网站：http://www.cplusplus.com/doc/

**一、变量和类型**

**1.1 变量-初始化变量**

|  |  |
| --- | --- |
| // initialization of variables  #include  using namespace std;  int main ()  { int a=5; // initial value: 5  int b(3); // initial value: 3  int c{2}; // initial value: 2  int result; // initial value undetermined   a = a + b;   result = a - c;   cout << result;  return 0; } | 6 |

**1.2 类型推倒（Type deduction）: auto and decltype**

When a new variable is initialized, the compiler can figure out what the type of the variable is automatically by the initializer. For this, it suffices to use auto as the type specifier for the variable:

|  |  |  |
| --- | --- | --- |
| 1 2 | int foo = 0; auto bar = foo; // the same as: int bar = foo; |  |

Here, bar is declared as having an auto type; therefore, the type of bar is the type of the value used to initialize it: in this case it uses the type of foo, which is int.  
  
Variables that are not initialized can also make use of type deduction with the decltype specifier:

|  |  |
| --- | --- |
| 1 2 | int foo = 0; decltype(foo) bar; // the same as: int bar; |

**1.3 初始化字符串**

string mystring = "This is a string";

string mystring ("This is a string");

string mystring {"This is a string"};

**二 、常量**

**2.1 整数**

|  |  |
| --- | --- |
| **Suffix** | **Type modifier** |
| u *or* U | unsigned |
| l *or* L | long |
| ll *or* LL | long long |

75 // int

75u // unsigned int

75l // long

75ul // unsigned long

75lu // unsigned long

**2.2 浮点数**

3.14159 // 3.14159

6.02e23 // 6.02 x 10^23

1.6e-19 // 1.6 x 10^-19

3.0 // 3.0

**三 运算符**

**3.1 递增和递减**

|  |  |
| --- | --- |
| **Example 1** | **Example 2** |
| x = 3; y = ++x; // x contains 4, y contains 4 | x = 3; y = x++; // x contains 4, y contains 3 |

Here there are some examples:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 | (7 == 5) // evaluates to false (5 > 4) // evaluates to true (3 != 2) // evaluates to true (6 >= 6) // evaluates to true (5 < 5) // evaluates to false |  |

Of course, it's not just numeric constants that can be compared, but just any value, including, of course, variables. Suppose that a=2, b=3 and c=6, then:

|  |  |
| --- | --- |
| 1 2 3 4 | (a == 5) // evaluates to false, since a is not equal to 5 (a\*b >= c) // evaluates to true, since (2\*3 >= 6) is true (b+4 > a\*c) // evaluates to false, since (3+4 > 2\*6) is false ((b=2) == a) // evaluates to true |

**3.2逗点运算符（，）**

The comma operator (,) is used to separate two or more expressions that are included where only one expression is expected. When the set of expressions has to be evaluated for a value, only the right-most expression is considered.  
  
For example, the following code:

|  |  |
| --- | --- |
|  | a = (b=3, b+2); |

**结果：**a=5

**3.3 sizeof**

This operator accepts one parameter, which can be either a type or a variable, and returns the size in bytes of that type or object:

|  |  |  |
| --- | --- | --- |
|  | x = sizeof (char); |  |

Here, x is assigned the value 1, because char is a type with a size of one byte.

**四、标准输入\输出**

**4.1输入Standard input (cin)**

The extraction operator can be used on cin to get strings of characters in the same way as with fundamental data types:  
**输入单词，**

|  |  |  |
| --- | --- | --- |
| 1 2 | string mystring;  cin >> mystring; |  |

However, **cin extraction** always considers **spaces (whitespaces, tabs, new-line...) as terminating the value being extracted, and thus extracting a string means to always extract a single word, not a phrase or an** entire sentence.  
  
To get an entire line from cin, there exists a function, called **getline（输入句子）**, that takes the stream (cin) as first argument, and the string variable as second. For example:

|  |  |
| --- | --- |
| // cin with strings  #include <iostream>  #include <string>  using namespace std;  int main ()  { string mystr;   cout << "What's your name? ";   getline (cin, mystr);   cout << "Hello " << mystr << ".\n";   cout << "What is your favorite team? ";   getline (cin, mystr);   cout << "I like " << mystr << " too!\n";  return 0; } | What's your name? Homer Simpson Hello Homer Simpson. What is your favorite team? The Isotopes I like The Isotopes too! |

**五、控制流（while for switch等）**

**http://www.cplusplus.com/doc/tutorial/control/**

**六 函数和返回值**

**6.1 main函数的返回值**

ou may have noticed that the return type of main is int, but most examples in this and earlier chapters did not actually return any value from main.  
  
Well, there is a catch: If the execution of main ends normally without encountering a return statement the compiler assumes the function ends with an implicit return statement:

|  |  |  |
| --- | --- | --- |
|  | return 0; |  |

Note that this only applies to function main for historical reasons. All other functions with a return type shall end with a proper return statement that includes a return value, even if this is never used.  
  
When main returns zero (either implicitly or explicitly), it is interpreted by the environment as that the program ended successfully. Other values may be returned by main, and some environments give access to that value to the caller in some way, although this behavior is not required nor necessarily portable between platforms. The values for main that are guaranteed to be interpreted in the same way on all platforms are:

|  |  |
| --- | --- |
| **value** | **description** |
| 0 | The program was successful |
| [EXIT\_SUCCESS](http://www.cplusplus.com/EXIT_SUCCESS) | The program was successful (same as above). This value is defined in header [<cstdlib>](http://www.cplusplus.com/%3Ccstdlib%3E). |
| [EXIT\_FAILURE](http://www.cplusplus.com/EXIT_FAILURE) | The program failed. This value is defined in header [<cstdlib>](http://www.cplusplus.com/%3Ccstdlib%3E). |

Because the implicit return 0; statement for main is a tricky exception, some authors consider it good practice to explicitly write the statement.

**6.2传引用和传值**

|  |  |
| --- | --- |
| // passing parameters by reference  #include <iostream>  using namespace std;  void duplicate (int& a, int& b, int& c)  { a\*=2; b\*=2; c\*=2; }  int main ()  { int x=1, y=3, z=7;   duplicate (x, y, z);   cout << "x=" << x << ", y=" << y << ", z=" << z;   return 0; } | x=2, y=6, z=14 |

**6.3 默认值**

|  |  |
| --- | --- |
| // default values in functions  #include <iostream>  using namespace std;  int divide (int a, int b=2)   { int r; r=a/b; return (r); }  int main ()  { cout << divide (12) << '\n';   cout << divide (20,4) << '\n';   return 0;   } | 6 5 |

**七 重载和模板**

**7.1 重载（overloads）**

In C++, two different functions can have the same name if their parameters are different; either because they have a different number of parameters, or because any of their parameters are of a different type. For example:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 | // overloading functions  #include <iostream>  using namespace std;  int operate (int a, int b)  { return (a\*b); }  double operate (double a, double b)  { return (a/b); }  int main ()  { int x=5,y=2;   double n=5.0,m=2.0;   cout << operate (x,y) << '\n';   cout << operate (n,m) << '\n';  return 0; } | 10 2.5 |

**7.2 模板**

|  |  |  |  |
| --- | --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 | // overloaded functions  #include <iostream>  using namespace std;  int sum (int a, int b)  {  return a+b;  }  double sum (double a, double b)  {  return a+b;  }  int main ()  {  cout << sum (10,20) << '\n';  cout << sum (1.0,1.5) << '\n';  return 0;  } | 30  2.5 | [Edit & Run](http://www.cplusplus.com/doc/tutorial/functions2/) |

Here, sum is overloaded with different parameter types, but with the exact same body.  
  
The function sum could be overloaded for a lot of types, and it could make sense for all of them to have the same body. For cases such as this, C++ has the ability to define functions with generic types, known as *function templates*. Defining a function template follows the same syntax than a regular function, except that it is preceded by the template keyword and a series of template parameters enclosed in angle-brackets <>:  
  
template <template-parameters> function-declaration   
The template parameters are a series of parameters separated by commas. These parameters can be generic template types by specifying either the class or typename keyword followed by an identifier. This identifier can then be used in the function declaration as if it was a regular type. For example, a generic sum function could be defined as:

|  |  |
| --- | --- |
| 1 2 3 4 5 | template <class SomeType>  SomeType sum (SomeType a, SomeType b)  {  return a+b;  } |

**实际例子1：**

|  |  |
| --- | --- |
| // function template  #include <iostream>  using namespace std;  template <class T>  T sum (T a, T b)  {  T result;  result = a + b;  return result;  }  int main () {  int i=5, j=6, k;  double f=2.0, g=0.5, h;  k=sum<int>(i,j);  h=sum<double>(f,g);  cout << k << '\n';  cout << h << '\n';  return 0;  } | 11  2.5 |

**实际例子2：**

|  |  |  |
| --- | --- | --- |
| // function templates  #include <iostream>  using namespace std;  template <class T, class U>  bool are\_equal (T a, U b)  {  return (a==b);  }  int main ()  {  if (are\_equal(10,10.0))  cout << "x and y are equal\n";  else  cout << "x and y are not equal\n";  return 0;  } | x and y are equal | [Edit & Run](http://www.cplusplus.com/doc/tutorial/functions2/) |

**7.3 Non-type template arguments**

|  |  |  |  |
| --- | --- | --- | --- |
| 2 3 4 5 6 7 8 9 10 11 12 13 14 | // template arguments  #include <iostream>  using namespace std;  template <class T, int N>  T fixed\_multiply (T val)  {  return val \* N;  }  int main() {  std::cout << fixed\_multiply<int,2>(10) << '\n';  std::cout << fixed\_multiply<int,3>(10) << '\n';  } | 20  30 | [Edit & Run](http://www.cplusplus.com/doc/tutorial/functions2/) |

**8 Name Visible**

[**http://www.cplusplus.com/doc/tutorial/namespaces/**](http://www.cplusplus.com/doc/tutorial/namespaces/)

// using

#include <iostream>

using namespace std;

namespace first

{

int x = 5;

int y = 10;

}

namespace second

{

double x = 3.1416;

double y = 2.7183;

}

int main () {

using namespace first;

cout << x << '\n';

cout << y << '\n';

cout << second::x << '\n';

cout << second::y << '\n';

return 0;

}

**9数组**

**9.1 数组作为参数**

**/**/ arrays as parameters

#include <iostream>

using namespace std;

void printarray (int arg[], int length) {

for (int n=0; n<length; ++n)

cout << arg[n] << ' ';

cout << '\n';

}

int main ()

{

int firstarray[] = {5, 10, 15};

int secondarray[] = {2, 4, 6, 8, 10};

printarray (firstarray,3);

printarray (secondarray,5);

**9.2 Library Arrays**

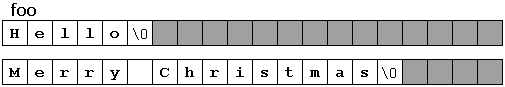
|  |  |
| --- | --- |
| **language built-in array** | **container library array** |
| #include <iostream>  using namespace std;  int main()  {  int myarray[3] = {10,20,30};  for (int i=0; i<3; ++i)  ++myarray[i];  for (int elem : myarray)  cout << elem << '\n';  } | #include <iostream>  #include <array>  using namespace std;  int main()  {  array<int,3> myarray {10,20,30};  for (int i=0; i<myarray.size(); ++i)  ++myarray[i];  for (int elem : myarray)  cout << elem << '\n';  } |

**10 字符序列（Character sequences）**

For example, the following array:

|  |  |  |
| --- | --- | --- |
|  | char foo [20]; |  |

is an array that can store up to 20 elements of type char. It can be represented as:  
  
http://www.cplusplus.com/doc/tutorial/ntcs/c_strings1.png

In this case, the array of 20 elements of type char called foo can be represented storing the character sequences"Hello" and "Merry Christmas" as:  
  


**10.1 空中止符字符串的初始化（Initialization of null-terminated character sequences）**

**效果相同**

|  |  |  |
| --- | --- | --- |
| 2 | char myword[] = { 'H', 'e', 'l', 'l', 'o', '\0' };  char myword[] = "Hello"; |  |

In both cases, the array of characters myword is declared with a size of 6 elements of type char: the 5 characters that compose the word "Hello", plus a final null character ('\0')

Expressions (once *myword* has already been declared as above), such as:  
一旦被初始化，下面的声明就无效

|  |  |
| --- | --- |
| 1 2 | myword = "Bye";  myword[] = "Bye"; |

would **not** be valid, like neither would be:

|  |  |  |
| --- | --- | --- |
|  | myword = { 'B', 'y', 'e', '\0' }; |  |

This is because arrays cannot be assigned values. Note, though, that each of its elements can be assigned a value individually. For example, this would be correct:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 | myword[0] = 'B';  myword[1] = 'y';  myword[2] = 'e';  myword[3] = '\0'; |  |

**10.2Strings and null-terminated character sequences**

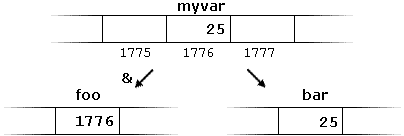
|  |  |  |
| --- | --- | --- |
| strings and NTCS:  #include <iostream>  #include <string>  using namespace std;  int main ()  {  char question1[] = "What is your name? ";  string question2 = "Where do you live? ";  char answer1 [80];  string answer2;  cout << question1;  cin >> answer1;  cout << question2;  cin >> answer2;  cout << "Hello, " << answer1;  cout << " from " << answer2 << "!\n";  return 0;  } | What is your name? Homer  Where do you live? Greece  Hello, Homer from Greece! | [Edit & Run](http://www.cplusplus.com/doc/tutorial/ntcs/) |

**11 指针（Pointer）**

**11.1 操作符地址（&）**

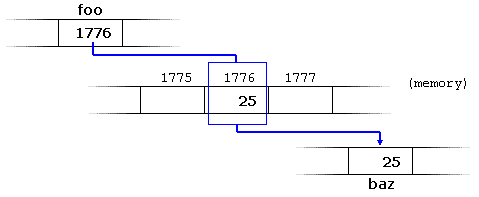
In this case, consider the following code fragment:

|  |  |  |
| --- | --- | --- |
| 1 2 3 | myvar = 25;  foo = &myvar;  bar = myvar; |  |

The values contained in each variable after the execution of this are shown in the following diagram:   
  


### 11.2 间接引用操作符（\*）Dereference operator (\*)

baz = \*foo;

his could be read as: "baz equal to value pointed to by foo", and the statement would actually assign the value 25to baz, since foo is 1776, and the value pointed to by 1776 (following the example above) would be 25.  
  


baz = foo; // baz equal to foo (1776)

baz = \*foo; // baz equal to value pointed to by foo (25)

The reference and dereference operators are thus complementary:

* & is the *address-of operator*, and can be read simply as "address of"
* \* is the *dereference operator*, and can be read as "value pointed to by"

**11.3 指针声明**

**例子1**

|  |  |
| --- | --- |
| // my first pointer  #include <iostream>  using namespace std;  int main ()  {  int firstvalue, secondvalue;  int \* mypointer;  mypointer = &firstvalue;  \*mypointer = 10;  mypointer = &secondvalue;  \*mypointer = 20;  cout << "firstvalue is " << firstvalue << '\n';  cout << "secondvalue is " << secondvalue << '\n';  return 0;  } | firstvalue is 10  secondvalue is 20 |

**例子2**

|  |  |
| --- | --- |
| // more pointers  #include <iostream>  using namespace std;  int main ()  {  int firstvalue = 5, secondvalue = 15;  int \* p1, \* p2;  p1 = &firstvalue; // p1 = address of firstvalue  p2 = &secondvalue; // p2 = address of secondvalue  \*p1 = 10; // value pointed to by p1 = 10  \*p2 = \*p1; // value pointed to by p2 = value pointed by p1  p1 = p2; // p1 = p2 (value of pointer is copied)  \*p1 = 20; // value pointed by p1 = 20    cout << "firstvalue is " << firstvalue << '\n';  cout << "secondvalue is " << secondvalue << '\n';  return 0;  } | firstvalue is 10  secondvalue is 20 |

**11.4 指针和数组（Pointer and Arrays）**

|  |  |
| --- | --- |
| // more pointers  #include <iostream>  using namespace std;  int main ()  {  int numbers[5];  int \* p;  p = numbers; \*p = 10;  p++; \*p = 20;  p = &numbers[2]; \*p = 30;  p = numbers + 3; \*p = 40;  p = numbers; \*(p+4) = 50;  for (int n=0; n<5; n++)  cout << numbers[n] << ", ";  return 0;  } | 10, 20, 30, 40, 50, |

**11.4 指针初始化**

Pointers can be initialized to point to specific locations at the very moment they are defined:

|  |  |  |
| --- | --- | --- |
| 1 2 | int myvar;  int \* myptr = &myvar; |  |

The resulting state of variables after this code is the same as after:

|  |  |
| --- | --- |
| 1 2 3 | int myvar;  int \* myptr;  myptr = &myvar |

When pointers are initialized, what is initialized is the address they point to (i.e., myptr), never the value being pointed (i.e., \*myptr). Therefore, the code above shall not be confused with:

**下面是不对的：**

|  |  |
| --- | --- |
| 1 2 3 | int myvar;  int \* myptr;  \*myptr = &myvar; |

### 11.5 指针运算Pointer arithmetics

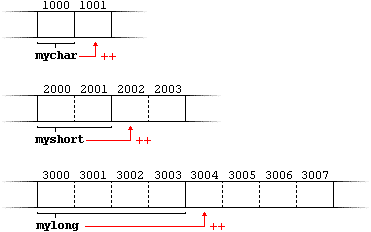
. For example, let's **imagine** that in a given system, **char takes 1 byte, short takes 2 bytes, and long takes 4**.Suppose now that we define three pointers in this compiler: 

|  |  |  |
| --- | --- | --- |
| 1 2 3 | char \*mychar;  short \*myshort;  long \*mylong; |  |

and that we know that they point to the memory locations 1000, 2000, and 3000, respectively.   
  
Therefore, if we write:

|  |  |  |
| --- | --- | --- |
| 1 2 3 | ++mychar; 地址加1  ++myshort; 地址加2  ++mylong; 地址加4 |  |

mychar, as one would expect, would contain the value 1001. But not so obviously, myshort would contain the value 2002, and mylong would contain 3004, even though they have each been incremented only once



**++操作符的优先级高于\*，所以\*p++相当于（\*（p++））。实际结果先取p指向的内容，指针p再+1（指向下一个位置）.**

**例子1**

|  |  |
| --- | --- |
| 1 2 3 4 | \*p++ // same as \*(p++): increment pointer, and dereference unincremented address  \*++p // same as \*(++p): increment pointer, and dereference incremented address  ++\*p // same as ++(\*p): dereference pointer, and increment the value it points to  (\*p)++ // dereference pointer, and post-increment the value it points to |

**例子2 ：**

\*p++ = \*q++ 相当于

\*p = \*q;

++p;

++q;

### 11.6 指针和const（Pointers and const）

**11.6.1**

int x;

int y = 10;

const int \* p = &y;

x = \*p; // ok: reading p

\*p = x; // error: modifying p, which is const-qualified

**11.6.2**

|  |  |
| --- | --- |
| // pointers as arguments:  #include <iostream>  using namespace std;  void increment\_all (int\* start, int\* stop)  {  int \* current = start;  while (current != stop) {  ++(\*current); // increment value pointed  ++current; // increment pointer  }  }  void print\_all (const int\* start, const int\* stop)  {  const int \* current = start;  while (current != stop) {  cout << \*current << '\n';  ++current; // increment pointer  }  }  int main ()  {  int numbers[] = {10,20,30};  increment\_all (numbers,numbers+3);  print\_all (numbers,numbers+3);  return 0;  } | 11  21  31 |

**11.6.3**

int x;

int \* p1 = &x; // non-const pointer to non-const int

const int \* p2 = &x; // non-const pointer to const int

int \* const p3 = &x; // const pointer to non-const int

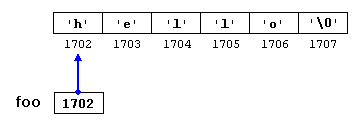
const int \* const p4 = &x; // const pointer to const int

const int \* p2a = &x; // non-const pointer to const int

int const \* p2b = &x; // also non-const pointer to const int

### 11.7 Pointers and string literals

|  |  |
| --- | --- |
| const char \* foo = "hello"; |  |

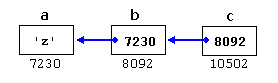
This declares an array with the literal representation for "hello", and then a pointer to its first element is assigned to foo. If we imagine that "hello" is stored at the memory locations that start at address 1702, we can represent the previous declaration as:  
   
Note that here foo is a pointer and contains the value 1702, and not 'h', nor "hello", although 1702 indeed is the address of both of these.  
  
The pointer foo points to a sequence of characters. And because pointers and arrays behave essentially in the same way in expressions, foo can be used to access the characters in the same way arrays of null-terminated character sequences are. For example:

|  |  |  |
| --- | --- | --- |
| 1 2 | \*(foo+4)  foo[4] |  |

Both expressions have a value of 'o' (the fifth element of the array).

### 11.8 指向指针的指针（Pointers to pointers）

|  |  |
| --- | --- |
| char a;  char \* b;  char \*\* c;  a = 'z';  b = &a;  c = &b; |  |

This, assuming the randomly chosen memory locations for each variable of 7230, 8092, and 10502, could be represented as:  
  


### 11.9 无类型指针（void pointer）

The void type of pointer is a special type of pointer. In C++, void represents the absence of type. Therefore, voidpointers are pointers that point to a value that has no type (and thus also an undetermined length and undetermined dereferencing properties).

|  |  |
| --- | --- |
| // increaser  #include <iostream>  using namespace std;  void increase (void\* data, int psize)  {  if ( psize == sizeof(char) )  { char\* pchar; pchar=(char\*)data; ++(\*pchar); }  else if (psize == sizeof(int) )  { int\* pint; pint=(int\*)data; ++(\*pint); }  }  int main ()  {  char a = 'x';  int b = 1602;  increase (&a,sizeof(a));  increase (&b,sizeof(b));  cout << a << ", " << b << '\n';  return 0;  } | y, 1603 |

### 11.10 无效指针和空指针（Invalid pointers and null pointers）

无效指针

int \* p; // uninitialized pointer (local variable)

int myarray[10];

int \* q = myarray+20; // element out of bounds

空指针

|  |  |  |
| --- | --- | --- |
| 1 2 | int \* p = 0;  int \* q = nullptr; |  |

Here, both p and q are *null pointers*, meaning that they explicitly point to nowhere, and

[**http://www.cplusplus.com/doc/tutorial/pointers/**](http://www.cplusplus.com/doc/tutorial/pointers/)

**11.11 函数指针（pointers to function）**

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 | // pointer to functions  #include <iostream>  using namespace std;  int addition (int a, int b)  { return (a+b); }  int subtraction (int a, int b)  { return (a-b); }  int operation (int x, int y, int (\*functocall)(int,int))  {  int g;  g = (\*functocall)(x,y);  return (g);  }  int main ()  {  int m,n;  int (\*minus)(int,int) = subtraction;  m = operation (7, 5, addition);  n = operation (20, m, minus);  cout <<n;  return 0;  } | 8 |

**d**

**d**

**d**