第三章 程序设计基础

模块3.2：处理数据

主讲教师：同济大学计算机科学与技术学院 陈宇飞
同济大学计算机科学与技术学院 龚晓亮

目录

- 简单变量
- const限定符
- 浮点数
- C++算术运算符

目录

- 简单变量
  - 变量名
  - 整型
  - 整型short、int、long和long long
  - 无符号类型
  - 选择整型类型
  - 整型字面值
  - C++如何确定常量的类型
  - char类型：字符和小整数
  - bool类型

1.1 变量名

- 变量：在程序运行中，值能够改变的量（有名字、值）

- 标识符：用来标识变量名、符号常量名、函数名、数组名、结构体名、类名等的有效字符序列，称为标识符（变量名）

1.1 变量名

C++的标识符命名规则：

✓C++提倡使用有一定含义的变量名。**cost_of_trip**或**costOfTrip**

✓在名称中智能使用字母字符、数字和下划线（_）

✓名称的第一个字符不能是数字

✓区分大写字符与小写字符

✓不能将C++关键字用作名称

1.1 变量名

C++的标识符命名规则：

✓以两个下划线打头或以下划线和大写字母打头的名称被保留给现实（编译器及其使用的资源）使用

✓以一个下划线开头的名称被保留给现实，用作全局标识符

✓C++对于名称的长度没有限制，名称中所有的字符都有意义，但有些平台有长度限制

变量	是否合法	备注
int poodle;		
int Poodle;		
int POODLE;		
Int terrier;		
int my_stars3;		
int _Mystars3;		
int 4ever;		
int double;		
int begin;		
int __fools;		
int the_very_best_variable_i_can_be_version_112;		
int honky-tonk;		

变量	是否合法	备注
int poodle;	//valid	
int Poodle;	//valid	distinct from poodle
int POODLE;	//valid	even more distinct
Int terrier;	//invalid	has to be int,not Int
int my_stars3;	//valid	
int _Mystars3;	//valid	but reserved-starts with underscore
int 4ever;	//invalid	starts with a digit
int double;	//invalid	double is a C++ keyword
int begin;	//valid	begin is a Pascal keyword
int __fools;	//valid	but reserved-starts with 2 underscores
int the_very_best_variable_i_can_be_version_112;	//valid	
int honky-tonk;	//invalid	no hyphens allowed

1.1 变量名

✓标识符区分大小写(也称为大小写敏感)

✓一般长度<=32(或64，每种编译器还有参数可设，不再讨论)

✓取名时通常按变量的含义

✓必须先定义、后使用

(某些语言不定义可直接使用，称为弱类型，因此对应也称为强类型)

✓同级不能同名(不同级的同名问题后续讨论)

1.2 整型

✓C++提供好几种整型，能够根据程序的具体要求选择最适合的整型

✓不同C++整型使用不同的内存量来存储整数

✓使用的内存量越大，可以表示的整数值范围越大

✓C++的基本整型（按宽度递增的顺序排列）：

char、short、int、long、long long

1.3 整型short、int、long和long long

✓C++对整型short、int、long和long long的长度规则定义如下：

- short　　至少16位
- int　　至少和short一样长
- long　　至少32位，且至少和int一样长
- long long 至少64位，且至少和long一样长

1.3 整型short、int、long和long long

1）运算符sizeof和头文件limits

✓对类型名（如int）使用sizeof运算符时，应将名称放在括号中；但对变量名（如n_short）使用该运算符，括号是可选的

```
cout << sizeof(int) << endl;

cout << sizeof n_short << endl;
```

✓头文件climits定义了符号常量来表示类型限制

```
#define INT_MAX 32767
```

1.3 整型short、int、long和long long

1）运算符sizeof和头文件limits

表 3.1 climits 中的符号常量

符号常量	表示
CHAR_BIT	char 的位数
CHAR_MAX	char 的最大值
CHAR_MIN	char 的最小值
SCHAR_MAX	signed char 的最大值
SCHAR_MIN	signed char 的最小值
UCHAR_MAX	unsigned char 的最大值
SHRT_MAX	short 的最大值
SHRT_MIN	short 的最小值
USHRT_MAX	unsigned short 的最大值

1.3 整型short、int、long和long long

1）运算符sizeof和头文件limits

续表

符号常量	表示
INT_MAX	int 的最大值
INT_MIN	int 的最小值
UNIT_MAX	unsigned int 的最大值
LONG_MAX	long 的最大值
LONG_MIN	long 的最小值
ULONG_MAX	unsigned long 的最大值
LLONG_MAX	long long 的最大值
LLONG_MIN	long long 的最小值
ULLONG_MAX	unsigned long long 的最大值

```
// limits.cpp -- some integer limits
#include <iostream>
#include <climits>                              // use limits.h for older systems
int main()
{
    using namespace std;
    int n_int = INT_MAX;                        // initialize n_int to max int value
    short n_short = SHRT_MAX;                   // symbols defined in climits file
    long n_long = LONG_MAX;
    long long n_llong = LLONG_MAX;
// sizeof operator yields size of type or of variable
    cout << "int is " << sizeof (int) << " bytes." << endl;
    cout << "short is " << sizeof n_short << " bytes." << endl;
    cout << "long is " << sizeof n_long << " bytes." << endl;
    cout << "long long is " << sizeof n_llong << " bytes." << endl;
    cout << endl;
    cout << "Maximum values:" << endl;
    cout << "int: " << n_int << endl;
    cout << "short: " << n_short << endl;
    cout << "long: " << n_long << endl;
    cout << "long long: " << n_llong << endl << endl;
    cout << "Minimum int value = " << INT_MIN << endl;
    cout << "Bits per byte = " << CHAR_BIT << endl;
    return 0;
}
```

```
int is 4 bytes.
short is 2 bytes.
long is 4 bytes.
long long is 8 bytes.

Maximum values:
int: 2147483647
short: 32767
long: 2147483647
long long: 9223372036854775807

Minimum int value = -2147483648
Bits per byte = 8
```

1.3 整型short、int、long和long long

2）初始化

✓初始化将赋值与声明合并在一起

✓可以使用字面值常量来初始化

```
int uncles = 5;
```

✓可以将变量初始化为另一个变量，条件是后者已经定义过

```
int aunts = uncles;
```

✓可以使用表达式来初始化变量，条件是当程序执行到该声明时，表达式中所有值都是已知的

```
int chairs = aunts + uncles + 4;
```

1.3 整型short、int、long和long long

2）初始化

警告：如果不对函数内部定义的变量进行初始化，该变量的值将是不确定的。该变量的值将是它被创建之前，相应内存单元保存的值

在声明变量时对它进行初始化，可避免以后忘记给它赋值！！！

1.3 整型short、int、long和long long

2）初始化

- int n_int = INT_MAX; // initialize n_int to max
- int uncles = 5; //initialize uncles to 5
- int aunts = uncles; //initialize aunts to 5
- int chairs = aunts + uncles + 4;//initialize chairs to 14
- int owls = 101; //traditional C initialization, sets owls to 101
- int wres(432); //alternative C++ syntax, set wrens to 432

1.3 整型short、int、long和long long

3）C++11初始化方式

将{ }初始化器用于任何类型初始化（可以使用等号，也可以不使用），可以单值变量，也可以用于数组和结构体

```
❖int emus{7};          // set emus to 7
❖int rheas = {12};     // set rheas to 12
❖int rocs = {};       //set rocs to 0
❖int psychics{};    // set psychics to 0
```

1.4 无符号类型

✓整型中无符号类型不能存储负数值，优点是可以增大变量存储的最大值

```
unsigned short change; // unsigned short type
unsigned int rovert;   // unsigned short type
unsigned quarterback;  // also unsigned int type
unsigned long gone;    // unsigned long type
unsigned long long lang_lang;// unsigned long long type
```

```
// exceed.cpp -- exceeding some integer limits
#include <iostream>
#define ZERO 0      // makes ZERO symbol for 0 value
#include <climits>  // defines INT_MAX as largest int
int main()
{   using namespace std;
    short sam = SHRT_MAX;    // initialize a variable to max value
    unsigned short sue = sam;// okay if variable sam already defined
    cout << "Sam has " << sam << " dollars and Sue has " << sue;
    cout << " dollars deposited." << endl << "Add $1 to each account." << endl << "Now ";
    sam = sam + 1;
    sue = sue + 1;
    cout << "Sam has " << sam << " dollars and Sue has " << sue;
    cout << " dollars deposited.\nPoor Sam!" << endl;
    sam = ZERO;
    sue = ZERO;
    cout << "Sam has " << sam << " dollars and Sue has " << sue;
    cout << " dollars deposited." << endl;
    cout << "Take $1 from each account." << endl << "Now ";
    sam = sam - 1;
    sue = sue - 1;
    cout << "Sam has " << sam << " dollars and Sue has " << sue;
    cout << " dollars deposited." << endl << "Lucky Sue!" << endl;
    return 0;
}
```

```
Sam has 32767 dollars and Sue has 32767 dollars deposited.
Add $1 to each account.
Now Sam has -32768 dollars and Sue has 32768 dollars deposited.
Poor Sam!
Sam has 0 dollars and Sue has 0 dollars deposited.
Take $1 from each account.
Now Sam has -1 dollars and Sue has 65535 dollars deposited.
Lucky Sue!
```

1.4 无符号类型

```
short a=32767, b=a+1;
a= 0111111111111111 = 32767
b= 1000000000000000 = -32768
```

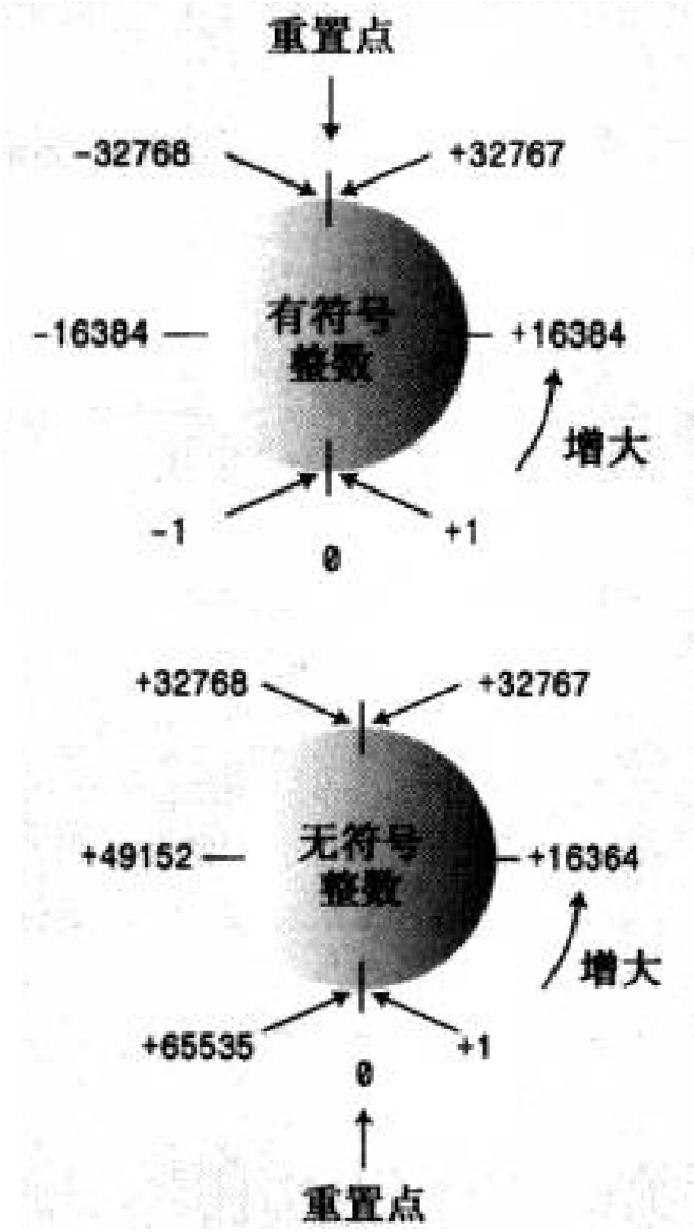
```
short a=-32768, b=a-1;
a= 1000000000000000 = -32768
b= 0111111111111111 = 32767
```

```
unsigned short a=65535, b=a+1;
a=   1111111111111111 = 65535
b= 1 0000000000000000 = 0
```

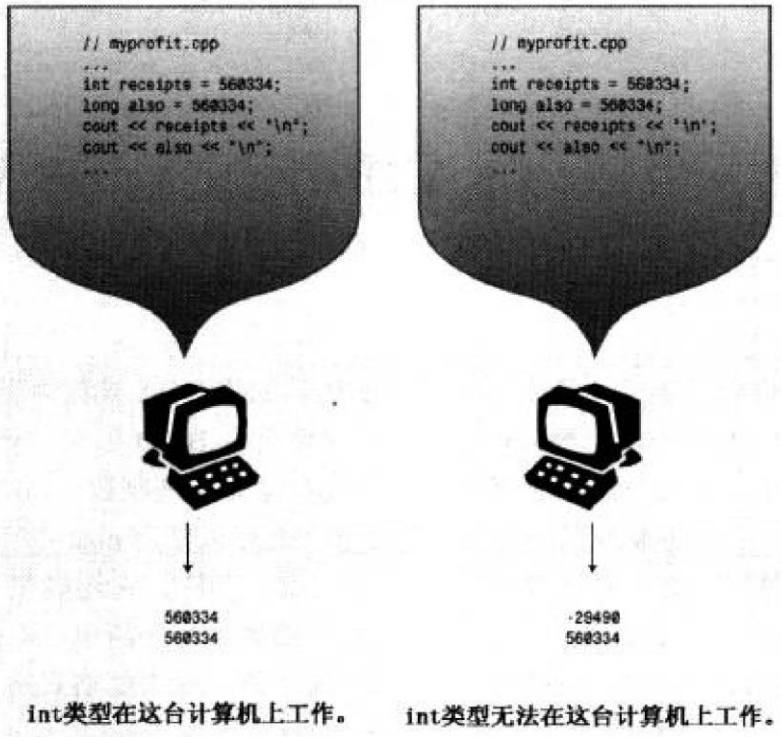
（高位溢出，舍去）

```
unsigned short a=0, b=a-1;
a= 0000000000000000 = 0
b= 1111111111111111 = 65535
```

（借位不够，虚借一位）



典型的整型溢出行为



为了提高可移植性，请使用long

1.5 选择整型类型

- ✓ int为计算机处理效率最高的长度。如果没有非常说服力的理由来选择其他类型，则应使用int
- ✓ 如果变量的值不可能为负数，则可以使用无符号类型
- ✓ 如果知道变量可能表示的整数值大于16位，则应使用long，即使系统上的int类型为32位，也应该这样做，这样程序在移值到16位的系统上时，就不会突然无法正常工作
- ✓ 如果要存储的值超过20亿，可以使用long long
- ✓ 如果short比int小，则可以使用short来节省内存，通常，仅当有大型整型数组时，才有必要使用short。如果节省内存很重要，则应使用short而不是使用int，即使它们的长度是一样的

1.6 整型字面值

✓整型字面值（常量）是显式的书写的常量

如：212或1776

✓和C相同，C++能够以三种不同的基数方式来书写整数：

基数为10

基数为8（老式UNIX版本）

基数为16（硬件黑客的最爱）

1.6 整型字面值

✓C++使用前一（两）位来标识数字常量的基数

- ❖如果第一位为1-9，则基数为10(十进制)
- ❖如果第一位是0，第二位为1-7，则基数是8(八进制)
- ❖如果前两位为0x或者0X，则基数为16（十六进制）
- ❖不管值书写为10、012、0xA，都将以相同的方式存储在计算机中-二进制（2为基数）

1.6 整型字面值

✓C++中cout默认是十进制输出

✓如果要以十六进制或者八进制方式显示值，可以使用cout的特殊特性

✓iostream中提供了控制符dec、hex和oct，分别用于cout以十进制、十六进制和八进制格式显示整数

1.6 整型字面值

```
// hexoct1.cpp -- shows hex and octal literals
#include <iostream>
int main()
{
    using namespace std;
    int chest = 42;      // decimal integer literal
    int waist = 0x42;    // hexadecimal integer literal
    int inseam = 042;    // octal integer literal

    cout << "Monsieur cuts a striking figure!\n";
    cout << "chest = " << chest << " (42 in decimal)\n";
    cout << "waist = " << waist << " (0x42 in hex)\n";
    cout << "inseam = " << inseam << " (042 in octal)\n";
    return 0;
}
```

```
Monsieur cuts a striking figure!
chest = 42 (42 in decimal)
waist = 66 (0x42 in hex)
inseam = 34 (042 in octal)
```

```
// hexoct2.cpp -- display values in hex and octal
#include <iostream>
using namespace std;
int main()
{
    using namespace std;
    int chest = 42;
    int waist = 42;
    int inseam = 42;

    cout << "Monsieur cuts a striking figure!" << endl;
    cout << "chest = " << chest << " (decimal for 42)" << endl;
    cout << hex;       // manipulator for changing number base
    cout << "waist = " << waist << " (hexadecimal for 42)" << endl;
    cout << oct;       // manipulator for changing number base
    cout << "inseam = " << inseam << " (octal for 42)" << endl;
    return 0;
}
```

```
Monsieur cuts a striking figure!
chest = 42 (decimal for 42)
waist = 2a (hexadecimal for 42)
inseam = 52 (octal for 42)
```

1.7 C++如何确定常量的类型

```
cout << "my salary is:" << 10000 << endl;
```

10000是什么类型？

- ✓ 除非有理由存储为其他类型（例如使用了特殊后缀来表示特定类型，或者值太大，不能存储为int整型），否则C++将常量存储为int整型。

1.7 C++如何确定常量的类型

✓后缀

后缀是放在常量后面的字母，用于表示类型

- l或者L用来表示整数为long型常量
- u或者U后缀表示unsigned int常量
- ul表示unsigned long常量（由于小写的l看起来像数字1，因此建议写后缀的时候使用大写字母L）
- ll、LL、ull、Ull、uLL和ULL

1.7 C++如何确定常量的类型

✓长度

❖在C++中，对十进制采用的规则与十六进制略有不同

❖对于不带后缀的十进制整数：将使用下面几种数据类型中能够存储该数的最小类型：int, long ,long long

❖对于不带后缀的十六进制或八进制整数：将使用下面几种类型中能够存储该数的最小类型：

int, unsigned int，long, unsigned long, long long

1.7 C++如何确定常量的类型

✓长度

例如：在int为16位，long为32位的机器中：

❖十进制数字 20000 被表示为int类型

40000 被表示为long类型

3000000000 被表示为long long类型

❖十六进制 0X9C40 (40000)被表示为unsigned int

这是因为十六进制常用来表示内存地址，而内存地址是没有符号的，因此，unsigned int比long更适合用来表示十六位的地址。

1.8 char类型：字符和小整数

✓char类型是专门为存储字符（数字和字母）设计的

✓编程语言通过字母的数值编码解决存储字母。因此，char是另外一种整型，能够表示目标计算机系统中所有基本符号-所有的字母、数字、标点符号等，很多系统的字母大都不超过128个

✓char常用来处理字符，但也可以当做比short更小的整型来使用，以节约空间

1.8 char类型：字符和小整数

✓最常用的符号集是ASCII字符集（必会！详见附录C ASCII字符集，但不用刻意背）

✓C++实现使用的是其主机系统的编码-如IBM大型机使用EBCDIC编码

✓C++支持的宽字符类型（wchar_t）可以存储更多的值，如国际Unicode字符集使用的值（了解即可）

1.8 char类型：字符和小整数

```
// chartype.cpp -- the char type
#include <iostream>
int main()
{
    using namespace std;
    char ch;          // declare a char variable

    cout << "Enter a character: " << endl;
    cin >> ch;
    cout << "Hola! ";
    cout << "Thank you for the " << ch << " character." << endl;

    return 0;
}
```

```
Enter a character:
m
Hola! Thank you for the m character.
```

为什么输入是m，输出的时候不是数字77，而直接由数值转换成了字符？

```
// morechar.cpp -- the char type and int type contrasted
#include <iostream>
int main()
{
    using namespace std;
    char ch = 'M';          // assign ASCII code for M to ch
    int i = ch;             // store same code in an int
    cout << "The ASCII code for " << ch << " is " << i << endl;
    cout << "Add one to the character code:" << endl;
    ch = ch + 1;            // change character code in ch
    i = ch;                 // save new character code in i
    cout << "The ASCII code for " << ch << " is " << i << endl;
    // using the cout.put() member function to display a char
    cout << "Displaying char ch using cout.put(ch): ";
    cout.put(ch);
    // using cout.put() to display a char constant
    cout.put('!');
    cout << endl << "Done" << endl;
    return 0;
}
```

```
The ASCII code for M is 77
Add one to the character code:
The ASCII code for N is 78
Displaying char ch using cout.put(ch): N!
Done
```

- C++中书写字符字面值：将字符用单引号，对字符串使用双引号
- ch实际上是一个整数，因此可以使用整数操作
- cout.put()成员函数，可以显示一个字符常量和变量ch

1.8 char类型：字符和小整数

```
char ch;

cin >> ch;
```

读取字符"5"，并将其对应字符编码（ASCII码53）存储到变量ch中

请问都输入5并回车，实际存储到变量中的值是几？

```
int n;

cin >> n;
```

读取字符"5"，并将其转换为相应的数字值5，并存储到变量中

1.8 char类型：字符和小整数

✓在C++中，书写字符常量最简单的方法是将字符用单引号括起来

```
char ch = 'M';
```

✓转义序列的编码（控制字符中，除空格外，都不能直接表示，\ ’ ” 等特殊图形字符也不能直接表示）

- ❖\a 表示振铃
- ❖\n 表示换行符
- ❖\”表示双引号（而不是字符串分隔符）
- ❖\? 表示问号

1.8 char类型：字符和小整数

✓转义序列的编码

表 3.2 C++转义序列的编码

字 符 名 称	ASCII 符号	C++代码	十进制 ASCII 码	十六进制 ASCII 码
换行符	NL (LF)	\n	10	0xA
水平制表符	HT	\t	9	0x9
垂直制表符	VT	\v	11	0xB
退格	BS	\b	8	0x8
回车	CR	\r	13	0xD
振铃	BEL	\a	7	0x7
反斜杠	\	\\	92	0x5C
问号	?	\?	63	0x3F
单引号	'	\'	39	0x27
双引号	"	\"	34	0x22

1.8 char类型：字符和小整数

✓转义序列的编码

❖可以将换行符嵌入到较长的字符串中，通常比endl方便

```
cout << endl << endl << "What next? " << endl << "Enter
a number:" << endl;
```

```
cout << "\n\nWhat next?\nEnter a number:\n";
```

1.8 char类型：字符和小整数

✓转义序列的编码

❖显示数字时，使用endl比输入"\n"或'\n'更容易些，但显示字符串时，在字符串末尾添加一个换行符，所需要的输入量要少些

```
cout << x << endl; //easier than cout << x << "\n"


cout << "Dr.X.\n";//easier than cout << "Dr.X." << endl;
```

1.8 char类型：字符和小整数

✓转义序列的编码

❖提示：在可以使用数字转义序列或符号转义序列（如\0x8和\b）时，应该使用符号序列。数字表示与特定的编码方式（如ASCII码）相关，而符号表示适用于任何编码方式，可读性也更强

1.8 char类型：字符和小整数

```
1   // bondini.cpp -- using escape sequences
2   #include <iostream>
3   int main()
4   {
5       using namespace std;
6       cout << "\aOperation \"HyperHype\" is now activated!\n";
7       cout << "Enter your agent code:________\b\b\b\b\b\b\b\b";
8       long code;
9       cin >> code;
10      cout << "\aYou entered " << code << "...\n";
11      cout << "\aCode verified! Proceed with Plan Z3!\n";
12      // cin.get();
13      // cin.get();
14      return 0;
15  }
```

```
operation "HyperHype" is now activated!
Enter your agent code:42007007
You entered 42007007...
Code verified! Proceed with Plan z3!
```

1.8 char类型：字符和小整数

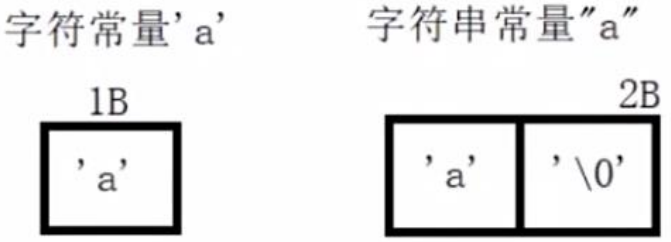
✓转义序列的编码

char ch='\0';

注意 '\0'，'0'和数值0的区别：

	'\0'	'0'	0
意义	代表ASCII码中的空字符，也称为null字符或者零终止符号	代表ASCII码为48的数字字符	表示十进制中的数字0
类型	字符类型	字符类型	整数类型
所占空间	1个字节	1个字节	4个字节

1.8 char类型：字符和小整数



✓字符常量和字符串常量的区别

	字符常量	字符串常量
表示方式不同	用单引号括起来， 例如 'a'	用双引号括起来， 例如 "hello world"
数据类型不同	一个字符， char类型	一串字符数组，const char*
存储方式不同	存储在内存中的一个字符单元中	是以 null 结尾的字符数组，存储在静态存储区域（即常量区）
赋值方式不同	可以直接赋值给 char 类型的变量， 例如 char a = 'a';	需要使用 strcpy() 函数或者初始化语法赋值给 char* 类型的变量， 例如 const char* str = "hello world"; 或者 char str[] = "hello world";
操作方式不同	可以进行逻辑运算、比较操作等	只能使用字符串函数库进行操作，例如 strlen、strcpy、strcat 等

1.8 char类型：字符和小整数

✓通用字符名

❖C++有一种表示特殊字符的机制，它独立于任何特定的键盘，使用的是通用字符名（universal character name）

❖通用字符名的用法类似于转义序列

❖可以用\u或者\U开头，\u后面是4个十六进制位，\U后面则是8个十六进制位。这些位表示的是字符的ISO 10464码点

例如：\u00F6

1.8 char类型：字符和小整数

✓通用字符名

- ISO 10464是一种正在制定的国际标准，为大量字符提供数字编码
- Unicode提供了一种表示字符集的解决方案-为大量字符和符号提供标准数字编码 例如：U-222B
- ASCII码是Unicode的一个子集
- ISO 10646小组和Unicode小组从1991年开始合作，保证他们的标准是同步的

1.8 char类型：字符和小整数

✓通用字符名

序号	具体符号	Unicode
1	㎡	\u33A1
2	×	\u00D7
3	≥	\u2265
4	≤	\u2264
5	∽	\u223D
6	δ	\u03B4
7	·	\u00B7
8	÷	\u00F7

1.8 char类型：字符和小整数

✓signed char 和 unsigned char

❖C++中，有3种不同的字符类型：

```
char fodo;//may be signed, may be unsigned

signed char snark;//definitely signed

unsigned char bar;//definitely unsigned
```

1.8 char类型：字符和小整数

✓signed char 和 unsigned char

❖如果将字符类型用作数字，那么：

signed char，表示范围[-128 ~ 127]

unsigned char，表示范围 [0 ~ 255]

1.8 char类型：字符和小整数

✓signed char 和 unsigned char

❖如果用于文本，则使用未加限定的char，是类似于 'a', '0'的类型，或是组成C字符串"abcde"的类型

❖到底是signed char还是unsigned char？这得看编译器：VC编译器、x86上的GCC都把char定义为signed char，而arm-linux-gcc却把char定义为unsigned char

❖字符('a', 'b', 'c'等）本质上也是一个整数，只是字符代表的值是0～127，我们可以用char表示一个不太大的整数(小整数）

1.8 char类型：字符和小整数

✓wchar_t

❖传统的字符数据类型为char，占用一个字节，存放的数据内容为ASCII编码，最多可以存放255种字符，基本的英文以及常用字符都可以涵盖

❖随着计算机在国际范围内普及，大量使用其它语言的计算机用户也纷纷出现，传统的ASCII编码已经无法满足人们的使用，因此一种新的字符存放类型wchar_t应运而生

❖wchar_t为宽字符类型或双字符类型，它占用两个字节，因此能够存放更多的字符

1.8 char类型：字符和小整数

✓wchar_t

```
wchar_t bob = L'P';   //a wide-character constant
wcout << L"tall" << endl;  //outputting a wide-character string
```

❖定义wchar_t时,需要以L开头,如果没有L,程序将会将wchar_t转换为char

❖对于ASCII码能够存放的数据类型,其高位存放的数据为0x00

❖char类型的字符串以'\0'结尾,wchar_t类型的字符串以'\0\0'结尾

1.8 char类型：字符和小整数

✓wchar_t

```
wchar_t bob = L'P';   //a wide-character constant
wcout << L"tall" << endl;  //outputting a wide-character string
```

❖要想输出wchar_t类型的字符,必须使用wcout,其在iostream中声明

❖后续例子不使用宽字符类型，但学习者需要知道这种类型，尤其是在进行国际编程或使用Unicode或ISO 10646时

1.8 char类型：字符和小整数

✓C++11新增的类型：char16_t和char32_t

```
char16_t ch1 = u'q';     //basic character in 16 bit form
char32_t ch2 = U'\U0000222B';   // universal character name in 32 bit form
```

❖在计算机系统上进行字符和字符串编码时，仅适用Unicode码点并不够

❖C++11新增了类型char16_t和char32_t两者都是无符号型

❖类型char16_t与/u00F6形式的通用字符名匹配

❖类型char32_t与/U0000222B形式的通用字符名相匹配

1.8 char类型：字符和小整数

✓C++11新增的类型：char16_t和char32_t

```
char16_t ch1 = u'q';     //basic character in 16 bit form
char32_t ch2 = U'\U0000222B';   // universal character name in 32 bit form
```

❖前缀u和U分别指出字符字面值的类型为char16_t和char32_t

❖char16_t与char32_t也都有底层类型——一种内置的整型，但底层类型可能随系统而异

❖后续例子不使用此新增类型，但学习者需要知道这种类型

1.9 布尔(bool)类型

✓在C语言中，是没有bool这个基础类型的。在C语言中，当要表示真或假的时候，都是定义一个非bool类型来使用的

✓ANSI/ISO C++标准添加了一种名叫bool的新类型，名称来源于英国数学家George Boole，是他开发了逻辑律的数学表示法

✓bool类型变量是用true和false来表示，也可以用非0值来表示true，用0来表示false，布尔类型在C++中是占用一个字节

```
bool is_ready = true;
```

1.9 布尔(bool)类型

❖字面值true和false都可以提升转换成int类型，true转换为1，而false转换为0

```
int ans = true;          // ans assigned 1
int promise = false;  // promise assigned 0
```

❖任何数字值或者指针都可以被隐式转换（即不用显示强制转化）为bool值。非零值转换为true，而零转换为false

```
bool start = -100;    // start assigned true
bool stop = 0;        // stop assigned false
```

非0为真，0为假

目录

- 简单变量
- const限定符
- 浮点数
- C++算术运算符

2.1 const限定符

✓如果程序在多个地方使用同一个常量，则需要修改常量时，只需要修改一个符号定义即可

✓#define 语句可以实现，符号常量-预处理器方法

```
#define ZERO 0        // makes ZERO symbol for 0 value
```

✓C++有一种更好的符号常量方法，使用const关键字来修饰变量声明和初始化

```
const int Months = 12; // Months is symbolic constant for 12
```

✓常量被初始化后，其值就被固定了，编译器将不允许再修改该常量

2.1 const限定符

✓创建常量的通用格式：

```
const type name = value;
```

✓常见常量命名法：首字母大写（Months）；整个名称大写（ZERO,常见于#define创建时）；以字母k打头的约定（kmonths）

✓const对象必须初始化，应在声明中就进行初始化

2.1 const限定符

✓const比#define好的原因：

- ❖const可以明确指定类型
- ❖可以使用C++的作用域规则将定义限制在特定的函数或文件中（作用域规则描述了名称在各种模块中的可知程度，后续课程讨论）
- ❖可以将const用于更复杂的类型如数组和结构体（后续课程讨论）
- ❖可以用const值来声明数组长度（后续课程讨论）

目录

- 简单变量
- const限定符
- 浮点数
- C++算术运算符

目录

- 浮点数
  - 书写浮点数
  - 浮点类型
  - 浮点常量
  - 浮点数的优缺点

3.1 书写浮点数

✓浮点数可以表示诸如 2.5、3.1415和100033.23 这样的数字，即带小数部分的数字

✓计算机将这样的值分成两部分储存。一部分表示值，另一部分用于对值进行放大和缩小

3.1415表示为0.31415（基准值）和 10 （缩放因子）

3141.5表示为0.31415（基准值相同）和 10000（缩放因子更大）

3.1 书写浮点数

✓放因子的作用是移动小数点的位置，浮点数可以表示非常大和非常小的数值，它们的内部表示方法和整数天壤之别

✓浮点数的书写：

❖第一种常规标准小数表达：

3.14、0.21 、8.0和 -14233.02

❖第二种浮点值法用 e 或者 E 表示

3.14E6 表示 3.14 与1000000 相乘

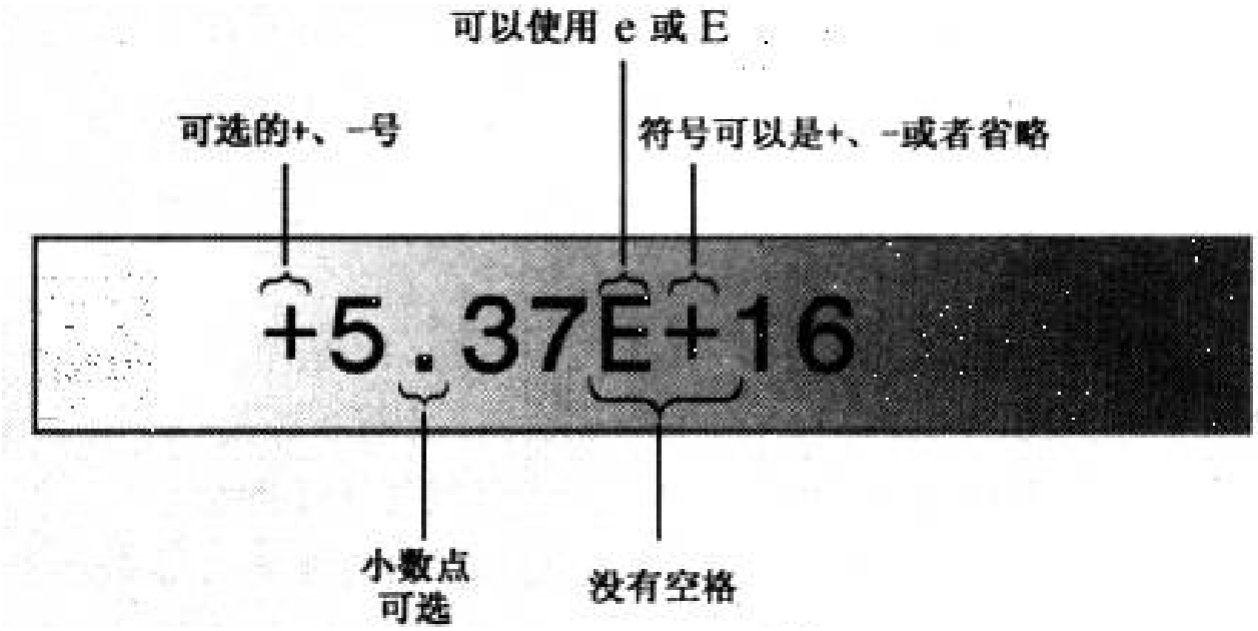
-0.12e4 表示 -0.12 与10^4（10000）相乘

3.1 书写浮点数

❖第二种浮点值法用 e 或者 E 表示

3.14E6 表示 3.14 与1000000 相乘（ 3.14称为尾数，6称为指数）

❑ 前面的符号用于数字，而指数的符号用于缩放

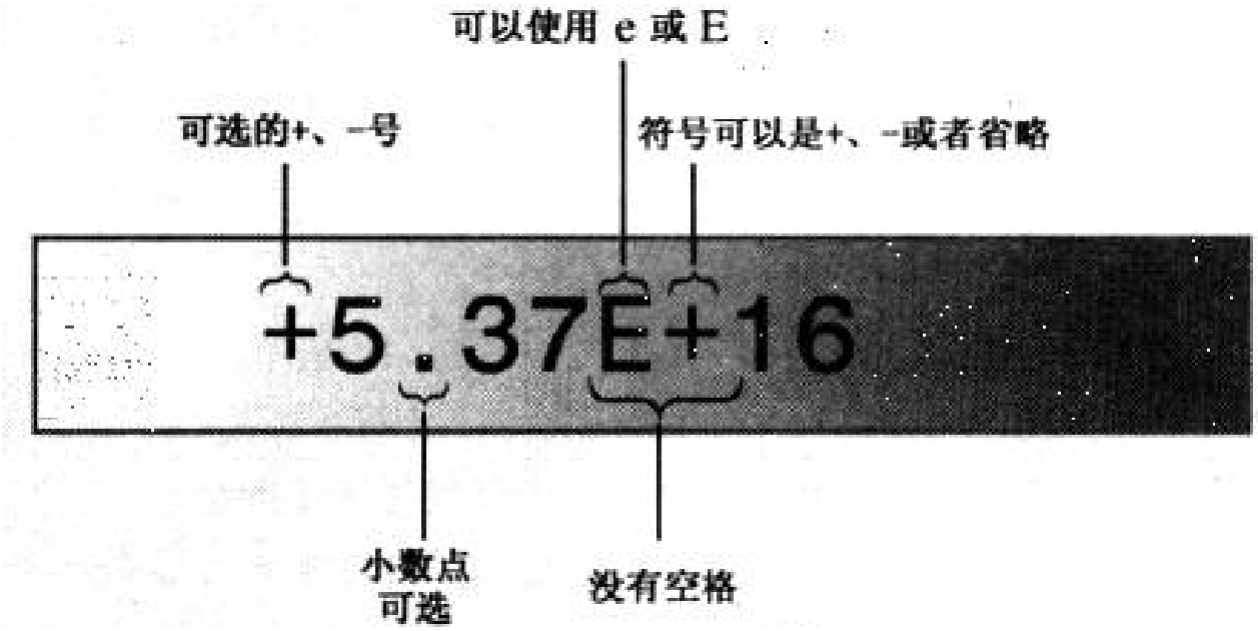


浮点值E表示法

3.1 书写浮点数

❖第二种浮点值法用 e 或者 E 表示

- d. dddE+n指的是将小数点向右移n位
- d. dddE-n指的是将小数点向左移n位



浮点值E表示法

3.2 浮点类型

✓c++ 有 3 种浮点类型：float、double 和 long double

✓这些类型是按照它们可以表示的有效位数和允许的指数最小范围来描述的。float至少32位，double至少48位，且不少于float，long double至少和double一样多

✓有效位（significant figure）是数字中有意义的位

3.2 浮点类型

✓浮点数在内存中存储分为三部分：符号位、指数部分和尾数部分

✓浮点数有指定有效位数(float:6位/double:15位)，超出有效位数则舍去(四舍五入)，因此会产生误差

✓浮点常量缺省为double型，如需表示为float型，可以在后面加f(F)

1.23(double) 1.23F(float)

```
// floatnum.cpp -- floating-point types
#include <iostream>
int main()
{
    using namespace std;
    cout.setf(ios_base::fixed, ios_base::floatfield); // fixed-point
    float tub = 10.0 / 3.0;      // good to about 6 places
    double mint = 10.0 / 3.0;    // good to about 15 places
    const float million = 1.0e6;
    cout << "tub = " << tub;
    cout << ", a million tubs = " << million * tub;
    cout << ",\nand ten million tubs = ";
    cout << 10 * million * tub << endl;
    cout << "mint = " << mint << " and a million mints = ";
    cout << million * mint << endl;
    return 0;
}
```

```
tub = 3.333333, a million tubs = 3333333.250000,
and ten million tubs = 33333332.000000
mint = 3.333333 and a million mints = 3333333.333333
```

3.3 浮点常量

✓浮点常量通常为 double 类型，如果希望常量为 float 类型，可以使用 f 或 F 后缀，如果希望使用 long double 类型，可以使用 l 或者 L 后缀（更推荐L）。如 3.14f 或者 3.14L

```
1.23f          // a float constant
2.45E20F       // a float constant
2.345324E28    // a double constant
2.2L           // a long double constant
```

3.4 浮点数的优缺点

✓浮点数的优点

- ❖ 可以表示整数之间的值
- ❖ 由于存在缩放因子，它们可以表示的范围要大得多

✓浮点数的缺点

- ❖ 计算效率低
- ❖ 精度会降低

3.4 浮点数的优缺点

```
// fltadd.cpp -- precision problems with float
#include <iostream>
int main()
{
    using namespace std;
    float a = 2.34E+22f;
    float b = a + 1.0f;

    cout << "a = " << a << endl;
    cout << "b - a = " << b - a << endl;
    return 0;
}
```

```
a = 2.34e+022
b - a = 0
```

2.34E+022是一个23位的数字。加上1就是在最末位加1。但float类型只能表示数字中的前6位或前7位，因此修改第23位对这个值不会有任何影响

将类型分类

C++对基本类型进行分类，形成了若干个族。类型 signed char、short、int 和 long 统称为符号整型；它们的无符号版本统称为无符号整型；C++11 新增了 long long。bool、char、wchar_t、符号整数和无符号整型统称为整型；C++11 新增了 char16_t 和 char32_t。float、double 和 long double 统称为浮点型。整数和浮点型统称算术（arithmetic）类型。

- 基本类型（算术arithmetic类型）
  - 整型
    - 符号整型
      - 字符型(signed char)
      - 短整型(short)
      - 整型 (int)
      - 长整型(long)
    - 无符号整型
    - 其他整型
      - 布尔型(bool)
      - char
      - wchar_t
    - C++11新增
      - long long
      - char16_t
      - char32_t
  - 浮点型
    - 单精度型(float)
    - 双精度型(double)
    - 长双精度型(long double)

目录

- 简单变量
- const限定符
- 浮点数
- C++算术运算符

目录

- C++算术运算符
  - 运算符优先级和结合性
  - 除法分支
  - 求模运算符
  - 类型转换
  - C++11中的auto声明

4. C++算术运算符

✓ C++ 支持的算术运算符

假设变量 A 的值为 21，变量 B 的值为 10，则：

运算符	描述	实例
+	把两个操作数相加	A + B 将得到 31
-	从第一个操作数中减去第二个操作数	A - B 将得到 11
*	把两个操作数相乘	A * B 将得到 210
/	分子除以分母	A / B 将得到 2
%	取模运算符，整除后的余数（两个操作数必须为整型）	A % B 将得到 1
++	自增运算符，整数值增加 1	A++ 将得到 22
--	自减运算符，整数值减少 1	B -- 将得到 9

```cpp
// arith.cpp -- some C++ arithmetic
#include <iostream>
int main()
{
    using namespace std;
    float hats, heads;

    cout.setf(ios_base::fixed, ios_base::floatfield); // fixed-point
    cout << "Enter a number: ";
    cin >> hats;
    cout << "Enter another number: ";
    cin >> heads;

    cout << "hats = " << hats << "; heads = " << heads << endl;
    cout << "hats + heads = " << hats + heads << endl;
    cout << "hats - heads = " << hats - heads << endl;
    cout << "hats * heads = " << hats * heads << endl;
    cout << "hats / heads = " << hats / heads << endl;
    return 0;
}
```

- 61.41998是由于float类型表示有效位数的能力有限。对于float，C++只保证6位有效位。如果61.41998四舍五入成6位，将得到61.4200，这是保证精度下的正确值
- 如果需要更高的精度，请使用double或long double

```
Enter a number: 50.25
Enter another number: 11.17
hats = 50.250000; heads = 11.170000
hats + heads = 61.419998
hats - heads = 39.080002
hats * heads = 561.292480
hats / heads = 4.498657
```

```cpp
// arith.cpp -- some C++ arithmetic
#include <iostream>
int main()
{
    using namespace std;
    double hats, heads;

    cout.setf(ios_base::fixed, ios_base::floatfield); // fixed-point
    cout << "Enter a number: ";
    cin >> hats;
    cout << "Enter another number: ";
    cin >> heads;

    cout << "hats = " << hats << "; heads = " << heads << endl;
    cout << "hats + heads = " << hats + heads << endl;
    cout << "hats - heads = " << hats - heads << endl;
    cout << "hats * heads = " << hats * heads << endl;
    cout << "hats / heads = " << hats / heads << endl;
    return 0;
}
```

- 61.41998是由于float类型表示有效位数的能力有限。对于float，C++只保证6位有效位。如果61.41998四舍五入成6位，将得到61.4200，这是保证精度下的正确值
- 如果需要更高的精度，请使用double或long double

```
Enter a number: 50.25
Enter another number: 11.17
hats = 50.250000; heads = 11.170000
hats + heads = 61.420000
hats - heads = 39.080000
hats * heads = 561.292500
hats / heads = 4.498657
```

4.1 运算符优先级和结合性

✓C++使用优先级规则来决定首先使用哪个运算符

- ❖算术运算符遵循通常的代数优先级，先乘除，后加减
- ❖可以使用括号来执行自己定义的优先级（1-15级，数字越小优先级越高）
- ❖当优先级相同时，C++将看操作数的结合性（associativity）是从左到右，还是从右到左

4.1 运算符优先级和结合性

✓C++使用优先级规则来决定首先使用哪个运算符

❖多数运算符的结合性都是从左往右，只有三类运算符是从右往左的

❑所有单目运算符

-	负号运算符	-表达式	右到左	单目运算符
(类型)	强制类型转换	(数据类型)表达式		
++	自增运算符	++变量名 变量名++		单目运算符
--	自减运算符	--变量名 变量名--		单目运算符
*	取值运算符	*指针变量		单目运算符
&	取地址运算符	&变量名		单目运算符
!	逻辑非运算符	!表达式		单目运算符
~	按位取反运算符	~表达式		单目运算符

4.1 运算符优先级和结合性

✓C++使用优先级规则来决定首先使用哪个运算符

❖多数运算符的结合性都是从左往右，只有三类运算符是从右往左的

❑所有单目运算符

❑唯一一个三目运算符 ?:

❑双目运算符中的赋值运算符（包括=、*=、/=、%=、+=、-=、&=、^=、|=、<<=、>>=等）。双目运算符中只有赋值运算符的结合性是从右往左的，其他的都是从左往右

✓运算符优先级和结合性详见：附录D 运算符优先级

4.2 除法分支

✓ C++除法分为两种情况，一种是整型之间的除法，一种是浮点数之间的除法

❖ 对于整型之间的除法（int），除法的结果只会保留商的整数部分，小数部分将被丢弃

❖对于浮点数之间的除法（float，double），除法的结果会保留小数部分，结果为浮点数

```cpp
// divide.cpp -- integer and floating-point division
#include <iostream>
int main()
{   using namespace std;
    cout.setf(ios_base::fixed, ios_base::floatfield);
    cout << "Integer division: 9/5 = " << 9 / 5 << endl;
    cout << "Floating-point division: 9.0/5.0 = ";
    cout << 9.0 / 5.0 << endl;
    cout << "Mixed division: 9.0/5 = " << 9.0 / 5 << endl;
    cout << "double constants: 1e7/9.0 = ";
    cout << 1.e7 / 9.0 << endl;
    cout << "float constants: 1e7f/9.0f = ";
    cout << 1.e7f / 9.0f << endl;
    return 0;
}
```

```
Integer division: 9/5 = 1
Floating-point division: 9.0/5.0 = 1.800000
Mixed division: 9.0/5 = 1.800000
double constants: 1e7/9.0 = 1111111.111111
float constants: 1e7f/9.0f = 1111111.125000
```

4.2 除法分支

✓除法运算符表示了3种不同运算：int除法、float除法和double除法

✓使用相同符号进行多种操作叫作运算符重载（operator overloading）

✓C++有一些内置重载示例

✓C++还允许扩展运算符重载，以便能够用于用户定义的类，这就是面向对象的属性

//此页属于后续进阶内容，了解即可

各种除法

4.3 求模运算符

✓求模运算符%返回整数除法的余数，取余数的意思

重点：求模运算符只用于整数，不能用于浮点

✓求模运算符%与整数除法相结合，适用于解决要求将一个量分成不同的整数单元的问题

4.3 求模运算符

```
// modulus.cpp -- uses % operator to convert lbs to stone
#include <iostream>
int main()
{
    using namespace std;
    const int Lbs_per_stn = 14;
    int lbs;
    cout << "Enter your weight in pounds: ";
    cin >> lbs;
    int stone = lbs / Lbs_per_stn;      // whole stone
    int pounds = lbs % Lbs_per_stn;     // remainder in pounds
    cout << lbs << " pounds are " << stone
         << " stone, " << pounds << " pound(s).\n";
    return 0;
}
```

```
Enter your weight in pounds: 181
181 pounds are 12 stone, 13 pound(s).
```

4.4 类型转换

✓ 由于有11种整型和3种浮点型，因此计算机需要处理大量不同的情况，尤其是对不同的类型进行运算

✓ C++自动执行很多类型转换：

- ❖将一种算术类型的值赋给另一种算术类型的变量时
- ❖表达式中包含不同的类型时
- ❖将参数传递给函数时

4.4 类型转换

1）初始化和赋值进行的转换

❖C++允许将一种类型的值赋给另一种类型的变量，值将被转换为被接收变量的类型

```
short thirty; long so_long; so_long = thirty; // assigning a short to a long
```

转换	潜在的问题
将较大的浮点类型转换为较小的浮点类型，如将double转换为float	精度（有效数位）降低，值可能超出目标类型的取值范围，这种情况下，结果将是不确定的
将浮点类型转换为整型	小数部分丢失，原来的值可能超出目标类型的取值范围，这种情况下，结果将是不确定的
将较大的整型转换为较小的整型，如将long转换为short	原来的值可能超出目标类型的取值范围，通常只赋值右边的字节

4.4 类型转换

```
// assign.cpp -- type changes on assignment
#include <iostream>
int main()
{
    using namespace std;
    cout.setf(ios_base::fixed, ios_base::floatfield);
    float tree = 3;      // int converted to float
    int guess = 3.9832; // float converted to int
    int debt = 7.2E12;  // result not defined in C++
    cout << "tree = " << tree << endl;
    cout << "guess = " << guess << endl;
    cout << "debt = " << debt << endl;
    return 0;
}
```

- 浮点型转整型，C++采用截取（丢弃小数部分）而不是四舍五入（查找最接近的整数）
- debt无法存储7.2E12，导致C++没有对结果进行定义的情况发生

```
tree = 3.000000
guess = 3
debt = 1634811904
```

4.4 类型转换

2）以{}方式初始化时进行的转换（C++11）

✓C++11将使用大括号{}的初始化称为列表初始化（list-initialization），这种初始化常用于给复杂的数据类型提供列表

✓列表初始化不允许缩窄（narrowing），即变量的类型可能无法表示赋给它的值

4.4 类型转换

2）以{}方式初始化时进行的转换（C++11）

```
const int code = 66;
int x = 66;
char c1{31325}; // narrowing, not allowed
char c2 = {66}; // allowed because char can hold 66
char c3 {code}; // ditto 同上
char c4 = {x}; // not allowed, x is not constant
x = 31325;
char c5 = x; // allowed by this form of initialization
```

4.4 类型转换

3）表达式中的转换

✓在计算表达式时，C++将bool、char、unsigned char、signed char和short值自动转换为int，这称之为整型提升（integral promotion）（通常将int类型选择为计算机最自然的类型，此时运算速度最快）

4.4 类型转换

3）表达式中的转换

✓还有其他一些整型提升，如果short比int短，则unsigned short类型将被转换为int；如果两种类型长度相同，则unsigned short类型将被转换为unsigned int。这为保证对unsigned short进行提升时不会损失数据

✓wchar_t被提升为下列类型中第一个宽度足够存储wchar_t取值范围的类型：int、unsignedint、long或unsigned long

4.4 类型转换

3）表达式中的转换

✓当运算涉及两种类型时，较小的类型将被转换为较大的类型

✓C++11版本的校验表，编译器将依次查阅该列表

1. 如果有一个操作数的类型是long double，则将另一个操作数转换为long double
2. 否则，如果有一个操作数的类型时double，则将另一个操作数转换为double
3. 否则，如果一个操作数的类型时float，则将另一个操作数转换为float
4. 否则，说明操作数都是整型，因此执行整型提升
5. 在这种情况下，如果两个操作数都是有符号或者无符号的，且其中一个操作数的级别比另一个低，则转换为级别高的类型
6. 如果一个操作数为有符号的，另一个操作数为无符号的，且无符号操作数的级别比有符号操作数高，则将有符号操作数转换为无符号操作数所属类型
7. 否则，如果有符号类型可表示无符号类型的所有可能取值，则将无符号操作数转换为有符号操作数所属类型
8. 否则，将两个操作数都转换为有符号类型的无符号版本

4.4 类型转换

3）表达式中的转换

❖有符号整型按级别从高到低依次为long long、long、int、short和signed char

❖无符号类型的排列顺序与有符号整型相同

❖类型char、signed char和unsigned char的级别相同

❖类型bool的级别最低

❖wchar_t、char16_t和char32_t的级别与其底层类型相同

4.4 类型转换

4）传递参数时的转换

✓传递参数时的类型转换通常由C++函数原型控制（后续课程介绍）

✓在对取消原型对参数传递的控制时，C++将对char和short类型（signed和unsigned）应用整型提升

✓为了保持与传统C语言中大量代码的兼容性，在将参数传递给取消原型对参数传递控制的函数时，C++将float参数提升为double

特别说明：转换优先级的规则很复杂，这里只是一个适合初学者的简易规则，
不完全正确，如果与实际编译器的表现有不一致的地方，以编译器为准

➢ 转换的优先级（简易理解）

高 ↑
- long double
- double
- float
- unsigned long long
- long long
- unsigned long
- long
- unsigned [int]

低 ↑ int <- char, u_char, short, u_short <-整型提升(表示必定的转换)

4) 否则两个操作数均为整数。此情况下，
首先，两个操作数都会经历*整数提升*（见后述）。然后

- 若两类型在提升后相同，则该类型即为共用类型
- 否则，若两操作数在提升后有相同的符号性（均为有符号或均为无符号），则拥有较低*转换等级*（见后述）者会隐式转换成拥有较高*转换等级*的操作数的类型
- 否则，两者符号性不同：若无符号类型操作数拥有大于或等于有符号类型操作数的*转换等级*，则有符号类型操作数会隐式转换成无符号类型
- 否则，两者符号性不同且有符号操作数的等级大于无符号操作数的等级。此情况中，若有符号类型可以表达无符号类型的所有值，则有无符号类型的操作数被隐式转换成有符号操作数的类型。
- 否则，两个操作数都会经历隐式转换，到有符号类型的无符号类型对应者。

```
1.f + 20000001; // int 被转换成 float ，给出 20000000.00
                // 相加后舍入到 float ，给出20000000.00
(char)'a' + 1L; // 首先，char 被提升回 int。
                // 这是有符号+有符号的情形，等级不同
                // int 被转换成 long ，结果是 signed long 的 98
2u - 10; // 无符号/有符号，等级相同
         // 10 被转换成无符号，无符号数学运算为模 UINT_MAX+1
         // 对于 32 位 int ，结果是 unsigned int 类型的 4294967288 （即 UINT_MAX-7 ）
0UL - 1LL; // 无符号/有符号，相异等级，有符号的等级较大。
           // 若 sizeof(long) == sizeof(long long) ，则有符号数不能表示所有无符号数
           // 这是最后一种情况：两个操作数都被转换成 unsigned long long
           // 结果是 unsigned long long 类型的 18446744073709551615（ ULLONG_MAX ）
```

4.4 类型转换

5）强制类型的转换

✓C++还允许通过强制类型转换机制显式地进行类型转换

✓强制转换的通用格式如下：

```
(typeName)value // old C syntax:converts value to typeName type

typeName (value) // new C++ syntax:converts value to typeName type
```

4.4 类型转换

5）强制类型的转换

✓在C++语言中引入了4个强制类型转换运算符

static_cast、const_cast、reinterpret_cast和dynamic_cast

✓static_cast<typeName>(value)用于数据类型的强制转换，强制将一种数据类型转换为另一种数据类型

```
// typecast.cpp -- forcing type changes
#include <iostream>
int main()
{
    using namespace std;
    int auks, bats, coots;
    // the following statement adds the values as double,
    // then converts the result to int
    auks = 19.99 + 11.99;    // these statements add values as int
    bats = (int)19.99 + (int)11.99;   // old C syntax
    coots = int(19.99) + int(11.99);  // new C++ syntax
    cout << "auks = " << auks << ", bats = " << bats;
    cout << ", coots = " << coots << endl;
    char ch = 'Z';
    cout << "The code for " << ch << " is ";     // print as char
    cout << int(ch) << endl;                     // print as int
    cout << "Yes, the code is ";
    cout << static_cast<int>(ch) << endl;       // using static_cast
    return 0;
}
```

```
auks = 31, bats = 30, coots = 30
The code for Z is 90
Yes, the code is 90
```

注意：
将值转换成int，然后相加得到的结果，与先将值相加然后转换为int的值不同

4.5 C++11中的auto声明

✓C++重新定义了auto的含义，让编译器能够根据初始值的类型推断变量的类型

✓在初始化声明中，如果使用关键字auto，而不指定变量的类型，编译器将把变量的类型设置成与初始值相同：

```
auto n = 100;      // n is int

auto x = 1.5;      // x is double

auto y = 1.3e12L; // y is long double
```

4.5 C++11中的auto声明

✓自动类型推断并非为简单情况而设定。应用于简单情况时易产生错误，例如：假设要将x，y和z都指定为double类型，编写如下：

```
auto x = 0.0;    // ok, x is double
① double y = 0;  // ok, 0 automatically converted to 0.0
② auto z = 0;    // oops, z is int because 0 is int
```

① 显示声明类型时，将变量初始化为0不会导致任何问题

② 采用自动类型推断时，导致类型不符合要求的问题

4.5 C++11中的auto声明

✓自动类型推断的优势在处理复杂类型，如标准模块库（STL）中的类型时才能体现

```
std::vector<double> socres;
std::vector<double>::iterator pv = scores.begin(); //C++98
```

```
std::vector<double> socres;
auto pv = scores.begin();    //C++11
```

总结

- 简单变量
  - 变量名
  - 整型
  - 整型short、int、long和long long
  - 无符号类型
  - 选择整型类型
  - 整型字面值
  - C++如何确定常量的类型
  - char类型：字符和小整数
  - bool类型

总结

- const限定符
- 浮点数
  - ➢书写浮点数
  - ➢浮点类型
  - ➢浮点常量
  - ➢浮点数的优缺点
- C++算术运算符
  - ➢运算符优先级和结合性
  - ➢除法分支
  - ➢求模运算符
  - ➢类型转换
  - ➢C++11中的auto声明