**Chapter 3 Dealing with data types**

* **C++’s built-in integer types**: *unsigned long*, ***long***; *unsigned* ***int***, *int*; *unsigned*

*short*, ***short***; ***char***, *unsigned char*, *signed char*; ***bool***;

C++11’s additions: *unsigned long long* and *long long*;

C++’s basic integer types, in order of increasing width, are **char**, **short**, **int**, **long**, and,

with C++11**, long long. Each comes in both signed and unsigned versions.**

* **C++’s built-in floating-point types**: *float*, *double*, and *long double*;
* **C++变量命名规则**

1. The only characters you can use in names are **alphabetic characters**, **numeric digits**, and the **underscore (\_) character**,
2. **The first character in a name cannot be a numeric digit,**
3. Uppercase characters are considered distinct from lowercase characters,
4. Names beginning with **two underscore characters** or with **an underscore character followed by an uppercase letter** are reserved for use by the implementation—that is, the compiler and the resources it uses. Names beginning with a single underscore character are reserved for use as global identifiers by the implementation,
5. C++ places **no limits on the length of a name**, and all characters in a name are significant. However, some platforms might have their own length limits.

Using a name such as **\_\_time\_stop** or **\_Donut** doesn’t produce a compiler error; instead, **it leads to undefined behavior**. In other words, there’s no telling what the result will be. The reason there is no compiler error is that the names are not illegal but rather are reserved for the implementation to use.

----------------------------------------------------------------------------------------------------------------

Int terrier; // invalid -- has to be int, not Int

int my\_stars3; // valid

int \_Mystars3; // valid but reserved -- starts with underscore

int 4ever; // invalid because 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 two underscores

int the\_very\_best\_variable\_i\_can\_be\_version\_112; // valid

int honky-tonk; // **invalid -- no hyphens allowed**

The integer myWeight might be named **nMyWeight**; here, the n prefix is used to represent an integer value, or be named **intMyWeight**, which is more precise and legible;

**str or sz** might be used to represent a null-terminated string of characters, b might represent a Boolean value, p a pointer, c a single character;

* C++关于几种类型的standard：

1. A ***short*** integer is at **least 16 bits** wide,
2. An ***int*** integer is **at least as big as short**,
3. A ***long*** integer is ***at least 32 bits wide and at least as big as int***,
4. A ***long long*** integer is *at least 64 bits wide and at least as big as long*,

Typically, ***int*** is 32 bits (the same as long) for Windows XP, Windows Vista, Windows 7, Macintosh OS X, VAX, and many other minicomputer implementations.

The four type***, int, short, long, and long long***, are ***signed*** types.

* 检测数据宽度的方法

1. First, the **sizeof**operator returns the size, **in bytes**, of a type or a variable；

You can apply the **sizeof** operator **to a type name or to a variable name**. When you use the **sizeof operator with a type name, such as int**, you enclose the name in parentheses. But **when you use the operator with the name of the variable**, such as n\_short, parentheses are optional:

cout << "int is " << **sizeof (int)** << " bytes.**\n**";

cout << "short is " << **sizeof n\_short** << " bytes.**\n**";

1. Second, the ***climits*** header file (or, for older implementations, the limits.h header file) contains **information about integer type limits**. In particular, it defines symbolic names to represent different limits. For example, it defines INT\_MAX as the largest possible int value and CHAR\_BIT as the number of bits in a byte.

* You can use #define to define your own symbolic constants. However, the ***#define*** directive is a C relic. C++ has a better way of creating symbolic constants, using the ***const*** keyword.
* **Initialization with C++11**

int owls = 101; // traditional C initialization, sets owls to 101

int wrens(432); // alternative C++ syntax, set wrens to 432

int rheas = {12}; int rheas{12}; // set rheas to 12

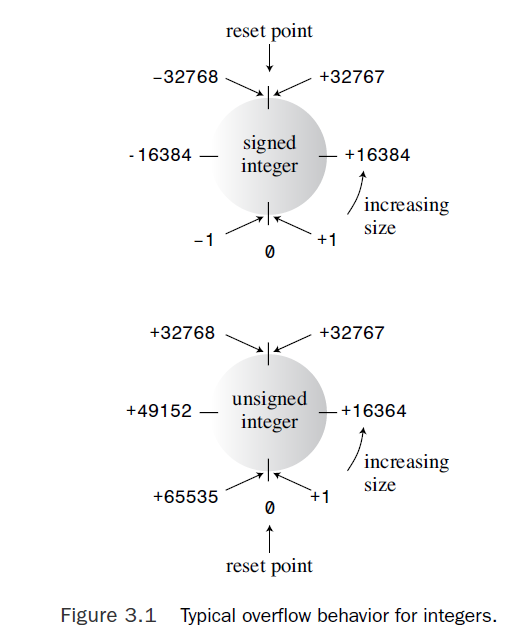
The braces can be left empty, in which case the variable is initialized to 0:

int rocs = {}; // set rocs to 0

int psychics{}; // set psychics to 0

unsigned quarterback; // also unsigned int

Integers behave much like an odometer. If you go past the limit, the values just start over at the other end of the range (see Figure 3.1). C++ guarantees that unsigned types behave in this fashion. However, C++ doesn’t guarantee that signed integer types can exceed their limits (overflow and underflow) without complaint, but that is the most common behavior on current implementations.



If you know that the variable might have to represent **integer values too great for a 16-bit integer, you should use long. This is true even if int is 32 bits on your system**. That way, if you transfer your program to a system with a 16-bit int, your program won’t embarrass you.

int chest = 42; // decimal integer literal

int waist = **0x**42; // hexadecimal integer literal

int inseam = 042; // octal integer literal

**If you want to display a value in hexadecimal or octal form**, you can use some special features of **cout**. It provides the **dec**, **hex**, and **oct** manipulators to give cout the messages to display integers in decimal, hexadecimal, and octal formats, respectively. Listing below uses hex and oct to display the decimal value 42 in three formats. (**Decimal is the default format, and each format stays in effect until you change it**.)

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;

结果是：

Monsieur cuts a striking figure!

chest = 42 (decimal for 42)

waist = 2a (hexadecimal for 42)

inseam = 52 (octal for 42)

Also note that because the **identifier hex is part of the std namespace** and the program uses that namespace, **this program can’t use hex as the name of a variable**. **However, if you omitted the *using* directive and instead used std::cout, std::endl, std::hex, and std::oct, you could still use plain hex as the name for a variable.**

* **The char Type: Characters and Small Integers**

Char is also an integer type that is typically smaller than short.

----------------------------------------------------------------------------

char ch;

cout << "Enter a character: " << endl;

**cin >> ch**;

cout << "Hola! ";

cout << "Thank you for the " << ch << " character." << endl;

----------------------------------------------------------------------------

运行结果：

Enter a character:

**M**

Hola! Thank you for the M character.

On input, cin converts the keystroke input M to the value 77. On output, cout converts the value 77 to the

displayed character M; **cin and cout are guided by the type of variable**.

C++ uses **single quotation** marks for a **character** and **double quotation** marks for a **string**. The cout object can handle either, but, as Chapter 4 discusses, the two are quite different from one another. Finally, the program introduces a cout feature, the **cout.put()** function, **which displays a single character**.

-------------------------------------------------------------------------

using namespace std;

char ch = 'M';

int i = ch;

cout << "The ASCII code for " << ch << " is " << i << endl;

cout << "Add one to the character code:" << endl;

ch = ch + 1;

i = ch;

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('!'); // 注意这里是单引号；

-------------------------------------------------------------------------

运行结果：

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!

Even digits entered via the keyboard are read as characters. Consider the following sequence:

**char ch**;

cin >> ch;

If you type 5 and Enter, this code reads the 5 character and stores the character code for the 5 character (53 in ASCII) in ch. Now consider this code:

**int n**;

cin >> n;

The same input results in the program reading the 5 character and **running a routine converting the character to the corresponding numeric value of 5, which gets stored in n**.

为什么在有cout <<的情况下还要用cout.put()呢？

这是历史原因. 在**C++2.0以及之前，C++把character constant, 比如‘M’, ’N’这种的，存为int型**，也就是他们的ASCII码. So a statement like the following would print the ASCII code for the $ character rather than simply display $:

cout << '$';

因为cout根据数据类型输出，而’$’是int型的，所以输出他的ASCII码；

But the following would print the character, as desired:

cout.put('$');

Meanwhile, **char variables typically occupied 8 bits**. A statement like the following copied 8 bits (the important 8 bits) from the constant 'M' to the variable ch:

char ch = 'M';

‘M’是int型的，ch是char型的，只有8bit，所以ch其实没能把’M’的ASCII码拷贝过来；

**Now, after Release 2.0, C++ stores single-character constants as type char, not type int. Therefore, cout now correctly handles character constants.** 所以, cout.put()是一个历史遗留产物.

The newline character provides an alternative to endl for inserting new lines into output. You can use the newline character in character constant notation ('\n') **or as character in a string ("\n")**. All three of the following move the screen cursor to the beginning of the next line:

----------------------------------------------------

cout << endl; // using the endl manipulator

cout << '\n'; // using a character constant

cout << "\n"; // using a string

----------------------------------------------------

For example, Ctrl+Z has an ASCII code of 26, which is 032 in octal and 0x1a in hexadecimal. You can represent this character with either of the following escape sequences: \032 or \x1a.You can make character constants out of these by enclosing them in single quotes, as in '\032', and you can use them as parts of a string, as in "hi\x1a there".

Unlike int, **char is not signed by default. Nor is it unsigned by default**. The choice is left to the C++ implementation in order to allow the compiler developer to best fit the type to the hardware properties. If it is vital to you that char has a particular behavior, you can use **signed char** or **unsigned char** explicitly as types.

wchar\_t, char16\_t and char32\_t未加总结;

* **The bool Type**

C++ interprets **nonzero values as true** and **zero values as false**.

bool is\_ready = true;

The literals **true and false** can be converted to type int by promotion, **with true converting to 1 and false to 0:**

int ans = true; // ans assigned 1

int promise = false; // promise assigned 0

Also any numeric or pointer value can be converted implicitly (that is, without an explicit type cast) to a bool value. Any nonzero value converts to true, whereas a zero value converts to false:

bool start = -100; // start assigned true

bool stop = 0; // stop assigned false

* **The const Qualifier**

If the program uses the constant in several places and you need to change the value, you can just change the single symbol definition.

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

Now you can use Months in a program instead of 12. After you initialize a constant such as Months, its value is set. The compiler does not let you subsequently change the value Months. If you try to, for example, g++ gives an error message that the program used an assignment of a **read-only** variable.

**const type name = value;**

Note that you initialize a const in the declaration. The following sequence is no good:

const int toes; // value of toes undefined at this point

toes = 10; // too late!

If you don’t provide a value when you declare the constant, it ends up with an unspecified value that you cannot modify.

**Difference between const and #define**: const is better. For one thing, it lets you **specify the type explicitly**. Second, you can use **C++’s scoping rules to limit the definition to particular functions or files**.

* **Floating-Point Numbers -- Numbers with fractional parts**

A computer stores such values in two parts. One part represents a value, and the other part scales that value up or down. Here’s an analogy. Consider the two numbers 34.1245 and 34124.5. They’re identical except for scale. You can represent the first one as 0.341245 (the base value) and 100 (the scaling factor).You can represent the second as 0.341245 (the same base value) and 100,000 (a bigger scaling factor).**The scaling factor serves to move the decimal point, hence the term floating-point**. C++ uses a similar method to represent floating-point numbers internally, **except it’s based on binary numbers, so the scaling is by factors of 2 instead of by factors of 10**.

**Writing Floating-Point Numbers**

12.34 // floating-point

939001.32 // floating-point

0.00023 // floating-point

8.0 // still floating-point

2.52e+8 // can use E or e, + is optional

8.33E-4 // exponent can be negative

7E5 // same as 7.0E+05

-18.32e13 // can have + or - sign in front

1.69e12 // 2010 Brazilian public debt in reais

5.98E24 // mass of earth in kilograms

9.11e-31 // mass of an electron in kilograms

这里的E指的是10的次方, 小数点前面的叫mantissa,小数点后面的叫exponent. As you might have noticed, E notation is most useful for very large and very small numbers. **E notation guarantees that a number is stored in floating-point format**, even if no decimal point is used.

Like ANSI C, C++ has **three floating-point types**: ***float***, ***double***, and ***long double***.

These types are described in terms of the **number of significant figures** they can represent and the minimum allowable range of exponents. **Significant figures** are the meaningful digits in a number. **14,179** uses **five significant figures**, **14,000 uses two significant figures**, for the result is rounded to the nearest thousand feet; in this case, the **remaining three digits are just placeholders**, and **14.179 thousand** uses **five significant digits.**

In effect, the C and C++ requirements for **significant digits** amount to ***float* being at least 32 bits**, ***double* being at least 48 bits** and certainly **no smaller than float**, and ***long double*** being at least as big as double.

Typically, however, **float is 32 bits**, **double is 64 bits**, and long double is 80, 96, or 128 bits. Also the **range in exponents for all three types is at least –37 to +37**.You can look in the cfloat or float.h header files to find the limits for your system.

-----------------------------------------------------------------------------

// the following are the minimum number of significant digits

#define DBL\_DIG 15 // double

#define FLT\_DIG 6 // float

#define LDBL\_DIG 18 // long double

// the following are the number of bits used to represent the mantissa

#define DBL\_MANT\_DIG 53

#define FLT\_MANT\_DIG 24

#define LDBL\_MANT\_DIG 64

// the following are the maximum and minimum exponent values

#define DBL\_MAX\_10\_EXP +308

#define FLT\_MAX\_10\_EXP +38

#define LDBL\_MAX\_10\_EXP +4932

#define DBL\_MIN\_10\_EXP -307

#define FLT\_MIN\_10\_EXP -37

#define LDBL\_MIN\_10\_EXP -4931

-----------------------------------------------------------------------------

// 输出的数字的小数点后面有10位；

printf("The minimum value of float = %.10e\n", FLT\_MIN)

-----------------------------------------------------------------------------

再来看一个例子

-----------------------------------------------------------------------------

#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;

}

-----------------------------------------------------------------------------

运行结果如下：

**tub = 3.333333**, a million tubs = 3333333.250000,

and ten million tubs = 33333332.000000

**mint = 3.333333** and a million mints = 3333333.333333

The program previews an **ostream** method called **setf()** from Chapter 17, Input, Output, and Files. This particular call **forces output to stay in fixed-point notation** so that you can better see the precision. It **prevents the program from switching to E notation** for large values and causes the program to **display six digits to the right of the decimal**. The arguments ***ios\_base::fixed*** and ***ios\_base::floatfield*** are constants provided by including **iostream**. 小数点右边输出6位可能是默认值.

Normally cout drops trailing zeros. For example, it would display 3333333.250000 as 3333333.25. The call to cout.setf() overrides that behavior, 使得最后出现0补位的现象. 前面提到示例系统支持double有15个significant digit:

#define DBL\_DIG 15 // double

但最后a million mints = 3333333.333333只有13个. 因为例子里的setf()是用来设置cout输出时的属性而不是数字本身的属性的.

-----------------------------------------------------------------------------

#include <iostream>

int main ()

{

std::cout.setf ( std::ios::hex, std::ios::basefield ); // set hex as the basefield

std::cout.setf ( std::ios::showbase ); // activate showbase

std::cout << 100 << '\n';

std::cout.unsetf ( std::ios::showbase ); // deactivate showbase

std::cout << 100 << '\n';

return 0;

}

-----------------------------------------------------------------------------

**Output:**

0x64

64

**Floating-Point Constants**

**By default**, **floating-point constants** such as 8.24 and 2.4E8 are type ***double***. If you want a constant to be type **float, you use an f or F suffix**. For type **long double, you use an l or L suffix**.

1.234f // a float constant

2.45E20F // a float constant

2.345324E28 // a double constant

2.2L // a long double constant

---------------------------------------

float a = 2.34E+22f;

float b = a + 1.0f;

cout << "a = " << a << endl;

cout << "b - a = " << b - a << endl;

---------------------------------------

最后结果可能是0，因为type float can represent only the first 6 or 7 digits (significant digits) in a number.

* **C++ Arithmetic Operators**

1. The **/** operator divides its first operand by the second. For example, **1000 / 5 evaluates to 200**. If both **operands are integers, the result is the integer portion of the quotient**. For example, **17 / 3 is 5**, with the fractional part discarded.
2. The % operator finds the modulus of its first operand with respect to the second. That is, **it produces the remainder of dividing the first by the second**. For example, **19 % 6 is 1** because 6 goes into 19 three times, with a remainder of 1. **Both operands must be integer types**; using the % operator with floating-point values causes a compile-time error.

You have yet to see the rest of the story about the division operator (/).The behavior of this operator depends on the type of the operands. If both operands are integers, C++ performs integer division. That means any fractional part of the answer is discarded, making the result an integer. If one or both operands are floating-point values, the fractional part is kept, making the result floating-point.

----------------------------------------------------------

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;

}

----------------------------------------------------------

**Output from the program:**

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

* **Type Conversions**

Assigning a large long value such as 2111222333 to float variable results in the loss of some precision. Because float can have just six significant figures, the value can be rounded to 2.11122E9.

A zero value assigned to a bool variable is converted to false, and a nonzero value is converted to true.

Assigning floating-point values to integer types poses a couple problems. First, converting floating-point to integer results in truncating the number (discarding the fractional part). Second, **a float value might be too big to fit in a cramped int variable**. In that case, **C++ doesn’t define what the result should be**; that means different implementation can respond differently.

cout.setf(ios\_base::fixed, ios\_base::floatfield);

float tree = 3; // int converted to float

int guess(3.9832); // double converted to int

int debt = 7.2E12; // result not defined in C++

cout << "tree = " << tree << endl; // 3.000000

cout << "guess = " << guess << endl; // 3

cout << "debt = " << debt << endl; **// 1634811904**

* **Initialization Conversions Using {}**

In particular, **list initialization doesn’t permit narrowing**, which is when the type of the variable may not be able to represent the assigned value. For example, conversions of floating types to integer.

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

**char c4 = {x}; // not allowed, x is not constant**

x = 31325;

char c5 = x; // allowed by this form of initialization

**For the initialization of c4**, **we know x has the value 66, but to the compiler**, **x is a variable** and conceivably **could have some other, much larger value**. It’s not the compiler’s job to keep track of what may have happened to x between the time it was initialized and the time it was used in the attempted initialization of c4.

* **Type Casts**

(typeName) value // converts value to typeName type

typeName (value) // converts value to typeName type

The first form is straight C. The second form is pure C++. The idea behind the new form is to make a type cast look like a function call.

More generally, you can do the following:

static\_cast<typeName> (value) // converts value to typeName type

* **auto Declarations in C++11**

C++11 introduces a facility that allows the compiler to deduce a type from the type of an initialization value. Just use auto instead of the type name in an initializing declaration, and the compiler assigns the variable the same type as that of the initializer:

auto n = 100; // n is int

auto x = 1.5; // x is double

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

auto x = 0.0; // ok, x is double because 0.0 is double

double y = 0; **// ok, 0 automatically converted to 0.0**

auto z = 0; **// oops, z is int because 0 is int**

**Using 0 instead of 0.0 doesn’t cause problems with explicit typing, but it does with automatic type conversion.**

Automatic type deduction becomes much more useful when dealing with complicated types, such as those in the STL.