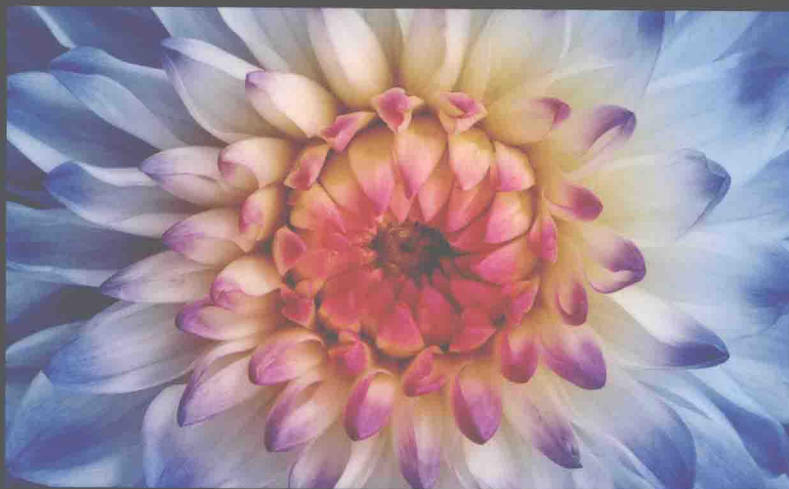


PEARSON



# C Primer Plus

第6版

英文版（下册）

[美] Stephen Prata 著

C Primer Plus (6th Edition)



中国工信出版集团



人民邮电出版社  
POSTS & TELECOM PRESS



# C Primer Plus

第6版

英文版（下册）

[美] Stephen Prata 著

C Primer Plus (6th Edition)

人民邮电出版社  
北 京

# 目录

<b>12 Storage Classes, Linkage, and Memory Management</b>	<b>511</b>
<b>第 12 章 存储类别、链接和内存管理</b>	
Storage Classes / 存储类别	511
Scope / 作用域	513
Linkage / 链接	515
Storage Duration / 存储期	516
Automatic Variables / 自动变量	518
Register Variables / 寄存器变量	522
Static Variables with Block Scope / 块作用域的静态变量	522
Static Variables with External Linkage / 外部链接的静态变量	524
Static Variables with Internal Linkage / 内部链接的静态变量	529
Multiple Files / 多文件	530
Storage-Class Specifier Roundup / 存储类别说明符	530
Storage Classes and Functions / 存储类别和函数	533
Which Storage Class? / 存储类别的选择	534
A Random-Number Function and a Static Variable / 随机数函数和静态变量	534
Roll'Em / 掷骰子	538
Allocated Memory: malloc() and free() / 分配内存: malloc() 和 free()	543
The Importance of free() / free() 的重要性	547
The calloc() Function / calloc() 函数	548
Dynamic Memory Allocation and Variable-Length Arrays / 动态内存分配和变长数组	548
Storage Classes and Dynamic Memory Allocation / 存储类别和动态内存分配	549
ANSI C Type Qualifiers / ANSI C 类型限定符	551
The const Type Qualifier / const 类型限定符	552
The volatile Type Qualifier / volatile 类型限定符	554
The restrict Type Qualifier / restrict 类型限定符	555

The _Atomic Type Qualifier (C11) / _Atomic 类型限定符 (C11)	556
New Places for Old Keywords / 旧关键字的新位置	557
Key Concepts / 关键概念	558
Summary / 本章小结	558
Review Questions / 复习题	559
Programming Exercises / 编程练习	561
<b>13 File Input/Output</b>	<b>565</b>
<b>第 13 章 文件输入/输出</b>	
Communicating with Files / 与文件进行通信	565
What Is a File? / 文件是什么	566
The Text Mode and the Binary Mode / 文本模式和二进制模式	566
Levels of I/O / I/O 的级别	568
Standard Files / 标准文件	568
Standard I/O / 标准 I/O	568
Checking for Command-Line Arguments / 检查命令行参数	569
The fopen() Function / fopen() 函数	570
The getc() and putc() Functions / getc() 和 putc() 函数	572
End-of-File / 文件结尾	572
The fclose() Function / fclose() 函数	574
Pointers to the Standard Files / 指向标准文件的指针	574
A Simple-Minded File-Condensing Program / 一个简单的文件压缩程序	574
File I/O: fprintf(), fscanf(), fgets(), and fputs()	
/ 文件 I/O: fprintf()、fscanf()、fgets() 和 fputs()	576
The fprintf() and fscanf() Functions / fprintf() 和 fscanf() 函数	576
The fgets() and fputs() Functions / fgets() 和 fputs() 函数	578
Adventures in Random Access: fseek() and ftell()	
/ 随机访问: fseek() 和 ftell()	579
How fseek() and ftell() Work / fseek() 和 ftell() 的工作原理	580
Binary Versus Text Mode / 二进制模式和文本模式	582
Portability / 可移植性	582
The fgetpos() and fsetpos() Functions / fgetpos() 和 fsetpos() 函数	583
Behind the Scenes with Standard I/O / 标准 I/O 的机理	583

<b>Other Standard I/O Functions / 其他标准 I/O 函数</b>	<b>584</b>
The int ungetc(int c, FILE *fp) Function / int ungetc(int c, FILE *fp) 函数	585
The int fflush() Function / int fflush() 函数	585
The int setvbuf() Function / int setvbuf() 函数	585
Binary I/O: fread() and fwrite() / 二进制 I/O: fread() 和 fwrite()	586
The size_t fwrite() Function / size_t fwrite() 函数	588
The size_t fread() Function / size_t fread() 函数	588
The int feof(FILE *fp) and int ferror(FILE *fp) Functions / int feof(FILE *fp) 和 int ferror(FILE *fp) 函数	589
An fread() and fwrite() Example / 一个程序示例	589
Random Access with Binary I/O / 用二进制 I/O 进行随机访问	593
<b>Key Concepts / 关键概念</b>	<b>594</b>
<b>Summary / 本章小结</b>	<b>595</b>
<b>Review Questions / 复习题</b>	<b>596</b>
<b>Programming Exercises / 编程练习</b>	<b>598</b>
<b>14 Structures and Other Data Forms</b>	<b>601</b>
<b>第 14 章 结构和其他数据形式</b>	
Sample Problem: Creating an Inventory of Books / 示例问题: 创建图书目录	601
Setting Up the Structure Declaration / 建立结构声明	604
Defining a Structure Variable / 定义结构变量	604
Initializing a Structure / 初始化结构	606
Gaining Access to Structure Members / 访问结构成员	607
Initializers for Structures / 结构的初始化器	607
Arrays of Structures / 结构数组	608
Declaring an Array of Structures / 声明结构数组	611
Identifying Members of an Array of Structures / 标识结构数组的成员	612
Program Discussion / 程序讨论	612
Nested Structures / 嵌套结构	613
Pointers to Structures / 指向结构的指针	615
Declaring and Initializing a Structure Pointer / 声明和初始化结构指针	617
Member Access by Pointer / 用指针访问成员	617
Telling Functions About Structures / 向函数传递结构的信息	618

Passing Structure Members / 传递结构成员	618
Using the Structure Address / 传递结构的地址	619
Passing a Structure as an Argument / 传递结构	621
More on Structure Features / 其他结构特性	622
Structures or Pointer to Structures? / 结构和结构指针的选择	626
Character Arrays or Character Pointers in a Structure / 结构中的字符数组和字符指针	627
Structure, Pointers, and malloc() / 结构、指针和 malloc()	628
Compound Literals and Structures (C99) / 复合字面量和结构 (C99)	631
Flexible Array Members (C99) / 伸缩型数组成员 (C99)	633
Anonymous Structures (C11) / 匿名结构 (C11)	636
Functions Using an Array of Structures / 使用结构数组的函数	637
Saving the Structure Contents in a File / 把结构内容保存到文件中	639
A Structure-Saving Example / 保存结构的程序示例	640
Program Points / 程序要点	643
Structures: What Next? / 链式结构	644
Unions: A Quick Look / 联合简介	645
Using Unions / 使用联合	646
Anonymous Unions (C11) / 匿名联合 (C11)	647
Enumerated Types / 枚举类型	649
enum Constants / enum 常量	649
Default Values / 默认值	650
Assigned Values / 赋值	650
enum Usage / enum 的用法	650
Shared Namespaces / 共享名称空间	652
typedef: A Quick Look / typedef 简介	653
Fancy Declarations / 其他复杂的声明	655
Functions and Pointers / 函数和指针	657
Key Concepts / 关键概念	665
Summary / 本章小结	665
Review Questions / 复习题	666
Programming Exercises / 编程练习	669

<b>15 Bit Fiddling</b>	<b>673</b>
<b>第 15 章 位操作</b>	
Binary Numbers, Bits, and Bytes / 二进制数、位和字节	674
Binary Integers / 二进制整数	674
Signed Integers / 有符号整数	675
Binary Floating Point / 二进制浮点数	676
Other Number Bases / 其他进制数	676
Octal / 八进制	677
Hexadecimal / 十六进制	677
C's Bitwise Operators / C 按位运算符	678
Bitwise Logical Operators / 按位逻辑运算符	678
Usage: Masks / 用法: 掩码	680
Usage: Turning Bits On (Setting Bits) / 用法: 打开位 (设置位)	681
Usage: Turning Bits Off (Clearing Bits) / 用法: 关闭位 (清空位)	682
Usage: Toggling Bits / 用法: 切换位	683
Usage: Checking the Value of a Bit / 用法: 检查位的值	683
Bitwise Shift Operators / 移位运算符	684
Programming Example / 编程示例	685
Another Example / 另一个例子	688
Bit Fields / 位字段	690
Bit-Field Example / 位字段示例	692
Bit Fields and Bitwise Operators / 位字段和按位运算符	696
Alignment Features (C11) / 对齐特性 (C11)	703
Key Concepts / 关键概念	705
Summary / 本章小结	706
Review Questions / 复习题	706
Programming Exercises / 编程练习	708
<b>16 The C Preprocessor and the C Library</b>	<b>711</b>
<b>第 16 章 C 预处理器和 C 库</b>	
First Steps in Translating a Program / 翻译程序的第一步	712
Manifest Constants: #define / 明示常量: #define	713



Tokens / 记号	717
Redefining Constants / 重定义常量	717
Using Arguments with #define / 在#define 中使用参数	718
Creating Strings from Macro Arguments: The # Operator	
/ 用宏参数创建字符串: #运算符	721
Preprocessor Glue: The ## Operator / 预处理器粘合剂: ##运算符	722
Variadic Macros: ... and __VA_ARGS__ / 变参宏: ...和 __VA_ARGS__	723
Macro or Function? / 宏和函数的选择	725
File Inclusion: #include / 文件包含: #include	726
Header Files: An Example / 头文件示例	727
Uses for Header Files / 使用头文件	729
Other Directives / 其他指令	730
The #undef Directive / #undef 指令	731
Being Defined—The C Preprocessor Perspective / 从 C 预处理器角度看已定义	731
Conditional Compilation / 条件编译	731
Predefined Macros / 预定义宏	737
#line and #error / #line 和#error	738
#pragma / #pragma	739
Generic Selection (C11) / 泛型选择 (C11)	740
Inline Functions (C99) / 内联函数 (C99)	741
_Noreturn Functions (C11) / _Noreturn 函数 (C11)	744
The C Library / C 库	744
Gaining Access to the C Library / 访问 C 库	745
Using the Library Descriptions / 使用库描述	746
The Math Library / 数学库	747
A Little Trigonometry / 三角问题	748
Type Variants / 类型变体	750
The tgmath.h Library (C99) / tgmath.h 库 (C99)	752
The General Utilities Library / 通用工具库	753
The exit() and atexit() Functions / exit() 和 atexit() 函数	753
The qsort() Function / qsort() 函数	755
The Assert Library / 断言库	760
Using assert / assert 的用法	760



<code>_Static_assert (C11) / _Static_assert (C11)</code>	762
<code>memcpy()</code> and <code>memmove()</code> from the <code>string.h</code> Library	
/ <code>string.h</code> 库中的 <code>memcpy()</code> 和 <code>memmove()</code>	763
Variable Arguments: <code>stdarg.h</code> / 可变参数: <code>stdarg.h</code>	765
Key Concepts / 关键概念	768
Summary / 本章小结	768
Review Questions / 复习题	768
Programming Exercises / 编程练习	770
<b>17 Advanced Data Representation</b>	<b>773</b>
<b>第 17 章 高级数据表示</b>	
Exploring Data Representation / 研究数据表示	774
Beyond the Array to the Linked List / 从数组到链表	777
Using a Linked List / 使用链表	781
Afterthoughts / 反思	786
Abstract Data Types (ADTs) / 抽象数据类型 (ADT)	786
Getting Abstract / 建立抽象	788
Building an Interface / 建立接口	789
Using the Interface / 使用接口	793
Implementing the Interface / 实现接口	796
Getting Queued with an ADT / 队列 ADT	804
Defining the Queue Abstract Data Type / 定义队列抽象数据类型	804
Defining an Interface / 定义一个接口	805
Implementing the Interface Data Representation / 实现接口数据表示	806
Testing the Queue / 测试队列	815
Simulating with a Queue / 用队列进行模拟	818
The Linked List Versus the Array / 链表和数组	824
Binary Search Trees / 二叉查找树	828
A Binary Tree ADT / 二叉树 ADT	829
The Binary Search Tree Interface / 二叉查找树接口	830
The Binary Tree Implementation / 二叉树的实现	833
Trying the Tree / 使用二叉树	849
Tree Thoughts / 树的思想	854

Other Directions / 其他说明	856
Key Concepts / 关键概念	856
Summary / 本章小结	857
Review Questions / 复习题	857
Programming Exercises / 编程练习	858

## **A Answers to the Review Questions** 861

### **附录 A 复习题答案**

Answers to Review Questions for Chapter 1 / 第 1 章复习题答案	861
Answers to Review Questions for Chapter 2 / 第 2 章复习题答案	862
Answers to Review Questions for Chapter 3 / 第 3 章复习题答案	863
Answers to Review Questions for Chapter 4 / 第 4 章复习题答案	866
Answers to Review Questions for Chapter 5 / 第 5 章复习题答案	869
Answers to Review Questions for Chapter 6 / 第 6 章复习题答案	872
Answers to Review Questions for Chapter 7 / 第 7 章复习题答案	876
Answers to Review Questions for Chapter 8 / 第 8 章复习题答案	879
Answers to Review Questions for Chapter 9 / 第 9 章复习题答案	881
Answers to Review Questions for Chapter 10 / 第 10 章复习题答案	883
Answers to Review Questions for Chapter 11 / 第 11 章复习题答案	886
Answers to Review Questions for Chapter 12 / 第 12 章复习题答案	890
Answers to Review Questions for Chapter 13 / 第 13 章复习题答案	891
Answers to Review Questions for Chapter 14 / 第 14 章复习题答案	894
Answers to Review Questions for Chapter 15 / 第 15 章复习题答案	898
Answers to Review Questions for Chapter 16 / 第 16 章复习题答案	899
Answers to Review Questions for Chapter 17 / 第 17 章复习题答案	901

## **B Reference Section** 905

### **附录 B 参考资料**

Section I : Additional Reading / 参考资料 I : 补充阅读	905
Online Resources / 在线资源	905
C Language Books / C 语言书籍	907
Programming Books / 编程书籍	907
Reference Books / 参考书籍	908

C++ Books / C++书籍	908
<b>Section II: C Operators / 参考资料 II: C 运算符</b>	<b>908</b>
Arithmetic Operators / 算术运算符	909
Relational Operators / 关系运算符	910
Assignment Operators / 赋值运算符	910
Logical Operators / 逻辑运算符	911
The Conditional Operator / 条件运算符	911
Pointer-Related Operators / 与指针有关的运算符	912
Sign Operators / 符号运算符	912
Structure and Union Operators / 结构和联合运算符	912
Bitwise Operators / 按位运算符	913
Miscellaneous Operators / 混合运算符	914
<b>Section III: Basic Types and Storage Classes</b>	
/ 参考资料 III: 基本类型和存储类别	915
Summary: The Basic Data Types / 总结: 基本数据类型	915
Summary: How to Declare a Simple Variable / 总结: 如何声明一个简单变量	917
Summary: Qualifiers / 总结: 限定符	919
<b>Section IV: Expressions, Statements, and Program Flow</b>	
/ 参考资料 IV: 表达式、语句和程序流	920
Summary: Expressions and Statements / 总结: 表达式和语句	920
Summary: The while Statement / 总结: while 语句	921
Summary: The for Statement / 总结: for 语句	921
Summary: The do while Statement / 总结: do while 语句	922
Summary: Using if Statements for Making Choices / 总结: if 语句	923
Summary: Multiple Choice with switch / 带多重选择的 switch 语句	924
Summary: Program Jumps / 总结: 程序跳转	925
<b>Section V: The Standard ANSI C Library with C99 and C11 Additions</b>	
/ 参考资料 V: 新增 C99 和 C11 的 ANSI C 库	926
Diagnostics: assert.h / 断言: assert.h	926
Complex Numbers: complex.h (C99) / 复数: complex.h (C99)	927
Character Handling: ctype.h / 字符处理: ctype.h	929
Error Reporting: errno.h / 错误报告: errno.h	930
Floating-Point Environment: fenv.h (C99) / 浮点环境: fenv.h (C99)	930

Floating-point Characteristics: float.h / 浮点特性: float.h	933
Format Conversion of Integer Types: inttypes.h (C99) / 整数类型的格式转换: inttypes.h	935
Alternative Spellings: iso646.h / 可选拼写: iso646.h	936
Localization: locale.h / 本地化: locale.h	936
Math Library: math.h / 数学库: math.h	939
Non-Local Jumps: setjmp.h / 非本地跳转: setjmp.h	945
Signal Handling: signal.h / 信号处理: signal.h	945
Alignment: stdalign.h (C11) / 对齐: stdalign.h (C11)	946
Variable Arguments: stdarg.h / 可变参数: stdarg.h	947
Atomics Support: stdatomic.h (C11) / 原子支持: stdatomic.h (C11)	948
Boolean Support: stdbool.h (C99) / 布尔支持: stdbool.h (C99)	948
Common Definitions: stddef.h / 通用定义: stddef.h	948
Integer Types: stdint.h / 整数类型: stdint.h	949
Standard I/O Library: stdio.h / 标准 I/O 库: stdio.h	953
General Utilities: stdlib.h / 通用工具: stdlib.h	956
_Noreturn: stdnoreturn.h / _Noreturn: stdnoreturn.h	962
String Handling: string.h / 处理字符串: string.h	962
Type-Generic Math: tgmath.h (C99) / 通用类型数学: tgmath.h (C99)	965
Threads: threads.h (C11) / 线程: threads.h (C11)	967
Date and Time: time.h / 日期和时间: time.h	967
Unicode Utilities: uchar.h (C11) / 统一码工具: uchar.h (C11)	971
Extended Multibyte and Wide-Character Utilities: wchar.h (C99)	
/ 扩展的多字节字符和宽字符工具: wchar.h (C99)	972
Wide Character Classification and Mapping Utilities: wctype.h (C99)	
/ 宽字符分类和映射工具: wctype.h (C99)	978
<b>Section VI: Extended Integer Types / 参考资料 VI: 扩展的整数类型</b>	<b>980</b>
Exact-Width Types / 精确宽度类型	981
Minimum-Width Types / 最小宽度类型	982
Fastest Minimum-Width Types / 最快最小宽度类型	983
Maximum-Width Types / 最大宽度类型	983
Integers That Can Hold Pointer Values / 可储存指针值的整型	984
Extended Integer Constants / 扩展的整型常量	984
<b>Section VII: Expanded Character Support / 参考资料 VII: 扩展字符支持</b>	<b>984</b>

Trigraph Sequences / 三字符序列	984
Digraphs / 双字符	985
Alternative Spellings: iso646.h / 可选拼写: iso646.h	986
Multibyte Characters / 多字节字符	986
Universal Character Names (UCNs) / 通用字符名 (UCN)	987
Wide Characters / 宽字符	988
Wide Characters and Multibyte Characters / 宽字符和多字节字符	989
<b>Section VIII: C99/C11 Numeric Computational Enhancements</b>	
<b>/ 参考资料VIII: C99/C11 数值计算增强</b>	<b>990</b>
The IEC Floating-Point Standard / IEC 浮点标准	990
The fenv.h Header File / fenv.h 头文件	994
The STDC FP_CONTRACT Pragma / STDC FP_CONTRACT 编译指示	995
Additions to the math.h Library / math.h 库增补	995
Support for Complex Numbers / 对复数的支持	996
<b>Section IX: Differences Between C and C++ / 参考资料IX: C 和 C++ 的区别</b>	<b>998</b>
Function Prototypes / 函数原型	999
char Constants / char 常量	1000
The const Modifier / const 限定符	1000
Structures and Unions / 结构和联合	1001
Enumerations / 枚举	1002
Pointer-to-void / 指向 void 的指针	1002
Boolean Types / 布尔类型	1003
Alternative Spellings / 可选拼写	1003
Wide-Character Support / 宽字符支持	1003
Complex Types / 复数类型	1003
Inline Functions / 内联函数	1003
C99/11 Features Not Found in C++11 / C++11 中没有的 C99/C11 特性	1004

# Storage Classes, Linkage, and Memory Management

You will learn about the following in this chapter:

- Keywords:  
`auto`, `extern`, `static`, `register`, `const`, `volatile`, `restricted`, `_Thread_local`,  
`_Atomic`
- Functions:  
`rand()`, `srand()`, `time()`, `malloc()`, `calloc()`, `free()`
- How C allows you to determine the scope of a variable (how widely known it is) and the lifetime of a variable (how long it remains in existence)
- Designing more complex programs

One of C's strengths is that it enables you to control a program's fine points. C's memory management system exemplifies that control by letting you determine which functions know which variables and for how long a variable persists in a program. Using memory storage is one more element of program design.

## Storage Classes

C provides several different models, or *storage classes*, for storing data in memory. To understand the options, it's helpful to go over a few concepts and terms first.

Every programming example in this book stores data in memory. There is a hardware aspect to this—each stored value occupies physical memory. C literature uses the term *object* for such a chunk of memory. An object can hold one or more values. An object might not yet actually have a stored value, but it will be of the right size to hold an appropriate value. (The phrase *object-oriented programming* uses the word *object* in a more developed sense to indicate class

objects, whose definitions encompass both data and permissible operations on the data; C is not an object-oriented programming language.)

There also is a software aspect—the program needs a way to access the object. This can be accomplished, for instance, by declaring a variable:

```
int entity = 3;
```

This declaration creates an *identifier* called `entity`. An identifier is a name, in this case one that can be used to designate the contents of a particular object. Identifiers satisfy the naming conventions for variables discussed in Chapter 2, “Introducing C.” In this case, the identifier `entity` is how the software (the C program) designates the object that’s stored in hardware memory. This declaration also provides a value to be stored in the object.

A variable name isn’t the only way to designate an object. For instance, consider the following declarations:

```
int * pt = &entity;
int ranks[10];
```

In the first case, `pt` is an identifier. It designates an object that holds an address. Next, the expression `*pt` is not an identifier because it’s not a name. However, it does designate an object, in this case the same object that `entity` designates. In general, as you may recall from Chapter 3, “Data and C,” an expression that designates an object is called an *lvalue*. So `entity` is an identifier that is an *lvalue*, and `*pt` is an expression that is an *lvalue*. Along the same lines, the expression `ranks + 2 * entity` is neither an identifier (not a name) nor an *lvalue* (doesn’t designate the contents of a memory location). But the expression `*(ranks + 2 * entity)` is an *lvalue* because it does designate the value of a particular memory location, the seventh element of the `ranks` array. The declaration of `ranks`, by the way, creates an object capable of holding ten `ints`, and each member of the array also is an object.

If, as with all these examples, you can use the *lvalue* to change the value in an object, it’s a *modifiable lvalue*. Now consider this declaration:

```
const char * pc = "Behold a string literal!";
```

This causes the program to store the string literal contents in memory, and that array of character values is an object. Each character in the array also is an object, as it can be accessed individually. The declaration also creates an object having the identifier `pc` and holding the address of that string. The identifier `pc` is a *modifiable lvalue* because it can be reset to point to a different string. The `const` prevents you from altering the contents of a pointed-to string but not from changing which string is pointed to. So `*pc`, which designates the data object holding the ‘B’ character, is an *lvalue*, but not a *modifiable lvalue*. Similarly, the string literal itself, because it designates the object holding the character string, is an *lvalue*, but not a *modifiable one*.

You can describe an object in terms of its *storage duration*, which is how long it stays in memory. You can describe an identifier used to access the object by its *scope* and its *linkage*, which together indicate which parts of a program can use it. The different storage classes offer



different combinations of scope, linkage, and storage duration. You can have identifiers that can be shared over several files of source code, identifiers that can be used by any function in one particular file, identifiers that can be used only within a particular function, and even identifiers that can be used only within a subsection of a function. You can have objects that exist for the duration of a program and objects that exist only while the function containing them is executing. With concurrent programming, you can have objects that exist for the duration of a particular thread. You also can store data in memory that is allocated and freed explicitly by means of function calls.

Next, let's investigate the meaning of the terms *scope*, *linkage*, and *storage duration*. After that, we'll return to specific storage classes.

## Scope

*Scope* describes the region or regions of a program that can access an identifier. A C variable has one of the following scopes: block scope, function scope, function prototype scope, or file scope. The program examples to date have used block scope almost exclusively for variables. A *block*, as you'll recall, is a region of code contained within an opening brace and the matching closing brace. For instance, the entire body of a function is a block. Any compound statement within a function also is a block. A variable defined inside a block has *block scope*, and it is visible from the point it is defined until the end of the block containing the definition. Also, formal function parameters, even though they occur before the opening brace of a function, have block scope and belong to the block containing the function body. So the local variables we've used to date, including formal function parameters, have block scope. Therefore, the variables *cleo* and *patrick* in the following code both have block scope extending to the closing brace:

```
double blocky(double cleo)
{
    double patrick = 0.0;
    ...
    return patrick;
}
```

Variables declared in an inner block have scope restricted just to that block:

```
double blocky(double cleo)
{
    double patrick = 0.0;
    int i;
    for (i = 0; i < 10; i++)
    {
        double q = cleo * i; // start of scope for q
        ...
        patrick += q;
    } // end of scope for q
    ...
}
```

```

    return patrick;
}

```

In this example, the scope of `q` is limited to the inner block, and only code within that block can access `q`.

Traditionally, variables with block scope had to be declared at the beginning of a block. C99 relaxed that rule, allowing you to declare variables anywhere in a block. One new possibility is in the control section of a `for` loop. That is, you now can do this:

```

for (int i = 0; i < 10; i++)
    printf("A C99 feature: i = %d", i);

```

As part of this new feature, C99 expanded the concept of a block to include the code controlled by a `for` loop, `while` loop, `do while` loop, or `if` statement, even if no brackets are used. So in the previous `for` loop, the variable `i` is considered to be part of the `for` loop block. Therefore, its scope is limited to the `for` loop. After execution leaves the `for` loop, the program will no longer see that `i`.

*Function scope* applies just to labels used with `goto` statements. This means that even if a label first appears inside an inner block in a function, its scope extends to the whole function. It would be confusing if you could use the same label inside two separate blocks, and function scope for labels prevents this from happening.

*Function prototype scope* applies to variable names used in function prototypes, as in the following:

```

int mighty(int mouse, double large);

```

Function prototype scope runs from the point the variable is defined to the end of the prototype declaration. What this means is that all the compiler cares about when handling a function prototype argument is the types; the names you use, if any, normally don't matter, and they needn't match the names you use in the function definition. One case in which the names matter a little is with variable-length array parameters:

```

void use_a_VLA(int n, int m, ar[n][m]);

```

If you use names in the brackets, they have to be names declared earlier in the prototype.

A variable with its definition placed outside of any function has *file scope*. A variable with file scope is visible from the point it is defined to the end of the file containing the definition. Take a look at this example:

```

#include <stdio.h>
int units = 0;          /* a variable with file scope */
void critic(void);
int main(void)
{
    ...
}

```