**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.  
  
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.  
  
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.  
  
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.

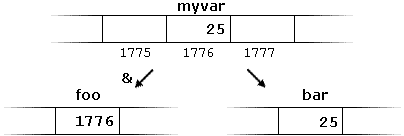
**Address-of operator (&)**

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:

|  |  |  |
| --- | --- | --- |
|  | 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.  
  
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.  
  
In this case, consider the following code fragment:

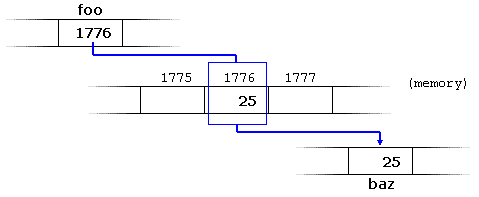
|  |  |  |
| --- | --- | --- |
| 1 2 3 | myvar = 25;  foo = &myvar;  bar = myvar; |  |

The values contained in each variable after the execution of this are shown in the following diagram:   
  
   
  
First, we have assigned the value 25 to myvar (a variable whose address in memory we assumed to be 1776).  
  
The second statement assigns foo the address of myvar, which we have assumed to be 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.  
  
The main difference between the second and third statements is the appearance of the *address-of operator* (&).  
  
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.

**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.  
  
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".  
  
Therefore, following with the values of the previous example, the following statement:

|  |  |  |
| --- | --- | --- |
|  | baz = \*foo; |  |

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

|  |  |  |
| --- | --- | --- |
| 1 2 | 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"

Thus, they have sort of opposite meanings: An address obtained with & can be dereferenced with \*.  
  
Earlier, we performed the following two assignment operations:

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

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

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 | 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 of1776. 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.  
  
So, after all that, you may also infer that for as long as the address pointed by foo remains unchanged, the following expression will also be true:

|  |  |  |
| --- | --- | --- |
|  | \*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.  
  
The declaration of pointers follows this syntax:  
  
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:

|  |  |  |
| --- | --- | --- |
| 1 2 3 | 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 adouble. 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.  
  
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.  
  
Let's see an example on pointers:

|  |  |  |  |
| --- | --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 | // 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 | [Edit & Run](http://www.cplusplus.com/doc/tutorial/pointers/) |

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:  
  
First, mypointer is assigned the address of firstvalue using the address-of operator (&). Then, the value pointed to bymypointer is assigned a value of 10. Because, at this moment, mypointer is pointing to the memory location offirstvalue, this in fact modifies the value of firstvalue.  
  
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.  
  
Here is an example a little bit more elaborated:

|  |  |  |  |
| --- | --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 | // 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 | [Edit & Run](http://www.cplusplus.com/doc/tutorial/pointers/) |

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".  
  
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).  
  
Another thing that may call your attention is the line:

|  |  |  |
| --- | --- | --- |
|  | 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:

|  |  |  |
| --- | --- | --- |
|  | 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 statemet for each variable.

**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:

|  |  |  |
| --- | --- | --- |
| 1 2 | int myarray [20];  int \* mypointer; |  |

The following assignment operation would be valid:

|  |  |  |
| --- | --- | --- |
|  | 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:

|  |  |  |
| --- | --- | --- |
|  | myarray = mypointer; |  |

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

|  |  |  |  |
| --- | --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 | // 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, | [Edit & Run](http://www.cplusplus.com/doc/tutorial/pointers/) |

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.  
  
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:

|  |  |  |
| --- | --- | --- |
| 1 2 | a[5] = 0; // a [offset of 5] = 0  \*(a+5) = 0; // pointed 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.

**Pointer initialization**

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:

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

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

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

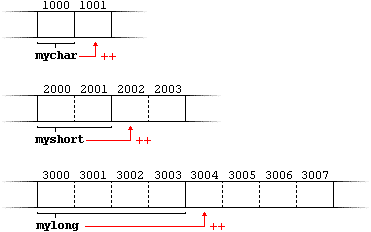
**Pointer arithmetics**

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

|  |  |  |
| --- | --- | --- |
| 1 2 3 | 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:

|  |  |  |
| --- | --- | --- |
|  | \*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).  
  
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):

|  |  |  |
| --- | --- | --- |
| 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 |  |

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

|  |  |  |
| --- | --- | --- |
|  | \*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:

|  |  |  |
| --- | --- | --- |
| 1 2 3 | \*p = \*q;  ++p;  ++q; |  |

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

**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 by the pointer as const. For example:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 | 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.  
  
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.

|  |  |  |  |
| --- | --- | --- | --- |
| 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 28 29 | // 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 | [Edit & Run](http://www.cplusplus.com/doc/tutorial/pointers/) |

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.  
  
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):

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 | 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 ofconst and pointers. More use cases will show up in coming chapters.  
  
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:

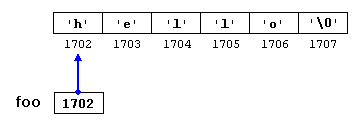
|  |  |  |
| --- | --- | --- |
| 1 2 | const int \* p2a = &x; // non-const pointer to const int  int const \* p2b = &x; // also non-const pointer to const int |  |

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.

**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.  
  
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:

|  |  |  |
| --- | --- | --- |
|  | 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:

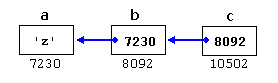
|  |  |  |
| --- | --- | --- |
| 1 2 | \*(foo+4)  foo[4] |  |

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

**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:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 | 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:  
  
   
With the value of each variable represented inside its corresponding cell, and their respective addresses in memory represented by the value under them.  
  
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:

* 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, voidpointers are pointers that point to a value that has no type (and thus also an undetermined length and undetermined dereferencing properties).  
  
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 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.  
  
One of its possible uses may be to pass generic parameters to a function. For example:

|  |  |  |  |
| --- | --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 | // 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 | [Edit & Run](http://www.cplusplus.com/doc/tutorial/pointers/) |

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 is has always a size of one byte. 

**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:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 | 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.  
  
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:

|  |  |  |
| --- | --- | --- |
| 1 2 | 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:

|  |  |  |
| --- | --- | --- |
|  | 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).  
  
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.

**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:

|  |  |  |  |
| --- | --- | --- | --- |
| 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 | [Edit & Run](http://www.cplusplus.com/doc/tutorial/pointers/) |

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:

|  |  |
| --- | --- |
|  | int (\* minus)(int,int) = subtraction; |

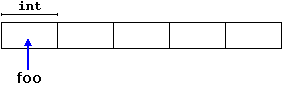
**Dynamic memory**

In the programs seen in previous chapters, all memory needs were determined before program execution by defining the variables needed. But there may be cases where the memory needs of a program can only be determined during runtime. For example, when the memory needed depends on user input. On these cases, programs need to dynamically allocate memory, for which the C++ language integrates the operators new and delete.

**Operators new and new[]**

Dynamic memory is allocated using operator new. new is followed by a data type specifier and, if a sequence of more than one element is required, the number of these within brackets []. It returns a pointer to the beginning of the new block of memory allocated. Its syntax is:   
  
pointer = new type  
pointer = new type [number\_of\_elements]  
  
The first expression is used to allocate memory to contain one single element of type type. The second one is used to allocate a block (an array) of elements of type type, where number\_of\_elements is an integer value representing the amount of these. For example:

|  |  |  |
| --- | --- | --- |
| 1 2 | int \* foo;  foo = new int [5]; |  |

In this case, the system dynamically allocates space for five elements of type int and returns a pointer to the first element of the sequence, which is assigned to foo (a pointer). Therefore, foo now points to a valid block of memory with space for five elements of type int.  
  
   
  
Here, foo is a pointer, and thus, the first element pointed to by foo can be accessed either with the expression foo[0] or the expression \*foo (both are equivalent). The second element can be accessed either with foo[1] or \*(foo+1), and so on...  
  
There is a substantial difference between declaring a normal array and allocating dynamic memory for a block of memory using new. The most important difference is that the size of a regular array needs to be a *constant expression*, and thus its size has to be determined at the moment of designing the program, before it is run, whereas the dynamic memory allocation performed by new allows to assign memory during runtime using any variable value as size.  
  
The dynamic memory requested by our program is allocated by the system from the memory heap. However, computer memory is a limited resource, and it can be exhausted. Therefore, there are no guarantees that all requests to allocate memory using operator new are going to be granted by the system.   
  
C++ provides two standard mechanisms to check if the allocation was successful:  
  
One is by handling exceptions. Using this method, an exception of type bad\_alloc is thrown when the allocation fails. Exceptions are a powerful C++ feature explained later in these tutorials. But for now, you should know that if this exception is thrown and it is not handled by a specific handler, the program execution is terminated.  
  
This exception method is the method used by default by new, and is the one used in a declaration like:

|  |  |  |
| --- | --- | --- |
|  | foo = new int [5]; // if allocation fails, an exception is thrown |  |

The other method is known as nothrow, and what happens when it is used is that when a memory allocation fails, instead of throwing a bad\_alloc exception or terminating the program, the pointer returned by new is a *null pointer*, and the program continues its execution normally.  
  
This method can be specified by using a special object called nothrow, declared in header [<new>](http://www.cplusplus.com/%3Cnew%3E), as argument for new:

|  |  |  |
| --- | --- | --- |
|  | foo = new (nothrow) int [5]; |  |

In this case, if the allocation of this block of memory fails, the failure can be detected by checking if foo is a null pointer:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 | int \* foo;  foo = new (nothrow) int [5];  if (foo == nullptr) {  // error assigning memory. Take measures.  } |  |

This nothrow method is likely to produce less efficient code than exceptions, since it implies explicitly checking the pointer value returned after each and every allocation. Therefore, the exception mechanism is generally preferred, at least for critical allocations. Still, most of the coming examples will use the nothrow mechanism due to its simplicity.

**Operators delete and delete[]**

In most cases, memory allocated dynamically is only needed during specific periods of time within a program; once it is no longer needed, it can be freed so that the memory becomes available again for other requests of dynamic memory. This is the purpose of operator delete, whose syntax is:

|  |  |  |
| --- | --- | --- |
| 1 2 | delete pointer;  delete[] pointer; |  |

The first statement releases the memory of a single element allocated using new, and the second one releases the memory allocated for arrays of elements using new and a size in brackets ([]).  
  
The value passed as argument to delete shall be either a pointer to a memory block previously allocated with new, or a*null pointer* (in the case of a *null pointer*, delete produces no effect).

|  |  |  |  |
| --- | --- | --- | --- |
| 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 28 | // rememb-o-matic  #include <iostream>  #include <new>  using namespace std;  int main ()  {  int i,n;  int \* p;  cout << "How many numbers would you like to type? ";  cin >> i;  p= new (nothrow) int[i];  if (p == nullptr)  cout << "Error: memory could not be allocated";  else  {  for (n=0; n<i; n++)  {  cout << "Enter number: ";  cin >> p[n];  }  cout << "You have entered: ";  for (n=0; n<i; n++)  cout << p[n] << ", ";  delete[] p;  }  return 0;  } | How many numbers would you like to type? 5  Enter number : 75  Enter number : 436  Enter number : 1067  Enter number : 8  Enter number : 32  You have entered: 75, 436, 1067, 8, 32, | [Edit & Run](http://www.cplusplus.com/doc/tutorial/dynamic/) |

Notice how the value within brackets in the new statement is a variable value entered by the user (i), not a constant expression:

|  |  |  |
| --- | --- | --- |
|  | p= new (nothrow) int[i]; |  |

There always exists the possibility that the user introduces a value for i so big that the system cannot allocate enough memory for it. For example, when I tried to give a value of 1 billion to the "How many numbers" question, my system could not allocate that much memory for the program, and I got the text message we prepared for this case (Error: memory could not be allocated).  
  
It is considered good practice for programs to always be able to handle failures to allocate memory, either by checking the pointer value (if nothrow) or by catching the proper exception.

**Dynamic memory in C**

C++ integrates the operators new and delete for allocating dynamic memory. But these were not available in the C language; instead, it used a library solution, with the functions [malloc](http://www.cplusplus.com/malloc), [calloc](http://www.cplusplus.com/calloc), [realloc](http://www.cplusplus.com/realloc) and [free](http://www.cplusplus.com/free), defined in the header[<cstdlib>](http://www.cplusplus.com/%3Ccstdlib%3E) (known as <stdlib.h> in C). The functions are also available in C++ and can also be used to allocate and deallocate dynamic memory.  
  
Note, though, that the memory blocks allocated by these functions are not necessarily compatible with those returned bynew, so they should not be mixed; each one should be handled with its own set of functions or operators.

**Data structures**

**Data structures**

A *data structure* is a group of data elements grouped together under one name. These data elements, known as*members*, can have different types and different lengths. Data structures can be declared in C++ using the following syntax:  
  
struct type\_name {  
member\_type1 member\_name1;  
member\_type2 member\_name2;  
member\_type3 member\_name3;  
.  
.  
} object\_names;  
  
Where type\_name is a name for the structure type, object\_name can be a set of valid identifiers for objects that have the type of this structure. Within braces {}, there is a list with the data members, each one is specified with a type and a valid identifier as its name.  
  
For example:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 | struct product {  int weight;  double price;  } ;  product apple;  product banana, melon; |  |

This declares a structure type, called product, and defines it having two members: weight and price, each of a different fundamental type. This declaration creates a new type (product), which is then used to declare three objects (variables) of this type: apple, banana, and melon. Note how once product is declared, it is used just like any other type.  
  
Right at the end of the struct definition, and before the ending semicolon (;), the optional field object\_names can be used to directly declare objects of the structure type. For example, the structure objects apple, banana, and melon can be declared at the moment the data structure type is defined:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 | struct product {  int weight;  double price;  } apple, banana, melon; |  |

In this case, where object\_names are specified, the type name (product) becomes optional: struct requires either atype\_name or at least one name in object\_names, but not necessarily both.  
  
It is important to clearly differentiate between what is the structure type name (product), and what is an object of this type (apple, banana, and melon). Many objects (such as apple, banana, and melon) can be declared from a single structure type (product).  
  
Once the three objects of a determined structure type are declared (apple, banana, and melon) its members can be accessed directly. The syntax for that is simply to insert a dot (.) between the object name and the member name. For example, we could operate with any of these elements as if they were standard variables of their respective types:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 | apple.weight  apple.price  banana.weight  banana.price  melon.weight  melon.price |  |

Each one of these has the data type corresponding to the member they refer to: apple.weight, banana.weight, andmelon.weight are of type int, while apple.price, banana.price, and melon.price are of type double.  
  
Here is a real example with structure types in action:

|  |  |  |  |
| --- | --- | --- | --- |
| 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 28 29 30 31 32 33 34 35 36 37 38 | // example about structures  #include <iostream>  #include <string>  #include <sstream>  using namespace std;  struct movies\_t {  string title;  int year;  } mine, yours;  void printmovie (movies\_t movie);  int main ()  {  string mystr;  mine.title = "2001 A Space Odyssey";  mine.year = 1968;  cout << "Enter title: ";  getline (cin,yours.title);  cout << "Enter year: ";  getline (cin,mystr);  stringstream(mystr) >> yours.year;  cout << "My favorite movie is:\n ";  printmovie (mine);  cout << "And yours is:\n ";  printmovie (yours);  return 0;  }  void printmovie (movies\_t movie)  {  cout << movie.title;  cout << " (" << movie.year << ")\n";  } | Enter title: Alien  Enter year: 1979  My favorite movie is:  2001 A Space Odyssey (1968)  And yours is:  Alien (1979) | [Edit & Run](http://www.cplusplus.com/doc/tutorial/structures/) |

The example shows how the members of an object act just as regular variables. For example, the member yours.year is a valid variable of type int, and mine.title is a valid variable of type string.  
  
But the objects mine and yours are also variables with a type (of type movies\_t). For example, both have been passed to function printmovie just as if they were simple variables. Therefore, one of the features of data structures is the ability to refer to both their members individually or to the entire structure as a whole. In both cases using the same identifier: the name of the structure.  
  
Because structures are types, they can also be used as the type of arrays to construct tables or databases of them:

|  |  |  |  |
| --- | --- | --- | --- |
| 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 28 29 30 31 32 33 34 35 36 37 38 | // array of structures  #include <iostream>  #include <string>  #include <sstream>  using namespace std;  struct movies\_t {  string title;  int year;  } films [3];  void printmovie (movies\_t movie);  int main ()  {  string mystr;  int n;  for (n=0; n<3; n++)  {  cout << "Enter title: ";  getline (cin,films[n].title);  cout << "Enter year: ";  getline (cin,mystr);  stringstream(mystr) >> films[n].year;  }  cout << "\nYou have entered these movies:\n";  for (n=0; n<3; n++)  printmovie (films[n]);  return 0;  }  void printmovie (movies\_t movie)  {  cout << movie.title;  cout << " (" << movie.year << ")\n";  } | Enter title: Blade Runner  Enter year: 1982  Enter title: The Matrix  Enter year: 1999  Enter title: Taxi Driver  Enter year: 1976    You have entered these movies:  Blade Runner (1982)  The Matrix (1999)  Taxi Driver (1976) | [Edit & Run](http://www.cplusplus.com/doc/tutorial/structures/) |

**Pointers to structures**

Like any other type, structures can be pointed to by its own type of pointers:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 | struct movies\_t {  string title;  int year;  };  movies\_t amovie;  movies\_t \* pmovie; |  |

Here amovie is an object of structure type movies\_t, and pmovie is a pointer to point to objects of structure type movies\_t. Therefore, the following code would also be valid:

|  |  |  |
| --- | --- | --- |
|  | pmovie = &amovie; |  |

The value of the pointer pmovie would be assigned the address of object amovie.  
  
Now, let's see another example that mixes pointers and structures, and will serve to introduce a new operator: the arrow operator (->):

|  |  |  |  |
| --- | --- | --- | --- |
| 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 28 29 30 31 | // pointers to structures  #include <iostream>  #include <string>  #include <sstream>  using namespace std;  struct movies\_t {  string title;  int year;  };  int main ()  {  string mystr;  movies\_t amovie;  movies\_t \* pmovie;  pmovie = &amovie;  cout << "Enter title: ";  getline (cin, pmovie->title);  cout << "Enter year: ";  getline (cin, mystr);  (stringstream) mystr >> pmovie->year;  cout << "\nYou have entered:\n";  cout << pmovie->title;  cout << " (" << pmovie->year << ")\n";  return 0;  } | Enter title: Invasion of the body snatchers  Enter year: 1978    You have entered:  Invasion of the body snatchers (1978) | [Edit & Run](http://www.cplusplus.com/doc/tutorial/structures/) |

The arrow operator (->) is a dereference operator that is used exclusively with pointers to objects that have members. This operator serves to access the member of an object directly from its address. For example, in the example above:

|  |  |  |
| --- | --- | --- |
|  | pmovie->title |  |

is, for all purposes, equivalent to:

|  |  |  |
| --- | --- | --- |
|  | (\*pmovie).title |  |

Both expressions, pmovie->title and (\*pmovie).title are valid, and both access the member title of the data structure pointed by a pointer called pmovie. It is definitely something different than:

|  |  |  |
| --- | --- | --- |
|  | \*pmovie.title |  |

which is rather equivalent to:

|  |  |  |
| --- | --- | --- |
|  | \*(pmovie.title) |  |

This would access the value pointed by a hypothetical pointer member called title of the structure object pmovie (which is not the case, since title is not a pointer type). The following panel summarizes possible combinations of the operators for pointers and for structure members:

|  |  |  |
| --- | --- | --- |
| **Expression** | **What is evaluated** | **Equivalent** |
| a.b | Member b of object a |  |
| a->b | Member b of object pointed to by a | (\*a).b |
| \*a.b | Value pointed to by member b of object a | \*(a.b) |

**Nesting structures**

Structures can also be nested in such a way that an element of a structure is itself another structure:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 | struct movies\_t {  string title;  int year;  };  struct friends\_t {  string name;  string email;  movies\_t favorite\_movie;  } charlie, maria;  friends\_t \* pfriends = &charlie; |  |

After the previous declarations, all of the following expressions would be valid:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 | charlie.name  maria.favorite\_movie.title  charlie.favorite\_movie.year  pfriends->favorite\_movie.year |  |

(where, by the way, the last two expressions refer to the same member).

C++ Intro

### Lecture 8: struct

## C++ structure: aggregation of data fields

|  |
| --- |
| In C++, an array is an aggregate of elements of the same type. A C++ **struct** is an aggregate of elements of arbitrary types. For example:  **struct** address {  string person\_name;  **int** street\_number;  string street\_address;  string town;  **char** state[2];  **int** zipcode;  };  This defines a new type called **address** consisting of the items you need in order to send mail to someone. Note the semicolon at the end. This is one of very few places in C++ where it is necessary to have a semicolon after a curly brace, so people are prone to forget it. |

### C++ objects and access to their data members

|  |
| --- |
| Variables of type **address** are often referred to as C++ objects. They can be declared exactly as other variables, and the individual members can be accessed using the "." (dot) operator. For example:  **#include** <iostream>  **#include** <string>  **struct** address{ **/\*...\*/** };  **int** main(**int** argc, **char**\* argv[])  {  address bcc;  bcc.person\_name = "Joe Doe";  bcc.street\_number = 777;  bcc.street\_address = "ELSBREE STREET";  bcc.town = "FALL RIVER";  bcc.state[ 0 ] = 'M';  bcc.state[ 1 ] = 'A';  bcc.zipcode = 2720;  **return** 0;  } |

### Pointers to objects

|  |
| --- |
| Structure objects are often accessed through pointers using the "->" operator, also known aspointer-to-object dereference operator. For example:  **#include** <iostream>  **#include** <string>  **struct** address{ **/\*...\*/** }; **// address becomes new C++ type**  **int** main(**int** argc, **char**\* argv[])  {  address adr; **// instantiate address object**  address\* padr = &adr; **// create pointer to object**  paddr->person\_name = "Frederick Charles Krueger";  **return** 0;  } |

### Object as function argument

|  |
| --- |
| Objects of structure types can be passed as function arguments (either by value, or by object reference):  **#include** <iostream>  **#include** <string>  **struct** address{ **/\*...\*/** }; **// address becomes new C++ type**  **int** main(**int** argc, **char**\* argv[])  {  address fictional\_character;  populate\_address( fictional\_character );  **return** 0;  }  **void** populate\_address( address& addr\_ )  {  addr\_.person\_name = "Frederick Charles Krueger";  addr\_.street\_name = "Elm Street";  addr\_.town = "Springwood";  } |

### Object initialization

|  |
| --- |
| The **address** structure that we've used so far did not require a programmer to provide any kind of initialization. We were able to instantiate an object, and then use individual member variables to store the data. It is possible to establish a more formal protocol, which will require a set of initial values that will be used when the object is created. Consider  **struct** date{  **int** month;  **int** day;  **int** year;  };  The structure named **date** could benefit from an explicit initialization of its member variables. For example, it would be nice to create a **date** object along with a set of values for its month, day, and year:  **int** main(**int** argc, **char**\* argv[])  {  date birthday( 12, 5, 1997 ); **// instantiate and initialize the object**  **return** 0;  }  In order to allow this type of initialization, we must declare a special function inside our structure, which is called a constructor. The entire program will look like this:  **struct** date{  **int** month;  **int** day;  **int** year;  **// Constructor:**  date( **int** month\_, **int** day\_, **int** year\_ )  {  month = month\_;  day = day\_;  year = year\_;  }  };**//struct date**  **int** main(**int** argc, **char**\* argv[])  {  date birthday( 12, 5, 1997 ); **// instantiate and initialize the object**  **return** 0;  } |

### Default constructor

|  |
| --- |
| Adding a constructor function to an object means that all objects of the user-defined type will be initialized, because each declaration will require a set of parameters to initialize the data members. In order to permit the old way of object creation, the default constructor must be provided; that is, a cnstructor that has no arguments. The default constructor typically initializes individual data members by setting them to the appropriate default values:  **struct** date{  **int** month;  **int** day;  **int** year;  **// Default constructor:**  date()  {  month = 0;  day = 0;  year = 0;  }  **// Constructor:**  date( **int** month\_, **int** day\_, **int** year\_ )  {  month = month\_;  day = day\_;  year = year\_;  }  };**//struct date**  **int** main(**int** argc, **char**\* argv[])  {  date today; **// default constructor is invoked**  date birthday( 12, 5, 1997 ); **// instantiate and initialize the object**  **return** 0;  } |

### struct provides a way to establish a new user-defined C++ type

|  |
| --- |
| The definition of a structure provides a new user-defined C++ type to the program. A structure can have many different constructors to provide convenient ways to init the objects. Once created, an object, like other fundamental C++ types, can be used in many different ways. For example, the following program illustrates a **vector** of user-defined type **date**:  **#include** <iostream>  **#include** <vector>  **struct** date{  **int** month;  **int** day;  **int** year;  **// Default constructor:**  date()  {  month = 0;  day = 0;  year = 0;  }  **// Constructor:**  date( **int** month\_, **int** day\_, **int** year\_ )  {  month = month\_;  day = day\_;  year = year\_;  }  };**//struct date**  **int** main(**int** argc, **char**\* argv[])  {  std::vector< date > week( 7 );  week[ 0 ] = date( 4, 8, 2007 );  week[ 1 ] = date( 4, 9, 2007 );  week[ 2 ] = date( 4, 10, 2007 );  week[ 3 ] = date( 4, 11, 2007 );  week[ 4 ] = date( 4, 12, 2007 );  week[ 5 ] = date( 4, 13, 2007 );  week[ 6 ] = date( 4, 14, 2007 );  **return** 0;  } |