第 2 章　基本数据类型和表达式

学习内容和目标

本章将讲解 C++ 基本数据类型、对象的基本属性、const 修饰符、类型自动推导、运算符和表达式计算。学完本章内容后，读者将可以：

- 理解基本数据类型的内存结构；
- 理解对象的基本属性；
- 学会运用 const 修饰符和类型自动推导；
- 掌握表达式求值的基本方法。

2.1　C++ 语句基本元素

C++ 语言的基本字符集包括 26 个大小写字母、10 个阿拉伯数字，以及符号 + - * / = , . _ : ; ? " ' ~ | ! # % & ( ) [ ] { } ^ < > 和空格等。这些字符集构成了 C++ 语句的基本元素，它们包括标识符、关键字、字面值常量、运算符及标点符号。

2.1.1　标识符

C++ 标识符(identifier)由字母、数字和下画线组成，而且必须以字母或下画线开头。

由程序员自己定义的标识符称为用户自定义标识符，它是由程序员定义的符号和单词，用来给对象、函数、类等实体进行命名。例如，m_radius、volume 和 x1 等是合法的标识符，而 2nd 和 no. 是非法的标识符。

C++ 严格区分大小写字母，如 name、Name 和 NAME 是三个不同的标识符。

2.1.2　关键字

如表 2.1 所示，C++ 语言保留了一些名字供语言自身使用，它们具有特定的含义，被称为关键字(keyword)，这些名字不能作为用户自定义标识符。表 2.1 中带后缀标记的关键字为 C++11 和 C++17 标准下新增关键字或者是对原含义做了修改。

关于表 2.1 中的关键字的含义和使用方法将在后面章节介绍。除上面的关键字外，运算符替代名 and(&&)、bitand(&)、compl(~)、not_eq(!=)、or_eq(|=)、xor_eq(^=)、and_eq(&=)、bitor(|)、not(!)、or(||) 和 xor(^) 也不能作为用户自定义标识符名。

表 2.1 C++ 关键字

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

§：C++11 标准新增关键字。

†：在 C++11 标准下修改了原含义或者增加了新含义。

‡：在 C++17 标准下修改了原含义。

建议：良好的编程习惯从标识符命名开始

不建议使用 C++ 标准库里面的名字，如库函数名、对象名和类名等。另外，也不提倡使用标准库保留的名字，如连续出现两个下画线、以下画线紧接大写字母开头和定义在函数体外以下画线开头的标识符。

为了提高程序的可读性，对于用户自定义标识符的命名，建议使用下面约定俗成的规则：

- 对象名一般用小写字母，如 name，而不是 Name 或 NAME；
- 应使用能帮助记忆的名字，如 salary，而不是 s；
- 由多个单词组成时，单词之间可用下画线或每个内嵌单词的第一个字母大写，如 student_name 或 studentName，而不是 studentname。

2.2 基本数据类型

数据类型是程序的基础，它定义了数据的意义及与数据相关的操作。C++ 定义了几种基本数据类型，包括算术类型（arithmetic type）和空类型（void）。算术类型也称为内置类型（build-in type），包括布尔型、字符型、整型和实型。空类型没有具体的值，仅用于特殊的场合。

2.2.1 内置类型

内置类型在内存中所占用的存储空间(Byte，字节)在不同的机器上或者不同的编译环境下有所不同，C++ 规定了每种内置类型所占存储空间的最小值或精度。表 2.2 列出了 C++ 规定的每种内置类型的最小内存空间(精度)及在 Visual C++ 编译环境下的内存空间。

表 2.2 C++ 基本内置类型

类 型	含 义	尺寸(单位:字节 最小尺寸/精度	Visual C++
bool	布尔型	未定义	1
char	字符型	1	1
wchar_t	宽字符型	2	2
char16_t	Unicode 字符型	2	2
char32_t	Unicode 字符型	4	4
short	短整型	2	2
int	整型	2	4
long	长整型	4	4
long long	双长整型	8	8
float	单精度浮点型	6 位有效数字(IEEE 754)	4
double	双精度浮点型	15 位有效数字(IEEE 754)	8
long double	扩展的精度浮点型	精度不低于 double 类型	8

位(bit)是计算机中存储数据的最小单位，指二进制数中的一个位数，其值为 0 或 1。

字节(Byte)是计算机存储容量的基本单位，一个字节由 8 位二进制数组成，即 8 个比特位。

布尔型(bool)的取值为 true(真)或 false(假)。

字符型 char 用来保存机器基本字符集中对应的整数值，即该字符的 ASCII 码值(见附录 A)，占用一个字节，如大写字母 A 和小写字母 a 的 ASCII 码值分别为 65 和 97。除 char 基本字符型外，C++ 还支持扩展的字符集，如 wchar_t、char16_t 和 char32_t。wchar_t 又称双字节字符型，可以存放扩展字符集中任意一个字符，如中文字符。char16_t 和 char32_t 用来表示 Unicode 字符集，Unicode 为每种语言中的每个字符设定了统一且唯一的二进制编码。

浮点型可表示单精度(float)、双精度(double)和扩展精度(long double)，用来保存实数。C++ 标准没有指定它们的尺寸，通常根据 IEEE 754 标准，float 以 4 个字节来表示，double 以 8 个字节表示，long double 以 8、12 或 16 个字节来表示。一般来说，编译器都能保证 float 和 double 分别有 6 和 15 位有效数字。

整型用来保存整数。除 bool 类型和扩展的字符类型外，其他的整型可分为带符号的(signed)和无符号的(unsigned)两种。无符号数表示大于等于 0 的值，而带符号数可以表示正数、负数和 0。类型 int、short、long 和 long long 都是带符号的。如果要表达无符号类型，需要在这些类型名前面添加 unsigned 修饰符，例如 unsigned int。

提示：整型数在计算机内存中以补码形式存储

一个数在计算机中以二进制形式表示，称为机器数。机器数是带符号的，二进制的最高位(左边第一位)存放符号位，正数为 0，负数为 1。例如：十进制数 3，转换成 8 位二进制数为 0000 0011，-3 为 1000 0011。这种表示方式称为原码。正数的反码是其本身，负数的反码是在其原码的基础上，符号位不变，其余按位取反，如 -3 的原码为 1000 0011，反码为 1111 1100。正数的补码是其本身，负数的补码是在其反码的基础上加 1，如 -3 的补码为 1111 1100 + 1 = 1111 1101。因此，一个 signed char 类型表示范围为 -128 ~ 127。

2.2.2 常量

常量也称为字面值常量(literal)，例如 -3 和 3.14。常量的表示形式取决于它的数据类型。

1. 整型常量

C++ 中的整型常量可以用十进制、八进制或十六进制数表示。以 0 开头的数代表八进制数，八进制数由数字 0 ~ 7 组成。以 0x 或 0X 开头的数代表十六进制数，十六进制数由数字 0 ~ 9 和字母 A ~ F(大小写均可)组成。例如，整型常量 58 的三种表示形式分别为：

58(十进制)　　072(八进制)　　0x3A(十六进制)

2. 实型常量

实型常量以小数或指数形式(科学计数法)表示。指数形式由尾数、阶数和 E 或 e 组成，其中在 E 或 e 前面的尾数部分必须有数字，后面的阶数必须为整数，例如：

```
3.14159    3.14159E0    0.    0E0    .314
```

实型常量默认为 double 类型。

3. 字符和字符串常量

用单引号引起来单个字符称为字符常量，而由双引号引起来的零个或多个字符称为字符串常量，例如：

```
'a'    '1'    '@'          //字符常量
"a"    "Mandy"             //字符串常量
```

一些无法打印的字符是无法直接使用的，如回车符、换行符、制表符和退格符等空白字符(white-space character)。另外，有一些具有特殊含义的字符，如单引号、双引号、问号和反斜线等，也不能直接使用。这种情况下，C++ 提供了转义序列(escape sequence)，转义序列以反斜线开始，包括如下字符：

```
\n 换行符,光标到下行行首      \t 水平制表符              \a 报警(响铃)符
\f 换页符                     \r 回车符,光标到本行行首   \b 退格符
```

\v 垂直制表符　　\\ 反斜线　　\' 单引号
\" 双引号　　\? 问号

C++ 也支持泛化的转义字符，如 \x 后紧跟一个或多个十六进制数，\后紧跟 1 ~ 3 个八进制数。

\0	空字符(null)	\nnn	八进制	\xnn	十六进制
\unnnn	Unicode(UTF-8)	\Unnnnnnnn	Unicode (UTF-16)		

因此，'\141'和'\x61'都表示字符 'a'。注意反斜杠 \ 后面的数字最多不超过 3 位有效数位，例如，"\1234" 表示两个字符(八进制数 123 对应的字符和字符 4)，"\459" 表示两个字符(八进制数 45 对应的字符和字符 9)。与八进制数不同，\x 用到后面的所有有效数位，例如，"\x0034" 表示十六进制数 34 对应的一个 8 位的字符(即字符 4)，"\x1234" 表示一个 16 位的字符。因为大多数机器的 char 类型为 8 位，所以对于上面的例子，编译器很可能会报错。

可以像使用普通字符那样使用转义字符，例如：

```
cout << "Hi \"\103++\"\n";     //输出 Hi "C++", 转到新一行
```

对于字符串常量，编译器在内存中逐个存放字符串的每一个字符的 ASCII 值，而且在结尾处加上一个空字符('\0')，用来标记字符串的终止，因此，字符串常量 "a" 要比字符常量'a'在内存中多一个字节的存储空间。

4. 前缀和后缀

可以为整型、实型和字符型常量添加前缀或后缀，来改变其默认类型，例如：

3.14159L	扩展精度实型字面值常量，类型为 long double
-84L	长整型，类型为 long int
3.14E-3F	单精度实型常量，类型为 float
L'a'	宽字面值常量，类型为 wchar_t
0x2aLU	十六进制表示的无符号长整型数 42，类型为 unsigned long int

其中，整型常量后缀 u 或 U 表示 unsigned，l 或 L 表示 long，ll 或 LL 表示 long long。实型常量后缀 f 或 F 表示 float，l 或 L 表示 long double。字符或字符串常量前缀 u 表示 char16_t，U 表示 char32_t，L 表示 wchar_t。

2.3 对象

对象是数据和操作的载体，程序通过对象获取内存中的数据，以及依赖于数据的操作。理解对象是把握 C++ 内存管理方式和掌握面向对象程序设计方法的基础。对于程序中的任何一个对象，需要掌握几个属性，包括数据类型、名字、内存结构、支持的操作、生命期(lifetime)和作用域(scope)。例如，当创建一个 double 类型对象时，它会占用连续 8 个字节的内存空间；它虽然支持加法、减法、乘法和除法操作，但不支持求余操作；另外，还需要知道它在程序中的可见范围及在内存中消亡的时刻。

2.3.1 对象的定义和初始化

1. 对象的定义

定义对象的一般格式是：以类型说明符(type specifier)开头，紧跟一个或多个对象名，其中对象名以逗号分隔，并以分号结束，例如：

```
int counter, age;          //定义两个整型类型对象 counter 和 age
Cylinder object;           //定义一个 Cylinder 类型(见例 1.3)对象 object
```

当一个对象在内存中被创建时，编译器会根据其数据类型为其分配一块相应大小的内存空间，该对象的生命期由此开始。访问一个对象所在内存空间中的数据，需要知道其内存空间所在的内存地址，这是通过对象的名字来实现的，也就是说对象名本质上是内存地址的一个映射。

提示："对象"和"变量"

传统上，对象和变量这两个术语可以互换。但本书强调对象而不使用变量，原因在于变量一般被认为是一块具有某种数据类型的内存空间，强调内存空间中存放的内容，而对象不仅包含内存结构而且还强调与之关联的操作，是一种面向对象的思维。

2. 对象的初始化

上面定义整型对象 age 时，并没有提供一个初始值，那么 age 对应的内存空间里面的内容是什么？读者可以思考并测试一下。当一个对象在内存中被创建时，被赋予了一个默认值，这个默认值是什么取决于对象的类型及对象被定义的位置(见 5.2 节)。如果对象在创建时没有被初始化，则该对象是不能直接参与计算的。在定义一个对象时，如果知道它的初始值，则务必提供一个初始值，然后就可以使用这个对象了，例如：

```
int age = 18;
```

可以在同一条定义语句中用先定义的对象的值初始化后面定义的对象，例如：

```
//year 先于 birthYear 创建和初始化，可用于初始化 birthYear
int year = 2017, birthYear = year;
```

在上面的定义中，虽然利用赋值运算符(=)对两个对象进行初始化，但并不是赋值操作。二者有本质的区别，例如：

```
int year = 0;          //定义对象并初始化，在此之前，year 在内存中不存在
year = 2017;           //赋值操作，在对 year 赋值之前，它在内存中已经存在
```

上面第一条语句是初始化操作，在执行这条语句之前，year 是不存在的。第二条语句是赋值操作，在执行赋值的时候，year 已经在内存中了。

利用赋值运算符(=)初始化对象，称为复制初始化(copy initialization)。除复制初始化方式外，C++ 还提供了直接初始化(direct initialization)和列表初始化(list initialization)，例如：

```
int year(2017);      //直接初始化
int year{2017};      //列表初始化，C++11
int year={2017};     //列表初始化，C++11
int year{};          //可以不提供初始值，默认为0，C++11
```

其中，列表初始化方式是在 C++11 新标准下引入的，已经得到了全面的应用。初始化是非常复杂的，将在后面继续讨论这个问题。

警告：使用未初始化的对象很危险

如果使用一个未初始化的对象，则会引起严重错误。比如，一个在函数内部创建的内置类型对象，如果没有提供初始值，那么它的值是未定义的随机值(见 5.2.2 节)。因此，定义一个对象时，应尽量为其提供初始值。

2.3.2 对象的声明

一个程序往往由多个代码文件组成，C++ 也支持分离式编译方式，即每个源文件可以单独编译。在多文件程序中，通常在一个文件里需要访问另外一个文件里面的对象。为了支持在程序文件之间共享代码，C++ 引入了声明(declaration)机制。对象的声明和定义有着本质的不同，定义一个对象编译器需要为其分配存储空间，但声明一个对象编译器并不会为其分配存储空间，只是通过声明表示这个对象已经定义过并告诉编译器对象的名字和数据类型。

声明对象，需要利用关键字 extern，而且不能提供初始值，形式如下：

```
int i(1);            //在一个源文件里定义对象 i 并初始化
extern int i;        //在另外一个文件里声明对象 i，而非定义 i，i 已被定义过
int j;               //可以理解为声明并定义 j
```

当然，如果利用 extern 声明对象时提供一个初始值，则声明就变成了定义：

```
extern int i=0;      //定义 i
int i=5;             //错误：对象 i 已经定义过
```

综上所述，可以得知一个对象只能被定义一次，但可以被声明多次。这在多文件中共享同一个对象是非常重要的。

2.3.3 作用域和生命期

每一个对象名字都对应唯一的内存空间，在代码中都有它的可见范围。作用域(scope)指定了每个名字在代码中的使用范围，通常以花括号分隔。同一个名字在不同的作用域下可能指向不同的内存空间。名字的作用域起始于它的声明处，结束于声明语句所在的域块的结束处，如以下代码。

例 2.1　把两个数的和保存到一个对象中并输出。

代码清单 2.1　例 2.1

```
1  #include<iostream>
2  using  namespace std;
3  int main() {
```

```
4       int sum = 0;      //用于存放两个数的和
5       {
6           int val1 = 10, val2 = 10;
7           sum = val1 + val2;
8       }
9       cout << sum;
10      return 0;
11  }
```

代码清单 2.1 定义了三个整型对象：sum、val1 和 val2。其中，sum 的作用域从第 4 行开始到函数体的结束处(第 11 行)结束。也就是说，在此之间的所有地方都可以访问 sum。类似地，val1 和 val2 的作用域从第 6 行开始，到其所在的语句块结束处结束(第 8 行)。在其作用域外访问 val1 和 val2 都是非法的，因为在 val1 和 val2 作用域外，它们要么还没有创建(第 6 行之前)，要么已经消亡了(第 8 行之后)。在这种情况下，称它们具有块域(block scope)或局部作用域。

通常，具有块域的对象的生命期从定义处开始，在作用域结束时消亡。一个对象的消亡意味着该对象生命期的结束，它在内存中占用的存储空间将被释放。当然，具有块域的对象的生命期结束的时刻还与它的数据类型有关。例如对于具有块域的静态对象，它们在程序结束时才消亡(见 5.2.2 节)。

如果一个块域包含了另外一个块域，则构成了作用域的嵌套。

例 2.2 作用域嵌套。

代码清单 2.2 例 2.2

```
1   #include <iostream>
2   int main() {
3       int sum = 10;
4       {
5           int sum = 0;
6           std::cout << sum << std::endl;        //访问内层 sum，打印输出 0
7       }
8       std::cout << sum << std::endl;            //访问外层 sum，打印输出 10
9       return 0;
10  }
```

在代码清单 2.2 中，先定义了一个对象 sum(第 3 行)，初始值为 10。该对象的作用域到第 10 行结束。然后，在该对象的作用域内定义一个语句块(第 4 ~ 7 行)，在该语句块内部又定义了一个名字为 sum 的对象(第 5 行)。内层对象 sum 的作用域和外层对象 sum 的作用域重叠。遇到这种情况，C++ 采用局部优先的原则，即外层对象的作用域被内层同名对象的作用域屏蔽。所以，第 6 行访问的是内层对象 sum 而非外层对象 sum。

建议：对象在使用的时候定义

一方面可以很容易找到对象的定义，另一方面也可以更合理地赋初始值。另外，内层对象的名字不要和外层对象的名字相同。

2.4 常量修饰符和类型推导

本节将介绍 const 修饰符、constexpr 和常量表达式、类型推导。

2.4.1 const 修饰符

数据存放到对象里，如果需要，则可以改变对象里的内容。但有时候，不希望对象的内容发生变化，比如对象里存放的是字面值常量。遇到这种情况时，可以利用关键字 const 对对象的类型加以限制，例如：

```
const double pi = 3.14159;          //圆周率
const int numStudent = 30;          //一个班级的学生人数
```

上面用两个字面值常量来初始化两个 const 对象，也意味着 pi 和 numStudent 为常量。在用一个常量初始化一个 const 对象时，编译器会把代码中用到这个常量的地方用它的值来替换，此时的常量也可以理解为其值的别名。所以在编译时，编译器会将代码中的 pi 用 3.14159 替换。

利用 const 修饰符定义的对象并不意味着该对象与常量总是等价的，例如：

```
int i = 100;
const int ci = i;       //利用对象 i 的值初始化 ci
```

利用一个非常量值来初始化 const 对象 ci，在编译时，编译器无法计算 ci 的值。因此，ci 只有在程序运行期间才体现常量特性。

任何试图对 const 修饰的对象进行写操作的行为都是非法的，例如：

```
numStudent = 50;      //错误：不能对 numStudent 进行写值操作
ci = 50;              //错误：不能对 const 对象 ci 进行写值操作
```

因为 const 对象一旦创建后，其值就不能再改变，因此 const 对象必须初始化。

```
const double pi;      //错误：const 对象必须初始化
```

const 修饰符不但提高了程序的安全性，避免了因为不小心而改变一些具有常量特性的对象的值，而且还改善了程序的可读性。例如，程序中只要用到了圆周率，只需要使用对象名 pi 就可以了。

2.4.2 constexpr 和常量表达式

常量表达式(const expression)是指值不会改变且在编译期间就能得到计算结果的表达式(表达式的概念将在 2.5 节中介绍)。利用常量表达式初始化的常量也称为编译时常量(compile-time constant)。例如，字面值是常量表达式，用字面值初始化的 const 对象也是常量表达式，但下面的 const 对象是非常量表达式：

```
int num = 100;
const int numStudent = num;
```

尽管 numStudent 是一个 const 对象，但它的值只有在程序运行期间可以获取，在编译期间不能得到，所以它不是常量表达式。这种情况在上一节中已经介绍过。

为了能让编译器更好地了解我们的意图，C++11 提供了 constexpr 关键字，用来帮助编译器自动识别常量表达式。与 const 类似，constexpr 修饰的对象是一个常量，而且必须用常量表达式初始化，例如：

```
constexpr int number = 10;                  //10 是常量表达式
constexpr int maxNumber = number + 1;       //number + 1 是常量表达式
constexpr int num = getNumber();            //是否合法取决于函数 getNumber 的属性
```

在上面第三条语句中，调用函数 getNumber 并用其返回值初始化 num。只有当 getNumber 是一个 constexpr 函数时，num 才是一个 constexpr 对象，否则会出现语法错误。在后续章节中将进一步讨论编译时常量及 constexpr 函数(见 5.5.4 节)。

2.4.3 类型推导

为了让代码更简洁，C++ 提供了一些类型处理机制。C++11 引入了 auto 和 decltype 两个关键字，使得 C++ 程序更现代化、智能化，大大提高了程序的开发效率。

1. 类型别名

类型别名(type alias)指把已经定义的数据类型换个新的名字，这样做的好处是使代码更易于理解和使用，能够帮助读者了解该类型的实际意义。定义类型别名的方法有两种：第一种是使用关键字 typedef，例如：

```
typedef double price;       //price 是 double 的一个类型别名
```

在程序中，price 和 double 具有同样的功能，都代表 double 数据类型，读者可以利用 price 来定义对象，例如：

```
price car = 1.0E5, mobile = 100.;     //car 和 mobile 存放的都是价格
```

通过使用 price 定义对象，可以很清楚地知道开发者的意图：price 类型对象用于存放价格信息。

第二种方法是使用 using 关键字来声明别名，例如：

```
using price = double;       //price 是 double 的一个类型别名
```

using 关键字紧跟别名和 = ，作用是把 = 左侧的名字声明为 = 右侧类型名的别名。

提示：推荐使用 using 声明

using 声明更符合编程习惯，其使用方式和定义对象的方式类似。

2. auto 类型推导

当定义一个对象时，需要先显式地告诉编译器需要的数据类型，然后再提供一个名字

和一个初始值。显式指明数据类型显然有些麻烦，因为提供的初始值的数据类型已经表明了用户的意图。为了改进该不足，C++11 将关键字 auto 赋予了新的含义，编译器利用它可以根据初始值的类型自动推导出需要的数据类型，例如：

```
auto pi = 3.14159, rad = 1.0;          //pi 和 rad 都为 double 类型
auto area = pi * rad * rad;            //area 为 double 类型
```

显然，如果要利用 auto 关键字将用户的意图告诉给编译器，则必须为对象提供一个初始值，而且初始值的类型必须符合用户意图：

```
auto i = 0, pi = 3.14159;          //错误：i 和 pi 的类型不一致
```

当初始值是一个 const 对象时，auto 将忽略 const 属性，例如：

```
const double pi = 3.14159;
auto rad = pi;          //rad 是一个 double 类型数，pi 的 const 属性被忽略
```

若希望编译器推断出 rad 具有 const 属性，则需要显式指出，例如：

```
const auto rad = pi;          //rad 是 const double 类型
```

注意：auto 使用说明

auto 的功能很强大，但不能肆意使用，否则会造成代码的可读性和可维护性下降，使用时需要权衡利弊。

3. decltype 关键字

auto 能够利用表达式的值推导出用户想要定义的对象的数据类型，并用表达式的值初始化定义的对象，但是有时只想用表达式的类型而不想用表达式的值来定义对象。为此，C++11 引入了 decltype 关键字，它能够在不用计算表达式的情况下获取表达式的数据类型，语法格式为 decltype (expr)，例如：

```
int i = 0;
decltype (i) j = 1;          //j 为 int 类型
decltype (i + j) k = 0;      //k 为 int 类型
```

decltype 分析 i + j 的值的数据类型，但不会计算 i + j 的值。

注意，当 decltype 遇到 const 时，和 auto 处理方式不同，它不会忽略 const 属性，例如：

```
const double pi = 3.14159;
decltype (pi) rad = 1.0;      //rad 为 const double 类型
```

2.5 表达式

表达式(expression)是指由运算符(operator)和操作对象(operand)组成的式子。字面值和对象是最简单的表达式。任何一个表达式都有一个确定数据类型的值。若要理解表达式

求值的过程，则需要知道运算符的优先级（precedence）、结合性（associativity）和求值次序（order of evaluation）。

2.5.1 基本知识

运算符是用来运算或处理对象的符号，参与运算的对象称为操作数。C++ 定义了一元（目）运算符（unary operator）、二元（目）运算符（binary operator）和三元（目）运算符。每一种运算符都有特定的含义和运算法则，运算符的目数决定了参与运算的操作数的个数，例如赋值运算符 = 为二元运算符，用于把 = 右边的表达式的值赋给 = 左边的对象。有些特殊的运算符在不同的环境下具有不同的功能。例如，符号 - 既可以是一元运算符，用于取反操作（比如 -4），也可以是二元运算符，执行减法运算（比如 4 -5）。

1. 左值和右值

任何一个表达式，要么是左值（lvalue），要么是右值（rvalue）。对于程序员来说，左值所在的内存空间的地址是可以获取（用取址符 & 获取）的，但右值的地址是无法得到的。因此，左值对象既可以读又可以写，而对右值对象只能进行读操作，不能对它进行写操作。显然，常量（如 'a'、10、3.14 等）都是右值，而由程序员定义的用来存放并能够改变值的对象是左值。一般来说，右值只能在 = 符号的右边，对左值没有限制，例如：

```
int i = 0;              //正确：用右值常量 0 初始化左值对象 i
10 = i;                 //错误，赋值运算符左侧必须为左值
int j = i;              //左值对象 i 可以当成右值，只对其内容进行读操作
const int N = 100;      //N 为右值对象
N = 40;                 //错误：不能改变右值对象 N 的值
```

一般来说，左值对象由程序员创建并命名，具有持久性。右值对象中除字面值常量外，都是临时对象（temporary object），它们大多是在执行运算的过程中由编译器创建的无名对象，运算完毕之后便消亡，没有持久性。

2. 优先级和结合性

含有多个运算符的表达式称为复合表达式（compound expression）。对于复合表达式求值，需要知道每一个运算符的优先级和结合性，才能确定它们的组合顺序。优先级高的运算符先运算，如乘法和除法运算符的优先级高于减法和加法运算符，即采用先乘除后加减的原则进行运算。例如，对于表达式 1 +2 * 3，有两个二元运算符，乘法的优先级高于加法，因此先计算 2 * 3，得到的值是 6，然后计算 1 +6，最终表达式的值为 7。结合性规定了在运算符优先级相同的情况下，是从左到右（左结合）计算还是从右到左（右结合）计算。例如，对于表达式 2 -1 -1，根据减法运算的左结合性，表达式的值是 0，而不是 2。C++ 运算符优先级表和结合性见附录 B。

建议：括号法求解复杂表达式的值

括号不受优先级和结合性约束。对于表达式 1 +4/(3 *(2 -1)),首先要计算括号里面的表达式的值,直到把所有的括号内的表达式都计算完,才考虑优先级和结合性。正因为如此,对于复杂的表达式求值,可以通过添加括号的方法来求解,例如,根据优先级和结合性,表达式 1 +4/3 *2 -1 与 (1 +((4/3) *2)) -1 等价。

2.5.2 算术运算符

在表 2.3 中,一元运算符的优先级最高,其次是乘法、除法和求余运算,减法和加法的优先级最低。所有算术运算符均为左结合性,即当运算符优先级相同时,表达式从左到右计算。算术运算符可以作用于任意算术类型或能转换为算术类型的类型,其运算结果是一个右值,保存在一个编译器创建的临时对象里。

表 2.3 算术运算符

运 算 符	功 能	用 法
+	一元正值运算	+5
-	一元负值运算	-5
*	乘法	4 *5
/	除法	5/3
%	求余或模运算	两个运算数必须是整型类型,如 5 % 3
+	加法	3 +2
-	减法	5 - 3

整数的算术运算的结果还是整数,例如:

```
21 / 6      //结果是 3,余数被舍弃
```

对于求余运算 %,要求两个操作数都是整数,计算被除数除以除数的余数。如果操作数中有负数,则运算的结果和被除数的符号一致,例如:

```
21 % -4     //结果为 1
-21 % 4     //结果为 -1
```

在表达式中,如果运算对象的数据类型不相同,编译器会自动进行类型转换,也称隐式类型转换,转换的规则是小数据类型向大数据类型转换。假设 i 是 int 类型,u 是 unsigned int 类型,f 是 float 类型,d 是 double 类型,如下表达式计算过程为:

```
'a' + 10 + u + d - i / f
```

根据优先级,先计算 i/f,将 i 转换成 float,得到的结果存放到一个 float 类型的临时对象 t1 中。根据左结合性,需要把字符 'a'进行整型提升①,转换成 int 类型,并和整型字面值 10 做加法运算,得到的结果存放到一个 int 类型的临时对象 t2 中。再计算 t2 + u,因为

① 对于 bool、char、unsigned char、short 和 unsigned short,只要它们有可能被保存为 int 类型,它们就会提升为 int 类型。

unsigned int 类型不小于 int，将 t2 转换成 unsigned 类型，相加的结果存放到一个 unsigned int 类型的临时对象 t3 中。然后计算 t3 + d，根据小数据类型向大数据类型转换的规则，将 t3 转换成 double 类型，结果存放到一个 double 类型的临时对象 t4 中。最后计算 t4 + t1，将 t1 转换成 double 类型，得到的最终结果存放在一个 double 类型的临时对象里，该临时对象是个右值。表达式运算完之后，其生命期结束。

警告：数据溢出

算术表达式有可能产生未定义的结果，例如：

```
int i = 32 / 0;          //除数为 0
short val = 32767;       //short 类型占 2 个字节，其表示的最大值为 32767
val += 1;                //val 本身加 1，结果超出其表示范围，出现数据溢出
std::cout << val;
```

在我们的测试环境下，输出结果为 -32768。在其他系统下，可能会出现不同结果，甚至崩溃。

2.5.3 赋值运算符

赋值运算符 = 的功能是，把赋值符号右侧表达式的值写入赋值符号左侧的操作对象里，所以赋值运算符的左侧操作对象必须是一个支持写操作的左值。运算的结果是左侧操作对象本身，且是左值，表达式的值是左侧对象的值。在 2.3.1 节中已经强调了赋值操作和初始化操作的区别：

```
int i = 0, j = i;        //初始化而非赋值
i = 0;                   //赋值而非初始化
i + j = 10;              //错误：算术表达式为右值
```

如果左、右两侧的操作对象的类型不相同，则编译器自动把右侧对象的值转换为左侧对象的类型，例如：

```
int i = 0;
double d = 3.14159;
i = d;                   //表达式的结果的类型是 int，值是 3
```

注意在转换的过程中，右侧对象本身并不会发生任何变化，d 的值还是 3.14159。C++11 允许用列表的值作为右操作数，但是如果列表的值转换后存在信息丢失的风险，编译器会以错误来处理，例如：

```
i = {3.14159};           //错误：窄化转换
```

赋值运算满足右结合性，例如：

```
i = j = 5;               //i 和 j 的值都是 5
```

根据右结合性，先把 5 赋给 j，然后把 j = 5 的值赋给 i。

因为赋值运算符的优先级是比较低的，所以通常需要加上括号才能正确执行用户的意图，例如：

```
i = 2 + j = 4;        //错误：2 + j 为右值，不能作为第 2 个赋值符号的左侧对象
i = 2 + (j = 4);      //正确：把 j = 4 的值加上 2 赋给 i，i 和 j 的值分别为 6 和 4
```

如果把一个左值对象经过简单算术运算之后的结果存放到这个左值对象里面，比如定义一个 int 类型对象 counter 用来计数操作，代码如下：

```
counter = counter + 1;
```

可以把二元算术运算符和赋值运算符组合成一个复合赋值运算符，形式如下：

```
+=     -=     /=     *=     %=
```

例如：

```
counter += 1;      //等效于 counter = counter + 1;
i *= j + 3;        //等效于 i = i * (j + 3);
```

复合赋值运算符仍然属于赋值运算符，因此上面的两个表达式均为赋值表达式。使用复合赋值运算符能够提高运算效率。比如，上面第一个复合赋值表达式在计算过程中直接在 counter 上操作，不会像右边的表达式一样产生临时对象。

2.5.4 自增和自减运算符

为了在书写上简化一个整型对象 i 的自增 1(i += 1)和自减 1(i -= 1)操作，C++ 提供了自增(++)和自减(--)运算符，用来对整型操作对象的当前值加 1 或减 1。自增、自减运算符分为前置和后置两种形式，例如：

```
int i = 0, j;
j = i++;        //后置，i 的值自增变为 1，表达式 i++ 的值为 i 自增之前的值，即 j 的值为 0
j = ++i;        //前置，i 的值自增变为 2，表达式 ++i 的值为 i 自增之后的值，即 j 的值为 2
```

注意，自增、自减运算符的操作对象必须为左值。前置版本返回左值对象本身，后置版本将原始值的副本作为右值返回。

建议：能用前置版本不用后置版本

前置版本避免了额外的运算代价，直接返回操作对象本身，而后置版本编译器需要产生一个临时对象来保存变化之前的值。

2.5.5 逻辑和关系运算符

表 2.4 列出了逻辑和关系运算符，以及它们的结合性和相关语义。逻辑运算符和关系运算符返回值都为 bool 类型。运算对象的值为 0 表示 false(假)，否则表示 true(真)，例如 5 和 -5 都为真。

在这些运算符中，逻辑非的优先级最高，其次是关系运算符，然后是逻辑与，逻辑或的优先级最低。在关系运算符中，< 、<= 、> 和 >= 的优先级高于 == 和 != 。

只有逻辑非运算符是右结合，其余都是左结合。这些运算符的运算结果都为右值。

表 2.4　逻辑和关系运算符，以及它们的结合性和相关语义

运算符	功能	结合性	用法	语　义
!	逻辑非	右	! 5	操作数的值为真，结果为假；反之结果为真
<	小于	左	5 < 4	左操作数小于右操作数，结果为真；反之为假
<=	小于等于	左	5 <= 4	左操作数小于或等于右操作数，结果为真；反之为假
>	大于	左	5 > 4	左操作数大于右操作数，结果为真；反之为假
>=	大于等于	左	5 >= 4	左操作数大于或等于右操作数，结果为真；反之为假
==	等于	左	5 == 4	左操作数等于右操作数，结果为真；反之为假
!=	不等于	左	5 != 4	左操作数不等于右操作数，结果为真；反之为假
&&	逻辑与	左	5 && 4	两个操作数的值都为真，结果为真；反之为假
||	逻辑或	左	5 || 4	两个操作数的值有一个或全为真，结果为真；反之为假

1. 表达式构造

我们经常会有这样的需求：只有当某个条件满足时，我们才去做一件事情。假如，这个条件是 $i \leqslant j \leqslant k$ 。如果构造出如下表达式：

```
i <= j <= k
```

仔细分析上面的表达式，根据左结合，需要先计算关系表达式 i <= j 的值，i <= j 的值要么为 1（真），要么为 0（假）。那么只要 k 大于等于 1，表达式的值永远为真。所以正确的表达式应该是：

```
i <= j && j <= k
```

2. 短路求值

逻辑与和逻辑或运算符都是先计算左侧对象的值，然后根据左侧对象的值判断是否计算右侧运算对象的值，规则如下：

- 对于逻辑与运算符来说，仅当左侧运算对象的值为真时，才计算右侧运算对象的值；
- 对于逻辑或运算符来说，仅当左侧运算对象的值为假时，才计算右侧运算对象的值。

这种策略称为短路求值（short – circuit evaluation），例如：

```
int i = 1, j = 2;
bool b = ! i && ++ j;          //b 的值是 0，j 的值是 2
```

根据运算符的优先级，表达式 !i && ++ j 等价于（!i）&&（++ j）。由于 i 的值为真（非 0），所以 !i 的值为假。根据逻辑与的短路求值策略，编译器不会计算逻辑与右侧的表达式的值，因此，j 的值依然为 2。最终，b 的值为 0。

警告：赋值运算符不是等号运算符

对于初学者来说，切勿将赋值运算符当成等号运算符 ==，判断两个对象的值是否相等应该是 i==j 而不是 i=j。

2.5.6 逗号运算符

用逗号运算符(comma operator)连接起来的表达式称为逗号表达式，格式如下：

```
exp1, exp2,...
```

逗号表达式的求值方法为依次从左向右计算每个运算对象，表达式的结果为最右边的运算对象。例如：

```
int i, j;
i = (j =3, j +=6, 5 +6);        //i 的值为 11, j 的值为 9
```

赋值运算符右侧为一个逗号表达式，从左向右依次运算里面的每一个表达式，最终得到 i 的值为 11，j 的值为 9。

在所有的运算符当中，逗号运算符的优先级最低。

2.5.7 条件运算符

条件运算符(?:)是唯一的一个三目运算符，格式如下：

```
cond ? expr1:expr2
```

其中，cond 一般是条件判断表达式。运算过程为：首先计算 cond 表达式的值，如果值为真则运算 expr1，并返回 expr1 的值；否则运算 expr2，并返回 expr2 的值。如果两个表达式都为左值或能转换为同一类型左值，则条件表达式为左值，否则为右值。

条件运算符的优先级低于逻辑或运算符，高于赋值运算符，且为右结合性。

条件运算符允许嵌套使用，例如：求三个整型数 a，b，c 中的最大值。

```
int a =4, b =5, c =6, max;
max = a > b ? (a > c ? a:c):(b > c ? b:c);
```

上面的赋值语句把三个对象中值最大的赋给 max。

建议：条件运算符不宜嵌套使用

嵌套条件运算符虽然能减少代码的书写，但代码的可读性急剧下降，不提倡使用嵌套条件运算符。

2.5.8 sizeof 运算符

sizeof 运算符返回一个表达式或一个类型所占内存的字节数。一般格式为：

```
sizeof (type) 或
sizeof (expr)
```

例如：

```
cout << sizeof (int);       //输出 4
int i=0;
cout << sizeof (++i);       //输出 4，i 的值为 0
```

注意，sizeof(expr) 形式只是返回表达式结果的数据类型的字节数，并不会实际运算表达式，因此上面执行完第 2 个输出语句后，i 的值仍然为 0。

2.5.9 位运算符

位运算符包括 ~（按位取反）、<<（左移）、>>（右移）、&（位与）、|（位或）和^（位异或），其中 ~ 为一元运算符，其余都为二元运算符。运算的对象是整型对象，用来处理二进制数，在运算过程中不会改变操作对象本身的值，操作的结果为右值。例如：

```
short a=3, b=5;
```

假设 short 类型占用 16 位，每一种位运算操作的结果如下：

```
b   ~ 00000000 00000101     <<1 00000000 00000101     >>1 00000000 00000101
      11111111 11111010           00000000 00001010           00000000 00000010

a     00000000 00000011           00000000 00000011           00000000 00000011
b   & 00000000 00000101         | 00000000 00000101         ^ 00000000 00000101
      00000000 00000001           00000000 00000111           00000000 00000110
```

对于位移运算符，移动的位数不能为负，且其值必须小于所得结果的位数。操作过程中，移动到边界外面的二进制数被舍弃。左移运算符在右侧插入 0。右移运算符在左侧插入的值取决于操作对象的数据类型，如果是无符号数，则左侧插入 0；如果是有符号数，则左侧插入符号位的副本。上面对 b 进行左移 1 位和右移 1 位的结果分别为：b << 1 的值为 10，b >> 1 的值为 2。

按位取反运算符(~)将运算对象逐位取反后得到一个新的值，即将 1 置为 0，0 置为 1。上面对 b 进行按位取反(~b)的结果为 -6。位与(&)、位或(|)与位异或(^)运算符按照相应的逻辑对两个操作数逐位进行运算。位与运算的规则为：两个操作数对应位都为 1 则结果中对应位为 1，否则为 0。对于位或运算，两个操作数对应位都为 0 则结果中对应位为 0，否则为 1。对于位异或运算，两个操作数对应位相同则结果中对应位为 0，否则为 1。

警告：位运算符不宜处理带符号数

位运算符虽然可以处理带符号数，但没有规定如何处理符号位，所以位运算符不宜处理带符号数。

2.5.10 求值次序

表达式求值除和运算符的优先级和结合性相关外，还与求值次序相关。求值次序与结合性无关，例如加法运算符并没有规定是先运算左操作对象还是右操作对象，假如有：

```
int i = 0, j;
j = i*2 + i ++;
```

优先级表明 i 与 2 相乘，结合性表明 i * 2 的结果再加上 i ++，但无法推断编译器是先计算 i ++ 还是 i * 2，程序的行为是未定义的。如果先计算表达式 i * 2 再计算表达式 i ++，那么结果为 0；反过来，结果则为 2。在 Visual C++ 编译器下面，j 的值为 0，但在 GCC 编译器下面，j 的值为 2。

在附录 B 列出的运算符中，C++ 只规定了四种运算符的求值次序：逻辑或、逻辑与、逗号和条件运算符，具体规则在前面已经介绍过。

建议：不要对表达式中同一个对象进行既读又写的操作

在复合表达式书写过程中，里面的每个子表达式的求值应互相无关，不要出现对同一个对象既读又写的情况，否则结果可能是未知的。因此，可以认为这样的表达式是错误的。

2.6 类型转换

在表达式运算过程中，一般会发生类型转换(type conversion)。C++ 中的类型转换有两种形式：一种是隐式的，另一种是显式的。隐式类型转换由编译器根据需要自动进行。显式类型转换由程序员根据需要来进行。

2.6.1 隐式类型转换

如果同一个表达式中的运算对象的类型不一致，那么编译器首先会尝试将它们转换为相同的类型，如果它们之间具有关联性并可以相互转换，则会自动发生隐式类型转换。前面已经提到了一些隐式类型转换的情形，总结如下：

- 一般情况下，比 int 类型小的整型类型提升为较大的整型类型。比如对于 'a' + 1 和 2U + 5，字符 'a'要转换成与其 ASCII 值相对应的整型数，有符号整型数 5 转化为无符号整型数。类似的还有浮点数提升(floating point promotion)，即 float 类型转换为 double 类型。
- 表达式的值需要转换为布尔值。将非布尔值转换为布尔值，如在条件表达式里的第一个表达式。
- 在初始化过程中，初始值转换成定义对象的类型。比如，对于 int i = 3.14 将 double

类型的 3.14 转换为 int；在赋值语句中，把赋值符号右侧运算对象转换成左侧运算对象的类型。

- 在算术表达式中，运算结果转换为运算对象中最宽(大)的数据类型。比如，将 int 转换为 float，float 转换为 double 等。

2.6.2 显式类型转换

除编译器的隐式类型转换外，也可以将一个对象显式地转换为另一种相关联的类型，这种方法也称为强制类型转换。C++ 提供了四种强制类型转换方式，分别为 static_cast、dynamic_cast、const_cast 和 reinterpret_cast，格式如下：

```
cast - name < type > (expr)
```

在算术表达式中，常用 static_cast 执行以下两种操作。

- 执行浮点数操作，例如：

```
int i = 5, j = 3;
double k = i / static_cast < double > (j);        //强制将 j 转化为 double 类型
```

- 告诉编译器用户有意将宽类型转换成窄类型，请关闭警告信息，例如：

```
double i = 5., j = 3.;
int k = static_cast < int > (i / j);        //强制将 i / j 的结果转化为 int
```

const_cast 常用来去掉对象的 const 属性，即把 const 对象转换为非 const 对象。将一个 const 对象移除其 const 属性后，编译器将不会抱怨对该对象进行写操作。在程序中，一般不提倡使用 const_cast 和 reinterpret_cast，除非无路可走，因此本书不再介绍它们。对于 dynamic_cast，将在 12.5.1 节中介绍其用法。

在早期版本的 C++ 代码中，会看到如下格式的强制类型转换：

```
type (expr)        //函数方式，或者
(type) expr        //C 语言方式
```

例如：

```
double k = i / (double)j;            //强制将 j 转化为 double 类型
double k = i / double (j);
```

提示：类型转换不会改变对象本身的值

无论是隐式类型转换还是强制类型转换，操作对象本身的值不会受到影响。

习题 2

2.1 C++ 中有哪几种基本的数据类型？确定一个数据的类型有什么作用？

2.2 一个对象的作用域和生命期指的是什么?

2.3 一个表达式由什么组成?影响表达式求值的因素有哪些?

2.4 什么是左值和右值?

2.5 在 C++ 程序中,什么情况下会对数据进行类型转换?转换的方式有哪些?

2.6 下列哪些是合法的用户自定义标识符?

① begin ② $ amount ③ new ④ _1first ⑤ Salary 94

⑥ file_name ⑦ struct ⑧ Salary94 ⑨ _while ⑩ number3.5

2.7 下列哪些是 C++ 语言中的合法常量?

① 3.14e1L ② 100L ③ 0237 ④ 'abc' ⑤ "A"

⑥ "ABC" ⑦ 0xABCD ⑧ '\581' ⑨ 1.43E3.5 ⑩ 'x0H'

2.8 'b'、\142 和\x62 分别代表什么?其 ASCII 值是多少?如何输出 " 和 '?

2.9 判断下列定义中 auto 和 decltype 推断出的类型是什么?

```
const  int i=42;
auto j=i;
decltype (i) j2=i;
int x=0;
auto j3=x;
decltype (x) j4=i;
```

2.10 分别根据以下已知条件,求下列表达式的值:

(1)
```
float x=2.5, y=4.7, c=3.5, d=2.5;
int a=2, b=3;
(int)(x+y) / 24 + (float)(a+ b) / 2 + (int)c / (int)d
```

(2)
```
charCh1 = 'a', ch2 = '5', ch3 = '0', ch4;
ch4 = Ch3 - ch2 + ch1;
```

(3)
```
int x=3, y=5, z=1, a=0, b=0, c=0, d;
d = (x+ z > y) + (x < y==y < z) + (a || (b+=5) || (c -=3))
```

(4)
```
int a=10, b=20, c=30;
float x=1.2, y=2.1;
a < b && x > y ||a < b - !c
```

(5)
```
int i=5, j=5, m, n;
m=i++;
n= ++j;
```

2.11 试根据 C++ 语言中运算符的优先级和结合性,给下列表达式添加括号而不改变其求值结果:

(1)
```
a=b+c*d < 2 && 8
```

(2)
```
a && 077 !=3
```

(3)
```
a==b ||a==c && c < 5
```

(4)
```
c=x !=0
```

(5)a < b==c==d

2.12　将下列算式或叙述用 C++ 表达式描述。

(1)|x| > 1。

(2)a 和 b 之一为 0，但不同时为 0。

(3)位于圆心在原点，内外半径分别为 a 和 b 的圆环中的点。

2.13　参考代码清单 1.2，编写程序。要求输入长方体的长、宽、高，计算并输出长方体的体积和表面积。