C++ Programming

For BCA 4th Semester

Lecture 2

[Pointer, Pointer and Array, Pointer to Pointer, Void Pointer, Invalid Pointer, Null Pointer, Pointer to Function]

Compiled
By
Subhadip Mukherjee
Dept. of Computer Science and BCA
Kharagpur College,
Kharagpur 721305

Pointer in C++

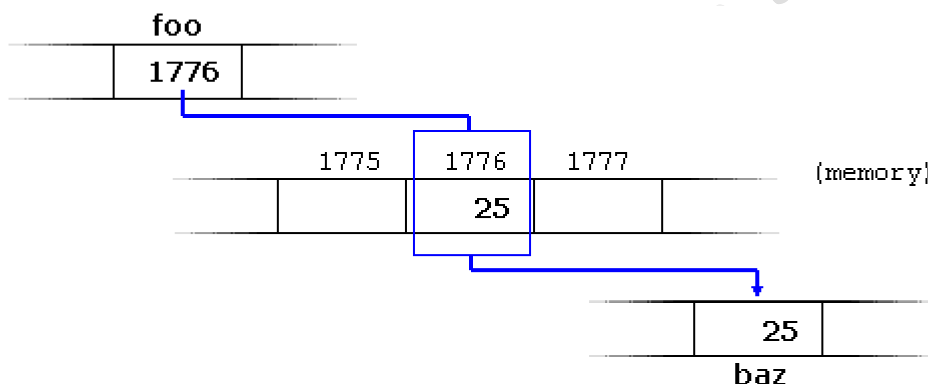
As just seen, a variable which stores the address of another variable is called a *pointer*. Pointers are said to "point to" the variable whose address they store.

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` to `baz`, 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 baz = foo;   // baz equal to foo (1776)
2 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 myvar = 25;
2 foo = &myvar;
```

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

Sc. BCA, Kharagpur College By Subhadip

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

Example 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
```

Edit & Run

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 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`.

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`.

Example 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 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;
}
```

```
firstvalue is 10
secondvalue is 20
```

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".

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 int myarray [20];
2 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 // more pointers
2 #include <iostream>
3 using namespace std;
4
5 int main ()
6 {
7   int numbers[5];
8   int * p;
9   p = numbers;  *p = 10;
10  p++;  *p = 20;
11  p = &numbers[2];  *p = 30;
12  p = numbers + 3;  *p = 40;
13  p = numbers;  *(p+4) = 50;
14  for (int n=0; n<5; n++)
15    cout << numbers[n] << ", ";
16  return 0;
17 }
```

```
10, 20, 30, 40, 50,
```

Edit & Run

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

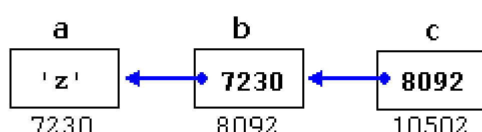
These two expressions are equivalent and valid, not only if `a` is a pointer, but also if `a` is an array. Remember that if an array, its name can be used just like a pointer to its first element.

Pointer to pointer

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 char a;
2 char * b;
3 char ** c;
4 a = 'z';
5 b = &a;
6 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, `void` pointers 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 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.

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

```
1 // increaser
2 #include <iostream>
3 using namespace std;
4
5 void increase (void* data, int psize)
6 {
7   if ( psize == sizeof(char) )
8   { char* pchar; pchar=(char*)data;
9 ++(*pchar); }
10  else if (psize == sizeof(int) )
11
```

```
y, 1603
```

Edit & Run

```
  { 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.

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:

```
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:

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

```
8
```

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;
```

By Subhadip ... nur College

By Subhadip Mukherjee, Dept. of Comp. Sc. & BCA, Kharagpur College