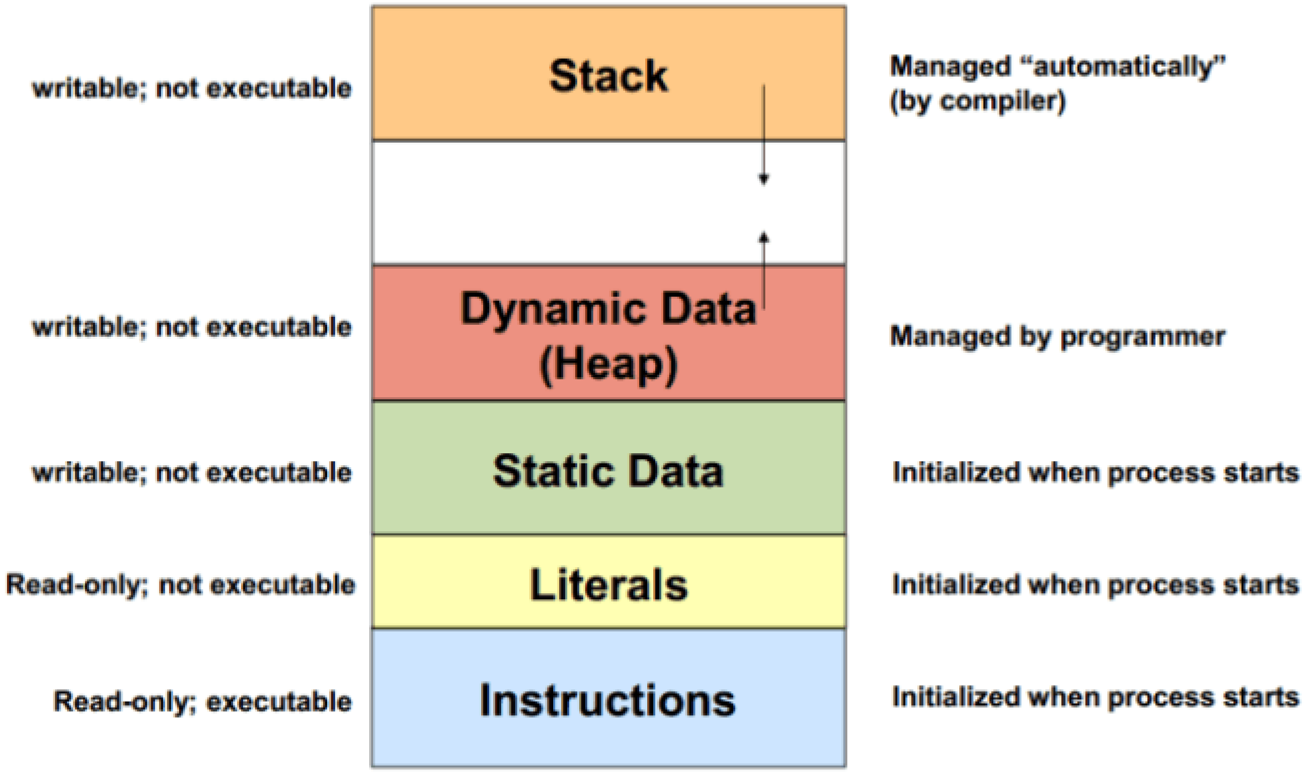
C++

20. marts 2025

*

&

Stack og heap



Stack og heap

- Stakken kaldes også *runtime stack.*
- Stakken indeholder *frames* eller *activation records.*
- Groft sagt en frame for hvert metodekald med værdier af variable etc.
- Når en metode terminerer, fjernes den tilhørende frame fra stakken.
- Stakken er mindre end heapen og løber af og til fuld.
- Stakken administreres af operativsystemet.
- Heapen er større end stakken og allokeres dynamisk af programmet – i C++ ved hjælp af pointere samt kommandoerne `new` og `delete`.

Memory allocation

- Indtil videre har vi beskrevet variable som steder i computerens hukommelse, som kan tilgås ved deres *identifier* (navn).
- På den måde skal programmet ikke bekymre sig om variablens fysiske adresse i memory.
- Den anvender simpelthen bare navnet, når der er behov for at referere til variable.
- For et C++ program er memory en række på hinanden følgende memory addresser, som hver har en længde på én byte og med en entydig adresse.
- Disse enkelt-byte memory celler er ordnet således, at det er muligt at organisere repræsentationer af data, som er længere en en byte i på hinanden følgende (consecutive) addresser.

Memory allocation

- På den made kan hver celle let lokaliseres i memory ved hjælp af sin adresse.
- Fx følger memory cellen med adressen 1776 umiddelbart efter cellen med adressen 1775 og kommer lige før cellen med adressen 1777.
- Den kommer desuden præcis 1000 celler efter 776 og 1000 celler før 2776.
- Når en variabel bliver erklæret, skal den del af memory, som skal bruges til at gemme dens værdi, tildeles en specific adresse i memory.
- Generelt tildeler C++ programmer ikke eksplicit præcise positioner i memory (addresser) til sine variable.
- Den opgave varetages typisk af operativsystemet, som afvikler programmet, og bestemmer de enkelte addresser på kørselstidspunktet (runtime).
- Det kan imidlertid være nyttigt for et program at kunne hente en variabels adresse i runtime for at kunne tilgå data som er på en bestemt position i forhold til adressen, fx det næste element i et array.

Pointere i heapen

- De efterfølgende eksempler viser, hvordan operatorerne & og * virker.
- Ikke nødvendigvis hvordan man skal bruge pointere.
- Fx har jeg svært ved at se nytten af at erklære en pointer til en simpel datatype, fx int.
- Men pointere bør anvendes til *containere,* fx array og `vector`.
- Min anbefaling er konsekvent at anvende vector i stedet for array.

Pointere

- De efterfølgende slides er en klassisk gennemgang af * og &.
- Nyere versioner af C++ lægger op til mere begrænset anvendelse af pointere end tidligere.
- Desuden anbefales det, at overveje at bruge *smart pointers.*

Reference operator &

The address of a variable can be obtained by preceding the name of a variable with an ampersand sign (&), known as*reference operator*, and which can be literally translated as "address of". For example:

```
foo = &myvar;
```

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

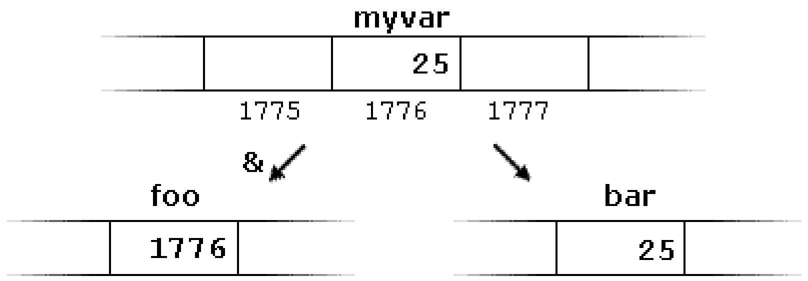
The actual address of a variable in memory cannot be known before runtime, but let's assume, in order to help clarify some concepts, that myvar is placed during runtime in the memory address 1776.

In this case, consider the following code fragment:

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

Reference operator &

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 *reference operator* (&).

Pointere

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.

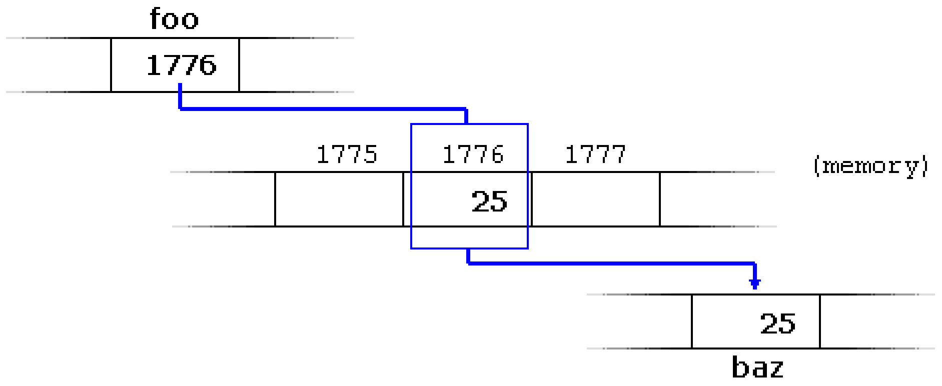
An interesting property of pointers is that they can be used to access the variable they point to directly. This is done by preceding the pointer name with the *dereference operator* (*). The operator itself can be read as "value pointed to by".

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

```
baz = *foo;
```

`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.

baz and foo

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* :

```
baz = foo;  //baz equal to foo(1776)
baz = *foo; //bas = value pointed to
            //by foo (25)
```

& and *

The reference and dereference operators are thus complementary:

& is the *reference operator*, and can be read as "address of"

* is the *dereference operator*, and can be read as "value pointed to by"

Thus, they have sort of opposite meanings: A variable referenced with & can be dereferenced with *.

Examples

Earlier we performed the following two assignment operations:

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

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

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

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:

Declaring pointers

```
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.

Declaring pointers

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.

Example

```
// 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;
}
```

What is printed?

```
firstvalue is 10

secondvalue is 20
```

Explanation

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 reference 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.

Another example

```
#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;
}
```

Explanation

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).

Declaration of pointers

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.