Note of C++ Primer 5th Edition

john107

2017 年 1 月 18 日

目录

1 Getting started

1.1 Reading an unknown number of inputs

On Windows we enter an end-of-file by typing ctrl+z, while on Unix and Mac OS, ctrl+d

```
while (std::cin >> value){}
```

1.2 First look at I/O

- cin :standard input
- cout :standard output
- cerr :standard error
- clog :general information about the execution of the program

1.3 Comments

We often need to comment out a block of code during debugging. Because that code might contain nested comment pairs, the best way to comment a block of code is to insert single-line comments at the beginning of each line in the section we want to ignore.

1.4 Use User-Defined Class

Headers from the standard library are enclosed in angle brackets (<>). Those that are not part of the library are enclosed in double quotes (" ").

```
#include <iostream>
#include "Sales_item.h"
```

A member function is a function that is defined as part of a class. Member functions are sometimes referred to as method .

The dot operator applies only to objects of class type, its left-hand operand must be an object of class type, and its right-hand operand must name a member of that type.

2 Variables and Basic Types

2.1 Primitive Built-in Types

The Standard C++ library implement a rich library of class types and associated functions.

A few rules of thumb can be useful in deciding which type to use:

- Use an unsigned type when you know that the values cannot be negative
- Use int for integer arithmetic. short is usually too small and, in practice, long often has the same size as int. If your data values are larger than the minimum guaranteed size of an int, then use long long.
- Do not use plain char or bool in arithmetic expressions. Use them only to hold characters or truth values. Computations using char are especially problematic because char is signed on some mathines and unsigned on others. If you need a tiny integer, explicitly specify either signed char or unsigned char.
- Use double for floating-point computations; float usually does not have enough precision, and the cost of double-precision calculations versus single-precision is negligible. In fact, on some machines, double-precision operations are faster than single. The precision offered by long double usually is unnecessary and often entails considerable run-time cost.

2.2 Expressions Involving Unsigned Types

Regardless of whether one or both operands are unsigned, if we subtract a value from an unsigned, we must be sure that the result cannot be negative.

It is essential to remember that signed values are automatically converted to unsigned.

2.3 Literals

A value, such as 42, is known as a literal because its value self-evident.

Integer literals that begin with 0(zero) are interpreted as octal. Those that begin with either 0x or 0X are interpreted as hexadecimal.

Prefix	Meaning	Type
u	Unicode 16 character	char16_t
U	Unicode 32 character	char32_t
L	wide character	wchar_t
u8	utf-8(string literals only)	char

Character and Character String Literals

Sufix	Minimum Type
u or U	unsigned
l or L	long
ll or LL	long long

Integer Literals

Sufix	Minimum Type
f or F	float
l or L	long double

Floating-Point Literals

A character enclosed within single quotes is a literal of type char, Zero or more characters enclosed in double quotation marks is a string literal.

The type of a string literal is array of constant char$. The complier appends a null character (' \0') to every string literal. Thus, the actual size of a string literal is one more than its apparent size.

Two string literals that appear adjacent to one another and that are separated only by spaces, tabs, or new-lines are conactenated into a single literal.

```
std::cout<<" a really, really long string literal "
        " that spans two lines" <<std::endl;
```

newline	\n	horizontal tab	\t	alert(bell)	\a
vertical tab	\v	backspace	\	double quote	\"
backslash	\\	question mark	\?	single quote	\'
carriage return	\r	formfeed	\f		

Several escape sequences

We use an escape sequence as if it were a single character.

We can also write a generalized escape sequence, which is \x followed by one or more hexadecimal digits or a \followed by one, two, three octal digits. The value represents the numerical value of the character. For example:

\7	(bell)	\12	(newline)	\40	(blank)
\0	(null)	\115	(' M')	\x4d	(' M')

Note that if a \is followed by more than three octal digits, only the first three are associated with the \. In contrast, \x used up all the hex digits following it.

We can override the default type of an integer, floating-point, or character literal by supplying a suffix or prefix as listed below.

L' a'	wide character literal, type if wchar_1
u8" hi!"	utf-8 string literal (utf-8 encodes a unicode character in 8 bits)
42ULL	unsigned integer literal, type is unsigned long long
1E-3F	single precision floating-point literal, type is float
3.14159L	extended-precision floating-point literal, type is long double

2.4 Variables

Like iostream, string is defined in namespace std. The string library gives us several ways to initialize string objects.

It is tempting to think of initialization as a form of assignment, but initialization and assignment are different operations in C++.

2.5 List Initialization

All four ways of initialization below are allowed:

- int i=0;
- int i ={0};
- i(0);
- i{0};

The generalized use of curly braces for initialization was introduced as part of the new standard, and is referred to as List Initialization . When used with variables of built-in type, this form of initialization has one important property: The compiler will not let us list initialize variables of built-in type if the initializer might lead to the loss of information:

```
long double ld=3.1415926536;
int a{ld},b={ld}; //error:narrowing conversion required
int a(ld),d=ld; //ok, but value will be truncated
```

2.6 Variable Declarations and Definitions

A declaration makes a name known to the program. A definition creates the associated entity.

A variable declaration specifies the type and name of a variable. A variable definition is a declaration. In addition to specifying the name and type, a definition also allocates storage and may provide the variable with an initial value.

To obtain a declaration that is not also a definition, we add the extern keyword and may not provide an explicit initializer.

Any declaration that includes an explicit initializer is a definition. We can provide an initializer on a variable defined as extern, but doing so override the extern. An extern that has an initializer is a definition .

```
extern int i; //declares but does not define i
int j; //declares and defines j
extern int k = 1; //definition.
```

It is an error to provide an initializer on an extern inside a function.

2.7 Scope of a Name

The name main— like most names defined outside a function — has global scope. Names defined inside a function has block scope.

To fetch a global scope variable from inner scope, use ::identifier

2.8 Compound Types

A compound type is a type that is defined in terms of another type. C++ has several compound types, two of which are references and pointers .

A reference defines an alternative name for an object. A reference type ” refers to” another type. We define a reference type by writing a declarator of the form &d, where d is the name being declared:

```
int ival=1024;
int &refVal=ival; //refVal refers to (is another name for) ival
int &refVal2; //error:a reference must be initialized
```

Ordinarily, when we initialize a variable, the value of the initializer is copied into the object we are creating. When we define a reference, instead of copying the initializer’ s value, we bind the reference to its initializer. Once initialized a reference remains bound to its initial object. There is no way to rebind a referenceto refer to a different object. Because of that, reference must be initialized.

A reference is not an object, a reference is just another name for an already existing object.

```
int &refVal3 = refVal; //ok, refVal3 is bound to the object to which refVal is bound, i.e., to ival
int i = refVal; //ok, initialize i to the same value as ival
```

Because references are not objects, we may not define a refer to a reference.

A reference may be bound only to an object, not to a literal or to the result of a more general expression.

2.9 Pointers

A pointer is a compound type that " points to" another type.

Like references, pointers are used for indirect access to other objects.

Unlike a reference, a pointer is an object in its own right. Pointers can be assigned and copied; a single pointer can point to several different objects over its lifetime.

Unlike a reference, a pointer need not be initialized at the time it is defined.

Like other built-in types, pointers defined at block scope have undefined value if they are not initialized.

A pointer holds the address of another object. We get the address of an object by using the address-of operator (the & operator).

```
int ival=42;
int *p = &ival; //p holds the address of ival; p is a pointer to ival.
*p = 0; //* yields the object, we assign a new value to ival through p
cout<<*p;
```

Several ways to obtain a null pointer:

```
int *p1 = nullptr; //introduced by the new standard, equivalent to int *p1=0
int *p2 = 0; //directly initialize p2 from the literal constant 0;
int *p3 = NULL;//must #include <cstdlib>, equivalent to int *p3=0; NULL is defined as 0 by the cstdlib header, and is
    called a preprocessor variable
```

Because references are not objects, they don' t have addresses. Hence, we may not define a pointer to a reference.

It can be easier to understand complicated pointer or reference declaration if you read them from right to left.

```
int i=42;
int *p;
int *&r=p; //r si a reference to the pointer p
r=&i;
*r=0;
```

By default, const objects are local to a file. Sometimes we have a const variable that we want to share across multiple files but whose initializer is not a constant expression. To solve this problem and define a single instance of a const variable, we use the keyword extern on both its definition nd declaration(s):

```
//file_1.cc defines and initializes a const that is accessible to other files
extern const int bufSize=fcn(); //specify extern in order for bufSize to be used in other files
//file_1.h
extern const int bufSize; //extern signifies that bufSize is not local to file_1.h and that its definition will occur
    elsewhere.
```

2.9.1 const pointer VS pointer to const

```
double pi=3.14;
const double *ptr = &pi; /*ok,ptr thinks it points to a const double type object, but it may not be true. Yet you can
    not chnge pi through ptr, because ptr ptr thinks it points to a const double type object*/
double *const ptr2=&pi; //ok, ptr2 is a const pointer, it can not be changed to point to another object
```

It may be helpful to think of pointers and references to const as pointers or references " think they point or refer to const " .

2.10 constexpr and Constant Expressions

A constant expression is an expression whose value cannot change and that can be evaluated at compile time.

Whether a given object (or expression) is a constant expression depends on the types and the initializer.

Under the new standard, we can ask the complier to verify that a variable is a constant expression by declaring the variable in a constexpr declaration. Variables declared as constexpr are implicitly const and must be initialized by constant expression.

It is important to understand that when we define a pointer in a constexpr declaration, the constexpr specifier applies to the pointer, not the type to which the pointer points. The difference is a consequence of the fact that constexpr imposes a top-level const on the objects it defines.

```
const int *p=nullptr; //p is a pointer to a const int
constexpr int *q=nullptr;//q is a const pointer to int
```

2.11 Type Aliases

A type alias is a name that is a synonym for another type. We can define a type alias in two ways:

Traditionally, we use a typeof:

```
typeof double wages;//wages is a synonym for double
typeof wages base, *p;//base is a synonym for double, p for double*
```

The new standard introduced a second way to define a type alias, via an alias declaration:

```
using SI=Sales_item;//SI is a synonym for Sales_item
```

2.12 The auto Type Specifier

2.13 The decltype Type Specifier

It is worth noting that decltype is the only context in which a variable defined as a reference is not treated as a synonym for the object to which it refers.

2.14 Brief Introduction to the Preprocessor

The most common technique for making it safe to include a header multiple times relies on the preprocessor. C++ program use the preprocessor to define header guard. Header guards rely on preprocessor variables. Preprocessor variables have one of two possible state: defined or not defined. The #define directive takes a name and defines that name as a preprocessor variable. There are two other directives that test whether a given preprocessor variable has or has not been defined:#ifdef and #inndef.

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

3 Strings, Vectors, and Arrays

3.1 Library string Type

To use the string type, we must include the string header.

```
#include<string>
using std::string;
int main(){
  string s1;
  string s2=s1;
  string s3="hiya";
  string s4(10,'c'); //s4 is "cccccccccc";direct initialization
  string s5=string(10,'c');//copy initialization
}
```

os«s	Write s onto output os. Returns os.
is»s	Reads whitespace-separated
getLine(is,s)	Reads a line of input from is into s. Returns is.
s.empty()	Returns true if s is empty; otherwise returns false
s[n]	Returns a reference to the char at position n in s;position start at 0.
s1+s2	Returns a string that is a conactenation of s1 and s2
s1=s2	Replaces characters in s1 with a copy of s2
s1==s2	The strings s1 and s2 are equal if they contain the same characters
s1!=s2	Equality is canse-sensitive.
<,<=,>,>=	Comparisons are case-sensitive and use dictionary ordering.

string Operations