Chapter 2. Variables and Basic Types

***2.4. const Qualifier***

const Objects Are Local to a File By Default. Nonconst variables are extern by default. To make a const variable accessible to other files we must explicitly specify that it is extern.

***2.5. References***

A const reference can be initialized to an object of a different type or to an rvalue, such as a literal constant:

int i = 42;

// legal for const references only

const int &r = 42;

const int &r2 = r + i;

The same initializations are not legal for nonconst references. Rather, they result in compile-time errors. The reason is subtle and warrants an explanation.

This behavior is easiest to understand when we look at what happens when we bind a reference to an object of a different type. If we write

double dval = 3.14;

const int &ri = dval;

the compiler transforms this code into something like this:

int temp = dval; // create temporary int from the double

const int &ri = temp; // bind ri to that temporary

If ri were not const, then we could assign a new value to ri. Doing so would not change dval but would instead change temp. To the programmer expecting that assignments to ri would change dval, it would appear that the change did not work. Allowing only const references to be bound to values requiring temporaries avoids the problem entirely because a const reference is read-only.

My Codes:

int main(int argc, char\* argv[])

{

int aa = 10;

const int &ref\_aa = aa;

double dd = 89.983;

const int &ref\_dd = dd;

int dd\_aa = dd;

cout << ref\_aa << endl;

cout << ref\_dd << endl;

cout << dd\_aa << endl;

cout << endl;

cout << "&aa = " << &aa << endl;

cout << "&ref\_aa = " << &ref\_aa << endl;

cout << "&dd = " << &dd << endl;

cout << "&ref\_dd = " << &ref\_dd << endl;

system("PAUSE");

return 0;

}

Output:

10

89

89

&aa = 0018FC24

&ref\_aa = 0018FC24

&dd = 0018FC08

&ref\_dd = 0018FBF0

***2.8. Class Types***

C++ supports a second keyword, [struct](mk:@MSITStore:D:\\Zhipeng\\books\\c++primer.chm::/0201721481/ch02lev1sec11.html" \l "gloss02_53), that can be used to define class types. The struct keyword is inherited from C.

If we define a class using the class keyword, then any members defined before the first access label are implicitly private; ifwe usethe struct keyword, then those members are public. Whether we define a class using the class keyword or the struct keyword affects only the default initial access level.

***2.9. Writing Our Own Header Files***

When designing a header it is essential to remember the difference between definitions, which may only occur once, and declarations, which may occur multiple times ([Section 2.3.5](mk:@MSITStore:D:\Zhipeng\books\c++primer.chm::/0201721481/ch02lev1sec3.html#ch02lev2sec14), p. [52](mk:@MSITStore:D:\Zhipeng\books\c++primer.chm::/0201721481/ch02lev1sec3.html#ch02lev2sec14)). The following statements are definitions and therefore should not appear in a header:

extern int ival = 10; // initializer, so it's a definition

double fica\_rate; // no extern, so it's a definition

Although ival is declared extern, it has an initializer, which means this statement is a definition. Similarly, the declaration of fica\_rate, although it does not have an initializer, is a definition because the extern keyword is absent. Including either of these definitions in two or more files of the same program will result in a linker error complaining about multiple definitions

There are three exceptions to the rule that headers should not contain definitions: classes, const objects whose value is known at compile time, and inline functions are all defined in headers.

**A Brief Introduction to the Preprocessor**

A #include directive takes a single argument: the name of a header. The pre-processor replaces each #include by the contents of the specified header. Our own headers are stored in files. System headers may be stored in a compiler-specific format that is more efficient.

Avoiding Multiple Inclusions

#ifndef SALESITEM\_H

#define SALESITEM\_H

// Definition of Sales\_itemclass and related functions goes here

#endif

Chapter 3. Library Types

***3.1. Namespace using Declarations***

#include <string>

#include <iostream>

using std::cin;

using std::string;

int main()

{

string s; // ok: string is now a synonym for std::string

cin >> s; // ok: cin is now a synonym for std::cin

cout << s; // error: no using declaration; we must use name

std::cout << s; // ok: explicitly use cout from namepsace std

}

***3.2. Library string Type***

|  |  |
| --- | --- |
| string s1; | Default constructor; s1 is the empty string |
| string s2(s1); | Initialize s2 as a copy of s1 |
| string s3("value"); | Initialize s3 as a copy of the string literal |
| string s4(n, 'c'); | Initialize s4 with n copies of the character 'c' |

**3.2.2. Reading and Writing strings**

// Note: #include and using declarations must be added to compile this code

int main()

{

string s; // empty string

cin >> s; // read whitespace-separated string into s

cout << s << endl; // write s to the output

return 0;

}

This program begins by defining a string named s. The next line,

cin >> s; // read whitespace-separated string into s

reads the standard input storing what is read into s. The string input operator:

* Reads and discards any leading whitespace (e.g., spaces, newlines, tabs)

It then reads characters until the next whitespace character is encountered

**Using getline to Read an Entire Line**

Unlike the input operator, getline does not ignore leading newlines. Whenever getline encounters a newline, even if it is the first character in the input, it stops reading the input and returns. The effect of encountering a newline as the first character in the input is that the string argument is set to the empty string.

int main()

{

string line;

// read line at time until end-of-file

while (getline(cin, line))

cout << line << endl;

return 0;

}

Any variable used to store the result from the string size operation ought to be of type string::size\_type. It is particularly important not to assign the return from size to an int.

## *Advice: Use the C++ Versions of C Library Headers*

In addition to facilities defined specifically for C++, the C++ library incorporates the C library. The <cctype> header makes available the C library functions defined in the C header file named <ctype.h>.

The standard C headers names use the form name.h. The C++ versions of these headers are named cname. The C++ versions remove the .h suffix and precede the name by the letter c. This indicates that the header originally comes from the C library. Hence, cctype has the same contents as ctype.h, but in a form that is appropriate for C++ programs. In particular, the names defined in the cname headers are defined inside the std namespace, whereas those defined in the .h versions are not.

***3.3. Library vector Type***

|  |  |
| --- | --- |
| vector<T> v1; | vector that holds objects of type T; |
|  | Default constructor v1 is empty |
| vector<T> v2(v1); | v2 is a copy of v1 |
| vector<T> v3(n, i); | v3 has n elements with value i |
| vector<T> v4(n); | v4 has n copies of a value-initialized object |

Although we can preallocate a given number of elements in a vector, it is usually more efficient to define an empty vector and add elements to it (as we'll learn how to do shortly).

An element must exist in order to subscript it; elements are not added when we assign through a subscript.

***3.4. Introducing Iterators***

The library defines an iterator type for each of the standard containers, including vector. Iterators are more general than subscripts: All of the library containers define iterator types, but only a few of them support subscripting. Because iterators are common to all containers, modern C++ programs tend to use iterators rather than subscripts to access container elements, even on types such as vector that support subscripting.

Each of the library container types defines a member named iterator that is a synonym for the actual type of its iterator.

My own words: Iterator can be interpreted as the address of one specific element in a container. So in order to get the value of the element, dereference operator \* should be used.

Each container defines a pair of functions named begin and end that return iterators. The iterator returned by the end operation does not denote an actual element in the container. Instead, it is used as a [sentinel](mk:@MSITStore:D:\\Zhipeng\\books\\c++primer.chm::/0201721481/ch03lev1sec7.html" \l "gloss03_16) indicating when we have processed all the elements in the container.

Iterators use the increment operator (++) to advance an iterator to the next element in the container.

Each container type also defines a type named const\_iterator, which should be used when reading, but not writing to, the container elements.

A const\_iterator should not be confused with an iterator that is const. When we declare an iterator as const we must initialize the iterator. Once it is initialized, we may not change its value:

vector<int> nums(10); // nums is nonconst

const vector<int>::iterator cit = nums.begin();

\*cit = 1; // ok: cit can change its underlying element

++cit; // error: can't change the value of cit

const vector<int> nines(10, 9); // cannot change elements in nines

// error: cit2 could change the element it refers to and nines is const

const vector<int>::iterator cit2 = nines.begin();

// ok: it can't change an element value, so it can be used with a const vector<int>

vector<int>::const\_iterator it = nines.begin();

\*it = 10; // error: \*it is const

++it; // ok: it isn't const so we can change its value

***3.5. Library bitset Type***

|  |  |
| --- | --- |
| bitset<n> b; | b has n bits, each bit is 0 |
| bitset<n> b(u); | b is a copy of the unsigned long value u |
| bitset<n> b(s); | b is a copy of the bits contained in string s |
| bitset<n> b(s, pos, n); | b is a copy of the bits in n characters from s starting from position pos |