Pointers

In earlier chapters, variables have been explained as locations in the computer's memory which can be accessed by their identifier (their name). This way, the program does not need to care about the physical address of the data in memory; it simply uses the identifier whenever it needs to refer to the variable.

在前面的章节中，变量被解释为计算机内存中的位置，可以通过它们的标识符(它们的名字)来访问。这样，程序就不需要关心内存中数据的物理地址;它只是在需要引用变量时使用标识符。

For a C++ program, the memory of a computer is like a succession of memory cells, each one byte in size, and each with a unique address. These single-byte memory cells are ordered in a way that allows data representations larger than one byte to occupy memory cells that have consecutive addresses.

对于一个c++程序，计算机的内存就像一个连续的内存单元，每个字节大小，每个都有一个唯一的地址。这些单字节存储单元的排列方式允许大于一个字节的数据表示占用具有连续地址的存储单元。

This way, each cell can be easily located in the memory by means of its unique address. For example, the memory cell with the address `1776` always follows immediately after the cell with address `1775` and precedes the one with `1777`, and is exactly one thousand cells after `776` and exactly one thousand cells before `2776`.

通过这种方式，每个单元格都可以通过其唯一地址轻松地定位到内存中。例如，地址为1776的存储单元总是紧跟在地址为1775的存储单元之后，在地址为1777的存储单元之前，并且在776之后恰好是一千个细胞，在2776之前恰好是一千个细胞。

When a variable is declared, the memory needed to store its value is assigned a specific location in memory (its memory address). Generally, C++ programs do not actively decide the exact memory addresses where its variables are stored. Fortunately, that task is left to the environment where the program is run - generally, an operating system that decides the particular memory locations on runtime. However, it may be useful for a program to be able to obtain the address of a variable during runtime in order to access data cells that are at a certain position relative to it.

当声明一个变量时，需要存储它的值的内存被分配到内存中的一个特定位置(它的内存地址)。通常，c++程序不会主动决定变量存储的确切内存地址。幸运的是，这个任务留给了程序运行的环境—通常是一个操作系统，它决定运行时的特定内存位置。然而，对于程序来说，能够在运行时获取变量的地址，以便访问相对于该变量处于某个位置的数据单元，这可能是有用的。

Address-of operator (&)

The address of a variable can be obtained by preceding the name of a variable with an ampersand sign (`&`), known as *address-of operator*. For example:

变量的地址可以在变量名前面加上一个&符号(' & ')，称为*address-of操作符*。例如:

```
foo = &myvar;
```

This would assign the address of variable `myvar` to `foo`; by preceding the name of the variable `myvar` with the *address-of operator* (`&`), we are no longer assigning the content of the variable itself to `foo`, but its address.

这将把变量myvar的地址赋给foo;通过在变量myvar的名称前面加上地址操作符(&)，我们不再把变量本身的内容赋给foo，而是把它的地址赋给它。

The actual address of a variable in memory cannot be known before runtime, but let's assume, in order to help clarify some concepts, that `myvar` is placed during runtime in the memory address `1776`.

内存中变量的实际地址在运行时之前是无法知道的，但是为了帮助澄清一些概念，我们假设myvar在运行时被放置在内存地址1776中。

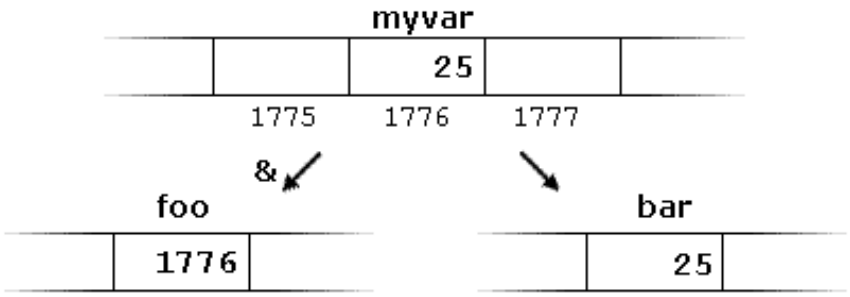
In this case, consider the following code fragment:

在这种情况下，考虑以下代码片段:

```
myvar = 25;
foo = &myvar;
bar = myvar;
```

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

执行后每个变量所包含的值如下图所示:



First, we have assigned the value `25` to `myvar` (a variable whose address in memory we assumed to be `1776`).

首先，我们将值25赋给myvar(一个在内存中地址为1776的变量)。

The second statement assigns `foo` the address of `myvar`, which we have assumed to be `1776`.

第二个语句给foo赋值myvar的地址，我们假设它是1776。

Finally, the third statement, assigns the value contained in `myvar` to `bar`. This is a standard assignment operation, as already done many times in earlier chapters.

最后，第三条语句将myvar中包含的值赋给bar。这是一个标准的赋值操作，在前面的章节中已经做过很多次了。

The main difference between the second and third statements is the appearance of the *address-of operator* (`&`).

第二个和第三个语句之间的主要区别是地址操作符(&)的外观。

The variable that stores the address of another variable (like `foo` in the previous example) is what in C++ is called a *pointer*. Pointers are a very powerful feature of the language that has many uses in lower level programming. A bit later, we will see how to declare and use pointers.

存储另一个变量(如前面例子中的foo)地址的变量在c++中被称为指针。指针是该语言的一个非常强大的特性，在低级编程中有很多用途。稍后，我们将看到如何声明和使用指针。

Dereference operator (*)

As just seen, a variable which stores the address of another variable is called a *pointer*. Pointers are said to "point to" the variable whose address they store.

正如刚才看到的，存储另一个变量地址的变量称为*指针*。指针被称为“指向”它们所存储的地址的变量。

An interesting property of pointers is that they can be used to access the variable they point to directly. This is done by preceding the pointer name with the *dereference operator* (`*`). The operator itself can be read as "value pointed to by".

指针的一个有趣的特性是，它们可以用来直接访问它们所指向的变量。这是通过在指针名前面加上*解引用操作符*(' * ')来实现的。操作符本身可以读为“由所指向的值”。

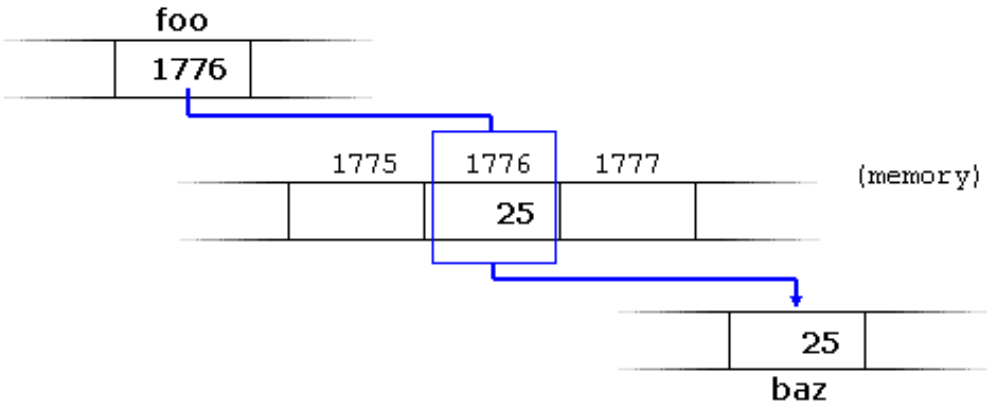
Therefore, following with the values of the previous example, the following statement:

因此，在前一个例子的值后面，下面的语句:

```
baz = *foo;
```

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

这可以被理解为:“baz等于foo指向的值”，并且该语句实际上会将值25赋给baz，因为foo是1776，而1776指向的值(下面的例子)将是25。



It is important to clearly differentiate that `foo` refers to the value `1776`, while `*foo` (with an asterisk `*` preceding the identifier) refers to the value stored at address `1776`, which in this case is `25`. Notice the difference of including or not including the *dereference operator* (I have added an explanatory comment of how each of these two expressions could be read):

重要的是要清楚地区分foo指向的值1776，而*foo(标识符前面有星号)*指向的是存储在地址1776的值，在本例中是25。请注意包含或不包含解引用操作符的区别(我添加了解释性注释，说明这两个表达式的读取方式):

```
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:

因此，引用和解引用操作符是互补的:

- `&` is the *address-of operator*, and can be read simply as "address of"
- &为操作符的地址，可以简单地读作“地址”。
- `*` is the *dereference operator*, and can be read as "value pointed to by"
- *是解引用操作符，可以读为“指向的值”。

Thus, they have sort of opposite meanings: An address obtained with `&` can be dereferenced with `*`.

因此，它们具有相反的含义:用&获得的地址可以用*解除引用。

Earlier, we performed the following two assignment operations:

前面，我们执行了以下两个赋值操作:

```
myvar = 25;
foo = &myvar;
```

Right after these two statements, all of the following expressions would give true as result:

在这两个命题之后，下列所有表达式的结果都是true:

```
myvar == 25
&myvar == 1776
foo == 1776
*foo == 25
```

The first expression is quite clear, considering that the assignment operation performed on `myvar` was `myvar=25`. The second one uses the address-of operator (`&`), which returns the address of `myvar`, which we assumed it to have a value of `1776`. The third one is somewhat obvious, since the second expression was true and the assignment operation performed on `foo` was `foo=&myvar`. The fourth expression uses the *dereference operator* (`*`) that can be read as "value pointed to by", and the value pointed to by `foo` is indeed `25`.

考虑到在myvar上执行的赋值操作是myvar=25，第一个表达式非常清楚。第二个使用地址操作符(&)，它返回myvar的地址，我们假设它的值为1776。第三个比较明显，因为第二个表达式是true，在foo上执行的赋值操作是foo=&myvar。第四个表达式使用了解引用操作符(*)，可以读作“被指向的值”，而foo指向的值确实是25。

So, after all that, you may also infer that for as long as the address pointed to by `foo` remains unchanged, the following expression will also be true:

所以，在所有这些之后，你还可以推断，只要foo指向的地址保持不变，下面的表达式也将为真:

```
*foo == myvar
```

Declaring pointers

Due to the ability of a pointer to directly refer to the value that it points to, a pointer has different properties when it points to a `char` than when it points to an `int` or a `float`. Once dereferenced, the type needs to be known. And for that, the declaration of a pointer needs to include the data type the pointer is going to point to.

由于指针可以直接指向它所指向的值，所以当指针指向char类型时，它的属性与指向int或float类型时不同。解除引用后，需要知道类型。为此，指针的声明需要包含指针要指向的数据类型。

The declaration of pointers follows this syntax:

指针的声明遵循以下语法:

```
type * name;
```

where `type` is the data type pointed to by the pointer. This type is not the type of the pointer itself, but the type of the data the pointer points to. For example:

其中type是指针所指向的数据类型。这个类型不是指针本身的类型，而是指针所指向的数据的类型。例如:

```
int * number;
char * character;
double * decimals;
```

These are three declarations of pointers. Each one is intended to point to a different data type, but, in fact, all of them are pointers and all of them are likely going to occupy the same amount of space in memory (the size in memory of a pointer depends on the platform where the program runs). Nevertheless, the data to which they point to do not occupy the same amount of space nor are of the same type: the first one points to an `int`, the second one to a `char`, and the last one to a `double`. Therefore, although these three example variables are all of them pointers, they actually have different types: `int*`, `char*`, and `double*` respectively, depending on the type they point to.

这是指针的三个声明。每个指针都指向不同的数据类型，但实际上，它们都是指针，并且都可能占用相同的内存空间(指针在内存中的大小取决于程序运行的平台)。然而，它们所指向的数据并不占用相同的空间，也不属于相同的类型:第一个指向int型，第二个指向char型，最后一个指向double型。因此，尽管这三个示例变量都是指针，但它们实际上有不同的类型:int、*char*和double*，这取决于它们所指向的类型。

Note that the asterisk (`*`) used when declaring a pointer only means that it is a pointer (it is part of its type compound specifier), and should not be confused with the *dereference operator* seen a bit earlier, but which is also written with an asterisk (`*`). They are simply two different things represented with the same sign.

注意，声明指针时使用的星号()*只意味着它是一个指针(它是其类型复合说明符的一部分)，不应该与前面看到的解引用操作符混淆，但它也是用星号()*编写的。它们只是用同一个符号表示的两个不同的东西。

Let's see an example on pointers:

让我们看一个关于指针的例子:

```
// 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;
}
```

Notice that even though neither `firstvalue` nor `secondvalue` are directly set any value in the program, both end up with a value set indirectly through the use of `mypointer`. This is how it happens:

注意，即使firstvalue和secondvalue都没有在程序中直接设置任何值，它们最终都通过mypointer的使用间接地设置了一个值。事情是这样发生的:

First, `mypointer` is assigned the address of firstvalue using the address-of operator (`&`). Then, the value pointed to by `mypointer` is assigned a value of `10`. Because, at this moment, `mypointer` is pointing to the memory location of `firstvalue`, this in fact modifies the value of `firstvalue`.

首先，使用address-of操作符(&)为mypointer分配firstvalue的地址。然后，mypointer所指向的值被赋值为10。因为，此时，mypointer正指向firstvalue的内存位置，这实际上修改了firstvalue的值。

In order to demonstrate that a pointer may point to different variables during its lifetime in a program, the example repeats the process with `secondvalue` and that same pointer, `mypointer`.

为了证明一个指针在程序的生命周期内可以指向不同的变量，这个例子用secondvalue和同一个指针mypointer重复了这个过程。

Here is an example a little bit more elaborated:

下面是一个更详细的例子:

```
// 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 to by p1
  p1 = p2;           // p1 = p2 (value of pointer is copied)
  *p1 = 20;          // value pointed to by p1 = 20
```

```
  cout << "firstvalue is " << firstvalue << '\n';
  cout << "secondvalue is " << secondvalue << '\n';
  return 0;
}
```

Each assignment operation includes a comment on how each line could be read: i.e., replacing ampersands (`&`) by "address of", and asterisks (`*`) by "value pointed to by".

每个赋值操作都包含了一行如何读取的注释:例如，用"address of"替换&号，用"value pointing by"替换*号。

Notice that there are expressions with pointers `p1` and `p2`, both with and without the *dereference operator* (`*`). The meaning of an expression using the *dereference operator* (*) is very different from one that does not. When this operator precedes the pointer name, the expression refers to the value being pointed, while when a pointer name appears without this operator, it refers to the value of the pointer itself (i.e., the address of what the pointer is pointing to).

注意，有带有指针p1和p2的表达式，有带解引用操作符(*)*的表达式也有不带解引用操作符的表达式。使用解引用操作符(*)*的表达式与不使用解引用操作符的表达式的含义非常不同。当这个操作符位于指针名之前时，表达式指向指针所指向的值，而当指针名没有使用这个操作符时，它指向指针本身的值(即指针所指向的地址)。

Another thing that may call your attention is the line:

另一件可能会引起你注意的事是台词:

```
int * p1, * p2;
```

This declares the two pointers used in the previous example. But notice that there is an asterisk (`*`) for each pointer, in order for both to have type `int*` (pointer to `int`). This is required due to the precedence rules. Note that if, instead, the code was:

这声明了前面示例中使用的两个指针。但是请注意，每个指针都有星号(*)，以便两者都具有int*类型(指向int的指针)。由于优先级规则，这是必需的。请注意，如果代码是:

```
int * p1, p2;
```

`p1` would indeed be of type `int*`, but `p2` would be of type `int`. Spaces do not matter at all for this purpose. But anyway, simply remembering to put one asterisk per pointer is enough for most pointer users interested in declaring multiple pointers per statement. Or even better: use a different statement for each variable.

P1的类型确实是int*，但是p2的类型是int。对于这个目的，空间根本不重要。但是无论如何，只要记住在每个指针上放一个星号，对于大多数想要在每个语句中声明多个指针的指针用户来说就足够了。或者更好的做法是:对每个变量使用不同的语句。

Pointers and arrays

The concept of arrays is related to that of pointers. In fact, arrays work very much like pointers to their first elements, and, actually, an array can always be implicitly converted to the pointer of the proper type. For example, consider these two declarations:

数组的概念与指针有关。事实上，数组的工作方式非常类似于指向第一个元素的指针，而且数组总是可以隐式地转换为适当类型的指针。例如，考虑以下两个声明:

```
int myarray [20];
int * mypointer;
```

The following assignment operation would be valid:

下面的赋值操作是有效的:

```
mypointer = myarray;
```

After that, `mypointer` and `myarray` would be equivalent and would have very similar properties. The main difference being that `mypointer` can be assigned a different address, whereas `myarray` can never be assigned anything, and will always represent the same block of 20 elements of type `int`. Therefore, the following assignment would not be valid:

之后，mypointer和myarray就等价了，并且有非常相似的属性。主要的区别是mypointer可以被分配一个不同的地址，而myarray永远不能被分配任何东西，并且总是代表同一个由20个int类型的元素组成的块。因此，下列赋值将无效:

```
myarray = mypointer;
```

Let's see an example that mixes arrays and pointers:

```
// 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;
}
```

Pointers and arrays support the same set of operations, with the same meaning for both. The main difference being that pointers can be assigned new addresses, while arrays cannot.

指针和数组支持相同的操作集，两者的含义相同。主要的区别在于指针可以被分配新的地址，而数组则不能。

In the chapter about arrays, brackets (`[]`) were explained as specifying the index of an element of the array. Well, in fact these brackets are a dereferencing operator known as *offset operator*. They dereference the variable they follow just as `*` does, but they also add the number between brackets to the address being dereferenced. For example:

在关于数组的一章中，括号([])被解释为指定数组元素的下标。实际上，这些括号是一个解引用操作符，称为偏移操作符。它们会像*一样解除对后面的变量的引用，但它们也会将括号之间的数字加到要解除引用的地址上。例如:

```
a[5] = 0;       // a [offset of 5] = 0
*(a+5) = 0;     // pointed to by (a+5) = 0
```

These two expressions are equivalent and valid, not only if `a` is a pointer, but also if `a` is an array. Remember that if an array, its name can be used just like a pointer to its first element.

这两个表达式等价且有效，不仅当a是指针时有效，而且当a是数组时也有效。记住，如果一个数组，它的名称可以像指向它的第一个元素的指针一样使用。

Pointer initialization

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

指针可以在定义时初始化为指向特定位置的指针:

```
int myvar;
int * myptr = &myvar;
```

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

这段代码之后的变量结果状态与之后相同:

```
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:

当指针初始化时，初始化的是指针指向的地址(例如，myptr)，而不是指针指向的值(例如，*myptr)。因此，上述代码不应与下列代码混淆:

```
int myvar;
int * myptr;
*myptr = &myvar;
```

Which anyway would not make much sense (and is not valid code).

无论如何，这都没有多大意义(并且不是有效的代码)。

The asterisk (`*`) in the pointer declaration (line 2) only indicates that it is a pointer, it is not the dereference operator (as in line 3). Both things just happen to use the same sign: `*`. As always, spaces are not relevant, and never change the meaning of an expression.

指针声明(第2行)中的星号(*)仅表明它是一个指针，而不是解引用操作符(如第3行)。这两个操作符恰好使用了相同的符号:。与往常一样，空格是无关的，并且永远不会改变表达式的含义。

Pointers can be initialized either to the address of a variable (such as in the case above), or to the value of another pointer (or array):

指针可以初始化为变量的地址(如上面的例子)，也可以初始化为另一个指针(或数组)的值:

```
int myvar;
int *foo = &myvar;
int *bar = foo;
```

Pointer arithmetics

To conduct arithmetical operations on pointers is a little different than to conduct them on regular integer types. To begin with, only addition and subtraction operations are allowed; the others make no sense in the world of pointers. But both addition and subtraction have a slightly different behavior with pointers, according to the size of the data type to which they point.

对指针进行算术操作与对普通整数类型进行算术操作略有不同。首先，只允许加减运算;其他的在指针的世界里没有意义。但是，根据它们所指向的数据类型的大小，加法和减法对指针的行为略有不同。

When fundamental data types were introduced, we saw that types have different sizes. For example: `char` always has a size of 1 byte, `short` is generally larger than that, and `int` and `long` are even larger; the exact size of these being dependent on the system. For example, let's imagine that in a given system, `char` takes 1 byte, `short` takes 2 bytes, and `long` takes 4.

在引入基础数据类型时，我们看到类型有不同的大小。例如:char的大小总是1字节，short通常比他大，int和long甚至更大;它们的确切大小取决于系统。例如，假设在给定的系统中，char占用1个字节，short占用2个字节，long占用4个字节。

Suppose now that we define three pointers in this compiler:

假设现在我们在这个编译器中定义了三个指针:

```
char *mychar;
short *myshort;
long *mylong;
```

and that we know that they point to the memory locations `1000`, `2000`, and `3000`, respectively.

我们知道它们分别指向内存位置1000 2000 3000。

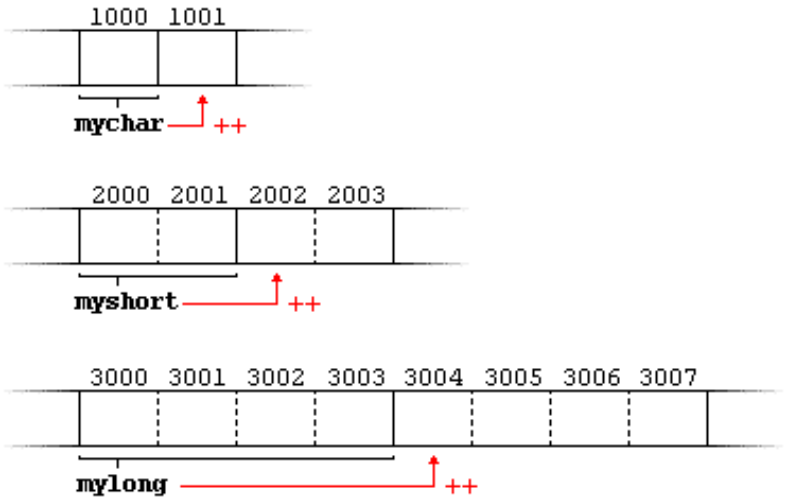
Therefore, if we write:

因此，如果我们写:

```
++mychar;
++myshort;
++mylong;
```

`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. The reason is that, when adding one to a pointer, the pointer is made to point to the following element of the same type, and, therefore, the size in bytes of the type it points to is added to the pointer.

如人们所料，Mychar将包含值1001。但不是很明显，myshort将包含值2002,mylong将包含值3004，尽管它们每个只增加了一次。原因是，当给指针加1时，指针将指向同一类型的下一个元素，因此，指针所指向的类型的字节大小将被添加到指针上。



This is applicable both when adding and subtracting any number to a pointer. It would happen exactly the same if we wrote:

这适用于对指针进行任何数字的加减运算。如果我们写:

```
mychar = mychar + 1;
myshort = myshort + 1;
mylong = mylong + 1;
```

Regarding the increment (`++`) and decrement (`--`) operators, they both can be used as either prefix or suffix of an expression, with a slight difference in behavior: as a prefix, the increment happens before the expression is evaluated, and as a suffix, the increment happens after the expression is evaluated. This also applies to expressions incrementing and decrementing pointers, which can become part of more complicated expressions that also include dereference operators (`*`). Remembering operator precedence rules, we can recall that postfix operators, such as increment and decrement, have higher precedence than

prefix operators, such as the dereference operator (`*`). Therefore, the following expression:

对于increment(++)和increment(——)操作符，它们都可以用作表达式的前缀或后缀，但行为略有不同:作为前缀，increment发生在表达式求值之前，作为后缀，increment发生在表达式求值之后。这也适用于指针自增和自减的表达式，它们可以成为包含解引用操作符()*的更复杂表达式的一部分。记住操作符优先级规则，我们可以回忆起后缀操作符(如自增和自减)的优先级高于前缀操作符(如解引用操作符*())。因此，表达式如下:

```
*p++
```

is equivalent to `*(p++)`. And what it does is to increase the value of `p` (so it now points to the next element), but because `++` is used as postfix, the whole expression is evaluated as the value pointed originally by the pointer (the address it pointed to before being incremented).

等价于*(p++)。它所做的是增加p的值(因此它现在指向下一个元素)，但是由于++用作后缀，整个表达式的值被计算为指针最初指向的值(它在递增之前指向的地址)。

Essentially, these are the four possible combinations of the dereference operator with both the prefix and suffix versions of the increment operator (the same being applicable also to the decrement operator):

从本质上讲，这是解引用操作符与自增操作符的前缀和后缀版本的四种可能组合(同样适用于自减操作符):

```
*p++   // same as *(p++): increment pointer, and dereference unincremented address 自增指针和解引用未自增地址
*++p   // same as *(++p): increment pointer, and dereference incremented address 自增指针和解引用自增地址
++*p   // same as ++(*p): dereference pointer, and increment the value it points to 解引用指针，并使其指向的值递增
(*p)++ // dereference pointer, and post-increment the value it points to  解引用指针，并对指针所指向的值进行后加
```

A typical -but not so simple- statement involving these operators is:

包含这些操作符的一个典型但不那么简单的语句是:

```
*p++ = *q++;
```

Because `++` has a higher precedence than `*`, both `p` and `q` are incremented, but because both increment operators (`++`) are used as postfix and not prefix, the value assigned to `*p` is `*q` before both `p` and `q` are incremented. And then both are incremented. It would be roughly equivalent to:

因为++的优先级高于`*`，所以p和q都是自增的，但是因为两个自增操作符(++)都用作后缀而不是前缀，所以在p和q都被自增之前，赋给*p的值是*q。然后两者都加1。它大致相当于:

```
*p = *q;
++p;
++q;
```

Like always, parentheses reduce confusion by adding legibility to expressions.

和往常一样，括号通过增加表达式的可读性来减少混淆。

Pointers and const

Pointers can be used to access a variable by its address, and this access may include modifying the value pointed. But it is also possible to declare pointers that can access the pointed value to read it, but not to modify it. For this, it is enough with qualifying the type pointed to by the pointer as `const`. For example:

指针可用于根据变量的地址访问变量，这种访问可能包括修改所指向的值。但也可以声明指针，使其可以访问所指向的值以读取它，但不能修改它。为此，只要将指针所指向的类型限定为const就够了。例如:

```
int x;
int y = 10;
const int * p = &y;
x = *p;          // ok: reading p
*p = x;          // error: modifying p, which is const-qualified
```

Here `p` points to a variable, but points to it in a `const`-qualified manner, meaning that it can read the value pointed, but it cannot modify it. Note also, that the expression `&y` is of type `int*`, but this is assigned to a pointer of type `const int*`. This is allowed: a pointer to non-const can be implicitly converted to a pointer to const. But not the other way around! As a safety feature, pointers to `const` are not implicitly convertible to pointers to non-`const`.

在这里，p指向一个变量，但以常量限定的方式指向它，这意味着它可以读取所指向的值，但不能修改它。还要注意，表达式&y的类型是int，*但它赋值给的是const int*类型的指针。这是允许的:指向非const的指针可以隐式转换为指向const的指针。但不是反过来!作为安全特性，指向const的指针不能隐式转换为指向非const的指针。

One of the use cases of pointers to `const` elements is as function parameters: a function that takes a pointer to non-`const` as parameter can modify the value passed as argument, while a function that takes a pointer to `const` as parameter cannot.

指向const元素的指针的一个用例是作为函数形参:接受非const指针作为形参的函数可以修改作为实参传递的值，而接受const指针作为形参的函数则不能。

```
// 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;
}
```

Note that `print_all` uses pointers that point to constant elements. These pointers point to constant content they cannot modify, but they are not constant themselves: i.e., the pointers can still be incremented or assigned different addresses, although they cannot modify the content they point to.

注意，print_all使用指向常量元素的指针。这些指针指向它们不能修改的常量内容，但它们本身不是常量:也就是说，指针仍然可以递增或赋值不同的地址，尽管它们不能修改它们所指向的内容。

And this is where a second dimension to constness is added to pointers: Pointers can also be themselves const. And this is specified by appending const to the pointed type (after the asterisk):

这就是向指针添加const的第二个维度:指针本身也可以是const。这是通过在指针类型后加const来指定的(在星号后面):

```
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
```

The syntax with `const` and pointers is definitely tricky, and recognizing the cases that best suit each use tends to require some experience. In any case, it is important to get constness with pointers (and references) right sooner rather than later, but you should not worry too much about grasping everything if this is the first time you are exposed to the mix of `const` and pointers. More use cases will show up in coming chapters.

使用const和指针的语法绝对是棘手的，识别最适合每种用法的情况往往需要一些经验。在任何情况下，尽早使用指针(和引用)获得const值是很重要的，但如果这是您第一次接触到const和指针混合使用的情况，则不必太担心掌握所有内容。更多的用例将在接下来的章节中出现。

To add a little bit more confusion to the syntax of `const` with pointers, the `const` qualifier can either precede or follow the pointed type, with the exact same meaning:

为了让const和指针的语法更容易混淆，const限定符可以在指针类型的前面或后面，其含义完全相同:

```
const int * p2a = &x;  //      non-const pointer to const int
int const * p2b = &x;  // also non-const pointer to const int
```

As with the spaces surrounding the asterisk, the order of const in this case is simply a matter of style. This chapter uses a prefix `const`, as for historical reasons this seems to be more extended, but both are exactly equivalent. The merits of each style are still intensely debated on the internet.

与星号周围的空格一样，在本例中，const的顺序只是样式问题。本章使用了前缀const，因为历史原因，这似乎要更广泛一些，但两者是完全等价的。每种风格的优点在互联网上仍有激烈的争论。

Pointers and string literals

As pointed earlier, *string literals* are arrays containing null-terminated character sequences. In earlier sections, string literals have been used to be directly inserted into `cout`, to initialize strings and to initialize arrays of characters.

如前所述，*字符串字面值*是包含以null结尾的字符序列的数组。在前面的章节中，字符串字面量被用于直接插入' cout '，用于初始化字符串和字符数组。

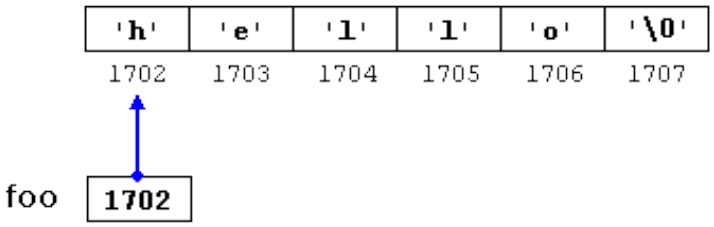
But they can also be accessed directly. String literals are arrays of the proper array type to contain all its characters plus the terminating null-character, with each of the elements being of type `const char` (as literals, they can never be modified). For example:

但它们也可以直接访问。字符串字面值是正确数组类型的数组，包含它的所有字符和结束空字符，每个元素的类型为const char(作为字面值，它们永远不能被修改)。例如:

```
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:

它声明了一个具有“hello”字面值表示的数组，然后将指向数组第一个元素的指针赋给foo。如果我们假设“hello”存储在以地址1702开始的内存位置，我们可以将前面的声明表示为:



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.

注意这里foo是一个指针，包含值1702，不是'h'，也不是"hello"，尽管1702确实是这两个地址的地址。

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:

指针foo指向一个字符序列。因为指针和数组在表达式中的行为本质上是相同的，所以foo可以用与以空结尾的字符序列数组相同的方式访问字符。例如:

```
*(foo+4)
foo[4]
```

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

这两个表达式的值都是'o'(数组的第五个元素)。

Pointers to pointers

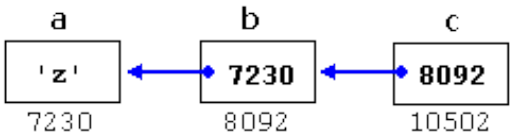
C++ allows the use of pointers that point to pointers, that these, in its turn, point to data (or even to other pointers). The syntax simply requires an asterisk (`*`) for each level of indirection in the declaration of the pointer:

c++允许使用指向指针的指针，这些指针反过来指向数据(甚至指向其他指针)。该语法只需要在指针的声明中为每一层的间接指向加一个星号(*):

```
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:

假设每个变量7230、8092和10502的随机选择的内存位置可以表示为:



With the value of each variable represented inside its corresponding cell, and their respective addresses in memory represented by the value under them.

每个变量的值表示在其相应的单元格中，它们在内存中的各自地址由它们下面的值表示。

The new thing in this example is variable `c`, which is a pointer to a pointer, and can be used in three different levels of indirection, each one of them would correspond to a different value:

这个例子中的新东西是变量' c '，它是一个指针指向一个指针，可以在三种不同的间接层中使用，每一种间接层对应不同的值:

- `c` is of type `char**` and a value of `8092`
- `*c` is of type `char*` and a value of `7230`
- `**c` is of type `char` and a value of `'z'`

void pointers

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

指针的void类型是一种特殊类型的指针。在c++中，void表示没有类型。因此，空指针是指向没有类型的值的指针(因此也有不确定的长度和不确定的解引用属性)。

This gives `void` pointers a great flexibility, by being able to point to any data type, from an integer value or a float to a string of characters. In exchange, they have a great limitation: the data pointed to by them cannot be directly dereferenced (which is logical, since we have no type to dereference to), and for that reason, any address in a `void` pointer needs to be transformed into some other pointer type that points to a concrete data type before being dereferenced.

这为空指针提供了很大的灵活性，因为它能够指向任何数据类型，从整数值或浮点数到字符串。作为交换,他们有一个伟大的限制:他们指出的数据不能直接引用(这是合乎逻辑的,因为我们没有间接引用类型),因此,任何一个空指针地址需要被转换成其他指针类型指一个具体的数据类型被引用之前。

One of its possible uses may be to pass generic parameters to a function. For example:

它可能的用途之一是向函数传递泛型参数。例如:

```
// 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;
}
```

`sizeof` is an operator integrated in the C++ language that returns the size in bytes of its argument. For non-dynamic data types, this value is a constant. Therefore, for example, `sizeof(char)` is 1, because `char` has always a size of one byte.

sizeof是c++语言中集成的操作符，以字节为单位返回参数的大小。对于非动态数据类型，此值为常量。例如，sizeof(char)是1，因为char的大小总是一个字节。

Invalid pointers and null pointers

In principle, pointers are meant to point to valid addresses, such as the address of a variable or the address of an element in an array. But pointers can actually point to any address, including addresses that do not refer to any valid element. Typical examples of this are *uninitialized pointers* and pointers to nonexistent elements of an array:

原则上，指针是指向有效地址的，比如变量的地址或数组中元素的地址。但是指针实际上可以指向任何地址，包括不指向任何有效元素的地址。典型的例子是未初始化的指针和指向数组中不存在元素的指针:

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

int myarray[10];
int * q = myarray+20;  // element out of bounds
```

Neither `p` nor `q` point to addresses known to contain a value, but none of the above statements causes an error. In C++, pointers are allowed to take any address value, no matter whether there actually is something at that address or not. What can cause an error is to dereference such a pointer (i.e., actually accessing the value they point to). Accessing such a pointer causes undefined behavior, ranging from an error during runtime to accessing some random value.

p和q都没有指向已知包含值的地址，但是上面的语句都不会导致错误。在c++中，指针被允许接受任何地址值，不管这个地址上是否真的有东西。导致错误的是对这样的指针进行解引用(即实际访问它们所指向的值)。访问这样一个指针会导致未定义的行为，范围从运行时的错误到访问某个随机值。

But, sometimes, a pointer really needs to explicitly point to nowhere, and not just an invalid address. For such cases, there exists a special value that any pointer type can take: the *null pointer value*. This value can be expressed in C++ in two ways: either with an integer value of zero, or with the `nullptr` keyword:

但是，有时，指针确实需要显式地指向任何地方，而不仅仅是一个无效的地址。对于这种情况，存在一个任何指针类型都可以接受的特殊值:空指针值。在c++中，这个值可以用两种方式表示:用0的整数值表示，或者用nullptr关键字表示:

```
int * p = 0;
int * q = nullptr;
```

Here, both `p` and `q` are *null pointers*, meaning that they explicitly point to nowhere, and they both actually compare equal: all *null pointers* compare equal to other *null pointers*. It is also quite usual to see the defined constant `NULL` be used in older code to refer to the *null pointer* value:

这里，p和q都是空指针，这意味着它们显式地指向任何地方，而且它们实际上是相等的:所有空指针与其他空指针的比较是相等的。在较早的代码中，也经常看到定义的常量NULL被用来引用空指针值:

```
int * r = NULL;
```

`NULL` is defined in several headers of the standard library, and is defined as an alias of some *null pointer* constant value (such as `0` or `nullptr`).

NULL在标准库的几个头文件中定义，并被定义为某个空指针常量值(如0或nullptr)的别名。

Do not confuse *null pointers* with `void` pointers! A *null pointer* is a value that any pointer can take to represent that it is pointing to "nowhere", while a `void` pointer is a type of pointer that can point to somewhere without a specific type. One refers to the value stored in the pointer, and the other to the type of data it points to.

不要混淆空指针和空指针!空指针是一个任何指针都可以接受的值，表示它是“没有指向”，而空指针是一种指针，可以指向没有特定类型的地方。一个指向存储在指针中的值，另一个指向指针所指向的数据类型。

Pointers to functions

C++ allows operations with pointers to functions. The typical use of this is for passing a function as an argument to another function. Pointers to functions are declared with the same syntax as a regular function declaration, except that the name of the function is enclosed between parentheses () and an asterisk (`*`) is inserted before the name:

c++允许使用指向函数的指针进行操作。它的典型用法是将一个函数作为参数传递给另一个函数。函数指针的声明语法与常规函数声明相同，只是函数名用圆括号()括起来，并在函数名前插入星号(*):

```
// 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;
}
```

In the example above, `minus` is a pointer to a function that has two parameters of type `int`. It is directly initialized to point to the function `subtraction`:

在上面的例子中，minus是一个指向有两个int型形参的函数的指针。它被直接初始化为指向函数减法:

```
int (* minus)(int,int) = subtraction;
```