HZ BOOKS
华章IT

C语言编程魔法书

基于C11标准

陈轶 著

- 基于C11标准，涵盖透彻的语法讲解、高级特性、主流底层编译器支持，配合大量示例与讲解（How-Why），掌握C魔法精髓。
- C语言与汇编语言重度用户与拥趸者撰写，10余年开发经验结晶，讲究透彻而实用，字字珠玑。

机械工业出版社
China Machine Press

C语言编程魔法书：基于C11标准

陈轶　著

ISBN：978-7-111-56521-5

本书纸版由机械工业出版社于2017年出版，电子版由华章分社（北京华章图文信息有限公司，北京奥维博世图书发行有限公司）全球范围内制作与发行。

客服热线：+ 86-10-68995265

客服信箱：service@bbbvip.com

官方网址：www.hzmedia.com.cn

新浪微博 @华章数媒

微信公众号 华章电子书（微信号：hzebook）

目录

前言

为什么要写这本书

本人在2001年上了大学本科，读计算机科学与技术专业。在第一年的上半学期，对计算机编程还没什么感觉。但是就在考“C语言程序设计”这门专业课的前一个月，感觉这门课学了那么久几乎什么都不会，可把我急坏了。然后就在这短短一个月的时间里又是看书，又是上机实验，终于考了70多分，算是过关了……不过奇怪的是在考试结束后，就发现自己对编程有了感情。到了大二，我们上“数据结构”所使用的教材是基于C++编程语言的，因为之前没学过C++语言，所以只能自学。而在这个过程中，我发现自己对编程更加热爱。在上完大三之后，我在暑假里又把之前的C语言重新巩固一番。有了计算机组成、操作系统、汇编语言、数据结构等知识积淀之后再去看C语言编程就感觉容易多了。我也是由此喜欢上了C编程语言。

10年之后，发现国内市面上很多C语言参考书仍然显得非常陈旧。不仅基于古老的C89/90标准，而且还在用Visual C++6.0这种既收费又老旧的开发环境教学生。对于比较新的C99标准的讲解屈指可数，更鲜有针对最新的C11标准的书籍。出于对C语言的热爱，在此热切希望能把最新标准的C语言奉献给各位读者，也想把C语言的方方面面讲透并且能讲得通俗易

懂，方便读者去思考实践，所以这也是我写这本书的主要原因。当各位阅读完本书之后，会发现C语言竟然如此强大！而且在大部分时候，尤其是我们想集中注意力解决某个特定问题的时候，使用C语言要比用其他一些基于面向对象的类C编程语言（比如C++、Java等）要直观得多！

本书之所以叫“C语言编程魔法书”，是因为像“宝典”、“圣经”之类的词已经被用滥了。再者，C语言本身就拥有极其强大的魔力，你能用它做几乎所有的事情。而且几乎每一个C语言编译器都能内联汇编语言，或者与C++、Objective-C直接兼容，而对于像Java、C#、Python等许多编程语言也有相应的接口。所以，我认为C语言在计算机编程语言领域中就好比数学在自然科学中的地位和作用，它是很多编程语言的基础，而且很多编程语言的编译器或解释器也都是基于C语言来写的。

就在2015年2月，Khronos标准组织发布了最具现代化的图形API——Vulkan，其主机端接口用的API是纯C语言。此外，像OpenGL、OpenCL、OpenAL、OpenVG等开放标准都基于纯C语言。此外，最近10年来TIOBE每月的编程语言排名，C语言排名始终能进前两名，也能说明它的使用范围之广，而且许多开源项目也多多少少会使用C语言来编写。况且学了C语言之后，再学习C++、Java等面向对象编程语言也会轻松很多。尤其像C++和Objective-C，没有C语言基础是完全不行的。所以个人十分推荐计算机系的大学生将C语言作为自己的计算机入门编程语言！

本书特色

从技术层面上讲，本书介绍了C语言的最新标准，即ISO/IEC 9899：2011。同时，也介绍了主流开源C语言编译器GCC与Clang对标准C语言语法的扩充。而且所基于的编译器和开发环境也是比较新的Visual Studio Community 2017、GCC 5，以及Clang 3.8（Apple LLVM 8.0，基于Xcode 8）。

从适合读者阅读和掌握知识的结构安排上讲，本书分为“预备知识篇”、“基础语法篇”、“高级语法篇”，以及“语法扩展篇”，还有最后的“项目实践篇”。从基础到高级，循序渐进地为读者描述C语言编程方法。本书尤其着重C语言标准语法上的精确描述，通过许多代码片段给读者介绍各种C语言语法知识，并且能反映出C语言的灵活性以及在使用上的约束。

本书推崇读者使用合法免费的C语言编译器以及集成开发环境，希望读者能有正确的软件版权意识，这样才能更好地为我国软件事业增添光彩，为打造良好的应用市场以及生态环境作出贡献。因此，本书主要选择使用GCC、Clang这两个主流开源免费的C语言编译器，而集成开发环境（IDE）则采用Visual Studio Community、Eclipse、Xcode这三个常用的免费开发工具，其中，Visual Studio Community不是开源的，而Xcode则是部分开源的。

本书虽然会讲解整个C编程语言，涉及了几乎所有的语法点，但是考虑到本书读者可能是初学C语言，且没有多少计算机专业知识，所以本书措辞会尽量通俗，而不过于追求学术化。某些描述可能会不太严谨，但对于本书所用到的GCC、Clang这两大主流编译器而言将完全适用。另外，考

虑到不少读者从事嵌入式系统开发工作，所以对于C语言标准中出现的所谓“由实现定义的”场合会尽量区分情况分别阐明。本书的最终的目的就是让读者至少能熟练掌握C语言编程，能将它灵活地运用于实际工程中。

读者对象

·嵌入式系统开发者

·移动或桌面客户端应用程序开发者

·服务器端应用程序开发者

·系统架构师

·计算机、电子工程、通信专业的大学生

·其他对C语言编程感兴趣的人员

如何阅读本书

本书一共分为四大篇。

预备知识篇（第1～3章），简单描述C语言的概况、学习C语言的预备知识，以及在Windows、macOS和Linux三大桌面环境下搭建编写C环境的方法。

第1章　C魔法概览。主要介绍C语言的来历和演化，用它编写代码的编程模式以及我们可以用于实践的主流C语言编译器。

第2章　学习C语言的预备知识。这一章主要为不太熟悉计算机系统的读者提供一些基础的计算机理论知识和相关概念，比如整数与浮点数在计算机中的表示方法、字符编码格式、按位逻辑计算、移位操作等。

第3章　C语言编程的环境搭建。这一章主要介绍了Windows、macOS以及Linux系统下如何安装并使用主流编译器与集成开发环境。

基础语法篇（第4～11章）讲解C语言的基本语法。这是C语言程序员必须掌握的。

第4章　C语言中的基本元素。这一章描述了C语言中常用字符集以及合法token的构成。此外还介绍了标识符、关键字以及标点符号的使用说明。

第5章　基本数据类型。这一章介绍了整数类型、字符类型、浮点类型数据的表示，以及它们之间的类型转换。此外还描述了对于这些基本数据类型的算术逻辑操作、投射操作以及通过sizeof操作符获取数据类型与对象相应的字节数。

第6章　用户自定义类型。这一章描述了枚举、结构体以及联合体这三种用户自定义类型，并介绍了它们的特性以及各种使用方式。

第7章　C语言的数组和指针。这一章十分关键，也是C语言的语法难

点。这里详细介绍了C语言中一维数组与多维数组的表示以及如何对它们进行操作，然后介绍了C语言中的指针类型，详细阐述了指针类型的使用技巧以及需要注意的事项。

第8章　C语言的控制流语句。这一章介绍了C语言的条件语句、选择语句以及循环等控制流语句。

第9章　C语言的函数。这一章介绍了C语言中的函数概念，包括C语言函数的声明及定义，还有C函数的调用。此外还介绍了C语言函数标识符作为表达式时的类型。

第10章　C语言的预处理器。这章包含了目前C11标准中所支持的所有预处理器特性，包括宏定义、预处理条件、预编译指示符与操作符以及C代码的注释。

第11章　C语言的编译上下文。这一章介绍了C语言对象与函数的作用域和名字空间。详细介绍了C语言中的四大作用域以及在不同作用域中的对象的生命周期。此外还介绍了对象与函数的连接属性，包括外部连接和内部连接。

高级语法篇（第12～16章）讲述C语言的一些高级特性。这一部分内容不需要C语言程序员必须掌握，但需要对此有个大概了解。

第12章　C语言中的类型限定符。该章介绍了C11标准中支持的const、volatile、restrict与_Atomic这四种限定符。详细说明了限定符用于修饰含有

指针的对象时，在*号的不同位置所起到的不同作用。然后分别介绍这四种限定符的具体含义。

第13章　C语言中的类型系统。这一章把C语言语法体系中的整个类型系统再梳理了一遍。这一章介绍了对于一些复杂类型的对象如何去剖析、理解，然后自己如何去声明自己想要的复杂类型的对象和函数。这一章所描述的其实是整个C语言语法体系的核心，如果大家能掌握的话，那么基本就算是真正掌握C语言了。其实，对于任一强类型的编程语言而言，其系统类型总是扮演着十分重要的角色，我们学习此类语言都需要透彻理解其整个类型系统。

第14章　C11标准中的表达式、左值与求值顺序。该章先介绍了C11标准中各类表达式以及它们的计算优先级。然后介绍了“左值”这个概念，并讲解了表达式之间的求值顺序。

第15章　函数调用约定与ABI。该章与C语言标准并无太大关系，但却与实际项目开发有关。这一章介绍了主流C语言编译器所采用的函数调用约定，然后详细描述了函数调用的过程，包括参数传递和返回值的具体处理。该章对嵌入式系统开发者以及需要将C语言与汇编语言进行交互使用的高性能计算开发者而言，将大为有用。

第16章　创建动态库与静态库。这一章介绍了用主流C语言编译工具构建静态库以及动态库的方法，并介绍如何使用这些库文件。

语法扩展篇（第17～19章）讲述了GCC与Clang编译器对C语言的扩

展。

第17章　GCC对C11标准的扩展。该章先简单介绍GNU语法扩展，然后介绍GCC编译器中常用的扩展语法。

第18章　Clang编译器对C11标准的扩展。该章介绍了Clang编译器对C11标准的语法扩展。最后还介绍了Apple开源的Grand Central Dispatch库的简单使用。

第19章　对C语言的未来展望。该章主要介绍了C语言的设计理念以及当前C语言标准委员会的工作组正在为C语言新增的内容，还谈到了哪些特性不会被添加到C语言中去。

项目实践篇（第20～21章），这里通过两个实际的C语言项目来介绍我们如何利用C语言来创作出自己的程序。

第20章　描述了UTF-8编码格式的字符串与UTF-16编码格式的字符串进行相互转换的例子。

第21章　介绍一个看似简单而功能很丰富的基于控制台的计算器程序。

建议零基础的读者要了解第一篇的预备知识，这对于后面深入学习C语言编程很有帮助。另外，这部分读者可以先不用强行看第三篇，尤其是第15章。因为第三篇涉及的知识比较深，而第15章又会直接引入汇编语言，这对于没有一定计算机专业知识的读者会比较难以理解。如果是有一

定计算机专业知识的读者可以略过第一篇，直接阅读第二篇。另外，如果是从事嵌入式系统开发的、或从事系统底层开发的资深程序员，建议仔细阅读第三、第四篇，相信这部分内容会对你的工作很有帮助。

勘误和支持

由于笔者的水平有限，编写时间仓促，书中难免会出现一些错误或者不准确的地方，恳请读者批评指正。如果你有更多的宝贵意见，欢迎你访问我的个人博客网站http://blog.csdn.net/zenny_chen进行专题讨论，我会尽量在线上为你提供最满意的解答。同时，你也可以通过微博http://weibo.com/zenny1chen与我联系，或发送电子邮件到zenny_chen@163.com。期待能够得到你们的真挚反馈，在技术之路上互勉共进。另外，本书最后两章的代码可以在作者的GitHub上获取：https://github.com/zenny-chen。

致谢

首先感谢我的父母和妻子对我写作此书的大力支持，尤其是我妻子在我忙于工作、编写此书时帮忙照顾孩子和做饭。然后感谢我公司老板对我写作此书的鼓舞与期待。

这里还要感谢机械工业出版社华章公司的编辑高婧雅，在一年多的时

间里给予我的大力支持和帮助。

最后感谢支持我的技术爱好者，感谢你们对我的支持以及对我的信任。

我想和作者聊聊

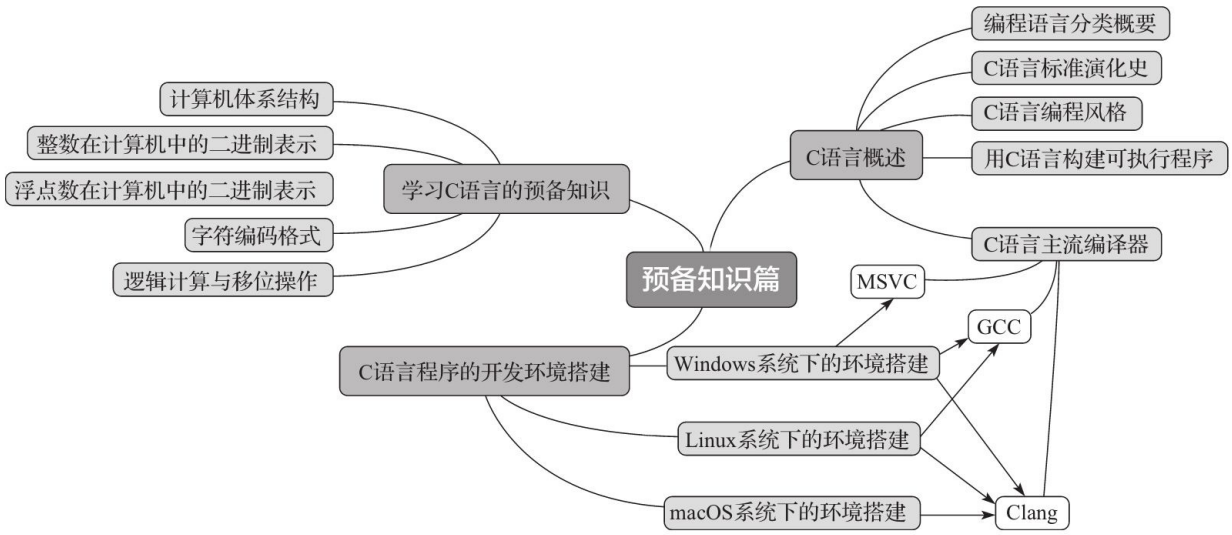
为了能更好地与读者进行联系，笔者这里留了一个QQ讨论群。各位如果在阅读此书中有任何疑问可以来本群询问，大家可以一起探讨。各位可以扫一扫下方的二维码，进此群的提示语为：“C语言编程魔法书”，或者查询群号86540289申请入群。

zenny_chen作者读书群

扫一扫二维码，加入该群。

陈轶

第一篇　预备知识篇


编程语言分类概要
C语言标准演化史
C语言编程风格
用C语言构建可执行程序
C语言概述
计算机体系结构
整数在计算机中的二进制表示
浮点数在计算机中的二进制表示
字符编码格式
逻辑计算与移位操作
学习C语言的预备知识
C语言主流编译器
预备知识篇
MSVC
GCC
C语言程序的开发环境搭建
Windows系统下的环境搭建
Linux系统下的环境搭建
macOS系统下的环境搭建
Clang

第1章　C魔法概览

本章内容主要对C编程语言（以下简称C语言）进行大体介绍，包括它的历史以及C语言标准的演化进程。然后介绍一下C语言编程思想，当前主流C语言编译器以及GNU语法扩展。最后简单介绍一下从用C语言编写程序到编译、构建一个可执行程序的大致过程。

计算机编程语言从对计算机硬件底层的抽象程度进行分类，可分为：机器语言、汇编语言以及高级语言。下面由底层到高层分别介绍这几种类别的编程语言。

1.1 例说编程语言

1）机器语言是直接通过十六进制数表示当前处理器架构的机器指令码。指令码包含了当前指令的功能（比如算术逻辑运算、移位、分支、中断、I/O等）、寄存器、立即数等多种元素。每种处理器架构所对应的机器码的字节长度也各不相同，有些是固定长度的（比如ARM、MIPS等架构），有些是可变长度的（比如x86架构）。

2）汇编语言（Assembly Language）通过简单的指令助记符（memonics）来表示对应机器指令的功能、寄存器编号、立即数（immediates）等元素。汇编语言是对机器指令的简单抽象，通过汇编器（assembler）可以将汇编语句翻译成对应的机器指令码。

3）高级语言的表达形式更为抽象且贴近我们日常的语言表述。而且，高级语言比起汇编语言往往更具有表达力，且拥有更加丰富的语法特性，以便将程序进行结构化和模块化。比如，高级语言具有自定义变量标识符、自定义数据结构、分支与循环、更形象自然的表达式等。高级语言一般通过编译器（compiler）可直接将表达式翻译为对应的机器指令码；也可以将高级语言先翻译为中间语言（类似于汇编，但可能比汇编适用范围更广、更利于跨平台的字节码），最后将中间语言翻译为最终的机器指令码。

当然，有些书中还介绍了第四代语言，它基于高级语言，比高级语言

更抽象，只需要一些简单的描述语句就能让计算机做比较复杂的工作。比如SQL（结构化查询语言，用于数据库查询）算是一种第四代语言。

下面，为了能让大家对这三种层次的编程语言有一个感性的认识，这里将列举ARMv8架构处理器下的机器语言、汇编语言，加上它们相应的C语言。读者如果手头有Xcode，并且有包含Apple A7或更高版本处理器的iOS设备的话，可以直接编译运行，并能看到最终效果。

下面首先列出一个文件名为my_sub.s的汇编源文件，其中包含了机器语言和汇编语言。见代码清单1-1：

代码清单1-1　机器语言与汇编语言

```
.text
.align 4

#ifdef __arm64__

.globl _my_sub_machine
.globl _my_sub_assembly

// 用机器语言实现减法操作
_my_sub_machine:

    .long 0x4b010000

    .long 0xd65f03c0

// 用汇编语言实现减法操作
_my_sub_assembly:

    sub w0, w0, w1

    ret

#endif
```

在代码清单1-1中，_my_sub_machine程序片段中的两条.long语句即为机器指令。这两条机器指令正好与_my_sub_assembly中的两条汇编指令相

对应。也就是说，“0x4b010000”这串32位的十六进制代码意思就是“sub w0，w0，w1”，表示将寄存器w0与寄存器w1的值进行相减，然后将结果写回w0寄存器中。而“0xd65f03c0”指令码对应于“ret”（更确切地说是ret x30），表示返回当前过程（procedure）。在汇编语言中，一般会使用过程或者例程（routine）来表示一个可执行的程序片段。在C语言中一般都用函数（function）表示。我们在这里能够明显看到，汇编语言采用指令助记符的方式比写机器指令码要直观得多，而且也不容易出错。“sub”指令的功能从助记符上就能知道是“减法”功能；而w0、w1也明确指明了使用的寄存器是w0和w1。这些在“0x4b010000”这种机器指令码上都无法直观地表现出来。

代码清单1-2列出C语言是如何表达一个减法操作的。

代码清单1-2　减法操作对应的C语言

```
static int my_sub_c(int a, int b)
{
    return a - b;
}
```

代码清单1-2所列出的C语言代码与代码清单1-1中的机器指令码和汇编语言完全对应，意思一目了然——将参数变量a的值与参数变量b的值进行相减，然后将结果返回。从这里我们就能看到机器语言、汇编语言以及以C语言为代表的高级语言之间在表达力上的差距了。高级语言的目的就是为了给程序员提供更良好的编程工具，更简洁、更富有表达力的语言，使得我们程序员能提升生产力，并且能构思出更多精彩炫酷的应用，而不是

把太多的精力都投入在如何让计算机执行的细节上。

代码清单1-3能让我们在主函数或其他函数中测试上述已经编写好的函数。

代码清单1-3　展示减法操作的结果

```
#ifdef __arm64__

extern int my_sub_machine(int a, int b);
extern int my_sub_assembly(int a, int b);

int result_machine = my_sub_machine(10, 2);
int result_assembly = my_sub_assembly(5, 3);
int result_c = my_sub_c(6, 2);

printf("Three results: %d, %d, %d\n", result_machine, result_assembly, result_c);

#endif
```

执行了上述代码之后，我们最后能在控制台看到输出结果：“Three results：8，2，4”。可见，上述三种不同的编程语言，计算功能是完全一致的，都是对两个输入参数做减法操作，然后返回差值。然而就可读性、可理解性以及编程便利性而言，显然C语言比起其他两者要强得多。而可读性最差的无疑就是机器指令码了。

1.C语言的类别与产生

对于高级语言来说，从表达上又可分为命令式编程语言（imperative programming language）和陈述型编程语言（declarative programming language）。命令式语言主要包括过程式（procedural）、结构化（structured）以及面向对象（object-oriented）的编程语言；陈述型编程语

言主要包括函数式（functional）以及逻辑型（logical）编程语言。而C语言则属于结构化的命令式编程语言。不过现在很多命令式编程语言也包含了一些函数式编程语言的特征。在本书中，后面第18章中谈到的Blocks语法就是一个很典型的函数式编程语言的语法。

C语言最初由Dennis Ritchie于1969年到1973年在AT&T贝尔实验室里开发出来，主要用于重新实现Unix操作系统。此时，C语言又被称为K&R C。其中，K表示Kernighan的首字母，而R则是Ritchie的首字母。K&R C语言与后来标准化的C语言有很大差异。比如，如果函数返回类型为int，则int可省：int my_function（）{}，也可以写成my_function（）{}。编译器不会有任何警告，更不会报错。另外，还有现在看来比较奇葩的函数定义，像我们现在定义这么一个函数——void my_function（int a，char*p）{}，如果是用K&R C语法定义的话要写成：void my_function（a，p）int a；char*p；{}。K&R的C语法中，定义一个函数时，其形参列表先列出形参的标识符，然后在函数声明的后面紧跟着对形参标识符的完整声明，最后是函数体。这在现行标准中已经被逐步废弃使用了。另外，当时的第一本C语言专业书《The C Programming Language》也并非一个正式的编程语言规范，但被用了许多年。

2.C90标准

由于C语言被各大公司所使用（包括当时处于鼎盛时期的IBM PC），因此到了1989年，C语言由美国国家标准协会（ANSI）进行了标准化，此时C语言又被称为ANSI C。而仅过一年，ANSI C就被国际标准化组织ISO

给采纳了。此时，C语言在ISO中有了一个官方名称——ISO/IEC 9899：1990。其中，9899是C语言在ISO标准中的代号，像C++在ISO标准中的代号是14882。而冒号后面的1990表示当前修订好的版本是在1990年发布的。对于ISO/IEC 9899：1990的俗称或简称，有些地方称为C89，有些地方称为C90，或者C89/90。不管怎么称呼，它们都指代这个最初的C语言国际标准。这个版本的C语言标准作为K&R C的一个超集（即K&R C是此标准C的一个子集），把后来引入的许多非官方特性也一起整合了进去。其中包括了从C++借鉴的函数原型（Function Prototypes），指向void的指针，对国际字符集以及本地语言环境的支持。在此标准中，尽管已经将函数定义的方式改为现在我们常用的那种方式，不过K&R的语法形式仍然兼容。

3.C99标准

在随后的几年里，C语言的标准化委员会又不断地对C语言进行改进，到了1999年，正式发布了ISO/IEC 9899：1999，简称为C99标准。C99标准引入了许多特性，包括内联函数（inline functions）、可变长度的数组、灵活的数组成员（用于结构体）、复合字面量、指定成员的初始化器、对IEEE754浮点数的改进、支持不定参数个数的宏定义，在数据类型上还增加了long long int以及复数类型。毫不夸张地说，即便到目前为止，很少有C语言编译器是完整支持C99的。像主流的GCC以及Clang编译器都能支持高达90%以上，而微软的Visual Studio 2015中的C编译器只能支持到70%左右。

4.C11标准

2007年，C语言标准委员会又重新开始修订C语言，到了2011年正式发布了ISO/IEC 9899：2011，简称为C11标准。C11标准新引入的特征尽管没C99相对C90引入的那么多，但是这些也都十分有用，比如：字节对齐说明符、泛型机制（generic selection）、对多线程的支持、静态断言、原子操作以及对Unicode的支持。本书将主要针对C11标准为大家详细讲解C编程语言。关于C语言历史与演化进程的详细介绍可参考维基百科：https://en.wikipedia.org/wiki/C_%28programming_language%29。

笔者近两年也是在不断地了解C语言标准委员会的最新动态（可参见：http://www.open-std.org/jtc1/sc22/wg14/），其中看到有人提出想为C语言添加面向对象的特性，包括增加类、继承、多态等已被C++语言所广泛使用的语法特性，但是最终被委员会驳回了。因为这些复杂的语法特性并不符合C语言的设计理念以及设计哲学，况且C++已经有了这些特性，C语言无需再对它们进行支持。笔者将在第19章给大家谈谈C语言设计理念与发展方向。

1.2 用C语言编程的基本注意事项

C语言的发明其实基于Unix操作系统。当时在C语言未面世之前，Dennis Ritchie所在的AT&T贝尔实验室用的Unix系统是完全用汇编语言写的。汇编语言的优势是直接面向处理器本身，能直接对底层硬件进行控制，充分发挥处理器的硬件能力。然而，它的缺陷也是显而易见的。

1.汇编语言的不足

首先，不可移植性。每种处理器，其指令集都大相径庭，比如ARM有ARM的指令集架构（ISA），Intel x86有x86的ISA，还有MIPS、Power（原来为PowerPC），Motorola 68000等；再加上各类微控制器单元（Micro-Controller Unit，MCU）、各类数字信号处理器（Digital Signal Processor，DSP），每种ISA都有其相应的汇编语言。那么多处理器如果对每一种都使用不同的汇编语言来实现同一个操作系统，那操作系统的开发人员真要崩溃了……而且即便实现出来，可能各个处理器上的实现也会有所不同，标准也很难被统一起来。

其次，汇编语言本身要比高级语言精密。因为汇编语言面对的都是寄存器、存储器以及各类底层硬件，而不是一种抽象的数据模型，所以代码编写时需要非常谨慎，而且调试程序也十分麻烦，且非常容易出错。所以，如果有一种既能面向底层硬件，又能对数据以及程序进行抽象的高级语言出现，那势必既能不太影响程序执行效率，又能大大提升程序的可执

行性、可读性以及编写的效率，这将是非常伟大的贡献。C语言也就是在这种背景下诞生的。

如果说，汇编语言面向的是底层硬件、一种过程化的编程风格的话，那么C语言就是面向数据流和算法、一种结构化的编程风格。C语言是一种结构化的、静态类型的编译型编程语言。也就是说，用C语言编写了源代码之后，需要通过C语言编译器进行编译，构建为相应的处理器能直接执行的机器码，然后处理器可以对生成出来的机器码进行执行。所以在各个处理器上，处理器厂商或第三方只需要为当前处理器写一个对应的C语言编译器即可。然后任何符合C语言标准的程序都能在上面编译后执行，除了需要支持某些机器特定的功能和特性外（后面会介绍）。

2.C语言编写程序要注意什么

那么我们在用C语言写程序的时候应该注意哪些方面呢?

1）可移植性：C语言被设计出来的一大初衷就是为了能将同一个源代码放到各个不同的平台上编译运行。因此，如果我们的代码要在多种不同架构的处理器上运行的话，我们就得注意C语言标准规定了哪些特性是编译器必须遵守的，哪些特性是平台或编译器自己实现的。我们要尽量使用标准中已明文规定的编程规范，尽可能避免在不同平台可能会产生不同行为的语法特性。当然，由于上面提到的处理器种类太过多样，尤其在嵌入式开发领域，很多MCU用的还都是8位处理器，这种情况下C源代码就很难被移植到32位或64位系统下了。本书后面将会指出大部分主流平台对C语

言标准中所提到的“实现定义”行为的区别。另外，也会提到一些技巧来应对不同的平台特性。

2）可维护性：可维护性在实际工程项目的研发中非常重要。它体现在最初工程架构的设计、对各个功能模块的划分、相应的开发人员安排，还有后期的测试。一般来说，现在一个工程如果是从无到有进行开发的话会采用螺旋式开发模型。也就是说，一个项目启动后，可以先做一个功能简单但能正常工作的产品原型。然后在此基础上不断地为它增加更多功能，或对之前的功能进行修改。在此期间，我们如何对整个工程进行模块化划分，从而能安排不同开发人员针对不同功能模块进行开发就变得尤为重要。另外，在工程开发过程中，如果有人员流动，那么如何将即将离职的开发人员手中的工作交付给新人也关系到整个项目的进展。因此，一个良好的C语言代码应该具有可读性、良好的文档化注释风格，以及较详细的设计文档。对于一个较大的工程项目来说，开发人员不仅仅需要把自己的代码写好，而且要写得能让别人看懂，并且要做好详细的设计文档，这样才能把项目风险降低。

3）可延展性：大家或许已经知道，像微软的Windows操作系统由数千名工程师合作研发；Linux操作系统对外开源，参与其中的研发人员也有数百上千人。如果我们在一个开发团队中负责一个需要由多人合作开发的工程项目，那么我们写的功能模块需要与其他人写的功能模块进行对接。所以，我们在开发一个较大工程项目时，需要协调好各自对外的模块接口（Application Program Interface，API）。由于C语言没有全局名字空间

（namespace）这个概念，所以命名一个对外接口也是非常重要的，否则可能会与其他功能模块的接口名发生冲突。本书后面会对C语言函数命名以及符号连接做进一步介绍。

4）性能：性能是提升程序使用者效率和生产力的体现。一个应用程序的性能越高，那么计算一个任务所花费的时间越短，也越节省计算机的耗电。而对于如何提升性能，一方面需要程序员对处理器架构、硬件特性有一定了解；另一方面需要程序员拥有比较丰富的算法知识，能针对实际需求灵活采用高效的算法。而像C语言这种十分接近硬件底层的高级编程语言，能极大限度地发挥处理器的特长，从而达到高效的运行性能。

1.3 主流C语言编译器介绍

对于当前主流桌面操作系统而言，可使用Visual C++、GCC以及LLVM Clang这三大编译器。其中，Visual C++（简称MSVC）只能用于Windows操作系统；其余两个，除了可用于Windows操作系统之外，主要用于Unix/Linux操作系统。像现在很多版本的Linux都默认使用GCC作为C语言编译器。而像FreeBSD、macOS等系统默认使用LLVM Clang编译器。由于当前LLVM项目主要在Apple的主推下发展的，所以在macOS中，Clang编译器又被称为Apple LLVM编译器。MSVC编译器主要用于Windows操作系统平台下的应用程序开发，它不开源。用户可以使用Visual Studio Community版本来免费使用它，但是如果要把通过Visual Studio Community工具生成出来的应用进行商用，那么就得好好阅读一下微软的许可证和说明书了。而使用GCC与Clang编译器构建出来的应用一般没有任何限制，程序员可以将应用程序随意发布和进行商用。不过由于MSVC编译器对C99标准的支持就十分有限，加之它压根不支持任何C11标准，所以本书的代码例子不会针对MSVC进行描述。所幸的是，Visual Studio Community 2017加入了对Clang编译器的支持，官方称之为——Clang with Microsoft CodeGen，当前版本基于的是Clang 3.8。也就是说，应用于Visual Studio集成开发环境中的Clang编译器前端可支持Clang编译器的所有语法特性，而后端生成的代码则与MSVC效果一样，包括像long整数类型在64位编译模式下长度仍然为4个字节，所以各位使用的时候也需要注意。为了方

便描述，本书后面涉及Visual Studio集成开发环境下的Clang编译器简称为VS-Clang编译器。

而在嵌入式系统方面，可用的C语言编译器就非常丰富了。比如用于Keil公司51系列单片机的Keil C51编译器；当前大红大紫的Arduino板搭载的开发套件，可用针对AVR微控制器的AVR GCC编译器；ARM自己出的ADS（ARM Development Suite）、RVDS（RealView Development Suite）和当前最新的DS-5 Studio；DSP设计商TI（Texas Instruments）的CCS（Code Composer Studio）；DSP设计商ADI（Analog Devices，Inc.）的Visual DSP++编译器，等等。通常，用于嵌入式系统开发的编译工具链都没有免费版本，而且一般需要通过国内代理进行购买。所以，这对于个人开发者或者嵌入式系统爱好者而言是一道不低的门槛。不过Arduino的开发套件是可免费下载使用的，并且用它做开发板连接调试也十分简单。Arduino所采用的C编译器是基于GCC的。还有像树莓派（Raspberry Pi）这种迷你电脑可以直接使用GCC和Clang编译器。此外，还有像nVidia公司推出的Jetson TK系列开发板也可直接使用GCC和Clang编译器。树莓派与Jetson TK都默认安装了Linux操作系统。在嵌入式领域，一般比较低端的单片机，比如8位的MCU所对应的C编译器可能只支持C90标准，有些甚至连C90标准的很多特性都不支持。因为它们一方面内存小，ROM的容量也小；另一方面，本身处理器机能就十分有限，有些甚至无法支持函数指针，因为处理器本身不包含通过寄存器做间接过程调用的指令。而像32位处理器或DSP，一般都至少能支持C99标准，它们本身的性能也十分强大。而像ARM出的RVDS编译器甚至可用GNU语法扩展。

图1-1展示了上述C语言编译器的分类。

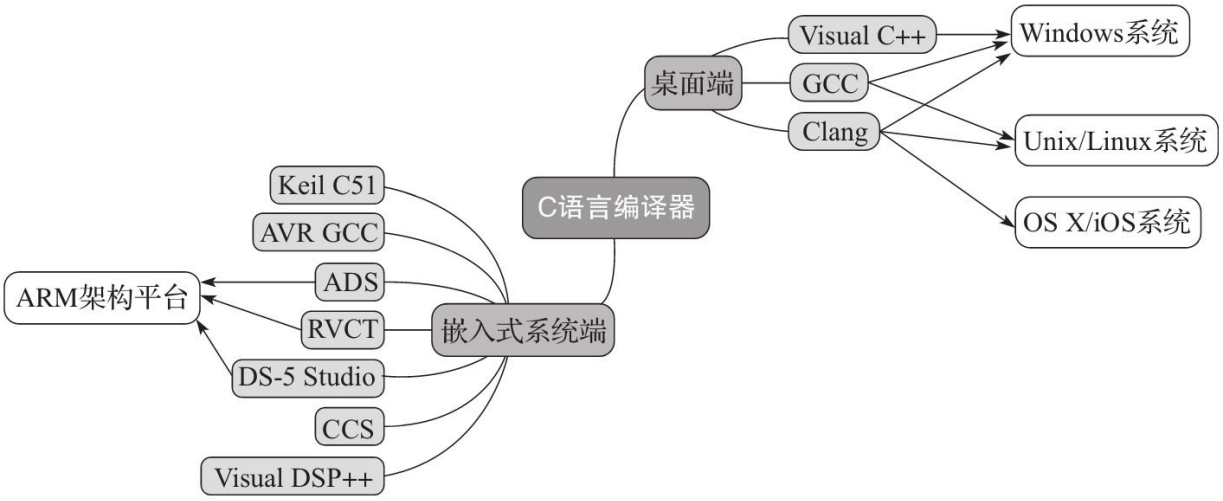


图1-1　C语言编译器的分类

1.4 关于GNU规范的语法扩展

GNU是一款能用于构建类Unix操作系统的计算机软件合集，由自由软件之父Richard Stallman开创，于1983年9月27日对外发布。GNU完全由自由软件（free software）构成。GNU语法扩展源自于GCC编译器，在1987年发布1.0版本，称为GNU C Compiler。随后，GCC编译器前端[1]支持了C++、Objective-C/C++、Fortran、Ada、Java以及最近跃升的Go等编程语言，因此现在GCC被称为GNU Compiler Collection。由于在20世纪90年代，GNU C编译器就对C90标准做了相当多的语法扩展，包括复合字面量、匿名结构体和数组、可指定的初始化器等，这些语法扩展被广泛使用，尤其是大量用于Linux内核代码中，因此C99标准将这些语法特性全都列入标准之中。

正因为GCC本身是开源自由软件，因此很多商用编译器也基于GCC进行扩展。像ARM的RVCT（RealView Compiler Toolkit）本身就支持GNU扩展。还有不少开发平台本身就直接使用GCC编译工具。由于有不少大公司顶级开发人员的参与，因此GCC编译器的目标代码优化能力相当高，而且还支持许多不同的处理器。所以，GCC当前被广泛使用并博得开发者的好评。像Linux操作系统基本默认使用GCC作为默认编译器，包括Android的NDK开发工具一开始也是如此。

然而，由于GCC基于比较严格的GPL许可证，许多大型商业开发商对

它望而却步。该许可证允许使用者免费使用软件，但是要求不能随意对它进行篡改并重新发布。如果开发者对它进行篡改，然后发布自己修改之后的软件，那么必须要把自己修改的那部分也开源出来。因此，在2003年诞生了一个LLVM开源项目，基于更为宽松的BSD许可证，其编译器称为Clang。BSD许可证允许开发者随意对软件进行修改并重新发布，甚至可以将修改过的版本作为自主版权，因而这个许可证深受大公司的欢迎。现在Apple对LLVM项目的投入非常大。macOS上的开发工具Xocde从4.0版本起就开始使用Clang编译工具链，随后Apple将自己改写的Clang编译器称为Apple LLVM。当前最新的Xcode 8所使用的Apple LLVM版本为8.x。而当前Android NDK也支持了Clang编译器工具链。Clang编译器并非基于GCC，它是从头开始写的。但是它的目标是尽量与GCC编译器兼容，所以Clang编译器包含大部分GNU语法扩展，除此之外还含有它自己特有的C语言扩展。当然也有一些特性是GCC含有而Clang不具备的，不过这些特性一般很少使用。

我们现在可以看到GNU语法扩展适用性十分广泛。如果读者当前在做Linux/Unix或Windows上的C语言编程开发，或者是在开发macOS/iOS应用，又或者是在开发Android应用，那么完全可以毫无顾忌地使用GNU语法扩展。本书最后几个章节会分别介绍GCC编译器特定的语法扩展以及Clang编译器特定的语法扩展。由于Clang编译器已经包含了大部分GNU语法扩展，因此在介绍GCC语法扩展的时候，如果当前特性Clang不支持，则会指明。

[1] 源代码编译流程请见1.5节图1-2。

1.5 用C语言构建一个可执行程序的流程

从用C语言写源代码，然后经过编译器、连接器到最终可执行程序的流程图大致如图1-2所示。

从图1-2中我们可以清晰地看到C语言编译器的大致流程。首先，我们先用C语言把源代码写好，然后交给C语言编译器。C语言编译器内部分为前端和后端。前端负责将C语言代码进行词法和语法上的解析，然后可以生成中间代码。中间代码这部分不是必须的，但是它能够为程序的跨平台移植带来诸多好处。比如，同样的一份C语言源代码在一台计算机上编译完之后，生成一套中间代码。然后针对不同的目标平台（比如要将这一套代码分别编译成ARM处理器的二进制机器码、MIPS处理器的二进制机器码以及x86处理器的二进制机器码），只需要编写相应目标平台的编译器后端即可。所以，这么做就可以把编译器的前端与后端剥离开来（这在软件工程上又可称为解耦合），不同处理器厂商可以针对自家的处理器特性，对中间代码生成到目标二进制代码的过程再度进行优化。接下来，由C语言编译器后端生成源文件相应的目标文件。目标文件在Windows系统上往往是.obj文件；而在Unix/Linux系统上往往是.o文件。C语言的源文件在所有平台上都统一用.c文件表示。最后，对于各个独立的目标文件，通过连接器将它们合并成一个最终可执行文件。连接器与C语言编译器是完全独立的。所以，只要最终目标代码的ABI（应用程序二进制接口）一致，我们可以把各个编译器生成的目标代码都放在一起，最后连接生成一个可执

行文件。比如，有些源代码可用GCC编译，有些使用Clang编译，还有些汇编语言源文件可直接通过汇编器生成目标代码，最后将所有这些生成出来的目标代码连接为可执行文件。最终用户可以在当前的操作系统上加载可执行文件进行执行。操作系统利用加载器将可执行文件中相关的机器码存放到内存中来执行应用程序。

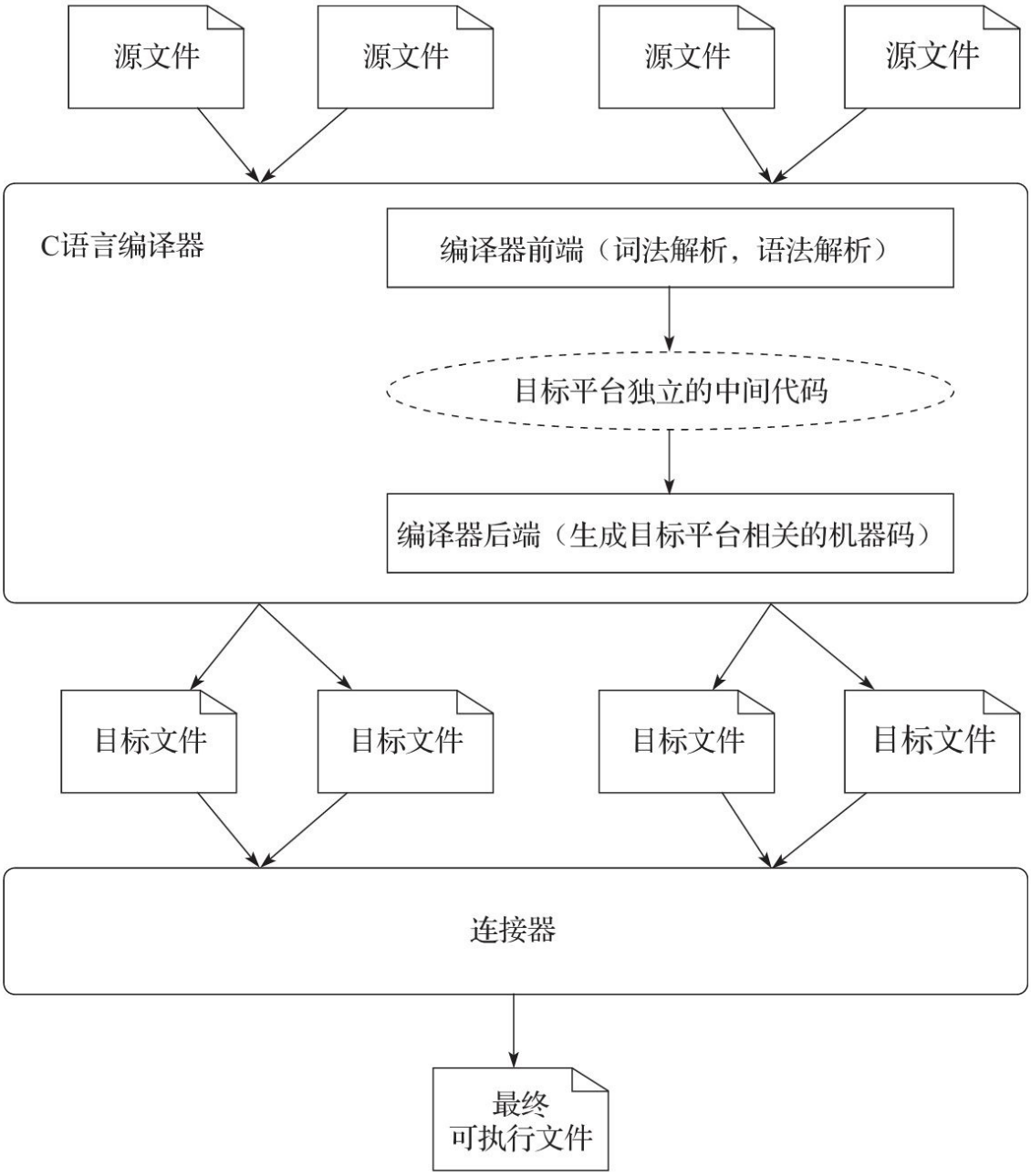


图1-2　C语言源代码编译流程图

1.6 本章小结

本章简要地介绍了计算编程语言的分类，描述了C语言的历史及演化，以及C语言的编程思想。此外还介绍了GNU的来龙去脉以及C语言编译器将C语言代码翻译成最终机器码的大致流程。

C语言作为一门更接近硬件底层的高级编程语言具有良好的抽象力、表达力和灵活性。此外，它具有非常高效的运行时性能。当前的C语言编译器最终翻译成的机器指令码与我们手工写汇编语言所得到的性能在大部分情况下相差无几。C语言基本能达成我们对性能的要求，而在某些对性能要求十分严苛的热点（hotspot）上，我们可以对这些功能模块手工编写汇编代码。C语言与汇编语言的ABI是完全兼容的，而且大部分C语言编译器还支持直接内联汇编语言。因此，C语言从1970年直到现在都是系统级编程的首要编程语言。

第2章　学习C语言的预备知识

我们在第1章已经大致介绍了C语言的概念以及编译、连接流程。我们知道C语言是高级语言中比较偏硬件底层的编程语言，因此对于用C语言的编程人员而言，了解一些关于处理器架构方面的知识是很有必要的，对于嵌入式系统开发的程序员而言更是如此了。

另外，C语言中有很多按位计算以及逻辑计算，所以对于初学者来说，如果对整数编码方式等计算机基础知识不熟悉，那么对这些操作的理解也会变得十分困难。因此，本章将主要给C语言初学者、同时也是计算机编程初学者，提供计算机编程中会涉及的基本知识，这样，在本书后面讲解到一系列相关概念时，初学者也不会感到陌生。

2.1 计算机体系结构简介

图2-1为一个简单的计算机体系结构图。

一个简单的计算机系统包含了中央处理器（CPU）以及存储器和其他外部设备。而在CPU内部则由计算单元、通用目的寄存器、程序序列器、数据地址生成器等部件构成。下面我们将从外到内分别简单地介绍这些组件。

2.1.1 贮存器

贮存器（Storage）尽管在图2-1中没有表示出来，但我们对它一定不会陌生，比如我们在PC上使用的硬盘（Hard Disk）就是一种贮存器。贮存器是一种存储器，不过它可用于持久保存数据而不丢失。因此我们通常把具有可持久保存的存储器统称为贮存器。现在PC上用得比较现代化的贮存器就是SSD（Solid-State Disk）了，俗称固态硬盘。当然，贮存器就其存储介质来说属于ROM（Read-Only Memory），即只读存储器。这类存储器的特点是数据能持久保留，比如我们PC上的文件，即便在关闭计算机之后也一直会保存在你的硬盘上，而且PC上的软件往往也是以可执行文件的形式保存在硬盘上的。但是它的读写速度非常缓慢，尤其是老式的SATA磁盘，写操作则更慢。因为通常对ROM的数据修改都要通过先读取某段数据所在的扇区，然后对该数据进行修改，再擦除所涉及的扇区，最后把修改好的

数据所包含的扇区再写回去。而对于ROM来说，其扇区是有写入次数限制的，所以写入次数越多，损耗就越大。当我们发现一个硬盘访问很慢的时候，通常就是其扇区（或磁道）已经破损严重了，这是在不断纠错并交换良好的扇区所引发的延迟。在嵌入式系统中，我们用的ROM一般是EPROM、EEPROM、Flash ROM等。这些硬件的详细资料各位可以从网上轻易获得，这里不再赘述。

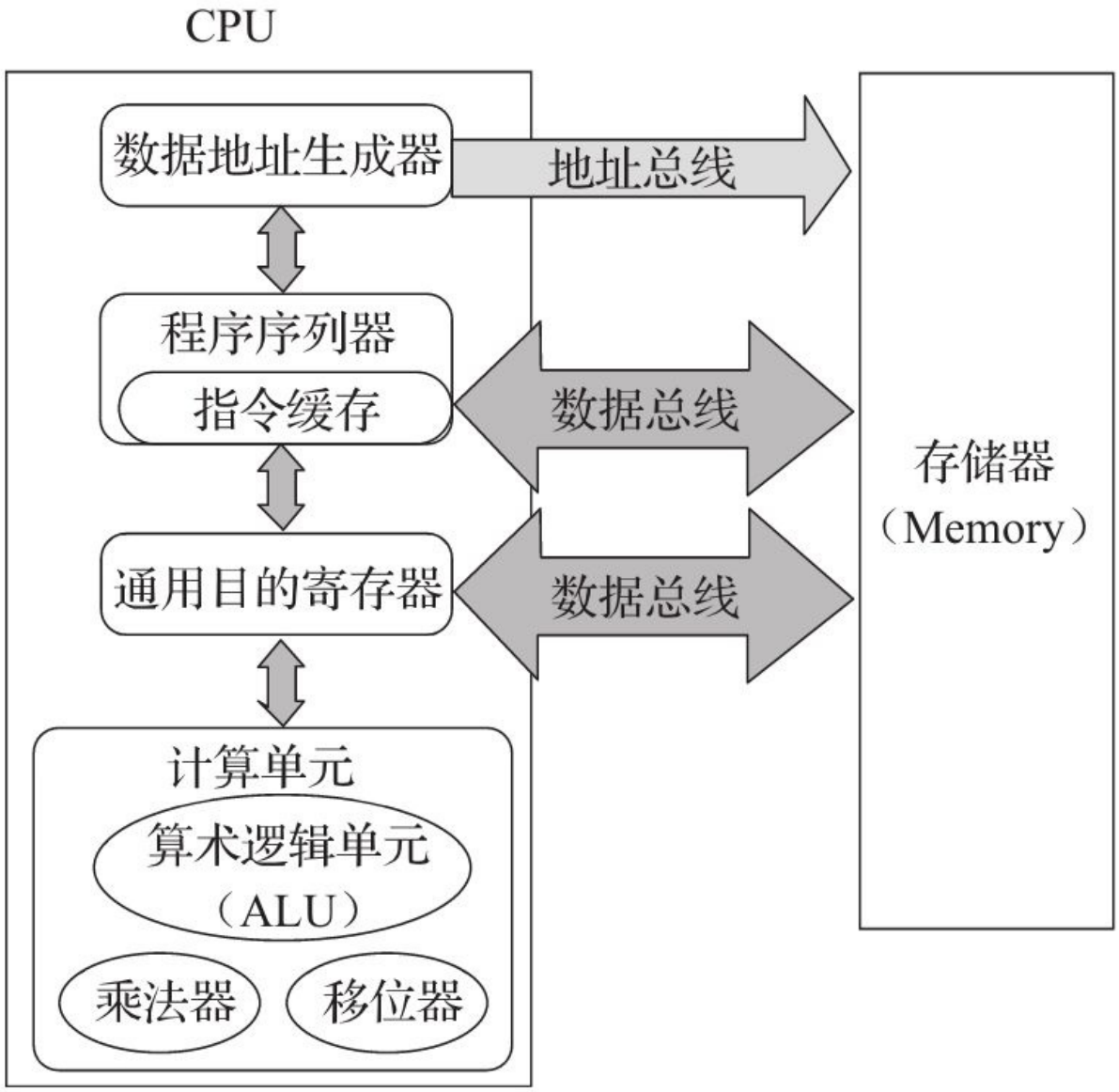


图2-1　简单的计算机体系结构图

2.1.2　存储器

存储器（Memory）一般是指我们通常所说的内存或主存（Main Memory）。其存储介质属于RAM（Random Access Memory），即随机访问存储器。它的特点是访问速度快，可对单个字节进行读写，这与ROM需要擦除整个扇区再对整个扇区写入的方式有所不同，因此更高效、灵活。但是RAM的数据无法持久化，掉电之后就会消失。此外，RAM的成本也比ROM高昂得多，我们对比一下16GB的内存条与256GB SSD的价格就能知道。然而正因为RAM的访问速度快，并且离CPU更近，所以在许多系统中都是将程序代码与数据先读取到RAM中之后再让CPU去执行处理的。当然，在一些嵌入式系统中也有让CPU直接执行ROM中的代码并访问读ROM中常量数据的情况，因为这类系统中总线频率以及CPU频率都相对较低，并且ROM也是与CPU以SoC（System-On-Chip，系统级芯片）的方式整合在一块芯片上的，所以访问成本要低很多。而有些环境对ROM的读取速度甚至比读取RAM还更快些。

 注意：在本书中所出现的“存储器”均表示内存，即RAM。而将可持久保存数据的存储器都一律称为“贮存器”。了解了这些概念后，我们在国外网站购买Mac或PC时，看到相关的术语就不会手足无措了。这里提供Apple美国官网的Mac配置信息网页，各位可以参

考：www.apple.com/macbook-pro/specs/。

2.1.3 寄存器

寄存器是在CPU核心中的、用于暂存数据的存储单元。一般处理器内部对数据的算术逻辑计算往往都需要通过寄存器（Register），而不是直接对外部存储器进行操作。因此，如果我们要计算一个加法或乘法计算，需要先把相关数据从外部存储器读到处理器自己的通用目的寄存器中，然后对寄存器做计算操作，再将计算结果也放入寄存器，最后将结果寄存器中的数据再写入外部存储器。寄存器的访问速度非常快，它是这三种存储介质中速度最快的，但是数量也是最少的。像在传统的32位x86处理器体系结构下，程序员一般能直接用的通用目的寄存器只有EAX、EBX、ECX、EDX、ESI、EDI、EBP这7个。还有一个ESP用于操作堆栈，往往无法用来处理通用计算。

2.1.4 计算单元

计算单元一般由算术逻辑单元（ALU）、乘法器、移位器构成。当然，像一般高级点的处理器还包含除法器，以及用于做浮点数计算的浮点处理单元（FPU）。它们一般都直接对寄存器进行操作。而涉及数据读写的指令会由专门的加载、存储处理单元进行操作。

2.1.5 程序执行流程

处理器在执行一段程序时，通常先从外部存储器取得指令，然后对指令进行译码处理，转换为相关的一系列操作。这些操作可能是对寄存器的算术逻辑运算，也可能是对存储器的读写操作，然后执行相关计算。最后把计算结果写回寄存器或写回到存储器。不过处理器在执行一系列指令的时候并不是每条指令都必须先经过上面所描述的整个过程才能执行下一条，而是采用流水线的方式执行，如图2-2所示。

图2-2体现了一个简单的处理器执行完一条指令的完整过程。我们这里假设从第一个取指令阶段到最后的写回阶段，这5个阶段均花费1个周期，倘若不是采用流水线的方式，而是每完成一条指令的执行再执行下一条指令，那么每条指令的处理都需要5个周期。而一旦采用流水线方式处理，那么我们可以看到，在第一条指令执行到译码阶段时，处理器可以对第二条指令做取指令操作；当第一条指令执行到执行阶段时，第二条指令执行到了译码阶段，此时第三条指令开始做取指令阶段，然后以此类推。这样，当整条流水线填充满之后，即执行到了第5条指令，那么对于后续指令而言，处理每一条指令的时间均只需要一个周期。

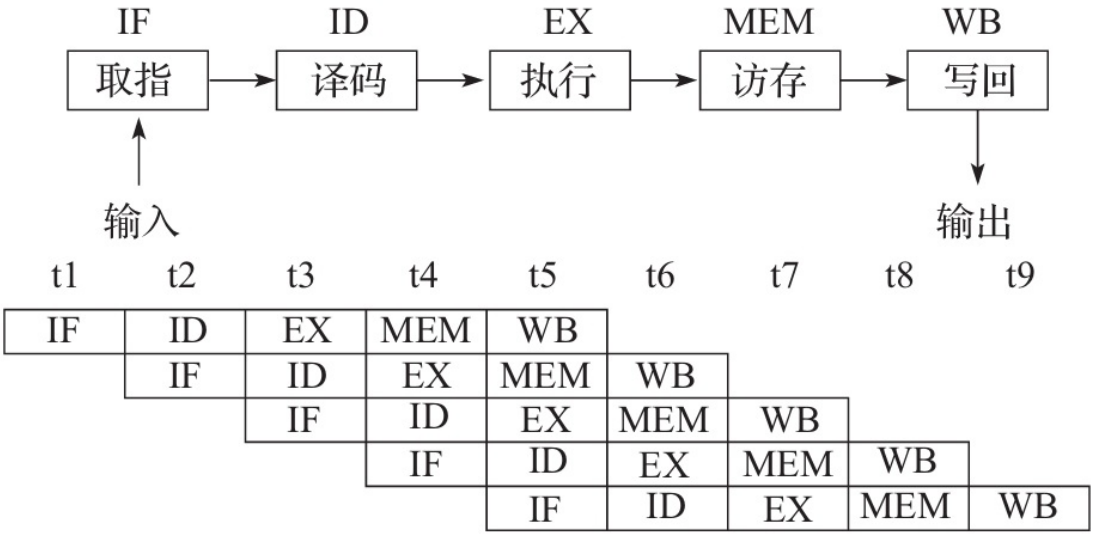


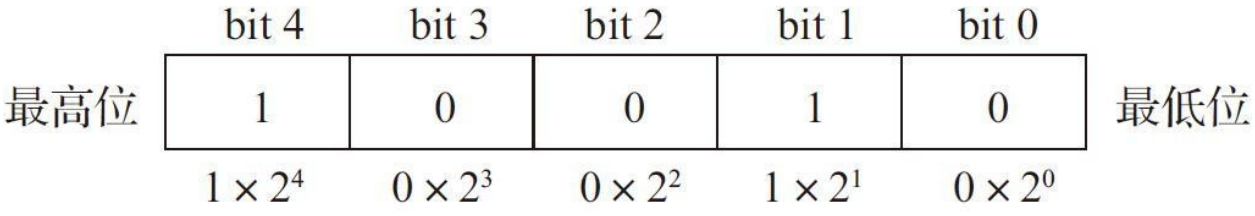
图2-2　处理器执行流水线

这里需要注意的是，并不是每条指令都需要访存操作，只有当需要对外部存储器做读写操作时才会动用访存执行单元。然而大部分指令都需要写回寄存器操作，即便像一条用于比较大小的指令，或一条系统中断指令，它们也会影响状态寄存器。当然，很多处理器会有空操作（NOP）指令，它仅仅占用一个时钟周期，而不会对除了指令指针寄存器以外的任何寄存器产生影响。

2.2 整数在计算机中的表示

我们日常用的整数都是十进制数（Decimal），也就是我们通常所说的逢十进一。因为我们人类有十根手指，所以自然而然地会想到采用十进制的计数和计算方式。然而，现在几乎所有计算机都采用二进制数（Binary）编码方式，所以我们日常所用到的整数如果要用计算机来表示的话，需要表示成二进制的方式。

二进制数则是逢二进一，所以在整串数中只有0和1两种数字。比如，十进制数0，对应二进制为0；十进制数1，对应二进制数1；十进制数2，对应二进制数10；十进制数3，对应二进制数11。因此，对于非负整数而言，二进制数第n位（n从0开始计）如果是1，那么就对应十进制数的2^n，然后每个位计算得到的十进制数再依次相加得到最终十进制数的值。比如，一个5位二进制数10010，最低位为最右边的位，记为0号位，数值为0；最高位为最左边的位，记为4号位，数值为1。那么它所对应的十进制数为：$2^4+2^1=18$。因为该二进制数除了4号位和1号位为1之外，其余位都是0，因此0乘以2^n肯定为0。图2-3为二进制数10010换算成十进制数的方法图。



二进制数10010对应的十进制数值为：

$1 \times 2^4+0 \times 2^3+0 \times 2^2+1 \times 2^1+0 \times 2^0 = 18$

图2-3　5位二进制数对应十进制的计算

在计算机术语中，把二进制数中的某一位数又称为一个比特（bit）。比特这个单位对于计算机而言，在度量上是最小的单位。除了比特之外，还有字节（byte）这个术语。一个字节由8个比特构成。在某些单片机架构下还引入了半字节（nybble或nibble）这个概念，表示4个比特。然后，还有字（word）这个术语。字在不同计算机架构下表示的含义不同。在x86架构下，一个字为2个字节；而在ARM等众多32位RISC体系结构下，一个字表示为4个字节。随着计算机带宽的提升，能被处理器一次处理的数据宽度也不断提升，因此出现了双字（double word）、四字（quad word）、八字（octa word）等概念。双字的宽度为2个字，四字宽度为4个字，所以它们在不同处理器体系结构下所占用的字节个数也会不同。

我们上面介绍了非负整数的二进制表达方法，那么对于负数，二进制又该如何表达呢？在计算机中有原码和补码两种表示方法，而最为常用的是补码的表示方法。下面我们分别对原码和补码进行介绍。

2.2.1　原码表示法

对于无正负符号的原码，其二进制表达如上节所述。而对于含有正负符号的原码，其二进制表示含有一位符号位，用于表示正负号。一般都是以二进制数的最高有效位（即最左边的比特）作为符号位，其余各位比特表示该数的绝对值大小。比如，十进制数6用一个8位的原码表示为

00000110；如果是-6，则表示为10000110。二进制的原码表示示例如图2-4所示。

图2-4　二进制数的原码表示

原码的表示非常直观，但是对于计算机算术运算而言就带来了许多麻烦。比如，我们用上述的6与-6相加，即00000110+10000110，结果为10001100，也就是十进制数-12，显然不是我们想要的结果。所以，如果某个处理器用原码表示二进制数，那么它参与加减法的时候必须对两个操作数的正负符号加以判断，然后再判定使用加法操作还是减法操作，最后还要判定结果的正负符号，可谓相当麻烦。所以，当前计算机的处理器往往采用补码的方式来表达带符号的二进制数。

2.2.2　补码表示法

正由于原码含有上述缺点，所以人们开发出了另一种带符号的二进制

码表示法——补码。补码与原码一样，用最高位比特表示符号位，其余各位比特则表示数值大小。如果符号位为0，说明整个二进制数为正数或零；如果为1，那么表示整个二进制数为负数。当符号位为0时，二进制补码表示法与原码一模一样，但是当符号位为负数时，情况就完全不同了。此时，对二进制数的补码表示需要按以下步骤进行：

1）先将该二进制数以绝对值的原码形式写好；

2）对整个二进制数（包括符号位），每一个比特都取反。所谓取反就是说，原来一个比特的数值为0时，则要变1；为1时，则要变0。

变换好之后，将二进制数做加1计算，最终结果就是该负数的补码值了。

下面我们还是用6来举例，+6的二进制补码跟原码一样，还是00000110。而-6的计算过程，按照上述流程如下：

1）先将-6用绝对值+6的形式表示：00000110；

2）对每个比特位取反，包括符号位在内，得到：11111001；

3）将变换好的数做加1计算，最终得到：11111010。

由于二进制补码的表示与通常我们可直接读懂的二进制数的表示有很大不同，所以给定一个二进制补码，我们往往需要先获得其绝对值大小才能知道它的具体数值。获得其绝对值的过程为：先判定符号位，如果符号

位为0，那么就以通常的二进制数表示法来读即可。如果符号位为1，那么就以上述同样的过程得到其对应的绝对值。比如，如果给定11111010这个二进制数，我们看到最高位符号位为1，说明是负数，我们就以上述过程来求解：

1）先将该二进制数每个比特做取反计算，得到：00000101；

2）然后将变换得到的值做加1计算，最终获得：00000110。

所以11111010的绝对值为00000110，即6。

对于补码表示，我们已经知道最高位比特表示符号位，其余的表示具体数值。但是这里有一个特殊情况，即符号位为1，其余位比特为都为0的情况。比如一个8位二进制补码：10000000，此时它的值是多少？因为我们通过上述流程，求得其绝对值的大小也是10000000，所以当前大部分计算机处理器的实现将它作为-128，但估计仍然有一些处理器会把它作为-0。因为C语言标准中对于数值范围的表示已经明确表示出8位带符号的整数范围可以是-128到+127，也可以是-127到+127，但最小值不得大于-127，最大值不得小于+127。第5章会有更详细的描述。

补码的这种表示法的优点就是可以无视符号位，随意进行算术运算操作。比如，像我们上面所举的例子：6+（-6），计算结果：

00000110+11111010=00000000

最后，上述计算结果的最高位符号位所产生的进位被丢弃（在处理器

中可能会设置相应的进位标志位）。我们自己计算的话也非常方便，在计算过程中，无需关心两个二进制补码的正负数的情况，也无需关心符号位所产生的影响。我们只需要像计算普通二进制数一样去计算即可。把最终的计算结果拿出来判断，是正数还是负数。当然，二进制补码会产生溢出情况，比如两个8位二进制补码加法：

120+50=01111000+00110010=10101010

然而，这个数并不是170，而是-86。首先，170已经超出了带符号8位二进制数可表示的最大范围了；其次，最高位变为1，用补码表示来讲就是负数表示形式。所以，这两个正数的加法计算就产生了负数结果，这种现象称为上溢。如果我们要避免在计算过程中出现上溢情况，需要用更高位宽的二进制数来表示，以提升精度。比如，如果我们将上述加法用16位二进制数表示，那么就不会有上溢问题了。

另外，在C语言标准中没有明确规定C语言编译器的实现以及运行时环境必须采用哪种二进制编码方式，而是对整数类型标明最大可表示的数值范围。目前大部分C语言实现都是对带符号整数采用补码的表示方式。这些会在第5章做进一步讲解。

2.2.3 八进制数与十六进制数

上面我们对二进制数编码形式做了比较详细的介绍。我们在编写程序

或者查看一些计算机相关的技术文档时常常还会碰到八进制数与十六进制数的表示，尤其是十六进制数用得非常多。下面我们就简单介绍一下这两种基数（radix）的表示方法。

这里跟各位再分享一个术语——基数。基数也就是我们通常所说的，某一个数用多少进制表达。对于像“01001000是几进制数”这种话，如果用更专业的表达方式来说的话就是，“01001000的基数是几”。基数为2就是二进制；基数为10则是十进制。

八进制数是逢八进一，因此每位数的范围是从0～7。八进制数转十进制数也很简单，我们可以用二进制数转十进制数类似的方法来炮制八进制数转十进制数——以一个八进制数每位数值作为系数，然后乘以8^n，然后计算得到的结果全都相加，最后得到相应的十进制数。其中，n表示当前该位所对应的位置索引（同样以0开始计）。比如，八进制数5271对应的十进制数的计算过程如图2-5所示。

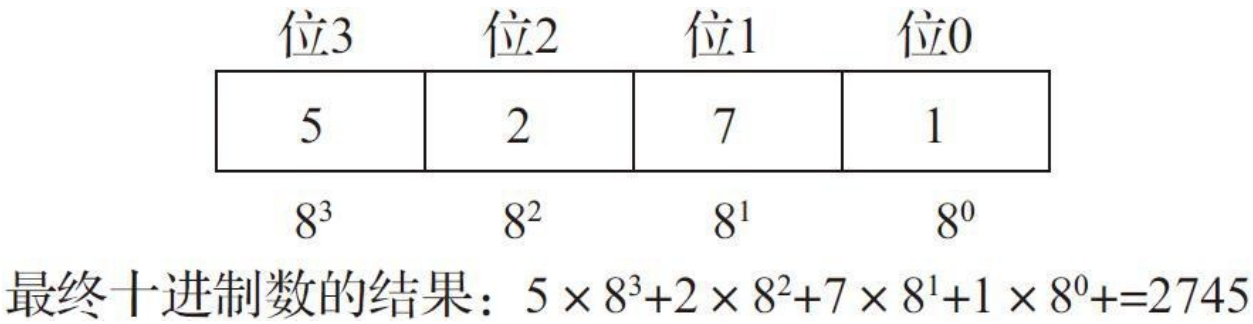


图2-5　八进制数转十进制数

八进制数对应于二进制数的话正好占用3个比特（范围从000～111），一般在通信领域以及信息加密等领域会用到八进制编码方式。而十六进制

数比八进制数用得更多，因为十六进制数正好占用4个比特，即4位二进制数（范围从0000～1111）。4个比特相当于半个字节。所以，无论是开发工具还是程序调试工具，一般都会用十六进制数来表示计算机内部的二进制数据，这样更易读，而且也更省显示空间（因为一个字节原本需要8位二进制数，而十六进制数只要两位即可表示）。下面就介绍一下十六机制数的表示方法。

十六进制数逢十六进一，因此每一位数的范围是从0到15。由于我们通常在数学上所用的十进制数无法用一位来表示10～15这6个数，因而在计算机领域中，我们通常用英文字母A（或小写a）来表示10；B（或小写b）来表示11；C（或小写c）来表示12；D（或小写d）来表示13；E（或小写e）来表示14；F（或小写f）来表示15。十六机制数转十进制数的方式与八进制数转十进制数类似——以一个十六进制数每位数值作为系数，然后乘以16^n，然后计算得到的结果全都相加，最后得到相应的十进制数。其中，n表示当前位所对应的位置索引（同样以0开始计）。比如，一个4位十六进制数C0DE的计算过程如图2-6所示：

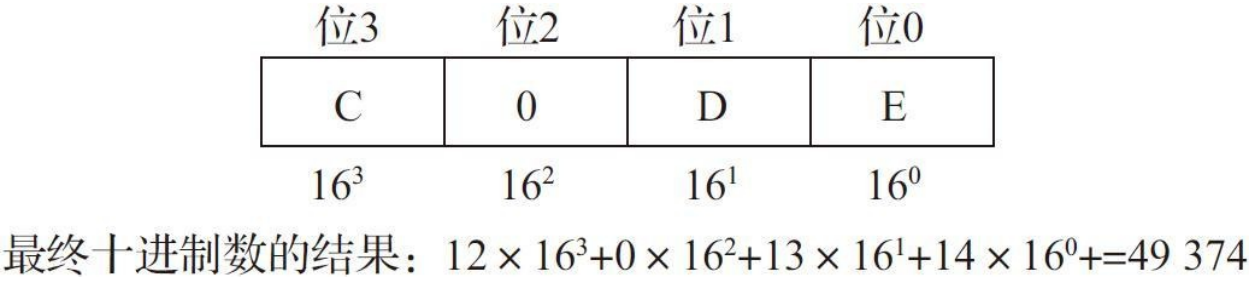


图2-6　十六进制数转十进制数

上述4位十六进制数C0DE，倘若用二进制数表示，则为：

1100000011011110。可见，用十六进制数表示要简洁得多，而且换算成十进制数也相对比较容易，尤其对于一个字节长度的整数来说。为了能更快速地换算二进制数、十进制数与十六进制数，请各位读者务必熟记下表:

表2-1　二进制数、十进制数与十六进制数的换算表

二进制数	十进制数	十六进制数
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

习惯上，用0或0o打头的数表示八进制数，0x打头的数表示十六进制数。比如，0123、0777表示八进制数；0x123，0xABCD表示十六进制数。

2.3　浮点数在计算机中的表示

当前主流处理器一般都能支持32位的单精度浮点数与64位的双精度浮点数的表示和计算，并且能遵循IEEE754-1985工业标准。现在此标准最新的版本是2008，其中增加了对16位半精度浮点数以及128位四精度浮点数的描述。C语言标准引入了一个浮点模型，可用来表达任意精度的浮点数，尽管当前主流C语言编译器尚未很好地支持半精度浮点数与四精度浮点数的表示和计算。关于C语言标准对浮点数的描述，我们稍后将在5.2节做更详细的介绍。

为了更好地理解IEEE754-1985中规格化（normalized）浮点数的表示法，我们先来介绍一下浮点数用一般二进制数的表示方法。一个浮点数包含了整数部分和尾数（即小数）部分。整数部分的表示与我们之前所讨论过的一样，第n位就表示2^n，n从0开始计。而尾数部分则是第m位表示2^{-m}，m从1开始计。对于一个0101.1010的二进制浮点数对应十进制数的计算如图2-7所示：

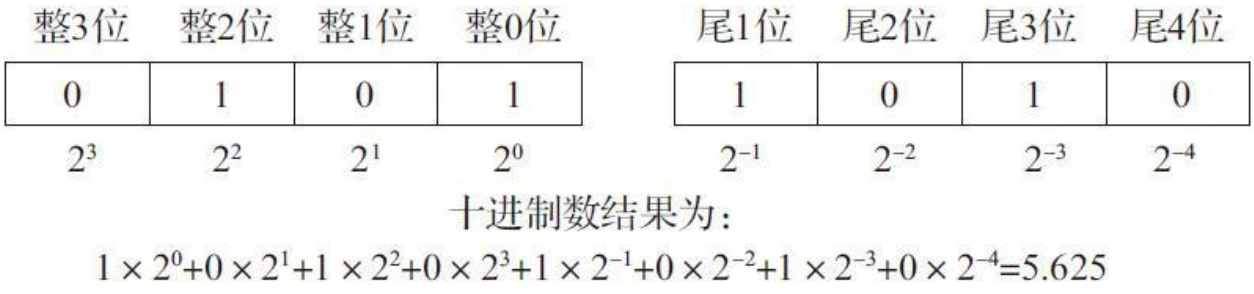


图2-7　二进制浮点数转十进制数

图2-7中，整i位即表示第i位整数；尾i位即表示第i位尾数。其中，第3位整数为最高位整数；第4位尾数表示最低位尾数。对二进制浮点数的表示有了概念之后，我们就可以看IEEE754-1985标准中对规格化浮点数的描述了。IEEE754-1985对32位单精度与64位双精度两种精度的浮点数进行描述。32位单精度浮点可表示的数值范围在$\pm1.18\times10^{-38}$到$\pm3.4\times10_{38}$，大约含有7位十进制有效数；64位双精度浮点可表示的数值范围在$\pm2.23\times10^{-308}$到$\pm1.80\times10^{308}$，大约含有15位十进制有效数。我们看到IEEE定义的浮点数的绝对值范围可以是一个远大于1的数，也可以是一个大于零但远小于1的数，即它的小数精度是可浮动的，所以称之为浮点数。如果说是定点数的话，它也可表示一个小数，但是其整数位数与小数位数的精度都是固定的。比如一个16.16的定点数表示整数部分采用16个比特，尾数部分也采用16个比特。而对于一个32位浮点数来说，既能使用16.16的格式，也能使用30.2的格式（即30个比特表示整数，2个比特表示尾数）或其他各种形式。而IEEE754-1985对规格化单精度浮点数的格式如下定义：

1）1位符号位，一般是最高位（31位），表示正负号。0表示正数，1表示负数。

2）8位指数位，又称阶码，位于23到30位。（阶码的计算后面会详细介绍。）

3）23位尾数，位于0到22位。

我们下面举一个实际的例子来详细说明一个十进制小数5.625如何表示

成IEEE754标准的规格化32位单精度浮点数。

1）5.625是一个正数，所以符号位为0，即第31位为0。

2）我们将5.625依照图2-7那样写成一般小数的表示法——0101.101。

3）我们将此二进制浮点数用科学计数法来表示，使得二进制整数位为最高位的1。这里最高位为1的比特是从左往右数是第二个比特，所以将小数点就放到该比特的后面，得到1.01101×2^2。二进制数的科学记数法，底数的值显然就是2。

4）此时，我们能看到尾数部分是小数点后面的那串二进制数，即01101，而指数为2。现在我们来求阶码。阶码用的是中经指数偏差（exponent bias）处理后的指数，即用上述得到的指数加上偏差值所求得的和。IEEE754在单精度浮点中规定，偏差值为127。所以本例中，阶码部分为2+127=129，用二进制数表示就是10000001。

5）尾数部分从大到小照抄，低位的用0填充即可，所以这里的尾数部分二进制数为：01101000000000000000000。

6）将整个处理完的二进制数串起来获得：0（符号位）10000001（阶码）01101000000000000000000（尾数），用十六进制数表达就是：40B40000。

十进制小数转64位双精度浮点数的方法与上述雷同，只不过阶码用11位比特来表示，尾数则用52位比特表示，而偏差值则规定为1023。

2.4 地址与字节对齐

由于C语言是一门接近底层硬件的编程语言，它能直接对存储器地址进行访问（当前大部分处理器在操作系统的应用层所访问到的逻辑地址，而部分嵌入式系统由于不含带存储器管理单元，因此可直接访问物理地址）。在计算机中，所谓“地址”就是用来标识存储单元的一个编号，就好比我们住房的门牌号。没有门牌号，快递就没法发货；如果门牌号记错了，那么快递就会把货物送错地方。计算机中的地址也是一样，我们为了要访问存储器中特定单元的一个数据，那么我们首先要获悉该数据所在的地址，然后我们通过这个地址来访问它。访问存储器，我们也简称为“访存”（Memory Access）。访问地址，我们也简称为“寻址”（Addressing）。我们在图2-1中也看到，一般计算机架构中都会有地址总线和数据总线。CPU先通过地址总线发送寻址信号，以指定所要访问存储器单元的地址。然后再通过数据总线向该地址读写数据，这样就完成了一次访存操作。这好比于快递送货，我们先打电话告诉快递通信地址，然后快递员把货送到该地址（写数据），或者去该地址拿货（读数据）送到别家。

一般对于32位系统来说，处理器一次可访问1个（8比特）字节、2个字节或4个字节。当访问单个字节时，对CPU不做对齐限制；而当访问多个字节时，比如要访问N个字节，由于计算机总线设计等诸多因素，要求CPU所访问的起始地址满足N个字节的倍数来访问存储器。如果在访问存储器时没有按照特定要求做字节对齐，那么可能会引发访存性能问题，甚至直

接导致寻址错误而引发异常（引发异常后通常会导致当前应用意外退出，在嵌入式系统中可能就直接死机或复位）。

下面我们给出一张图2-8来描述，看看一般对32位系统而言如何正确地做到访存字节对齐。

图2-8展示了如何正确对齐访问1个字节、2个字节和4个字节的情况。图中画出了6个存储单元内容，地址低16位从0x1000到0x1005，每个存储单元为1个字节。对于仅访问1个字节的情况，图2-8所有地址都能直接访问并满足字节对齐的情况。对于一次访问2个字节的情况，要满足对齐要求，只能访问0x1000、0x1002、0x1004等必须要能被2整除的地址。对于一次访问4字节的情况，要满足对齐要求，则只能访问0x1000、0x1004等必须要能被4整除的地址。

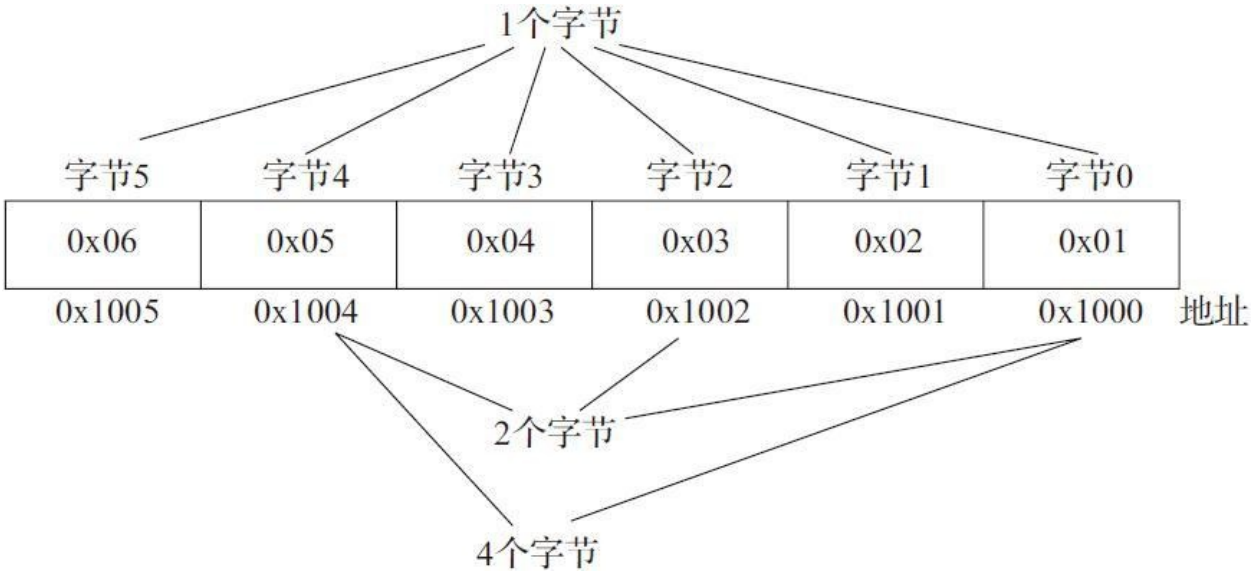


图2-8　字节对齐

然而，并不是说要访问多少字节，就必须要保证访问能被多少整除的地址才能满足对齐要求。如果一次访问8字节，对于32位系统而言，通过32位通用目的寄存器来读写存储器的话，某些CPU会自动将8字节的访存分为两次进行操作，每次为4字节，因此只要保证4字节对齐就能满足对齐要求。这些都根据特定的处理器来做具体处理。

就笔者用过的一些处理器而言，像x86、ARM等处理器，当访存不满足对齐要求时并不会引发总线异常，但是访问性能会降低很多。因为原本可一次通信的数据传输可能需要拆分为多次，并且前后还要保证数据的一致性，所以还可能会有锁步之类的操作。而像Blackfin DSP则会直接引发总线异常，导致整个系统的崩溃（如果不对此异常做处理的话）。另外，像ARMv5或更低版本的处理器，在对非对齐的存储器地址进行访问时，CPU会先自动向下定位到对齐地址，然后通过向右循环移位的方式处理数据，这就使得传输数据并不是原本想一次传输的数据内容，也就是说写入的或读出的数据是失真的。比如，根据图2-8所示内容，如果我们要对一款ARM7EJ-S处理器（ARMv5TEJ架构）从地址0x1002读4字节内容，那么实际获取到的数据为0x02010403；而在x86架构或ARMv7架构的处理器下，则能获得0x06050403。

2.5　字符编码

我们从2.2节到2.4节讲述的都是数值信息（整数与浮点数），本小节我们将讨论字符信息。在计算机中我们所处理的字符信息，即文本信息（包括数字、字母、文字、标点符号等）是以一种特定编码格式来定义的。为了使世界各国的文本信息能够通用，就需要对字符编码做标准化。我们现在最常用也最基本的字符编码系统是ASCII码（American Standard Code for Information Interchange，美国信息交换标准码）。ASCII码定义每个字符仅占一个字节，可表示阿拉伯数字0～9、26个大小写英文字母，以及我们现在在标准键盘上能看到的所有标点符号、一些控制字符（比如换行、回车、换页、振铃等）。ASCII码最高位是奇偶校验位，用于通信校验，所以真正有编码意义的是低7个比特，因此只能用于表示128个字符（值从0～127）。由于ASCII是美国国家标准，所以后来国际化标准组织将它进行国际标准化，定义为了ISO/IEC 646标准。两者所定义的内容是等价的。

ISO/IEC 646对于英文系国家而言是基本够用了，但是对于拉丁语系、希腊等国家来说就不够用了。所以后来ISO组织就把原先ISO/IEC 646所定义字符的最高位也用上了，这样就又能增加128个不同的字符，发布了ISO/IEC 8859标准。然而，欧洲大陆虽小，但国家却有数百个，128种扩展字符仍然不够用。因此后来就在8859的基础上，引入了8859-n，n从1～16，每一种都支持了一定数量的不同的字母，这样基本能满足欧美国家的文字表示需求。当然，有些国家之间仍然需要切换编码格式，比如

ISO/IEC8859-1的语言环境看8859-2的就可能显示乱码，所以，还得切换到8859-2的字符编码格式下才能正常显示。

而在中国大陆，我们自己也定义了一套用于显示简体中文的字符集——GB2312。它在1981年5月1日开始实施，是中国国家标准的简体中文字符集，全称为《信息交换用汉字编码字符集·基本集》。它收录了6763个汉字，包括拉丁字母、希腊字母、日语假名、俄语和蒙古语用的西里尔字母在内的682个全角字符。然后又出现了GBK字符集，GBK1.0收录了21886个符号，其中汉字就包含了21003个。GBK字符集主要扩展了繁体中文字。由于像GB2312与GBK能表示成千上万种字符，因此这已经远超1个字节所能表示的范围。它们所采用的是动态变长字节编码，并且与ASCII码兼容。如果表示ASCII码部分，那么仅1个字节即可，并且该字节最高位为0。如果要表示汉字等扩展字符，那么头1个字节的最高位为1，然后再增加一个字节（即用两个字节）进行表示。所以，理论上，除了第1个字节的最高位不能动之外，其余比特都能表示具体的字符信息，因而最多可表示2^7+2^{15}=32896种字符。

当然，正由于GB2312与GBK主要用于亚洲国家，所以当欧美国家的人看到这些字符信息时显示的是乱码，他们必须切换到相应的汉字编码环境下看才能看到正确的文本信息。为了能真正将全球各国语言进行互换通信，出现了Unicode（Universal Character Set，UCS）标准。它对应于编码标准ISO/IEC 10646。Unicode前后也出现了多个版本。早先的UCS-2采用固定的双字节编码方式，理论上可表示2^{16}=65536种字符，因此极大地涵盖了

各种语言的文字符号。

不过后来，标准委员会意识到，对于像希伯来字母、拉丁字母等压根就不需要用两个字节表示，而且定长的双字节表示与原有的ASCII码又不兼容，因此后来出现了现在用得更多的UTF-8编码标准。UTF-8属于变长的编码方式，它最少可用1个字节表示1个字符，最多用4个字节表示1个字符，判别依据就是看第1个字节的最高位有多少个1。如果第1个字节的最高位是0，那么该字符用1个字节表示；最高3位是110，那么用2个字节表示；最高4位是1110，那么用3个字节表示；最高位是11110，那么该字符由4个字节来表示。所以UTF-8现在大量用于网络通信的字符编码格式，包括大多数网页用的默认字符编码也都是UTF-8编码。尽管UTF-8更为灵活，而且也与ASCII码完全兼容，但不利于程序解析。所以现在很多编程语言的编译器以及运行时库用得更多的是UTF-16编码来处理源代码解析以及各类文本解析，它与之前的UCS-2编码完全兼容，但也是变长编码方式，可用双字节或四字节来表示一个字符。如果用双字节表示UTF-16编码的话，范围从0x0000到0xD7FF，以及从0xE000到0xFFFF。这里留出0xD800到0xDFFF，不作为具体字符的编码表示，而是用于四字节编码时的编码替换。当UTF-16表示0x10000到0x10FFFF之间的字符时，先将该范围内的值减去0x10000，使得结果落在0x00000到0xFFFFF范围内。然后将结果划分为高10位与低10位两组。将低10位的值与0xDC00相加，获得低16位；高10位与0xD800相加，获得高16位。比如，一个Unicode定义的码点（code point）为0x10437的字符，用UTF-16编码表示的步骤如下。

1）先将它减去0x10000——0x10437-0x10000=0x0437。

2）将该结果分为低10位与高10位，0x0437用20位二进制表示为00000000010000110111，因此高10位是0000000001=0x01；低10位则是0000110111，即0x037。

3）将高10位与0xD800相加，得到0xD801；将低10位与0xDC00相加，获得0xDC37。因此最终UTF-16编码为0xD801DC37。

我们看到，尽管UTF-16也是变长编码表示，但是仅低16位就能表示很多字符符号，况且即便要表示更广范围的字符，也只是第二种四字节的表示方法，这远比UTF-8四种不同的编码方式要简洁很多。因此，UTF-16用在很多编程语言运行时系统字符编码的场合比较多。像现在的Java、Objective-C等编程语言环境内部系统所表示的字符都是UTF-16编码方式。

另外，现在还有UTF-32编码方式，这一开始也是Unicode标准搞出来的UCS-4标准，它与UCS-2一样，是定长编码方式，但每个字符用固定的4字节来表示。不过现在此格式用得很少，而且HTML5标准组织也公开声明开发者应当尽量避免在页面中使用UTF-32编码格式，因为在HTML5规范中所描述的编码侦测算法，故意不对它与UTF-16编码做区分。

2.6　大端与小端

现代计算机系统中含有两种存放数据的字节序：大端（Big-endian）和小端（Little-endian）。所谓大端字节序是指在读写一个大于1个字节的数据时，其数据的最高字节存放在起始地址单元处，数据的最低字节存放在最高地址单元处。所谓小端字节序是指在读写一个大于1个字节的数据时，其数据的最低字节存放在起始地址单元处，而数据的最高字节存放在最高地址单元处。比如，我们要在地址0x00001000处存放一个0x04030201的32位整数，其大端、小端存放情况如图2-9所示。

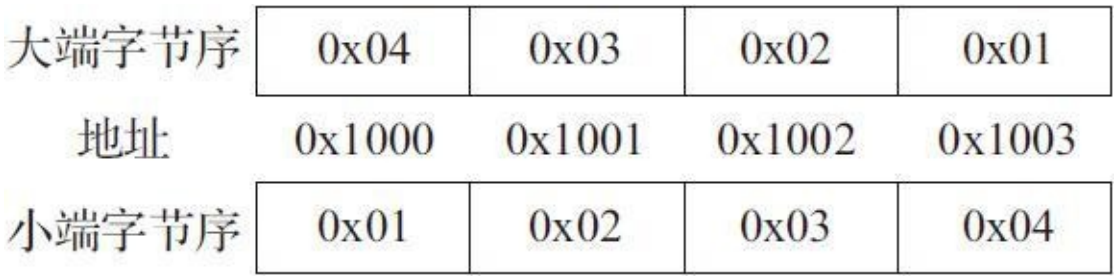


图2-9　大端与小端

当前，通用桌面处理器以及智能移动设备的处理器一般都用小端字节序。通信设备中用大端字节序比较普遍。

本书后续所要叙述的内容中，若无特殊说明，都是基于小端字节序进行描述。

2.7 按位逻辑运算

按位逻辑运算在计算机编程中会经常涉及，这些运算都是针对二进制比特进行操作的。所谓的“按位”计算就是指对一组数据的每个比特逐位进行计算，并且对每个比特的计算结果不会影响其他位。常用的按位逻辑运算包括“按位与”、“按位或”、“按位异或”以及“按位取反”四种。下面将分别介绍这4种运算方式。

1）按位与：它是一个双目操作，需要两个操作数，在C语言中用&表示。两个比特的按位与结果如下：

0&0=0；　0&1=0；　1&0=0；　1&1=1

也就是说，两个比特中如果有一个比特是0，那么按位与的结果就是0，只有当两个比特都为1的时候，按位与的结果才为1。比如，对两个字节01001010和11110011进行按位与的结果为01000010。按位与一般可用于判定某个标志位是否被设置。比如，我们假定处理一个游戏手柄的按键事件，用一个字节来存放按键被按下的标志，前4个比特分别表示“上”、“下”、“左”、“右”。比特4表示按下了“A”键，比特5表示按下了“B”键，比特6表示按下了“X”键，比特7表示按下了“Y”键。那么当我们接收到二进制数01010100时，说明用户同时按下了“左”方向键、“A”键和“X”键。那么我们判定按键标志时可以通过按位与二进制数1来判定是否按下了“上”键，按位与二进制数10做按位与操作来判定是否按下

了“下”键，跟二进制数100做与操作来判定是否按下了“左”键，以此类推。如果按位与的结果是0，说明当前此按键没有被按下，如果结果不为零，说明此按键被按下。

2）按位或：它是一个双目操作符，需要两个操作数，在C语言中用“|”表示。两个比特的按位或结果如下：

0|0=0；　0|1=1；　1|0=1；　1|1=1

也就是说，只要有一个比特的值是1，那么按位或的结果就是1，只有当两个比特的值都为0的时候，按位或的结果才是0。比如，对于两个字节01001010和11110011进行按位或的结果为11111011。按位或一般可用于设置标志位。就如同上述例子，如果用户按下了“上”键，那么系统底层会将最低位设置为1；如果用户按下了“Y”键，那么系统底层会将最高位设置为1。随后系统会将这串信息发送到应用UI层。

3）按位异或：它是一个双目操作，需要两个操作数，在C语言中用^表示。两个比特的按位异或结果如下

0^0=0；　0^1=1；　1^0=1；　1^1=0

也就是说，如果两个比特的值相同，那么按位异或的结果为0，不同为1。比如，对于两个字节01001010和11110011进行按位或的结果为10111001。按位异或适用于多种场景，比如我们用一个输入比特与1进行异或就可以反转该输入比特的值，输入为0，那么结果为1；输入为1，那么结

果为0。任一比特与0异或，那么结果还是原比特的值。按位异或跟按位与和按位或不同，它可以对数据信息进行叠加组合。因为给定任一比特，对于另外一个比特的输入，不同的输入值对应不同的输出，所以我们通过异或能还原信息。比如，我们有两个整数a和b，我们设c=a^b。对于c，我们可以通过c^a重新得到b，也可以通过c^b来重新得到a。所以异或在信息编码、数据加密等技术上应用得非常多。

4）按位取反：它是一个单目操作，只需要一个操作数，在C语言中用~表示。一个比特的按位取反结果如下：~0=1；~1=0。比如，对一个字节01001010进行按位取反的结果为10110101。

2.8 移位操作

现代处理器的计算单元中一般都会包含移位器。移位器往往能执行算术左移（Arithmetic Shift Left）、算术右移（Arithmetic Shift Right）、逻辑左移（Logical Shift Left）、逻辑右移（Logical Shift Right）、循环右移（Rotational Shift Right）这些操作。

下面我们将分别介绍这些移位操作，这里需要提醒各位的是，移位操作一般总是对整数数据进行操作，并且移入移出的都是二进制比特。然而，不同的处理器架构对移位操作的实现可能会有一些不同。比如，如果对一个32位寄存器做移位操作，倘若指定要移动的比特数超过了31，那么在x86处理器中是将指定的比特移动位数做模32处理（也就是求除以32的余数，比如左移32位相当于左移0位、右移33位相当于右移1位）；而在ARM、AVR等处理器中，对一个32位的整数做左移和逻辑右移超出31位的结果都将是零。

2.8.1 算术左移与逻辑左移

由于算术左移与逻辑左移操作基本是相同的，仅仅对标志位的影响有些区别，所以合并在一起讲。左移的操作步骤十分简单，假设我们要左移N位，那么先将整数的每个比特向左移动N位，然后空出的低N位填零。图2-10展示了对一个8位整数分别做左移1位与左移2位的过程。

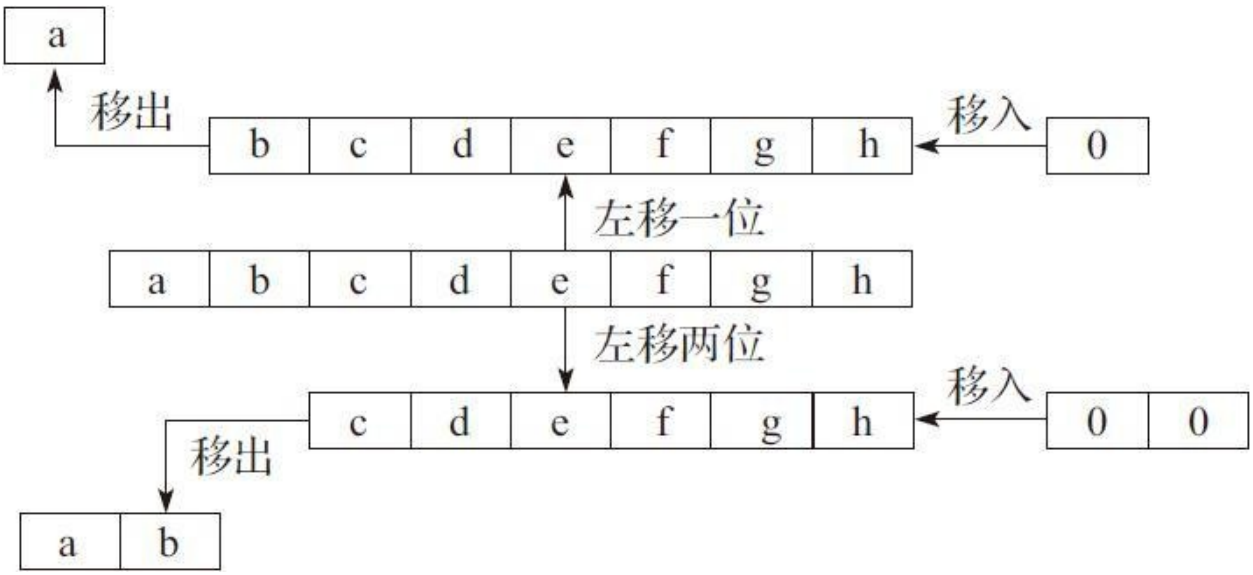


图2-10　算术左移与逻辑左移

图2-10中间由小写字母a～h构成的方格图即表示一个8位二进制整数，每个小写字母表示一比特，并且字母a作为最高位比特，字母h作为最低位比特。左移1位后，原来的8位二进制数就变成了bcdefgh0；左移2位后，原来的8位二进制数就变成了cdefgh00。

2.8.2　逻辑右移

逻辑右移的操作步序是：先将整数的每一个比特向右移动N位，然后高N位用零来填补。图2-11展示了一个8位二进制整数分别逻辑右移1位和2位的过程。

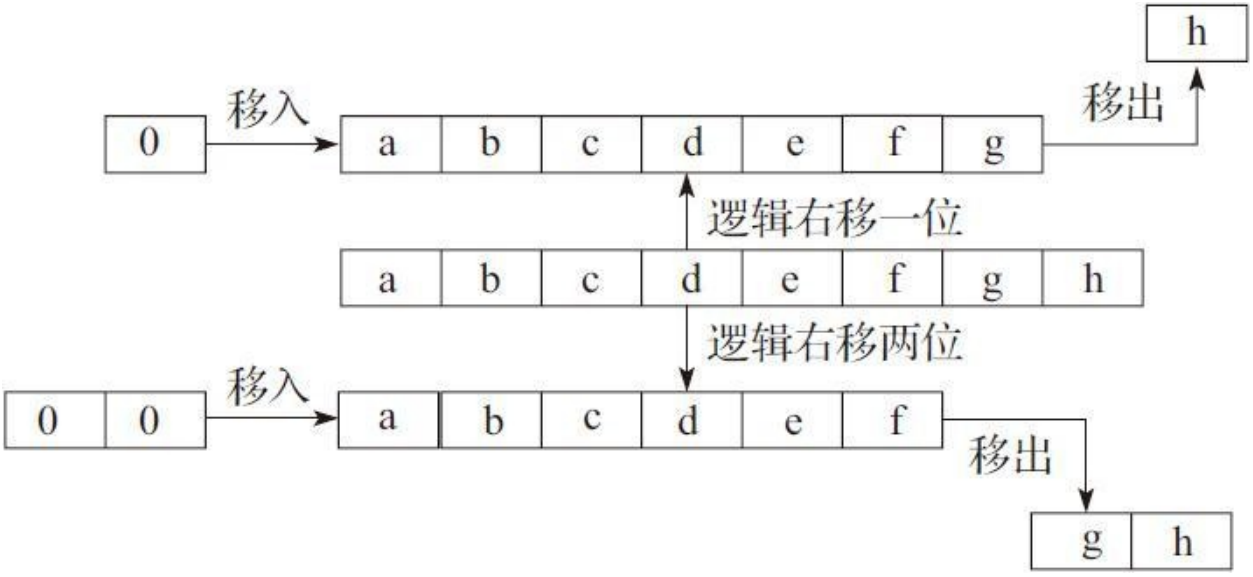


图2-11　逻辑右移

图2-11中间由小写字母a～h构成的方格图即表示一个8位二进制整数，每个小写字母表示一位比特，并且字母a作为最高位比特，字母h作为最低位比特。将原始二进制8位数据逻辑右移1位后，二进制数据变为0abcdefg；逻辑右移2位后，二进制数据变为00abcdef。

2.8.3　算术右移

算术右移与逻辑右移类似，只不过移出N位之后，高N位不是用零来填充，而是根据原始整数的最高位，如果原始整数的最高位为1，那么移位后的高N位用1来填充；如果是0，则用0来填充。图2-12展示了一个8位二进制整数分别算术右移1位和2位的过程。

图2-12　算术右移

图2-12中间由小写字母a～h构成的方格图，即表示一个8位二进制整数，每个小写字母表示一位比特，并且字母a作为最高位比特，字母h作为最低位比特。将原始8位二进制整数算术右移1位之后，该二进制数变为aabcdefg；将它算术右移2位之后，变为aaabcdef。

2.8.4　循环右移

循环右移的步序是：先将原始二进制整数右移N位，移出的N位依次放入到高N位。图2-13展示了将一个8位二进制整数分别循环右移1位和2位的过程。

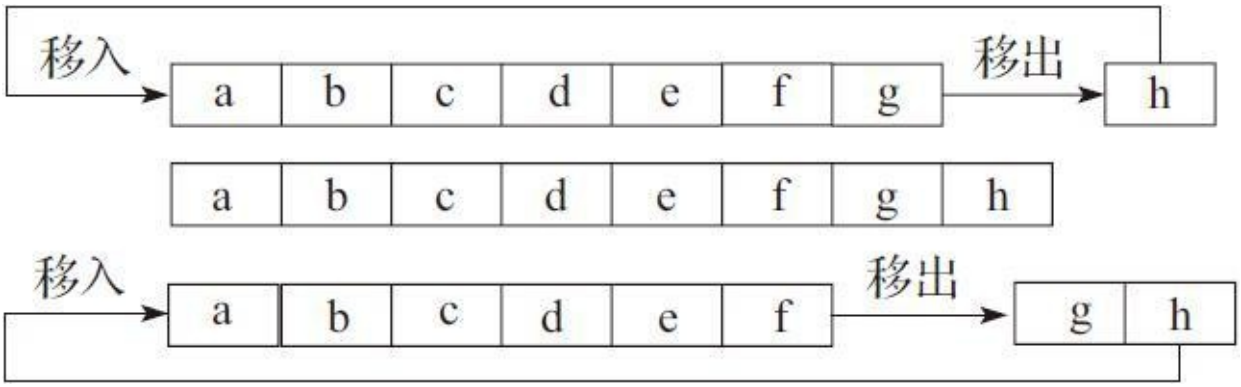


图2-13　循环右移

图2-13中间由小写字母a～h构成的方格图，即表示一个8位二进制整数，每个小写字母表示一位比特，并且字母a作为最高位比特，字母h作为最低位比特。将原始8位二进制整数循环右移1位之后，该二进制整数变为habcdefg；将它循环右移2位之后，该二进制整数变为ghabcdef。

2.9 本章小结

本章大致介绍了计算机体系结构以及程序执行的大致流程，然后描述了整数以及浮点数在计算机中的存储方式，之后还介绍了地址与字节对齐、字符编码、处理器大端与小端字节序，以及按位逻辑运算和移位操作。由于这些知识都是学习C语言必备的，C语言中有相关语法与这些概念对应，所以各位最好能先理解、掌握这些基本知识，这样对后续学习C语言将有很大帮助。

第3章　C语言编程的环境搭建

我们在第2章讲述了学习C语言所必需的一些预备知识。本章将给大家介绍常用桌面操作系统下的C语言环境搭建。这里所讲述的C语言编译器以及集成开发环境（IDE）都是可合法免费下载的，本书不鼓励各位使用盗版或破解软件，所以下面会列出下载这些合法免费软件的官方链接，大家把编程环境搭建完之后即可上机实践编程。

3.1 Windows操作系统下搭建C语言编程环境

Windows操作系统下默认不自带任何C语言编译器，大家必须从网上下载自己所需要的C语言编译器。如果各位想通过C语言开发Windows系统平台相关的应用，或者主要想在Windows平台对C语言程序进行调试，那么往往首选Visual Studio Community。这款开发环境是免费的，里面自带了微软自家的C语言编译器——简称为MSVC。不过当前MSVC无法支持最新的C11标准新特性，而且即便是C99标准也是支持得比较有限，所以它并不适合学习C11最新标准。但对于C语言初学者而言，这款集成开发环境还是非常适合的。幸运的是，2017年3月微软最新推出的Visual Studio Community 2017包含了Clang编译器前端工具，如果我们勾选安装的话即可使用Clang来作为C语言编译器。尽管Visual Studio下的Clang编译器尚处于试验阶段，但大部分功能都可用了。目前笔者测试下来，它对原子操作还没支持好，另外像UTF-8、UTF-16等字符编码问题还与Windows操作系统本身相关，所以要涉及这些问题的话，我们只能使用系统特定的接口去解决或者使用下面提到的MinGW以及Clang官方提供的编译工具链去解决。

所以，如果大家想在Windows操作系统下学习更为完整的C11标准最新特性，那么建议下载MinGW，如果是64位的Windows系统的话则最好下载Mingw-w64。如果还想学习Clang编译器语法扩展的话，也可以再下载单独的Clang编译器。

3.1.1 安装Visual Studio Community 2017

Visual Studio Community最新版本可在微软的Visual Studio官方网站下载：

https://www.visualstudio.com/thank-you-downloading-visual-studio/?sku=Community&rel=15。

当我们下载好Visual Studio Community的安装程序之后，将它打开运行。随后会看到一个选择安装组件的对话框。我们在该对话框的右侧能看到已经勾选上的组件以及一些没有勾选上的组件。这里我们必须勾选上“Clang/C2（实验）”这一项，如图3-1所示。因为不安装Clang，后面就无法用它编译C源代码。

图3-1　Visual Studio Community安装界面

安装完成之后，我们打开Visual Studio Community 2017，首先出现欢迎界面。Visual Studio在首次启动时就会很明显地提示我们注册账号或用账号登录。我们先用Hotmail或MSN账号登录注册，如果不注册仅有30天左右的试用时间，但一旦注册完之后就能永久使用了。我们登录完自己的账号之后就可以开始新建一个C语言的项目工程了。

我们找到菜单栏最左边的“文件”，然后选择“新建”，再点击“项目”，如图3-2所示。

图3-2　欢迎界面中新建项目

随后我们会看到新建项目的对话框。在左侧边栏中找到“Visual C++”，然后选中“Win32”，随后在中间栏选择“Win32 Console Application”，最后，在底下输入此工程创建后存放的目录路径以及工程名，如图3-3所示。

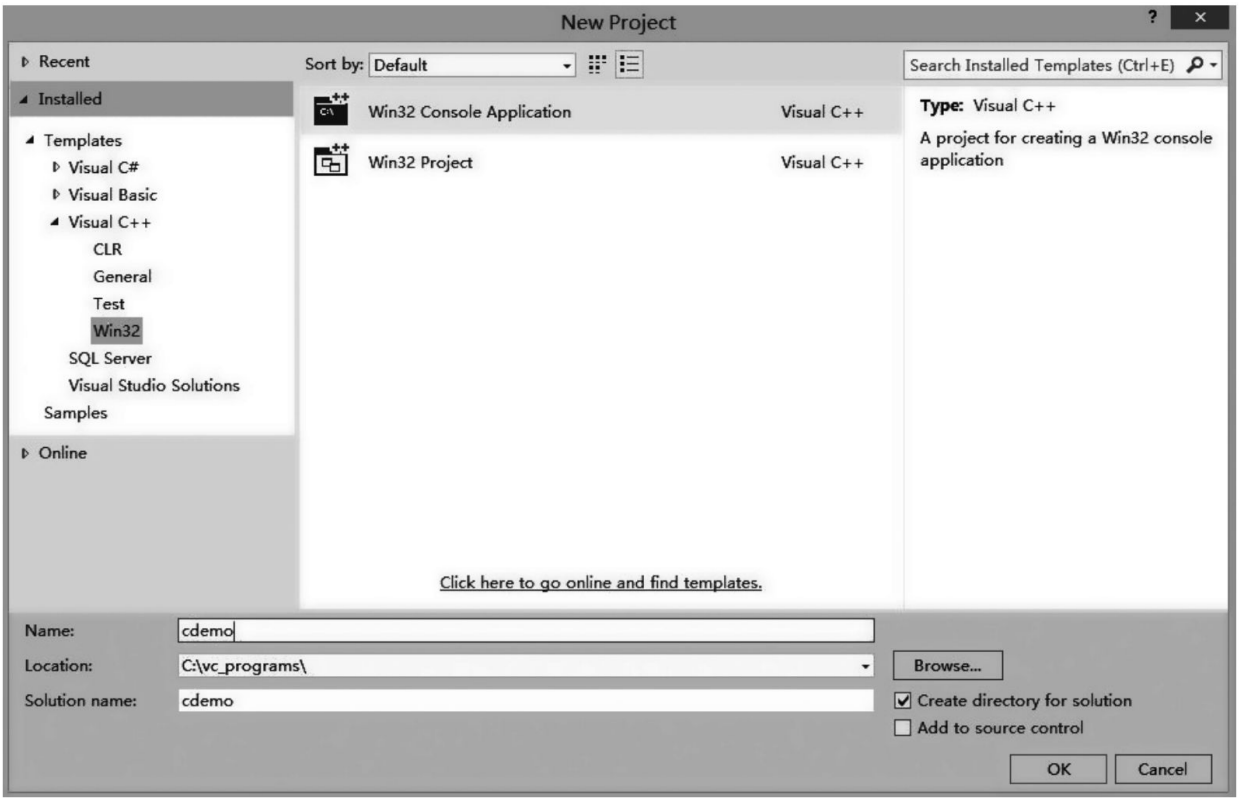


图3-3　Visual Sutdio新建项目

点击“OK”按钮后进入应用设置向导界面，如图3-4所示。

我们看到图3-4这个界面时，先别着急点击“下一步”按钮，应先点击左边边栏中的“应用程序设置”，对此进行初步配置。然后进入图3-5所示的界面。

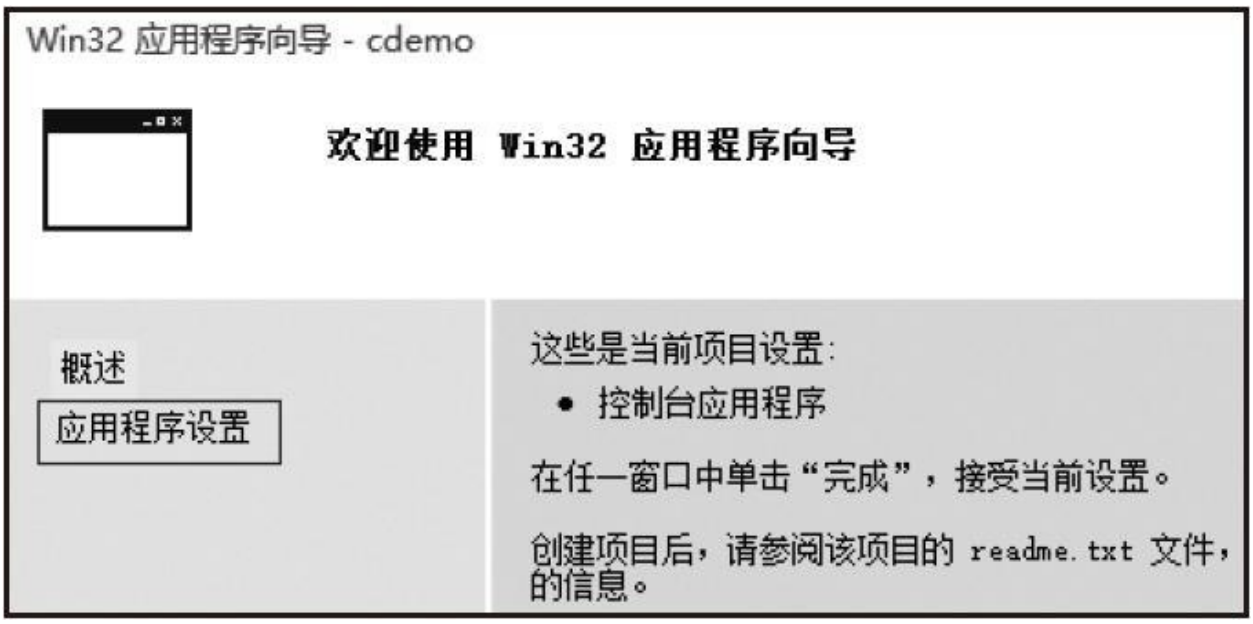


图3-4　Visual Studio应用设置向导

图3-5　Visual Studio项目创建时的应用设置

图3-5所示的界面中，在“附加选项”中，先取消勾选“预编译头”，然后勾选“空项目”。最后，点击“完成”按钮进入到我们所创建的cdemo项目工程

的主界面。此时，整个工程是空的，只有文件夹而没有任何文件，需要手工新建C源文件。用鼠标右键单击“源文件”，选择“添加”，然后再点击“新建项”，如图3-6所示。

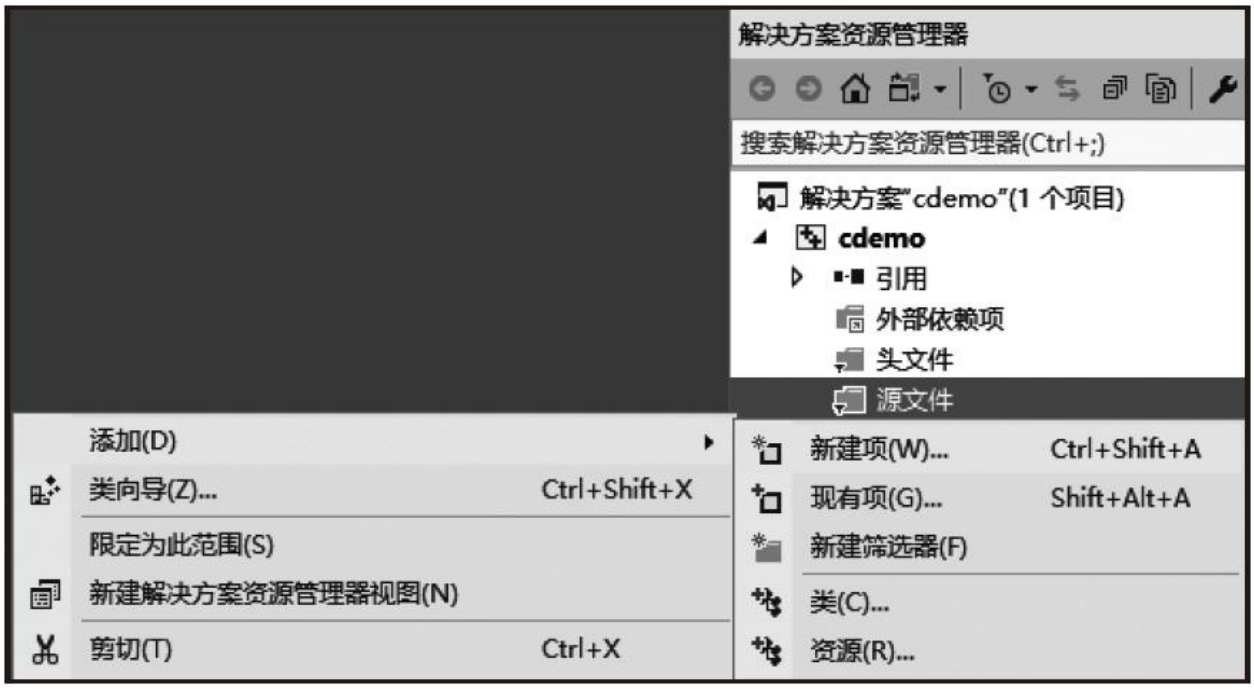


图3-6　Visual Studio添加C源文件

在随后弹出的如图3-7所示的对话框中，选中中间栏中的“C++文件（.cpp）”那一项，然后在底下“名称”一栏输入源文件名。

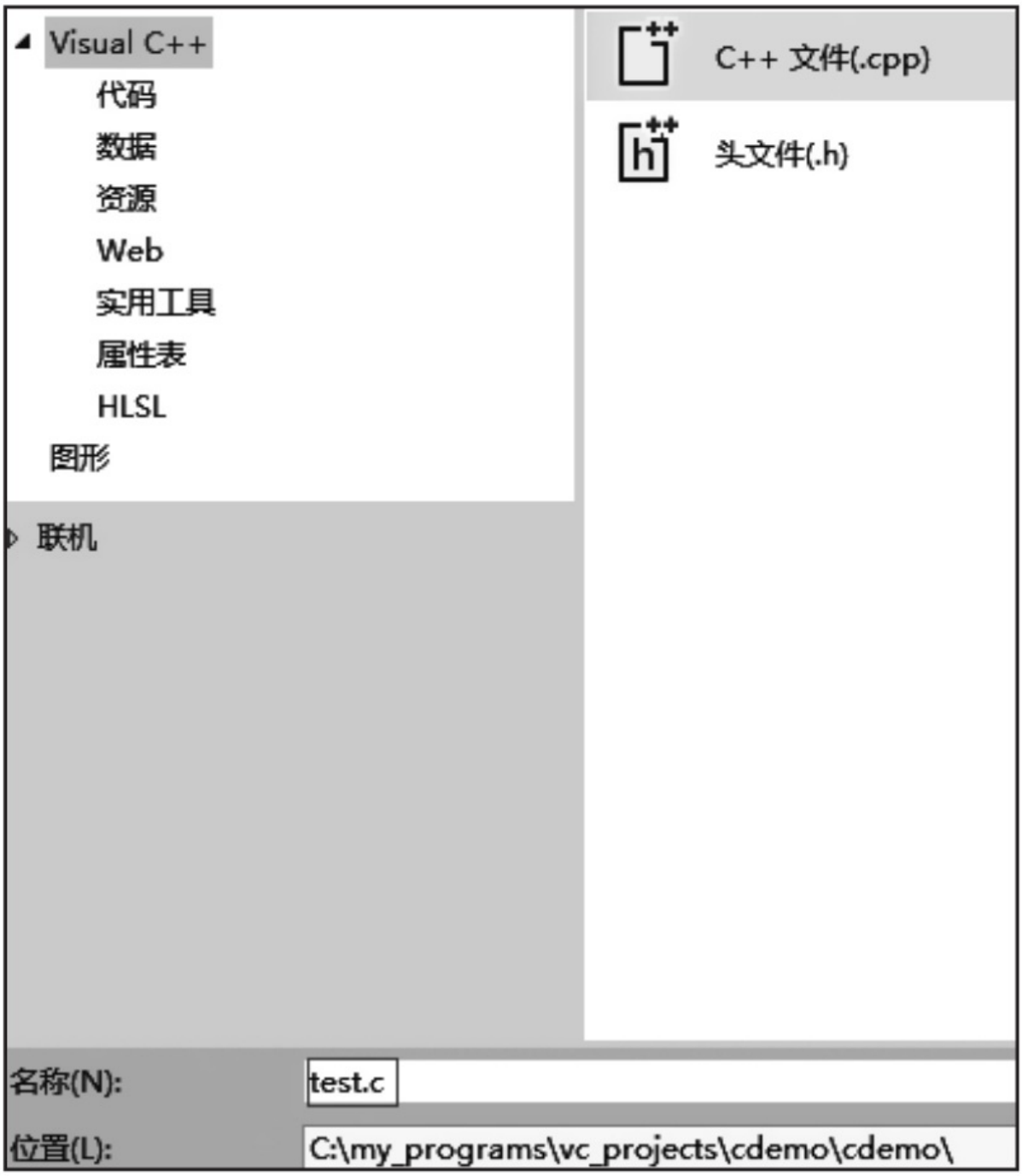


图3-7　Visual Studio命名C源文件名

注意：这里需要注意，默认文件后缀名是.cpp，即C++源文件，因为Visual C++默认采用C++编程语言，因此我们这里要手工填写.c文件后缀

名，使得后续我们用C编译器进行编译构建整个控制台应用。

完成之后，我们点击“添加”按钮，然后再次进入工程主界面，此时即可看到C源文件的编辑界面了。

我们在进入源文件编辑界面后，先对Visual Studio的文本编辑选项做些处理，以便于我们后续可以流畅地编写代码。如图3-8所示，我们在上面的菜单栏找到“工具”，然后选择“选项”。

图3-8　Visual Studio准备设置编辑选项

点击进入后能看到如图3-9所示的对话框。在左边栏找到“文本编辑器”这个选项，然后将它展开，选中“所有语言”，随后我们勾选上“行号”，这样，在编辑每个文本文件时都能看到行号，便于我们查找代码中的语法

错误以及调试代码用。

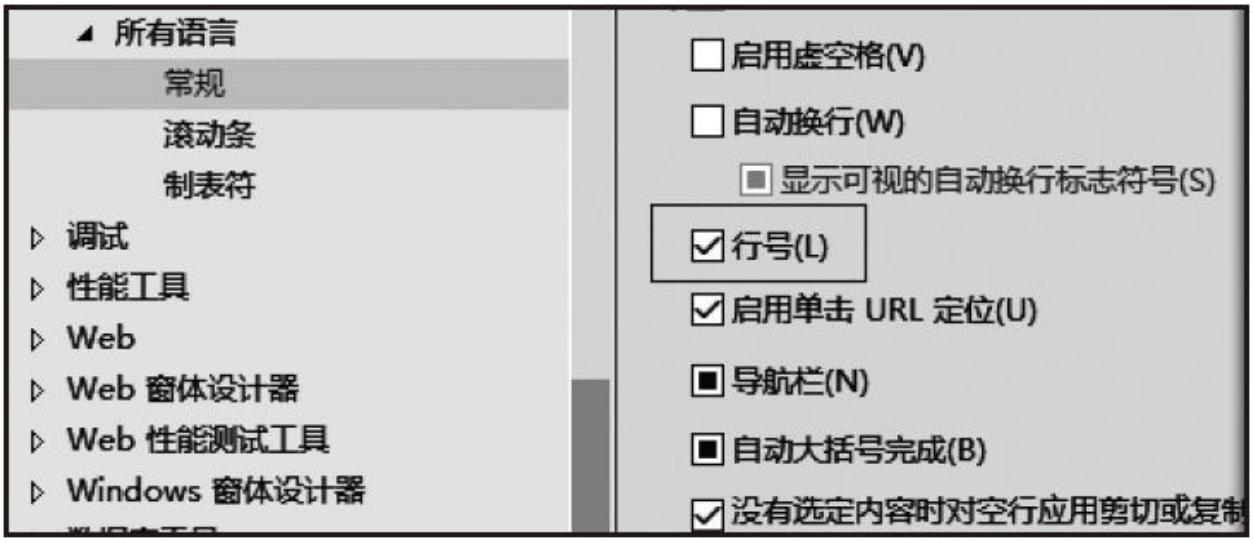


图3-9　设置编辑选项

最后，再选中“制表符”选项，对制表符进行设置，如图3-10所示。习惯上，我们一般将Tab Size设置为4个半角字符，缩进大小也是4个半角字符，然后每个制表符用4个空格代替，这样用其他编辑器浏览Visual Studio编辑过的源文件也不会导致格式错乱。

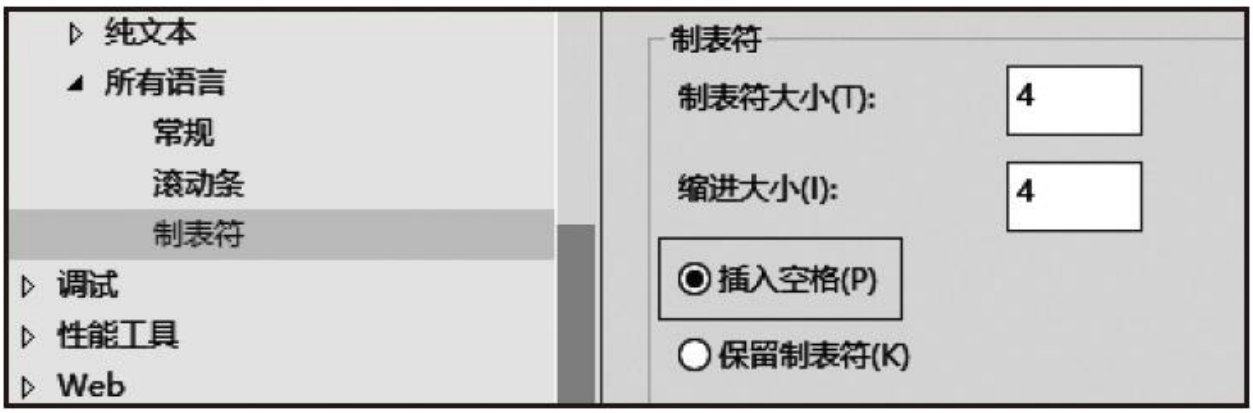


图3-10　Visual Studio设置制表符

接下来我们设置当前的项目工程的属性选项。我们找到菜单栏的“项目”，然后点击“cdemo属性”，如图3-11所示。

图3-11　Visual Studio设置项目属性

在配置界面的常规页面中（见图3-12），先找到左上角的“配置”选项，选择“所有配置”。这样，我们后续做的所有配置都对Debug模式与Release模式同时有效。然后，在右侧找到“平台工具集”，这里需要选择使用“Visual Studio 2017-Clang with Microsoft CodeGen”，这个选项使得我们对当前的项目工程使用Clang编译工具链进行编译构建。

图3-12　Visual Studio对cdemo项目工程的常规设置

随后我们展开C/C++这一项，此时仍然需要先将左上角的“配置”设置为“所有配置”。然后找到“语言”，将“C语言标准”设置为GNU11标准。这样我们就能在Visual Studio Community集成开发环境下编写调试大部分基于GNU11标准的C语言代码了。设置如图3-13所示。

图3-13　Visual Studio设置C语言标准

全都设置完成之后，我们就可以编写第一个C语言程序了。同一般C语

言教程一样，我们这里也通过输出一个“Hello，world！”字样，作为第一个C语言代码的演示程序。我们输入图3-14中所示的代码，然后点击工具栏中的绿色三角箭头（图3-14中用矩形框圈出）即可编译运行了。在程序最后的getchar（）作用在于：弹出的控制台应用不会在程序终止时马上自动关闭，而是等用户输入一个回车时再关闭。

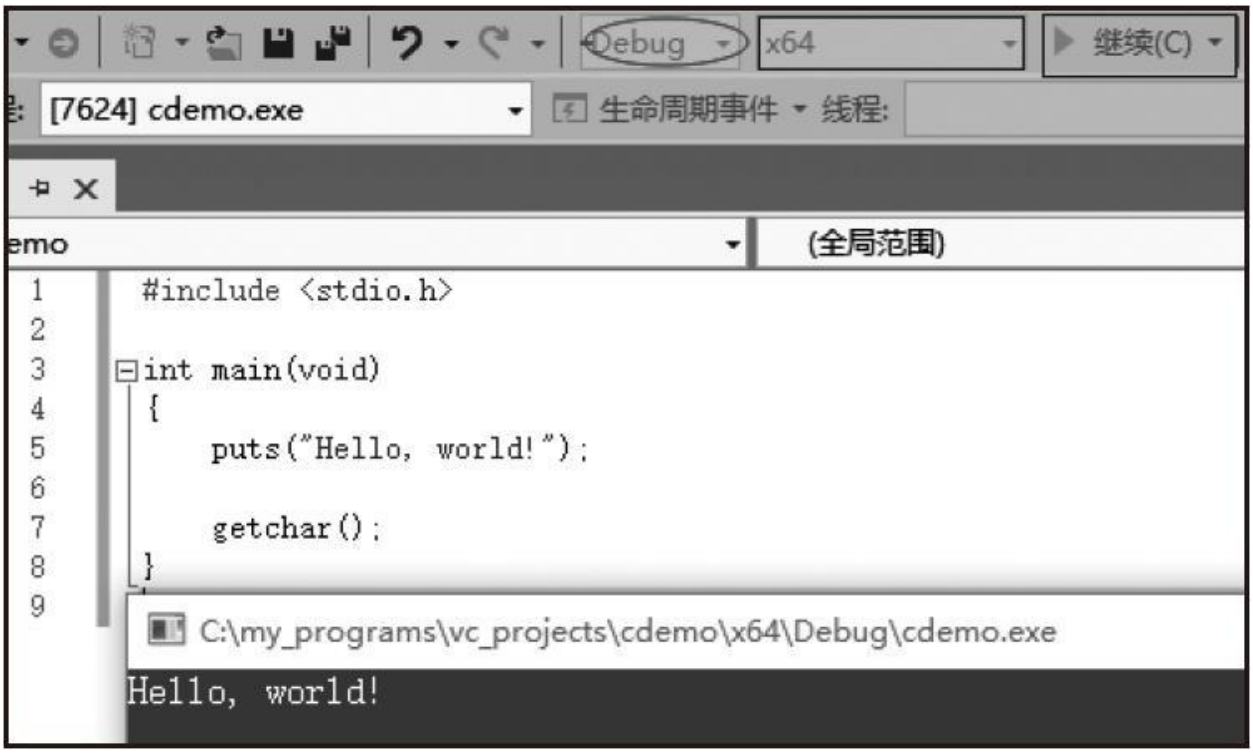


图3-14　在Windows控制台输出字符串

在图3-14所示的界面中，椭圆圈出来的部分用于设置当前程序以调试模式构建还是以发布模式构建。如果以调试模式构建，我们可以利用Visual Studio内建的调试器做断点跟踪，查看局部对象与全局对象状态以及寄存器状态等，便于调试程序。如果以发布模式构建，那么当前程序会被大幅优化，使得程序运行性能大幅提升，但难以调试。图3-14中，中间用

矩形框圈出的部分是设置当前目标程序的执行模式，默认为x86，即32位执行模式。这里我们将它设置成了64位执行模式。

3.1.2 安装MinGW编译器

MinGW编译器是著名开源C语言编译器GCC对Windows操作系统的一个移植版本。通过MinGW，我们就可以在Windows下享用大部分GCC编译器所带来的强大功能了。这对跨平台的C语言开发而言十分有用。下面我们就来介绍如何下载安装MinGW编译器。

首先，我们直接进入这个网址下载安装文件：http://sourceforge.net/projects/mingw/files/latest/download?source=files。这个文件非常小，因为MinGW采用的是在线安装模式，萃取线上各个最新release版本的组件进行下载。

然后，我们双击安装包，初步安装完毕后弹出对话框如图3-15所示。绿色进度条表示已经安装好了。

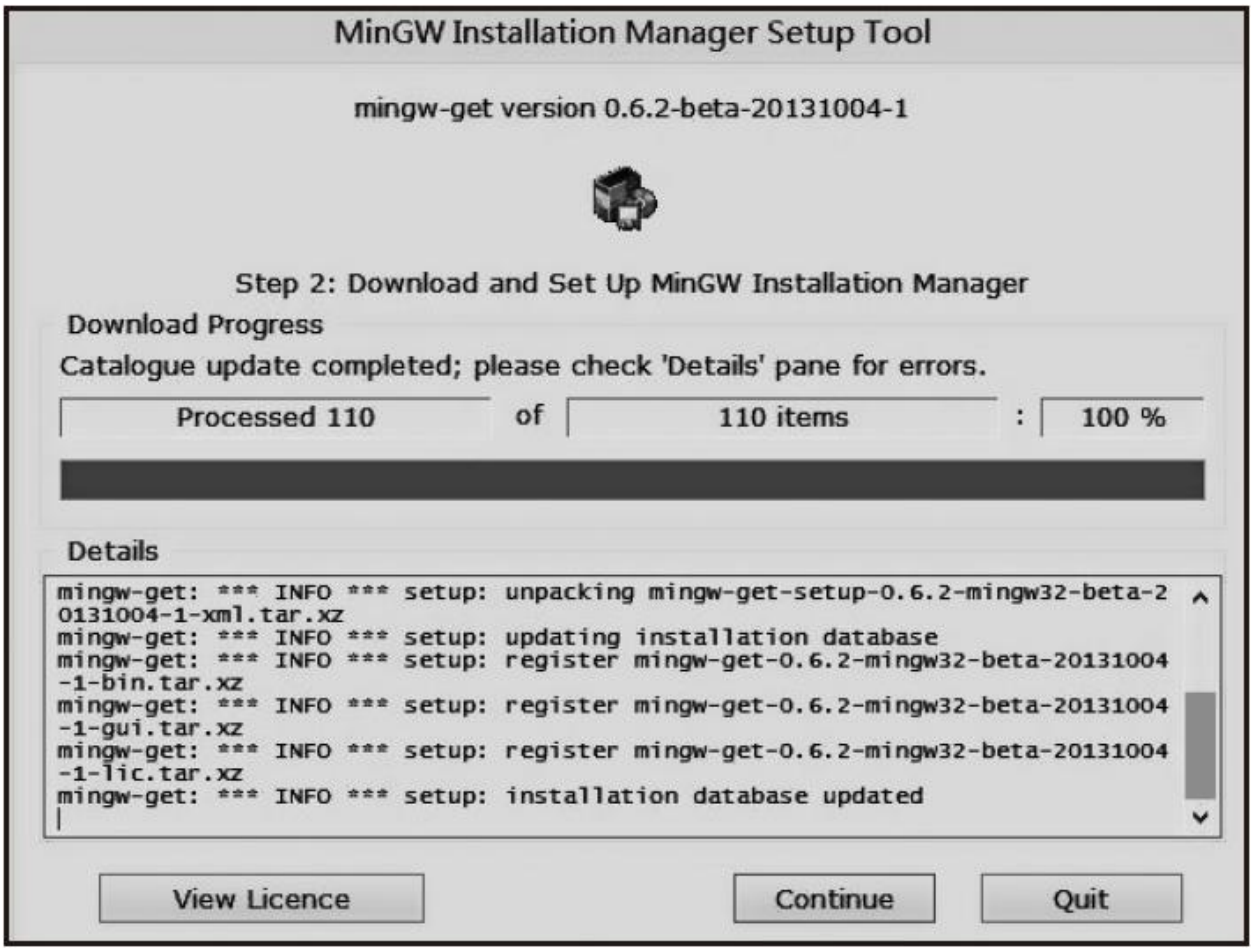


图3-15　MinGW初步安装成功

我们点击“Continue”按钮后，出现选择安装更多组件的对话框。我们在左侧栏点击“Basic”，即采用基本安装。然后，在右侧栏安装上全部列出的组件。要选中某个安装组件，鼠标右键该包名，然后在快捷菜单中选择“Mark for Installation”命令，如图3-16所示。

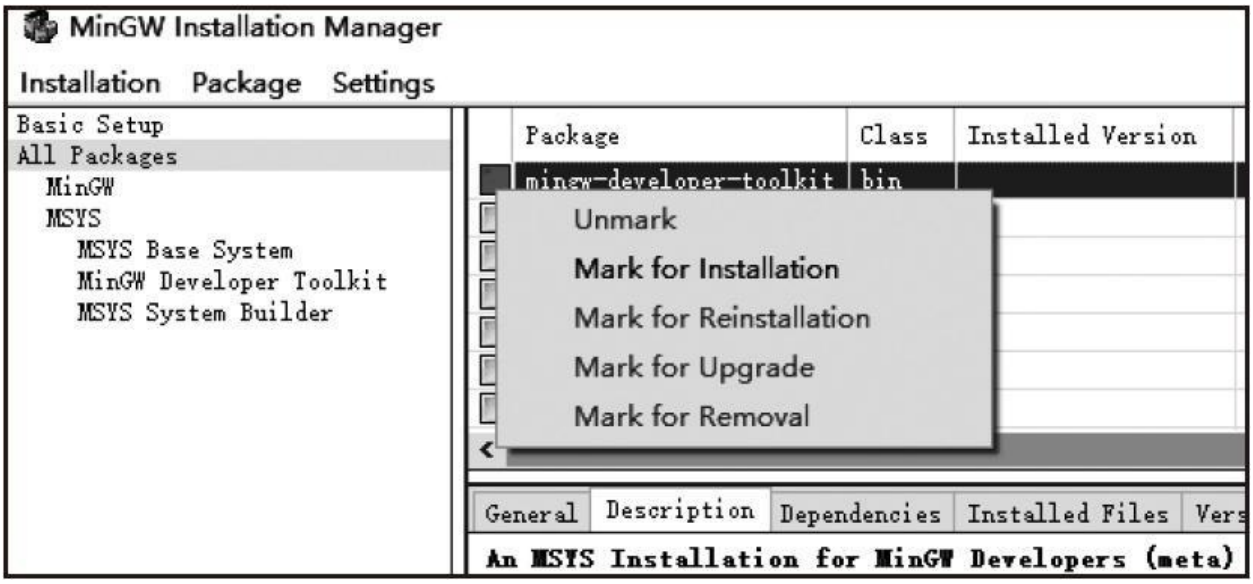


图3-16　MinGW安装，选中安装包

全都选择好之后，我们最后更新刚选好的安装包。我们在菜单栏选中“Installation”，然后点击“Update Catalogue”，如图3-17所示。

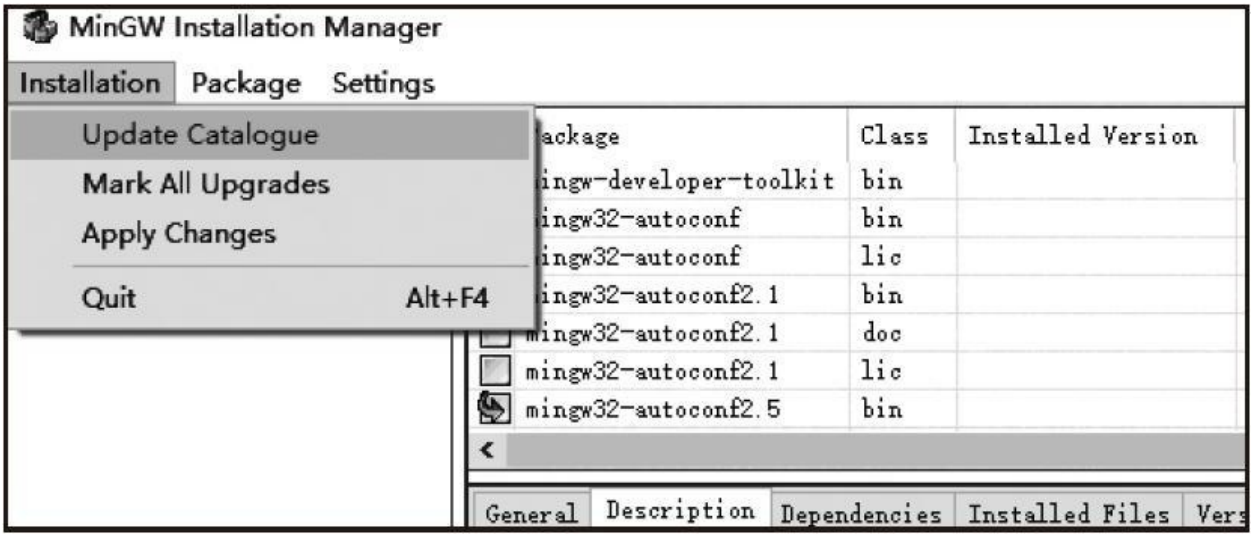


图3-17　MinGW更新安装包

之后会弹出如图3-18所示的界面，点击最左边的“Review Changes”按钮，会弹出如图3-19所示的对话框。

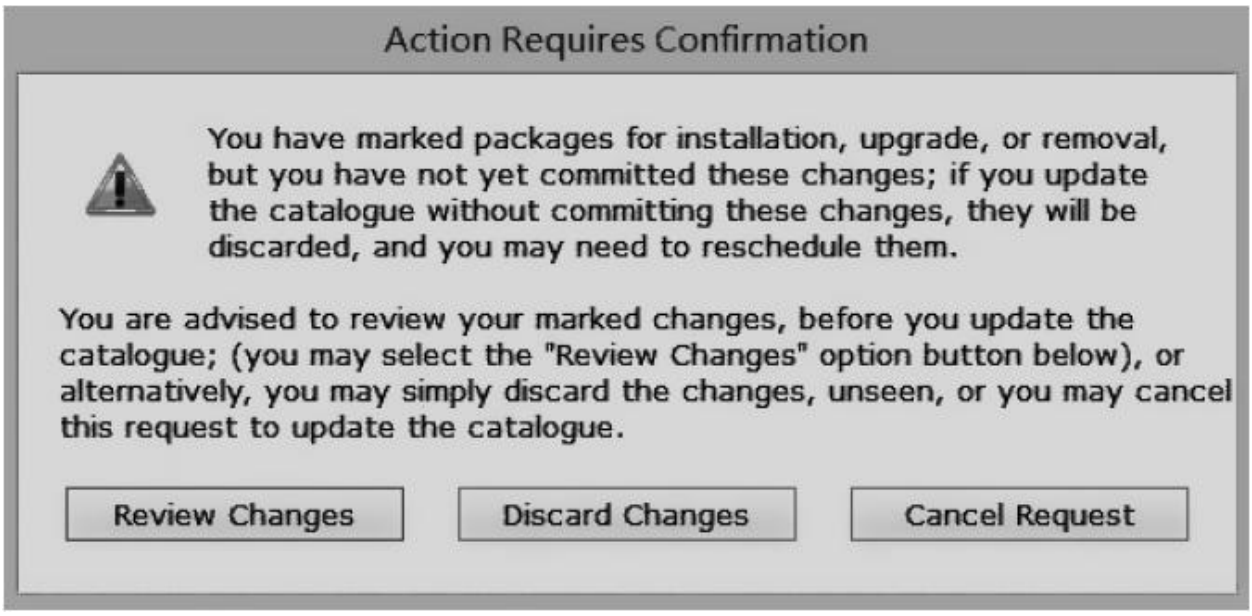


图3-18　MinGW安装要求确认

点击“Apply”按钮之后，就会下载安装设置更新后的安装包。等待全都安装完毕后，点击“Close”按钮，退出整个安装程序。

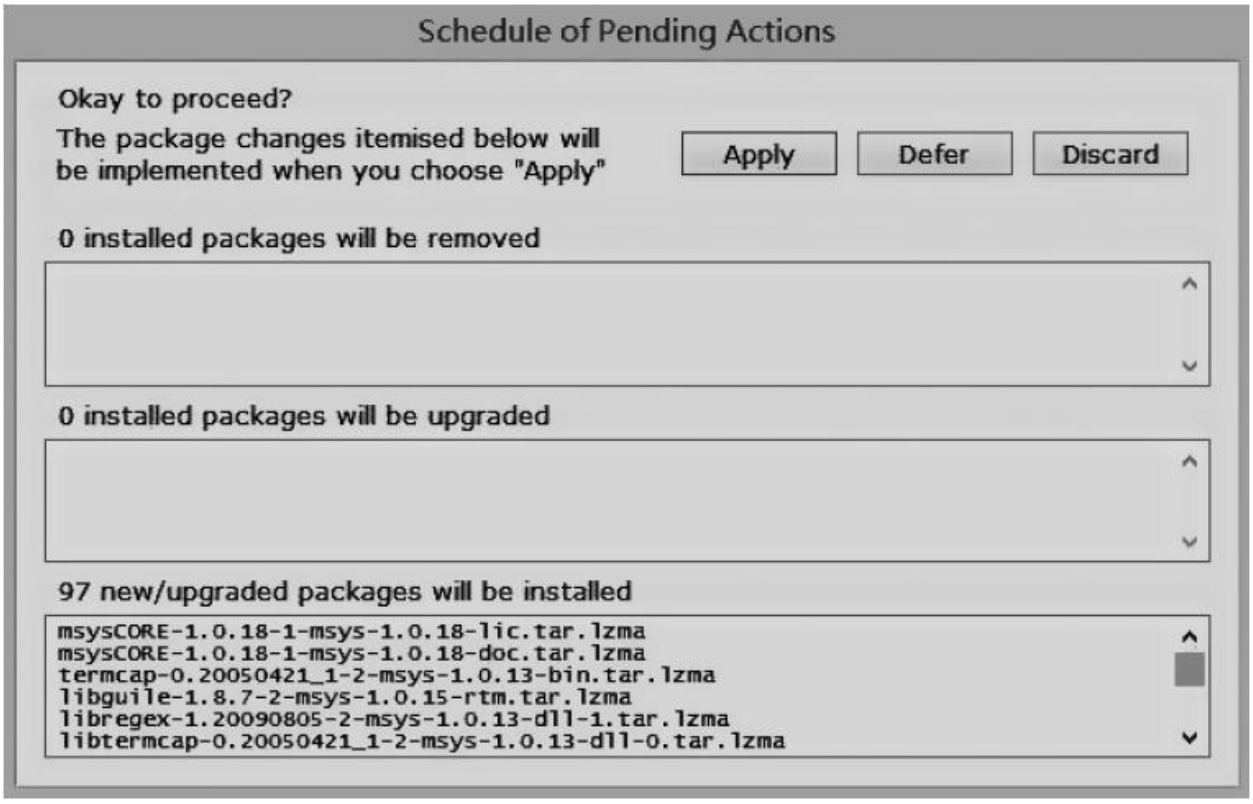


图3-19　MinGW安装更新

安装结束后，不要着急使用，而是先将MinGW的bin文件夹注册到环境变量中。先打开“文件资源管理器”，在左侧栏中找到“此电脑”或“我的电脑”，鼠标右键单击它，选择“属性”，进入后点击左侧的“高级系统设置”，如图3-20所示。

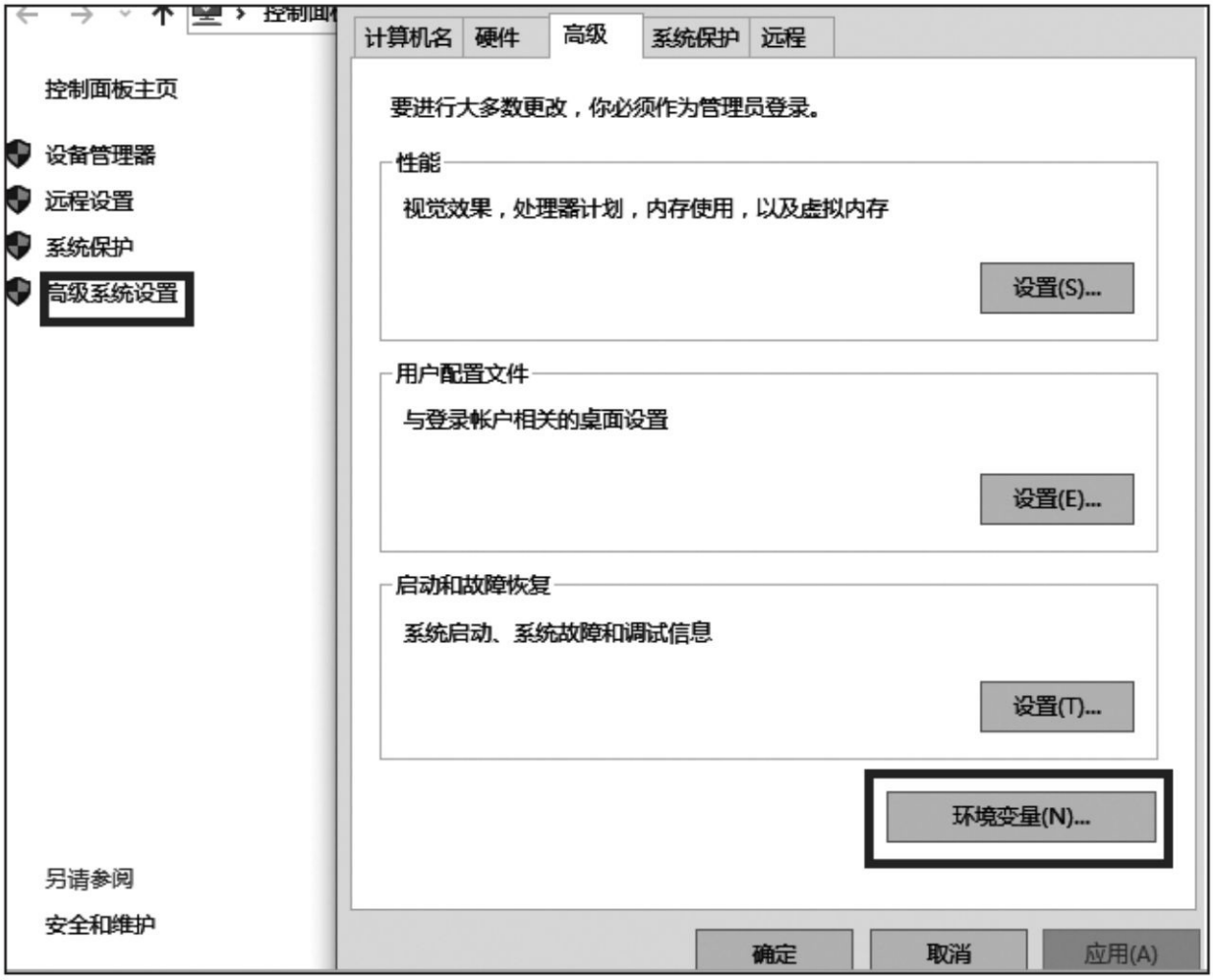


图3-20　进入环境变量的设置界面

进入图3-20的对话框之后，点击“环境变量”按钮，进入到“环境变量”对话框。我们在“系统变量”区域选中“Path”变量，然后点击“编辑”按钮，弹出“编辑系统变量”对话框。在“变量值”中往后添加刚才安装后的MinGW中的bin文件夹所在目录。在环境变量中的每个值之间用半角分号“；”进行分隔，如图3-21所示。

图3-21　进入环境变量设置Path

完成之后，我们就可以打开控制台程序（方法是右键桌面上左下角“开

始”按钮，然后选择命令提示符），然后进入要编译的C源文件所在的目录。然后用gcc命令对指定C源文件进行编译构建，如图3-22所示。

这里，我们借用之前在Visual Studio Community下编辑好的源文件test.c。我们先用cd命令定位到test.c所在的目录。然后用gcc--version命令查看当前GCC编译器的版本。最后，用gcc-std=gnu11 test.c进行编译，最终在当前目录生成a.exe可执行文件。我们直接键入a，回车，即可看到程序输出结果。

要注意的是，MinGW是32位的C语言编译器，所以它构建出来的程序也是32位的。如果各位用的Windows操作系统是64位的，那么可以使用Mingw-w64编译器。下载地址如下：https://sourceforge.net/projects/mingw-w64/files/latest/download？source=files。

Mingw-w64的安装、设置过程与32位的MinGW类似，这里不再赘述。

3.1.3 安装LLVM Clang编译器

LLVM（Low Level Virtual Machine）起源于一个大学项目，它是一个编译器基础架构项目，用于设计一组具有良好定义的、可重用的库。LLVM起先用于替代GCC（这里的GCC是指GNU Compiler Collection）栈中的代码生成器，然后对GCC中已有的许多编译器进行修改以适配LLVM。后来LLVM发起了开发一个全新的适用于不少编程语言的编译器

前端，称为Clang。Clang主要支持C、C++、Objective-C等编程语言，并且主要由Apple公司大力支持和维护。LLVM与Clang都基于BSD许可证，比GPL更宽松。正因如此，现在许多硬件商都逐渐开始投入对LLVM的支持，像Khronos开放标准组织也基于LLVM IR（Intermediate Representation）开发出了自己的一套SPIR-V。Clang编译器在语法上力争支持各大主流编译器的语法扩展，包括GCC和MSVC，所以微软也已经把Clang纳入Visual Studio集成开发环境的工具集中。

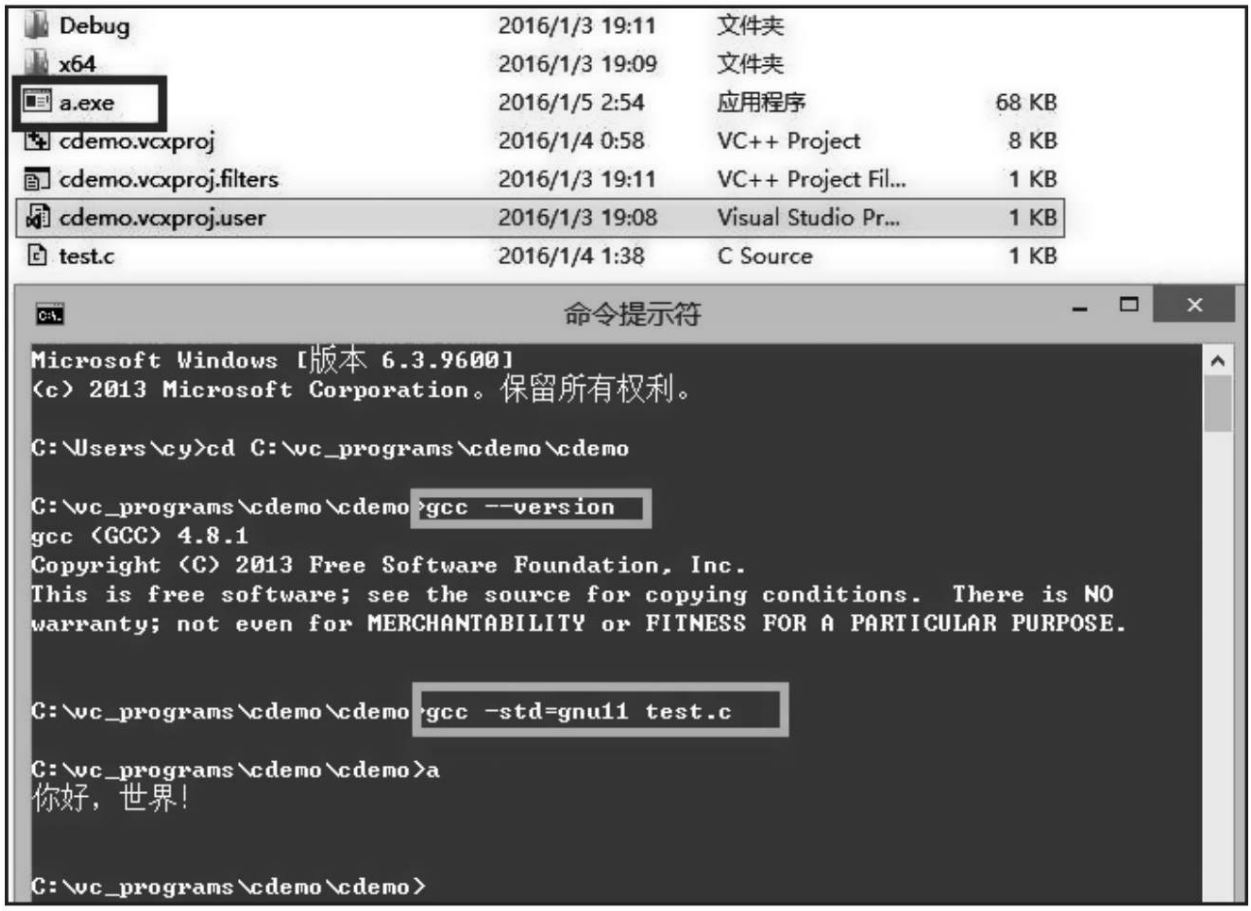


图3-22　用GCC构建C程序

我们首先在LLVM Clang官网下载最新稳定发布版本的Clang安装包：http://llvm.org/releases/download.html。然后，要注意的是选择32位版本，如图3-23所示。

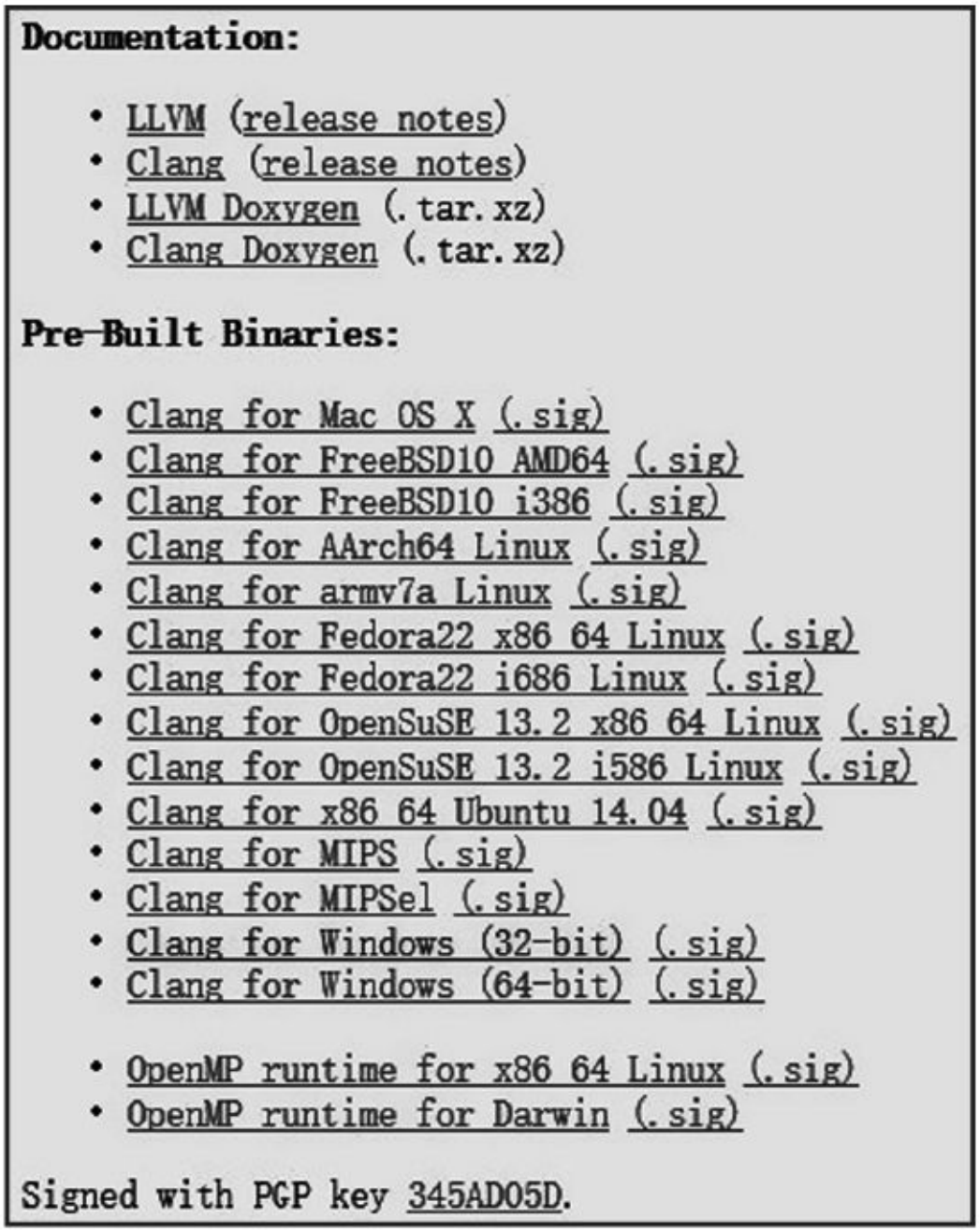


图3-23　下载Clang for Windows（32-bit）

由于Clang主要是一个编译器前端，因此它需要依赖其他编译器的连接器以及某些运行时库。所以，我们光安装Clang是无法直接成功构建应用程序的，因而我们要使用Clang的话，必须在此之前先把MinGW安装好。

MinGW是32位的，因此为了二进制兼容，我们所选取的Clang也必须是32位的。当然，如果之前安装的是64位的MingW-W64，那么这里需要下载安装64位的Clang。

安装Clang的过程非常简单，可根据安装向导简单地做些选择即可完成安装。安装完成后，可以去“系统”里的环境变量中看，把LLVM目录下的bin文件夹的路径添加到Path环境变量中，如图3-24所示。然后就可以再次使用命令行工具直接编译运行程序了。

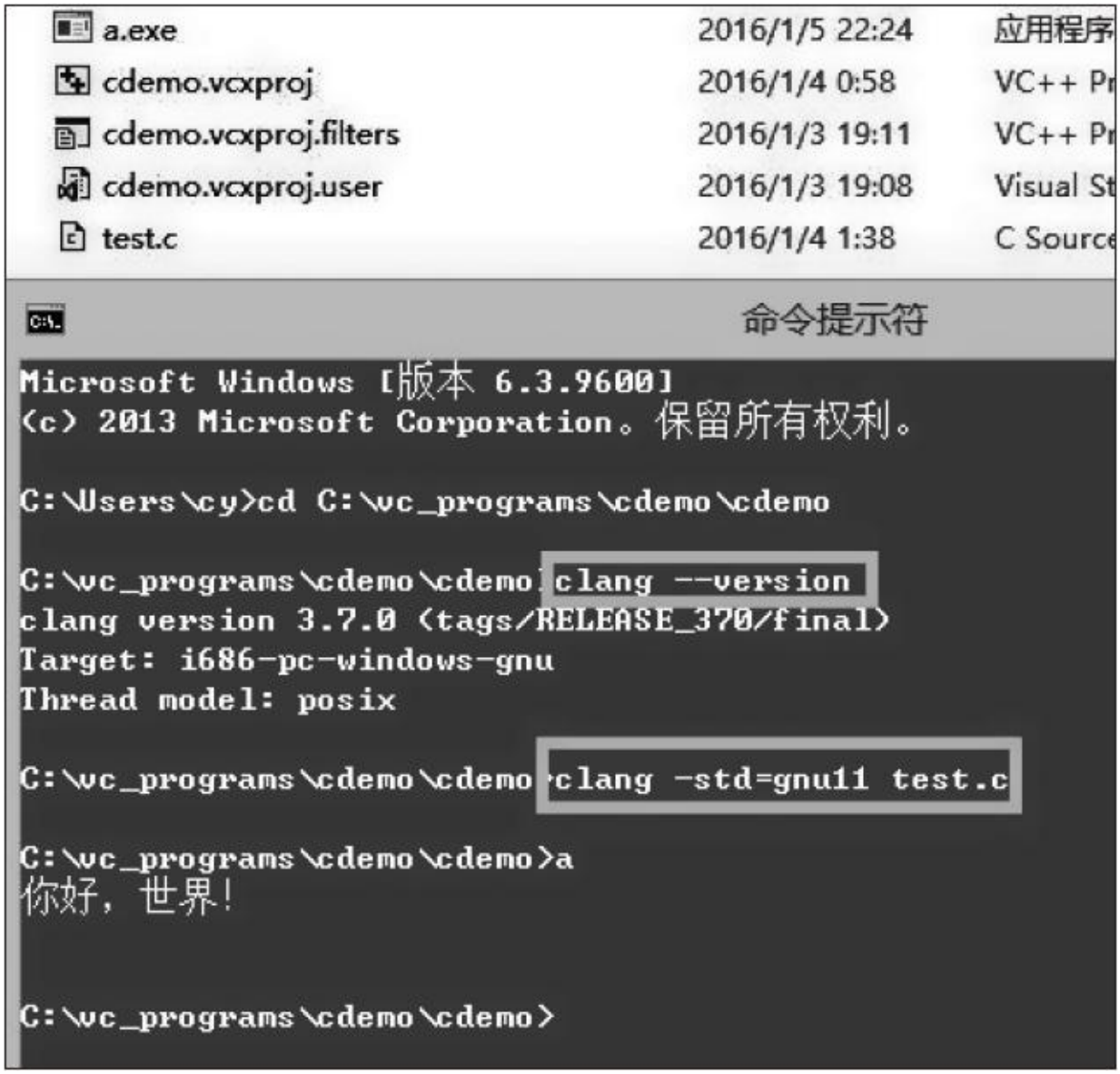


图3-24　用Clang编译器构建应用程序

3.2 macOS系统下搭建C语言编程环境

macOS系统也不默认自带C语言编译器。然而，用户可以自己去Mac App Store免费下载macOS下的强大开发工具——Xcode：https://itunes.apple.com/cn/app/xcode/id497799835？mt=12。该集成开发工具采用Apple定制版本的Clang编译器，称为Apple LLVM编译器。它自带C、C++、Objective-C以及Apple自己新推出的Swift编程语言编译器，还有一系列功能强大的代码静态分析以及性能剖析工具。

下载完Xcode之后，把它打开。如果是第一次启动，Xcode会自动更新一些资源，完了之后弹出主界面，如图3-25所示。

我们选择第二个选项，点击它即可创建应用程序工程。第一个选项仅用于操练把玩Swift编程语言，而第二个选项用于创建真正的应用或库。当然，有些应用可直接提交到App Store审核，有些则不行。

点击“Create a new Xcode project”之后，出现图3-26所示的对话框。在图3-26中，我们看到在上面一栏中所选的项目工程为macOS的应用。然后在下边，我们选择“Command Line Tool”，即命令行工具。最左边的Cocoa Application用于创建macOS系统上基于GUI以及沙盒机制的应用，它可以上传到Mac App Store。中间的“Game”专门用于游戏应用，也可上传到Mac App Store。而最右边的“Command Line Tool”构建出来的应用则无法上传到Mac App Store，但是它能访问macOS的整个文件系统，并且没有采用沙盒

机制。另外，开发者用Command Line Tool开发出来的应用也可以直接放到网上供其他人下载使用。

图3-25　Xcode欢迎界面

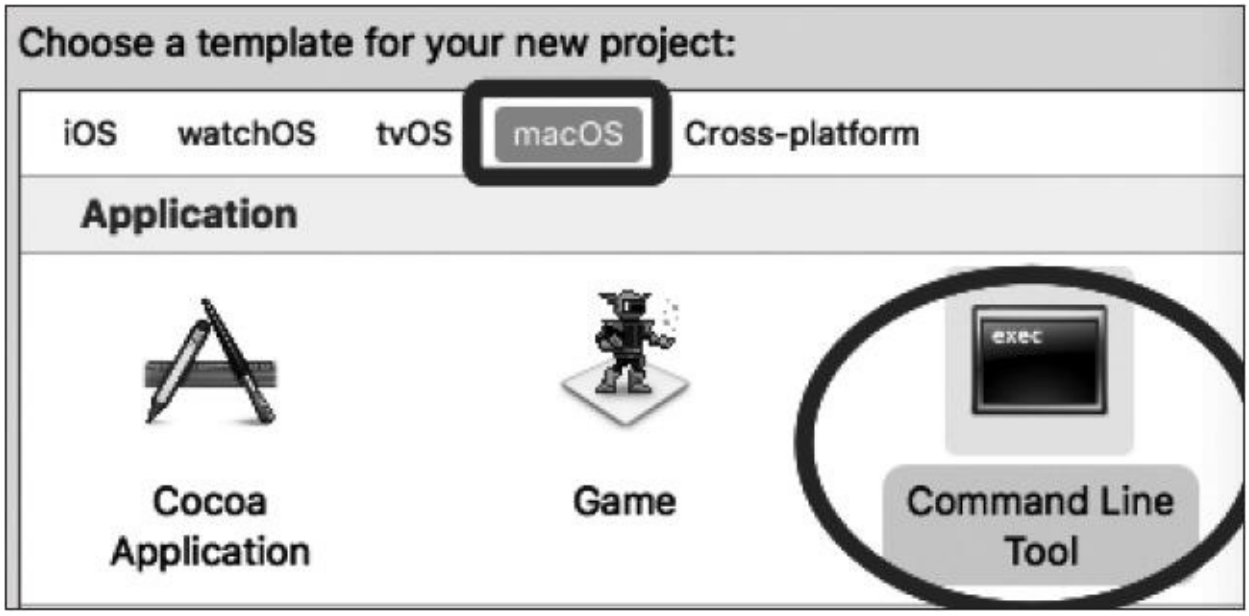


图3-26　选择MacOS命令行工具应用项目

我们点击“Next”按钮之后出现如图3-27所示的对话框。在第1行用英文输入自己的产品名称，这个后面将用于自动生成的工程名称。然后第2行填写组织名。第3行填写组织标识，格式为com.<公司名>.<产品名>。当然，第2、第3行对于我们的demo而言可以随意填写。第5行我们要选择C，表示使用C语言。

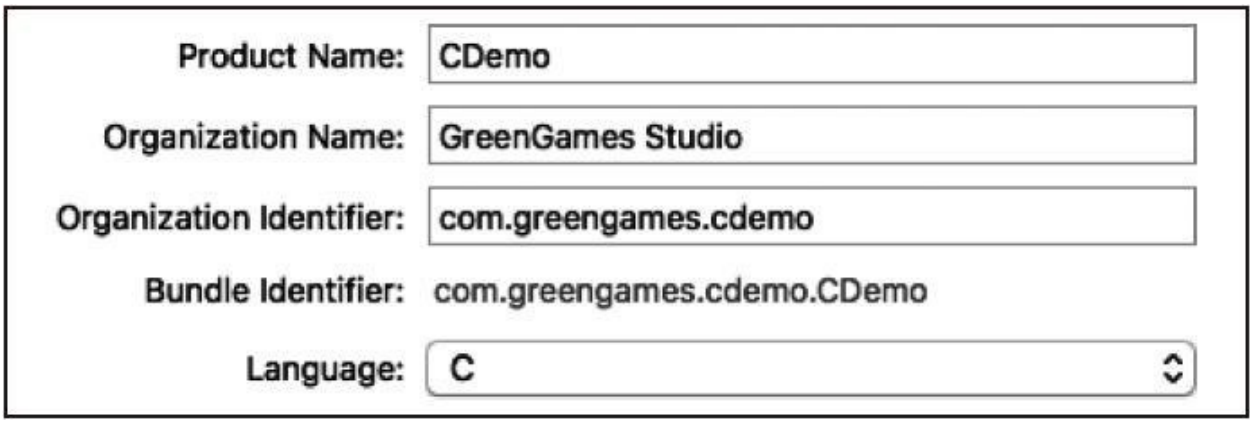


图3-27　输入macOS命令行应用的属性

点击“Next”按钮可看到图3-28所示的目录选择对话框。

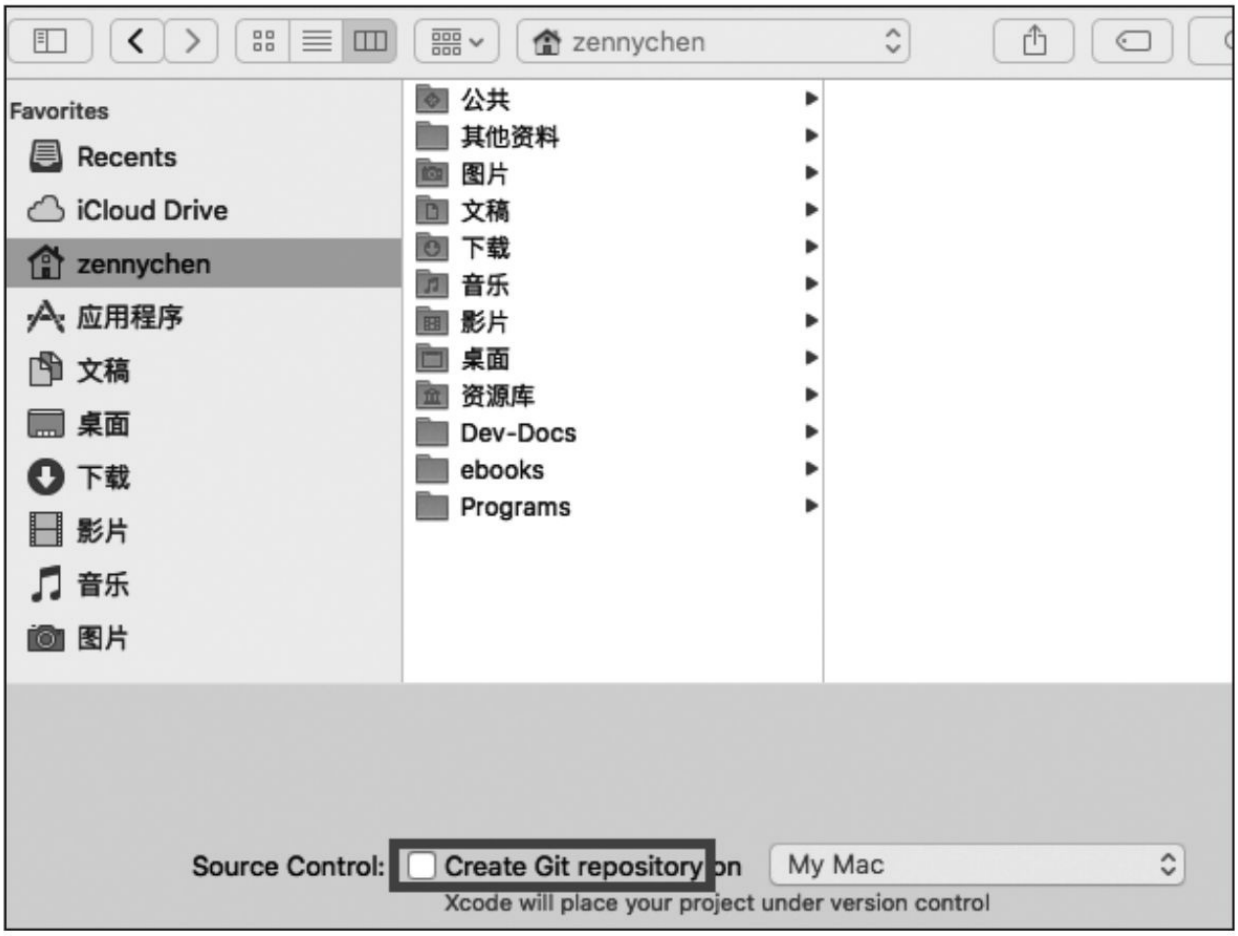


图3-28　macOS命令行应用生成目录选择对话框

这里选择将新创建的项目工程放到哪个目录下。另外，这里要注意的是，我们不要勾选“Create Git repository”这一选项。因为它会在工程本地做git版本管理，对于我们一般应用而言没有任何必要，而且这会随着工程构建的次数增多而增大，很占磁盘空间。而且如果要将本地工程拷贝到其他环境，也会带来许多不便。我们最后点击“Create”按钮之后，工程就会被创建好。

工程被创建完之后，Xcode默认会打开，包括会自动创建一个main.c的C语言源文件。此时，我们不用着急编辑、运行，可以先设置一下编译选项。

我们首先点击蓝色的“CDemo”项目工程图标，然后点击中间一栏“TARGETS”下的“CDemo”控制台图标，最后在右边栏的最上方选中“Build Settings”，然后在下面选中“All”和“Combined”。随后，我们找到“Apple LLVM x.x-Language”这一栏，将“C Language Dialect”选为gnu11，这个选项将贯穿本书内容。到此，我们的C语言编译选项就设定好了，如图3-29所示。

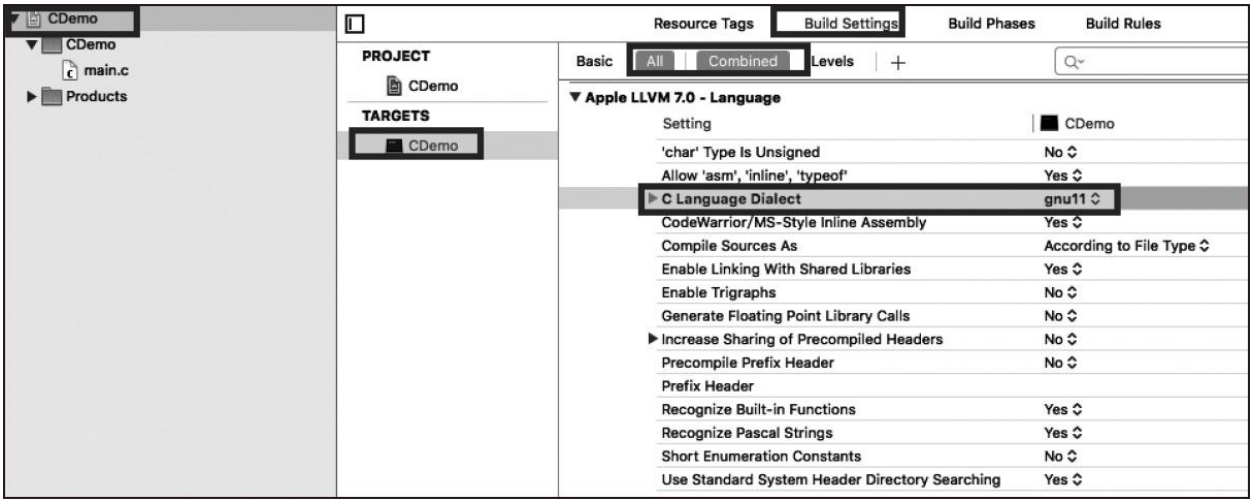


图3-29　macOS项目设置工程配置选项

如果我们想对最终生成的代码再做一些优化，可以设置图3-30中的一些选项。

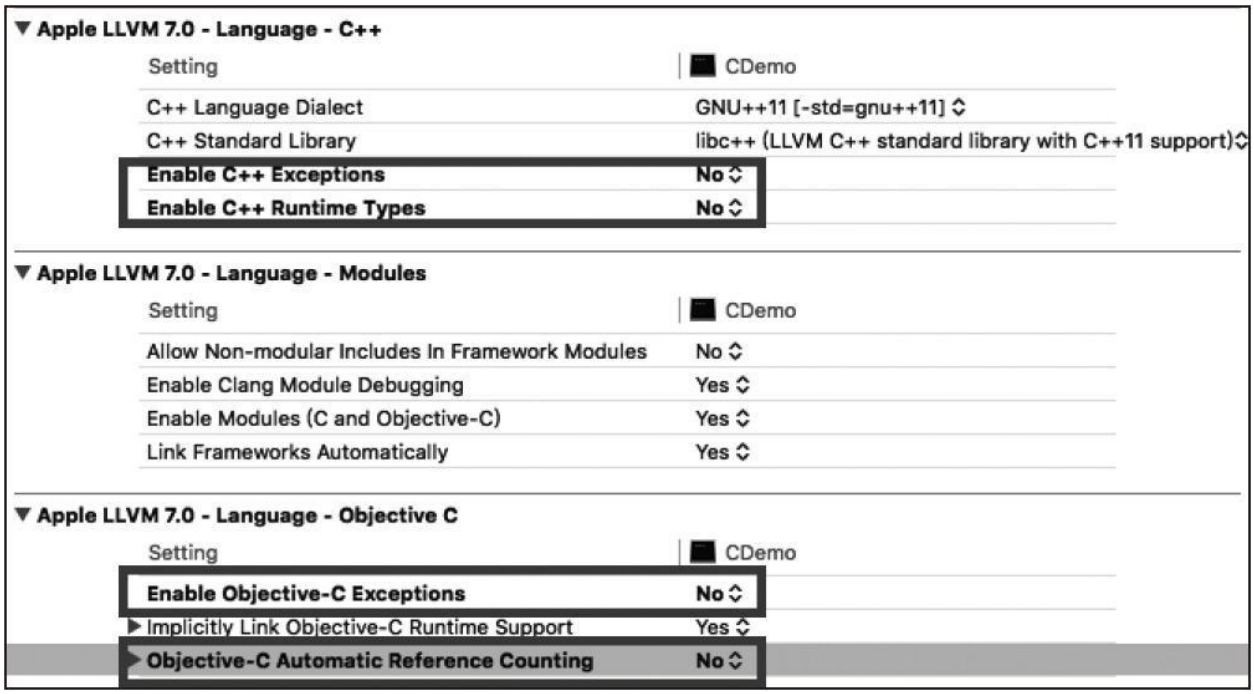


图3-30　macOS项目设置其他编译选项

我们将C++的异常以及运行时类型（RTTI）全都关闭，另外也将Objective-C的异常关闭。这样，最终的应用程序中将不会包含异常栈，同时，编译器后端优化也能更省力不少。大家可以观察到，将这几个选项关闭后，最终生成的可执行文件会比开启时要小一些。

最后，我们可以设置一下Xcode自身的偏好设置，将行号显示出来，如图3-31所示。

图3-31　打开Xcode偏好设置

我们在菜单栏上，选择“Xcode”，然后点击“Pre-ferences...”，弹出图3-32所示的对话框。我们把“Line numbers”勾选上即可在文本编辑框中看到行号。另外，Xcode默认字符编码已经是UTF-8了，因此不需要我们做额外的设置。

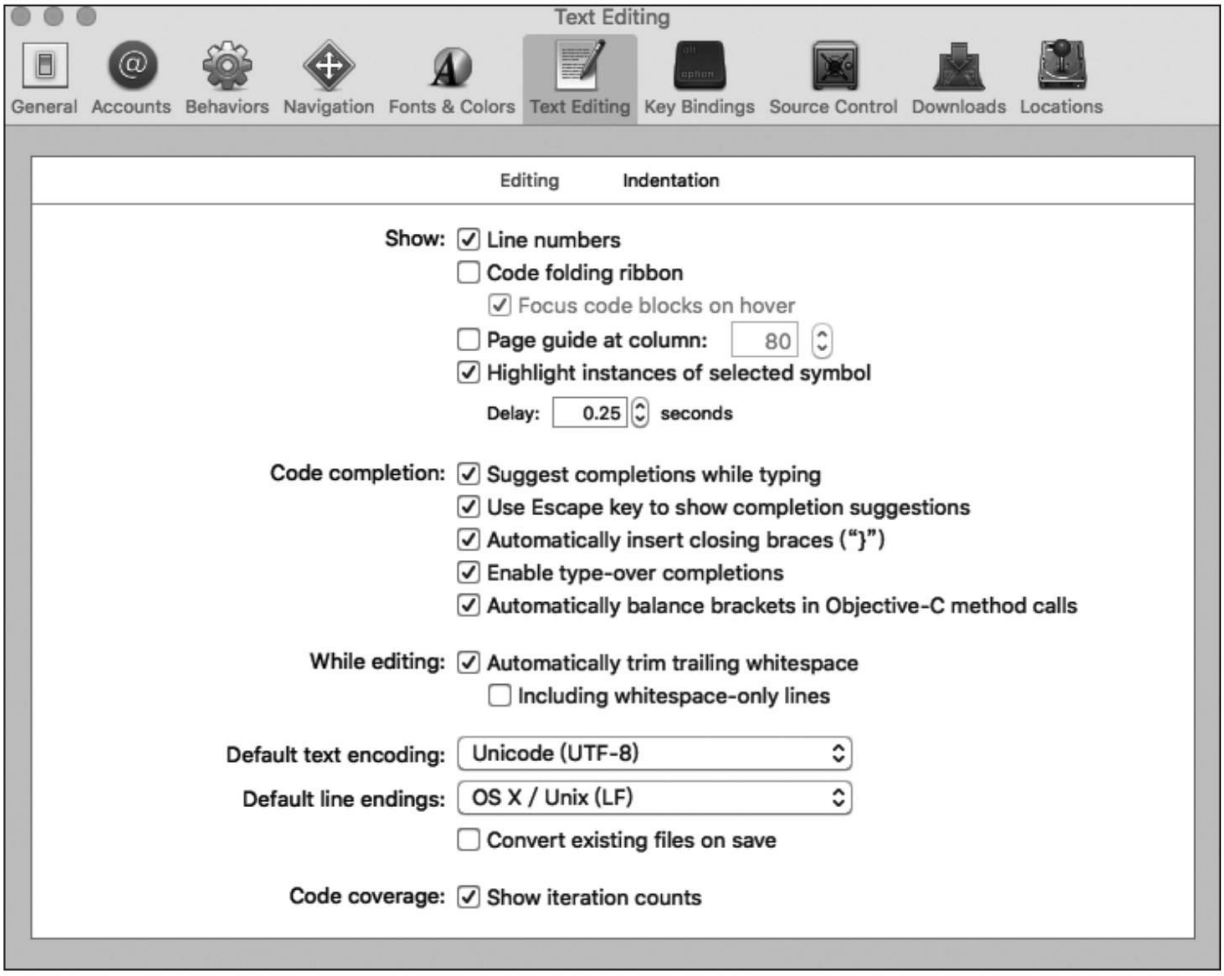


图3-32　Xcode设置文本编辑属性

由于Xcode默认字体可能会显得比较小，因此如果想设置字体以及背景颜色的话可以选择“Fonts&Colors”选项。

在进入到此对话框后，我们点击左侧栏下边的“+”号，添加一个新的字体，并且选择“Duplicate‘Default’”，如图3-33所示。这使得我们所新增的字体以默认字体和颜色作为基准，然后对它做大小修改。

图3-33　Xcode字体设置，添加新字体

如图3-34所示，我们这里新增了一个叫“Defualt_Big”的字体，然后在中间这栏，我们先选中“Plain Text”，然后将滚动条滚动到最下方，按住Shift键再选中最后一条“Other Preprocessor Macros”，这样可以将所有种类的文字格式全都选中，随后我们点击“T”字样的按钮来调整这些文字格式的字体大小。这里，原先的字体大小为“Menlo Regular-11.0”，设置之后这里变为“Menlo Regular-14.0”。

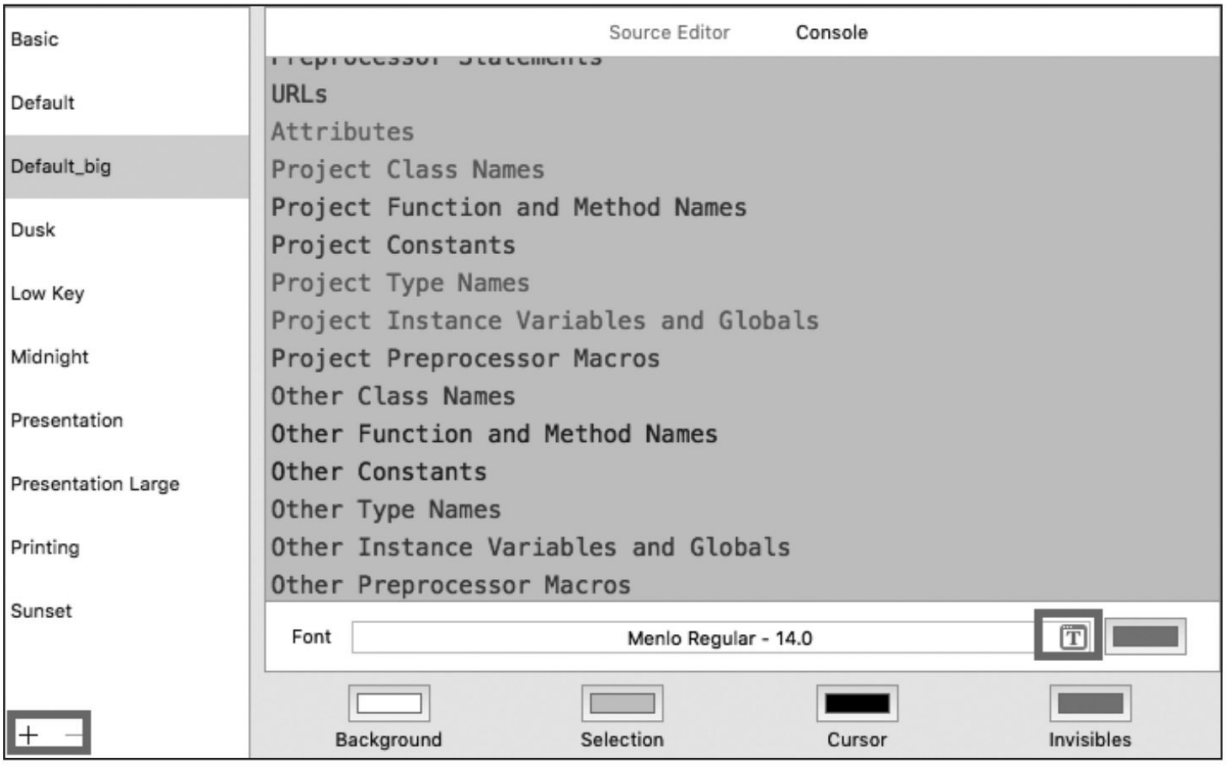


图3-34　Xcode设置新字体格式与大小

现在，我们就可以直接运行Xcode自动帮我们生成好的main.c中的C源代码了。我们直接点击右上角的三角箭头按钮即可编译并运行这段代码，如图3-35所示。

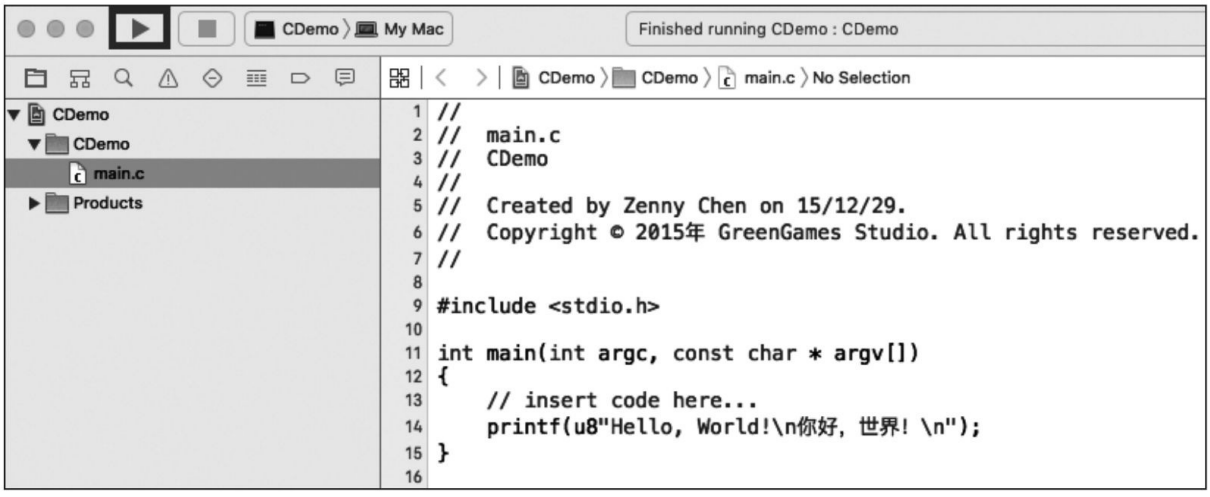


图3-35　编译运行macOS控制台应用

我们在下面的调试控制台中能看到图3-35这两行文字。其中，最后一句是应用退出后系统自动打印的。我们可以看到，macOS下能非常轻松地直接输出中文，而不需要各种复杂的编码转换。

图3-36　macOS控制台程序运行结果

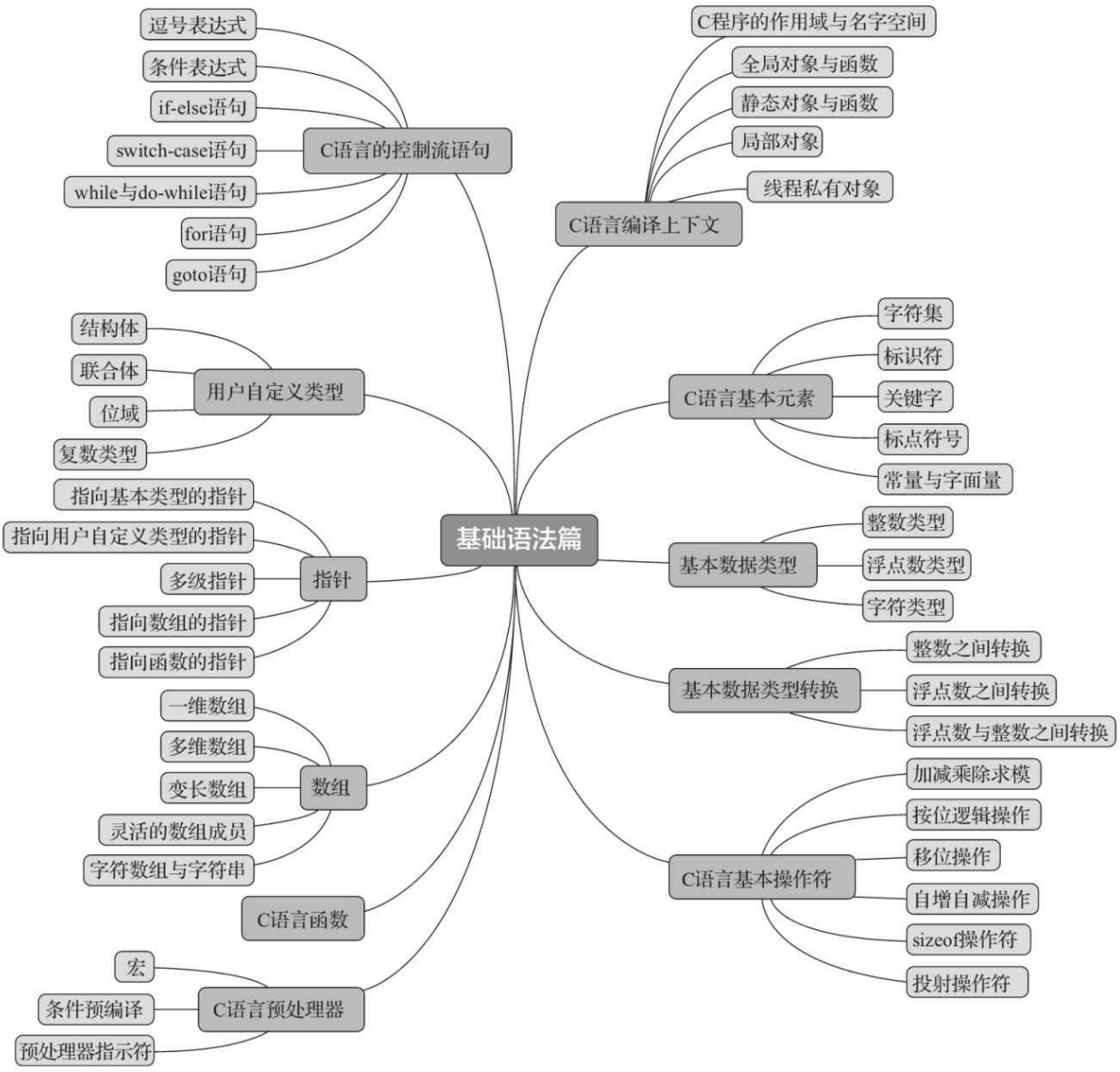
3.3 本章小结

本章主要讲述了Windows操作系统下如何使用Visual Studio Community、MinGW和LLVM Clang进行C语言程序开发，同时也讲解了如何在macOS下使用Xcode做C语言程序开发。因为Windows操作系统与macOS系统用得比较广泛，而且它们都主要基于GUI的集成开发环境进行编程，所以我们做重点讲解。而在各个版本的Linux下基本都默认安装了GCC编译器，各位可以直接在Linux系统下的命令行终端使用gcc命令对C语言源文件做编译构建。而当前FreeBSD最新发布版本默认使用了LLVM Clang编译器，各位也可以直接在命令行终端使用clang命令对C语言源文件做编译构建。

另外，Linux、FreeBSD系统下，笔者推荐使用的集成开发环境是Eclipse。它拥有比较基本的代码智能感知，设置断点进行调试的功能，而且它也是一款开源免费的软件。当然，要启动Eclipse必须先下载JRE（Java Runtime Environment），这个可以从Oracle官网下载。

截至本章，第一部分的讲解结束，各位读者应该对C语言的由来、用途以及各种准备工作都了解得差不多了吧？下面我们将进入第二部分，正式开启C语言魔法的大门！

第二篇　基础语法篇


基础语法篇
C语言的控制流语句
逗号表达式
条件表达式
if-else语句
switch-case语句
while与do-while语句
for语句
goto语句
C语言编译上下文
C程序的作用域与名字空间
全局对象与函数
静态对象与函数
局部对象
线程私有对象
用户自定义类型
结构体
联合体
位域
复数类型
指针
指向基本类型的指针
指向用户自定义类型的指针
多级指针
指向数组的指针
指向函数的指针
数组
一维数组
多维数组
变长数组
灵活的数组成员
字符数组与字符串
C语言函数
C语言预处理器
宏
条件预编译
预处理器指示符
C语言基本元素
字符集
标识符
关键字
标点符号
常量与字面量
基本数据类型
整数类型
浮点数类型
字符类型
基本数据类型转换
整数之间转换
浮点数之间转换
浮点数与整数之间转换
C语言基本操作符
加减乘除求模
按位逻辑操作
移位操作
自增自减操作
sizeof操作符
投射操作符

第4章　C语言中的基本元素

本章将正式进入C语言编程话题。我们在第1章已经大致介绍了C语言的编译、连接和加载运行流程，参见图1-2。我们首先介绍C语言单个源文件的基本构成以及基本元素。

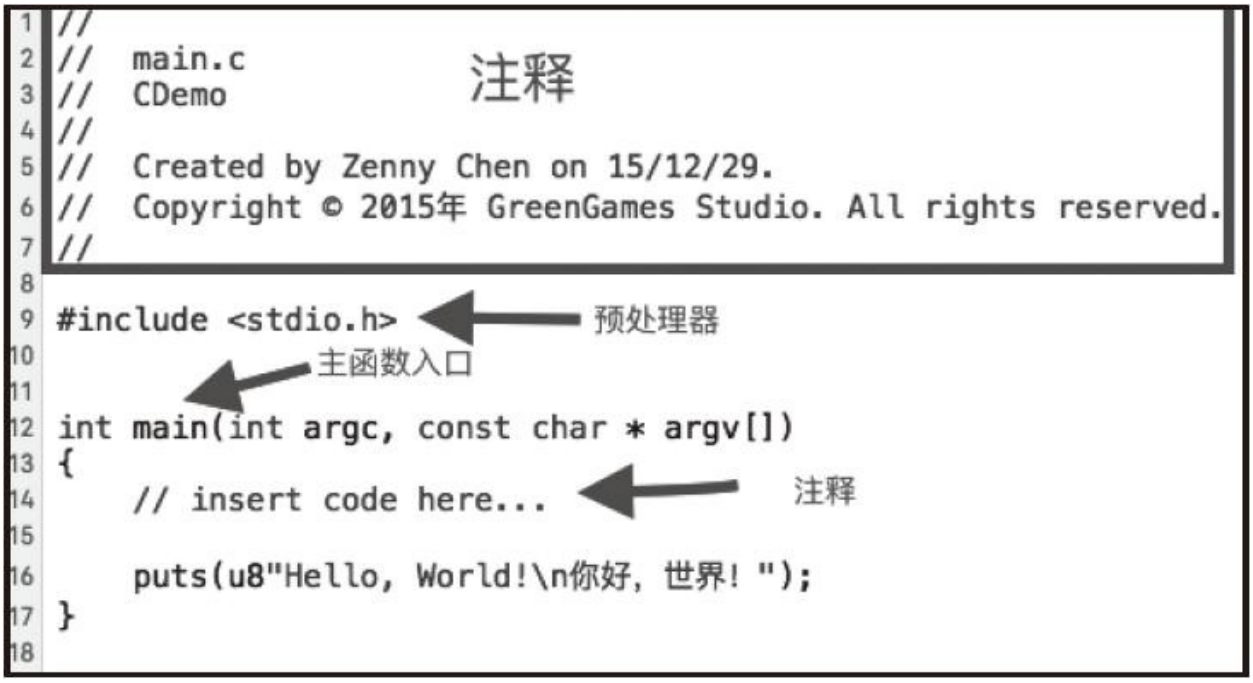


图4-1　C源文件的基本构成

我们在图4-1中能看到，一个可用来编译执行的基本C源文件主要包含4个部分。

第1部分是注释。注释主要用于给源代码做批注，方便阅读和维护。编译器会忽略所有注释部分，而且注释部分在预编译处理结束后就不存在

了。我们将在10.9节讨论程序注释。

第2部分是预处理器（Preprocessor）。图4-1中的第9行代码就是一条#include预处理器，它将标准库头文件“stdio.h”中的所有内容都直接放到当前源文件中，这样我们就可以将它看作在第9行这个位置插入此头文件的所有内容（这里我们可以先无视上面的注释部分的处理）。“stdio.h”文件包含了第16行所用到的puts标准库函数的原型。我们将在第10章详细讨论预处理器。

第3部分是主函数入口main。它是C程序的入口函数。也就是说，当操作系统加载完我们构建生成的C程序后，率先执行的就是main函数。关于main函数，我们将在9.9节中介绍。

第4部分是用{}包围着的函数具体实现代码（第13～17行）。这里的实现就是第16行打印输出两行文字。

C语言的头文件一般用.h后缀表示，源文件一般用.c后缀表示。C源文件是一个文本文件，所以它是由一系列字符构成的。下一节将介绍C源文件中可用的字符集以及执行C程序时可用的字符集。

4.1 C语言中的字符集

一般来说，编程语言的字符集都可分为两组：一组叫源字符集，另一组叫执行字符集。所谓“源字符集”是指在写C源代码时用的字符集，也就是呈现在C源文件中的字符集。而“执行字符集”是指编译构建完源文件后的目标二进制文件中所表示的字符集，它将用于运行在当前的执行环境中。比如，我们在控制台或者GUI窗口视图上所看到的文字信息就属于执行字符集。

C语言标准允许C语言实现采用多字符扩展字符集，但是必须要满足一组基本字符集。基本字符集都包含在ASCII码可显字符集中，包括半角的大写字母A～Z、小写字母a～z、半角的阿拉伯数字0到9以及下列符号：

```
！ " # % & ' （ ） * + ， - . / ： ； < =
> ？ [ \ ] ^ _ { | } ~
```

为了叙述方便，上述这排符号后续将统称为“标点符号”；而大小写半角英文字母统称为“字母”；半角阿拉伯数字0到9统称为“数字”。

由于在C语言的上述基本字符集中有9个字符超出了ISO 646不变字符集的范围，分别是：# \ ^ [ ] | { } ~。所以，在C90标准中就引入了三字符连拼（Trigraph）的方式来表达这9个字符：

```
？？= 对应于 #
```

? ? ）　对应于　]

? ? ！　对应于　|

? ? （　对应于　[

? ? '　对应于　^

? ? >　对应于　}

? ? /　对应于　\

? ? <　对应于　{

? ? -　对应于　~

例如：

```
??=def?ine arraycheck(a, b)  a??(b??)  ??!??!  b??(a??)
printf(“Eh???/n”);
// 上述代码等价于：
#define arraycheck(a, b)  a[b]  ||  b[a]
printf(“Eh?\n”);
```

这里我们还能再呈现一下源字符集与执行字符集的差异。上述代码中，“? ? ? /n”表示源字符集，它在C源文件中就是如此写的；而最后翻译成的“\n”就相当于执行字符集，显示在命令行程序中就是一个换行。

由于C++17标准打算废弃三字符连拼，笔者估计下一个C语言标准也将废弃三字符连拼机制，因此不建议各位使用，大家只要了解一下这个历史

即可。

C99标准中引入了对其中5个字符的双字符连拼（Digraph）表示。

<:　　对应于　[

:>　对应于　]

<%　对应于　{

%>　对应于　}

%:　　对应于　#

双字符连拼在下一个标准中还能正常使用。尽管Trigraph与Digraph基本用不上，不过在看一些较早之前欧洲一些国家的人所写的代码时能知道那是什么。由于笔者在日本做过一些项目，所以知道在Windows系统下的日语环境中，“\”这个符号会被显示成“￥”。因此当我们看到“￥”符号时能反应出是“\”就行。

4.2 C语言中的token

在编程语言中经常会涉及“token”这个词，token这里不是指网络通信中所谓的“令牌”，而是用于词法解析的，通过指定一个词位（词的单位）的类别来结构化表示该词位。如以下代码：

```
int a = 3 << 2;
```

这里就有7个token，分别是：int、a、=、3、<<、2以及最后的分号；。这一行代码中就已经列出了C语言中的常用几种token，分别是关键字（int）、标识符（a）、字面量（3和2）、操作符（=和<<）、其他标点符号（；）。每个token之间用空白符或标点符号进行分隔。空白符主要包括空格（white space）、制表符（tab）以及换行回车。像上述代码也能写成以下形式，两者是等价的。

```
int a=3<<2;
```

但是，这里int与a之间必须用空白符分割。

C语言标准中定义了token和预处理token，分别用于在编译时和预编译时的符号解析。token包括关键字、标识符、常量、字符串字面量以及标点符号。预处理token主要包括头文件名、标识符、预处理数、字符常量、字符串字面量、标点符号以及不属于上述符号的每个非空白字符。

下面我们将分别描述标识符、关键字、常量与字符串字面量、标点符号这几种token。预处理token将放在第10章做详细描述。

4.2.1 C语言中的标识符

在C11标准中提到，C语言中的标识符可以表示一个对象（object），一个函数（function），一个结构体（structure）、联合体（union）或枚举（enumeration）的一个名字（C11标准中将结构体、联合体以及枚举类型的名字称为tag）或其中一个成员、一个typedef名、一个跳转标签（label）名、一个宏（macro）名或一个宏的形参（parameter）。当我们提到“标识符”时，要意识到标识符不仅仅是上述所描述实体的名称，而且也是对它们的引用（reference）。

一般C语言的实现约定，一个标识符由基本字符集中的所有大小写英文字母、阿拉伯数字0到9以及下划线_构成，并且标识符不能以数字开头。比如：aBc、_ab、C11、_3_都是有效的标识符；5ab、a（2、886都是无效的标识符。有些C语言实现允许将$作为构成标识符的有效字符，但有些是将含有$的标识符作为一种内部使用的特殊符号来用，所以我们在命名标识符的时候应该避免使用$符号。此外，C11标准允许使用多字节扩展字符集（通用字符名）来命名标识符，但不能违背上述基本约定。比如，在Apple LLVM编译器中，允许使用中文、拉丁字母、希腊字母等作为标识符：αντιο、bonné、小鳥遊·六花、ラーメン等都是有效标识符，但是像3百九、

十*二，这些就是无效的标识符。此外，C语言标准中还规定，如果一个标识符含有通用字符名，那么每一个通用字符名必须落在ISO/IEC 10646编码方式的以下范围内（用十六进制表示）：

1）00A8，00AA，00AD，00AF，00B2～00B5，00B7～00BA，00BC～00BE，00C0～00D6，00D8～00F6，00F8～00FF；

2）0100～167F，1681～180D，180F～1FFF；

3）200B～200D，202A～202E，203F～2040，2054，2060～206F；

4）2070～218F，2460～24FF，2776～2793，2C00～2DFF，2E80～2FFF；

5）3004～3007，3021～302F，3031～303F；

6）3040～D7FF；

7）F900～FD3D，FD40～FDCF，FDF0～FE44，FE47～FFFD；

8）10000～1FFFD，20000～2FFFD，30000～3FFFD，40000～4FFFD，50000～5FFFD，60000～6FFFD，70000～7FFFD，80000～8FFFD，90000～9FFFD，A0000～AFFFD，B0000～BFFFD，C0000～CFFFD，D0000～DFFFD，E0000～EFFFD。

此外，标识符的第一个通用字符名不能落在以下范围内：0300～036F，1DC0～1DFF，20D0～20FF，FE20～FE2F。

在C语言标准中没有特别设定一个标识符的最大长度。不过具体的C语言实现可以根据自己的情况设定标识符最大长度。

在同一作用域（scope）内，一个标识符应该指定一个确切的实体。如果编译器在当前上下文中无法判定某个标识符用于引用哪个实体，那么就会发生编译错误。关于作用域的详细介绍请参见11.1节。

4.2.2 C语言中的关键字

在编程语言中所谓的“关键字”（keyword）是指被编程语言编译器保留用作特定语义的token，它们不能被程序员当作其他标识符来使用。C11标准中的关键字见表4-1。

表4-1 C11标准中的关键字

关键字	描　　述	可用的 C 语言标准
auto	自动存储类说明符	C90
break	循环与选择分支的退出语句	C90
case	选择语句块中的 case 语句	C90
char	字符类型	C90
const	常量类型限定符	C90
continue	循环中跳过当前迭代	C90

（续）

关键字	描　　述	可用的 C 语言标准
default	默认条件的 case	C90
do	与 while 语句连用，引出循环语句块	C90
double	双精度浮点类型	C90
else	其他条件的分支	C90
enum	枚举类型说明符	C90
extern	外部存储类说明符	C90
float	单精度浮点类型	C90
for	引出 for 循环语句	C90
goto	跳转语句	C90
if	条件分支语句	C90
inline	内联函数说明符	C99
int	整数类型	C90
long	长整数类型	C90
register	寄存器存储类说明符	C90
restrict	访存模式受限的指针类型限定符	C99
return	函数返回语句	C90
short	短整数类型	C90
signed	带符号的整数类型	C90
sizeof	获取类型与对象大小操作符	C90
static	静态存储类说明符	C90
struct	结构体类型说明符	C90
switch	选择语句	C90
typedef	类型定义存储类说明符	C90
union	联合体类型说明符	C90
unsigned	无符号整数类型说明符	C90
void	无类型	C90
volatile	易变存储对象的类型限定符	C90
while	引出 while 循环语句	C90
_Alignas	对齐说明符	C11
_Alignof	获取对象对齐操作符	C11
_Atomic	原子类型限定符	C11
_Bool	布尔类型	C99
_Complex	复数类型说明符	C99
_Generic	引出泛型表达式	C11
_Imaginary	虚数类型说明符	C99
_Noreturn	无返回函数说明符	C11
_Static_assert	静态断言	C11
_Thread_local	线程本地私有存储类说明符	C11

在上述关键字中有些是由大写、小写以及下划线混合组成的，各位在编写代码的时候需要注意大小写。这些关键字会从第5章开始分别进行介绍。

看到以上这些关键字读者可能会感到奇怪，为何有些关键字是以下划线打头的呢？以下划线打头的关键字均是从C99标准开始引入的。由于在C99之前，有不少C语言编译器已经对C99标准新引入的特性给予支持，为了防止C99标准的关键字与一些编译器已有的扩展关键字冲突，从而通过以下划线作为前缀，然后首字母大写来定义这些关键字。而通过C语言新标准引入新的标准库可使得这些关键字能被统一。

所以，大家在使用以下划线打头的关键字时，请尽量先引入相应的标准库头文件，然后使用非下划线形式的相应关键字。比如，<stdbool.h>头文件中将_Bool类型定义为了bool类型；<complex.h>头文件中将_Complex定义为了complex，等等。我们最好使用bool、complex来代替_Bool和_Complex，这样一来书写更为简洁，二来又有更好的向前兼容以及跨平台等特性。当然，还有一些关键字是没有相应标准库定义形式的，比如_Generic，我们在使用的时候直接用_Generic即可。

4.2.3 C语言中的常量与字符串字面量

C语言中，常量（contant）有4种，分别是整数常量、浮点数常量、枚举常量以及字符常量。每个常量都具有一个特定的类型以及该常量所指定

的值，常量值必须在其类型所能表示的范围内。整数常量和字符常量将在5.1节中描述；浮点数常量将在5.2节中描述；枚举常量将在6.1节描述。

字符串字面量我们之前已经见过了，图4-1中的u8“Hello，world\n你好，世界！”就是一个字符串字面量。在C11中，一个字符串字面量由一对双引号包裹的一系列的字符构成。如果字符串中含有诸如回车、双引号等字符的话，需要对它们进行转义，转义字符将在5.1.6节中描述。此外，字符串的第一个双引号前可以加u8、u和U这三种前缀。u8指定了该字符串字面量是一个UTF-8字符串；u表示该字符串字面量是一个UTF-16字符串；U表示该字符串字面量是一个UTF-32字符串。如果不加前缀，则默认为当前系统实现的字符编码格式。字符串字面量将在7.10节做进一步描述。

4.2.4 C语言中的标点符号

C语言的标点符号如下：

[ ] ( ) { } . ->

++ -- & * + - ~ !

/ % << >> < > <= >=

== != ^ | && || ? :

; ... = *= /= %= += -=

<<=　>>=　&=　^=　|=　，　#　##

<:　：>　<%　%>　%:　%:%:

标点符号是具有独立语法和语义意义的符号。它作为一个要执行的操作时，又称为操作符（operator）。操作符所作用的实体称为操作数（operand）。比如，3+2这个表达式中，+是一个操作符，表示整数加法操作。而3和2则是+的操作数，3作为+的左操作数；2作为+的右操作数。

上述列出的标点符号中，有些无法单独成为一个操作符，比如[、]，（、）等，而是需要将它们组合起来[　]、（　）才行。而在（　）操作符里边的表达式则作为该操作符的操作数。比如：（3+2）的操作数是表达式3+2。此外，有些标点符号可进行组合形成一个操作符，比如<<、+=、>>=等。这些组合标点符号之间不允许带有空白符，比如<<表示左移操作，而<　<仅仅表示两个小于号。

C语言中，操作符按照可作用的操作数个数来分可分为单目操作符（unary operator）、双目操作符（binary operator）和三目操作符（ternary operator）。

1）单目操作符有！（表示逻辑非）、&（用作地址操作符时）、*（作为间接操作符时）、+（表示正数符号时）、-（表示负数符号时）、~（表示按位取反）。

2）双目操作符有++（表示自增操作）、--（表示自减操作）。

3）三目操作符只有一组，即？与：的组合，作为条件表达式的操作符，这将在8.2节中详细描述。

其余的，除了#和##作为预处理操作符之外，上述列出的操作数中都是双目操作符。

不同的操作符可能会有不同的计算优先次序。在计算一个表达式时，如果该表达式含有多个操作符，那么这些操作符按照优先级高的先开始计算，然后再计算低优先级的操作。如果几个操作符具有相同优先级，那么按照从左到右的顺序依次计算。在C11标准中定义了如下表达式的计算优先次序，排列从高到低。

1）基本表达式：标识符、常量、字符串字面量、圆括号表达式（比如（3+2））、泛型表达式。

2）后缀操作符：数组下标（比如a[0]）、函数调用、结构体与联合体成员访问操作符（.和->）、后缀自增及自减操作符（比如a++；a--）、复合字面量（比如（int[]）{1，2，3}）。

3）单目操作符：前缀自增与自减操作符、地址操作符与间接操作符（比如++a；--a）、单目算术操作符（+、-、！、~，其中这里的+和-表示正负号）、sizeof操作符与_Alignof操作符。

4）类型投射操作符（详见5.6节）。

5）乘法操作符（包括乘、除、求余数*、/、%）。

6）加法操作符（+、-）。

7）移位操作符（左移、右移）。

8）关系操作符（大于、小于、大于等于、小于等于）。

9）相等性操作符（等于和不等于，==、！=）。

10）按位与操作符。

11）按位异或操作符。

12）按位或操作符。

13）逻辑与操作符。

14）逻辑或操作符。

15）条件操作符（即三目表达式）。

16）赋值操作符。

17）逗号操作符。

下面举一个简单的例子:

```
int a = 3 + 2 * 10 / 4 - -(3 - 2);
```

上述代码中，（3-2）最先被计算得到结果1，然后再计算-（1）的结果是-1，然后计算2*10的结果等于20，再计算20/4的结果等5，再是3+5的结果等于8，然后是8-（-1）的结果是9，最后是将结果9赋值给变量a。

4.3 关于C语言中的“对象”

C11标准将“对象”定义为执行环境中的数据存储区域，对象中的内容用于表达它的值。当引用了某一对象时，该对象就可称为具有一个特定类型。言下之意，C语言标准中的“对象”是指数据实体，而不是一个函数。此外，它具有一个特定的存储区域，无论是在寄存器中还是在存储器中。另外，它具有一个特定的类型。

C语言不是一门面向对象的编程语言，所以这里的“对象”与面向对象编程语言所涉及的对象概念有些差别，不过从范围上来讲，这里的“对象”比面向对象中的对象范围更广。从总体上将对象进行划分可分为两大类——变量和常量。

·变量是指在程序运行时，允许该对象所存放的值被修改。

·常量是指在程序运行时，该对象所存放的值不允许被修改。

在C语言实现中，常量可以被写入ROM，尤其对于嵌入式设备而言，更有可能如此。这样，一旦对某个常量对象进行修改，那么系统会直接发出异常。而在通用桌面操作系统中，常量也被分配在RAM中，所以我们仍然可以通过类型转换或是其他奇技淫巧对常量对象进行修改，不过后果是无法预估的。

在计算机编程语言中还有一个比较常见的概念就是字面量。在传统编

程语言中，字面量就是指在源代码中用于表示一个固定值的文字记号。

比如，像3、-10、3.14、"hello"等都属于字面量。

其中：

·3、-10表示整数字面量。

·3.14表示浮点数字面量。

·"hello"表示一个字符串字面量。

这些字面量往往都是常量，而像一般的整数字面量在概念上我们也无需关心它到底是不是一个对象，即不需要关心它有没有自己的存储空间。由于字面量以及像（3+2）等常量表达式是在编译时就能计算出结果的，所以对于这些字面量的算术逻辑计算也无需在程序运行时体现出来。

另外，C11还包括了结构体、联合体以及数组的复合字面量。这些复合字面量无需是常量，而且它们自己所包含的元素也完全可以是变量，并且在运行时也完全可被修改。

4.4 C语言中的“副作用”

在很多编程语言中都会提到“副作用”（side effects）这个概念。在C11标准中对副作用是这么描述的：对一个易变对象的访问、对一个对象的修改、对一个文件的修改，或调用一个函数，所有这些操作都具有副作用。副作用对执行环境中的状态做了改变。对一个表达式的计算通常包含了对值的计算以及对副作用的初始化。对一个左值表达式的值计算包含了判定该表达式所表示对象的标识。

通常来讲，所谓副作用就是在C源代码中的某一条表达式在目标程序中执行时，对当前程序的执行状态产生了或潜在产生改变，那么我们称该表达式产生了副作用。所谓程序执行状态包含了许多元素，比如对目标程序指令、寄存器的值、存储器中的数据等。

4.5 C语言标准库中的printf函数

我们这里先简单介绍一下本书后续会大量使用的控制台字符串输出函数printf。这是一个C语言标准库函数。printf函数的原型为：

```
int printf(const char * restrict format, ...);
```

此函数第一个参数format是一个字符串格式符，后面的省略号表示不定个数的参数，这些参数的数据类型需要分别与format所指向的字符串中的格式匹配。函数最后返回的是一个int类型整数，表示被传递到控制台的字符的个数。如果输出或者字符串编码发生错误，那么该函数将返回一个负值。但当前大部分编译器的实现并非返回传递到控制台的字符个数，而是字节个数，这对输出UTF-8编码的字符串时尤为如此。

下面简单介绍一下本书中常用的format字符串中的格式符。

1）%c：对应参数是一个int类型，但实际运行时会将该int类型对象转换为unsigned char类型。

2）%d：对应参数是一个int类型。

3）%f：对应参数是一个double类型。

4）%ld：对应参数是一个long int类型。

5）%s：对应参数是一个const char*类型，表示输出一个字符串。

6）%u：对应参数是一个unsigned int类型。

7）%zu：对应参数是一个size_t类型。

8）%td：对应参数是一个ptrdiff_t类型。

9）%x（或%X）：对应参数是一个int类型，不过会以十六进制形式输出，其中大于9的数字根据字母x大小写进行转换，如果是%x，则大于9的数用a～f表示；如果是%X，则用A～F表示。

10）%%：输出一个%符号。

各位可以在自己的计算机上尝试编写下列代码，熟悉一下pritnf函数的使用方式：

```
#include <stdio.h>
#include <math.h>

int main(int argc, const char * argv[])
{
    int len = printf("你好\n");
    printf("长度为：%d\n", len);
    printf("输出字符是:%c，输出浮点数是：%f\n", 'A', M_PI);
    printf("100的十六进制数为：0x%X\n", 100);

    const char *s = "hello, world!";
    printf("几乎100%会出现在编程语言教科书上的字符串是：%s\n", s);
}
```

各位可以编译运行上述代码。如果各位在某些Unix/Linux上实践，没有中文输入法也没有关系，可以用相应的英文来代替上述中文。另外，上述字符串中所出现的\n是一个转义字符，关于转义字符，我们将在5.1.6节

中加以描述。

4.6 本章小结

本章我们大概描述了C语言构成的基本元素。一开始，我们列出了一个完整的C语言源文件应该包含的几个部分。然后我们提到了C语言中的可用字符集以及各类符号与它们的定义。关于C语言执行环境限制的更多详细信息可参考此博文：http://www.cnblogs.com/zenny-chen/p/4251813.html。

通过本章学习，各位应该已经能体会到C语言书写的大致格式，并且通过本章列出的一些代码片段，自己能试试身手写一些简单短小的代码出来，然后利用printf函数可以打印出一些计算结果。

第5章　基本数据类型

本章将介绍C语言中的基本数据类型以及相关的算术逻辑运算。C语言中的基本数据分为两大类，一类是整数类型，另一类是浮点类型。整数类型还包括字符类型以及布尔类型。浮点数类型包括单精度浮点数、双精度浮点数以及扩展双精度浮点类型。

对任一整数对象和浮点数对象，我们都能用+、-、*、/对它们做加、减、乘、除算术运算操作，当然做除法时除数不能为零，否则会导致程序运行时异常。另外，对于整数之间的操作还能使用%（求模操作）进行求余数，比如5%2的结果为1，但浮点数之间不能进行求模操作。我们还能对整数做按位操作以及移位操作，同样这些操作不支持浮点数。

5.1 整数类型

C语言中整数类型包括int、short、long、long long、布尔、字符等，除了字符与布尔类型以外，其他所有整数类型都支持带符号与无符号的表示方式。关于带符号与无符号整数类型的表示方式可以参考第2章的内容。此外，C语言标准中没有明确规定每一种整数类型所占用的字节数，这些全都是由C语言的实现来定义的，但是C语言标准给了若干约束，所以C语言实现应该至少能满足这些约束。为了方便叙述，我们这里仍然根据主流桌面端编译器（GCC、Clang）以及主流32位与64位处理器环境的实现进行讲解。

5.1.1 int类型

用关键字int声明的一个整数对象具有int类型。在具体的C语言执行环境中，int数据的最小值与最大值分别定义为<limits.h>头文件中的INT_MIN和INT_MAX。在我们常用的32位与64位环境中，int默认为是带符号的（相当于signed int），占用4个字节（即32位），其最小值为-2^{31}（即0x80000000），最大值为$2^{31}-1$（即0x7FFFFFFF）。int所对应的无符号类型是unsigned int，通常在32位与64位环境下也占用4个字节，最小值为0，最大值为$2^{32}-1$（即0xFFFFFFFF）。在具体C语言执行环境中的最大值定义为<limits.h>头文件中的UINT_MAX。

int类型对应的整数字面量可直接按照自然方式书写，比如0、-128、127、+2233等都默认表示为int类型。此外，整数字面量可以分别使用八进制、十进制以及十六进制的方式进行表达。八进制的整数字面量表达方式为以0打头，比如：01、023、-0477这些都是属于八进制整数字面量。而十六进制整数字面量则是以0x或0X打头，比如：0x123、-0x0045、0xabcdef这些都是有效的十六进制整数字面量。而其他没有任何前缀的整数字面量都表示为十进制整数。如果想要表达一个unsigned int类型的整数字面量，可在一般整数字面量后直接添加字母u或U。本书习惯上使用大写的U。比如0U、01U、-128U、2048U、+2233U等都属于unsigned int类型。当然，即便字面量后面不加U后缀，这些数也能赋值给unsigned int类型的对象，因为它们会被编译器进行默认转换。此外，当我们要声明一个unsigned int类型的对象时，int可以省略。比如，unsigned a=0；，其中对象a的类型即为unsigned int类型，=是一个赋值操作符（assignment operators），将其右操作数0赋值给左操作数a。下面举一些例子，各位也可以在自己的计算机上试试，如代码清单5-1所示。

代码清单5-1　int类型介绍

```
#include <stdio.h>
#include <limits.h>

int main(int argc, const char * argv[])
{
    int a = 10;                             // 声明了int类型对象a，并且将整数10赋值给它
    // 以下语句声明了unsigned int类型对象b，并且将整数100赋值给它
    unsigned int b = +100U;

    unsigned c = -1U;                       // 声明了unsigned int类型对象c

    printf("a + b = %d\n", a + b);          // 这里a+b的结果为110
    printf("c = 0x%X\n", c);                // 这里，c为0xFFFFFFFF
    printf("a + c = %d\n", a + c);          // 这里加法结果溢出，但可将它看作为10-1的结果
```

```
    a = 0x7FFFFFFF;
    a += 1;                                 // a += 1相当于a = a + 1
    printf("a = %d\n", a);// 加法结果溢出，结果为-2147483648，相当于0x80000000

    printf("INT_MIN = %d\n", INT_MIN);  // 查看当前C语言实现下int类型的最小值
    printf("INT_MAX = %d\n", INT_MAX);  // 查看当前C语言实现下int类型的最大值

    // 查看当前C语言实现下unsigned int类型的最大值
    // 由于unsigned int的最小值已被标准定义为0
    printf("UINT_MAX = %u\n", UINT_MAX);
}
```

这里顺便再提一下，根据C语言标准，我们在一个C语言源文件的末尾处最好添加一个换行符，并且不再添加任何其他的空白符。在某些老版本的GCC上（比如3.x版本），如果源文件末不是以换行符结尾，则GCC编译器在编译后会有警告，要求在源文件末尾处添加一个换行符。

上述代码已经涉及了很多额外的知识，比如加减法运算结果溢出的行为，还有+=操作符的意义等。由于这些知识比较简单易懂，所以我们就不在正文中加以赘述，而直接以代码注释的方式给出。各位在自己的计算机上编译运行后自然就能知晓其用途。当然，各位在敲上述代码的时候，注释部分（//以及其后面的文字）不需要打出来，这些仅仅是对代码的注解，对程序本身没有其他作用。

5.1.2 short类型

short类型（标准表达为signed short int类型，其中signed与int均可省略）我们一般称之为短整型。在我们通常的32位及64位系统下占用2个字节（即16位），其最小值为-2^{15}（即0x8000），最大值为$2^{15}-1$（即0x7FFF）。在C语言执行环境下，其最大、最小值分别定义为<limits.h>头

文件中的SHRT_MAX和SHRT_MIN。short类型所对应的无符号类型为unsigned short（标准表达为unsigned short int，其中int可省）。它通常在32位及64位系统下占2个字节，最小值为0，最大值为2^{16}-1（即0xFFFF）。在C语言执行环境中，其最大值定义为<limits.h>头文件中的USHRT_MAX。

short类型与unsigned short类型没有特别对应的整数字面量，它们可直接用int与unsigned int相应的整数字面量进行赋值，如代码清单5-2所示的用法。

代码清单5-2　short类型介绍

```
#include <stdio.h>
#include <limits.h>

int main(int argc, const char * argv[])
{
    // 这里同时声明了short类型对象a和b
    // 并且将a赋值为100，b赋值为200
    short a = 100, b = 200;

    printf("a - b = %d\n", a - b);

    // 这里声明了unsigned short类型的对象c
    unsigned short c = 100;
    c -= 200;                           // 相当于c = c - 200;

    printf("c = %hu\n", c);             // 结果为65436（即65536 - 100）

    printf("SHRT_MIN = %d\n", SHRT_MIN);// 查看当前C语言实现下short类型的最小值
    printf("SHRT_MAX = %d\n", SHRT_MAX);// 查看当前C语言实现下short类型的最大值

    // 查看当前C语言实现下unsigned short类型的最大值
    // 由于unsigned short的最小值已被标准定义为0
    printf("USHRT_MAX = %u\n", USHRT_MAX);
}
```

代码清单5-2中，字符串格式符%hu对应一个unsigned short类型的参数。65536表示为2^{16}。这里表述了第2章介绍的概念，即如何将一个带符号整数（补码形式）转为无符号整数的表示形式。

5.1.3 long类型

long类型（标准表达为signed long int类型，其中signed与int均可省略）我们一般称之为长整型。在我们通常的32位环境下long类型占用4个字节（即32位），而在64位系统下，当前几个主流桌面编译器就有所区别了。MSVC与VS-Clang仍然为4个字节，而GCC与Clang则是8个字节（即64位）。long类型所对应的无符号类型为unsigned long（标准表达为unsigned long int，int可省），我们一般称之为无符号长整型，它的字节长度与long类型一致。在C语言执行环境中，long类型的最小值与最大值分别定义为<limits.h>头文件中的LONG_MIN与LONG_MAX。unsigned long类型的最大值定义为<limits.h>头文件中的ULONG_MAX，其最小值为0。

long类型对应的整数字面量是在int整数字面量后面加上英文字母l或L，本书都用大写字母L作为后缀。unsigned long对应的整数字面量是在unsigned int字面量后面加上字母l或L，本书采用的都是以UL作为后缀。一般情况下，我们直接用int字面量赋值给long类型的变量也不会有问题，但是当我们要表达一个超出int范围的整数时，我们就得加上后缀L，否则数据可能会被截断。不过这里，不同的编译器会有不同行为。如代码清单5-3所示的例子。

代码清单5-3　long类型介绍

```
#include <stdio.h>
#include <limits.h>

int main(int argc, const char * argv[])
```

```
{
    long a = 100;                                       // 声明了long类型对象a，并给它赋值为100

    // 这里0x100000000已经超出了一般int的范围
    // 所以我们在它后面加上后缀L，表示一个long类型的字面量
    // 倘若不加后缀L，在某些编译器上会出现警告，在运行时也可能出现数据被截断的情况
    long b = 0x100000000L;

    // 这里用%ld表示对应一个long类型的参数
    printf("a + b = %ld\n", a + b);

    // 声明一个unsigned long类型对象c
    // 100UL即表示一个unsigned long的整数字面量
    unsigned long c = 100UL;

    // 声明一个unsigned long类型对象d
    // 1000000表示一个int类型的整数字面量
    // 但由于unsigned long的精度至少不小于int类型的精度，
    // 所以int可以隐式转为unsigned long类型
    unsigned long d = 1000000;

    // 这里用%lu表示对应一个unsigned long类型的参数
    printf("c * d = %lu\n", c * d);

    printf("LONG_MIN = %ld\n", LONG_MIN);// 查看当前C语言实现下long类型的最小值
    printf("LONG_MIN = %ld\n", LONG_MAX);// 查看当前C语言实现下long类型的最大值

    // 查看当前C语言实现下unsigned long类型的最大值
    // 由于unsigned long的最小值已被标准定义为0
    printf("ULONG_MAX = %lu\n", ULONG_MAX);
}
```

对于代码清单5-3，我们最好在64位的Linux或macOS（或FreeBSD）下用GCC或Clang编译器编译运行，因为在Windows 64位下用MSVC或MS-Clang编译器的话，long类型仍然占4个字节，看不到某些效果，除非使用MingW-W64或64位的Clang编译器。

不过，正因为long和unsigned long在不同环境下字节长度不同，所以我们在定义一个整数对象时应当尽量避免使用long类型，除非涉及系统相关的一些属性。比如C语言标准库中将获取文件当前位置（ftell）等函数的返回类型作为long类型。但对于一般应用程序而言，我们需要慎用long类型。

5.1.4　long long类型

C语言标准对long long（标准表达为signed long long int，其中signed与int可省）类型提得不多，仅仅阐述了long long类型的精度至少为long int类型的精度。不过在当前几大主流桌面编译器中，无论是32位系统还是64位系统，long long的宽度均为8个字节（即64位）。其最小值为-2^{63}，最大值为$2^{63}-1$。long long对应的无符号类型为unsigned long long（标准表达为unsigned long long int，int可省），同样也是8字节的宽度，最小值为0，最大值为$2^{64}-1$。在C语言执行环境下，long long的最小值与最大值分别定义为<limits.h>头文件中的LLONG_MIN与LLONG_MAX。unsigned long long类型的最大值定义为<limits.h>头文件中的ULLONG_MAX。

long long对应的整数字面量表示为int整数字面量后加后缀ll或LL，本书采用LL后缀。unsigned long long对应的整数字面量表示为unsigned int整数字面量后面加后缀ll或LL，本书采用ULL作为后缀。参见代码清单5-4。

代码清单5-4　long long类型介绍

```
#include <stdio.h>
#include <limits.h>

int main(int argc, const char * argv[])
{
    long long a = 100;          // 声明了long long类型对象a，并给它赋值为100

    // 这里0x100000000已经超出了一般int的范围
    // 所以我们在它后面加上后缀LL，表示一个long long类型的字面量
    long long b = 0x100000000LL;

    // 这里用%lld表示对应一个long long类型的参数
    printf("a + b = %lld\n", a + b);

    // 声明一个unsigned long long类型对象c
    // 100ULL即表示一个unsigned long long的整数字面量
```

```
    unsigned long long c = 100ULL;

    // 声明一个unsigned long long类型对象d
    // 1000000表示一个int类型的整数字面量
    // 但由于unsigned long long的精度至少不小于int类型的精度,
    // 所以int可以隐式转为unsigned long long类型
    unsigned long long d = 1000000;

    // 这里用%llu表示对应一个unsigned long long类型的参数
    printf("c * d = %llu\n", c * d);

    // 查看当前C语言实现下long long类型的最小值
    printf("LLONG_MIN = %lld\n", LLONG_MIN);

    // 查看当前C语言实现下long long类型的最大值
    printf("LLONG_MAX = %lld\n", LLONG_MAX);

    // 查看当前C语言实现下unsigned long long类型的最大值
    // 由于unsigned long long的最小值已被标准定义为0
    printf("ULLONG_MAX = %llu\n", ULLONG_MAX);
}
```

5.1.5 布尔类型

在计算机编程语言中，布尔类型的对象是一个二值数据对象。布尔类型用于表达真假逻辑关系，一般用true表示真，false表示假。产生布尔值的表达式称为逻辑表达式或关系表达式（比如，大于、等于、小于、不等于等关系操作的结果）。在C11标准中，布尔类型用关键字_Bool声明，并说明布尔类型只要能够存放0和1值就行，也就是至少为1个比特。所以现在大部分对_Bool的C语言实现都将它作为1个字节的宽度。此外，_Bool类型不能用signed和unsigned来修饰。

在C语言刚被创建的时候，它并不具备“布尔类型”这个概念，而仅仅用0（浮点数则为0.0）与对象比较来判定真假。如果对象的值等于零，那么表示“假”，否则表示“真”。所以，即便从C99开始引入了_Bool布尔类型，之前的这个约定依然沿用。为了能与C++兼容，C语言从C99标准开始

就引入了<stdbool.h>头文件，里面用bool这个宏来定义_Bool，用true定义为1，false定义为0。bool、true以及false都不属于C语言中的关键字，它们仅属于标准库中定义的类型和常量。在C11标准的语言核心中，依然只定义了_Bool这个关键字表示布尔类型，而没有定义真值和假值的字面量。所以，我们在使用布尔类型的对象时，最好引入<stdbool.h>头文件，然后用bool定义布尔类型对象，用true表示真值常量，false表示假值常量。下面我们举一些例子来说明。

代码清单5-5　布尔类型介绍

```
#include <stdio.h>
#include <stdbool.h>

int main(int argc, const char * argv[])
{
    // 声明了一个布尔类型的对象a，并将它初始化为真
    bool a = true;

    // 声明了一个布尔对象b，并将它初始化为假
    bool b = false;

    // 声明了一个布尔对象c，并用a与b是否相等的比较结果对它初始化
    bool c = a == b;                    // 这里c为假
    printf("c = %d\n", c);              // 输出c = 0

    c = a != b;                         // 用a != b（a不等于b）的比较结果给对象c
    printf("a!=b的结果：%d\n", c);  // 输出1

    c = 10 > 5;
    printf("10 > 5? %d\n", c);              // 输出1

    c = 10 < 5;
    printf("10 < 5? %d\n", c);              // 输出0

    a = 0x10000000;
    printf("a = %d\n", a);              // 输出1

    b = 0.0;
    printf("b = %d\n", b);              // 输出0

    // 输出_Bool数据对象类型的宽度
    printf("_Bool type size is: %zu\n", sizeof(_Bool));
}
```

在代码清单5-5中，我们用GCC或Clang编译器编译之后能发现，_Bool

类型的对象只占用一个字节。另外，对于a=0x10000000，a的值不是简单地对0x10000000进行高位截断，只取最低1个字节获得结果，而是相当于将0x10000000！=0的结果给了布尔对象a。同样，下面的b=0.0也不是说就把0.0这个双精度浮点数赋值给布尔对象b，而是把0.0！=0.0的结果赋值给布尔对象b。这些都会由C语言编译器自动转换处理。

5.1.6 字符类型

C语言中用关键字char来声明一个字符类型。C11标准阐明了一个char类型的对象必须至少能存放基本执行字符集，并且如果一个基本执行字符存放在一个char类型的对象中的话，那么该char类型的对象的值必须保证为非负整数。所以，通常C语言的实现都会将char类型的宽度设置为一个字节，这样正好至少能存放ASCII码字符集。

这里各位需要当心的是，有些编译器会默认将char类型设定为无符号的，即char类型的整数是一个无符号的8位整数。所以，我们如果要用char类型定义一个带符号的8位整数，需要显式地使用signed char，这里signed不应该被省略。如果要声明一个无符号8位整数，则使用unsigned char。C11标准明确指出，char、signed char与unsigned char统称为字符类型，但三者在类型上是不兼容的，尽管char在数值表示范围上可能与unsigned char相同，或与signed char相同。因此如前所述，我们在编写程序的时候，用signed char来指定8位带符号整数；unsigned char来指定无符号8位整数；

char用于指定一个基本字符对象。signed char的最大、最小值分别定义为<limits.h>中的SCHAR_MAX与SCHAR_MIN。unsigned char的最大值定义为<limits.h>中的UCHAR_MAX，最小值为0。char的最大、最小值分别定义为<limits.h>中的CHAR_MAX和CHAR_MIN。

8位带符号与无符号的整数字面量没有特定的字面量表示方式，直接用int与unsigned int类型的整数字面量即可。而字符字面量则是用单引号，里面包含一个或多个字符。比如'a'、'123'都是有效的字符字面量。C11标准明确规定，一个字符字面量具有int类型，如果将一个字符字面量赋值给一个char类型的对象，那么将该字符字面量的最低有效位赋值给它，比如在我们通常的执行环境中就是将字符字面量的最低字节赋值给char类型的对象。

在C语言中，不是所有的字符字面量都能回显在文本编辑器中，另外还有一些字符具有特殊作用，比如换行、制表符等，而且像单引号本身也表示一个字符字面量的开头或结尾，所以对于这些特殊字符，我们通过使用转义字符的方式来表示它们。下面列举一下C语言中的转义字符。

1）单引号：用\'。

2）双引号：用\"。

3）问号：用\? ，不过一般我们可以直接使用'? '，无需使用此转义字符。

4）倒斜杠\：用\\。

5）用八进制编码表示的一个字符：\后面紧跟1到3个八进制数。比如：\7、\12、\123等。

6）用十六进制编码表示一个字符：\x后面跟一个十六进制数。比如：\x0a、\x30等。

 注意：\x后面所跟的所有能有效表示为十六进制数的字符（即0～9，大写字母A～F以及小写字母a～f）都作为当前单个十六进制编码的字符，直到遇到无法有效表示十六进制数的字符为止。另外，如果\x后面不是紧跟一个有效的十六进制字符，那么编译器将会报错。所以\x后必须至少跟一个有效的十六进制字符。

7）\a：表示报警。该字符会产生一个可听到的或可见到的警报，但不改变当前的游标位置。

8）\b：表示回退。该字符将当前游标移动到当前行的前一个位置。

9）\f：表示换页。该字符将当前游标移动到下一个逻辑页的初始位置。

10）\n：表示换行。该字符将当前游标移动到下一行的初始位置。

11）\r：表示回车。该字符将当前游标移动到当前行的初始位置。

12）\t：表示水平制表符。该字符将当前游标移动到当前行的下一个水平表格单元位置。

13）\v：表示垂直制表符。该字符将当前游标移动到下一垂直表格单元位置的初始位置。

14）\0：表示空。值为0的字符在C语言中一般用于字符串的结束符。C语言标准库中的很多库函数都以\0字符作为一个字符串末尾的判断依据。

下面我们将举一些例子来描述这些字符类型及其使用方法。

代码清单5-6　字符类型介绍

```
#include <stdio.h>
#include <limits.h>

int main(int argc, const char * argv[])
{
    signed char a = 100;                        // 声明了一个带符号8位整数对象a
    signed char b = -10;                        // 声明了一个带符号8位整数对象b
    printf("a - b = %d\n", a - b);      // 结果输出110

    unsigned char c = 200;              // 声明了一个无符号8位整数对象c
    unsigned char d = 50;               // 声明了一个无符号8位整数对象d
    printf("c - d = %d\n", c - d);      // 输出结果150

    // 输出signed char的最小值
    printf("SCHAR_MIN = %d\n", SCHAR_MIN);

    // 输出signed char的最大值
    printf("SCHAR_MAX = %d\n", SCHAR_MAX);

    // 输出unsigned char的最大值
    printf("UCHAR_MAX = %d\n", UCHAR_MAX);

    char ch = 'a';                      // 声明一个字符对象ch，并用字符'a'对它初始化
    printf("ch = %c\n", ch);            // 输出a

    // 对于多字符的字符字面量，某些编译器会给出警告
    // 这里会将int类型的字符字面量'abc'截取其最低字节'c'给对象ch
    // 其他高字节部分被舍弃
    ch = 'abc';
    printf("ch = %c\n", ch);            // 输出c

    // 这里不会有警告，并且
    // 字符a位于对象s的最低字节位置，字符\0位于对象s的最高字节位置
    int s = '\0cba';
    printf("s = %s\n", (char*)&s);  // 输出abc
```

```
    // 八进制数060相当于十六进制数0x30，对应于ASCII码的罗马数字0
    ch = '\060';
    printf("ch = %c\n", ch);            // 输出0

    // 八进制数0101相当于十六进制数0x41，对应于ASCII码的大写字母A
    ch = '\101';
    printf("ch = %c\n", ch);            // 输出A

    ch = '\x42';                                    // 十六进制数0x42对应于ASCII码中的大写字母B
    printf("ch = %c\n", ch);            // 输出B

    // 无效的转义符，在Clang中此时ch的值为'8'
    // 因为八进制数中的每一位数都是在0到7的范围内
    ch = '\8';

    // 无效的转义符，在Clang编译器中直接编译报错
    // 因为字母g不是一个能表达十六进制数的有效字符
    ch = '\xg0';

    // 输出char类型的最小值
    printf("CHAR_MIN = %d\n", CHAR_MIN);

    // 输出char类型的最大值
    printf("CHAR_MAX = %d\n", CHAR_MAX);
}
```

代码清单5-6中，对于printf（"s=%s\n"，（char*）&s）；这条代码我们用到了投射操作与指针的概念，这些知识稍后会做详细介绍。这里仅仅给大家阐明多字节字符字面量赋值给一个int对象后，其在小端模式下存储数据的方式。说到这里，聪明的读者可能会有这么一个问题：因为'\060'表示的是一个八进制数，如果想要表达一个{'\0'，'4'，'3'，'2'}这么一个多字节字符该怎么做呢？因为'\043'本身是一个有效的八进制转义字符，对应于十六进制数0x23，在ASCII码表中对应于#字符。所以'\0432'相当于'#2'。此时，我们能想到的是，对于一个八进制转义字符，在\后面最多只能跟3个八进制数，所以我们索性将\0写作为\000即可解决这个问题。当然，我们也可以分别将后面的三个字符'4'、'3'、'2'分别用八进制或十六进制转义字符来代替。我们可以实践一下以下代码:

```
#include <stdio.h>

int main(int argc, const char * argv[])
```

```
{
    // 这里s的值相当于'\0#2'
    int s = '\x0\0432';
    printf("s = %s\n", (char*)&s);  // 输出2#

    // 这里s的值相当于最高字节为字符'\0'
    // 低三个字节依次为'432'
    s = '\000432';
    printf("s = %s\n", (char*)&s);  // 输出234

    // 或者可以这么写
    s = '\0\x34\x33\x32';
    printf("s = %s\n", (char*)&s);  // 输出234
}
```

5.1.7 宽字符以及Unicode字符类型

从C99标准中引入了wchar_t类型来表示一个多字节字符。wchar_t并不是C语言的一个关键字，而是定义在<stddef.h>头文件中的一个宏类型。wchar_t类型在不同环境，其长度也可能不一样，C语言标准没有规定它必须占用多少字节。当前编译器一般将wchar_t定义为4个字节的宽度，有些老的编译器可能为2个字节。宽字符的字面量为一般字符字面量前加大写字母L前缀，这里各位要注意，必须是大写字母，不能是小写的。比如，L'a'、L'你'等都属于wchar_t类型的宽字符字面量。宽字符在C语言中的定义比较模糊，它主要根据当前系统的语言环境设置，可能是UTF-16编码、GB2312、拉丁系编码格式等。宽字符在不同语言环境下，其相应的所显示出来的字样都可能会不同。由此，C语言标准组织在C11标准中引入了Unicode字符类型。

C11中主要引入了UTF-8字符串、UTF-16字符以及字符串类型和UTF-32字符及字符串类型。正如在2.6节所描述的，UTF-8编码的长度范围为1～4个字节，所以在C语言中可以直接用char类型来表示当前一个UTF-8编码

字符的一个字节，它已经涵盖了基本的ASCII码。如果要表示中文、日文等UTF-8字符的话，则需要使用字节数组。C11中，引入了新的头文件<uchar.h>，其中定义了UTF-16字符类型----char16_t以及UTF-32字符类型----char32_t。不过C语言标准委员会做得非常灵活，在标准中提到，当C语言编译器预先定义了__STDC_UTF_16__这个宏时，char16_t才保证被用作为UTF-16编码；当预先定义了__STDC_UTF_32__这个宏时，char32_t才保证被用作为UTF-32编码；否则char16_t和char32_t可能会留作其他字符编码类型使用。在C11中，UTF-16的字面量是在普通字符字面量前加小写字母u，比如u'a'、u'我'等都是UTF-16字符字面量；UTF-32的字面量是在普通字符字面量前加大写字母U，比如U'b'、U'好'等都是UTF-32字符字面量。同样，C11标准没有明确提到char16_t与char32_t的宽度，现在编译器一般将char16_t定义为unsigned short类型，占2个字节；将char32_t定义为unsigned int类型，占4个字节。

鉴于现在UTF-8和UTF-16都用得非常普遍，我们将在以下代码例子中简单描述一下宽字符以及UTF16字符和UTF-8字符串的使用如代码清单5-7所示。而关于字符串，我们将在7.10节做详细描述。

代码清单5-7　宽字符与Unicode字符介绍

```
#include <stdio.h>
#include <uchar.h>
#include "zenny_utftrans.h"

int main(int argc, const char * argv[])
{
    // 定义了一个长度为32个字节的char数组
    // 这里后面用于存放 UTF-8多字节字符
    char buffer[32];

    // 声明了一个wchar_t类型的对象a
```

```
    wchar_t a = L'你';

    // 声明了一个UTF-16变量utf16Char
    char16 utf16Char = u'好';

    // 在utf16Str数组中依次存放a中的宽字符、utf16Char的UTF-16字符
    // 以及u'\0'表示UTF-16字符串的终结符
    char16 utf16Str[] = { a, utf16Char, u'\0' };

    ZennyUTF16ToUTF8(buffer, utf16Str, NULL);

    printf(u8"字符串为：%s\n", buffer);

    printf("wchar_t宽度为：%zu\n", sizeof(a));
}
```

各位需要注意的是，头文件<uchar.h>尚未包含在macOS等部分Unix系统中，所以我们用unsigned short来代替char16_t，或者我们可以用代码清单5-8的代码自己建一个uchar.h头文件。而在Windows系统中倒是已经包含了，各位可以在VS-Clang、MinGW等编译器中直接使用。不过无论在哪种系统环境下，GCC和Clang编译器对UTF-16以及UTF-32字符的字面量都已经支持。

代码清单5-8　一个简单的uchar.h头文件

```
#pragma once
#define _UCHAR

#include <stdint.h>

#define __STDC_UTF_16__
#define __STDC_UTF_32__

#if !defined(__cplusplus)
typedef uint_least16_t char16_t;
typedef uint_least32_t char32_t;
#endif
```

各位可以看到，我们在代码清单5-8中用的是uint_least16_t类型来定义char16_t；使用uint_least32_t类型来定义char32_t。这都是在C11标准中所采用的定义方式。关于类型定义的相关内容，各位可以详细参考13.4节内

容。

如果当前系统没有支持<uchar.h>，那么我们也就无法使用Unicode库函数，因此笔者这里自己实现了UTF-16字符串与UTF-8字符串编码之间相互转换的函数库，头文件为"zenny_utftrans.h"，各位可在第20章中获得完整的源代码。一般，C语言运行时环境对宽字符的输入/输出做得都不太成熟，宽字符库函数使用起来也比较麻烦，所以通常我们还是以UTF-8字符串的形式加以输出。最后，以上代码各位最好运行在默认语言环境以UTF-8进行编码的系统上。在Windows系统上，由于新建文件的编码格式都是根据本地语言环境设置的，中文环境下默认为GBK编码。所以，如果我们想要将当前源文件的字符编码格式转换为UTF-8，那么可以通过打开记事本，然后将当前文件另存为UTF-8编码格式，最后再覆盖原有文件即可。不过，由于Windows系统的打印函数的输入字符串也需要与当前系统支持的语言环境编码格式一致，所以如果要打印输出字符串的话，需要最终通过系统API将UTF-8格式编码转换为当前系统默认的编码格式。我们看代码清单5-9的内容。

代码清单5-9　Windows系统下将UTF-8编码字符串转为系统默认编码字符串

```
#include <Windows.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

static void UTF8ToDefaultString(char dst[], const char *pUTF8SrcStr)
{
    if (dst == NULL || pUTF8SrcStr == NULL)
        return;

    // 源UTF-8字符串的字节个数
```

```
    const size_t srcLen = strlen(pUTF8SrcStr);
    if (srcLen == 0)
        return;

    // 声明wideChars作为中间转换的宽字符串
    wchar_t *wideChars = NULL;
    // 获取宽字符串的实际转换长度
    const int wideStrLen = MultiByteToWideChar(CP_UTF8, 0, pUTF8SrcStr,
                           (int)srcLen, NULL, 0);
    // 动态分配wideChars
    wideChars = malloc(wideStrLen);

    // 做UTF-8字符串到宽字符串的转换
    MultiByteToWideChar(CP_UTF8, 0, pUTF8SrcStr, (int)srcLen,
                        wideChars, wideStrLen);

    // 获取系统默认编码的目的多字节字符串的长度
    const int dstLen = WideCharToMultiByte(CP_ACP, 0, wideChars,
                       wideStrLen, NULL, 0, NULL, NULL);
    // 将中间宽字符串转换为系统默认编码的目的多字节字符串
    WideCharToMultiByte(CP_ACP, 0, wideChars, wideStrLen, dst, dstLen,
                        NULL, NULL);

    // 在目的字符串最后添加'\0'作为结束符
    dst[dstLen] = '\0';

    // 释放wideChars
    free(wideChars);
}

int main(void)
{
    // 声明存放系统默认的多字节字符数组
    char dstChars[32];
    const char *utf8Str = u8"你好，世界！";

    // 将UTF-8编码格式的字符串转换为系统默认编码的多字节字符串
    UTF8ToDefaultString(dstChars, utf8Str);
    // 输出目的字符串
    puts(dstChars);

    getchar();
}
```

各位在编译运行代码清单5-9之前请务必确认当前的C源文件的字符编码格式为UTF-8，否则编译会不通过。此外，上述代码如果使用的Visual Studio集成开发环境，那么只能使用MSVC，因为当前VS-Clang对UTF-8编码格式的源文件支持不好，会引发编译错误。当然，如果我们在Windows系统下直接使用MinGW或Clang编译器也能正常编译通过。

另外，笔者是在macOS系统上运行的上述代码。在macOS系统下，默认的宽字符编码格式正好为UTF-16编码，所以与char16_t类型兼容。在其

他环境下就未必能正常显示上述输出字符串了，请各位读者注意。

5.1.8 size_t与ptrdiff_t类型

size_t在之前的标准中主要用于sizeof操作符的返回类型。C11标准引入了_Alignof操作符之后，它的返回类型也是size_t。size_t定义在<stddef.h>头文件中。通常我们使用size_t作为一个指针（或地址）转换一个整数的方式，它一般是无符号的。在MSVC与MS-Clang编译器中，32位环境下被定义为unsigned int，64位环境下被定义为unsigned long long。在GCC和Clang编译器中，无论是32位还是64位环境，size_t都被定义为unsigned long，因为unsigned long在GCC和Clang中，在32位环境下是32位的，在64位环境下是64位的。这么一来，无论是哪个编译器，size_t数据类型都能存放当前系统环境下的一个地址长度。我们将在5.5节详细讲解sizeof操作符；6.5.1节介绍_Alignof操作符；第7章详细讲解指针与地址。

ptrdiff_t类型用于两个指针相减后的结果类型，它是带符号的，在<stddef.h>头文件中定义。在通常C语言实现中，它的宽度与size_t相同，仅有的区别是ptrdiff_t是带符号的，而size_t则往往是无符号的。下面的例子会涉及后续章节的知识（见代码清单5-10），读者可以先了解一下，等学到相关知识后可以回过头来再仔细思考。

代码清单5-10　size_t与ptrdiff_t

```
#include <stdio.h>
```

```
#include <stddef.h>

int main(int argc, const char * argv[])
{
    // 用size_t声明了对象a
    size_t a = sizeof(a);       // a的值为a类型的宽度（即占用多少字节）

    // 定义了指向size_t类型的指针对象p，并将它指向对象a的地址
    size_t *p = &a;

    int b = 100;

    // 定义了指向int类型的对象q，并将它指向对象b的地址
    int *q = &b;

    size_t s1 = (size_t)p;
    size_t s2 = (size_t)q;

    // 这里，对size_t类型的格式符需要加前缀z
    printf("Address of a is: 0x%16zX\n", s1);
    printf("Address of b is: 0x%16zX\n", s2);

    // 这里，对ptrdiff_t类型的格式符需要加前缀t
    ptrdiff_t diff = (ptrdiff_t)q - (ptrdiff_t)p;
    printf("Address of a minus address of b is: %td\n", diff);
}
```

在代码清单5-10中，#include<stddef.h>可省，因为它已经包含在<stdio.h>头文件中了。

5.1.9 C语言中的标准整数类型

在前面所讲述的整数类型中，我们已经提起过像int、long之类的类型在不同的编译运行环境下可能会有不同字节长度，尤其是long类型。为了能使代码适应更广泛的编译执行环境，我们在编写C语言代码时可以考虑使用从C99标准就已经引入的标准整数类型。标准整数类型一般被定义在<stdint.h>头文件中，主要包含以下几类。

1）固定宽度的整数类型：当前标准能够支持int8_t、uint8_t、int16_t、uint16_t、int32_t、uint32_t、int64_t、uint64_t这些常用的类型。使用这些

类型之后，我们就无需纠结signed char的字节宽度、short的字节宽度、long的字节宽度等都是多少，因为这些类型从字面上就已经表明了它们分别占多少字节。比如int8_t的宽度就是1个字节（8比特）；int32_t则是4个字节（32比特）。此外，以int作为前缀的类型表示是带符号的类型；以uint为前缀的类型表示无符号类型。所以，我们今后写C语言代码时应当优先考虑这些标准整数类型。除了这些常用的标准整数类型外，C11标准还定义了其他固定宽度的标准类型，不过这些类型都是可选的，C语言实现没必要一定支持，比如：int24_t、uint24_t、int40_t、uint40_t、int48_t、uint48_t、int56_t、uint56_t。

2）最小宽度整数类型：这些整数类型表示至少需要满足所指定的比特位数，但允许占用更多的比特位。这些类型主要有：int_least8_t、uint_least8_t、int_least16_t、uint_least16_t、int_least32_t、uint_least32_t、int_least64_t、uint_least64_t。此外，还有可选的24、40、48、56比特宽度的最小宽度整数类型。

3）最快最小宽度的整数类型：这些整数类型往往用于快速计算。它们与最小宽度整数类型类似，至少需要满足所指定的比特位数。不过与最小宽度整数类型不同的是，它们往往具有更快速的计算速度。比如说，有些硬件（比如AMD的基于GCN架构的GPU）具有24位整数的快速乘法计算，那么当程序员使用了int_fast24_t时，则能暗示编译器生成利用这种快速乘法的指令。这些类型主要包括：int_fast8_t、uint_fast8_t、int_fast16_t、uint_fast16_t、int_fast32_t、uint_fast32_t、int_fast64_t、uint_fast64_t。另

外，还有可选的24、40、48、56比特宽度。

4）能存放对象指针的整数类型：该类型有intptr_t与uintptr_t两个。前者是带符号的，后者是无符号的。这个类型用于将一个对象的地址或是一个指针对象的值用一个整数存放起来。

5）最大宽度的整数类型：这种类型表示当前C语言实现能容纳所有整数的最大整数类型，有intmax_t和uintmax_t这两个。代码清单5-11描述了以上这些标准整数类型的使用。

代码清单5-11　标准整数类型

```
#include <stdio.h>

// 由于<stdbool.h>与<stdint.h>没有被包含在<stdio.h>头文件中，所以需要另外引入
#include <stdbool.h>
#include <stdint.h>

int main(int argc, const char * argv[])
{
    // 声明一个标准布尔类型对象b
    bool b = true;

    // 声明一个字符类型对象c
    char c = 'A';

    // 声明一个带符号8位整数对象s8
    int8_t s8 = 10;

    // 声明一个无符号16位整数对象u16
    uint16_t u16 = 100;

    // 声明一个带符号32位整数对象s32
    int32_t s32 = 1000;

    // 声明一个无符号64位整数对象u64
    // 这里的整数字面量仍然可以用ULL作为后缀，以确保精度不丢失
    uint64_t u64 = 10000ULL;

    printf("b = %d, c = %c, s8 = %d, u16 = %u, s32 = %d, u64 = %llu\n",
           b, c, s8, u16, s32, u64);

    // 声明了至少需要8比特的整数对象l8
    int_least8_t l8 = 30;

    // 声明了至少需要16比特的快速计算整数类型对象f16
    uint_fast16_t f16 = 40;

    printf("l8 + f16 = %d\n", l8 + f16);
```

```
    // 声明了一个能存放对象指针的无符号整数类型对象p
    uintptr_t p = (uintptr_t)&b;    // p这里存放了对象b的地址

    // 声明了一个能存放对象指针的带符号整数类型对象diff,
    // 并且用对象b的地址与对象c的地址的差对其初始化。
    // 这里也可以使用ptrdiff_t类型
    intptr_t diff = (intptr_t)&b - (intptr_t)&c;

    printf("p = 0x%.16zX\ndiff = %td\n", p, diff);

    // 这里输出最大整数类型占多少字节
    printf("intmax_t size: %zu bytes\n", sizeof(intmax_t));
}
```

5.2 浮点类型

当前在C语言中有3种实数浮点类型，分别为float、double与long double。C语言标准仅仅规定了float类型的精度是double类型精度的子集；double类型精度是long double精度的子集。在一般C语言实现中，将float类型设定为32位单精度浮点型，并采用IEEE754中的规格化浮点数表示方法；将double类型设定为64位双精度浮点型，并采用IEEE754中的规格化浮点数表示方法；long double在x86架构处理器下表示扩展双精度浮点（80位浮点数，一般占用16个字节），这是Intel自己扩展出来的浮点数格式。而在ARM等其他处理器架构下，long double可能与double类型一样，表示双精度浮点类型，但宽度仍然可能是16字节，而不是8字节。在一些GPU、DSP或嵌入式处理器中，浮点类型可能支持部分IEEE754标准，甚至使用其他表示法也有可能，所以各位使用时需要注意。但在大部分桌面环境以及智能设备上都基本可以满足IEEE754标准。

在C语言中，浮点数字面量的表达方式非常丰富，最基本的就是如0.1、-100.05等这种正常的十进制浮点在数学上的表示方法。在十进制浮点数后面添加f或F后缀，表示该字面量是float类型浮点数；不添加任何后缀表示double类型浮点字面量；添加l或L后缀表示long double类型的浮点字面量。另外，如果浮点数的小数部分为0，那么我们也可以写为：10.、-5.等形式，10.相当于10.0。同样，如果整数部分为0，那么我们也可以写作为.25、.1001等形式，.25相当于0.25。

此外，C语言还引入了对浮点数的科学计数法的表示。比如，3e5或3E5表示$3*10^5$；6e-3或6E-3表示$6*10^{-3}$。这里需要注意的是，e或E和它前后两个数之间都不能有空白字符，像3 e 5就不是$3*10^5$的科学计数法表示了。此外，e或E右边的数必须是整数，不能是浮点数。

C语言还引入了十六进制浮点表示法，即一个十六进制数跟p或P，再跟一个十进制数，表示p或P之前的十六进制数乘以$2^{p之后的数}$。比如$0x3P^2$相当于0x3*22；$0x3.5P-3$相当于$0x3.5*2^{-3}$。这里需要注意的是，p或P的左边必须是一个十六进制整数或浮点数，p或P的右边必须是一个十进制整数，并且十六进制浮点数表示中，p与指数部分不可缺省。十六进制浮点数到十进制浮点数的转换可以参考第2章的2.4节内容。这里十六进制浮点数并非采用IEEE754所描述的规格化浮点数表示法，而是一般的二进制浮点表示法，比如0x3.5p0相当于二进制数的整数部分为0011，小数部分为0101（一个十六进制数的位数占4个比特）。其小数部分的计算直接用$2^{-2}+2^{-4}$=0.3125即可，或者更简单地，直接用5除以16即可得到小数部分结果，所以0x3.5就相当于十进制浮点数3.3125。跟十六进制整数计算方式类似，对于像0x1.2345p0这种十六进制浮点数的小数部分（一般术语上又称为尾数部分）的计算方式为：$2/16+3/16^2+4/16^3+5/16^4$。

下面我们将举一些例子给大家操练一下，见代码清单5-12。

代码清单5-12　浮点类型

```
#include <stdio.h>
#include <float.h>
```

```
int main(int argc, const char * argv[])
{
    // 定义一个单精度浮点数对象f
    float f = -3.5E+3f; // f的值为3.5 * 1000 = -3500.0。这里的+表示正号，可省
    printf("f = %f\n", f);

    f = .25f;                   // f的值为0.25
    printf("f = %f\n", f);

    f = -0x5p+10f;              // f的值为-5 * 1024 = -5120
    printf("f = %f\n", f);

    // 定义了一个双精度浮点对象d
    double d = -100.;           // d的值为-100.0
    printf("d = %f\n", d);

    d = -1200e-5;                       // d的值为-1200 * 0.00001 = -0.012
    printf("d = %f\n", d);

    d = 0x30.8p-2;              // d的值为0x30.8 * 0.25 = 48.5 * 0.25 = 12.125
    printf("d = %f\n", d);

    // 定义一个long double类型的对象g
    long double g = -0x3.5P0L;  // g的值为-0x3.5 * 1 = -3.3125
    printf("g = %Lf\n", g);

    g = 0x18.F8P2L * 2E3L;      // (24+0.96875)*4 * 2*1000 = 199750.0
    printf("g = %Lf\n", g);

    printf("long double size is: %zu bytes\n", sizeof(g));

    // 以下都是用浮点数的科学计数法来打印出各种类型浮点数的最大、最小值
    printf("float min value is: %g\n", FLT_MIN);
    printf("float max value is: %g\n", FLT_MAX);

    printf("double min value is: %g\n", DBL_MIN);
    printf("double max value is: %g\n", DBL_MAX);

    printf("long double min value is: %Lg\n", LDBL_MIN);
    printf("long double max value is: %Lg\n", LDBL_MAX);
}
```

5.3 数据精度与类型转换

几乎所有计算机编程语言都会涉及数据精度以及类型转换的问题。一般来说，一个类型所占用的字节个数越多（即宽度越大），其精度也就越高。在C语言中，将整数精度等级称为“整数转换等级”。

C11标准提出以下约定。

1）任意两个不同类型的带符号整数不会具有相同等级，即使它们所表示的值一模一样。比如在32位环境中，一般int类型与long类型在整数数值上表现是完全一样的，都表示32位带符号整数，但long的等级仍然高于int的等级。

2）更高精度的带符号整数类型的转换等级应该高于较低精度的带符号整数类型的等级。

3）一个无符号整数类型的转换等级与其相应的带符号整数类型的等级相同。比如，short类型的转换等级与unsigned short类型的一样。

4）具体的带符号整数的转换等级如下（从高到低）：long long int>long int>int>short int>signed char。

5）char的等级应该与signed char和unsigned char相同。

6）size_t与ptrdiff_t的等级不应该高于signed long int，除非C语言实现

需要支持更大的对象。

就这一点而言，GCC与Clang编译器在64位执行模式下将size_t的实现定义为unsigned long，这是合乎标准的；而MSVC与VS-Clang却将size_t定义为了unsigned long long，这显然没有遵守标准的一般规定。

7）_Bool类型（即布尔类型）的等级在所有标准整数类型中是最低的。

5.3.1 整数晋升

对于一个整数类型，如果它的精度等级在计算过程中从低转换到高，那么这个过程称为“整数晋升”，它是一个隐式转换，无需程序员写投射操作符进行类型转换。在C语言标准中，之所以称为整数晋升是因为低转换等级提升到高转换等级类型，在整数数据上不会有任何变化，而仅仅是类型变为更高等级了。比如，代码清单5-13所示的整数类型转换都属于整数晋升。

代码清单5-13 整数晋升

```
#include <stdio.h>

int main(int argc, const char * argv[])
{
    // 声明了一个signed char类型的对象c
    signed char c = -128;

    // 声明了一个short类型对象s
    short s = c;                          // 这里对象c做整数晋升到short类型，并将值赋给对象s

    // 声明了一个int类型对象i
    int i = s;                    // 这里对象s做整数晋升到int类型，并将值赋给对象i
```

```
    // 声明了一个对象l，并将对象i做整数晋升到long类型，并将值赋给l
    long l = i;

    // 声明了一个对象g，并将对象l做整数晋升到long long类型，并将值赋给g
    long long g = l;

    printf("g = %lld\n", g);    // 输出-128

    // 声明了一个unsigned char类型对象a
    unsigned char a = 255;

    // 声明了一个unsigned int类型对象u
    unsigned int u = a;         // 这里将对象a做整数晋升到unsigned int，并将值赋给u

    printf("u = %u\n", u);      // 输出255
}
```

5.3.2 带符号与无符号整数之间的转换

当一个带符号（无符号）整数类型要转换为另一个无符号（带符号）整数类型时，如果转换目标类型有足够大的精度来容纳原始类型，那么转换后的值是保持不变的。

当一个原始整数类型要转换为无符号整数类型时，如果目标无符号整数类型无法容纳原始整数，那么，原始整数值通过不断地加（或减）目标类型最大能表示的值再加1，直到该值恰好能落在目标无符号整数可表示范围内。比如，一个-129的short类型要转换为unsigned char类型，那么将-129加上unsigned char最大能表示的值255再加1，即-129+（255+1）=127。127正好在unsigned char所表示的范围内，加法停止，127就是最终转换到unsigned char类型的值。当然，这个是正式的数学上的表达方式。在实际应用中，我们无需那么麻烦去计算，直接将超出目标无符号类型的位全都舍去即可。比如，-129的16位二进制数为1111111101111111，如果将它转换为8位无符号类型，那么我们直接将高8位舍去，取其低8位作为目标无符

号8位整数类型即可，所以转换后的结果就是01111111，即0x7F，对应于十进制数127。这是高精度原始整数转为低精度无符号整数的情况。如果是一个低精度的带符号整型转换为更高精度的无符号整数类型，那么需要先判断原始低精度整数的符号位，如果是0（表示非负整数），则用0填充到高精度无符号整数；如果是1（表示负整数），那么用1填充到高精度无符号整数。比如，带符号8位整数-1（二进制数为11111111）将它转为无符号16位整数，结果为1111111111111111，即65535。

当一个原始类型要转为带符号整数类型时，如果目标带符号整数类型无法容纳原始整数，那么结果是由实现定义的，C语言实现也可选择发出异常信号。而现在主流C语言编译器（比如MSVC、GCC和Clang）对目标类型为带符号整数类型的转换与目标为无符号整数类型类似。如果是高精度整型转低精度带符号整型，那么直接做二进制数的高位截断即可。如果是低精度的无符号整型转高精度带符号整型，那么高位直接用0扩充。比如，无符号8位整数255转为带符号16位整数，仍然是255；无符号16位整数1023转为带符号8位整数，即0000001111111111转为带符号8位整数，直接截断高8位，保留低8位，得到11111111，结果为-1。

综上所述，对于当前主流C语言编译器而言，整数类型转换主要看源操作数类型，如果源操作数是无符号整数类型，那么做低精度到高精度的整数类型转换时采用高位填0的方式；如果源操作数是带符号整数类型，那么做低精度到高精度的整数类型转换时采用的是高位填符号位的方式。而对于高精度到低精度的整数转换，无论源操作数是带符号整数还是无符号

整数，都是采用高位截断的方式。下面将给大家举一些例子来更好地帮助大家理解带符号与无符号数之间的转换，如代码清单5-14所示。

代码清单5-14　带符号与无符号数之间的类型转换

```
#include <stdio.h>

int main(int argc, const char * argv[])
{
    short s = -129;

    unsigned char uc = s;               // 这里是将s + 256 = 127赋值给uc

    printf("uc = %u\n", uc);            // 输出：uc = 127

    unsigned short us = 1023;

    uc = us;  // 直接截取us高8位赋值给uc

    printf("uc = %u\n", uc);            // 输出：uc = 0

    uc = 128;
    s = uc; // uc为无符号整数，直接扩充0到short宽度
    printf("s = %d\n", s);              // 输出：s = 128

    signed char sc = -1;
    s = sc; // sc是带符号整数，将符号位（这里是1）扩展到short类型宽度
    printf("s = %d\n", s);              // 输出：s = -1

    sc = us;                            // 直接截断us高8位，取其低8位给sc
    printf("sc = %d\n", sc);            // 输出：sc = -1

    us = uc;    // uc是无符号整数，直接扩充0到unsigned short类型的宽度
    printf("us = %u\n", us);            // 输出：us = 128

    // sc是带符号整数，将符号位（这里是1）扩展到unsigned short类型宽度
    us = sc;

    printf("us = 0x%.4X\n", us);        // 输出：us = 0xffff

    s = 257;
    uc = s;                             // 直接截断s的高8位，取其低8位
    printf("d = %d\n", uc);             // 输出：d = 1
}
```

在有些较老版本的编译器或者把编译器警告等级调得较高的情况下，高转换等级的整数类型转换为低转换等级的整数可能会出现警告，此时我们可以用投射操作符（cast operator）做显式的类型转换来规避这些警告。不过，C11标准提到的是，投射操作符主要用于涉及指针的类型转换，对

于基本数据类型，可用也可不用。

投射操作符非常简单，就是用圆括号将某个类型包围住。比如：（int）、（long long）等。

代码清单5-15描述了投射操作符的使用方法。

代码清单5-15　投射操作符的使用

```
#include <stdio.h>

int main(int argc, const char * argv[])
{
    signed char c = 100;

    // signed char的转换等级比int要低，所以在任何情况下都无需投射操作符
    // 但是加上投射操作符做显式类型转换也没问题
    int i = c;

    // short的转换等级比int低，在某些情况下不用投射操作符可能会有警告
    short s = (short)i;          // 将变量i通过投射操作符强制转为short类型

    s = (short)65536L + (short)65536LL;
    printf("s = %d\n", s);       // 输出0
}
```

投射操作符的优先级仅次于单目操作符，比乘法操作符优先级高。关于投射操作符的详细介绍请见5.6节内容。

5.3.3　浮点数与浮点数的转换以及浮点数与整数之间的转换

浮点数之间的转换与整数之间的转换不同。因为一般处理器架构采用的是IEEE754规格化浮点表示法，所以大多数十进制浮点数无法精确地用

二进制浮点数来表示，这是由于其尾数部分实际上都是通过2^n进行相加拟合而成的。所以，我们在比较两个浮点数的时候必须谨慎使用==相等性操作符。

如果一个处理器架构支持IEEE754标准，那么单精度浮点与双精度浮点的转换直接可根据IEEE754标准中浮点数的表示方式进行。对于单精度浮点数转双精度浮点数，双精度浮点数的符号位以及尾数高位有效数都不需要变动，仅仅是尾数低位添0；而阶码部分则是用原始单精度浮点的指数加上双精度浮点数指定的中经指数偏差即可。而双精度转单精度则可能会产生精度丢失。

C11标准提到，一个有限浮点型实数（即它不是一个非数NaN，也不是一个无穷大数INF）可以转换为除布尔类型之外的其他所有整数类型，然后其小数部分被丢弃，也就是说它的值向零截断。如果一个浮点数转为较低精度的无符号整数（比如一个单精度浮点数转为一个无符号8位整数），那么该浮点数是先转为与它相同宽度的带符号整数（先转signed int），然后再转为相应更低精度的无符号整数（再转unsigned char），还是直接转为与低精度无符号整数相同宽度的带符号整数（先转signed char），然后再转为该相同精度的无符号整数（再转unsigned char），这一点在标准中没有提到，也是由实现定义的。同样，当一个整数转为一个浮点数时，倘若目标浮点数无法容纳原始整数的数值范围，那么结果也是未定义的。

下面我们将举一些例子来描述浮点数与浮点数之间的转换，以及浮点

数与整数之间的相互转换，如代码清单5-16所示。

代码清单5-16　浮点数与整数之间的转换

```
#include <stdio.h>

int main(int argc, const char * argv[])
{
    float f = 3.1415926f;

    double d = f;

    // 输出1，说明两者相等
    printf("d == 3.1415926f? %d\n", d == 3.1415926f);

    // 输出0，说明两者不相等。
    // 因为3.1415926这个字面量在单精度与双精度浮点的表达上不具有相同尾数，
    // 由于双精度具有更高精度，所以其尾数部分不是简单地通过在单精度浮点的基础上添加零所得，
    // 而是继续对该小数做进一步拟合，使得双精度浮点数与所表达的字面量的值更接近。
    // 所以，这里同样都是3.1415926，但多个f和少个f就有千差万别了
    printf("d == 3.1415926? %d\n", d == 3.1415926);

    d = 3.1415926;
    f = d;
    // 输出1，说明两者相等。
    // 由于3.1415926在单精度浮点数的精度范围内，所以精度在转换过程中没有损失
    printf("f = 3.1415926f? %d\n", f == 3.1415926f);

    int i = f;
    printf("i = %d\n", i);// 输出3，说明仅保留浮点数的整数部分，而把小数部分全都截断

    i = 100.5 + 200.5;

    // 输出301，说明这里两个浮点数相加，是将结果先以双精度浮点的形式保存，
    // 最后再转换为int类型
    printf("i = %d\n", i);

    i = (int)100.5 + (int)200.5;

    // 输出300，由于这里是先分别将100.5和200.5转为int类型整数。
    // 所以，在它们相加之前，小数部分已经都被截断，使得结果为100 + 200
    printf("i = %d\n", i);

    d = -100000000000.99999999999;
    unsigned char uc = d;
    printf("uc = %u\n", uc);    // 输出0

    uc = (long long)d;
    printf("uc = %u\n", uc);    // 输出255

    uc = (int)d;
    printf("uc = %u\n", uc);    // 输出0
}
```

代码清单5-16的示例是采用Apple LLVM 8.0编译器，基于Intel Haswell架构处理器，在macOS 10.12.3系统上运行后得出的。

5.4 C语言基本运算操作符

对于我们上面提到的整数类型对象，可以使用加减乘除、求模等算术运算操作符，还有正负号单目操作符，自增、自减操作符，按位与、按位或、按位异或、按位取反这四种按位逻辑运算操作符，以及移位操作符。对于浮点类型对象，可以使用加减乘除算术运算操作符，还有正负号单目操作符，自增、自减操作符。这些操作符的计算优先级可参考4.2.4节中的内容。

5.4.1 加、减、乘、除与求模运算操作符

计算两个整数和浮点数的和、差、积、商、余数（仅整数可用）可以分别通过加、减、乘、除、求模这些操作来实现。这些操作对应的操作符在C语言中分别记为：+、-、*、/、%。这些都属于双目操作符，即每种操作都需要两个操作数（operand）。在C语言中还有对左边操作数与右边操作数进行计算，然后直接把计算结果赋给左操作数的简易操作符，与上述操作符分别对应为：+=、-=、*=、/=、%=。在C语言标准中称之为复合赋值操作符（compound assignment operator）。这里各位要注意的是，在做除法和求模计算的时候，除数（即右操作数）不能为零，否则整个应用可能会导致异常。下面我们举一些简单的例子来看看这些操作符的使用，如代码清单5-17所示。

代码清单5-17　基本算术运算操作符

```
#include <stdio.h>

int main(int argc, const char * argv[])
{
    int a = +100;                       // 这里的+可省
    int b = -a;                         // b = -100
    int c = a + b;                      // a + b的值为0，所以c的值为0

    a *= b;                             // 相当于 a = a * b，a的值为-10000
    c /= a;                             // 相当于 c = c / a，c的值为0
    b %= 3;                             // 相当于 b = b % 3，-100除以3的余数为-1
    printf("a + c = %d\n", a + b + c);  // 输出-10001
    b += 1 + 2 + 3;                     // 这里相当于：b = b + (1 + 2 + 3)

    float x = 10.25f;
    float y = -0.25f;

    x += y; // 相当于x = x + y，x的值为10.0f
    y -= x; // 相当于y = y - x，y的值为-0.25f - 10.0f = -10.25f
    printf("y / x = %f\n", y / x);      // 输出-1.025000
}
```

5.4.2　按位逻辑操作符

按位逻辑操作符可对任何整数类型进行操作，包括布尔类型，但不包括浮点数类型。按位与、按位或、按位异或以及按位取反，在C语言中的操作符分别对应为：&、|、^、~。同样，按位与、按位或和按位异或，都有直接赋值的操作表达方式，分别为：&=、|=、^=。

下面举几个例子来介绍按位逻辑操作符的基本用法，并且给出一些需要注意的细节，如代码清单5-18所示。

代码清单5-18　基本按位逻辑操作符

```
#include <stdio.h>

int main(int argc, const char * argv[])
{
```

```
    bool b = false;
    b |= true;                  // 相当于b = b | 1
    printf("b = %d\n", b);      // 输出1

    // 各位注意，~b的结果仍然是1!
    // 之前已经提到，在C语言中任何具有返回类型的表达式都可充当一个布尔表达式，
    // 然后对于上面的布尔变量b，其内部二进制表示是00000001（在Clang中它占用1个字节）。
    // 所以，这里对b采用按位取反操作后，内部结果应该变为11111110，十六进制数为0xfe。
    // 最后，0xfe与0比较，当然不相等，所以结果为true
    b = ~b;                     // 由于按位取反只需要一个操作数，所以它没有~=这种形式
    printf("b = %d\n", b);      // 输出1

    unsigned long long x = 0xffffffffffffffffULL;

    // 0原本是int类型，但&=的左操作数是一个unsigned long long类型，
    // 转换等级比int要高，所以这里的0会先被隐式地做整数晋升，
    // 把它提升到unsigned long long的宽度，然后再对它做按位取反操作
    x &= ~0;    // 这里相当于x = x & ~0。操作后，x的值仍然为0xffffffffffffffff

    int a = 0;
    x &= ~a;    // 这里与上述情况一样
    printf("x = 0x%.16llX\n", x);  // 输出0xffffffffffffffff
}
```

5.4.3 自增、自减操作符

自增、自减操作符可用于整数类型与浮点数类型的变量，也能用于指针变量。不过它们只能作用于可被修改的左值（关于左值的概念请参考14.4节）。自增、自减操作符有两种形式，一种是作为前缀操作符，另一种是作为后缀操作符。作为前缀操作符时，其计算优先级作为单目操作符的优先级，所以其作为后缀操作符的计算优先级比作为前缀操作符的优先级要高一级。当自增作为前缀操作符使用时，它的计算过程就相当于（操作数+=1）。前缀++操作符的结果就是操作数执行递增之后的值。比如：int b=++a；就好比int b；a+=1；b=a；，或int b=a+1；a+=1；前缀自减操作符也同样如此。当自增作为后缀操作符使用时，后缀++操作符的结果是操作数执行++之前的值。然后，操作数对象的值将作为副作用在原有基础上递增。对后缀++操作结果的值计算，其顺序将在更新操作数本身值的副作

用之前。比如：int b=a++；就好比int b=a；a+=1；后缀--操作符也同样如此。

下面举一些简单易懂的例子，如代码清单5-19所示。

代码清单5-19　自增自减操作符

```
#include <stdio.h>

int main(int argc, const char * argv[])
{
    int a = 1;
    int b = ++a;                // 相当于int b = a += 1;。b的值为原来a的值加1的结果
    int c = a++ + ++b;  // c的值为a + (b + 1)，然后a与b各自递增

    // 输出：a = 3, b = 3, c = 5
    printf("a = %d, b = %d, c = %d\n", a, b, c);

    float f = 10.5f;
    float g = f++;      // g的值为原来f的值，然后f递增

    // 输出：f = 11.500000, g = 10.500000
    printf("f = %f, g = %f\n", f, g);
}
```

在使用递增、递减的时候往往会牵涉程序执行顺序问题，关于此问题读者可参考14.5节。

5.4.4　关系操作符、相等性操作符与逻辑操作符

在C语言中，关系操作符（relational operators）的表示基本与数学上的表示一致——用>表示大于关系，用<表示小于关系，用>=表示大于等于关系，用<=表示小于等于关系。相等性操作符（equality operators）就两个：一个是==表示相等；另一个是！=表示不等。用这些操作符计算得到的结果是一个int类型。如果这些操作符的左操作数满足右操作数的指定关系，

那么返回1，否则返回0。在5.1.5节中已经提到了，由于C语言一开始并没有引入“布尔类型”这个概念，而仅仅是通过与0比较来得出真假值。所以就当前的C11标准而言，这些操作符所返回的类型仍然被定义为int类型，而不是_Bool类型，这是为了能与之前C90标准的C代码进行兼容。但是对现代化C语言编程而言，我们应该勤用布尔类型，这不仅有助于对编程语言相关概念的理解，而且在代码上也更具可读性与逻辑性！同时，当我们要转到Java、Swift等比C语言类型更强的编程语言上时也能更顺手些。所以，笔者这里强烈推荐各位将关系操作符以及相等性操作符的结果类型视为bool类型（需要引入<stdbool.h>标准库头文件），然后将满足指定关系的值看作为“真”（true），将不满足关系的值看作为“假”（false）。

由于关系操作符与相等性操作符比较简单，而且与我们日常用的数学上的操作很接近，所以不过多介绍。本节主要为大家介绍一下逻辑操作符。C语言中有3种逻辑操作符：逻辑与&&表示并且；逻辑或||表示或者；逻辑非！表示否。C语言标准规定，逻辑操作符的操作数都应该是int类型的表达式，计算结果也是int类型。但这里也推荐各位将逻辑操作符的操作数以及计算结果视为布尔类型。下面我们分别介绍这3种逻辑操作符。

①逻辑与是一个双目操作符，其计算过程是先判定左操作数的值是否为真，如果为真则继续判定右操作数，如果左右两个操作数都为真，那么计算结果为真，否则计算结果为假。但如果左操作数表达式的值为假，那么直接得到为假的结果，而不会再去计算右操作数。

②逻辑或也是一个双目操作符，其计算过程是先判定左操作数的值是

否为真，如果为真则直接得到为真的结果，而不会再去计算右操作数。如果左操作数表达式的值为假，那么再去判定右操作数，如果右操作数的值为真，那么结果为真，如果其值为假，那么结果为假。

③逻辑非是一个单目操作符，如果其操作数表达式为真，那么结果为假；否则结果为真。这个操作其实就是对操作数取逻辑非，也就相当于我们日常逻辑中对某一陈述的否定。

下面我们看一下代码清单5-20所举的一些例子。

代码清单5-20　关系操作符与逻辑操作符

```
#include <stdio.h>
#include <stdbool.h>

int main(void)
{
    int x = 0, y = 0;

    // 这里由于x < 0的结果为假，所以直接得到b为假。
    // ++y > 0这一表达式不会被计算
    bool b = x < 0 && ++y > 0;
    // 输出：b = 0, y = 0
    printf("b = %d, y = %d\n", b, y);

    // 这里由于y == 0的结果为真，所以直接得到b为真。
    // --x > 0这一表达式不会被计算
    b = y == 0 || --x > 0;
    // 输出：b = 1, x = 0
    printf("b = %d, x = %d\n", b, x);

    // 输出：!b = 0
    printf("!b = %d\n", !b);
}
```

5.4.5　移位操作符

之前在2.9节为大家介绍了移位操作。在C语言标准中将移位操作也称

为“按位移位操作”，并且没有明确指定移位操作用的是算术移位还是逻辑移位。C语言中使用<<操作符表示左移操作，使用>>操作符表示右移操作，它们都是双目操作符，并且其左右操作数都必须是整数类型。a<<b表示对整数a向左移b位；a>>b表示对整数a向右移b位。C语言标准中明确声明了，如果移位操作的右操作数为负数，或者右操作数的值大于等于左操作数值的位宽（即由多少个比特构成），那么行为是未定义的。比如，在一般32位系统环境下，1<<-1、2>>32这些移位操作的结果都是未定义的。而当前主流C语言编译器的移位行为都直接与当前所运行的处理器的移位特性相关。也就是说，处理器如何处理移位的右操作数为负数，或者右操作数的数值大于等于左操作数位宽的情况，那么计算结果就如此被处理执行。关于这点，各位可以回顾一下2.9节中的相关内容。当然，对于未定义行为的情况我们还是需要慎重对待，应当尽量避免移位的右操作数为负数的情况，以及右操作数的数值大于等于左操作数位宽的情况。

不过当前主流编译器以及大部分嵌入式系统相关的C语言编译器仍然区分了逻辑右移与算术右移。对于右移操作，采用的是算术右移还是逻辑右移主要看右移操作的左操作数，即移位操作对象。如果左操作数是带符号整数类型，那么采用的是算术右移；如果是无符号整数类型，那么采用的是逻辑右移。我们看一下代码清单5-21的例子。

代码清单5-21　移位操作

```
#include <stdio.h>

int main(void)
{
```

```
    int a = 1;
    unsigned b = 1;

    // 相当于a = a << 2，将带符号整数对象a向左移2位，结果为4
    a <<= 2;

    // 相当于b = b << 3，将无符号整数对象b向左移3位，结果为8
    b <<= 3;

    printf("a = %d, b = %d\n", a, b);

    // 当前a的值用十六进制表示为：0xfffffff6
    a = -10;
    b = 0xfffffff6;

    // 相当于a = a >> 30，由于a是带符号整数，所以这里做的是算术右移
    a >>= 30;

    // 相当于b = b >> 30，由于b是无符号整数，所以这里做的是逻辑右移
    b >>= 30;

    // 输出：a = -1, b = 3
    printf("a = %d, b = %d\n", a, b);
}
```

我们从代码清单5-21可以看到，带符号整数a在做右移的时候，高位填充的是符号位，所以采用的是算术右移，结果为-1。而无符号整数b在做右移的时候，高位填充的是0，所以采用的是逻辑右移，结果为3。

5.4.6 圆括号操作符

圆括号操作符主要用于提升其操作数表达式的计算优先级，并且能将其操作数与其他操作符进行分隔。带有圆括号操作符的表达式称为圆括号表达式（Parenthesized Expression），它属于基本表达式，这也是计算优先级最高的表达式。圆括号操作符有一个很厉害的特性是，其操作数表达式可以是一个左值，然后将得到的结果表达式也作为一个左值，并可对它进行修改。关于左值的概念可参见14.4节。我们下面举一些简单的例子，介绍一下圆括号操作符（见代码清单5-22）。由于它使用起来非常自然，一

般来说与我们在数学上用于计算式中的效果类似，所以不做过多描述。

代码清单5-22　圆括号操作符

```
#include <stdio.h>

int main(void)
{
    int a = 10;

    // 这里使用圆括号操作符提升a + 1子表达式的优先级
    a = (a + 1) * 2;

    // 这里对对象a使用圆括号操作符尽管没有实际意义，但也完全合法。
    // 这里相当于：int b = a;
    int b = (a);

    // 这里输出：a = 22, b = 22
    printf("a = %d, b = %d\n", a, b);

    int *p = (int[]) { 1, 2 };

    // 这里p + 1子表达式不是左值，但*(p + 1)子表达式却是一个左值，
    // 然后(*(p + 1))子表达式也是一个左值，所以可以对它做自增操作
    (*(p + 1))++;

    // 输出：p[1] = 3
    printf("p[1] = %d\n", p[1]);
}
```

5.5　sizeof操作符

sizeof属于单目操作符，其语法形式为：

1）sizeof单目表达式

2）sizeof（类型名）

sizeof操作符返回其操作数的大小（即占用多少字节），用一个整数值来表示，一般C语言实现都用size_t来表示sizeof操作符的返回类型。如果sizeof的操作数是一个可变修改类型（将在7.3节中介绍）的表达式，那么该操作数需要在运行时计算；否则sizeof仅仅在编译时对操作数的类型进行获取，而不会去计算操作数所对应的整个表达式的结果。也就是说C语言实现在这种情况下只需要生成对应sizeof操作符的结果相关代码，而无需生成作为sizeof操作数的表达式的代码。

下面举一些例子来帮助各位理解，如代码清单5-23所示。

代码清单5-23　sizeof操作符

```
#include <stdio.h>

int main(int argc, const char * argv[])
{
    int a = 10;

    // 这里假定argc为1，array是一个变长数组
    // 其长度需要在运行时通过变量argc与a相加获得
    int array[a + argc];

    // 这里将会对array的大小在运行时做计算
    size_t size = sizeof(array);
```

```
    printf("size = %zu\n", size);                // 输出44

    size = sizeof(++a);

    printf("size = %zu, a = %d\n", size, a);     // a仍然为10，++a没有被执行

    double d = 10.05;

    size = sizeof d;    // sizeof操作符可以不加括号，但前提是操作数是单目表达式
    printf("sizeof d = %zu\n", size);            // 输出8

    size = sizeof d + sizeof a;
    printf("size = %zu\n", size);                // 输出12
}
```

在代码清单5-23中涉及的数组概念，我们将在第7章重点描述。此外，尽管C语言允许我们可以使用不带圆括号的sizeof操作符，但笔者这里建议各位加上圆括号，这样会使得代码更为直观，也不会受到其他中缀表达式的干扰。本书后续章节中所涉及的sizeof操作符都将带有圆括号。

5.6　投射操作符

投射操作符（Cast Operator）的语法很简单，就是在圆括号中放上类型名，用于修饰一个表达式。它与圆括号操作符的区别在于，圆括号操作符中是将表达式作为其操作数；而投射操作符则是在圆括号中放类型名。投射操作符是一个单目操作符，这里被修饰的表达式作为投射操作的操作数。投射操作的实际含义是：将一个表达式的类型投射为该投射操作符所指定的类型，这个动作也被称为类型投射（Type Casting）。C语言标准为何用“投射”这个词，而不是直接用“转换”（type conversion）呢？因为这里面其实牵涉两个动作：首先根据当前表达式的内容拷贝出一个临时对象，然后将该临时对象做目标类型的转换以及值的调整。因此整个投射操作过程其实是隐式地创建了一个临时对象，然后我们后续的操作都是针对该临时对象进行的，与原始表达式无关。这就类似湖周围的树通过阳光投射到湖面中的倒影。树就类似原始表达式，倒影就好比临时对象，它们属于两个不同的对象。

我们了解了这个概念之后就能对为何投射表达式不能作为左值（我们将在14.4节中介绍）做出合理的解释了。一般C语言教科书中把投射操作称为“类型转换”是不严谨的。我们可以先简单看一下代码清单5-24的例子。

代码清单5-24　投射表达式不能作为左值

```
#include <stdio.h>
int main(int argc, const char * argv[])
```

```
{
    int i = 100;
    ((short)i)++;                          // 这句是非法的，对投射表达式不能做自修改操作
    short *sp = &(short)i;  // 这句也是非法的，对非左值不能取其地址
    (short)i += 10;              // 对左值进行投射操作一般也是非法的
    printf("*sp = %d\n", *sp);
}
```

代码清单5-22展示了对一个投射表达式做赋值计算是非法的，取其地址也是非法的。而且C语言标准也明确指出，一个投射表达式将不会产生一个左值。尽管C语言标准中没有指明投射操作的操作数是否可以是一个左值，但在主流C语言编译器中这是非法的。

然而有趣的是，在Visual Studio中的MSVC编译器上，代码清单5-22的代码能正常通过编译运行。因此，MSVC编译器对投射操作放宽了限制，使得它更像是单纯的“类型转换”，仿佛丢弃了投射操作。不过C语言标准中明确规定：一个投射操作不产生左值，而对非左值做自增、取地址等操作本身就是非法的，所以这也是为何笔者不赞成各位使用Visual Studio自带的MSVC编译器来写C语言的主要原因，它对标准的支持不仅有限，而且还违背了一些基本原则，因此笔者更推荐各位使用VS-Clang编译器。

最后，我们再简单讲一下C11标准对投射操作符的约束。

1）用于投射操作的类型名应该指定一个标量类型，即非数组类型，并且可以对该类型添加_Atomic、const、volatile等限定符；用于投射操作的类型名也可以指定void类型（详细请见7.9节内容）。而投射操作的操作数应该具有标量类型。

2）涉及指针类型的转换，除了某些允许指针类型隐式转换的情况，应

该显式使用投射操作。

3）指针类型不应该被转换为任一浮点类型；浮点类型也不应该被转换为任一指针类型。

比如int*不能转换为float类型；double类型也不能转换为double*类型。

5.7 本章小结

本章介绍了C语言中常用的基本类型，包括整数类型、字符类型以及浮点数类型，同时介绍了常用的基本算术逻辑计算、自增自减操作、按位操作、逻辑操作等。最后还讲了我们常用的sizeof操作符以及投射操作符。

这里各位要注意的是整数类型及其相关表示的范围是根据特定处理器以及特定操作系统环境来决定的。当然，本章中也列举了常用的32位系统以及64位系统中各种整数类型所表示的数值范围。其次，自增自减操作的操作顺序需要各位好好理解，这需要多实践，当然这里也不建议各位把自增自减操作写得过于复杂，否则会影响代码的可理解性。此外，笔者还推荐各位将关系操作符、相等性操作符以及逻辑操作符的结果类型视作布尔类型，这样能提升代码的逻辑性与可读性。最后，本书也澄清了类型投射的问题，它不单单是一种类型转换，然后描述了投射过程概念上的逻辑。

第6章　用户自定义类型

本章我们将介绍C语言中的用户自定义类型。通过用户自定义类型，我们可以把一些数据高度抽象化，从而能够把这些数据信息组织成为我们概念上更易于理解的模型。例如，我们可以定义一个用于学籍管理的“学生信息”类型，或是定义一个可用于表示交通灯：红灯、黄灯、绿灯这三种状态的数据类型，等等。本章还会介绍C11标准对复数（Complex Number）类型的支持。

6.1　枚举类型

枚举（enumeration）类型一般用于表示离散有限有个数值的数据对象。比如刚才提到的交通灯的3种颜色的灯，阳光可被分离的7种颜色的光，一颗骰子上的6种点的花色，等等。声明一个枚举的一般形式如下：

```
enum  标识符 { 枚举符列表 }
```

以上这种声明枚举的形式可以完整地称为枚举说明符（enum-specifier）。其中，枚举标识符在C语言标准中又被称为枚举标签（tag）。枚举符列表由一个个枚举符（enumerator）构成，而一个枚举符则是一个枚举常量（enumeration constant），或带有一个常量表达式的枚举常量。下面我们先举一些简单例子，请看代码清单6-1：

代码清单6-1　枚举类型介绍

```
// 声明一个名为LIGHT的枚举类型
enum LIGHT;

// 定义了一个名为TRAFFIC_LIGHT的枚举类型
enum TRAFFIC_LIGHT
{
    TRAFFIC_LIGHT_RED,      // 0
    TRAFFIC_LIGHT_YELLOW,   // 1
    TRAFFIC_LIGHT_GREEN     // 2
};

// 定义了一个名为LIGHT的枚举类型
enum LIGHT
{
    LIGHT_RED = -2,                     // -2
    LIGHT_ORANGE,                       // -1
    LIGHT_YELLOW = 1,                   // 1
    LIGHT_GREEN,                        // 2
    LIGHT_BLUE,                         // 3
    LIGHT_INDIGO = LIGHT_RED + -100,    // -102
    LIGHT_PURPLE                        // -101
    ,    // 一个枚举符列表的末尾可加一个逗号，这便于用程序自动生成C语言代码
```

```
};
```

在代码清单6-1中，TRAFFIC_LIGHT和LIGHT都是枚举标签，我们一般就称为枚举标识符或枚举名。像TRAFFIC_LIGHT_RED以及LIGHT_RED=-2等都是枚举符。而像TRAFFIC_LIGHT_RED以及LIGHT_RED等都属于枚举常量。枚举符列表中的每一个枚举常量都被声明为具有int类型的常量。如果枚举符常量带有一个常量表达式，那么该表达式的类型必须与int兼容，同时，该常量表达式的值将作为该枚举常量的值（比如LIGHT_RED）。如果第一个枚举常量不带有常量表达式，那么其值默认为0（比如TRAFFIC_LIGHT_RED）。对于之后的所有枚举常量，如果它没有常量表达式为它赋值，那么该枚举常量的值为它前一个枚举常量的值加1（比如LIGHT_ORANGE）。一个枚举中的枚举符也被称为该枚举的成员。比如，TRAFFIC_LIGHT枚举类型就有三个枚举成员，分别是TRAFFIC_LIGHT_RED、TRAFFIC_LIGHT_YELLOW和TRAFFIC_LIGHT_GREEN。

像代码清单6-1中所声明的“enum LIGHT；”以及“enum TRAFFIC_LIGHT{TRAFFIC_LIGHT_RED，TRAFFIC_LIGHT_YELLOW，TRAFFIC_LIGHT_GREEN}；”这整段代码就称为枚举说明符。

每个枚举类型应该与char、一个带符号整数类型或一个无符号整数类型相兼容。具体对于枚举类型使用哪种整数类型是由实现定义的。比如，在ADS1.2开发环境中，枚举类型被默认定义为与unsigned char类型兼容，

而在当前主流桌面C语言编译器中，枚举类型默认为与int类型相兼容。注意，这里说的是枚举类型，而不是枚举常量的类型。

要声明一个枚举类型的对象，需要关键字enum加枚举标识符一起来声明。代码清单6-2阐明了如何定义、使用枚举对象：

代码清单6-2　声明及使用枚举对象

```
#include <stdio.h>
#include <stdint.h>
// 定义了一个名为TRAFFIC_LIGHT的枚举类型
enum TRAFFIC_LIGHT
{
    TRAFFIC_LIGHT_RED,      // 0
    TRAFFIC_LIGHT_YELLOW,   // 1
    TRAFFIC_LIGHT_GREEN     // 2
} g_traffic;  // 定义了一个全局的enum TRAFFIC_LIGHT变量g_traffic
// 定义了一个名为LIGHT的枚举类型
static enum LIGHT
{
    LIGHT_RED = -2,                     // -2
    LIGHT_ORANGE,                       // -1
    LIGHT_YELLOW = 1,                   // 1
    LIGHT_GREEN,                        // 2
    LIGHT_BLUE,                         // 3
    LIGHT_INDIGO = LIGHT_RED + -100,    // -102
    LIGHT_PURPLE                        // -101
    // 定义了一个静态的enum LIGHT变量s_light
    // 并用LIGHT_BLUE枚举常量对其初始化
} s_light = LIGHT_BLUE;
// 匿名枚举
enum
{
    DICE_ONE = 1,
    DICE_TWO,
    DICE_THREE,
    DICE_FOUR,
    DICE_FIVE,
    DICE_SIX
   // 定义了一个匿名枚举类型，并用它定义了一个全局变量g_dice
} g_dice = DICE_TWO;

int main(int argc, const char * argv[])
{
    // 定义了enum TRAFFIC_LIGHT的变量traffic，这里的enum不能省，
    // 并对它用TRAFFIC_LIGHT_YELLOW枚举常量对其初始化
    enum TRAFFIC_LIGHT traffic = TRAFFIC_LIGHT_YELLOW;
    g_traffic = TRAFFIC_LIGHT_GREEN;
    printf("traffic = %d, g_traffic = %d\n", traffic, g_traffic);
    // 枚举变量也可以进行算术运算，尽管并不推荐这么做
    g_dice += 2;
    // 一个枚举变量也能赋值给一个整数变量
    int32_t a = g_dice;
    // 一个整数变量也能赋值给一个枚举变量，尽管并不推荐这么做。
    // 此时light变量不是任一其有效的枚举常量值
    enum LIGHT light = a;
```

```
    printf("light is: %d\n", light);
    a += DICE_ONE;
    printf("Is a == DICE_FIVE? %d\n", a == DICE_FIVE);
    printf("light = %d\n", s_light + LIGHT_PURPLE);
    // 在函数内定义一个枚举类型
    enum SOME_ENUM
    {
        SOME_ENUM1,
        SOME_ENUM2,
        SOME_ENUM3
    }se = SOME_ENUM1;
    // 将(se + 2)的结果类型显式转换为enum SOME_ENUM类型
    enum SOME_ENUM se2 = (enum SOME_ENUM)(se + 2);
    printf("se2 = %d\n", se2);
}
```

6.2　结构体类型

C语言中的结构体（structure）是对一组数据元素的有序组织。通过结构体，我们可以建立丰富多彩的数据结构，对现实世界建立概念模型。在结构体中我们可以存放任意类型的数据元素，包括整数类型、浮点类型、枚举类型，甚至嵌套其他结构体类型。C语言标准又将数组和结构体称为聚合（aggregate）类型。要定义一个结构体类型，我们使用关键字struct，后面紧跟的标识符就作为该结构体的类型名，该类型名在C语言标准中又称为此结构体的标签（tag）。结构体类型既可以定义在文件作用域，也可以定义在函数中。

6.2.1　结构体概述

结构体类型的一般定义形式为：

```
struct 标识符可省 { 结构体声明列表 }
```

结构体声明列表由各个声明（包括静态断言声明）构成。静态断言将在14.3节中详细介绍。

代码清单6-3展示了结构体类型的声明以及定义。

代码清单6-3　结构体声明符

```c
#include <stdio.h>
#include <stdint.h>
// 声明了一个名为StructTest的结构体
struct StructTest;

enum MyEnum
{
    MY_ENUM1,
    MY_ENUM2
};

// 定义了一个名为MyStruct的结构体
struct MyStruct
{
    // 声明了MyStruct的一个int32_t类型成员a
    int32_t a;

    // 声明了一个MyStruct的一个enum MyEnum类型成员e
    enum MyEnum e;

    // 声明了MyStruct的一个double类型成员d
    double d;

    // 声明了指向struct StructTest类型的指针成员pTest
    struct StructTest *pTest;

    // 用一个静态断言作为声明，但不占用此结构体对象的存储空间
    _Static_assert(MY_ENUM1 == 0, "NG");
};

// 定义结构体StructTest
struct StructTest
{
    // 声明了StructTest的一个int16_t类型成员s
    int16_t s;

    // 声明了StructTest的一个struct MyStruct类型成员m
    struct MyStruct m;

    // 声明了StructTest中的一个匿名结构体
    struct {
        int32_t a, b;
        float f;
        ;    // 结构体声明列表末尾允许包含一个分号，便于程序自动生成C语言代码
    }i; // 直接用此匿名结构体声明了对象i作为StructTest的成员
};
```

代码清单6-3中，StructTest以及MyStruct就是结构体标签，我们一般就称为结构体标识符或结构体名。像“struct StructTest；”这种对StructTest结构体的声明以及后面对StructTest的整个定义（从{到}；）就称为结构体说明符。而像struct MyStruct里{}中的“int a；”则作为结构体中的声明，其中a就作为MyStruct结构体的成

员。“_Static_assert（MY_ENUM1==0，“NG”）；”这条语句就是静态断

言，如果各位把这条语句中的0改为1的话，在编译时就会报错，报错信息为“NG”。

在结构体内除了可定义基本类型对象作为其成员之外，还可定义枚举、结构体等用户自定义数据类型对象。比如，在MyStruct中的成员e，以及在StructTest中的成员m、i等。

C语言中可以定义匿名结构体类型，即结构体的标识符可缺省。我们在代码清单6-1中最后几行能看到在MyStruct内定义了一个匿名结构体类型，并用它声明了对象i。匿名结构体通常无法被直接引用，所以我们只能通过直接在它后面声明该类型的对象或指向该结构体类型的指针进行使用。

6.2.2 用结构体创建对象并访问其成员

上面我们讲述了结构体类型的声明和定义。下面我们将在代码清单6-4中描述如何用结构体声明对象并对它们进行初始化，以及如何访问结构体的成员。

代码清单6-4 结构体对象的声明、初始化以及成员访问

```
#include <stdio.h>

// 声明了一个名为MyPoint的结构体
static struct MyPoint
{
    float x, y;
}g_point;   // 然后立即声明g_point对象，其成员在程序加载完之后被初始化为零

int main(int argc, const char * argv[])
```

```
{
    // 在main函数中声明了对象point，但没有对它做显式的初始化
    struct MyPoint point;   // 此时，point的x和y成员对象的值都是不确定的

    // 访问一个结构体对象中的某个成员使用“.”访问操作符
    printf("g_point.x = %f, g_point.y = %f\n", g_point.x, g_point.y);

    // 结构体对象标识符、. 访问操作符，以及成员对象标识符之间都可以用空白符分隔。但习惯上，
    // 我们更偏向于直接用“结构体对象标识符.结构体成员对象”这种样式，中间都不包含任何空白符
    printf("point.x = %f, point.y = %f\n", point .x, point . y);

    // 声明一个指向MyPoint结构体类型的指针，并将它初始化为指向point对象的地址
    struct MyPoint *pPoint = &point;

    // 访问一个指向结构体指针的对象，使用“->”操作符
    pPoint->x = 100.0f;     // 此时，point.x的值也变为了100.0f
    printf("point.x = %f\n", point.x);

    // 用MyPoint结构体声明了point2对象，并对它显式初始化
    // 对一个结构体对象显式初始化使用一个大括号，然后依次指明该结构体对应成员的值
    // 这里，point2.x被初始化为100.0f，point2.y被初始化为50.0f
    struct MyPoint point2 = { 100.0f, 50.0f };
    printf("point2.x = %f, point2.y = %f\n", point2.x, point2.y);

    // 结构体对象允许直接用等号赋值，此时C语言实现会直接将此赋值操作拆分为
    // 依次用“=”右操作数的结构体成员对象的值赋值给=左操作数结构体成员对象的值，
    // 不过各个成员对象的赋值执行次序是由实现定义的，
    // 未必严格按照成员对象在结构体中定义的顺序进行
    point = point2; // 此表达式相当于point.x = point2.x; point.y = point2.y;
    printf("point.x = %f, point.y = %f\n", point.x, point.y);

    // 在main函数中定义名为MyRectangle的结构体
    struct MyRectangle
    {
        enum {
            COLOR_RED,
            COLOR_GREEN,
            COLOR_BLUE
            // 这里定义了一个匿名枚举类型，并直接用它声明枚举对象color
            // 作为MyRectangle的一个成员
        }color;

        struct MyPoint position;
        struct {
            float width, height;
            // 这里定义了一个匿名结构体，并直接用它声明结构体对象size
            // 作为MyRectangle的一个成员
        }size;

        // 这里直接用MyRectangle结构体类型声明了一个对象rect，并对它直接做初始化。
        // 在初始化过程中依次对其成员color、position以及匿名结构体对象size进行初始化
    } rect = { COLOR_RED, 10.0f, 20.0f, 90.0f, 35.0f },

    // 然后再直接用MyRectangle结构体类型声明了一个对象rect2，并直接对它进行初始化。
    // 这里在对position以及size成员对象初始化时使用了{ }，因为它们也是结构体类型，
    // 从而可以对部分成员进行初始化为指定的值。
    // 这里，仅对成员position对象中的x成员进行初始化为30.0f；
    // 而对其成员size对象中的所有成员直接出初始化为零
    rect2 = { COLOR_GREEN, { 30.0f }, { } };

    printf("rect.color = %d, rect.position.x = %f, \
           rect.position.y = %f, rect.size.width = %f, \
           rect.size.height = %f\n", rect.color, rect.position.x,
           rect.position.y, rect.size.width, rect.size.height);

    printf("rect2.color = %d, rect2.position.x = %f, \
           rect2.position.y = %f, rect2.size.width = %f, \
           rect2.size.height = %f\n", rect2.color, rect2.position.x,
           rect2.position.y, rect2.size.width, rect2.size.height);
```

```
    // 用MyRectangle结构体声明了对象rect3，并利用指定成员的初始化器对它进行初始化。
    // 这里，rect3.color被初始化为COLOR_BLUE；rect3.position.x初始化为0.0f；
    // rect3.position.y被初始化为200.0f；rect3.size.width被初始化为0.0f；
    // rect3.size.height被初始化为50.0f
    struct MyRectangle rect3 = { .color = COLOR_BLUE,
        .position.y = 200.0f, { .height = 50.0f } };

    printf("rect3.color = %d, rect3.position.x = %f, "
           "rect3.position.y = %f, rect3.size.width = %f, "
           "rect3.size.height = %f\n", rect3.color, rect3.position.x,
           rect3.position.y, rect3.size.width, rect3.size.height);
}
```

如果一个结构体类型声明的对象具有静态存储周期（将在11.3节中介绍）或线程存储周期（将在11.7节中介绍）——比如代码清单6-4中在文件作用域声明的对象g_point，那么该结构体对象的所有成员在执行main函数前被系统默认初始化为0。如果一个结构体对象没有被初始化且具有自动存储周期（将在11.4节介绍），那么其成员对象的值是未定义的，比如代码清单6-4中main函数里的第一行语句。我们在对一个结构体对象初始化时使用{}作为其初始化器（initializer）。在{}中用逗号分隔的表达式为当前结构体对象的相应成员的初始化器，可依次对结构体对象相应的成员进行初始化，它们也称为结构体对象的初始化器列表（initializer-list）。比如代码清单6-4中的“struct MyPoint point2={100.0f，50.0f}；”。其中，{}里的100.0f与50.0f就是对point2对象成员的初始化器，分别将point2对象的成员x与成员y初始化为100.0f和50.0f，它们构成了一个针对结构体对象point2的初始化器列表。而整个{100.0f，50.0f}就是对结构体对象point2的初始化器。如果一个初始化中的初始化器少于当前结构体成员的个数，那么未被初始化到的成员对象被清零（也就是说，如果未被初始化到的成员是数值对象，那么被初始化为0，如果是指针类型对象则被置为空）。如果一个结构体中还含有一个结构体类型的成员，那么对它初始化时，其初始化器可

用{}来分隔其他成员。比如上述代码中的rect2={200，{30.0f}，{}}。另外，由于在对rect2进行初始化的时候，其匿名结构体类型的成员对象size没有任何初始化器对size的成员进行初始化，所以size的所有成员都为0。

要访问一个结构体对象的成员，使用“.”操作符，这在C语言标准中称为成员访问操作符（member-access operator），它属于后缀操作符。各位需要注意的是，成员访问操作符是一个单目操作符，其操作数是该操作符左边表示结构体对象的表达式，而其右边的结构体成员名则不属于操作数。这就类似于投射操作中（）操作符中的类型也不作为投射操作的操作数一样，投射操作符的操作数是它后面的表达式。比如代码清单6-4中的“printf（"point.x=%f\n"，point.x）；”就是读point对象中x成员的值。如果要访问一个指向结构体类型的指针对象，那么使用->操作符（注意，->必须作为整体，即“-”与“>”之间不能存在任何空白符）它和操作符一样，也属于单目后缀操作符，我们将在7.6节做详细介绍。比如代码清单6-4中pPoint->x=100.0f；就是将100.0f赋值给pPoint指针对象所指结构体对象（即point对象）的成员x。我们将在第7章详细介绍指针对象。此外，各位要注意的是，“.”和“->”访问操作符只能访问结构体中的对象成员，而不能访问该结构体中所定义的类型。

最后，上述代码呈现了如何使用指定成员的初始化器（designated initializer）来对结构体成员进行初始化。这个语法特性从C99开始引入，因此当前MSVC、VS-Clang、GCC和Clang都能支持。指定成员的初始化器通过“.”操作符来指定当前对哪个结构体成员进行初始化。而且一旦用了指定

成员的初始化器，对成员的初始化也无需按照它们在结构体中的排列次序进行。比如，我们可以用“struct MyPoint point={.y=50.0f，.x=10.0f}；”对point对象进行初始化。其中，“.y=50.0f”以及“.x=10.0f”均为指定成员的初始化器。这样一来初始化完之后，point.x就是10.0f，point.y为50.0f。

6.2.3 结构体复合字面量

从C99标准开始对结构体还引入了结构体复合字面量（structural compound literal），也称之为“匿名结构体对象”，可以对某个结构体对象直接用结构体字面量进行赋值。这样对于无法在初始化阶段确定值的结构体对象而言，有了一种新的更简便的赋值方式，而无需通过一个临时结构体对象，如代码清单6-5所示。

代码清单6-5 结构体复合字面量

```
#include <stdio.h>
#include <stdint.h>

int main(int argc, const char * argv[])
{
    // 定义了一个结构体S，并用此结构体类型声明了一个变量s
    struct S
    {
        int32_t a, b;
    }s;     // s在这里尚未初始化

    // 在C99之前，我们只能这么对s赋值
    s.a = 100;
    s.b = -50;

    // 或者是：
    struct S tmp = { 100, -50 };
    s = tmp;

    // 从C99标准开始，对变量s通过结构体复合字面量进行赋值
    // 注意，这里不是对s的初始化
    s = (struct S){ .a = 100, .b = -50 };

    printf("s.a = %d, s.b = %d\n", s.a, s.b);
```

```
    // 一个结构体复合字面量能够作为一个变量充当左值，
    // 从而可以对其内部变量成员做修改操作，尽管这通常来说没有实践意义
    ((struct S) { 10, 20 }).a += 10;

}
```

在代码清单6-5中，“（struct S）{.a=100，.b=-50}；”就是一个结构体复合字面量。其形式与初始化非常类似，其实就是在初始化器{}的左边加上对该结构体的投射操作（关于投射操作符可详细参考5.3.2节以及5.6节的内容）。我们看到，对结构体类型S的一个变量s进行赋值时没有通过一个临时变量，而直接使用该复合字面量进行赋值。在代码清单6-5的最后也可以看到，一个结构体复合字面量可以充当左值，从而可直接修改其内部变量成员的值，这一点与其他一般的字面量只能作为常量不同。当然，对结构体复合字面量直接做修改操作通常没有什么实践意义，不过如果我们通过指针直接指向该字面量的话，有时也能方便某些操作。

 注意：所谓初始化是指在声明了一个对象之后，立即用等号操作符=对它做初始赋值。如果在声明时没有用等号操作符对该对象进行操作，那么在此之后就不能再用初始化器对它进行初始化了，而只能对它做一般的赋值处理。所以在上述代码中，如果我们直接用“s={.a=100，.b=-50}；”将会出现编译器报错。

最后，关于匿名结构体还有一个非常有趣的设定。当一个结构体中含有一个匿名结构体，且没有用它来声明一个成员对象时，那么该匿名结构体中所声明的对象成员可直接视为其外部结构体成员那样被直接访问。如代码清单6-6所示。

代码清单6-6　匿名结构体的特性

```
#include <stdio.h>
#include <stdint.h>

int main(int argc, const char * argv[])
{
    // 定义了一个结构体S
    struct S
    {
        // 声明成员对象a
        int32_t a;

        // 定义了一个匿名结构体
        struct
        {
            // 在此定义了成员对象b和c
            int32_t b;
            int32_t c;          // 这里c被字节填充，扩展了4字节
        };  // 这里没声明该匿名结构体的对象作为结构体S的成员

        // 声明成员对象d
        double d;
    }s; // 直接用结构体S声明变量s，且未对它初始化

    s.a = 0;            // 对结构体变量s的成员a赋值
    s.b = 1;            // 对结构体变量s的匿名结构体成员b赋值
    s.c = 2;            // 对结构体变量s的匿名结构体成员c赋值
    s.d = 10.5;         // 对结构体变量s的成员d赋值

    // 结构体S与以下结构体S2的存储布局完全一致
    struct S2
    {
        int32_t a;
        int32_t b;

        int32_t c;      // 这里c被字节填充，扩展了4字节

        double d;
    } s2;

    // 以下代码分别用于获得结构体各个成员的偏移位置
    printf("Offset of b1: %td\n", (ptrdiff_t)&s.b - (ptrdiff_t)&s.a);
    printf("Offset of c1: %td\n", (ptrdiff_t)&s.c - (ptrdiff_t)&s.a);
    printf("Offset of d1: %td\n", (ptrdiff_t)&s.d - (ptrdiff_t)&s.a);

    printf("Offset of b2: %td\n", (ptrdiff_t)&s2.b - (ptrdiff_t)&s2.a);
    printf("Offset of c2: %td\n", (ptrdiff_t)&s2.c - (ptrdiff_t)&s2.a);
    printf("Offset of d2: %td\n", (ptrdiff_t)&s2.d - (ptrdiff_t)&s2.a);
}
```

我们可以看到，代码清单6-6中结构体对象s对成员a、d的访问与其内部匿名结构体中的成员b和c的访问完全相同。另外，C语言标准也明确指出，匿名结构体的成员会被认作包含该匿名结构体的结构体成员，所以存储布局也会与没有该匿名结构体的结构体一样。而对于上述代码中的最后6

行代码，则是用于测试结构体变量s与s2的存储布局的。这里牵涉到结构体字节填充的概念，我们将在6.5节详细介绍。此外，结构体类型不能用作投射操作所指定的类型，即不能将一个对象类型强制转换为一个结构体类型；结构体对象不能直接使用关系操作符来比较大小或判定是否相等。

6.3　联合体类型

联合体（union）类型是C语言中比较特别的类型，在整个计算机编程语言中，除了与C语言基本兼容的C++和Objective-C支持外，也就只有COBOL支持这种类型了。联合体的声明形式与结构体类似，但是与结构体不同的是，联合体对象中的所有成员共享同一存储区域。联合体使用关键字union来声明，union后面可跟一个标识符作为该联合体的名字，如果该标识符缺省，那么称此联合体为匿名联合体。我们先通过代码清单6-7来对比一下联合体与结构体在存储布局上的区别。

代码清单6-7　联合体与结构体存储布局对比代码

```
#include <stdint.h>
int main(int argc, const char * argv[])
{
    struct
    {
        int8_t a;
        int16_t b;
        int32_t c;
    }s = { 0, 1, 2 };

    union
    {
        int8_t a;
        int16_t b;
        int32_t c;
    }u = { 0 };
}
```

代码清单6-7中，我们在main函数里定义了一个匿名结构体，并用它声明了对象s。然后定义了一个匿名联合体，并用它声明了对象u。对象s与对象u的存储布局如图6-1所示（假定使用桌面操作系统与编译器、小端模式的处理器）：

我们假设结构体按照一般主流C语言编译器实现方式做字节对齐。在图6-1中，结构体对象s中的成员a是int8_t类型，占用1个字节，然后填充1个字节，总共耗费2个字节的存储空间；成员b所以从偏移位置2开始，它是int16_t类型，占用2个字节，由于跟a共同构成了4字节块，且下一个成员c占4字节，所以无需字节填充；成员c从偏移位置4开始，占用4字节空间。所以，对象s总共占用了8字节的存储空间。

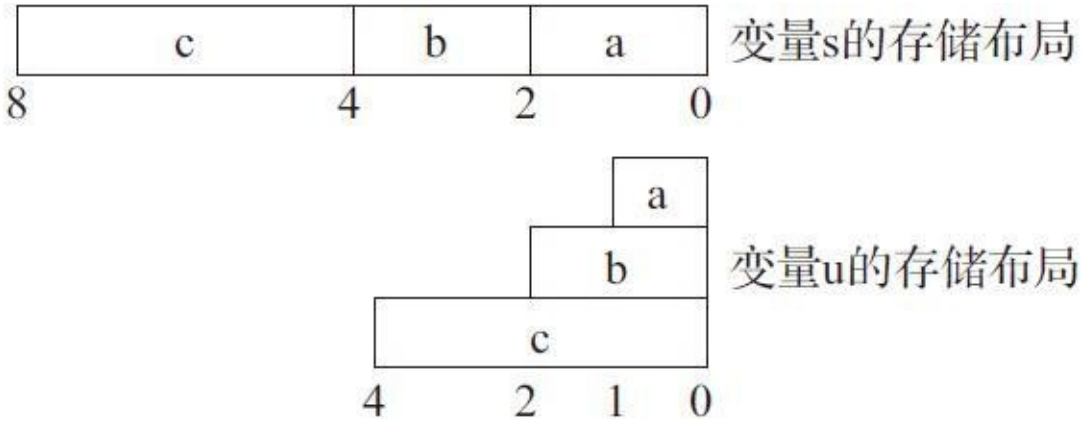


图6-1　结构体与联合体的存储布局

然后再看对象u。联合体对象u的a、b、c三个成员都从偏移位置0开始存放，所以它们完全共享0号字节的存储空间中的内容；然后成员b与成员c共享1号字节所存储的内容；而变量c则自己独享2和3号字节所存储的内容。所以联合体变量u总共只占用4个字节的存储空间。下面我们通过代码清单6-8举一个比较直观的例子来验证上述联合体变量u的存储布局。

代码清单6-8　联合体存储布局可视化代码

```
#include <stdio.h>
#include <stdint.h>

int main(int argc, const char * argv[])
```

```
{
    union
    {
        int8_t a;
        int16_t b;
        int32_t c;
    }u = { .c = 0x04030201 };

    // 输出: a = 0x00000001, b = 0x00000201, c = 0x04030201
    printf("a = 0x%.8x, b = 0x%.8x, c = 0x%.8x\n", u.a, u.b, u.c);
}
```

代码清单6-8先对联合体变量u进行初始化，将它占用字节最多的成员c进行初始化，我们用比较容易观察的十六进制数0x04030201写入到成员c。然后，我们观察成员a、成员b和成员c的值。输出后我们发现成员a的值就是成员c的值的最低8位（即字节0）；成员b则是成员c的低16位（即字节0和字节1）的内容；成员c则是初始化时的值。这样就与我们上面所描述的存储布局完全吻合了。

联合体在构造上与结构体类似，也可以在里面存放其他类型的对象作为其成员，包括枚举、结构体以及联合体对象。下面举一些定义联合体类型的例子。

代码清单6-9　联合体类型的定义

```
// 定义一个名为Union1的联合体
// 它包含三个共享的成员
union Union1
{
    int32_t a;

    int8_t b;

    int16_t s;
};

// 定义一个名为Union2的联合体
static union Union2
{
    // 在Union2中定义一个匿名结构体并直接用它声明成员变量s
    struct {
        int32_t a;
        int32_t b;
```

```
    }s;

    // 声明成员变量f，并与s共享同一联合体存储单元
    float f;

    // 直接用Union2声明一个静态变量un
} un;

enum MyEnum
{
    ENUM1,
    ENUM2
};

union MyUnion
{
    // 用MyEnum枚举类型声明对象e作为MyUnion联合体的成员
    enum MyEnum e;

    // 用联合体Union2声明对象u作为MyUnion联合体的成员
    union Union2 u;

    // 联合体内也可放置静态断言
    _Static_assert(sizeof(un) == 8, "NG");
};

// 定义一个名为MyStruct的结构体
struct MyStruct
{
    int32_t a;

    // 用上面定义的Union2声明u1作为MyStruct的成员
    union Union2 u1;

    // 定义一个匿名联合体，并直接用它声明其对象u2作为MyStruct的成员
    union{
        double d;
        float f;
    }u2;

    double d;
};
```

对联合体对象的初始化也与结构体类似，从C99标准起可使用指定成员的初始化器。不过对联合体的初始化只能初始化其中一个成员，因为所有成员都共享同一存储位置，所以如果对多个成员指定初始化值时编译器可能会发出警告。代码清单6-10是对联合体对象初始化以及赋值的例子。

代码清单6-10　联合体对象的初始化以及成员访问

```
#include <stdio.h>
#include <stdint.h>

int main(int argc, const char * argv[])
```

```
{
    // 定义了一个名为MyUnion的联合体
    union MyUnion
    {
        // 声明了成员a
        int32_t a;

        // 在MyUnion联合体内定义了一个匿名结构体
        struct {
            int16_t b;
            int8_t c;
        }s; // 直接用该结构体声明对象s作为MyUnion的成员

        // 声明了成员f
        float f;

        // 这里直接用MyUnion联合体类型声明对象un，并直接对它初始化
        // 由于这里s.b与s.c作为同一结构体变量s的成员，所以完全可行
    }un = { .s.b = 0x0201, .s.c = 0x03 },

    // 这里，Apple LLVM发出警告，.s.b的初始化器会覆盖它之前的.a的初始化器
    unWarning = { .a = 100, .s.b = -100 };

    // 在C99标准之前，用常规方法只能对联合体对象的第一个成员进行初始化
    union MyUnion un2 = { 100 };    // 即un2.a被初始化为100

    // 这里用MyUnion声明了对象un3，但未对它进行初始化
    union MyUnion un3;

    // 这里(union MyUnion){ .f = 10.5f }是一个联合体的复合字面量
    // 与结构体复合字面量类似。联合体复合字面量可直接对一个联合体对象进行赋值
    un3 = (union MyUnion){ .f = 10.5f };

    // 我们对un.s初始化之后观察un.a的值
    // 输出：un.a = 0x00030201
    printf("un.a = 0x%.8x\n", un.a);

    // 对un2.a初始化之后，我们观察un2.s.b的值
    // 输出：un2.s.b = 100
    printf("un2.s.b = %d\n", un2.s.b);

    // 对un3.f初始化之后，存放在此联合体起始地址的数据为10.5f的
    // 规格化浮点数的二进制表示。
    // 所以我们这里先将un3.a的地址转为浮点，然后再转整型来观察un3.a的值
    // 输出：un3.a = 10
    printf("un3.a = %d\n", (int32_t)*(float*)&un3.a);
}
```

代码清单6-10比较详细地介绍了联合体对象初始化、赋值以及联合体复合字面量的各种情况。此外，这个代码示例也进一步给大家呈现了联合体中所有成员共享存储空间的这一特性。由于联合体对其成员的访问形式与结构体完全一样，因此这里不再赘述。

此外，联合体与结构体一样，不能用作投射操作符所指定的类型；联

合体对象不能直接使用关系操作符来比较大小、判断是否相同等。

6.4　位域

位域（bit-fields）也是C语言比较独特的语法之一。通过位域，我们可以将一串比特流进行结构化描述，这在通信领域用得尤为广泛。在C语言中，位域是在结构体或联合体中指定位宽的成员。通常我们都用结构体来指定位域，尽管联合体也可以指定成员的位宽，但基本没什么实际意义，因为联合体的成员都共享同一存储单元。

6.4.1　位域的一般特性

如果结构体中的某一成员要作为位域，那么它必须是一个整数类型（包括布尔和枚举类型），而不能是浮点或其他类型。指定该位域宽度的表达式也应该是一个整数常量表达式，且表达式的值不能是负数。位域的基本表达式为：

```
<类型> <标识符> : <位宽表达式>;
```

这里<位宽表达式>就用于指定此位域的宽度（即取值范围），单位是比特。比如代码清单6-11所示。

代码清单6-11　结构体位域的简单介绍

```
struct BitField
{
    int32_t a : 5;      // a由5个比特构成，取值范围在[-16, 15]
    uint32_t b : 6;     // b由6个比特构成，取值范围在[0, 63]
```

```
    int32_t c : 7;      // c由7个比特构成，取值范围在[-64, 63]
};
```

代码清单6-11中，BitField的成员a、b和c都是位域。其中，a的宽度为5个比特，由于它是带符号整数，因此这5个比特中最高位需要作为符号位，因此其取值范围在[-2^4，2^4-1]。成员b的宽度为6个比特，由于它是一个无符号整型，因此其取值范围在[0，2^6-1]。整个BitField类型大小为4个字节。由于a、b、c这3个成员的宽度加起来为18个比特，少于32比特，一般C语言实现会将其扩充到4个字节（正好是一个int32_t类型的宽度）。

C语言对位域还有以下限制：

1）不能对位域成员做取地址操作；

2）位域成员不能作为sizeof的操作数；

3）不能用对齐属性来修饰位域；

4）指定位域位宽的常量值不能超过该类型可表示的范围。

代码清单6-12展示了对位域的一些错误的用法。

代码清单6-12　位域的一些错误用法

```
#include <stdio.h>
#include <stdint.h>

int main(int argc, const char * argv[])
{
    struct BitField
    {
        _Alignas(int32_t) int32_t a : 5; // 这句报错！_Alignas属性不能用于位域
        uint32_t b : 6;
        char c : 9;     // 这句报错！一个char类型对象最多只能由8个比特构成
```

```
    }bf = { 10, 20 };

    int32_t *p = &bf.a;                 // 这句报错！不能对位域做取地址操作

    size_t size = sizeof(bf.b);         // 这句报错！sizeof操作符不能用于位域
}
```

上面讲述了位域的形式以及约束。下面我们将描述位域的二进制表达方式，请见代码清单6-13。

代码清单6-13　结构体中位域的二进制表达形式

```
#include <stdio.h>
#include <stdbool.h>

int main(int argc, const char * argv[])
{
    enum MyEnum
    {
        ENUM1,
        ENUM2,
        ENUM3
    };

    struct MyStruct
    {
        int32_t a : 6;                  // a的范围为[-32, 31]
        int16_t b : 5;                  // b的范围为[-16, 15]
        int8_t c : 8;                   // c的范围为[-128, 127]

        char x : sizeof(enum MyEnum);   // 相当于：char x : 4
        bool y : 1;

        enum MyEnum e : ENUM3;          // 相当于：enum MyEnum e : 2

    }s = { 0x18, 0x0a, 0x77, '\0', true, ENUM3 };

    // s的二进制表示：  (0) 10 1 0000 01110111 (00000) 01010 011000
    // 整理后得：0101 0000 0111 0111 0000 0010 1001 1000
    printf("The size is: %zu\n", sizeof(s));

    // 输出0x50770298
    printf("The content of s is: 0x%.8x\n", *(uint32_t*)&s);
}
```

代码清单6-13中，我们在main函数里定义了一个名为MyStruct的结构体类型，其中的成员均为位域。我们看到，位域成员的类型可以是各类整数类型、字符类型、布尔类型以及枚举类型。而指定位域宽度的整数常量

表达式可以是整数字面量、sizeof操作符所得到的结果、枚举值等，它们必须是在编译时能确定值的常量。

6.4.2 位域成员的存放与布局

代码清单6-13中，我们分别对MyStruct结构体对象的每个成员进行初始化，得到s内部表示的二进制如注释中所示。MyStruct结构体类型大小为4个字节，而s的二进制表示中，用圆括号括起来的二进制比特表示是被填充的比特，其他比特表示每个成员本身的值。其中，成员a是最右边的6个比特，然后之后的每个位域成员都依次从右向左排列。比特填充对于不同编译器、不同执行环境可能会不同。不过C语言标准指明的是，对于同一结构体内毗邻两个位域成员，如果第一个位域成员仍然有足够的存储空间容纳第二个位域成员，那么第二个位域成员可以与前一个位域成员打包在一起，一起构成同一单元的毗邻比特串，这个可以参考代码清单6-13中的成员a与b。成员a的类型为int32_t，位宽为6个比特；成员b的类型为int16_t，位宽为5个比特，它与成员a正好可以组合在一起，构成11个比特的串，存放在同一单元。

如果构成一个单元的存储空间不够，那么后一个位域是被放在下一个存储单元还是与之前的单元迭交存放，这是由实现自己定义的。在一个单元内的位域分配次序（从高位序到低位序还是从低位序到高位序）也是由实现自己定义的。此外，可寻址存储单元如何做字节对齐在标准中也是未

指定的。在代码清单6-13中，我们看到成员c占8个比特，如果将它与之前a和b构成的11比特串拼接在一起，那么它超出了之前a和b所在的16比特可寻址存储单元。因此，在Apple LLVM的实现中，它采用的是将成员c放在下一个可寻址存储单元内，所以成员a和b所在的单元最后高5位用0填充，直到正好满16比特。倘若成员c的位宽为5个比特，那么成员c就能与a和b拼接在一起，存放在同一可寻址存储单元，如代码清单6-14所示。此外，对于现在桌面操作系统以及当前主流C编译器而言，在同一可寻址存储单元中采用从低位序到高位序的排列方式。

代码清单6-14　成员c与成员a和b一起拼接的结构体

```
#include <stdio.h>
#include <stdbool.h>

int main(int argc, const char * argv[])
{
    enum MyEnum
    {
        ENUM1,
        ENUM2,
        ENUM3
    };

    struct MyStruct
    {
        int32_t a : 6;                  // a的范围为[-32, 31]
        int16_t b : 5;                  // b的范围为[-16, 15]
        int8_t c : 5;                   // c的范围为[-16, 15]

        char x : sizeof(enum MyEnum);   // 相当于：char x : 4
        bool y : 1;

        enum MyEnum e : ENUM3;          // 相当于：enum MyEnum e : 2

    }s = { 0x18, 0x0a, 0x07, '\6', true, ENUM3 };

    // s的二进制表示：  (000000000) 10 1 0110 00111 01010 011000
    // 整理后得：0000 0000 0101 0110 0011 1010 1001 1000
    printf("The size is: %zu\n", sizeof(s));

    // 输出0x00563a98
    printf("The content of s is: 0x%.8x\n", *(uint32_t*)&s);
}
```

我们上面提到几个邻近的位域成员可以被拼接到同一个可寻址存储单元，那么我们会有疑问，如何确定这一可寻址存储单元的大小呢？一般C语言实现可以先将第一个和第二个位域成员先拼接在一起，如果能成功拼接，那么观察它们已占用的比特个数，向上取满足2^n的最小整数。比如代码清单6-13中，成员a和先与成员b拼接，由于拼接后，没有超出它们俩任一类型的最大宽度范围，所以可以拼接在一起，这样就形成了一个11个比特的串，占用了两个字节的存储单元。然后到了成员c，在代码清单6-13中，c的位宽为8个比特，倘若与a和b拼接在一起，那么就超过双字节的存储单元，此时C语言标准允许C语言实现做一个选择，让成员c放在下一个存储单元，或是让它与a和b也拼接在一起，那么c就横跨了两个存储单元（也就是标准所提到的叠交存放的情况），而Apple LLVM选择了前者，即让成员c存放在下一个单元。在代码清单6-14中，由于成员c占5个比特，与a和b已经串好的11比特拼接在一起后，形成16比特串，正好能存放在当前存储单元。因此在代码清单6-14中，a、b和c都在同一可寻址存储单元。

最后，为了让各位对可寻址存储单元大小的分配能有进一步的认识，我们对代码清单6-13再做一些小修改（见代码清单6-15），然后看看结果。

代码清单6-15　将成员b和c均改为int类型

```
#include <stdio.h>
#include <stdbool.h>
#include <stdint.h>

int main(int argc, const char * argv[])
{
    enum MyEnum
    {
```

```
        ENUM1,
        ENUM2,
        ENUM3
    };

    struct MyStruct
    {
        int32_t a : 6;      // a的范围为[-32, 31]
        int32_t b : 5;      // b的范围为[-16, 15]
        int32_t c : 8;      // c的范围为[-128, 127]

        char x : sizeof(enum MyEnum);
        bool y : 1;

        enum MyEnum e : ENUM3;

    }s = { 0x18, 0x0a, 0x77, '\1', true, ENUM3 };

    // s的二进制表示：  (000000) 10 1 0001 01110111 01010 011000
    // 整理后得：0000 0010 1000 1011 1011 1010 1001 1000
    printf("The size is: %zu\n", sizeof(s));

    // 输出0x028bba98
    printf("The content of s is: 0x%.8x\n", *(uint32_t*)&s);
}
```

代码清单6-15中，我们将成员b与成员c类型晋升到int32_t之后，它们所存放的可寻址存储单元就有32个比特之多。此时，我们可以看到这样就能将此结构体剩余位域成员全都装入其中，所以比特填充只出现在最后几个高位。

6.4.3 匿名位域

在C11标准中，位域可以是匿名的。当某个位域只给出其类型及宽度，而不给出标识符名时，该位域则称为匿名位域。匿名位域一般仅用作比特填充（通常C语言的实现都用0来填充），而不能被直接访问该填充区域。而当匿名位域的宽度为0时，则表示该位域是其前面的位域所组成的存储区域的末尾，即作为一个结束标志而使用。该存储区域的后续比特都将被填充，其下面的位域都将作为一个新的存储区域。下面我们将举一个使

用匿名位域的例子，见代码清单6-16。

代码清单6-16　匿名位域使用示例

```
#include <stdio.h>
#include <stdint.h>

int main(int argc, const char * argv[])
{
    struct
    {
        int32_t a : 8;
        // 这里使用了一个匿名位域，使得位域a与位域b之间空出8比特，
        // 并且这空出的8比特均用0来填充
        int32_t: 8;
        int32_t b : 16;

        // 这里分别给成员a和b进行初始化
        // 对象s仍然具有两个成员（a和b），中间填充部分无法访问
    }s = { 0x21, 0x6543 };

    // 这里输出：s is: 0x65430021
    printf("s is: 0x%.8X\n", *(uint32_t*)&s);

    struct
    {
        int32_t a : 8;
        // 这里使用了一个匿名位域，且宽度为0，
        // 这样位域a所在的整个区域后续都将被0填充，然后终结该区域
        int32_t : 0;

        // 位域b将被安排在位域a的下一个存储区域，而不会跟a存放在同一区域
        int32_t b : 16;

        // 下面的x与y都会与位域b存放在同一存储区域
        int16_t x : 8;

        int16_t y : 8;

        // 这里分别给成员a，b，x，和y进行初始化
    }t = { 0x10, 0x4321, 0x65, 0x76 };

    // 这里输出：t is: 0x7665432100000010
    printf("t is: 0x%.16llX\n", *(uint64_t*)&t);
}
```

6.4.4　位域使用示例

在实践中，我们往往把需要拼接在一起的位域成员用相同类型或相同字节宽度的类型毗邻排放，这样可以避免一些不必要的比特填充，从而能

够获得更好的位域结构化描述。我们下面举一个关于初中学生信息部分属性的例子来描述位域在实践中的使用方式，如代码清单6-17所示。

代码清单6-17　位域在实践中的使用

```
#include <stdio.h>
#include <stdbool.h>
#include <stdint.h>

int main(int argc, const char * argv[])
{
    struct Student
    {
        uint32_t grade : 2;     // 年级（1-3，对于初中生可用0表示预备班）
        uint32_t class : 4;     // 班级，一个年级最多可有16个班级
        uint32_t number : 7;    // 学号，一个班级最多128个学生

        uint32_t weight : 7;    // 体重，一个学生最重128kg
        uint32_t height : 8;    // 身高，一个学生最高256cm

        uint32_t isMale : 1;    // true表示男生；false表示女生
        uint32_t age : 3        // 年龄（10到17岁）
    };

    // 声明了一个学生对象，1年级5班，26号，50kg，160cm，男生，14岁
    struct Student s = { 1, 5, 26, 50, 160, true };

    printf("size is:%zu\n", sizeof(s));
}
```

代码清单6-17中我们定义了名为Student的一个结构体，整个结构体的大小为4个字节，而就这4个字节我们就定义了一个学生的7个属性。由于在这个学生的结构体中很多属性不需要一个字节（8比特）来表示，而只要很少几个比特就能表示其范围了。比如第一个属性“年级”，一所中学的年级一般为1到3年级，所以我们用2个比特就能表示这个范围，2比特的无符号整数可表示的范围是[0，3]。这里，对一个学生对象s初始化的各个属性为：1年级，5班，学号为26号，体重50kg，身高160cm，男生，14岁。我们可以看到，通过位域可以将信息做些压缩，使得数据对象信息尽可能少地占用存储空间。如果以上这些属性都要用8位无符号整数来表示（比如像

Java语言就不支持位域），那么就至少需要7个字节。

6.5 字节对齐与字节填充

在前几节我们已经提到了一些关于字节对齐与字节填充的使用场景，本节将详细描述C语言中的字节对齐与字节填充。

在2.5节中，我们已经大致描述了字节对齐这个概念，并解释了为何计算机系统需要字节对齐。由于C语言是非常贴近底层硬件的高级编程语言，所以它本身就含有诸多字节对齐的语法特性。然而字节对齐对于不同架构的处理器可能会有不同要求。对于一般32/64位系统环境，通常编译器会默认对指令与数据做4字节对齐。也就是说，一个函数的起始地址以及一个数据对象的起始地址都会是4字节的倍数（比如地址0x01001000，0x00400204等）。而像在ARM处理器架构下，每条ARM指令所在的地址都必须是4字节对齐（即4字节的倍数），否则执行指令时就会引发异常；而在AArch32环境下，对每个函数的栈起始地址要求是8字节对齐（即8字节的倍数），由于这样能优化LDP指令（用于一下子读取两个连续的4字节字的操作）对数据的读取；而在AArch64环境下，则要求每个函数的栈的起始地址应该为16字节对齐。

6.5.1 _Alignof操作符

在C语言中，每个完整的对象类型都具有对齐要求。所谓N字节对齐是指分配该对象存储空间时，其起始地址必须是N字节的倍数。比如，一个

对象要求4字节对齐则意味着给该对象分配的起始地址必须能被4整除。尽管每个C编译器的实现对于对齐要求可能有些不同，不过在传统的桌面环境中，一般基本数据类型的对齐参照其数据类型本身的大小。比如，一个char类型的对齐要求为1个字节；一个short类型的对齐要求为2个字节；一个int类型的对齐要求为4个字节，等等。C11标准引入了_Alignof关键字来查看当前指定对象的对齐要求。

_Alignof的用法与sizeof类似，但有两点不同：

1）sizeof后面可直接跟一个对象标识符而不需要圆括号，但_Alignof则不行。_Alignof后面必须跟着一对圆括号，然后在圆括号里存放操作数。

2）_Alignof的操作数只能是类型名（基本类型、枚举类型、结构体、联合体等），而不能是一个表达式（包括对象标识符都不行）。不过GNU语法扩展能使得_Alignof的操作数为表达式，这样我们在GCC、Clang编译器中就能使用此特性。

此外，_Alignof同样也返回一个size_t类型的值表示指定类型的对齐要求。

顺应C11标准的实现都应该含有一个max_align_t类型，表示当前实现对基本对齐要求的最大对齐的支持。如果某个C语言实现无法直接使用max_align_t，那么可以通过包含<stddef.h>头文件来解决。所谓基本对齐要求是指C语言编译器在没有受到程序员指定的情况下自己做出的默认对齐要求。比如上面提到的char、short、int类型的默认对齐要求。而对于一个

数组或一个结构体，如果它们的存储空间非常大，此时编译器将会按照_Alignof（max_align_t）字节对齐要求对它们做对齐。max_align_t这个类型也是编译器自己实现的，像Apple LLVM编译器是将它定义为long double类型（大小为16字节）；在GCC、MSVC与VS-Clang中被定义为double类型（大小为8字节）。此外，我们在查询对齐要求的时候可以引入标准头文件<stdalign.h>，这里面会有对_Alignof所定义的宏——alignof。alignof是用在C++中的关键字，其作用与C语言的_Alignof一样。下面我们通过代码清单6-18来看看_Alignof的具体使用方式。

代码清单6-18　_Alignof的使用

```
#include <stdio.h>
#include <stdbool.h>
#include <stdalign.h>
#include <stddef.h>

int main(int argc, const char * argv[])
{
    // 这里查询当前编译器的最大对齐字节数
    size_t size = _Alignof(max_align_t);
    printf("Max alignment is:%zu\n", size);

    // 这里查询布尔类型的对齐字节数
    // 我们引入了<stdalign.h>头文件后可直接使用alignof
    size = alignof(bool);
    printf("Boolean alignment is: %zu\n", size);

    // 定义一个结构体S，然后声明一个变量s并对它初始化
    struct S
    {
        int a;
        float f;
        double d;
        long double ld;
    }s = { 0, 1.0f, 10.5, 1000.0005L };

    // 在C11标准中，alignof操作数必须是一个类型名，不能是一个表达式
    // 所以这里只能用alignof(struct S)，而不能用alignof(s)
    size = alignof(struct S);
    printf("S alignment is: %zu\n", size);
}
```

6.5.2 _Alignas对齐说明符

6.5.1节介绍的是类型默认对齐以及_Alignof操作符的使用。但在很多情况下，通过编译器的默认对齐方式无法满足对齐要求，尤其涉及高性能计算领域时，我们可能要将一个结构体或一个数组以16字节对齐甚至64字节对齐的方式存放。

C11标准引入了对齐说明符——_Alignas操作符，用于显式指定某个对象以多少字节对齐。_Alignas操作符的用法与_Alignof类似，不过_Alignas的操作数可以是一个常量表达式。该常量表达式用于显式指明当前指定对象以多少字节对齐。另外与_Alignof类似的是，当我们引入<stdalign.h>头文件后，我们可直接使用alignas来代替_Alignas。

一般C11标准的实现对_Alignas操作数也有一些约束和限制：

1）_Alignas操作数所指定的字节对齐大小不能小于当前C语言实现默认的对齐大小。比如，一个int对象的默认对齐大小如果是4字节，那么当我们使用_Alignas（1）int a；来声明对象a的时候，编译器可能会报错。

2）_Alignas的操作数应该是0、1、2、4等无符号整数。当其操作数大于4时，该操作数必须是4的倍数，否则编译器可能报错。比如，_Alignas（5），_Alignas（10）均可能是无效的对齐说明符。当_Alignas的操作数为0时，表示忽略此对齐说明符，此时编译器将会以默认方式对指定对象采取字节对齐要求。

3）编译器实现一般都会对对齐说明符指定一个最大可对齐的字节数，在Apple LLVM编译器中指定为256MB，如果_Alignas的操作数的值超过256×1024×1024，那么编译器将会报错。

此外，在一个对象声明符中，可多次使用对齐说明符，如果在一条声明语句中存在多个对齐说明符，则声明的对象将按照最大指定的对齐字节数来对齐。

下面，我们将通过代码清单6-19来描述对齐说明符的用法以及效果。为了观察实际使用对齐说明符后的效果，我们需要动用GNU语法扩展，所以清单6-19中的代码大家需要在GCC或Clang编译器中尝试。

代码清单6-19　_Alignas操作符的使用及效果

```
#include <stdio.h>
#include <stdbool.h>
#include <stdalign.h>

int main(int argc, const char * argv[])
{
    // 这里声明了int类型对象a1，并观察其默认的对齐字节数
    int a1 = 0;
    size_t align = alignof(a1);
    printf("a1 alignment is: %zu\n", align);

    // 用double对齐属性来给int类型对象a2做字节对齐。
    // Alignas对齐说明符可以放在类型后面，对象标识符前面
    int _Alignas(double) a2 = 0;
    align = alignof(a2);
    printf("a2 alignment is: %zu\n", align);

    // int类型对象a3的对齐有些复杂，
    // 它是从int、double以及指定的64字节对齐这三种对齐方式中，
    // 选出最大对齐字节数的那个作为对齐要求
    alignas(int) alignas(double) alignas(64) int a3 = 0;
    align = alignof(a3);
    printf("a3 alignment is: %zu\n", align);

    // 这里，对对象d采用指定为0的对齐，那么它仍然用默认的对齐方式
    double alignas(0) d = 0;
    align = alignof(d);
    printf("d alignment is: %zu\n", align);

    // 这里演示了当我们吃不准当前环境某一类型对象的基本对齐要求时，
    // 我们可以使用条件表达式来判断我们要求的对齐字节数是否小于基本对齐要求，
```

```
    // 我们选取自己要求的对齐字节数与基本对齐要求字节数之间的最大值
    long long alignas(alignof(long long) > 8 ? alignof(long long) : 8)
        ll = 0LL;
    align = alignof(ll);
    printf("ll alignment is: %zu\n", align);

    // 对齐说明符也能修饰一个结构体成员
    struct
    {
        // 这里，成员对象a以16字节对齐
        int alignas(16) a;
        int b;
    }s = {0, 0};

    printf("s size is: %zu\n", sizeof(s));
    printf("s alignment is: %zu\n", alignof(s));

    // 这条声明语句是错误的。
    // 因为一般桌面环境中，int类型的基本对齐要求字节数为4字节，
    // 而这里指定的字节对齐要求为1个字节，小于4字节的基本对齐要求
    int alignas(char) err = 0;
}
```

代码清单6-19展示了_Alignas操作符的若干种用法。包括显式指定对齐大小与默认对齐大小的区别，由_Alignas引出的对齐说明符放置的位置（可在类型名之后，但必须在对象标识符之前），在一条声明语句中同时存在多个对齐说明符的情况，以及_Alignas操作数使用常量表达式的情况。

通过以上两个例子，我们应该能够基本掌握在C11标准中如何查询某个类型的对齐字节数以及如何自己指定某个对象的对齐字节数。

6.5.3 结构体成员的字节对齐与字节填充

下面我们将介绍结构体成员的字节对齐以及字节填充。C语言中，结构体的字节填充一般也是根据其成员的对齐情况来确定的。C11标准只是提到了一个结构体或联合体对象的每个非位域成员，以实现定义的、适合

该成员对象类型的方式进行对齐；在一个结构体或联合体的末尾可以做字节填充。因此，结构体成员的字节对齐以及字节填充也主要根据C编译器自身的实现而定。对于当前桌面系统上的主流C语言编译器而言，主要遵守以下规则来判定每个成员的字节对齐与填充:

1）结构体第一个成员所在的偏移地址为0（即从0开始计）。

2）每个成员根据其类型或程序员指定的对齐字节数来判定它所在的偏移地址。如果该成员要求4字节对齐，那么它所处的偏移地址就应该是4的倍数，如果不是4的倍数，则向上取不小于当前偏移值的4的倍数的最小整数。

3）确定了当前成员对象的偏移地址之后，它的起始地址到前一个对象之间空白的存储空间用0来填充。

4）结构体对象的字节对齐要求与其成员中最大字节对齐要求相一致。

代码清单6-20给出了C编译器默认对齐情况的结构体成员字节对齐与填充。

代码清单6-20　默认对齐的结构体成员字节对齐与填充

```
#include <stdio.h>
#include <stdalign.h>
#include <stdint.h>

int main(int argc, const char * argv[])
{
    struct
    {
        int8_t c;
        int32_t i;
        int16_t s;
```

```
        int64_t d;
    }s = { 0x10, 0x20, 0x30, 0x40 };

    printf("s alignment: %zu\n", alignof(s));
    printf("size is: %zu\n", sizeof(s));
}
```

代码清单6-20中声明了一个匿名结构体对象s，其成员有c、i、s和d。这里，成员c以1个字节对齐；成员i以4个字节对齐；成员s以2个字节对齐；成员d以8个字节对齐。根据规则4，该结构体对象s是以8字节对齐的（即对齐要求与其成员d一致）；而s的大小为24个字节。

我们下面来分析一下代码清单6-20中匿名结构体中成员的字节对齐情况，见表6-1。

表6-1　代码清单6-20中匿名结构体成员布局

成员对象标识符	对齐字节数	占用存储空间大小	偏移量地址
c	1	1	0
i	4	4	4
s	2	2	8
d	8	8	16

表6-1列出了结构体中每个成员的偏移地址、对齐字节数以及占用存储空间大小。

1）对象c类型为int8_t，所以占1个字节大小，对齐要求为1字节。由于它是第一个成员，所以存放在偏移地址0单元处。

2）对象i是第二个成员，它是int32_t类型，占4个字节，因此对齐要求为4字节。由于偏移地址1不是4的倍数，所以需要向上找4的倍数的最小整数，这样正好是偏移地址4，那么成员i就占用了偏移地址单元4到7，而偏

移地址单元1到3正好是空出来的，用0来填充。

3）对象s是第三个成员，类型为int16_t，占2个字节，因此对齐要求也是2个字节。由于偏移地址8满足2的倍数，因此s就存放在偏移地址8处，并占用偏移地址单元8和9。

4）对象d是第四个成员，它是int64_t类型，占用8个字节，在Clang、GCC与MSVC中的对齐要求也是8字节。由于偏移地址10不满足8的倍数，因此需要向上找到是8的倍数的最小整数，因此它被存放在偏移地址16处，并占用偏移地址单元16到23。

由上可知，整个结构体对象的大小就是24字节。下面图6-2展示了结构体对象s的存储布局，该图是在Apple LLVM 8.0编译环境下通过Xcode集成开发环境捕获到的。

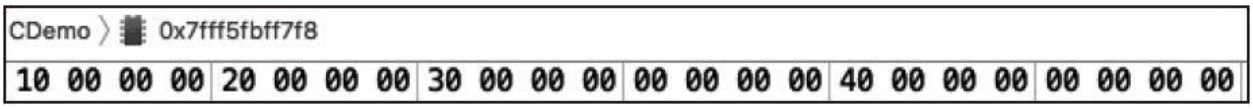


图6-2　结构体对象s的存储布局

图6-2中，每个数都是十六进制的，表示一个字节。以竖线划分4个字节为一组，这样容易观察。其中，数值0x10处于偏移地址单元0处；数值0x40在偏移地址16处。通过这个图我们可以很直观地看清字节填充与对齐的实现。

此外，在C语言标准中还有一个offsetof宏，用于获取当前成员所在的

偏移位置。它定义在<stddef.h>头文件中。代码清单6-21展示了它的用法。

代码清单6-21　offsetof的使用

```
#include <stdio.h>
#include <stddef.h>
#include <stdalign.h>
#include <stdbool.h>
int main(int argc, const char * argv[])
{
    struct S
    {
        int16_t s;

        // 这里对象b按照int32_t类型的对齐要求进行对齐，即4字节对齐
        alignas(int32_t) bool b;

        int32_t i;

        // 这里定义了一个匿名联合体，并用它声明一个对象un
        // un按照16字节对齐。
        // 这里需要注意，un的对齐说明符的值不能小于8，因为其成员最大对齐字节数为8
        alignas(16) union
        {
            // 指定成员c的对齐要求为8字节对齐
            alignas(8) char c;
            float f;
        }un;    // un的大小为4字节，且按16字节对齐
    };

    size_t offset = offsetof(struct S, s);
    printf("s offset is: %zu\n", offset);

    offset = offsetof(struct S, b);
    printf("b offset is: %zu\n", offset);

    offset = offsetof(struct S, i);
    printf("i offset is: %zu\n", offset);

    offset = offsetof(struct S, un);
    printf("un offset is: %zu\n", offset);

    printf("S alignment is: %zu\n", alignof(struct S));
}
```

以上我们描述了结构体内的成员对齐以及字节填充，而之前提到过，C语言标准允许实现对结构体或联合体的末尾做字节填充，而填充大小也是由实现定义的。一般来说，结构体末尾的填充根据该结构体当前成员最大对齐字节数进行填充。比如，如果当前成员的最大对齐字节数为8字节，那么倘若当前结构体的大小不满足8字节的倍数，则填充满8字节的倍数为

止。换句话说，结构体对象的大小应该是其对齐字节数的倍数。因此，当确定一个结构体最后一个成员的位置与大小之后，后面所填充的字节需要保证整个结构体的大小正好与其对齐字节数成倍数关系。我们下面来看代码清单6-22所举的一个例子。

代码清单6-22　结构体末尾字节填充

```
#include <stdio.h>
#include <stddef.h>
#include <stdalign.h>
#include <stdbool.h>

int main(int argc, const char * argv[])
{
    struct S
    {
        int i;                      // 偏移地址为0；大小为4字节；4字节对齐

        alignas(64) long double ld; // 偏移地址为64；大小为16字节；64字节对齐

        char c;                     // 偏移地址为80；大小为1字节；1字节对齐
    };

    printf("S alignment is: %zu\n", alignof(struct S));
    printf("S size is: %zu\n", sizeof(struct S));

    size_t offset = offsetof(struct S, i);
    printf("i offset: %zu\n", offset);

    offset = offsetof(struct S, ld);
    printf("ld offset: %zu\n", offset);

    offset = offsetof(struct S, c);
    printf("c offset: %zu\n", offset);
    struct s2
    { int i;
        char x, y, z, w;
    }
    printf("S2 alignment is %zu\n", alignof(struct S2));
    printf("S2 size is %zu\n", sizeof(struct S2));
    printf("w offset: %zu\n", offsetof(struct S2, w));
}
```

代码清单6-22中可清晰看到，由于结构体S的最大对齐字节数的成员ld为64字节对齐，所以结构体S的对齐要求即为64字节。当成员c放置在偏移地址80之后，需要在末尾填充值为0的字节，一直到结构体S的大小能满足

64的倍数为止。因此结构体S的最终大小为128字节。而对于结构体S2，其最大对齐成员类型为int，也就是对象i，因此它也以4字节要求对齐。最后4个x、y、z、w成员都是单字节对象，因此均满足对齐要求，所以S2中的所有成员都不存在字节填充的情况。此外，S2大小在默认情况下已经是8字节了，满足4字节要求对齐的倍数，因此在w成员之后也不需要额外的字节填充。

6.6 复数类型

之前我们在第5章给大家介绍了整数类型与浮点数类型，这两种类型统称为实数类型。本节我们将给大家介绍复数（complex number）类型。之所以在这里介绍复数类型是因为复数本身就是一个复合类型，我们知道一个复数是由实部和虚部构成的，因此在C语言中一个复数对象也具有两个成员，一个表示复数的实数部分，另一个表示复数的虚数部分。

复数类型从C99标准开始引入，并且复数的实部与虚部的数据类型只能是float、double与long double这三种浮点类型。遵循GNU语法扩展的C语言实现允许复数类型为整数类型，因此在GCC与Clang编译器中都能使用整数类型的复数。声明一个复数对象时，使用_Complex关键字，再加浮点类型。比如，我们要声明一个实部与虚部都为float单精度浮点类型的复数，使用float_Complex comp；。这里，关键字_Complex与类型名可以互换位置。

我们在使用复数时，一般需要包含头文件<complex.h>。此头文件包含了诸多对复数进行操作的库函数，包括取复数的实部和虚部的值等API。此外，这个头文件中还定义了_Complex的简化、标准形式——complex。因此，我们后面可直接用complex来声明一个复数对象，这样既简洁，而且又能跨平台。比如像当前的MSVC并不直接支持复数类型，但支持<complex.h>标准库头文件，并以结构体方式给出复数实现。要表示一个复

数字面量可以直接用算术表达式，比如3.0f+0.5fi表示一个实部为3.0、虚部为0.5的一个单精度浮点型复数。这里，后缀i或I表示复数的虚部，一般建议使用大写的I，这也是在<complex.h>所定义的表示虚部的后缀。I可以放在表示单精度浮点的后缀f之前，比如0.5If也是合法的。此外，虚部也可以写成诸如0.5f*I这样的表达式。复数除了上述这种直接写表达式的形式之外也可以像结构体对象那样初始化，比如：float_Complex comp={3.0f，0.5f}；，这里复数对象comp的实部为3.0f、虚部为0.5f。

上面我们描述了复数对象的声明方法以及初始化方式。除了上述这些特性外，复数类型也支持类似于结构体那样的复合字面量。比如，（float_Complex）{-1.25f，2.75f}表示一个实部为1.25、虚部为2.75的单精度浮点复数对象。

遵循GNU语法扩展的编译器可直接通过__real__操作符来获取复数的实部，通过__imag__操作符来获取复数的虚部。这两个操作符的操作数应该是一个复数类型的对象，且操作数中的计算副作用仍然会保留，这点跟sizeof操作符有所不同。如果__real__与__imag__的操作数是一个实数，那么该实数将会被隐式地转为一个复数，该复数的实部与实数相同（除非涉及浮点数的转换，那么精度可能会有所影响）；虚数部分可以是正0（对于浮点数而言）或无符号的0。当我们引入了头文件<complex.h>之后，我们可以使用标准库的crealf、creal和creall来获取复数的实部，crealf函数用于获取类型为单精度浮点（float）的复数的实部，creal用于获取类型为双精度浮点（double）的复数的实部，creall用于获取类型为扩展双精度浮点

（long double）的复数的实部。而cimagf、cimag、cimagl这些库函数则用于获取复数的虚部。其中，cimagf用于获取类型为单精度浮点的复数的虚部，cimag用于获取类型为双精度浮点的复数的虚部，cimagl用于获取类型为扩展双精度浮点的复数的虚部。此外，头文件<complex.h>还包含了用于计算复数的三角函数、对数等多种库函数工具。

复数的加减乘除四则运算与实数一样，可直接通过+、-、*、/这些运算符进行计算。此外，一个复数可以转换为一个实数，此时复数的实数部分转为实数（数据类型转换遵循实数的类型转换规则），而虚数部分则完全舍弃。复数之间不能判别大小，但可以判别是否相等。下面我们将举一些例子来描述复数的一些使用方法，见代码清单6-23。

代码清单6-23　复数的使用

```
#include <stdio.h>
#include <stdbool.h>
#include <complex.h>

int main(int argc, const char * argv[])
{
    // 声明了一个单精度浮点型复数对象comp,
    // 并用一个复数计算表达式对它进行初始化
    float complex comp = 2.5f + 1.5fI;

    // 对复数comp的虚部加3.0f
    comp += 3.0f * I;

    // 对复数comp的实部加0.5f
    comp += 0.5f;

    // 分别获取复数comp的实部和虚部
    float r = crealf(comp);
    float i = cimagf(comp);
    printf("r = %f, i = %f\n", r, i);

    // 这里声明一个双精度浮点型的复数对象comp2,
    // 并用类似结构体的方式对它进行初始化，其实部为10.5、虚部为0.5
    double complex comp2 = {10.5, 0.5};

    r = creal(comp2);
    i = cimag(comp2);
    printf("r = %f, i = %f\n", r, i);
```

```
    // 这里声明了一个扩展双精度浮点型的复数对象comp3，
    // 然后没有直接对它进行初始化
    long double complex comp3;

    // 这里用一个扩展双精度浮点型的复数字面量赋值给comp3
    comp3 = (float complex){ -1.25L, 2.75L };

    // 分别获取comp3的实部与虚部
    r = creall(comp3);
    i = cimagl(comp3);
    printf("r = %f, i = %f\n", r, i);

    // 这里复数comp2直接转为double类型，然后赋值给f变量。
    // 此时，comp2的实部将double类型转为float类型赋值给f，并丢弃虚数部分
    float f = comp2;
    printf("f = %f\n", f);

    // 可以将一个实数直接赋值给一个复数对象，
    // 这里将整数100转换为float类型，然后赋值给复数的实数部分，其虚数部分为0
    comp = 100;
    printf("comp real is: %f, imag is: %f\n", crealf(comp), cimagf(comp));

    // 这里对comp2做复数的四则运算
    comp2 = (comp2 + (0.25 - 3.0I)) / comp3;
    printf("comp2 real is: %f, imag is: %f\n", creal(comp2), cimag(comp2));

    // 复数之间不能用>、>=、<、<=来比较大小，但可以使用==及!=来判断是否相等
    bool b = comp == (float complex){100.0f, 0.0fI};
    printf("Is equal? %d\n", b);
}
```

代码清单6-23展示了上述所讲的关于复数使用的一些基本语法特征，包括对复数的初始化、赋值、复合字面量、获取实部与虚部、四则运算、比较是否相等、与实数之间的转换等。

C11标准还提供了C语言编译器可选实现的_Imaginary类型，表示纯虚数。不过这个特征没有被当前主流编译器的任何一款所支持，这里不做展开介绍。不过其用法与_Complex差不多，只不过表示的数据对象为一个纯虚数。如果纯虚数类型被C编译器支持，那么在<complex.h>头文件中我们也能直接使用imaginary。不过由于complex所引入的复数类型已经涵盖了纯虚数，所以一般编译器也不会再去支持imaginary。

6.7　本章小结

本章给大家介绍了枚举、结构体、联合体等用户自定义类型，其中结构体又称为聚合类型。最后又介绍了复数类型。自此，C语言中的对象类型就已经全部介绍完了。其他出现的类型都将是这些对象类型的引申。

下一章我们将介绍C语言中的一个重点语法——指针。它是数据对象的一个属性，并且伴随着本章与第5章所介绍的类型。

第7章　C语言的数组与指针

本章将介绍C语言中的数组与指针。有不少程序员都称C语言中的指针是整个C语言的精髓，尽管个人认为C语言的真正精髓是其整个类型系统（type system），但指针确实也扮演着非常核心的角色。所以对于C语言初学者来说，学好指针是十分关键的。

本章将先介绍C语言中的数组，然后介绍指针以及指针与数组的关系，随后再介绍一下C语言的字符串字面量，最后介绍C语言的完整类型与不完整类型。其中，数组与指针都属于对象类型的一个类别（category），其中数组与结构体一样，也属于聚合类型。

7.1 一维数组

一维数组是C语言中比较常用的聚合类型，它表示一组具有相同类型的多个元素的有序组合。声明一个一维数组的基本形式为：

```
< 类型名 > < 对象标识符 > [ < 数组元素个数 > ];
```

其中，数组元素个数可以是一个计算表达式。在C99之前，计算表达式必须是一个整数常量表达式，而在C99之后，它可以不必为整数常量表达式，一个需要在运行时计算的整数变量表达式也能用来指定一个数组的元素个数。如果指定数组长度的表达式是一个变量，或者需要在运行时才能确定的值，那么该数组又被称为变长数组（variable length array），这将在后续的7.3节中做详细介绍。表示数组元素个数的常量表达式的值应该是一个正整数；而在GNU语法扩展中，允许这里的常量表达式为0，表示一个长度为0的空数组，该数组对象占用0个字节的存储空间。

由于数组由一个或多个相同类型的对象组成，所以我们可以访问数组对象中的任一元素。访问数组的某一元素时，我们使用数组对象标识符，然后后面加[i]。其中，数组对象标识符是一个后缀表达式，[]是一个下标操作符（subscript operators），它是一个单目操作符，其前面的后缀表达式则作为它的操作数。[i]表示的是数组下标，数组下标中可以指定整数常量表达式或整数变量作为指定当前的数组元素索引（index），比如上面[]中的i。数组下标索引值可以是正数，也可以是负数，或者0。负数索引值

一般用于指针对象，而不直接用于数组对象本身。访问数组对象的第1个元素时，使用下标[0]，访问第2个元素使用下标[1]，然后依此类推。

对一个数组对象的初始化使用{}，作为其初始化器，里面存放相应元素进行初始化的初始化器列表。如果列表长度小于数组本身的长度，那么后面多出来的元素部分被初始化为0。从C99标准开始，可以通过指定数组下标的方式为数组元素进行初始化，这也被称为受指定的初始化器（designated initializer）。

下面举一些简单的例子来描述一维数组的对象声明以及对其元素的访问。

代码清单7-1　一维数组的初始化及元素访问

```
#include <stdio.h>
#include <stdbool.h>
#include <complex.h>
#include <stdint.h>

int main(int argc, const char * argv[])
{
    // 声明一个含有5个类型为char的元素的数组对象c,
    // 并分别用'a'、'b'、'c'、'd'、'e'对其进行初始化
    char c[2 * 2 + 1] = { 'a', 'b', 'c', 'd', 'e' };

    // 分别将数组c的第1个元素到第5个元素的值打印出来
    printf("c[0] = %c, c[1] = %c, c[2] = %c, c[3] = %c, c[4] = %c\n",
           c[0], c[1], c[2], c[3], c[4]);

    // 声明一个含有4个类型为int32_t的元素的数组对象a,
    // 并分别用100, -1, 0, 0对它进行初始化,
    // 数组的初始化列表末尾也可以添加一个额外的逗号
    int32_t a[4] = { 100, -1, };

    // 将数组a的第1个到第4个元素的值都打印出来
    printf("a[0] = %d, a[1] = %d, a[2] = %d, a[3] = %d\n",
           a[0], a[1], a[2], a[3]);

    // 声明一个含有5个类型为float的元素的数组对象f,
    // 对其第二个元素初始化为10.5f, 第4个元素初始化为-0.5f, 其余元素均为0.0f
    float f[5] = { [1] = 10.5f, [3] = -0.5f };

    // 将数组f的第1到第5个元素的值都打印出来
    printf("f[0] = %f, f[1] = %f, f[2] = %f, f[3] = %f, f[4] = %f\n",
           f[0], f[1], f[2], f[3], f[4]);
```

```
    // 声明一个含有3个类型为int32_t的元素的数组b,
    // 然后用之前的c[0], a[0], f[1]对其初始化。
    // 这里在声明数组b的时候不指明其大小,
    // 此时将通过初始化列表中元素的个数来确定其元素个数
    int32_t b[] = { c[0], a[0], f[1] };

    // 这里仅声明了一个含有3个类型为int32_t的元素的数组d,
    // 但数组d的每个元素都没有被初始化；如果没有初始化列表，那么必须指定数组元素个数
    int32_t d[3];

    // 分别为数组d的三个元素赋值
    d[0] = b[2];    // 将数组b的第3个元素赋值给数组d的第1个元素
    d[1] = b[1];    // 将数组b的第2个元素赋值给数组d的第2个元素
    d[2] = b[0];    // 将数组b的第1个元素赋值给数组d的第3个元素

    // 将数组d的三个元素的值分别打印出来
    printf("d[0] = %d, d[1] = %d, d[2] = %d\n",
           d[0], d[1], d[2]);

    // 这里声明了一个含有8个int16_t类型元素的数组s，并对它进行初始化,
    // 使用了指定下标的初始化器与顺序下标初始化器的混合方式,
    // 此时确定数组对象元素个数的方式为：判定初始化列表中指定下标的最大索引值；
    // 然后将此最大下标索引值加1就是该数组对象元素的个数,
    // 这里，指定下标初始化器的下标最大索引值为6，所以一共是7个元素。
    // 其中第1个元素为-1；第2个元素为2；第3个元素为0；第4个元素为10；
    // 第5个元素为1；第6个元素为20；第7个元素为5。
    // 当指定下标初始化器之后出现一个顺序下标初始化器时,
    // 此顺序下标的索引值为前一个指定下标的索引值加1，所以这里最后一个值为20的元素的下标索引为5
    int16_t s[] = { -1, 2, [3] = 10, [6] = 5, [4] = 1, 20 };

    // 将数组s的各个元素的值都分别打印出来
    printf("s[0] = %d, s[1] = %d, s[2] = %d, s[3] = %d\n",
           s[0], s[1], s[2], s[3]);
    printf("s[4] = %d, s[5] = %d, s[6] = %d\n",
           s[4], s[5], s[6]);

    struct T
    {
        int32_t a, b;
    };

    // 这里声明了一个含有4个元素的struct T结构体类型的元素的数组
    // 此声明包含了各种有效的结构体对象元素初始化器与数组初始化器进行有效混用的方式。
    // 其中第一个元素的初始化器展示了可完全将其各个成员的值按次序摆在数组初始化列表中；
    // 第二个元素的初始化器则是使用了更直观的{ }来初始化其结构体成员；
    // 第三个元素是用了指定下标的数组初始化器结合指定成员的结构体初始化器；
    // 第四个元素则是分别对其结构体成员进行初始化
    /** 数组对象t的内容如下所示（由Xcode调试器展示出）：
     (T [4]) t = {
     [0] = (a = 10, b = 20)
     [1] = (a = 1, b = 2)
     [2] = (a = -1, b = -2)
     [3] = (a = 4, b = 5)
     }
     */
    struct T t[] = { 10, 20, { 1, 2 }, [2] = {.a = -1, .b = -2},
        [3].a = 4, [3].b = 5 };
}
```

代码清单7-1详细描述了数组对象的声明、初始化以及元素访问。为了直观，后面我们统一将a[0]精简地描述为数组对象a的0号元素，这也表示

数组a的第一个元素，这样表述更为直观，且不会引起概念上的紊乱，而且也比“下标索引为0的元素”的表达要更为精简。

一个一维数组对象，比如int a[10]；其类型为int[10]，表示含有10个int类型对象的数组类型，对象a的类型类别为数组。一个数组对象中的元素通常是按照从低地址到高地址的顺序来存放的。0号元素放在数组对象的起始地址，1号元素放在数组对象的起始地址加上数组元素类型的长度所得的地址处。当一维数组对象作为sizeof的操作数时，sizeof操作符所返回的结果为该数组元素个数乘以该数组元素类型的长度。比如，上述的数组对象a，sizeof（a）的结果就相当于10*sizeof（int），在32位或64位系统下，结果就是40。也就是说，数组对象a所占的存储空间为40个字节。图7-1展示了int b[3]={1，2，3}的存储空间布局（在Apple LLVM 8.0，macOS 10.12系统下运行得出）。

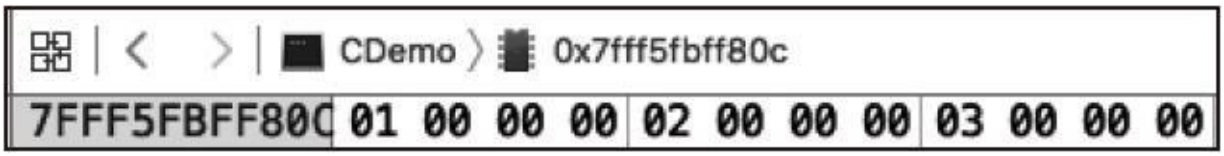


图7-1　一维数组对象b的某个存储布局

图7-1展示了上述数组对象b在macOS 10.12下运行时的存储布局。这里说明一下，数组b对象自身所在的地址为0x7fff5fbff80c，其0号元素1所在地址就是0x7fff5fbff80c，其1号元素2所在地址为0x7fff5fbff810，其2号元素3所在地址为0x7fff5fbff814。数组对象b的总大小为12个字节。

数组类别的对象与其他对象不同，不能将一个数组对象直接赋值给另

一个数组对象，即便对一个数组对象进行初始化时也同样如此，但这里有一个例外就是匿名数组对象（即数组的复合字面量）可以在对某个数组对象初始化时直接赋值给它。要把一个数组对象中的某些元素赋值给另一个数组对象的某些元素时，通常使用单个元素赋值的方式，或是通过<string.h>中声明的库函数memcpy做字节拷贝。代码清单7-2展示了匿名数组的形式以及给数组元素赋值的方式。

代码清单7-2　匿名数组的形式以及给数组元素赋值的方式

```
#include <stdio.h>
#include <string.h>

int main(int argc, const char * argv[])
{
    // 这里声明了数组对象a，其类型为int[3]，并直接用初始化列表对它初始化
    int a[] = { 1, 2, 3 };

    // 输出数组对象a的大小
    printf("a size is: %zu\n", sizeof(a));

    // 这里声明了数组对象b，其类型为int[3]，并用一个匿名数组对象对它初始化，
    // 该匿名数组对象的类型也是int[3]。
    // 这里倘若使用int b[] = a; 这种方式对b进行初始化，则会引发编译错误
    int b[] = (int[]){ 4, 5, 6 };

    // 输出数组对象b的元素个数
    // 这里我们通过数组对象b的总字节数除以其每个元素占用多少字节来获得其元素个数
    printf("b elments count: %zu\n", sizeof(b) / sizeof(b[0]));

    // 在使用匿名数组给一个数组对象进行初始化时，类型必须匹配。
    // 这里一维数组对象c与匿名数组对象的类型均为int[10]。
    // 如果这里匿名数组的元素个数不是10，则会引发编译报错。
    // 这里，匿名数组对象一共含有10个int类型对象，
    // 其中0号元素值为1；  2号元素值值为-1；  3号元素值为-2，其余元素值均为0
    int c[10] = (int[10]){ [0] = 1, [2] = -1, -2 };

    // 声明一个数组对象d，其类型为int[20]，表示含有20个int对象元素的一维数组
    // 然后没有对d直接做初始化，其每个元素的值都是不确定的
    int d[20];

    // 我们用库函数memcpy先对数组对象d的前10个元素进行赋值。
    // 这里使用数组对象c的所有元素拷贝到数组d的前10个元素中
    // memcpy的第一个参数为要拷贝的目的数据地址；第二个参数为源数据地址；
    // 第三个参数表示要拷贝的总字节数
    memcpy(d, c, sizeof(c));

    // 随后我们分别对数组对象d的12、14、16号元素进行赋值
    d[12] = b[0];
    d[14] = a[0];
    d[16] = a[1] + b[1];

    // 最后，我们用一个匿名数组对象对数组d的17到19号元素进行赋值。
```

```
    // 由于这里的memcpy是一个宏，而不是普通函数，
    // 因此，这里在匿名数组对象外层又用了一个圆括号来避免宏内部处理所引发的问题，
    // 一般情况下，最外层的圆括号是可省的
    memcpy(&d[17], ((int[]){7, 9, 8}), 3 * sizeof(int));
}
```

代码清单7-2详细介绍了匿名一维数组对象的使用方法以及数组的初始化与赋值的约束。这里大家要注意的是，使用库函数memcpy时，必须先引入头文件<string.h>。在语句块作用域声明的一个数组对象，若未经初始化，那么其每个元素的值都是不确定的。如果访问一个数组对象的下标索引超出了数组对象本身的大小，则可能引发程序异常。比如在代码清单7-2中，如果有像“a[3]=4；”这样的赋值语句，那么在执行这条语句时就可能引发程序崩溃，由于数组a一共只有3个元素，有效下标索引值从0～2，而下标索引值3超出了这个范围，所以此时对a[3]进行访问就发生了数组访问越界问题。

7.2　多维数组

7.1节我们介绍了一维数组对象以及数组对象的各种特性，包括如何初始化、如何为其元素赋值等。本节我们将介绍多维数组。多维数组在科学计算上用得尤为多，比如，如果一个一维数组能表示一个向量的话，那么二维数组就能用来表示一个M×N的矩阵，而更多维的数组则能表示更多维度的数据。

形式如int a[2][3]；表示声明了一个二维数组对象a，其类型为int[2][3]，表示含有2个int[3]类型的数组。习惯上，我们很多时候也会将其描述为具有2行3列的int类型元素的数组。不过无论怎么描述，有一点必须清楚，那就是数组对象a的每个元素都是int[3]类型。a[0]表示数组a的0号元素，它是一个具有3个int元素的数组；a[0][0]表示数组a的0号元素（即一个int[3]类型数组）中的0号元素。a[1][2]则表示数组a的1号元素中的2号元素。

二维数组的初始化与元素访问如代码清单7-3所示。

代码清单7-3　二维数组的初始化与元素访问

```
#include <stdio.h>
#include <string.h>

int main(int argc, const char * argv[])
{
    // 声明了一个二维数组对象a，其类型为int[2][3]，
    // 表示具有2个int[3]类型的元素的数组，然后用数组初始化列表进行初始化
    int a[2][3] = {
        // 对数组a的0号元素进行初始化
        // 数组a的0号元素是一个int[3]数组，其每个元素初始化完之后
```

```
        // 分别为: 1, 2, 3
        {1, 2, 3},

        // 对数组a的1号元素进行初始化
        // 数组a的1号元素是一个int[3]数组，其每个元素初始化完之后
        // 分别为: 4, 5, 6
        {4, 5, 6}
    };

    // 这里分别打印数组a的0号元素中的1号元素，
    // 以及数组a的1号元素中的2号元素
    printf("a[0][1] = %d, a[1][2] = %d\n",
           a[0][1], a[1][2]);
    // 输出数组对象a的大小
    // 数组对象a的大小为2 * 3 * sizeof(int)，一共24个字节
    printf("size of a is: %zu\n", sizeof(a));
    // 对一个二维数组的初始化也能扁平化为像对一个一维数组那样进行初始化。
    // 这里数组b的每个元素的值与数组a中的完全一致，
    // 所以，这里的初始化列表与上述a的初始化列表是等效的
    int b[2][3] = {
        1, 2, 3, 4, 5, 6
    };

    // 输出b[0]的大小；由于b[0]为int[3]类型，所以大小为3 * 4 = 12字节
    printf("size of b[0] is: %zu\n", sizeof(b[0]));

    // 在声明一个二维数组对象时，倘若直接对它进行初始化，
    // 那么其元素个数可以不指明，但每个元素的数组类型必须明确指定。
    // 这里，数组对象c的每个元素为int[3]类型，这里的3不能缺省。
    // 最终c的类型为int[4][3]
    int c[][3] = {
        // 对其0号元素进行初始化
        [0] = { 1, 2, 3 },
        [2] = { 4, 5 },     // 1号元素的数组元素均为0，c[2][2]也为0
        // 这里使用指定元素的初始化器
        [3][0] = b[1][2], [3][2] = 7  // c[3][1]的值为0
    };

    printf("c[0][0] = %d, c[1][1] = %d, c[2][2] = %d, c[3][0] = %d\n",
           c[0][0], c[1][1], c[2][2], c[3][0]);

    // 如果在语句块作用域中声明一个二维数组，但没有对它直接初始化，
    // 那么必须指明该数组对象的元素个数，因此这里的2不能省
    int d[2][3];

    // 将数组b的元素拷贝到数组d对象中
    memcpy(d, b, sizeof(d));

    // 这里修改d[1][2]的值
    d[1][2] = c[3][2];

    printf("d[0][0] = %d, d[1][2] = %d\n", d[0][0], d[1][2]);

    // 声明了一个二维数组对象e，并用一个匿名二维数组对象对齐初始化
    // e的类型为int[3][4]
    int e[][4] = (int[][4]){
              { 1, 2, 3 },
        [1] = { 4, 5, 6 },
              { 7, 8, 9 },    // 初始化列表的末尾允许添加一个额外的逗号
    };

    printf("e[0][0] = %d, e[1][1] = %d, e[2][2] = %d\n",
           e[0][0], e[1][1], e[2][2]);
}
```

代码清单7-3展示了二维数组的多种初始化方式，包括直接使用初始化列表以及使用匿名二维数组对象；另外，也介绍了二维数组初始化列表中的指定元素下标的初始化器以及顺序下标的初始化器；随后，又介绍了二维数组元素的拷贝以及对单个元素的访问方式。

二维数组的元素存储方式与一维数组类似。由低地址到高地址，先存放0号元素中的所有元素数据，然后再存放1号元素中的所有元素数据，以此类推。图7-2展示了代码清单7-3中二维数组对象a的元素存储布局。

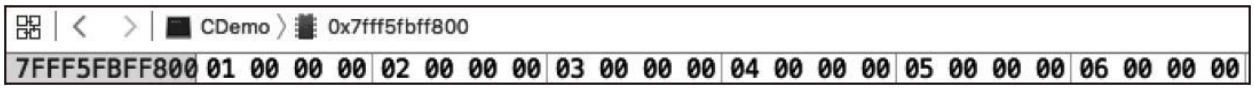


图7-2　二维数组对象a的元素存储布局

图7-2中，二维数组对象a的首地址为0x7fff5fbff800，其0号元素为一个int[3]类型的数组，这个数组中的元素数据先存放在与a相同的首地址处，然后每个元素依次存放在0x7fff5fbff800、0x7fff5fbff804、0x7fff5fbff808处；二维数组对象a的1号元素也是一个int[3]类型的数组，它存放在0x7fff5fbff80c处，并且其每个元素依次存放在地址0x7fff5fbff80c、0x7fff5fbff810、0x7fff5fbff814处。整个数组对象a所占存储空间为24个字节。

除了二维数组，我们还可以声明三维、四维，甚至更高维度的数组，至于最多能支持多少维数组是由C语言实现自己定义的，不过一般支持到十维数组问题都不大。多维数组的构造形式与二维数组类似。比如，int a[2][3][4]；声明了一个三维数组对象a，a的类型为int[2][3][4]，a[0]的类型

为int[3][4]，a[0][0]的类型为int[4]。这也就是说，三维数组对象a中含有两个元素，且每个元素都是一个int[3][4]的二维数组对象。四维数组也类似，比如int b[2][3][4][5]；声明了一个四维数组对象b，其类型为int[2][3][4][5]，共有两个元素，每个元素为类型是int[3][4][5]的三维数组对象。三维、四维数组的元素存储布局也与二维数组的类似，都是按照从低地址到高地址依次存放每个元素的数据内容的。代码清单7-4描述了三维数组的一些基本操作。

代码清单7-4　三维数组的一些基本操作

```
#include <stdio.h>
#include <string.h>

int main(int argc, const char * argv[])
{
    // 声明一个三维数组对象a，其类型为int[2][3][4]
    // 其每个元素的类型为int[3][4]
    int a[2][3][4] = {
        // 0号元素
        {
            { 1, 2, 3, 4 },
            { 5, 6, 7, 8 },
            { 9, 10, 11, 12 },  // 这里可添加一个额外的逗号
        },

        // 1号元素
        {
            { -1, -2, -3, -4 },
            {-5, -6, -7, -8 },
            { -9, -10, -11, -12 }
        },  // 这里的添加一个额外的逗号
    };

    printf("a[0][0][0] = %d, a[1][1][1] = %d\n",
           a[0][0][0], a[1][1][1]);

    // 声明一个三维数组对象b，其类型为int[2][3][4]。
    // 这里可以用与一维数组初始化列表相同的方式为其初始化。
    // 三维数组的元素个数可省，这样根据初始化列表来确定其元素个数，
    // 但其每个元素的类型必须指明，所以这里的[3][4]中的任何值都不能缺省
    int b[][3][4] = {
        // 0号元素
        1, 2, 3, 4,
        5, 6, 7, 8,
        9, 10, 11, 12,

        // 1号元素
        -1, -2, -3, -4,
        -5, -6, -7, -8,
        -9, -10, -11, -12
```

```
    };

    // 此时，数组b的元素内容与数组a的完全相同
    printf("b[0][1][1] = %d, b[1][2][2] = %d\n",
           b[0][1][1], b[1][2][2]);

    // b的大小为2 * 3 * 4 * sizeof(int) = 96，即占用96个字节的存储空间
    printf("b size is: %zu\n", sizeof(b));

    // 三维数组对象的初始化列表也能使用指定元素索引的初始化器
    // 这里未受指定的元素的值都将被初始化为0
    int c[3][2][4] = {
        // 初始化0号元素
        [0] = { {1, 2, 3, 4}, {5, 6, 7, 8} },
        // 初始化1号元素
        [1] = { [0] = { -1, -2, -3, -4 }, [1][2] = -5, -6 },
        // 分别指定2号元素中的若干元素进行初始化
        [2][0] = { 10, 11, 12, 13 }, [2][1][1] = 14, 15
    };

    printf("c[1][1][0] = %d, c[1][1][3] = %d, c[2][1][2] = %d\n",
           c[1][1][0], c[1][1][3], c[2][1][2]);

    // 声明一个二维数组对象d，且不对它进行直接初始化
    int d[3][4];

    // 将a[1]中二维数组对象的元素内容拷贝到二维数组对象d中
    memcpy(d, a[1], sizeof(d));

    printf("d[0][0] = %d, d[1][1] = %d, d[2][2] = %d\n",
           d[0][0], d[1][1], d[2][2]);
}
```

代码清单7-4展示了三维数组的声明、初始化以及对其元素的访问。通过以上这些例子各位应该能很容易地总结出来：二维数组的每个元素是一个一维数组对象，三维数组的每个元素是一个二维数组对象，那么N维数组的每个元素则是一个N-1维数组对象。

7.3　变长数组

我们在7.1节与7.2节中描述的数组都是固定长度的数组。C99标准引入了一种叫可变长度的数组（variable length array），这类数组在声明时，其元素个数不是用常量表达式来指定的，而是通过变量。因此变长数组不能在文件作用域中声明，不能用static存储类说明符来修饰。此外，变长数组以及指向变长数组的指针类型统称为可变修改类型（variably modified type）。当可变修改类型作为sizeof的操作数时，sizeof操作符的结果将在运行时计算，并且操作数所产生的副作用也将会体现出来。随后，变长数组声明之后不能直接对它进行初始化，我们只能通过memcpy等库函数或通过直接访问其元素的方式对该数组中的指定元素进行赋值。正由于变长数组不能直接使用初始化器进行初始化列表，所以不存在匿名变长数组对象，即变长数组的复合字面量。

变长数组在有些场合还是比较实用的。比如说，我们要在一个函数内部定义一个数组，但其大小需要通过函数参数来指定，且元素个数也不会太多。此时，如果随便定义一个比较大的数组也会造成栈空间的不必要的浪费，而使用变长数组则正好能满足需求。

代码清单7-5描述了变长数组的一些使用方式以及特性。

代码清单7-5　变长数组

```
#include <stdio.h>
```

```
int main(int argc, const char * argv[])
{
    int a = 5;

    // 声明了一个含有a个元素的变长数组对象b，类型为int[a]，
    // 这里不能对数组对象b直接做初始化
    int b[a];

    // 我们这里先将a做增1操作，此时a变为6
    a++;

    // 观察数组b的大小，这里sizeof(b)就可能不是在编译时得出的，而是在运行时计算得到。
    // 尽管变量a增加了1，但此时b的大小仍为20个字节，说明一共含有5个int类型的元素。
    // 说明数组b的大小在声明的时候就已经确定而不变了
    printf("b size is: %zu\n", sizeof(b));

    int x = 0;

    // 声明一个指向可变长度为int[a]的指针p，p是一个可变修改类型的指针对象。
    // 指针p指向数组b的地址。这里要注意的是，
    // 由于a的值已经被修改过了（从5增加到了6），
    // 所以，这里(*p)的类型虽然仍然为int[a]，但int[a]显然具有6个int类型的元素
    int (*p)[a] = &b;

    // 这里，(*p)的大小为int[a]的大小，计算得到24个字节
    printf("p[0] size is: %zu\n", sizeof(p[++x]));

    // 这里，x的值为1，说明sizeof操作数中产生了++x的副作用，
    // 这里sizeof的计算是在运行时得到结果的，而不是在编译时
    printf("x = %d\n", x);

    // 如果在变量前加了一个const修饰，则说明它是一个常量
    const int n = 10;

    // 这里一维数组对象d就不是一个变长数组，而是一个定长数组
    // 尽管这里数组元素个数用n来指定，不过d的类型可以看作为int[10]，而不是int[n]
    int d[n] = { 1, 2, 3 };

    // 这里用常量n定义了一个指向数组的指针对象q，
    // 不过，这里的q不是一个可变修改类型，(*q)的类型为int[10]
    int (*q)[n] = &d;

    // 这里的++x不产生任何副作用，因为(*q)不是一个变长数组类型
    printf("q[0] size is: %zu\n", sizeof(q[++x]));

    // x的值仍然为1
    printf("x = %d\n", x);

    // 大家要注意的是，只有当sizeof操作数类型为可变修改类型时，计算才会在运行时执行，
    // 否则的话，仍然在编译时就能获得结果，且不会产生副作用。
    // 比如这里，尽管b是一个变长数组类型int[a]，但b[++x]的类型为int，
    // 不是一个可变修改类型，所以这里的sizeof仍然在编译时得到结果，且++x不产生副作用
    printf("b[0] size is: %zu\n", sizeof(b[++x]));

    // x的值仍然为1
    printf("x = %d\n", x);
}
```

代码清单7-5展示了变长数组的一些基本特性，同时还涉及了本章稍后将会介绍的指针相关的话题。为了说明可变修改类型作为sizeof操作数时的

运行时特性，以及操作数产生的副作用的特性，所以这里先借用一下。此外，还谈到了用const限定符来修饰对象的情况，此时该对象也作为一个常量而不是变量。const限定符将在12.1节中介绍。

7.4　一级指针与对象地址

从本节开始，我们将正式涉及C语言中的一个核心概念——指针（pointer）的话题。在前面一些章节中，为了表达C语言的某些语法特性已经引入了关于指针的一些表达形式。各位在看完整个第7章后，回头去看那些内容将会有更为深刻的理解。现在市面上有不少高级语言都宣称摒弃了“指针”这个概念，但实际上还能看到其中的一些影子。比如Java中，虽说没有直接引入指针语法系统，但它里面涉及到的引用（reference）本质上仍然是一个指针，只不过Java没有引入直接取其内容或取某个对象地址的方式，所以在Java中你也无法取一个引用的地址。而在C语言中，只要是一个对象（对象都拥有自己的存储地址），那么就能取其地址。

7.4.1　地址与指针的基本概念

要解释指针这个概念，我们先说地址（address）。在C语言中，无论我们是在文件作用域声明一个对象，还是在语句块作用域声明一个对象，它们都具有自己的地址。比如，我们在main函数中声明了一个对象：int a=10；，那么对象a就有它自己的地址。我们通过单目操作符&来取对象a的地址——&a。这里的&操作符在C语言标准中也称为地址操作符（address operators），它是一个单目前缀操作符，跟在它后面的表达式作为其操作数。一个对象的地址长度根据不同的系统环境会有所不同。比

如，通常在32位处理器系统模式下，地址的长度为4个字节；在64位处理器系统模式下，地址的长度为8个字节。一个地址表征了用于存放一个对象的数据内容的位置。当然，现代处理器基本都有一套内存管理系统，所以我们在应用程序中拿到的都是虚拟地址；而在一些简单的嵌入式系统下，如果没有引入虚拟地址特性，那么获取到的对象地址就是物理地址。本书暂时不考虑对象所在的是虚拟地址还是物理地址，我们将它们都抽象为“地址”。

有了地址之后，我们如何把对象的地址保存起来以便后续使用呢？这个时候，C语言就引入了一个称为指针的类型类别。一个指向int类型对象的指针就能存放int类型对象的地址。我们在声明一个对象时，在对象标识符前添加*号就能把它声明为一个指针对象。比如，要声明一个指向int类型对象的指针对象p，就用“int*p；”。这里，*可以紧贴int和p，像“int*p；”这也完全没问题。不过在习惯上，我们往往会将*与对象标识符紧贴，而在表示一个类型时，会与类型标识符紧贴。比如：“int*p；sizeof（int*）；”等。指向一个普通非指针对象的指针又被称为一级指针。代码清单7-6描述了一级指针对象的声明、初始化以及使用。

代码清单7-6　一级指针的基本使用

```
#include <stdio.h>

int main(int argc, const char * argv[])
{
    // 声明了一个int类型对象a，并给它初始化为10
    int a = 10;

    // 声明了一个指向int类型的指针对象p，类型为int*；
    // 并用对象a的地址对它初始化，此时，指针p的值就是对象a的地址。
    // 这也被称为“指针p指向对象a”
    int *p = &a;
```

```
    // 一个指针对象可以转为一个无符号整数类型来观察该指针对象的值。
    // 不过我们一般使用uintptr_t类型来表示一个对象地址的值。
    // 这里其实就是输出对象a的地址值
    printf("p value is: 0x%.16tX, size is: %zu\n",
           (uintptr_t)p, sizeof(p));
}
```

代码清单7-6简单明了地阐述了如何声明一个指针对象并为它进行初始化的方法，最后还输出了指针对象p的值。以上代码通过Apple LLVM 8.0编译并在macOS 10.12系统下运行，由于是64位系统环境，所以地址长度为8个字节。图7-3展示了对象a与指针对象p的存储布局。

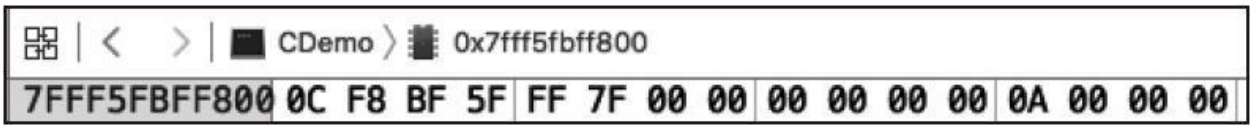


图7-3　对象a与指针对象p的存储布局

图7-3所展示的内容是基于代码清单7-6的运行结果。最后打印输出指针对象p的值（即对象a的地址）为0x00007FFF5FBFF80C，并且对象a的值为10。在图7-3上我们能看到，地址0x00007FFF5FBFF800是指针对象p的地址，而它的内容正是0x00007FFF5FBFF80C（地址从左到右依次为从低地址到高地址），在地址0x00007FFF5FBFF80C处的内容为0A 000000（用十六进制表示），即十进制数10，它就是对象a的值。所以通过这个图我们能清晰地认识到，指针对象本身也有它自己的地址（这里的0x00007FFF5FBFF800就是指针对象p的地址），而指针对象的值就是它所指向的那个对象的地址（这里p的值就是它所指向的对象a的地址0x00007FFF5FBFF80C）。

7.4.2　访问指针对象所指对象的内容

当一个指针对象指向了某一对象之后，我们就可以使用间接操作符（indirection operator）通过指针对象间接地访问它所指向对象的值。间接操作符也是*，属于单目前缀操作符，跟在它后面的表达式作为其操作数。它与用来声明指针类别对象的*不属于同种功能。间接操作符只能作用于指针类型的对象，也就是说间接操作符的操作数必须是一个指针类型的对象。代码清单7-7简单介绍了间接操作符的使用以及效果。

代码清单7-7　间接操作符的使用及效果

```
#include <stdio.h>

int main(int argc, const char * argv[])
{
    // 声明了一个int类型对象a，并给它初始化为10
    int a = 10;

    // 声明了一个指向int类型的指针对象p，类型为int*;
    // 并用对象a的地址对它初始化。其中，&a表达式的类型也是int*
    int *p = &a;

    // 我们通过间接操作符对指针p进行操作，所以这里p就是间接操作符*的操作数，
    // (*p)的类型为int。这里将指针p所指的内容修改为20
    *p = 20;

    // 然后我们可以观察到，对象a的值变为了20
    printf("a = %d\n", a);

    // 声明了一个short类型对象b，并用p所指的对象的值对它进行初始化
    short b = *p;

    printf("b = %d\n", b);
}
```

运行代码清单7-7之后，我们就能神奇地发现，当执行了“*p=20；”这条语句之后，对象a的值被修改为20了。这是怎么发生的呢？这就是间接操作符的神奇之处！我们可以借助图7-3来分析。指针对象p所在地址为

0x00007FFF5FBFF800，它的值就是对象a的地址0x00007FFF5FBFF80C。而当对指针对象p动用了间接操作，所访问的就是以指针对象p的值（0x00007FFF5FBFF80C）作为地址，然后获取该地址中的内容。“*p=20；”这条语句其实做了两步操作：首先获得指针对象p的值；然后以这个值作为地址，把20写入到这个地址中去。这样地址0x00007FFF5FBFF80C（也就是对象a的地址）的内容由原本的10变为了20。而后面的“short b=*p；”也是同样，先获得指针p的值，然后以该值作为地址取该地址中的内容，将该数据赋值给对象b。

因此，所谓的间接操作其实就是以操作数的值作为地址，然后访问该地址的内容。这与取对象地址正好是一个逆操作。

7.4.3 指针对象的其他操作

在C语言中，两个指针对象可以进行大小比较，也就是比较它们所指向对象的地址大小，比如0x00007FFF5FBFF800要小于0x00007FFF5FBFF80C，所以代码清单7-6中的指针对象p的地址要小于对象a的地址。而指向不同类型的指针之间的转换通常来说不能做隐式转换。我们在第5章讲解整数之间的类型转换时谈到，在C语言中高精度与低精度整数相互转换都可以隐式执行，不需要通过投射操作显式给出。指针类型则不然，一个int*类型与short*类型之间就无法进行隐式转换，必须通过投射操作进行显式转换，否则会有编译警告。

本节一开始谈到了对象的地址。一个对象在C语言中通常是一个左值（lvalue），而对于一个右值（rvalue）是无法取它地址的。C语言标准对右值给出的定义是：一个表达式的值。比如，一个sizeof操作符返回的值、一个整数字面量、对一个对象做取地址操作的表达式，还有++、--等表达式。不过当以上这些表达式通过某种形式套上了间接操作符之后，就又能使用&进行取地址操作了。但此时，&取地址操作符的语义其实不是取它表达式的地址，而是用于脱去间接操作的效果。代码清单7-8将会清楚地给大家描述以上这些内容的实际效果。

代码清单7-8　指针与取地址的其他特性

```
#include <stdio.h>
#include <stdbool.h>
#include <stdint.h>

int main(int argc, const char * argv[])
{
    // 声明了两个int32_t类型的对象，分别为a和b
    int32_t a = 10, b = 5;

    // 声明了三个指向int32_t类型的指针对象，分别为p、q和r。
    // 注意，这里用逗号分隔的声明符列表中，如果要指明是指针类型，
    // 那么每个指针对象标识符前都必须添加*。
    // 对象c是一个int32_t类型的对象
    int32_t *p = &a, *q = &b, c = 0, *r = &c;

    // 指针可以进行比较大小以及判断是否相等
    bool e = p > q;
    printf("Is p > q? %d\n", e);

    e = p == r;
    printf("Is p equal to r? %d\n", e);

    // 比较r是否与对象e的地址相同
    printf("Is r equal to &c? %d\n", r == &c);

    // 声明了一个int16_t类型对象s以及一个指向int16_t类型的指针对象t
    int16_t s = 1, *t = &s;

    // 以下这句会出现编译警告，因为int32_t*类型与int16_t*类型不匹配
    t = &a;

    // 以下这句也会出现警告，因为int16_t*类型与int32_t*类型不匹配
    r = &s;

    // 如果要用指针t指向对象a，那么可以使用投射操作来做类型强制转换
    t = (int16_t*)&a;      // 这里不会有编译警告
```

```
    *t = 2048;  // 通过指针t将对象a的值间接地修改为2048

    // 尽管可以通过投射操作将指针p指向对象s，不会出现编译警告，但不建议这么做！
    // 由于对象s是int16_t类型，宽度没有p所指向的int32_t类型的宽度大，
    // 倘若通过指针p写入了一个超出int16_t类型能表示范围的值，那可能引发无法预料的事情
    p = (int32_t*)&s;

    *p = 1024;  // 这里不会有任何问题，因为1024在int16_t类型可表示的范围内

    printf("a = %d, s = %d\n", a, s);

    // 以下这些表达式都是错误的，都将引发编译报错
    &b++; &++b; &1234; &sizeof(b); &&a;

    // 不过一个匿名数组、匿名结构体等表达式可以进行取地址操作
    int32_t (*pa)[3] = &(int32_t[]){1, 2, 3};   // pa是指向一个匿名数组的指针

    struct S { int32_t a, b; };
    struct S *ps = &(struct S){10, 20}; // ps是指向一个匿名结构体的指针

    // 以下表达式都是有效的，并且取地址与间接操作相互抵消
    &*p++; &*++q;   // 效果如同与：p++; ++q;
    p = &*r;        // 效果如同与：p = r;
    printf("*p = %d\n", *p);

    a = 10;

    // 取地址操作符与间接操作符允许多次嵌套，尽管这看上去比较繁复
    b = *&*&*&a;    // 这句等同于：b = a;

    printf("b = %d\n", b);

    // 重新让指针对象p指向对象a
    p = &a;

    // 指针与整数之间也可以通过投射操作来相互转换。
    // 之前已经提到过，用于存放指针值或地址值最佳类型为intptr_t，
    // 或uintptr_t。这里，address的值就是指针p的值，即对象a的地址值。
    // 这里通过投射操作将指向int32_t类型的指针类型转换为uintptr_t整数类型
    uintptr_t address = (uintptr_t)p;

    // 这里声明了一个指向int32_t类型的指针对象p2，
    // 然后直接将address的值对它进行初始化。
    // 这里通过投射操作将address的类型转换为一个指向int32_t的指针类型
    int32_t *p2 = (int32_t*)address;

    printf("*p2 = %d\n", *p2);
}
```

代码清单7-8给大家介绍了声明指针对象时的注意事项，其中大家必须要注意，指针说明符*一般是紧跟对象标识符的，而不是类型名。指针之间可以比较大小以及判别是否相等。取地址操作符的操作数一般不能是一个右值。对于毗邻的间接操作符与取地址符，它们的作用将会相互抵消。一个指针类型可以通过投射操作而转换为一个整型；同样，一个整型也可以

通过投射操作转换为一个指针类型。

以上这些就是关于一个指针对象的基本概念和作用。这里大家要理解，一个指针对象本身也是个对象，它也有地址，而它的值则是它所指向的一个对象的地址。不过，我们可以通过投射操作直接给一个指针对象赋值为一个具体指定的地址值，比如“int*p=（int*）0x00FF0100；”。这里，指针对象p就指向地址0x00FF0100。当然，这么做之前我们必须要清楚0x00FF0100这个地址是否合法，并且当前是否能安全地访问存放在该地址里的数据内容。

7.5　多级指针

上一节介绍了一个一级指针的基本概念和各种特性，并且提到了一个指针对象本身也有地址。那么我们会有疑问，我们应该如何用一个指针对象去指向另一个指针对象的地址呢？此时，我们将引入多级指针这一概念。

如果我们要指向一个一级指针对象的地址，比如：“int*p；”。此时如果要指向p的地址——&p，那么我们就需要一个二级指针对象，比如：“int**q=&p；”。这里，q就是指向一级指针对象p的一个二级指针，其类型为int**，表示指向一个（指向一个int类型的）指针的指针。我们通过代码清单7-9来进一步观察二级指针、一级指针以及普通对象之间的关系。

代码清单7-9　多级指针概貌

```
#include <stdio.h>
#include <stdbool.h>
#include <stdint.h>

int main(int argc, const char * argv[])
{
    // 声明了一个int32_t对象x和对象y
    int32_t x = 10, y = 20;

    // 声明了一个指向int32_t类型的指针对象p和q,
    // 并分别将对象x的地址与对象y的地址对它们初始化
    int32_t *p = &x, *q = &y;

    // 声明了一个指向int32_t指针对象的指针对象pp,
    // 并用对象p的地址对它初始化
    int32_t **pp = &p;

    // 我们可以用间接操作符来取pp的内容
    // 这里，*pp的值就是指针对象p的值，即对象x的地址
    bool b = *pp == &x;
    printf("b = %d\n", b);

    // 修改pp的内容，将它修改为指针对象q的值；
    // 这实际上是就是将指针对象p的值修改成了q的值，即对象y的地址
```

```
    *pp = q;

    printf("p == q? %d\n", p == q);

    // 这里用了两次间接操作符，第一次同样是访问pp所指对象地址的内容，
    // 即指针对象q的值。第二次则是以q的值作为地址，访问其内容。
    // 由于q指向的是y，所以这步操作直接将对象y的值修改为了30
    **pp = 30;
    printf("y = %d\n", y);
}
```

下面我们根据代码清单7-9中的示例来讲解这段代码通过Apple LLVM 8.0构建后，在macOS 10.12.4下运行时的情况。其中主要观察指针对象pp、p、q的内容变化。通过在bool b=*pp==&x；这句打上断点之后，这5个对象的存储位置如图7-4所示。

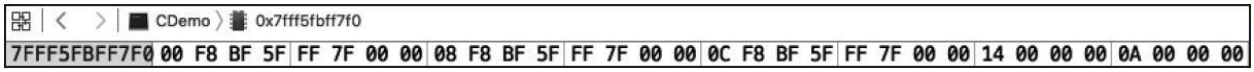


图7-4　执行到第4句时的存储器内容

图7-4中，pp对象的地址为00007F FF 5F BF F7 F0（此图中看最上面，CDemo的右边那一串文字）。我们很容易发现对象y的内容及其地址，只要查找到十六进制数14就是y的内容数值，而它的地址为00007F FF 5F BF F808，对象x的内容只要查找到十六进制数0A即可，其地址为00007F FF 5F BF F80C。然后根据对象x的地址来查找指针对象p的地址，可以看到是00007F FF 5F BF F800；根据对象y的地址找到指针对象q的地址——00007F FF 5F BF F7 F8。这样，我们已经清晰地看到指针对象pp的内容了，正是指针对象p的地址值！当我们对指针对象pp做一次间接操作，那么其实就是访问它所指指针对象的值，即p的值。所以，我们通过*pp与x的地址进行比较能获得两者相等的结果。从类型上讲，pp是int**类型，那么*pp就是int*类型了，**pp则是int类型。

之后，我们对*pp进行修改，将q的值赋值给*pp，这样其实就是将指针对象p的值间接修改成了q的值。我们看图7-5所示的变化。

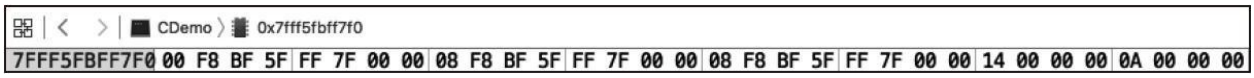


图7-5　*pp修改之后的内容变化

我们看到，图7-5中，对象p地址（00007F FF 5F BF F800）的内容发生了改变，由原来的00007F FF 5F BF F80C（对象x的地址）变为了00007F FF 5F BF F808（对象y的地址）。所以，*pp的操作就是将pp对象的值（00007F FF 5F BF F800，即对象p的地址）作为地址，然后访问其内容（在赋值前得到00007F FF 5F BF F80C，即对象x的地址）。这里的赋值操作其实就是将q的值（即地址00007F FF 5F BF F7 F8所包含的内容）写入到00007F FF 5F BF F800地址中去。这样就间接实现了将指针对象q的值赋值给了指针对象p，使得指针p也指向了对象y的地址。

最后**pp其实就是先对pp做一次间接操作，将pp地址的内容作为地址，访问其内容；然后以该内容，即（*pp）的值作为地址，再访问一次内容。最终获得的就是对象y的值。“**pp=30；”其实就是将30写入到对象y的地址中去，使得此双重间接操作间接地修改了对象y的值。下面我们可以用一段C语言程序更深入地解释一下“**pp=30；”执行的过程。

代码清单7-10　**pp=30；的执行过程分解

```
#include <stdio.h>
#include <stdint.h>

int main(int argc, const char * argv[])
```

```
{
    int32_t a = 10;
    int32_t *p = &a;
    int32_t **pp = &p;

    // 下面是执行**pp = 30; 的分解动作
    // 第一步：先获取pp所指对象的地址，用address1保存
    uintptr_t address1 = (uintptr_t)pp;

    // 第二步：以address1作为地址，再访问其内容
    // 然后将此内容保存为address2。这一步相当于执行了第一次间接操作
    uintptr_t address2 = *(uintptr_t*)address1;

    // 第三步：最后以address2作为地址，将30写入其中。
    // 这一步相当于执行了第二次间接操作
    *(int32_t*)address2 = 30;

    // 最后可以看到a的值被修改为30
    printf("a = %d\n", a);
}
```

代码清单7-10更直观地解释了两次间接操作的整个过程。通过这个例子，大家对于间接操作的理解应该更为深入了吧。最后跟大家再总结一下：在声明一个指针对象时，*号表示该对象为多少级的指针对象；在表达式中，*号作为单目操作符使用时就是间接操作，表示以它的操作数的值作为地址，取该地址中的内容。所取内容的长度则根据类型来判定，如果操作数的类型为int32_t*，那么取int32_t类型长度（即32位）整数；如果操作数的类型为int32_t**类型，那么取int32_t*类型长度（即一个地址长度）的整数；如果操作数的类型是float*，那么取float类型（即32位单精度）浮点数。

我们下面用图7-6来展示代码清单7-9中通过pp二级指针对象间接修改指针对象p的值以及对象y的值的过程。

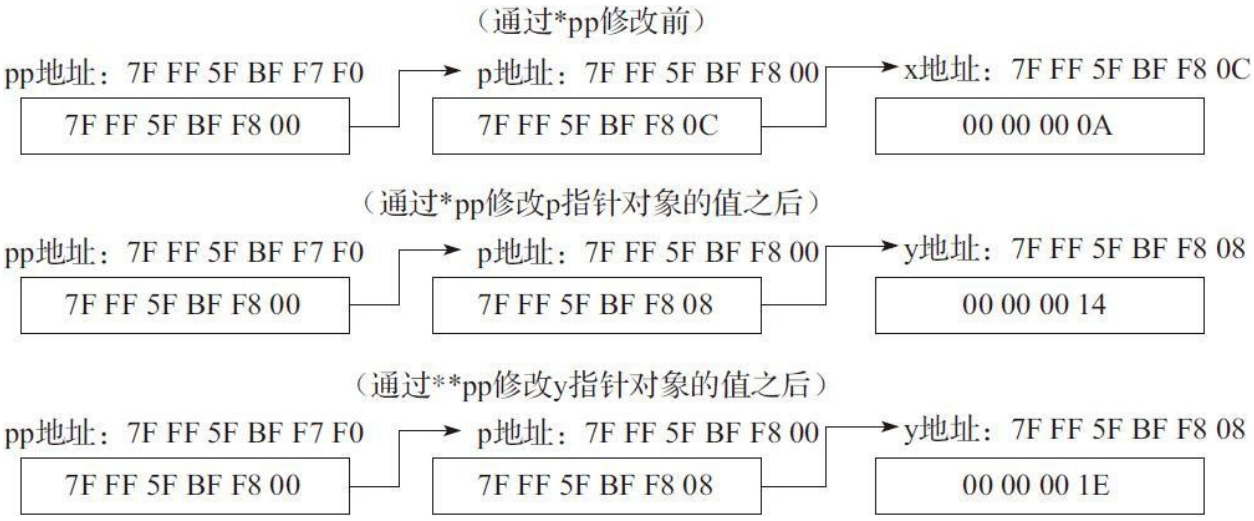


图7-6　通过二级指针对象pp修改p指针对象及y对象值的过程示意图

图7-6中，矩形上方的文字表示当前对象名以及其地址，矩形中的十六进制数串表示当前对象的值，箭头表示当前指针对象指向某一个对象。

7.6　指向用户自定义类型的指针

在上两节中，我们基本以指向整数类型的指针作为例子。其实除了指向整数、浮点数等基本类型的指针外，我们还能定义指向枚举、结构体以及联合体类型的指针。指向枚举类型的指针与一般指向基本类型的指针差不多，只不过类型变为枚举类型而已。对于结构体以及联合体这种带有各自成员对象的类型而言，这里会涉及如何用一个指向结构体和联合体类型的指针对象去访问它们成员的问题。我们在6.2节已经知道，对于一个普通的结构体对象要访问其成员时，我们使用“.”操作符。而如果我们要通过一个指向结构体类型的指针去访问它所指的结构体对象的成员时，我们必须使用“->”成员访问操作符。

下面我们举一些例子来描述指向用户自定义类型的指针的声明以及使用，请见代码清单7-11。

代码清单7-11　指向用户自定义类型的指针

```
#include <stdio.h>

int main(int argc, const char * argv[])
{
    // 定义一个名为TRAFFIC_LIGHT的枚举，并包含三个枚举值
    enum TRAFFIC_LIGHT
    {
        TRAFFIC_LIGHT_RED,
        TRAFFIC_LIGHT_YELLOW,
        TRAFFIC_LIGHT_GREEN
        // 然后，直接声明一个对象light，以及指向该枚举的指针对象pe，
        // 并且直接指向对象light
    } light, *pe = &light;

    // 我们也可以用以下这种更普遍的形式声明一个指向枚举类型的指针对象
    enum TRAFFIC_LIGHT *pe2 = pe;

    // 由于light并没有初始化，所以它的值是不确定的，这里通过指针给它赋值
```

```
    *pe2 = TRAFFIC_LIGHT_YELLOW;

    // 输出1
    printf("light = %d\n", light);

    // 定义一个名为S的结构体
    struct S
    {
        int a;
        float f;
        enum TRAFFIC_LIGHT *pe;

        // 并直接声明一个对象s以及指向该结构体类型的指针对象p
    }s, *p = &s;

    // 利用指针对象p间接为对象s的成员赋值
    p->a = 10;
    p->f = -0.5f;
    p->pe = &light;

    // 当然，我们也可以用以下方式来访问成员，
    // 但是我们需要注意的是，这里必须加一个圆括号，
    // 因为成员访问操作符“.”的优先级高于间接操作符“*”
    (*p).a += 10;
    (*p).f -= 1.0f;

    // 将成员指针pe所指的枚举对象的值修改为TRAFFIC_LIGHT_GREEN，
    // *(*p).pe表达式相当于：*((*p).pe)
    *(*p).pe = TRAFFIC_LIGHT_GREEN;

    // 我们观察对象s的成员，分别为20和-1.5和2
    printf("a = %d, f = %f, light = %d\n", s.a, s.f, light);

    // 以下这句表达式就相当于*(p->pe)，因为成员访问操作符“->”的优先级高于间接操作符“*”
    *p->pe = TRAFFIC_LIGHT_RED;

    // 输出0
    printf("light = %d\n", light);

    // 这里用结构体S声明了一个对象s2，并用*p对它初始化。
    // 这就相当于是用对象s对s2进行初始化
    struct S s2 = *p;

    // 输出20，-1.5，0
    printf("a = %d, f = %f, light = %d\n", s2.a, s2.f, *s2.pe);

    // 这里声明了一个指向结构体S的二级指针对象pp，并用p的地址对它初始化
    struct S **pp = &p;

    // 我们只能通过以下方式来访问pp的成员
    (*pp)->a -= 10;
    (**pp).f += 1.0f;
    *(*pp)->pe = TRAFFIC_LIGHT_YELLOW;

    // 输出10，-0.5，1
    printf("a = %d, f = %f, light = %d\n", s.a, s.f, light);
}
```

代码清单7-11中我们看到了成员访问操作符的优先级要高于间接操作符的优先级，所以我们在刚开始写代码时需要注意这点，以免搞错了逻辑。此外，对于指向结构体或联合体的二级指针对象，在C++中也能直接

使用“->”操作符进行成员访问（比如代码清单7-11中声明的pp，可直接用“pp->a-=10；”，但C语言则不行，这点对于学过C++的朋友需要格外注意。

7.7 指针与数组的关系

我们之前已经提到过，指针对象与数组对象属于两种不同的类别，数组属于聚合类型，而指针属于标量类型。但有趣的是，一个数组对象能合法地隐式转为相应的指针类型。比如，一个int[5]类型能被转为int*类型，使得一个指针对象能指向一个数组的某一元素的地址。而在C11标准中也是明文指出：除了当数组对象标识符作为sizeof、_Alignof、单目&操作符的操作数之外，表示数组类型的表达式会被转换为指向该数组元素类型的指针类型（即[type]类型被转换为type*类型），而该表达式的值则指向该数组对象的初始元素，并且不再是一个左值。所以，我们通常在表达式中引用数组对象标识符的时候，实际使用的是指向该数组首个元素地址的指针，我们可以将数组对象标识符作为间接操作符的操作数。另外，一个指针对象也能通过使用下标操作符来访问它所指向的数组缓存中的相应元素。在C语言标准中，其实是将下标操作翻译为指针与整数的加减法操作。指针与整数之间的加减法操作与一般整数之间的加减法有所不同，假定这里声明了一个指针对象p为type*p，那么（p+1）的值为p+sizeof（type）；（p+2）的值为p+sizeof（type）*2，以此类推。C语言可以将*（p+1）写作为p[1]；将*（p+2）写作为p[2]；而（*p）就相当于*（p+0），可写作为p[0]。

同理，对间接操作的逆操作——取地址操作而言，对于一个一维数组，对其某个元素做取地址操作就相当于从该数组首地址加上指定元素所

在的偏移地址。所以，像（p+1）与&p[1]是等同的。这里大家要注意的是，下标操作符属于后缀操作符，其计算优先级要高于单目取地址操作符&，所以&p[1]相当于&（p[1]）。从类型上分析，p是属于type*类型，（p+1）仍然属于type*类型；而p[1]就是type类型了（由于p[1]相当于*（p+1））。所以，对p[1]再做取地址操作就好比&*（p+1），根据我们在7.4节最后所提到的，一个取地址符如果与间接操作符两者毗邻，则相互抵消，所以可直接得到它与（p+1）完全等同。

代码清单7-12比较详细地描述了指针与数组之间的转换以及指针加减运算的规则与特性。这里各位要注意的是，指针对象只能用加法和减法对它们进行算术计算，而不能用乘除法。当然，两个指针对象之间或一个指针与一个整数之间也不能使用按位逻辑运算。

代码清单7-12　指针与数组的关系及算术运算

```
#include <stdio.h>
#include <stdint.h>

int main(int argc, const char * argv[])
{
    // 声明了一个含有5个int类型元素的数组对象，并对它初始化
    int32_t a[5] = { 1, 2, 3, 4, 5 };

    // 声明了一个指针对象p，并用数组a的首个元素的地址对它初始化
    // 这里相当于：int32_t *p = &a[0];
    int32_t *p = a;

    // 将p的值，即数组a的首地址打印出来
    printf("a address: 0x%.16tX\n", p);

    printf("*p = %d\n", *p);

    // 上述对p的初始化等同于下面这条赋值语句
    // 由于数组索引操作符[]的优先级比取地址操作符&要高，
    // 所以这里p = &a[1]; 又相当于 p = &(a[1]);
    p = &a[1];

    // 这里*p的值就是2，即a[1]的值
    printf("*p = %d\n", *p);

    // 将p的值打印出来
```

```
    printf("a[1] address: 0x%.16tX\n", p);

    // 声明一个address变量，用于记录当前指针对象p的值
    uintptr_t address = (uintptr_t)p;

    p++;                          // 相当于p += 1; 此时，p指向了数组a[2]元素的地址
    printf("p value: 0x%.16tX\n", p);

    printf("*p = %d\n", *p);

    // 上面的p++就好比：
    address += sizeof(*p);      // *p类型为int32_t，所以这里为4

    // 这里address与p的值相等
    printf("Is equal? %d\n", address == (uintptr_t)p);

    // 我们将指针对象p重新指向数组a的起始地址
    p = a;

    p[3] += 10;                 // 相当于*(p + 3) += 10
    *(p + 4) -= 5;              // 相当于p[4] -= 5

    printf("a[3] = %d, a[4] = %d\n", a[3], a[4]);

    /** 除了指针对象与整数对象之间可做加减运算之外，两个指针对象之间也能做减法计算 */
    // 这里让指针p指向数组a的1号元素。这里a + 1与&a[1]是等价的，
    // 说明这里的数组对象标识符a已经作为指向其首个元素地址的指针类型
    p = a + 1;
    int32_t *q = &a[3]; // 这里声明指针对象q，并将它指向数组a的3号元素
    ptrdiff_t diff = q - p;   // 我们计算q与p之间的差值

    // 我们可以观察到q与p之间的差值为2，说明两者跨了2个元素
    printf("diff is: %td\n", diff);

    // 这里p与q两个指针之间的计算相当于：
    diff = (intptr_t)q - (intptr_t)p;
    diff /= sizeof(*p);
    printf("diff is: %td\n", diff);

    // 这里声明了一个数组对象array，其类型为int32_t* [5]，
    // 即array是一个包含5个元素的数组，其每个元素的类型为int32_t*。
    // 这里由于array[3]和array[4]没有被显式初始化，
    // 因此它们被默认初始化为空
    int32_t* array[5] = { p, q, a };

    // 由于当前p指向a[1]地址，所以输出：array[0][0] = 2
    // array[0][0]相当于*array[0]，我们也可以用**array。
    // **array则完全可以体现出，当数组标识符用于表达式时，
    // 它就相当于指向它首个元素地址的指针，因而才能作为间接操作符的操作数
    printf("array[0][0] = %d\n", **array);

    // 这里相当于访问指针对象q所指的对象的内容，输出：*array[1] = 14
    printf("*array[1] = %d\n", *array[1]);

    // 由于array[2]的值即为数组对象a的首地址，
    // 所以我们可以借助array[2]来间接访问数组a的元素。
    // 这里array[2][4]也就相当于a[4]，输出：array[2][4] = 0
    printf("array[2][4] = %d\n", array[2][4]);
    // 这里能输出OK!
    if (array[3] == NULL && array[4] == NULL)
        puts("OK!");
}
```

代码清单7-12清晰地描述了指针与数组之间转换以及指针的算术运

算。不过这里需要再度提醒的是，尽管一个数组类型作为表达式时可以被隐式地转换成相应的指针类型，但数组对象类别与指针对象类别仍然是两个不同的类别，而像int[5]类型与int*类型也属于不同的类型。此外，一个指针类型无法被转换为一个数组类型，即便用投射操作也不行。更确切地说，数组类型无法作为一个投射操作符的操作数，即像“（int[5]）p”这种表达式是非法的。

代码清单7-12也提到了指针与整数对象之间以及两个指针对象之间的算术计算。这里指针所参与的计算只能是加减算术计算，而不能是乘除。当指针对象与一个整数对象参与计算时，指针的值相当于加或减了该整数值的sizeof（*指针对象）倍。当两个指针对象相减时，计算结果是两个指针的差值再除以sizeof（*指针对象）。最后要注意的是，两个类型相兼容的指针对象只能做减法计算，不能做加法计算。

7.8　指向数组的指针

我们上一节讲述了一维数组与一级指针的关系，谈到了一个一维数组对象可以被隐式地转为一个一级指针对象，使得一个一级指针对象能指向该数组的某个元素。但之前我们已经提到过，任何对象都有地址，所以在C语言中都能定义指向任一对象类型的指针，数组也不例外。如果我们定义了一个数组a：“int a[3]；”，那么若要声明一个指向int[3]类型的指针对象p并指向a，则可用此形式：“int（*p）[3]=&a；”。这里，对象p就是指向数组a的指针，其类型为int（*）[3]，表示指向一个int[3]数组类型的指针。这里要注意的是，[]内的3不能省，因为这里的数组对象a是一个定长数组。（*p）的类型则是int[3]类型，所以我们可以通过“（*p）[i]”或“p[0][i]”（上节已经提到过，（*p）相当于p[0]）来访问指针p所指数组的某个元素。

下面，我们将通过代码清单7-13来详细地给大家介绍一下指向数组的指针的特性与用法。

代码清单7-13　指向一维数组的指针

```
#include <stdio.h>

int main(int argc, const char * argv[])
{
    // 声明了一个数组对象a，具有5个元素
    int a[] = { 1, 2, 3, 4, 5};

    // 声明了一个指向int类型的指针对象p，并将它指向数组a的首个元素的地址
    // int *p = a; 就相当于：int *p = &a[0];
    int *p = a;

    // 声明了一个指向int[5]数组类型的指针对象，并用数组a的地址对它初始化
```

```
int (*pa)[5] = &a;

// 与sizeof(p)结果相同，因为两者都是指针类型的对象
printf("size of pa is: %zu\n", sizeof(pa));

// 这里sizeof(*pa)与sizeof(*p)的结果就不同了：
// 因为(*pa)的类型是int[5]，所以sizeof(*pa)就相当于sizeof(int[5])；
// 而(*p)的类型是int，所以sizeof(*p)的结果就相当于sizeof(int)
printf("size of (*pa) = %zu, sizeof (*p) = %zu\n",
       sizeof(*pa), sizeof(*p));

int sum = 0;

for(int i = 0; i < 5; i++)
    sum += (*pa)[i];        // 这里需要注意，这里的圆括号不能省

printf("sum = %d\n", sum);

// 这里声明了一个二维数组b[3][5]，并对其初始化
int b[3][5] = {
    {1, 2, 3, 4, 5},
    {6, 7, 8, 9, 10},
    {11, 12, 13, 14, 15}
};

// 我们用pa指针对象指向b的首个元素的地址。这里相当于：pa = &b[0]
// 一个二维数组其本质是一个数组，该数组的每个元素是一个一维数组。
// b是一个含有3个元素的数组，其中每个元素是一个int[5]的数组对象
pa = b;

pa[0][0]++;
(*(pa + 1))[0]--;           // 相当于pa[1][0]--

uint64_t address = (uint64_t)pa;

pa += 1;                            // 相当于pa = &b[1];

// pa += 1也可看作：
address += sizeof(b[0]);

// b[0]的类型为int[5]
printf("address == pa? %d\n", (uint64_t)pa == address);

printf("b[0][0] = %d, b[2][3] = %d\n", pa[-1][0], pa[1][3]);

int n = a[2];

// 声明了一个变长数组对象v，此时，n为3，所以v具有三个元素
int v[n];

// 对数组对象v的每个元素进行赋值
v[0] = 1; v[1] = 2; v[2] = 3;

// 声明了一个指向变长数组int[n]的指针对象pv
// 并用数组对象v的地址对它初始化
int (*pv)[n] = &v;

n++;

// n++对数组对象v以及指向数组对象v的指针pv均无影响
// v的元素个数仍然是3个；(*pv)的类型int[n]
// n也为3（这里的n不是声明的变量n的值，而是与声明变长数组时所绑定的值）
printf("n = %d\n", n);
printf("size of (*pv) is: %zu\n", sizeof(pv[0]));
printf("size of v is: %zu\n", sizeof(v));

// 声明了一个指向int[n]的数组的指针对象pv2
int (*pv2)[n] = pv;
```

```
    // 这里，(*pv2)的大小即为sizeof(int) * 4，即n为加1后的值
    printf("size of (*pv2) is: %zu\n", sizeof(pv2[0]));

    printf("v[2] = %d\n", pv2[0][2]);
}
```

代码清单7-13详细描述了指向一维数组的指针的特性以及使用方法。这里涉及指向一维数组的指针与一般指针之间的区别，比如int*p和int（*pa）[5]，前者指向了数组对象a的首个元素的地址，而后者则指向了数组a的地址。其实这两个地址值都是相同的，但它们的概念、含义都不相同。

然后我们又通过将指向一维数组的指针对象pa指向了一个二维数组首个元素的地址，这样我们就能对指针与数组之间的关系有进一步的了解了。与一维数组其实类似，a的类型为int[5]，它能被隐式地转换为int*；同样，b的类型为int[3][5]，那么它能被隐式地转换为int（*）[5]。这里大家能进一步摸透数组与指针之间的关系，其实一个数组对象能隐含地表示其首个元素的地址。像数组a，当其标识符用于一般的表达式时，就相当于&a[0]；数组b也是如此，当其标识符用作=的右操作数时，就相当于&b[0]。由于b[0]的类型是int[5]，所以&b[0]自然而然地就被表达为int（*）[5]。

最后一部分我们描述了指向变长数组的指针的特性。关于变长数组，我们在7.3节已经有了比较详细的介绍，这里不再赘述。

以上描述的是指向一维数组的指针，那么是否存在指向更多维度数组的指针呢？答案当然是肯定的。C语言的类型系统是相当完备的，既然存

在指向一维数组的指针，那么肯定就会存在指向更多维的数组指针。上面已经提到了，声明一个指向含有3个元素的一维数组的指针p的形式为：“int（*p）[3]；”。要声明指向一个int[2][3]二维数组的指针q的形式为：“int（*q）[2][3]；”，q的类型为“int（*）[2][3]”，则这里[2][3]这两个方括号里的数都不能省。（*q）的类型则为int[2][3]，所以如果我们要查询sizeof（*q），那么得到的大小为“sizeof（int）*3*2”。代码清单7-14简单介绍了指向二维数组的方式以及用法。

代码清单7-14　指向二维数组的指针

```
#include <stdio.h>

int main(int argc, const char * argv[])
{
    // 声明了一个二维数组a，具有2个元素，
    // 其中每个元素为一个int[3]的数组
    int a[2][3] = {
        { 1, 2, 3 },
        { 4, 5, 6 }
    };

    // 声明了一个指向二维数组int[2][3]的指针对象p，
    // 并用数组a的地址对它初始化
    int (*p)[2][3] = &a;

    // 修改二维数组元素a[0][0]的值
    (*p)[0][0] += 10;
    printf("a[0][0] = %d\n", a[0][0]);

    // 修改二维数组a[1][2]的值
    p[0][1][2] += 100;
    printf("a[1][2] = %d\n", a[1][2]);

    // (*p)的大小为2 * 3 * sizeof(int)
    printf("size of (*p) = %zu\n", sizeof(*p));
}
```

代码清单7-14简单地描述了指向二维数组的指针的使用以及特性，这些可依次类推到三维，甚至更高维的数组指针。此外，指向二维数组的指针能直接指向一个三维数组的首个元素的地址，这与指向一维数组的指针

可直接指向一个二维数组的首个元素的地址的特性类似。

7.9 void类型、指向void类型的指针与空指针

我们之前讲到的类型都是有其具体意义的类型。在C语言中还有一种类型表示“无”或“空”，用void关键字来表示。void一般用于函数返回类型以及表示空的形参列表，也可作为表达式的类型。具有void类型的表达式称为void表达式，表示该表达式不返回任何值，也不具有任何有意义的类型。在C语言中，大部分表达式都具有某个有意义的类型，不过我们可以通过投射操作，将某个表达式强制转为void表达式，比如：（void）0，（void）（a+1），（void）（a=3）等都是属于void表达式。void表达式除了可用作为逗号操作符的操作数以及三目条件操作符的“？”后面和“：”后面的操作数之外，一般不可用作为其他操作符的操作数。此外，void表达式也只能作为对void类型投射操作的操作数，而不能作为对其他类型投射的操作数。像以下表达式都是合法的：

```
// 这里，对于(void)(void)(1 + 2)表达式，(void)(1 + 2)这个子表达式就作为
// void投射操作的操作数。
(void)0, (void)1; (void)(void)(1 + 2); 1 > 2 ? (void)0 : (void)1;
```

在C语言中，一个指针可以指向void类型，即该指针对象的类型为void*。如果一个指针对象是指向void类型的指针，那么该指针可被隐式转换为指向任一对象类型的指针。而指向任一对象的指针也都能被隐式转为指向void类型的指针。在5.6节最后描述C11标准对投射操作符的约束时已经提到：“涉及指针类型的转换，除了某些允许指针类型隐式转换的情况，应该显式使用投射操作”。这里，所谓的“某些允许指针类型隐式转换的情

况”就是指void*类型的情况。因而void*指针类型往往在C语言社区中也被戏称为“万用指针类型”（universal pointer）。

代码清单7-15举了一些使用指向void指针的例子。

代码清单7-15　指向void指针的一些例子

```
#include <stdio.h>

int main(int argc, const char * argv[])
{
    int a = 10;

    // 这里声明了一个指向void类型的指针对象p,
    // 并用对象a的地址对它初始化
    void *p = &a;

    // 这里声明了一个指向int的指针对象q,
    // 并直接用p对它初始化。这里不需要使用投射操作做类型转换
    int *q = p;

    *q += 10;
    printf("a = %d\n", a);

    // 将指向void类型的指针指向一个临时数组对象的地址
    p = &(int[]){ 1, 2, 3 };

    // 这里直接将p转换为指向int[3]数组的指针类型,
    // 然后对该数组的索引为1的元素值做加2修改
    ((int(*)[3])p)[0][1] += 2;

    // 这里声明一个指向int[3]数组的指针t,
    // 并且直接用p进行初始化，这里也不需要任何投射操作做类型转换
    int (*t)[3] = p;

    // 这里(*t)的类型为int[3]，可直接将它赋值给指向int的指针q。
    // int[3]到int*是可隐式转换的
    q = *t;
    printf("q[1] = %d\n", q[1]);
}
```

在C语言中我们通常使用一个空指针表示一个无效的、或未经初始化的指针对象，空指针使用宏NULL来表示。C语言中的NULL往往会被定义为（void*）0，即类型为指向void指针、值为0的整数常量。通常，我们不能对一个空指针所指的内容进行读写。举一个简单的例

子：“int*p=NULL；”，这里p就被初始化为一个空指针。

一般来说，我们在函数里声明一个局部指针对象时，如果未对它做有意义的初始化，那么可以先将它指向空，否则该局部指针对象的值是不确定的，这种情况在C语言社区中也将它戏称为“野指针”（wild pointer），即不受控的指针。如果用空（NULL）来标识该指针为无效指针，那么显然更容易做判断处理。而如果一个指针对象指向了一块动态分配的内存空间，当此内存空间被释放之后，那么我们也可以将该指针指向空，表示它已经失效了，否则该指针对象也会成为“野指针”。这对声明在文件的作用域并且被多个模块所访问的指针对象来说尤为有用。

7.10　字符数组与字符串字面量

在C语言中，字符串字面量是一个比较特殊的类型。在C99标准中只有默认的ASCII字符集的字符串字面量以及系统环境定义的宽字符串字面量两种，而C11标准还引入了UTF-8字符串、UTF-16字符串以及UTF-32字符串。

一个字符串字面量会自动添加一个'\0'字符，用来表示该字符串的结束符。该结束符可被库函数strlen等使用，用于获取当前字符串的长度（即字符个数通常也实现为字节个数）。比如，"abc"是含有3个ASCII码字符的一个字符串，但它的类型是char[4]，即实质上是一个含有4个ASCII码字符的数组（最后一个字符为'\0'）。u"abc"是含有3个UTF-16编码字符的一个字符串，它的类型为char16_t[4]的数组（最后一个字符为u'\0'）。

一个字符串字面量的类型虽然是一个数组类型，但相比于数组的语法体系却还有一些例外特性。首先，尽管字符串字面量的类型是一个字符数组类型，且不是常量，但C语言标准明确指出，对字符串字面量中的字符进行修改是一个未定义的行为。此外，字符串字面量可直接给一个字符数组进行初始化。代码清单7-16列出了这些特性。

代码清单7-16　字符串字面量的一些特性

```
#include <stdio.h>
#include <string.h>
#include <wchar.h>
```

```
#include "uchar.h"
int main(int argc, const char * argv[])
{
    // "abc"是一个兼容ASCII码的系统编码形式的字符串
    const char *asciiString = "Hello, world";

    // u8"你好，世界"是一个UTF-8编码的字符串
    const char *utf8String = u8"你好，世界";

    // u"你好，世界"是一个UTF-16编码的字符串
    const char16_t *utf16String = u"你好，世界";

    // U"你好，世界"是一个UTF-32编码的字符串
    const char32_t *utf32String = U"你好，世界";

    // L"你好，世界"是一个系统编码形式的宽字符串
    const wchar_t *wideString = L"你好，世界";

    printf("asciiString: %s\n", asciiString);
    printf("utf8String: %s\n", utf8String);

    // 可以直接用一个字符串字面量给一个字符数组对象进行初始化
    // 此时，s的类型为char[4]
    char s[] = "abc";

    // 这里要注意的是，数组对象s所在的地址并不与字符串"abc"所在的地址相同
    printf("The address of s is: %.16tX\n", (uintptr_t)s);
    printf("The address of string is: %.16tX\n", (uintptr_t)"abc");

    printf("size of s is: %zu\n", sizeof(s));

    // 输出结果为6，除了hello这5个字符外最后还有一个'\0'结束符，一共6个字符
    // 因而"hello"的类型为char[6]类型
    printf("The size of string is: %zu\n", sizeof("hello"));

    // 当然，我们也可以用一个字符数组字面量（即匿名数组）对一个数组对象进行初始化
    char a[] = (char[]){ 'a', 'b', 'c', '\0' };

    // 我们调用strcmp库函数来比较两个字符数组的内容是否相同。
    // 如果相同返回0，否则返回一个负数说明s的内容小于a的内容；
    // 返回一个正数说明s的内容大于a的内容
    int equal = strcmp(s, a);
    printf("Result is: %d\n", equal);

    // 这里字符数组对象b尽管含有6个字符元素，但字符串长度仍然为3，
    // 因为索引3元素为'\0'，表示字符串结束符
    char b[] = { 'a', 'b', 'c', '\0', 'd', 'e' };

    printf("array b size: %zu\n", sizeof(b));
    printf("b string length: %lu\n", strlen(b));

    // 比较字符数组b与字符串s是否相同
    equal = strcmp(b, s);
    printf("equal is: %d\n", equal);

    char b[] = s;  这句是非法的！不能将一个数组对象给另一个数组对象进行初始化
}
```

标准库头文件<string.h>中列出了许多丰富的字符串操作函数，比如字符串比较、获取字符串长度、字符串拷贝等。代码清单7-16展示了C语言中字符串字面量的特性以及如何用字符数组表示一个字符串的方式。如果各

位在编译环境中没有<uchar.h>头文件，那么可以使用5.1.7节中的代码清单5-8作为头文件进行包含。

C语言中，字符串字面量另一个有趣特性就是可进行拼接。相邻几个字符串字面量可拼接在一起，形成一个字符串字面量，比如：字符串"abc""def"即可表示一个完整的字符串"abcdef"。两个字符串字面量之间可以有零个或多个空白符分隔。但这里要注意的是，进行拼接的几个字符串的类型必须一致，比如几个UTF-8编码的字符串可进行拼接，几个宽字符串字面量也可进行拼接，但一个UTF-16字符串与一个宽字符串字面量之间就无法进行拼接。在字符串字面量进行拼接时，无前缀的字符串字面量可被任意拼接，这将形成在所拼接的字符串字面量中含有某一前缀的字符串类型。代码清单7-17将介绍字符串拼接特性。

代码清单7-17　字符串字面量的拼接特性

```
#include <stdio.h>
#include <string.h>
#include <wchar.h>

#include <uchar.h>

int main(int argc, const char * argv[])
{
    // 这里，"a"与"b"之间用一个空格分隔，"b"与u"c"之间没有任何空白符。
    // 在所有字符串字面量中由于第三个字面量u"c"具有前缀u,
    // 所以整个字符串为一个UTF-16编码的字符串u"abc"
    const char16_t *utf16String = "a" "b"u"c";

    // 这里与上述字符串字面量等价
    utf16String = u"a""b" u"c";

    // 习惯上，一般我们可以这么写
    utf16String = u"a" "b" "c";

    // 字符串字面量也可以用换行分隔，换行也属于空白符
    const char *s = "d" "e"
                    "f";
    printf("s = %s\n", s);

    // 以下这种字符串拼接方式是错误的。u前缀字面量与U前缀的字面量不能拼接在一起
    const char16_t *errorString = u"a" U"b" "c";
```

```
}
```

7.11　完整与不完整类型

在C语言中，类型被划分为对象类型和函数类型。在一个翻译单元内，一个对象类型在多个位置可能是不完整的，也可能是完整的。所谓不完整类型就是指缺乏足够的信息去判定用该类型所声明对象的大小。

在C语言中，一个void表达式表示不存在值，此类型是一个不完整类型，并且不能对它隐式或显式转换为其他类型。如果具有其他类型表达式的计算结果为一个void表达式，那么其值和所表示的标识符都会被丢弃。比如，（void）（2+3）；这个表达式就是一个void表达式，它不能给任一类型的对象赋值。

只含有一个枚举类型、结构体、联合体标识符的声明也是不完整类型，因为它们没有任何信息来描述自己。

一个指向不完整类型的指针类型是一个完整类型，因为指针类型对象的大小明确，所以指向void类型的指针也属于完整类型。代码清单7-18举了一些例子进行说明。

代码清单7-18　完整与不完整类型

```
#include <stdio.h>

// 这里声明了一个名为MyStruct的结构体类型，
// 由于缺乏对该类型的明确定义，因此在当前翻译单元的这个位置，它是一个不完整类型
struct MyStruct;

static void foo(void)
{
    // 这里对象p是一个指向MyStruct结构体的指针，
```

```
    // struct MyStruct* 是一个完整类型
    struct MyStruct *p = NULL;

    // 这里如果写：struct MyStruct s;将会报错。
    // 一个不完整类型不能用来声明该类型的一个对象。

    printf("The size is: %zu\n", sizeof(p));

    // 这里表达式(void)(2 + 3)是一个void表达式，
    // 因此可以加在return语句后面返回。如果这里缺省(void)，将会编译报错
    return (void)(2 + 3);
}

// 由于这里对结构体MyStruct进行了明确的定义，所以在当前翻译单元的这个位置，
// 它现在是一个完整类型了。
struct MyStruct
{
    int i;
    float f;

    // 由于在此刻，MyStruct尚未定义完成，因此它在此位置仍然是不完整类型。
    // 如果要定义：struct MyStruct s; 编译器将会报错
    // 但可定义指向MyStruct结构体类型的指针对象作为成员
    struct MyStruct *p;

    // 如果一个结构体至少含有一个命名成员对象，那么最后可跟一个不指定大小的数组。
    // 最后不指定大小的数组对象也是一个不完整类型，但数组元素的类型必须是完整的，
    // 所以这里array[i]的类型是void*，它是一个完整类型
    // 成员array不占MyStruct结构体类型的大小
    void* array[];
};

int main(int argc, const char * argv[])
{
    foo();

    // 一个完整类型可以声明该类型的一个对象
    struct MyStruct s = { 10, 0.5f };
    printf("The value is: %f\n", s.i + s.f);

    // 在64位系统下，结构体MyStruct类型的大小为16个字节
    printf("size is: %zu\n", sizeof(s));
}
```

由于一个不完整类型无法确定其大小，因此它不能作为sizeof、_Alignof的操作数。

7.12　灵活的数组成员

在C语言中，在至少含有一个命名成员对象的结构体中，其最后一个成员可以是一个不完整数组类型，即不指定数组长度（如7.11节的代码清单7-18中定义的结构体所示），该成员被称为灵活的数组成员。在大部分情况下，灵活的数组成员是被忽略的，而结构体的大小也不把灵活的数组成员给算进去，但是字节填充会受到灵活的数组成员类型的影响。当我们用“.”或“->”成员访问操作符去访问灵活的数组成员时，该成员的偏移量往往就是结构体自身大小。

代码清单7-19给出了灵活的数组成员的一些特性以及常用实践方式。

代码清单7-19　灵活的数组成员

```
#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>

int main(int argc, const char * argv[])
{
    struct Test1
    {
        int8_t b;

        // 这里d就是灵活的数组成员，这里如果不声明此成员，
        // 那么结构体Test1的大小仅为1个字节；但声明了此成员之后，
        // 为了对该成员访问的需要，而对整个Test1类型做了字节填充，到8个字节
        double d[];
    };

    // 获得成员d所在Test结构体中偏移量
    size_t offset = offsetof(struct Test1, d);

    // 这里偏移量与结构体Test的大小都一样，均为8个字节
    printf("offset is: %zu\n", offset);
    printf("size is: %zu\n", sizeof(struct Test1));

    struct Test2
    {
        int a;
```

```
        // 这里array是灵活的数组成员。
        // 由于array所在的偏移位置正好是与int对齐的，
        // 因此整个结构体Test2无需再做字节填充
        int array[];
    };

    printf("size of struct Test2: %zu\n", sizeof(struct Test2));

    // 这里声明了一个Test2结构体数组对象ts，它共有三个元素。
    // 这里相当于：ts[0].a = 10; ts[1].a = 20; ts[2].a = 30;
    struct Test2 ts[] = { { 10 }, { 20 }, { 30 } };

    // 这里我们可以很清晰地观察到，
    // 结构体Test2中的array成员所在偏移也正好是Test2的大小，
    // 这样当我们访问ts数组的第一个元素的array成员时，
    // 该array正好指向第二个元素的起始地址。
    // 这就好比：ts[0].array = &ts[1]; ts[1].array = &ts[2];
    int sum = ts[0].array[0] + ts[1].array[0];

    // 这里sum的结果为 20 + 30 = 50
    printf("sum = %d\n", sum);

    // 灵活的数组成员还有一种常用使用方式是对整个结构体类型做动态存储分配，
    // 这样，我们可以根据当前上下文来确定灵活的数组成员到底给它分配多大的存储空间。
    // 比如这里给pt所指向的Test2结构体对象中的array成员，
    // 分配了由变量sum所指定的数组元素个数
    struct Test2 *pt = malloc(sizeof(*pt) + sizeof(double[sum]));

    // 这里用到了在第8章将会讲到的for循环语句
    for(int i = 0; i < sum; i++)
        pt->array[i] = i + 1.0;

    double result = pt->a;

    for(int i = 0; i < sum; i++)
        result += pt->array[i];

    printf("result is: %f\n", result);
}
```

代码清单7-19中用了C语言标准库的malloc函数，该函数从存储堆空间动态分配指定的存储空间大小，其原型包含在了<stdlib.h>标准库头文件中。

7.13　本章小结

本章主要讲述了C语言中的数组与指针相关话题。本章也是相对比较有难度的章节，希望C语言初学者能反复阅读、实践。数组与指针类型可进行各种组合、变化，这也体现出C语言类型系统的强大与灵活，但又不失简洁性。如果能把本章内容掌握好，那么对于C语言的一大主要问题也就解决了。

至此，关于C语言中的类型就全都讲解完了。第8章与第9章将分别介绍C语言中的控制流语句以及函数。

第8章　C语言的控制流语句

第5～7章主要讲解了C语言中的各种对象类型以及各种字面量，随之讲解了对象的声明以及一些简单的算术逻辑计算、赋值语句的写法。本章将为大家介绍C语言的控制流语句，用来表达语句的分支和循环执行。我们将先介绍逗号表达式与条件表达式，这两个表达式具有魔法般的表现力，往往能把一些语句的表达进行简化。随后，我们会介绍C语言中的选择语句（selection statement）if-else以及switch-case，C语言中的选择语句就是我们通常所谓的分支语句。紧接着会介绍while、do-while以及for迭代语句来表示循环，执行一次循环又可称为执行一次迭代。最后我们会介绍C语言中的goto跳转语句，这是最能体现处理器控制流执行的表达式，但在使用时需要注意一些事项。

8.1　逗号表达式

C语言支持逗号表达式，用于将若干表达式按指定次序执行，最终返回最后一个表达式的结果。比如，像int a=（0，2，3）；这条语句执行后，a的值为3。逗号操作符是一个双目操作符，其左边的操作数称为左操作数，其右边的操作数则称为右操作数。比如表达式（0，2）中，0是左操作数，2是右操作数。逗号表达式的左操作数与右操作数之间有一个顺序点，因此右操作数能确保在左操作数所产生的所有副作用完成后再进行执行。这个特点就与分号的作用类似。不过之所以要引入逗号表达式，其主要目的是将最后一个表达式的计算结果给返回出来；而分号则表示整个表达式的终结，并不具备这个特性。

在使用逗号表达式的时候还需注意，由于逗号操作符的优先级是最低的，因此逗号表达式往往会用圆括号操作符括起来，使得它们作为一个整体的基本表达式不会被其他操作符分隔开。代码清单8-1列举了逗号表达式的特性以及需要注意的事项。

代码清单8-1　逗号表达式

```
#include <stdio.h>

int main(int argc, const char * argv[])
{
    // =前的int a, b表示对象a和b的声明符，它们之间的逗号表示对标识符的分隔，
    // 而不是作为逗号操作符使用，因为逗号操作符的功能仅用于表达式。
    // =后面的(2, 3, 4)则是一个逗号表达式，表示给对象b赋值为4
    // 这条语句相当于：int a; 2; 3; int b = 4;
    int a, b = (2, 3, 4);

    // 这里相当于：a = 2; b = 3; 4;
    a = 2, b = 3, 4;
```

```
    printf("a = %d, b = %d\n", a, b);

    // 这里相当于b = 3; a = 4;，因为赋值操作符的优先级要大于逗号操作符
    a = (b = 3, 4);
    printf("a = %d, b = %d\n", a, b);

    // 尽管这里有后缀++操作符，但之前已经提到过，逗号操作符的左右操作数之间存在顺序点
    // 因此右操作数执行前需要处理完左操作数所产生的所有副作用
    // 因而，这里相当于：b++; int tmp = b += 4; a = tmp;
    // a和b的值均为8
    a = (b++, b += 4);
    printf("a = %d, b = %d\n", a, b);

    // 这里就相当于：b = a++; b = b - 1;
    b = (b = a++, b - 1);
    printf("a = %d, b = %d\n", a, b);
}
```

在使用逗号表达式时还需要注意一点，当逗号作为特定上下文的特定分隔语义使用时，它就不能做逗号表达式的操作符使用了，此时可以用圆括号将整个逗号表达式括起来作为一个整体的基本表达式，这样可以将作为操作符的逗号与表示分隔用的逗号进行区分。比如：printf（“a+b=%d，a-b=%d\n”，（b++，a+b），a-b）；里，表达式（b++，a+b）的圆括号就不能省，因为其圆括号外的逗号用于分隔参数。

8.2　条件表达式

在C语言中有一种非常便捷的用于执行条件分支的表达式，即条件表达式。它是一个三目表达式，由标点符号？与：共同构成。其表达式的形式为：

```
布尔表达式 ？ 表达式1 ： 表达式2
```

这组表达式的执行逻辑为：如果布尔表达式的值为“真”，那么执行表达式1；否则执行表达式2。正由于这里的？与：共同构成了一个三目操作符，所以其操作数也就有3个——“？”之前的布尔表达式、“：”之前的表达式1，以及“：”之后的表达式2。三目操作符的优先级比较低，它仅比赋值操作符要高一级而已。

代码清单8-2展示了表达式的用法以及一些需要注意的地方。

代码清单8-2　条件表达式

```
#include <stdio.h>

int main(int argc, const char * argv[])
{
    int a = 10;
    int b;

    // 由于条件表达式的?与：操作符优先级大于赋值操作符，
    // 所以后面的(b = -1)这句子表达式需要加上圆括号，
    // 否则a > 0? b = 1 : b = -1相当于：(a > 0? b = 1 : b) = -1;
    // 这样就会引发编译报错
    // 这里a > 0的表达式为真，所以执行b = 1子表达式
    a > 0? b = 1 : (b = -1);
    printf("b = %d\n", b);

    // 这里a < 0的结果为假，所以执行a - 10的表达式
    // 因此，整个结果表达式为：b = a - 10;
    b = a < 0? a + 10 : a - 10;
```

```
    printf("b = %d\n", b);
}
```

8.3 if-else语句

if-else语句在C语言中属于一种选择语句，用来表达不同的执行分支。if语句的形式为：

```
if ( 表达式 ) 语句;
```

表示当表达式为真时执行语句。

如果我们要表达：当表达式为假时执行另一条语句，那么可以添加else。其形式为：

```
if ( 表达式 ) 语句1
else 语句2;
```

表示当表达式为真时执行语句1，否则执行语句2。

C语言中的if-else语句与我们在工作中遇到的分支流程图很类似。比如图8-1所示的场景。

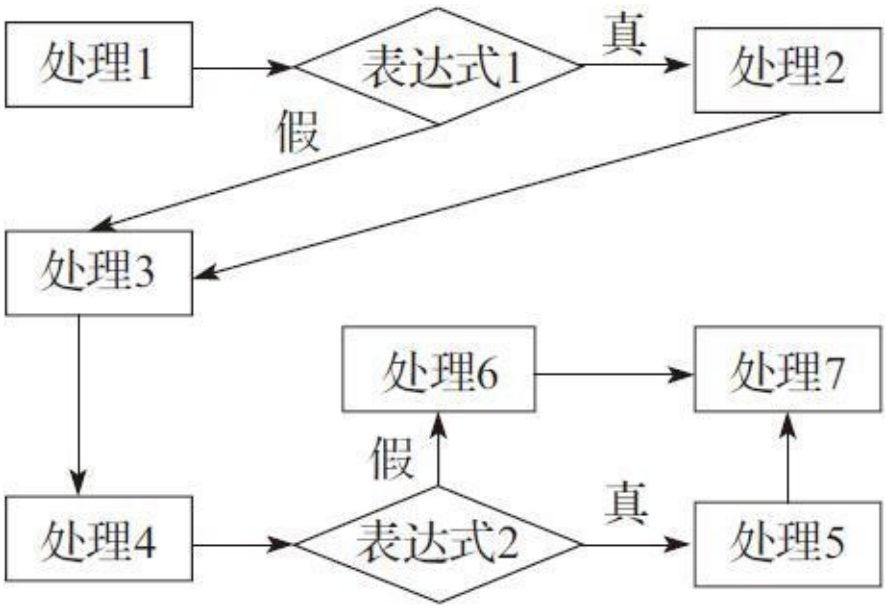


图8-1　if-else逻辑流程图

图8-1中，处理1、处理2到处理3的分支逻辑就是if逻辑。表示：

```
if(表达式1为真) { 执行处理2 } 执行处理3
```

说明只有当表达式为真时执行处理2的逻辑，但无论表达式是否为真，最终都会执行到处理3的逻辑上。

而从处理4到处理7展现的是if-esle的逻辑。表示：

```
if(表达式2为真) { 执行处理5 } else { 执行处理6 } 执行处理7
```

说明当表达式2为真时，执行的是处理5的逻辑，如果为假则执行处理6的逻辑，最终都执行到处理7的逻辑。

代码清单8-3则展示了if-else语句的具体用法以及需要注意的一些事

项。

代码清单8-3　if-else语句

```
#include <stdio.h>

int main(int argc, const char * argv[])
{
    int a = 10, b = 0, c = 1;

    // 这里的if语句表明，当a > 0时执行b = -1;
    // if语句后面可以跟一条语句
    if(a > 0)
        b = -1;

    b++;

    // 这里需要注意的是，下面的b++;语句无论if中表达式的条件是否为真都会执行。
    // 因为if语句后只能跟一条语句（这里的b=-1;就是一条语句，而b++;则属于另一条），
    // 因此这里的b++与上述的b++属于相同位置，即在if语句之外。
    if(a > 20)
        b = -1; b++;

    // 如果我们要在一条if语句内包含多条语句，那么可以用语句块
    if(a > 0)
    {
        // 当a > 0为真时，执行这两条语句
        b++;
        c++;
    }

    // 这里使用了if-else语句，说明当a < 0为真时执行b--;
    // 如果为假则执行c++;
    if(a < 0)
        b--;
    else                // 这里else的用法与if一样，后面可跟一条语句
        c++;

    if(c > 0)
    {
        a++;
        b++;
    }
    else
    {
        c = 0;
        a = 1;
        b = 2;
    }

    printf("a = %d, b = %d, c = %d\n", a, b, c);

    // if语句本身也属于一条语句，因此
    if(a > 0)
        if(b > 0)
            if(c > 0)
                puts("OK!");

    // 相当于：
    if(a > 0)
    {
        if(b > 0)
```

```
        {
            if(c > 0)
            {
                puts("OK!");
            }
        }
    }

    // if与else后面可直接跟一个空语句，这么一来，if或else则不执行任何操作。
    if(a > 0);        // 这是一条跟在if()后面的空语句
    if(a > 0)
        a++;
    else;          // 这是一条跟在else后面的空语句
}
```

代码清单8-3中涉及C语言中的语句（statement）语法问题，这里先简单提一下。C语言一共有以下几种语句：标签语句、复合语句、表达式语句、选择语句、迭代语句、跳转语句。这里像if（a>0）、b=-1；b++；等都属于一条语句。if（a>0）是一条选择语句；b=-1；则是表达式语句；而复合语句是由{}包围的一个语句块，其中{}中可包含多条语句。这里除了表达式语句，其他语句都不具有返回类型，类似于一条void表达式语句。

8.4 switch-case语句

switch-case语句是C语言中第二种选择语句，switch-case的一般表达形式为：

```
switch ( 表达式 )
{
    case 整数常量表达式1:
        语句1;
        break;
    case 整数常量表达式2:
        语句2;
        break;
    default:
        语句3;
        break;
}
```

这里，switch（）选择语句中的表达式应该具有一个整数类型（包括字符类型和布尔类型），而不能是其他类型，后面走哪条case分支就根据该表达式的值进行选择。一个switch语句块{}中可包含多条case标签语句。每个case标签语句的表达式应该是一条整数常量表达式，并且在一个switch语句块中不应该存在包含相同整数常量的任意两条case标签语句。如果switch中的表达式的值与某一个case标签语句中的常量值相同，那么就执行该case标签下的语句。break跳转语句用于跳出当前整个switch语句块而不继续往下执行。在switch语句块中最多允许存在一条default标签语句，表示当switch选择语句中的表达式的值没有与任一case标签语句后的常量表达式的值匹配时，则执行default标签后的语句。倘若switch表达式的值没有与任一case标签的常量表达式的值匹配，且不存在default标签，则不执行整个switch语句块中的内容。

在case或default标签下的语句中不能声明一个临时对象，由于它在整个switch语句块的作用域内，因此它可能对其他case分支可见，但它执行到该分支时，其值可能还没有被正确初始化。因此，如果我们在某一case分支下要表达一些比较复杂的语句，则必须使用复合语句，即加上{}，在{}中可声明临时对象。

代码清单8-4将描述switch-case语句的一般用法以及一些需要注意的地方。

代码清单8-4　switch-case语句

```
#include <stdio.h>

int main(int argc, const char * argv[])
{
    int a = 10;
    int b = 0;

    // 这里对a + 1的值进行判断
    switch(a + 1)
    {
        // 如果表达式a + 1的值为10，则执行case 10标签下的语句
        case 10:
            b += a;      // 执行b += a
            break;       // 跳出整个switch语句块

        // 如果表达式a + 1的值为11，则执行case 11标签下的语句
        case 11:
            b -= a;
            break;       // 跳出整个switch语句块
    }
    printf("b = %d\n", b);

    switch(a)
    {
        case 10:
            b = 0;       // 这里没有break语句，因此当执行完case 10下的所有语句时，
                         // 紧接着还会执行它下面的case 12下的语句
        case 12:
            b++;
            break;       // 这里的break将跳出整个switch语句块
    }
    printf("b = %d\n", b);

    switch(b)
    {
        case 0:
            a = 0;
            break;
```

```
        // 当switch选择语句中的表达式b的值
        // 无法匹配这里switch语句块中所有的case常量表达时，
        // 则执行defualt标签语句下的语句
        default:
        {
            // 如果要在switch语句块中的任一标签语句下声明一个临时对象，
            // 则必须使用复合语句，即用{ }将整个上下文包围起来
            int c = 10;
            a = 1;
            b = c + a;
            break;
        }
    }
    printf("a = %d, b = %d\n", a, b);

    switch(b)
    {
        // 在switch语句块中可以放其他表达式或声明对象，在这里的表达式都不会被执行，
        // 因为switch语句块中只执行匹配switch表达式的case标签下的语句
        // 或default标签下的语句。因此不建议在switch语句块中使用除case或
        // default标签语句之外的其他语句
        int c = 0;    // 这里声明了switch语句块作用域的对象c，但c不会被初始化为0
        printf("c = %d\n", c);    // 这里的打印语句不会被执行

        case 1:
        default:
            c = 10;  // 此语句在这里是合法的，因为c没有被声明在case或default标签下
            break;

        case 11:
            c = 2;
            a += c;
            b -= c;
            break;
    }
    printf("a = %d, b = %d\n", a, b);

    // 空switch选择语句
    switch(a);

    switch(a)
    {
        // case标签下的空语句
        case 0:
            ;

        // default标签下的空语句
        default:
            ;
    }
}
```

正如各位在代码清单8-4中所见，switch-case的用法非常灵活，但也充满了各种限制。这里建议各位使用一些常规的用法，避免在switch语句块作用域下声明临时对象，更不要在非case标签语句下使用任何表达式。

以上就介绍完了C语言中的所有选择语句，下面我们将开始介绍C语言

中用于表示循环执行的迭代语句，包括while、do-while和for语句。

8.5 while与do-while迭代语句

C语言中，如果我们想要循环迭代地执行一系列语句，那么我们就要用到迭代语句。我们本小节将先来介绍while与do-while迭代语句。

while语句的一般表达形式为：

```
while ( 表达式 ) 语句;
```

表示如果表达式的值为真，那么执行语句，等到语句执行完之后，再去判断表达式的值，直到表达式的值为假时跳出当前循环。图8-2展示了while迭代语句的执行流程。

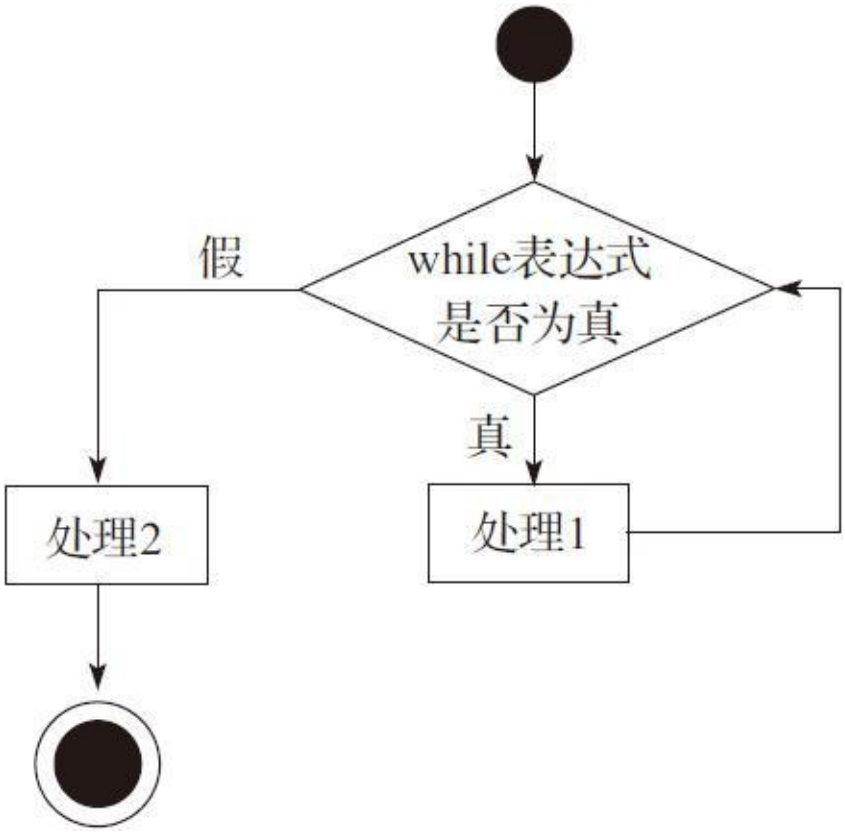


图8-2　while迭代语句的执行流程

图8-2简单明了地展示了while迭代语句的整个执行流程。当进入while语句之后就开始判断其表达式是否为真，如果为真，则执行其循环语句（处理1）。执行完处理1之后再立刻回到while表达式处继续判断，当表达式的值为假时才跳出循环执行处理2。

代码清单8-5列出while的一些使用方法以及一些注意事项。

代码清单8-5　while迭代语句的使用

```
#include <stdio.h>
#include <stdbool.h>

int main(int argc, const char * argv[])
{
    int a = 0, b = 10, c = 5;

    // 这里while语句的判别条件为c > 0，也就是说，当对象c大于0时执行while语句块
    // 中的语句，否则跳出整个while循环语句块
    while(c > 0)
    {
        a++;
        b--;
        c--;
        // 执行完上面三条语句之后，再次回到while(c > 0)处
    }

    // 当c > 0为假时，跳出整个while循环语句块
    printf("a = %d, b = %d, c = %d\n", a, b, c);

    // 这里使用true表示while迭代语句表达式总是为真。这里就是通常所说的“无限循环”
    while(true)
    {
        a++;
        b--;

        // 由于while语句表达式的判别条件总是为真，因此它将会一直循环下去，
        // 此时，我们可以通过加入if选择语句进行判断，
        // 然后使用break跳转语句来跳出整个while循环
        if(b == 0)
            break;  // 这里当b == 0为真时，使用break跳出整个循环
    }

    printf("a = %d, b = %d\n", a, b);

    while(c < 5)
    {
        a++;

        if(a == 11)
```

```
        {
            // continue跳转语句用于while循环语句块中，
            // 表示直接跳转到while语句的判别表达式处，而不执行以下语句。
            // 因此这里当a的值为11时，则不执行下面两句，而直接继续判断c < 5
            continue;
        }

        b++;
        c++;
    }

    printf("a = %d, b = %d, c = %d\n", a, b, c);

    // 下面举一个比较实用的例子，将给定的一个数组中的元素进行倒序排序
    // 将array中元素进行倒序排序
    int array[] = { 1, 2, 3, 4, 5 };

    // 先获取数组长度
    int length = sizeof(array) / sizeof(array[0]);

    // 将变量a作为索引
    a = 0;
    // 判断当前索引是否小于长度的一半，如果小于长度的一半则执行循环
    while(a < length / 2)
    {
        // 交换数组首尾两个元素
        int temp = array[a];
        array[a] = array[length - 1 - a];
        array[length - 1 - a] = temp;

        // 索引递增
        a++;
    }

    printf("array elements: ");
    // 我们这里再使用循环将数组array中的元素打印出来
    a = 0;
    while(a < 5)
    {
        printf("%d  ", array[a]);
        a++;
    }
    // 输出一个换行符
    puts("");

    // 一条空while语句
    while(a > 100);

    // while与if类似，后面可跟一条语句
    while(b > 0)
        b--;

    // 有时，我们为了能将某些语句每次都能放在循环判断之前执行，
    // 可以利用逗号表达式来实现这个效果。
    // 这里每次执行循环迭代前，先执行b+=2; a-=2;，然后再判断c > 0的结果
    while(b += 2, a -= 2, c > 0)
    {
        printf("a = %d, b = %d\n", a, b);

        c--;
    }
}
```

代码清单8-5中还介绍了continue与break这两种跳转语句在while循环体

中的用法。continue用于直接跳转到while表达式处做判别执行，而break则表示跳出当前循环体。

下面我们来介绍一下do-while循环语句。do-while的表达形式为：

```
do 语句 while ( 表达式 ) ;
```

do-while与while十分类似，但不同的是：do-while先执行循环逻辑，然后再通过while来判断当前表达式的真假情况。因此其执行流程就如图8-3所示。

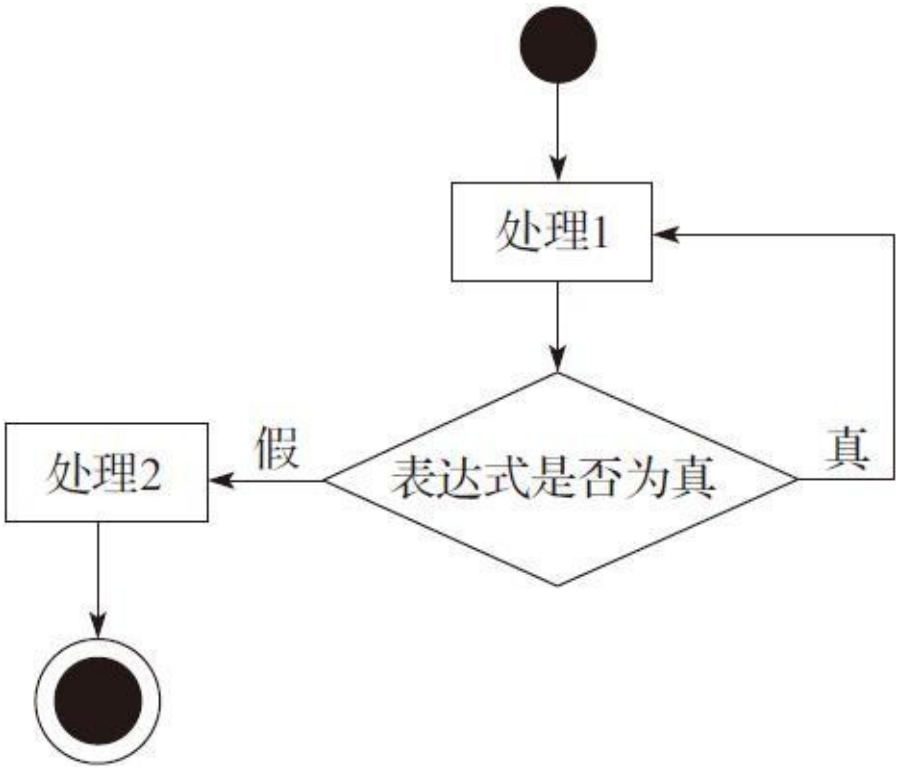


图8-3　do-while语句的执行流程

图8-3中，处理1就好比是do{}中的语句块，当表达式为真时，执行流

直接跳转到处理1中继续迭代执行，直到while（）中的表达式的结果为假，则跳出整个循环，顺序执行循环外的逻辑。

代码清单8-6描述了do-while的使用方式以及相应的一些特性。

代码清单8-6　do-while循环的使用

```
#include <stdio.h>
#include <stdbool.h>

int main(int argc, const char * argv[])
{
    int a = 0, b = 10, c = 5;

    // 这里使用do-while循环
    do
    {
        // 以下三句话必定会先执行一次
        a++;
        b--;
        c += 2;
    }
    while(a < 5);   // 然后判断a < 5的结果，如果为真则继续执行do里的语句
    // 上述循环一共执行了5次
    printf("a = %d, b = %d, c = %d\n", a, b, c);

    do
    {
        // 在do语句块中声明的对象，在其外部不可被访问
        int tmp = a + 1;
        a--;
        b += tmp;
        c--;
    }
    // 这里在while表达式中就不能引用tmp对象
    while(a > 1);

    printf("a = %d, b = %d, c = %d\n", a, b, c);

    do
    {
        b--;

        // 在do语句块中使用continue则直接跳转到while表达式处，
        // 而不执行continue之后的语句
        if(b == 20)
            continue;

        a++;

        // 在do语句块中使用break效果与while语句块相同，都是跳出当前循环体
        if(a == 5)
            break;
    }
    while(c--, true);   // 这里同样可使用逗号表达式

    // 以上，b--执行了5次，而a++与c--则执行了4次
    printf("a = %d, b = %d, c = %d\n", a, b, c);
```

```
    // do语句与if语句类似，后面直接跟一条语句
    do
        a--;
    while(a > 0);

    // 空的do语句。各位要注意的是，do语句后面必须跟while语句
    do;
    while(false);

    do
    {
        a++;

        // 在循环内还可嵌套循环
        while(b > 10)
        {
            b--;
            c++;

            // 这里的break语句仅仅跳出当前的while(b > 10)循环，
            // 而不是外部的do-while循环
            if(c == 10)
                break;
        }
    }
    while(a < 5);

    printf("a = %d, b = %d, c = %d\n", a, b, c);
}
```

---

while与do-while循环介绍完了，下面我们将介绍for迭代语句。

8.6 for迭代语句

for迭代语句的表达形式为:

```
for ( 子表达式1;表达式2;表达式3 ) 语句
```

其中，子表达式1可以是一个声明，也可以是一个表达式。如果它是一个声明，那么在这里声明的对象可在整个循环体中访问。子表达式1在整个循环开始时执行一次，后续的迭代都不会被执行。表达式2是一个控制表达式，用于判定循环执行条件是否满足，如果表达式2的计算结果为真，则执行循环，否则退出当前for循环。表达式3则是在每执行完一次迭代之后执行一次。在每次迭代结束后，总是先执行表达式3，然后再执行表达式2判断是否继续循环，如果可继续循环则继续执行循环语句。

这里，子表达式1与表达式3可缺省。如果表达式2缺省，那么循环条件总是为真。

图8-4描述了for迭代语句的执行流程。

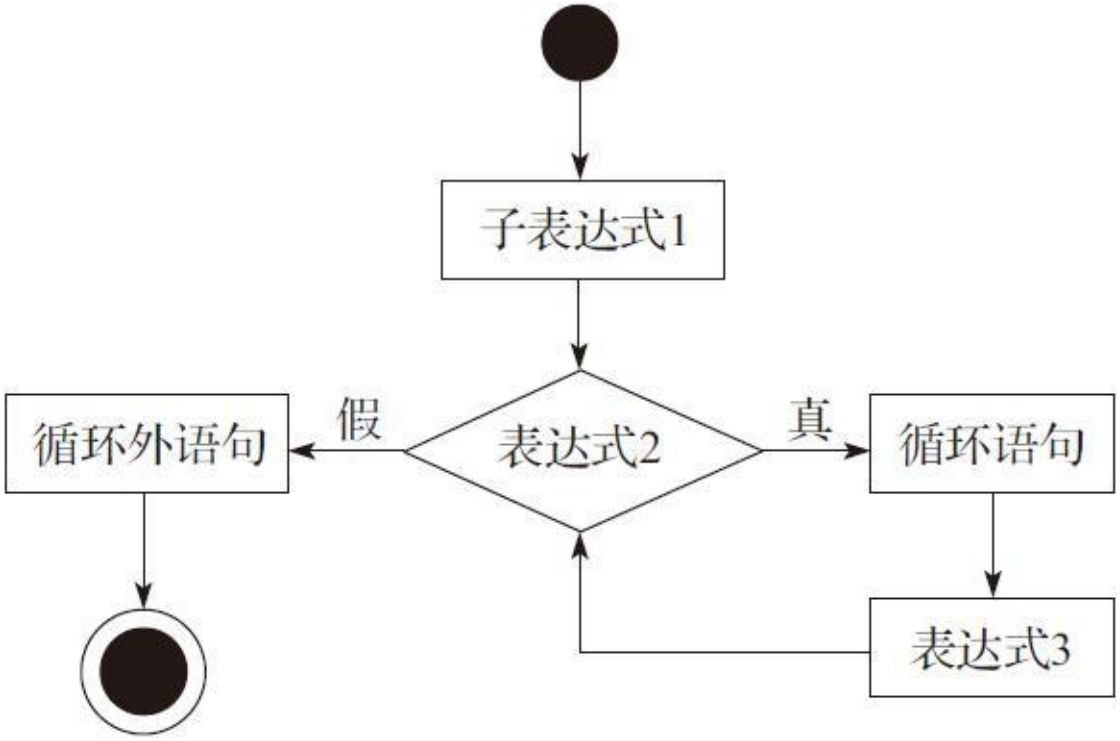


图8-4　for语句执行流程

下面我们将通过代码清单8-7进一步介绍for迭代语句的使用方法以及一些细节问题。

代码清单8-7　for迭代语句的使用

```
#include <stdio.h>

int main(int argc, const char * argv[])
{
    int a = 0, b = 5, c = 10;

    // 这里for里的子表达式1是一个声明，它声明了对象i并初始化为0;
    // 表达式2是判断i的值是否小于5，如果小于5则进行循环，否则跳出循环;
    // 这里声明的i仅对当前for的循环语句可见，对循环语句外部的作用域不可见。
    // 表达式3是i++;在每次迭代时，当循环语句a++;执行完毕后执行一次i++;
    for(int i = 0; i < 5; i++)
        printf("i = %d\n", i), a++;

    // 上述循环语句a++;被执行了5次
    printf("a = %d\n", a);
    i++    // 这条语句错误，标识符i在当前main函数的语句块作用域内不可见!

    // 这里子表达式1是一个表达式，它将0赋值给对象a
    for(a = 0; a < b; a++)
    {
```

```
        c--;
        printf("b + c = %d\n", b + c);
    }

    // 这里的for语句缺省子表达式1与表达式3，仅存在控制表达式2
    for(; c < 10; )
    {
        a--;
        c++;
        printf("c = %d\n", c);
    }

    // 这里的for语句中缺省了子表达式1与表达式2，因此表达式2条件一直为真。
    // 这里表达式3为：a == 3? puts("continue reached!") : (void)0
    // 表示当a等于3时，执行输出命令，否则不做任何事情，(void)0是一条void表达式，
    // 这里表示不做任何动作，相当于一条空语句
    for(; ; a == 3? puts("continue reached!") : (void)0)
    {
        a++;

        // 当a等于3时执行continue语句，从而跳过之后的语句重新跳转到for的表达式3，
        // 然后再判定表达式2的结果（这里一直为true）
        if(a == 3)
            continue;

        b++;

        if(b == 10)
            break;      // 当b等于10时，跳出整个for循环

        printf("a + b + c = %d\n", a + b + c);
    }

    // for后面跟空语句
    for(a = 0, b = 0; a < 10; a++, b++);

    printf("a = %d, b = %d\n", a, b);
}
```

代码清单8-7列出了各种对for迭代语句的用法。这里需要注意的是，当在for循环体内使用continue跳转语句时，控制流是跳转到表达式3的位置，而不是直接去做表达式2的判定。等表达式3执行完成后再做表达式2的判定。

8.7 goto语句

在C语言中，把goto、continue、break、return这4种语句统称为跳转语句（jump statements）。continue与break语句，我们之前在讲述switch选择语句以及循环语句的时候已经介绍了。continue仅用于循环语句中，在while与do-while中表示直接跳转到while循环条件处；在for循环体中表示跳转到表达式3处。而break在case语句中表示退出当前switch内的执行，用在循环体内表示退出当前循环执行。return语句用于函数返回，我们将在下一章进行介绍。本节主要介绍goto语句。

goto语句相当于处理器中的一条分支指令（比如jump、branch等），这条语句也表征了C语言与计算机硬件架构非常贴合。尽管现代各种面向对象的编程语言都不支持goto语句，也有不少开发者对goto语句进行各种冷嘲热讽，但只要使用得当，goto确实也是一个非常不错的编程语法工具。

C语言中要使用goto语句，后面必须加一个标签名（lable）。我们可以在一个函数中几乎任意位置添加一个标签。标签与对象不一样，当出现一条goto语句时，即便goto指定的那个标签在当前goto语句之前未曾出现也没关系，goto允许向下跳转。正因为goto语句十分灵活，所加的限制很小，所以也很有可能会出现一些问题，比如跳转到某一位置，而在该位置处所引用到的对象未被初始化，这就可能产生不可预期的运行结果。因此我们

在使用goto语句的时候要确保当前可见的对象都被正确初始化了。

代码清单8-8展示了goto语句的一些用法以及一些需要注意的地方。

代码清单8-8　goto语句的使用

```
#include <stdio.h>

int main(int argc, const char * argv[])
{
    int a = 0;

    a++;

    // 如果a大于0，则跳转到NEXT标签后的语句
    if(a > 0)
        goto NEXT;

    // 这里的a--;被跳过执行
    a--;

NEXT:
    // 跳转到此循环
    for(int i = 0; i < 5; i++)
    {
        a += i;

        // 如果a > 5，则跳出此循环，到NEXT2标签后面的语句
        if(a > 5)
            goto NEXT2;

        printf("a = %d\n", a);
    }

NEXT2:

    a++;

    int b = 10;

    // 如果a与b的和大于20，则跳转到NEXT下的for循环语句处
    if(a + b < 20)
        goto NEXT;

    // 若a > 0，则跳转到NEXT3标签后的语句处
    if(a > 0)
        goto NEXT3;

    int c = 10;

NEXT3:
    // 这里要注意的是，如果跳转到此处，之前对对象c的初始化就不会执行，
    // 所以此时c的值是不确定的
    printf("c = %d\n", c);

    goto FINISH;

FINISH:
    // 一个标签后面必须跟一条语句，
    // 这里用一个空语句（分号）来表示
```

```
    ;
}
```

代码清单8-8列出了各种对goto语句的使用方式，包括向前跳转和向后跳转，从循环语句中跳出，等等。另外，还展示了一次跳转可能会引发对象值不确定的情况。各位在使用goto时应当注意这些问题。

另外，为了编写一个结构良好的代码，在很多时候我们都避免使用goto语句。下面我们列举一个使用do-while语句来实现类似goto语句效果的代码，如代码清单8-9所示。

代码清单8-9　可替代goto语句的代码示例

```
#include <stdio.h>
#include <stdbool.h>

int main(int argc, const char * argv[])
{
    int a = 0, b = 1, c = 2;

    // goto语句效果如下：
    if(a < 0)
        goto FINISH;

    printf("a = %d\n", a);

    if(b < 0)
        goto FINISH;

    printf("b = %d\n", b);

    if(c < 0)
        goto FINISH;

    printf("c = %d\n", c);

FINISH:

    if(a < 0 || b < 0 || c < 0)
        puts("Error");

    // 使用do-while来实现上述代码效果：

    do
    {
        if(a < 0)
            break;

        printf("a = %d\n", a);
```

```
            if(b < 0)
                break;

            printf("b = %d\n", b);

            if(c < 0)
                break;

            printf("c = %d\n", c);
        }
        while(false);

        if(a < 0 || b < 0 || c < 0)
            puts("Error");
    }
```

代码清单8-9列举了常用错误处理的方法。我们看到，if语句与goto语句联用能比较方便地处理因某些参数错误而需要进一步处理的逻辑。然而，我们有时也可以用do-while同样简洁地表达这一逻辑。当然，如果是在多层嵌套的循环中要跳转出整个循环，这时还不如goto来得直接。根据当前上下文我们可以选择比较得体的解决方案，同时这里也建议尽量避免使用goto语句，但如果需要花很大力气、写不少冗余代码才能避免goto的话，有时还不如直接用goto来得更体面一些。

8.8 本章小结

本章介绍了C语言中所有的控制流语句，包括选择语句、循环语句以及跳转语句（return语句将放在下一章详细描述）。这些通常也是一个计算机程序的执行顺序控制模式。当然，对于计算机处理器的执行而言，它往往只有跳转指令，而结构化的分支、循环是高级语言对机器执行的一种高层抽象，使得高级语言编写的程序更易读。

下一章，我们将介绍更高级、更具模块化的分支功能——函数。当各位把函数学完后，整个C语言的语法框架就掌握得差不多了。

第9章　C语言的函数

我们在第8章介绍了C语言中几种基本的控制流语句。但是为了使一个C语言程序写出更良好的结构化、模块化的代码，我们需要借助函数（function）。

在前几个章节中我们在不少示例代码中调用了一些库函数，比如memcpy、memset、printf等，它们都完成一些特定的功能，比如memcpy是将一个源数据缓存空间中指定长度的内容（字节）拷贝到目的缓存。而printf的功能则是将指定的字符串输出到控制台。因此在C语言中，一个函数是由若干语句构成的用于完成一系列功能的代码块。一个函数可以有输入也可以有输出。输入数据通过参数传递的方式从函数调用者（function caller）传入函数内，输出数据则是由函数返回给其调用者的操作计算结果。

这里自然而然地引出了调用者与被调用者（callee）的概念。所谓调用者是指调用某一函数的函数，被调用者则是当前被调用的函数。比如，我们在main函数中调用printf函数，那么对于printf函数而言，调用者就是main函数，printf则是被调用者，而调用该printf函数时所处的位置则称为函数调用点（invoke point）。

函数的执行与其他控制流语句不同，在我们调用一个函数时，当被调

用函数执行完自己的逻辑之后会把控制交还给其调用者继续执行。该执行流程如图9-1所示。

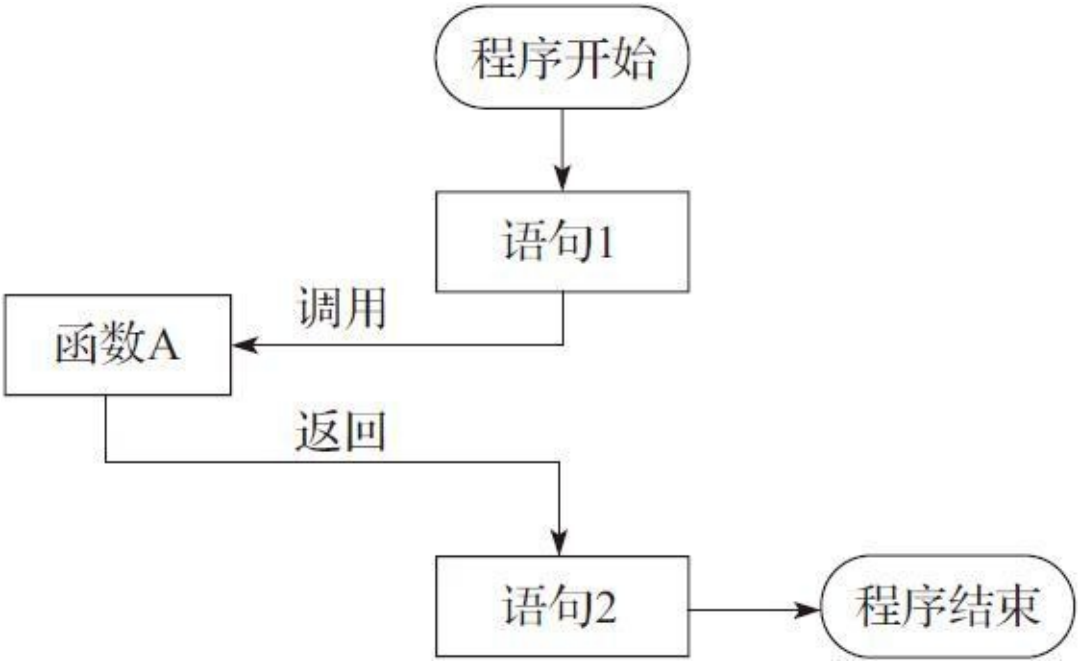


图9-1　函数的执行流程

如图9-1所示，程序一开始先进入main函数执行语句1，然后调用函数A，在执行了函数A之后将控制返回给main函数，紧接着执行main函数中的语句2，最后退出当前程序。

本章将介绍函数的声明、定义、调用、参数传递，此外后续章节还会介绍指向函数的指针以及对主函数的介绍。

9.1 函数的声明与定义

C语言中，一个函数由返回类型、函数名以及形式参数（简称形参）列表构成。在C语言标准中，又把函数名称作为函数标志（function designator）。函数声明以如下形式表示：

```
返回类型 函数名（形参列表）
```

形参列表中，每个形式参数（parameter）用逗号分隔。比如：int foo（int a，float b）；这就是一个函数声明。该声明给出了一个名为foo的函数，其返回类型为int，其形参列表为（int a，float b）；该列表声明了函数foo带有两个形参，第一个形参为int类型，第二个形参为float类型。如果一个函数不返回任何值，那么其返回类型用void表示。如果一个函数不含有任何形式参数，那么参数列表可用（）或（void）来表示。本书偏向使用（void）来表示不带任何形参的形参列表，因为这样函数声明与函数调用不容易被搞混。在老式的C语言编译器中，允许缺省函数返回类型，如果缺省，则默认函数返回类型为int。但对于现代C语言编译器而言，如果不写函数返回类型则会报出警告，尽管也会将该函数的返回类型默认作为int类型处理。

一个函数声明给出了该函数的原型（prototype），一个函数原型能确定该函数的一个比较完整的类型。如果函数声明不作为函数定义的一部分，那么形参列表中的形参可含有不完整类型的形参，另外可以用[*]来表

示变长数组类型；否则每个形参都必须是完整类型的。在一个函数原型中，函数形参列表中每个形参的标识符可省。比如上述的foo函数可声明为：

```
int foo ( int, f loat );
```

函数定义的一般形式为：

```
存储类说明符可缺省  返回类型  函数名  (形参列表) 复合语句
```

在函数定义中，要么形参列表中的每个形参类型必须是完整类型，要么形参列表为空列表。返回类型也必须是完整类型或void类型。另外，存储类说明符只能用extern或static，或者可缺省。如果缺省存储类说明符，则默认为extern。extern存储类说明符表示一个函数具有外部连接；static存储类说明符表示一个函数具有内部连接。存储类型说明符将在第11章详细介绍。

下面我们将通过代码清单9-1来给大家先简单介绍一下函数的声明与定义的具体C语言代码。

代码清单9-1　函数的声明与定义

```
#include <stdio.h>

// 这里声明一个结构体类型MyStruct，对它尚未定义
struct MyStruct;

// 这里声明了一个函数MyFunc，它具有静态存储类，返回类型为MyStruct，
// 带有一个形式参数s，其类型为MyStruct。MyStruct这里为不完整类型
static struct MyStruct MyFunc(struct MyStruct s);

// 这里直接定义了一个函数Foo，它具有外部存储类，返回类型为void，
```

```
// 并且其形参列表为空
void Foo(void)
{
    puts("Hello, world!");
}

// 这里定义MyStruct结构体类型
struct MyStruct
{
    int a;
    float b;
};

// 这里直接声明了三个函数，第一个函数是int Func1(void);
// 第二个函数是int* Func2(int);
// 第三个函数是int (*Func3(void))[3];即返回一个int(*)[3]类型的不带形参的函数。
// 对函数声明后但不定义，且不去实际引用，也不会存在连接时错误
int Func1(void), *Func2(int), (*Func3(void))[3];

int main(int argc, const char * argv[])
{
    // 调用Foo函数
    Foo();

    // 这里调用MyFunc函数，给它传递的实际参数是MyStruct结构体的一个复合字面量，
    // 然后MyFunc所返回的MyStruct结构体对象赋值给s对象
    struct MyStruct s = MyFunc((struct MyStruct){10, 1.5f});
    printf("s.a = %d, s.b = %f\n", s.a, s.b);
}

// 这里定义函数MyFunc，这里MyStruct已经是完整类型了
static struct MyStruct MyFunc(struct MyStruct s)
{
    return (struct MyStruct){ s.a + 10, s.b - 0.5f };
}
```

代码清单9-1中我们看到先声明了一个MyFunc，然后定义了一个函数Foo。在定义函数Foo的时候，其原型也就同时给出了。因此在main函数中，我们可以分别调用这两个函数。如果在某一函数中要调用另一个函数，而所调函数的原型在该点处未知，那么编译器就会发出警告。

在main函数中，当执行Foo（）；这条语句时，就是对Foo函数进行调用，此时控制流会跳转到Foo函数中的代码进行执行，同时会保护当前上下文。等Foo函数都执行完之后将会恢复main之前调用处的上下文，然后把控制流返回给当前的main函数，继续执行Foo（）；之后的代码。然后调用了MyFunc函数，MyFunc函数构造了一个临时MyStruct结构体对象，

将该对象初始化后返回出去。在main函数中的调用端，我们声明了一个MyStruct结构体对象s，将MyFunc返回的结构体对象对它进行初始化，这样s的成员a被初始化为20，成员b被初始化为1.0f。

9.2 函数调用与实现

9.1节主要介绍了函数的声明与定义，并且简单提到了函数调用形式。本节将详细介绍函数调用及其实现原理。

像代码清单9-1中main函数里的Foo（）这一表达式在C语言标准中称为函数调用表达式。而Foo后面的（）就是函数调用操作符（function call operators），它是一个单目后缀操作符，它前面的函数标志就是一个后缀表达式，作为其操作数。而整个函数调用表达式也是一个后缀表达式，后面圆括号中可以不加任何东西，也可以添加一组用逗号分隔的表达式，这组表达式也称为指定函数实参的表达式列表。一个实参（即实际参数，actual argument，简称argument）可以是任一完整对象类型的表达式。在函数调用之前，所有实参都要完成计算，并且赋值给对应形参。函数的每个形参，在该函数被调用时都需要有一个实参与之对应。所以这里实参与形参的概念十分容易区分：实参是属于函数调用者的对象，然后传递给函数被调者；而形参是属于函数被调者的。

这里就涉及了圆括号出现在表达式中的所有各类情况——如果圆括号中放的是类型名，那么此时圆括号扮演的投射操作符的角色；如果圆括号中是一个表达式，并且前面含有一个表示函数标志的表达式（包括指向函数的指针），那么此时圆括号扮演的是函数调用操作符的角色；如果圆括号中存放的是一个表达式，并且前面没有表示函数标志的表达式，那么此

时圆括号扮演的就是圆括号操作符的角色。

9.2.1 函数调用的顺序点

这里各位要注意的是，在对函数标志（即函数名）计算与对实参计算之后、在函数实际调用之前，有一个顺序点。但是，C语言标准没有说函数标志的计算与对实参的计算之间有顺序点，因此对函数标志的计算与对实参的计算谁先谁后是不确定的。下面用代码清单9-2来举一个简单的例子进行说明。

代码清单9-2　函数调用过程中的顺序点

```
#include <stdio.h>

static void Func1(int a)
{
    printf("f1 a = %d\n", a);
}

static void Func2(int a)
{
    printf("f2 a = %d\n", a);
}

static void Func3(int a, int b)
{
    printf("b - a = %d\n", b - a);
}

int main(int argc, const char * argv[])
{
    // 这里借助一个指向函数指针的数组对象来表明一个函数标志。
    // 这样，对pFuncs[i]作为函数标志符的计算就能体现出来了
    void (*pFuncs[2])(int) = { &Func1, &Func2 };

    int i = 0;

    // 这里编译器会发出警告，因为pFuncs[i++]的计算与i的计算是未确定顺序的
    pFuncs[i++](i);

    // 这里编译器会发出警告，因为pFuncs[i]的计算与++i的计算是未确定顺序的
    pFuncs[i](++i);

    // 这里编译器会发出警告，因为第一个实参的计算与第二个实参的计算是未确定顺序的
    Func3(i++, i);
}
```

代码清单9-2引入了将在9.8节中介绍的指向函数的指针语法，这里先借助一下用来表明对函数标志的计算顺序问题。这里各位可以看到，无论是函数标志的计算还是实参列表中对每个实参的计算，它们之间都没有顺序点。

没有顺序点有何好处？现代高级一点的处理器都具有超标量流水线（superscalar pipeline）以及无序执行引擎（out-of-order execution engine），使得处理器对多条前后没有依赖关系的指令进行并行执行，这也被称为指令级并行（Instruction-Level Parallelism，ILP）。我们以代码清单9-2为基础，写一条pFuncs[i+1]（i-1）；函数调用表达式语句，那么汇编指令可能是这样的（基于ARMv7架构指令集）：

```
ldr  R1, =i;
adr  R12,  pFuncs;
add   R12, R12, R1, lsl #2;
sub  R0, R1, #1;
ldr R12, [R12, #4];
blx  R12;
```

第1条指令是将变量i的内容读取到R1寄存器中；

第2、3条指令则是将R12寄存器指向pFuncs数组的第i个元素的地址；

第4条指令是将i-1的值作为第一个实参的值；

第5条指令则完成了对整个pFuncs[i+1]的计算，获得了当前要调用的函数指针的值；

第6条指令做函数调用。

这里，第1、2条指令没有依赖关系，因此可并行执行；第3条指令因为动用了寄存器移位操作加算术操作，本身比较复杂，因此一般无法与其他指令做并行执行了；第4、第5条指令之间也不存在相互依赖，可并行执行。由此我们可以看到，不安插顺序点对于现代处理器的加速执行而言是非常有好处的。而处理器在做诸如函数调用等分支指令时，自然会清算之前相关计算的副作用（包括对函数标志的计算以及实参的计算）。因此，我们在涉及不指定顺序点的表达式语句中，不要在整条语句中出现对同一对象做产生顺序点副作用的操作（比如自增、自减操作）。

下面通过代码清单9-3对代码清单9-2进行改进，来说明如何避免代码清单9-2中所产生的警告问题。

代码清单9-3　避免产生顺序点副作用

```
#include <stdio.h>

static void Func1(int a)
{
    printf("f1 a = %d\n", a);
}

static void Func2(int a)
{
    printf("f2 a = %d\n", a);
}

static void Func3(int a, int b)
{
    printf("b - a = %d\n", b - a);
}

int main(int argc, const char * argv[])
{
    void (*pFuncs[2])(int) = { &Func1, &Func2 };

    int i = 0;

    pFuncs[i](i + 1); i++;
```

```
    pFuncs[i](i + 1); i++;

    Func3(i, i + 1); i++;

    // 或者
    i = 0;
    int j = 0;

    // 由于i与j是两个不同的对象，因此这里不会引发顺序点的冲突问题
    pFuncs[j++](++i);

    pFuncs[j++](++i);

    Func3(j, ++i);
}
```

上面我们讲述了函数调用的基本形式以及实参、函数标志的计算与函数调用之间的顺序点。下面我们将讲述函数调用执行的细节。尽管C语言标准没有规定函数执行的过多细节，这是由于这样可以让各种编译器针对当前运行环境有更多更自由灵活的实现方式，不过我们通过对函数内部执行细节的学习可以更深入透彻地理解C语言函数的整个概念。

正如上文所描述的，对一个函数的调用要经历三个步骤，前两个步骤分别是对函数标志表达式的计算以及对函数实参的计算，这两步之间不分先后顺序，最后是对函数进行调用。把实际参数传递到函数形式参数的实现是由各个处理器以及系统环境决定的，这里涉及函数调用约定，此内容将在第15章做详细介绍。

9.2.2 函数的栈空间

对于大部分处理器以及操作系统环境而言，每个函数都具有自己独立的上下文存储空间，此存储空间往往是栈式存储的，所以又被称为栈

（stack）空间。相应地，一般处理器会有一个专用的栈指针寄存器（Stack Pointer Register，一般简称为SP）用于存放当前函数所属的栈空间的地址。初始时，SP会指向进程给当前程序分配栈空间的最大地址处。然后我们在函数中定义一个局部对象，那么它可能就会被保存在当前函数的栈空间中，此时栈指针SP会先减该局部对象所占存储空间的字节大小，然后将该对象的内容存放在此存储单元内。我们观察一下代码清单9-4所列出的main函数在执行代码的过程中对栈指针的变化。

代码清单9-4　栈指针操作大概示意代码

```
int main(void)
{
    int a = 10;    // 相当于SP -= 4; [SP] = 10;
    a += 20;       // 相当于reg = [SP]; reg += 20; [SP] = reg;

    // 函数返回前需要恢复栈空间，即把当前函数所使用的栈空间内容给自动回收
    // 这里相当于：reg = [SP]; SP += 4;
    return a;
}
```

代码清单9-4中，注释里写的是处理器的部分汇编指令，其中[SP]表示访问SP当前所指向的栈空间的内容，reg表示某个寄存器。上述代码假定运行在32位系统中，一个int类型的对象所占的存储空间为4个字节。因此一开始SP-=4表示SP指针先减去4个字节，然后[SP]=10表示将常量10存储到减去4之后的SP所指向的栈空间。而这两条指令（即SP-=4；[SP]=10）又被称为压栈（push）操作，许多处理器会直接用PUSH指令助记符来表示，比如可以把这两条指令表示为一条指令：PUSH 10。随后，在做a+=20；时，先获取将当前SP所指向的栈空间地址值，然后读取该地址中所存放数据的值（这里存放的是对象a的值）加载到寄存器reg中，再将该寄存器的值加上

20，最后把该寄存器的值写入SP所指的栈空间中。最后一句返回语句则是将SP所指的栈空间地址中存放的值加载到reg寄存器中用于返回结果，然后将栈指针加4，即恢复函数调用前的SP的位置。这一过程又称为出栈（pop）操作，许多处理器会直接使用POP指令助记符来表示，比如可以把最后两条指令表示为：POP reg。图9-2展示了代码清单9-4中执行main函数时的SP操作过程。

图9-2假定在系统调用main函数时，SP栈指针指向0x8000地址，并且将从0x7000到0x8000这4KB存储空间作为当前进程的栈空间。当执行了int a=10；这条语句之后，相当于将常量10压入栈，此时对象a的存储空间可看作为以0x7FFC作为起始地址，一直到0x7FFF，这4字节的存储单元，所以对象a的地址（&a）就是0x7FFC。最后执行return a；语句之后，先将对象a的值加载到用于返回值的寄存器reg中，然后恢复SP到初始时的位置。最后执行RET指令，将被调函数所保存的函数调用者之前所执行CALL指令的下一条指令的地址作为目标指令地址进行跳转。

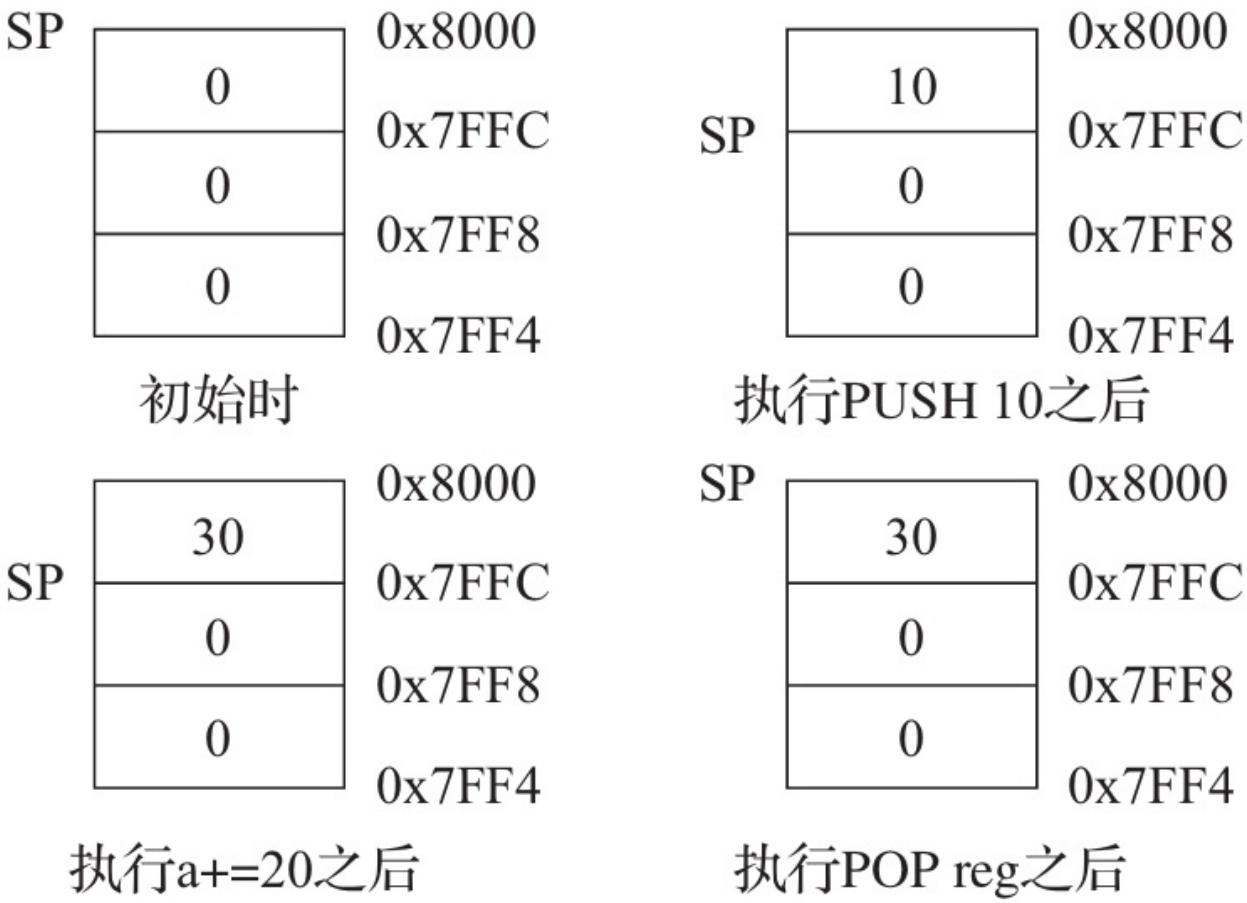


图9-2　SP栈指针操作过程

9.2.3　函数的参数传递与返回

上面介绍了对一个函数中局部对象的操作，并且初步描述了处理器中栈指针SP的工作方式。而在调用某个函数时，将实际参数传递给被调函数的形式参数时，有时也需要对栈空间进行操作，尤其是对于寄存器数量不多的处理器。一般会将被调函数的形参所在的栈存储空间分配在调用函数的栈空间区域中，也就是说让函数调用者负责将实参内容压栈，以此形式给被调函数的形参赋值。倘若被调函数要返回一个结构体对象，以至于无

法将返回内容保存在一个寄存器内，那么该对象存放在被调函数的栈空间中，然后将该结构体对象的首地址放在结果寄存器中返回出去。函数调用者需要通过SP指针将被调函数所返回的结构体内容拷贝到自己的栈空间中去。

下面我们通过代码清单9-5大致介绍一下在函数中调用另一个函数时，在参数传递以及获取返回数据时，对栈指针的操作。

代码清单9-5　参数传递与函数返回时的代码示例

```
#include <stdio.h>

struct MyStruct
{
    int i;
    float f;
};

static struct MyStruct Foo(int a, float b)
{
    struct MyStruct s = { a, b };

    s.i += 10;
    s.f -= 0.5f;

    return s;
}

int main(int argc, const char * argv[])
{
    struct MyStruct s = Foo(10, 1.5f);

    printf("s.i = %d, s.f = %f\n", s.i, s.f);
}
```

代码清单9-5中，在main函数中先用实际参数10与1.5f来调用函数Foo，然后将函数Foo所返回的MyStruct类型的对象给main函数中声明的局部对象s进行初始化。这里，main函数称为函数调用者（caller），而Foo函数则被称为函数被调用者（callee）。Foo的形参都位于main函数的栈空间范围

内，图9-3展示了在main函数中调用Foo函数前后栈指针的变化。

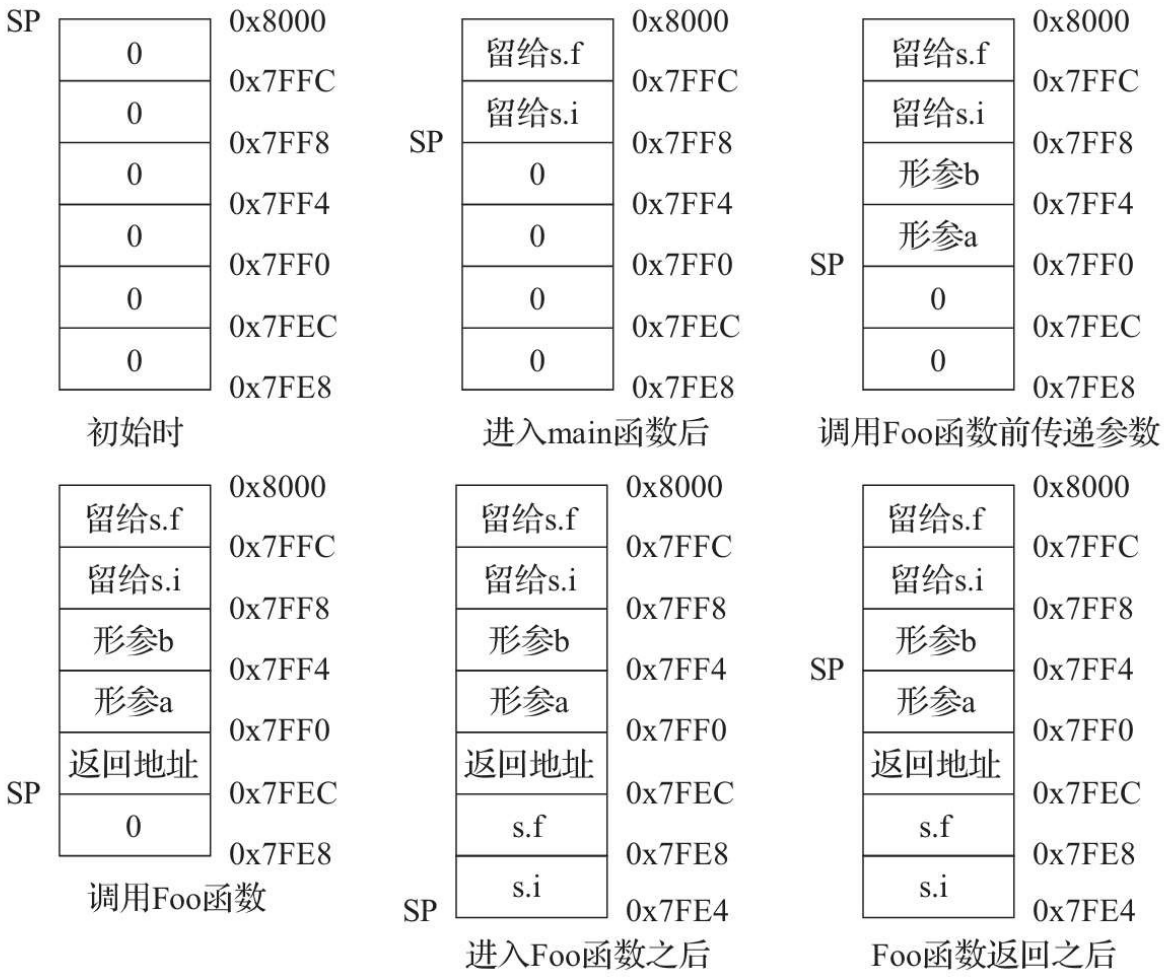


图9-3　函数调用时的参数传递与返回结果的获取

图9-3中，在进入main之前，栈指针SP假定指向0x8000这个地址。进入main之后，假定SP指针移动到了0x7FF8的位置，这是为了给main函数中的局部对象s留出存储空间。也就是说对象s的起始地址即为0x7FF8。当开始调用Foo函数时，先将实参10和1.5f传递给Foo函数的形参a和b，此时，先压入形参b的值，再压入形参a的值，形参a的地址自然就比形参b要低了。此时，形参a的地址为0x7FF0，形参b的地址为0x7FF4。传递完参数后，开始调用函数Foo。关于函数调用，有些处理器（比如x86）是将函数调用指

令的下一条指令的地址自动压栈，而有些处理器（比如ARM）则是有专门的连接寄存器（link register）用于存放函数调用指令的下一条指令地址。这里假定我们用的是类似x86处理器那种，将返回地址做压栈操作的。进入Foo函数后，跟main函数类似，再将SP栈指针减8，留给Foo函数中的局部对象s。这样对象s的起始地址就是0x7FE4。此时，我们从图中可以观察到，从栈空间地址0x7FF0到0x7FFF这段存储空间是属于函数调用者main函数的区域；而栈空间地址0x7FE4到0x7FEF这段存储空间是属于被调者Foo函数的区域。当Foo函数返回之后，SP栈指针回到0x7FF0处。此时，编译器会自动生成代码，将Foo函数中所返回的的局部对象s（地址在0x7FE4处）的首地址通过寄存器返回给函数调用者main，然后在main函数中将获得的Foo所返回的局部对象s的内容拷贝到它自己的局部对象s中（地址在0x7FF8处）。在没有调用另一个函数之前，所有之前调用过的函数所留在栈空间中的数据都会被保留，这样有助于做返回值的拷贝。但是当执行完main函数中的struct MyStruct s=Foo（10，1.5f）；这条语句之后，我们就应该认为之前所调用的Foo函数在栈空间留下的任何数据都已经无效了。此后，栈指针SP会恢复到0x7FF8，即把之前压入的形参对象也“销毁”掉。

下一条语句是调用库函数printf，此时与调用Foo函数类似，会对栈指针进行操作，先做参数传递，然后调用printf函数。在调用printf函数过程中，之前Foo函数所留在栈空间的数据都会被printf函数所留下的数据给覆盖掉。也就是说，一个系统进程的整个栈空间都是该进程中所有被调函数共享的。因此，栈空间是一个可活动的、可复用的、用于存放函数中临时产生数据的存储空间。

以上便描述了调用一个函数后编译器以及处理器大概会做的一些事情。看到这里，相信读者应该对栈这个概念有所了解了，并且对函数内定义的局部对象的生命周期也有了大概的认识。那么说到这儿我们应该知道，被调函数的形参与调用者的实参其实是两个不同的对象。函数调用者的实参在自己的栈空间内，而在传递给被调函数时，是将保存在自己栈空间的对象做压栈处理，拷贝到被调函数可访问的栈空间区域，尽管这部分区域仍然属于函数调用者。为了更清晰地表达这一点，我们将通过代码清单9-6进行陈述。

代码清单9-6　呈现被调函数形参与调用者的实参属于不同对象

```
#include <stdio.h>

static void Foo(int a)
{
    // 这里将输出：a address is: 00007FFF5FBFF7EC
    printf("a address is: %.16tX\n", (uintptr_t)&a);

    a += 10;

    // 这里将输出：a = 110
    printf("a = %d\n", a);
}

int main(int argc, const char * argv[])
{
    int x = 100;

    // 这里将输出：x address is: 00007FFF5FBFF80C
    printf("x address is: %.16tX\n", (uintptr_t)&x);

    Foo(x);

    // 这里将输出：x = 100
    printf("x = %d\n", x);
}
```

通过代码清单9-6我们可以发现，main函数中将其局部对象x传递给Foo函数的形参，x的地址为0x 00007FFF5FBFF80C，而Foo形参a的地址为

00007FFF5FBFF7EC，两者属于不同的对象。所以，即便在Foo函数中任意修改形参a的值都不会影响main函数中x对象的值。

9.2.4 通过形参修改实参的值

那么我们可能会问，如何通过函数来修改函数调用者对象的值呢？答案很简单：通过指针！如果我们将函数调用者的某个对象的地址作为实参传递给被调函数的形参（被调函数的形参为一个指针类型的对象），那么在被调函数中可利用间接操作对形参所指对象的内容进行修改。代码清单9-7将通过实现交换两个整数实参值的函数来描述如何利用指针来修改实参值。

代码清单9-7　实现交换两个整数对象值的函数

```
#include <stdio.h>

static void MySwapFunc(int *p, int *q)
{
    // temp先保存形参p所指对象的值
    int temp = *p;

    // 将形参q所指对象的值赋给形参p所指对象
    *p = *q;

    // 将temp保存的值赋给形参q所指对象，这样正好完成了整个交换操作
    *q = temp;
}

int main(int argc, const char * argv[])
{
    int a = 10, b = 20;

    // 这里调用MySwapFunc时，
    // 分别将对象a的地址与对象b的地址作为实参传递给MySwapFunc函数
    MySwapFunc(&a, &b);

    // 我们通过打印可以看到，对象a的值与b的值两者被交换了
    printf("a = %d, b = %d\n", a, b);
}
```

代码清单9-7中，通过将main函数中的局部对象a与b的地址作为实参传递给Foo函数的形参，然后由Foo函数通过对形参的间接操作来实现交换两个函数调用者对象的目的。像*p=*q；这条语句执行完之后，main函数中的对象a的值就变为了20。而当*q=temp；这条语句执行完后，main函数中的对象b的值变为了10。

9.3　数组类型作为函数形参

如果一个函数的形参是一个数组类型的对象，那么它会被调整为指向该数组元素类型的指针，同时如果类型还有限定符（比如const、volatile），那么可以在表示数组对象的[]里添加。如果在[]中含有static关键字，那么实参必须确保至少能访问该形参所指定元素个数的元素数量。

对于一个纯函数声明而言（即声明该函数之后不直接对它定义），形参可以具有不完整类型，并且可以使用[*]来表示变长数组类型。

代码清单9-8将描述这些特性。

代码清单9-8　数组类型对象作为函数形参

```
#include <stdio.h>

// 这里是对函数Func1的声明，不对它进行定义，因此可用[*]表示一个变长数组类型。
// 这里需要注意的是，a的类型为int[*]，它是一个不完整类型
static void Func1(int a[*]);

// 这里对函数Func1进行了定义，这里不能使用[*]，但可以用[]。
// 因为int[*]是一个不完整类型，而int[]是完整类型，
// 因为int[]会被自动转换为int*。当函数形参为数组类型时，
// 会被自动转换为指向该数组元素类型的指针，所以这里的形参a就是int*类型
static void Func1(int a[])
{
    if(a != NULL)
        printf("a[0] = %d\n", a[0]);

    // 这里sizeof(a)的大小就相当于sizeof(int*)的大小
    printf("size of a = %zu\n", sizeof(a));
}

// 当一个数组类型对象作为函数形参时，无论指定数组长度是多少都不会有用。
// 因为它们都会被转换为指向该数组元素类型的指针。
// 这里的形参a也是int*类型
static void Func2(int a[10])
{
    if(a == NULL)
        puts("nil!");

    // 这里sizeof(a)的大小就相当于sizeof(int*)的大小
    printf("size of a = %zu\n", sizeof(a));
```

```
}

// 这里用static表示调用Func3时，实参所指定的数组或缓存应该至少含有5个int元素对象。
// 这里加上const限定符，表示对形参a做了常量限定，a不能指向其他对象地址。
// 因此，这里a的类型为int * const
static void Func3(int a[static const 5])
{
    int sum = 0;

    // 对数组元素求和
    for(int i = 0; i < 5; i++)
        sum += a[i];

    // OK
    a[0] = 100;

    // 这句话错误：a = NULL;，a不能指向其他对象，即a的值不能被修改
}

// 这里声明Func4，其形参类型为一个元素类型为int的二维数组，
// 其中a[i]的类型被声明为int[*]，是一个不确定个数的数组，
// 它是一个不完整类型
static void Func4(int a[static const 2][*]);

// 这里对Func4进行定义，并明确表示a[i]的类型为int[3]。
// 对于一个函数形参类型为二维数组类型的也类似，
// 这里a的类型会被自动转换为int(* const)[3]，即指向int[3]数组的常量指针
static void Func4(int a[static const 2][3])
{
    // 这里将输出sizeof(int[3])一样的大小
    printf("size of a[0] = %zu\n", sizeof(*a));

    // 将a[1][2]的值修改为20
    a[1][2] = 20;

    // 这句话错误：a = NULL;，a的值不能被修改
}

int main(int argc, const char * argv[])
{
    // 用一个数组字面量作为实参来调用Func1
    Func1((int[]){ 1, 2 });

    // 这里直接用空值作为形参来调用Func2
    Func2(NULL);

    int array[] = { 1, 2, 3, 4, 5, 6 };

    // 将数组array作为实参来调用Func3
    Func3(array);

    // array的第一个元素被修改成了100
    printf("array[0] = %d\n", array[0]);

    int darray[][3] = {
        1, 2, 3,
        4, 5, 6
    };

    Func4(darray);

    // darray[1][2]的值被修改成了20
    printf("darray[1][2] = %d\n", darray[1][2]);
}
```

代码清单9-8涉及const限定符的问题，关于这个话题我们将在第12章详细描述，这里仅用来陈述当数组类型作为函数形参时可以如何表达。

既然我们知道了当函数形参是一个数组类型时，它会被自动转为相应的指针类型，也就是说我们不能将一个数组直接以元素拷贝的方式传递给函数形参，因此我们往往以一个数组的某个元素的地址传递给被调函数的形参，然后可以再加一个参数表示当前提供数组的长度。代码清单9-9给出了一个示例代码，里面实现了给一个指定数组的所有元素进行倒序排列的函数。

代码清单9-9　对指定数组进行倒序排列的函数实现

```
#include <stdio.h>

/**
 * 这里定义一个名为SwapArray的函数，用于实现给指定数组的所有元素进行倒序排列
 * @param a 指向一个传入数组元素的地址，其类型为int*
 * @param count 用于指定传入数组的长度（即元素个数）
*/
void SwapArray(int a[], int count)
{
    // 这里只需要遍历一遍长度
    for(int i = 0; i < count / 2; i++)
    {
        // 交换数组首尾两个元素的值
        int temp = a[i];
        a[i] = a[count - i - 1];
        a[count - i - 1] = temp;
    }
}

int main(int argc, const char * argv[])
{
    int array[] = { 1, 2, 3, 4, 5 };

    // 从array第0个元素开始，对所有元素进行倒序排序
    SwapArray(array, 5);

    // 输出排序结果
    printf("Elements: ");

    for(int i = 0; i < 5; i++)
        printf("%d  ", array[i]);

    puts("");

    // 再从array[1]元素开始，对它及后序元素进行倒序排序
```

```
    SwapArray(&array[1], 4);

    // 输出排序结果
    printf("Elements: ");

    for(int i = 0; i < 5; i++)
        printf("%d  ", array[i]);

    puts("");
}
```

通过代码清单9-9，我们对如何将数组对象作为实参传递给被调函数有了一些清晰的思路。上述倒序排序数组元素的算法也十分简单，即将数组第一个元素与最后一个元素进行交换，第二个元素与倒数第二个元素进行交换，以此类推，最后一直到中间那个元素，这样整个数组的元素就被倒序排序了一遍。

9.4 带有不定参数类型及个数的函数声明与调用

C语言函数的形参类型列表的最后可以带有不定参数类型及个数的形参列表，用（，...）来表示。C语言标准明确规定，含有不定参数个数的形参列表中，必须要有一个确定的命名形参，并且...后不能再跟其他形参。比如以下函数声明是错误的：

```
// 错误！在 ... 之前必须至少要有一个命名形参
void Func1(...);

// 错误！在 ... 之后不能再跟任何形参
void Func2(int a, ..., int b);
```

对于调用带有不定参数个数的函数时所要传递的实参而言，由于不定参数列表中的每个参数类型不确定，因此C语言编译器将采用默认的实参晋升（argument promotion）机制。也就是说，对于任何整数转换等级小于int类型的实参，其类型都将被晋升为int类型；单精度浮点类型（float）的实参将被晋升为双精度浮点类型（double）。

另外，当我们要实现一个带有不定参数个数的函数时，需要借助<stdarg.h>标准头文件中的宏来迭代获取每个形参。<stdarg.h>标准头文件定义了4个宏用于遍历传入的实参列表。正如上面所述，带有不定实参个数及类型的函数的形参列表中，至少会有一个形参（即最开始的命名形参），所以在...之前的那个形参在实参访问机制中将扮演一个特殊的角色。

<stdarg.h>标准头文件中定义的va_list类型是一个完整类型，用来保存va_start、va_arg、va_end以及va_copy这4个宏函数在操作过程中所需要的状态信息，我们通常会在获取不定实参列表之前先用va_list类型来声明一个对象。我们也能将va_list声明的实参传递给另一个函数，如果它的最后一个形参为va_list类型的话。为了叙述方便，我们假定这里声明了一个va_list对象，名为ap。然后下面对4个宏函数的介绍中都用ap作为va_list声明的对象。

在开始访问...所对应的实参之前，必须先调用va_start宏。va_start对ap进行初始化，这样ap才能后续为va_arg、va_end等宏所使用。这里大家需要注意的是，对于同一个不定参数类型与个数的参数列表而言，va_start只能被调用一次。va_start的第二个参数需要传，...之前的那个形参对象，以定位不定参数列表从哪个命名参数开始起获取。

va_arg宏扩展为一个表达式，用于指定在函数调用中下一个实参的类型与值。va_arg宏的第一个参数为ap，在每执行一次va_arg时，ap会被自动修改为下一个实参的值，然后返回。因此，va_arg返回的是当前ap所指定的下一个实参的值，同时ap也会指定到下一个实参位置。第二个形参应该是一个类型名，该类型名与调用者传入的实参类型对应。这里各位要注意的是，由于调用者传入的实参会做默认的实参晋升，所以va_arg的第二个参数不能是char、_Bool、short等低于int类型转换等级的整数类型，如果实参传入的是这些类型的对象，那么这些对象会被自动晋升为int类型。因此要获取char、short等类型实参时，都要用va_arg（ap，int）。如果实参传入

的是float类型对象，那么会被自动晋升为double类型。

当获取完不定参数列表后，调用va_end宏来无效化ap，这样ap之后就不可再使用了。

va_copy宏初始化其第一个参数dest作为其第二个参数src的一个拷贝，两者都是va_list类型的对象。如果src对象已经通过va_start进行了初始化，并且通过几次va_arg的迭代，那么这就好比先对dest调用了va_start宏，然后迭代地调用va_arg宏，直到dest的状态与src的状态相同。

代码清单9-10将给出定义与调用不带参数个数的函数的方法以及使用这些宏的详细方法。

代码清单9-10　不带参数个数及类型的函数的定义与调用

```
#include <stdio.h>
#include <stdarg.h>

// 定义一个含有一个int形参，再跟一个不定参数类型及个数的形参列表
static void Mytest1(int n, ...)
{
    // 首先声明va_list类型的对象ap
    va_list ap;

    // 对ap初始化，并指明形参n是紧跟在 ... 之前的形参
    va_start(ap, n);

    // 开始获取第一个实参，其类型为int
    int a = va_arg(ap, int);

    // 迭代获取第二个实参，其类型为unsigned
    unsigned b = va_arg(ap, unsigned);

    // 迭代获取第三个形参，其类型为double
    double d = va_arg(ap, double);

    // 结束迭代，对ap无效化处理
    va_end(ap);

    printf("result = %f\n", n + a + b + d);
}

// 这里定义函数MyFunc，其第二个形参为va_list类型
static double MyFunc(int n, va_list ap)
```

```
{
    // 在此函数中，无需对ap做va_start初始化以及va_end的无效化
    int a = va_arg(ap, int);

    unsigned b = va_arg(ap, unsigned);

    double d = va_arg(ap, double);

    return n + a + b + d;
}

static void MyTest2(int n, ...)
{
    va_list ap;

    va_start(ap, n);

    // 这里将初始化完的ap对象作为实参，传递给MyFunc
    double result = MyFunc(n, ap);

    va_end(ap);

    printf("result = %f\n", result);
}

struct MyStruct { int a, b; };
union MyUnion { char c; short s; };

// 我们也可以用自定义类型作为不定参数类型与个数的参数列表的实参
static void MyTest3(int a, ...)
{
    va_list ap;

    va_start(ap, a);

    // 这里指定了MyStruct结构体类型
    struct MyStruct s = va_arg(ap, struct MyStruct);

    // 尽管un的大小为2个字节，但一般C语言实现都会对它做默认的晋升，
    // 因此这里指定union MyUnion也没有关系
    union MyUnion un = va_arg(ap, union MyUnion);
    printf("size of un: %zu\n", sizeof(un));

    va_end(ap);

    int result = a + s.a + s.b - un.s;
    // 输出：result = 10
    printf("result = %d\n", result);
}

int main(int argc, const char * argv[])
{
    int8_t a = 10;
    uint16_t b = 20;

    // 实参a的类型将被晋升为int
    // 实参b的类型将被晋升为unsigned
    // 实参10.5f的类型将被晋升为double
    Mytest1(3, a, b, 10.5f);

    MyTest2(5, a, b, 10.5f);

    // 这里将一个结构体对象与一个联合体对象作为不定参数的实参传递
    MyTest3(10, (struct MyStruct) { 1, 2 }, (union MyUnion) { .s = 3 });
}
```

代码清单9-10中，MyTest1函数完整地通过一般的方式来获取不定参数列表部分的实参，而MyTest2函数则是借助MyFunc来获取其不定参数个数列表部分的实参。像我们经常使用的printf函数就是一个典型的带有不定参数个数与类型的库函数。该函数在实现中通过第一个实参的字符串格式符来解析后续每个实参的类型，从而可以恰当地获取实参的值。

9.5　函数的递归调用

C语言的函数调用有一个十分有趣的特性，就是可递归调用。什么是递归调用？其实在我们中学数学课上就有所接触了。比如，函数f（x）=x*f（x-1），我们称f（x）为一个递推方程。如果把它映射到C语言中，那么函数f（x）就是递归调用的，也就是在计算这个函数的时候借用了该函数本身。

在一个函数中的某个位置调用该函数自己，这也被称为直接递归调用。如果函数A在某个位置调用了函数B，而函数B在某个位置处又调用了函数A，那么这被称为间接递归调用。

下面我们用代码清单9-11来举一个简单的例子来看看，函数递归调用是如何执行的。

代码清单9-11　函数递归调用简介

```
#include <stdio.h>

static void Func(int n)
{
    // 如果n等于0，则直接返回
    if(n == 0)
    {
        puts("last level!");
        return;
    }

    // 打印当前形参n的值，以确定现在是第几层递归调用
    printf("n = %d\n", n);

    // 递归调用Func，并且将n - 1作为实参传入
    Func(n - 1);

    puts("call over");
}
```

```
int main(int argc, const char * argv[])
{
    Func(3);
}
```

代码清单9-11中定义了一个Func函数，在其内部实现中它做了递归调用。在每次递归调用时，都会先重新进入Func函数，直到最内部的调用返回，然后从里到外逐个进行调用返回。在main函数中用实参3来调用Func函数，因此最后输出结果是：

```
n = 3
n = 2
n = 1
last level!
call over
call over
call over
```

下面我们用图9-4来描述调用Func函数之后的整个控制流的执行。

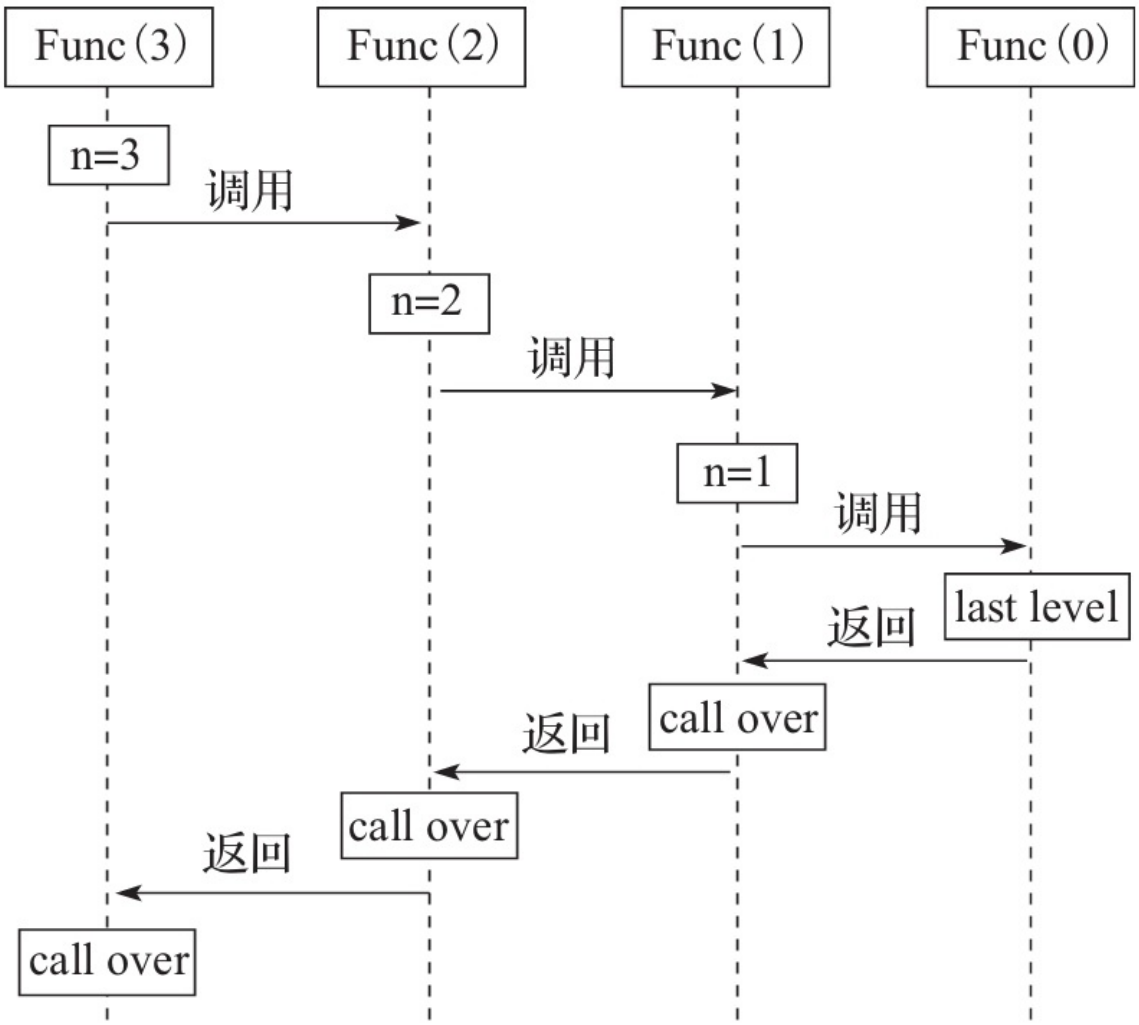


图9-4　函数递归调用的执行流程

图9-4中，矩形方框中的内容表示当前函数中要执行打印的内容。我们通过这个顺序图可以清晰地看到，当函数Func做递归调用时，它就把执行控制权直接交给了新的被调的Func，只有等最后被调的Func返回之后，之前被递归调用的Func才能再获得执行权继续执行。我们看到调用与返回非常具有层次感。

下面，我们将列举若干实例来进一步说明递归函数的使用方式，以及

使用递归函数的优缺点。

我们先以比较简单的阶乘算法来介绍一个递归函数实现的具体算法方式。我们在中学时应该学过阶乘的表达方式，即f（n）=n*（n-1）*（n-2）*...*1。而如果当n为0时，则f（0）=1。所以这可以用简单的递推式来表达——当n==0时，f（0）=1；否则，f（n）=n*（n-1）。而这种递推形式的函数表达式就能方便地用C语言代码来表达了。代码清单9-12展示了阶乘算法的递归实现与循环迭代实现两种方式。

代码清单9-12　阶乘的递归实现与循环迭代实现

```
#include <stdio.h>

int FactorialRecursion(int n)
{
    if(n < 1)
        return 1;

    // 这句表达式语句正如我们所提到的类似：f(n) = n * f(n - 1)这种形式
    return n * FactorialRecursion(n - 1);
}

int FactorialIteration(int n)
{
    if(n < 1)
        return 1;

    for(int i = n - 1; i > 0; i--)
        n *= i;

    return n;
}

int main(int argc, const char * argv[])
{
    int result = FactorialRecursion(5);

    printf("result = %d\n", result);

    result = FactorialIteration(5);

    printf("result = %d\n", result);
}
```

代码清单9-12中，第1个函数FactorialRecursion用的是以递归调用的方

法实现的阶乘计算，第2个函数FactorialIteration则是用循环迭代的方法实现的阶乘计算。从表达上来看，我们可以很明显地看到，采用递归的方式比采用循环的方式要简洁很多。从执行效率上看，由于每次做递归调用都需要做当前函数的上下文保护，所以难免会做一些堆栈操作；此外还有函数调用本身会对处理器的执行流水线造成一定影响，因此运行性能肯定比直接循环迭代来得低些。

通过上面的代码清单9-12，我们能看到，在表达上递归调用形式比循环迭代更为简洁，而在运行效率上则是循环迭代更有优势。在这种单一线性数据处理上我们很容易使用循环迭代来代替递归调用；然而，如果是树状数据处理顺序，那么用循环就很难去表达了。这个时候我们更倾向于直接使用递归调用来处理数据。代码清单9-13将为大家呈现更为复杂的递归函数调用方式。

代码清单9-13　更复杂些的函数递归调用

```
#include <stdio.h>

void ListNumber(int n)
{
    if(n > 9)
    {
        printf("10\n");
        return;
    }
    if(n < -9)
    {
        printf("-10\n");
        return;
    }

    printf("%d  ", n);

    ListNumber(n * 2);
    ListNumber(-n * 2);
}

int main(int argc, const char * argv[])
{
```

```
    ListNumber(2);
}
```

代码清单9-13的计算输出结果为:

```
2  4  8  10
-10
-8  -10
10
-4  -8  -10
10
8  10
-10
```

这种结果输出显然难以用单纯的循环迭代来表达，因为它是一种树状输出。我们通过图9-5来列出结果输出过程。

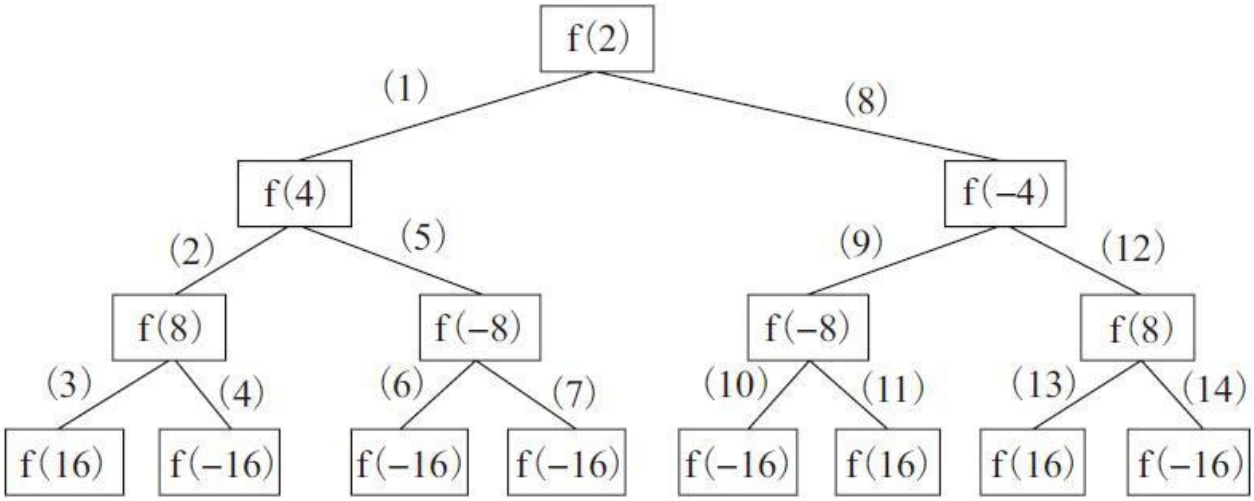


图9-5　树状递归执行顺序图

图9-5中，矩形方框中的f就表示代码清单9-13中的ListNumber函数。直线上的用圆括号包围的数字表示当前调用次序，比如（1）表示第1次重新进入ListNumber函数，（5）表示在第5次重新进入ListNumber函数。显然，如果要用循环迭代来表示这种树状执行次序将会十分复杂。所以，这

里用递归形式表达不仅简洁，而且更有效率。

最后，我们再举一个斐波那契数列的例子来讲一下函数递归调用。斐波那契数列指的是这么一个数列：0，1，1，2，3，5，8，13，21……其中0属于第0项，1属于第1项。从第2项开始，当前项的数是其之前两项数的和，所以可以用这个数学方程来表达：Fn=F（n-1）+F（n-2），n>1且n属于自然数。代码清单9-14展示了分别使用循环迭代法以及函数递归调用的方法来求得第n项的斐波那契数列的值。

代码清单9-14　获取斐波那契数列的指定项的值

```
#include <stdio.h>

/** 用循环迭代实现获取斐波那契数列第nItem项的值 */
static int FibonacciIteration(unsigned nItem)
{
    // 这里声明的former表示一开始作为第0项的值，current则表示第1项的值
    int former = 0, current = 1;

    if (nItem == 0)
        return former;
    else if (nItem == 1)
        return current;

    for (int n = 2; n <= nItem; n++)
    {
        // 计算当前项的值
        int newValue = former + current;
        // 将前一项的值赋值给前第二项
        former = current;
        current = newValue;
    }

    return current;
}

/** 用函数递归调用实现获取斐波那契数列第nItem项的值 */
static int FibonacciRecursion(unsigned nItem)
{
    if (nItem == 0)
        return 0;
    else if (nItem < 2)
        return 1;

    // 递归调用FibonacciRecursion，求得当前第nItem项的值
    return FibonacciRecursion(nItem - 1) + FibonacciRecursion(nItem - 2);
}

int main(int argc, const char * argv[])
```

```
{
    int value = FibonacciIteration(8);
    printf("value = %d\n", value);
    // 第8项的值为21
    value = FibonacciRecursion(8);
    printf("value = %d\n", value);
}
```

图9-6展示了递归调用代码清单9-14中的FibonacciRecursion（5）的调用流程。

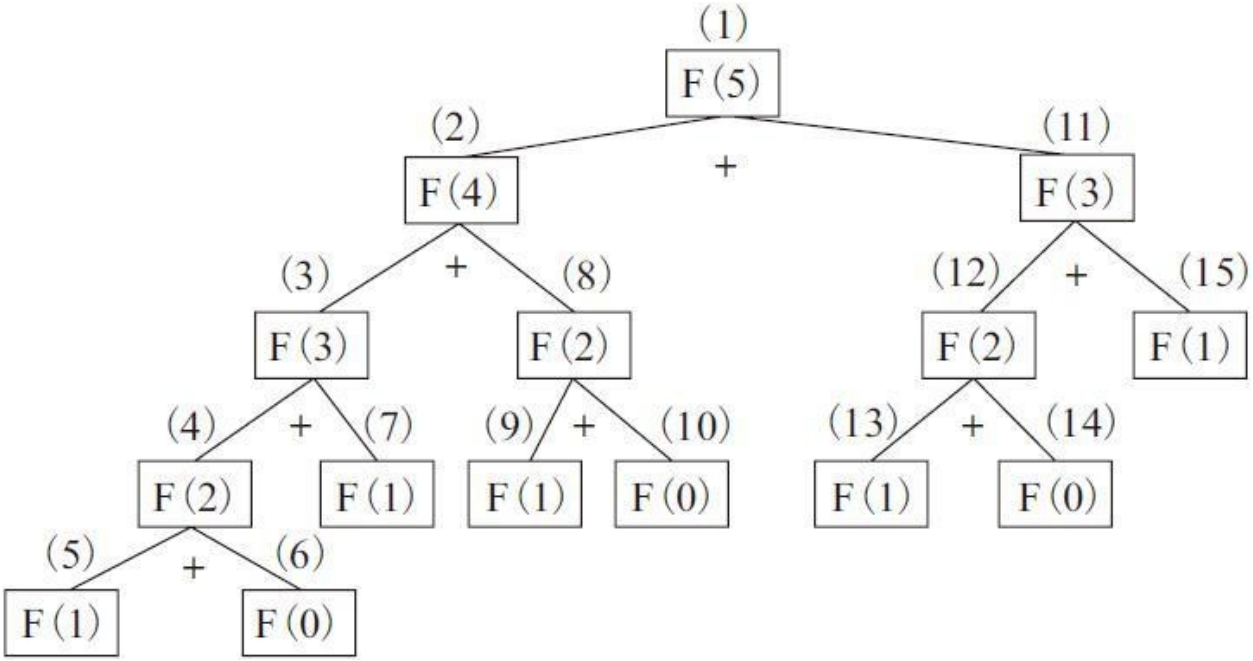


图9-6　斐波那契数列的函数递归调用流程图

图9-6中，矩形框表示当前调用的FibonacciRecursion函数，其上方的数字表示当前调用是第几次调用，比如（1）表示第一次调用，（2）表示第二次调用。从这个图中我们也能容易发现，某一次函数调用的结果就是它下面两次调用结果的和。所以FibonacciRecursion（5）函数调用的最终结果就是5。

9.6 内联函数

从C99标准开始起，inline关键字被正式纳入C语言标准。在C11标准中，inline与下一小节将描述的_Noreturn都属于函数说明符（function specifier）。C11标准中明确指出，函数说明符应该仅用在对一个函数标识符的声明中。

如果一个函数用inline函数说明符进行声明，那么该函数是一个内联函数。内联函数是对C语言编译器的暗示，建议编译器对该函数的调用尽可能地快。

具有内部连接的任一函数都可以作为一个内联函数。而对于具有外部连接的函数则具有以下限制：如果一个函数用inline函数说明符进行声明，那么它也应该定义在同一翻译单元中。如果对一个函数在所有文件作用域的声明，在某一翻译单元中包含了inline函数说明符，而没有extern，那么在该翻译单元中的定义是一个内联定义。内联定义不提供对该函数的外部定义，并且也不禁用它在另一个翻译单元中的外部定义。内联定义提供了对一个外部定义的替代品，编译器可以用来实现在同一翻译单元中对该函数的任一调用（选择内联定义的调用或外部定义的调用）。对函数的调用使用的是内联定义还是外部定义则是未指定的。

具有外部连接的一个函数的内联定义，不应该包含具有静态或线程存储周期的一个可修改对象，并且也不应该包含对一个具有内部连接标识符

的引用。

代码清单9-15展示了内联函数的使用以及一些注意事项。

代码清单9-15　内联函数的定义与使用

```
// main.c源文件
#include <stdio.h>

// Func是一个具有内联定义的函数
inline int Func(int n)
{
    return n * 2;
}

extern inline int Func2(int n)
{
    // 对于具有外部连接的一个函数的内联定义，不应该包含可修改的静态存储周期对象。
    // 这里编译器可能会报出警告
    static int s;

    s += n;

    return s + n;
}

static inline int Func3(int n)
{
    // 对于具有内部连接的一个内联函数，可以包含可修改的静态存储对象
    static int s;

    s += n;

    return s + n;
}

// MyTest函数定义在hello.c源文件中
extern void MyTest(void);

int main(int argc, const char * argv[])
{
    int result = Func(3);
    printf("result = %d\n", result);

    MyTest();

    // 调用完MyTest()函数之后，Func2函数中静态对象s的值变为了20
    result = Func2(10);
    printf("result in main is: %d\n", result);

    // 在调用Func3函数之前，它所包含的静态对象s的值为0
    result = Func3(1);
    printf("result 1 = %d\n", result);

    // 在调用了一次Func3函数之后，它所包含的静态对象s的值为1
    result = Func3(2);
    printf("result 2 = %d\n", result);
}
// hello.c源文件
#include <stdio.h>
```

```
// 这里将Func定义为具有外部连接的一个函数
int Func(int n)
{
    return n * 3;
}

inline int Func2(int n)
{
    static int s;

    s += n;

    return s + n;
}

void MyTest(void)
{
    printf("value is: %d\n", Func(2));

    int result = Func2(20);
    printf("result in MyTest is: %d\n", result);
}
```

代码清单9-15分为两个源代码，第一个是main.c，第二个是hello.c。这两个源文件放在同一工程内，然后编译后连接生成可执行文件。在main.c中，定义了一个具有内联定义的Func函数（它没有用extern修饰，仅具有inline函数说明符），而在hello.c中则定义了相同函数名Func的外部定义，在连接时不会引发符号重定义的冲突。但是，在main.c的Func定义中，倘若在inline前面添加extern，则会引发Func符号重定义的连接错误。因为此时它真具有了外部连接，而不是一个仅具有内联定义的函数。

在main函数中，第一条调用Func函数的语句，它可以被编译器直接翻译为：int result=3*2；。内联函数的作用就是建议编译器以最快的方式调用函数。那么将函数中的语句内容直接扩展出来，使得与当前函数调用者的上下文结合进行优化，无疑是最快的方式。当然，inline仅仅起到建议的作用，内联函数的调用是做代码展开还是直接做普通的函数调用完全由编译器做最后的决定。

函数Func2与Func3则呈现了上文所描述的对内联函数中包含静态存储周期对象的限制。Func2之所以不能在函数体内包含静态对象，是因为内联定义的函数其本质上仍然是外部的，也就是说尽管它可以出现在不同的翻译单元，但它仍然只具有一个实体。也就是说，其内部定义的静态对象对于整个执行程序而言也是唯一的，所以它在不同源文件中的调用，其内部静态对象s的值会受到影响（即具有不可见的副作用）。而具有static存储类的内联函数本身就具有内部连接，因此每个翻译单元具有一个独立的对象副本，所以在多个源文件中进行调用时相互之间不会有任何影响。对函数中包含静态存储周期对象的情况的详细介绍，请参考11.3节。

9.7　函数的返回类型与无返回函数

前面几节讲解了函数的形参与实参的传递以及函数的调用等，本节将详细描述函数的返回。

在C语言中，函数的返回类型几乎可以是任意类型，包括整型、浮点型等基本类型，枚举、结构体、联合体等用户自定义类型，也可以是指向上述这些类型的指针，但唯独不允许数组类型。除了返回类型为void的情况外，一个函数中的任一分支代码最终必须要触碰一条return语句进行函数返回。对于返回类型为void的函数，在其函数体结尾处会默认隐含一条return语句。当函数体中出现return语句时，return后面跟着的表达式的类型必须要与函数返回类型兼容，如果return后面是一条空表达式（比如直接以分号结尾），那么表示返回的是一个void表达式。

代码清单9-16展示了函数返回的一些常用技巧。

代码清单9-16　C语言函数返回的常用技巧

```
#include <stdio.h>

// 这里定义了一个返回类型为int的函数
static int MyIntFunc(int a)
{
    // 确保函数最终执行完时都要触碰一条return语句。
    // 这里的if、if-else、以及else语句后面的return语句都不能省，
    // 如果else后面的return 0省了之后，
    // 在a等于0的情况下函数的返回值是不确定的
    if(a > 0)
        return 100;
    else if(a < 0)
        return -100;
    else
        return 0;
}
```

```
// 这里定义了一个返回类型为int*的函数
static int* MyIntPtrFunc(void)
{
    static int s = 100;

    return &s;
}

// 这里定义了一个返回类型为一个结构体的函数
static struct SA { int a; float f; } MyStructFunc(void)
{
    return (struct SA){ 100, -10.25f };
}

// 这里定义了一个返回类型为一个枚举的函数
static enum { MY_ENUM1, MY_ENUM2 } MyEnumFunc(void)
{
    return MY_ENUM2;
}

static void MyVoidFunc(int a)
{
    if(a > 0)
    {
        puts("a is above zero!");
    }

    // 这里使用(void)投射操作将表达式转为void表达式。
    // 这里要注意的是，printf的返回类型为int，而不是void
    return (a > 10)? (void)printf("a = %d\n", a) : (void)0;
}

int main(int argc, const char* argv[])
{
    int a = MyIntFunc(1);
    printf("a = %d\n", a);

    int *p = MyIntPtrFunc();
    printf("*p = %d\n", *p);

    struct SA sa = MyStructFunc();
    printf("sum of sa is: %.2f\n", sa.a + sa.f);

    a = MyEnumFunc();
    printf("enum is: %d\n", a);

    MyVoidFunc(100);
}
```

C11标准引入了_Noreturn关键字，它也是函数说明符。如果一个函数用一个_Noreturn函数说明符来声明，那么该函数不应该返回给其调用者。所以用_Noreturn修饰的函数，其返回类型应该是void类型。此外，C11标准也引入了<stdnoreturn.h>头文件，其中将_Noreturn定义为了noreturn，因此我们应该尽量包含<stdnoreturn.h>头文件，然后直接使用noreturn。

在应用端开发中，_Noreturn很少使用。该函数一般用于嵌入式系统中某些异常处理例程，或是用于线程处理例程，倘若这些例程（routine）不做任何返回的话。在C11标准库中，像longjmp、abort、exit、quick_exit等库函数均以_Noreturn进行声明。代码清单9-17将说明_Noreturn的一些使用方式与规则。

代码清单9-17　_Noreturn的使用

```
#include <stdio.h>
#include <stdlib.h>
#include <stdnoreturn.h>
// 用noreturn声明了函数Routine。
// 在Routine函数内不能出现任何return语句
noreturn void Routine(int n)
{
    if(n > 0)
    {
        printf("n = %d\n", n);

        // 这里将中止程序执行，应用程序执行到abort()时将会引发异常
        abort();
    }
    else
    {
        // 这里退出整个程序的执行
        puts("Exit!");
        exit(n);
    }
}

int main(int argc, const char * argv[])
{
    Routine(-10);
}
```

代码清单9-17中所调用的abort（）以及exit（）库函数的原型都声明在<stdlib.h>头文件中。

9.8 指向函数的指针

C语言中的指针是一个非常灵活、强大、适用范围广的属性，一个指针可指向几乎所有对象类型，它也能指向任何函数。其实在C语言中，一个函数标志用于表达式时就已经表征了一个指向该函数类型的指针。

比如我们声明了一个函数Func——“void Func（void）；”那么对于函数调用表达式——Func（）而言，其实这里的Func后缀表达式就已经表示了一个指向返回类型为void，且参数列表为空的函数的指针，该类型表示为：

```
void (*)(void)
```

函数指针类型的通用表达形式为：

```
返回类型（* cv限定符可选）（形参列表）
```

指针对象的标识符放在可缺省的cv限定符之后、“）”之前。cv限定符即为const、volatile限定符，这些将在第12章中详细介绍。另外，对于有些C语言实现含有函数调用约定的，那么在函数指针类型中将函数调用约定说明符放在*前面，比如void（__stdcall（*）（void）。函数调用约定详见第15章。

声明一个指向函数的指针对象时，其形参列表与返回类型的要求与一

般函数声明的要求一样，返回类型以及形参类型可以是不完整类型。通过函数指针对象可以立即对它所指向的函数进行调用（这在处理器中对应的是寄存器间接调用指令）。一个函数标志本身即可表示为一个指向该函数类型的指针，而如果在它前面加地址操作符&，同样也表示指向该函数的指针，两者在类型上是完全等同的。也就是说，如果有：void Func（void），那么Func与&Func的类型都为void（*）（void）类型。

此外，正如我们很早之前所说的，任一对象都有其地址，那么指向函数的指针对象也不例外，如果对指向函数的指针对象取了其地址，那么其类型表示为在（）中的“*”之前、“（”之后再加一个“*”。比如，如果存在一个指向函数的指针对象——void（*pFunc）（void），那么&pFunc的类型即为void（**）（void）。这里要注意的是，pFunc与上面描述的Func是不同的，因为Func本身是一个函数标志，所以&Func仍然可表示为指向一个函数的指针，以至于它与Func在类型上是等同的；而pFunc本质上就是一个指向函数的指针对象标识符，所以&pFunc就是指向函数指针对象的指针。

代码清单9-18展示了指向函数指针的用法。

代码清单9-18　指向函数的指针示例

```
#include <stdio.h>
#include <stdarg.h>

// 这里仅声明了MyStruct结构体类型
struct MyStruct;

// 这里声明了一个静态的指向函数的指针p,
// 所指向的函数其返回类型是一个不完整类型MyStruct类型,
// 其形参s与a同样也都是不完整类型。
```

```
// 这里形参中可加标识符，当然形参标识符也可缺省，这与声明函数原型时一样
static struct MyStruct (*p)(struct MyStruct s, int a[*]) = NULL;

// 这里声明函数Test，在main函数下面定义
static void Test(void);

// 这里定义了Foo函数，它含有一个不定参数个数与类型的参数列表
// 其实现为对不定参数进行求和运算，然后将结果返回
static int Foo(int n, ...)
{
    va_list ap;
    va_start(ap, n);

    int sum = 0;
    for(int i = 0; i < n; i++)
        sum += va_arg(ap, int);

    va_end(ap);

    return sum;
}

int main(int argc, const char * argv[])
{
    // 这里声明了指向函数Test的指针对象pf
    void(*pf)(void) = Test;

    // 通过函数指针pf间接调用Test函数，这里也可以使用pf()进行调用。
    // 由于函数调用操作符()的优先级大于间接操作符*，
    // 所以这里需要使用(*pf)将它作为整体，作为函数标志的后缀表达式
    (*pf)();

    // 这里声明了指向函数的指针pFunc，其形参标识符缺省，
    // 并用Foo函数地址对它进行初始化。这里的&可缺省
    int(*pFunc)(int, ...) = &Foo;

    // 这里用指向函数的指针pFunc进行间接的函数调用
    int result = pFunc(3, 10, 20, 30);
    printf("result = %d\n", result);

    // 这里声明了指向函数指针对象的指针
    int(**pp)(int, ...) = &pFunc;

    // 这里通过指向函数指针的指针pp做Foo函数的间接调用
    result = (*pp)(5, 1, 2, 3, 4, 5);
    printf("result2 = %d\n", result);

    // 将pp所指对象的值置空，使得pFunc对象的值为空
    *pp = NULL;

    if(pFunc == NULL)
        puts("Null!");
}

// 这里对MyStruct进行定义
struct MyStruct
{
    int a;
    float f;
};

static struct MyStruct Func(struct MyStruct s, int a[])
{
    printf("sum = %f\n", s.a + s.f + a[0]);

    return s;
}

// 对Test函数做定义
```

```
static void Test(void)
{
    // 在文件作用域声明的指向函数的指针p指向Func函数
    p = Func;

    // 通过指向函数的指针p进行间接函数调用
    p((struct MyStruct){.a = 10, .f = 0.5f}, (int[]){1, 2, 3});
}
```

代码清单9-18列出了指向函数的指针的各种使用方式。此外，指向函数的指针对象也能作为一个函数的参数，同时一个函数的返回类型也可以为指向函数的指针类型。比如：void（*func（int（*p）（void））（int）；就声明了一个返回类型为void（*）（int）、带有一个类型为int（*）（void）形参的函数func。

9.9　C语言中的主函数main

在C语言中，将应用启动时调用的函数命名为main函数。C语言实现不需要对此函数做原型声明。它应该被定义为返回类型为int，并且不带任何形参的函数，如：int main（void）{/*...*/}；或者带有两个形参的函数，如：int main（int argc，char*argv[]）{/*...*/}。这两个形参所对应的实参是在执行该程序时传入的。argc一般存放执行当前程序时输入的命令字符串个数；argv则存放了指向各个输入字符串的指针。假设我们现在对C源文件编译构建后，生成了一个名为test的可执行文件。那么我们在控制台中输入test arg1 arg2，再按回车，那么此时，main函数的第一个参数argc的值为3，因为test其实就属于要传入到argv数组的第1个参数，然后后面跟着2个命令行参数arg1和arg2；所以argv对应的实参内容为：{“test”，“arg1”，“arg2”}，即由应用程序名与其命令行参数字符串所构成的数组。如果将main函数声明为带有两个形参的形式，那么它们应该遵循以下约束：

1）argc的值应该是一个非负数。

2）argv[argc]应该是一个空指针。

3）如果argc的值大于零，那么数组成员argv[0]到argv[argc-1]应该包含指向字符串的指针，这些字符串在程序启动前由主机环境给出。如果argc的值大于1，那么argv[1]到argv[argc-1]所指向的字符串才表示程序形参，即

当前应用名后面的命令行参数。

4）形参argc和argv以及argv数组所指向的字符串可以在程序中修改，并且在程序启动和终止之间保留其最后被修改的值。

main函数的返回值相当于调用库函数exit所传入的实参值。如果main函数中缺省return语句，那么默认为return 0。

代码清单9-19展示了main函数参数的使用。

代码清单9-19　main函数的执行

```
#include <stdio.h>

int main(int argc, const char * argv[])
{
    if(argc == 0)
        return -1;

    // argv[0]指向的字符串为当前程序名
    printf("The program name is: %s\n", argv[0]);

    // 以下依次输出程序名后面的参数名
    for(int i = 1; i < argc; i++)
        printf("arg %d: %s\n", i, argv[i]);

    // 我们可以对argc、argv参数进行任意修改
    argc = 10;

    argv[0] = "hello";
    argv[1] = “world";
    argv = NULL;

    // 缺省return语句，这里默认为return 0
}
```

假定我们将代码清单9-19编译构建后生成了test可执行文件。然后在控制台下，进入到test当前路径，再输入test arg1 arg2，那么结果将会输出：

```
The program name is: test
arg 1: arg1
arg 2: arg2
```

这里，程序默认的入口函数就是main函数，因此main函数必须要有外部连接，它不能是static的。此外，main函数也不能带有inline、_Noreturn等函数说明符。

9.10　函数与函数调用作为sizeof操作符

C语言标准明确规定，sizeof操作符不应该应用于：①具有函数类型；②一个不完整类型的表达式；③访问一个位域成员的表达式。_Alignof操作符不应该应用于一个函数类型或不完整类型。这里大家要注意的是，当一个函数标志作为sizeof或_Alignof的操作数时，它不会被隐式转换为指向该函数类型的指针类型，这个与它单独用于其他计算表达式有所不同。因此，假定我们定义了一个函数：void Foo（void），那么sizeof（Foo）的结果是未定义的；而sizeof（&Foo）是合法的，其结果相当于sizeof（void（*）（void）），也就是一个指针对象大小。

如果sizeof的操作数是一个函数调用表达式，那么它的结果相当于sizeof（函数返回类型），同时，作为sizeof操作数的函数调用将不会发生。由于函数返回类型不能是一个可变修改类型，因此这里不会涉及在运行时对可变修改类型对象所占存储空间大小的计算。

代码清单9-20展示了sizeof作用域函数的一些代码。

代码清单9-20　sizeof与代码

```
#include <stdio.h>

static int Func1(void)
{
    puts("Func1");
    return 0;
}

static void Func2(void)
{
```

```
    puts("Func2");
}

int main(int argc, const char * argv[])
{
    // 这里由于Func1返回的是int类型，所以sizeof的结果相当于sizeof(int)
    size_t size = sizeof(Func1());
    printf("Func1() size is: %zu\n", size);

    // 由于Func2的返回类型是void，它属于不完整类型，
    // 因此理论上这里的sizeof结果在标准里是未定义的，
    // 在GCC与Clang的实现上，结果为1
    size = sizeof(Func2());
    printf("Func2() size is: %zu\n", size);

    // 这里将Func1，一个函数标识作为sizeof操作数，其行为是未定义的
    size = sizeof(Func1);
    printf("Function size is: %zu\n", size);

    // 这里对&fFunc1，即一个函数指针类型作为sizeof操作数，
    // 该值与sizeof(void(*)(void))相同
    size = sizeof(&Func1);
    printf("Function pointer size is: %zu\n", size);
}
```

9.11　本章小结

本章主要介绍了C语言的函数，对函数返回类型、形参以及函数调用与实参传递等知识进行了全方位的讲解。通过对本章的学习，各位应该能理解并自己会写函数，将自己的一些功能逻辑给模块化、抽象化。在本章中，比较难以理解的可能就属函数递归调用了，如果大家是专攻于一些数学算法的，希望能再好好消化一下。

第10章　C语言预处理器

C语言编译器前端还分为预处理阶段与编译阶段。预处理阶段是通过C语言的各类预处理器将指定的一些字符符号直接替换到即将编译的源代码。预处理器在跨平台整合上很有帮助，我们一般可以利用预处理器针对有差异的系统平台安插不同的源代码。除此之外，当前遵循C11标准的预处理器的功能已经十分强劲，我们可以利用预处理器生成灵活强大的代码，可以把某些代码逻辑通过宏定义来高度抽象化。

用于预处理的指示符称为预处理指示符（preprocessing directives），C语言主要有三大类预处理指示符——条件段（if-section）预处理指示符、控制行（control-line）预处理指示符、空指示符（null directive）。本章将会为大家介绍条件预处理、文件包含、宏替换、行控制、错误指示符、编译指示（pragma）指示符、空指示符以及C11标准中预定义的宏名。

对于任意一条预处理指示符，除了_Pragma之外，其他的都必须是以#符号打头，并且#符号必须出现在每一行的最前面。也就是说，如果我们要在源文件中某一行使用一条预处理指示符，那么最开头就得写上#，前面除了空白符之外不允许出现其他任何字符。一条预处理指示符的最后都无需加分号。对于预处理组（比如条件段预处理指示符）而言，其作用范围从组的起始指示符开始的下一行一直到该组结束指示符的上一行。对于控制行预处理器指示符（比如宏定义）而言，一行就定义了某个符号或执行某

个动作，它们不作用到下一行。

 注意：C语言的预处理器拥有自己独立的文法，我们可以将它看作为嵌入在C源代码中的一段编译指示脚本，用于在当前源文件中构建指定的后续要进行编译的C源代码。所以大家不要将它与之前描述的C语言的一些语法特征给搞混，而使用预处理器就好比在使用元编程（metaprogramming）。

10.1 宏定义

宏定义属于控制行预处理指示符。以#define定义的一个符号称为宏（macro）。这里，#与define之间可以存在空白符（换行符除外），但对于某些C代码编辑器而言，一旦#与define之间存在空白符，可能会导致编辑器在词法上无法识别，从而无法获得该有的语法高亮。因此建议各位在用以#打头的任何预处理指示符的时候，#与跟在它后面的指示符之间不要有任何空白符。

在C语言中，宏的定义有两种形式，一种是类似对象的宏定义，另一种是类似函数的宏定义。类似对象的宏定义的形式为：

```
# define    标识符    替换列表    换行符
```

类似函数的宏定义形式为：

```
# define    标识符(参数列表)    替换列表    换行符
```

以上定义中，“替换列表”可缺省。这里要注意的是，类似函数的宏定义中，标识符与（之间不应该存在任何空白符，否则预编译器可能会将（）作为类似对象宏定义的替换列表中的一部分。

在C语言预处理过程中，替换列表会将当前宏标识符给完全替换掉。我们称两个替换列表是完全等同的，当且仅当该两个替换列表中的预处理

符号（preprocessing token）的数量、次序、拼写以及空白分隔符的数量相同，对于所有空白分隔符都认为是等同的（比如一个tab制表符与一个空格符是完全相同的，因此N个空格符与一个空格符是等同的，当然这里N必须大于零）。

对类似函数的宏（以下简称为宏函数）的“调用”与一般C函数调用还有一点不同，即宏函数的实参可以不传，此时在宏替换时会使用占位标记（placemarker）预处理符号来代替。占位标记预处理符号在C语言语法上不会体现出来，它作为C语言预处理器的一种标准实现方式进行定义。

宏定义的作用范围是从它定义完的那个位置起一直到当前源文件结束，它不受语句块作用域、函数作用域等影响，因为正如本章开头所提到的，预处理部分与C源代码正文部分采用的是完全不同的文法体系，而且预处理器（preprocessor）是独立于编译器而存在的。因此从严格意义上来说，我们在使用类似函数的宏的时候也不能将它称之为“调用”，所以后续统一采用“使用”。

前面谈了关于宏的基本概念以及一些注意事项之后，下面我们就来谈谈宏的基本使用。

10.1.1 宏的基本使用

前面提到，#define用于定义一个宏，当在源代码中使用宏的时候，宏

在预编译处理期间会被替换为其替换列表中的内容。下面我们将通过代码清单10-1举一些例子来初步说明宏的定义以及使用方式。

代码清单10-1　宏的初步使用

```
#include <stdio.h>

// 这里定义了一个缺省替换列表的宏MY_MACRO
#define MY_MACRO

// 这里定义了一个宏对象MY_MACRO1，其替换列表为：100
#define MY_MACRO1       100

// 这里定义了一个宏对象MY_MACRO2，其替换列表为：10 + a
#define MY_MACRO2       10 + a

// 这个宏对象直接定义了一个函数
// 如果替换列表中的代码较长，那么可以用\后面紧跟换行符来进行换行，
// 这里要注意的是，\后面不应该再跟其他空白符，而是要直接紧跟换行符
#define MY_MACRO3       static int Foo(void) {  \
return 1;           \
}

// 这里使用了宏MY_MACRO3，因此在预处理期间，这里会将MY_MACRO3扩展为上述替换列表中的函数定义。
// 所以这里使用MY_MACRO3，就相当于安插了这段代码：
// static int Foo(void){return1;}
MY_MACRO3

// 这里定义了一个宏函数MY_SWAP，用于交换两个实参的值。
// 这里要注意，MY_SWAP与(之间不应该出现空白符。
// 这里假定x和y都是整数类型
#define MY_SWAP(x, y)   { int tmp = x; x = y; y = tmp; }

static void Dummy(void)
{
    // 这里MY_MACRO4是一个宏对象，而不是函数，
    // 因为MY_MACRO4与(之间有一个空格（空白符）。
    // 这里尽管定义在了Dummy函数的语句块作用域内，
    // 但仍然可以在当前文件作用域中任何位置使用
    #define MY_MACRO4 (a, b)
}

int main(int argc, const char * argv[])
{
    // 在预处理期间，MY_MACRO的替换不包含任何预处理符号；
    // MY_MACRO1会被自动替换为100
    int a = MY_MACRO MY_MACRO1;

    // 在预编译处理期间，MY_MACRO2会被自动替换为10 + a，
    // 因此整个表达式在编译前会变为：int b = 10 + a * 3
    int b = MY_MACRO2 * 3;

    // 这里a的值为100，b的值为10 + a * 3等于310
    printf("a = %d, b = %d\n", a, b);

    // 这里直接调用了Foo函数，由于在main函数上已经使用了宏MY_MACRO3，
    // 所以它在预处理阶段扩展后，被替换成了对Foo函数的定义
    a = Foo();
    printf("a = %d\n", a);
```

```
    // 这里使用了宏函数MY_SWAP，并且将对象标识符a和b作为其实参传入，
    // 此时预处理器在扩展时就会将MY_SWAP(a, b)替换为以下代码：
    // { int tmp = a; a = b; b = tmp; }
    // 各位可以注意到，)符号后面没有添加分号，因为}作为复合语句结尾时分号可省
    MY_SWAP(a, b)
    // 交换之后a的值为310，b的值为1
    printf("a = %d, b = %d\n", a, b);

    // 这里使用了MY_MACRO4宏对象，在预处理阶段进行扩展时会变为：a = (a, b);
    // 因此，这条表达式语句会将对象b的值赋给对象a。
    // 此外，尽管MY_MACRO4定义在了Dummy函数内，但仍然可以在main函数中使用
    a = MY_MACRO4;

    // 这里会输出OK
    if(a == b)
        puts("OK!");
}
// 错误的宏定义，在#符号之前不允许出现除空白符之外的任何其他字符
MY_MACRO #define ERROR_MACRO

// 以下宏定义没问题，由于在#之前只有空白符。
// #与define之间允许存在除了换行符以外的其他空白符
    #  define OK_MACRO
```

代码清单10-1中所展示的宏定义与宏替换尽管不算复杂，但已经能充分呈现出宏定义的形式以及宏替换后的代码样式。正如本书第一章所提到的，C语言编译器在对C源代码编译前需要先做一次预编译，也就是预处理。在预编译阶段会将预处理器符号全都替换为它所定义的替换列表中的内容，同时合并多余的空白符。在代码清单10-1中我们可以看到，在替换过程中，替换列表中的任一符号（除了某些被合并的空白符之外）都会得到保留，且原封不动地替换到即将编译的源代码中。

当然，除了我们在源代码中显式定义宏之外，编译器一般也会提供全局的宏定义。比如，当我们在使用GCC或Clang编译器时，可以使用编译选项-D，后面紧跟所要定义的宏名，然后可跟=来指定该宏的替换列表。比如，-DMY_MACRO=10这个命令选项（command option）指定了定义一个全局的宏MY_MACRO，并且该宏的替换列表为一个常量整数10。

10.1.2　宏定义中的#操作符

在一个宏函数定义中，如果在替换列表中使用#后面跟形参名，那么在宏替换时可以将此形参部分所对应的实参内容以字符串字面量的形式表示。在使用这种宏函数时，在作为实参的预处理符号中的每个空白符都会在预处理时作为字符串字面量中的一个空格符。在实参中，第一个预处理符号之前的空白符以及预处理符号之后的空白符都会在预处理时被删除。其他情况下，实参中每个预处理符号的原始拼写都会保留在宏替换之后的字符串字面量中，除非出现转义字符需要处理。如果宏替换后的结果不是一个有效的字符串字面量，那么结果是未定义的。

代码清单10-2展示了宏定义中#操作符的使用方法以及效果。

代码清单10-2　宏定义中#操作符的使用

```
#include <stdio.h>
#include <string.h>

// 这里定义了一个简单的带有#操作符的宏MY_MACRO1。
// 这里#与x之间有一个空格，这是合法的，并且与#x的效果一致
#define MY_MACRO1(x)    # x

// 这里定义了一个带有两个形参的宏函数，
// 其替换列表是将第一个实参所指定的符号作为字符串字面量与一个换行符拼接，
// 然后再与第二个实参所指定的符号作为字符串字面量进行拼接
#define MY_MACRO2(x, y)    #x "\n" #y


int main(int argc, const char * argv[])
{
    // 这里的MY_MACRO1(10ab)会被替换为"10ab"
    const char *s = MY_MACRO1(10ab);
    printf("The literal is: %s\n", s);

    // 这里字符串比较结果是相同的
    if(strcmp(s, "10ab") == 0)
        puts("Equal!");

    // 对于#操作符所作用的形参对应的实参，其前后空白符都会被删除，
    // 同时，这里的\符号在宏替换为字符串之后就变为'\\'
```

```
    s = MY_MACRO1(  10"ab\n"  );

    // 这里字符串比较结果是相同的
    if(strcmp(s, "10\"ab\\n\"") == 0)
        puts("Equal!");

    // 这里尽管第一个实参中有一个逗号，但它被包围在圆括号中，因此不作为实参分隔符。
    // 同样，在第二个参数中的逗号在''中，它作为一个字符token，因此也不作为实参分隔符。
    s = MY_MACRO2((123abc,45;'0'), [1a2b3c:?','=]);
    printf("string is: %s\n", s);

    // 这里第二个实参传的是不含任何预处理符号的实参，这里仅用一个逗号作为分隔符。
    // 此时，MY_MACRO2(abcd, )会被替换为"abcd\n"，而忽略后续与#y的替换与拼接，
    // 由于y形参对应的实参不包含任何预处理符号，因此它用占位标记预处理符号代替，
    // 并且在宏替换的最后阶段#与占位标记预处理符号的拼接被完全移除
    s = MY_MACRO2(abcd, );
    printf("s = %s\n", s);

    // 这里与上述代码类似，只不过第一个实参作为不含任何预处理符号的实参，
    // 这里仅用一个逗号作为参数分隔符使用。MY_MACRO2(, abcd)这里会被替换为"\nabcd"
    s = MY_MACRO2(, abcd);
    printf("s = %s\n", s);

    // 以下宏实参是非法的，由于它不是一个有效的预处理符号，
    // 由于出现了一个'，但没有找到另一个'与之匹配
    // s = MY_MACRO1(123'p);
}
```

代码清单10-2详细描述了宏定义中#操作符的使用方式。这里还引出了预处理符号（preprocessing token）的概念。我们可以看到，代码清单10-2中在main函数中作为宏的实参的符号（token）非常丰富，诸如10ab，它在C语言一般源代码中压根就不是一个合法的标识符，也不是一个合法的数字字面量，但它却是一个合法的预处理符号，因此可以作为宏函数的实参。

C语言标准规定了哪些元素可作为预处理符号，哪些不能。下面列出可作为预处理符号的元素：

·头文件名；

·标识符；

·预处理数字；

·字符常量；

·字符串常量；

·标点符号；

所有非空白字符，并且它们不能是以上提到的符号。

因此，像代码清单10-2中的123'p就不是一个有效的预处理符号，因为123'p既不是一个合法的预处理数字，也不是一个合法的字符常量，所以在'后面必须再要出现一个'，以至于能构成一个完整的合法字符常量才算是一个有效的预处理符号，也就是说123'p'就是一个合法的预处理符号了。而像123\p这种也不属于有效的预处理符号，因为\只能作为字符转义符号使用，一般不作为他用。如果宏函数实参中含有逗号，那么必须注意，应该要用圆括号围起来，否则逗号将作为实参分隔符的功能而使用。

此外，在代码清单10-2中我们也提到了在使用宏函数时传入缺省实参的形式，比如MY_MACRO2（abcd，）以及MY_MACRO2（，abcd）。此时，缺省的宏函数实参在宏替换时会先用占位标记预处理符号代替，然后与它前后其他预处理符号进行拼接。比如像MY_MACRO2（abcd，），在宏替换开始前就好比：#abcd"\n"#placemarker；在替换后，#placemarker将被完全移除，所以结果字符串为"abcd\n"。

10.1.3　宏定义中的##操作符

在宏函数定义中，如果在替换列表中含有##操作符，并且##操作符的前面或后面跟一个该宏函数的一个形参，那么在宏替换时，该形参用其相应实参的预处理符号序列进行替换。如果在宏替换时，相应的实参没有预处理符号，那么形参将用一个“占位标记”预处理符号进行替换。

简单来说，##操作符起到的作用是将宏函数的形参与替换列表中的内容完全融合起来。比如说，我们要定义一个宏函数，使得该形参能与10拼接在一起，那么我们可以用#define CONCAT（x）x##10。这样在宏替换时，相应实参所对应的预处理符号能与10完全融为一体。比如使用CONCAT（32）之后，它就会被替换为3210。一个占位标记预处理符号与一个非占位标记预处理符号的拼接，结果为那个非占位标记预处理符号。

##操作符与#操作符不同，##操作符不仅可作用于宏函数形参，而且也可用于在替换列表中将其前后两个预处理符号序列拼接在一起。因此##操作符的适用范围比#更广，#操作符的操作数只能是宏函数形参。

代码清单10-3展示了##操作符的一些基本用法。

代码清单10-3　宏定义中的##操作符

```
#include <stdio.h>

// 定义了一个宏函数MY_MACRO1，功能是将形参x对应的实参预处理器符号序列与10拼接在一起
#define MY_MACRO1(x)    x ## 10
// 定义了一个宏函数MY_MACRO2，功能是将0x与形参x对应的实参预处理器符号序列拼接在一起
#define MY_MACRO2(x)    0x ## x
// 定义了一个宏函数MY_MACRO3，功能是将形参x对应的实参预处理器符号序列与10拼接在一起
```

```
// 后面再加上将0x与形参y对应的实参预处理符号序列拼接在一起的数
#define MY_MACRO3(x, y) x##10 + 0x##y

// MY_MACRO4是一个宏对象，它直接表示++操作符
#define MY_MACRO4       + ## +

// 这里定义了宏函数CONCAT，将x与y对应的实参预处理符号序列拼接在一起
#define CONCAT(x, y)    x ## y

int main(int argc, const char * argv[])
{
    // 这里的MY_MACRO1(32)将被替换为3210
    int a = MY_MACRO1(32);
    // 这里将打印出3210
    printf("a = %d\n", a);

    // 这里对MY_MACRO1的实参传递不包含任何预处理符号的实参序列，
    // 这样使得这里的MY_MACRO1宏函数最终被替换为10，
    // 形参x用一个占位标记预处理符代替。
    // 占位标记预处理符与10（非占位标记预处理符）拼接，结果就是10
    a = MY_MACRO1();
    printf("a = %d\n", a);

    // 这里的MY_MACRO2(64)将被替换为0x64
    a = MY_MACRO2(64);
    // 这里将打印出100
    printf("a = %d\n", a);

    // 这里的MY_MACRO3(10, 16)将被替换为1010 + 0x16
    a = MY_MACRO3(10, 16);
    // 这里将打印出1032
    printf("a = %d\n", a);

    // 这里的MY_MACRO4将被替换为++，
    // 因此这条语句就相当于：a++;
    a MY_MACRO4;
    // 这里将打印出1033
    printf("a = %d\n", a);

    // 这里CONCAT(an, obj)将被替换为anobj，
    // 从而这里声明了一个名为anobj的int类型对象
    int CONCAT(an, obj);

    // 这里可直接引用anobj标识符
    anobj = a;
}
```

代码清单10-3展示了##操作符的典型使用方式。我们上面已经讲解了#与##操作符，并且讲了宏的基本使用方式。下面我们将进一步讲解宏替换，同时将#操作符与##操作符结合起来进行描述。

10.1.4 宏替换

本节将更详细地描述宏替换。首先，在同一文件作用域内，不能出现两个相同名称的宏标识符，除非这两个宏标识符的替换列表完全等同。也就是说相同的宏定义可以出现多次。

在宏定义中，替换列表之前的空白符以及替换列表之后的空白符全都不作为替换列表中的一部分。

当使用一个宏时，宏实例会用替换列表中的预处理符号进行替换，完了之后预处理器还会再次扫描更多的宏名。也就是说，一个宏定义中可以引用另外一个已定义的宏，因此预处理器会不断迭代解析替换列表中所有出现的宏，直到把它们全都解析完成。当预处理器识别到使用一个宏函数时，其形参会用实参进行替换。替换列表中的形参（若不作为#或##的操作数）在该宏的替换列表中所包含的所有宏名全都被扩展之后，才用相应的实参进行替换。在做实参替换之前，每个实参的预处理符号需要进行完全的宏替换。也就是说，宏函数的替换顺序是先处理替换列表中出现的#与##操作符，然后对替换列表中所出现的宏进行展开替换；接着检查实参是否引用了宏，如果引用了则先对所有引用了宏的实参进行完全的宏替换；最后才将替换列表中出现的形参替换为宏扩展后的实参对应的预处理符号。

在对宏函数调用过程中，当替换列表中的所有形参已被替换，并且对#与##的处理也完成之后，所有占位标记预处理符号被移除，然后所获得的预处理符号序列再被重新扫描。

宏定义与函数定义不同，不能做“递归式”定义，也就是说在一个宏的

替换列表中不能出现所定义宏自身的标识符；此外，“间接递归式”定义也不行，比如宏A的替换列表中引用了宏B，而宏B的替换列表中又引用了宏A。C语言标准指出，发生以上这两种情况时宏名不会被替换。这些没被替换的宏名预处理符号对于后续的替换也不再可用。

最后要提到的一点是，如果在宏替换之后，恰好出现了诸如#define这种结果，那么出现这种预处理符号序列不会作为预处理指示符进行处理，即便它是一个“良好定义的”宏，此时编译器（C语言实现）也会报错。

代码清单10-4进一步描述了宏替换的操作次序以及一些需要注意的地方。

代码清单10-4　宏替换的进一步描述

```
#include <stdio.h>

// 这里先定义一个宏函数MY_MACRO1
#define MY_MACRO1(x)    x + x ## 0

// 这里定义的MY_MACRO1与上述定义的完全等同，因此是允许的
#define MY_MACRO1(x)    x + x ## 0

// 这里定义LITERAL，将形参所对应的实参预处理符号序列进行字符串字面量化
#define LITERAL(x)      #x

// 这里定义宏函数MY_MACRO2，其替换列表引用了宏函数LITERAL
#define MY_MACRO2(x)    LITERAL(x)

// 这里定义了一个用于拼接的宏函数CONCAT，
// 它将x与y所对应的实参预处理符号进行拼接，然后再拼接一个ELLO
#define CONCAT(x, y)    x ## y ## ELLO

int main(void)
{
    // 这里，MY_MACRO1(10)会被替换为：10 + 100
    int a = MY_MACRO1(10);
    printf("a = %d\n", a);

    // 这里是对宏LITERAL的使用：LITERAL(MY_MACRO1(20))，
    // 第一步就是先对LITERAL进行宏替换，直接就是#x，
    // 由于对 # 操作符的处理次序比实参中出现宏名进行替换的次序要优先，
    // 所以这里的第1步完成后直接做第2步，将实参MY_MACRO1(20)替换掉形参x。
    // 替换后相当于：# MY_MACRO1(20)，因此最后的替换结果就是字符串"MY_MACRO1(20)"
    const char *s = LITERAL(MY_MACRO1(20));
    printf("s = %s\n", s);
```

```
    // 这里对宏MY_MACRO2的使用：MY_MACRO2(MY_MACRO1(20))，
    // 第一步先对MY_MACRO2进行宏替换，结果为LITERAL(x)，由于这里不涉及
    // #与## 操作符，然而却存在宏名LITERAL，因此需要进一步对它扩展。
    // 所以第二步就是宏展开LITERAL(x)，得到 #x 。
    // 这里需要大家注意，由于这里的#x不是MY_MACRO2宏里的替换列表中的，
    // 而是被替换扩展出来的，因此此时不需要直接对 # 符号做处理。
    // 所以第三步就是对实参中出现的宏MY_MACRO1进行替换；
    // 对实参MY_MACRO1(20)进行宏替换后结果为：20 + 200，
    // 然后第四步将扩展后的实参20 + 200传给形参x，得：#<20 + 200>。
    // 最终获得字符串字面量——"20 + 200"
    s = MY_MACRO2(MY_MACRO1(20));
    printf("s = %s\n", s);

    // 下面这个对CONCAT的使用结果会替换为：#define ELLO，
    // 使得宏替换结果为一个预编译指示符，这会导致编译报错。
    CONCAT(#, define)

#define def 100
#define H abcd

    // 这里对宏MY_MACRO2的使用与上述类似，
    // 第一步先对MY_MACRO2进行宏替换，结果为LITERAL(x)；
    // 第二步是再对LITERAL进行宏展开，得到#x；
    // 第三步对实参CONCAT(if, def H)进行宏替换，
    // 得到：x ## y ## ELLO ；
    // 由于这里CONCAT宏的替换列表含有##操作符，
    // 因此不会对实参宏def和H进行扩展，
    // 所以第四步将实参if和def H替换形参之后，结果为ifdef HELLO ；
    // 第五步将扩展后的实参传给形参x，得：#<ifdef HELLO> ；
    // 最终获得字符串字面量——"ifdef HELLO"
    s = MY_MACRO2(CONCAT(if, def H));
    printf("s = %s\n", s);
}
```

代码清单10-4详细介绍了宏替换的操作步骤，相信读者读到这里已经对宏定义与宏替换有了更深层的理解了。此代码中的某些注释中含有#<>的符号，这里的<>表示将其中的预处理器符号作为一个整体，比如#<ifdef HELLO>，表示将ifdef HELLO预处理符号序列作为整体替换掉形参。否则#ifdef HELLO的语义跟#<ifdef HELLO>是不一样的。

10.1.5 可变参数的宏定义

从C99标准起，C语言开始加入了不定参数个数（又称可变参数）的宏定义。与可变参数的函数类似，定义可变参数的宏函数时，宏的形参列表

使用...来表示。而在替换列表中，用__VA_ARGS__（前后各有两条下划线）表示形参...对应的参数内容。比如：#define VARIADIC_MACRO（...）printf（__VA_ARGS__）。这里将VARIADIC_MACRO宏定义为printf函数，同时可变参数部分用__VA_ARGS__表示。在使用可变参数宏的时候，...所对应的实参列表将完全等同地取代替换列表中的__VA_ARGS__部分。比如，如果我们使用上述定义的VARIADIC_MACRO（“The integer is：%d\n”，100）；，那么在做宏替换之后就成为printf（“The integer is：%d\n”，100）；。

这里各位要当心的是，一般我们在定义可变参数宏的时候，参数列表直接使用（...），而不是在它前面再加某个形参，比如：#define VARIADIC_PRINT（str，...）print（str，__VA_ARGS__）这种形式。由于...部分的可变形参可以不传任何实参与之对应，比如VARIADIC_PRINT（“Hello”）；像这个宏在做宏替换时会变为：printf（“Hello”，）；。请大家注意，在实参“Hello”后面还有一个逗号！因为__VA_ARGS__表示的是...部分，在替换列表print（str，__VA_ARGS__）中，在__VA_ARGS__之前有一个逗号作为printf函数的实参分隔符。但是替换列表在预处理阶段不会解析此语义，预处理器只会解析宏本身的参数分隔符（比如：VARIADIC_PRINT（str，...）中的逗号作为其形参分隔符）。所以在宏替换之后，__VA_ARGS__之前的逗号仍然保留，而这就会导致编译错误。在使用宏的时候可传缺省的实参内容，但在函数调用的时候却不能这么做，函数调用中每个实参逗号分隔符后必须跟一个对应的表达式作为实参。

代码清单10-5展示了可变参数宏的一些用法。

代码清单10-5　可变参数宏定义与使用

```
#include <stdio.h>

/**
 * 定义了带有可变参数的宏VARIADIC_MACRO1,
 * 其替换列表是将整个实参列表内容作为字符串字面量,
 * 然后与"The string is: "进行拼接
*/
#define VARIADIC_MACRO1(...)    "The string is: "  #__VA_ARGS__

/**
 * 定义了带有可变参数的宏VARIADIC_MACRO2,
 * 其替换列表是将实参列表与100结合，然后用( )包围起来
 */
#define VARIADIC_MACRO2(...)    (__VA_ARGS__ ## 100)

/**
 * 定义了带有两个形参的宏VARIADIC_MACRO3，其第二个形参为可变参数;
 * 其替换列表是先执行形参a对应的表达式，然后执行可变实参列表，最后放表达式10。
 * 它们用( )包围起来
 */
#define VARIADIC_MACRO3(a, ...)    (a  __VA_ARGS__  10)

int main(void)
{
    // 这里VARIADIC_MACRO1(Good luck!)将会被替换为:
    // "The string is: " "Good luck!"
    const char *s = VARIADIC_MACRO1(Good luck!);
    // 输出: s = The string is: Good luck!
    printf("s = %s\n", s);

    // 这里实参列表Say "Hi"!, Byebye, Thank you作为一个整体,
    // 然后通过#操作符被替换为"Say \"Hi\"!, Byebye, Thank you"
    s = VARIADIC_MACRO1(Say "Hi"!, Byebye, Thank you);
    // 输出: s = The string is: Say "Hi"!, Byebye, Thank you
    printf("s = %s\n", s);

    // 这里使用VARIADIC_MACRO1宏的时候没有传任何实参,
    // 因此宏替换后就是"The string is: "这单一的字符串字面量
    s = VARIADIC_MACRO1();
    // 输出: s = The string is:
    printf("s = %s\n", s);

    // 这里VARIADIC_MACRO2(20)将被替换为: (20100)
    int a = VARIADIC_MACRO2(20);
    // 输出: a = 20100
    printf("a = %d\n", a);

    // 这里VARIADIC_MACRO2(10, 20, 30)将被替换为: (10, 20, 30100),
    // 很明显它是一个逗号表达式。而整条语句为: a = (10, 20, 30100);
    a = VARIADIC_MACRO2(10, 20, 30);
    // 输出: a = 30100
    printf("a = %d\n", a);

    int b = 0;

    // 这里VARIADIC_MACRO3(++b, -20 + )将被替换为: (++b - 20 + 10),
    // 其中第一个实参为++b，后面的-20 + 作为可变参数的实参
```

```
    a = VARIADIC_MACRO3(++b, -20 + );
    // 输出: a = -9, b = 1
    printf("a = %d, b = %d\n", a, b);

    // 这里缺省VARIADIC_MACRO3的第一个实参，可变实参部分为30 * ，
    // 因此VARIADIC_MACRO3(, 30 * )被替换为: (30 * 10)
    a = VARIADIC_MACRO3(, 30 * );
    // 输出: a = 300
    printf("a = %d\n", a);

    // 这里同时缺省了第一个实参以及可变参数列表的实参，
    // 因此VARIADIC_MACRO3( , )被替换为: (10)
    a = VARIADIC_MACRO3( , );
    // 输出: a = 10
    printf("a = %d\n", a);
}
```

代码清单10-5中我们可以清晰地看到，在使用可变参数宏时，可变形参对应的实参是连同实参逗号分隔符一同换入替换列表中去的，取代__VA_ARGS__部分。在替换完之后，如果表达式有语法错误则会编译报错，替换列表在预编译时是不做语法分析的。

10.2　C语言中预定义的宏

C语言标准指出了C语言实现（即C语言编译工具）要求必须实现的预定义宏以及可选实现的预定义宏。所谓预定义宏即不是由C语言程序员自己定义，而是由编译工具预先已经定义好的宏。

C语言中的预定义宏分为三类：

·C语言标准强制要求预定义的宏；

·环境宏，它可选实现的预定义宏；

·条件特征宏，这也是实现可选实现的。

当然除此之外，C语言实现可根据当前平台环境等因素自己定义特定的预定义宏。

10.2.1　C语言强制要求的预定义宏

我们在写C语言代码的时候出于调试或备注的目的，往往想方便获得当前C语言源文件名、当前代码所在的行号、当前代码所在的函数名、编译当前源文件时的时间日期等。因此，C语言标准提供了以下我们常用的预定义宏来方便地获取这些信息。

1）__DATE__：此预定义宏用于表示当前日期的一个字符串字面量，其形式为"Mmm dd yyyy"。这里，Mmm是月份的缩写，比如1月是Jan，2月是Feb。如果日小于10，那么前面的一个d用一个空格字符表示。所以，如果是2015年2月6日，__DATE__就表示为"Feb 22015"，这里Feb与2之间有两个空格。

2）__FILE__：表示当前源文件名，用字符串字面量表示。

3）__LINE__：表示当前源文件的当前行号，用一个整数常量表示。

4）__STDC__：它表示一个常量值，如果该宏的值为1，那么说明当前C语言实现顺应C语言标准。

5）__STDC_HOSTED__：如果当前C语言实现是一个主机端实现，那么该预定义宏的值为1；如果是独立式实现，那么该预定义宏的值为0。所谓主机端实现是指当前C源代码最终编译为当前目标平台兼容的二进制代码文件，随后可以连接成能在当前目标平台上可直接执行的可执行文件。而独立式实现类则比较灵活，比如可以将当前C源代码编译为一种中间代码，然后可以以虚拟机的方式对该中间代码做解释执行。

6）__STDC_VERSION__：该预定义宏表示一个整数常量（long类型），用于指明当前正使用的C语言标准版本。比如，如果当前用的是2011年12月发布的C语言标准，那么整数常量为201112L。

7）__TIME__：表示当前时间的一个字符串字面量，其形式为"hh:

mm：ss"。比如，22时55分30秒则表示为"22：55：30"。

除了上述这些预定义宏之外，从C99标准起规定，C语言实现还需要实现一个预定义标识符__func__，该预定义标识符表示当前函数名，它被定义为：static const char__func__[]="function-name";

代码清单10-6展示了使用上述这些宏的示例。

代码清单10-6　C语言强制要求预定义的宏

```
#include <stdio.h>

int main(void)
{
    printf("The current date is: %s\n", __DATE__);
    printf("The current time: %s\n", __TIME__);
    printf("The current file is: %s\n", __FILE__);
    printf("The current function: %s\n", __func__);
    printf("The current line is: %d\n", __LINE__);
    printf("The current standard is: %ld\n", __STDC_VERSION__);
    printf("Conform standard C %d\n", __STDC__);
    printf("Is current hosted environment: %d\n", __STDC_HOSTED__);
}
```

此外，C语言标准还强制规定了不能预定义__cplusplus该预定义宏，如果此宏被预定义，那么说明当前编译环境为C++语言，而不是C语言。

10.2.2　环境宏

环境宏主要用于指明当前源文件输入字符编码的支持情况。这些宏都是可选实现，而不是必须实现的。

1）__STDC_ISO_10646__：该预定义宏被定义为一个整数常量，形式

与__STDC_VERSION__类似。如果遵循的ISO10464是1997年12月发布的，那么此宏的值为199712L。如果C语言实现定义了这个宏，那么在wchar_t类型对象中存放的值，在Unicode要求字符集中的每一个Unicode与该字符所对应的短标识符具有相同的值。Unicode要求字符集由ISO/IEC 10646定义的所有字符组成，此外还包括一些修改与技术勘误表。如果C语言实现使用了其他编码方式，那么就不能定义此宏。

2）__STDC_MB_MIGHT_NEQ_WC__：如果该预定义宏的值为1，那么指明当在一个整数字符常量中用作为单个字符时，wchar_t类型的字符编码中，一些基本字符集的编码值不需要与其值相等。

3）__STDC_UTF_16__：如果该预定义宏的整数常量值为1，那么指明char16_t类型的值表示的是UTF16编码的。如果C语言实现对char16_t类型的值采用的是其他编码类型，那么就不该定义此宏。

4）__STDC_UTF_32__：如果该预定义宏的整数常量值为1，那么指明char32_t类型的值是UTF-32编码的。如果C语言实现对char32_t类型的值采用的是其他编码方式，那么就不该定义此宏。

10.2.3　条件特征宏

C语言标准中含有一些语法特征是C语言编译器可选实现的，编译器可根据这些宏来指明自己是否支持C语言中的一些可选实现的语法特性。

1）__STDC_ANALYZABLE__：如果该预定义宏的整数常量值为1，那么指明当前C语言实现顺从C语言标准在附录L中标出的运行时代码检查，这些代码检查主要包括数组边界是否越界、栈是否越界等。

2）__STDC_IEC_559__：如果该预定义宏的整数常量值为1，那么指明当前C语言实现顺从C语言标准中附录F中标出的浮点数表示以及浮点算术运算。

3）__STDC_IEC_559_COMPLEX__：如果该整数常量值为1，那么指明当前C语言实现遵守C11标准手册附录G中的规格说明（ISO/IEC 60559兼容的复数）。

4）__STDC_LIB_EXT1__：如果该预定义宏被定义，那么它是一个整数常量，形式与__STDC_VERSION__类似，用于指明当前C语言实现支持C11标准中附录K定义的扩展（主要是边界检查接口）。

5）__STDC_NO_ATOMICS__：如果该预定义宏的整数常量值为1，那么指明当前C语言实现不支持原子类型（包括_Atomic类型标识符）以及<stdatomic.h>头文件。

6）__STDC_NO_COMPLEX__：如果该预定义宏的整数常量值为1，那么指明当前C语言实现不支持复数类型，并且也不支持<complex.h>头文件。如果C语言实现定义了此宏，那么当前实现就不该定义__STDC_IEC_559_COMPLEX__这个宏。

7）__STDC_NO_THREADS__：如果该预定义宏的整数常量值为1，那么指明当前C语言实现不支持<threads.h>头文件。这里要注意的是，C语言标准中没说有了此预定义宏就不支持_Thread_local，这个关键字仍然可以被支持。

8）__STDC_NO_VLA__：如果该预定义宏的整数常量值为1，那么指明当前C语言实现不支持变长数组，也不支持可变修改类型。

10.2.4 主流编译器及平台预定义的宏

以下预定义宏不是C语言标准指出的，而是当前主流桌面平台、处理器及主流编译器可能支持的预定义宏。我们通过这些宏可以了解到当前所用的是哪款C语言编译器，以及当前编译目标环境所在的系统与所用的处理器架构等。

1）_MSC_VER：如果C语言实现预定义了这个宏，说明当前的编译器为MSVC。

2）__GNUC__：如果C语言实现预定义了这个宏，说明当前的编译器为GCC或兼容GCC的编译器。那么，如果我们在使用Clang编译器时，此宏也是被定义的。

3）__clang__：如果C语言实现预定义了这个宏，说明当前的编译器为Clang编译器。

4）__i386__：如果C语言实现预定义了这个宏，说明编译生成的目标为32位的x86处理器。

5）__x86_64__：如果C语言实现预定义了这个宏，说明编译生成的目标为x86_64处理器，并运行在64位系统模式下的指令集。

6）__arm__：如果C语言实现预定义了这个宏，说明编译生成的目标为32位ARM处理器。

7）__arm64__：如果C语言实现预定义了这个宏，说明编译生成的目标为64位的ARM处理器。

8）__APPLE__：如果C语言实现预定义了这个宏，说明编译生成的目标为Apple系统（包括macOS、iOS、tvOS、watchOS等）上的。

9）__unix__：如果C语言实现预定义了这个宏，说明编译生成的目标是Unix或与其兼容的系统上的。

10）__linux__：如果C语言实现预定义了这个宏，说明编译生成的目标是Linux或与其兼容的系统上的。

11）_WIN32：如果C语言实现预定义了这个宏，说明编译生成的目标是32位Windows系统。

12）_WIN64：如果C语言实现预定义了这个宏，说明编译生成的目标是64位Windows系统。

13）__LP64__：如果C语言实现预定义了这个宏并且其整数常量为1，那么说明当前程序运行环境为64位系统环境，此时long int类型的长度为64位，int类型的长度仍然为32位。

10.3 条件预编译

条件预编译用来控制所要编译的代码。当条件预编译中的条件为真时，这段预编译块中的代码参与编译，否则不参与编译。实际上，当条件预编译的条件为真时，预处理器才会将该预编译块中的预处理字符序列替换到源代码中，等待后续编译。

控制条件包含的表达式应该是一个整数常量表达式，此外还支持defined表达式。defined表达式的形式类似于sizeof表达式，有两种形式：

```
def ined 标识符
```

以及

```
def ined ( 标识符 )
```

defined表达式只能与#if条件包含控制语句联合起来使用，而不能单独用于其他场合。defined后面的标识符应该是一个宏名，如果该宏在此前已被定义，那么defined表达式的值为1，否则defined表达式的值为0。

在预处理常量表达式中，defined表达式可以参与算术逻辑运算，而且像！defined也用得比较多，如果！defined后面的标识符定义过，那么表达式的值为0，否则表达式的值为1。这里的！符号相当于逻辑运算中的非操作符。

下面介绍#if、#elif预处理指示符，这两个预处理指示符的形式都一样：

```
#if   常量表达式      换行符
条件预编译组可选
#elif   常量表达式      换行符
条件预编译组可选
```

在上面的描述中，“#if常量表达式”以及“#elif常量表达式”后面必须跟一个换行符，而不能用其他符号作为分隔符。它们用于检查控制常量表达式计算结果是否为非零。如果#if后面的常量表达式的计算结果不为零，那么后面将编译该条件预编译组的代码。如果#if后面的常量表达式的计算结果为零，且后面跟有#elif语句，那么判定#elif语句后面的常量表达式，如果计算结果不为零，那么后面将编译该条件预编译组的代码。#elif的语义类似于else if，只不过在预处理器中的#else后面不能跟任何其他表达式和语句，只能跟换行，再跟条件预编译组。所以，#elif也可以用下述语句代替：

```
#else     换行符
#if   常量表达式      换行符
条件预编译组可选
```

这里的“常量表达式”应该是一个有效的、适用于预处理的常量表达式。当本组条件预编译组的条件包括判定都结束之后，最后必须用#endif来结尾。其中，#endif后面只能跟换行符，而不能跟其他字符。

此外，如果#if、#elif后面的常量表达式中包含一个标识符（在预处理器中就是宏），并且该标识符在此时没有被定义，那么此条件包含的判定

结果即为“假”，其下面的条件预编译组的代码不会参与编译。

代码清单10-7展示了上述几个控制条件包含语句的使用方式。

代码清单10-7　#if、#elif、#else预处理指示符的使用

```
#include <stdio.h>

int main(void)
{
    // 这里3 + 5是一个非零整数常量表达式，
    // 因此下面预编译组的puts函数调用表达式将被编译
#if 3 + 5
    puts("Non-zero expression");
#endif

    // 这里做一次puts函数调用，而实参内容则根据预编译条件来选
    puts(

// 这里#if后面的表达式为0，因此不会将"0"编译进去
#if 0
    "0"
// 这里#elif后面的整数常量表达式结果为0，因此后面的"1"不会编译进去
#elif 3 - 3
    "1"
// 由于上述条件判定均失败，所以编译#else下面的预编译组，"2"
#else
    "2"
#endif
         );     // 因此这里将输出2

// 这里定义了宏HELLO
#define HELLO

// 这里判定的是倘若定义了HELLO这个宏，则编译下面的puts函数调用。
// 显然，这里将会输出HELLO defined
#if defined(HELLO)
    puts("HELLO defined!");
#endif

// 这里判定的是倘若定义了HELLO这个宏，同时也定义了HI宏，
// 则编译下面的puts函数调用。
// 显然，这里的puts函数调用不会被编译
#if defined(HELLO) && defined(HI)
    puts("Both defined");
#endif

// 这里定义了宏HI，并将该宏的值定义为2
#define HI  2

// 这里判定的是倘若定义了HELLO这个宏，同时也定义了HI宏，
// 并且两个defined表达式的值加起来等于HI的值时，则编译下面的puts函数调用。
// 显然，defined表达式的值均为1，因为这两个宏此时都被定义。
// 同时，HI的值被定义为了2，因此条件完全满足，下面的printf函数调用将被编译
#if defined(HELLO) && defined(HI) && (defined(HELLO) + defined(HI)) == HI

    // 这里输出：HI value: 2
    printf("HI value: %d\n", HI);

#endif
```

```
    int aa = 100;

// 这里大家要注意，由于上面定义的aa不是预处理中定义的宏，因此在预处理中找不到该标识符，
// 因此这里由于找不到aa标识符，#if条件不成立，输出：aa is not defined!
#if aa == 100
    puts("aa is defined!");
#else
    puts("aa is not defined!");
#endif

#define aa  20

// 由于此时定义了aa宏，并且其替换列表的整数常量表达式确实为20，所以这里输出Yep!
#if aa == 20
    puts("Yep!");
#endif

    // 这里将输出aa = 20，因为aa宏定义将之前的aa对象标识符给覆盖掉了
    printf("aa = %d\n", aa);

    enum
    {
        MY_ENUM1,
        MY_ENUM2,
        MY_ENUM3
    };

// 对于枚举符而言，它也不属于预定义标识符，因此下面将输出：MY_ENUM2 is not defined!
#if MY_ENUM2 == 1
    puts("MY_ENUM2 is defined!");
#else
    puts("MY_ENUM2 is not defined!");
#endif
    // 以下预编译语句会发生预处理错误！由于sizeof操作符只能用于C代码的编译期间，
    // 而不能用于预处理期间，因此sizeof(int)是一个非法的预处理常量表达式
#if aa == sizeof(int)
    puts("Yep!");
#endif
}
```

除了上述介绍的#if、#elif与#else这些条件包含预处理指示符之外，还有#ifdef与#ifndef表示条件包含的预处理指示符。#ifdef类似于#if defined的简略表达方式，不过#ifdef后面只能跟标识符（即宏名），而不能是常量表达式。而#ifndef则类似于#if！defined的简略表达方式，不过#ifndef后面也只能跟标识符，而不能跟常量表达式。

代码清单10-8将利用这些条件包含预处理与10.2节中列出的C语言实现可选预定义宏相结合，给出当前C语言实现与运行环境所提供的语言特性与系统特征。

代码清单10-8　条件包含预处理与C语言实现预定义宏

```
#include <stdio.h>

int main(void)
{
#ifdef __GNUC__
    puts("GNU C or compatible compiler!");
#endif

#ifdef __clang__
    puts("Current compiler is Clang!");
#endif

#ifdef _MSC_VER
    puts("Current compiler is Microsoft Visual C!");
#endif

#if defined(__i386__) || defined(__x86_64__)
    puts("x86 processor!");
#elif defined(__arm__) || defined(__arm64__)
    puts("ARM processor");
#endif

#ifdef __APPLE__
    puts("Apple Inc. OS");
#elif defined(_WIN32) || defined(_WIN64)
    puts("Windows OS");
#endif

#ifdef __unix__
    puts("Unix or compatible OS");
#endif

#ifdef __linux__
    puts("Linux or compatible OS");
#endif

#ifdef __LP64__
    puts("long type is 64-bit!");
#endif

// 相当于#if __STDC_ANALYZABLE__ != 0
#if __STDC_ANALYZABLE__
    puts("C Analyzability available!");
#endif

// 相当于#if __STDC_ISO_10646__ != 0
#ifdef __STDC_ISO_10646__
    printf("ISO 10646 %ld supported!\n", __STDC_ISO_10646__);
#endif

// 相当于#if __STDC_UTF_16__ != 0
#if __STDC_UTF_16__
    puts("UTF-16 supported!");
#endif

#if __STDC_UTF_32__ != 0
    puts("UTF-32 supported!");
#endif

#if __STDC_IEC_559__ == 1
    puts("ISO/IEC 559 comformed!");
#endif
```

```
#if __STDC_NO_ATOMICS__ == 1
    puts("Atomics not available!");
#endif

#if __STDC_NO_THREADS__ == 1
    puts("<threads.h> not available!");
#endif
}
```

10.4　源文件包含预处理指示符

C语言是一个可共享的编程语言，我们可以将自己的源代码编译成库，然后给其他开发人员使用。这样，一来可以对自己的源代码进行保护，因为库中的代码已经全都转为了平台相关的机器指令码，二来也不影响其他开发者对库中的函数接口进行调用。那么我们如何将自己源码中的对外函数接口以及数据类型等共享给其他开发者呢？答案就是通过头文件（header）！

C语言中的头文件一般以.h作为后缀名，而且C语言编译器一般不会对.h文件进行编译，而.h头文件中的代码如果存在语法错误，则往往是在包含进源文件参与编译后由编译器发现的。在C语言预处理中使用#include预处理指示符将指定的文件包含到当前源文件中。

#include有两种形式，一种是：

```
#include <头文件名> 换行符
```

还有一种是：

```
#include “头文件名” 换行符
```

第一种使用<>的形式会使得预处理器将<>中所指定文件的整个内容将#include<头文件名>这整个预处理指示符全都替换掉。其中，对<>中指定

的文件路径的搜索是实现自定义的。对于主流桌面系统而言，<>中指定的文件路径一般就是操作系统默认存放库头文件的系统路径或是由用户指定的系统环境路径。

而第二种“”的方式是由实现先通过另一种自定义的搜索方式对指定文件进行搜索，如果搜索到则进行内容替换；倘若搜索不到，则换用<>形式的搜索方式进行搜索，如果再搜索不到，则会出现编译出错。这里所谓的“另一种实现自定义搜索方式”，对于主流桌面系统而言通常就是当前C语言项目工程下的路径。

在具体实践上通常来说，对于系统自带头文件，包括C语言标准库的头文件都会放置在每个操作系统的指定位置，我们可以直接用<>的方式；另外，我们通过设置当前C语言项目工程的默认头文件搜索路径，也可以用<>的形式来包含指定搜索路径下的头文件；而对于我们自己写的头文件，一般直接放在当前C语言工程项目中，我们可采用“”的方式包含。就如上面所提到的那样，使用“”的方式包含头文件，其搜索范围比用<>的形式来得广。

此外，#include后面可以跟一个宏名，我们可以用一个宏对象来指定所要包含的文件名。代码清单10-9展示了对#include预处理指示符的使用方法与作用。

代码清单10-9　#include预处理指示符的使用与效果

```
/** 以下是自己写的a1.h头文件里的内容：*/
```

```
// 这里使用条件预编译是为了防止此头文件被某一源文件多次包含
#ifndef a1_h
#define a1_h

#include <stdio.h>

#define MY_MACRO        100

struct MyStruct
{
    int a;
    float f;
};

static void MyFunction(struct MyStruct s)
{
    printf("The value is: %f\n", s.a + s.f);
}

#endif /* a1_h */


/** 以下是main.c源文件内容 */

// 由于a1.h中已经包含了<stdio.h>，因此这里可以不用包含
#include "a1.h"

// 这里即便再次包含a1.h也没有问题，由于a1.h中已经通过条件预编译进行了重复包含的保护
#include "a1.h"

int main(void)
{
    printf("The macro value is: %d\n", MY_MACRO);

    struct MyStruct s = { 10, 1.5f };

    MyFunction(s);
}
```

在代码清单10-9中分别列出了a1.h头文件与main.c源文件中的代码。在main.c源文件中，使用了#include“a1.h”之后，其实是将整个a1.h头文件中的内容全都包含到了main.c文件中的#include“a1.h”位置处。因此在两句源文件包含的预处理完成之后，整个main.c的内容是这样的：

```
#define a1_h

#include <stdio.h>

#define MY_MACRO        100

struct MyStruct
{
    int a;
    float f;
};
```

```
static void MyFunction(struct MyStruct s)
{
    printf("The value is: %f\n", s.a + s.f);
}

int main(void)
{
    printf("The macro value is: %d\n", MY_MACRO);

    struct MyStruct s = { 10, 1.5f };

    MyFunction(s);
}
```

我们看到，当包含了a1.h头文件之后，在main.c源文件的翻译单元中自动就定义了a1_h以及MY_MACRO这两个宏。在main.c中，当遇到第2条#include“a1.h”时，由于a1_h宏已经被定义，因此其后续内容都不会被再次包含到main.c源文件中。

然后我们再看代码清单10-10，#include加宏名的使用场合。

代码清单10-10　在#incude预处理指示符后跟宏名

```
/** 以下是自己写的a1.h头文件里的内容: */

// 这里使用条件预编译是为了防止此头文件被某一源文件多次包含
#ifndef a1_h
#define a1_h

#include <stdio.h>

#define MY_MACRO        100

struct MyStruct
{
    int a;
    float f;
};

static void MyFunction(struct MyStruct s)
{
    printf("The value is: %f\n", s.a + s.f);
}

#endif /* a1_h */

/** 以下是自己写的a2.h头文件里的内容: */

// 这里使用条件预编译是为了防止此头文件被某一源文件多次包含
#ifndef a2_h
#define a2_h
```

```
#define MY_MACRO        -10

struct MyStruct
{
    short s;
    double d;
};

static void MyFunction(struct MyStruct s)
{

}

#endif /* a2_h */


/**  以下是main.c源文件中的内容 */

#include <stdio.h>

// 这里先定义一个USE_A2_HEADER宏
#define USE_A2_HEADER

// 以下条件预编译判别的是：如果定义了USE_A2_HEADER宏，
// 那么将HEADER_NAME定义为"a2.h"；否则，将HEADER_NAME定义为"a1.h"
#ifdef USE_A2_HEADER

#define HEADER_NAME     "a2.h"

#else

#define HEADER_NAME     "a1.h"

#endif

// 这里通过HEADER_NAME所定义的头文件名进行包含
// 当前包含的是"a2.h"
#include HEADER_NAME

int main(void)
{
    printf("The macro value is: %d\n", MY_MACRO);

    struct MyStruct s = { 10, 1.5f };

    MyFunction(s);
}
```

在main.c中，一开始定义了宏USE_A2_HEADER，因此HEADER_NAME宏所指定的是“a2.h”，如果我们把#define USE_A2_HEADER注释掉，那么包含的将是“a1.h”。

10.5　#error预处理指示符

#error预处理指示符用于在预处理过程中报出指定的错误诊断信息。其形式为：

```
#error    预处理符号可选    换行符
```

如果源文件中含有#error预处理指示符，并且它发生作用，那么编译器就会报#error后面指定预处理符号信息字样的错误。如代码清单10-11所示。

代码清单10-11　#error预处理指示符

```
// #define MY_SAFE_MACRO

#ifndef MY_SAFE_MACRO
// 如果MY_SAFE_MACRO没有被定义，那么这里就会报错，显示：safe macro not defined!
#error safe macro not defined!
#endif

// 这里直接报错，但不输出错误信息
#error

int main(void)
{

}
```

代码清单10-11中，先把MY_SAFE_MACRO宏定义给屏蔽掉，所以在编译整个源代码时会报错误信息。一般#error与条件预编译一起使用的场合较多。

此外，有些编译器还支持#warning，其用法与#error一样，只不过编译

器输出的是警告信息，而不阻碍编译器继续编译。

10.6　#line预处理指示符

#line预处理指示符用于作为行号控制。其形式为：

```
#line　数字序列　　换行符
```

#line后面的数字即指定它下一行的行号。使用#line预处理指示符，预处理器在计算行号时就以它作为基准开始算，而忽略在该预处理指示符之前的行号状态。

此外，#line还有一种形式可用来修改当前的源文件名：

```
#line　数字序列　“源文件名”　换行符
```

这样，除了当前行号，连源文件名都能一起被修改。代码清单10-12展示了#line预处理指示符的使用方式以及效果。

代码清单10-12　#line预处理指示符的使用与效果

```
#include <stdio.h>

int main(void)
{
#line 10
    printf("The current line is: %d\n", __LINE__);
    // 上面输出：The current line is: 10

    printf("The current line is: %d\n", __LINE__);  // 这里输出13

#line 100 "hello.cc"
    printf("The current line: %d, and file name: %s\n", __LINE__,
           __FILE__);
    // 上面输出：The current line: 100, and file name: hello.cc
}
```

10.7　#undef预处理指示符

#undef预处理指示符用于取消之前定义过的宏。其形式为：

```
#undef     标识符     换行符
```

这里的标识符就表示一个宏名。如果#undef后面的宏名之前没被定义过也没关系，预处理器不会报错。如果#undef后面的宏名之前被定义过，那么之前定义的宏就被撤销。代码清单10-13展示了#undef的使用与效果。

代码清单10-13　#undef的使用与效果

```
#include <stdio.h>

#define MY_MACRO    100

// 这里使用 #undef 将MY_MACRO宏取消定义
#undef MY_MACRO

// 随后这里再重新定义MY_MACRO宏
#define MY_MACRO    200

int main(void)
{
    // 这里将会输出200
    printf("MY_MACRO value is: %d\n", MY_MACRO);

    // 即便没有定义过YOUR_MACRO宏，放在#undef后也没问题
#undef YOUR_MACRO

    // 再次取消定义MY_MACRO
#undef MY_MACRO

#ifndef MY_MACRO
// 这里编译器会报警：MY_MACRO not defined!
#warning MY_MACRO not defined!
#endif
}
```

代码清单10-13演示了#undef的使用，通过这段代码我们可以清晰地了解到#undef将一个指定宏给取消定义的效果。这里大家要注意的是，对于

一个宏，只有当它被取消定义之后才能给它重新定义一个替换列表。

10.8 pragma预编译指示符与操作符

#pragma预处理指示符用于指示当前翻译单元使用某种编译特性进行编译。比如，可以指定哪些函数用某个优化选项进行优化，从哪里开始使用标准浮点约定等。其形式为：

```
#pragma  预处理符号  换行符
```

这里，预处理符号就是pragma指定的编译选项，这些编译选项一般由编译器实现自己定义。当然，C语言标准也列出了若干标准的编译选项，以STDC作为前缀，其形式为：

```
#pragma  STDC  标准支持的编译选项  开关值  换行符
```

这里，开关值有三种，分别是：ON、OFF与DEFAULT。

除了#pragma预处理指示符外，C语言标准从C99开始引入了_Pragma操作符。

_Pragma操作符的语义与#pragma预处理指示符一样，只不过它更为灵活，可用于宏定义的替换列表中，而#pragma预处理指示符则不能。

_Pragma操作符的形式为：

```
_Pragma ( 字符串字面量 )
```

它后面可以直接跟函数、结构体类型等元素的定义。代码清单10-14介绍了#pragma预处理指示符与_Pragma操作符的使用。

代码清单10-14　#pragma预处理指示符与_Pragma操作符的使用

```
#include <stdio.h>
// 这里使用#pragma预处理指示符开启遵循浮点数标准的编译选项
#pragma STDC FP_CONTRACT ON

// 指示后面的代码用-O2优化选项进行优化
#pragma O2

static void MyFunc(int a)
{
    a += 100;
    printf("a = %d\n", a);
}

// 指示后面的代码用-O0优化选项编译
#pragma O0

// 这里根据当前编译器是否预定义了DEBUG宏来指示MY_PRAGMA_OPTION宏的状态。
// 如果预定义了DEBUG宏，那么MY_PRAGMA_OPTION宏用-O0的编译选项；
// 否则使用-O2的编译选项
#ifdef DEBUG
#define MY_PRAGMA_OPTION    _Pragma("O0")
#else
#define MY_PRAGMA_OPTION    _Pragma("O2")
#endif

MY_PRAGMA_OPTION
static int MyFunc2(int a, int b)
{
    return a * a + b * b;
}

// 指示从main函数开始的代码用-O1优化选项编译
_Pragma("O1") int main(void)
{
    MyFunc(10);

    int value = MyFunc2(3, 4);
    printf("The value is: %d\n", value);
}
```

通过代码清单10-14，我们可以看到利用条件预编译与_Pragma操作符的结合使用能非常方便地通过当前环境上下文来指定自己需要的编译选项。

10.9　空指示符与C语言中的程序注释

预处理指示符中的空指示符比较简单，形式为：

```
# 换行符
```

#与换行符之间可以存在空白符。它在实际代码中没有任何效果。有时在写一连串预处理指示符时，添加一些空指示符可能会使得代码格式更好看一些。代码清单10-15展示了空指示符的一般使用。

代码清单10-15　空指示符的使用

```
#include <stdio.h>
#
#ifdef HELLO
#warning HELLO is defined
#endif
#
#define HELLO   100
#

int main(void)
{
    printf("HELLO = %d\n", HELLO);
}
```

下面主要谈论C语言的注释。对于程序代码来说，注释的重要性不言而喻，无论是给自己的编程思路留个注解还是给其他人看源码，这些一般都需要注释加以辅助。我们之前的所有代码示例中几乎都含有注释，用于说明当前语句的作用以及效果。在计算机编程语言中，注释（comment）是不参与编译的，而只是作为描述性的文字片段。注释可以用来描述当前源代码主要实现什么功能，描述函数实现什么功能，结构体、联合体等用

户自定义类型表示什么含义等。

在C11标准中，注释有两种形式，第一种是行注释，在一行代码中只要不是出现在“”或‘’内的//符号，一直到这一行的换行符，都属于注释部分。从下一行开始则不属于注释部分，而是正式的代码部分。另一种是语句块注释，从出现不包含在“”或‘’内的/*符号开始一直到*/结束都属于注释部分。/*到*/之间可以存在多个换行符。

在C语言中，如果出现注释，那么C语言实现在预处理阶段就会去掉它所遇到的注释行和注释段，一般会用一个空白符来代替，不过C语言标准也没有具体指定用什么符号去替换整个注释行与注释段，但是对于C语言程序员来说，需要将注释看作代码中的一个空白符。代码清单10-16展示了注释的一些用法以及效果。

代码清单10-16　注释的使用与效果

```
#include <stdio.h>
#include <stdbool.h>

/* 这是一个注释段，
   可以用来大篇幅地介绍当前源文件、函数等功能描述。
 */

#define MY_LITERAL(expr)    #expr

int main(void)  // 这是一个注释行，可以用来描述某句代码表示什么含义
{
    const char *s = "// 这不是一个注释，而是一个字符串"
    printf("s = %s, %c\n", s, '// ');  // '// '也不是一个注释，而是一个字符常量

    int var = 100;

    // 下面这句是错误的，这里并不是var += 100，而是被看作为v ar += 100
#if false
    v/*这里用注释隔断*/ar += 100;
#endif

    s = MY_LITERAL(you/*这里用注释隔断*/win);

    // 这里将输出：you win。这两个单词中间用一个空格隔断。
```

```
    printf("var = %d, s = %s\n", var, s);

    // 如果在一个行注释中出现/*，那么它已经被融为当前行注释中的内容，
    // 而不起到注释段开头的作用。所以后面再用*/是无效的
    /* 同样，如果在注释段中出现//，那么它也被融为注释内容的一部分，也不再充当注释行的开头
     */
}
```

代码清单10-16展示了比以往更多形式的注释，以及注释在代码中扮演的作用。我们在代码清单10-16中可以清楚地看到，注释段被Clang编译器在预处理时转为了一个空格符。

10.10　本章小结

本章介绍了C语言预处理器的所有特性，展示了C语言强大的宏功能以及条件预编译等高效、灵活的特性。通过将宏、条件预编译、pragma等组合使用，可使得C语言代码更具跨平台性、可移植性以及紧凑性。当然，对宏的滥用也可能导致程序调试的不便捷，所以设计宏的时候需要小心谨慎。

此外，本章对C语言的注释做了进一步详细介绍，描述了注释的作用以及在代码中的使用效果。通过本章的学习，大家可以写出更丰富、更“专业”的C语言程序代码。

第11章　C语言程序的编译上下文

本章将为大家介绍C语言中的编译上下文相关话题。其中包括对象、函数以及类型标识符的作用域和名字空间，对象与函数的连接以及对象的生命周期。由于在编程语言中，对数据对象的管理、函数接口的抽象化在维护较大项目工程的时候会显得非常重要，因此通过学习本章，我们可以将自己写的C语言程序中的函数、对象、类型进行更好地安排，使得我们的程序或库有着更加良好的封装性，同时也能使代码写得更安全、健壮。

11.1 C语言程序中的作用域和名字空间

C语言程序中，一个标识符（identifier）可用来表示一个对象，函数，结构体、联合体以及枚举的名称或是它们的成员（枚举的成员又被称为枚举常量），typedef名（13.4节将会详细描述），跳转标签名，宏名，或是一个宏形参。同一个标识符在程序中的不同位置可能会表示不同的实体，也就是说在C语言中，标识符允许被覆盖。

对于一个标识符所指代的每个不同的实体，只有在一个程序文本的区域内，该标识符才是“可见的”，这个区域就被称为作用域（scope）。如果某个标识符同时指代了多个不同的实体，那么这些实体要么具有不同的作用域，要么在不同的名字空间中。C语言一共有4种作用域，分别是：函数作用域、文件作用域、语句块作用域以及函数原型作用域。下面，我们将分别介绍这4种作用域。

11.1.1 文件作用域

在C语言标准中，对具有文件作用域的标识符的定义非常有意思，用的是排除法——如果一个标识符声明在所有语句块之外，同时也在形参列表之外，那么我们称该标识符具有文件作用域。文件作用域从当前源文件开始一直到源文件末尾结束。

这里有几个非常典型的例子。如果读者学过Java、C++等面向对象的编程语言，那么应该知道这些编程语言中的类可以嵌套定义类，并且嵌套类的作用域属于其外部类。C语言也能在结构体或联合体中嵌套定义结构体或联合体，但嵌套定义的结构体和联合体具有与其外部结构体或联合体同等的作用域。也就是说，如果定义的外部结构体在文件作用域中，那么其内部嵌套定义的结构体也处于文件作用域中。同样，定义在文件作用域的枚举类型中的枚举常量也具有文件作用域。在C语言中，结构体、联合体与枚举定义块本身不具有“作用域”属性，它们不属于语句块。

此外，对于预处理器中的宏、条件预编译等指示符，它们始终具有文件作用域，而不受其他作用域的影响（参见第10章的介绍）。代码清单11-1展示了具有文件作用域的标识符。

代码清单11-1　具有文件作用域的标识符

```
#include <stdio.h>

// 这里OutStruct结构体类型处于文件作用域
struct OutStruct
{
    // a是OutStruct结构体的成员对象，不具有作用域属性
    int a;

    // InnerStruct是定义在OutStruct内部的结构体，但具有文件作用域
    struct InnerStruct
    {
        int i;
    } inner;        // 这里inner作为OutStruct结构体的成员对象

    // 在OutStruct结构体内定义的枚举类型也具有文件作用域
    enum MY_ENUM
    {
        // 这里面的枚举常量也同样具有文件作用域
        MY_ENUM_1,
        MY_ENUM_2
    } e;
};

// 这里对象sa处于文件作用域，
// 另外我们可以看到定义在OutStruct结构体内部的InnerStruct可以直接被访问，
// 也就是说InnerStruct是全局可见的，因此它具有文件作用域
```

```
static int sa = sizeof(struct InnerStruct);

// MyTest函数具有文件作用域
static void MyTest(void)
{
// 尽管宏MY_MACRO定义在MyTest函数内，但它仍然具有文件作用域，不受函数语句块的影响
    #define MY_MACRO    100
}

// main函数处于文件作用域
int main(void)
{
    struct OutStruct out = { 10, { 100 }, MY_ENUM_2 };

    // 这里可直接访问InnerStruct结构体类型
    struct InnerStruct inner = { 20 };

    // 这里也可以直接访问宏MY_MACRO
    printf("The value is: %d\n", out.a + out.inner.i + inner.i + sa -
           MY_MACRO);
}
```

代码清单11-1中我们可以清晰地看到，在具有文件作用域的结构体内再定义一个结构体，其内部结构体也具有文件作用域。所以，为了避免全局名字空间的污染，我们在编写C语言程序时应当尽量避免在文件作用域中定义的结构体、联合体内定义命名结构体或联合体，取而代之的是，可以用匿名结构体或联合体。

此外，代码清单11-1中我们还看到了宏不受其他作用域的影响，始终保持着文件作用域的特性，这也是预处理指示符的特性。

11.1.2 函数作用域

C语言标准明确规定，跳转标签名是仅有的具有函数作用域的标识符。函数作用域从一个函数体的开始（即紧跟在{符号的后面），一直到函数体结束（即紧跟在}符号前面）。跳转标签可以出现在函数体中的任何位置，在使用时，随着goto语句一起出现。

代码清单11-2将简单描述仅有的作为函数作用域标识符的跳转标签名。

代码清单11-2　具有函数作用域的标识符（跳转标签）

```
#include <stdio.h>
int main(void)
{
    int count = 5;
    // 这里声明了一个HELLO_LABEL跳转标签，它具有函数作用域
HELLO_LABEL:
    puts("Hello, world!");
    if(--count > 0)
        goto HELLO_LABEL;   // 跳转到HELLO_LABEL标签处
    switch(count)
    {
        case 1:
            count = 0;
            break;
        case 0:
            count = 3;
            // 这里尽管在switch语句块内声明SWITCH_INNER_LABEL跳转标签
            // 但它仍然属于函数作用域
        SWITCH_INNER_LABEL:
            printf("count = %d\n", count);
            break;
        default:
            break;
    }
    if(--count > 0)
        goto SWITCH_INNER_LABEL;    // 跳转到HELLO_LABEL标签处
}
```

代码清单11-2可以清晰地看到，跳转标签始终具有函数作用域，而不受其他语句块作用域的影响。这也是为什么C语言标准将跳转标签专门独立出一个函数作用域的原因。

11.1.3　函数原型作用域

C语言标准指出，如果一个标识符是声明在一个函数原型（该函数原型不作为函数定义的一部分）中的形参列表内，那么该标识符则具有函数

原型作用域。函数原型作用域从形参列表的“（”开始，到函数声明符的末尾结束。

不过当前主流编译器中，普遍不满足函数原型作用域到函数声明符的末尾结束，而一般是将作用域范围缩短到了形参列表中紧跟）符号之前。所以我们在利用函数原型作用域中的标识符的时候需要注意实现上的差异性。代码清单11-3展示了函数原型作用域的概念。

代码清单11-3　函数原型作用域

```
/** 这里，形参标识符a以及形参标识符pArray都具有函数原型作用域。
 * 我们可以看到，在声明pArray形参的时候还用到了形参a的标识符，
 * 说明形参a标识符在此形参列表中可见。
 * 在Clang编译器中，如果我们把[sizeof(int)]中的int改为a，
 * 那么编译器就会报错：标识符'a'没有声明
*/
static int (*Foo(int a, int (*pArray)[sizeof(a)]))[sizeof(int)];
```

这里补充说明一下，代码清单11-3中，函数Foo被声明为具有int（*）[sizeof（int）]指向数组的指针返回类型，带有int类型的形参a以及带有int（*）[sizeof（a）]指向数组的指针类型的形参pArray。

11.1.4　语句块作用域

C语言标准指出，如果一个标识符的声明出现在一个语句块内，或是出现在一个函数定义中的形参声明列表中，那么该标识符具有语句块作用域。语句块作用域是从当前语句块的{开始一直到}结束。

这里要注意的是，之前也提到过，结构体、联合体、枚举类型定义时的花括号不算语句块，它们均属于类型说明符（type specifier）的一部分。在结构体与联合体中，花括号内的数据对象属于该结构体或联合体的成员，数据对象成员不具有“作用域”这个属性；如果是内嵌定义命名结构体或联合体，那么内嵌的结构体或联合体的作用域与其外部结构体或联合体的作用域一样。枚举类型中的枚举常量，其作用域与枚举类型一样。代码清单11-4展示了语句块作用域以及上述提到的一些注意事项。

代码清单11-4　语句块作用域的描述

```
#include <stdio.h>

// MyStruct结构体具有文件作用域
struct MyStruct
{
    int a;

    // 下面这个数组对象声明会引发错误，由于成员a不作为任何作用域，
    // 因此在声明结构体成员对象的时候无法被引用
    int b[sizeof(a)];

    // InnerStruct结构体同样具有文件作用域
    struct InnerStruct
    {
        int i;
    } inner;

    // 由于InnerStruct具有文件作用域，
    // 因此它可以在对象成员声明中被引用，而inner对象则不行
    int c[sizeof(struct InnerStruct)];
};

// MyEnum具有文件作用域
enum MyEnum
{
    // MyEnum1与MyEnum2两个枚举常量都具有文件作用域
    MyEnum1 = 1,
    MyEnum2,

    // 由于MyEnum1与MyEnum2两个枚举常量都具有文件作用域，
    // 所以它们可以在声明后续枚举常量中被引用
    MyEnum3 = MyEnum1 + MyEnum2
};

/** 这里的函数Foo具有文件作用域，并且由于它是一个函数定义，
 * 所以形参对象a以及arr具有语句块作用域，并且第二个形参arr则直接引用了标识符a。
 * 不过在Clang实现中，a仍然不能在()形参列表外可见，但可以在函数体内可见。
 * 因此，这里sizeof(int)中的int改为a就会编译报错。
 * 这里Foo函数的返回类型为：int (*)[sizeof(int)]
*/
```

```
static int (*Foo(int a, int arr[sizeof(a)]))[sizeof(int)]
{
    // 由于形参对象a具有语句块作用域，因此在函数体内完全可以被访问
    printf("a = %d\n", a);

    // 由于形参a已经在Foo函数的语句块作用域，因此这里不能再以标识符a来声明一个对象
    // int a = 10;

    // 由于在函数体内，Foo函数已经具有完整的函数原型，因此可以在语句块作用域内被引用
    printf("size of return type is: %zu\n", sizeof(Foo(0, NULL)[0]));

    return NULL;
}

int main(void)
{
    // 这里MyObject结构体类型具有语句块作用域
    struct MyObject
    {
        int a;

        // 这里Inn结构体也同样具有语句块作用域
        struct Inn
        {
            int n;
        } inn;
    };

    // 枚举类型TheEnum具有语句块作用域
    enum TheEnum
    {
        // 下面的TheEnum1与TheEnum2枚举常量都具有语句块作用域
        TheEnum1,
        TheEnum2
    };

    // 引用文件作用域的MyStruct与InnerStruct
    struct MyStruct ms = { 10, (struct InnerStruct){ 20 } };

    // 引用文件作用域的MyEnum
    enum MyEnum en = MyEnum2;

    // 引用语句块作用域的MyObject与Inn
    struct MyObject obj = { 30, (struct Inn){ 40 } };

    // 引用语句块作用域的TheEnum
    enum TheEnum te = TheEnum1;

    // 调用文件作用域的Foo函数
    Foo(ms.a + ms.inner.i + en + obj.a + obj.inn.n + te, NULL);
}

void foo(void)
{
    struct MyObject obj;                // 错误，MyObject无法被访问
    enum TheEnum en = TheEnum1; // 错误，TheEnum与TheEnum1无法被访问

    // 引用文件作用域的MyStruct与InnerStruct，OK
    struct MyStruct ms = { 10, (struct InnerStruct){ 20 } };

    // 引用文件作用域的MyEnum，OK
    enum MyEnum en = MyEnum2;
}
```

代码清单11-4比较详细地展示了语句块作用域的特点以及它与文件作

用域和函数原型作用域的差异性。

由于之前提到，在一个源代码上下文中，同一标识符在不同位置可指代多个不同实体，因此在C语言中，不同作用域的相同标识符是可被重定义的，下面我们将介绍C语言中标识符的重定义以及作用域的叠交。

11.1.5　标识符的重定义与作用域的叠交

C语言标准明确指出，同一标识符在程序中的不同位置可以指代不同实体。此外，在同一名字空间中，如果一个标识符指代了两个不同的实体，那么作用域可能叠交。如果在同一名字空间中有相互叠交的作用域，那么内部作用域必须在外部作用域之前结束，也就是说，内部作用域的范围是外部作用域的真子集。如果一个标识符分别在外部作用域和内部作用域声明了两个不同的实体，那么在内部作用域中，用此标识符所声明的实体将会覆盖掉（即隐藏掉）外部作用域声明的实体。

代码清单11-5介绍了标识符的重定义与作用域的叠交的作用与效果。

代码清单11-5　标识符的重定义与作用域的叠交的作用与效果

```
#include <stdio.h>
#include <stdint.h>

// 在文件作用域声明了一个整型对象sa
static int sa = 10;

// 在文件作用域定义了一个结构体类型MyTest
struct MyTest
{
    int a;
    float f;
};
```

```
/**
 * 在文件作用域声明了一个名为Foo的函数，
 * 其中该函数的形参具有函数原型作用域，这里的形参sa覆盖掉了文件作用域的sa，
 * 形参MyTest覆盖掉了文件作用域的MyTest结构体类型
*/
static void Foo(int16_t sa, double MyTest);

/**
 * 在文件作用域定义了函数Foo，其形参列表中的形参对象具有语句块作用域。
 * 其中该函数的形参具有函数原型作用域，这里的形参sa覆盖掉了文件作用域的sa，
 * 形参MyTest覆盖掉了文件作用域的MyTest结构体类型
*/
static void Foo(int16_t sa, double MyTest)
{
    printf("sa = %d, MyTest = %f\n", sa, MyTest);

    // 这条语句信息量很大!
    // 这里通过显式加上struct以指明所使用的MyTest是文件作用域的结构体类型。
    // 然后，.f = MyTest 表达式中所引用的MyTest标识符指代的是形参对象。
    struct MyTest mt = { .a = sa, .f = MyTest };

    printf("The sum result: %f\n", mt.a + mt.f);
}   // 这里，Foo函数语句块作用域结束

int main(int argc, const char* argv[])
{
    // 这里调用文件作用域的Foo函数，并引用了文件作用域的sa整型对象
    Foo(sa, 100.5);

    // 在main函数体的语句块作用域声明了整型对象aa
    int aa = 0;

    if(sa > 0)
    {
        // 这里if语句块的作用域覆盖了main函数体的语句块作用域

        // 这里在if语句块作用域中声明了sa整型对象，将文件作用域的sa给覆盖掉了
        int sa = aa;

        // 这里在if语句块作用域中声明了整型对象aa，将其外部语句块的aa对象覆盖掉。
        // 这里要注意的是，在声明语句中一旦出现了像这里的int aa 对象标识符的声明，
        // 那么该对象标识符立即生效。所以= 操作符右边的aa也表示这里刚声明的对象aa，
        // 而不是外部语句块作用域中的aa。所以对于以下语句编译器会发出警告：
        // “变量aa未被初始化，由于用它自身给自己初始化了”
        int aa = aa + sizeof(aa);
        aa = sa - 10;

        // 这里printf函数中的实参aa与sa都是此if语句块作用域中声明的对象，
        // 所以这里aa是-10，sa为0，结果为-10
        printf("inner sum = %d\n", aa + sa);
    }   // 这里，if语句块作用域结束

    // 这里printf函数中的实参aa是此main函数体语句块作用域中声明的对象，
    // sa则是文件作用域中声明的对象。这里aa是0，sa是10，所以结果是10。
    printf("outer sum = %d\n", aa + sa);

    // 在for语句中声明的对象具有当前for语句块的作用域。
    // 因此，这里的aa把外部声明的aa对象给覆盖掉了，并且其作用域为for语句块的作用域。
    // 这里省略了{ }，但此for语句块作用域到printf函数调用的;符号后结束
    for(int aa = 0; aa < 10; aa++)
        printf("loop aa = %d\n", aa);

    // 这里外部aa的值仍然为0
    printf("outer aa = %d\n", aa);

    // 这里在main函数语句块作用域定义了MyTest结构体类型，
    // 将文件作用域中定义的MyTest结构体类型给覆盖掉了
    struct MyTest
```

```
    {
        float f;
        int i;
    };

    // 这里引用的是main函数语句块作用域中的MyTest结构体
    struct MyTest mt = { -10.5f, 100 };

    if(aa == 0)
        goto Foo;

    puts("Hello, world!");

    // 这里在函数作用域中声明了作为跳转标签的Foo
Foo:
    // 尽管存在表示跳转标签的Foo，但这并不妨碍对Foo函数的调用。
    // 因为当Foo出现在goto后面，则表示跳转标签；
    // 作为函数调用表达式出现，则作用函数标识符，这一点C语言编译器能做出准确区分。
    // 它们属于两种不同的名字空间
    Foo(mt.i, mt.f);
    puts("Completed!");
}
```

代码清单11-5展示了比较完整的标识符重定义以及作用域叠交特性。在不同作用域，如果标识符所指代的类别不一样，编译器可以通过表达式以及语句的表达形式做出区分，比如是结构体类型还是某个对象标识符，还是函数标识符或跳转标签。如果是相同类别的，那么内部作用域的对象会把外部作用域的给覆盖掉。

11.1.6 标识符的名字空间

本节是对上一节的补充说明。C语言标准明确指出，如果同一标识符的多个声明在某个翻译单元中的任一点可见，那么C语言实现的语法上下文将通过不同实体类别来消除歧义。所以，在C语言中对于不同类别的标识符具有自己独立的名字空间（namespace）。

C语言标准指定了4类名字空间：

·跳转标签名：跳转标签可以在其声明与使用过程中在语法上加以区分。

·结构体、联合体以及枚举的标签（tag）：根据关键字struct、union以及enum加以区分。注意，这三者属于同一名字空间。

·结构体与联合体的成员（member）：通过结构体或联合体对象的成员访问操作符“.”或“->”进行区分。

·所有其他标识符，也被称为普通标识符：以普通（ordinary）声明符声明的标识符，或是作为枚举常量。

这里要注意的是，同一类别的名字空间是无法进行区分的，而C语言标准将结构体、联合体以及枚举标签都放在一类，这就说明了结构体、联合体以及枚举的类型标识符是无法同时存在于同一作用域内的。代码清单11-6进一步描述了在同一作用域内不同名字空间类别的标识符的使用。

代码清单11-6　标识符的名字空间

```
#include <stdio.h>

// 在文件作用域声明了一个整型对象sa
static int sa = 10;

// 在文件作用域定义了一个结构体类型sa,
// 由于结构体标签与普通对象标识符属于不同的名字空间，因此这里可以直接使用
struct sa
{
    int a;
    float f;
};

/**
 * 在文件作用域定义了一个枚举类型SA。
 * 由于枚举标签与结构体标签属于同一名字空间,
 * 因此这里不能使用sa作为枚举类型的标识符
*/
```

```
enum SA
{
    // 由于枚举常量与对象标识符同属于一个名字空间,
    // 因此这里不能用sa作为枚举常量标识符
    SA1,
    SA2
};

/**
 * 在文件作用域中定义了联合体类型un,
 * 同样, 由于联合体标签与结构体标签属于同一名字空间,
 * 因此这里不能使用sa作为联合体类型的标识符
*/
union un
{
    // 这里可以用sa作为该联合体的成员
    int sa;

    // 同样, 这里也能用un作为该联合体的成员
    float un;
};

/**
 * 这里在文件作用域定义了函数SA。
 * 由于SA属于其他标识符的名字空间, 因此与枚举类型不冲突。
 * 此外, 形参对象sa属于语句块作用域, 因此与文件作用域的sa也不冲突
*/
static void SA(int sa)
{
    // 这里引用的是SA函数的语句块作用域中的形参对象sa
    printf("sa = %d\n", sa);
}

int main(int argc, const char* argv[])
{
    // 这里用文件作用域的枚举类型SA声明了对象es,
    // 并用SA2枚举常量对其初始化
    enum SA es = SA2;

    // 这里调用了文件作用域的函数SA,
    // 并用文件作用域对象sa与语句块作用的es对象的和作为实参
    SA(sa + es);

    // 这里用文件作用域的sa结构体类型声明了对象s
    struct sa s = { es, 10.5f };

    // 这里用文件作用域的联合体类型un声明了对象un,
    // 并用语句块作用域的对象s的成员f对其un成员进行初始化
    union un un = { .un = s.f };

    // 这里访问的是声明在语句块作用域的对象un的un成员
    printf("un is: %f\n", un.un);

    // 以下这条语句错误! 因为以下语句声明的es标识符属于其他标识符名字空间,
    // 而在此语句块作用域内, 已经声明了es标识符作为枚举类型的对象, 它也属于其他标识符名字空间
    int es = 0;

    // 在语句块作用域内重新定义了枚举类型SA
    enum SA { SA1, SA2, SA3 };

    // 这句声明符没有问题。因为这里声明的SA作为数据对象,
    // 而之前的SA标识符用于枚举类型
    int SA = 0;

    // 以下这条语句错误! 因为SA1在此语句块作用域内已经被用作枚举常量
    // 而数据对象标识符的名字空间与枚举常量标识符同属于其他标识符名字空间,
    // 因此会产生冲突
    int SA1 = SA;
```

```
    // 这句声明没有问题，在语句块作用域声明了sa对象，并用SA对象对其初始化。
    // 这里，sa对象覆盖了文件作用域的sa对象
    int sa = SA;

    if(sa == 0)
        goto SA;

    puts("Dummy output!");

    // 这里SA用作为跳转语句标签，因此与上面同一语句块作用域中的用作枚举类型，
    // 以及用作整型对象的标识符不冲突。
    // 而C语言更是将跳转标签专门独立出一个作用域——即函数作用域
SA:
    SA = sa + 1;
    printf("SA = %d\n", SA);
}
```

代码清单11-6介绍了在同一作用域内，同一标识符能同时指代不同实体的效果。当然，为了不引起混淆，各位在实际项目中应当尽量避免标识符重叠的情况，尤其像函数内的局部标识符覆盖了文件作用域的全局标识符，有很多难以发现的Bug都是这种标识符重叠所产生的。因此，这里对作用域与名字空间详细的介绍不是为了鼓励程序员有针对性地加以利用，更多的是告诉C语言编译器的实现者，让这些开发者知道C语言标准对作用域以及名字空间的实现情况，使得自己开发的C语言编译器能遵循C语言标准。

讲完了C语言程序的作用域与名字空间。下面三节将介绍C语言中对象与函数的连接（linkage）。C语言标准中，一个对象或函数标识符可以在不同作用域或同一作用域内进行多次声明，而这些重复的声明可以通过称为连接的过程来引用同一对象或函数。C语言中一共有三种连接类型：外部连接（external linkage）、内部连接（internal linkage）与无连接（none）。下面我们将分别介绍这些相关内容。

11.2　全局对象与函数

用extern存储类说明符（storage-class specifier）声明的对象与函数具有外部连接。对于一个数据对象，如果只有含extern的声明，那么它还不具有一个实体，只有当它在文件作用域中用不带extern的声明之后才具有实体，并且作为一个全局对象。对于一个函数，如果用extern或缺省存储类说明符对函数声明，那么该函数具有外部连接。如果只声明了一个函数原型，那么该函数也没有实体，只有对该函数定义之后才有实体。在定义函数时，如果用extern或缺省存储类说明符进行定义，那么函数具有外部连接，并且称该函数为全局函数。

另外，C11标准也明确指出，在C语言未来版本中，存储类说明符应该只能放在整个对象与函数声明的最前面，而不能放在其他位置。

全局对象与函数的声明在同一翻译单元中可出现多次，即便是在同一作用域中。然而，对对象的定义只能有一次。如果在声明一个全局对象的同时，又对它进行初始化，那么此时这声明就对该对象进行了定义，同时该对象也具有了实体。对全局对象与函数的定义只能有一次，否则在连接时，连接器会报重定义外部符号的连接错误。

代码清单11-7展示了全局对象与函数的声明与定义。

代码清单11-7　全局对象与函数的声明与定义

```
/** 以下是 test.c 源文件里的内容 */
// 声明了具有外部连接的全局对象ga
extern int ga;

// 这里再次声明具有外部连接的全局对象ga，
// 然而此时ga对象仍然不具备实体。
// 另外，由于ga的实体是在main.c翻译单元中产生的，因此这里对ga的声明必须加extern，
// 否则会引发连接时的符号重定义错误
extern int ga;

// 声明了具有外部连接的全局函数test
extern void test(void);

// 再次声明具有外部连接的全局函数test，
// 这里缺省了存储类说明符extern，但默认为extern
void test(void);

// 这里对test进行定义。
// 在函数定义时，通常extern缺省，也表示当前函数为具有外部连接的全局函数
void test(void)
{
    ga = 100;
}
/** 以下是 main.c 源文件中的内容 */
#include <stdio.h>

// 声明了全局对象ga，并且此时ga具有了实体
int ga;

// 再次声明了全局对象ga
int ga;

// 这里声明了全局函数test，使得test标识符在当前翻译单元中可见。
// 由于在test.c中已经对函数test做了定义，所以在main.c中就不能再次对它定义，
// 否则在连接时会报test符号重定义的错误
extern void test(void);

// 声明了具有外部连接的全局对象ma，此时ma已具备实体
int ma;

// 再次声明了全局对象ma，并且同时对它进行初始化。此时，这个声明就是对ma的定义
int ma = 10;

// 这里仍然可以对ma做声明
int ma;

// 这里错误！由于之前对全局对象ma做过初始化，
// 因此这里再进行初始化会引发外部符号重定义的连接错误
int ma = 20;

int main(int argc, const char* argv[])
{
    // 全局函数与对象的声明也可放在语句块作用域，
    // 使得当前作用域对此标识符可见
    void test(void);

    // 在语句块作用域中对具有外部连接的对象进行声明时，
    // 必须添加extern存储类说明符，否则声明的标识符将不具有连接，
    // 而是作为该语句块作用域中的局部对象，而覆盖掉文件作用域的全局对象
    extern int ma;

    // 调用全局函数test
    test();

    // 在当前的main.c翻译单元以及test.c翻译单元中，
    // 全局对象标识符ga都指代同一个实体；而全局函数test也是同一个函数实体
    printf("result is %d\n", ga + ma);
}
```

代码清单11-7中含有两个源文件test.c与main.c的代码，这样可以将这两个源文件分别作为两个不同的翻译单元，从而体现出具有外部连接的全局对象与函数标识符都引用的是同一个实体。各位在上机实践时，也是分别用两个源文件来输入这些代码，并且将这两个源文件放在同一项目工程目录下的。代码清单11-7展示了具有外部连接的全局对象与函数的特性以及在使用时需要注意的地方。这里要注意的是，在一个源文件中对具有外部连接的对象或函数做了定义，使得它有了实体之后，那么在其他源文件中就不能再次对它进行定义，否则会引发连接时的符号重定义错误。在其他源文件中，为了能使得当前翻译单元识别到此符号的存在，只能对具有外部连接的全局对象或函数进行声明。另外，还需要注意的是，对不产生实体的全局对象做外部声明时需要加上extern存储类说明符，否则也会引发连接时的符号重定义错误。

11.3 静态对象与函数

用存储类说明符static修饰的对象与函数称为静态对象与函数。C语言标准明确指出：如果一个对象或函数的声明在一个文件作用域中，并且包含static存储类说明符，那么该对象或函数具有内部连接。然而，与全局对象不同，静态对象除了可以定义在文件作用域之外，还可以定义在语句块作用域中，而定义在语句块中的静态对象没有连接。与全局对象与函数类似的是，静态对象与函数的声明也可以在同一文件作用域中出现多次，但是在同一文件作用域中对静态对象的初始化只能出现一次；同样，对静态函数的定义也只能出现一次。对C语言中的函数而言，无论是具有外部连接的全局函数还是具有内部连接的静态函数，都只能在文件作用域内进行定义，同时也只有具有外部连接的全局函数才可以在语句块作用域中声明，而具有内部连接的静态函数则不允许在语句块作用域内声明。同时，如果一个静态对象声明在语句块作用域中，那么不能对它进行重复声明。因为在语句块作用域中，对一个静态对象的声明就已经相当于对它的定义，它已经被实例化了，无论它有没有同时被初始化。

另外，与外部连接不同的是，具有内部连接的静态对象与函数并不是在整个程序或库中指代同一个实体，而是在其各自的翻译单元中指代同一对象或函数。而一个翻译单元也就是我们通常所谓的一个C源文件。因此，如果我们在两个C源文件，比如a.c和b.c中，都在文件作用域定义了一个静态对象，比如——“static int staticObject；”，那么当编译之后，a.c编译

之后所产生的目标文件a.o与b.c编译后所产生的目标文件b.o中各自都含有一个名为staticObject的、具有内部连接的整型对象实体，因此当连接器在连接之后，这两个对象将是同时存在的，而它们在各自的翻译单元中分别表示当前所处翻译单元中的状态。

静态对象的情况比较复杂，我们通过代码清单11-8这一详细的示例代码做细节上的进一步讲解。

代码清单11-8　静态对象与函数

```
/** 以下是 test.c 源文件里的内容 */
#include <stdio.h>

// 在test.c翻译单元中声明静态对象sa，并对它用-10进行初始化
static int sa = -10;

static void SFoo(void)
{
    sa++;

    printf("sa in %s = %d\n", __FILE__, sa);
}

// 这里定义了具有外部连接的全局函数test，用于间接调用具有静态连接的SFoo函数以观察结果
void test(void)
{
    SFoo();
}
/** 以下是 main.c 源文件中的内容 */
#include <stdio.h>

// 在文件作用域声明静态对象sa，sa具有内部连接
static int sa;

// 这里再次声明静态对象sa，并对它进行初始化
static int sa = 100;

// 这里对静态对象sa进行再次声明，但不能再进行初始化
static int sa;

// SFoo函数在文件作用域内被定义为静态函数，从而具有内部连接
static void SFoo(void)
{
    // 在语句块作用域中声明静态对象，则立即对它进行了实例化的定义。
    // 声明在语句块作用域的静态对象尽管没有连接，但它的行为就类似于文件作用域的静态对象。
    // 当第一次调用SFoo函数，这里inner静态对象即被初始化。
    // 而当第二次调用SFoo函数时，inner静态对象不会被再次初始化，但还保留着之前的值
    static int inner = 100;

    // 下面再用static对inner对象进行声明，编译器会报inner重定义的错误
    // static int inner;
```

```
    // 在每次调用SFoo函数时，都对inner静态对象做一次自增操作
    inner++;

    printf("inner = %d, sa = %d\n", inner, sa);
}

int main(int argc, const char* argv[])
{
    // 以下声明错误！在语句块作用域中不允许声明静态函数
    static void SFoo(void);

    // 第一次调用SFoo函数时，inner静态对象被初始化，随后做了一次自增操作
    // 所以这里将输出：inner = 101
    SFoo();

    // 当第二次调用SFoo函数时，inner静态对象没有再次初始化，
    // 但保留了上次的结果101，并且随后也做了一次自增操作，所以这里输出：inner = 102
    SFoo();

    // 这里声明test全局函数
    extern void test(void);

    // 通过两次调用定义在test.c中的全局函数test，
    // 以观察在test.c中定义的sa静态对象以及SFoo静态函数的状态
    test();
    test();

    // 这里声明静态sa将把文件作用域的静态对象sa给覆盖掉，
    // 同时这里的sa将不具有连接
    static int sa = 0;

    printf("inner sa = %d\n", sa);

    // 我们通过再次调用SFoo函数可以观察到文件作用域中定义的sa对象的当前值
    SFoo();
}
```

代码清单11-8也同样创建了两个C源文件，一个名为test.c，另一个名为main.c。我们看到，test.c中的sa静态对象与main.c中的sa静态对象是两个不同的实体，两者都分别作用在自己的翻译单元上下文中。同样，test.c中的SFoo静态函数与main.c中定义的SFoo静态函数也是两个不同的函数实体，里面的实现也完全不一样，但标识符完全一样，它们具有内部连接。因此连接器在做符号连接时，在当前目标文件中具有内部连接的符号就作为当前上下文的一个实体，而对于具有外部连接的符号，则需要全局考虑。分布在各个目标文件中的同一个具有外部连接的符号，最终都将指代为同一个实体。所以，从这点上来看，具有外部连接的对象与函数是名副

其实的全局对象与函数。

11.4 局部对象

声明为一个函数形参、或者在一个语句块作用域内没有用extern或static存储类说明符声明的对象称为局部对象（local objects）。之前提到过，声明在语句块作用域中的静态对象不具有连接，在此对于不具有连接的对象的范围又进一步扩充了——局部对象也不具有连接。因此，对于对象的连接分类现在就很明确了——在文件作用域用extern存储类说明符或不用任何存储类说明符声明的对象，以及在语句块中用extern存储类说明符声明的对象具有外部连接；在文件作用域中用static存储类说明符声明的对象具有内部连接；其余的都没有连接。当然，之前也提到过，函数只能被定义在文件作用域，而只有具有外部连接的函数才可以声明在语句块作用域内。同时，函数必须具有连接，不是外部连接就是内部连接。

各位对局部对象应该已经是相当熟悉了。像作为函数的形参的对象，以及在语句块作用域中声明的不带任何存储类说明符的对象都属于没有连接的局部对象。局部对象只对它所在的作用域内可见，并且出了它所在的作用域，那么它的生命周期也就结束了。

这里，C语言中还有一个auto存储类说明符以及register存储类说明符来声明一个局部对象。在早期C语言中（尤其还没有被标准化时），auto关键字用于显式声明一个对象是局部变量，它不具有连接，其生命周期也是由函数实现自动回收的。我们之前提到过，函数中定义的局部对象往往是存

放在栈空间的，当函数返回前，该函数所用的栈会被推出，使得该函数之前所持有的局部对象全都被自动销毁，所以局部对象在那时也被称为自动的。而现代C语言（在C90标准确立后），auto关键字就被逐步废弃了。而register存储类说明符则是暗示C语言编译器，当前声明的对象最好被存放在寄存器中，由于它可能被重复使用而应该得到最快速的访问。然而，现代C语言编译器做得越来越智能，尤其是自动内联函数、循环展开优化等技术成熟之后，优化策略也丰富得多，因此当前编译器能自己更好地合理分配寄存器，我们使用register关键字往往会阻碍编译器的进一步优化，所以从C语言下一个版本的标准起，标准委员会可能会逐步弃用register关键字或者为它赋予一种新的语义，因此大家在当前的C语言代码中尽量避免这两个关键字，除非有些编译器对它们做了语义上的扩展（比如，auto在C++中已经被用作可自动推导的类型）。

代码清单11-9简单地给大家介绍了声明局部对象时的auto与register存储类说明符的使用方式。

代码清单11-9　局部对象以及auto与register存储类说明符

```
#include <stdio.h>

/**
 * 这里在文件作用域定义了具有内部连接的静态函数foo,
 * 其形参对象a和b都是无连接的局部对象。
 * 其中，形参b用register存储类说明符修饰,
 * 暗示编译器将此形参对象尽量存放在寄存器中
*/
static void foo(int a, register int b)
{
    printf("sum = %d\n", a + b);
}

int main(int argc, const char* argv[])
{
    foo(100, 200);
```

```
    // 这里声明了一个自动局部对象a,
    // 其语义与int a没有差别。此外，auto存储类说明符不能用于声明一个函数形参对象
    auto int a = 10;

    // 这里用register存储类说明符声明了对象r，暗示编译器将对象r尽量存放在寄存器中
    register int r = 20;

    // 以下语句会引发编译错误！由于寄存器类型的对象在概念上“没有存储器地址”，
    // 因此，不能对register修饰的对象做取地址操作
    int *p = &r;

    printf("value is: %d\n", a + r);
}
```

代码清单11-9中也谈到了使用register存储类说明符声明的对象在使用时的限制，既然一个对象是暗示用于存放在寄存器中的，那么它就不具有存储器地址所以我们不能将它作为地址操作符的操作数。此外，auto存储类说明符不能用于修饰一个形参对象。

11.5　对象的存储与生命周期

本节将描述对象的存储与生命周期。在C语言中，对象如果从存储区域进行划分的话，可分为全局数据存储区、栈存储区以及被动态分配的堆存储区。而全局数据存储区又可分为可读写存储区以及只读存储区。只读存储区也称为常量存储区。

之前在第1章中，我们提到了若干C语言源文件从编译到连接，最终生成可执行文件的流程。可执行文件中就包含了对函数（指令码）、全局数据分布的描述。当我们点击可执行文件，或是在控制台中输入可执行文件名然后按回车键之后，操作系统自带的特定加载器就根据可执行文件中所描述的函数以及全局对象的布局分配相应的存储空间。函数代码一般会被加载到指定的指令存储区；一些全局常量（比如字符串字面量）根据实现，可能会被存放到常量存储区；可被修改的全局数据对象则会被分配到可读写的存储区。在大部分实现中，仅有一个声明而未被初始化的全局对象会以零进行初始化，但这个行为并不是C语言标准明确指出的，而是当前大部分环境都是这么实现的。也就是说在我们的程序正式执行之前，加载器已经给代码以及全局数据分配好了存储空间，并且对全局对象完成了初始化，最后加载器会自动调用main函数使程序正式开始执行。因此，全局对象的生命周期与当前程序的一样长，直到当前运行的程序被关闭前，全局对象一直可用。这里所描述的全局对象包括具有外部连接的全局对象，以及具有内部连接的静态对象。

在一个函数内定义的局部对象以及函数形参对象，它们一般会被存放在栈存储空间。它们的生命周期从声明开始，一直到函数调用结束，最后的栈指针复位即把函数中所有定义的局部对象全都销毁。另外，这里还有一点需要提出的是，对于函数中的一个嵌套语句块作用域的局部对象，其生命周期是从其声明开始，一直到该语句块结束，而不是函数调用结束。这个是从编程语言的逻辑上来讲的，即便某个C语言实现可使得语句块作用域的对象与函数的生命周期一样长，但我们不能做这种断言。

之前已经提到过，由于函数的栈空间十分有限，如果我们要分配一段较大的存储空间，我们就需要调用malloc等C语言标准库函数来做存储空间的动态分配。动态分配的存储空间由于传统上使用堆数据结构进行管理，因此我们也把动态分配的存储空间称为堆空间（heap memory）。堆空间的生命周期从malloc成功执行后，一直到调用free之类的库函数进行释放掉。这里大家要注意的是，如果一个指针对象指向了一个动态分配的存储空间，那么后续如果还要用这个指针去指向某个动态分配的存储空间，则必须先把之前动态分配的存储空间给释放掉，否则会引发内存泄漏（memory leak）。如果在一个程序中，在某个函数中不断动态分配存储空间，却一直不释放，那么该程序所占用的内存会不断提升，最终可能导致整个系统运行缓慢，或是当前程序无法再申请到更多的存储空间而导致崩溃。用malloc库函数动态分配出来的存储空间不受栈的影响，所以即便退出当前函数，该存储空间仍然有效，直到对它调用free进行释放为止。

代码清单11-10展示了全局对象、局部对象以及动态分配存储空间的生

命周期。

代码清单11-10　对象的存储与生命周期

```
#include <stdio.h>
#include <stdlib.h>

// 声明全局对象ga，具有整个程序的生命周期
int ga;

// 声明静态对象sa，具有整个程序的生命周期
static int sa;

// 函数foo的形参对象a的生命周期到函数foo调用完成之前
static int* foo(int a)
{
    // 语句块作用域的静态对象inner也拥有整个程序的生命周期
    static int inner = 1;

    int *p = NULL;

    if(inner > 0)
    {
        int tmp = inner + a;
        p = &tmp;
    }
    // 这里用指针p去引用if语句块中的局部对象tmp。
    // 由于tmp的生命周期在逻辑上已经在if语句块结束后就结束了，
    // 所以这里我们不应该在实际项目中做这样的引用，尽管在macOS环境下能输出正确结果
    if(p != NULL)
    {
        printf("tmp = %d\n", *p);
    }

    // 这里不能返回对象a的地址，也不能返回对象p的地址以及指针p的值，
    // 因为对象a与p都是函数foo的局部对象，其生命周期在foo调用结束后就全都结束了。
    // 而静态对象inner的生命周期为整个程序的生命周期，因此可以将它地址返回出来，
    // 然后在其他函数中进行使用
    return &inner;
}

static int* test(void)
{
    // 动态分配了100个int对象存储空间
    int *p = malloc(100 * sizeof(*p));

    for(int i = 0; i < 100; i++)
        p[i] = i;

    // 由于动态分配的存储空间会被一直保留，直到它被释放为止。
    // 因此，这里返回动态分配存储空间的首地址不会有任何问题
    return p;
}

int main(int argc, const char* argv[])
{
    // 在macOS运行环境下，ga与sa都被加载器初始化为0
    printf("ga + sa = %d\n", ga + sa);

    int *p = foo(100);

    // 这里通过指针对象p间接地修改了foo函数中静态对象inner的值。
```

```
    // 此时，inner的值变为了101
    *p += 100;

    foo(-100);

    // 调用test函数之后，将动态分配的首地址传给main中声明的指针对象p
    p = test();

    printf("p[0] = %d, p[99] = %d\n", p[0], p[99]);

    // 释放p所指向的动态分配的存储空间，
    // 在test函数中动态分配的存储空间生命周期结束
    free(p);
}
```

代码清单11-10分别展示了全局对象、静态对象以及局部对象的生命周期。我们尤其要注意的是，如果一个函数返回的是指针对象类型，那么返回的指针对象应该指向一个全局对象、静态对象或是动态分配存储空间的首地址。另外在一般系统中，字符串字面量属于全局常量，其任一元素的地址都可作为函数的返回值。

11.6 _Thread_local对象

_Thread_local关键字首次在C11标准中出现，它也是一个存储类说明符，用它声明的一个对象表示该对象在任一线程中是私有的。_Thread_local可以与extern或static一同声明，但是一个_Thread_local对象必须要么具有外部连接，要么具有内部连接。_Thread_local对象的生命周期为“线程存储周期”（thread storage duration），其整个生命周期从线程启动执行一直到线程退出。当线程启动执行之前，_Thread_local对象会为当前线程制作一个对象副本，然后将其值拷贝到该副本中，此后，无论在线程中如何修改副本值，其声明的原版对象的值是不受影响的。因此，_Thread_local对象也被称为线程私有对象。

代码清单11-11展示了_Thread_local对象的特征与使用方式。

代码清单11-11 _Thread_local对象的使用

```
#include <stdio.h>
#include <stdbool.h>
#include <pthread.h>

// 声明一个静态_Thread_local对象ta，它具有内部连接。
// 此外，它是原版的_Thread_local对象
static _Thread_local volatile int ta = 1;

// 声明一个静态布尔对象isComplete，此对象用于标识用户线程是否执行完的标志
static volatile bool isComplete = false;

/** 这是一个用户线程处理函数，用于执行一个用户线程分派的任务 */
static void* MyThreadProcedure(void* param)
{
    // 在启动用户线程之前，
    // 系统会将文件作用域声明的静态ta对象复制到当前线程作为一个副本。
    // 因此，一开始在当前用户线程中ta副本的值为1
    printf("Firstly, ta in user thread is: %d\n", ta);

    // 这里将当前用户线程的ta副本赋值为100
    ta = 100;
```

```
    printf("ta in user thread = %d\n", ta);

    // 将用户线程执行完成标志设置为true
    isComplete = true;

    return NULL;
}

int main(int argc, const char* argv[])
{
    // main函数的执行是在主线程上，
    // 一开始静态_Thread_local对象ta被复制到当前线程，
    // 那么主线程就有了ta的一个副本。尽管我们在主线程中仍然可以引用ta，
    // 但它已经不是之前在文件作用域声明的那个ta了，而是在主线程中的一个独立副本
    printf("ta = %d\n", ta);

    // 这里将主线程的ta副本赋值为20
    ta = 20;

    pthread_t thread = NULL;
    // 创建用户线程并执行
    pthread_create(&thread, NULL, &MyThreadProcedure, NULL);

    // 等待用户线程执行完毕
    while(!isComplete);

    // 输出主线程中ta副本的值
    printf("ta in main thread = %d\n", ta);
}
```

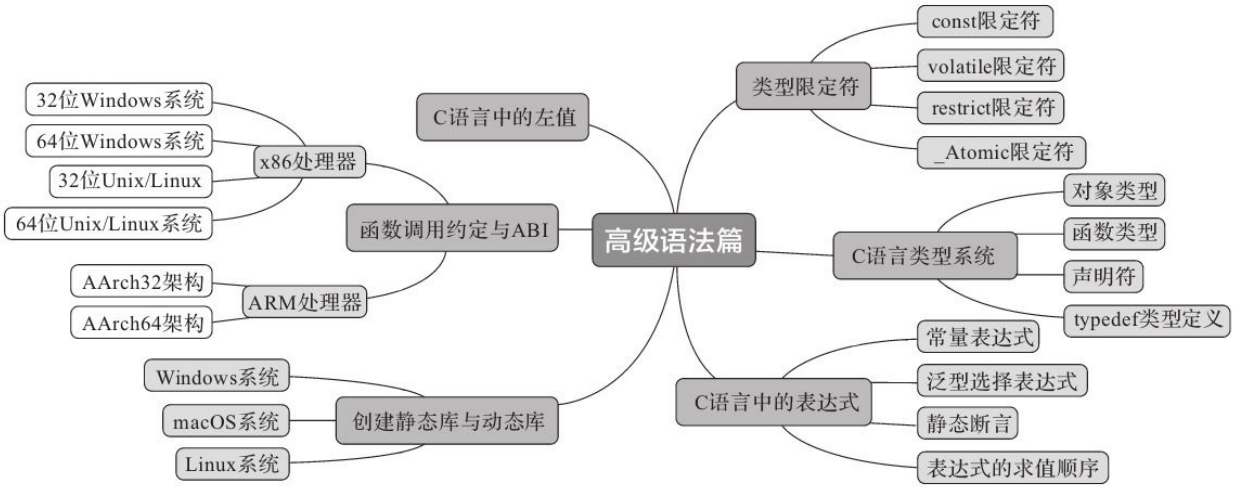
代码清单11-11使用了现在几乎所有类Unix操作系统自带的pthread库，Windows系统也有对pthread的支持，但可能需要开发者自己添加pthread的连接库。如果各位无法正常编译连接代码清单11-11，可以在网上查询如何使用pthread库，这方面的资料非常多。此外，代码清单11-11中还用到了volatile关键字，此关键字将在12.2节做详细介绍。

11.7　本章小结

本章主要给大家介绍了C语言代码编译的上下文以及不同种类的对象、存储空间在运行时的行为和生命周期。通过本章学习，大家可以掌握如何控制好对象与函数的连接，应该对公共开放的函数和全局对象作为外部连接进行声明，而仅在单个源文件内使用的函数和对象应该使用内部连接。此外，在函数中声明的局部对象在出了相应的语句块之后，其生命周期即结束。如果要持续保留对象的有效性，应该使用动态分配存储空间。另外，如果需要一个很大的内存作为缓存使用也应该使用动态分配的存储空间，因为一个函数的栈空间往往比较小。而对于文件作用域声明的全局对象以及静态对象则具有整个程序的生命周期。

在本章最后一节谈到了_Thread_local对象，这种类型的对象是声明并初始化了之后，它的值不会被修改。当每个线程在执行之前，系统都会将该对象的值拷贝到当前线程的特定存储空间，作为当前线程的一个副本使用，所以每个线程内对该对象标识符的引用其实都是当前线程对它所拥有的相应对象副本的操作，不会影响到全局声明的对象本身。不过当前支持_Thread_local的主流编译器有GCC和Clang，其他编译器可能对该特性的支持比较有限。如果大家在用GCC 4.9或更高版本，或者是Clang 3.7（或Apple LLVM 7.0）或更高版本，则可以放心大胆地使用。

第三篇　高级语法篇


高级语法篇
C语言中的左值
函数调用约定与ABI
x86处理器
32位Windows系统
64位Windows系统
32位Unix/Linux
64位Unix/Linux系统
ARM处理器
AArch32架构
AArch64架构
创建静态库与动态库
Windows系统
macOS系统
Linux系统
类型限定符
const限定符
volatile限定符
restrict限定符
_Atomic限定符
C语言类型系统
对象类型
函数类型
声明符
typedef类型定义
C语言中的表达式
常量表达式
泛型选择表达式
静态断言
表达式的求值顺序

第12章　C语言中的类型限定符

C语言中的类型限定符（type qualifier）用于指明一个对象的访存属性。C11标准中一共含有4种类型限定符，分别是const、volatile、restrict以及_Atomic。除了_Atomic这一类型限定符比较特殊外，对于其余三个类型限定符，当一个对象为指针类型的对象时，类型限定符与*号之间的摆放位置不同，该限定符所修饰的类型也会有所不同。此外，这些类型限定符可叠加使用。

我们将在12.1节中详细描述除_Atomic以外的其他三种类型限定符的摆放位置与修饰类型之间的关系，后面几节将不再赘述。而类型限定符的摆放与对类型的修饰也是C语言中的难点之一，希望各位能仔细阅读12.1节中的内容。

12.1 const限定符

const限定符用于修饰一个对象，表明该对象是一个常量（constant）。被const修饰的对象只能初始化一次，之后它的值就不能被修改。在很多嵌入式系统中，全局const对象可能会与代码一起被存入ROM存储介质中。

当const限定符用于修饰一个对象时，如果将它放置于紧挨着对象标识符的前面，那么表明该const修饰此对象本身，该对象的值就不能被修改。在这种情况下，const也可以放置在类型的前面。像下面两条语句分别将对象a与对象b声明为常量对象。

```
int const a = 100;          // 这里声明常量对象a，它具有int类型
const float b = 10.5f;      // 这里声明常量对象b，它具有float类型
```

上述代码中，无论是对象a还是对象b，都不能对它们的值进行修改。倘若通过指针做间接引用进行修改，那么行为是未定义的，比如以下代码片段：

```
int const a = 10;           // 定义了常量对象a
void *p = (void*)&a;        // 这里通过万用指针对象p指向对象a的地址
*(int*)p += 10;             // 然后通过指针p来修改对象a的值
```

我们尝试在某个函数中编写上述代码然后运行，在一般桌面操作系统中常量对象a的值往往会变为20，由于常量对象a的存储空间在栈空间，栈存储空间本身是一个可读可写的存储空间，因此在这种情况下即便C语言编译器会对常量对象在编程语言语法上进行保护，但也不能保证在运行时

常量对象的值一定是不变的。不过我们仍然不应该通过这种方式去修改一个常量对象。C语言标准明确指出，通过指针解引用（dereference）的方式去修改一个常量对象，其行为是未定义的。所谓“解引用”是指通过对指针对象做间接操作以访问该指针所指对象的值。当然，在某些嵌入式系统中，如果对一个存入ROM的全局常量对象进行修改，那么在运行时该常量的值要么不变，即该操作被存储器控制器视作为一个无效操作；要么系统直接发生异常。

这里再谈谈const的位置放置问题。在大部分源代码中，我们会看到const修饰一个对象时，往往会放在类型的前面。但在后面与指针类型结合的时候我们会发现，将const紧挨着放置于它所修饰的对象标识符之前，更容易判断当前修饰的是哪个类型，或者说哪一层解引用被视作一个常量。

const可以用于修饰任一类型的对象，包括基本类型，枚举、结构体、联合体等用户自定义类型，以及各种指针类型和数组元素类型。这里需要注意是，由于数组对象本身其地址是固定不变的，数组对象仅仅表征了一段存储空间的首地址以及对其元素的访问模式，所以C语言中没有一种限定符能用于修饰数组对象本身，限定符只能用于修饰数组元素。

下面将分别介绍const限定符如何修饰普通标量对象、数组元素、指针类型的对象，以及修饰后的对象的访问状态。

12.1.1 const限定符修饰普通对象

当const修饰一个普通对象时，该对象的值将不能被修改。当一个const修饰一个复合类型对象时（比如一个结构体对象），那么该结构体对象中的所有成员的值都不能被修改。

代码清单12-1将展示这些常量对象的声明与使用。

代码清单12-1　const修饰普通对象

```
#include <stdio.h>

int main(int argc, const char* argv[])
{
    // 声明了一个int类型的常量对象a，但没有对它初始化
    int const a;

    // 即便如此，我们也不能再对常量对象a进行赋值，因此以下这条表达式是错误的：
    a = 100;

    // 声明了一个常量枚举对象e，并用MY_ENUM2枚举值对它初始化
    enum { MY_ENUM1, MY_ENUM2 } const e = MY_ENUM2;

    // 定义了一个匿名结构体，并用它声明了一个常量结构体对象s，这里的const也能放在struct前面
    struct
    {
        int a;

        struct
        {
            float f;
            double d;
        }inner;
    } const s = { .a = 10, .inner = { 10.5f, -20.5 } };

    // 这里对结构体对象s中的任一成员，包括其内嵌类型的成员，都不能修改
    s.inner.d = 30;     // 这条语句是错误的

    printf("result = %.1f\n", e + s.a + s.inner.f - s.inner.d);

    struct Object
    {
        int a, b;
        const int c;    // 这里成员对象c是一个常量
    };

    struct Object obj = { 10, 20, 30 };
    obj.a += obj.b;     // 这条语句没有问题
    // 以下这条语句将会出现编译报错，
    // 因为Object结构体中的成员对象c是常量，所以它的值不能被修改
    obj.c += 10;
}
```

代码清单12-1简单地介绍了const修饰一个普通类型的对象的使用情

景。这里很明确地描述了当const修饰一个复合类型时的特性。

下面我们将描述const修饰数组元素的情况。

12.1.2 const限定符修饰数组元素

在C语言中，一个数组对象仅仅表征了对一个连续存储空间的引用，所以在C语言实现中，一个数组对象的地址与其起始元素的首地址都是同一个地址，并且一个数组对象本身不具有“值”这个概念。因此，任一限定符都不能用于修饰一个数组对象，而只能用于修饰数组对象的元素。

代码清单12-2展示了const修饰数组元素的方式以及效果。

代码清单12-2 const修饰数组对象元素

```
#include <stdio.h>

int main(int argc, const char* argv[])
{
    // 声明了一个带有5个const int类型元素的数组对象a
    // 这里的const修饰的是a[i]，而不是a自身
    int const a[] = { 1, 2, 3, 4, 5 };

    // 以下这条语句将产生编译错误，
    // 因为数组对象a中的每个元素都是常量，不能被修改
    a[0]++;

    // 这里首先定义了一个匿名枚举，然后用该枚举类型声明了一个
    // 带有3个常量元素的数组对象e。e的每个元素类型为const枚举类型，
    // 这里的const也能放在enum之前
    enum { MY_ENUM1, MY_ENUM2, MY_ENUM3 }
    const e[] = { MY_ENUM1, MY_ENUM2, MY_ENUM3 };

    e[1] = MY_ENUM3;      // 这条语句也是无法通过编译的

    // 这里定义了一个UN联合体类型，并用该类型声明了一个带有2个常量元素的数组对象u。
    // u的每个元素的类型为const union UN，这里的const也能放在union之前。
    union UN { int a; float f; }
    const u[] = { {.a = 10}, {.f = 2.5f} };

    u[1].f = 0.0f;        // 这条语句也无法通过编译
```

```
    printf("The value is: %.1f\n", a[0] + e[1] + u[1].f);
}
```

通过代码清单12-2的例子我们可以看到，const修饰一个数组对象时，产生作用的是该数组中的每个元素。然而，对于像代码清单12-2中数组对象a的类型，它仍然被表达为const int[5]，表示具有5个const int类型元素的数组。我们要查看上述数组对象类型的话其实非常方便，比如我们要查看数组对象a的类型，我们在声明数组对象a语句下面写一句：a++；。然后，编译器会提示编译出错信息——Cannot increment value of type'const int[5]'，这就说明数组对象a的类型为const int[5]了。

下面我们将描述const所使用的最复杂的情景——与指针类型混用。

12.1.3 const限定符修饰指针类型对象

在C语言中，限定符修饰一个对象是一个非常神奇的设定，这种神奇不亚于指向数组的指针与指向函数的指针这种表达形式。我们之前已经描述了const修饰一个普通对象的例子，比如——const int a=10；表示将对象a指定为一个常量，对象a的类型为const int。而且这里const与int之间的位置可以互换，不影响语义。而当const要修饰一个指针类型的对象时，内容就丰富了。这里涉及一个问题，我们需要指定是将指针对象本身指定为常量还是将该指针对象做了间接操作之后的值作为常量，或是将两者同时作为常量。下面我们将分别列出这三种表达方式。

```
int * const p1;         // 指定指针对象p1为常量
```

```
int const * p2;         // 指定指针对象p2做间接操作后的值作为常量
int const * const p3;  // 指定指针对象p3为常量，并且对它做间接引用后的值也作为常量
```

我们先对比看一下上述代码片段中的p1对象。这里const修饰的是p1指针对象，说明p1指针对象自身是一个常量，这意味着p1一旦被声明，它的值就不允许被修改，比如：p1=NULL；这条语句就是错误的。也就是说它不能指向其他地址，但是对*p1可以做修改，比如：*p1=20；这完全可行。我们看到，修饰p1指针对象的const紧靠在p1对象标识符之前（即在p1的左侧位置），并且在*号之后（即在*的右侧位置），此时p1的类型为int*const。

p2指针对象不是一个常量，const修饰的是（*p2），说明对p2做间接操作后，其值为一个常量，不允许对它进行修改。也就是说，像p2=NULL；这条语句是完全有效的，而像*p2=10；这条语句则是非法的。p2对象的类型为const int*。

p3指针对象是一个常量，并且对它做间接操作后的值也是一个常量。这意味着，像p3=NULL；以及*p3=10；这些都是非法的。p3指针对象的类型为const int*const。

通过上述三个指针对象的不同例子，我们可以发现const限定符在修饰一个指针对象时所产生的丰富多样性。这里大家要记住的是，在声明一个指针对象时，当const限定符摆放在所有*号的后面、紧靠在对象标识符之前，那么该const限定符修饰的是指针对象本身，即指针对象不能指向其他任何对象，它的值不能被修改。当const摆放在最靠近对象标识符的*号之

前时，const修饰的实际上是*连同其后面的对象标识符，就比如（*p）的值，也就是说对*p的值不能进行修改。

所以从const所修饰的对象来看，我们就可以看它后面的*号位置。如果它后面没有*号，那么修饰的就是对象本身，否则它修饰的就是所有在此对象标识符之前、const之后的*做间接操作之后的对象值。像const int**pp；这里pp与（*pp）都不是常量，只有（**pp）才是常量。

除了从const所修饰对象的标识符这个视角看之外，还能通过const修饰的类型来看。

比如：p1是int*const类型，我们看const前面的类型，这里是int*，所以很显然这里的int*类型的对象是常量，不可被修改，也就是p1对象自身，因为p1的类型去除限定符之后就是int*类型。所以从类型角度看的话就是要看const之前的类型。

然后我们看p2的类型，它是const int*，我们不妨把const与int交换一下位置，这里的交换是不会影响语义的，只要不涉及越过*号的交换。我们看到p2的类型可描述为int const*，我们看摆放在const前面的类型是int，所以这里对于p2对象而言，（*p2）是一个常量，它是const int类型。

这么一来，当我们要声明一个指针对象，它自身是一个常量还是说对它做间接操作之后的对象是一个常量就有两种方式去判别了。当然，一般我们用这种类型判别作为一种验证方式，我们在思考时还是首选上一段所描述的const限定哪个对象的方法，不过主要还在于自己怎么思考和理解，

而不用去强求采用哪种方式。当我们确定了*相对于const的位置之后也就能确定当前的const修饰的是哪种类型——const修饰的是跟在它后面的连同*包含在一起的那个对象。像对于p1来说，const后面就是p1，所以此时const修饰的就是p1对象，而p1对象的类型就是int*const；对于p2，const后面跟的是*p2，所以const修饰的就是（*p2），而（*p2）的类型很显然就是const int；对于p3则有两个const，最前面的const后面跟着的是（*const p3），所以（*p3）就是常量类型，即const int；而后一个const紧贴着p3，说明它修饰的就是p3对象，所以p3自身就是一个常量，这么一来，p3的类型就是const int*const。这里我们可以发现，当对一个对象做一次间接操作之后，比如（*p3），其类型就看该*之前的部分，而*之后的所有限定符都可以直接无视，所以（*p3）的类型就是const int，*后面的const可以无视之。而p3类型则需要把所有的const限定符都带上。

另外，我们还能观察到这么一个用来判定当前对象的值是否可修改的规律：如果当前对象标识符的前面紧贴着const，那么该对象的值无法修改，如果前面紧贴着的是*，那么该指针对象的值则可修改。比如p1之前紧贴着const，所以p1的值无法被修改；而p2之前紧贴着的是*，所以p2可被修改，但是（*p2）前面就紧贴着const了，所以（*p2）的值就不能被修改了。

当我们掌握了这些识别const如何修饰指针对象的技巧之后，我们可以来点更复杂的情况。请见代码清单12-3。

代码清单12-3　const修饰指针对象的综合情况

```
#include <stdio.h>

// 这里定义了一个dummy函数，稍后会用到
static void dummy(int a)
{
    printf("param a = %d\n", a);
}

int main(int argc, const char* argv[])
{
    // 声明一个常量对象a，并将它初始化为10
    const int a = 10;

    // 声明一个普通对象b，并将它初始化位20
    int b = 20;

    // 声明一个变量指针对象p，并用对象a的地址对它初始化
    int const *p = &a;              // p的类型为const int *

    *p = 20;                    // 这句非法！(*p)是一个常量，其值不能被修改
    p = &b;                         // 这句没问题

    // 声明了一个常量指针对象q，并用对象b的地址对其初始化
    int * const q = &b;             // q的类型为int * const

    q = NULL;                           // 这句非法！q是一个常量，其值不能被修改
    *q += 10;                           // 这句没问题

    // 这里声明了一个变量指针对象cpp，并用指针对象p的地址对它初始化，
    // 注意，这里的const修饰的是(**cpp)，只有(**cpp)不能被修改。
    // 此外，这里(*cpp)的类型为const int *，(**cpp)的类型为const int
    int const **cpp = &p;           // cpp的类型为const int **
    *cpp = &a;                      // 这里通过间接操作，使得指针对象p又指向了a
    if(p == &a)
        puts("p points to a!");

    cpp = NULL;                         // 这句没有问题
    **cpp = 0;                          // 这句错误！因为(**cpp)是一个常量，其值不允许被修改

    // 这里声明了一个常量指针对象cqq，并用指针对象q的地址对它初始化。
    // cqq前的const修饰的是cqq，说明cqq自身是一个常量，
    // int*后面的const修饰的是(*cqq)，说明(*cqq)也是一个常量。
    // 也就是说，这里(*cqq)的类型为int * const，(**cqq)的类型为int
    int* const * const cqq = &q;    // cqq的类型为int * const * const
    **cqq += 100;       // 这句没有问题，(**cqq)的类型是int，不是一个常量
    printf("b = %d\n", b);

    *cqq = NULL;        // 这句错误！(*cqq)的类型为int * const，是一个常量
    cqq = NULL;         // 这句错误！cqq的类型为int * const * const，是一个常量

    // 声明一个数组对象arr，它含有3个int类型的元素
    int arr[3] = { 1, 2, 3 };

    // 这里声明了一个常量指针对象pArray，指向数组对象arr的地址，
    // pArray的类型为int (* const)[3]，const修饰的是pArray对象标识符，
    // 这也说明const修饰的类型为int(*)[3]，即pArray自身是一个常量
    int (* const pArray)[3] = &arr;
    (*pArray)[1] = 0;   // OK，没有问题
    pArray = NULL;      // 这句话则是非法的，pArray是常量，其值不允许被修改
    printf("arr[1] = %d\n", arr[1]);

    // 这里声明了一个指向函数的指针常量对象pFunc。
    // 这里的const修饰的是pFunc标识符，说明pFunc自身是一个常量。
    // const所修饰的类型则是void (*)(int)
    void (* const pFunc)(int) = &dummy;
    pFunc(100);                 // 没问题
```

```
    pFunc = NULL;        // 语法错误! pFunc是常量，其值不允许被修改
}
```

代码清单12-3中也体现了如何去判定一个指针对象的常量情况。正如之前提到的，对于从const修饰哪个对象而言则是看const后面的*号情况。我们把const后面的*号连同对象标识符一起放进去看，比如像对象p，它的声明符中const后面有一个*号，那么这个const就修饰了整个（*p），说明（*p）是一个常量。而对于指针对象q，const后面就只有一个对象标识符q，说明const修饰的就是q本身，那么q就是一个常量，而（*q）则不是常量。

对于一个指针对象被多个const修饰的情况我们也无需慌张，我们可以从右往左看各个const修饰的对象是啥。当我们在看某个const修饰哪个对象时可把其余的const全都忽略。比如我们看代码清单12-3中的cqq这个比较复杂的指针对象，首先在cqq之前有一个const限定符，说明这个const修饰的就是cqq自身。而对于int*后面的那个const，我们从这个const位置起从左往右看，可以看到const后面跟着的是*const cqq。正如之前所说的，我们把*之后的const都忽略掉可得到：*cqq，说明这里*之前的const修饰的是（*cqq），这意味着（*cqq）的值不允许被修改。而对于（**cqq），由于最左边的*前面没有const修饰，所以它不是常量，（**cqq）的值可以被修改。

代码清单12-3也描述了对于指向数组的指针对象以及指向函数的指针对象如何用const修饰。其实原理也一样，直接在指针对象标识符前添加

const即可。

以上讲的是const如何修饰一个指针对象的问题，而对于整个C语言类型系统而言还有一个比较重要的问题是——如何匹配一个含有const修饰的对象类型。这里除了涉及C语言的类型系统之外，还涉及一个类型安全问题。比如说，我们声明了一个常量对象const int a=10；，如果用它赋值给另一个普通变量对象，这是完全没有问题的，比如：int b=a；。因为无论变量b如何修改其值，常量a的值是不会受到影响的。但如果用一般的指针对象（如int*p=&a；）去指向某个常量对象会发生什么呢？当我们对指针对象p采用间接操作符来修改值的时候，比如*p=20；此时对象a的值就被改变了，这与对象a作为常量这一属性是相违背的。所以在C语言中，对一个常量对象做取地址操作之后的指针类型是在const后面直接加*号。比如对于“const type obj；”，&obj的类型就是const type*。这么做可使得对该指针类型做间接操作之后仍然保持常量类型。所以像上面的const int a；&a的类型为const int*，当然表示为int const*则更容易做类型判定。而一个const int*类型的指针对象是不能赋值给一个int*类型的指针对象的，除非用投射操作做类型强制转换。我们在判定一个对象取其地址之后的类型的方法也很简单——直接将原本对象的标识符变为*号即可。比如，这里的a声明为const int a，那么取其地址之后，&a的类型则是const int*（把a变成了*）。而对于代码清单12-3中的q，它声明为int*const q，那么取其地址之后，&q的类型为int*const*（把q变成了*）。

正由于存在类型安全问题，所以等号操作符的左边表达式的类型如何

去匹配等号操作符右边表达式的类型有一定学问。正如上一段所述，等号操作符右边的const int*类型不能隐式转换为等号操作符左边的int*类型；然而，等号操作符右边如果是int*类型，那么可以隐式转换为const int*类型与等号操作符左边的表达式匹配。换句话说，低限定的类型可以隐式转为高限定类型，而高限定类型则不允许隐式转为低限定类型。除了这种情况——int**不能隐式转换为const int**类型。代码清单12-4给出了int**不能隐式转换为const int**的理由。

代码清单12-4　展示const int**如何巧妙地修改了一个常量对象的值

```
#include <stdio.h>

int main(int argc, const char* argv[])
{
    // 这里先声明一个常量对象a
    const int a = 10;

    // 这里声明一个普通指针对象p，初始化为空
    int *p = NULL;

    // 这里声明一个const int**的指针对象pp，指向p的首地址。
    // 这里用投射操作做类型强制转换就是因为，
    // int**不能隐式转换为const int **类型。
    const int **pp = (const int**)&p;

    // 这里大家注意，由于*pp的类型是const int*，
    // &a的类型也是const int*，所以两者完全兼容。
    // 这条语句执行之后，p也就被间接指向了常量对象a的地址
    *pp = &a;

    if(p == &a)
        puts("p points to a!");

    // 最后通过指针对象p来间接修改常量对象a的值
    *p = 10;

    printf("a = %d\n", a);
}
```

从代码清单12-4中可以看到，倘若int**能隐式转换为const int**，那么我们通过一个中间普通指针对象就能绕过原先常量对象的访问权限，从而间接修改常量对象值的情况。这里，最具破坏力的语句就是——

*pp=&a；。由于a是一个常量对象，而（*pp）的类型为const int*，完全与&a的类型相同，所以这个赋值没有任何问题，但所呈现的问题则是这条间接操作赋值语句把常量对象a的地址传递给了普通指向对象p。随后，普通指针对象p可以通过间接操作即可随意修改常量对象a的值。

因此，在C语言中不允许直接将int**隐式转换为const int**，而只能将int**隐式转换为const int*const*类型。如果代码清单12-4中的指针对象pp的类型是const int*const*，那么当对pp做第一次间接操作时，（*pp）的类型为int const*const，其值不允许被修改，从而保证了无法通过（*pp）将原先初始化时指向的指针对象间接地将它修改为指向某个常量对象。

以上已经基本把const限定符如何修饰一个对象以及修饰后所产生的效果都详细描述了。各位在看完这些内容后务必要反复实践，这样才能加深对限定符的了解，这块内容也确实不容易掌握。

12.1.4 const限定符修饰函数形参类型为数组的对象

下面再谈一下const限定符如何修饰函数形参为数组类型的场合。我们在9.3节中已经谈过，一个函数的形参可以被表达为一个数组类型，但它本质上仍然是一个指针，既然是一个指针，那么它跟原生的数组对象就会有所不同。前面讲了，原生的数组对象本身是不可被修改的，因此没有所谓的用限定符修饰数组对象的这个概念，然而对于指针则不同，指针对象的值是可被修改的。如果我们要对以数组类型呈现的函数形参对象施加const

限定，使得该形参值无法被修改，那么我们只需要将const限定符放置在[]下标操作符里面即可。如果[]中含有数值字面量或其他标识符，那么const放在它们的前面，即左侧位置。代码清单12-5展示了const修饰函数形参为数组类型的例子。

代码清单12-5　const修饰函数形参为数组类型的例子

```
#include <stdio.h>

// 这里，形参a相当于int * const类型
// 形参b相当于const int *类型
// 形参c相当于int const * const类型
static void Fun(int a[const 5], const int b[3],
                const int c[static const 4])
{
    a[0]++;      // OK！没有问题
    a = NULL;    // 错误！a是一个常量指针对象

    b[0]++;      // 错误！b[0]是const int类型，一个常量

    c[0]++;      // 错误！c[0]是const int类型，是一个常量
    c = NULL;    // 错误！c本身是一个常量，不能被修改

    printf("The sum is: %d\n", a[0] + b[1] + c[2]);

    b = NULL;    // OK！没有问题
}

int main(int argc, const char* argv[])
{
    int a[] = { 1, 2, 3, 4, 5 };
    int b[] = { 7, 8, 9 };
    int c[] = { 10, 11, 12, 13 };

    Fun(a, b, c);
}
```

代码清单12-5中给出了3种不同类型的形参，我们看每个形参的本质类型时也非常简单，直接将[]去掉，把里面的static等存储类说明符也全都去掉，只留限定符，然后把标识符变为*号即可。另外，如果我们对使用数组下标操作符的对象类型是否为一个常量看不清，可以把数组下标形式变为间接操作形式，比如a[0]++；可以转换为（*（a+0））++；，语义是相同

的，这样我们就能看到a[0]的类型其实与（*a）一样，都是int类型，而这里const修饰的是a自身。

12.1.5　类型限定符的本质含义

最后，我们来描述一下类型限定符的本质含义。所谓类型限定符很显然，它主要是限定类型的，尽管在文法上它也可以看作修饰一个对象，而像我们上面那种const限定的判定主要是以对象作为参照的，这里我们将再详细介绍如何根据类型进行判定。类型限定符所限定的是类型说明符（比如基本类型，枚举、结构体、联合体等用户自定义类型，还有稍后会讲的原子类型说明符），以及与类型说明符相结合的指针类型。我们假定有某个含有N级指针的类型Type（Type自身或许也包含着const限定符），那么Type const或者const Type都表示类型Type受到const限定。然而在C语言中，我们只能通过typedef将整个类型组合为一个类型标识符Type，这一点会在下一章描述。而像const int*这个类型，int*没有被视为一个整体，这里的const限定的仅仅是int类型，而不是整个int*类型。所以，在C语言中使用限定符后置法，将int*const这种写法看作为const限定int*类型。因此我们碰到除_Atomic之外的类型限定符，都看整个类型声明的最右边的限定符，最右边的限定符限定了在它左边的所有类型。如果对当前对象做了N次间接操作，那么我们就从右往左跳过N个*号，看类型限定符限定的情况。

像代码清单12-5中的Fun函数形参对象a，已知其类型为int*const，那么

对于a自身来说，其类型最右边有一个const，那很显然，它就是一个常量；而对于（*a）或a[0]而言，我们用了一次间接操作符，那么我们从右往左看跳过一个*号，前面有没有const，说明（*a）的类型就是int，不是一个常量。同样，我们再分析一下形参对象c，它的类型可以通过上述方式解析出来，是一个int const*const类型。那么对于c而言，它前面就有一个const，所以c就是一个常量；而对于（*c）或c[0]而言，做了一次间接操作之后看第一个*前面是否跟着一个const，我们看到确实有一个const，所以（*c）也是一个常量，其类型为const int。

我们可以再引申，观察代码清单12-3中的cpp和cqq。cpp指针对象的类型为int const**，那么对于cpp而言，类型最右边是一个*号而没有碰到const，所以cpp本身不是一个常量；而（*cpp）做了一次间接操作之后，跳过最右边的*号，其类型为int const*。很显然，这个类型的最右边也是一个*号而不存在const，所以（*cpp）也不是一个常量；最后再看（**cpp），做了第二次间接引用之后，其类型为int const，这里很明显就有一个const，所以（**cpp）是一个常量。而cqq也同样分析，cqq的类型为int*const*const，它前面就有一个const，所以cqq自身是一个常量；而做了第一次间接操作之后，（*cqq）的类型为int*const，很显然最右边是一个const，所以（*cqq）也是一个常量；最后看做第二次间接操作之后，（**cqq）的类型，它是int，没有const修饰，所以（**cqq）不是一个常量。

由于像const以及volatile类型限定符在C90标准中就已经引入了，所以

对于类型的限定看上去有些别扭。但这种表达方式也是相当完备的，能适用于任何类型组合，这也是C语言从标准化开始起就注定要贯彻设计为一门简洁、灵活且强大的编程语言这一目标。

12.2 volatile限定符

volatile在英语中的意思是“不稳定的”，“易变的”，“易挥发的”。C语言用volatile限定符修饰一个对象时，指明该对象的值可能会被异步修改，这暗示了编译器不要对该对象做寄存器暂存优化，在读写它的时候总需要显式地从它的存储器地址中获取值。一般而言，C语言实现会将一个函数中多次出现的同一对象的值尽可能地存放在寄存器中，如果该对象的内容可以存放在寄存器中（不超过一个寄存器所能容纳的字节数），且寄存器数量足够。毕竟，寄存器访问比读写内存要快得多。

volatile一般用于多个线程所共享的资源，包括用于数据同步的锁对象。另外，嵌入式系统也会将MMR（存储器映射的寄存器）地址类型定义为指向volatile的指针类型。volatile修饰对象时所摆放的位置与它所起的修饰效果同const一样，这里不再赘述。此外，volatile可以与const一同使用，尽管这么做的场合不多，不过对于MMR的访问来说倒也不错——比如：int data=*（const volatile int*）0xff800000UL；表示从0xff800000地址所映射的外设寄存器中获取int类型的相关数据。这里使用volatile限定符表示在每次出现*（const volatile int*）0xff800000UL时，都要显式地读取该地址中的内容，而不是在第一次读取之后就默认将该值存放在CPU的寄存器中，然后后续的读取都直接从该寄存器中获取数据。

下面，我们将通过代码清单12-6来描述volatile限定符的使用以及效

果。

代码清单12-6　volatile的使用及效果

```
#include <stdio.h>
#include <pthread.h>

// 这里定义了一个Fun函数，其形参a的类型为：int * const volatile
static void Fun(int a[static volatile const 2])
{
    printf("a[0] = %d, a[1] = %d\n", a[0], a[1]);
}

// 这里先声明一个普通的int类型静态对象
static int normalInt;

// 这里声明了一个volatile的int类型静态对象
static volatile int volatileInt;

// 这个函数用于用户线程执行例程
static void* ThreadProc(void *param)
{
    // 这里的考察很简单，做一个1000000次循环，
    // 每次分别将normalInt与volatileInt递增，
    // 最后看主线程中这两个值的变化
    for(int i = 0; i < 1000000; i++)
    {
        normalInt++;
        volatileInt++;
    }

    return NULL;
}

int main(int argc, const char* argv[])
{
    // 在主线程中，一开始将normalInt与volatileInt初始化为0
    normalInt = 0;
    volatileInt = 0;

    pthread_t threadID;
    // 我们用pthread API创建一个用户线程，
    // 该线程中normalInt与volatileInt在不断变化
    pthread_create(&threadID, NULL, &ThreadProc, NULL);

    // 我们这里就循环10000次，明显少于用户线程的1000000次，
    // 这样，用户线程在对这两个考察对象的最终修改不会做综合（synthesize）
    while(volatileInt < 10000);

    // 我们在Release模式下能观察到，normalInt对象的值始终为0；
    // 而volatileInt则得到了当前修改后的值
    printf("normalInt = %d\n", normalInt);
    printf("volatileInt = %d\n", volatileInt);

    // 这里声明一个指针对象p，(*p)的类型为volatile int
    volatile int *p = &volatileInt;

    // 这里声明了一个指针对象q，它自身是volatile的。
    // (*q)的类型则是int，不是volatile的
    int * volatile q = &normalInt;

    Fun((int[]){ *p, *q });
```

```
}
```

各位在运行代码清单12-6中的代码示例时必须注意，一定要将当前编译环境设置为Release模式（一般开发环境默认的设置是Debug模式），只有这样，编译器才会对代码做出优化，我们才能看到效果。而如果用命令行编译的话，我们可以直接用-O2命令选项，并且不添加-g命令选项即可。另外，在Linux环境下，我们需要使用--pthread连接命令选项来连接pthread的安全实现运行时库。而在macOS环境下默认已经把底层的库都连接好了，无需手工设置。

12.3 restrict限定符

restrict限定符是从C99标准开始引入的。它的用法与之前的const和volatile有所不同，它只能用于修饰一个指针类型的对象，而不能用于修饰一个普通对象。通过受restrict限定的一个指针所访问的一个对象与该指针具有一种特殊的关联性。这种关联性要求，对该对象的所有访问都要直接或间接使用那个特定指针的值，而不受其他指针的干涉。使用restrict限定符可暗示编译器对通过指针访问的数据进行优化，比如说我们可以直接将受restrict限定的指针所读取到的值存放在寄存器中，后续再次出现对该指针的访问时可直接拿寄存器中的数据，而不需要做真正的访存操作。

下面我们通过代码清单12-7举一个简单的示例，这样大家就能有一定的感性认识了。

代码清单12-7　初窥restrict限定符

```
#include <stdio.h>
#include <stdint.h>

int main(int argc, const char* argv[])
{
    // 这里声明一个数组，它含有8个uint8_t类型的元素
    uint8_t bytes[] = { 0x10, 0x20, 0x30, 0x40,
        0x50, 0x60, 0x70, 0x80 };

    // 声明一个指向int32_t类型的指针对象p，它直接指向bytes数组的首地址
    int32_t *p = (int32_t*)bytes;
    // *p的初始值为0x40302010
    printf("First, *p = 0x%.8X\n", *p);

    // 声明一个指向int32_t类型的指针对象q，它直接指向bytes[2]位置元素的地址
    int32_t *q = (int32_t*)&bytes[2];
    // *q的初始值为0x60504030
    printf("First, *q = 0x%.8X\n", *q);

    *q += 16;
```

```
    // 输出: *p = 0x40402010
    printf("Now, *p = 0x%.8X\n", *p);
}
```

各位注意，代码清单12-7中的示例代码必须在x86处理器或ARMv7或更高版本的处理器中执行。各位可以看到，代码清单12-7中，指针对象p与指针对象q之间是有叠加部分的，即bytes[2]与bytes[3]部分，这两个元素所在的地址分别作为p所指对象的高2字节以及q所指对象的低2字节。这意味着无论是指针对象p还是指针对象q，它们都不能作为一个指向独立对象的指针，也就是说，（*p）与（*q）所表示的对象不是相互独立的，而是有叠交（aliasing）的。在这种情况下，指针p与指针q都不能用restrict限定符去修饰。

很显然，如果我们这里允许对（*p）与（*q）的访问做寄存器暂存优化，那么当对（*p）做修改时，（*q）的低2字节也被改变，而寄存器中的内容却得不到更新；同样，如果对（*q）做修改，那么（*p）的高2字节的值也被修改，而其寄存器中的内容也得不到更新，这会导致不可预期的结果。这也是为何无法对存储空间存在叠交的两个指针对象做restrict限定的原因。

对于一个受restrict限定符限定的指针对象，它所指向的存储空间应该在当前执行环境下是唯一的，没有其他指针与它指向同一个存储空间，并且也不存在任何与其他指针所指向的存储空间有重叠的情况。代码清单12-8进一步描述了restrict限定符的使用。

代码清单12-8　restrict限定符的进一步使用

```c
#include <stdio.h>
#include <stdint.h>

// 这里参数a的类型为：int * const volatile restrict
static void Fun(int a[static const volatile restrict 2])
{
    printf("The result is: %d\n", a[0] + a[1]);
}

/**
 * 下面我们自制一条利用restrict限定符的存储数据拷贝函数
 * @param pDst 指向目的存储空间
 * @param pSrc 指向源存储空间
 * @param count 指定要拷贝多少个int32_t的数据元素
*/
void MyfastMemCpy(int32_t* restrict pDst, const int32_t* restrict pSrc,
                  size_t count)
{
    // 如果元素个数为偶数，我们直接2个元素2个元素进行拷贝，以提升效率
    if((count & 1) == 0)
    {
        const size_t nLoop = count / 2;
        uint64_t* restrict p = (uint64_t*)pDst;
        const uint64_t* restrict q = (const uint64_t*)pSrc;

        for(size_t i = 0; i < nLoop; i++)
            p[i] = q[i];
    }
    else
    {
        for(size_t i = 0; i < count; i++)
            pDst[i] = pSrc[i];
    }
}

int main(int argc, const char* argv[])
{
    // 声明了两个int类型对象a和b
    int a = 10, b = 20;

    // 声明了一个指向int类型的受restrict限定的指针对象p，
    // 用对象a的地址对它初始化
    int* restrict p = &a;

    // 声明了一个指向int类型的受restrict限定的指针对象q，
    // 用对象b的地址对它初始化
    int* restrict q = &b;

    // 以下这条语句是非法的！两个restrict限定的指针不能指向同一个存储空间。
    // 尽管编译器对此不会有任何警告，但可能会引发未定义的结果
    p = q;

    Fun((int[]){*p, *q});

    // 以下这条语句是非法的，restrict不能用于修饰非指针类型
    restrict int x = 0;

    // 以下这条语句是非法的，(*t)类型为int，不是指针类型
    restrict int *t = NULL;

    // 下面定义了两个数组，均含有1024个元素
    int32_t dst[1024] = { 0 };
    int32_t src[1024];

    // 对src数组元素做初始化
    for(int i = 0; i < 1024; i++)
        src[i] = i;
```

```
    // 调用我们自制的高效存储器拷贝函数
    MyfastMemCpy(dst, src, 1024);

    // 最后我们验证一下结果
    for(int i = 0; i < 1024; i++)
    {
        if(src[i] != dst[i])
        {
            puts("Result not equal!");
            return -1;
        }
    }

    puts("Result equal!");
}
```

从代码清单12-8中可知，对restrict限定的指针的使用有其随意性，我们在写代码的时候如果无意间破坏了restrict的要求，编译器也无法识别，因为要实现这种识别对编译器来说开销会比较大。所以restrict关键字一般用于函数形参，提示函数使用者所传入的对象地址要确保其唯一性。我们在满足restrict要求的时候，我们可以提供更高效的运行时库，比如像代码清单12-8中的MyFastMemCpy函数。当我们确保pDst与pSrc这两个指针所指向的存储空间相互独立且不叠交时，我们可以采取各种优化措施，比如可以多个元素多个元素一起拷贝，只要CPU有这种能力。但是，倘若我们传入的指针所指向的存储空间不能满足唯一性，或者说当中有叠交，那就别说多个元素一起拷贝了，即便连指针所指向对象类型的粒度（这里是int32_t类型）进行拷贝都无法确保数据正确性（比如代码清单12-7所呈现的情况），只能一个字节一个字节进行拷贝。

12.4 _Atomic限定符

_Atomic限定符是在最新的C11标准中所引入的。所以它限定类型的方式比起const、volatile以及restrict有所不同，它直接用_Atomic（类型名）这种方式作为原子类型说明符。为何_Atomic能使用这种形式作为类型限定符呢？因为_Atomic一般修饰的是非指针对象类型，所以不牵涉限定指向数组的指针以及指向函数的指针这些比较特殊的类型表达形式。

如果将一个对象声明为原子类型，那么说明该对象是原子的，这也称为“原子对象”。原子对象的访问与非原子的有所不同，对一个原子对象的读和写都是不可被打断的，此外有很多针对原子对象的修改操作（比如加减算术计算以及各种逻辑计算等原子操作），这些原子操作也是不可被打断的。一个操作不可被打断意味着在执行整个操作过程中，即便有一个硬件中断信号过来，该中断信号也不能立即触发处理器的中断执行例程，处理器必须执行完整条原子操作之后才可进入中断执行例程。对于中断控制器而言往往会有“未决”（Pending）这个状态，说明当前中断尚未被处理。我们在使用原子操作的时候不用担心当前的执行线程会被切换，因为中断处理都不会发生。原子对象往往用于多核多线程并行计算中对多个线程共享变量的计算。

原子操作的另一大特点是，对于来自多个处理器核心对同一个存储空间的访问，存储器控制器会去仲裁当前哪个原子操作先进行访存操作，哪

个后进行，这些访存操作都会被串行化，所以这对于多核多线程并行计算的数据同步而言是必需的处理器特征。我们无法通过简单的开关中断去控制各个核心同时执行不同线程的行为与状态，所以在多核心多线程并行计算的环境下，原子操作是唯一的数据同步手段。另外在此环境下，像互斥体（mutex）、信号量（semaphore）等同步原语的实现也都基于原子操作。

我们在C11标准下的C语言中使用原子操作时，应当包含<stdatomic.h>标准库头文件，该头文件中已经预定义了一些当前主流处理器所能支持的原子对象类型，此外还有相应的原子操作函数。我们在实际使用原子类型时应当避免直接使用_Atomic（类型名）这种形式，而是直接用<stdatomic.h>头文件中已经定义好的原子类型。当前C11标准中所罗列的能够支持原子对象类型的基本类型均为整数类型，也就是说除整数类型外的其他类型都无法作为原子对象类型（包括浮点类型）。我们常用的原子对象类型有：atomic_bool、atomic_char、atomic_schar、atomic_uchar、atomic_ushort、atomic_short、atomic_int、atomic_uint、atomic_long、atomic_ulong、atomic_char16_t、atomic_char32_t、atomic_wchar_t、atomic_intptr_t、atomic_uintptr_t、atomic_size_t、atomic_ptrdiff_t等。像这里面atomic_int类型就被定义为_Atomic（int），而像atomic_size_t类型就被定义为_Atomic（size_t）。

此外，原子对象的初始化与普通对象也有所不同，在<stdatomic.h>头文件中定义了两个接口，分别用于对全局原子对象与函数内局部原子对象

进行初始化。另外，对原子对象的读写也不应该直接用=赋值操作符，而是需要通过使用atomic_load函数进行读，atomic_store函数进行写。代码清单12-9将先简单介绍原子对象的一些基本操作。

代码清单12-9　原子对象的初步使用

```
#include <stdio.h>
#include <stdatomic.h>

// 这里声明了一个int类型的静态原子对象sIntAtom
// 我们通过ATOMIC_VAR_INIT宏函数对其初始化为100
static atomic_int sIntAtom = ATOMIC_VAR_INIT(100);

int main(int argc, const char* argv[])
{
    // 这里在main函数中声明了局部原子对象a
    atomic_int a;

    // 我们通过atomic_init函数对原子对象a进行初始化为10
    atomic_init(&a, 10);

    // 我们通过atomic_store函数将原子对象a修改为20
    atomic_store(&a, 20);

    // 我们通过atomic_load函数将原子对象a的值加载到普通对象b中
    int b = atomic_load(&a);

    // 我们利用atomic_fetch_add函数，对原子对象sIntAtom与普通对象b做原子加法操作。
    // 此时返回的结果是做原子加法操作之前的sIntAtom的值
    int oldValue = atomic_fetch_add(&sIntAtom, b);

    printf("oldValue = %d\n", oldValue);

    // 我们将原子加法操作之后的sIntAtom原子对象的值，加载到对象b中
    b = atomic_load(&sIntAtom);

    printf("sIntAtom = %d\n", b);
}
```

代码清单12-9清晰而又精简地介绍了对原子对象的各类操作。这里给大家呈现的是对原子对象的初始化、加载、存储以及原子加法操作。除了原子加法操作之外，C11标准还定义了以下原子算术逻辑操作：atomic_fetch_sub（原子减法操作）、atomic_fetch_or（原子按位或操作）、atomic_fetch_xor（原子按位异或操作）、atomic_fetch_and（原子按

位与操作）。这里要注意的是，这些算术逻辑原子操作都不能用于atomic_bool类型，即布尔原子类型。另外这里需要注意的是，对原子对象的初始化函数本身并非原子的，也就是说，atomic_init函数是可被打断的。因此我们在对原子对象做初始化时应当统一在一个线程中完成（通常是主线程），然后再做线程分派调度。另外，我们不应该使用atomic_store原子存储操作对原子对象进行初始化，对原子对象的初始化操作只有ATOMIC_VAR_INIT与atomic_init这两个接口。同时，其他原子操作必须作用于已初始化的原子对象，否则结果可能是未知的。

代码清单12-10将给大家带来一个比较实用的例子来描述原子对象以及原子操作的使用与效果。在这个例子中，我们将定义一个10000×100的一个int类型的二维数组，并对它的所有元素进行求和操作。我们将使用双核双线程并行计算来达成这个目的。

代码清单12-10　双核双线程对二维数组求和

```
#include <stdio.h>
#include <stdatomic.h>
#include <stdbool.h>
#include <stdint.h>
#include <pthread.h>

// 声明一个静态unsigned long long类型的原子对象，初始化为0，
// 用于存放原子计算操作的求和计算结果
static volatile atomic_ullong sAtomResult = ATOMIC_VAR_INIT(0);

// 声明一个静态的int类型的原子对象，初始化为0，
// 用于存放原子计算操作的当前计算数组的行索引
static volatile atomic_int sAtomIndex = ATOMIC_VAR_INIT(0);

// 声明一个静态普通的uint64_t类型对象，并将它初始化为0，
// 用于存放普通计算操作的求和计算结果
static volatile uint64_t sNormalResult = 0;

// 声明一个静态普通的int类型对象，并将它初始化为0，
// 用于存放普通计算操作中当前计算数组的行索引
static volatile int sNormalIndex = 0;

// 由于这个标志在用户线程中只写，且在主线程中只读，
```

```
// 因此在这两者线程中并不会产生数据竞争，所以无需使用原子对象
static volatile bool sIsThreadComplete = false;

// 声明即将用于计算的二维数组
static int sArray[10000][100];

// 定义普通计算操作的线程例程
static void* NormalSumProc(void *param)
{
    // 这里使用一个currIndex对象，使得sNormalIndex在每次迭代中仅被读取一次，
    // 减少外部修改的干扰
    int currIndex;

    // 在每次迭代时，先读取当前行索引的值，然后立即对它做递增操作
    while((currIndex = sNormalIndex++) < 10000)
    {
        // 得到当前行索引之后，对当前行的数组做求和计算
        uint64_t sum = 0;
        for(int i = 0; i < 100; i++)
            sum += sArray[currIndex][i];

        sNormalResult += sum;
    }

    // 用户线程计算结束，将sIsThreadComplete标志置为true
    sIsThreadComplete = true;

    return NULL;
}

// 定义原子操作计算的线程例程
static void* AtomSumProc(void *param)
{
    int currIndex;

    while((currIndex = atomic_fetch_add(&sAtomIndex, 1))
            < 10000)
    {
        uint64_t sum = 0;
        for(int i = 0; i < 100; i++)
            sum += sArray[currIndex][i];

        atomic_fetch_add(&sAtomResult, sum);
    }

    sIsThreadComplete = true;

    return NULL;
}

int main(int argc, const char* argv[])
{
    // 我们先对sArray数组进行初始化
    for(int i = 0; i < 10000; i++)
    {
        for(int j = 0; j < 100; j++)
            sArray[i][j] = 100 * i + j;
    }

    // 我们先在主线程中计算出标准正确的计算结果
    uint64_t standardResult = 0;

    for(int i = 0; i < 10000; i++)
    {
        for(int j = 0; j < 100; j++)
            standardResult += sArray[i][j];
    }

    printf("The standard result is: %llu\n", standardResult);
```

```
    // 下面我们先观察不用原子对象与原子操作的计算
    pthread_t threadID;

    pthread_create(&threadID, NULL, &NormalSumProc, NULL);

    // 在主线程中也做类似的计算处理
    int currIndex;

    // 使用原子加法操作对当前原子数组行索引做后缀递增操作
    while((currIndex = sNormalIndex++) < 10000)
    {
        uint64_t sum = 0;
        for(int i = 0; i < 100; i++)
            sum += sArray[currIndex][i];

        sNormalResult += sum;
    }

    // 等待用户线程完成
    while(!sIsThreadComplete);

    if(sNormalResult == standardResult)
        puts("Normal compute compared equal!");
    else
    {
        printf("Normal compute compared not equal: %llu\n",
               sNormalResult);
    }

    // 我们最后对原子操作的线程做并行计算
    sIsThreadComplete = false;

    pthread_create(&threadID, NULL, &AtomSumProc, NULL);

    while((currIndex = atomic_fetch_add(&sAtomIndex, 1))
          < 10000)
    {
        uint64_t sum = 0;
        for(int i = 0; i < 100; i++)
            sum += sArray[currIndex][i];

        atomic_fetch_add(&sAtomResult, sum);
    }

    // 等待用户线程完成
    while(!sIsThreadComplete);

    if(atomic_load(&sAtomResult) == standardResult)
        puts("Atom compute compared equal!");
    else
        puts("Atom compute compared not equal!");
}
```

代码清单12-10不仅有原子操作的求和计算，而且还有不采用原子操作的求和计算。我们可以实际操作一下，能观察到若采用普通求和计算往往无法得到正确的计算结果，且计算结果的值每次执行还都不一样。这就是因为像++操作、+操作的非原子性造成的。像这类修改操作其实有三个步

骤：读取数据、修改数据、存储数据。比如像a++；这个操作，如果用处理器指令来表示的话至少需要三条指令——load reg，[a]（将对象a的值加载到reg寄存器中）；inc reg（对reg寄存器做递增操作）；store reg，[a]（将reg寄存器的值再写回对象a中）。对于原子加法操作而言，它们将被组合成一单条指令，并且整个操作过程不能被打断。而对于普通操作，这三条指令每条执行完之后都能被打断，这就使得一个线程对寄存器做了修改之后，但在写回之前被其他线程先写回了，然后等该线程再写回就把先前线程修改的内容给覆盖了，从而造成了数据的不一致性。关于这个时序问题我们可以通过图12-1来清晰看到。

图12-1中上半部分是非原子的修改操作，下半部分是原子的修改操作。我们可以清晰地观察到对于非原子的读- 修改- 写操作之间存在着间隙，这些间隙都会被CPU利用，一旦有中断信号过来就会被打断，或者被其他处理器核心的相关操作给覆盖。像图12-1中，线程A与线程B几乎同时对一个共享存储单元读取值，然而线程A操作比较快，先写回数据，而线程B操作稍慢后写回，但是等到线程B写回数据的时候就直接把线程A已修改好的数据给完全覆盖了！换句话说，线程B并没有基于线程A先修改好的数据做相应操作。而原子操作则不一样，它们是作为一个整体的操作，如果同时有两个原子操作对同一共享存储单元进行操作，那么存储器控制器会做仲裁哪个操作优先、哪个操作断后，并且稍后执行的原子操作必定基于之前修改完的结果进行。因为一个原子操作在被允许操作之前，连读取操作都不会执行，而当存储器控制器允许某个原子操作执行时，那么读取- 修改- 写回这三个操作才会捆绑着执行。

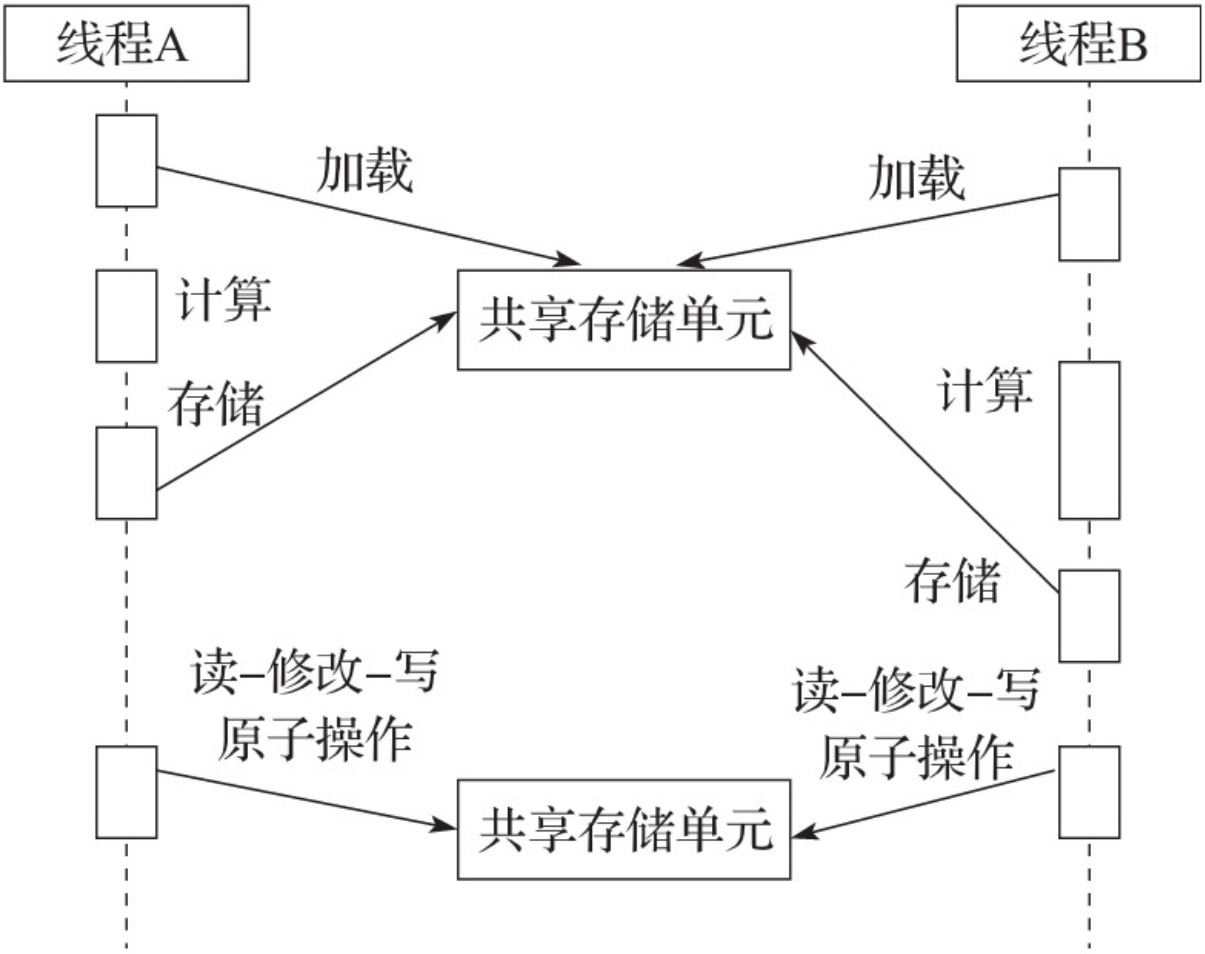


图12-1　原子操作与非原子操作的顺序图

注意：当前Visual Studio Community 2017中的VS-Clang对原子操作的编译器后端还没支持好，所以各位如果要在Windows系统上测试代码清单12-9与代码清单12-10中的内容的话，请使用基于GCC的Mingw或纯Clang编译器。或参考GitHub上的代码：https://github.com/zenny-chen/simple-stdatomic-for-VS-Clang。这里提供了基于VS-Clang环境中内建函数对部分原子操作的实现。各位将此GitHub中的stdatomic.h以及stdatomic.c放到自己的工程项目中，然后用＃include.“stdatomic.h”进行包

含。

原子操作其实属于一个比较大的问题领域，这里仅仅揭露了其冰山一角。C11标准对于原子库还有相关的存储器次序这个概念，此外还有lock-free（无锁）同步算法所需的原子操作，这些更高级的话题我们将放到本书姊妹篇标准库卷做详细描述。

12.5　本章小结

本章内容属于C语言中比较高端的话题，也是比较晦涩难懂的部分。如果大家对本章所讲述的类型限定符掌握到驾驭自如的境界的话，那么离C语言大师也就不远了。对于本章，笔者认为对于大部分初学者来说光看一遍还远远不够，大家需要不断实践，然后再去巩固阅读，相信每次阅读都会有新的收获。

最后，在12.4节中提到_Atomic所使用的_Atomic（类型名）这种表达方式，如果我们在使用const、volatile以及restrict时，在不涉及指向数组的指针与指向函数的指针这些表达方式的情况下也可以这么玩，代码清单12-11将给大家呈现这种奇妙的表达方式。

代码清单12-11　有趣的类型限定符表达方式

```
#include <stdio.h>

// 我们自己定义一个类似于_Atomic用法的宏CONST
#define CONST(type)     type  const

int main(int argc, const char* argv[])
{
    // 声明一个int类型的常量对象a，并初始化为10
    CONST(int) a = 10;

    int b = 0;

    // 声明一个普通指针对象，指向a的地址
    // (*p)的类型为const int
    CONST(int) *p = &a;

    // 声明一个常量指针对象q，指向对象b的地址
    // q的类型为int * const
    CONST(int*) q = &b;
    *q += 20;
    // 声明一个指针对象pp，指向指针p的地址
    // 其类型为const int * const *
    CONST(CONST(int)*) *pp = &p;
    printf("The value is: %d\n", **pp + *q);
```

```
}
```

对于代码清单12-11我们可以看到，只有在CONST圆括号里包围的类型才是常量类型。像指针对象p，CONST所包围的是int类型，所以指针p本身不是常量，而（*p）才是。而指针对象q则相反，CONST包围的是int*，所以指针对象q本身是常量，但（*q）则不是。而对于比较复杂的pp对象，最外围的CONST包围的是CONST（int）*类型，所以pp自身不是一个常量；但（*pp）就是了，它的类型就是CONST（CONST（int）*）；而（**pp）的类型则是CONST（int），明显也是个常量。

所以，只要把基本的概念掌握之后，我们可以自己抽象出一套解析方法。无论怎么变，万变不离其宗。

第13章　C语言的类型系统

类型系统（type system）在许多强类型编程语言中扮演了非常重要的角色，比如C、C++、Java、Objective-C、Swift，等等。这些编程语言中，不同类型的对象本身所具有的含义、能表示的值的范围等都可能有所不同。我们在前几章中已经详细讲述了C语言的基本类型、用户自定义类型、指针、数组等，此外还有类型限定符。这些类型、类别与限定符的相互组合能构成多种不同的完整类型，使得C语言的类型具有丰富多样性，同时也具有灵活性与统一性。C语言的类型系统是相当完备的，这意味着我们通过C语言现有的语法特征，无论怎么组合，都能构造出一个合法的类型来。

本章将给大家总结并更深入地介绍C语言的类型系统，让大家深刻体会到其中的博大精深。然后给大家介绍一下C11标准中对类型的各种分类方法。最后一小节将引入C语言的另一个非常有用的语法特征——typedef，可以将任一类型抽象为一个类型标识符。

13.1　对象类型与函数类型

C语言的对象类型由三大部分组成——类型说明符（type specifier）、类型限定符（type qualifier）、对象类别（type category）。类型说明符包括了像void、char、short、int等基本类型，还有结构体、联合体说明符，枚举说明符，原子类型说明符以及typedef名。类型限定符包括const、volatile以及restrict（这里已经把_Atomic限定的类型归类为原子类型说明符）。对象类别主要就是数组与指针类别。

而函数类型由函数的返回类型与形参列表组成。其中，返回类型与形参列表中每个形参的类型都是对象类型。如果形参列表为空，那么形参列表可用void声明。

我们在判定一个对象的类型时，首先应该先判定当前对象的类别，即它是一个普通对象还是一个数组对象或是指针对象。对象类别是互斥的，也就是说，如果一个对象是普通对象，那么它就不可能再是指针对象或数组对象。因此，一个对象的类别只能是普通、指针与数组中的一个。知道了类别之后，我们就可以再判定该对象的完整类型，包括如何被限定符修饰等等。代码清单13-1给出了一些对象类型与函数类型的例子。

代码清单13-1　对象类型与函数类型

```
#include <stdio.h>

// 这里声明了一个静态全局数组对象sArray，其类型为int [3]
// 它具有3个int类型的元素
```

```
static int sArray[3] = { 1, 2, 3 };

// 定义了Func1函数，其类型为：int* (int)，
// 表示函数返回类型为int*，形参列表中含有一个参数，其类型为int
static int* Func1(int a)
{
    printf("Func1 a = %d\n", a);
    return sArray;
}

// 定义了Func2函数，其类型为：int (* (int, float) )[3]，
// 表示该函数返回类型为int (*) [3]，形参列表具有两个参数，第一个形参a具有int类型，
// 第二个形参f具有float类型
static int (*Func2(int a, float f))[3]
{
    printf("Func2 sum = %f\n", a + f);
    return &sArray;
}

int main(int argc, const char* argv[])
{
    // 这里声明了一个指针对象p，其类型为const int *
    // 并用Func1函数调用的返回值为它初始化
    const int *p = Func1(3);
    printf("*p = %d\n", *p);

    // 这里声明了一个指针对象pFunc，它指向函数Func2
    int (*(*pFunc)(int, float))[3] = &Func2;
    pFunc(3, 10.5f)[0][1]++;

    // 声明了int类型的普通对象elem
    int elem = sArray[1];
    printf("elem = %d\n", elem);
}
```

代码清单13-1中，静态全局对象sArray的类别是一个数组，main函数中声明的对象p与pFunc的类别都是指针，而对象elem则是一个普通对象。这里也清楚地描述了函数的返回类型以及形参类型，它们都是用对象类型来表示的。

代码清单13-1中的Func2函数的类型以及main函数中声明的pFunc函数指针对象类型比较复杂，我们将在13.3节中详细分析这种复杂的类型声明方式以及对其类型的剖析。另外，本节大致介绍了对象与函数类型的构成，这将作为下面几节的基础。

13.2 对声明符的进一步说明

为了能让各位更深刻地理解一个对象与函数的类型，我们这里将引入比较完整的C11标准所给出的声明文法。

declaration：

declaration-specifier　init-declarator-list$_{opt}$；

static_assert-declaration

declaration-specifier：

storage-class-specifier　declaration-specifier$_{opt}$

type-specifier　declaration-specifier$_{opt}$

type-qualifier　declaration-specifier$_{opt}$

function-specifier　declaration-specifier$_{opt}$

alignment-specifier　declaration-specifier$_{opt}$

init-declarator-list：

init-declarator

init-declarator-list，init-declarator

init-declarator：

declarator

declarator=initializer

我们从上述关于声明的文法中可以看到，C语言中声明包含了两大部分：一个是声明说明符（declaration-specifier），另一个是声明符（declarator）。我们在看声明符之前可以尝试代码清单13-2中这些奇怪但又符合文法的写法。

代码清单13-2　无声明符的声明

```
int main(int argc, const char* argv[])
{
    static;
    extern const int;
    _Alignas(8);
}
```

代码清单13-2中列出的三个声明都是合法有效的声明，尽管在编译时编译器会报出“此声明没有声明任何东西”之类的警告。因为在声明的文法中也明确指出，在声明说明符中后面的初始化声明符列表可省，当然最后的分号是必须有的。下面我们再来看声明符的文法。

declarator：

$\text{pointer}_{\text{opt}}$　direct-declarator

direct-declarator：

identifier

（declarator）

direct-declarator [type-qualifier-list$_{opt}$ assignment-expressionopt]

direct-declarator [statictype-qualifier-list$_{opt}$ assignment-expression]

direct-declarator [type-qualifier-list$_{opt}$ static assignment-expression]

direct-declarator [type-qualifier-list$_{opt}$ *]

direct-declarator （parameter-type-list）

direct-declarator （identifier-list$_{opt}$）

pointer：

*type-qualifier-list$_{opt}$

*type-qualifier-list$_{opt}$ pointer

type-qualifier-list：

type-qualifier

type-qualifier-list type-qualifier

我们这里可以看到，指针与数组类别都是在声明符中体现出来的。此外，无论在上述的声明说明符中还是在这里的直接说明符（direct-declarator）中，我们都看到了类型限定符的身影。因此，我们也可以将类型限定符看作既可以修饰类型，也可以修饰类别。此外，我们在数组[]里也可以看到其中能够包含一条赋值表达式。这个在之前描述数组的时候并未涉及太多，由于确实用得很少，并且这也是自C99标准中引入了可变长度数组之后才引入的语法特性。代码清单13-3先对此语法特性进行描述。

代码清单13-3　数组声明的下标中含有赋值表达式的情况

```
#include <stdio.h>

// 这里定义了静态函数Func1，带有两个形参，形参a是int类型；
// 形参s则是一个int * const类型，这里用一个变长数组来表示
static void Func1(int a, int s[static const a += 5])
{
    // 注意，这里形参s中动用了a += 5，
    // 这意味着当函数Func1被调用时，形参a的值会自动加5
    printf("Func1 a = %d\n", a);
}

int main(int argc, const char* argv[])
{
    int a = 1;

    Func1(10, &a);

    // 这里声明了一个变长数组对象s，
    // 在数组下标里使用了一个赋值表达式，使得局部对象a的值加10
    int s[a += 10];

    // s数组具有11个int元素
    printf("The size of s = %zu\n", sizeof(s) / sizeof(s[0]));

    // a的值为11
    printf("a = %d\n", a);
}
```

代码清单13-3描述了声明数组对象时，在数组下标里使用赋值表达式的情况这得益于C语言中二赋值操作符与十二等复合赋值操作符直接返回它们左操作数的值的语法特性。当然，对于这种写法不建议滥用，因为这

并不会让代码看上去有多简洁，反而容易让人看得眩晕。这里引出这个代码示例主要是针对C11标准中给出声明符的文法而言的，大家看到了文法定义，自然就可写出很多有趣的（但可能并不实用的）表达式和语句。

下面我们将详细谈谈直接声明符中第二个文法——（declarator）。这种形式一般用于声明指向数组的指针以及指向函数的指针对象。不过对于普通对象，其他指针对象以及数组对象也均能使用这种声明符表达方法。代码清单13-4将给出使用例子。

代码清单13-4　（declarator）的声明方式

```
#include <stdio.h>

int main(int argc, const char* argv[])
{
    // 声明了int类型对象a
    int (a) = 0;

    // 声明了指向const int类型的指针对象p
    const int (*p) = &a;

    // 声明了指向int类型的常量指针对象q
    int (* const q) = &a;

    // 声明了一个具有三个int类型元素的数组对象b
    int (b)[3] = { 1, 2, 3 };

    // 声明了一个具有三个int*类型元素的数组对象c
    int* (c)[3] = { &a };

    // 这里声明了一个指向int[3]数组的常量指针r
    int (* const r)[3] = &b;

    // 这里声明了包含两个int类型元素的数组对象d
    int (d[2]) = { 0 };

    // 这里声明了包含两个int类型元素的数组对象e
    int ((e)[2]) = { 0 };

    // const int (*)可以直接作为完整类型进行类型转换，这相当于const int *
    printf("a = %d\n", *(const int(*))p);

    // 同样，int((*))[3]也可直接作为完整类型进行类型转换，
    // 这里((*))与(*)的作用也是一样的，相当于int (*)[3]
    printf("b[0] = %d\n", ((int((*))[3])r)[0][0]);

    printf("Remain result: %d\n", *q + *c[0] + d[0] + e[1]);
}
```

从代码清单13-4中我们可以看到原本为指向数组的指针与指向函数的指针而设计的（declarator），由于在实际使用中C语言标准没有对其他声明做太多约束，所以我们可以对此文法玩出各种花样来。在代码清单13-4中，除了声明指向数组的常量指针对象r，其他标识符周围的（）都可以省去，而语义不变。

此外，在做类型转换时，如果（）内含有*，那么此圆括号可以保留，这也是为了迎合指向数组的指针与指向函数的指针类型而设计的；如果没有*，则不能加（）。像int（）这种是非法的类型表达方式。

13.3 更复杂的声明

我们前两节介绍了对象类型以及对象与函数的声明，本节我们将利用这些知识构造出更丰富且看上去更复杂的对象类型。尽管在大部分时候，像以下介绍的复杂类型没太大用武之地，但在关键时候总能帮上忙，而且一旦学会对这些类型的解析，那么相信再复杂的语法特性也难不倒你了。我们将分三步为大家介绍自己如何编写或去读懂一些更复杂的对象类型以及函数类型。

13.3.1 将某一类型转换为指向该类型的指针

首先交给大家的是，如何表达指向某一个对象的指针类型。

例1：像比较简单的，我们已经知道如果声明了一个对象a——int a；，那么对象a的类型就是int，而指向对象a的指针类型则是int*类型。当然正如上一节所述，这里的int*也可以表示为int（*）。

例2：如果声明了一个数组对象a——int a[3]；，那么数组对象a的类型为int[3]，而指向数组对象a的指针类型则为int（*）[3]。所以我们可以发现，要将某一类型转换为指向该类型的指针其实非常容易，只需要把用该类型声明的对象标识符直接替换为（*）即可！这里其实就是把数组对象标识符a变为了（*）。

例3：如果是一个含有3个指向函数的数组，比如：void（*pArray[3]）（int）；，如果要获得指向pArray数组对象的指针类型该怎么做？很简单！直接将pArray变为（*）即可——void（*（*）[3]）（int）。该类型表示指向一个含有3个元素的数组的指针，每个元素是指向void（int）类型的函数指针。

13.3.2 判定当前类型属于哪种对象类型

我们了解了上述对象转为指向该对象的指针类型之后，接下来就要区分该类型的类别，也就是说，给出的类型是普通对象类型还是指针类型抑或是数组类型。普通对象类型非常简单，只要在类型中不出现*号，也不出现[]下标符号的，那么就是普通对象类型。这里难以区分的是，当在一个类型中同时出现了*号以及[]符号时，如何判别该类型是指向一个数组的指针还是就是数组类型。这时，我们必须看类型表达式中最里面的那个（*）右边是否紧跟着一个[]，如果是，则表明它属于指向数组的指针；如果[]出现在（）内，跟在*右边，则表示它是一个数组对象。像13.1.1节中的例3中的pArray就是一个数组对象，因为我们将pArray标识符拿掉之后很明显就能看到，（）里的*后面有一个[3]。而&pArray则是指向数组的指针类型，因为它的类型是void（*（*）[3]）（int），最里面的（*）后面直接跟着一个[3]。

我们这里也举一些例子。

例1：对于void（**funcArray[3]）（int）；，对象funcArray就是一个数组，而不是一个指针，该数组中的每个元素是指向函数类型void（int）的指针的指针，即void（**）（int）类型。在C语言中，*作用于一个数组对象或函数时，有没有圆括号，其语义是完全不同的。典型的例子有以下这些。

例2：int*p[3]；表示对象p是一个数组，每个元素是int*类型的对象。

例3：int（*q）[3]；表示对象q是一个指向int[3]数组类型的指针。

例4：void*func（int）；表示func是一个函数，其类型为void*（int）。

例5：void（*pFunc）（int）；表示pFunc是一个指向void（int）类型的函数的指针。

类似地，判定一个声明是声明一个函数还是声明一个对象，就看最里层的（*）后面是否立即跟着（）形参列表，如果是，说明声明的是一个指向函数的指针类型，否则说明是一个函数类型。比如：void（*（float））（int）；表示声明的是一个函数类型，因为这里面的*并不是以（*）形式出现的，并且它具有一个float类型的形参，返回类型为void（*）（int）；。而void（*（*）（float））（int）；表示指向一个返回类型为void（*）（int），并带有一个float类型形参的函数的指针。如果（*）后面既没有直接跟[]数组下标符号，也没有跟（）函数形参列表，那么*号两旁的圆括号完全可省。所以对于函数声明来说，其形参列表就对应于数组

声明的下标。我们可以类比一下int（void）与int[3]，以及int（*）（void）与int（*）[3]。随后，我们通过代码清单13-5将上面所述的小例子都串起来。

代码清单13-5　更复杂的类型声明

```
#include <stdio.h>

static void func(int a)
{
    printf("a = %d\n", a);
}

// MyFunc的返回类型为void (*)(int)，并带有一个float类型的形参
static void (* MyFunc(float f) )(int)
{
    printf("f = %f\n", f);

    return &func;
}

int main(int argc, const char* argv[])
{
    // pArray是带有3个void(*)(int)类型元素的数组，
    // 其类型为：void (* [3])(int)
    void (*pArray[3])(int) = { NULL };

    // pp指向pArray数组对象，其类型为：void (* (*)[3])(int)
    // 注意，这里最里面的(*)两旁的()不可省
    void (* (*pp)[3])(int) = &pArray;

    pp[0][1] = &func;

    (*pp)[1](10);

    // pFunc指向MyFunc函数，其类型为：void (* (*)(float) )(int)
    void (* (*pFunc)(float))(int) = &MyFunc;

    // 这里分别做了两次调用。第一次调用的是MyFunc，
    // 第二次则调用的是在MyFunc函数最后所返回的func函数
    pFunc(2.5f)(20);
}
```

当我们知道了如何表达自己想要的类型之后，那么下面我们就再看一下如何去解析给定的含有比较复杂类型的对象与函数。

13.3.3　复杂复合类型的判断

通常，对于一个含有指针符号的对象来说，我们先从最里面（即最右边）的*号开始，或者如果类型中含有多重圆括号，那么从最里面那层圆括号开始解析，因为它是离对象标识符最近的，也是最能直接表示该对象所具备的第一层类型的。

例1：void（*（*pArray）[3]）（int）。像这里的pArray对象是什么类型呢？pArray对象的声明中含有指针，也含有圆括号，我们要判定其类型就从最里面的那个圆括号开始。最里面的圆括号中含有*号，也含有pArray对象标识符，说明该对象的类别肯定是一个指针。然后看该圆括号的右边有没有跟下标或函数形参列表。我们看到，这里右边紧跟着的是[3]，说明pArray对象是一个指向含有3个元素的数组的指针，那么这一层就结束了。然后看外面一层，这里也是（*）的形式，其左边是一个void类型，右边是一个（int）形参列表，说明它是一个指向返回类型为void，带有一个int类型形参的函数的指针类型。这么一分析之后，我们就可以得到pArray对象是一个指向3个元素的数组的指针，数组的每个元素是指向返回类型为void，并带有一个int类型形参的函数指针类型。

例2：int（*（*pFunc）（void））[3]。pFunc对象的类型中也含有*符号，并且也含有圆括号。我们仍然从最里面的那层圆括号开始分析。最里面的圆括号中包含了*符号以及pFunc标识符，说明pFunc一定是一个指针。然后看该圆括号右边跟着的是（void），它很明显是一个形参列表，表示无任何形参，说明pFunc是一个指向函数的指针类型。最后看外面一层圆括号，也是（*）的样式，其左边是int类型，右边是[3]，说明这是一个指向

int[3]数组的指针类型。因此，我们就可以把pFunc的完整类型给推断出来——它是一个指向不含任何形参的，返回类型为int（*）[3]的函数的指针。

例3：void（**array[3]）（int）。对象array的类型中也包含了*号以及圆括号。不过由于这里只有一个圆括号，所以我们可以直接对圆括号里的内容进行分析。这里，array标识符没有与某个*进行捆绑在一个圆括号中（即（*array）这种形式），它与*号是分开的，然后看array标识符后面紧跟着[3]，说明它是一个含有3个元素的数组。之后再看标识符所在的圆括号中其余内容，是两个*号，也就是（**），再看其右边跟着的是（int），说明它是一个指向函数指针的指针，在它左侧看到是一个void类型，说明函数返回类型是void。那么完整地看array，其类型为一个含有3个元素的数组，数组的每个元素的类型为指向返回类型为void，并带有一个类型为int形参的函数的指针的指针。当然，像这里的array也可以被声明为：void（*（*array[3]））（int）。即便在*array[3]外围再加一层圆括号其实也容易分析。这么一来就更能看出array是一个数组对象，然后array[0]则是一个指针类型了，而最外围的void（*）（int）就是array[0]这个指针所指向的类型。

从上面3个例子可以看出，对于诸如T（*id）<postfix>的声明而言，id就是一个指针对象，而它所指向的类型即为其圆括号外围的T<postfix>类型。

我们通过代码清单13-6来验证上面所做的类型推论是否正确。

代码清单13-6　对象类型推断验证

```
#include <stdio.h>

// 在全局作用域声明一个静态int类型元素的数组，并对它初始化
static int sArray[] = { 1, 2, 3 };

// 定义一个返回类型为void，带有一个int类型参数的静态函数func
static void func(int a)
{
    printf("a = %d\n", a);
}

// 定义一个返回类型为int(*)[3]，不带有任一形参的静态函数Fun
static int (*Fun(void))[3]
{
    return &sArray;
}

int main(int argc, const char* argv[])
{
    /** 先验证pArray对象 */

    // 这里先声明一个含有3个元素的数组对象arr，
    // 每个元素的类型为指向void(int)函数的指针
    void (*arr[3])(int) = { &func };

    // 在文中已推断出了pArray的类型为指向一个含有3个元素的数组的指针，
    // 数组中每个元素的类型为指向void(int)函数的指针
    void (*(*pArray)[3])(int) = &arr;

    // 通过pArray指向数组的指针来调用函数
    pArray[0][0](100);

    /** 然后验证pFunc对象 */

    // pFunc是一个指向函数的指针，该函数的返回类型为int (*)[3]，
    // 并不带任何形参
    int (*(*pFunc)(void))[3] = &Fun;

    // 通过pFunc做函数调用，
    // 由于pFunc所指函数的返回类型是int (*)[3]，所以pFunc()[0]则是int[3]，
    // 这里的pFunc()[0]与(*pFunc())是等价的
    int *p = pFunc()[0];
    printf("Sum is: %d\n", p[0] + p[1] + p[2]);

    /** 最后验证array对象 */

    // array对象类型为含有3个元素的数组，该数组的每个元素的类型是void (**)(int)，
    // 即指向返回类型为void，带有一个int类型形参的函数的指针的指针
    void (** array[3])(int) = { &arr[0] };

    // 通过array数组对象做函数调用
    (*array[0])(10);
}
```

通过代码清单13-6，通过一些类型相对简单的对象为这些复杂类型的

对象赋值，我们就能很快验证出这些类型确实如上述所推断的那样。

上述所描述的一些例子中对象的类型已经足够复杂了，当然我们还可以写出更复杂的类型，只要大家掌握上面所讲的关键点就能比较容易地判定某个对象或函数的类型，并且也能自己写出想要的类型。如果各位掌握了这部分知识的话，那么可以说基本已经能够随意驾驭C语言了。

13.4 typedef类型定义

我们上一节介绍了比较复杂的对象类型与函数类型。通常来说，我们看到那样复杂的类型第一感觉就是头晕想吐……因此C语言引入了类型定义，可以将复杂的类型抽象为一个类型标识符，这样我们就可以直接用定义好的类型标识符去声明一个对象或作为函数声明的一部分了。C语言中类型定义的语法与声明一个对象的语法很类似，仅仅是在最前面加上typedef关键字。比如，我们要用int类型来定义一个名为INT的类型，那么可以这么写：typedef int INT；。这里也是需要用分号作为结束符的。我们可以看到，如果把typedef去掉，那么就相当于定义了一个名为INT的对象。而前面一旦添加了typedef，那么INT标识符就不是一个对象标识符，而是一个类型标识符了，C语言标准也把类型标识符称为typedef名（typedef name）。

typedef名与结构体标签、联合体标签和枚举标签一样，都属于用户自定义类型，而且typedef也可以放在文件作用域和语句块作用域中。当然，如果typedef定义的是一个可变修改类型，那么它只能放在语句块作用域中。typedef还能嵌套定义，也就是用一个typedef名去定义另一个类型，比如：typedef INT MYINT；，这里用上述定义好的INT自定义类型再定义了一个MYINT这个类型，它与INT都一样，都属于int类型。

下面我们将分3个部分来描述typedef进行类型定义的方式与功能。第1

部分将介绍typedef一般的用法，我们如何用它来定义普通的对象类型以及函数类型，并且如何用typedef名去声明一个对象或一个函数。第2部分我们将描述typedef如何与限定符相结合，以及相结合之后限定符究竟限定什么类型。第3部分将介绍通过typedef来定义某个结构体或联合体类型。

13.4.1 typedef的一般使用

前面已经描述了，使用typedef来做类型定义时，其声明形式与声明一个对象的形式非常类似，仅仅是在最前面添加typedef关键字。而定义好的类型标识符就相当于用于定义该标识符的那个类型。与对象声明一样，对同一类型标识符的定义在同一作用域中可出现多次，但所定义的类型必须都相同。另外，用typedef做类型定义时，其声明只能放在文件作用域或语句块作用域，而不能声明在函数原型作用域中。代码清单13-7详细描述了typedef的一般使用方式以及一些需要注意的地方。

代码清单13-7 typedef的一般使用方式

```
#include <stdio.h>

// 这里使用了声明列表形式，将INT类型标识符定义为了int对象类型，
// 将FUNC类型标识符定义为了int(void)函数类型。
// 相当于：
// typedef int INT;
// typedef int FUNC (void);
typedef int INT, FUNC (void);

// 这里再次定义INT，仍然使用类型int，没有问题
typedef int INT;

// 以下语句将出现编译错误。由于INT标识符已经被定义为int，
// 不能用其他与int不同的类型再定义INT
typedef short INT;

// 这里定义了一个名为ARRAY的数组类型int[3]
typedef int ARRAY[3];
```

```
// 以下这条语句声明了一个名为func的函数，该函数的类型为FUNC，即int(void)
static FUNC func;

// 这里直接用ARRAY去声明一个数组对象array，array的类型为int[3]
static ARRAY array;

int main(int argc, const char* argv[])
{
    // 调用函数func
    func();

    // 对array数组进行赋值
    for(int i = 0; i < 3; i++)
        array[i] = i;

    // 这里用已定义好的类型INT又声明了一个PINT类型与PPINT类型。
    // PINT类型为int*，而PPINT类型为int**
    typedef INT *PINT, **PPINT;

    // 这里用PINT声明了指针对象p，并将它初始化为指向array数组首个元素。
    // p的类型为int*
    PINT p = &array[0];

    // 这里用PPINT声明了指针对象q，并用p的地址对它初始化。q的类型为int**
    PPINT q = &p;

    **q += 10;
    printf("array[0] = %d\n", array[0]);

    *q = NULL;
    if(p == NULL)
        puts("p is null!");

    // 这里用PINT*来声明一个指针对象r。由于PINT本身是int*类型，
    // 所以在对象标识符r之前再添加一个*，使得r的类型变为了int**
    PINT *r = &p;
    *r = &array[2];
    if(*p == array[2])
        puts("Equal!");

    // 同样，我们这里通过FUNC类型后面添加*来声明一个函数指针对象pFunc。
    // pFunc的类型即为int(*)(void)
    FUNC *pFunc = &func;
    INT a = pFunc();
    printf("a = %d\n", a);

    // 这里需要注意的是，FUNC类型本身是函数类型，用它声明的标识符是一个函数，而不是一个对象。
    // 一个函数标识符只有在充当非左值的情况下才能隐式地转换为指向函数的指针对象。
    // 它是不允许作为左值的！所以以下语句是错误的
    FUNC aFunc = func;

    // 同样，我们也可以在ARRAY类型后面添加*来声明一个指向数组的指针对象pArray
    ARRAY *pArray = &array;
    printf("array[1] = %d\n", pArray[0][1]);

    // 类型定义与对象声明一样，*在不同的位置所表达的类型语义有所不同，
    // 这里ARRAY_PTR的类型为int*[3]
    typedef INT* ARRAY_PTR[3];

    // 这里PARRAY的类型为int(*)[3]
    typedef INT (*PARRAY)[3];

    ARRAY_PTR array2 = { &a, p };
    printf("The value is: %d\n", *array2[0] + array2[1][0]);

    PARRAY pArray2 = &array;
    if(pArray == pArray2)
        puts("OK!");
```

```
}

// 我们这里要注意的是，定义一个函数必须使用完整的函数原型，而不能使用类型定义的类型标识符
static int func(void)
{
    printf("%s is called!\n", __func__);
    return 100;
}
```

代码清单13-7展示了typedef进行类型定义的一般用法。我们从中可以看到，用typedef来定义一个类型标识符的方式与声明一个对象的方式相当类似。此外，用typedef定义好的一个类型标识符可以再次用于定义其他类型，并且可以随意与*号、下标与形参列表相结合，构成种类丰富的指针类型、数组类型以及函数类型。用了类型定义之后，原本像指向函数的指针、指向数组的指针等类型顷刻间就变得简单多了，这是C语言对类型的一种抽象，尤其用于实际项目的开发过程中还是比较有用的。这可以使得底层库的实现者将一些复杂的类型抽象掉，使得上层开发人员不必去关心某个类型标识符具体是哪种类型，而只要适当使用即可。

另外，大家还需要注意的是，typedef定义好的类型标识符是属于类型，所以我们不能用存储类说明符去修饰类型标识符，而类型标识符本身也没有连接这一概念，整个类型系统也不存在存储类限定的说法。因此存储类说明符只能用于修饰对象与函数，表示该对象或函数所具有的连接以及存储属性和生命周期。

我们之前也谈到宏可以用来预定义某个类型，那么宏与typedef在用于类型定义时有哪些差异呢?

首先，typedef可以定义几乎所有类型，比如指向函数的指针类型、指

向数组的指针类型等，而这一点宏是做不到的。

其次，宏用来做类型定义时，我们可以在任何地方通过#undef去取消当前的宏定义，将当前宏替换成另一种类型；typedef则无法实现取消定义的处理。

最后，宏与类型定义所受影响的作用域不一样。宏完全属于文件作用域（更确切地说，是当前整个翻译单元），不受语句块作用域的影响，而typedef定义的类型则具有文件作用域或语句块作用域。当然，宏与类型定义在本质上的不同就是，宏属于预处理，独立于C源代码的正式编译；而类型定义则是在编译期间处理的。

13.4.2 typedef与类型限定符相结合的使用

前面讲述了typedef类型定义的普通用法，本节将结合类型限定符来描述typedef的使用。

首先，用typedef定义的类型标识符本身可以用类型限定符进行修饰。其次，在typedef定义中，我们可以将类型限定符与类型名组合在一起来定义某个类型。如果在用typedef进行类型定义的过程中已经含有了某个类型限定符，比如const，那么在用该定义的类型标识符去声明某个对象时也伴随着同样的类型限定符，此时不会引发类型冲突，见如下代码片段所示。

```
typedef const int CINT;
CINT const a = 10;
```

上述代码片段中，我们用const int来定义了一个类型CINT，然后再用CINT来声明了一个常量对象a。这里我们看到，在用CINT声明对象a的时候，还伴随着一个const类型限定符，此时编译器不会报错，也不会有警告，这条声明语句仍然有效，并且对象a是一个int类型的常量，即const int类型。

当用typedef定义了一个指针类型之后，当它与类型限定符相结合时究竟是什么类型呢？比如：

```
typedef int *PINT;
const PINT p;
```

这里，PINT类型是一个完整的int*类型，随后我们用PINT类型去声明一个指针对象p，并且在PINT之前添加了const限定符，那么这时候，p究竟是什么类型呢？我们在第12章中已经讲过，当类型限定符限定一个类型时，如果该类型是一个指针类型，那么看它与*号之间的位置。我们在这个代码片段中看到，PINT就是指代int*类型，根据之前的判定方式，似乎p应该是const int*类型。然而，这里的PINT是直接将int与*号绑定在一起的，并且从整个对象声明上来看，这里就出现了const限定符与CINT类型说明符这两个声明说明符，所以这就意味着这里的const限定符限定的是完整的PINT类型，即int*，所以等同于PINT const p；，也就相当于int*const p；了。

我们在第12章中已有所描述，对于C语言初学者来说，我们在对一个含有类型限定符的对象进行声明时，尽量将类型限定符写在所限定类型的

后面，这样更容易看出类型限定符修饰的究竟是什么类型，并且也不容易搞混。代码清单13-8详细描述了typedef定义的类型标识符与类型限定符相结合的使用。

代码清单13-8　typedef类型名与类型限定符的结合

```
#include <stdio.h>

int main(int argc, const char* argv[])
{
    // 这里定义了类型PINT，其类型为int*
    // 并且该定义与 typedef int* PINT; 等同
    typedef int (*PINT);

    int a = 0;

    // 用PINT类型声明了一个指针对象p，并初始化为指向对象a的地址
    const PINT p = &a;

    // 以下这条语句错误。由于p是int * const类型，p的值不能被修改
    p = NULL;

    *p = 10;    // 这条语句OK
    printf("a = %d\n", a);

    // 将CPINT定义为const int *类型
    typedef const int *CPINT;

    // 这里用CPINT类型声明了指针对象q，
    // 由于这里还用了限定符const，因此q的类型为const int* const
    const CPINT q = &a;

    // 以下两条语句均不合法
    *q = 10;
    q = NULL;

    // 这里将PCINT定义为int * const类型
    typedef int * const PCINT;

    // 这里用PCINT声明了指针对象r，并且这里在声明中也用了const限定符，
    // r的类型仍然为int * const
    const PCINT r = &a;
    *r += 10;   // OK
    r = NULL; 不合法
}
```

13.4.3　用typedef来定义结构体与联合体的类型

我们之前提到过，用typedef定义的一个类型能起到抽象具体类型的作

用。这一点我们在实际项目工程中会比较多见，比如我们将一个描述矩形位置与宽高的结构体定义为Rect，此时上层应用开发者无需知道Rect究竟是一个结构体还是联合体或某些类型的组合，而是直接根据文档中的用法去使用即可。我们使用typedef也可对枚举、结构体以及联合体做类型别名的抽象。当我们用typedef去定义某个枚举、结构体或联合体的一个抽象类型时，那么用定义好的类型标识符去声明一个对象时，enum、struct以及union关键字必须缺省。如果不缺省，那么编译器会根据这些类型关键字去查找相应的类型。比如，如果有“typedef struct Test{int a，b；}TEST；”，那么倘若我们这么声明一个Test结构体对象“struct TEST t；”，那么编译器就会报错：“变量t具有不完整类型‘struct TEST’”。所以，我们要么用“struct Test t；”要么用“TEST t；”进行声明。

此外，我们一般也会用typedef将一个匿名枚举、结构体或联合体来定义为某一抽象类型。代码清单13-9给出了更多的用法。

代码清单13-9　typedef用于定义一个枚举、结构体和联合体的情况

```
#include <stdio.h>

// 通过typedef将结构体MyRect定义为RECT
typedef struct MyRect
{
    struct
    {
        float x, y;
    } positon;

    struct
    {
        float width, height;
    } size;
} RECT;

// 这里将一个匿名枚举类型定义为TRAFFIC_LIGHT
typedef enum
{
    RED_LIGHT,
```

```
    YELLOW_LIGHT,
    GREEN_LIGHT
} TRAFFIC_LIGHT;

// 我们可以先用typedef定义好一个类型，
// 尽管此时union Vertex_Attr是一个不完整类型，但是不影响类型定义
typedef union Vertex_Attr VERTEX_ATTR;

// 这里直接定义Vertex_Attr类型即可，typedef可省
union Vertex_Attr
{
    float position[4];
    float color[4];
};

// 这里用typedef先定义了NODE类型与PNODE类型
typedef struct Node NODE, *PNODE;

struct Node
{
    int data;

    // 由于之前已经定义好了PNODE类型，因此这里可直接使用。
    // link的类型为struct Node*
    PNODE link;
};

int main(int argc, const char* argv[])
{
    // 这里我们用RECT去声明对象rect时，前面不能添加struct关键字
    RECT rect = { 10.0f, 20.0f, 50.0f, 60.0f };

    // 同样，这里TRAFFIC_LIGHT前不能添加enum
    TRAFFIC_LIGHT light = GREEN_LIGHT;

    // 这里的VERTEX_ATTR前不能添加union
    VERTEX_ATTR vertexColor = { .color = {0.1f, 0.9f, 0.1f, 1.0f} };

    printf("The value is: %f\n",
           rect.positon.y + light + vertexColor.color[1]);

    // 我们用NODE类型声明了一个node_list数组对象
    // 下面我们做一个简单的线性单链表的搜索算法
    NODE node_list[3];

    // 先为node_list数组的各个成员进行初始化
    node_list[0].data = 1;
    node_list[0].link = &node_list[1];

    node_list[1].data = 2;
    node_list[1].link = &node_list[2];

    node_list[2].data = 3;
    node_list[2].link = NULL;

    // 然后我们声明对象p作为线性列表的表头节点
    PNODE p;

    // 下面我们找到data值为3的那个节点
    for(p = node_list; p != NULL; p = p->link)
    {
        if(p->data == 3)
        {
            puts("The node is found!");
            break;
        }
    }

    if(p == NULL)
```

```
        puts("Node not found!");
}
```

从代码清单13-9中我们可以看到，typedef用于定义一个抽象的枚举、结构体和联合体类型时显得十分灵活。当然，这里面还没展示出与类型限定符的结合，但原理都一样，只需要在typedef后面或类型标识符前添加即可，存在*号的情况下则按自己的需要摆放好位置。我们还看到了，在使用typedef时，如果此时用来定义类型的枚举、结构体或联合体本身尚未被定义，那也不影响类型定义，稍后可以补上对这些具体类型的定义。

13.5　本章小结

本章我们更深入地讲解了C语言的类型系统，并且详细地介绍了对象与函数的声明文法。通过对本章的学习，各位对C语言的设计理念，尤其是类型系统的设计上会有更深刻的认识。本章最后也描述了C语言中的类型定义使用方法以及注意事项。各位如果将本章学习透彻，那么可以说基本上就踏入了C语言大师的行列了。

至此，C语言标准中的大部分语法都讲解得差不多了，下一章将主要描述C11标准新引入的泛型表达式与静态断言。

第14章　C11标准中的表达式、左值与求值顺序

C11标准细分了17种表达式，而这17种表达式中有相互包含的情况。我们先大致看一下这17种表达式：

1）基本表达式：包括了标识符、常量、字符串字面量、圆括号表达式（即（表达式）这种形式），以及泛型选择表达式。14.2节将详细介绍C11标准新引入的泛型选择表达式。

2）后缀表达式：包括了基本表达式、带有下标的表达式（比如：array[10]）、函数调用（比如：Func（10））、结构体或联合体的成员访问（比如：obj.a或pObj->a）、后缀++和--、匿名结构体或联合体的初始化列表（比如：（struct S）{.a=10，.b=20}）。

3）单目表达式：包括了后缀表达式、前缀++和--、单目操作符结合投射表达式（比如：-（int）10.5）、sizeof表达式、_Alignof表达式。这里要注意的是，C语言中所规定的单目操作符只有&、*、+、-、~、！，前缀++和--。这里的&表示地址操作符，而不是按位与；这里的*表示间接操作，而不是乘法计算；这里的+和-表示正负号，而不是加和减。

4）投射表达式：包括了单目表达式、（类型名）投射表达式。比如

（int）10.5。

5）乘法表达式：包括了投射表达式，带有乘法操作符、除法操作符或求模操作符的表达式（比如：3*5、6/2、7%3）。乘法表达式中，如果含有乘法操作符、除法操作符或求模操作符，那么操作符左边是乘法表达式，右边是投射表达式。

6）加法表达式：包括了乘法表达式，带有加法操作符或减法操作符的表达式。如果加法表达式中含有加法操作符或减法操作符，那么操作符左边为加法表达式，右边为乘法表达式。

7）移位表达式：包括了加法表达式，带有左移或右移的表达式。如果移位表达式中含有左移或右移操作符，那么操作符左边为移位表达式，右边为加法表达式。

8）关系表达式：包括了移位表达式，带有小于、大于、小于等于、大于等于操作符的表达式。如果关系表达式中含有小于、大于、大于等于或小于等于操作符，那么操作符左边为关系表达式，右边为移位表达式。

9）相等表达式：包括了关系表达式，带有==或！=操作符的表达式。如果相等表达式中带有==或！=操作符，那么操作符左边为相等表达式，右边为关系表达式。

10）按位与表达式：包括了相等表达式，含有按位与操作符&的表达式。如果按位与表达式中含有按位与操作符，那么操作符左边为按位与表

达式，右边为相等表达式。

11）按位异或表达式：包括了按位与表达式，带有按位异或操作符^的表达式。如果按位异或表达式中含有按位异或操作符，那么操作符左边是按位异或表达式，操作符右边是按位与表达式。

12）按位或表达式：包括了按位异或表达式，带有按位或操作符|的表达式。如果按位或表达式带有按位或操作符，那么操作符左边是按位或表达式，右边是按位异或表达式。

13）逻辑与表达式：包括了按位或表达式，带有逻辑与操作符&&的表达式。如果逻辑与表达式中带有逻辑与操作符，那么操作符左边为逻辑与表达式，右边为按位或表达式。

14）逻辑或表达式：包括了逻辑与表达式，带有逻辑或操作符||的表达式。如果逻辑或表达式中带有逻辑或操作符，那么操作符左边为逻辑或表达式，右边为逻辑与表达式。

15）条件表达式：包括了逻辑或表达式，？：结合的三目表达式。三目表达式的形式为：逻辑或表达式？表达式：条件表达式。

 注意：这里对条件表达式的描述是C11标准中的描述，而我们平时在使用条件表达式的时候应该参考8.2节中描述的形式，也就是将这里？之前的表达式视作为布尔表达式。

16）赋值表达式：包括了条件表达式，以及这种形式的表达式：单目表达式赋值操作符赋值表达式。赋值操作符为以下操作符：=、*=、/=、%=、+=、-=、<<=、>>=、&=、^=、|=。

17）表达式：这里的表达式即为基本表达式中圆括号里的那个表达式。它包括了赋值表达式，以及这种形式的表达式：表达式，赋值表达式。这也就是我们在8.1节中所描述的逗号表达式。

我们可以看到，表达式的排列顺序表明了表达式所涵盖操作符的计算优先级。这里优先级最高的显然就是圆括号了，优先级最低的则是逗号。这里各位要着重注意的是，按位操作符的优先级小于关系操作符，因此我们在使用条件判定语句的时候一定要给按位操作符加上圆括号。如代码清单14-1所示。

代码清单14-1　注意按位操作符与关系操作符的优先级

```
#include <stdio.h>
#include <stdbool.h>

int main(int argc, const char* argv[])
{
    int a = 10;

    // 这里的条件判断是真，后面会输出OK
    if((a & 1) == 0)
        puts("OK");

    // 这里的if语句中相当于：
    // a & (1 == 0)，所以先计算1 == 0这个相等表达式
    // 然后计算a & false，所以整个表达式的结果为false
    if(a & 1 == 0)
        puts("Ooops");

    bool result = a & 1 == 0;
    printf("result = %d\n", result);
}
```

14.1 常量表达式

常量表达式是指该表达式在编译期间就能够被计算出来，而不需要生成相应运行时代吗。C语言的常量表达式是在“条件表达式”这个层级，也就是说赋值表达式不能作为一个常量表达式，当然并不是所有在条件表达式这一层级的表达式都能作为常量表达式。常量表达式不应该含有赋值、递增、递减、函数调用以及逗号操作符，除非它们作为包含在一个不被计算的子表达式中（比如包含在sizeof操作数中）。

对于初始化器中的常量表达式可以有更宽松的限定，它可以是：

1）一个算术常量表达式；

2）一个空指针常量；

3）一个地址常量；

4）一个完整对象类型的地址常量加上或减去一个整数常量所构成的一个表达式。

一个算术常量表达式应该具有算术类型，并且其操作数应该仅为整数常量、浮点常量、字符常量、不含有可变修改类型的sizeof表达式，以及_Alignof表达式。而在一个算术常量表达式中的投射操作也应该只是将一种算术类型转换为另一个算术类型，除非作为sizeof与_Alignof的操作数。

我们要判定一个表达式是否为常量表达式其实比较简单，我们对一个全局对象进行初始化，如果能通过编译，那么为它初始化的表达式就基本是一个常量表达式，否则它就不是一个常量表达式。代码清单14-2展示了判定常量表达式的方法。

代码清单14-2　常量表达式的判定

```
#include <stdio.h>
#include <stdint.h>
// 这里声明了一个静态变量对象a，并且这里的100是一个常量表达式
static int a = 100;

// 由于静态文件作用域对象a不是一个常量，所以这里的a不是一个常量表达式。
// 这条语句会引发编译错误—初始化器元素不是一个编译时常量
static int b = a;

// 这里用const声明了一个常量对象c，并且这里的(200 - 100) / 2是一个常量表达式
static const int c = (200 - 100) / 2;

// 由于对象c是一个常量，所以这里的(c + 2) << 1是一个常量表达式
static const int64_t d = (sizeof(c) + 2) << 1;

// 整个(sizeof(d) > sizeof(c))? sizeof(d) + 1 : sizeof(c) - 1表达式，
// 是一个常量表达式
static int e = (sizeof(d) > sizeof(c))? sizeof(d) + 1 : sizeof(c) - 1;

// 如果一个常量表达式作为一个初始化器，那么它可以是一个地址常量。
// 这里，&e 这个表达式就是一个地址常量
int *p = &e;

// 在GCC与Clang中，被const修饰的一个常量对象也能作为初始化器的一个常量表达式，
// 但在MSVC中却不被允许，C语言标准没有指定const修饰的对象是否能作为一个常量表达式，
// 但C语言标准声明了，允许C语言实现接受其他形式的常量表达式
static int64_t maybeError = c + d;

int main(int argc, const char* argv[])
{
    printf("a = %d, c = %d, d = %lld, e = %d\n", a, c, d, e);
}
```

代码清单14-2列举了一些能作为常量表达式的表达式形式。对于常量表达式而言，比如上述的（200-100）/2，在实际编译后的二进制代码中不会包含整个计算表达式的过程，而仅仅是一个计算结果，即50。如果在函数中出现像int a=（200-100）/2；这种语句，最终整条语句对应的指令可能

仅仅就是“mov reg，50”，假设reg表示变量a所在的寄存器，而不会有减法、除法等指令出现，这些计算全都由编译器负责计算。

当然，一些编译器可能会根据上下文对代码进行优化。比如像int a=100；和int b=a+50；这两条语句中，a和b都是变量，但是在编译int b=a+50；这句时编译器可能会直接将它编译为：mov reg，150，假设reg表示变量b所用的寄存器。然而在C语言语法上，a+50仍然不属于常量表达式。

14.2 泛型选择表达式

泛型选择表达式是C11标准新引入的一个重大语法特性。它可以使得C语言使用轻量级的泛型机制，其表达形式如下：

```
_Generic (赋值表达式, 泛型关联列表)
```

这里的“赋值表达式”其实就是我们在本章开头所列出的第16条表达式，也是范围第二大的。而泛型关联列表的形式是一组用逗号分隔的类型与表达式相联结的表达式，形式为：

```
类型名 ： 赋值表达式。
```

整个泛型选择表达式的语义为：C语言实现先获取最左边的赋值表达式的类型，这里要注意的是，此获取类型的动作与sizeof表达式一样，仅获取类型而不对表达式做计算，也不会生成相关的运行时代码；然后将获取到的类型与泛型关联列表中每一个“类型名”部分进行比较，如果两者兼容则选择该“类型名”所对应的赋值表达式作为整个泛型表达式的计算结果，否则跳过当前的泛型关联，尝试匹配下一条。其中，类型名部分还可以用default来表示当泛型关联列表中没找到与最左边的赋值表达式的类型相匹配的类型时所选出的表达式。

代码清单14-3列举了泛型选择表达式的基本用法以及一些注意事项。

代码清单14-3　泛型选择表达式的基本使用

```
#include <stdio.h>

int main(int argc, const char* argv[])
{
    // 这里列举了一个简单的泛型选择表达式。
    // 该表达式中，对表达式100进行获取类型，然后对后面的泛型关联进行匹配。
    // 我们知道100属于int类型，因此最终整个泛型选择表达式的结果为表达式1。
    // 这条语句在编译完成后其实就相当于：int a = 1;
    int a = _Generic(100, float:-1, int:1, default:0);
    printf("a = %d\n", a);              // 这里输出：a = 1

    // 在这条泛型选择表达式中，
    // (++a, a + 1.5f)这一逗号表达式最终计算出的类型是float，
    // 由于泛型选择表达式中最左边用于类型匹配源的赋值表达式不被计算，
    // 所以这里的++a没有任何效果
    a = _Generic((++a, a + 1.5f), int:a + 10, float:a + 100,
                 double:a + 1000, default:a);
    printf("second a = %d\n", a);       // 这里输出：a = 101

    // 尽管在一般赋值表达式中，float能隐式地转换为double类型，
    // 但是在泛型选择表达式中，类型匹配是相当严格的！a + 1.5f是float类型，
    // 那么由于在这里找不到float类型，
    // C语言实现就会选择default中的表达式作为整个泛型选择表达式的结果。
    // 这条语句就相当于a = a; 其实就如同一条空语句
    a = _Generic(a + 1.5f, int:a + 10, long:a + 100,
                 double:a + 1000, default:a);
    printf("third a = %d\n", a);        // 这里仍然输出：a = 101

    const char *output = "none";
    // 当我们的匹配源表达式是一个字符串字面量的时候必须注意，
    // C语言标准中是将字符串字面量视作为char*类型，但有些C语言实现在匹配泛型关联的时候，
    // 可能仍然会将字符串类型设定为const char[N]，N表示字符串中字符个数再加一个'\0'结束符。
    // 因此使用时应当小心，尽量使用投射操作做显式的类型转换
    output = _Generic("abc", const char* : "const char*", char* : "char*",
                      const char[4] : "const char[4]",
                      char[4] : "char[4]");

    // 我们尝试下面这条语句，
    // 会看到当前编译器在做编译报错时仍然会把"abc"作为char[4]或const char[4]类型
    "abc" = 100;

    printf("output is: %s\n", output);

    struct Point { int x, y; };
    struct Size { int width, height; };
    struct Point p = { };

    // 结构体的类型匹配也同样如此
    output = _Generic(p, struct Point:"Point",
                      struct Size:"Size", default:"none");
    printf("p is a %s\n", output);

    int x, y;

    // 下面我们通过投射操作加逗号表达式作为泛型关联，分别给x、y两个对象进行初始化。
    // 我们在每个泛型关联中的表达式的前面加上(void)，表示整个表达式为void表达式。
    // 要注意，如果一个泛型关联中出现多个赋值表达式，需要用逗号分隔，不能使用分号。
    // 此外，需要给这些赋值表达式加上圆括号，以防止逗号作为泛型关联的分隔符
    _Generic(x + y, int:(void)(x = 1, y = 2),
            float:(void)(x = 0.1f, y = 0.2f), default:(void)0);

    printf("x = %d, y = %d\n", x, y);
}
```

泛型选择表达式最常用的是用于做一些标准库。比如，像C语言标准库中不少数学函数都带有类型前缀与后缀，通过泛型表达式我们可以将这些前缀与后缀给应用开发者给抽象掉，不暴露出来，这样便于开发者更便捷地使用这些标准库API。下面再举一些例子来说明这种用法。

代码清单14-4　泛型选择表达式用于标准库的制作

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

// 这里定义了一个gen_abs宏函数，用于判定expr表达式的类型。
// 如果是int，则使用abs库函数；如果是long，则使用labs；
// 如果是float，则使用fabsf库函数；如果是double，则使用fabs库函数；
// 如果是long double，则使用fabsl库函数；默认使用fabs
#define gen_abs(expr)  _Generic((expr), int:abs, long:labs, float:fabsf, \
                       double:fabs, long double:fabsl, default:fabs)     \
                       (expr)

// 这里定义了一个gen_pow宏函数，判定base + exponent这一表达式的类型
#define gen_pow(base, exponent) _Generic((base) + (exponent), \
                                         float:powf, \
                                         double:pow, long double:powl, \
                                         default:pow) (base, exponent)


int main(int argc, const char* argv[])
{
    // 由于－100表达式是int类型，所以这里选择了abs另外这里再要提醒各位的是，
    // 一个函数标志用在表达式中则默认表示为一个指向函数的指针类型，
    // 所以这里整个泛型选择表达式的结果为：abs (-100)
    int a = gen_abs(-100);

    // 由于1000L表达式是long类型，所以这里选择了labs
    long l = gen_abs(1000L);

    // 由于0.5f表达式是float类型，所以这里选择了fabsf
    float f = gen_abs(0.5f);

    printf("value = %f\n", a + l + f);

    // 这里base的类型为float，exponent的类型为6.0，两者相加的类型则取double。
    // 因此，这里泛型选择的最终表达式为pow(2.0f, 6.0)
    double d = gen_pow(2.0f, 6.0);
    printf("d = %d\n", (int)d);
}
```

我们在使用泛型表达式的时候必须要注意，在泛型关联列表中不能出

现两个一样的类型，也不能找不到任何与赋值表达式的类型匹配的泛型关联，否则会引发编译报错。对于无法找到所匹配类型的情况，我们应该要加上default泛型关联。另外，泛型关联中类型后面要跟的是一个表达式，而不是一条语句，所以不能出现分号，也不能出现{}这种语句块。

除了上述提到的基本类型外，对于我们自定义的枚举、结构体、联合体类型以及各种指针类型也都可以用泛型表达式。然而，当泛型关联中的表达式为赋值表达式时，当前主流的C语言编译器（包括GCC 5.4以及Clang 3.8）对用户自定义类型的泛型选择支持都不太好，因此我们当前应尽量使用C语言中的基本类型作为匹配类型，等以后C编译器完善了再使用用户自定义类型也不迟。

14.3 静态断言

静态断言这一语法特性也是C11标准新引入的。其意图是让C语言程序员在编译时就能提示出可在编译时判定出的错误，并输出相应提示。静态断言是一条声明，即一条语句，而不是表达式，不过这个语法特性在其他章节讲也不是十分合适，所以就放到这里介绍。

静态断言的语法形式为：

```
_Static_assert (常量表达式,字符串字面量);
```

它表示：如果常量表达式的值为假（即计算结果为0或者是一个空指针），那么断言失败，C语言实现将在编译时产生一条诊断信息，诊断信息中包含后面字符串字面量的信息；如果常量表达式的值计算结果为真，那么断言成功，该声明不起任何作用。各位要注意的是，这里的常量表达式必须是一个整数常量表达式。一个整数常量表达式应该具有整数类型，并且其操作数应该仅为整数常量、枚举常量、字符常量、sizeof表达式、_Alignof表达式，以及经过整数类型投射操作的浮点常量。此外，这里的sizeof与_Alignof的操作数不应该是可变修改类型。而像一个地址常量等表达式在这里都不能作为一个整数常量表达式。

我们在使用静态断言的时候应该添加上标准库头文件<assert.h>，然后直接使用预定义的宏函数static_assert，而不是直接使用_Static_assert。代码

清单14-5展示了静态断言的基本使用方式与效果。

代码清单14-5　静态断言的基本使用与效果

```
#include <stdio.h>
#include <assert.h>
#include <stdbool.h>

int main(int argc, const char* argv[])
{
    // 这里常量表达式直接是真值，所以断言成功，不产生任何效果
    static_assert(true, "This is true");

    // 这里常量表达式直接是一个假值，所以断言失败，编译器将产生诊断信息
    static_assert(false, "This is false");

    int a = 10;

    // 由于这里&a不是一个整数常量表达式，所以编译器直接报错：
    // 静态断言表达式不是一个整数常量表达式
    static_assert(&a, "&a is not null!");

    // 由于sizeof(a)的大小不为0，因此断言成功，这里不产生任何效果
    static_assert(sizeof(a), "p is null!");

    enum COLOR { RED, GREEN, BLUE };

    // 由于枚举常量RED是0，所以这里断言失败，编译器将产生诊断信息
    static_assert(RED, "This is red color");
}
```

从代码清单14-5我们可以看到，使用静态断言的条件非常有限。我们一般用静态断言对当前编译环境进行判定，而决不能用于对运行时对象值的判定。

14.4　C语言中的左值

关于编程语言中的“左值”，一般通俗地来说就是可作为=赋值操作符左侧操作数的表达式，这也意味着等号左侧的表达式要求是一个可被修改的左值。C11标准给C语言的左值做了明确的定义：一个左值表达式能隐式地用来表示一个对象；如果一个左值在计算时无法用来表示一个对象，那么行为是未定义的。一个可修改的左值不能是一个数组类型，不能是一个不完整类型，不能有const限定符修饰；并且如果该左值是一个结构体或联合体类型的话，其任一成员也不能有const限定符修饰。

当一个左值作为单目&操作符、++操作符、--操作符的操作数，或成员访问操作符.和赋值操作符=的左操作数时，整个表达式就不具备左值特性了，这在C语言中称为左值转换。

下面我们就通过代码清单14-6来举一些左值表达式以及非左值表达式的例子。

代码清单14-6　C语言中的左值

```
#include <stdio.h>

int main(int argc, const char* argv[])
{
    int a = 10;
    int *p;

    // 这里的a是一个左值
    a += 5;

    // 这里的p是一个左值
    p = &a;
```

```
    // 当a作为++操作符的操作数时，它就不再是左值了。
    // 所以a++表达式不是左值，不能作为=操作符的左操作数。这条语句将会引发编译错误
    // a++ = 0;

    // 声明一个类型为array[5]的数组对象
    int array[5] = { 1, 2, 3 };

    // 数组类型的对象不能作为左值，所以它也不能作为=操作符的左操作数，以下语句会引发编译错误
    array = (int[]){ 4, 5, 6 };

    // array[1]表达式是一个左值
    array[1]++;

    // main是int(int, const char**)的函数类型，不能作为左值
    main = NULL;    // 这条语句将会引发编译错误

    int (*pFunc)(int, const char**);

    // pFunc是指向函数的指针类型，可以作为左值
    pFunc = &main;

    // 这条语句将会引发编译错误。作为=操作符的左操作数之后，a就不再作为左值，
    // 因此整个(a = 10)表达式不是一个左值，它不能作为=操作符的左操作数
    (a = 10) = 100;

    p = array;

    a = 0;

    // a++不是一个左值表达式，但p[a++]表达式是一个左值
    p[a++] = 30;

    // 当左值p[a]作为&的操作数之后就不再是左值，因此以下语句将会引发编译错误
    &p[a] = NULL;

    // 而当&p[a]前面再加*进行解引用之后，*&p[a]表达式又再度成为了左值
    *&p[a] += 10;
}
```

代码清单14-6中后面牵涉指针的引用与解引用部分可能会感觉比较绕。不过从逻辑上很容易理解，我们取一个对象的地址时得到的是该对象的地址值，地址值是不能被修改的，因此对对象的引用自然就是一个常量，不能作为一个左值了。而取一个指针的内容时，由于前面没有const修饰，自然就能访问相应地址的数据内容，因此解引用表达式很自然就能作为左值。

另外，代码清单14-6也暗示了，当一个函数标志作为表达式时，只有当它扮演“右值”时，其类型才会被隐式转为指向该函数的指针类型，否则

它依然保持为函数类型。另外，当一个函数标志后面跟着（）后缀操作符表示函数调用时，该函数标志就不再是一个左值了，它具有指向该函数的指针类型。对于一个数组对象，通常它均以指向其元素类型的指针的形式作为表达式的，除非作为单目操作符&、sizeof、_Alignof等操作符的操作数时，保留其数组类型。

“右值”在C11标准中被描述为“一个表达式的值”，在7.4节也有过一些介绍。不过右值这个概念在C语言中用得不多，因为C语言的类型系统相对来说还是比较简单的，而C++则要复杂许多，并且也会频繁用到右值这个概念（比如C++11的右值引用，等等）。所以各位务必不要混淆了C语言跟C++语言对左值和右值的概念，两者是不太一样的。C++是将右值定义为“作为一个临时对象”，显然范围比C语言的要广很多。

下面举个例子。在C语言中，我们把之前6.2.3节中提到的匿名结构体、6.3节中提到的匿名联合体以及7.1节中提到的匿名数组统称为复合字面量（compound literal）。C语言标准对复合字面量的定义很明确：由一个圆括号包起来的类型名后面跟着用花括号包起来的初始化器列表所构成的一个后缀表达式。复合字面量提供了一个匿名对象，其值由初始化器列表提供。在标准中，这个定义有一个脚标，注明了：复合字面量与投射表达式不同。比如，投射操作仅仅指定了对标量类型或void类型的转换，并且投射表达式的结果不是一个左值。这句话非常关键，信息量庞大！这其实已经意味着复合字面量可以作为左值使用，而且实际上也确实如此。但在C++中将这种类似的、产生匿名对象的表达式称为右值。我们看一下代码

清单14-7所示的C++14代码。

代码清单14-7　C++14语言中的临时对象

```
int main(void)
{
    struct Test
    {
        int a, b;
        void set(int i) { a = i; b = i + 1; }
    };

    // 这里的表达式Test{ 10, 20 }是一个右值
    Test{ 10, 20 };
    // C++14中允许对一个临时对象，即一个右值调用非const方法来修改其成员值
    Test{ 10, 20 }.set(100);

    // 但是以下3条语句都是错误的。
    // 在C++14标准中不允许对临时对象做取地址操作，也不允许修改该临时对象的成员值
    Test *p = &Test{ 10, 20 };
    Test{ 10, 20 }.a++;
    Test().a = 0;
}
```

代码清单14-7中可以看出，C++14中的临时对象其实是一个右值，所以不能对它做取地址操作，也不能做自增自减等修改操作。但是C++因为有成员方法这个特性，所以可以通过成员方法来修改其成员属性的值。因此，从这个角度上来看，C++中所谓的右值并不是严格意义上不可修改的。然后，我们再看看C11标准中的对复合字面量的对待情况，见代码清单14-8。

代码清单14-8　C11语言中的复合字面量

```
int main(int argc, const char * argv[])
{
    // 对匿名数组做取地址操作毫无问题
    int (*p)[3] = &(int[]) { 1, 2, 3 };

    // 对匿名数组做元素修改也毫无问题
    (int[]) { 1, 2, 3 }[1]++;

    struct Test { int a, b; };

    // 对匿名结构体做成员修改毫无问题
```

```
    (struct Test) { 10, 20 }.a++;

    // 对匿名结构体做取地址操作也毫无问题
    struct Test *t = &(struct Test) { 10, 20 };
}
```

我们对比代码清单14-7与代码清单14-8可以清晰地看出，对待临时匿名对象，C++与C语言的方式是截然不同的。笔者之所以在此花费大量笔墨描述C语言与C++语言对于右值概念的区别，主要原因就是笔者在工作时以及在一些技术社区经常发现许多程序员会把这两个编程语言的左值和右值概念搞混。这也是由于C++与C语言的兼容性比较强所导致的，所以不少程序员会习惯性地把C++的一些概念搬到C语言上。在大部分情况下可能没什么问题，但这两者在不少细节上还是有一些差别的。

当我们知道了“左值”这个概念之后，我们就能更清晰地了解到哪些表达式能作为赋值操作符=的左操作数以及递增、递减操作符的操作数，而哪些则不能。

14.5 C语言中表达式的求值顺序

C语言标准对程序执行的语义做了一种抽象机行为的描述，使得对于C语言生成执行代码的优化可根据自己的环境进行处理。对一个表达式的计算通常来说同时包括了对值的计算以及相应副作用的引发。对一个左值表达式的值计算包含了对它所指派对象的标识的判定。

所谓次序都是一种相对关系。我们说两个表达式哪个执行在前、哪个执行在后，都是针对这两个表达式而言的，所以这两个表达式之间就存在一种次序关系。C语言中有4种次序关系：执行在前的次序关系（sequenced before）、执行在后的次序关系（sequenced after）、无执行先后的次序关系（unsequenced）、不确定的次序关系（indeterminately sequenced）。而当我们要判定两个表达式的计算执行顺序的时候需要一个参考点，以这个位置点来判定这两个表达式的执行先后次序关系，那么这个点就称为顺序点（sequence point）。如果表达式A与表达式B具有前后次序关系，并且A的计算发生在B的之前，那么在这两个表达式之间就存在一个顺序点，在这个点处，与A相关的计算和副作用发生在与B相关的计算和副作用之前。

1）执行在前的次序关系是在同一线程中所执行的两个计算之间的一个非对称、可传递的二元关系，它在两个计算之间导出一个偏序。给定两个计算A和B，如果A的次序在B之前，那么A的执行应该在B的执行之前发生。比如：a=b+c；这条语句中有两个计算，一个是对b+c的计算，还有一

个是对a的赋值计算。这里显然是先执行b+c的计算，然后再执行对a的赋值计算，因此表达式b+c与表达式a的关系为执行在前的次序关系，这里的顺序点为=操作符。这意味着，在对a的赋值操作之前，表达式b+c的值计算必须先被计算完成。

2）执行在后的次序关系则是执行在前的次序关系的逆关系。如果计算A的次序在B之后，那么A的执行应该在B的执行之后发生。比如：a=b+array[10]；，我们这里仅观察b+array[10]表达式。这里，表达式b+array[10]的计算发生在array[10]之后，所以对于整个表达式而言，b+array[10]的执行次序在其子表达式array[10]之后，并且这里的顺序点为+操作符。

3）无执行先后的次序关系是指如果计算A的次序既不在B之前发生，也不在B之后发生。如果两个表达式的计算是无执行先后的次序关系，那么这两个表达式的计算可以交错执行，甚至并行执行。比如：a=（a+b）+（c+d）；。这里，表达式（a+b）与表达式（c+d）是无执行先后的次序关系。如果处理器支持指令级的并行执行的话，这两条表达式可并行执行。

4）不确定的次序关系是指在表达式A与表达式B之间不能确定A发生在B之前还是之后，但A一定要么发生在B之前，要么发生在B之后，两者不可交错执行。比如，在函数标志（function designator）与实参的计算之后，但在实际调用之前存在一个顺序点。在函数标志表达式（即表示当前函数）中的每个计算（包括对其他函数的调用）如果没有特别指明是在被

调函数体的执行之前还是之后执行，那么相对于被调函数的执行来说，它与这些计算是不确定的次序关系。我们下面将通过代码清单14-9来举一个比较复杂的例子来说明不确定的次序关系。

代码清单14-9　不确定的次序关系

```
#include <stdio.h>

static int fun1(int a, int b)
{
    puts("fun1 is called!");
    return 1;
}

static int fun2(int a, int b)
{
    puts("fun2 is called!");
    return 2;
}

static int fun3(int a, int b)
{
    puts("fun3 is called!");
    printf("a + b = %d\n", a + b);
    return 3;
}

static int fun4(int a, int b)
{
    puts("fun 4 is called!");

    return 0;
}

int main(int argc, const char* argv[])
{
    // 这里声明了一个指向int(int, int)函数指针的数组
    int (*pFuncList[])(int, int) = {
        &fun1, &fun4, &fun2, &fun3
    };

    pFuncList[fun1(1, 2)](fun2(10, 20), fun3(-2, 10) + fun4(0, 0));
}
```

代码清单14-9中，表达式pFuncList[fun1（1，2）]就作为一个函数标志，它指明了即将调用的一个函数。而这里，顺序点就是在对表达式pFuncList[fun1（1，2）]以及fun2（10，20）、fun3（-2，10）+fun4（0，0）的计算之后，在实际函数调用发生之前那一刻。C语言标准明确指出，

fun1、fun2、fun3、fun4可以以任一次序进行调用，在Apple LLVM 8.0编译器实现中，先调用的是fun1，然后是fun2，再是fun3，最后是fun4。为何函数调用之间的顺序点是不确定的，而不是无执行次序的呢？其实对于经典的处理器执行模型而言，一个线程中处理器同时只能处理一个分支跳转，所以当前后同时有两个分支指令或函数调用指令时，处理器必须一个一个执行，而无法同时执行，因此这里的次序关系就是不确定的次序关系，并且此次序关系可根据编译器的优化需要等上下文环境来确定。

讲完了执行次序之后，我们再来详细讨论一下C语言中的顺序点。下面描述顺序点会存在的地方。

1）在一个函数调用中函数指派符和实参的计算与函数实际调用之间（见代码清单14-9）。

2）以下操作符的第一个操作数与第二个操作数的计算之间：逻辑与（&&）、逻辑或（||）、逗号（，）。

3）在条件操作符？：的第一个操作数与第二个或第三个操作数的计算之间。比如：expr1？expr2：expr3；表达式expr1与表达式2或表达式3的计算之间存在一个顺序点。

4）在一条完整声明符的末尾。

5）在对一条完整表达式的计算与下一条要被计算的完整表达式之间。所谓完整表达式即为：不作为一个复合字面量一部分的一个初始化器；一

条表达式语句中的表达式；一条选择语句（if或switch）的控制表达式；一条while或do语句的控制表达式；一条for语句的每条可选的表达式；一条return语句中的可选表达式。它不作为一个声明符中另一个表达式的某一部分。

6）紧放在一个库函数返回之前。

7）与每个格式化的输入输出函数转换说明符相关的行为之后。

8）在对一个比较函数的每一次调用之前与之后，以及在对一个比较函数的任一次调用与作为实参传递的对象传值之间。

代码清单14-10描述了顺序点的一些例子以及相应的执行顺序。

代码清单14-10　关于顺序点的一些例子

```
#include <stdio.h>

int main(int argc, const char* argv[])
{
    int a = 10;
    // 在int a = 10;这条声明符之后有一个顺序点
    int b = 20;

    // 在表达式++a > 0与表达式b-- < 10之间有一个顺序点，
    // 并且++a > 0的执行一定在b-- < 10中任一子表达式的执行之前执行完成。
    // 同时，这里if中的完整表达式与if下面的printf函数调用之间存在顺序点
    if(++a > 0 && b-- < 100)
    {
        // 对于这条打印函数调用，在第一个a = %d的%d之后有一个顺序点，
        // 使得在打印出a = 11之后才能打印出b = 19
        printf("a = %d, b = %d\n", a, b);
    }

    // 表达式a < b与后面两个表达式之间有一个顺序点，
    // 并且表达式a < b一定先执行完成，然后再执行第二个或第三个表达式
    a = a < b? a + 1 : a - 1;

    // 这里，表达式++b + 1与++a - 1之间存在一个顺序点，
    // 并且在++ b + 1完成之前，++a - 1中的任一子表达式计算都不能开始执行
    a = (++b + 1, ++a - 1);
}
```

最后要提醒各位的是，如果在一条语句中出现对同一标量对象的多个无执行次序关系的修改，那么行为是未定义的。代码清单14-11列出了这种情况。

代码清单14-11　对同一个对象的多个无执行次序的修改

```
#include <stdio.h>

int main(int argc, const char* argv[])
{
    int array[] = { 1, 2, 3, 4 };

    int *p = array;

    int a = 0;

    // 在这条赋值表达式中，作为=操作符左操作数的p[a++]表达式中含有对对象a的修改，
    // 而右操作数表达式++a也同时对对象a进行修改，那么这将会引发未定义行为
    p[a++] = ++a;

    struct Test
    {
        int x, y;
    }t = { 0, 1 };

    // 这条语句没有问题，尽管这里的++操作都是针对同一个对象t，但t是一个结构体，
    // 不是一个标量类型，同时，++分别对t的x成员与y成员进行操作，这是两个不同的元素。
    // 因此这条语句不会发生未定义行为
    p[++t.x] = ++t.y;
}
```

另外，还有一个C语言程序员讨论得比较多的问题，即表达式x+++++y；最后产生一个什么结果的问题。其实，C语言标准已明确指出：程序片段x+++++y会被解析为x++++y，这违反了递增操作符的约束（因为x++已经不是一个左值，它不能再次作为++操作符的操作数），即便x+++++y这种解析可能产生一个正确的表达式。这段文字其实很清楚地表明了x+++++y最终会被解析成什么，这也是对词法解析器的一种约束。而对于x+++++y表达式来说，x++与++y是两个无次序关系的表达式，并且对x和对y的修改是作用在两个不同标量对象上的，因此本身没有什么问题。

14.6　C语言中的语句

以上描述的都是与表达式相关的概念。这里我们将再简单介绍一下C语言中的语句。C语言中一共含有6种语句，分别为：标签语句（labeled statement）、复合语句（compound statement）、表达式语句（expression statement）、选择语句（selection statement）、迭代语句（iteration statement）、跳转语句（jump statement）。因为语句在本书前面的各个章节中都介绍得差不多了，因此我们以代码清单14-12中的内容来对以上6种语句做个排号入座。

代码清单14-12　C语言中的语句

```
#include <stdio.h>

int main(int argc, const char * argv[])
{
    // 这是一条声明，而不是语句
    int a, b;

    // 以下两条是表达式语句
    a = 10;
    b = a * 2 + 3;

    // 这是一条复合语句
    {
        // 这三条是在一条复合语句中的表达式语句
        printf("a = %d, b = %d\n", a, b);
        a++;
        b--;
    }

    // 这是一条选择语句（从if到 } ），
    // 其中从 { 一直到 } 是该选择语句中的复合语句
    if (a > 10)
    {
        puts("Greater than ten!");

        // 这是一条在选择语句中的跳转语句
        goto HELLO;
    }

    // 这也是一条选择语句（从if一直到else后面的分号）
    if (b > 10) puts("Greater than ten!");
    else puts("Less than ten!");
```

```
    // HELLO: a++; 是一条标签语句
HELLO:
    a++;    // 这是一条标签语句中的表达式语句
    b++;    // 这条表达式语句不属于标签语句中的语句

    // 这是一条选择语句
    switch (a)
    {
        // case0: puts("zero"); 是一条选择语句中的标签语句
    case 0:
        puts("zero");   // 这是一条标签语句中的表达式语句
        break;          // 这是一条跳转语句，并且不属于标签语句中的语句

        // 整个 case 12: puts("twelve"); 是一条选择语句中的标签语句
        // puts("twelve"); 是一条标签语句中的表达式语句
    case 12: puts("twelve");
        break;          // 这是一条跳转语句，并且不属于标签语句中的语句

        // 从default一直到 } 是一条选择语句中的标签语句
        // { break; } 是一条标签语句中的复合语句
    default: {
        break;          // 这是一条复合语句中的跳转语句
        }
    }

    // 这是一条迭代语句，a--； 是迭代语句中的表达式语句
    while (a > 0) a--;

    // 从do一直到分号是一条迭代语句，
    // 其中从 { 一直到 } 是该迭代语句中的复合语句
    do
    {
        b--;
    } while (b > 0);

    // 这是一条迭代语句，int i = 0 是其中的一条声明，
    // i < 5 以及 i++ 则属于表达式，
    // a++, b++; 是该迭代语句中的表达式语句
    for (int i = 0; i < 5; i++) a++, b++;

    // 从for一直到 } 是一条迭代语句
    for (a = 0; a < 5; a++)
    {
        // 从if一直到分号是迭代语句中的选择语句
        if (a == 3)
            continue;   // 这是选择语句中的一条跳转语句

        b--;    // 这是迭代语句中的一条表达式语句
    }

    ;       // 这是一条空语句

    { }     // 这是一条不包含任何语句的复合语句，注意，复合语句后面无需加分号

    return 0;   // 这是一条跳转语句
}
```

14.7 本章小结

本章首先对C语言的表达式做了系统的介绍，同时详细介绍了泛型选择表达式以及常量表达式。然后补完了C11标准中新引入的静态断言语句。

之后，我们提到了C语言中关于左值的概念，最后讲述了C语言中表达式的求值顺序以及顺序点等相关概念。

通过对本章的学习，各位能对C语言具体实现中的一些细节更深入地了解，同时也能感受到C语言标准所体现出的灵活性，而又不缺乏约束性。C语言标准在某种程度上给了C语言具体实现可针对当前运行的处理器以及操作系统环境做特定的执行优化。

第15章　函数调用约定与ABI

我们在第1章中提到过，C语言的一大优点是我们即便把源代码编译好，打包成库，我们在另一个项目中仍然可以去连接静态库或动态库中的全局外部对象以及全局外部函数。这样，我们就可以将静态连接库或动态连接库作为插件或中间件交给其他开发者来使用了。比如像我们所使用的标准库函数都是以静态库或动态库的方式做默认连接的。那么这里就会存在一个问题，我在调用库中外部全局函数的时候究竟是如何调用成功的呢？毕竟库中的上下文与当前项目中的上下文可能会不太一样。为了解决这一问题，很多操作系统给出了针对当前系统环境下的函数调用约定（Function Calling Convention）。有了函数调用约定，我们就可以在某个操作系统上使用多种不同的C语言编译器了。尽管很多C语言编译器自成一派，比如MSVC编译器与GCC就完全不同，但只要它们都能遵守同一函数调用约定，那么生成出来的目标文件也能正确地相互连接。

ABI在之前章节中也有所提及，它的英文全称是Application Binary Interface，即应用二进制接口。它是在整个操作系统中对二进制接口规范的详细描述，不仅包含了函数调用约定，而且还规定了数据类型的对齐规则、布局、大小等；还规定了应用程序如何做系统调用；另外还有目标文件、程序库的二进制格式，使得跨编译器的连接成为可能，同时操作系统的加载器也能成功地加载由不同连接器最终所构成的可执行文件，并进行

执行。

由于ABI详细规则与各个操作系统紧密相关，并且规范本身也不是太简单，因此我们下面将主要简单介绍Windows操作系统以及Unix/Linux操作系统下的函数调用约定。至于ABI的规范文档，各位可以在网上搜索到相关资料，像大部分Unix/Linux系统在64位模式的x86处理器中所用的ABI遵循的是System-V规范，各位能够下载到。

15.1 Windows操作系统环境下x86处理器的函数调用约定

Windows操作系统中一般x86处理器用得比较多，因此我们这里也针对x86处理器做详细描述。从AMD推出了64位处理器开始起，x86处理器也能支持64位模式了。由于32位模式与64位执行模式在寄存器使用等方面会有所不同，所以操作系统为了能更大效率地支持64位模式程序的执行，其函数调用约定也会与32位模式下的有所不同。下面将分别介绍32位执行模式与64位执行模式下的Windows操作系统中的函数调用约定。我们一般对Windows下MSVC或现在新引入的VS-Clang编译器所使用的ABI简称为MS-ABI。以下两节中的代码既可以用MSVC编译器构建，也可以用VS-Clang编译器构建，两者都默认采用MS-ABI的函数调用约定，并且相互兼容。

15.1.1 Windows操作系统下32位x86执行模式的函数调用约定

Windows系统中，x86在32位执行模式下的函数调用约定有3种：C函数调用约定，标准调用约定，快速调用约定。在所有这3种调用约定下，所有参数的位宽都会被扩展到32位。比如，我们要传一个short类型的对象，那么在实际压栈的时候，就会对它进行带符号扩展到32位，因此在函数实现中，该形参尽管在类型上仍然是short类型，但获取数据的时候其实可以用

int类型来获取，当然我们不建议这么做。而对于返回值，如果要返回的对象少于4个字节，那么它会被扩展到4字节，然后放入EAX寄存器中进行返回。如果是一个64位数据或一个8字节的结构体对象，那么实现会将该8字节对象以EDX：EAX寄存器对进行存放。其中，EDX寄存器存放高4字节，EAX存放低4字节。如果要返回的对象大于8字节，那么就会将该数据的起始地址放入EAX寄存器中，然后在函数返回之后通过EAX作为基地址将该结构体对象完整地拷贝出来。在函数实现中，如果我们需要使用EBX、ESI、EDI以及EBP寄存器的话，那么需要先将它们压栈保护，等函数返回之前则推出堆栈恢复之前的值。

在默认情况下，如果我们不指明，在C语言中用的就是C函数调用约定了。我们下面来分别介绍这三种调用约定。

C函数调用约定参照了C语言函数具有不定参数个数的函数声明特性，所以该调用约定是将传递给函数的参数全都压入栈中，并且以从右到左的顺序依次将参数压入栈中，也就是说最后一个参数先压入栈中，第一个参数最后被压入栈中。在MSVC编译器中，默认使用C函数调用，如果我们要显式地指明遵循C函数调用约定的函数时，我们可以对该函数用__cdecl调用约定限定符进行修饰，此关键字是MSVC编译器对标准C语言的扩展，不属于标准C语言的范畴。采用C函数调用约定时，函数调用者负责将压入堆栈的实参清除掉，即把栈指针恢复到传参之前的位置。

当需要指定一个函数做标准调用约定时，我们在声明函数的时候使用__stdcall调用约定限定符对该函数标识符进行修饰。标准函数调用约定的

参数传递与C调用约定类似，在参数传递上也是将参数全都压入栈中，不过有所区别的是，标准调用约定是由函数实现自己清理传入函数的形参所占的栈空间。由于x86中的RET函数返回指令可直接带操作数，指示将栈指针往上加多少字节，使得栈指针在函数返回的同时就能立即恢复到传参之前的位置，这也会节省一部分运行时开销。

当我们需要指定一个函数做快速调用约定时，我们在声明函数的时候使用__fastcall调用约定限定符。快速调用约定是先尝试将前两个参数依次放入ECX寄存器与EDX寄存器（如果前两个参数的字节长度小于等于4），然后将其余参数仍然按照从右到左的次序压入堆栈。快速调用与__stdcall调用约定一样，当函数返回之前，函数实现必须自己将之前传入参数所占用的栈空间做清除操作，而不是给函数调用者去做。

下面我们将为大家介绍如何通过Visual Studio 2017 Community来实验函数调用约定。首先，我们根据第3章提到的内容，创建一个Win32控制台的空项目，项目名为demo。然后在该项目工程中新建main.c和func.asm两个源文件。新建func.asm时，我们也是选择“C++源文件”，然后对源文件名命名为func.asm即可。下面我们要对项目本身进行设置，使得它能支持汇编语言的编译和连接。我们鼠标右键点击cdemo项目名，然后在下拉栏中找到“生成依赖项”，然后点击“生成自定义”，如图15-1所示。

图15-1　设置编译汇编源文件第一步

然后我们会看到一个设置“生成自定义项文件”的对话框，我们勾选上“masm”即可，如图15-2所示。

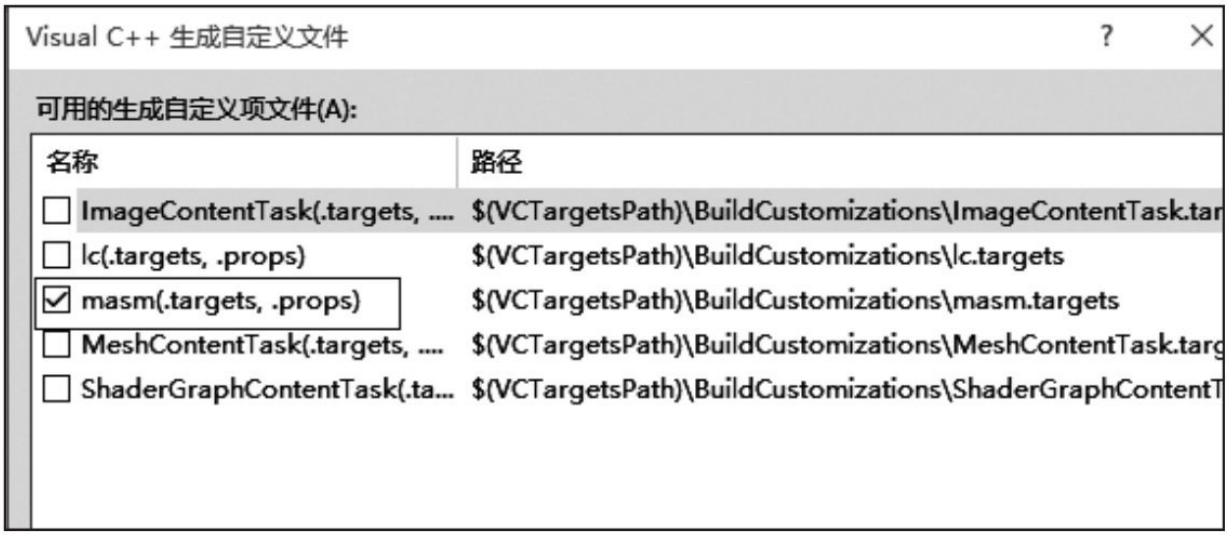


图15-2　设置编译汇编源文件第二步

下面我们先看看main.c源文件中的内容，见代码清单15-1。

代码清单15-1　MSVC在x86处理器32位环境下的函数调用约定C源文件

```
#include <stdio.h>
#include <stdint.h>
#include <stdbool.h>

// 计算(a - b)
extern int __cdecl CFunc(int a, int b);

extern uint64_t CFunc2(void);

struct Test
{
    int a;
    int b;
    int c;
    int d;
};

extern struct Test CFunc3(void);

// 计算(a - b) - (c - d)
extern int __fastcall FastFunc(int a, int b, int c, int d);

int main(void)
{
    int value = CFunc(5, 2);
    printf("The C value is: %d\n", value);

    uint64_t llValue = CFunc2();
    printf("The long value is: 0x%16llx\n", llValue);

    value = FastFunc(6, 2, 3, 1);
    printf("The fast value is: %d\n", value);

    struct Test test = CFunc3();
    printf("a = %d, b = %d, c = %d, d = %d\n",
            test.a, test.b, test.c, test.d);
}
```

代码清单15-1中我们声明了4个函数，分别是CFunc1、CFunc2、CFunc3和FastFunc。CFunc1用于观察传递两个32位整型参数时，采用C调用约定的参数传递情况；CFunc2则是用于观察返回值为8字节时函数所返回的情况；CFunc3则观察当返回值是一个较大结构体对象时，函数返回情况；FastFunc则是用于观察调用一个快速调用约定的函数时，给它传递4个

参数的情况。为了便于观察，CFunc1的汇编实现用的是返回第1个参数减去第2个参数的结果；FastFunc的实现是先用第1个参数减去第2个参数结果，再减去第3个参数与第4个参数的结果，然后将最终值返回出来。

下面我们看一下汇编源文件。

代码清单15-2　MSVC在x86处理器32位环境下的函数调用约定汇编源文件

```
; 汇编源文件func.asm

.model flat
.code

_CFunc   proc public

    mov     eax, [esp + 4]    ; EAX存放第一个参数
    mov     ecx, [esp + 8]    ; ECX存放第二个参数
    sub     eax, ecx
    ret

_CFunc   endp


_CFunc2     proc public

    mov     edx, 12345678H    ; 存放高4字节
    mov     eax, 90abcdefH    ; 存放低4字节
    ret

_CFunc2     endp


_CFunc3     proc public

    push    40          ; 给成员d赋值
    push    30          ; 给成员c赋值
    push    20          ; 给成员b赋值
    push    10          ; 给成员a赋值
    mov     eax, esp    ; 将当前栈指针赋给EAX作为所返回结构体对象的起始地址
    add     esp, 16     ; 将栈指针恢复到返回地址处
    ret

_CFunc3     endp


@FastFunc@16   proc public

    mov     eax, ecx        ; 获取第一个参数值，传给EAX
    sub     eax, edx        ; 将第一个参数值与第二个参数值相减，结果再存到EAX
    mov     ecx, [esp + 4]  ; 读取第三个参数，存放到ECX
    mov     edx, [esp + 8]  ; 读取第四个参数，存放到EDX
    sub     ecx, edx        ; 将第三个参数减去第四个参数的值存放回ECX
```

```
    sub     eax, ecx        ; 将第一个差值与第二个差值再相减，存放到EAX返回

    ; 这里用ret 8是将压栈的两个参数在函数返回后直接推出栈
    ; 相当于:  ret;    add esp, 8;
    ret     8

@FastFunc@16   endp

end
```

通过代码清单15-2，我们看到，CFunc2最终返回0x12345678_90abcdef。而CFunc3最终所返回的结构体的成员依次是10、20、30、40。我们注意到，FastFunc子过程最后用ret 8表示函数返回后，ESP自动加8，以回收压入栈中作为形参的栈空间。

下面通过图15-3来描述CFunc1的参数传递时栈空间的数据存放情况。

①从图15-3a中可以看到，一开始在调用CFunc之前假定此时ESP栈指针指向0x010C的位置，由于当前栈指针所指的栈空间地址属于函数调用者的上下文，所以其数据不用动，用X表示。其余数据用？表示未知数据。

②在调用CFunc1之前，先传递参数，并且是以从右到左的顺序传递，先传右边参数，即第二个参数，我们发现此时ESP到了0x108的位置，并且0x108到0x10B存放的是32位带符号整数2（见图15-3b）。

③如图15-3c所示，传入左边参数，即第一个参数，此时栈指针ESP再向下减4，到了0x0104的位置，这里0x0104到0x0107存放的就是第一个参数的数据，即32位带符号整数5。

④做CFunc1函数的调用（见图15-3d），执行CALL指令之后，会自动将当前CALL指令的下一条指令的地址压入栈中，此时ESP也移动到了

0x100的位置。因此我们访问[ESP+4]就是访问第一个参数的数据，而访问[ESP+8]就是访问第二个参数的数据。

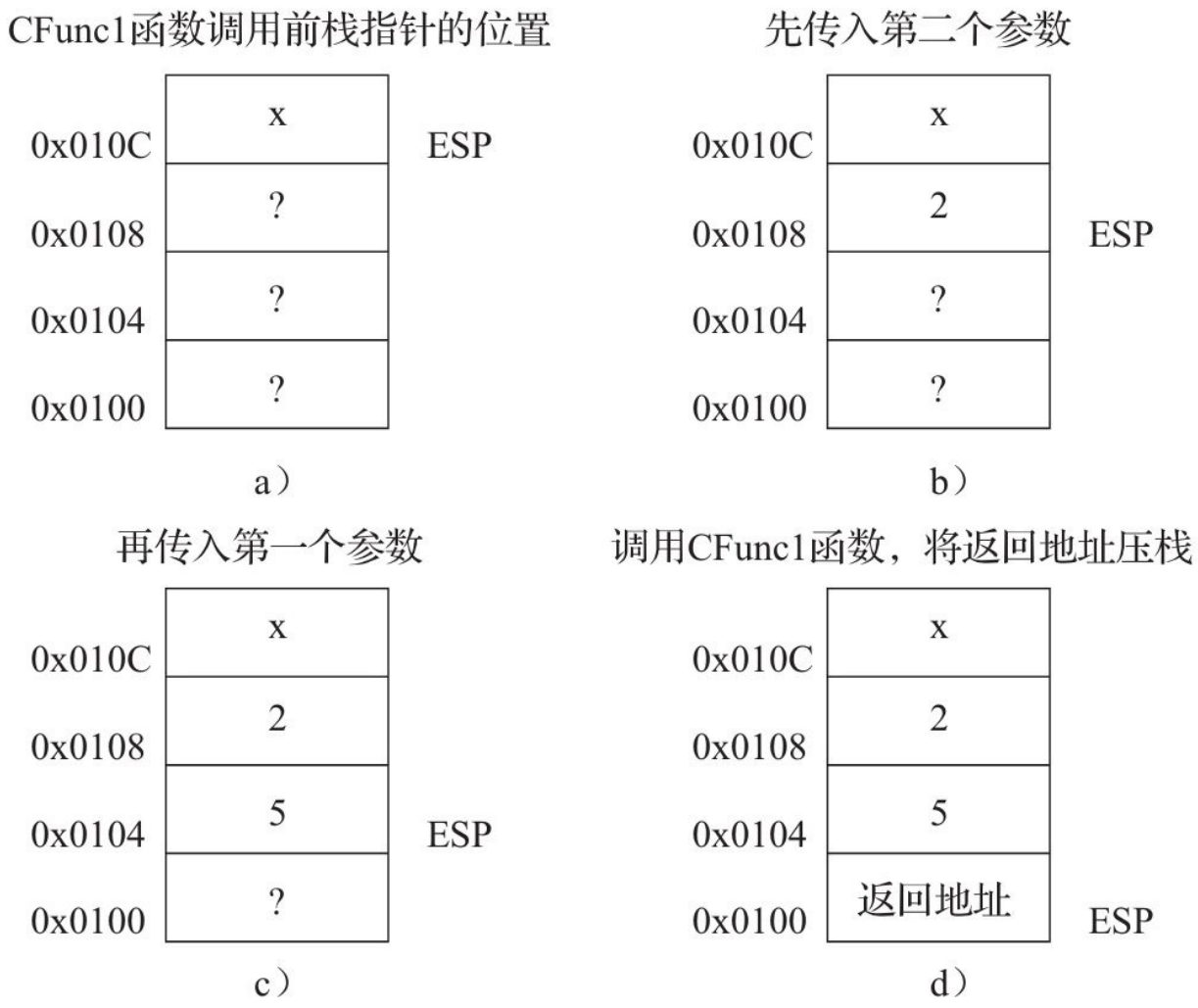


图15-3　CFunc1调用前后的参数传递及栈指针的变化

图15-4描述了调用CFunc3之后，结构体成员与栈的对应关系以及栈指针的变化，最后函数返回前后需要做的工作。

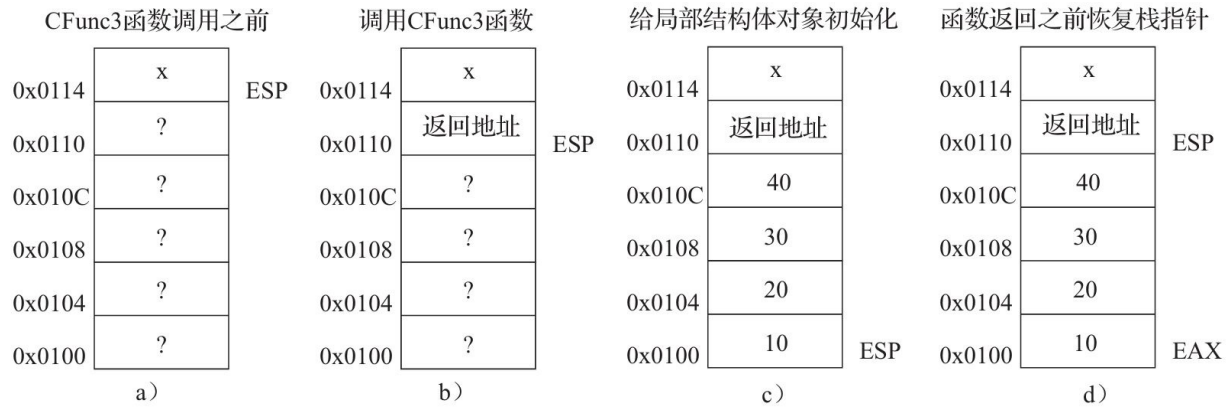


图15-4　CFunc3函数调用前后栈数据以及栈指针变化

①由于我们在调用CFunc3的时候，它没有参数，所以不需要先做压栈操作。我们这里也是用？表示此时栈空间中的数据见图15-4a。

②调用CFunc3之后，x86处理器会先将当前CALL指令的下一条指令的地址压入栈中，然后ESP到0x0110的位置（见图15-4b）。

③在CFunc3内，对要返回的结构体对象进行初始化，其成员a到d依次被赋值为10、20、30和40，此时ESP会在0x0100的位置（见图15-4c）。

④在函数返回之前，我们先用EAX寄存器记录当前ESP的位置，这样EAX就会作为指向该结构体对象的指针被返回出去了。

然后我们将ESP恢复到之前保存返回地址的那个位置，执行返回指令（见图15-4d）。

⑤在函数调用端，编译器实现将会先获取返回出来的EAX的值，然后将EAX所指向的结构体对象拷贝到其栈上下文中。

这个过程是立即实现的，因为我们在图15-4中也看到了，返回的结构体对象的实体其实已经被回收了，所以在做其他函数调用或需要使用栈空间的动作之前，必须将此对象的值给拷贝出来。当然，如果我们仅仅调用CFunc3函数，而不是通过=赋值操作符将其返回值赋值给调用者的结构体对象，那么CFunc3中的临时结构体对象也不需要被拷贝出来。

15.1.2 Windows操作系统下64位x86执行模式的函数调用约定

x86处理器在64位模式下，通用寄存器的位宽不仅增加到了64位，而且可用的通用寄存器的数量也增长了1倍。表15-1中列出了x86处理器在32位与64位模式下所有可作为通用目的计算使用的寄存器。

表15-1 x86处理器在32位与64位执行模式下所有可作为通用目的寄存器列表

寄存器类型	32 位执行模式	64 位执行模式
8 位寄存器	AL、BL、CL、DL、AH、BH、CH、DH	AL、BL、CL、DL、DIL、SIL、BPL、SPL、R8L - R15L
16 位寄存器	AX、BX、CX、DX、DI、SI、BP、SP	AX、BX、CX、DX、DI、SI、BP、SP、R8W - R15W
32 位寄存器	EAX、EBX、ECD、EDX、EDI、ESI、EBP、ESP	EAX、EBX、ECD、EDX、EDI、ESI、EBP、ESP、R8D - R15D
64 位寄存器	不可用	RAX、RBX、RCX、RDX、RDI、RSI、RBP、RSP、R8 - R15

表15-1中，SP、ESP以及RSP是作为栈指针寄存器使用，一般只能用它做栈指针操作，而不能做其他通用目的用途。

在Windows系统的函数调用约定中，x86处理器在64位模式下，函数实现需要自己保存RBX、RDI、RSI、RBP、R12、R13、R14与R15这些通用目的寄存器，如果在当前函数中用了这些寄存器的话。而其他通用目的寄存器都由函数调用者自己维护。另外，所有SIMD寄存器（MMX、XMM、YMM与ZMM寄存器）也都由函数调用者自己维护，函数实现直接使用即可。

对于参数传递，64位模式下就没32位模式那么复杂了，它只有一种调用方式。由于有充足的寄存器用来存放参数，所以MS-ABI规定，x86处理器在64位模式下，对于前4个整数参数，依次放入RCX、RDX、R8与R9寄存器；之后的参数都是以从右到左的次序依次压入栈中。这里需要注意的是，即便前面4个参数都没有实际压入栈空间，但函数调用者一般会为它们保留对应的在调用函数中的栈位置，这样看上去就仿佛这些寄存器也被压入了调用函数的栈空间一样。因此我们在函数实现中要访问第5个参数，其实也需要将RSP栈指针加40进行访问；而第6个参数则需要将RSP加48进行访问。图15-5展示了将在代码清单15-3中所声明的函数MyFunc2（int a，int b，int c，int d，int16_t e，int16_t f）的函数调用前后栈指针的变化。

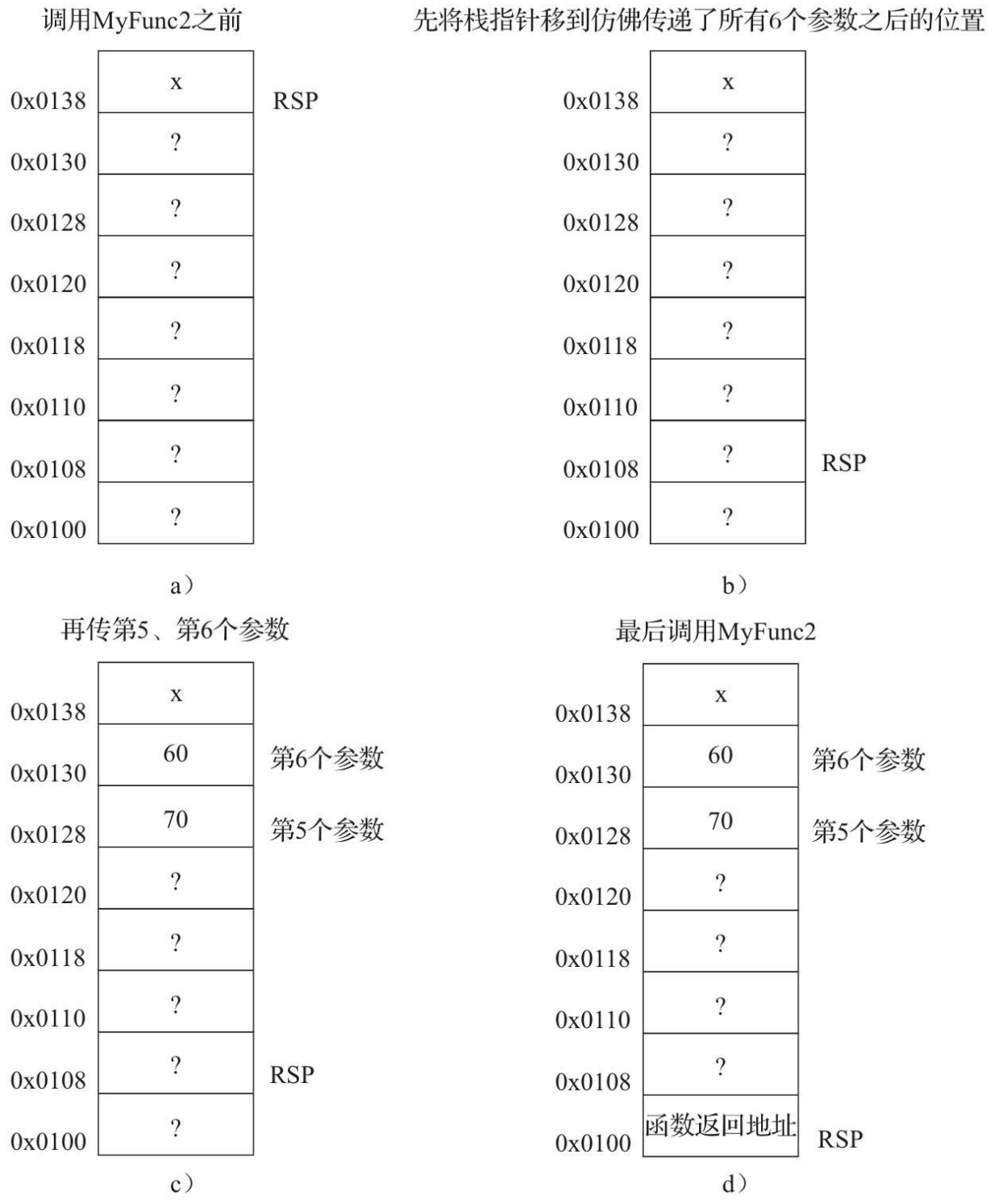


图15-5　Windows系统64位模式下调用MyFunc2前后的栈空间变化

从图15-5我们能清楚看到，当我们调用MyFunc2函数之前，函数调用

者会做不少工作。先把栈指针往下移到正好能放入6个实参的位置，随后将第5、第6个实参放入到MyFunc2栈空间相应的栈空间位置作为其形参，最后调用MyFunc2函数。

下面，我们在展示代码之前先教大家如何在Visual Studio 2017 Community下生成64位程序的步骤。与之前一样创建一个Win32控制台应用程序的空项目。然后右击源文件文件夹，选择新建项，新建main.c与func.asm两个源文件并添加到项目工程中。然后，与图15-1和图15-2一样，添加MASM的编译选项。由于asm文件因为版本不同，可能默认不作为汇编源文件参与编译，所以需要为它添加编译项类型。我们先右键点击func.asm，如图15-6所示：

然后选择“属性”，进入func.asm文件的设置，如图15-7所示。

图15-6　右键点击func.asm

图15-7　设置func.asm的Item Type

一开始，func.asm的“项类型”是“不参与生成”，现在我们点击右边的三角箭头，然后选中“Microsoft Macro Assembler”，这样此文件能用MASM汇编器进行编译了。

 注意：为了使该配置同时适用于调试模式与发布模式，我们在左上角选择“所有配置”，同时注意当前的平台是x64，因为我们要测试的程序是64位应用程序。如果我们之前没有设置当前活动平台为x64，那么可以根据图15-8进行设置。在工具栏中“Debug”那个选择控件右边有一个原本默认显示“x86”的选择框，现在将它选择为“x64”。这个选择框就是用于指定解决方案平台的。这样我们编译生成的应用就是64位应用了。

图15-8　将解决方案平台设置为64位模式，以生成64位应用

这些设置完成之后，我们就来看代码清单15-3。由于64位模式下函数调用约定就一种，所以这里就列出了两个函数的例子，分别为MyFunc1与MyFunc2。

代码清单15-3　Windows系统x86处理64位模式下的函数调用约定C源文件

```
// main.c源文件
#include <stdio.h>
#include <stdint.h>

// 执行(a - b) / (c - d)操作
extern int MyFunc1(int64_t a, int8_t b, int32_t c, int16_t d);

// 执行(a + b + c + d) / (e - f)操作
extern int MyFunc2(int a, int b, int c, int d, int16_t e, int16_t f);

int main(void)
{
    int result = MyFunc1(12, 4, 3, -1);
    printf("MyFunc1 division result: %d\n", result);

    result = MyFunc2(10, 20, 30, 40, 70, 60);
    printf("MyFunc2 division result: %d\n", result);

    puts("\nProgram completed!");

    getchar();
}
```

代码清单15-3所列出的代码很简单，这里不多做介绍。不过这里要提醒各位的是，请留意一下这两个函数的每个参数类型，后面在汇编源文件中会做相关处理。Windows系统x86处理64位模式下的函数调用约定汇编源文件如代码清单15-4所示。该源文件列出了MyFunc1与MyFunc2的具体实现。

代码清单15-4　Windows系统x86处理64位模式下的函数调用约定汇编源文件

```
; func.asm汇编源文件
.code

MyFunc1     proc public

    ; 由于第二个参数是8位整数，第四个参数是16位整数，因此需要将它们做带符号扩展，
    ; 否则，高位可能存在其他非零数据，会影响后续计算
    movsx   rdx, dl
    movsx   r9, r9w

    sub     rcx, rdx    ; 将第一个参数与第二个参数相减，结果存放到RCX寄存器
    mov     rax, rcx
    xor     rdx, rdx    ; 将RDX寄存器清零，它将作为被除数的高64位

    sub     r8, r9      ; 将第三个参数与第四个参数相减，结果存放到R8寄存器作为除数
    idiv    r8

    ret
```

```
MyFunc1     endp


MyFunc2     proc public

    add     rcx, rdx    ; 将第一个参数与第二个参数相加，结果存入RCX寄存器
    add     r8, r9      ; 将第三个参数与第四个参数相加，结果存入RDX寄存器
    add     rcx, r8     ; 将两个求和结果再次求和，结果存入RCX寄存器
    mov     rax, rcx    ; 将结果移入RAX寄存器，作为被除数的低位部分
    xor     rdx, rdx    ; 将RDX寄存器清零，它将作为被除数的高位部分

    mov     rcx, [rsp + 40]     ; 获取第5个参数
    mov     r8, [rsp + 48]      ; 获取第6个参数

    ; 由于最后两个参数都是无符号16位整数，因此将这两个参数做清零高位扩展
    ; 使得高48位比特都为0
    movzx   rcx, cx
    movzx   r8, r8w

    sub     rcx, r8
    idiv    rcx

    ret

MyFunc2     endp

end
```

这里大家可以看到Windows系统在x86处理器64位模式下参数传递情况。此外，压入函数的参数也是由函数调用者自己清理，不需要由函数实现来处理。

最后笔者对于函数调用约定中谁负责清理传入参数的栈空间这个问题上发表一些个人看法。各位感兴趣的也可以与笔者联系大家共同探讨。尽管像x86处理器在32位下的__stdcall与__fastcall函数调用约定中由函数实现负责清理传入参数所占的栈空间，从而会节省一些运行时开销。但其实从工程学角度来看，这是一个不对称行为。这就相当于函数调用者分配的空间，但由函数实现去回收，所以在64位模式的函数调用约定中，采用的是类似__cdecl函数调用约定的做法。此外，像Apple在对象引用计数管理方面也是做了这么一个规定：是你分配的就由你去释放；不是由你分配的，

则不用你去释放。

15.2 Unix/Linux操作系统环境下x86处理器的函数调用约定

Unix/Linux操作系统环境下，x86处理器在32位执行模式下的C函数调用约定与Windows操作系统上的相差不多。只不过在Unix/Linux系统上规定，在函数调用处，栈指针所指的栈空间地址必须是16字节对齐的。此外，对于各种数据类型的对齐也做了相关规定，在32位模式下，单字节整数以1个字节对齐；双字节整数以2个字节对齐；其他所有整数与浮点类型都是4字节对齐的，除了long double是16字节对齐之外。

在Unix系操作系统中，通常使用GCC或Clang作为编译器，这些编译器也提供了x86处理器在32位模式下的三种函数调用约定，只不过函数调用约定限定符用的是__attribute__来声明的（在第17章中做详细介绍）。默认的函数调用约定仍然是cdecl，即C函数调用约定，如果我们通过显式声明的话，可以用诸如void__attribute__（（cdecl））foo（void）；来声明。此外，stdcall与fastcall也一样，分别通过__attribute__（（stdcall））与__attribute__（（fastcall））来声明。

64位执行模式下则与Windows的完全不同了。Unix系操作系统在x86处理器64位执行模式下遵循的是针对AMD64架构的System-V（这里的V表示罗马数字5，不是英文字母）ABI。在64位模式下，long以及long long类型都是64位的，并且它们也要求以8字节对齐。函数实现需要自己保存的通用

目的寄存器有RBX、RBP、R12、R13、R14、R15，其他通用寄存器都由函数调用者去维护。而在函数调用上，System-V ABI则能将更多的整数参数通过通用目的寄存器做参数传递，从左到右依次放入RDI、RSI、RDX、RCX、R8与R9。

下面举一个与代码清单15-3相同的例子来看看macOS下使用64位执行程序的函数调用约定使用情况。在macOS下，我们使用Xcode，用默认选项配置即可，十分方便。我们还是在生成的工程中使用main.c源文件，汇编源文件用func.s，在GCC与Clang编译器中，汇编源文件通常用.s作为后缀名的文件。用Xcode新建汇编文件非常方便，直接在macOS一栏中将滚动条拉动到最下面找到Other一栏，然后选中“Assembly File”即可，如图15-9所示。

图15-9　用Xcode新建汇编源文件

代码清单15-5给出与15-3相应的汇编源文件。

代码清单15-5　macOS下实现与代码清单15-4相同效果的汇编代码

```
//  func.s

.text
.align 4

.globl _MyFunc1, _MyFunc2

_MyFunc1:

    // 将第二个参数与第四个参数做带符号扩展
    movsx   %si, %rsi
    movsx   %cx, %rcx

    sub     %rsi, %rdi  // 第一个参数减去第二个参数，差存入RDI
    sub     %rcx, %rdx  // 第三个参数减去第四个参数，差存入RDX
    mov     %rdi, %rax  // 将第一个差值放入RAX作为被除数
    mov     %rdx, %rcx  // 将第二个差值放入RCX作为除数
    xor     %rdx, %rdx  // 清空RDX寄存器，作为被除数高位
    idiv    %rcx

    ret


_MyFunc2:

    add     %rsi, %rdi  // 将第一个参数与第二个参数相加，结果存入RDI
    add     %rcx, %rdx  // 将第三个参数与第四个参数相加，结果存入RDX
    add     %rdx, %rdi  // 将两个求和结果相加，结果存入RDI寄存器

    movzx   %r8w, %r8   // 无符号扩展第五个参数
    movzx   %r9w, %r9   // 无符号扩展第六个参数
    sub     %r9, %r8    // 将第5个参数减去第6个参数的值，存入R8作为除数

    mov     %rdi, %rax  // 将求和结果放入RAX作为被除数
    xor     %rdx, %rdx  // 清空RDX寄存器，作为被除数高位
    idiv    %r8

    ret
```

在Linux环境下（包括Android编译环境下），GAS汇编器现在已经不要求C语言的外部函数符号在汇编代码中需要显式加上前导_符号，汇编器会默认加上。因此上述汇编代码如果拿到Linux或Android上编译的话，我们必须将_MyFunc1和_MyFunc2前面的下划线给去掉。

15.3 ARM处理器环境下的函数调用约定

由于ARM处理器由ARM官方定义了其函数调用约定，所以对于各类嵌入式系统以及操作系统，基本都按照ARM官方制定的调用约定进行实现，当然有些系统会在此基础上做一些扩展。尽管大部分ARM处理器都是32位的，不过现在ARMv8架构处理器也有32位执行模式与64位执行模式，ARM官方分别将它们称为AArch32与AArch64。下面我们分别对AArch32与AArch64的函数调用约定进行介绍。

15.3.1 AArch32架构环境下的函数调用约定

在AArch32架构体系下，ARM处理器可访问16个通用目的寄存器R0～R15，其中R13作为栈指针寄存器（SP），R14作为连接寄存器（LR）用于存放函数返回地址，R15作为程序计数寄存器（PC）。这里，R14通常也能作为通用目的寄存器来使用，所以我们一般能用R0～R12以及R14这14个通用目的寄存器。这里需要注意的是R9寄存器，R9寄存器在不同环境所扮演的角色可能会不太一样。在某些嵌入式平台上，如果将代码编译为地址无关的（position-independent），那么R9将作为静态基地址寄存器而使用，这时当我们在函数中要访问全局数据对象的时候就需要用R9加上此数据对象所在的偏移来定位真正的物理地址，像ADS1.2、RVCT4.0这些编译器编译出来的地址无关的代码都会将R9作为静态基地址寄存器使用，所以

我们不能在代码中自己使用R9寄存器。此外，R9寄存器也有可能被用于线程寄存器，比如用于访问C11标准中的_Thread_local对象，所以各位在使用R9寄存器的时候需要注意各个平台上的相关文档，以免误用而引发程序异常、崩溃的现象。在iOS、Android系统中，R9寄存器都没有特殊用途，各位可以放心将它视为通用目的寄存器来使用。

在AArch32架构环境下，需要函数实现自己保存的寄存器有R4～R11以及R14。R12寄存器可用作过程间调用的共享寄存器。比如，我在子过程A中需要传递一个值给子过程B使用，那么可以将这个值保存在R11中，子过程B即可直接通过访问寄存器R11而获取到。所谓子过程（Procedure或Routine）就是我们用汇编写的函数实现，它与C函数的区别不仅仅是在语言层面上，而且在实现上更为灵活，能自己制定各类数据的传递方式以及跳转方式。C函数所生成的代码会有更多的规定与约束。

在AArch32架构环境下，函数调用前的参数传递规则是：对于前4个不超过32位的整数参数，从左到右依次放入R0～R3寄存器，后续参数根据特定环境从右到左依次压入栈中，不同环境对函数调用前的栈指针地址的对齐要求可能会不同，但一般在类Unix系操作系统中，都要求8字节或16字节对齐。

下面我们还是以代码清单15-3的C代码作为样本来做AArch32架构下iOS系统环境的函数调用约定的测试。用iOS来做函数调用约定的测试十分方便，我们用Xcode创建一个iOS的Single View Application项目模板，然后在工程里添加一个func.s的汇编源文件即可。在iOS项目工程中添加func.s的

时候与macOS工程类似，在iOS一栏里的Others分类能找到Assembly File，选择它，然后将文件名命名为func即可。随后，我们可以直接在ViewController.m里做测试，如代码清单15-6所示。由于ViewController.m是一个Objective-C源文件，所以是以.m作为后缀名结尾的，但各位不用担心，Objective-C完全兼容C语言，它是货真价实的C语言的超类，这与C++对C语言做兼容还不一样。

代码清单15-6　iOS工程中的ViewController.m源文件代码

```
//  ViewController.m
//  arm_test

#import "ViewController.h"

@interface ViewController ()

@end

/**
 * 由于AArch32架构下，ARM-A的指令集中没有整数除法指令，
 * 所以这里我们自己定义一个全局外部函数来定义一个通用的除法函数
*/
int MyDivide(int dividend, int divisor)
{
    return dividend / divisor;
}

// 执行(a - b) / (c - d)操作
// size_t在32位执行模式下是32位，64位执行模式下为64位
extern int MyFunc1(size_t a, int8_t b, int32_t c, int16_t d);

// 执行(a + b + c + d) / (e - f)操作
extern int MyFunc2(int a, int b, int c, int d, int16_t e, int16_t f);

@implementation ViewController

-       (void)viewDidLoad {
    [super viewDidLoad];

    int result = MyFunc1(12, 4, 3, -1);
    printf("MyFunc1 division result: %d\n", result);

    result = MyFunc2(10, 20, 30, 60, 70, 50);
    printf("MyFunc2 division result: %d\n", result);
}

@end
```

由于ARM处理器从ARMv4架构一直到ARMv7-AM架构中不包含整数除法指令，只有ARMv7-R才包含，所以我这里定义名为MyDivision函数来做通用的整数除法操作。下面我们通过代码清单15-7来看对应的MyFunc1与MyFunc2的函数实现。

代码清单15-7　iOS工程中在32位模式下对MyFunc1与MyFunc2函数的实现

```
//
//  func.s
//  armv7_test

.text
.align 4

.globl _MyFunc1, _MyFunc2
.globl _MyDivide

#ifdef __arm__

.arm

_MyFunc1:

    // 这里压入两个寄存器，以保持栈空间始终以8字节对齐
    push    {r4, lr}

    sub     r0, r0, r1  // 将第一个参数与第二个参数相减，结果放入R0寄存器
    sub     r1, r2, r3  // 将第三个参数与第四个参数相减，结果存入R1寄存器

    blx     _MyDivide   // 调用MyDivide函数，执行除法计算

    pop     {r4, pc}


_MyFunc2:

    push    {r4, lr}

    add     r0, r0, r1  // 将第一个参数与第二个参数相加，结果放入R0寄存器
    add     r2, r2, r3  // 将第二个参数与第三个参数相加，结果放入R2寄存器
    add     r0, r0, r2  // 将两个求和结果再次相加，结果放入R0寄存器

    ldr     r4, [sp, #8]    // 读取第5个参数放入R4寄存器
    ldr     r12, [sp, #12]  // 读取第6个参数放入R12寄存器
    sub     r1, r4, r12 // 将第5个参数与第6个参数相减，结果放入R1寄存器

    blx     _MyDivide   // 调用MyDivide函数，执行除法计算

    pop     {r4, pc}

#endif
```

从代码清单15-7中我们可以看到，ARM指令集对于这两个函数的实现指令非常精简，参数传递机制与x86的也比较相似，先将前4个整数参数传入寄存器，更多的参数以从右到左的次序压入栈中。并且压入参数的栈空间最后也是由函数调用者回收，函数实现无需关心。

此外，如果各位在Android系统下试验的话，那么在汇编源文件中需要把函数标识符的前导下划线_给删除。

最后提一点，如果当前ARMv7架构处理器支持NEON技术的话，那么在iOS系统中如果在函数实现中使用了向量寄存器，就需要将Q4～Q7这4个向量寄存器保存到栈中。

15.3.2 AArch64架构环境下的函数调用约定

从ARMv8架构起，ARM处理器进入了64位时代。ARMv8架构处理器与x86处理器类似，能同时支持32位的ARMv7架构的指令集，即AArch32模式，以及64位的指令集，即AArch64模式。在15.3.1节中已经介绍了AArch32模式下的函数调用约定，这里将介绍AArch64架构环境下的函数调用约定。

在AArch64模式下，ARM处理器可用的通用目的寄存器数量从16个增加到32个，分别为R0～R31。其中原来AArch32中的PC寄存器已经不显式地给出了，它作为系统隐藏的寄存器而不开放到ISA中。这里，R31作为栈

指针寄存器（SP），R30作为连接寄存器（LR）以存放函数返回地址。这些通用目的寄存器中，R19～R28需要由函数实现来保存。此外，ARMv8架构必须支持SIMD技术，因此它也包含了32个向量寄存器，分别为V0～V31。其中，V8～V15需要由函数实现自己保存。

由于有充足的通用目的寄存器，AArch64模式下参数传递能让更多整数参数放入寄存器中。其中前8个整数参数分别存入R0～R7寄存器，后续更多参数才压入栈中。此外，AArch64中，压入参数的栈空间也是由函数调用者来回收，函数实现无需关心。

在iOS系统下，R18寄存器用于系统特殊功能使用，所以我们在用汇编自己实现函数的时候不能对它进行使用。

代码清单15-8也是根据代码清单15-6的Objective-C源文件来完成AArch64架构下的函数实现。

代码清单15-8　iOS工程中在64位模式下对MyFunc1与MyFunc2函数的实现

```
//
//  func.s
//  armv8_test

.text
.align 4

.globl _MyFunc1, _MyFunc2

#ifdef __arm64__

_MyFunc1:

    sxtb    x1, w1      // 带符号扩展第2个参数
    sxth    x3, w3      // 带符号扩展第4个参数
    sub     x0, x0, x1  // 将第1个参数与第2个参数相加，结果放入X0寄存器
    sub     x2, x2, x3  // 将第2个参数与第3个参数相加，结果放入X2寄存器
```

```
    sdiv    x0, x0, x2  // 将第1个差值除以第2个差值，结果放入X0

    ret


_MyFunc2:

    sxth    x4, w4      // 带符号扩展第5个参数
    sxth    x5, w5      // 带符号扩展第6个参数
    add     x0, x0, x1  // 将第1个参数与第2个参数相加，结果存入X0寄存器
    add     x2, x2, x3  // 将第3个参数与第4个参数相加，结果存入X2寄存器
    add     x0, x0, x2

    sub     x4, x4, x5  // 将第5个参数与第6个参数相减，结果存入X4寄存器
    sdiv    x0, x0, x4

    ret

#endif
```

代码清单15-8与代码清单15-7可以合并为一个汇编源文件，因为这里面已经通过编译器预定义的宏来判定当前编译环境是AArch32模式还是AArch64模式。__arm__表示AArch32模式，__arm64__表示AArch64模式。对于iOS设备，从iPod Touch 6、iPhone 5S起，iPad Air与iPad mini 2起用的都是64位Apple A处理器，即从Apple A7开始的处理器都是ARMv8架构的、支持AArch64执行模式，而之前的处理器都只能用AArch32执行模式。

15.4　本章小结

本章描述了在Windows操作系统以及Unix系操作系统下x86处理器与ARM处理器的函数调用约定。大部分应用程序员可能并不需要关心函数调用约定，但是编译器实现者、高性能计算、嵌入式系统等领域的程序员则需要关心。这里面牵涉不同编程语言、不同编译器所编译出的二进制兼容性问题，另外还有很关键的是C语言与汇编语言如何相互调用问题，这对于高性能计算的开发人员以及嵌入式系统开发人员来说就显得格外重要了。

本章包含了大量汇编语言代码，这里不要求大家能对此深入研究多少，如果各位不从事比较专业领域的开发的话，只需要了解即可，因此在示例代码中也仅仅使用加减乘除这些基本的算术运算，不涉及太多复杂的指令。不管怎么说，通过对本章的学习，各位至少能对函数调用过程，包括如何传参、如何把结果返回、栈指针在函数调用前后是如何移动的……会有更深刻的理解，这对于学习C语言本身来说也是很有帮助的。

第16章　创建静态库与动态库

在第15章各位已经了解了C语言函数的ABI。如果某些系统具有相互兼容的ABI，那么对于兼容C语言标准库的系统环境，我们可以使用相同的静态库，甚至是动态库。静态库与动态库统称为库文件，它们是用于提供给应用程序连接所使用的打包好的一种二进制中间文件。库文件是一个较为完整的目标文件的集合，我们可以在一个系统环境中创建一个用于创建静态库或动态库的工程，然后将工程中的所有源文件进行编译、打包输出为静态库或动态库文件。

静态库文件是参与整个应用程序一起连接的库文件。根据特定实现，静态库可不做符号连接，也可做部分连接，不过无论是哪种实现，静态库文件中允许存在未解决的符号（unresolved symbol），即没有在打包静态库的工程中定义的某些具有外部连接的全局对象和函数，而这些具有外部连接的全局对象和函数在静态库中的某处被引用了。

动态库与静态库不同的是：它是在程序加载时，或在运行时进行加载相关符号，因此动态库必须进行完整的连接处理。如果我们在一个应用程序中，通过在运行时加载动态库中特定的全局函数或全局对象，那么我们将通过当前系统环境特定的系统调用进行。

下面，我们将分别针对Windows、macOS以及Linux系统来讲解在这些

操作系统中如何创建静态库与动态库，并且如何使用静态库和动态库。

16.1 Windows系统下创建静态库与动态库

由于在Windows系统下，我们用Visual Studio开发工具更多一些，所以这里将主要介绍如何通过Visual Studio 2017 Community来创建静态库与动态库，然后在主应用程序中使用这些库。各位如果没有2017版本，那么使用2015、2013版也差不多。

16.1.1 Windows系统下创建并使用静态库

我们首先打开Visual Studio，然后准备创建一个Win32控制台应用程序，如图16-1所示。

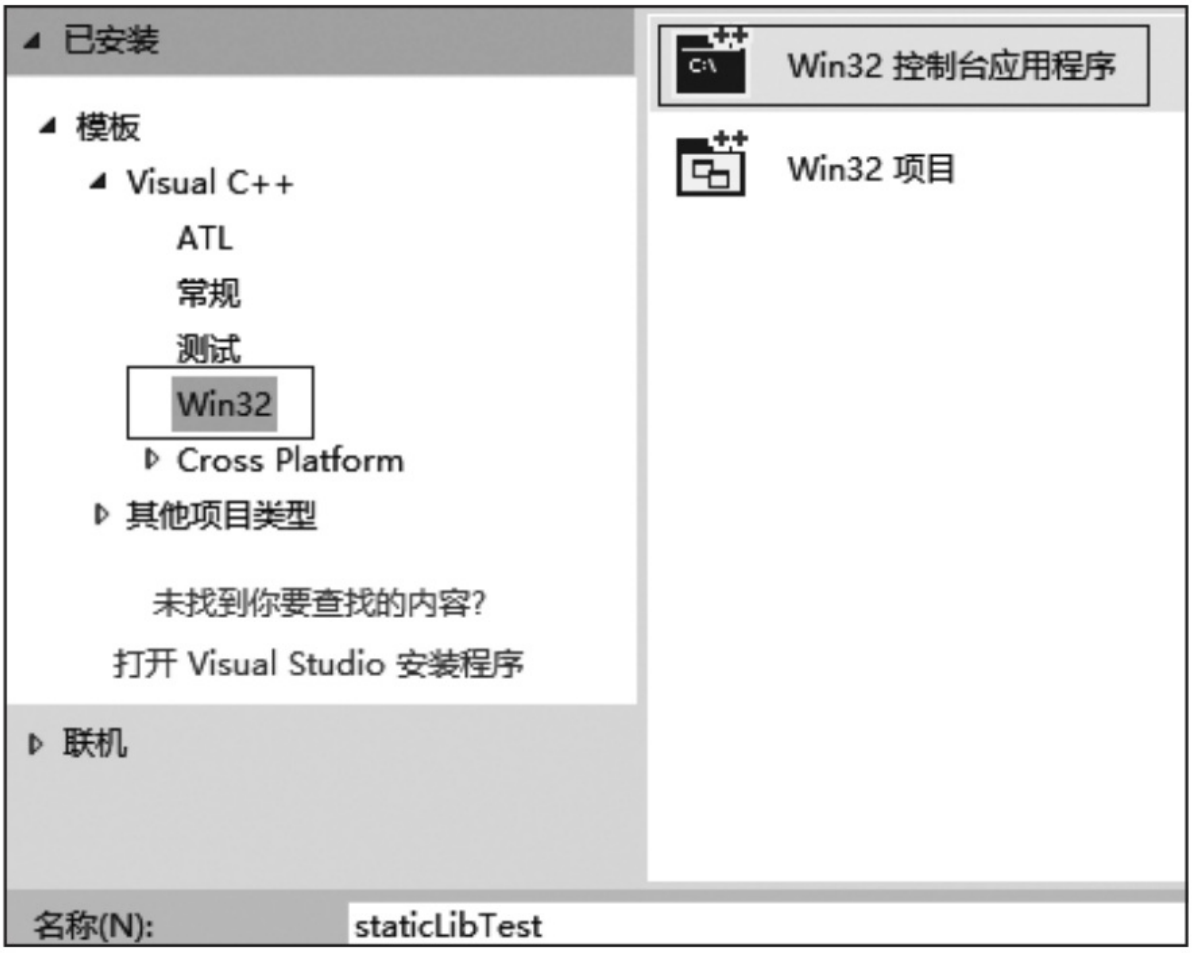


图16-1　选择Win32控制台应用程序

随后，我们填写项目名为staticLibTest，点击“下一步”到下一个界面。

在下一个界面中，我们先点击“应用程序设置”，设置应用属性。这里，在“应用程序类型”中选择“静态库”单选按钮，表示我们要创建的是一个静态库工程。在“附加选项”中把所有选项都取消，如图16-2所示。

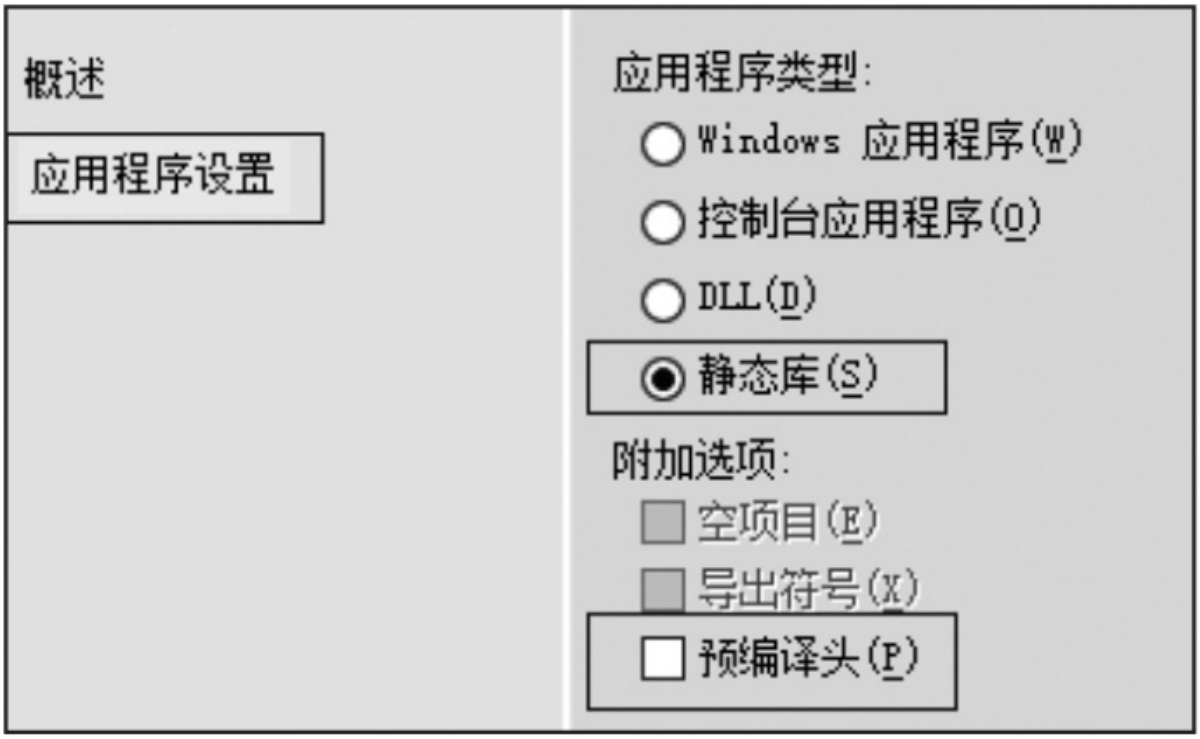


图16-2　创建静态库工程

最后，我们点击“完成”按钮，进入工程主界面。我们在右侧找到“源文件”文件夹，鼠标右键点击它，然后选择“添加”，再点击“新建项”，然后我们在左侧选择“代码”，再点击“C++File”之后，在“名称”文本框中输入“lib.c”。最后点击“添加”按钮。这样我们就能看到一个文本编辑框，这里就能编写lib.c源文件中的C代码了。

静态库中的C代码见代码清单16-1。

代码清单16-1　静态库的C代码

```
#include <stdio.h>
// 此函数将在主应用程序中定义，当前库中不解决此符号
extern void MainFunction(void);

// 在静态库中定义了具有内部连接的函数InnerFunction，
```

```
// 它对主应用程序的符号连接没有任何影响，仅作用于静态库项目中的当前源文件
static int InnerFunction(void)
{
    puts("This is a static library inner function!");

    return 10;
}

// 在静态库中定义了StaticLibTest具有外部连接的全局函数
void StaticLibTest(int a)
{
    // 调用了MainFunction，即对MainFunction符号进行了引用。
    // 因此必须在主应用程序中对MainFunction进行定义，否则主应用程序将通不过连接
    MainFunction();

    int b = InnerFunction();
    printf("value = %d\n", a + b);
}
```

我们编写完上述代码之后，在菜单栏找到“生成”按钮，然后点击之后选择“生成解决方案”，这样在我们项目工程文件夹的Debug目录中就会出现staticlibTest.lib文件了。

接下来，要创建主工程目录了。一开始与图16-1一样，选择Win32控制台应用程序。然后在“应用程序类型”中，选择“控制台应用程序”，然后同样，“附加选项”中不选中任何选项。点击“完成”按钮则创建好了当前的主应用程序项目工程。

然后在编辑界面，我们仍然在“源文件”文件夹处鼠标右键点击一下，然后选择“添加”，再选择“新建项”，然后创建一个名为main.c的源文件。随后，我们把刚才得到的staticlibTest.lib文件放入当前工程中main.c所在的目录下。我们回到Visual Studio，再右击“Source Files”，选择“添加”，然后再选择“现有项”，在弹出的文件选择对话框中选中staticlibTest.lib，最后点击“添加”按钮，则把staticlibTest.lib静态库文件也加入到了“源文件”文件夹中了，如图16-3所示。

图16-3　添加静态库文件

接下来，我们在main.c文件中输入代码清单16-2中所示的代码。

代码清单16-2　主程序源代码

```
#include <stdio.h>

// 在主函数中定义具有外部连接的全局函数MainFunction
void MainFunction(void)
{
    puts("This is a function in main project!");
}

// 在主应用程序中定义具有内部连接的静态函数InnerFunction
static int InnerFunction(void)
{
    puts("This is an inner function in main!");
    return 200;
}

// 声明定义在静态库中的具有外部连接的全局函数StaticLibTest
extern void StaticLibTest(int a);

int main(void)
{
    StaticLibTest(1);
```

```
    int a = InnerFunction();
    printf("a = %d\n", a);

    getchar();
}
```

输入完成后，这次我们可以点击工具栏中的绿色小三角按钮，让编译器编译、连接后直接运行。此时连接器会把main.c生成的目标文件与静态库staticlibTest.lib文件做总和连接，从而解决所有外部符号，最终生成可执行文件。如果代码没有输入错误，整个程序能直接跑起来。

这里我们可以看到，在main.c源文件中定义了静态库中所引用到的MainFunction函数。此外，在main函数中则调用了在静态库中定义的StaticLibTest函数。而静态库中与main中所定义的InnerFunction函数都是它们各自独有的，因此相互之间没有任何影响。对静态库中符号的连接主要针对具有外部连接的符号。

16.1.2 Windows系统上创建并使用动态库

动态库又称为动态连接库，是指应用程序在启动前被加载时或在运行时加载的连接库。它的优点是无需重新连接所有的库从而重新生成新的可执行文件，而是只需替换当前功能模块对应的动态连接库文件。这样分发给最终用户的时候，或者更新应用时无需把整个可执行文件全都连接更新一遍，而只需要提供修改过的功能模块所对应的动态连接库文件，只要当应用程序启动执行时即可产生更新后的效果。

在Windows系统上，我们对于“补丁”这个词已经是耳熟能详了，而应用程序的“补丁”往往就是通过更新动态库文件来实现的，而不需要更新可执行文件本身。下面我们就来介绍如何通过Visual Studio 2017 Community来构建我们自己的动态库文件。

首先，我们仍然根据图16-1创建一个Win32控制台工程，名字为dllTest。然后，右击“应用程序设置”中，在“应用程序类型”一栏选择DLL；在“附加选项”中取消勾选其他选项，而勾选上“空项目”，如图16-4所示。

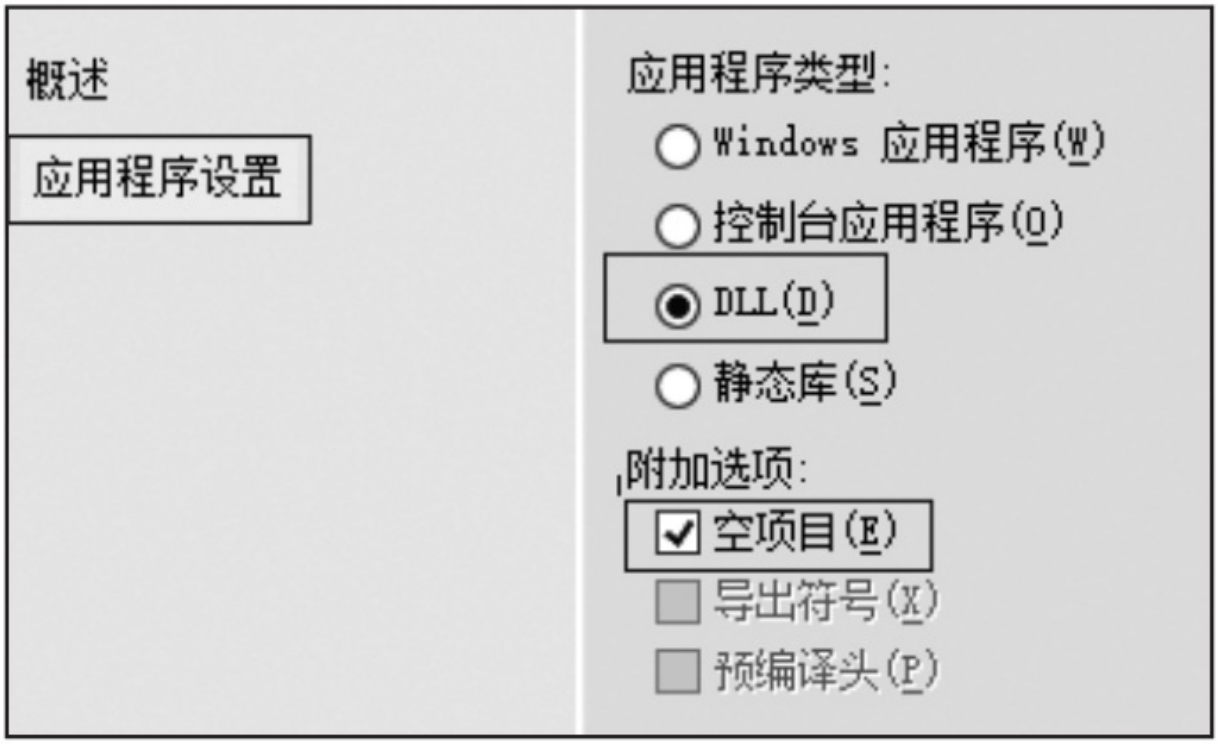


图16-4　设置DLL工程

这里大家一定要勾选上“空项目”，否则DLL项目工程会自动导入一些杂七杂八的头文件与资源文件，使得我们后期构建的动态库文件无法正常

工作。

最后点击“完成”即可创建好一个DLL工程项目。这里我们类似地在“源文件”文件夹中创建一个名为”dll.c”的源文件，然后敲入代码清单16-3所示的代码内容。

代码清单16-3　动态库代码内容

```
#include <stdio.h>

// 在Windows中使用__declspec(dllexport)来指定
// 当前的函数作为可被动态加载的外部函数
void __declspec(dllexport) DLLFunction(int a)
{
    printf("This is a dll function! a = %d\n", a);
}

// 这个函数将在主程序中将以运行时加载的形式进行调用
int __declspec(dllexport) DLLLoadedFunction(void)
{
    puts("DLL loaded function is called!");
    return 100;
}
```

在Windows中使用了一个C语言扩展关键字——__declspec（dllexport）函数说明符来指明当前函数可被动态加载。__declspec（dllexport）只能用于修饰具有外部连接的全局函数或对象。

另外在代码清单16-3中，函数DLLFunction将在主程序加载时被加载到程序内存中；而函数DLLLoadedFunction将在运行时直接通过Windows API动态加载到程序内存中，然后通过函数指针做间接调用。

写完上述代码之后，我们同样在菜单栏找到“生成”按钮，点击之后选择“生成解决方案”，这样在我们项目工程文件夹的Debug目录中就会同时

出现dllTest.lib文件与dllTest.dll文件了。其中，dllTest.lib包含了函数实现本体，而dllTest.dll存放的是符号映射等内容，所以两者都需要添加到主程序的工程中。Windows中，dll文件就被称为动态连接库文件。

我们回到之前创建好的main工程，将刚才构建得到的dllTest.lib文件放到main.c源文件所在的工程目录下，然后再将dllTest.dll再复制粘贴到工程主目录下的Debug目录中，与生成的可执行文件放在一起，由于可执行文件后面在加载过程中以及运行时查找dllTest.dll文件时会以它所在的目录作为默认搜索路径，这么处理显然更容易些。随后将之前添加好的staticlibTest.lib文件删除，将新增的dllTest.lib文件添加到“源文件”文件夹中，如图16-5所示。

图16-5　在main工程中添加dllTest.lib并将dllTest.dll放在可执行文件所在目录

这里dllTest.dll无需添加到工程中，因为它不参与连接。随后，我们修改main.c的内容，如代码清单16-4所示。

代码清单16-4　Windows动态加载dll的main源文件

```
#include <windows.h>
#include <stdio.h>

// 用 __declspec(dllimport)声明当前函数是通过加载器在加载程序时做动态库连接的
extern void __declspec(dllimport) DLLFunction(int a);

int main(void)
{
    DLLFunction(100);   // 这里可以直接调用加载时载入的DLLFunction函数

    // 使用Windows API库函数LoadLibrary动态加载dllTest.dll库
    HMODULE dllLibHandle = LoadLibrary(L"dllTest.dll");

    // 使用Windows API库函数GetProcAddress获得动态库中
    // DLLLoadedFunction外部函数符号
    int (*pFunc)(void) = (int(*)(void))GetProcAddress(dllLibHandle,
                          "DLLLoadedFunction");

    // 通过函数指针间接调用DLLLoadedFunction
    int a = pFunc();
    printf("a = %d\n", a);

    getchar();
}
```

这里由于用到Windows API，因此引入了<windows.h>系统库文件。__declspec（dllimport）说明符用于声明当前函数或对象将在加载时做动态连接，这样它所对应的符号能安全地在当前代码中进行引用。

在代码清单16-4中，对DLLFunction函数调用以下的部分就是通过Windows API对dllTest.dll的运行时动态加载过程了。GetProcAddress系统函数基于得到的动态库的句柄（handle）来获取“DLLLoadedFunction”符号所在的相对地址，然后赋值给一个相应类型的函数指针对象，最后通过该函数指针做间接调用。

这里大家要注意的是，dll文件的位置要放对，如果程序找不到dll文件则会报错。此外，dll文件可以放在与C源文件的相同路径下，用于工程调试时进行加载；而在Debug目录下与可执行文件放在一起的dll文件用于在Debug目录下直接运行可执行文件时做dll文件的加载。

16.2　macOS系统下创建静态库与动态库

在macOS中，我们可以方便地使用Xcode这一灵活强大的集成开发环境来创建静态库与动态库。Xcode可以在Mac App Store免费下载，而且无需注册开发者账号即可使用。不过Apple开发者账号申请起来比较容易，而且也是免费的，直接去http://developer.apple.com即可。这里用的Xcode版本是8.2.1，不过对于创建静态库与动态库而言，更老、更新的版本都差不多一样。

在macOS系统下，静态库的创建和使用与Windows环境下的差不多，并且在使用时也是需要加入主工程一同进行连接的。而动态库的使用则与Windows平台的不太一样，在macOS以及其他类Unix系统中，主应用程序可以直接在加载时由加载器去动态加载动态连接库中的函数与对象，但是需要指明动态连接库的路径，在macOS中使用DYLD_LIBRARY_PATH环境变量去指定。否则的话，加载器会默认找/usr/local/lib/路径下的动态库。动态连接库中所有具有外部连接的全局函数与对象的符号默认都是可被动态连接的，因此不需要像Windows里那样使用__declspec（dllexport）去指定。如果我们希望在动态库中的某些符号不允许被外部连接或动态加载，可以使用17.14.1节中所描述的第17条——可见性属性来指明当前符号的对外可见性。

16.2.1　macOS系统下创建并使用静态库

首先，我们打开Xcode，然后选择左侧的“Create a New Xcode Project”，出现选择新项目的模板对话框。我们在此对话框中，在对话框上侧选择“macOS”一栏下的“Framework&Library”选项区域，然后点击下面大框里的“Library”图标，如图16-6所示。

图16-6　macOS库项目创建

然后，我们点击“Next”按钮到下一步，来到项目选项对话框。这里可以填上自己的项目名，本demo用的是“StaticLibTest”，下面的组织名与组织标识可以自己随便填写，然后“Framework”可以选择“None（Plain C/C++Library）”，由于我们不需要建立基于Cocoa Framework的库，所以这里只要选普通C语言的库即可。下面的“Type”选择“Static”，表示要创建的是静态库。这个对话框设置完后的效果如图16-7所示。

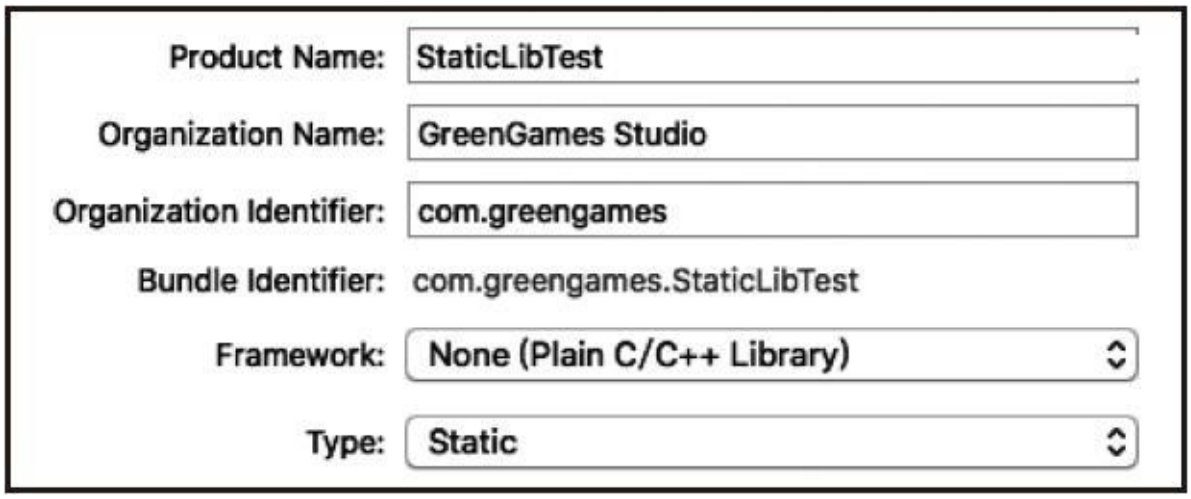


图16-7　macOS创建静态库项目

完成之后点击“Next”按钮，来到了工程存放路径选择对话框，这里各位选择将此静态库工程存放到哪个文件目录下。选择完之后点击“Finish”按钮，然后就进入了静态库工程的主界面。这里我们能看到红色的libStaticLibTest.a，这个文件将是我们后面成功构建后生成的静态连接库文件。

我们首先设置项目工程的编译选项，点击中间栏“TARGETS”下面的“StaticLibTest”，然后选择右侧栏的“Build Settings”，找到“Apple LLVM x.y-Language”这一栏，将“C Language Dialect”这一项选择设置为“gnu11”，如图16-8所示。

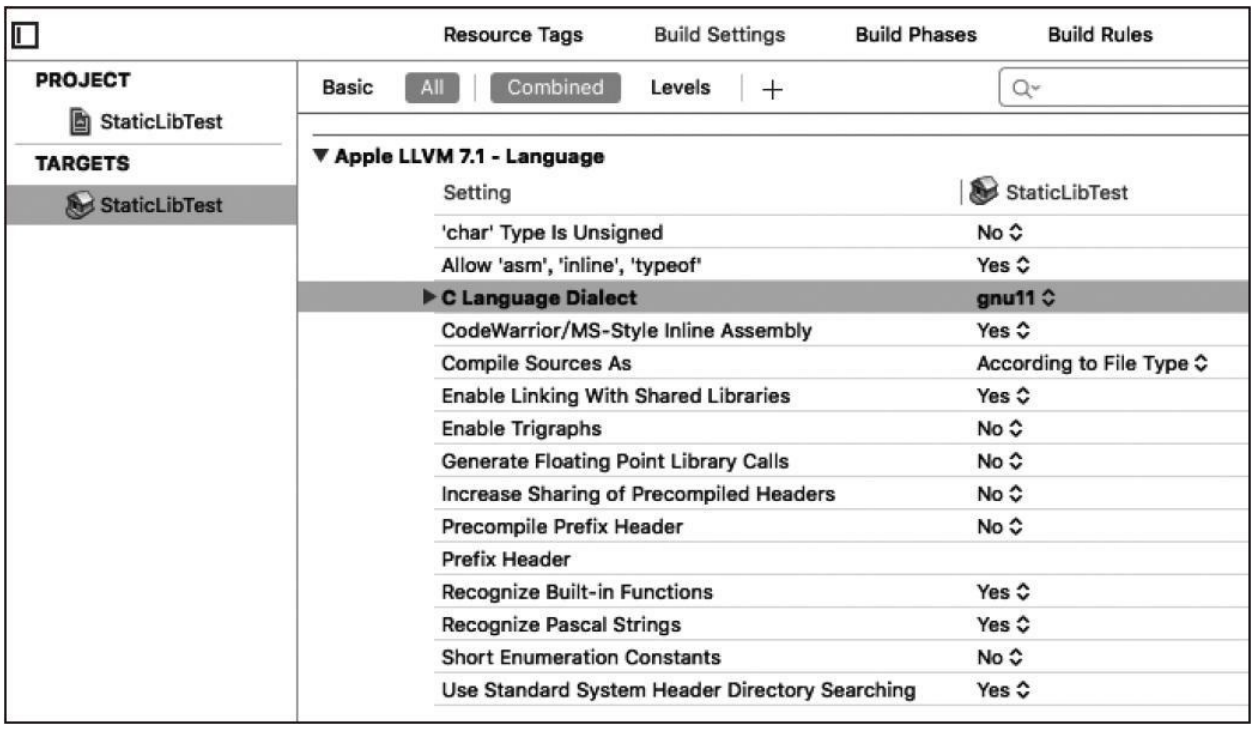


图16-8　Xcode环境设置库工程的编译选项

随后，我们鼠标右键点击带有蓝色图标的工程文件“StaticLibTest”，然后选择“New File”，这样能看到一个选择源文件类型的对话框。我们仍然在上侧栏找到“macOS”，然后点击它下面的“Source”选项区域中的“C File”，如图16-9所示。

然后点击“Next”按钮，输入C源文件名“lib”，这里“.c”后缀可省，默认为.c后缀。最后可以将“Also Create a header file”选项给去掉，这里我们不需要创建一个头文件，然后点击“Next”按钮后，直接点击“Create”按钮就在我们的StaticLibTest项目中新建好了lib.c源文件了。

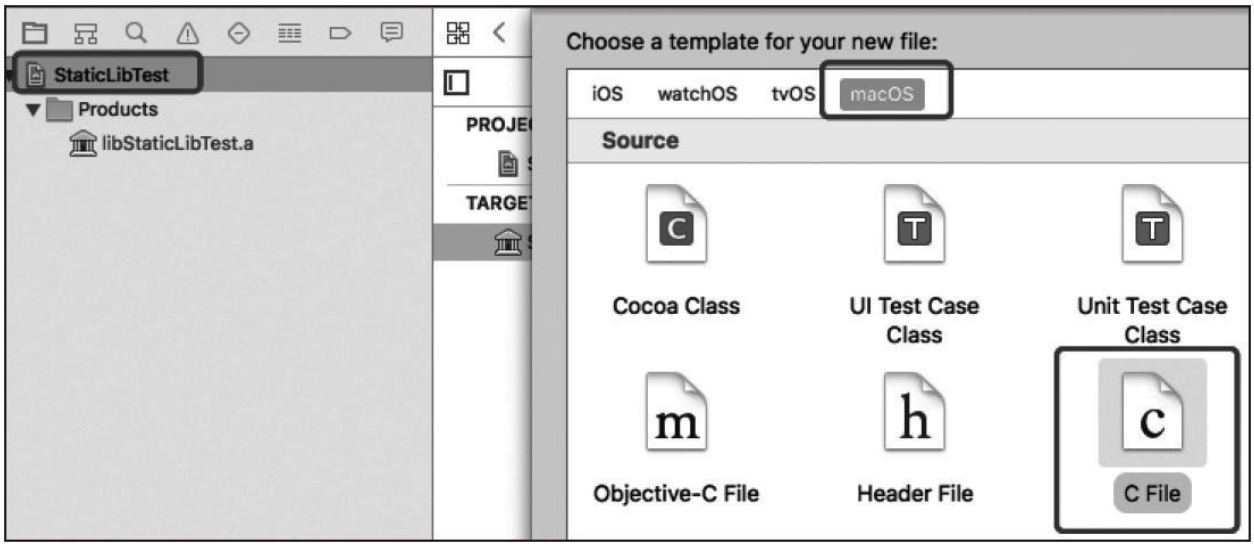


图16-9　Xcode新建C源文件

我们对lib.c进行编辑，直接将代码清单16-1中的内容复制粘贴进去。完成之后，点击菜单栏上的“Product”，再选择“Build”，然后我们就能看到原本红色的“libStaticLibTest.a”变成黑色文字了。此时，我们右键点击此“libStaticLibTest.a”，然后选择“Show in Finder”，这样就能跳转到“libStaticLibTest.a”文件所在的目录了。我们复制该文件，然后粘贴到主应用工程的目录下即可。

下面我们用Xcode创建主应用工程。打开Xcode后同样先选择“Create a New Xcode Project”，然后在选择新项目模板的对话框中选择“macOS”一览下的“Application”中选择“Command Line Tool”，如图16-10所示。

然后点击“Next”按钮，弹出新建项目选项设置。这里我们可以将项目名设置为“Main”，然后“Language”这一栏选择“C”，如图16-11所示。

完成后点击“Next”按钮，出现项目工程存放路径的选择对话框，我们

选择好之后点击“Create”按钮就创建好了主工程项目。此时，在Main文件夹下自动生成了一个main.c的C源文件。我们鼠标右键点击main.c源文件，然后选择“Show in Finder”，跳转到main.c文件所在的目录。然后将刚才生成的“libStaticLibTest.a”静态连接库文件拖进该文件夹。完成后，鼠标右键点击Xcode项目中的“Main”文件夹（黄色文件夹图标的那个），选择“Add Files to Main...”，选中“libStaticLibTest.a”，然后点击“Add”按钮。这样就把静态连接库加入到Main项目工程中了，如图16-12所示。

图16-10　用Xcode选择macOS命令行工具应用项目

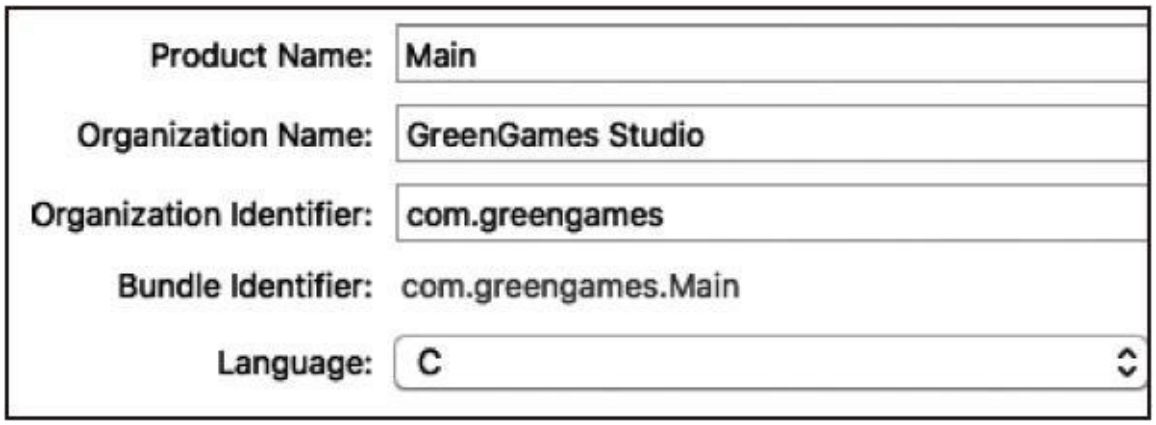


图16-11　用Xcode创建C项目主工程

图16-12　用Xcode将静态连接库加入到主项目工程

我们像静态库项目设置那样设置主项目工程的编译选项，将C语言标准选择为“gnu11”。随后，我们将代码清单16-2中的内容复制粘贴到此main.c源文件中。最后，我们点击工具栏左侧的黑色三角箭头就可以编译main.c然后连接静态库文件，Xcode会自动执行生成好的Main可执行文件，将结果输出在下方的输出框中。

在Windows系统中，静态库文件的后缀为“.lib”；而在几乎所有类Unix系统中，静态库文件的后缀全都是“.a”。

16.2.2　macOS系统下创建并使用动态库

macOS系统下创建动态库的过程与创建静态库十分类似，我们在设置项目选项的时候，仅仅将“Type”选择为“Dynamic”即可。这里我们创建一个名为“DynLibTest”的动态库项目工程，如图16-13所示。

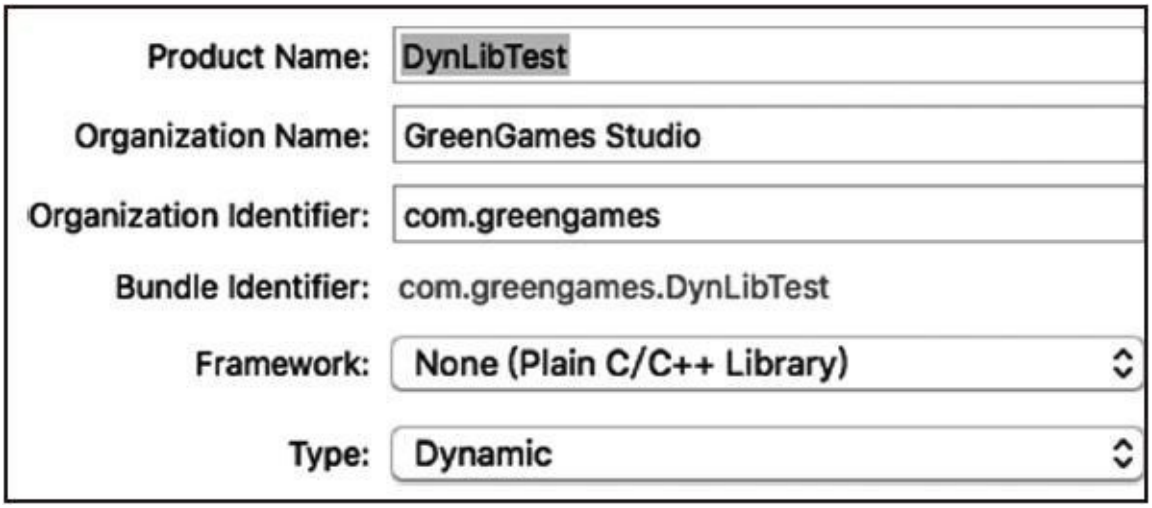


图16-13　Xcode设置动态库项目工程

我们这里创建一个名为“dylib.c”的源文件，随后基本参考代码清单16-3的内容，但这里必须去掉__declspec（dllexport）这一说明符。在macOS以及其他类Unix中，动态库中所有具有外部连接的函数与对象都默认可被主应用进行动态加载。这里，dylib.c的内容如代码清单16-5所示。

代码清单16-5　macOS环境下的动态库代码

```
#include <stdio.h>

// 以下函数与对象将在主程序中以加载时或运行时进行加载的方式做动态连接
int RuntimeLoadedFunction(void)
{
    puts("DLL loaded function is called!");
    return 100;
}
int dyn_runtime = 20;
```

完成之后，我们点击菜单栏中的“Product”，然后点击“Build”即可完成构建，最终生成“libDynLibTest.dylib”文件。

我们现在打开之前创建的主应用工程，将之前的静态库文件删除，然

后与添加静态类库类似的方式将“libDynLibTest.dylib”文件拷贝到此目录下。最后，我们编辑main.c源文件，如代码清单16-6所示。我们先使用加载时连接的方式。

代码清单16-6　macOS在加载程序时加载动态库符号的主程序

```
#include <stdio.h>
// 声明动态库中的RuntimeLoadedFunction全局函数
extern int RuntimeLoadedFunction(void);

// 声明动态库中的dyn_runtime全局对象
extern int dyn_runtime;

int main(int argc, const char * argv[])
{
    printf("Hello, World!\n");

    // 调用动态库中定义的RuntimeLoadedFunction全局函数
    int value = RuntimeLoadedFunction();
    printf("value = %d\n", value);

    // 调用动态库中定义的dyn_runtime全局对象
    printf("dyn_runtime = %d\n", dyn_runtime);

    return 0;
}
```

当我们编辑好代码清单16-6中的代码之后，点击工具栏中的运行按钮直接运行会出现加载时错误——在/usr/local/lib/路径中找不到libDynLibTest.dylib文件。此时，我们可以打开控制台程序，然后进入到libDynLibTest.dylib文件所在的目录，然后使用export DYLD_LIBRARY_PATH=来指定用户目录下的动态库路径。随后，我们进入可执行文件Main所在的目录，然后直接运行。整个过程如图16-14所示。

```
localhost:~ zennychen$ cd /Users/zennychen/Desktop/Main/Main/
localhost:Main zennychen$ export DYLD_LIBRARY_PATH=/Users/zennychen/Desktop/Main/Main
localhost:Main zennychen$ cd /Users/zennychen/Library/Developer/Xcode/DerivedData/Main-
    djvwehzvisibujbdqyncbaakgedb/Build/Products/Debug/
localhost:Debug zennychen$ ./Main
Hello, World!
DLL loaded function is called!
value = 100
dyn_runtime = 20
localhost:Debug zennychen$
```

图16-14　在控制台中动态加载动态库文件并执行Main程序

上面描述的是在控制台中使用加载程序时加载动态库的方式来运行主程序。下面我们将描述如何在Main程序运行时加载动态库中的符号。这里我们需要在做一些准备。我们在上面原有的Main工程的基础上，需要把动态库文件添加到与Main可执行文件的同一路径中。我们右击Product里的Main，然后选择“Show in Finder”，然后系统会弹出当前Main所在的目录，我们将libDynLibTest.dylib文件复制到该目录下。然后在main.c源文件中输入代码清单16-7的内容。

代码清单16-7　macOS使用运行时加载动态库符号的主程序

```
#include <stdio.h>
#include <string.h>
#include <stdbool.h>

// 此头文件包含了运行时动态加载动态库中外部符号的API
#include <dlfcn.h>

// 此头文件包含了_NSGetExecutablePath系统API
#include <mach-o/dyld.h>
int main(void)
{
    // 获取当前可执行程序所在路径
    char path[512];
    uint32_t size = sizeof(path);
    _NSGetExecutablePath(path, &size);

    // 将当前路径与动态库文件名进行拼接，得到动态库的完整路径
    strcat(path, "/libDynLibTest.dylib");

    // 使用dlopen函数加载动态库，返回动态库文件句柄
    void *dylibHandle = dlopen(path, RTLD_NOW);
    if(dylibHandle == NULL)
    {
```

```
            puts("dylib file not found!");
            return -1;
        }

        do
        {
            // 使用dlsym函数加载外部函数符号
            int (*pFunc)(void) = dlsym(dylibHandle, "RuntimeLoadedFunction");
            if(pFunc == NULL)
            {
                puts("RuntimeLoadedFunction function not found!");
                break;
            }

            int a = pFunc();
            printf("a = %d\n", a);

            // 使用dlsym函数加载外部对象符号
            int *p = dlsym(dylibHandle, "dyn_runtime");
            if(p == NULL)
            {
                puts("dyn_runtime object not found!");
                break;
            }

            printf("dyn_runtime = %d\n", *p);
        }
        while(false);

        // 关闭动态库文件句柄
        dlclose(dylibHandle);
    }
```

代码清单16-7中用到了macOS中的一些系统API，其中<dlfcn.h>系统头文件是大部分类Unix系统都自带的，这其中也包括Linux，所以在大部分类Unix系统中我们都能使用dlopen函数来打开并加载指定的一个动态库文件；然后用dlsym函数加载该动态库中指定的某个函数或对象；最后用dlclose关闭动态库文件。而这里的<mach-o/dyld.h>是macOS专用的，它包含的_NSGetExecutablePath函数用于获取当前可执行文件所在的目录路径。这里各位要注意的是，在macOS中，如果我们指定“./xxx”路径并不一定是可执行文件当前路径，而是当前用户的home目录路径，这一点与Windows系统不一样。

16.3 Linux系统下创建并使用静态库与动态库

在Linux系统下我们往往直接使用命令行来编译整个C语言工程，包括打包成静态库、动态库、生成可执行文件等。当然，在Linux系统下我们也可以使用像Eclipse等集成开发环境（IDE）进行开发，不过这里我们将使用命令行来描述。在几乎所有类Unix系统中，静态库与动态库都用lib作为文件名前缀。比如，libdispatch.a表示一个dispatch静态库文件；libdispatch.so表示一个dispatch动态库文件。后缀名.a就是压缩包archive的首字母；后缀名.so是shared object的首字母缩写。本书下面所描述的编译环境为GCC 4.9.0，运行环境为CentOS 7.1。

16.3.1 Linux系统下创建并使用静态库文件

我们首先在Linux环境下用编辑器输入代码清单16-1中的内容，然后将文件保存为static_lib.c。我们先用以下命令生成与之对应的.o目标文件：

```
gcc  -c  -std=gnu11  static_lib.c
```

完成之后就会在当前目录下生成static_lib.o文件。这里的-c命令选项就是将源文件编译为目标文件，而不做连接处理。然后我们将此目标文件打包成静态库文件：

```
ar  -cr  libstatic_lib.a  static_lib.o
```

这样，在当前路径中就会出现libstatic_lib.a静态库文件。这里的命令选项中c表示创建静态库文件；r表示如果当插入的模块名已经在库中存在，则替换同名的模块。如果我们要对多个目标文件打包，那么可以再往后添加目标文件名。然后用编辑器输入代码清单16-2中的内容，再将文件保存为main.c，与之前的static_lib.o存放在同一路径下。做完之后，我们用以下命令生成可执行文件test：

```
gcc main.c -std=gnu11 -L./ -lstatic_lib -o test
```

完成之后就会在当前目录中生成test可执行文件。我们直接用./test即可运行test程序。这里-L用于指定静态库的搜索路径，后面跟./表示当前目录路径；-l就是用于连接静态库的命令选项，大家应该注意到了，-l命令后面没有lib前缀名，也没有.a后缀名，而直接跟静态库模块名。-o命令选项用于指定输出最终文件名。

16.3.2 Linux系统下创建并使用动态库

Linux下创建并使用动态库的方式与macOS系统下差不多。我们用编辑器输入代码清单16-5所述内容，将它保存为dynamic.c源文件。然后我们用以下命令对它进行编译，生成目标文件：

```
gcc  -std=gnu11  -fPIC  -c  dynamic.c
```

这条命令中，-fPIC选项是将源文件编译为位置无关的代码，这对于构

成动态库的目标文件而言既能增强安全性，又能增强灵活性。由于macOS上的Xcode默认编译选项已经加上了-fPIC编译选项，因此我们无需手动编辑更改。而在Linux系统中，我们需要显式指定，默认的选项是位置相关的。“PIC”也就是“Position Independant Code”的缩写。

然后，我们用以下命令创建动态库文件：

```
gcc -shared dynamic.o -o libdynamic.so
```

这里的命令选项-shared就是指定GCC编译器工具链对输入的目标文件生成共享目标文件，也就是Linux下的动态库文件。

Linux下使用加载程序时加载动态库的方式与macOS类似，只不过我们需要使用LD_LIBRARY_PATH环境变量来设置用户指定的动态库路径，否则加载器默认使用的是系统指定的动态库路径。我们可以使用代码清单16-6中的代码来进行尝试，这里不再赘述。

这里，我们将举一个使用程序运行时加载动态库的例子，由于运行加载的方式与macOS还稍微有些不同。我们把代码清单16-8的内容输入到main.c文件中。

代码清单16-8　Linux系统下通过运行时加载动态库方式的主程序

```
#include <stdio.h>
#include <string.h>
#include <stdbool.h>

// 此头文件包含了运行时动态加载动态库中外部符号的API
#include <dlfcn.h>
```

```
// 此头文件包含了readlink系统函数，用于获取当前可执行文件的完整路径
#include <unistd.h>


int main(void)
{
    // 获取当前可执行程序所在路径
    char path[512];
    int size = readlink("/proc/self/exe", path, 512);
    // 由于这里的path得到的是当前可执行文件的完整路径，因此需要萃取出它所在的路径名
    while(path[--size] != '/' && size > 0);
    path[size] = '\0';

    // 将当前路径与动态库文件名进行拼接，得到动态库的完整路径
    strcat(path, "/libdynamic.so");

    // 使用dlopen函数加载动态库，返回动态库文件句柄
    void *dylibHandle = dlopen(path, RTLD_NOW);
    if(dylibHandle == NULL)
    {
        printf("so file not found: %s\n", path);
        return -1;
    }

    do
    {
        // 使用dlsym函数加载外部函数符号
        int (*pFunc)(void) = dlsym(dylibHandle, "RuntimeLoadedFunction");
        if(pFunc == NULL)
        {
            puts("RuntimeLoadedFunction function not found!");
            break;
        }

        int a = pFunc();
        printf("a = %d\n", a);

        // 使用dlsym函数加载外部对象符号
        int *p = dlsym(dylibHandle, "dyn_runtime");
        if(p == NULL)
        {
            puts("dyn_runtime object not found!");
            break;
        }

        printf("dyn_runtime = %d\n", *p);
    }
    while(false);

    // 关闭动态库文件句柄
    dlclose(dylibHandle);
}
```

由于Linux与macOS相比，在获取当前可执行文件所在的路径上有些区别，因此这里重新贴一下完整代码，而在其他方面则差不多。然后大家可以用以下命令来编译main.c：

```
gcc  -std=gnu11  main.c  -o  test  –ldl
```

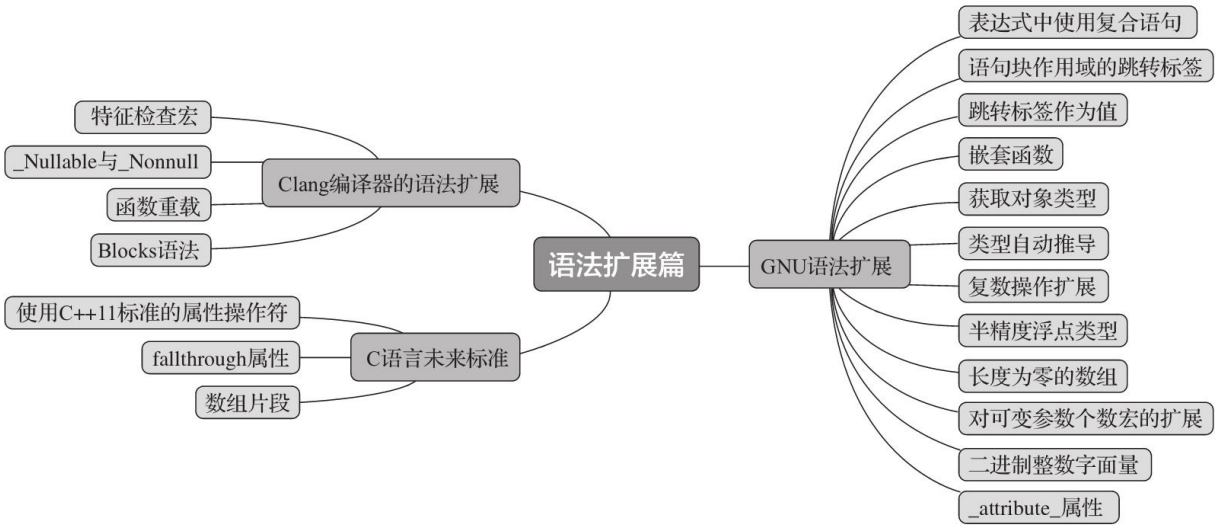
这条命令最终生成名为test的可执行文件。这里，-ldl命令选项用于连接dl库。dl库包含了Linux系统下对dlopen、dlsym、dlclose函数的实现，在默认情况下dl库是不被GCC编译器自动连接的，因此需要我们显式地添加进行连接。

16.4 本章小结

本章为大家初步讲解了静态库与动态库的创建与使用。在实际工程项目中，我们往往会涉及创建静态库与动态库的需求，而且库对于工程来说也是对功能模块的封装与抽象，为其他功能模块的开发者屏蔽掉对当前所用模块的实现细节。此外，将一些成熟的代码打包成库也能节省不少编译构建的时间，因此打包成库确实非常有用。

除了这些C语言标准的静态库与动态库之外，有些系统平台还有自己特定的库，比如macOS上有framework库文件（实质上是一个文件夹），用于更好地实现功能模块化，它主要用于Objective-C与Swift编程语言。而像Java编程语言也有自己的JAR库文件等等。另外，像Java、Python等语言可以调用C语言实现的函数，而主要手段就是通过动态库文件进行加载，然后通过这些语言特定的桥接API与C函数进行交互。

第四篇　语法扩展篇


语法扩展篇
GNU语法扩展
表达式中使用复合语句
语句块作用域的跳转标签
跳转标签作为值
嵌套函数
获取对象类型
类型自动推导
复数操作扩展
半精度浮点类型
长度为零的数组
对可变参数个数宏的扩展
二进制整数字面量
_attribute_属性
Clang编译器的语法扩展
特征检查宏
_Nullable与_Nonnull
函数重载
Blocks语法
C语言未来标准
使用C++11标准的属性操作符
fallthrough属性
数组片段

第17章　GCC对C11标准的语法扩展

从桌面系统一直到嵌入式系统，现在GCC编译器或遵循GNU语法扩展规范的C语言编译器已经十分普遍。在桌面系统端，像GCC（包括使用GCC核心的MinGW、Dev-C++等Windows上的编译器）、Clang编译器（包括Visual Studio中集成的VS-Clang）等都遵循GNU语法扩展。当然，Clang编译器自己还有一些扩展，另外GCC有些特性它也没有支持，下一章会讲到这些情况。在嵌入式系统中，像用于交叉编译的编译器，如ARM-GCC、MIPS-GCC则用得更是普遍了；现在ARM官方最新推荐使用的ARM Compiler 6则完全基于Clang编译器。此外，像Arduino也是基于AVR-GCC或ARM-GCC等，根据采用特定的处理器而定，所以我们在Arduino集成开发环境中也完全可以使用GNU语法扩展。而像macOS、iOS中所用的Apple LLVM，以及Android开发用的NDK也都采用基于Clang编译器的编译工具链。因此，我们当前在绝大多数场合均可使用更灵活、更强大的GNU语法扩展。

由于GNU语法扩展种类较多，并且有些也是由于老的标准没有，而自己加了之后新的C语言标准又予以支持的，所以以下介绍的GNU语法扩展主要就是C11标准所不具备的，但又比较实用的，有些GCC编译器支持但Clang编译器不支持的也会另作说明。

为了甄别当前编译器是否支持GNU语法扩展，我们可以使用

__GNUC__这个宏（前后各有两条下划线）。代码清单17-1给出一个简单的例子。

代码清单17-1　鉴别当前C编译器是否支持GNU语法扩展

```
#include <stdio.h>

#ifdef __GNUC__
#warning "This is a GNU compatible compiler!"
#endif

int main(void)
{
    return 0;
}
```

如果当前编译器支持GNU语法扩展，那么编译器在编译时就会发出警告。

17.1　在表达式中使用复合语句与声明

在标准C语言中我们知道，在一条表达式中只能使用表达式作为其子表达式，而不能使用语句，更不能使用声明。声明以及其他语句中可包含一条或多条表达式。但在GNU语法扩展中则可通过使用一个圆括号将一条复合语句包住作为一条表达式。在这复合语句中当然还能包含对对象的声明等其他语句。

其语法形式为：(复合语句)

整个圆括号包裹的复合语句表达式的计算结果是由该复合语句中最后一条语句的计算结果所表示的。代码清单17-2展示了这种语法的表达以及效果。

代码清单17-2　表达式中使用复合语句与声明

```
// main.c源文件
#include <stdio.h>

int main(void)
{
    int a = 10, b = 20;

    // 我们已经知道，C语言中用{ }包围的一系列语句称为一条复合语句。
    // 这里我们将( )包住的复合语句作为 = 操作符的右操作数，
    // 因此该复合语句的最后一条表达式的计算结果必须是一个int类型
    a = ({
        // 我们先声明一个对象x，并对其初始化
        int x = a > b? a + 1 : b - 1;
        a += x;
        b -= x;
        a + b;
    });

    printf("a = %d, b = %d\n", a, b);

    // 以下是将复合语句表达式作为for语句中用于初始化的表达式的例子
    for(int i = ({
```

```
        int x = a - b;
        a -= 6;
        x -= 10 * b;
    }); i < a; i++)
    {
        printf("Hello, world: %d\n", i);
    }

    // 复合语句表达式也属于一种表达式，因此是不能直接作为左值的，但可以以引用的方式出现，
    // 然后作为 * 间接操作符的操作数来使用
    *({
        &a;
    }) = 100;

    printf("a = %d\n", a);
}
```

代码清单17-2给出了三种复合语句表达式的使用场景。第一个例子是将复合语句表达式直接用于赋值操作符的场景。整条复合语句表达式的计算结果就是最后“a+b；”这条表达式语句的值。第二个例子是将复合语句表达式用于for语句中对第一条子语句中临时变量i的初始化。第三个例子则直接通过复合语句表达式最终所得到的对象a的地址作为间接操作的操作数进行访问。大家在这里就能看到，通过在一条复合语句两边加上左右圆括号就能使它作为一条表达式来使用，这在某些场合还是比较方便有用的。

17.2　声明语句块作用域的跳转标签

我们在11.1.2节中已经学习到，C语言中的跳转标签具有比较特殊的函数作用域，也就是说跳转标签无论在当前函数的哪个语句块中都可见。而为了对C语言能有更好的内部模块封装性，GNU语法扩展中添加了__label__关键字（label前后各有两条下划线），用此关键字声明的标签仅在当前语句块作用域可见，因而也被称为局部跳转标签。此外，在其他语句块作用域中也可声明同名的局部跳转标签，既不会引发符号重定义的冲突，而且也能正常跳转。代码清单17-3展示了局部跳转标签的使用。

代码清单17-3　局部跳转标签的使用

```
#include <stdio.h>

#define MY_INNER_PROCESS(a) {   \
    __label__ MY_JUMP, YOUR_JUMP, FINAL_PROCESS;    \
                                                    \
    int b = a - 5;              \
                                \
    if(b > 0)                   \
        goto MY_JUMP;           \
    else                        \
        goto YOUR_JUMP;         \
                                \
MY_JUMP:                        \
    printf("b is above zero! b = %d\n", b);         \
    goto FINAL_PROCESS;         \
                                \
YOUR_JUMP:                      \
    printf("b is not above zero! b = %d\n", b);     \
                                \
FINAL_PROCESS:                  \
    puts("This is a macro!!");                      \
}


int main(void)
{
    int a = 10;

    if(a > 0)
    {
        // 这里声明a > 0分支语句块中的局部跳转标签
        __label__ MY_JUMP, YOUR_JUMP, FINAL_PROCESS;
```

```
    int b = a - 5;

    if(b > 0)
        goto MY_JUMP;
    else
        goto YOUR_JUMP;

MY_JUMP:
    printf("b is above zero! b = %d\n", b);
    goto FINAL_PROCESS;

YOUR_JUMP:
    printf("b is not above zero! b = %d\n", b);

FINAL_PROCESS:
    puts("This is a > 0 path~");
}
else
{
    // 这里也同样声明a <= 0分支语句块中的局部跳转标签,
    // 且名字与a > 0分支语句块中的完全相同
    __label__ MY_JUMP, YOUR_JUMP, FINAL_PROCESS;

    int b = a + 5;

    if(b > 0)
        goto MY_JUMP;
    else
        goto YOUR_JUMP;

MY_JUMP:
    printf("b is above zero! b = %d\n", b);
    goto FINAL_PROCESS;

YOUR_JUMP:
    printf("b is not above zero! b = %d\n", b);

FINAL_PROCESS:
    puts("This is a < 0 path!");
}

a = -a;

// 调用宏, 这里MY_INNER_PROCESS宏中也有与上述两个语句块相同的跳转标签名,
// 而且结构也基本相同
MY_INNER_PROCESS(a);

a = ({
    __label__ MY_JUMP, YOUR_JUMP, FINAL_PROCESS;

    int b = a + 5;

    if(b > 0)
        goto MY_JUMP;
    else
        goto YOUR_JUMP;

MY_JUMP:
    printf("b is above zero! b = %d\n", b);
    goto FINAL_PROCESS;

YOUR_JUMP:
    printf("b is not above zero! b = %d\n", b);

FINAL_PROCESS:
    a + b;
});
```

```
    printf("a = %d\n", a);
}
```

代码清单17-3这个代码示例详细说明了局部跳转标签的使用与实际效果。代码清单17-3也是分了3个小例子。第1个例子用一个if-else条件分支设置了两个不同的语句块，在里面声明了相同名字的局部跳转分支，然后执行差不多功能的操作。第2个小例子是比较有用的，使用宏来封装语句块，并且在该语句块中含有局部跳转操作，这样就能很清晰地呈现局部跳转标签的威力了：如果没有局部跳转标签，那么宏定义中的跳转标签将是函数作用域的，倘若这个宏在某个函数中被使用2次，那么就会出现跳转标签重定义的情况。有了局部跳转标签，那么宏也能更好地起到封装抽象类似代码块的作用。第3个例子则结合了上一节我们提到的复合语句表达式，我们可以看到局部标签也能在复合语句表达式中很好地工作。

17.3　跳转标签作为值

在标准C语言中跳转标签只能用于goto语句，而无法单独作为一个表达式所使用。而在GNU语法扩展中，我们能够通过&&单目操作符取跳转标签的地址。这里的&&不是逻辑与，而是用于取跳转标签的地址的前缀操作符，要跳转标签的地址可用于跳转到它所指定的代码处。正是因为它的计算结果是代码指令的地址，所以不具有任何具体类型，在GNU语法扩展中只能用void*类型表示&&单目表达式的类型。

此外，goto语句的表达也相应做了扩展，除了可直接跟跳转标签名之外，还能跟指向标签地址的指针做一次间接引用后的表达式。这一点非常有意思，指向跳转标签地址的指针是void*类型，对它做一次间接操作后，表达式的类型就变为void了。因此，goto后面跟的是一个void表达式，这也与标签的类型相吻合，跳转标签也不具备任何实体类型。

代码清单17-4展示了跳转标签作为值使用的方式。

代码清单17-4　跳转标签作为值使用

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    // 声明一个局部跳转标签LOCAL_JUMP
    __label__ LOCAL_JUMP;

    // 声明一个指向跳转标签的指针p，并初始化为指向函数作用域的NORMAL_JUMP标签地址
    void *p = &&NORMAL_JUMP;

    // 声明一个指向跳转标签的指针q，并初始化为指向语句块作用域的NORMAL_JUMP标签地址
    void *q = &&LOCAL_JUMP;
```

```
    int a = 10;
    int *buffer = NULL;

    if(a <= 0)
        goto *p;    // 跳转到指针p所指向的标签

    buffer = malloc(sizeof(*buffer) * a);
    if(buffer == NULL)
        goto *p;    // 跳转到指针p所指向的标签

    for(int i = 0; i < a; i++)
        buffer[i] = i + 1;

    if(buffer[0] + buffer[a - 1] < 100)
        goto *q;    // 跳转到指针q所指向的标签

    printf("The value is: %d\n", buffer[0] + buffer[a - 1]);

LOCAL_JUMP:

    printf("buffer[a / 2] = %d\n", buffer[a / 2]);

NORMAL_JUMP:

    if(buffer != NULL)
        free(buffer);

    puts("Program complete!");
}
```

有了可取跳转标签值的能力，使得C语言在跳转上也有了直接跳转与间接跳转两种方式，这种语法对称性就与函数的直接调用与通过指向函数的指针做间接调用一样。除了个别较低级的单片机微控制单元（MCU）不支持函数间接调用与地址间接跳转之外，大部分处理器都能直接在指令上支持这两种跳转与函数调用方式。尽管在goto语句本身的使用上，有不少程序员秉持各自的意见和看法，但从语法体系结构来说，能支持这种通过指针做间接跳转的方式还是很赞的。

17.4　嵌套函数

我们在第9章介绍函数的时候已经提到过，C语言标准规定我们必须在文件作用域定义一个函数。然而在GNU语法扩展中，我们可以在一个函数中定义一个嵌套函数（Nested Function）。嵌套函数不具有任何连接，如果要在函数内声明某一嵌套函数的话，可以使用auto存储类说明符，表示该函数无连接，但不能使用extern或static存储类说明符。嵌套函数可被视为共享其外部函数的执行上下文（包括栈空间），这样使得嵌套函数可以访问其外部函数所声明的局部对象。嵌套函数可以通过函数指针的形式传递到外部，比如外部函数的返回类型是指向一个函数的指针，然后将其内部所定义的嵌套函数返回出去。如果嵌套函数中没有引用任何外部函数中所声明的局部对象，那么当外部函数返回之后，通过函数指针来调用嵌套函数还是安全的；倘若嵌套函数引用了外部函数所声明的局部对象，那么当外部函数返回之后，其执行上下文由于被回收，所以再通过函数指针来间接调用其嵌套函数，可能会引发运行时异常，或是得到意想不到的运行结果。代码清单17-5列举了嵌套函数的定义与使用，并分析其相关的执行上下文。

代码清单17-5　嵌套函数的定义与执行

```
#include <stdio.h>
#include <stdint.h>

/**
 * Test函数用于测试嵌套函数的特性
 * @param p 参数p用于输出指向局部对象i的指针
 * @return 返回指向嵌套函数的指针
```

```
*/
static void (*Test(int **p))(int)
{
    int i = 10, a = 100;

    // 分别输出局部对象i和a的地址
    printf("In %s, address of i: 0x%.16tX\n", __func__, (uintptr_t)&i);
    printf("In %s, address of a: 0x%.16tX\n", __func__, (uintptr_t)&a);

    // 这里定义Test函数中的嵌套函数InnerTest,
    // 它具有一个形参arg，这里的类型说明符auto可省
    auto void InnerTest(int arg)
    {
        // 这里对外部局部对象i的修改也将会影响到外部函数的执行
        printf("The value is: %d\n", ++i + arg);

        // 这里观察嵌套函数中外部函数的局部对象i的地址以及嵌套函数形参arg的地址
        printf("In %s, address of i: 0x%.16tX\n", __func__,
               (uintptr_t)&i);
        printf("In %s, address of arg: 0x%.16tX\n", __func__,
               (uintptr_t)&arg);
    }

    // 调用嵌套函数Innertest
    InnerTest(a);

    // 我们这里将会发现i的值加了1
    printf("i = %d\n", i);

    // 如果形参p不空，那么将i的地址赋值给它
    if(p != NULL)
        *p = &i;

    // 返回嵌套函数的地址
    return &InnerTest;
}

static int FetchData(int count)
{
    int array[count];

    for(int i = 0; i < count; i++)
        array[i] = i + 1;

    int sum = 0;

    for(int i = 0; i < count; i += 2)
        sum += array[i];

    // 在此函数中再次调用Test函数，观察运行时的执行情况
    // Test(NULL);

    return sum;
}

int main(void)
{
    // 声明指针对象pTrace，将用于指向Test函数中局部对象i的地址
    int *pTrace = NULL;

    // 声明函数指针pFunc，直接调用Test,
    // 将返回的嵌套函数为它初始化
    void (*pFunc)(int) = Test(&pTrace);

    // 我们这里通过打开或屏蔽FetchData函数中对Test函数的调用来观察执行状态
    int data = FetchData(100);
    printf("data = %d\n", data);

    // 先输出调用pFunc之前，Test函数中局部对象i的值
```

```
        printf("Original i = %d\n", *pTrace);

        // 如果FetchData函数中执行了对Test函数的调用,
        // 那么在这里执行pFunc的调用可能会引起程序崩溃
        pFunc(data);

        // 再输出调用pFunc之后，Test函数中局部对象i的值
        printf("Original i = %d\n", *pTrace);
    }
```

代码清单17-5中，在Test中所定义的嵌套函数InnerTest引用了该嵌套函数的外部函数Test所声明的对象i，然而我们在整个运行过程中发现执行一切正常。通过打印Test函数局部对象i的地址，我们发现该对象所占用的栈空间始终被维护着，没有遭到破坏。但是，当我们把FetchData函数中注释掉的Test（NULL）；重新打开时，再运行就可能会发生异常，除非此时把main函数中的pFunc（data）；这条语句给屏蔽掉。由于在调用Test之后，其嵌套函数InnerTest的执行上下文发生了变化，原有维护着的局部对象也无法继续维护，所以再次通过pFunc调用嵌套函数可能引发异常。

由于嵌套函数的执行上下文无法使用其他手段进行保护，所以它无法作为一个闭包传递到外部环境使用。此外，目前也只有GCC才支持嵌套函数定义，而LLVM Clang社区也明确指出将很有可能永远不会支持嵌套函数这个语法特性，取而代之的是，Clang编译器已经实现了更先进的Blocks语法，我们将在第18章做详细介绍。所以，笔者这里建议各位尽量避免将嵌套函数用在实际商业化的项目工程中，一来适用范围不大，二来可移植性也不高。

17.5　使用typeof来获取对象类型

GNU扩展语法特性中，笔者认为typeof操作符是贡献最大的语法特性之一。通过typeof，我们可以在编译时获取到指定对象的类型，并且typeof表达式可作为类型用于声明其他对象。我们可以通过实现更简洁的接口，实现更抽象的功能模块，从而使我们的C语言代码更为精简！

typeof与sizeof、_Alignof操作符具有相类似的特性。首先，它们一般都是在编译时进行计算，而不是在运行时，也不是在预处理阶段；其次，如果其操作数是一个可变修改类型或具有这种类型的表达式，那么对这些表达式的计算则要放到运行时。另外，typeof还有一个强大特性是，其操作数可以是一个函数，并且当其操作数是一个函数名时，整个表达式就表示为一个函数类型，而不是指向该函数的指针类型。这一点需要大家注意。

我们先通过代码清单17-6来看看typeof操作符的一般使用。

代码清单17-6　typeof操作符的一般使用

```
#include <stdio.h>
#include <stdint.h>

// 定义函数Func1
static void Func1(int a)
{
    printf("a = %d\n", a);
}

// 通过typeof来声明一个函数Func2，其返回类型以及参数列表都与Func1一样
static typeof(Func1) Func2;

// 定义函数Func2
```

```
static void Func2(int b)
{
    printf("b = %d\n", b);
}

// 定义打印当前基本数值类型的对象的类型的宏
#define PRINT_TYPE(expr)    _Generic((expr), float:puts("float type"),  \
                                    double:puts("double type"),         \
                                    default:puts("int type"))

// 利用typeof来定义泛型的取绝对值。这里将0强制转为expr的类型，然后进行比较
#define GENERAL_ABS(expr)   ((expr) >= (typeof(expr))0? (expr) : -(expr))

int main(int argc, const char * argv[])
{
    // 这里使用typeof(&Func1)表示指向Func1函数的指针类型
    typeof(&Func1) pFunc = &Func1;
    pFunc(100);

    pFunc = &Func2;
    (*pFunc)(200);

    // 我们这里先借用下一节将会描述的__auto_type来萃取表达式的类型，
    // 这样可以验证我们之前定义的宏没有破坏原有的对象类型
    __auto_type i = GENERAL_ABS(-1);
    __auto_type f = GENERAL_ABS(-1.5f);
    __auto_type d = GENERAL_ABS(-2.25);
    printf("i = %d, f = %f, d = %f\n", i, f, d);
    PRINT_TYPE(i);
    PRINT_TYPE(f);
    PRINT_TYPE(d);

    // typeof表达式可以嵌套，因为typeof的操作数既可以是一个表达式，也可以是一个类型
    typeof(typeof(100)) const a = 100;

    // 这里用typeof(a)表示为一个const int类型
    typeof(a) array[4] = (typeof(a)[]){1, 2, 3, 4};

    // 这里用typeof(array)表示一个const int[4]类型
    typeof(array) arr2 = {5, 6, 7, 8};

    // 这里声明了一个指向const int[4]数组的指针类型，即const int(*)[4]
    typeof(array) *pArray = &array;

    // 这里与上一条语句一样，同样表示const int(*)[4]类型
    typeof(&arr2) pArray2 = &arr2;

    int n = 5;

    // 这里声明了一个变长数组varr，其类型为int[n]，n == 5
    // 然后变量n的值变为了6
    int varr[n++];

    // 这里用varr声明了一个变长数组，其类型为int[3][n]，n == 5
    typeof(varr) varr2[3];
    printf("varr2 count: %zu\n", sizeof(varr2) / sizeof(varr2[0]));
    printf("size of varr2[0]: %zu\n", sizeof(varr2[0]) / sizeof(varr2[0][0]));

    // 声明了一个指向int[n]（n == 5）的变长数组的指针对象pv，
    // 其类型为：int(*)[n]，并用varr2的起始地址为它初始化
    typeof(varr) *pv = varr2;

    // 这里的空声明也是合法的，但因为不产生任何副作用，
    // 所以这里对表达式 pv[++n] 不进行计算，所以n的值也没有变化
    typeof(pv[++n]);

    // 这里用表达式 pv[++n] 的类型声明了一个变长数组对象varr3
    // varr3的类型为：int[n]（n == 5），这里变量n的值变为了7
    typeof(pv[++n]) varr3;
```

```
    printf("n = %d\n", n);
    printf("size of varr3: %zu\n", sizeof(varr3) / sizeof(varr3[0]));

    // 这里我们利用了GNU扩展语法的取跳转标签地址，用于表示const void*类型
    typeof((const void*)&&GOTO_LABEL) p = &a;

    if(*(const uint8_t*)p < 200)
        goto GOTO_LABEL;

    puts("Hello, world!");

GOTO_LABEL:

    printf("The value is: %d\n", (*pArray)[0] + pArray2[0][3]);
}
```

代码清单17-6中我们可以看到typeof操作符的强大威力。它不仅可对对象与函数进行操作，而且还能作用于具体类型，只要是合法的对象、函数以及类型，包括跳转标签的地址，这些都能顺利地转换为相应的类型，并且包含类型限定符。此外，我们还看到了使用typeof来定义泛型取绝对值操作。由于通过取宏参数表达式的类型来转换数值0，所以我们无需关心每种类型所对应的0的形态（比如单浮点型的0需要表达为0.0f，双精度浮点型的0需要表达为0.0），这样我们就能十分简单地实现对任一基本数值类型做取绝对值操作了。这要比通过C11标准所引入的泛型选择表达式根据每种特定类型做特定函数调用的语句表达要简洁多了。

typeof的威力还不仅仅体现在上述这些基本的使用上，我们在第6章描述结构体与联合体的时候提到过，如果在结构体内或联合体内定义一个嵌套的命名结构体与联合体，那么该嵌套的结构体或联合体仍然与其外部的结构体或联合体同处于相同的作用域。如果在文件作用域定义了一个结构体，并且该结构体内又定义了一个嵌套的命名结构体，那么势必就影响了该文件作用域的名字空间，倘若该结构体是定义在头文件中，并且要被引入到其他源文件中使用，那么所造成的污染会更大。因此我们一般不建议

在结构体中定义嵌套命名结构体，而是使用匿名嵌套结构体或联合体，然后直接用它声明一个对象作为外部结构体或联合体的一个成员。在标准C11中，我们是无法获取一个结构体或联合体中嵌套定义的匿名结构体或联合体的具体类型的，但在GNU扩展语法中，这将成为可能！代码清单17-7将为大家展示这种奇迹。

代码清单17-7　通过typeof来获取结构体中的匿名嵌套结构体类型

```
#include <stdio.h>

struct Frame
{
    struct
    {
        float x, y;
    }point;

    struct
    {
        float width, height;
    }size;
};

// 以下联合体作为一个名字空间来使用
union Namesapce
{
    struct { int a, b; } class1;
    struct { float c, d; } class2;
    struct { double e, f; } class3;
};

// 我们通过定义一个宏来方便获取一个结构体或联合体中的嵌套类型
#define FETCH_STRUCT_SUBTYPE(name, type)    typeof((struct name){}.type)
#define FETCH_UNION_SUBTYPE(name, type)     typeof((union name){}.type)

int main(int argc, const char * argv[])
{
    // 我们通过在typeof中构造一个匿名Frame结构体对象来获取其point的类型
    typeof((struct Frame){}.point) point = { 10.0f, 20.0f };

    // 我们通过在typeof中使用类型投射操作来访问Frame中的成员size，获取其类型
    typeof(((struct Frame*)NULL)->size) size = { 90.0f, 100.0f};

    struct Frame frame = { point, size };

    // 我们通过FETCH_UNION_SUBTYPE宏来获取Namespace中class2的类型
    FETCH_UNION_SUBTYPE(Namesapce, class2) object = {frame.point.x,
        frame.size.height};

    printf("object.c = %f, object.d = %f\n", object.c, object.d);
}
```

代码清单17-7展示了如何通过typeof操作符去获取结构体或联合体中嵌套定义的匿名结构体或联合体类型，然后跟普通的结构体与联合体那样去声明对象，正常使用。我们可以看到，通过这种方式也能在很大程度上避免名字空间的污染，尤其在做一些通用第三方库的场合会十分有用。

17.6　使用__auto_type做类型自动推导

从GCC 4.9版本以及Clang 3.8版本起，两者均引入了十分现代化的编程语言特性——类型自动推导。类型自动推导这一特性早先在一些具有动态特性的编程语言上用得很多，比如JavaScript、C#等，后来C++11标准引入了这一特性，在比较新的Swift编程语言上则直接以类型推导为主进行代码编写。GNU语法扩展引入了__auto_type关键字（前面包含两条下划线）用于声明一个对象，该对象类型将通过对它初始化的初始化器的类型进行推导。因此大家要注意的是，如果要通过类型推导来声明一个对象，那么必须为该对象直接初始化，否则该对象的类型在声明时是未知的，这将导致编译错误。

通过类型自动推导来声明对象能简化一些代码，此外对于相似代码的抽象也很有帮助。代码清单17-8展示了类型自动推导的使用方式及其便利之处。

代码清单17-8　GNU语法扩展的类型自动推导

```
#include <stdio.h>

// 为了方便起见，我们将__auto_type定义为obj，这样更为简洁
#define obj     __auto_type

// 我们利用类型推导来定义一个交换两个变量值的宏
#define GENERAL_SWAP(a, b)  {   __auto_type tmp = (a);  \
                                (a) = (b);              \
                                (b) = tmp;  }

int main(int argc, const char * argv[])
{
    obj a = 10, b = 20;
    GENERAL_SWAP(a, b);
    printf("a = %d, b = %d\n", a, b);
```

```
    obj c = 1.5f, d = -2.5f;
    GENERAL_SWAP(c, d);
    printf("c = %f, d = %f\n", c, d);

    struct MyStruct { int i; float f; };
    obj s = (struct MyStruct){1, 0.5f};
    obj t = (struct MyStruct){100, -10.25f};
    GENERAL_SWAP(s, t);
    printf("sum of s = %f, sum of t = %f\n", s.i + s.f,
           t.i + t.f);

    // 这里要注意的是，__auto_type不能对一个声明的对象做部分类型推导，
    // 而必须是完整类型。即用__auto_type声明的对象只能以单独的标识符的形式出现。
    // 所以这里我们不能将arr声明为：obj arr[] = { 1, 2, 3, 4 };，
    // 并且初始化器必须是一个匿名数组对象，以提供完整的类型信息
    obj arr = (int[]){ 1, 2, 3, 4 };

    const obj x = arr[2];
    // 这里使用类型推导时，用作初始化器的表达式x的类型限定符会被忽略，
    // 就跟普通的 int y = x; 这种声明方式一样
    obj y = x;
    y++;
    printf("y = %d\n", y);

    // 这里声明的指针对象p的类型为const int *类型
    const obj *p = arr;

    // 这里声明的指针对象q是const int* const类型，
    // 由于通过&x推导出的类型是const int*类型，
    // 所以如果单单用obj q来声明就是const int*类型。
    // 然后这里又用const去修饰指针对象q，那么q的类型就变为const int* const了
    const obj q = &x;

    q = NULL;    // 这句是错误的
    *q = 0;      // 这句也是错误的

    p = NULL;    // 这句没问题
    printf("*q = %d\n", *q);
}
```

从代码清单17-8可以看到，我们通过类型推导能用更简洁的方式来实现交换两个任意数据类型对象的值，这个比typeof的表达方式还要更简洁一些。另外，我们在代码清单17-8中将__auto_type定义为obj，这在表达上也更为简略，代码看上去也更舒服。最后又通过指针与数组对象的类型推导来呈现出__auto_type的使用效果以及约束。

17.7　对复数操作的扩展

6.6节谈到了复数类型的使用。在C语言中，复数类型是一个比较奇妙的数据类型，它像一个结构体，然而我们却可以用比较自然的方式来表达一个复数，比如3.0+2.0i。另外，我们访问一个复数的实部的时候不是通过访问其成员属性，而是通过creal、crealf或creall来访问；访问虚部的时候则需要通过cimagf、cimag或cimagl来访问。GNU语法扩展引入了__real__操作符（real前后各有两条下划线）用于访问一个复数的实数部分；使用__imag__操作符（imag前后各有两条下划线）用于访问一个复数的虚数部分。这么做的好处是我们无需关心当前复数实部与虚部的类型，可以直接获取相应的值。此外，GNU语法扩展中也引入了整数复数，也就是复数的实部与虚部都是整数，尽管整数在复数操作过程中起不到什么作用，但这完善了整个复数类型系统，并且再度突显出__real__与__imag__操作符的优势。

GNU语法扩展还引入了~单目操作符作用于一个复数，用于计算该复数的共轭复数。在C99标准中，我们必须通过<complex.h>标准头文件中的conjf、conj或conjl来获取指定复数的共轭复数。代码清单17-9给出了GNU语法扩展对复数类型的延伸特性。

代码清单17-9　GNU语法扩展对复数的延伸特性

```
#include <stdio.h>
#include <complex.h>
```

```
int main(int argc, const char * argv[])
{
    // 这里声明了一个实部与虚部都是short类型的复数对象compi,
    // 在complex后面必须跟原生基本类型，不能跟typedef类型名
    // 因此这里只能用char、short、int等，不能用int8_t、int32_t之类的类型名
    complex short compi = 4 - 2i;

    // 对compi求它的共轭复数
    compi = ~compi;

    // 我们通过__real__与__imag__操作符来获取compi的实部与虚部
    printf("real = %d, imaginary = %d\n",
           __real__(compi), __imag__(compi));
}
```

代码清单17-9以一个简短的代码示例将GNU语法扩展对复数的延伸特性都列举了出来。

17.8　半精度浮点类型

由于在图像以及音视频处理领域，数字信号数据所需要的精度要求不高，比如像当前用得较多的图像像素格式为RGBA8888，也就是一个像素由4个分量构成，分别表示红、绿、蓝与透明度，每个分量占1个字节大小，每个分量的取值范围一般为从0到255。在很多专用处理器的处理过程中对这种数字图像数据用单精度浮点显然会浪费带宽，因此IEEE754标准协会在2008年新发布的标准中正式引入了半精度浮点（half-precision binary floating-point）。

半精度浮点数的规格化表达方式与单精度类似，它具有1位符号位，5位指数位，尾数则有11位，其中经指数偏差为15。

不过由于当前CPU对半精度的支持十分有限，像x86处理器在引入了AVX/XOP操作时（Intel处理器是在SandyBridge架构上才刚引入了AVX）才引入了对半精度浮点数数据存储格式的支持；而ARM处理器则更早一些，在ARMv7架构出炉时则通过NEON技术引入了对半精度浮点数据存储格式的支持。大家注意到，这些处理器在指令上仅仅是引入了对半精度浮点数据的存储表示，而没有提供任何算术计算操作。它们所支持的仅仅是将半精度浮点数与单精度浮点数之间能相互转换的特性，如果要对半精度浮点数做实际计算，则还需要软件库的支持。当然，目前在处理器上一般不太会对半精度数据做算术计算，而是将它们交给GPU或其他硬件加速器

进行处理。但是，像OpenCL这种主要做数据密集型计算的平台已经引入了对半精度浮点数算术计算的支持，只要当前操作环境支持。而像Apple在2014年新出的Metal API则直接对半精度浮点数加以支持，因为能在Metal上运行的GPU都能支持半精度浮点数的算术运算。因此GNU语法扩展引入了对半精度浮点数的支持也算比较及时。

在GNU语法扩展中，使用__fp16关键字（前面带有两条下划线）来声明一个半精度浮点数。我们需要注意的是，在遵循GNU语法扩展的C语言中，我们只能将一个半精度浮点数对象用于数据存储，而不能用于计算，除非有第三方库的支持，否则编译器也是直接会把该算术运算编译为先将半精度浮点转换为单精度浮点，计算完毕后再转回半精度浮点。代码清单17-10展示了使用半精度浮点数的简单例子。

代码清单17-10　使用半精度浮点数

```
#include <stdio.h>

int main(int argc, const char * argv[])
{
    float a = 10.5f;
    __fp16 h = a;

    // 如果此段代码不经过优化，那么这个printf函数对实参h做了两次操作：
    // 第一次是将半精度浮点转为单精度浮点第二次则是将单精度浮点转为双精度浮点
    printf("The half-floating number is: %f\n", h);

    // 这里的减法计算是先将半精度浮点数h转换为单精度浮点
    // 然后计算完之后再转回半精度浮点数
    h -= 0.5f;
    printf("h = %f\n", h);

    h = 100000.5f;
    // 由于100000.5已经超过了半精度浮点所能表示的范围，所以这里将打印inf，表示无穷大
    printf("The half-floating number is: %f\n", h);
}
```

通过代码清单17-10，我们能看到半精度浮点数在C语言中使用时的特

性了。如果不涉及到GPU或其他计算加速器，不要使用半精度浮点数。如果我们需要将数据传到GPU等加速器执行，那么我们可以先用单精度浮点数进行计算处理，完成之后再转成半精度浮点数存放到存储器之中。

17.9 长度为零的数组

在GNU语法扩展中，C语言能支持声明长度为0的数组。这在通常情况下意义不大，但是当它作为一个结构体最后一个成员时，则功能相当于之前提到的灵活数组作为结构体最后一个成员，而且比C99标准所引入的这个语法特性更为灵活。

长度为0的数组对象作为结构体的成员与C99标准引入的灵活数组对象作为结构体成员相比，在语法特性上十分相似，但有以下这些不同。

1）由于灵活数组对象是不完整类型，所以我们不能用sizeof操作符对它进行操作。而长度为0的数组可以作为sizeof操作符的操作数，并且整个sizeof表达式的计算结果为0。

2）灵活数组对象作为结构体成员时，该结构体必须至少含有一个命名非空成员对象（即该成员对象用sizeof计算结果必须大于0）。而长度为零的数组作为结构体成员时没有这一要求，也就是说长度为零的数组对象完全可以作为结构体中唯一的成员。

3）在C99标准中，带有灵活数组对象作为成员的结构体或包含这种结构体对象的联合体，不能作为另一个结构体的成员或一个数组的某个元素。而在GNU语法扩展中，这些都允许。

正因如此，GNU语法扩展也允许定义不含任何成员的结构体与联合

体，这样的结构体与联合体的大小为0个字节。代码清单17-11展示了长度为0的数组作为结构体成员与灵活数组作为结构体成员的用法比较示例。

代码清单17-11　长度为0的数组作为结构体成员VS灵活数组作为结构体成员

```
#include <stdio.h>

// GNU语法扩展也允许定义一个不含任何成员的结构体
struct Dummy
{

};

// 如果是以灵活数组对象作为结构体成员，那么该结构体中至少必须含有一个命名成员对象
struct FlexibleArrayStruct
{
    int a;          // 这里命名成员对象a不能缺省
    int flex[];
};

// ZeroArrayStruct结构体类型的定义没有问题
struct ZeroArrayStruct
{
    // 这里不需要含有一个命名非空成员对象
    int zero[0];
};


int main(int argc, const char * argv[])
{
    // GNU语法扩展允许直接声明一个长度为0的数组对象
    int arr[0];

    // 长度为零的数组对象arr的大小为0
    printf("The size is: %zu\n", sizeof(arr));

    // 不含任何成员对象的Dummy结构体类型的大小也为0
    printf("Dummy size is: %zu\n", sizeof(struct Dummy));

    // ZeroArrayStruct结构体大小也为0
    printf("ZeroArrayStruct size is: %zu\n",
            sizeof(struct ZeroArrayStruct));

    struct FlexibleArrayStruct *pFS = NULL;
    // 以下sizeof(fs.flex)这一表达式是错误的！因为pFS->flex是不完整类型，
    // 它不能作为sizeof操作符的操作数
    printf("size = %zu\n", sizeof(pFS->flex));

    struct ZeroArrayStruct *pZS = NULL;

    // 以下的sizeof(pZS->zero)表达式是没问题的，大小为0
    printf("size = %zu\n", sizeof(pZS->zero));

    // 含有灵活数组成员以及长度为0的数组成员的结构体不能直接用初始化器进行初始化，
    // 也不能用该结构体构造一个匿名结构体对象，
    // 所以这里先使用匿名数组对象，然后再转为指向结构体的指针
    pFS = (struct FlexibleArrayStruct*)(int[]){4, 0, 1, 2, 3};
```

```
    pZS = (struct ZeroArrayStruct*)(int[]){1, 2, 3, 4};

    int sum = 0;
    for(int i = 0; i < pFS->a; i++)
        sum += pFS->flex[i];

    printf("flex sum is: %d\n", sum);

    sum = 0;
    for(int i = 0; i < pFS->a; i++)
        sum += pZS->zero[i];

    printf("zero sum is: %d\n", sum);
}
```

17.10　对可变参数个数的宏的语法扩展

C99标准引入了可定义可变参数个数的宏的语法特性，但是正如我们在10.1.5节中所描述的，用于表示可变实参的标识符__VA_ARGS__只能完全替换掉可变实参部分，如果可变实参是空，那么它也就相当于一个空白符。这会引发要定义类似printf这种函数的问题。我们在10.1.5节中提到，如果我们要用宏来定义printf，我们只能用#define MY_PRINT（...）printf（__VA_ARGS__）这种形式。

而在GNU语法扩展中，引入了逗号操作符与##宏拼接符相结合的方式，可让##拼接符左侧的逗号根据__VA_ARGS__所替换的实参来定。如果__VA_ARGS__所替换的实参不缺省，那么逗号保留，如果为缺省，那么逗号也会被省去。因此，在支持GNU语法扩展的情况下，我们可以将上述的MY_PRINT定义为：#define MY_PRINT（fmt，...）printf（fmt，##__VA_ARGS__）。这么一来，当我们使用MY_PRINT（“Hello，world\n”）；时就不会出现编译错误了，因为此时可变实参列表为空，那么在替换__VA_ARGS__的时候由于它用##拼接符与前面的逗号拼接，形成“，##”的形式，所以前面的逗号也会被省去，这样就不会出现printf（“Hello，world\n”，）这种宏替换了。

此外，GNU语法扩展中还能直接用#define MY_PRINT（fmt，args...）printf（fmt，##args）这种宏定义形式。这里，args后面紧跟...表示该参数

是可变个数的参数列表，这样在其替换列表中可以直接用args来表示可变个数的实参。命名的可变个数参数显然更具表达力，而且也使得代码更为整洁。代码清单17-12展示了可变参数宏的扩展使用。

代码清单17-12　可变个数实参的宏的GNU扩展

```
#include <stdio.h>

// 使用匿名可变参数个数的宏参数
#define MY_PRINT(fmt, args...)  printf(fmt, ## args)

// 使用命名可变个数的宏参数
#define MY_LOG(fmt, ...)        printf(fmt, ## __VA_ARGS__)

// 将MY_EXPR作为一个逗号表达式的宏来使用
#define MY_EXPR(a, args...)     (a, ## args)

int main(int argc, const char * argv[])
{
    MY_PRINT("Hello, world!\n");
    MY_LOG("Hello, world!\n");

    MY_PRINT("The string is: %s\n", "yes");
    MY_LOG("The string is: %s\n", "no");

    // 相当于：int a = (10);
    int a = MY_EXPR(10);
    printf("a = %d\n", a);

    // 相当于：a += (0, 20);
    a += MY_EXPR(0, 20);
    printf("a = %d\n", a);
}
```

这个GNU语法扩展可谓是对C99标准的一个补完，使得可变参数个数的宏定义在语法体系上更加完备，而且在使用上也不会有什么漏洞。

17.11　case语句中使用范围表达式

GNU语法扩展中引入了一个十分便捷的case语句范围表达式，可以使得当前的case条件作用于某个范围，而不仅仅是一个值上。比如，case 1...5就表示值如果在1～5的范围内则满足条件，执行该case语句中的逻辑。这里，省略号...就作为一个范围操作符，其左右两个操作数之间必须至少要用一个空白符进行分隔，如果写成1...5这种形式会引发词法解析错误。范围操作符的操作数可以是任一整数类型，包括字符类型。另外，范围操作符的左操作数的值应该小于或等于右操作数，否则该范围表达式就会是一个空条件范围，它永远不成立。代码清单17-13展示了对case语句范围表达式的使用示例。

代码清单17-13　case语句范围表达式

```
#include <stdio.h>

int main(int argc, const char * argv[])
{
    int a = 1;
    const int c = 10;

    switch(a)
    {
        // 这条case语句是合法的，并且与case 1等效
        case 1 ... 1:
            printf("a = %d\n", a);
            break;

        // 这条case语句中的范围操作符的左操作数大于右操作数，
        // 因此它是一个空条件范围，这条case语句下的逻辑永远不会被执行
        case 2 ... 1:
            puts("Hello, world!");
            break;

        // 使用const修饰的对象也可作为范围操作符的操作数
        case 8 ... c:
            puts("Wow!");
            break;

        default:
```

```
            break;
        }

        char ch = 'A';
        switch(ch)
        {
            // 从'A'到'Z'的ASCII码范围
            case 'A' ... 'Z':
                printf("The letter is: %c\n", ch);
                break;

            // 从'0'到'9'的ASCII码范围
            case '0' ... '9':
                printf("The digit is: %c\n", ch);

            default:
                break;
        }
    }
```

我们通过代码清单17-13可以看到，case范围表达式即可充当一条if语句，像case 1...5就好比if（a>=1&&a<=5），而从表达形式上看则更为简洁。当然，范围操作符的操作数必须是一个编译时的常量，不能是一个变量。

17.12　投射到一个联合体类型

我们之前已经学到过，在C语言标准中，我们不能将某个对象转换为一个结构体或联合体类型，我们只能将某个对象的地址转换为指向一个结构体或联合体类型的指针。不过在GNU语法扩展中，我们却可以将一个对象转换为一个包含该对象类型的联合体类型。该语法扩展增加了对联合体类型的投射操作，其实也在一些场合简化了代码。代码清单17-14展示了联合体类型的投射操作代码示例。

代码清单17-14　联合体类型的投射操作

```
#include <stdio.h>
#include <math.h>

struct MyPoint
{
    float x, y;
};

/** 定义了一个名为UnionTest的联合体，其中包含了三个对象成员 */
union UnionTest
{
    int a;

    double d;

    struct MyPoint point;
};

/** 计算并打印出形参t中的点point到原点之间的距离 */
static void OutputDistanceToOrigin(union UnionTest t)
{
    float distance = sqrtf(t.point.x * t.point.x +
                           t.point.y * t.point.y);
    printf("Distance to origin: %f\n", distance);
}

int main(int argc, const char * argv[])
{
    int a = 1;

    // 这里通过对联合体UnionTest的投射操作，将整型对象a转换为UnionTest联合体类型
    union UnionTest un = (union UnionTest)a;
    printf("value is: %d\n", un.a);

    double d = 5.0;
```

```
    // 这里通过对联合体UnionTest的投射操作，将双精度浮点对象d转换为UnionTest联合体类型
    un = (union UnionTest)d;
    printf("value is: %f\n", un.d);

    struct MyPoint mp = { 3.0f, -4.0f };

    // 这里通过对联合体UnionTest的投射操作，
    // 将MyPoint结构体类型对象mp转换为UnionTest联合体类型
    un = (union UnionTest)mp;
    printf("x = %f, y = %f\n", un.point.x, un.point.y);

    // 这里可直接用联合体的投射操作将mp结构体转换为UnionTest联合体类型
    OutputDistanceToOrigin((union UnionTest)mp);
}
```

17.13　使用二进制整数字面量

在C语言标准中，我们通过使用前缀0表示一个八进制整数（比如：012表示十进制整数10）；通过0x或0X前缀表示一个十六进制整数或十六进制浮点数（比如：0x10表示十进制整数16，0x3.8p0表示十进制浮点数3.5）。而在GUN语法扩展中可以通过0b或0B前缀来表示一个二进制整数，比如：0b01011010表示十进制整数90，0B0011表示十进制整数3。

17.14　使用__attribute__指定函数、对象与类型的属性

我们此前在10.8节提到过，可以用#pragma预编译指示符来描述一段代码的特性。而在GNU语法扩展中，我们除了可以用#pragma预编译指示符之外，还能使用__attribute__（attribute前后各有两条下划线）说明符（specifier）所指定的属性来指明用它所修饰的函数、对象或类型的相关特性。当__attribute__用于修饰对象时，它就如同C语言语法体系结构中的类型限定符（type qualifier），跟const、volatile、restrict等属一类。当__attribute__修饰一个函数时，它就相当于一个函数说明符（function specifier），跟inline、_Noreturn属同一类。倘若__attribute__在函数定义中进行修饰，那么该限定符可以放在函数声明的最前面，也可以放在函数标识符前；而如果只是函数声明，那么该限定符除了上述两个位置之外，还能放在函数声明的末尾。当__attribute__修饰一个结构体、联合体或枚举类型时，该限定符只能放在类型标识符之前。

__attribute__说明符的基本语法为：

```
__attribute__ (( attribute-list ))。
```

这里要注意的是，__attribute__所指定的属性列表必须前后用双圆括号包围，属性列表中如果有多个属性，那么用逗号分隔。当__attribute__要用来修饰一组函数声明时，可以将它放在第一个函数声明的最前面，那么后续声明的函数都会受到此__attribute__的修饰。代码清单17-15展示了

__attribute__的基本使用以及其语法特性。

代码清单17-15　__attribute__的基本使用及语法

```
#include <stdio.h>
#include <stdint.h>
#include <stdalign.h>

/** 用__attribute__修饰一个命名结构体类型 */
struct __attribute__((packed)) Test
{
    int8_t a;
    int b;
};

/** 使用__attribute__修饰一个匿名结构体类型 */
struct __attribute__((packed))
{
    int16_t s;
    double d;
}sStruct;

/** 使用3个属性来修饰同一个函数，分别是4字节对齐、总是内联、纯函数 */
static int  __attribute__((aligned(4), always_inline, pure))
MyTestFunc(void)
{
    return 0b01100100;
}

// 对MyFunc函数进行声明，此时__attribute__说明符可以放在函数声明的末尾
extern void MyFunc(int a) __attribute__((pure));

/** 使用__attribute__来修饰一个形参对象 */
static void foo(int __attribute__((aligned(8))) a)
{
    // 在GNU语法扩展中，alignof的操作数可以是一个表达式，而不单单只是类型名
    printf("The alignment of a is: %zu\n", alignof(a));
}

// 这里声明了三个函数，这三个函数的返回类型都是int，并且都用aligned(8)的属性修饰。
// 而weak属性只作用于f1；pure属性只作用于f2；always_inline属性只作用于f3
__attribute__((aligned(8))) int
__attribute__((weak)) f1(int a, int b),
__attribute__((pure)) f2(int a, int b),
__attribute__((always_inline)) f3(void);

// 以下分别对这三个函数进行定义。在定义时，__attribute__可缺省
int f1(int a, int b)
{
    return a * a + b * b;
}

int f2(int a, int b)
{
    return a + b;
}

int f3(void)
{
    return 0123;
}
```

```
int main(int argc, const char * argv[])
{
    // 这里我们将会看到，struct Test类型的大小为5个字节
    printf("size of Test: %zu\n", sizeof(struct Test));

    // 这里我们将会看到，sStruct的大小为10个字节
    printf("size of sStruct: %zu\n", sizeof(sStruct));

    int a = MyTestFunc();
    printf("a = %d\n", a);

    foo(100);

    printf("f1 = %d\n", f1(3, 4));
    printf("f2 = %d\n", f2(3, 4));
    printf("f3 = %d\n", f3());

    // 这里将声明的指针对象p用aligned(16)属性来修饰，
    // 同时，指明(*p)的属性为aligned(4)
    int __attribute__((aligned(4))) * __attribute__((aligned(16))) p = &a;
    printf("align of p is: %zu\n", alignof(p));
    printf("align of *p is: %zu\n", alignof(*p));
}
```

代码清单17-15比较详细地介绍了__attribute__说明符的使用方法以及其语法特性。下面我们将分别介绍__attribute__可用来修饰函数、变量以及类型的常用属性。

17.14.1 __attribute__用于修饰函数的属性

以下将列举常用的函数属性，这些属性一般能用于大部分处理器环境，并且在GCC编译器以及Clang编译器上均能支持。

1.aligned（alignment）

aligned属性修饰一个函数时，用于指示该函数的首地址至少需要alignment个字节对齐。如果我们所指定的alignment字节数小于默认的对齐字节数，那么以默认的字节对齐为准。如果我们使用编译命令行选项“-falign-functions”对所有函数做全局的字节对齐指定，那么我们再用aligned

属性修饰某个函数时，该函数将以我们当前所指定的字节数做对齐。

我们这里要注意的是，用aligned属性修饰一个函数后，该函数实际的字节对齐数仍然以连接器的安排为准，因此该属性也是一个暗示性的属性，当然在大部分情况下，函数会满足我们所指定的最低字节个数对齐的要求，如果我们所指定alignment不太大的话。最后要注意的是，我们要指定函数的字节对齐要求必须使用__attribute__说明符，而无法使用C11标准所引入的_Alignas说明符，_Alignas只能用于修饰对象。代码清单17-16展示了aligned属性的大概使用方式。

代码清单17-16　aligned属性修饰函数的使用

```
#include <stdio.h>

#include <stdint.h>
#include <stdalign.h>

/** 指定func1首地址至少满足16字节对齐，这里再使用inline函数说明符也没问题 */
static inline void __attribute__((aligned(16))) func1(int a)
{
    printf("a = %d\n", a);
    printf("%s address: 0x%.16tX\n", __func__, (uintptr_t)&func1);
}

static void __attribute__((aligned(64))) func2(void)
{
    func1(100);

    printf("func1 alignment: %zu\n", alignof(func1));
    printf("%s alignment: %zu\n", __func__, alignof(func2));
    printf("%s address: %.16tX\n", __func__, (uintptr_t)&func2);
}

int main(int argc, const char * argv[])
{
    func2();
}
```

2.always_inline

用此属性修饰一个函数时，指示编译器当前函数总是内联。如果编译

器由于某些限制而无法将指定的函数做内联处理，那么在编译时就会报错。用always_inline属性修饰的函数可以通过一个函数指针做间接调用，编译器将根据编译优化选项以及上下文来判定此间接调用是否也能进行内联处理，但如果间接调用内联失败则不会引发编译错误。代码清单17-15已经含有一些对always_inline属性的使用，而代码清单17-17则进一步展示了always_inline属性的用法。

代码清单17-17　always_inline属性的进一步使用

```
#include <stdio.h>

/** 用always_inline属性修饰函数func */
static int __attribute__((always_inline)) func(void)
{
    return 100;
}

int main(int argc, const char * argv[])
{
    // 这里对func的函数调用会被内联
    int a = func();

    // 这里获取func的地址也完全没问题
    int (*pFunc)(void) = &func;
    printf("func address: 0x%.16zX\n", (size_t)pFunc);

    a += pFunc();
    printf("a = %d\n", a);
}
```

3.flatten

用此属性修饰的函数，在该函数中调用的每一个函数都将尽可能地做内联处理。而用flatten属性所修饰的那个函数是否内联，则根据编译器当前的编译选项以及当前上下文来定。代码清单17-18展示了flatten的使用与效果。

代码清单17-18　flatten属性的使用与效果

```
#include <stdio.h>
static int func1(void)
{
    return 100;
}

static int func2(int a, int b)
{
    return a * a - b * b;
}

static void func3(int a)
{
    printf("a^2 = %d\n", a * a);
}

/** 使用flatten属性修饰函数FlattenTest */
static void __attribute__((flatten)) FlattenTest(void)
{
    int a = func1();
    printf("a = %d\n", a);

    a += func2(5, 4);
    printf("a = %d\n", a);

    func3(a);
}

int main(int argc, const char * argv[])
{
    FlattenTest();
}
```

从代码清单17-18我们可以在FlattenTest函数中设置断点，然后从反汇编中能看到，除了对标准库函数的printf调用没有内联外，对func1、func2、func3函数的调用全都内联了。由于printf函数属于库函数，在当前编译上下文中无法获得其具体实现，所以对它的调用无法内联。而在FlattenTest函数上面所定义的func1、func2和func3尽管没有显式地使用inline或__attribute__（（always_inline））去修饰，但在用flatten属性修饰的FlattenTest函数中仍然做了内联处理。

4.cdecl/stdcall/fastcall/ms_abi/sysv_abi

这些属性用在x86处理器系统平台上，分别表示所修饰的函数使用C函数调用约定、标准调用约定、快速调用约定，以及使用MSVC的函数调用约定或使用System-V的函数调用约定。当然，正如第15章已经介绍的，像cdecl、stdcall以及fastcall这三种调用约定都只能用于x86架构处理器的32位执行模式下。而ms_abi与sysv_abi则一般用于64位执行模式。正因GCC可同时支持ms_abi与sys_abi这两种调用约定，所以我们在类Unix系统环境下或在Windows系统环境下写C语言程序都能具备良好的可移植性，只要我们先定好两者都采用哪一种调用约定即可。

5.pure

用pure属性修饰的函数用来说明该函数除了返回值之外没有其他任何效果，并且该函数所返回的值仅仅依赖于函数的形参以及/或全局对象。用pure属性所修饰的函数可以用来辅助编译器做消除公共子表达式以及帮助做循环优化，使用这种函数就好比使用算术操作符一般。

用pure属性所修饰的函数体内不应该含有无限循环，不应该对volatile修饰的全局对象进行访问或是对多个线程所共享的全局对象进行访问，也不应该访问其他系统资源，比如对文件、套接字等进行操作。简而言之，对同一个使用pure属性修饰的函数连续做两次调用（如果该函数带有参数，那么两次调用应该用同样的实参），那么这两次调用所返回的结果应该始终是相同的。因此，用pure属性所修饰的函数也很容易让编译器做内联处理。代码清单17-19展示了pure属性的使用。

代码清单17-19　pure属性的使用

```
#include <stdio.h>

static int s = 10;

/** 以下定义了返回类型为void的pure函数 */
static void __attribute__((pure)) DummyFunc(void)
{
    puts("Hello, world!");
}

/** 以下定义的PureFunc函数可作为pure函数 */
static int __attribute__((pure)) PureFunc(int a, int b)
{
    return s + a * a + b * b;
}

/**
 * 以下定义的NormalFunc不能作为pure函数，
 * 因为函数内对全局对象进行了修改，从而使得返回结果会因为不同的全局对象的值而导致不同
*/
static int NormalFunc(int a, int b)
{
    s += 10;
    return s + a * a - b * b;
}

int main(int argc, const char * argv[])
{
    // 各位注意，DummyFunc在这里不会被调用
    DummyFunc();

    int a = PureFunc(3, 4);
    int b = PureFunc(3, 4);

    // 我们可以很自然地知道，使用相同的实参连续调用两次PureFunc，结果都是相同的
    printf("a = %d, b = %d\n", a, b);

    a = NormalFunc(5, 4);
    b = NormalFunc(5, 4);
    // NormalFunc不是一个pure函数，我们即便用相同的实参去做两次调用，结果也是不同的
    printf("a = %d, b = %d\n", a, b);
}
```

代码清单17-19中，DummyFunc是一个pure函数，但返回类型为void，在main函数中调用时由于编译器认为它对于程序执行不会造成任何影响，所以把它直接给消除了，我们在运行代码清单17-19时不会看到“Hello，world！”字符串的输出。因此大家要注意的是，一般pure函数的返回类型不应该是一个void，并且在调用时应该总是要有个对象去接收其返回值，否则该函数的调用就很可能被消除。另外，函数NormalFunc不是一个pure

函数，尽管我们使用__attribute__（（pure））对它进行修饰也不会有编译报错的情况，但一旦编译器在某些情况下把此函数误做优化（比如直接将它所访问的静态对象s的访问操作优化为直接存放在某个寄存器中，而使得其他线程对此修改操作不可见），那么执行效果与我们的预期将是不符的。

6.const

用const属性修饰的函数与用pure属性修饰的十分类似，不过const属性比pure更严格，它要求函数不能读全局对象。此外，用const属性修饰的函数的参数不能是一个指针类型，而且在用const属性修饰的函数内往往不能调用一个非const属性的函数。

7.constructor/destructor

constructor属性用于指定一个函数在程序进入main函数之前自动被程序加载器调用。用destructor属性修饰的函数则是在main函数执行结束之后，或是调用系统退出函数exit而退出当前应用之后，由系统自动调用。我们可以指定多个constructor函数以及destructor函数，它们的调用次序可能是随机的。不过我们可以为它们指定优先级来安排它们的调用次序，带有优先级的constructor属性为：constructor（priority）；带有优先级的destructor属性为：destructor（priority）。这里的priority是自己指定的一个整数，priority值越小，那么其优先级越高。对于constructor函数来说，优先级越高，那么就越先被执行；对于destructor函数来说，优先级越高则越晚被执行。

constructor对某些需要做全局初始化，但又无法直接在文件作用域通过指定常量的形式做初始化的对象进行初始化来说非常有用。代码清单17-20展示了constructor函数与destructor函数的用法。

代码清单17-20　constructor函数与destructor函数的使用

```
#include <stdio.h>
#include <string.h>

static int s = 10;

static int array[10];

static int sum = 0;

static void __attribute__((constructor(1))) MyInit1(void)
{
    puts("This is the first constructor!!");

    // 给array静态数组对象初始化
    int tmp[] = {s, s + 1, s + 2};
    memcpy(array, tmp, sizeof(tmp));
}

static void __attribute__((constructor(2))) MyInit2(void)
{
    puts("This is the second constructor!!");
    const int count = sizeof(array) / sizeof(array[0]);

    for(int i = 0; i < count; i++)
        sum += array[i];
}

static void __attribute__((destructor(1))) MyDeinit1(void)
{
    puts("This is the second destrctor!!");

    const int count = sizeof(array) / sizeof(array[0]);
    sum = 0;

    for(int i = 0; i < count; i++)
        sum += array[i];

    printf("sum = %d\n", sum);
}

static void __attribute__((destructor(2))) MyDeinit2(void)
{
    puts("This is the first destrctor!!");

    // 将array数组对象全都清零
    memset(array, 0, sizeof(array));
}

int main(int argc, const char * argv[])
{
    printf("sum = %d\n", sum);
}
```

我们执行代码清单17-20中的程序之后就能很清楚地看到两个constructor函数与两个destructor函数的执行次序了。

8.deprecated

用这个属性来修饰函数说明该函数已经被废弃了，如果程序员在自己的函数中调用此函数，那么编译器会报出警告。deprecated属性还能附加自己定制的消息，形式为：deprecated（message）。这里的message是一个C语言字符串字面量。代码清单17-21展示了deprecated属性的用法。

代码清单17-21　deprecated属性的使用

```
#include <stdio.h>

static void __attribute__((deprecated)) MyFunc(void)
{
    puts("This is MyFunc!!");
}

static void
        __attribute__((
            deprecated("Please use MyNewFunc instead")
               )) MyOldFunc(void)
{
    puts("This is MyOldFunc!!");
}

static void MyNewFunc(void)
{
    puts("This is MyNewFunc!");
}

int main(int argc, const char * argv[])
{
    // 这里编译器会报出警告—'MyFunc' is deprecated
    MyFunc();

    // 这里编译器会报出警告—
    // 'MyOldFunc' is deprecated: Please use MyNewFunc instead
    MyOldFunc();

    MyNewFunc();
}
```

9.dllimport/dllexport

这两个属性主要用于Windows系统以及塞班系统，指定具有外部连接的函数可作为动态连接库的符号进行导入或导出。__attribute__（（dllimport））相当于在Windows系统上的MSVC编译器中的__declspec（dllimport），dllexport属性也一样。详细可参考16.1.2节内容。

10.naked

这个属性主要用于ARM、x86_64、AVR等处理器平台。默认情况下，编译器会对一个函数实现自动生成某些编译器既定的上下文保护与恢复代码，前者称为prologue，后者称为epilogue。当用了此属性之后，函数实现就不会生成prologue以及epilogue代码。也就是说，用此属性修饰的函数对于我们来说就是一个纯粹的函数入口，我们可以在里面写内联汇编，并且即便函数返回类型不是void，我们也无需自己显式添加return语句，直接用汇编指令返回即可。另外，在x86_64环境下，naked函数中只能用内联汇编，而不能使用其他C语言语句。有了naked函数，我们就可以直接在C源文件里写汇编代码了，而且用内联汇编实现的C函数还能通过内联等处理做进一步的优化。代码清单17-22展示了naked函数的用法。

代码清单17-22　naked函数的用法

```
#include <stdio.h>

/** 定义一个naked函数MyASMFunc，该函数实现100 + (a - b)的功能 */
static int __attribute__((naked)) MyASMFunc(int a, int b)
```

```
{
    int a = 0; // 在naked函数中使用一般的C语言语句是错误的，
    // 这里应该使用纯内联汇编
    asm("sub %esi, %edi");
    asm("mov $100, %eax");
    asm("add %edi, %eax");

    // 最后不需要return语句，直接用RET指令做函数返回即可
    asm("ret");
}

int main(int argc, const char * argv[])
{
    int a = MyASMFunc(10, 20);
    printf("a = %d\n", a);
}
```

各位要注意的是，大家要执行代码清单17-22中的程序时必须要在x86_64环境中，因此需要确保当前系统是64位系统，并且编译器所用的输出目标是64位程序才能正常运行。

11.noinline

此属性与always_inline相反，用于指明一个函数不做内联处理。

12.nonnull

此属性可用于修饰带有指针类型形参的函数，指明该函数所有指针类型的形参不能为空。另外，我们也可以使用nonnull（arg-index，...）的形式来指定哪些指针对象的参数不能为空。其中，arg-index的最小值为1，所以计数从1开始，而不是从0开始。nonnull这个属性用来做低层的库或中间件非常有用，这样一来可以告诉上层应用开发人员哪些参数是不能为空的，二来还能防止上层应用开发人员误将不该为空的参数传空进去，否则会以编译器警告的方式呈现出来。代码清单17-23给出了nonnull属性的使用方式。

代码清单17-23　nonnull属性的使用

```
#include <stdio.h>

/** 定义一个函数MyFunc，指明该函数的所有指针类型的形参都不为空 */
static int __attribute__((nonnull)) MyFunc(int *p)
{
    // 这里对形参p做是否为空的判断会引发编译器警告，
    // 因为该参数已经被断言不能为空了
    if(p == NULL)
        return 0;

    return *p + 10;
}

/** 这里定义函数MyFunc2，并且指明其第一个形参p不能为空 */
static void __attribute__(( nonnull(1) )) MyFunc2(int *p, int *q)
{
    // 这里对形参p是否为空的判定会引发编译器警告
    if(p == NULL)
        return;

    // 这里对形参q是否为空的判定不会引发编译器警告
    if(q == NULL)
        *p = 0;
    else
        *p = *q + 1;
}

/** 这里定义了函数MyFunc3，并指明了第二个参数与第四个参数不能为空 */
static void __attribute__(( nonnull(2, 4) ))
        MyFunc3(int a, int *p, int b, int *q)
{

}

int main(int argc, const char * argv[])
{
    int a = 10;
    a = MyFunc(&a);
    printf("a = %d\n", a);

    // 这里如果传空来调用MyFunc函数，那么编译器会发出警告
    int b = MyFunc(NULL);
    printf("b = %d\n", b);

    MyFunc2(&b, &a);
    printf("b = %d\n", b);

    // 这里对第一个参数传空会引发编译器警告
    MyFunc2(NULL, &a);
    // 这里对第二个参数传空不会引发编译器警告
    MyFunc2(&a, NULL);
    printf("a = %d\n", a);
    MyFunc3(a, &a, b, &b);
}
```

13.returns_nonnull

该属性用于指明它所修饰的的返回值不会是一个空指针。returns_nonnull属性只能用于修饰返回类型为一个指针类型的函数。它的作用与nonnull属性类似，一般用于告诉上层应用开发人员，当前函数的返回值不会为空，因此不需要在自己函数内再去判定调用此属性修饰的函数之后的指针对象是否为空，从而使代码更为简洁。代码清单17-24展示了returns_nonnull属性的使用示例。

代码清单17-24　returns_nonnull属性的使用

```
#include <stdio.h>

static int s = 10;

/** 定义一个函数MyFunc，指明该函数的返回值不为空 */
static int* __attribute__((returns_nonnull)) MyFunc(int *p)
{
    if(p == NULL)
        return &s;

    return p;
}

int main(int argc, const char * argv[])
{
    int a = 1;
    const int *p = MyFunc(&a);
    printf("*p = %d\n", *p);

    p = MyFunc(NULL);
    printf("*p = %d\n", *p);
}
```

14.hot/cold：hot

该属性用来告知编译器，当前函数属于调用比较频繁的或是占用系统资源比较高的，属于性能热点（hot spot），编译器可以对此函数做深度优化。而cold属性则相反，它用于告诉编译器，当前函数很少执行，对运行性能影响微乎其微，编译器可将它安排到远离热点函数的子代码段中。编

译器将所有热点函数安排到一个段，将所有冷点代码安排到一个段对于运行时性能的影响还是不小的。尤其是当我们做高性能计算的时候，有时发现自己优化了一个函数后整体性能反而低了一点，这很可能说明你的代码改动对整个程序的代码结构安排造成了影响，使得代码执行时对指令Cache造成了不良影响。下面我们将讨论代码段这个话题。

15.section：section

该属性的用法是：section（“section-name”）。在基于GCC/Clang编译器的编译工具链中，会将编译好的代码放入text段。然而，就如我们上面第14条中所提及的，将代码完全交给连接器安排可能会导致几个调用挨得比较近的函数被安排在相互离得较远的地址位置，或者几个函数之间有调用关系的代码可能被隔得较远，这会导致这些代码在执行时造成指令Cache命中率低下，从而影响整体程序执行性能。为了避免因指令Cache的命中率过低而造成程序性能的影响，我们可以将这些调用挨得比较近的函数，或者彼此之间有调用关系的函数安排到同一个段中。比如像：

```
void funcA(void)
{
    funcB();
    funcC();
    funcD();
}
```

在funcA函数中依次调用了funcB、funcC和funcD，那么我们可以将这4个函数安排在同一个代码段中，或者将funcB、funcC、funcD安排在同一个代码段中。另外还有像：

```
void funcA(void)
{
    funcB();
}

void funcB(void)
{
    funcC();
}

void funcC(void)
{
    funcD();
}
```

像这种有彼此调用关系的函数可以看情况安排在同一个代码段。这里的调用关系为：funcA→funcB→funcC→funcD。尤其在同一个循环里所执行的一些函数放在同一个段中往往会有比较好的性能表现。

不同的处理器以及不同的操作系统，对于子段的定义方式可能不同，代码清单17-25展示的是具有mach-o目标格式的macOS系统下的子段指定方式。

代码清单17-25　macOS下对section属性的指定

```
#include <stdio.h>

static void __attribute__(( section("__TEXT,MySection") )) MyFunc1(void)
{
    puts("This is MyFunc1!");
}

static void __attribute__(( section("__TEXT,MySection") )) MyFunc2(void)
{
    puts("This is MyFunc2!");
}

static void Test(void)
{
    printf("MyFunc1 address: 0x%.16zX\n", (size_t)&MyFunc1);
    printf("MyFunc2 address: 0x%.16zX\n", (size_t)&MyFunc2);
    printf("Test address: %.16zX\n", (size_t)&Test);
}

int main(int argc, const char * argv[])
{
    MyFunc1();
    MyFunc2();
    Test();
```

```
    printf("main address: %.16zX\n", (size_t)&main);
}
```

在代码清单17-25中，我们发现对子段section的指定必须先包含一个段segment。这个段我们可以通过对当前main.c进行反汇编得到。用Xcode的反汇编见图17-1。

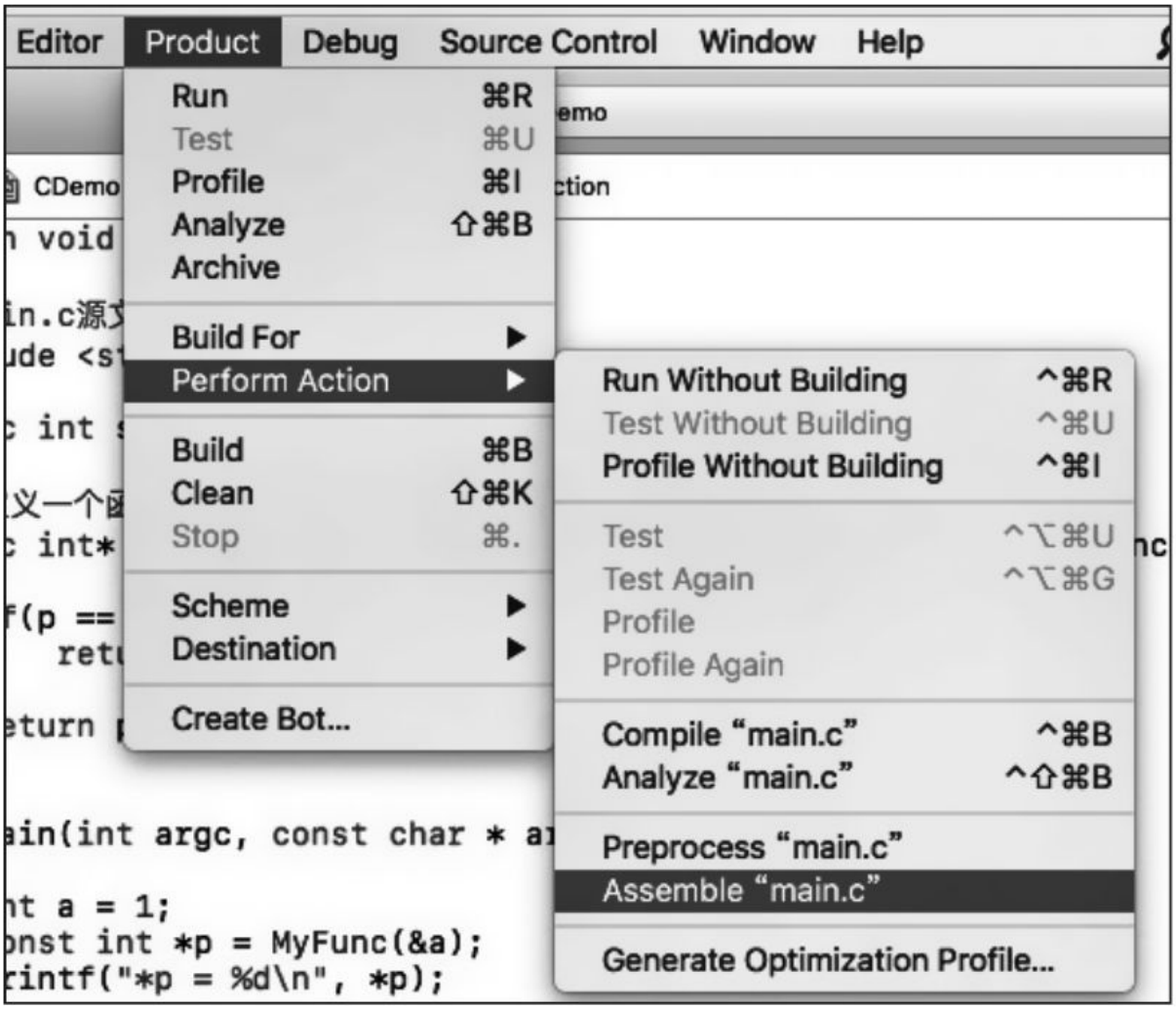


图17-1　利用Xcode查询当前文件的反汇编

首先在当前源文件的编辑状态下，在菜单栏选择“Product”，然后选中“Perform Action”，再选择“Assemble‘main.c’”即可，然后就会跳转到

main.c源文件的汇编代码界面。在汇编代码界面中，我们看到了第一行就是：.section__TEXT，__text，regular，pure_instructions。我们就用__TEXT作为segment名即可。

如果在Linux环境下，我们可以通过-S命令选项来输出当前C源文件对应的汇编文件。

代码清单17-25中，我们定义了名为MySection的子段。通过输出，我们发现MyFunc1与MyFunc2的地址非常接近，而没有指定子段的Test与main函数的地址则非常接近。在笔者测试环境下，myFunc1的地址为0x100004BF0；myFunc2的地址为：0x100004C10。Test函数的地址为：0x1000008D0；main函数的地址为：0x100000890。

16.used/unused

unused属性修饰一个函数意味着该函数很可能不会在整个程序中调用。现在的GCC以及Clang编译器对于不会被调用到的函数可能会发出编译器警告，如果我们对该函数指明了unused属性，那么编译器则不会再报出警告。而used属性则意味着该函数的代码必须在连接时生成，它不能被优化掉，无论该函数是否被其他函数调用。

17.visibility：visibility

该属性用于指示连接器当前函数符号对外部模块的可见性，一般用于动态连接库的制作。visibility属性的声明形式为visibility（“visibility-

type”），这里visibility-type有4个取值，分别为default、hidden、protected以及internal。

①default可见性是默认的符号连接可见性，如果我们不指定visibility属性，那么默认就使用此默认的可见性。默认可见性的对象与函数可以直接在其他模块中引用，包括在动态连接库中，它属于一个正常、完整的外部连接。

②hidden可见性指明了它所修饰的对象和函数具有“隐藏连接”。在同一共享目标文件（.so文件）中，具有隐藏连接的一个对象或函数的多个声明都将引用同一对象或函数。

③internal可见性与hidden可见性类似，不过它明确指示连接器，当前所修饰的对象或函数不能在其他模块中引用。不过根据不同的目标文件格式，对连接属性的定义可能会有些差异，比如在macOS中的mach-O格式的目标文件而言，hidden可见性同样也无法被外部模块所引用。

④protected可见性与default可见性相类似，不过该可见性指明了用它所修饰的对象或函数与当前模块所绑定，这意味着该对象或函数不能被另一模块所覆盖重写。

下面，我们将通过代码清单17-26与代码清单17-27来观察default可见性、hidden可见性以及internal可见性在macOS系统上的效果。各位也可以根据第16章所描述的内容在Linux系统上进行尝试。

代码清单17-26　macOS中对可见性属性的测试（动态库的代码）

```
//  lib2.c
/** 定义hidden可见性的函数MyHiddenTest */
int __attribute__((visibility("hidden"))) MyHiddenTest(void)
{
    return 100;
}

/** 定义internal可见性的函数internal */
int __attribute__((visibility("internal"))) MyInternalTest(void)
{
    return 200;
}

//  lib.c
#include <stdio.h>

// 这里对MyHiddenTest与MyInternalTest函数的声明不需要显式地添加可见性属性
// 它们引用之前声明过的相应函数
extern int MyHiddenTest(void);

extern int MyInternalTest(void);

/**这里定义默认可见性的函数MyExtDynFunc */
void MyExtDynFunc(void)
{
    puts("This is my so test!");

    printf("hidden value: %d\n", MyHiddenTest());
    printf("internal value: %d\n", MyInternalTest());
}
```

代码清单17-26展示了两个源文件，一个是lib2.c，另一个是lib.c，另外Xcode工程名用的是mydyn，编译构建后最终生成libmydyn.dylib。lib2.c定义了一个hidden可见性的函数MyHiddenTest，一个internal可见性的函数MyInternalTest。然后在lib.c源文件中对它们声明以及调用。笔者在测试的时候将最后生成的libmydyn.dylib动态连接库文件放置在了“/Users/zennychen/”用户根目录下。各位可以根据自己当前的环境来设置存放动态库的路径。

代码清单17-27　macOS中对可见性属性的测试（主函数）

```
// main.c源文件
```

```
#include <stdio.h>

// 此头文件包含了运行时动态加载动态库中外部符号的API
#include <dlfcn.h>

int main(int argc, const char * argv[])
{
    const char *path = "/Users/zennychen/libmydyn.dylib";

    // 使用dlopen函数加载动态库，返回动态库文件句柄
    void *dylibHandle = dlopen(path, RTLD_NOW);
    if(dylibHandle == NULL)
    {
        puts("dylib file not found!");
        return 0;
    }

    do
    {
        // 使用dlsym函数加载internal函数符号
        int (*pFunc)(void) = dlsym(dylibHandle, "MyInternalTest");
        if(pFunc == NULL)
        {
            puts("MyInternalTest function not found!");
        }
        else
        {
            int a = pFunc();
            printf("a = %d\n", a);
        }

        // 使用dlsym函数加载hidden函数符号
        pFunc = dlsym(dylibHandle, "MyHiddenTest");
        if(pFunc == NULL)
        {
            puts("MyHiddenTest function not found!");
        }
        else
        {
            int a = pFunc();
            printf("a = %d\n", a);
        }

        // 使用dlsym函数加载外部对象符号
        void (*p)(void) = dlsym(dylibHandle, "MyExtDynFunc");
        if(p == NULL)
        {
            puts("dyn_runtime object not found!");
            break;
        }

        p();
    }
    while(false);

    // 关闭动态库文件句柄
    dlclose(dylibHandle);
}
```

我们通过运行代码清单17-27的主程序之后就会发现，我们在动态库中所定义的hidden可见性的函数MyHiddenTest以及internal可见性的函数

MyInternalTest都无法在main函数中加载。我们只能加载到默认可见性的MyExtDynFunc函数。

18.weak

用weak属性修饰的具有外部连接的对象或函数具有一个弱符号。这意味着我们在同一模块中，或者在另一模块中定义相同外部符号名的对象或函数，可将具有weak属性的符号给覆盖重写。这对我们制作静态连接库来说十分有用。假如我们编写了一个静态连接库A.a，其中引用了一个第三方的静态连接库B.a，而最终应用开发者同时引用了A.a与C.a，而C.a这个静态连接库也包含了B.a的内容，此时如果不用weak属性修饰B.a的外部符号，那么就会引起外部符号重定义的连接时错误。为了避免引发重复包含的错误，将外部符号声明为weak属性是比较可靠的方式。

此外，我们通过weak属性还能判定当前应用是否包含了指定的静态连接库。比如，我们在自己的应用中用weak属性定义某个需要进行判定的外部函数，如果我们的应用包含了指定的静态连接库，那么静态连接库中非weak属性的相同外部符号将覆盖我们应用里自己写的weak属性的外部符号，从而能正常发挥作用。而如果没有使用相应的静态连接库，也不会引发外部符号未定义的错误。如果我们在主程序中含有对同一函数名包含多个weak属性的外部符号，并且没有对非weak属性的相应符号进行定义，那么具体使用哪个符号的定义将由连接器安排选择，这有可能是随机的。但如果有一个相应的非weak属性的符号存在，那么连接器必定用该非weak属性的外部符号所定义的内容。

代码清单17-28以及17-29将分别给出weak属性外部符号的使用以及效果。这两个例子都是在ubuntu 16.04中完成测试的。

代码清单17-28　weak属性的使用（静态库代码）

```
// libc.c源文件
int __attribute__((weak)) OverridenFunc(void)
{
    return 10;
}

const char *NonOverridenFunc(void)
{
    return "Hello";
}

// libc2.c源文件
int __attribute__((weak)) OverridenFunc(void)
{
    return 20;
}

// build.sh文件
gcc -std=gnu11 -c libc.c libc2.c
ar cr libStaticTest.a libc.o libc2.o
```

我们在输入代码清单17-28中的代码内容前可先新建一个文件夹，然后分别创建一个libc.c源文件、libc2.c源文件以及build.sh文件。这里大家要注意的是，对于具有相同函数名的不同weak属性函数的定义，必须将它们放置在不同源文件中，使得它们具有独立的翻译单元，从而拥有不同的文件作用域和上下文。如果将它们放在同一源文件中，那么在编译时就会报错。在libc.c与libc2.c中分别定义了名为OverridenFunc的具有weak属性的函数。而在libc.c源文件中还定义了一个具有正常外部连接的函数NonOverridenFunc，它的实现将不可被再次覆盖。

我们在用控制台进入该工程文件夹，然后输入bash build.sh，即可编译

生成静态库文件libStaticTest.a文件。随后，我们将这个静态库文件放入主程序的项目文件夹中。

代码清单17-29　weak属性的使用（主程序）

```
// main.c源文件
#include <stdio.h>

extern int OverridenFunc(void);

const char* __attribute__((weak)) NonOverridenFunc(void)
{
    return NULL;
}

int main(void)
{
    int a = OverridenFunc();
    printf("a = %d\n", a);

    const char *s = NonOverridenFunc();
    if(s == NULL)
    {
        puts("Static library is not loded!");
        return 0;
    }

    printf("s = %s\n", s);
}

// build.sh文件
gcc main.c -std=gnu11 -L./ -lStaticTest -o CTest
```

我们进入主程序main.c所在的文件夹，然后在控制台输入bash build.sh，即可编译生成最终的CTest可执行程序。我们运行这个程序之后就会发现OverridenFunc的返回结果会选择libStaticTest.a文件中的其中一个；而NonOverridenFunc的返回将始终是“Hello”字符串。

如果各位在macOS环境下，那么需要注意的是，从Xcode 8开始，我们在静态库工程下编译，默认的外部符号都默认为weak属性的，即便你不显式使用weak属性也是如此。因此这会使得我们做静态连接之后，即便在主

程序中定义了一个与静态库中相同名称的函数都不会有任何问题，即便这两者都没用weak属性去显式修饰。

17.14.2 __attribute__用于修饰对象的属性

用于修饰对象的属性与用于修饰函数的基本差不多，因此这里将简略介绍，碰到与函数有所不同的属性将会做详细介绍。

1）aligned：与用于修饰函数的aligned属性类似。这里各位要注意的是，C11标准所引入的_Alignas可用于修饰对象，因此我们最好用_Alignas来替代，这样对于可移植性而言也会更好一些。

2）deprecated：与用于修饰函数的deprecated属性类似。

3）mode：这个属性用于指定声明一个对象的数据类型，对于声明一个整数对象，如果通过mode属性，那么我们都只需要用int或unsigned类型即可，然后该属性将会把所声明的对象转换为能适应该模式长度的相应类型。GNU语法扩展支持三种数据类型模式的声明，分别为mode（byte），mode（word）以及mode（pointer）。mode（byte）表示数据类型是字节，往往对应char类型；mode（word）表示当前处理器的自然字长，在32位系统中一般为4个字节，64位系统中一般为8个字节；mode（pointer）表示当前处理器环境下，一个指针所占的字节个数。代码清单17-30将展示此属性的使用与效果。

代码清单17-30　对象属性mode的使用与效果

```
#include <stdio.h>

#define OUTPUT_TYPE(expr)   _Generic((expr),    \
                            signed char: puts("signed char"),   \
                            unsigned char: puts("unsigned char"),\
                            signed short: puts("signed short"), \
                            unsigned short: puts("unsigned short"),\
                            signed int: puts("signed int"),     \
                            unsigned int: puts("unsigned int"), \
                            signed long: puts("signed long"),   \
                            unsigned long: puts("unsigned long"),\
                            signed long long: puts("sllong"),   \
                            unsigned long long: puts("ullong"), \
                            default: puts("int"))

int main(int argc, const char * argv[])
{
    // 声明对象a，将它类型指明为字节模式
    int __attribute__(( mode(byte) )) a = 100;

    // 声明对象b，将它类型指明为机器字长模式
    unsigned __attribute__(( mode(word))) b = 10000;

    // 声明对象address，将它类型指明为地址模式
    int __attribute__((mode(pointer))) address;

    address = (size_t)&a;

    printf("a = %d\n", *(char*)address);
    printf("b = %tu\n", b);

    printf("size of a: %zu\n", sizeof(a));
    printf("size of b: %zu\n", sizeof(b));
    printf("size of address: %zu\n", sizeof(address));

    OUTPUT_TYPE(a);
    OUTPUT_TYPE(b);
    OUTPUT_TYPE(address);
}
```

4）packed：packed属性可用于修饰一个一般对象，也可用于修饰结构体或联合体内的成员对象，用于指示该对象应该具有最小可能的字节对齐。对于一个普通对象可能是一个字节；对于一个结构体中的位域，则可能是一个比特。如果我们同时用aligned属性或_Alignas来修饰同一对象，那么该对象的字节对齐将按照aligned属性来安排。代码清单17-31展示了packed属性的使用。

代码清单17-31　对象属性packed的使用

```
#include <stdio.h>
#include <stdint.h>
#include <stdalign.h>
#include <stddef.h>

struct Test
{
    int8_t a;

    // 如果仅仅对一个对象声明，__attribute__可以放到声明尾部
    // 这里的成员b为1个字节对齐
    int b __attribute__((packed));

    int16_t c;  // 成员c为2个字节对齐

    uint8_t d;

    // 这里的成员e为1个字节对齐
    int64_t __attribute__((packed)) e;
};// Test结构体最后以这里字节对齐数最大的2作为最终大小的倍数，所以最后填充了1个字节

int main(int argc, const char * argv[])
{
    int __attribute__((packed)) a = 100;
    int __attribute__((packed)) alignas(long) b = 20;

    printf("align of a: %zu\n", alignof(a));
    printf("align of b: %zu\n", alignof(b));
    printf("size of Test: %zu\n", sizeof(struct Test));

    // 各位可以通过以下成员的偏移地址就能观察到各成员的字节对齐情况了
    printf("offset of a: %zu\n", offsetof(struct Test, a));
    printf("offset of b: %zu\n", offsetof(struct Test, b));
    printf("offset of c: %zu\n", offsetof(struct Test, c));
    printf("offset of d: %zu\n", offsetof(struct Test, d));
    printf("offset of d: %zu\n", offsetof(struct Test, e));
}
```

这里各位要注意的是，在GCC更高版本以及Clang编译器中，对一般对象声明packed属性将可能会被忽略。而在结构体中对成员对象进行声明则没有问题。

5）section：用于修饰对象的section属性与用于修饰函数的类似，可参考用于修饰函数的介绍。

6）used/unused：用于修饰对象的used/unused属性与用于修饰函数的类

似，可参考用于修饰函数的介绍。

7）weak：用于修饰对象的weak属性与用于修饰函数的类似，可参考用于修饰函数的介绍。

8）dllimport/dllexport：用于修饰对象的dllimport/dllexport属性与用于修饰函数的类似，可参考用于修饰函数的介绍。

17.14.3 __attribute__用于修饰类型的属性

__attribute__也可用于修饰结构体、联合体以及枚举类型（有些还能用于修饰枚举常量）。用于修饰类型的属性与用于修饰函数和对象的也基本差不多。因此这里将简略介绍，碰到与函数或对象有所不同的属性将会做详细介绍。

1）aligned：aligned属性与用于修饰函数的类似，详细可参考函数属性部分。此属性可以用来修饰枚举、结构体、联合体以及类型定义。当然，最大可能的对齐字节数最终还是得看连接器，如果超过了连接器所能支持的对齐字节数，则以连接器的最大字节对齐数进行安排。代码清单17-32展示了aligned属性修饰用户自定义类型的示例。

代码清单17-32　aligned属性修饰用户自定义类型

```
#include <stdio.h>
#include <stdalign.h>

// 定义MyStruct，将它声明为8字节对齐
```

```
// __attribute__修饰一个用户自定义类型时，可以放在类型声明的末尾
struct MyStruct
{
    short a, b;

} __attribute__((aligned(8)));

// 定义MY_ENUM枚举类型，将它声明为8字节对齐
enum __attribute__((aligned(8))) MY_ENUM
{
    MY_ENUM_ONE,
    MY_ENUM_TWO
};

// 字节对齐属性用于修饰typedef名
typedef int MY_INT __attribute__((aligned(8)));
typedef short __attribute__((aligned(4))) MY_SHORT;

int main(int argc, const char * argv[])
{
    // MyStruct结构体由于8字节对齐，所以最终大小必须是其自身字节对齐的倍数。
    // 原本MyStruct为4个字节，这里需要在最后做4个字节的字节填充，扩充到8字节
    printf("The size of MyStruct: %zu\n", sizeof(struct MyStruct));

    enum MY_ENUM me = MY_ENUM_TWO;

    // 这里MY_ENUM枚举对象me仍然为4个字节
    printf("size of me: %zu\n", sizeof(me));

    // 不过这里MY_ENUM枚举对象me为8字节对齐
    printf("align of me: %zu\n", alignof(me));

    printf("align of MY_INT: %zu\n", alignof(MY_INT));
    printf("align of MY_SHORT: %zu\n", alignof(MY_SHORT));
}
```

2）packed：这里对类型使用packed属性时，可以针对结构体与联合体类型，效果与用于修饰对象的类似。

3）unused：这个属性与用于修饰函数和对象的类似，详细请参考修饰函数的unused属性。

4）deprecated：这个属性与用于修饰函数和对象的类似，详细请参考修饰函数的deprecated属性。另外，从GCC 4.9以及Clang 3.6起，deprecated可单独用于修饰一个枚举常量。

17.15　本章小结

本章主要介绍了主要的一些GNU扩展语法特性，这些语法特性可以在GCC 4.9或更高版本以及Clang 3.8及更高版本上进行使用。由于目前大部分主流桌面编译器以及嵌入式系统编译器都基于GCC与Clang编译器的核心，所以其适用范围相当广泛。同时，GNU语法扩展也是针对C语言相当好的语法体系的补充，增加了其体系的完备性。可以说，有了GNU语法扩展，C语言才能真正算是一门现代化的高级编程语言。

第18章　Clang编译器对C11标准的扩展

第17章给大家描述了GNU语法扩展，这些扩展可以给GCC和Clang编译器以及基于这些编译器核心打造出来的编译工具链进行使用。而本章将为大家介绍当前最为先进的LLVM Clang编译器针对C11语法扩展特性。

单独针对Clang编译器做介绍也是有很多缘由的。首先，Clang编译器是整个LLVM项目的一个子项目，它是C、C++以及Objective-C编程语言的编译器前端。而整个LLVM项目采用的是UIUC许可证、基于MIT/X11许可证以及3条款的BSD许可证。这个许可证比起GCC的著名GPL许可证要宽松很多。这意味着我们直接获取Clang的源代码，然后可以根据自己当前的目标平台做适配，而经过修改的代码部分则无需开放出来，这一点在GPL许可证上是不允许的。正因为如此，它广受各大厂商的欢迎。而且LLVM项目最初由Swift编程语言创始人Chris Lattner在大学里发起，然后被Apple看中，Apple在LLVM上做了大力投资，后来又开启了自己的一个分支，称为Apple LLVM。因此我们现在从Xcode 4开始起用的LLVM编译器都称为Apple LLVM编译器，而Apple LLVM编译器又是从标准的LLVM主干上拉下来的。

此外，Google也是从NDK 9开始起大力推广LLVM Clang编译工具链。而且在NDK 11中就有官方声明，GCC编译器只升级到4.9，后续将处于维护状态，然后NDK 13版本将直接被丢弃，而只使用LLVM Clang编译工具

链。再看看ARM，ARM官方的编译工具链ARM Studio 6也开始基于Clang。而像AMD则是把Clang直接用于自己的OpenCL编译器上。

到了2017年，微软也将Clang集成在Visual Studio开发环境中，作为可选的C语言编译器前端，而后端仍然采用MSVC的目标代码生成器以及运行时。

可见，Clang作为C语言的编译器前端已经被炒得如此狂热了。对于我们开发者来说，如果当前在做iOS以及Android应用开发，那么请别犹豫，直接使用Clang编译器（当然，这也没得选），使用它的语法扩展吧。而且在macOS、iOS、watchOS以及tvOS的开发框架中，已经有不少API都直接使用了Clang语法扩展，比如后面会讲到的Blocks语法。这些在Android开发上也都能使用，后面会进行介绍。

如果我们使用了Clang编译器，那么编译器就会自动定义预编译宏__clang__，表示当前用的是Clang或基于Clang的编译工具链。此外，__GNUC__这个预编译宏也会被定义，说明当前编译器遵循GNU语法扩展。

18.1 特征检查宏

特征检查宏是一组用于检查当前Clang编译器是否具有某些语法特性或是否支持指定的内建函数的宏，这些宏都以两条下划线打头。比如，像__has_builtin用于检查当前Clang编译器是否支持指定的内建函数。__has_feature与__has_extension这两个宏用得较多，用于检查当前编译器是否支持所指定的语法特征。__has_attribute则用于检查当前编译器是否支持所指定的属性。另外还有__has_include宏则用于检查当前上下文中是否包含了所指定的文件。关于特征检查这个语法特性，其内容比较繁杂，但同时语法却比较简单，各位可以参考这个网页上的内容：http://clang.llvm.org/docs/LanguageExtensions.html。

18.2 _Nullable与_Nonnull

Clang编译器当然也支持GUN语法扩展中的nonnull属性，但是对于每个需要指定函数形参不能为空的参数都要用__attribute__（（nonnull））去修饰显然太过繁琐，而且使得代码也显得比较冗长。而在Clang编译器中，则直接引入了_Nullable（前面带有一条下划线）限定符用于修饰指针类型的函数形参对象可以为空；用_Nonnull（前面带有一条下划线）限定符用来表示当前所修饰的指针类型的形参对象不可为空。此外，这两个关键字还能用于修饰函数的返回类型，如果函数返回类型为指针类型的话。引入这两个限定符一来是提高代码的简洁性，尽管我们也可以自己定义宏用来简化__attribute__（（nonnull））；二来是为了能更好地将我们用C语言定义的外部全局函数输出给其他编程语言进行使用，比如Swift。像Swift这种编程语言有Optional语法特性，需要指明当前函数形参是否可以为空，如果不能为空，我们必须使用_Nonnull限定符来修饰该形参指针对象。

另外各位要注意的是，Apple在Apple LLVM 6.0中就引入了__nonnull与__nullable，这两者都还能使用，但推荐使用Clang标准化的_Nonnull与_Nullable，它俩在其他基于Clang编译器前端的编译工具链上也能使用。代码清单18-1展示了_Nullable/_Nonnull属性的使用。

代码清单18-1　_Nullable/_Nonnull属性的使用

```
#include <stdio.h>
static void MyFunc(int* _Nonnull p, int* _Nullable q)
```

```
{
    if(p == NULL)
        return;

    if(q != NULL)
        *p = *q;
    else
        *p = 0;
}

int main(int argc, const char * argv[])
{
    int a = 10, b = 20;

    MyFunc(&a, &b);
    printf("a = %d, b = %d\n", a, b);

    // 如果第一个参数传空，那么编译器会发出警告
    MyFunc(NULL, NULL);

    // 由于第二个参数可以为空，所以这个调用不会有任何警告
    MyFunc(&a, NULL);
    printf("a = %d\n", a);
}
```

代码清单18-1中，函数MyFunc具有两个指向int类型的指针参数p和q。其中形参p用_Nonnull修饰，生命表示它不能为空；形参q用_Nonnull修饰，声明它可以为空。Clang编译器会在编译时做静态代码分析，如果在调用MyFunc时，将空（NULL）传递给形参p，那么编译器就会发出警告。

18.3　函数重载

函数重载是一个高级编程语言常用特性，现在很多高级编程语言都支持该语法特性，包括C++、Java、C#、Swift，等等。那么什么是函数重载呢？简单来说，就是在同一单元翻译中定义了一组相同名称的函数，这些函数具有不同的参数类型或参数个数，那么我们称这组函数为重载函数（overloaded functions）。倘若这组函数中有任意两个函数的参数类型与个数都完全相同，那么编译器仍然会报有类型冲突的错误。

Clang中通过使用__attribute__（（overloadable））这一函数属性说明符将一个函数指示为可重载的。各位要注意的是，对于一组重载函数，我们需要将它们每一个都用__attribute__（（overloadable））函数说明符进行修饰，如果漏了一个，那么那个函数将会报出类型冲突的错误。下面通过代码清单18-2来给大家展示一下函数重载的表现方式。

代码清单18-2　Clang编译器中使用函数重载特性

```
#include <stdio.h>

static void __attribute__((overloadable)) Func(void)
{
    puts("This is a function!");
}

static void __attribute__((overloadable)) Func(int a)
{
    printf("a = %d\n", a);
}

static void __attribute__((overloadable)) Func(float f)
{
    printf("f = %f\n", f);
}

static void __attribute__((overloadable)) Func(char c)
```

```
{
    printf("The character is: %c\n", c);
}

static void __attribute__((overloadable)) Func(unsigned a, short b)
{
    printf("The sum is: 0x%.8X\n", a + b);
}

/**
 * 这个函数会报类型冲突的错误，因为它与第一个Func一样，形参列表为空，
 * 即便它的返回类型与第一个Func有所不同
*/

static int __attribute__((overloadable)) Func(void)
{
    return 10;
}

int main(int argc, const char * argv[])
{
    // 这里调用的是void Func(void)函数
    Func();

    // 这里调用的是void Func(int a)函数
    Func(100);

    // 这里调用的是void Func(float f)函数
    Func(10.25f);

    // 各位注意，这里调用的也是void Func(int a)函数，
    // 因为之前我们已经讲到了，C语言中像'c'这种字符字面量默认为int类型
    Func('c');

    // 当对'c'做了char类型的投射操作之后，也就将int类型转换为了char类型，
    // 此时调用的就是void Func(char c)函数
    Func((char)'c');

    // 调用了void Func(int a, short b)函数
    Func(0xc0de0000, 0x1314);
}
```

通过代码清单18-2我们发现，使用函数重载将会使得函数接口变得十分简洁。对于实现相同功能的函数，我们无需通过改变函数名就能给不同类型的参数或者不同个数的参数做相应的函数调用了。我们在使用具有函数重载特性的一组函数时需要注意，像代码清单18-2中所示的，如果我们要传的一些实参的类型与想要调用的那个函数的形参类型不匹配，那么我们此时需要使用类型投射操作，将实参类型显式地转换为相应函数的形参类型，否则可能会调用到我们本不想调用的那个函数。

18.4 Blocks语法

Blocks语法是Apple在Apple LLVM 2.0中贡献给LLVM开源社区的C语言扩展语法特性。这里的Blocks不是指我们C语言中的语句块，而是一种Lambda表达式，由于它可以像语句块那样定义在函数体内，所以将此语法命名为Blocks。为了避免混淆，后续提及到Blocks语法时一律用Blocks英文给出，而对于普通语句块，则直接称其为“语句块”。

在开讲Blocks语法之前，我们首先简单介绍一下两个基本概念，一个是Lambda表达式（Lambda是希腊字母λ），还有一个是闭包（Closure）。

在计算机编程领域，Lambda表达式也称作为匿名函数，它可以像函数那么定义，也可以做函数调用，但不需要给出函数名。在计算机编程语言领域中，闭包也称为词法闭包（lexical closure）或函数闭包（function closure），在具有头等函数的编程语言（主要是函数式编程语言）中用于实现词法作用域的名字绑定（lexically scoped name binding）。在操作上，闭包是一条记录，它将一个函数与它自己的上下文存放在一起。当我们在某一个函数里创建闭包的时候，此闭包将该函数中每一个与自己相关联的局部对象映射为自己所绑定的名字。

至此我们可以看到，一个Lambda表达式如果可作为闭包，那么必须满足两个条件：第一，它能绑定自己所在函数的局部对象；第二，它必须有自己独立的执行环境，使得所绑定的局部对象能始终在其自己的执行环境

中维护，这一点也是判定一个Lambda表达式是否可作为闭包的最为关键的一点。根据这两个条件，我们可以得出，像Java 8中的匿名Lambda表达式严格意义上不属于闭包，因为它对自己所绑定的局部对象的维护特性非常薄弱，它只能绑定常量，而不能绑定变量，并且还有其他许多约束。而C++11的Lambda表达式自身也不具备保存自己执行环境的接口，只能通过借助std：：function做Lambda表达式执行环境的拷贝与封装，这样才能把Lambda表达式拷贝到函数外执行。我们后面可以看到，Clang中的Blocks语法就是相当标准的闭包。

下面我们将引入Blocks引用类型以及Blocks定义的语法形式，同时我们也将结合上面涉及的一些术语做更明了的介绍。Blocks的引用类型与函数指针类型十分类似，我们只需要把函数指针类型中的*符号改为^符号即可。比如，我们声明一个指向函数的指针对象，像void（*pFunc）（int）；表示声明一个函数指针对象pFunc，它所指向的函数类型为void（int）。而如果是一个Block引用对象的声明也类似：
void（^refBlock）（int）；表示声明了一个对Block的引用对象，它所引用的Block类型为void（^）（int）。从上面这两句话中我们就可以看到一个Block引用类型与指向函数指针类型的形式十分相似。而定义一个类型为void（^）（int）的Block可以如下：^void（int a）{printf（“a=%d\n”，a）；}；是非常简单。这个Block的返回类型为void，并且含有一个int类型的参数。从对Block的定义上我们也可以看到，Block自身是没有标识符的，它就是一个纯粹的表达式。

我们将通过代码清单18-3来介绍上面提到的一些术语。不过在此之前，如果各位是在Debian/Ubuntu系统上编程的话，那么可以通过在命令行输入以下命令来安装Clang以及使用Blocks和Grand Central Dispatch所需要的库。

```
sudo apt-get install llvm
sudo apt-get install clang
sudo apt-get install libblocksruntime-dev
sudo apt-get install libdispatch-dev
```

第三条命令可以先不用输入，因为Clang本身所带的一些库可能已经包含了Blocks的运行时。如果我们在使用Blocks时发生了连接错误，那么可以安装blocksruntime-dev。另外，我们在Windows以及除了macOS/iOS以外的类Unix系统环境下通过Clang来编译使用Blocks的C代码的时候，必须使用-fblocks编译命令选项，指示编译器开启Blocks语法，然后使用-lBlocksRuntime命令选项来连接Blocks所需要的运行时库。如果我们还想使用Grand Central Dispatch，则再添加-ldispatch命令选项。而在macOS中，Xcode已经帮我们做了一切，我们只需要直接点击三角箭头按钮编译构建即可。如果各位用的是Windows和Android，那么可以从这个地址下载到blocksruntime-dev的源代码，然后可以自己做成库或是直接将源代码放入自己的项目工程中：https://github.com/mackyle/blocksruntime。这个项目基于UIUC与MIT双重许可证，各位可以放心大胆地使用。这里只需要BlocksRuntime目录下的Block.h、Block_private.h、data.c以及runtime.c这4个文件，以及根目录下的config.h这个文件即可。此外，对于现在Visual Studio 2017 Community中的VS-Clang，如果使用__block对象进行捕获，那

么代码生成会有问题，一旦使用__block对象捕获就会引发异常。所以在Windows系统下，仍然建议各位使用纯Clang编译器做后续对Blocks的实践。

代码清单18-3　Blocks语法的初步使用

```
#include <stdio.h>

int main(int argc, const char * argv[])
{
    // 在main函数中声明局部对象a，并将它初始化为10
    int a = 10;

    // 声明一个对void(^)(int)类型的Block的引用对象，并用一个Block对它初始化
    void (^refBlock)(int) = ^void(int i) {
        printf("a + i = %d\n", a + i);
    };

    a = 20;

    // 对Block引用对象做一次调用
    refBlock(1);    // 输出：a + i = 11

    a = 30;

    // 对Block引用对象再做一次调用
    refBlock(1);    // 输出：a + i = 11

    printf("a = %d\n", a);  // 输出：a = 30
}
```

下面我们就来分析代码清单18-3中Block。代码清单18-3中，我们先在main函数中声明了一个局部对象a，然后声明了一个Block引用对象refBlock，并定义了一个Block对该引用对象进行初始化。这里，我们所创建的Block的词法作用域就是main函数中的语句块作用域。然后，Block中的对象a就是对main函数中的局部对象a的名字绑定。Block中的a是在创建此Block时就已经被创建好了，它是将main函数的局部对象a的当前值，也就是在创建该Block之前那一刻的值，拷贝到Block中的局部对象a里的，这一过程就是本节一开始所介绍的名字绑定。

接下去，我们发现无论main函数的局部对象a的值怎么修改，在Block中在它创建时所绑定的a的值不会有任何改变。因此我们先后两次调用了refBlock对象所引用的Block，无论main函数的对象a怎么修改，输出的结果都是一样的。

下面，我们用图18-1来描述Blocks语法与相关计算机编程术语的对应关系。

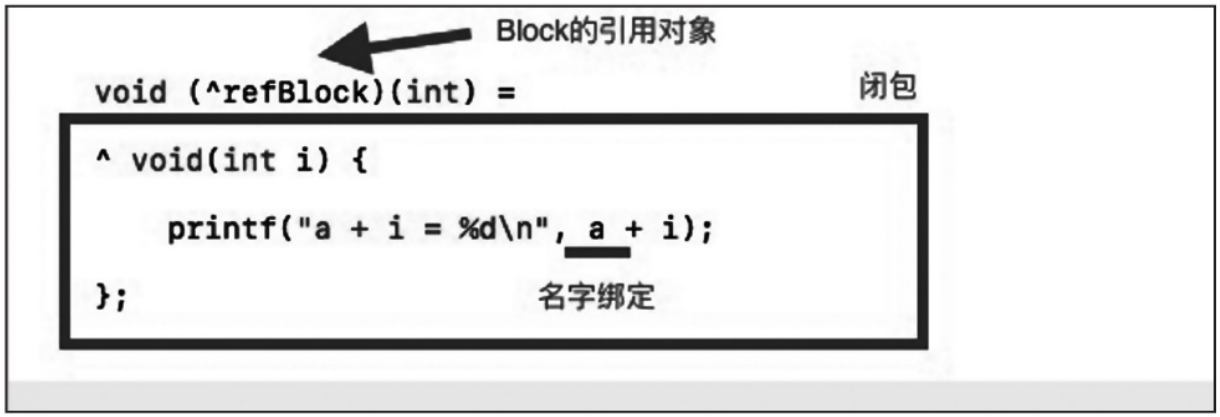


图18-1　Blocks语法与计算机编程术语的对应

在图18-1中我们可以看到，refBlock对象是对Block的一个引用，它就如同扮演函数指针一般的角色，通过refBlock可以直接调用由该对象所引用的Block。加粗的矩形围住的部分就是一个Block，它通过一个前导^符号引入，后面跟着此Block伴随实现的返回类型以及形参列表，这与函数体的定义相同，也是用一对{}来围住对此Block的实现。而用加粗的矩形所围住的定义整个Block的表达式就被称为Lambda表达式，它对于Block而言也是一个闭包。

在Block的实现中引用了该Block所在函数的局部对象a，由于Block有其自己独立的执行环境，所以Block中的对象a其实与函数局部对象a是两个不同的对象，而在这条Block表达式执行完之时，它所绑定的a已经将其外部函数局部对象a的值拷贝进来了。因此，在Block里的a的值是不能被修改的，因为它是只读的。同时，此后无论外部函数局部对象a的值怎么改变，对Block里的a的值是不会有任何影响的。

至此我们可能会有这么个疑问：如果Block中所绑定的外部对象的值不能修改，那么它跟真正的闭包不是还有一定差距么？而且实用性也大打折扣。确实如此，现在Java 8对Lambda表达式的实现也仅仅做到这一步而已，但Blocks语法则没那么简单。Blocks语法中引入了__block（前面带有两条下划线）关键字用来修饰可被Blocks以传引用的方式所绑定的对象。我们查看代码清单18-4的示例。

代码清单18-4　外部函数对象以引用的方式绑定给Block

```
#include <stdio.h>

int main(int argc, const char * argv[])
{
    // 在main函数中声明对象a
    int a = 10;

    // 在main函数中声明对象b，并且对象b将以引用的方式绑定到Block中去
    __block int b = 20;

    // 这里直接定义一个Block而不声明指向它的一个引用对象，然后直接调用
    ^void(void) {
        a++; // 这句对a的修改将会报错

        // 下面这句没有问题，因为外部局部对象b是以引用的方式绑定到Block中的对象b的。
        // 所以这里的b += a操作其实是有三步操作：
        // 1）先通过Block中的b获取它所指向的外部局部对象b的地址；
        // 2）对外部对象b做b += a的操作；
        // 3）将结果值存放到外部对象b中
        b += a;
    }
    // 直接对该Block进行调用
    ();
```

```
    printf("b = %d\n", b);  // 我们将观察到，b的值变为了30

    // 在定义一个Block时，如果其返回类型为void，那么void可省；
    // 如果其参数列表为空，那么参数列表也可省
    void (^refBlock)(void) = ^ {
        printf("Current b = %d\n", b);

        b += a;

        printf("After modified, b = %d\n", b);
    };

    // 我们这里将b的值修改为25
    b -= 5;

    // 调用Block，我们发现在Block中，b一开始的值为25，然后它被修改成了35
    refBlock();

    // 我们再观察当前外部函数局部对象b的值，发现b也变成了35
    printf("b = %d\n", b);
}
```

通过代码清单18-4我们可以发现，用__block所修饰的局部对象b，当把它绑定到一个Block中的时候，其实相当于是把它的地址传入进去，而不是它当前的值。而在Block中访问b的时候其实也是通过Block里所绑定的b的地址去访问其值的，这么一来就可以通过该地址来修改外部局部对象b的值了。因此说用__block所修饰的局部对象b绑定到Block的过程其实是传引用的过程，而没有用__block所修饰的局部对象a绑定到Block的过程其实是传值的过程。图18-2给出了Block绑定外部局部对象的结构示意图。

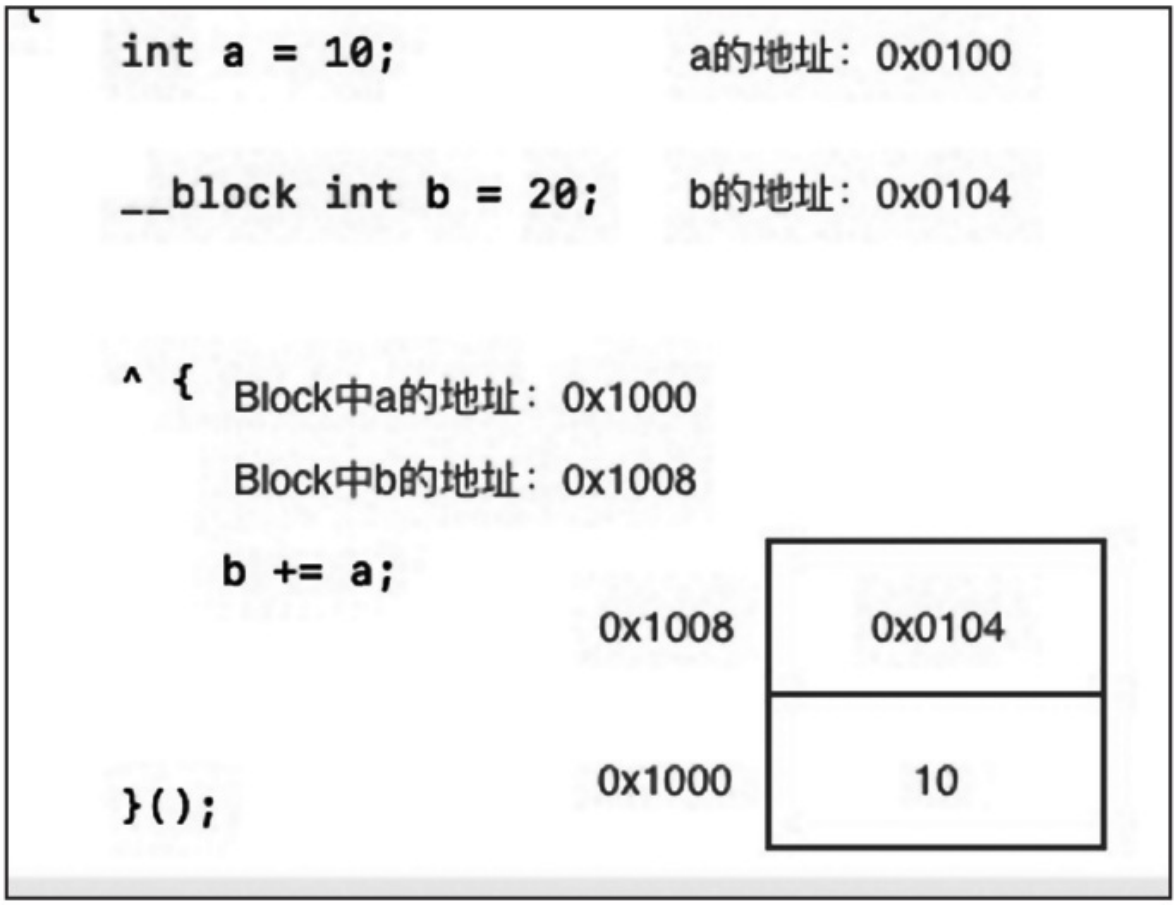


图18-2　Block绑定外部局部对象的结构示意图

从图18-2中我们可以看到，对于一条b+=a；这么简单的语句，在Block中的实际操作过程是比较复杂的。这里的复杂点是对b的处理——先获得外部局部对象b的地址，然后再读取该地址的内容，与Block中的a相加之后再写回到外部局部对象b的地址中去。另外，在图18-2中我们可以看到当Block创建完之后，Block中a的地址假设为0x1000，其存放的数据就是外部对象a的值；而b的地址假定为0x1008，而该地址所存放的数据则是外部对象b的地址，而不是此对象b的值。

我们清楚了Block内部的处理机制之后，下面我们就要讨论如何传递

Block引用了。因为Block是一个代码块，它的实现可以非常灵活，比如可以与它所在的函数共享同一代码空间，也可以把Block中的代码存放到其他代码存储段中。但是其执行上下文（也就是执行环境）必须得到安全维护！我们知道，一旦退出了一个函数体，那么该函数中的所有局部对象都会被自动销毁，而Block的执行上下文其实在存储类型上来说也是auto类型的，它与当前所在的函数共享同一个栈空间，跟该函数中的局部对象一样。这么做有两大好处，首先在我们不想把函数中所定义的Block传到外部的时候，在函数内对Block的实现可以做很大优化，比如在Clang的实现中是把外部函数中用__block所声明的对象与Block中跟它绑定的对象作为同一个实体，也就是说两者共享同一栈存储单元；其次，如果在每次调用函数的时候都要创建一次其内部的Block执行上下文，那么什么时候释放该Block的执行上下文呢？这是个问题。因此Blocks语法中引入了运行时库<Block.h>，这个库中有两个宏函数接口——Block_copy与Block_release。Block_copy接口是先动态分配一块存储空间，然后将所指定的Block的执行上下文拷贝到分配的存储空间中。Block_release则是释放所指定的Block执行上下文。当我们要把在某个函数中定义的Block传递到其他模块执行前，可以通过Block_copy接口将指定的Block上下文动态分配给该Block引用的持有者，然后由该持有者自己管理何时通过Block_release进行释放。代码清单18-5将展示这对接口的用法。

代码清单18-5　Block执行环境的保存与释放

```
#include <stdio.h>
struct MyObject
```

```
{
    int a;
    void (^block)(int);
};

/** 通过动态分配存储空间的方式创建一个MyObject结构体对象并返回将其首地址返回 */
struct MyObject *CreateMyObject(void(^ _Nullable refBlock)(int))
{
    struct MyObject *pObj = malloc(sizeof(*pObj));
    pObj->a = 1;

    // 如果refBlock不空，则拷贝其执行上下文，并将拷贝之后的引用赋值给block成员
    if(refBlock != NULL)
        pObj->block = Block_copy(refBlock);
    else
        pObj->block = NULL;

    return pObj;
}

/** 销毁指定的MyObject结构体对象 */
void DestroyMyObject(struct MyObject* _Nullable pObj)
{
    // 如果pObj所指向的对象不空，则将它释放
    if(pObj != NULL)
    {
        // 先释放pObj的block成员
        if(pObj->block != NULL)
            Block_release(pObj->block);

        free(pObj);
    }
}

static struct MyObject* Test(void)
{
    int a = 10;
    __block int b = 20;

    printf("address of a: %.16tX\n", (uintptr_t)&a);
    printf("address of a: %.16tX\n", (uintptr_t)&b);

    void (^block)(int) = ^(int i){

        printf("In block, address of a: %.16tX\n", (uintptr_t)&a);
        printf("In block, address of a: %.16tX\n", (uintptr_t)&b);

        b += a + i;
        printf("a = %d, b = %d\n", a, b);
    };

    // 这里先调用一次Block
    block(100);

    // 创建CreateMyObject对象，将block引用作为实参传递进去
    struct MyObject *pObj = CreateMyObject(block);

    return pObj;
}

/** Func函数将返回一个类型为int(^)(int)的Block引用对象 */
int (^Func(int a))(int)
{
    __block int c = a + 1;

    // 声明一个Block的引用对象block，并将它指向一个int(^)(int)类型的Block实现
    int (^block)(int) = ^int(int i) {
        c += i;
        return c;
```

```
    };

    // 各位必须注意的是，如果一个函数要把它内部的一个Block对象引用传递出去，
    // 则必须使用Block_copy，否则在外部再调用Block_copy则已经晚了，
    // 运行时可能会发生异常
    return Block_copy(block);
}

int main(int argc, const char * argv[])
{
    struct MyObject *pObj = Test();

    // 在main函数中调用了block引用之后，我们发现此Block中的a和b的值都被完整地保留
    // 此外，Block中a和b的地址都被改变了，
    // 因为它所在的执行上下文从Test函数的栈空间被移入了动态分配的存储空间
    pObj->block(pObj->a);

    // 我们再次调用Test函数，我们发现第二次进入Test后，
    // 其中Block的上下文会按照当前函数调用的执行环境再做安排，
    // 然后再通过动态分配的存储空间，拷贝到pObj2的block成员中
    struct MyObject *pObj2 = Test();

    // 我们可以再调用一次再观察情况，Block中b的值仍然安全地维护着
    pObj->block(pObj2->a);

    // 我们再调用pObj2的block
    pObj2->block(5);

    DestroyMyObject(pObj);
    DestroyMyObject(pObj2);

    // 调用Func函数
    int (^refBlock)(int) = Func(10);

    // 通过Func函数所返回的Block引用调用该Block
    int value = refBlock(3);
    printf("value = %d\n", value);  // 这里输出：value = 14

    // 我们可以再调用一次
    value = refBlock(2);
    printf("value = %d\n", value);  // 这里输出：value = 16

    // 由于这里的refBlock是通过Func函数最后的Block_copy得到的，
    // 所以用完之后，我们要使用Block_release将它释放
    Block_release(refBlock);
}
```

代码清单18-5提供了很多信息。首先，我们可以看到在做Block_copy之前Block里与外部函数绑定的对象的地址，然后在Block_copy操作之后这些对象地址的变化。然后，我们还能看到如何使用Block_copy与Block_release做Block引用的拷贝与释放，并且在Block_copy操作之后，原先函数中Block内部绑定对象的值都完好地保存着。Block执行环境在内部会有引用转移机制，也就是说，当Block所在的函数即将返回时，它会将所

绑定的__block对象解除引用，使得该Block内的名字绑定对象指向自己执行环境中的某个存储空间，并将那时刻的__block对象的值拷贝到那个存储空间中。这样，等函数返回之后，无论怎么调用此Block，访问用__block所修饰的名字绑定对象都访问同一个执行环境中的存储空间，而不会再受函数栈空间的影响。

最后，我们还获悉如果一个函数要返回它里面所定义的一个Block对象引用，那么将该Block引用返回之前必须通过Block_copy操作。如果不在此函数里使用Block_copy操作，而是在外部该函数调用结束之后再使用Block_copy操作，那么执行可能就会引发异常。此外，即便在函数返回之后立即调用Block_copy操作也不行，因为Block的执行上下文在做Block_copy操作之前都是与该函数的栈空间所共享的，一旦函数返回，则栈内会发生一定变化从而可能直接或间接地对Block的执行环境造成破坏。

Block除了可以定义在函数内之外还能定义在文件作用域。当然，如果一个Block实现定义在文件作用域，那么它其实就跟普通函数差不多了，因为它也不需要绑定任何局部对象，它的栈空间本身就是独立的。我们还可以在其他语句块内定义一个Block，但需要注意的是，C语言标准已经明确指出，当一个语句块中的局部对象出了这个语句块作用域，那么它的生命周期也就到头了，即便在有的时候通过指针来间接访问这些对象可能会好使。不过我们还是要注意，在语句块中定义的Block如果要出了该语句块之后对它调用的话还是应当先调用Block_copy。最后提醒大家的是，在Block中不能绑定一个数组对象。如果我们要把一个数组对象传递到一个Block中

可以通过参数传递的方式，或是通过用一个结构体来封装。代码清单18-6展示了以上这些Block使用上的特性。

代码清单18-6　Blocks语法的其他使用细节

```
#include <stdio.h>
#include <Block.h>
#include <string.h>

static int s_array[] = { 1, 2, 3 };

// 这里声明了一个返回类型为int(*)[3]，无参数列表的Block引用对象myBlock，
// 并且直接在文件作用域实现了该Block
static int (*(^myBlock)(void))[3] = ^int(*(void))[3] {
    return &s_array;
};

int main(int argc, const char * argv[])
{
    // 对于类型比较复杂的Block引用，我们可以直接用__auto_type，非常简便
    __auto_type block = myBlock;

    // 我们通过main函数中声明的Block引用对象block来调用此block
    int (*pArr)[3] = block();

    // 声明array数组对象
    int array[] = { pArr[0][0], pArr[0][1], pArr[0][2] };

    // 在Block中无法直接绑定外部函数的局部数组对象，因此这里直接通过结构体来封装
    struct { int arr[3]; } s;

    // 将数组array的数据拷贝到s对象中
    memcpy(&s, array, sizeof(array));

    void (^block2)(void) = NULL;

    if(array[0] > 0)
    {
        block2 = ^{
            int sum = 0;
            for(int i = 0; i < sizeof(s.arr) / sizeof(s.arr[0]);
                i++)
                sum += s.arr[i];

            printf("The sum is: %d\n", sum);

            // 当然，我们可以在Block中直接访问指向数组的指针对象
            sum = 0;
            for(int i = 0; i < 3; i++)
                sum += (*pArr)[i];
            printf("The second sum is: %d\n", sum);
        };

        // 我们在这里通过block2调用if语句块中定义的Block没有问题
        block2();

        // 但是一旦我们需要在if语句块作用域外调用block2，则必须用Block_copy
        block2 = Block_copy(block2);
    }
```

```
    if(block2 != NULL)
    {
        block2();

        // 用完之后释放
        Block_release(block2);
    }
}
```

在了解了Blocks语法之后，下面将给大家呈现如何通过Grand Central Dispatch利用多核多线程来对一个数组做求和计算。先看代码清单18-7。

代码清单18-7　通过Grand Central Dispatch做多核多线程并行求和计算

```
#include <stdio.h>
#include <stdbool.h>
#include <stdatomic.h>
#include <dispatch/dispatch.h>

static int s_buffer[5][10000];

int main(int argc, const char * argv[])
{
    // 对s_buffer进行初始化
    for(int i = 0; i < 5; i++)
    {
        for(int j = 0; j < 10000; j++)
            s_buffer[i][j] = 10000 * i + j;
    }

    // 声明用于做并行计算的行索引
    __block volatile atomic_int rowIndex;

    // 声明用于最后计算的结果
    __block volatile atomic_int result;

    // 通用标识用户线程的计算全部完成
    __block volatile bool isCompleted = false;

    // 先对行索引初始化为0
    atomic_init(&rowIndex, 0);

    // 对计算结果初始化为0
    atomic_init(&result, 0);

    // 定义计算Block
    void (^computeBlock)(void) = ^ {
        // 确定当前要计算的行
        int row;

        while(row = atomic_fetch_add(&rowIndex, 1), row < 5)
        {
            int sum = 0;
            for(int i = 0; i < 10000; i++)
                sum += s_buffer[row][i];

            // 将当前计算结果与result值相加
            atomic_fetch_add(&result, sum);
```

```
        }
    };

    // 线程分派执行计算Block
    dispatch_async(dispatch_get_global_queue(QOS_CLASS_USER_INTERACTIVE,
                                             0),
                   ^{ computeBlock(); isCompleted = true; });

    // 在主线程上执行计算Block
    computeBlock();

    // 如果用户线程没有执行完成，则一直等待
    while(!isCompleted)
        asm("pause");

    printf("The result is: %d\n", atomic_load(&result));

    // 我们下面用单线程传统算法计算，校验结果
    int sum = 0;
    for(int i = 0; i < 5; i++)
    {
        for(int j = 0; j < 10000; j++)
            sum += s_buffer[i][j];
    }

    printf("sum = %d\n", sum);
}
```

如果我们在Linux下编译运行代码清单18-7的话，则需要连接libdispatch库，可以用-ldispatch命令选项。

下面我们来分析一下代码清单18-7。首先，我们在文件作用域声明了一个s_buffer数组对象，它可以看作具有5行10000列int类型元素的二维数组。我们要做的事情就是利用双核双线程对这个数组中所有元素进行求和。这里所采用的方法其实也比较简单，每个线程都做同样的操作，即先获取当前它要操作的某一行元素的索引，然后对该行元素做求和计算，最后把得到的求和结果与总的结果进行相加。

main函数中，一开始先对s_buffer数组所有元素进行初始化。然后声明用于并行计算的当前行索引以及结果，这两个必须是原子对象，因为它们是两个线程所共享的对象，并且都会在这些线程中进行修改。isCompleted

对象仅仅在用户线程中进行写，而在主线程中进行读，因此这里不需要使用原子对象。

在Block中的计算过程也不复杂，先用原子加法对rowIndex对象做加1操作，由于它返回的正好是修改之前的值，所以处理起来很方便，直接用当前得到的行索引值进行判定，如果小于5则做求和计算，否则退出循环。对于当前行数组每个元素的求和不要使用原子操作，因为原子操作非常缓慢，会使得多核多线程性能还远不如单线程的性能，所以我们仅仅把当前行元素获得的求和结果与总和结果做一次原子加法操作即可。

dispatch_async函数就是Grand Central Dispatch中用于异步创建用户线程并分派执行的接口。第一个参数是指定当前用户线程的优先级，其中QOS_CLASS_USER_INTERACTIVE是最高优先级，QOS_CLASS_BACKGROUND是最低优先级。第二个参数总是传0，现在还没有任何作用，留在以后作为扩展使用。第三个参数则是void（^）（void）类型的Block引用。我们在这里又创建了一个新的Block，因为对于用户线程，我们需要它告知主线程当前计算任务是否已经执行完成，我们在这里也能看到，一个Block中也可以嵌套调用其他Block。下面就是在主线程上直接调用computeBlock进行计算，然后等待用户线程计算结束后获取最终结果。asm（“pause”）；是使用x86处理器的PAUSE指令，用来指示处理器当前线程正处于循环等待，可以使用超线程等硬件线程技术切换给其他线程执行。如果各位将上述代码运行在ARMv7或ARMv8架构的处理器上，那么可以使用asm（"yield"）；，效果一样。

18.5 本章小结

本章介绍了Clang编译器在GCC所支持的GNU语法扩展上自己又新增的一些语法特性。本章着重介绍了Blocks语法的使用，大家如果能掌握好这个语法那必有用武之地，因为OpenCL 2.0也已经引入了Blocks语法特性作为其Lambda表达式，可用于管道异步操作。而对于iOS、macOS的开发者来说学会Blocks语法就显得更为重要了，因为Apple Cooca Framework现在很多接口都含有Blocks，很多消息接口都是以Block引用的方式作为回调参数的。

至此，本书的内容就接近尾声了。下面将畅想一下C语言标准的现在与未来，希望能从中给广大C语言编程爱好者对C语言的设计以及理念带来更多启迪。

第19章　对C语言的未来展望

我们到目前为止已经把所有C11编程语言所包含的语法特性都讲述完了。本章我们将主要根据当前C语言标准委员会的WG14工作小组对下一个C语言标准——目前内部定为C2X已经做的工作成果来展望一下未来。

首先，C语言标准委员会已经非常明确了，C语言在未来不会增加面向对象的特性。因为当前基于C语言的带有面向特性的编程语言已经很多了，典型的有大部分兼容C语言语法的C++编程语言；基于C语言编译器本身，然后仅提供一个预编译器的Objective-C。Objective-C与C语言完全兼容，甚至共用同一个C编译器。它先将一些Objective-C中的语法标签翻译为相应的C函数调用，然后通过C语言编译器进行编译，然后与自身的运行时库进行连接生成最终的可执行程序。因此目前编程语言的市场反而是基于面向对象的编程语言比仅面向过程的要多，而C语言独善其身，不受面向对象语法特性的影响：一方面能维持其纯粹性，使得它能适用于更广泛的运行生产环境，因而也能长期成为工业标准化的编程语言；另一方面，这也能保持C语言可预见的内存模型，我们可以很容易地判断一个结构体中成员如何编排、占用多少存储空间等，这在其他面向对象的编程语言中是很难，甚至是无法做到的。而且这也使得C语言能更好地与汇编语言相结合，解决偏向底层硬件系统的问题。

其次，C语言在语法体系上的进化步伐不会迈得太大。C语言标准委员

会也在尽量控制C语言语法体系的膨胀度。如果一门编程语言太过膨胀则会导致编译器项目将难以维护，并且会引发各种Bug，其实这一点在C++编程语言上体现得十分明显。现在主流编译器都能支持到C++14标准，但关于这些C++编译器的Bug报告也在源源不断地产生。C++17标准还会添加不少东西，对于C++编程语言而言，它已经变得过于臃肿。在这一点上，C语言发展的谨慎还是非常难得的，这个时代做减法反而更难能可贵，这也是为什么笔者对C语言如此情有独钟的原因之一。不过C2X标准也会将一些比较好的现有C++特性引入进来，后续小节中会有所介绍，但不会导致语法特性过于庞杂。下一个C语言标准目前内部称为C2X，说明它将在2020～2029年之间的某个时候发布，不过笔者估计在2020～2022年之间发布的可能性高。

最后，笔者先大概描述一下自己希望未来C语言能提供的一些语法特性，然后在以下各节介绍现在WG14工作小组基本已经确定要在C2X中引入的语法特性。笔者希望C2X标准能引入以下这些语法特性。

1）支持半精度浮点数：随着3D图形渲染以及图像处理的多媒体应用需求增多，不少处理器开发商都已经在自己的处理器中引入了16位半精度浮点数的计算处理单元。而在2008年，IEEE754标准也在8月发布了最新浮点数格式表达标准，此后支持半精度浮点数存储格式的处理器的种类就更多了。这十分常见于GPU中，而当前基于Haswell微架构的Intel处理器以及基于支持VFPv4 ARMv7架构以及ARMv8架构的ARM处理器都支持半精度浮点数的存储。当前GCC和Clang编译器已经通过GNU语法扩展引入了半

精度浮点数类型__fp16，尽管现在大部分CPU不能直接对半精度浮点数做加减乘除算术运算，但是能够在单精度浮点类型与半精度浮点类型之间相互转换。再过几年之后，随着处理器支持半精度浮点的处理能力增强，C语言标准也应该增加对半精度浮点数的支持！根据当前C语言标准新增关键字的命名特性，使用_Half作为半精度浮点数的类型标识符比较合适。然后在标准库中引入<half.h>，可以将_Half宏定义为half。

2）将GNU语法扩展中的typeof纳入标准：萃取一个数据对象的类型在很多方面都会显得十分方便，而且C++11中也引入了decltype关键字，其作用与typeof十分类似，而且还能作为函数类型推导使用。因此C语言标准在引入typeof上也不会显得十分困难，毕竟GCC和Clang对此关键字已经用了十多年了。

3）类型自动推导：很多现代化编程语言都有类型自动推导特性，比如2014年新生的Swift编程语言，还有C++11。类型自动推导在一定程度上能省去不少代码以及一些不必要的类型转换，配合typeof使用将威力无穷。如果运用在宏定义中的话，能体现出无与伦比的灵活性和强大性能。而这个语法特性也在GCC 4.9以及Clang 3.8中通过引入__auto_type关键字实现了。

4）引入Lambda表达式：Lambda表达式在并行计算上能发挥很大优势，这一点在Apple开源的Grand Central Dispatch中就已经体现得淋漓尽致了。Apple将Blocks语法引入到Clang编译器中，它工作良好，不仅能起到简化代码的作用，而且还能提升并行计算的性能。因此笔者这里希望C2X

中能引入Blocks语法或者类似的Lambda表达式。

下面我们将挑选几条比较有意思的WG14工作小组基本已经确定要在C2X中引入的语法特性。

19.1　C语言中的属性

当前C语言标准中是没有“属性”这个语法特性的。其中，我们常用的像字节对齐这类属性已经转为相应的关键字_Alignof与_Alignas作为类型限定符使用。而如果要描述一个对象、函数或类型的其他属性只能借助编译器各自的实现定义。比如在MSVC中使用的是__declspec关键字来引出一个属性；在GCC与Clang编译器中则使用__attribute__关键字来引出一个属性。

C++11标准已经通过[[<属性表达式>]]这一语法特性来描述一个函数、对象或类型的属性。而这次C语言标准委员会也想借助这种表示法来描述属性。这样，以后我们要使用属性时就不需要根据不同编译器来使用__declspec或__attribute__了，而直接使用[[]]即可。

代码清单19-1将展示[[]]属性表达式的使用方式以及效果。

代码清单19-1　C2X的属性

```
#include <stdio.h>
// 定义一个总是被编译器内联的函数
static int [[ always_inline ]] foo(int a, int b)
{
    return a + b;
}

int main(int argc, const char * argv[])
{
    // 声明一个整数对象a，将它类型指明为字节模式
    int [[ mode(byte) ]] a = 100;
    // 定义一个以8字节对齐的结构体
    struct MyStruct {
        short a, b;
    } [[ aligned(8) ]];
```

```
}
```

代码清单19-1中可以看到，使用[[]]属性的方式与__attribute__很类似，而且摆放位置也差不多一样，但在表达上则更为简洁。

19.2 fallthrough属性

我们平时在写switch-case语句的时候，一般情况下每条case语句结尾处都会使用一个break或是return来跳出该选择分支。但是在有些时候，我们确实想在处理完当前case分支之后再直通（fallthrough）到下一条case语句紧接着处理。因此原有的C语言就有默许case语句直通的功能。但是程序员不是神仙，如果当我们在编写程序中不是有意要让当前case语句直通，而是漏加了一条break语句，那么程序就可能会发生意想不到的情况。而且在这种情况下，由于编译器也缺乏足够的条件来判定程序员是漏加了break语句还是故意想让case语句直通，所以也不会给出任何提示。因此C++17标准引入了[[fallthrough]]属性显式地提示当前case语句需要直通到下一条case，这样一来编译器能够判断出当前程序员是否漏加了break还是有意让case语句直通。而这个属性也将在C2X标准中被采纳。

[[fallthrough]]这个属性与其他属性有些不同，它不作为对象、函数、类型的限定符的样式进行使用，而是类似于一条语句，就像break语句那样。标准中也提到了，[[fallthrough]]只能用在case语句中，并且只能放在下一条case语句的上面。倘若两条case语句之间没有任何语句，那么可以不用[[fallthrough]]；不然必须用[[fallthrough]]显式指明当前case语句直通到下一条case语句，不然编译器应该报出警告。代码清单19-2将描述的[[fallthrough]]属性的大致用法。

代码清单19-2　[[fallthrough]]属性的大致用法

```
#include <stdio.h>

int main(void)
{
    int n = 1;

    switch (n)
    {
    case 0:
    // 由于case 0与case 1之间没有任何表达式，所以这里不需要使用[[fallthrough]]
    case 1:
        ++n;
        [[fallthrough]];  // 这里通过[[fallthrough]]进行直通

    case 2:
        n *= 2;
        // 由于case 2下面含有一条case标签语句，但没有出现[[fallthrough]]，
        // 而采用的是隐式直通，所以编译器这里应该给出警告
    case 3:
        n--;

        // 这里通过执行break语句而跳出了此switch选择语句块，
        // 所以前面的[[fallthrough]]都只能直通到这里
        break;

    default:
        n = 0;  // 这里的default分支不会被执行，因为通过直通执行到case 3条件

        // 这里使用[[fallthrough]]编译器会发出警告，因为后续没有case标签了。
        [[fallthrough]];
    }
    // 这里输出：n = 3
    printf("n = %d\n", n);
}
```

代码清单19-2中列出了[[fallthrough]]属性的基本使用方式以及一些约束限制。这里，n的值满足case 1，然后做完++n操作之后通过[[fallthrough]]直通到case 2继续做n*=2操作。case 2条件分支的最后使用了当前C语言的默认直通方式，这会引发C2X编译器的警告，不过这也能成功地直通到case 3语句的处理。case 3分支通过break语句跳出当前switch语句块。因此n对象经过了上述除了default之外的所有case语句的处理执行，结果为3。

19.3 数组片段

数组片段这个语法特性的灵感来源于比C语言更古老的Fortran编程语言。C2X标准增加数组片段特性其实不仅仅是为了加入一些语法糖，使得数组操作更为灵活、方便，而主要是为了激发当前现代化桌面处理器以及智能移动处理器的多线程与SIMD（单指令多数据）指令集的威力。

数组片段的语法其实非常简单：

```
<数组对象标识符> [ <起始元素下标索引表达式> : <长度表达式> : <跨度表达式> ]
```

这里，<起始元素下标索引表达式>指明了所要获取数组片段的首个元素的索引位置；<长度表达式>指明了所要获取数组片段的元素个数；<跨度表达式>指明了从当前元素萃取完之后跨多少个元素再萃取一次。这里，<跨度表达式>可省，如果缺省，那么跨度默认为1。我们现在举一个比较简单的例子，假设我们声明了一个数组对象A，那么A[1：3：2]表示萃取一个数组片段，该数组片段的起始元素为A[1]，一共有3个元素，然后每跨两个元素取一次，所以最终该数组片段的元素依次为A[1]，A[3]，A[5]。而像A[0：3]则表示所萃取的数组片段起始元素为A[0]，长度为3，跨度为1，所以萃取后的数组片段元素依次为A[0]，A[1]，A[2]。而如果是A[：]，则表示以数组A的所有元素作为数组片段的元素。

从以上操作可以看到，数组片段类似于取一个数组的子数组，不过各

位需要注意的是，数组片段一般都是与数组片段进行操作，也就是说等号的左表达式与右表达式都应该是一个数组片段。而在C语言的类型体系上，数组片段其实是不具备类型的，因为数组片段操作的本质是将右表达式数组循环<长度表达式>次，然后按照指定模式进行操作，最后将结果存入左表达式的数组中。所以说数组片段的实际实现会根据当前情况使用循环迭代拆分成一个个单独的标量操作，或是利用当前硬件特性直接通过多线程或SIMD的操作来实现。

下面我们将通过3节内容依次描述数组片段的赋值操作、算术计算操作以及函数调用的场合。

19.3.1 数组片段的赋值操作

正因为数组片段引入到C2X标准的主要原因是充分利用硬件特性，发挥多线程与SIMD指令集的性能，所以在对数组片段的赋值操作时，倘若左右表达式都是同一个数组对象，那么两者最后产生的数组片段的元素位置不应该存在“不完全叠交”的情况。因为这会产生元素依赖而破坏多线程以及SIMD的独立操作。代码清单19-3将描述数组片段的赋值操作。

代码清单19-3 数组片段的赋值操作

```
#include <stdio.h>
#include <intrin.h>

typedef __m128i     int4;

int main(void)
{
```

```
// 声明一个int[10]类型的数组对象a，并且对它进行初始化
int a[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

// 声明一个int[10]类型的数组对象b
int b[sizeof(a) / sizeof(a[0])];

// 将数组a的后三个元素的值赋给数组b的前三个元素
b[0:3] = a[7:3];

// 以上操作即类似于：
for (int i = 0; i < 3; i++)
    b[0 + i] = a[7 + i];

// 这条语句是将数组a的全部元素赋值给数组b
b[:] = a[:];

// 上述操作即类似于：
for (int i = 0; i < 10; i++)
    b[i] = a[i];

// 这条语句是将数组a从起始元素开始，跨3个元素，取3次元素赋值给数组b；
// 对数组b的存放则是从第1个元素开始，每跨2个元素存放一次，一共存放3次
b[1:3:2] = a[0:3:3];

// 以上操作即类似于：
for (int i = 0; i < 3; i++)
    b[1 + i * 2] = a[0 + i * 3];

// 以下语句会产生未定义行为，因为左表达式的数组片段长度
// 与右表达式的数组片段长度不一致
b[0:3] = a[5:4];

// 这条语句是合法的，将数组a的后5个元素赋值给前5个元素。
// 这里对数组a的操作没有元素叠交
a[0:5] = a[5:5];

// 这条语句也是合法的，将数组a的所有元素做一次赋值，
// 这种情况属于对同一个数组操作时，元素全部叠交的场合
a[:] = a[0:10];

// 这条语句会产生未定义行为，因为在操作过程中，
// 数组元素从a[2]到a[9]均是叠交元素，属于部分叠交
a[1:8] = a[2:8];

// 部分叠交为何不允许发生在数组片段呢?
// 因为通过多线程或SIMD的操作会同时读取数组a的相关元素，
// 然后一次性以向量的方式存放到左表达式数组片段指定位置。
// 倘若存在叠交情况，那么会出现不可期待的结果。
// 假设我们现在处理器支持SSE2，那么存在一个类型int4，
// 上述操作将可能是这么实现的：
for (int i = 0; i < 2; i++)
{
    // 从a[2 + 4 * i]元素的地址处一次读取4个元素，存放到value中
    int4 value = _mm_loadu_si128((int4*)&a[2 + 4 * i]);

    // 将value存放到a[1 + 4 * i]元素地址处
    _mm_storeu_si128((int4*)&a[1 + 4 * i], value);
}

// 这里就会引发一个问题，第一次迭代将a[2]到a[5]元素存放到a[1]到a[4]之后，
// a[1]到a[4]的原本元素值就被改变了。
// 第二次迭代将a[6]到a[9]元素存放到a[5]到a[8]之后，
// a[5]到a[8]原本元素的值就被改变了。
// 在多线程情况下，这两次操作的顺序是不确定的，倘若第二次迭代操作先执行，
// 那么稍后做的第一组迭代中，a[5]元素的值就不是原本的值，而是变成了a[6]的值了。

/** 下面我们可以再看一下多维数组的数组片段赋值 */
int c[5][5] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 };
int d[5][5];
```

```
    // 这里我们完成了一次向量转置操作，
    // 将数组二维数组c的第一行元素赋值给了数组d的第一列元素
    d[0:5][0] = c[0][0:5];
}
```

代码清单19-3详细介绍了数组片段赋值的方法、效果以及约束。

19.3.2 数组片段的算术计算操作

数组片段的算术操作与普通的标量计算类似，并且这里的乘法操作也是对数组片段的每个元素进行乘法操作，而不是作为向量内积，也不作为矩阵乘法操作。代码清单19-4列出了数组片段的算术计算操作的用法与效果。

代码清单19-4 数组片段的算术计算操作

```
#include <stdio.h>

int main(void)
{
    // 声明一个数组对象a，它含有10个元素
    int a[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

    // 声明一个数组对象b，它也有10个元素
    int b[10] = { 0 };

    // 将数组b的所有元素赋值为1
    b[:] = 1;

    // 将a[2]元素的值分别与b[0]、b[1]、b[2]相加，
    // 然后将结果存放到b[0]、b[1]、b[2]
    b[0:3] += a[2];

    // 将数组b的各个元素与2相乘，然后将结果存放到数组b的各个元素
    b[:] *= 2;

    // 对数组a的每个元素做递增操作
    a[:]++;

    // 声明一个二维数组c
    int c[5][5] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 };

    // 声明一个二维数组d
    int d[5][5];
```

```
    // 这个没问题，将c[0]的所有元素赋值到d[0]中
    d[0][:] = c[0][:];

    // 这条语句操作没有问题，它是定义良好的
    d[0:2][0:3] = c[1:2][2:3] + d[2:2][1:3];

    // 以上操作相当于：
    for (int i = 0; i < 2; i++)
    {
        for (int j = 0; j < 3; j++)
            d[0 + i][0 + j] = c[1 + i][2 + j] + d[2 + i][1 + j];
    }

    // 以下这条语句的行为是未定义的，
    // 由于取数组a的数组片段的维度与取数组d的数组片段的维度不一致。
    // 我们这里需要注意的是，一个标量可以给数组片段进行计算操作并赋值，
    // 但一个数组片段参与计算与赋值时，左右两边的数组片段维度必须一致
    d[0:2][0:2] = a[0:2];
}
```

19.3.3 数组片段用于函数调用的情况

在函数调用时，数组片段用作函数调用实参的场合非常能体现数组片段的本质特性以及实质作用。这里需要注意的是，数组片段用作函数实参时，函数形参的类型一般为数组的元素类型，而不是一个指向数组元素指针类型。代码清单19-5描述了数组片段用于函数调用时的实参传递情况。

代码清单19-5　数组片段用于函数调用时的实参传递情况

```
#include <stdio.h>
// 定义一个函数MyAdd，它的作用是将两个形参值相加，然后将结果返回
static int MyAdd(int a, int b)
{
    return a + b;
}

int main(void)
{
    // 声明一个数组对象a，它含有10个元素
    int a[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

    // 声明一个数组对象b，它也有10个元素
    int b[10] = { 11, 12, 13, 14, 15 };

    // 对函数MyAdd进行逐步调用，然后将结果依次赋值给b[5]到b[9]元素
    b[5:5] = MyAdd(a[5:5], b[0:5]);

    // 以上操作相当于：
```

```
        for (int i = 0; i < 5; i++)
            b[5 + i] = MyAdd(a[5 + i], b[0 + i]);
    }
```

19.4 其他语法特性

除了上述提到一些语法特性和属性之外，WG14工作组也基本确定了还会将当前Clang编译器已经实现了的__has_include加入到C2X中。

此外还新增了[[maybe_unused]]属性，这个属性用于修饰声明的静态对象、静态函数以及语句块作用域内声明的局部对象。如果这些对象声明之后没有被使用，那么编译器往往会给出警告，但用了这个属性之后，即便上述声明的对象或函数在当前上下文中没有被使用，编译器也不会发出警告。

另外还有对register关键字想做一些拓展，比如让它跟const限定符连用，构造一个常量表达式。常量表达式在C++11标准中是用constexpr限定符来表示的。具体如何实现，WG14工作小组仍然在研究探讨中。

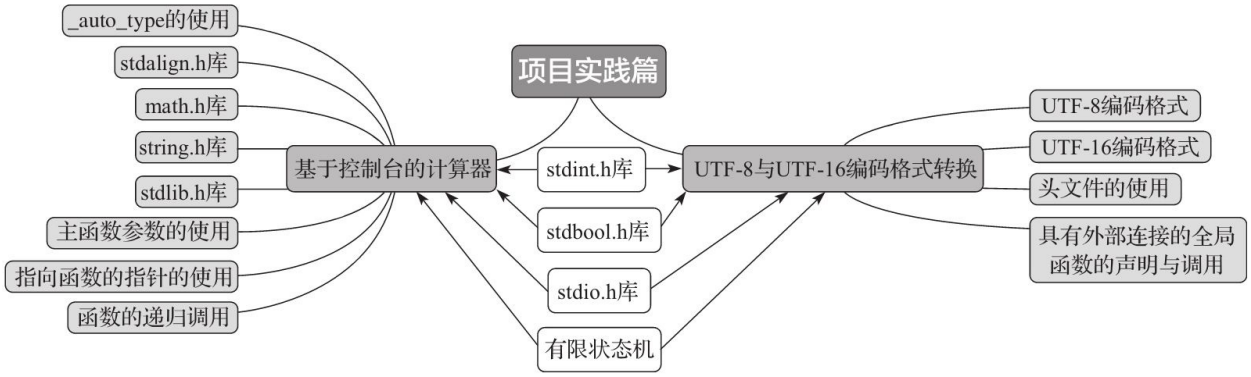
19.5 本章小结

本章对C语言未来的发展做了一些具有前瞻性的猜想，并根据当前WG14的工作情况做了一些汇总。不管C语言在5年后变成怎样，它不会一下子变得面目全非，语法体系上也不会有非常大幅度的修改，而是会充分考虑向前兼容，这一点与C++那种大刀阔斧的演化有所不同。

因此，我们可以相信，未来的C语言不会变得难学，它仍然会是一门相对比较简单、高效的高级编程语言，并且继续作为工业标准为各个生产、设计、计算平台所使用。

C语言所有语法特性的介绍到此结束。下面两章将通过两个不算太大但功能完整的设计项目来展示一下如何用C语言做实际项目。

第五篇　项目实践篇


项目实践篇
基于控制台的计算器
_auto_type的使用
stdalign.h库
math.h库
string.h库
stdlib.h库
主函数参数的使用
指向函数的指针的使用
函数的递归调用
stdint.h库
stdbool.h库
stdio.h库
有限状态机
UTF-8与UTF-16编码格式转换
UTF-8编码格式
UTF-16编码格式
头文件的使用
具有外部连接的全局函数的声明与调用

第20章　制作UTF-8与UTF-16编码字符串的转码器

UTF-8字符编码格式与UTF-16字符编码格式都是当今非常常用的字符编码格式，它们都是根据Unicode这一计算机工业编码标准进行制定的。当今现代化计算机操作系统几乎都使用UTF-8作为其默认的系统字符编码格式，大部分集成开发环境也几乎默认使用UTF-8编码格式。Google在2008年的报告中指出，在互联网上，UTF-8编码已经成为HTML文件使用最多的字符编码格式。

在当今许多文本解析上，大多都以UTF-16编码来收集源文件以及文本上的字符，因为UTF-16编码格式的可变性小，而且大部分情况下2个字节即可表示一个常用字符，因此用UTF-16编码格式的字符串一来不怎么耗费内存，二来能方便地确定字符串的长度，从而可以方便地进行插入、修改、删除等操作。因此对于字符串的编辑而言，它比UTF-8更具优势。

下面，我们将先分别介绍UTF-8编码格式与UTF-16编码格式的相关知识，然后再具体介绍笔者已经实现的两者之间的转换工具以及其他操作工具。

20.1 UTF-8字符编码格式

UTF-8字符编码是一种变长的字符编码格式，可兼容之前已有的ASCII编码格式。它在1993年1月首次官方发布，当时UTF-8编码的字符长度最多可长达6个字节，也就是说当时一个UTF-8编码的字符可占用1到6个字节，比如如果一个字符是与ASCII码相兼容的，那么它就只占1个字节。到了2003年11月，UTF-8做了一些改动然后再次发布，这是为了让UTF-8与UTF-16能完整兼容转换，对UTF-8的表达范围做了裁剪，将它的最大长度缩减到了4个字节，并且要求它必须满足RFC 3629标准中的约束。现在我们用的UTF-8编码格式都是基于2003年发布的标准实现的。

我们上面探讨了UTF-8是一种变长编码格式，其长度在1个字节到4个字节之间，那么一个UTF-8编码的字符是如何表达的呢？一个UTF-8编码的字符由两部分构成，第一部分是用于标识该字符一共需要多少字节的前缀比特标志。这个前缀比特要看第一个字节的高5位，如果是0，表示当前字符由1个字节构成；如果是3，表示当前字符由2个字节构成；如果是7表示当前字符由3个字节构成；如果是15，则表示当前字符由4个字节构成。其实这也就意味着通过观察第一个字节的前导1的个数即可判断出当前字符由多少个字节构成。第二部分则是该字符的码点（code point）。所谓码点就是用于表示一个特定字符具有实际意义的编码值，该编码值是由Unicode组织制定的。比如对于一个兼容ASCII码的UTF-8编码字符A来说，其完整的二进制编码为01000001。可见其前导1的个数为0，说明它就由1个字节构

成。然后它的码点即为1000001，即十六进制的0x41。表20-1列出了UTF-8不同字节数构成的字符编码信息。

表20-1　UTF-8编码信息表

字节个数	码点比特个数	起始码点	最后码点	字节 1	字节 2	字节 3	字节 4
1	7	0x0000	0x007F	0xxxxxxx	N/A	N/A	N/A
2	11	0x0080	0x07FF	110xxxxx	10xxxxxx	N/A	N/A
3	16	0x0800	0xFFFF	1110xxxx	10xxxxxx	10xxxxxx	N/A
4	21	0x10000	0x10FFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

表20-1中，x符号表示码点的一个比特。我们看到，为了与ASCII码完全兼容，Unicdoe在制定UTF-8编码时，将没有任何前导1的字节表示为单字节UTF-8编码，然后从由2个字节构成的UTF-8编码开始，有多少个前导1就表明当前字符占用多少个字节。然后，对于一个字符的UTF-8编码，后续字节的编码都以10开头，这么做的好处是可用于校验当前字符编码的正确性，也可以避免解析到用于表示UTF-8字符串的结束符\0字符，因为它的编码值为0，而有了前导10比特，则当前字节的最小值为0x80，所以不可能会出现0的情况。这种特性也称为自同步（self-synchronizing）特性，它可在遍历UTF-8编码字符串的时候方便校验当前字符编码的正确性。

我们下面举一个具体例子来说明一个字符的UTF-8编码是如何构成的，我们这里用欧元符号€进行举例说明。

在Unicode标准中，欧元符号€的码点为0x20AC。那么可以根据以下步骤来构造出其UTF-8编码。

1）根据表20-1中列出的模式，由于0x20AC这个码点坐落于0x0800到0xFFFF之间，因此它最终的UTF-8编码应该由3个字节构成。因此我们后面看表20-1的第3行。

2）我们将0x20AC用二进制数来表示，为0010000010101100。随后我们将这些比特插入到相应字节中。

3）字节1具有4位固定前导比特1110，而低4位用于存放码点的比特，因此字节1正好可以将码点的高4位放进去，那么得到11100010。

4）字节2具有2个固定的前导比特10，可存放6位码点比特，因此把后续6位码点的二进制比特插入进去得到10000010。

5）字节3具有两个固定的前导比特10，可存放6位码点比特，因此我们可以将剩余的6位码点二进制比特插入进去得到10101100。

这样，我们整理得到欧元符号€的UTF-8编码的二进制表示为：111000101000001010101100。用十六进制表达则是0xE282AC。

20.2　UTF-16字符编码格式

在20世纪80年代，人们开发出了双字节字符编码格式，那时就将它称为“Unicode”。随后，随着各种语言符号的加入，人们很快发现单单用双字节来表示一个字符远远不够，而此时已经有许多开发商基于这种双字节字符编码做了许多大型项目。比如Java一开始就是基于这种双字节编码的字符格式的，所以它能够支持使用汉字或其他文字来定义某个标识符，而不仅仅用ASCII码字符。到了1996年，Unicode标准开发了2.0版本，将原先的双字节字符编码改造为变长的字符编码格式，称为UTF-16字符编码，而之前的“Unicode”则改称为“UCS-2”编码格式，其中UCS表示通用字符集。

因此，UTF-16也是变长的Unicode字符编码格式，不过它与UTF-8不同的是，它只有两种长度，一种是占用2字节的编码格式，这种编码格式与UCS-2完全兼容；还有一种就是4字节编码格式。UTF-16也有“码点”这个概念，并且一个字符的码点与UTF-8编码中的码点值都是一样的，因为这些都是由Unicode组织来制定的。

UTF-16编码根据字符对应的码点值，由三种不同区间范围而做出了不同的定义。

（1）从0x0000到0x7FFF以及从0xE000到0xFFFF两个区间范围

坐落在这两个区间范围内的UTF-16编码表示起来非常简单，就是当前

字符所对应的码点值本身。这也是UTF-16与UCS-2相兼容的区间。Unicode将坐落于这两个区间范围内的码点称为基本多语言平面（Basic Multilingual Plane），简称BMP。比如，像美元$符号，它在Unicode中的码点与ASCII码中的一样，均为0x24，所以它的UTF-16编码即为0x0024。注意，尽管它用一个字节即可表示，但对于UTF-16来说，仍然需要占用2个字节。而欧元符号在Unicode中的码点为0x20AC，所以它对应的Unicode编码值即为0x20AC。

（2）从0x010000到0x10FFFF范围

我们看到，这个区间内的码点用两个字节已经无法表达，所以对于UTF-16编码而言就需要动用4个字节进行描述。这个范围区间又称为补充平面（Supplementary Plane），像Emoji、一些历史脚本、非常少用的汉字等都坐落于此平面中。那么在此范围内的UTF-16编码应该如何计算呢？可以通过以下三步来获得：

1）将码点值减去0x010000，然后取差的低20位，使得结果留在0x000000到0x0FFFFF。

2）取步骤1所得结果的高10位，范围在0x0000到0x03FF，将它加上0xD800，得到第一个16位编码单元，这也称为高位替换，该值的范围在0xD800到0xDBFF。

3）对于由步骤1所得结果的低10位（范围也在0x0000到0x03FF），将它加上0xDC00，得到第二个16位编码单元，这也称为低位替换，该值的范

围在0xDC00到0xDFFF。

我们这里举两个例子来说明码点落在0x010000到0x10FFFF范围字符的UTF-16编码如何表示。首先我们看一个德撒律字母𐐷，该字母的码点定义为0x10437。第一步，我们先将该码点值减去0x10000，得到0x0437。取该值的低20位，得到二进制值00000000010000110111。然后，取它高10位——0000000001，对应十六进制值为0x0001，将它加上0xD800之后得到0xD801，因此它的第一个16位编码单元的值就是0xD801。最后，取20位值的低10位，得到0000110111，对应十六进制值为0x0037，将它加上0xDC00得到0xDC37，因此它的第二个16位编码单元的值就是0xDC37。最后我们得到整个德撒律字母𐐷的UTF-16编码表示为：0xD801 DC37。

第二个例子是古代汉字“”（同“碎”），它的码点定义为0x24B62。第一步，我们先将该码点值减去0x10000，得到0x14B62。第二步取它的低20位，得到二进制值00010100101101100010。第三步，取前10位，得到0001010010，对应十六进制值为0x0052，将它加上0xD800之后得到0xD852，因此第一个16位编码单元的值就是0xD852。第四步，取刚才所得20位值的低10位，得到1101100010，对应十六进制值为0x0362，将它加上0xDC00得到0xDF62，因此第二个16位编码单元的值就是0xDF62。最终，该汉字的UTF-16编码表示为0xD852 DF62。

（3）范围从0xD800到0xDFFF的区间

Unicode永久保留了这两个区间不允许定义任何有意义的字符码点。因

为我们通过上面从0x010000到0x10FFFF范围的UTF-16编码表示法就已经知道这个区间用于4字节UTF-16编码的高位替换与低位替换区间，所以不能用来定义码点值，否则会在编码上产生歧义。

20.3　代码示例

我们前面已经将UTF-8编码规则以及UTF-16编码规则完整地描述过了。下面我们就要开始实现对这两种编码方式的相互转换并制作其他一些常用操作工具函数。

首先要声明的是，这份代码示例可以从笔者的GitHub上获得完整的代码资源：https://github.com/zenny-chen/UTF-8-and-UTF-16-string-utilities。

该项目由三个文件构成，zennyChar.h包含了编码转换工具的对外函数接口；zennyChar.c用于对编码转换工具的相关函数进行实现；main.m是一个Objective-C源文件，因为Objective-C中的字符串对象自身支持UTF-16编码，所以我们可以用来校验结果是否正确。如果我们在macOS下测试这些代码，那么可以创建一个Foundation控制台工程即可编译运行；倘若在Ubuntu系统下，那么可以根据笔者的这篇博文来安装Objective-C的运行时环境：http://www.cnblogs.com/zenny-chen/p/4080067.html。

下面，我们就先来看zennyChar.h头文件的内容，见代码清单20-1。

代码清单20-1　zennyChar.h头文件内容

```
#ifndef UTF_trans_zennyChar_h
#define UTF_trans_zennyChar_h

#include <stdbool.h>
#include <stdint.h>
#include <stddef.h>

/**
* 将UTF16字符串转为UTF8字符串
```

```
* @param pUTF16 指向目的存放UTF16字符串的缓存地址
* @param pUTF8 指向源UTF8字符串缓存首地址
* @param pUTF16Length 指向存放目的UTF16字符串长度的变量首地址；如果转换成功，
  且此指针不空，那么将最终UTF16字符串的长度存放进去
* @return 若转换成功返回true，否则返回false
*/
extern bool ZennyUTF8ToUTF16(uint16_t pUTF16[], const char *pUTF8,
size_t *pUTF16Length);

/**
* 从UTF8字符串获得相应UTF16字符串的长度
* @param utf8Str 指向UTF8字符串首地址
* @return 相应UTF16字符串长度
*/
extern size_t ZennyGetUTF16LengthFromUTF8(const char *utf8Str);

/**
* 将UTF16字符串转为UTF8字符串
* @param pUTF8 指向目的存放UTF8字符串的缓存首地址
* @param pUTF16 指向源存放UTF16字符串的缓存首地址
* @param pUTF8Length 指向存放目的UTF8字符串长度的变量指针；当转换成功时，
  若pUTF8Length不空，则将转换后的UTF8字符串的长度存放进去
* @return 若转换成功返回true，否则返回false
*/
extern bool ZennyUTF16ToUTF8(char pUTF8[], const uint16_t *pUTF16,
size_t *pUTF8Length);

/**
* 从UTF16字符串获得相应UTF8字符串的长度
* @param utf16Str 指向源UTF16字符串的首地址
* @return 相应UTF8字符串的长度
*/
extern size_t ZennyGetUTF8LengthFromUTF16(const uint16_t *utf16Str);

/**
* 获得指定UTF16字符串的长度
* @param s UTF16字符串首地址
* @return UTF16字符串长度
*/
extern size_t ZennyUTF16StrLen(const uint16_t *s);

/**
* 将dst所存放的字符串内容与src所存放的字符串内容拼接（dst内容为头，src内容为尾），
  然后将结果存放入dst的缓存中
* @warning dst必须有足够的存储空间来存放结果字符串内容
* @param dst 指向目的UTF16字符串以及作为源UTF16字符串的头部
* @param src 指向源UTF16字符串尾部内容
* @return 如果拼接成功，指向dst；否则指向空
*/
extern const uint16_t* ZennyUTF16StrCat(uint16_t dst[], const uint16_t src[]);

#endif
```

代码清单20-1已经将本项目所提供的字符串操作工具全都列出来了。这其中不仅包含了UTF-8编码字符串转UTF-16编码字符串，而且还包含了获取UTF-8字符串的长度以及UTF-16字符串的长度，还有UTF-16编码字符串的拼接。

代码清单20-2将列出这些对外接口函数的实现。

代码清单20-2　UTF-8与UTF-16编码工具外部函数接口的实现

```
#include "zennyChar.h"

bool ZennyUTF8ToUTF16(uint16_t pUTF16[], const char *pUTF8, size_t *pUTF16Length)
{
    if(pUTF16 == NULL || pUTF8 == NULL)
        return false;

    size_t orgIndex = 0, dstIndex = 0;
    char ch;

    while((ch = pUTF8[orgIndex]) != '\0')
    {
        uint32_t result = 0;
        int length = 0;

        // counting leading '1' for number of UTF-8 bytes
        uint32_t firstByteFlag = (uint32_t)ch & 0xfc;
        while((firstByteFlag & 0x80) != 0)
        {
            firstByteFlag <<= 1;
            length++;
        }

        // 若长度为0，则直接为兼容的ASCII码
        if(length == 0)
        {
            pUTF16[dstIndex++] = ch;
            orgIndex++;
            continue;
        }

        // 对于macOS系统，采用的是大端的Unicode

        // 先拼接第一个字节的剩余有效比特
        result = (uint32_t)ch & (0xffU >> (length + 1));

        // 对于UTF-8字节数小于4的，说明是基本多语言平面（BMP），对应于Unicode一定为
        // 两个字节。对于大于3字节数的UTF-8则被扩充到21位的Unicode
        int base;
        switch(length)
        {
            case 2:
                base = 11;
                break;

            case 3:
                base = 16;
                break;

            case 4:
            default:
                base = 21;
                break;
        }

        int shiftedBitPosition = base - (8 - (length + 1));

        result <<= shiftedBitPosition;
```

```
        int i = 1;

        // 再拼接其余字节的比特位
        do
        {
            i++;

            shiftedBitPosition -= 6;

            result |= ((uint32_t)pUTF8[++orgIndex] & 0x3f) << shiftedBitPosition;
        }
        while(i < length);

        // 对于UTF-8字节数小于4的，说明是基本多语言平面（BMP），
        // 对应于Unicode一定为两个字节
        if(length < 4)
            pUTF16[dstIndex++] = result;
        else
        {
            // 对于大于3字节数的UTF-8，则采用高低交替对码点格式
            result -= 0x00010000;
            uint16_t high = result >> 10;
            uint16_t low = result - (high << 10);
            pUTF16[dstIndex++] = high + 0xd800;
            pUTF16[dstIndex++] = low + 0xdc00;
        }

        orgIndex++;
    }

    pUTF16[dstIndex] = u'\0';

    if(pUTF16Length != NULL)
        *pUTF16Length = dstIndex;

    return true;
}

size_t ZennyGetUTF16LengthFromUTF8(const char *utf8Str)
{
    if(utf8Str == NULL)
        return 0;

    size_t orgIndex = 0, dstLength = 0;
    char ch;

    while((ch = utf8Str[orgIndex]) != '\0')
    {
        int length = 0;

        // 对UTF-8字节序列计算前导1的个数，有多少个前导1说明该字符由多少个字节构成
        uint32_t firstByteFlag = (uint32_t)ch & 0xfc;
        while((firstByteFlag & 0x80) != 0)
        {
            firstByteFlag <<= 1;
            length++;
        }

        if(length == 0)
            length = 1;

        orgIndex += length;

        int addition = length > 3? 2 : 1;
        dstLength += addition;
    }

    return dstLength;
```

```
}

bool ZennyUTF16ToUTF8(char pUTF8[], const uint16_t *pUTF16, size_t *pUTF8Length)
{
    if(pUTF8 == NULL || pUTF16 == NULL)
        return false;

    size_t orgIndex = 0, dstIndex = 0;
    uint16_t ch;

    while((ch = pUTF16[orgIndex]) != u'\0')
    {
        // 处理ASCII码兼容情况
        if(ch < 0x80)
        {
            pUTF8[dstIndex++] = ch;
            orgIndex++;
            continue;
        }

        // 处理16位Unicode的情况（最多3字节UTF-8）
        if(ch < 0xd800 || ch >= 0xe000)
        {
            if((ch & 0xf800) == 0)
            {
                // 高5位为0，说明是2字节UTF-8
                uint8_t value = (ch >> 6) | 0xc0;
                pUTF8[dstIndex++] = value;

                value = (ch & 0x3f) | 0x80;
                pUTF8[dstIndex++] = value;
            }
            else
            {
                // 否则为3字节UTF-8
                uint8_t value = (ch >> 12) | 0xe0;
                pUTF8[dstIndex++] = value;

                value = (ch >> 6) & 0x3f;
                value |= 0x80;
                pUTF8[dstIndex++] = value;

                value = ch & 0x3f;
                value |= 0x80;
                pUTF8[dstIndex++] = value;
            }
        }
        else
        {
            // 处理21位Unicode的情况
            uint32_t high = ch - 0xd800;
            uint32_t low = pUTF16[++orgIndex] - 0xdc00;
            uint32_t result = low + (high << 10);
            result += 0x00010000;

            uint8_t value = (result >> 18) | 0xf0;
            pUTF8[dstIndex++] = value;

            value = (result >> 12) & 0x3f;
            value |= 0x80;
            pUTF8[dstIndex++] = value;

            value = (result >> 6) & 0x3f;
            value |= 0x80;
            pUTF8[dstIndex++] = value;

            value = (result & 0x3f) | 0x80;
            pUTF8[dstIndex++] = value;
        }
```

```
        orgIndex++;
    }

    pUTF8[dstIndex] = '\0';

    if(pUTF8Length != NULL)
        *pUTF8Length = dstIndex;

    return true;
}

size_t ZennyGetUTF8LengthFromUTF16(const uint16_t *utf16Str)
{
    if(utf16Str == NULL)
        return 0;

    size_t orgIndex = 0, dstLength = 0;
    uint16_t ch;

    while((ch = utf16Str[orgIndex]) != u'\0')
    {
        // 处理ASCII码兼容情况
        if(ch < 0x80)
        {
            dstLength++;
            orgIndex++;
            continue;
        }

        // 处理16位Unicode的情况（最多3字节UTF-8）
        if(ch < 0xd800 || ch >= 0xe000)
        {
            if((ch & 0xf800) == 0)
            {
                // 高5位为0，说明是2字节UTF-8
                dstLength += 2;
            }
            else
            {
                dstLength += 3;
            }
        }
        else
        {
            // 否则为4字节UTF-8
            dstLength += 4;
            orgIndex++;
        }

        orgIndex++;
    }

    return dstLength;
}

size_t ZennyUTF16StrLen(const uint16_t *s)
{
    if(s == NULL)
        return 0;

    size_t index;

    for(index = 0; s[index] != u'\0'; index++);

    return index;
}

const uint16_t* ZennyUTF16StrCat(uint16_t dst[], const uint16_t src[])
```

```
{
    if(dst == NULL || src == NULL)
        return NULL;

    size_t index = ZennyUTF16StrLen(dst);

    for(size_t i = 0; src[i] != u'\0'; i++)
        dst[index++] = src[i];

    dst[index] = u'\0';

    return dst;
}
```

代码清单20-2给出了所有外部接口函数的完整的实现，并且在一些关键部分都写了注释。由于编码转换的算法本身并不太复杂，所以这段代码注释量不多，而且很容易就能看懂。

代码清单20-3给出了针对此编码转换工具的测试。

代码清单20-3　编码转换工具的测试

```
#import <Foundation/Foundation.h>
#include <string.h>

#include "zennyChar.h"

int main(int argc, const char * argv[])
{
    // 声明一个s字符数组，用于存放一个UTF-8字符串
    const char s[] = u8"你好，世界！αβγ□Hello, world!";

    printf("The length is: %zu, and the content is: %s\n", strlen(s), s);

    // 使用NSString对象来存放这个字符串，并且str对象引用所包含的字符编码已经转为了UTF-16
    NSString *str = [NSString stringWithUTF8String:s];
    NSLog(@"The length is: %zu, and the content is: %@", [str length], str);

    // 下面用我们自己实现的UTF-8转UTF-16的方法进行对比测试
    unichar buffer[64];
    size_t length;
    if(ZennyUTF8ToUTF16(buffer, s, &length))
        NSLog(@"\nThe transformed UTF16 length is: %zu, and the string is: %@",
        length, [NSString stringWithCharacters:buffer length:length]);

    NSLog(@"The UTF16 string length from UTF8 is: %zu, and the original
          UTF16 string length is: %zu", ZennyGetUTF16LengthFromUTF8(s),
          ZennyUTF16StrLen(buffer));

    // 再将UTF-16编码的字符串再转回UTF-8编码的字符串
    char chBuffer[64];
    ZennyUTF16ToUTF8(chBuffer, buffer, &length);
    printf("\nThe transformed UTF8 length is: %zu, and the string is: %s\n",
            length, chBuffer);
```

```
    length = ZennyGetUTF8LengthFromUTF16(buffer);
    printf("The UTF8 length from UTF16 is: %zu\n\n", length);

    // 最后测试一下ZennyUTF16StrCat函数
    ZennyUTF16StrCat(buffer, u"□拜拜～");

    length = ZennyUTF16StrLen(buffer);
    NSLog(@"The UTF16 length is: %zu, and the content is: %@", length,
          [NSString stringWithCharacters:buffer length:length]);

    return 0;
}
```

代码清单20-3中的测试字符串中既包含中文、英文、希腊文，同时还包括了Emoji，因此测试涉及的范围比较广。

20.4　本章小结

本章详细介绍了UTF-8编码格式以及UTF-16编码格式的具体规格。通过一个完整的项目我们介绍了如何在UTF-8编码的字符串与UTF-16编码的字符串之间相互转换，如何获取UTF-16字符串的长度等。通过这个项目各位能够学习到C语言程序设计中的常用技巧，比如对外函数接口通过一个头文件进行说明，还有C语言代码中对函数、对象命名的习惯规则，等等。

尽管本项目的实际算法不算复杂，但是它已经具备一定的模块化设计思想，我们直接将它作为库使用也完全没有问题。

第21章　制作控制台计算器

我们在前一章介绍了UTF-8与UTF-16之间相互转码的算法工具，如果说这个项目还算比较简单的话，那么本章将要介绍的项目将会复杂不少。我们将在本章给大家介绍如何制作基于控制台的计算器工具。

本计算器的功能特性有以下几点。

1）用户通过一个不带空格的完整的字符串作为该程序的参数，该字符串就是我们所要制作的程序对它进行解析的算术表达式。假定我们构建出来的程序名为calculator，那么我们在控制台输入calculator 1+2+3*4，然后输入回车，我们就能看到计算结果15。

2）本计算器支持的运算操作包括加法操作（+）、减法操作（-）、乘法操作（*）、除法操作（/），求模操作（%），指数求幂操作（^）以及括号操作。这里要注意的是，有些控制台将圆括号视作具有特殊意义的符号，因此不能作为程序的输入参数给出，而我们这里同时支持圆括号和方括号作为括号使用，在解析前会有预处理将碰到的方括号全都转换为圆括号。

3）本计算器支持以下这些常用数学函数：sin（正弦函数）、cos（余弦函数）、tan（正切函数）、cot（余切函数）、sinh（双曲正弦函数）、cosh（双曲余弦函数）、tanh（双曲正切函数）、asin（反正弦函数）、

acos（反余弦函数）、atan（反正切函数）、asnh（反双曲正弦函数）、acsh（反双曲余弦函数）、log（求底数为2的对数）、lg（求底数为10的对数）、ln（求底数为e的对数）、sqrt（求平方根）、cbrt（求立方根）、recp（求倒数）、rad（将角度值转为弧度值）、deg（将弧度制转为角度值）、exp（求e的幂）。这些数学函数都只含一个参数，并且需要使用括号将参数包裹起来，比如：calculattor log[4]+sqrt[9]。

4）本计算器还支持两个数学常量，一个是e，一个是pi。比如：calculator sin（pi）+　cos（0）-e^2。

5）在参数字符串中，所有字母均可以用大写和小写，解析程序的预处理器会将所有大写字母转换为小写字母。

下面，我们将先介绍此代码中的一些关键思想，然后给出完整的源代码。

21.1　对数字的解析

计算器中的算式解析过程中最重要的步骤之一就是要将数字、操作符、函数名等元素解析出来。其中对数字的解析是最复杂的，因此我们单独通过一节来为大家介绍如何解析算式中的数字。

首先，我们要定义好在算术表达式中哪些数字表达是合法的，哪些是不合法的。在本计算器程序中，这些都是合法的数字：10，20.5，.23，0。而这些不是合法的数字：36.52.34，因为在52之前已经有了一个小数点，而在34之前又出现了一个，此时解析器会在34之前的那个小数点处停止解析，然后直接返回当前状态。

有了上述合法数字表达的定义之后，我们就可以依此构造出一个状态机了，如图21-1所示。

图21-1中，实心圆圈表示起始状态；实心圆圈外面再加一圈的圈圈表示终止状态，在实际程序中则表示当前数字解析函数的返回；直角矩形表示一个动作行为；圆角矩形则表示当前状态；菱形表示条件判断。由于数学常量也表示一个特定的数，所以这里将数学常量与一般的数字放在一起解析。

21.2　对操作符的优先级处理

为了使我们的算术表达式尽可能与数学上的常用表达匹配，我们就需要对计算操作符做运算优先级的处理。在本计算器程序中安排了四个优先级，以下按照从小到大的次序排列：

1）+（加）、-（减）：加法与减法操作符的优先级是最低的。

2）*（乘）、/（除）、%（求模）：这些运算操作的优先级处于第二等级。

3）^（幂）：指数计算的优先级处于第三等级。

4）括号：括号的运算优先级是最高的。这里要说明的是，由于数学函数后面必须跟括号，所以函数操作的优先级也要大于以上3类操作符的优先级。

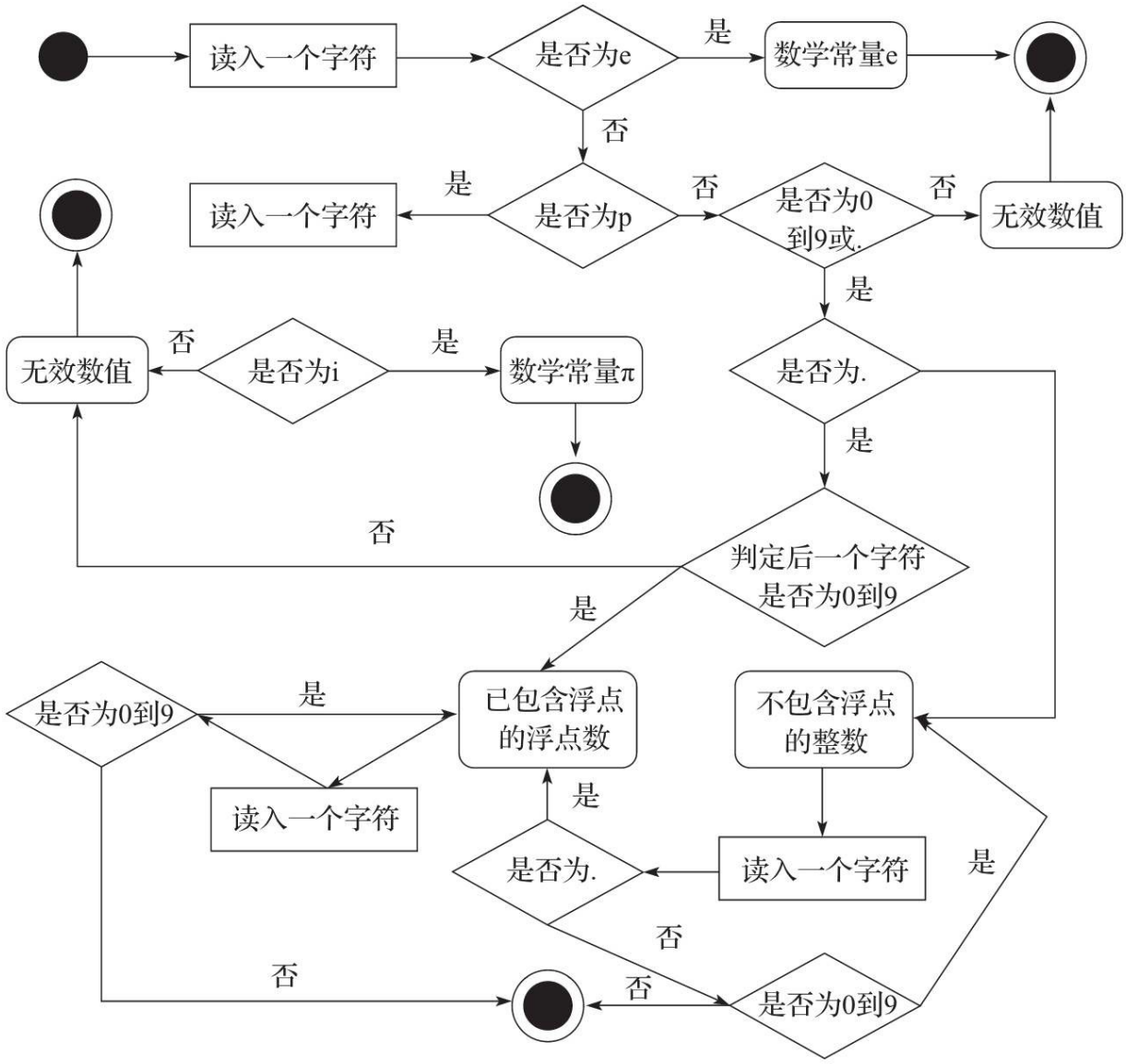


图21-1　数字解析状态机

对于相同优先级的运算操作，在同一函数调用层中按照从左到右的顺序做归约计算，比如：1+2-3，其计算次序就是先计算1+2的值，然后再计算3-3的值。

对于遇到比当前计算优先级更高的计算操作时，本计算器解析程序将

采用递归方式进行计算。比如1+2*（3+4），其递归调用次序如图21-2所示。

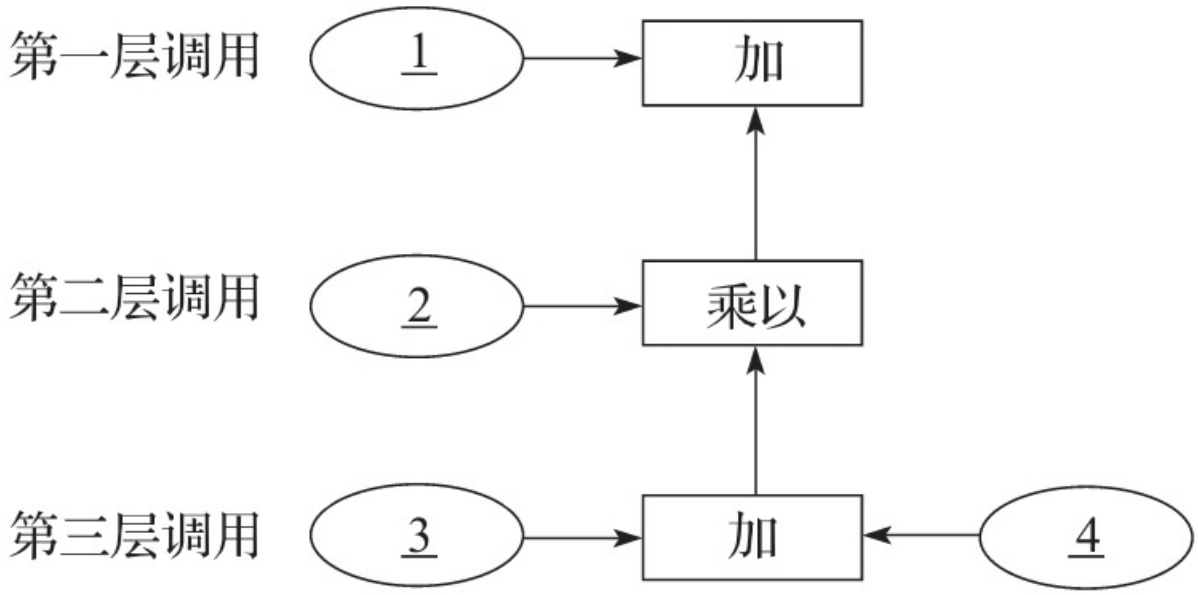


图21-2　四则运算的递归调用逻辑

图21-2中展示了计算式1+2*（3+4）的递归调用次序。首先，解析器遇到1+2时查看后面那个操作符的优先级，由于乘法优先级大于加法，因此将2*操作作为递归调用。随后遇到（符号，那么它的优先级比乘法操作高，因此再做一层递归调用。在第三层调用中，将3+4的结果计算完之后返回到第二层调用，做2*7的操作，然后完成该计算之后再返回到第一层调用，完成最后的1+14的计算，最终得到计算结果15。

而对于计算表达式2+3*4+5，2+处于第一层调用，而后面的3*4+5作为第二层调用，由于加法计算满足结合律，所以在这种情况下就相当于（2+3*4）+5=2+（3*4+5）。这里的一个副作用是减法不满足结合律，因

此我们要计算减法的时候，实际上要将减号的右操作数做取相反数操作，然后将减法变成加法。比如，我们要计算1-2*3-4，我们就要将它转为：1+（-2）*3+（-4）。

21.3　代码示例

我们前面已经大致介绍了这款控制台计算器程序。下面我们就要开始实现此控制台计算器的具体代码。

首先要声明的是，这份代码示例可以从笔者的GitHub上获得完整的代码资源：https://github.com/zenny-chen/SimpleCalculator

该项目就仅由一个main.c C源文件构成。这里面包含了main函数以及对计算表达式的解析和相关处理函数。代码清单21-1展示了main.c中的所有内容。

代码清单21-1　main.c源代码

```
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <stdlib.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdalign.h>

/** 我们这里使用简约的var作为对象类型的自动推导 */
#define var     __auto_type

/** 我们指定输入表达式的最大长度为2047字节，超出部分将被截断 */
#define MAX_ARGUMENT_LENGTH     2047

/** 用于标记解析符号时的当前状态 */
enum PARSE_PHASE_STATUS
{
    PARSE_PHASE_STATUS_LEFT_OPERAND = 0,
    PARSE_PHASE_STATUS_RIGHT_OPERAND,
    PARSE_PHASE_STATUS_LEFT_PARENTHESIS,
    PARSE_PHASE_STATUS_NEED_OPERATOR = 4,
    PARSE_PHASE_STATUS_HAS_NEG = 8
};

/** 当前算术计算优先级 */
enum OPERATOR_PRIORITY
{
    /** 加减法优先级 */
    OPERATOR_PRIORITY_ADD,
```

```
    /** 乘除以及求模优先级 */
    OPERATOR_PRIORITY_MUL,

    /** 幂运算优先级 */
    OPERATOR_PRIORITY_POW
};

/** 判定当前字符是否属于数字 */
static inline bool IsDigital(char ch)
{
    return ch >= '0' && ch <= '9';
}

/**
 * 判定当前是否为数学常量
 * @return 如果是数学常量，则返回该常量的字符个数，否则返回0
*/
static inline int IsMathConstant(const char *cursor)
{
    if(cursor[0] == 'p' && cursor[1] == 'i')
        return 2;

    // 由于本程序还支持exp函数用于计算e的指数幂，
    // 所以如果后面e后面包含了一个x，那么这个e就不会是数学常量
    if(cursor[0] == 'e' && cursor[1] != 'x')
        return 1;

    return 0;
}

/** 判定当前字符是否可能为函数 */
static inline bool IsMathFunction(char ch)
{
    return ch >= 'a' && ch <= 'z';
}

/** 对数字进行解析 */
static double ParseDigital(const char *cursor, int *pRetLength)
{
    // 我们先判断当前是否为数学常量
    var length = IsMathConstant(cursor);
    if(length > 0)
    {
        *pRetLength = length;
        return (cursor[0] == 'p')? M_PI : M_E;
    }

    char value[MAX_ARGUMENT_LENGTH + 1];

    char ch;
    var index = 0;
    var hasDot = false;

    do
    {
        ch = cursor[index];
        if(ch == '.')
        {
            // 如果之前已经出现了小数点，那么这里就中断解析
            if(hasDot)
                break;
            else
            {
                hasDot = true;
                value[index] = ch;
            }
        }
        else if(!IsDigital(ch))
```

```
            break;  // 在其余情况下，倘若不是数字，则立即中断解析

        value[index++] = ch;
    }
    while(ch != '\0');

    value[index] = '\0';

    *pRetLength = index;

    // atof函数是将一个字符数组中的内容转为一个double类型的浮点数值
    // 该函数在<stdlib.h>头文件中声明
    return atof(value);
}

static double AddOp(double a, double b)
{
    return a + b;
}

static double MinusOp(double a, double b)
{
    return a - b;
}

static double MulOp(double a, double b)
{
    return a * b;
}

static double DivOp(double a, double b)
{
    return a / b;
}

static double ModOp(double a, double b)
{
    // 由于求模操作时，操作数必须是整数，
    // 所以我们这里将a与b都转换为带符号的64位整数类型
    return (int64_t)a % (int64_t)b;
}

/** 定义了一个操作函数表，方便快速定位当前操作符所对应的操作函数 */
static double (* const opFuncTables[])(double, double) = {
    // 为了进一步节省全局存储空间，我们这里将根据ASCII码表找出最小的字符值，
    // 将该值作为0，后续的都减去该值。通过ASCII表可以知道，值最小的符号是%
    // 它的值为0x25。然后我们可以用指定索引的初始化器对opFuncTables进行初始化
    ['%' - '%'] = &ModOp,
    ['*' - '%'] = &MulOp,
    ['+' - '%'] = &AddOp,
    ['-' - '%'] = &MinusOp,
    ['/' - '%'] = &DivOp,
    ['^' - '%'] = &pow
};

/** 判定是否为有效操作符 */
static inline bool IsOperator(char ch)
{
    return (ch >= '%' && ch <= '/') || ch == '^';
}

static double radian(double degree)
{
    return degree * M_PI / 180.0;
}

static double degree(double radian)
{
    return radian * 180.0 / M_PI;
```

```
}

static double cot(double radian)
{
    return tan(M_PI * 0.5 - radian);
}

static double recp(double x)
{
    return 1.0 / x;
}

static const struct
{
    int name;
    double (*pFunc)(double);
} mathFuncList[] = {
    { '\0nis', &sin },
    { '\0soc', &cos },
    { '\0nat', &tan },
    { '\0toc', &cot },
    { 'hnis', &sinh },
    { 'hsoc', &cosh },
    { 'hnat', &tanh },
    { 'nisa', &asin },
    { 'soca', &acos },
    { 'nata', &atan },
    { 'hnsa', &asinh },
    { 'hsca', &acosh },
    { '\0gol', &log2 },
    { '\0\0gl', &log10 },
    { '\0\0nl', &log },
    { 'trqs', &sqrt },
    { 'trbc', cbrt },
    { 'pcer', &recp },
    { '\0dar', &radian },
    { '\0ged', &degree },
    { '\0pxe', &exp }
};

static double (*ParseMathFunction(const char *cursor, int *pLength))(double)
{
    // 这里buffer至少要求4字节对齐。因为我们后面会对前4个字节内容进行同时访问
    char alignas(4) buffer[8] = { '\0' };
    var index = 0;

    // 由于这里规定的数学符号最多占用4个字节，所以需要用一个计数器来防止访问越界
    for(var count = 0; count < 4; count++, index++)
    {
        var ch = cursor[index];
        if(!IsMathFunction(ch))
            break;

        buffer[index] = ch;
    }
    buffer[index] = '\0';

    // 同时取出刚才所存放的4个字节内容，方便比较
    var value = *(int*)buffer;

    // 查找函数表中是否含有该函数名
    const var length = sizeof(mathFuncList) / sizeof(mathFuncList[0]);
    for(typeof(length + 0) i = 0; i < length; i++)
    {
        if(mathFuncList[i].name == value)
        {
            *pLength = index;
            return mathFuncList[i].pFunc;
        }
```

```
    }

    return NULL;
}

/**
 * 解析当前的算术表达式
 * @param ppCursor 指向当前算术表达式字符串的地址。
 * 它既是输入又是输出。当当前算术表达式作为括号进行计算时，
 * 需要将右括号的位置输出到实参。
 * @param leftOperand 当前左操作数的值
 * @param status 当前计算状态
 * @param priority 当前计算的算术优先级
 * @param pStatus 输出解析状态
 * @return 输出计算表达式的结果
*/
static double ParseArithmeticExpression(const char **ppCursor, double
leftOperand, enum PARSE_PHASE_STATUS status, enum OPERATOR_PRIORITY
priority, bool *pStatus)
{
    const char *cursor = *ppCursor;
    var rightOperand = 0.0;

    var length = 0;

    // 指向操作符函数的指针
    double (*pOpFunc)(double, double) = NULL;

    // 指向数学函数的指针
    double (*pMathFunc)(double) = NULL;

    bool isSuccessful = true;

    char ch;

    do
    {
        ch = *cursor;

        // 先判定当前字符是否属于数字或数学常量
        if(IsDigital(ch) || IsMathConstant(cursor) > 0)
        {
            double value = ParseDigital(cursor, &length);
            cursor += length;

            if((status & PARSE_PHASE_STATUS_RIGHT_OPERAND) == PARSE_PHASE_
            STATUS_LEFT_OPERAND)
            {
                leftOperand = value;
                // 如果具有负数符号，则将左操作数做取相反数操作
                if((status & PARSE_PHASE_STATUS_HAS_NEG) != 0)
                    leftOperand = -leftOperand;
            }
            else
            {
                // 对于当前为右操作数的情况，根据操作符计算优先级，
                // 需要进一步判定后面的操作优先级是否大于前面的，
                // 如果大于前面的，则需要做递归计算
                rightOperand = value;
                // 如果具有负数符号，则将左操作数做取相反数操作
                if((status & PARSE_PHASE_STATUS_HAS_NEG) != 0)
                    rightOperand = -rightOperand;
            }

            // 清除负数标志
            status &= ~PARSE_PHASE_STATUS_HAS_NEG;

            // 添加后续需要算术操作符的状态标志
```

```
    status |= PARSE_PHASE_STATUS_NEED_OPERATOR;
}
else if(IsMathFunction(ch))
{
    pMathFunc = ParseMathFunction(cursor, &length);
    if(pMathFunc == NULL)
    {
        // 如果数学函数返回空，说明解析失败，立即中断解析
        isSuccessful = false;
        break;
    }
    cursor += length;
    if(*cursor != '(')
    {
        // 如果函数后面没有跟(，那也不是一个合法的表达式，立即中断解析
        isSuccessful = false;
        break;
    }
}
else if(IsOperator(ch))
{
    // 这个区间范围内包含了常用的算术操作符以及左右圆括号，
    // 因此我们在这个分支中同时对这两类符号进行解析判断
    if(ch == '(')
    {
        cursor++;

        double value = ParseArithmeticExpression(&cursor, 0.0,
        PARSE_PHASE_STATUS_LEFT_OPERAND | PARSE_PHASE_STATUS_LEFT_
        PARENTHESIS, OPERATOR_PRIORITY_ADD, &isSuccessful);

        // 如果当前游标所指向的字符不是')'，说明没有匹配到合适的)，中断解析
        if(!isSuccessful || *cursor != ')')
        {
            isSuccessful = false;
            break;
        }

        if((status & PARSE_PHASE_STATUS_RIGHT_OPERAND) == 0)
        {
            // 如果当前状态为左操作数
            if(pMathFunc != NULL)
            {
                leftOperand = pMathFunc(value);
                pMathFunc = NULL;
            }
            else
                leftOperand = value;
        }
        else
        {
            // 如果当前状态为右操作数
            if(pMathFunc != NULL)
            {
                rightOperand = pMathFunc(value);
                pMathFunc = NULL;
            }
            else
                rightOperand = value;
        }

        // 清除当前左括号状态
        status &= ~PARSE_PHASE_STATUS_LEFT_PARENTHESIS;
```

```
    // 添加后续需要算术操作符的状态标志
    status |= PARSE_PHASE_STATUS_NEED_OPERATOR;
}
else if(ch == ')')
{
    // 将当前游标位置输出给实参
    *ppCursor = cursor;

    return (pOpFunc != NULL)? pOpFunc(leftOperand, rightOperand) :
            leftOperand;
}
else
{
    if((status & PARSE_PHASE_STATUS_NEED_OPERATOR) == 0)
    {
        if(ch == '-')
        {
            // 如果当前不需要操作符，则将减号视作负数符号
            // 作为负数符号的话，后面必须跟一个数，否则也是无效的
            if(IsDigital(cursor[1]) || IsMathConstant(&cursor[1]) > 0)
               status |= PARSE_PHASE_STATUS_HAS_NEG;
            else
            {
                isSuccessful = false;
                break;
            }
        }
        else
        {
            // 对于其他情况，如果当前状态不需要操作符，那么表达式非法，
            // 立即中断解析
            isSuccessful = false;
            break;
        }
    }
    else
    {
        var tmpFunc = opFuncTables[ch - '%'];
        if(tmpFunc == NULL)
        {
            // 如果没找到对应的操纵符函数，说明当前输入字符是非法的，
            // 直接中断解析
            isSuccessful = false;
            break;
        }
        // 判定当前操作符的计算优先级
        var pry = OPERATOR_PRIORITY_ADD;
        if(tmpFunc == ModOp || tmpFunc == MulOp || tmpFunc == DivOp)
            pry = OPERATOR_PRIORITY_MUL;
        else if(tmpFunc == pow)
            pry = OPERATOR_PRIORITY_POW;

        if(pOpFunc == NULL)
            pOpFunc = tmpFunc;

        if((status & PARSE_PHASE_STATUS_RIGHT_OPERAND) == 0)
        {
            // 当前操作符解析成功，后续将需要该操作的右操作数
            status |= PARSE_PHASE_STATUS_RIGHT_OPERAND;
        }
        else
        {
            // 如果之前优先级不小于当前操作符的优先级，那么立即做归约
            if(priority >= pry)
```

```
                    {
                        leftOperand = pOpFunc(leftOperand, rightOperand);
                        rightOperand = 0.0;
                        // 随后更新当前操作函数以及计算优先级
                        pOpFunc = tmpFunc;
                    }
                    else
                    {
                        // 如果当前碰到了比之前优先级更改的操作符，
                        // 那么我们采用递归的方式进行计算
                        if(pOpFunc == MinusOp)
                        {
                            // 如果之前的计算是减法，那么根据减法不适用于结合律的
                            // 性质，我们这里将它作为一个加法，并且将右操作数取负
                            pOpFunc = AddOp;
                            rightOperand = -rightOperand;
                        }
                        else if(pOpFunc == DivOp)
                        {
                            // 如果之前的计算是除法，那么根据除法不适用于结合律的
                            // 性质，我们这里将它作为一个乘法，并且将右操作数取其倒数
                            pOpFunc = MulOp;
                            rightOperand = 1.0 / rightOperand;
                        }

                        // 递归做高优先级的运算操作
                        var value = ParseArithmeticExpression(&cursor,
                        rightOperand, PARSE_PHASE_STATUS_LEFT_OPERAND |
                        PARSE_PHASE_STATUS_NEED_OPERATOR, pry, pStatus);

                        // 由于我们可能会碰到在括号操作符中的高优先级运算的归约
                        // 比如考虑这个表达式：(1+2*3)
                        // 这里，2*3)会在同一个调用级中，所以)的输出无法影响到
                        // 先前的(，所以这里我们也要将当前游标位置进行输出，
                        // 返回给上一级的调用
                        *ppCursor = cursor;
                        return pOpFunc(leftOperand, value);
                    }
                }
                // 更新当前计算优先级
                priority = pry;
            }
            // 清除需要操作符标志
            status &= ~PARSE_PHASE_STATUS_NEED_OPERATOR;
        }

        // 对于所有操作符情况，最后都让游标往前走一格
        cursor++;
    }
    else
    {
        // 如果遇到其他字符，倘若不是字符串结束符则宣告解析失败
        if(ch != '\0')
        {
            isSuccessful = false;
            break;
        }
    }
}
while(ch != '\0');

// 若解析失败，则后续出结果时不做任何相关计算
if(!isSuccessful)
    pOpFunc = NULL;
```

```
    if(pStatus != NULL)
        *pStatus = isSuccessful;

    return (pOpFunc == NULL)? leftOperand : pOpFunc(leftOperand, rightOperand);
}

/**
 * 计算输入的算术表达式
 * @param expr 输入的算术表达式字符串
 * @param result 以字符串的形式输出结果，这里设置了实参至少需要提供的缓存长度
 * @return 如果表达式解析成功，返回true，否则返回false
*/
bool CalculateArithmeticExpression(char expr[], char result[static 32])
{
    if(expr[0] == '\0')
        return false;

    /*** 我们先对输入字符串做一些过滤，使得当中出现的一些符号能适配本程序 */
    var length = (int)strlen(expr);

    // 由于一些命令控制台不支持带有圆括号()的表达式，但支持方括号[]表达式，
    // 所以我们这里可以将输入中的[]再替换回()。
    // 此外，我们将出现的所有大写字母替换为小写字母
    for(var i = 0; i < length; i++)
    {
        var ch = expr[i];
        if(ch == '[')
            expr[i] = '(';
        else if(ch == ']')
            expr[i] = ')';
        else if(ch == '$')
            expr[i] = '^';
        else if(ch >= 'A' && ch <= 'Z')
        {
            // 由于ASCII码的巧妙设计，大写字母与小写字母正好相差0x20，
            // 所以我们这里只需通过加上0x20值就能方便地将大写字母转为小写字母
            expr[i] += 0x20;
        }
    }

    bool ret = false;

    var value = ParseArithmeticExpression((const char**)&expr, 0.0,
    PARSE_PHASE_STATUS_LEFT_OPERAND, OPERATOR_PRIORITY_ADD, &ret);

    if(!ret)
        return ret;

    // 我们将结果显示为小数点后面跟8位尾数
    sprintf(result, "%.8f", value);

    // 我们下面将把多余的.00000这种字样给过滤掉，使得结果输出更好看一些
    length = (int)strlen(result);
    var dotIndex = -1;
    var hasE = false;
    for(var i = 0; i < length; i++)
    {
        if(result[i] == '.')
            dotIndex = i;
        else if(result[i] == 'e')
            hasE = true;
    }
    // 我们只有在仅存在小数点的情况下做过滤
```

```
    if(dotIndex >= 0 && !hasE)
    {
        var index = length;

        while(--index > 0)
        {
            if(result[index] != '0')
                break;

            result[index] = '\0';
        }
        // 如果dotIndex后面没有具体数字了，那么我们将小数点也过滤掉
        if(result[dotIndex + 1] == '\0')
            result[dotIndex] = '\0';
    }

    return true;
}


int main(int argc, const char * argv[])
{
    // argc用于存放参数个数。在Windows以及各类Unix系统中，参数以空格符进行分隔。
    // 因此我们在使用该程序时，计算表达式中不应该含有空格符（包括空格和制表符）。
    // 假定我们生成的可执行程序名为SimpleCalculator，那么在控制台输入：
    // SimpleCalculator 1+2是合法的。
    // 而输入：SimpleCalculator 1 + 2，则直接输出结果1，后面的+2会被忽略
    if(argc < 2)
    {
        puts("No expression to calculate!");
        return 0;
    }

    var length = strlen(argv[1]);
    if(length == 0)
    {
        puts("No expression to calculate!");
        return 0;
    }

    // 对参数表达式长度做截断，取由宏指定的长度
    if(length > MAX_ARGUMENT_LENGTH)
        length = MAX_ARGUMENT_LENGTH;

    // 我们准备一个字符串缓存，将通过程序参数传递进来的字符串表达式拷贝到当前程序的栈上，
    // 便于后续解析操作
    char argBuffer[MAX_ARGUMENT_LENGTH + 1];

    // 我们这里使用strncpy也使得在做字符串拷贝的时候确保长度不超过指定的length大小。
    // 这里之所以使用strncpy而不是memcpy，
    // 是因为strncpy会在目标缓存最后添加一个字符串结束符'\0'。
    // 该函数在<string.h>头文件中
    strncpy(argBuffer, argv[1], length);

    char result[32];

    var state = CalculateArithmeticExpression(argBuffer, result);

    printf("The arithmetic expression to be calculated: %s\n", argBuffer);

    if(state)
        printf("The answer is: %s\n", result);
    else
        puts("Invalid expression!");
```

```
}
```

代码清单21-1中的代码必须在GCC 4.9或更高版本、Clang 3.8或更高版本，以及Apple LLVM 8.0或更高版本上才能通过编译构建。此外，编译选项中必须加入-std=gnu11。

21.4　本章小结

本章给大家介绍了如何使用C语言制作一个基于控制台的计算器。尽管这个功能看上去不算复杂，但实际实现起来还是有一定的复杂度。本程序中，笔者通过递归来解决算术表达式的计算优先级问题，通过使用有限状态机来逐字节地解析当前输入符号，将它归类。在一开始先建立了一个状态图，这有助于对此程序的理解。

我们可以在Windows以及其他类Unix系统中编译这段代码然后运行。当然对编译器的要求是GCC 4.9或更高版本、Clang 3.8或更高版本。如果各位在Windows系统上的话，可以使用MS-Clang Mingw编译器进行编译构建。