表 2.3：C++关键字

alignas	continue	friend	register	true
alignof	decltype	goto	reinterpret_cast	try
asm	default	if	return	typedef
auto	delete	inline	short	typeid
bool	do	int	signed	typename
break	double	long	sizeof	union
case	dynamic_cast	mutable	static	unsigned
catch	else	namespace	static_assert	using
char	enum	new	static_cast	virtual
char16_t	explicit	noexcept	struct	void
char32_t	export	nullptr	switch	volatile
class	extern	operator	template	wchar_t
const	false	private	this	while
constexpr	float	protected	thread_local	
const_cast	for	public	throw	

表 2.4：C++操作符替代名

and	bitand	compl	not_eq	or_eq	xor_eq
and_eq	bitor	not	or	xor	

2.2.3 节练习

练习 2.12：请指出下面的名字中哪些是非法的？

(a) `int double = 3.14;`　　(b) `int _;`

(c) `int catch-22;`　　(d) `int 1_or_2 = 1;`

(e) `double Double = 3.14;`

2.2.4 名字的作用域

不论是在程序的什么位置，使用到的每个名字都会指向一个特定的实体：变量、函数、类型等。然而，同一个名字如果出现在程序的不同位置，也可能指向的是不同实体。

48

作用域（scope）是程序的一部分，在其中名字有其特定的含义。C++语言中大多数作用域都以花括号分隔。

同一个名字在不同的作用域中可能指向不同的实体。名字的有效区域始于名字的声明语句，以声明语句所在的作用域末端为结束。

一个典型的示例来自于 1.4.2 节（第 11 页）的程序：

```
#include <iostream>
int main()
{
    int sum = 0;
    // sum 用于存放从 1 到 10 所有数的和
    for (int val = 1; val <= 10; ++val)
        sum += val; // 等价于 sum = sum + val
```

```
    std::cout << "Sum of 1 to 10 inclusive is "
              << sum << std::endl;
    return 0;
}
```

这段程序定义了3个名字：main、sum和val，同时使用了命名空间名字std，该空间提供了2个名字cout和cin供程序使用。

名字main定义于所有花括号之外，它和其他大多数定义在函数体之外的名字一样拥有**全局作用域**（global scope）。一旦声明之后，全局作用域内的名字在整个程序的范围内都可使用。名字sum定义于main函数所限定的作用域之内，从声明sum开始直到main函数结束为止都可以访问它，但是出了main函数所在的块就无法访问了，因此说变量sum拥有**块作用域**（block scope）。名字val定义于for语句内，在for语句之内可以访问val，但是在main函数的其他部分就不能访问它了。

建议：当你第一次使用变量时再定义它

一般来说，在对象第一次被使用的地方附近定义它是一种好的选择。因为这样做有助于更容易地找到变量的定义。更重要的是，当变量的定义与它第一次被使用的地方很近时，我们也会赋给它一个比较合理的初始值。

嵌套的作用域

作用域能彼此包含，被包含（或者说被嵌套）的作用域称为**内层作用域**（inner scope），包含着别的作用域的作用域称为**外层作用域**（outer scope）。

作用域中一旦声明了某个名字，它所嵌套着的所有作用域中都能访问该名字。同时，允许在内层作用域中重新定义外层作用域已有的名字：

49

```
#include <iostream>
// 该程序仅用于说明：函数内部不宜定义与全局变量同名的新变量
int reused = 42; // reused 拥有全局作用域
int main()
{
    int unique = 0; // unique 拥有块作用域
    // 输出#1：使用全局变量 reused;输出 42 0
    std::cout << reused << " " << unique << std::endl;
    int reused = 0; // 新建局部变量 reused，覆盖了全局变量 reused
    // 输出#2：使用局部变量 reused；输出 0 0
    std::cout << reused << " " << unique << std::endl;
    // 输出#3：显式地访问全局变量 reused；输出 42 0
    std::cout << ::reused << " " << unique << std::endl;
    return 0;
}
```

输出#1出现在局部变量reused定义之前，因此这条语句使用全局作用域中定义的名字reused，输出42 0。输出#2发生在局部变量reused定义之后，此时局部变量reused**正在作用域内**（in scope），因此第二条输出语句使用的是局部变量reused而非全局变量，输出0 0。输出#3使用作用域操作符（参见1.2节，第7页）来覆盖默认的作用域规则，因为全局作用域本身并没有名字，所以当作用域操作符的左侧为空时，向全局作用域发出请求获取作用域操作符右侧名字对应的变量。结果是，第三条输出语句使用全局变量reused，输出42 0。

如果函数有可能用到某全局变量，则不宜再定义一个同名的局部变量。

2.2.4 节练习

练习 2.13：下面程序中 j 的值是多少？

```
int i = 42;
int main()
{
    int i = 100;
    int j = i;
}
```

练习 2.14：下面的程序合法吗？如果合法，它将输出什么？

```
int i = 100, sum = 0;
for (int i = 0; i != 10; ++i)
    sum += i;
std::cout << i << " " << sum << std::endl;
```

2.3 复合类型

复合类型（compound type）是指基于其他类型定义的类型。C++语言有几种复合类型，本章将介绍其中的两种：引用和指针。

50

与我们已经掌握的变量声明相比，定义复合类型的变量要复杂很多。2.2 节（第 38 页）提到，一条简单的声明语句由一个数据类型和紧随其后的一个变量名列表组成。其实更通用的描述是，一条声明语句由一个**基本数据类型**（base type）和紧随其后的一个**声明符**（declarator）列表组成。每个声明符命名了一个变量并指定该变量为与基本数据类型有关的某种类型。

目前为止，我们所接触的声明语句中，声明符其实就是变量名，此时变量的类型也就是声明的基本数据类型。其实还可能有更复杂的声明符，它基于基本数据类型得到更复杂的类型，并把它指定给变量。

2.3.1 引用

C++11 中新增了一种引用：所谓的“右值引用（rvalue reference）”，我们将在 13.6.1 节（第 471 页）做更详细的介绍。这种引用主要用于内置类。严格来说，当我们使用术语“引用（reference）”时，指的其实是“左值引用（lvalue reference）”。

引用（reference）为对象起了另外一个名字，引用类型引用（refers to）另外一种类型。通过将声明符写成&d 的形式来定义引用类型，其中 d 是声明的变量名：

```
int ival = 1024;
int &refVal = ival;        // refVal 指向 ival（是 ival 的另一个名字）
int &refVal2;              // 报错：引用必须被初始化
```

一般在初始化变量时，初始值会被拷贝到新建的对象中。然而定义引用时，程序把引用和它的初始值**绑定**（bind）在一起，而不是将初始值拷贝给引用。一旦初始化完成，引用将和它的初始值对象一直绑定在一起。因为无法令引用重新绑定到另外一个对象，因此引用必须初始化。

引用即别名

引用并非对象，相反的，它只是为一个已经存在的对象所起的另外一个名字。

定义了一个引用之后，对其进行的所有操作都是在与之绑定的对象上进行的：

```
refVal = 2;        // 把 2 赋给 refVal 指向的对象，此处即是赋给了 ival
int ii = refVal; // 与 ii = ival 执行结果一样
```

51 为引用赋值，实际上是把值赋给了与引用绑定的对象。获取引用的值，实际上是获取了与引用绑定的对象的值。同理，以引用作为初始值，实际上是以与引用绑定的对象作为初始值：

```
// 正确：refVal3 绑定到了那个与 refVal 绑定的对象上，这里就是绑定到 ival 上
int &refVal3 = refVal;
// 利用与 refVal 绑定的对象的值初始化变量 i
int i = refVal; // 正确：i 被初始化为 ival 的值
```

因为引用本身不是一个对象，所以不能定义引用的引用。

引用的定义

允许在一条语句中定义多个引用，其中每个引用标识符都必须以符号&开头：

```
int i = 1024, i2 = 2048; // i 和 i2 都是 int
int &r = i, r2 = i2;     // r 是一个引用，与 i 绑定在一起，r2 是 int
int i3 = 1024, &ri = i3; // i3 是 int，ri 是一个引用，与 i3 绑定在一起
int &r3 = i3, &r4 = i2;  // r3 和 r4 都是引用
```

除了 2.4.1 节（第 55 页）和 15.2.3 节（第 534 页）将要介绍的两种例外情况，其他所有引用的类型都要和与之绑定的对象严格匹配。而且，引用只能绑定在对象上，而不能与字面值或某个表达式的计算结果绑定在一起，相关原因将在 2.4.1 节详述：

```
int &refVal4 = 10;         // 错误：引用类型的初始值必须是一个对象
double dval = 3.14;
int &refVal5 = dval;       // 错误：此处引用类型的初始值必须是 int 型对象
```

2.3.1 节练习

练习 2.15：下面的哪个定义是不合法的？为什么？

(a) `int ival = 1.01;`　　(b) `int &rval1 = 1.01;`
(c) `int &rval2 = ival;`　　(d) `int &rval3;`

练习 2.16：考查下面的所有赋值然后回答：哪些赋值是不合法的？为什么？哪些赋值是合法的？它们执行了什么样的操作？

```
int i = 0, &r1 = i;        double d = 0, &r2 = d;
```

(a) `r2 = 3.14159;`　　(b) `r2 = r1;`
(c) `i = r2;`　　(d) `r1 = d;`

练习 2.17：执行下面的代码段将输出什么结果？

```
int i, &ri = i;
i = 5; ri = 10;
std::cout << i << " " << ri << std::endl;
```

2.3.2 指针

指针（pointer）是“指向（point to）”另外一种类型的复合类型。与引用类似，指针 52 也实现了对其他对象的间接访问。然而指针与引用相比又有很多不同点。其一，指针本身就是一个对象，允许对指针赋值和拷贝，而且在指针的生命周期内它可以先后指向几个不同的对象。其二，指针无须在定义时赋初值。和其他内置类型一样，在块作用域内定义的指针如果没有被初始化，也将拥有一个不确定的值。

指针通常难以理解，即使是有经验的程序员也常常因为调试指针引发的错误而被备受折磨。

定义指针类型的方法将声明符写成*d 的形式，其中 d 是变量名。如果在一条语句中定义了几个指针变量，每个变量前面都必须有符号*：

```
int *ip1, *ip2;   // ip1 和 ip2 都是指向 int 型对象的指针
double dp, *dp2;  // dp2 是指向 double 型对象的指针，dp 是 double 型对象
```

获取对象的地址

指针存放某个对象的地址，要想获取该地址，需要使用**取地址符**（操作符&）：

```
int ival = 42;
int *p = &ival;  // p 存放变量 ival 的地址，或者说 p 是指向变量 ival 的指针
```

第二条语句把 p 定义为一个指向 int 的指针，随后初始化 p 令其指向名为 ival 的 int 对象。因为引用不是对象，没有实际地址，所以不能定义指向引用的指针。

除了 2.4.2 节（第 56 页）和 15.2.3 节（第 534 页）将要介绍的两种例外情况，其他所有指针的类型都要和它所指向的对象严格匹配：

```
double dval;
double *pd = &dval;   // 正确：初始值是 double 型对象的地址
double *pd2 = pd;     // 正确：初始值是指向 double 对象的指针

int *pi = pd;         // 错误：指针 pi 的类型和 pd 的类型不匹配
pi = &dval;           // 错误：试图把 double 型对象的地址赋给 int 型指针
```

因为在声明语句中指针的类型实际上被用于指定它所指向对象的类型，所以二者必须匹配。如果指针指向了一个其他类型的对象，对该对象的操作将发生错误。

指针值

指针的值（即地址）应属下列 4 种状态之一：

1. 指向一个对象。
2. 指向紧邻对象所占空间的下一个位置。
3. 空指针，意味着指针没有指向任何对象。
4. 无效指针，也就是上述情况之外的其他值。

试图拷贝或以其他方式访问无效指针的值都将引发错误。编译器并不负责检查此类错误，
53 这一点和试图使用未经初始化的变量是一样的。访问无效指针的后果无法预计，因此程序员必须清楚任意给定的指针是否有效。

尽管第2种和第3种形式的指针是有效的，但其使用同样受到限制。显然这些指针没有指向任何具体对象，所以试图访问此类指针（假定的）对象的行为不被允许。如果这样做了，后果也无法预计。

利用指针访问对象

如果指针指向了一个对象，则允许使用**解引用符**（操作符*）来访问该对象：

```
int ival = 42;
int *p = &ival;  // p存放着变量ival的地址，或者说p是指向变量ival的指针
cout << *p;      // 由符号*得到指针p所指的对象，输出42
```

对指针解引用会得出所指的对象，因此如果给解引用的结果赋值，实际上也就是给指针所指的对象赋值：

```
*p = 0;      // 由符号*得到指针p所指的对象，即可经由p为变量ival赋值
cout << *p;  // 输出0
```

如上述程序所示，为*p赋值实际上是为p所指的对象赋值。

解引用操作仅适用于那些确实指向了某个对象的有效指针。

关键概念：某些符号有多重含义

像&和*这样的符号，既能用作表达式里的运算符，也能作为声明的一部分出现，符号的上下文决定了符号的意义：

```
int i = 42;
int &r = i;      // &紧随类型名出现，因此是声明的一部分，r是一个引用
int *p;          // *紧随类型名出现，因此是声明的一部分，p是一个指针
p = &i;          // &出现在表达式中，是一个取地址符
*p = i;          // *出现在表达式中，是一个解引用符
int &r2 = *p;    // &是声明的一部分，*是一个解引用符
```

在声明语句中，&和*用于组成复合类型；在表达式中，它们的角色又转变成运算符。在不同场景下出现的虽然是同一个符号，但是由于含义截然不同，所以我们完全可以把它当作不同的符号来看待。

空指针

空指针（null pointer）不指向任何对象，在试图使用一个指针之前代码可以首先检查它是否为空。以下列出几个生成空指针的方法：

54
```
int *p1 = nullptr;       // 等价于int *p1 = 0;
int *p2 = 0;             // 直接将p2初始化为字面常量0
// 需要首先#include cstdlib
int *p3 = NULL;          // 等价于int *p3 = 0;
```

C++11 得到空指针最直接的办法就是用字面值 **nullptr** 来初始化指针，这也是C++11新标准刚刚引入的一种方法。nullptr是一种特殊类型的字面值，它可以被转换成（参见2.1.2节，

第 32 页）任意其他的指针类型。另一种办法就如对 p2 的定义一样，也可以通过将指针初始化为字面值 0 来生成空指针。

过去的程序还会用到一个名为 NULL 的**预处理变量**（preprocessor variable）来给指针赋值，这个变量在头文件 cstdlib 中定义，它的值就是 0。

2.6.3 节（第 68 页）将稍微介绍一点关于预处理器的知识，现在只要知道预处理器是运行于编译过程之前的一段程序就可以了。预处理变量不属于命名空间 std，它由预处理器负责管理，因此我们可以直接使用预处理变量而无须在前面加上 std::。

当用到一个预处理变量时，预处理器会自动地将它替换为实际值，因此用 NULL 初始化指针和用 0 初始化指针是一样的。在新标准下，现在的 C++程序最好使用 nullptr，同时尽量避免使用 NULL。

把 int 变量直接赋给指针是错误的操作，即使 int 变量的值恰好等于 0 也不行。

```
int zero = 0;
pi = zero;          // 错误：不能把 int 变量直接赋给指针
```

建议：初始化所有指针

使用未经初始化的指针是引发运行时错误的一大原因。

和其他变量一样，访问未经初始化的指针所引发的后果也是无法预计的。通常这一行为将造成程序崩溃，而且一旦崩溃，要想定位到出错位置将是特别棘手的问题。

在大多数编译器环境下，如果使用了未经初始化的指针，则该指针所占内存空间的当前内容将被看作一个地址值。访问该指针，相当于去访问一个本不存在的位置上的本不存在的对象。糟糕的是，如果指针所占内存空间中恰好有内容，而这些内容又被当作了某个地址，我们就很难分清它到底是合法的还是非法的了。

因此建议初始化所有的指针，并且在可能的情况下，尽量等定义了对象之后再定义指向它的指针。如果实在不清楚指针应该指向何处，就把它初始化为 nullptr 或者 0，这样程序就能检测并知道它没有指向任何具体的对象了。

赋值和指针

指针和引用都能提供对其他对象的间接访问，然而在具体实现细节上二者有很大不同，其中最重要的一点就是引用本身并非一个对象。一旦定义了引用，就无法令其再绑定到另外的对象，之后每次使用这个引用都是访问它最初绑定的那个对象。 55

指针和它存放的地址之间就没有这种限制了。和其他任何变量（只要不是引用）一样，给指针赋值就是令它存放一个新的地址，从而指向一个新的对象：

```
int i = 42;
int *pi = 0;          // pi 被初始化，但没有指向任何对象
int *pi2 = &i;        // pi2 被初始化，存有 i 的地址
int *pi3;             // 如果 pi3 定义于块内，则 pi3 的值是无法确定的

pi3 = pi2;            // pi3 和 pi2 指向同一个对象 i
pi2 = 0;              // 现在 pi2 不指向任何对象了
```

有时候要想搞清楚一条赋值语句到底是改变了指针的值还是改变了指针所指对象的值不太容易，最好的办法就是记住赋值永远改变的是等号左侧的对象。当写出如下语句时，

```
pi = &ival;           // pi 的值被改变，现在 pi 指向了 ival
```

意思是为 pi 赋一个新的值，也就是改变了那个存放在 pi 内的地址值。相反的，如果写出如下语句，

```
*pi = 0;              // ival 的值被改变，指针 pi 并没有改变
```

则*pi（也就是指针 pi 指向的那个对象）发生改变。

其他指针操作

只要指针拥有一个合法值，就能将它用在条件表达式中。和采用算术值作为条件（参见 2.1.2 节，第 32 页）遵循的规则类似，如果指针的值是 0，条件取 false：

```
int ival = 1024;
int *pi = 0;                  // pi 合法，是一个空指针
int *pi2 = &ival;             // pi2 是一个合法的指针，存放着 ival 的地址
if (pi)                       // pi 的值是 0，因此条件的值是 false
    // ...
if (pi2)                      // pi2 指向 ival，因此它的值不是 0，条件的值是 true
    // ...
```

任何非 0 指针对应的条件值都是 true。

对于两个类型相同的合法指针，可以用相等操作符（==）或不相等操作符（!=）来比较它们，比较的结果是布尔类型。如果两个指针存放的地址值相同，则它们相等；反之它们不相等。这里两个指针存放的地址值相同（两个指针相等）有三种可能：它们都为空、都指向同一个对象，或者都指向了同一个对象的下一地址。需要注意的是，一个指针指向某对象，同时另一个指针指向另外对象的下一地址，此时也有可能出现这两个指针值相同的情况，即指针相等。

因为上述操作要用到指针的值，所以不论是作为条件出现还是参与比较运算，都必须使用合法指针，使用非法指针作为条件或进行比较都会引发不可预计的后果。

3.5.3 节（第 105 页）将介绍更多关于指针的操作。

56 void* 指针

void*是一种特殊的指针类型，可用于存放任意对象的地址。一个 void*指针存放着一个地址，这一点和其他指针类似。不同的是，我们对该地址中到底是个什么类型的对象并不了解：

```
double obj = 3.14, *pd = &obj;
                              // 正确：void*能存放任意类型对象的地址
void *pv = &obj;              // obj 可以是任意类型的对象
pv = pd;                      // pv 可以存放任意类型的指针
```

利用 void*指针能做的事儿比较有限：拿它和别的指针比较、作为函数的输入或输出，或者赋给另外一个 void*指针。不能直接操作 void*指针所指的对象，因为我们并不知道这个对象到底是什么类型，也就无法确定能在这个对象上做哪些操作。

概括说来，以 void*的视角来看内存空间也就仅仅是内存空间，没办法访问内存空间中所存的对象，关于这点将在 19.1.1 节（第 726 页）有更详细的介绍，4.11.3 节（第 144 页）将讲述获取 void*指针所存地址的方法。

2.3.2 节练习

练习 2.18：编写代码分别更改指针的值以及指针所指对象的值。

练习 2.19：说明指针和引用的主要区别。

练习 2.20：请叙述下面这段代码的作用。

```
int i = 42;
int *p1 = &i;
*p1 = *p1 * *p1;
```

练习 2.21：请解释下述定义。在这些定义中有非法的吗？如果有，为什么？

```
int i = 0;
(a) double* dp = &i; (b) int *ip = i; (c) int *p = &i;
```

练习 2.22：假设 p 是一个 int 型指针，请说明下述代码的含义。

```
if (p) // ...
if (*p) // ...
```

练习 2.23：给定指针 p，你能知道它是否指向了一个合法的对象吗？如果能，叙述判断的思路；如果不能，也请说明原因。

练习 2.24：在下面这段代码中为什么 p 合法而 lp 非法？

```
int i = 42;      void *p = &i;        long *lp = &i;
```

2.3.3 理解复合类型的声明

如前所述，变量的定义包括一个基本数据类型（base type）和一组声明符。在同一条 57 定义语句中，虽然基本数据类型只有一个，但是声明符的形式却可以不同。也就是说，一条定义语句可能定义出不同类型的变量：

```
// i 是一个 int 型的数，p 是一个 int 型指针，r 是一个 int 型引用
int i = 1024, *p = &i, &r = i;
```

很多程序员容易迷惑于基本数据类型和类型修饰符的关系，其实后者不过是声明符的一部分罢了。

定义多个变量

经常有一种观点会误以为，在定义语句中，类型修饰符（*或&）作用于本次定义的全部变量。造成这种错误看法的原因有很多，其中之一是我们可以把空格写在类型修饰符和变量名中间：

```
int* p;          // 合法但是容易产生误导
```

我们说这种写法可能产生误导是因为 int*放在一起好像是这条语句中所有变量共同的类型一样。其实恰恰相反，基本数据类型是 int 而非 int*。*仅仅是修饰了 p 而已，对该声明语句中的其他变量，它并不产生任何作用：

```
int* p1, p2;     // p1 是指向 int 的指针，p2 是 int
```

涉及指针或引用的声明，一般有两种写法。第一种把修饰符和变量标识符写在一起：

```
int *p1, *p2; // p1 和 p2 都是指向 int 的指针
```

这种形式着重强调变量具有的复合类型。第二种把修饰符和类型名写在一起，并且每条语句只定义一个变量：

```
int* p1;          // p1 是指向 int 的指针
int* p2;          // p2 是指向 int 的指针
```

这种形式着重强调本次声明定义了一种复合类型。

上述两种定义指针或引用的不同方法没有孰对孰错之分，关键是选择并坚持其中的一种写法，不要总是变来变去。

本书采用第一种写法，将*（或是&）与变量名连在一起。

指向指针的指针

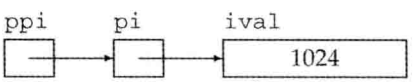
一般来说，声明符中修饰符的个数并没有限制。当有多个修饰符连写在一起时，按照其逻辑关系详加解释即可。以指针为例，指针是内存中的对象，像其他对象一样也有自己的地址，因此允许把指针的地址再存放到另一个指针当中。

58

通过*的个数可以区分指针的级别。也就是说，**表示指向指针的指针，***表示指向指针的指针的指针，以此类推：

```
int ival = 1024;
int *pi = &ival; // pi 指向一个 int 型的数
int **ppi = &pi; // ppi 指向一个 int 型的指针
```

此处 pi 是指向 int 型数的指针，而 ppi 是指向 int 型指针的指针，下图描述了它们之间的关系。



解引用 int 型指针会得到一个 int 型的数，同样，解引用指向指针的指针会得到一个指针。此时为了访问最原始的那个对象，需要对指针的指针做两次解引用：

```
cout << "The value of ival\n"
     << "direct value: " << ival << "\n"
     << "indirect value: " << *pi << "\n"
     << "doubly indirect value: " << **ppi
     << endl;
```

该程序使用三种不同的方式输出了变量 ival 的值：第一种直接输出；第二种通过 int 型指针 pi 输出；第三种两次解引用 ppi，取得 ival 的值。

指向指针的引用

引用本身不是一个对象，因此不能定义指向引用的指针。但指针是对象，所以存在对指针的引用：

```
int i = 42;
int *p;           // p 是一个 int 型指针
int *&r = p;      // r 是一个对指针 p 的引用

r = &i;           // r 引用了一个指针，因此给 r 赋值&i 就是令 p 指向 i
*r = 0;           // 解引用 r 得到 i，也就是 p 指向的对象，将 i 的值改为 0
```

要理解 r 的类型到底是什么，最简单的办法是从右向左阅读 r 的定义。离变量名最近的符号（此例中是&r 的符号&）对变量的类型有最直接的影响，因此 r 是一个引用。声明符的其余部分用以确定 r 引用的类型是什么，此例中的符号*说明 r 引用的是一个指针。最后，声明的基本数据类型部分指出 r 引用的是一个 int 指针。

面对一条比较复杂的指针或引用的声明语句时，从右向左阅读有助于弄清楚它的真实含义。

2.3.3 节练习

59

练习 2.25：说明下列变量的类型和值。

```
(a) int* ip,i, &r = i; (b) int i, *ip = 0; (c) int* ip, ip2;
```

2.4 const 限定符

有时我们希望定义这样一种变量，它的值不能被改变。例如，用一个变量来表示缓冲区的大小。使用变量的好处是当我们觉得缓冲区大小不再合适时，很容易对其进行调整。另一方面，也应随时警惕防止程序一不小心改变了这个值。为了满足这一要求，可以用关键字 **const** 对变量的类型加以限定：

```
const int bufSize = 512;        // 输入缓冲区大小
```

这样就把 bufSize 定义成了一个常量。任何试图为 bufSize 赋值的行为都将引发错误：

```
bufSize = 512;                  // 错误：试图向 const 对象写值
```

因为 const 对象一旦创建后其值就不能再改变，所以 const 对象必须初始化。一如既往，初始值可以是任意复杂的表达式：

```
const int i = get_size();       // 正确：运行时初始化
const int j = 42;               // 正确：编译时初始化
const int k;                    // 错误：k 是一个未经初始化的常量
```

初始化和 const

正如之前反复提到的，对象的类型决定了其上的操作。与非 const 类型所能参与的操作相比，const 类型的对象能完成其中大部分，但也不是所有的操作都适合。主要的限制就是只能在 const 类型的对象上执行不改变其内容的操作。例如，const int 和普通的 int 一样都能参与算术运算，也都能转换成一个布尔值，等等。

在不改变 const 对象的操作中还有一种是初始化，如果利用一个对象去初始化另外一个对象，则它们是不是 const 都无关紧要：

```
int i = 42;
const int ci = i;               // 正确：i 的值被拷贝给了 ci
int j = ci;                     // 正确：ci 的值被拷贝给了 j
```

尽管 ci 是整型常量，但无论如何 ci 中的值还是一个整型数。ci 的常量特征仅仅在执行改变 ci 的操作时才会发挥作用。当用 ci 去初始化 j 时，根本无须在意 ci 是不是一个常量。拷贝一个对象的值并不会改变它，一旦拷贝完成，新的对象就和原来的对象没什么关系了。

60 默认状态下，const 对象仅在文件内有效

当以编译时初始化的方式定义一个 const 对象时，就如对 bufSize 的定义一样：

```
const int bufSize = 512; // 输入缓冲区大小
```

编译器将在编译过程中把用到该变量的地方都替换成对应的值。也就是说，编译器会找到代码中所有用到 bufSize 的地方，然后用 512 替换。

为了执行上述替换，编译器必须知道变量的初始值。如果程序包含多个文件，则每个用了 const 对象的文件都必须得能访问到它的初始值才行。要做到这一点，就必须在每一个用到变量的文件中都有对它的定义（参见 2.2.2 节，第 41 页）。为了支持这一用法，同时避免对同一变量的重复定义，默认情况下，const 对象被设定为仅在文件内有效。当多个文件中出现了同名的 const 变量时，其实等同于在不同文件中分别定义了独立的变量。

某些时候有这样一种 const 变量，它的初始值不是一个常量表达式，但又确实有必要在文件间共享。这种情况下，我们不希望编译器为每个文件分别生成独立的变量。相反，我们想让这类 const 对象像其他（非常量）对象一样工作，也就是说，只在一个文件中定义 const，而在其他多个文件中声明并使用它。

解决的办法是，对于 const 变量不管是声明还是定义都添加 extern 关键字，这样只需定义一次就可以了：

```
// file_1.cc 定义并初始化了一个常量，该常量能被其他文件访问
extern const int bufSize = fcn();
// file_1.h 头文件
extern const int bufSize; // 与 file_1.cc 中定义的 bufSize 是同一个
```

如上述程序所示，file_1.cc 定义并初始化了 bufSize。因为这条语句包含了初始值，所以它（显然）是一次定义。然而，因为 bufSize 是一个常量，必须用 extern 加以限定使其被其他文件使用。

file_1.h 头文件中的声明也由 extern 做了限定，其作用是指明 bufSize 并非本文件所独有，它的定义将在别处出现。

如果想在多个文件之间共享 const 对象，必须在变量的定义之前添加 extern 关键字。

2.4 节练习

练习 2.26：下面哪些句子是合法的？如果有不合法的句子，请说明为什么？

```
(a) const int buf;          (b) int cnt = 0;
(c) const int sz = cnt;     (d) ++cnt; ++sz;
```

2.4.1 const 的引用

61 可以把引用绑定到 const 对象上，就像绑定到其他对象上一样，我们称之为**对常量的引用**（reference to const）。与普通引用不同的是，对常量的引用不能被用作修改它所绑定的对象：

```
const int ci = 1024;
const int &r1 = ci;  // 正确：引用及其对应的对象都是常量
```

```
r1 = 42;                    // 错误：r1 是对常量的引用
int &r2 = ci;               // 错误：试图让一个非常量引用指向一个常量对象
```

因为不允许直接为 ci 赋值，当然也就不能通过引用去改变 ci。因此，对 r2 的初始化是错误的。假设该初始化合法，则可以通过 r2 来改变它引用对象的值，这显然是不正确的。

术语：常量引用是对 const 的引用

C++程序员们经常把词组“对 const 的引用”简称为“常量引用”，这一简称还是挺靠谱的，不过前提是你得时刻记得这就是个简称而已。

严格来说，并不存在常量引用。因为引用不是一个对象，所以我们没法让引用本身恒定不变。事实上，由于 C++语言并不允许随意改变引用所绑定的对象，所以从这层意义上理解所有的引用又都算是常量。引用的对象是常量还是非常量可以决定其所能参与的操作，却无论如何都不会影响到引用和对象的绑定关系本身。

初始化和对 const 的引用

2.3.1 节（第 46 页）提到，引用的类型必须与其所引用对象的类型一致，但是有两个例外。第一种例外情况就是在初始化常量引用时允许用任意表达式作为初始值，只要该表达式的结果能转换成（参见 2.1.2 节，第 32 页）引用的类型即可。尤其，允许为一个常量引用绑定非常量的对象、字面值，甚至是个一般表达式：

```
int i = 42;
const int &r1 = i;          // 允许将 const int&绑定到一个普通 int 对象上
const int &r2 = 42;         // 正确：r1 是一个常量引用
const int &r3 = r1 * 2;     // 正确：r3 是一个常量引用
int &r4 = r1 * 2;           // 错误：r4 是一个普通的非常量引用
```

要想理解这种例外情况的原因，最简单的办法是弄清楚当一个常量引用被绑定到另外一种类型上时到底发生了什么：

```
double dval = 3.14;
const int &ri = dval;
```

此处 ri 引用了一个 int 型的数。对 ri 的操作应该是整数运算，但 dval 却是一个双精度浮点数而非整数。因此为了确保让 ri 绑定一个整数，编译器把上述代码变成了如下形式：

```
const int temp = dval;      // 由双精度浮点数生成一个临时的整型常量
const int &ri = temp;       // 让 ri 绑定这个临时量
```

62

在这种情况下，ri 绑定了一个**临时量**（temporary）对象。所谓临时量对象就是当编译器需要一个空间来暂存表达式的求值结果时临时创建的一个未命名的对象。C++程序员们常常把临时量对象简称为临时量。

接下来探讨当 ri 不是常量时，如果执行了类似于上面的初始化过程将带来什么样的后果。如果 ri 不是常量，就允许对 ri 赋值，这样就会改变 ri 所引用对象的值。注意，此时绑定的对象是一个临时量而非 dval。程序员既然让 ri 引用 dval，就肯定想通过 ri 改变 dval 的值，否则干什么要给 ri 赋值呢？如此看来，既然大家基本上不会想着把引用绑定到临时量上，C++语言也就把这种行为归为非法。

对 const 的引用可能引用一个并非 const 的对象

必须认识到，常量引用仅对引用可参与的操作做出了限定，对于引用的对象本身是不是一个常量未作限定。因为对象也可能是个非常量，所以允许通过其他途径改变它的值：

```
int i = 42;
int &r1 = i;                    // 引用 ri 绑定对象 i
const int &r2 = i;              // r2 也绑定对象 i，但是不允许通过 r2 修改 i 的值
r1 = 0;                         // r1 并非常量，i 的值修改为 0
r2 = 0;                         // 错误：r2 是一个常量引用
```

r2 绑定（非常量）整数 i 是合法的行为。然而，不允许通过 r2 修改 i 的值。尽管如此，i 的值仍然允许通过其他途径修改，既可以直接给 i 赋值，也可以通过像 r1 一样绑定到 i 的其他引用来修改。

2.4.2 指针和 const

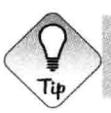
与引用一样，也可以令指针指向常量或非常量。类似于常量引用（参见 2.4.1 节，第 54 页），**指向常量的指针**（pointer to const）不能用于改变其所指对象的值。要想存放常量对象的地址，只能使用指向常量的指针：

```
const double pi = 3.14;         // pi 是个常量，它的值不能改变
double *ptr = &pi;              // 错误：ptr 是一个普通指针
const double *cptr = &pi;       // 正确：cptr 可以指向一个双精度常量
*cptr = 42;                     // 错误：不能给*cptr 赋值
```

2.3.2 节（第 47 页）提到，指针的类型必须与其所指对象的类型一致，但是有两个例外。第一种例外情况是允许令一个指向常量的指针指向一个非常量对象：

```
double dval = 3.14;             // dval 是一个双精度浮点数，它的值可以改变
cptr = &dval;                   // 正确：但是不能通过 cptr 改变 dval 的值
```

63 和常量引用一样，指向常量的指针也没有规定其所指的对象必须是一个常量。所谓指向常量的指针仅仅要求不能通过该指针改变对象的值，而没有规定那个对象的值不能通过其他途径改变。



试试这样想吧：所谓指向常量的指针或引用，不过是指针或引用“自以为是”罢了，它们觉得自己指向了常量，所以自觉地不去改变所指对象的值。

const 指针

指针是对象而引用不是，因此就像其他对象类型一样，允许把指针本身定为常量。**常量指针**（const pointer）必须初始化，而且一旦初始化完成，则它的值（也就是存放在指针中的那个地址）就不能再改变了。把*放在 const 关键字之前用以说明指针是一个常量，这样的书写形式隐含着一层意味，即不变的是指针本身的值而非指向的那个值：

```
int errNumb = 0;
int *const curErr = &errNumb;     // curErr 将一直指向 errNumb
const double pi = 3.14159;
const double *const pip = &pi;    // pip 是一个指向常量对象的常量指针
```

如同 2.3.3 节（第 52 页）所讲的，要想弄清楚这些声明的含义最行之有效的办法是从右向左阅读。此例中，离 curErr 最近的符号是 const，意味着 curErr 本身是一个常量对象，对象的类型由声明符的其余部分确定。声明符中的下一个符号是*，意思是 curErr

是一个常量指针。最后，该声明语句的基本数据类型部分确定了常量指针指向的是一个 int 对象。与之相似，我们也能推断出，pip 是一个常量指针，它指向的对象是一个双精度浮点型常量。

指针本身是一个常量并不意味着不能通过指针修改其所指对象的值，能否这样做完全依赖于所指对象的类型。例如，pip 是一个指向常量的常量指针，则不论是 pip 所指的对象值还是 pip 自己存储的那个地址都不能改变。相反的，curErr 指向的是一个一般的非常量整数，那么就完全可以用 curErr 去修改 errNumb 的值：

```
*pip = 2.72;          // 错误：pip 是一个指向常量的指针
                      // 如果 curErr 所指的对象（也就是 errNumb）的值不为 0
if (*curErr) {
    errorHandler();
    *curErr = 0;      // 正确：把 curErr 所指的对象的值重置
}
```

2.4.2 节练习

练习 2.27：下面的哪些初始化是合法的？请说明原因。

(a) `int i = -1, &r = 0;`　　(b) `int *const p2 = &i2;`
(c) `const int i = -1, &r = 0;`　　(d) `const int *const p3 = &i2;`
(e) `const int *p1 = &i2;`　　(f) `const int &const r2;`
(g) `const int i2 = i, &r = i;`

练习 2.28：说明下面的这些定义是什么意思，挑出其中不合法的。

(a) `int i, *const cp;`　　(b) `int *p1, *const p2;`
(c) `const int ic, &r = ic;`　　(d) `const int *const p3;`
(e) `const int *p;`

练习 2.29：假设已有上一个练习中定义的那些变量，则下面的哪些语句是合法的？请说明原因。

(a) `i = ic;`　　(b) `p1 = p3;`
(c) `p1 = &ic;`　　(d) `p3 = &ic;`
(e) `p2 = p1;`　　(f) `ic = *p3;`

2.4.3 顶层 const

如前所述，指针本身是一个对象，它又可以指向另外一个对象。因此，指针本身是不是常量以及指针所指的是不是一个常量就是两个相互独立的问题。用名词**顶层 const**（top-level const）表示指针本身是个常量，而用名词**底层 const**（low-level const）表示指针所指的对象是一个常量。

64

更一般的，顶层 const 可以表示任意的对象是常量，这一点对任何数据类型都适用，如算术类型、类、指针等。底层 const 则与指针和引用等复合类型的基本类型部分有关。比较特殊的是，指针类型既可以是顶层 const 也可以是底层 const，这一点和其他类型相比区别明显：

```
int i = 0;
int *const p1 = &i;          // 不能改变 p1 的值，这是一个顶层 const
const int ci = 42;           // 不能改变 ci 的值，这是一个顶层 const
const int *p2 = &ci;         // 允许改变 p2 的值，这是一个底层 const
```

```
const int *const p3 = p2; // 靠右的 const 是顶层 const，靠左的是底层 const
const int &r = ci;        // 用于声明引用的 const 都是底层 const
```

当执行对象的拷贝操作时，常量是顶层 const 还是底层 const 区别明显。其中，顶层 const 不受什么影响：

```
i = ci;      // 正确：拷贝 ci 的值，ci 是一个顶层 const，对此操作无影响
p2 = p3;     // 正确：p2 和 p3 指向的对象类型相同，p3 顶层 const 的部分不影响
```

执行拷贝操作并不会改变被拷贝对象的值，因此，拷入和拷出的对象是否是常量都没什么影响。

另一方面，底层 const 的限制却不能忽视。当执行对象的拷贝操作时，拷入和拷出的对象必须具有相同的底层 const 资格，或者两个对象的数据类型必须能够转换。一般来说，非常量可以转换成常量，反之则不行：

65

```
int *p = p3;              // 错误：p3 包含底层 const 的定义，而 p 没有
p2 = p3;                  // 正确：p2 和 p3 都是底层 const
p2 = &i;                  // 正确：int*能转换成 const int*
int &r = ci;              // 错误：普通的 int&不能绑定到 int 常量上
const int &r2 = i;        // 正确：const int&可以绑定到一个普通 int 上
```

p3 既是顶层 const 也是底层 const，拷贝 p3 时可以不在乎它是一个顶层 const，但是必须清楚它指向的对象得是一个常量。因此，不能用 p3 去初始化 p，因为 p 指向的是一个普通的（非常量）整数。另一方面，p3 的值可以赋给 p2，是因为这两个指针都是底层 const，尽管 p3 同时也是一个常量指针（顶层 const），仅就这次赋值而言不会有什么影响。

2.4.3 节练习

练习 2.30：对于下面的这些语句，请说明对象被声明成了顶层 const 还是底层 const？

```
const int v2 = 0;      int v1 = v2;
int *p1 = &v1, &r1 = v1;
const int *p2 = &v2, *const p3 = &i, &r2 = v2;
```

练习 2.31：假设已有上一个练习中所做的那些声明，则下面的哪些语句是合法的？请说明顶层 const 和底层 const 在每个例子中有何体现。

```
r1 = v2;
p1 = p2; p2 = p1;
p1 = p3; p2 = p3;
```

2.4.4 constexpr 和常量表达式

常量表达式（const expression）是指值不会改变并且在编译过程就能得到计算结果的表达式。显然，字面值属于常量表达式，用常量表达式初始化的 const 对象也是常量表达式。后面将会提到，C++语言中有几种情况下是要用到常量表达式的。

一个对象（或表达式）是不是常量表达式由它的数据类型和初始值共同决定，例如：

```
const int max_files = 20;          // max_files 是常量表达式
const int limit = max_files + 1;   // limit 是常量表达式
int staff_size = 27;               // staff_size 不是常量表达式
```

```
const int sz = get_size();          // sz 不是常量表达式
```

尽管 staff_size 的初始值是个字面值常量，但由于它的数据类型只是一个普通 int 而非 const int，所以它不属于常量表达式。另一方面，尽管 sz 本身是一个常量，但它的具体值直到运行时才能获取到，所以也不是常量表达式。

constexpr 变量

在一个复杂系统中，很难（几乎肯定不能）分辨一个初始值到底是不是常量表达式。66 当然可以定义一个 const 变量并把它的初始值设为我们认为的某个常量表达式，但在实际使用时，尽管要求如此却常常发现初始值并非常量表达式的情况。可以这么说，在此种情况下，对象的定义和使用根本就是两回事儿。

C++11 新标准规定，允许将变量声明为 **constexpr** 类型以便由编译器来验证变量的值是否是一个常量表达式。C++ 11 声明为 constexpr 的变量一定是一个常量，而且必须用常量表达式初始化：

```
constexpr int mf = 20;              // 20 是常量表达式
constexpr int limit = mf + 1;       // mf + 1 是常量表达式
constexpr int sz = size();          // 只有当 size 是一个 constexpr 函数时
                                    // 才是一条正确的声明语句
```

尽管不能使用普通函数作为 constexpr 变量的初始值，但是正如 6.5.2 节（第 214 页）将要介绍的，新标准允许定义一种特殊的 constexpr 函数。这种函数应该足够简单以使得编译时就可以计算其结果，这样就能用 constexpr 函数去初始化 constexpr 变量了。

一般来说，如果你认定变量是一个常量表达式，那就把它声明成 constexpr 类型。

字面值类型

常量表达式的值需要在编译时就得到计算，因此对声明 constexpr 时用到的类型必须有所限制。因为这些类型一般比较简单，值也显而易见、容易得到，就把它们称为“字面值类型”（literal type）。

到目前为止接触过的数据类型中，算术类型、引用和指针都属于字面值类型。自定义类 Sales_item、IO 库、string 类型则不属于字面值类型，也就不能被定义成 constexpr。其他一些字面值类型将在 7.5.6 节（第 267 页）和 19.3 节（第 736 页）介绍。

尽管指针和引用都能定义成 constexpr，但它们的初始值却受到严格限制。一个 constexpr 指针的初始值必须是 nullptr 或者 0，或者是存储于某个固定地址中的对象。

6.1.1 节（第 184 页）将要提到，函数体内定义的变量一般来说并非存放在固定地址中，因此 constexpr 指针不能指向这样的变量。相反的，定义于所有函数体之外的对象其地址固定不变，能用来初始化 constexpr 指针。同样是在 6.1.1 节（第 185 页）中还将提到，允许函数定义一类有效范围超出函数本身的变量，这类变量和定义在函数体之外的变量一样也有固定地址。因此，constexpr 引用能绑定到这样的变量上，constexpr 指针也能指向这样的变量。

指针和 constexpr

67

必须明确一点，在 constexpr 声明中如果定义了一个指针，限定符 constexpr 仅

对指针有效，与指针所指的对象无关：

```
const int *p = nullptr;        // p是一个指向整型常量的指针
constexpr int *q = nullptr;    // q是一个指向整数的常量指针
```

p和q的类型相差甚远，p是一个指向常量的指针，而q是一个常量指针，其中的关键在于constexpr把它所定义的对象置为了顶层const（参见2.4.3节，第57页）。

与其他常量指针类似，constexpr指针既可以指向常量也可以指向一个非常量：

```
constexpr int *np = nullptr; // np是一个指向整数的常量指针，其值为空
int j = 0;
constexpr int i = 42;        // i的类型是整型常量
// i和j都必须定义在函数体之外
constexpr const int *p = &i; // p是常量指针，指向整型常量i
constexpr int *p1 = &j;      // p1是常量指针，指向整数j
```

2.4.4 节练习

练习2.32：下面的代码是否合法？如果非法，请设法将其修改正确。

```
int null = 0, *p = null;
```

2.5 处理类型

随着程序越来越复杂，程序中用到的类型也越来越复杂，这种复杂性体现在两个方面。一是一些类型难于“拼写”，它们的名字既难记又容易写错，还无法明确体现其真实目的和含义。二是有时候根本搞不清到底需要的类型是什么，程序员不得不回过头去从程序的上下文中寻求帮助。

2.5.1 类型别名

类型别名（type alias）是一个名字，它是某种类型的同义词。使用类型别名有很多好处，它让复杂的类型名字变得简单明了、易于理解和使用，还有助于程序员清楚地知道使用该类型的真实目的。

有两种方法可用于定义类型别名。传统的方法是使用关键字**typedef**：

```
typedef double wages;     //wages是double的同义词
typedef wages base, *p;   //base是double的同义词，p是double*的同义词
```

68

其中，关键字typedef作为声明语句中的基本数据类型（参见2.3节，第45页）的一部分出现。含有typedef的声明语句定义的不再是变量而是类型别名。和以前的声明语句一样，这里的声明符也可以包含类型修饰，从而也能由基本数据类型构造出复合类型来。

C++ 11

新标准规定了一种新的方法，使用**别名声明**（alias declaration）来定义类型的别名：

```
using SI = Sales_item;    // SI是Sales_item的同义词
```

这种方法用关键字using作为别名声明的开始，其后紧跟别名和等号，其作用是把等号左侧的名字规定成等号右侧类型的别名。

类型别名和类型的名字等价，只要是类型的名字能出现的地方，就能使用类型别名：

```
wages hourly, weekly;          // 等价于double hourly、weekly;
```

```
SI item;                          // 等价于 Sales_item item
```

指针、常量和类型别名

如果某个类型别名指代的是复合类型或常量，那么把它用到声明语句里就会产生意想不到的后果。例如下面的声明语句用到了类型 pstring，它实际上是类型 char*的别名：

```
typedef char *pstring;
const pstring cstr = 0;   // cstr 是指向 char 的常量指针
const pstring *ps;        // ps 是一个指针，它的对象是指向 char 的常量指针
```

上述两条声明语句的基本数据类型都是 const pstring，和过去一样，const 是对给定类型的修饰。pstring 实际上是指向 char 的指针，因此，const pstring 就是指向 char 的常量指针，而非指向常量字符的指针。

遇到一条使用了类型别名的声明语句时，人们往往会错误地尝试把类型别名替换成它本来的样子，以理解该语句的含义。

```
const char *cstr = 0;     // 是对 const pstring cstr 的错误理解
```

再强调一遍：这种理解是错误的。声明语句中用到 pstring 时，其基本数据类型是指针。可是用 char*重写了声明语句后，数据类型就变成了 char，*成为了声明符的一部分。这样改写的结果是，const char 成了基本数据类型。前后两种声明含义截然不同，前者声明了一个指向 char 的常量指针，改写后的形式则声明了一个指向 const char 的指针。

2.5.2 auto 类型说明符

编程时常常需要把表达式的值赋给变量，这就要求在声明变量的时候清楚地知道表达式的类型。然而要做到这一点并非那么容易，有时甚至根本做不到。为了解决这个问题，C++11 新标准引入了 **auto** 类型说明符，用它就能让编译器替我们去分析表达式所属的类型。和原来那些只对应一种特定类型的说明符（比如 double）不同，auto 让编译器通过初始值来推算变量的类型。显然，auto 定义的变量必须有初始值：

C++11

69

```
// 由 val1 和 val2 相加的结果可以推断出 item 的类型
auto item = val1 + val2; // item 初始化为 val1 和 val2 相加的结果
```

此处编译器将根据 val1 和 val2 相加的结果来推断 item 的类型。如果 val1 和 val2 是类 Sales_item（参见 1.5 节，第 17 页）的对象，则 item 的类型就是 Sales_item；如果这两个变量的类型是 double，则 item 的类型就是 double，以此类推。

使用 auto 也能在一条语句中声明多个变量。因为一条声明语句只能有一个基本数据类型，所以该语句中所有变量的初始基本数据类型都必须一样：

```
auto i = 0, *p = &i;      // 正确：i 是整数、p 是整型指针
auto sz = 0, pi = 3.14;   // 错误：sz 和 pi 的类型不一致
```

复合类型、常量和 auto

编译器推断出来的 auto 类型有时候和初始值的类型并不完全一样，编译器会适当地改变结果类型使其更符合初始化规则。

首先，正如我们所熟知的，使用引用其实是使用引用的对象，特别是当引用被用作初始值时，真正参与初始化的其实是引用对象的值。此时编译器以引用对象的类型作为 auto 的类型：

```
int i = 0, &r = i;
auto a = r;              // a是一个整数（r是i的别名，而i是一个整数）
```

其次，auto一般会忽略掉顶层const（参见2.4.3节，第57页），同时底层const则会保留下来，比如当初始值是一个指向常量的指针时：

```
const int ci = i, &cr = ci;
auto b = ci;    // b是一个整数（ci的顶层const特性被忽略掉了）
auto c = cr;    // c是一个整数（cr是ci的别名，ci本身是一个顶层const）
auto d = &i;    // d是一个整型指针（整数的地址就是指向整数的指针）
auto e = &ci;   // e是一个指向整数常量的指针（对常量对象取地址是一种底层const）
```

如果希望推断出的auto类型是一个顶层const，需要明确指出：

```
const auto f = ci;          // ci的推演类型是int，f是const int
```

还可以将引用的类型设为auto，此时原来的初始化规则仍然适用：

```
auto &g = ci;               // g是一个整型常量引用，绑定到ci
auto &h = 42;               // 错误：不能为非常量引用绑定字面值
const auto &j = 42;         // 正确：可以为常量引用绑定字面值
```

70 设置一个类型为auto的引用时，初始值中的顶层常量属性仍然保留。和往常一样，如果我们给初始值绑定一个引用，则此时的常量就不是顶层常量了。

要在一条语句中定义多个变量，切记，符号&和*只从属于某个声明符，而非基本数据类型的一部分，因此初始值必须是同一种类型：

```
auto k = ci, &l = i;        // k是整数，l是整型引用
auto &m = ci, *p = &ci;     // m是对整型常量的引用，p是指向整型常量的指针
// 错误：i的类型是int而&ci的类型是const int
auto &n = i, *p2 = &ci;
```

2.5.2 节练习

练习2.33：利用本节定义的变量，判断下列语句的运行结果。

```
a = 42; b = 42; c = 42;
d = 42; e = 42; g = 42;
```

练习2.34：基于上一个练习中的变量和语句编写一段程序，输出赋值前后变量的内容，你刚才的推断正确吗？如果不对，请反复研读本节的示例直到你明白错在何处为止。

练习2.35：判断下列定义推断出的类型是什么，然后编写程序进行验证。

```
const int i = 42;
auto j = i; const auto &k = i; auto *p = &i;
const auto j2 = i, &k2 = i;
```

2.5.3 decltype类型指示符

C++11 有时会遇到这种情况：希望从表达式的类型推断出要定义的变量的类型，但是不想用该表达式的值初始化变量。为了满足这一要求，C++11新标准引入了第二种类型说明符**decltype**，它的作用是选择并返回操作数的数据类型。在此过程中，编译器分析表达式并得到它的类型，却不实际计算表达式的值：

```
decltype(f()) sum = x;    // sum的类型就是函数f的返回类型
```

编译器并不实际调用函数 f，而是使用当调用发生时 f 的返回值类型作为 sum 的类型。换句话说，编译器为 sum 指定的类型是什么呢？就是假如 f 被调用的话将会返回的那个类型。

decltype 处理顶层 const 和引用的方式与 auto 有些许不同。如果 decltype 使用的表达式是一个变量，则 decltype 返回该变量的类型（包括顶层 const 和引用在内）：

71

```
const int ci = 0, &cj = ci;
decltype(ci) x = 0;       // x 的类型是 const int
decltype(cj) y = x;       // y 的类型是 const int&，y 绑定到变量 x
decltype(cj) z;           // 错误：z 是一个引用，必须初始化
```

因为 cj 是一个引用，decltype(cj)的结果就是引用类型，因此作为引用的 z 必须被初始化。

需要指出的是，引用从来都作为其所指对象的同义词出现，只有用在 decltype 处是一个例外。

decltype 和引用

如果 decltype 使用的表达式不是一个变量，则 decltype 返回表达式结果对应的类型。如 4.1.1 节（第 120 页）将要介绍的，有些表达式将向 decltype 返回一个引用类型。一般来说当这种情况发生时，意味着该表达式的结果对象能作为一条赋值语句的左值：

```
// decltype 的结果可以是引用类型
int i = 42, *p = &i, &r = i;
decltype(r + 0) b;    // 正确：加法的结果是 int，因此 b 是一个（未初始化的）int
decltype(*p) c;       // 错误：c 是 int&，必须初始化
```

因为 r 是一个引用，因此 decltype(r)的结果是引用类型。如果想让结果类型是 r 所指的类型，可以把 r 作为表达式的一部分，如 r+0，显然这个表达式的结果将是一个具体值而非一个引用。

另一方面，如果表达式的内容是解引用操作，则 decltype 将得到引用类型。正如我们所熟悉的那样，解引用指针可以得到指针所指的对象，而且还能给这个对象赋值。因此，decltype(*p)的结果类型就是 int&，而非 int。

decltype 和 auto 的另一处重要区别是，decltype 的结果类型与表达式形式密切相关。有一种情况需要特别注意：对于 decltype 所用的表达式来说，如果变量名加上了一对括号，则得到的类型与不加括号时会有不同。如果 decltype 使用的是一个不加括号的变量，则得到的结果就是该变量的类型；如果给变量加上了一层或多层括号，编译器就会把它当成是一个表达式。变量是一种可以作为赋值语句左值的特殊表达式，所以这样的 decltype 就会得到引用类型：

```
// decltype 的表达式如果是加上了括号的变量，结果将是引用
decltype((i)) d;      // 错误：d 是 int&，必须初始化
decltype(i) e;        // 正确：e 是一个（未初始化的）int
```

切记：decltype((*variable*))（注意是双层括号）的结果永远是引用，而 decltype(*variable*)结果只有当 *variable* 本身就是一个引用时才是引用。

72

2.5.3 节练习

练习 2.36：关于下面的代码，请指出每一个变量的类型以及程序结束时它们各自的值。

```
int a = 3, b = 4;
decltype(a) c = a;
decltype((b)) d = a;
++c;
++d;
```

练习 2.37：赋值是会产生引用的一类典型表达式，引用的类型就是左值的类型。也就是说，如果 i 是 int，则表达式 i=x 的类型是 int&。根据这一特点，请指出下面的代码中每一个变量的类型和值。

```
int a = 3, b = 4;
decltype(a) c = a;
decltype(a = b) d = a;
```

练习 2.38：说明由 decltype 指定类型和由 auto 指定类型有何区别。请举出一个例子，decltype 指定的类型与 auto 指定的类型一样；再举一个例子，decltype 指定的类型与 auto 指定的类型不一样。

2.6 自定义数据结构

从最基本的层面理解，数据结构是把一组相关的数据元素组织起来然后使用它们的策略和方法。举一个例子，我们的 Sales_item 类把书本的 ISBN 编号、售出量及销售收入等数据组织在了一起，并且提供诸如 isbn 函数、>>、<<、+、+=等运算在内的一系列操作，Sales_item 类就是一个数据结构。

C++语言允许用户以类的形式自定义数据类型，而库类型 string、istream、ostream 等也都是以类的形式定义的，就像第 1 章的 Sales_item 类型一样。C++语言对类的支持甚多，事实上本书的第III部分和第IV部分都将大篇幅地介绍与类有关的知识。尽管 Sales_item 类非常简单，但是要想给出它的完整定义可在第 14 章介绍自定义运算符之后。

2.6.1 定义 Sales_data 类型

尽管我们还写不出完整的 Sales_item 类，但是可以尝试着把那些数据元素组织到一起形成一个简单点儿的类。初步的想法是用户能直接访问其中的数据元素，也能实现一些基本的操作。

既然我们筹划的这个数据结构不带有任何运算功能，不妨把它命名为 Sales_data 以示与 Sales_item 的区别。Sales_data 初步定义如下：

73

```
struct Sales_data {
    std::string bookNo;
    unsigned units_sold = 0;
    double revenue = 0.0;
};
```

我们的类以关键字 **struct** 开始，紧跟着类名和类体（其中类体部分可以为空）。类体由

花括号包围形成了一个新的作用域（参见 2.2.4 节，第 43 页）。类内部定义的名字必须唯一，但是可以与类外部定义的名字重复。

类体右侧的表示结束的花括号后必须写一个分号，这是因为类体后面可以紧跟变量名以示对该类型对象的定义，所以分号必不可少：

```
struct Sales_data { /* ... */ } accum, trans, *salesptr;
// 与上一条语句等价，但可能更好一些
struct Sales_data { /* ... */ };
Sales_data accum, trans, *salesptr;
```

分号表示声明符（通常为空）的结束。一般来说，最好不要把对象的定义和类的定义放在一起。这么做无异于把两种不同实体的定义混在了一条语句里，一会儿定义类，一会儿又定义变量，显然这是一种不被建议的行为。

很多新手程序员经常忘了在类定义的最后加上分号。

类数据成员

类体定义类的**成员**，我们的类只有**数据成员**（data member）。类的数据成员定义了类的对象的具体内容，每个对象有自己的一份数据成员拷贝。修改一个对象的数据成员，不会影响其他 Sales_data 的对象。

定义数据成员的方法和定义普通变量一样：首先说明一个基本类型，随后紧跟一个或多个声明符。我们的类有 3 个数据成员：一个名为 bookNo 的 string 成员、一个名为 units_sold 的 unsigned 成员和一个名为 revenue 的 double 成员。每个 Sales_data 的对象都将包括这 3 个数据成员。

C++11 新标准规定，可以为数据成员提供一个**类内初始值**（in-class initializer）。创建对象时，类内初始值将用于初始化数据成员。没有初始值的成员将被默认初始化（参见 2.2.1 节，第 40 页）。因此当定义 Sales_data 的对象时，units_sold 和 revenue 都将初始化为 0，bookNo 将初始化为空字符串。

C++11

对类内初始值的限制与之前（参见 2.2.1 节，第 39 页）介绍的类似：或者放在花括号里，或者放在等号右边，记住不能使用圆括号。

7.2 节（第 240 页）将要介绍，用户可以使用 C++语言提供的另外一个关键字 class 来定义自己的数据结构，到时也将说明现在我们使用 struct 的原因。在第 7 章学习与 class 有关的知识之前，建议读者继续使用 struct 定义自己的数据类型。

2.6.1 节练习

74

练习 2.39：编译下面的程序观察其运行结果，注意，如果忘记写类定义体后面的分号会发生什么情况？记录下相关信息，以后可能会有用。

```
struct Foo { /* 此处为空 */ } // 注意：没有分号
int main()
{
    return 0;
}
```

练习 2.40：根据自己的理解写出 Sales_data 类，最好与书中的例子有所区别。

2.6.2 使用 Sales_data 类

和 Sales_item 类不同的是，我们自定义的 Sales_data 类没有提供任何操作，Sales_data 类的使用者如果想执行什么操作就必须自己动手实现。例如，我们将参照 1.5.2 节（第 20 页）的例子写一段程序实现求两次交易相加结果的功能。程序的输入是下面这两条交易记录：

```
0-201-78345-X 3 20.00
0-201-78345-X 2 25.00
```

每笔交易记录着图书的 ISBN 编号、售出数量和售出单价。

添加两个 Sales_data 对象

因为 Sales_data 类没有提供任何操作，所以我们必须自己编码实现输入、输出和相加的功能。假设已知 Sales_data 类定义于 Sales_data.h 文件内，2.6.3 节（第 67 页）将详细介绍定义头文件的方法。

因为程序比较长，所以接下来分成几部分介绍。总的来说，程序的结构如下：

```
#include <iostream>
#include <string>
#include "Sales_data.h"
int main()
{
    Sales_data data1, data2;
    // 读入 data1 和 data2 的代码
    // 检查 data1 和 data2 的 ISBN 是否相同的代码
    // 如果相同，求 data1 和 data2 的总和
}
```

和原来的程序一样，先把所需的头文件包含进来并且定义变量用于接受输入。和 Sales_item 类不同的是，新程序还包含了 string 头文件，因为我们的代码中将用到 string 类型的成员变量 bookNo。

75 Sales_data 对象读入数据

第 3 章和第 10 章将详细介绍 string 类型的细节，在此之前，我们先了解一点儿关于 string 的知识以便定义和使用我们的 ISBN 成员。string 类型其实就是字符的序列，它的操作有>>、<<和==等，功能分别是读入字符串、写出字符串和比较字符串。这样我们就能书写代码读入第一笔交易了：

```
double price = 0; // 书的单价，用于计算销售收入
// 读入第 1 笔交易： ISBN、销售数量、单价
std::cin >> data1.bookNo >> data1.units_sold >> price;
// 计算销售收入
data1.revenue = data1.units_sold * price;
```

交易信息记录的是书售出的单价，而数据结构存储的是一次交易的销售收入，因此需要将单价读入到 double 变量 price，然后再计算销售收入 revenue。输入语句

```
std::cin >> data1.bookNo >> data1.units_sold >> price;
```

使用点操作符（参见 1.5.2 节，第 20 页）读入对象 data1 的 bookNo 成员和 units_sold 成员。

最后一条语句把 data1.units_sold 和 price 的乘积赋值给 data1 的 revenue 成员。

接下来程序重复上述过程读入对象 data2 的数据：

```
// 读入第 2 笔交易
std::cin >> data2.bookNo >> data2.units_sold >> price;
data2.revenue = data2.units_sold * price;
```

输出两个 Sales_data 对象的和

剩下的工作就是检查两笔交易涉及的 ISBN 编号是否相同了。如果相同输出它们的和，否则输出一条报错信息：

```
if (data1.bookNo == data2.bookNo) {
    unsigned totalCnt = data1.units_sold + data2.units_sold;
    double totalRevenue = data1.revenue + data2.revenue;
    // 输出：ISBN、总销售量、总销售额、平均价格
    std::cout << data1.bookNo << " " << totalCnt
              << " " << totalRevenue << " ";
    if (totalCnt != 0)
        std::cout << totalRevenue/totalCnt << std::endl;
    else
        std::cout << "(no sales)" << std::endl;
    return 0;          // 标示成功
} else {               // 两笔交易的 ISBN 不一样
    std::cerr << "Data must refer to the same ISBN"
              << std::endl;
    return -1;         // 标示失败
}
```

在第一个 if 语句中比较了 data1 和 data2 的 bookNo 成员是否相同。如果相同则执行第一个 if 语句花括号内的操作，首先计算 units_sold 的和并赋给变量 totalCnt，然后计算 revenue 的和并赋给变量 totalRevenue，输出这些值。接下来检查是否确实售出了书籍，如果是，计算并输出每本书的平均价格；如果售量为零，输出一条相应的信息。 76

2.6.2 节练习

练习 2.41：使用你自己的 Sales_data 类重写 1.5.1 节（第 20 页）、1.5.2 节（第 21 页）和 1.6 节（第 22 页）的练习。眼下先把 Sales_data 类的定义和 main 函数放在同一个文件里。

2.6.3 编写自己的头文件

尽管如 19.7 节（第 754 页）所讲可以在函数体内定义类，但是这样的类毕竟受到了一些限制。所以，类一般都不定义在函数体内。当在函数体外部定义类时，在各个指定的源文件中可能只有一处该类的定义。而且，如果要在不同文件中使用同一个类，类的定义就必须保持一致。

为了确保各个文件中类的定义一致，类通常被定义在头文件中，而且类所在头文件的名字应与类的名字一样。例如，库类型 string 在名为 string 的头文件中定义。又如，我们应该把 Sales_data 类定义在名为 Sales_data.h 的头文件中。

头文件通常包含那些只能被定义一次的实体，如类、const 和 constexpr 变量（参见 2.4 节，第 54 页）等。头文件也经常用到其他头文件的功能。例如，我们的 Sales_data 类包含有一个 string 成员，所以 Sales_data.h 必须包含 string.h 头文件。同时，使用 Sales_data 类的程序为了能操作 bookNo 成员需要再一次包含 string.h 头文件。

这样，事实上使用 Sales_data 类的程序就先后两次包含了 string.h 头文件：一次是直接包含的，另有一次是随着包含 Sales_data.h 被隐式地包含进来的。有必要在书写头文件时做适当处理，使其遇到多次包含的情况也能安全和正常地工作。

头文件一旦改变，相关的源文件必须重新编译以获取更新过的声明。

预处理器概述

确保头文件多次包含仍能安全工作的常用技术是**预处理器**（preprocessor），它由 C++ 77 语言从 C 语言继承而来。预处理器是在编译之前执行的一段程序，可以部分地改变我们所写的程序。之前已经用到了一项预处理功能#include，当预处理器看到#include 标记时就会用指定的头文件的内容代替#include。

C++程序还会用到的一项预处理功能是**头文件保护符**（header guard），头文件保护符依赖于预处理变量（参见 2.3.2 节，第 48 页）。预处理变量有两种状态：已定义和未定义。**#define** 指令把一个名字设定为预处理变量，另外两个指令则分别检查某个指定的预处理变量是否已经定义：**#ifdef** 当且仅当变量已定义时为真，**#ifndef** 当且仅当变量未定义时为真。一旦检查结果为真，则执行后续操作直至遇到**#endif** 指令为止。

使用这些功能就能有效地防止重复包含的发生：

```
#ifndef SALES_DATA_H
#define SALES_DATA_H
#include <string>
struct Sales_data {
    std::string bookNo;
    unsigned units_sold = 0;
    double revenue = 0.0;
};
#endif
```

第一次包含 Sales_data.h 时，#ifndef 的检查结果为真，预处理器将顺序执行后面的操作直至遇到#endif 为止。此时，预处理变量 SALES_DATA_H 的值将变为已定义，而且 Sales_data.h 也会被拷贝到我们的程序中来。后面如果再一次包含 Sales_data.h，则#ifndef 的检查结果将为假，编译器将忽略#ifndef 到#endif 之间的部分。

预处理变量无视 C++语言中关于作用域的规则。

整个程序中的预处理变量包括头文件保护符必须唯一，通常的做法是基于头文件中类的名字来构建保护符的名字，以确保其唯一性。为了避免与程序中的其他实体发生名字冲突，一般把预处理变量的名字全部大写。

头文件即使（目前还）没有被包含在任何其他头文件中，也应该设置保护符。头文件保护符很简单，程序员只要习惯性地加上就可以了，没必要太在乎你的程序到底需不需要。

2.6.3 节练习

练习 2.42：根据你自己的理解重写一个 Sales_data.h 头文件，并以此为基础重做 2.6.2 节（第 67 页）的练习。

小结

78

类型是C++编程的基础。

类型规定了其对象的存储要求和所能执行的操作。C++语言提供了一套基础内置类型，如 int 和 char 等，这些类型与实现它们的机器硬件密切相关。类型分为非常量和常量，一个常量对象必须初始化，而且一旦初始化其值就不能再改变。此外，还可以定义复合类型，如指针和引用等。复合类型的定义以其他类型为基础。

C++语言允许用户以类的形式自定义类型。C++库通过类提供了一套高级抽象类型，如输入输出和 string 等。

术语表

地址（address）是一个数字，根据它可以找到内存中的一个字节。

别名声明（alias declaration）为另外一种类型定义一个同义词：使用“名字=类型”的格式将名字作为该类型的同义词。

算术类型（arithmetic type）布尔值、字符、整数、浮点数等内置类型。

数组（array）是一种数据结构，存放着一组未命名的对象，可以通过索引来访问这些对象。3.5 节将详细介绍数组的知识。

auto 是一个类型说明符，通过变量的初始值来推断变量的类型。

基本类型（base type）是类型说明符，可用 const 修饰，在声明语句中位于声明符之前。基本类型提供了最常见的数据类型，以此为基础构建声明符。

绑定（bind）令某个名字与给定的实体关联在一起，使用该名字也就是使用该实体。例如，引用就是将某个名字与某个对象绑定在一起。

字节（byte）内存中可寻址的最小单元，大多数机器的字节占 8 位。

类成员（class member）类的组成部分。

复合类型（compound type）是一种类型，它的定义以其他类型为基础。

const 是一种类型修饰符，用于说明永不改变的对象。const 对象一旦定义就无法再赋新值，所以必须初始化。

常量指针（const pointer）是一种指针，它的值永不改变。

常量引用（const reference）是一种习惯叫法，含义是指向常量的引用。

常量表达式（const expression）能在编译时计算并获取结果的表达式。

constexpr 是一种函数，用于代表一条常量表达式。6.5.2 节（第 214 页）将介绍 constexpr 函数。

转换（conversion）一种类型的值转变成另外一种类型值的过程。C++语言支持内置类型之间的转换。

数据成员（data member）组成对象的数据元素，类的每个对象都有类的数据成员的一份拷贝。数据成员可以在类内部声明的同时初始化。

声明（declaration）声称存在一个变量、函数或是别处定义的类型。名字必须在定义或声明之后才能使用。 79

声明符（declarator）是声明的一部分，包括被定义的名字和类型修饰符，其中类型修饰符可以有也可以没有。

decltype 是一个类型说明符，从变量或表达式推断得到类型。

默认初始化（default initialization）当对象未被显式地赋予初始值时执行的初始化行

为。由类本身负责执行的类对象的初始化行为。全局作用域的内置类型对象初始化为 0；局部作用域的对象未被初始化即拥有未定义的值。

定义（definition）为某一特定类型的变量申请存储空间，可以选择初始化该变量。名字必须在定义或声明之后才能使用。

转义序列（escape sequence）字符特别是那些不可打印字符的替代形式。转义以反斜线开头，后面紧跟一个字符，或者不多于 3 个八进制数字，或者字母 x 加上 1 个十六进制数。

全局作用域（global scope）位于其他所有作用域之外的作用域。

头文件保护符（header guard）使用预处理变量以防止头文件被某个文件重复包含。

标识符（identifier）组成名字的字符序列，标识符对大小写敏感。

类内初始值（in-class initializer）在声明类的数据成员时同时提供的初始值，必须置于等号右侧或花括号内。

在作用域内（in scope）名字在当前作用域内可见。

被初始化（initialized）变量在定义的同时被赋予初始值，变量一般都应该被初始化。

内层作用域（inner scope）嵌套在其他作用域之内的作用域。

整型（integral type）参见算术类型。

列表初始化（list initialization）利用花括号把一个或多个初始值放在一起的初始化形式。

字面值（literal）是一个不能改变的值，如数字、字符、字符串等。单引号内的是字符字面值，双引号内的是字符串字面值。

局部作用域（local scope）是块作用域的习惯叫法。

底层 const（low-level const）一个不属于顶层的 `const`，类型如果由底层常量定义，则不能被忽略。

成员（member）类的组成部分。

不可打印字符（nonprintable character）不具有可见形式的字符，如控制符、退格、换行符等。

空指针（null pointer）值为 0 的指针，空指针合法但是不指向任何对象。

nullptr 是表示空指针的字面值常量。

对象（object）是内存的一块区域，具有某种类型，变量是命名了的对象。

外层作用域（outer scope）嵌套着别的作用域的作用域。

指针（pointer）是一个对象，存放着某个对象的地址，或者某个对象存储区域之后的下一地址，或者 0。

指向常量的指针（pointer to const）是一个指针，存放着某个常量对象的地址。指向常量的指针不能用来改变它所指对象的值。

预处理器（preprocessor）在 C++编译过程中执行的一段程序。

预处理变量（preprocessor variable）由预处理器管理的变量。在程序编译之前，预处理器负责将程序中的预处理变量替换成它的真实值。

引用（reference）是某个对象的别名。

80 **对常量的引用**（reference to const）是一个引用，不能用来改变它所绑定对象的值。对常量的引用可以绑定常量对象，或者非常量对象，或者表达式的结果。

作用域（scope）是程序的一部分，在其中某些名字有意义。C++有几级作用域：

全局（global）——名字定义在所有其他作用域之外。

类（class）——名字定义在类内部。

命名空间（namespace）——名字定义在命名空间内部。

块（block）——名字定义在块内部。

名字从声明位置开始直至声明语句所在的作用域末端为止都是可用的。

分离式编译（separate compilation）把程序分割为多个单独文件的能力。

带符号类型（signed）保存正数、负数或 0 的整型。

字符串（string）是一种库类型，表示可变长字符序列。

struct 是一个关键字，用于定义类。

临时值（temporary）编译器在计算表达式结果时创建的无名对象。为某表达式创建了一个临时值，则此临时值将一直存在直到包含有该表达式的最大的表达式计算完成为止。

顶层 const（top-level const）是一个 `const`，规定某对象的值不能改变。

类型别名（type alias）是一个名字，是另外一个类型的同义词，通过关键字 `typedef` 或别名声明语句来定义。

类型检查（type checking）是一个过程，编译器检查程序使用某给定类型对象的方式与该类型的定义是否一致。

类型说明符（type specifier）类型的名字。

typedef 为某类型定义一个别名。当关键字 `typedef` 作为声明的基本类型出现时，声明中定义的名字就是类型名。

未定义（undefined）即 C++语言没有明确规定的情况。不论是否有意为之，未定义行为都可能引发难以追踪的运行时错误、安全问题和可移植性问题。

未初始化（uninitialized）变量已定义但未被赋予初始值。一般来说，试图访问未初始化变量的值将引发未定义行为

无符号类型（unsigned）保存大于等于 0 的整型。

变量（variable）命名的对象或引用。C++语言要求变量要先声明后使用。

void* 可以指向任意非常量的指针类型，不能执行解引用操作。

void 类型 是一种有特殊用处的类型，既无操作也无值。不能定义一个 `void` 类型的变量。

字（word）在指定机器上进行整数运算的自然单位。一般来说，字的空间足够存放地址。32 位机器上的字通常占据 4 个字节。

&运算符（& operator）取地址运算符。

***运算符**（* operator）解引用运算符。解引用一个指针将返回该指针所指的对象，为解引用的结果赋值也就是为指针所指的对象赋值。

#define 是一条预处理指令，用于定义一个预处理变量。

#endif 是一条预处理指令，用于结束一个 `#ifdef` 或`#ifndef` 区域。

#ifdef 是一条预处理指令，用于判断给定的变量是否已经定义。

#ifndef 是一条预处理指令，用于判断给定的变量是否尚未定义。

第 3 章
字符串、向量和数组

内容

3.1 命名空间的 using 声明 74
3.2 标准库类型 string 75
3.3 标准库类型 vector 86
3.4 迭代器介绍 95
3.5 数组 101
3.6 多维数组 112
小结 117
术语表 117

除了第 2 章介绍的内置类型之外，C++语言还定义了一个内容丰富的抽象数据类型库。其中，`string` 和 `vector` 是两种最重要的标准库类型，前者支持可变长字符串，后者则表示可变长的集合。还有一种标准库类型是迭代器，它是 `string` 和 `vector` 的配套类型，常被用于访问 `string` 中的字符或 `vector` 中的元素。

内置数组是一种更基础的类型，`string` 和 `vector` 都是对它的某种抽象。本章将分别介绍数组以及标准库类型 `string` 和 `vector`。

82 第2章介绍的内置类型是由C++语言直接定义的。这些类型，比如数字和字符，体现了大多数计算机硬件本身具备的能力。标准库定义了另外一组具有更高级性质的类型，它们尚未直接实现到计算机硬件中。

本章将介绍两种最重要的标准库类型：string 和 vector。string 表示可变长的字符序列，vector 存放的是某种给定类型对象的可变长序列。本章还将介绍内置数组类型，和其他内置类型一样，数组的实现与硬件密切相关。因此相较于标准库类型 string 和 vector，数组在灵活性上稍显不足。

在开始介绍标准库类型之前，先来学习一种访问库中名字的简单方法。

3.1 命名空间的 using 声明

目前为止，我们用到的库函数基本上都属于命名空间 std，而程序也显式地将这一点标示了出来。例如，std::cin 表示从标准输入中读取内容。此处使用作用域操作符（::）（参见1.2节，第7页）的含义是：编译器应从操作符左侧名字所示的作用域中寻找右侧那个名字。因此，std::cin 的意思就是要使用命名空间 std 中的名字 cin。

上面的方法显得比较烦琐，然而幸运的是，通过更简单的途径也能使用到命名空间中的成员。本节将学习其中一种最安全的方法，也就是使用 **using 声明**（using declaration），18.2.2节（第702页）会介绍另一种方法。

有了 using 声明就无须专门的前缀（形如命名空间::）也能使用所需的名字了。using 声明具有如下的形式：

```
using namespace::name;
```

一旦声明了上述语句，就可以直接访问命名空间中的名字：

```
#include <iostream>
// using 声明，当我们使用名字 cin 时，从命名空间 std 中获取它
using std::cin;

int main()
{
    int i;
    cin >> i;          // 正确：cin 和 std::cin 含义相同
    cout << i;         // 错误：没有对应的 using 声明，必须使用完整的名字
    std::cout << i;    // 正确：显式地从 std 中使用 cout
    return 0;
}
```

每个名字都需要独立的 using 声明

按照规定，每个 using 声明引入命名空间中的一个成员。例如，可以把要用到的标准库中的名字都以 using 声明的形式表示出来，重写1.2节（第5页）的程序如下：

83

```
#include <iostream>
// 通过下列 using 声明，我们可以使用标准库中的名字
using std::cin;
using std::cout; using std::endl;
int main()
{
    cout << "Enter two numbers:" << endl;
```

```
    int v1, v2;
    cin >> v1 >> v2;
    cout << "The sum of " << v1 << " and " << v2
         << " is " << v1 + v2 << endl;
    return 0;
}
```

在上述程序中，一开始就有对 cin、cout 和 endl 的 using 声明，这意味着我们不用再添加 std::形式的前缀就能直接使用它们。C++语言的形式比较自由，因此既可以一行只放一条 using 声明语句，也可以一行放上多条。不过要注意，用到的每个名字都必须有自己的声明语句，而且每句话都得以分号结束。

头文件不应包含 using 声明

位于头文件的代码（参见 2.6.3 节，第 67 页）一般来说不应该使用 using 声明。这是因为头文件的内容会拷贝到所有引用它的文件中去，如果头文件里有某个 using 声明，那么每个使用了该头文件的文件就都会有这个声明。对于某些程序来说，由于不经意间包含了一些名字，反而可能产生始料未及的名字冲突。

一点注意事项

经本节所述，后面的所有例子将假设，但凡用到的标准库中的名字都已经使用 using 语句声明过了。例如，我们将在代码中直接使用 cin，而不再使用 std::cin。

为了让书中的代码尽量简洁，今后将不会再把所有 using 声明和#include 指令一一标出。附录 A 中的表 A.1（第 766 页）列出了本书涉及的所有标准库中的名字及对应的头文件。

读者请注意：在编译及运行本书的示例前请为代码添加必要的#include 指令和 using 声明。

3.1 节练习

练习 3.1：使用恰当的 using 声明重做 1.4.1 节（第 11 页）和 2.6.2 节（第 67 页）的练习。

3.2 标准库类型 string

标准库类型 **string** 表示可变长的字符序列，使用 string 类型必须首先包含 string 头文件。作为标准库的一部分，string 定义在命名空间 std 中。接下来的示例都假定已包含了下述代码：

84

```
#include <string>
using std::string;
```

本节描述最常用的 string 操作，9.5 节（第 320 页）还将介绍另外一些。

C++标准一方面对库类型所提供的操作做了详细规定，另一方面也对库的实现者做出一些性能上的需求。因此，标准库类型对于一般应用场合来说有足够的效率。

3.2.1 定义和初始化 string 对象

如何初始化类的对象是由类本身决定的。一个类可以定义很多种初始化对象的方式，只不过这些方式之间必须有所区别：或者是初始值的数量不同，或者是初始值的类型不同。表 3.1 列出了初始化 string 对象最常用的一些方式，下面是几个例子：

```
string s1;              // 默认初始化，s1 是一个空字符串
string s2 = s1;         // s2 是 s1 的副本
string s3 = "hiya";     // s3 是该字符串字面值的副本
string s4(10, 'c');     // s4 的内容是 cccccccccc
```

可以通过默认的方式（参见 2.2.1 节，第 40 页）初始化一个 string 对象，这样就会得到一个空的 string，也就是说，该 string 对象中没有任何字符。如果提供了一个字符串字面值（参见 2.1.3 节，第 36 页），则该字面值中除了最后那个空字符外其他所有的字符都被拷贝到新创建的 string 对象中去。如果提供的是一个数字和一个字符，则 string 对象的内容是给定字符连续重复若干次后得到的序列。

表 3.1：初始化 string 对象的方式

`string s1`	默认初始化，s1 是一个空串
`string s2(s1)`	s2 是 s1 的副本
`string s2 = s1`	等价于 s2(s1)，s2 是 s1 的副本
`string s3("value")`	s3 是字面值"value"的副本，除了字面值最后的那个空字符外
`string s3 = "value"`	等价于 s3("value")，s3 是字面值"value"的副本
`string s4(n, 'c')`	把 s4 初始化为由连续 n 个字符 c 组成的串

直接初始化和拷贝初始化

由 2.2.1 节（第 39 页）的学习可知，C++语言有几种不同的初始化方式，通过 string 我们可以清楚地看到在这些初始化方式之间到底有什么区别和联系。如果使用等号（=）初始化一个变量，实际上执行的是**拷贝初始化**（copy initialization），编译器把等号右侧的初始值拷贝到新创建的对象中去。与之相反，如果不使用等号，则执行的是**直接初始化**（direct initialization）。

当初始值只有一个时，使用直接初始化或拷贝初始化都行。如果像上面的 s4 那样初始化要用到的值有多个，一般来说只能使用直接初始化的方式：

```
string s5 = "hiya";     // 拷贝初始化
string s6("hiya");      // 直接初始化
string s7(10, 'c');     // 直接初始化，s7 的内容是 cccccccccc
```

85 对于用多个值进行初始化的情况，非要用拷贝初始化的方式来处理也不是不可以，不过需要显式地创建一个（临时）对象用于拷贝：

```
string s8 = string(10, 'c'); // 拷贝初始化，s8 的内容是 cccccccccc
```

s8 的初始值是 string(10, 'c')，它实际上是用数字 10 和字符 c 两个参数创建出来的一个 string 对象，然后这个 string 对象又拷贝给了 s8。这条语句本质上等价于下面的两条语句：

```
string temp(10, 'c');   // temp 的内容是 cccccccccc
string s8 = temp;       // 将 temp 拷贝给 s8
```