## Compound data types

# Pointers

We have already seen how variables are seen as memory cells that can be accessed using their identifiers. This way we did not have to care about the physical location of our data within memory, we simply used its identifier whenever we wanted to refer to our variable.

The memory of your computer can be imagined as a succession of memory cells, each one of the minimal size that computers manage (one byte). These single-byte memory cells are numbered in a consecutive way, so as, within any block of memory, every cell has the same number as the previous one plus one.

This way, each cell can be easily located in the memory because it has a unique address and all the memory cells follow a successive pattern. For example, if we are looking for cell 1776 we know that it is going to be right between cells 1775 and 1777, exactly one thousand cells after 776 and exactly one thousand cells before cell 2776.

## Reference operator (&)

As soon as we declare a variable, the amount of memory needed is assigned for it at a specific location in memory (its memory address). We generally do not actively decide the exact location of the variable within the panel of cells that we have imagined the memory to be - Fortunately, that is a task automatically performed by the operating system during runtime. However, in some cases we may be interested in knowing the address where our variable is being stored during runtime in order to operate with relative positions to it.

The address that locates a variable within memory is what we call a *reference* to that variable. This reference to a variable can be obtained by preceding the identifier of a variable with an ampersand sign (`&`), known as reference operator, and which can be literally translated as "address of". For example:

```
ted = &andy;
```

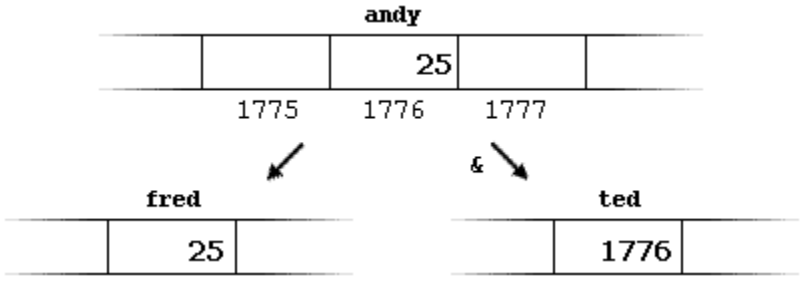
This would assign to `ted` the address of variable `andy`, since when preceding the name of the variable `andy` with the reference operator (`&`) we are no longer talking about the content of the variable itself, but about its reference (i.e., its address in memory).

From now on we are going to assume that `andy` is placed during runtime in the memory address `1776`. This number (`1776`) is just an arbitrary assumption we are inventing right now in order to help clarify some concepts in this tutorial, but in reality, we cannot know before runtime the real value the address of a variable will have in memory.

Consider the following code fragment:

```
andy = 25;
fred = andy;
ted = &andy;
```

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 `andy` (a variable whose address in memory we have assumed to be 1776).

The second statement copied to `fred` the content of variable `andy` (which is 25). This is a standard assignment operation, as we have done so many times before.

Finally, the third statement copies to `ted` not the value contained in `andy` but a reference to it (i.e., its address, which we have assumed to be `1776`). The reason is that in this third assignment operation we have preceded the identifier `andy` with the reference operator (`&`), so we were no longer referring to the value of andy but to its reference (its address in memory).

The variable that stores the reference to another variable (like `ted` in the previous example) is what we call a *pointer*. Pointers are a very powerful feature of the C++ language that has many uses in advanced programming. Farther ahead, we will see how this type of variable is used and declared.

# Dereference operator (*)

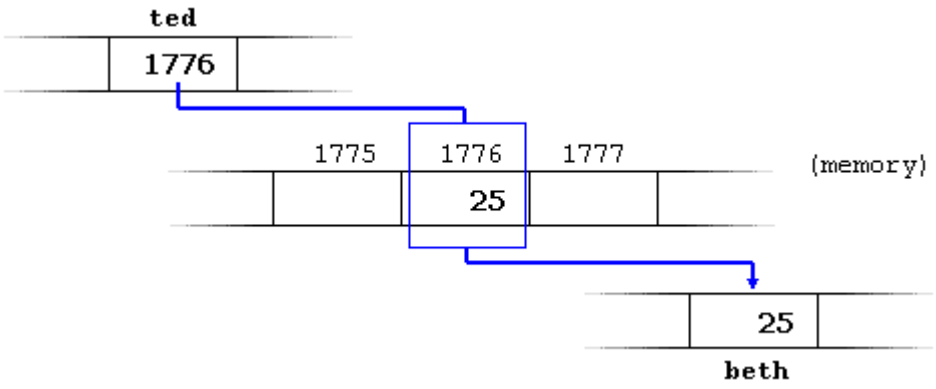
We have just seen that a variable which stores a reference to another variable is called a pointer. Pointers are said to "point to" the variable whose reference they store.

Using a pointer we can directly access the value stored in the variable which it points to. To do this, we simply have to precede the pointer's identifier with an asterisk (*), which acts as dereference operator and that can be literally translated to "value pointed by".

Therefore, following with the values of the previous example, if we write:

```
beth = *ted;
```

(that we could read as: "`beth` equal to value pointed by `ted`") `beth` would take the value `25`, since `ted` is `1776`, and the value pointed by 1776 is 25.




You must clearly differentiate that the expression `ted` refers to the value `1776`, while `*ted` (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 included an explanatory commentary of how each of these two expressions could be read):

```
beth = ted;   // beth equal to ted ( 1776 )
beth = *ted;  // beth equal to value pointed by ted ( 25 )
```

Notice the difference between the reference and dereference operators:

- & is the reference operator and can be read as "address of"
- * is the dereference operator and can be read as "value pointed by"

Thus, they have complementary (or opposite) meanings. A variable referenced with `&` can be dereferenced with `*`.

Earlier we performed the following two assignment operations:

```
andy = 25;
ted = &andy;
```

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

```
andy == 25
&andy == 1776
ted == 1776
*ted == 25
```

The first expression is quite clear considering that the assignment operation performed on `andy` was `andy=25`. The second one uses the reference operator (`&`), which returns the address of variable `andy`, 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 `ted` was `ted=&andy`. The fourth expression uses the dereference operator (`*`) that, as we have just seen, can be read as "value pointed by", and the value pointed by `ted` is indeed `25`.

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

```
*ted == andy
```

# Declaring variables of pointer types

Due to the ability of a pointer to directly refer to the value that it points to, it becomes necessary to specify in its declaration which data type a pointer is going to point to. It is not the same thing to point to a `char` as to point to an `int` or a `float`.

The declaration of pointers follows this format:

```
type * name;
```

where `type` is the data type of the value that the pointer is intended to point to. This type is not the type of the pointer itself! but the type of the data the pointer points to. For example:

```
int * number;
char * character;
float * greatnumber;
```

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 will occupy the same amount of space in memory (the size in memory of a pointer depends on the platform where the code is going to run). 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 `float`. Therefore, although these three example variables are all of them pointers which occupy the same size in memory, they are said to have different types: `int*`, `char*` and `float*` respectively, depending on the type they point to.

I want to emphasize that the asterisk sign (`*`) that we use 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 that we have seen a bit earlier, but which is also written with an asterisk (`*`). They are simply two different things represented with the same sign.

Now have a look at this code:

```
// 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 << endl;
  cout << "secondvalue is " << secondvalue << endl;
  return 0;
}
```

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

Notice that even though we have never directly set a value to either `firstvalue` or `secondvalue`, both end up with a value set indirectly through the use of `mypointer`. This is the procedure:

First, we have assigned as value of `mypointer` a reference to `firstvalue` using the reference operator (`&`). And then we have assigned the value 10 to the memory location pointed by `mypointer`, that because at this moment is pointing to the memory location of `firstvalue`, this in fact modifies the value of `firstvalue`.

In order to demonstrate that a pointer may take several different values during the same program I have repeated the process with `secondvalue` and that same pointer, `mypointer`.

Here is an example a little bit more elaborated:

```
// 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 by p1 = 10
  *p2 = *p1;         // value pointed 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 << endl;
  cout << "secondvalue is " << secondvalue << endl;
  return 0;
}
```

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

I have included as a comment on each line how the code can be read: ampersand (`&`) as "address of" and asterisk (`*`) as "value pointed by".

Notice that there are expressions with pointers `p1` and `p2`, both with and without 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).