C++ a more in depth look into the language

Martin Nestorov

January 6, 2018

Contents

Who am I?

- Martin Nestorov
- Junior 2nd sem.
- I like to type . . . a lot
- email: mdn150@aubg.edu
- Twitter: @mnestorov

Why I chose to make these small tutorials

- People seem to have some problems with the transition from C++ to FDS.
- I want to help!
- I want to also learn!

Over the course of the past several years I found a certain love for C++, but instead of just reading about the language and doing some small experiments with it, I wanted to share my knowledge and to get better at it. Seeing how some students struggle with C++ in the beginning courses, I decided that I can help out. That way not only am I learning more in-depth concepts, but also I am making the lives of students easier (hopefully).

How to look at C++

C++ as **four** sub-languages

- C
- OOP
- STL
- Templates

There is an old joke about C++ - "How do you shoot yourself in the foot? In C you just shoot yourself in the foot. In C++ you accidentally create a dozen instances of yourself and shoot them all in the foot. Providing emergency medical assistance is impossible since you can't tell which are bitwise copies and which are just pointing at others and saying, "That's me, over there."

C++ can sometimes be vary annoying, because we have absolutely no idea how it works and what it does. The fact that having very similar syntax for wildly different things doesn't help either. So instead of just banging our heads until bleeding sets in, we can try to understand a little bit better what is happening with this language and (hopefully) try to minimize **undefined** behavior. In fact C++ is that special language where if you don't know that you are doing, the language re-enforces that by acting in a way where it doesn't know what it might do. That's pretty scary. That's why I will try to minimize this effect with these tutorials and help you out to make better C++ programs and to introduce effective methods or writing C++ code.

C++ can be generally divided into four sub-categories or sub-languages. It's a daunting task to go over all of these aspects and trying to teach everything, mainly because I can't! But instead I will try to cover some topics that haven't been covered in depth in the C++ lectures and are somewhat elude to the students. Now our list seems like this

What we are going to cover

- File structure
- Memory management
- Inheritance
- Templates

- OOP
- STL

File structure

- Advantages:
  - Better code organization
  - Faster compiling
  - Separation of **implementation** from **interface**
- Disadvantages:
  - More complicated structure

It has come to my attention that many people, while writing their homeworks, neglect the advantages of structuring their **source** files. This is most noticeable when they are writing the first 3 FDS homeworks or their FDS project. Being able to properly manage your project into several files and to navigate between them in an optimal and cohesive way will minimize your C++ suffering (and coding experience in general).

Now most of you already know this, but there are several great advantages to separating your **implementation** (cpp) files from your **interface** (h) files. One advantage is that this will bring code clarity to your program. Instead of navigating through one big **400** line `Source.cpp` file, and wondering where that one definition of that object is and then going back to the `main()` method to fix one line, you will have a nicely structured system. Every class will have it's own implementation and interface file, thus minimizing code searching and optimizing code writing! But that's not all. This separation will allow you to compile your programs even faster. That is, every **source** file is compiled on its own. When you make a small change in your original 400 line monstrosity, you have to wait for all of that code to **re-compile**, but if you have the files separated, you just have to re-compile only the changes files, making waiting time much smaller (naturally this effect can be seen when compiling large programs, while small ones won't have such noticeable impact on time). And lastly, we also gain the benefit of implementation from interface separation.

But what we are sacrificing for these benefits is that we are increasing our project complexity. This shouldn't scare us that much, but we do have

to keep this in mind - **We have to limit ourselves not to over layer our program**.

Since most of you are working with Visual Studio, here is how your structure looks like now: Picture 1

And here is how you would want it to look: Picture 2

Now I am not going to tell you how to add files and where to put them, I will just show you how to properly implement header and source files and how to avoid collisions and also how to stop writing those damn "xfz.h" files.

First things first. What's the difference between headers and sources? Apart from the obvious name difference, header files are considered to be the **interface** part of the source code, that is, the place where we explain what the program does, but **not** how it does it. The source (cpp) files are the **implementation** part, this is where we explain **how** we implement the interface. Keeping them separate allows us to keep a clean directory and structure and to separate our logic from our implementation.

How to work with headers?

```
 0  #include <iostream> //header we always include
 1  #include "awesomeclass1.h"
 2  #include "awesomeclass2.h"
 3  #include "awesomeclass3.h" // our own class headers
 4
 5  int main()
 6  {
 7      awesomeclass1 ac;
 8      awesomeclass2 ac2;
 9      awesomeclass3 ac3;
10
11      ac.do_something(ac2);
12
13      ac3 = ac;
14
15      return 0;
16  }
```

So what are header files and what do they do?

Header files, usually ending with the `*.h` file extension (can be also `*.hxx`, `*.hpp`, or nothing at all), are code files that are **included** in our source files. We can look at header files as code that only serves as the interface to our

classes. That is, it holds a declaration which we later on implement and/or include in our source files. Although we will talk about header files and how we `#include` them later on, what we can say now is that for every header file that is included, the contents of that file are put in the place of the `#include` keyword. What we can expect header files to do is to hold **declarations** inside of them. They just tell us what we are going to implement, but not **how** we are going to implement it (that's the job of the source files).

NOTE: that for the `iostream` we used angle brackets and for our classes we used quotes. This just means that the header file **iostream** comes with the compiler and that the linker should look in the system directories for the definition. The quotes indicate that we are providing the header file and that the linker should look at the build directory.

Header guards

```
 0  // x.h
 1  class x { . . . };
 2
 3  // a.h
 4  class a { X x . . . };
 5
 6  // b.h
 7  class b { X x . . . };
 8
 9  // source.cpp
10  #include "a.h" // include x.h for the first time
11  #include "b.h" // error! include x.h again!
```

```
0  // x.h
1  #ifndef __X_H_INCLUDED__
2  #define __X_H_INCLUDED__
3
4  class x { . . . };
5
6  #endif
```

Because we are going to be using a lot of headers in our program, we would need to include all of them. But because these files are actually copied when we **include** them, we would have several problems. One would be the code duplication. Another one will be the multiple declarations of

classes and functions. To overcome this problem we would need to **guard** our header files in order not to use them more than once in our program. That is why we use the directives `#ifndef`, `#define`, and `#endif`. When we wrap our header file around with these directives, we ensure that the preprocessor will not copy a single file more than once into the source file.

Declarations vs Definitions

```
 0  // declarations
 1  extern int a;
 2  extern char b;
 3  class foo; // extern not allowed for user types
 4  double sum(double, int);  // no need for extern
 5
 6  // definitions
 7  int a; // the implementation of a
 8  char c; // the implementation of c
 9  double div(double a, int x) { return a / x; }
10  class foo { . . . };
```

I want to take some time now to revisit two old concepts before we continue - namely I want to talk about **declarations** and **definitions**. Here we will mention the role of the **compiler** and **linker** briefly, but we will cover them later on. Now a **declaration** introduces an identifier and its type, be that type of a variable, object, or a function. Basically we are saying to the compiler that there is something with this name and this type, but we are not specifying **how** this specific thing is implemented. This is what the **compiler** needs in order to make a reference to that declaration. We can imagine the situation where we have a function which we want to use in multiple files in our program, it's just a very useful function, but we don't want to provide the implementation of the function in every source file. That's why we just declare the function, the compiler sees the declaration, makes a reference to it without the need to know its implementation, and then when we build the final executable, the linker links every reference of the declared function to its implementation. If you remember from the C++ course, a function prototype is a declaration.

A **definition** in C++ is when we a define something in its complete form. This means that we need to both declare something and define it at the same time. We actually implement that specific identifier we have declared. When we write functions and give them a function body at the

same time, that is a form of definition. When we write `int x;` we are both declaring and defining the variable `x`. This means that we are telling the compiler that there is a variable `x` of type `int` that is in the global scope in the current source file. It's important to note that giving this variable a value is not necessary in order to define it. But in the examples we have this specific keyword `extern` that is associated with declarations. What `extern` does is that it marks the variable and tells the compiler that this is only a declaration and that a definition will be found somewhere else. In general we use `extern` when we want to declare global variables in header files and **define** them in some source file.

We can **declare** something as much as we like in as many files as we want, but we can only **define** something only **once**. Often when we get errors telling us that something is undefined (this is at linker time), what that means is that we have something declared, but we have not defined its implementation anywhere and we get a **missing symbols error**. On the other hand when we have multiple definitions of the same thing, the linker is confused as to which specific definition we want and we get a **duplicate symbols error**.

Okay now let us go back to header files.

When and how to use headers

`class A` ==> (uses) ==> `class B`

- do nothing if: A makes no references at all to B
- do nothing if: The only reference to B is in a friend declaration
- forward declare B if: A contains a B pointer or reference: `B* b;`
- forward declare B if: one or more functions has a B **object/pointer/reference** as a parameter, or as a return type: `B MyFunction(B myb);`
- `#include "b.h"` if: B is a parent class of A
- `#include "b.h"` if: A contains a B object: `B b;`

```
0  #ifndef __MY_CLASS_H_INCLUDED__
1  #define __MY_CLASS_H_INCLUDED__
2
3  // Forward declare dependencies
4  class F;
```

```
 5  class B;
 6
 7  // Include dependencies
 8  #include <map>
 9  #include "c.h"
10
11  // Our current class
12  class MyClass : public c
13  {
14  public:
15    std::map<int, char> matrix;  // a map object that is required by our class
16    F* f;                        // F pointer, so forward declare F
17    void Func(B& b);             // B reference, so forward declare B
18
19    friend class MyFriend;       // friend declaration is not a dependency
20                                 // don't do anything about MyFriend
21  }
22  #endif
```

Since we will be writing a lot of classes, there will be a lot of dependencies. As an example, a derived class will always be dependent of its parent class, which means that we must be aware of the parent class at compile time, so we can use the derived class. There are two general forms of dependencies - one is, that there will be things that can be **forward declared**, and the other is, that things need to be `#include`-ed. Let's take an example with the following. Image we have class `A`, which uses class `B`. This makes A dependent on B. To decide if we need to **forward declare** or to include A depends on what the interaction between the two classes is.

Some explanation is required here. First of we are protecting our class with the header guards. Then we are forward declaring two classes `F` and `B`. This means that we are telling the compiler that there is a user type class called **F** and **B** and that the implementation of the classes are somewhere else. This allows us to access the methods and variables of **F** and **B** without needing to specify their implementation. Forward declarations can be useful, because they can reduce compilation time. If we just want to access and use one or two functions from a class, instead of including the whole class (and copying its contents to the current file) we can just forward declare it and save ourselves some time.

In general we want to remember this simple rule - prefer forward declarations to includes, in order to save compilation time and circular dependency

issues. The benefits to these rules is that we are encapsulating the classes we use as much as possible, because the classes that use our classes don't need to know how they work. This means that if we restrict our use of includes and think about how the dependencies of the classes interact, we are writing encapsulated and decoupled code.

Circular Dependencies

```
0  // b.h
1  #include "a.h"
2  class B { A* a; };
3
4  // a.h
5  #include "b.h"
6  class B { B* b; };
```

I mentioned at some point about circular dependencies, but they are something that is very common when we are writing OOP code. In this example we see a circular dependency. If we were to just include both header files, we would be stuck in loop, trying to figure out which class to include first, giving us a compilation error. In order to overcome this (and keep in mind that there is nothing wrong with such circular dependencies) we can just forward declare one of the two classes and include the other so we can break from this cycle. Now before you ask "what about the situation where we have explicit object declarations instead of pointers to objects?", let's just say that such a situation is **impossible**. Not that you can't write such a program, but there is an inherit design flaw to such code, it's virtually impossible to make such a code work, and there is a clear flaw in logic if you find yourself in such a situation. The other question you might have is - well when should we use pointers to objects? This totally depends on your program. For instance, if we have a `class Car` and a `class Wheel`, it's logical for the **Car** to include the **Wheel** class, but the **Wheel** class only needs a pointer to the Car object (it doesn't need the whole Car).

How does compilation work?

There are 3 steps of compiling C++ code:

- Preprocessing
- Compilation

  - Compilation
  - Assembly
- Linking

When we are trying to run a piece of code (in our case C++), the computer goes through several steps before we see the output, or the errors.

Preprocessing

In order to get only the preprocessed file we can run the

```
g++ -E hello-world.cpp -o hello-world.ii
```

which will produce the `hello-world.ii` file and then we can look inside of it and find out what it includes.

The first part is the so called **preprocessing** step, where the **preprocessor** handles the `preprocessor directives`. These are the `#include`, `#define`, `#if`, `#ifndef`, `#ifdef` keywords we put on the top of our files. At this stage, one file at a time, each of these **directives** are replaces with their respective pieces of code from other files (they are usually only declarations). That is why when we have multiple source files, we include only the header files, because they only show us the declarations and not the definitions (thus we minimize time in this step). So after the directives have been replaced with the respective file contents or snippets of files (depending in the `#if` `#ifndef` and `#ifdef` and the macro `#define` keywords) we get at the end "pure C++" code. The preprocessor also adds line numbers so that the further steps can identify where the inserted code came from. As an example, if we write `#include <iostream>` we actually just insert the contents from the `iostream` library on the top of our main source file (again we must remember that most of the time, we are just including declarations).

As a side note, this whole process is very similar for C code as well.

So at the end of all of this copying, we get a temporary file that is just C/C++ code. It's indicated by the `*.i` or `*.ii` file extension, meaning that this file is just C/C++ code and must not be preprocessed.

We have to be careful where and how we put our **includes**.

Tip: Prefer using `const`, `enum`, and `inlines` to `#define`

```
#define A_RATIO = 1.18     // bad
const double ARatio = 1.18 // good
```

Tip: one of the things we want to do while writing C++ code is to minimize our reliance on the preprocessor. That is, if we are `#define`-ning macros as constants so that we can use them throughout our program, we might encounter strange errors, because these directives may be treated as not part of the language. As an example, if we write `#define A_RATIO 1.18` the preprocessor might skip the name and just include the double 1.18. Then if we get, or when we get, an error referring to 1.18, we might not know it, because it was a macro define lost from the preprocessor. Instead we can just use `const~s as such: ~const double ARatio = 1.18;` Now we know that the compiler will see this variable and we won't bang our head against the wall with unnecessary errors.

After we have our "pure C++" code (ending with the `*.i/*.ii`) suffix, we are ready to move to the next step - **Compilation**.

Compilation

- Step 1 - Compile
- Step 2 - Assemble

To get the object file we can run

```
g++ -c hello-world.ii
```

or

```
g++ -c hello-world.cpp
```

and we can then look inside what an object file looks like with

```
nm hello-world.o
```

or

```
objdump -t hello-world.o
```

The **compilation** step is another relatively simple phase, where the preprocessed pure C++ file is transformed into **assembly** code. From there the compiler invokes an underlying back-end (assembler tool-chain) and assembles the assembly code into **machine** code, thus producing an actual **binary file** (where there are different binary file formats such as: `EFL`, `a.out`, `COFF`, `SOM`). This is the so called **object file**, which contains the compiled code

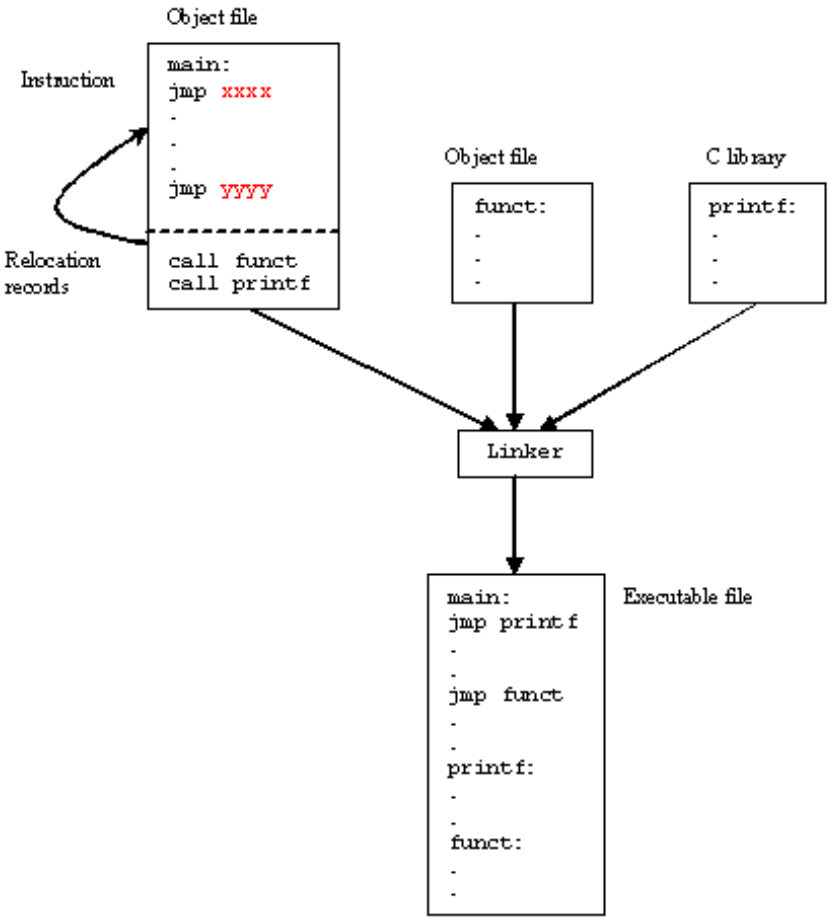
into binary form of the symbols defined in the input. This file is usually no directly executable. The object files also contain additional data in the form of sections, used for linking, debugging, symbolic cross-reference resolution, comments, re-allocations, program symbols, etc (sections can be `.text`, `.bss`, `.data`, `.reloc`, etc.). The object files contain the metadata that hold the memory locations (addressed) of the variables and functions (called symbols) into an associative data structure called a **symbolic table**. Note that these addresses might not be the final addresses of the symbol in the final executable. The things that might be interesting to us is the symbol table. This is a data structure in the object file that's basically a name and an index. It maps different items in the object file to names that the linker can understand. If you call a function from your code, the compiler doesn't put the final address of the routine in the object file. Instead, it puts a placeholder value into the code and adds a note that tells the linker to look up the reference in the various symbol tables from all the object files it's processing and stick the final location there.

One big advantage to this is that the **compiler** can stop the compilation at this phase. Because you won't need to re-compile every file, but only those that **have been** changed, you can specify which files to compile and save time. **IDEs** and some other tools can do this automatically and check the timestamps of the files and only compile those source codes which have been modified. On the compilation step we get the normal compiler errors, such as **syntax errors**, **failed function overload errors**, etc.

Once we have the object file we can transform it into special **executables**, **shared**, or **dynamic** libraries. Here the **linker** comes into play.

Linking

The **linker** just links all of the object files into one executable file.



The **linker** just links all of the object files into one executable file. The just of it is that the linker **links** object files by resolving undefined definitions of functions in the object files. That is, it goes through the object files and for every undefined function it tries to replace the reference of the undefined symbol with the correct address in another object file or in the standard library. The whole linking process is somewhat tedious and difficult to follow as it involves moving memory locations and relocation of symbols so we can skip this part, but for those who are interested, there are several links that explain exactly how the linker does its job.

One thing that we will encounter are the terms **dynamic** and **static** linking. **Static** linking is the process that links the program and the libraries together at normal link time. This means that the binding between the program and the library is known at link time. We are linking the program

statically to a **shared archive** of objects (libraries). An example would be the standard `libc.a` library for **C**. A draw back to this approach is that the size of the executable is quite big, because everything must be bundled together. These static libraries are identified by the `*.a` file extension.

Although the deployment of such **executables** is much easier and allows us to have **0 dependencies**, the size of the binary can get too big and such static linkage does not allow us to reuse memory for executable code between different processes. What this means is that when we have multiple executables that rely on the same library, unless our OS is very smart, it's very likely that we are loading the same piece of code over and over, incrementally increasing the memory we are using for the same piece of code. Another problem is that if we are to change something and have to **re-build** and **run** the executable, we would spend a lot of time reallocating with the static library.

To overcome this problem we can use **dynamic** libraries. For the Windows users, these are the famous `*.dll` files. In essence, we get an **incomplete** binary, which is told **during** runtime, where to search for the code in the respective library. That is - the linkage of the functions from the shared objects and our program is done during runtime right before the program starts. The linker just mentions to the executable that there is a function from a shared object used at this particular place and notes it in the binary, and then carries on. The symbols of the shared objects (the ones in the libraries we are using) are only verified and validated that they exist, but are not combined into the final executable binary. Thus we get several great advantages to using dynamic linking and libraries:

- Portable executables with smaller size.
- Standard libraries can be updated and re-patched without the need of re-linkage of every program.
- We can run multiple processes that use the same shared libraries without the need of copying the same code, thus saving large amounts of memory space.

This is the last step before we can take the `.exe` file, load it into memory and run it. At the linking stage we get different errors, such as **multiple function definitions**, or **undefined functions**, **missing references**, etc.

Loading and running - Now that we have a ready executable file we just have to **load** it into memory and run it. The **loader** is a general part of the OS and it operates in several steps. The general idea is this - first

we validate memory and access privileges to the exe. The OS reads the header of our binary, checks if we have enough space to run the program, checks what kind of access permissions we have, checks the ability to run the instructions, makes sure that this is a valid executable image, and then goes through several steps of loading. To be exact - it allocates primary memory to run the file, copies the address spaces from secondary to primary memory, copies the multiple sections of the executable to the primary memory, copies the command line arguments on to the stack, refreshes the register and re-points the **esp** (the stack pointer) to the top of the cleared stack, and finally jumps to the start of the program and runs the `main()` method.

Conclusion

Understanding undefined behavior better Understanding errors Good grip on how data is represented in C++

Conclusion - we can see that this is somewhat of a long process, where a lot of steps take place. This is done, from one point of view, for easier implementation and reduction of complexity. Being able to control all of these functionalities allows us to create big programs, to compile them in an easy and fast manner, and to understand what kind of errors we are getting at what stage. With the powers of "conditional compilation" we are able to create pre-compiled libraries that need only linking, this is called a "separate compilation model". Knowing the difference between the compilation phase and the link phase can make it easier to hunt for bugs. Compiler errors are usually syntactic in nature – a missing semicolon, an extra parenthesis. Linking errors usually have to do with missing or multiple definitions. If you get an error that a function or variable is defined multiple times from the linker, that's a good indication that the error is that two of your source code files have the same function or variable.

Memory management

The memory layout can be divided into **five** sections:

text data bss heap / free store stack

The different segments in memory are the **text**, **data**, **bss**, **stack**, and **heap**.

The **text** segment holds the executable instructions inside. The OS tries to make is so that if the same program is running on multiple instances, this part of the code is shared between the individual processes, instead of being copied multiple times.

The **data** segment is where the non zero initialized global and statically allocated variables are. Each running instance of the program has an individual segment holding this piece of data.

The **bss** segment (**B** lock **S** tarted by **S** ymbol) is where all of the zero initialized global and statically allocated variables are. Again, each running instance has an individual bss segment. While running the bss segment is stored in the data segment, but in the execution file it is stored in the bss section.

The **heap** is the dynamic part of the memory allocation (**C** uses `malloc()`, `calloc()`, and `realloc()`, while **C++** has `new`). We should make a quick clarification here. You might encounter two different terms that are often used interchangeably - one is the **heap** and the other is the **free store**. The difference between them in terms of their functionality is none, but following the C++ standards we can see that the heap is never mentioned apart from being an **abstract data structure**. This is so, because the heap is allocated or freed via `malloc()`, `calloc()`, `realloc()`, and `free()`, while the free store is allocated or deleted with `new` and `delete`. Although `new` and `delete` might be implemented in terms of `malloc()` and `free()`, these are not the same memory locations and they cannot be used **safely** interchangeably. For the sake of simplicity, we will continue referring to them under the **heap** term, but note that C++ does not used the heap the way C uses it.

Everything in this part of the memory is anonymous and needs a pointer to gain access to it. When we allocate new memory the process address space grows upwards. This means that as new items are added, the addresses of those items are numerically greater than the addresses of the previous ones. To free up memory from the heap we use `free()` for C and `delete` for C++, thus leaving holes in the memory. This means that when you are allocating objects to the heap and then deleting them, because of their different size, you might get into the situation where some deleted object free up space between objects that are still on the heap. Thus physically leaving free space that cannot be used by larger objects. This is the idea of leaving holes.

We can picture it as if we have a blank wall and then start arranging pictures on it. If we are not careful with our picture arrangement we might get most of the pictures on the wall, but at some point we might get small free spaces that are just blank wall. Thus we technically do have space for more pictures, but this space is fragmented and unusable for bigger pictures (presuming that we cannot chop up our pictures into pieces). This is the same with the memory allocation and de-allocation on the heap. On our machines, where we have virtual memory, we don't really experience this problem,

because it is important for the virtual memory to have the object into one continuous block. We can experience this problem of memory fragmentation when we start getting allocation errors (such as `malloc()` returning `null`), or when we cannot free up memory properly, or when our program takes too long to reallocate memory.

To overcome this problem we might use some tactical position of object creation to avoid such problems. We can allocate objects from different areas according to their size and/or their expected lifetime. So if you're going to create a lot of objects and destroy them all together later, allocate them from a memory pool. Any other allocations you do in between them won't be from the pool, hence won't be located in between them in memory, so memory will not be fragmented as a result (Using a good algorithm for allocating memory, we can, instead of allocating memory for a lot of small objects, pre-allocate memory for a contiguous array of those smaller objects. Sometimes being a little wasteful when allocating memory can go along way for performance and may save you the trouble of having to deal with memory fragmentation).

In general we don't have to worry that much for this sort of fragmentation unless our program is long running and has a wide mixture of long lived/short lived, big/small objects that are constantly created and destroyed. But even then the automatic memory allocation is on our side and helps us as much as it can. So we can start worrying about this only when we see clear sings of slow processes and blatant memory errors.

The great thing about C++ is that the **STL** handles these allocations very well and it's optimized so if we are relying on the STL (and we should), then we wouldn't have any problems.

The end of the heap is indicated by the `break` pointer. It is impossible to allocate more data beyond this range, but with system calls `brk()` and `sbrk()` we can move the break further up the memory and free up more space for our running program (keep in mind that such direct system calls are generally a bad practice and should be avoided).

The **stack** is the static part of the memory allocation in our program. Here local variables are allocated. These are all the variables that are declared inside a function body and are not set as `static`. Following the stack data structure, local variables, function parameters, addresses, etc. are popped up or pushed down for quick and easy access.

When a function is called a **stack frame** (a procedure activation record) is pushed on top of the stack. A stack frame holds information for the address from where the function was called, where to jump back when the function ends (**return address**), local variables, function parameters, and any other information needed by the function. When the function returns, the stack

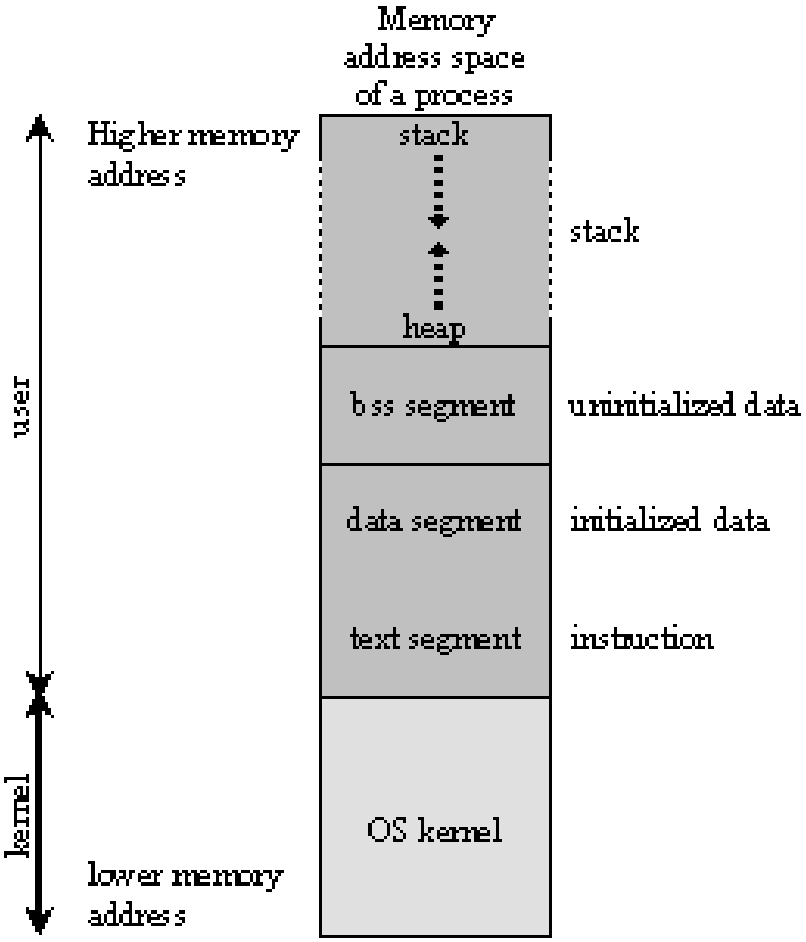
frame is popped from the top of the stack. The stack grows downwards, meaning that the address of each stack frame is numerically smaller than the previous one.

So when a program is running, the data, bss, and heap segments are aligned into one continuous memory block (area) called a data segment. The stack is kept separate from them. In theory it is possible for the stack and heap to collide and grow inside each other, but the OS prevents such collisions.

Memory space in C++

This is all the space and data the program needs in order to run properly.

$$addressspace = memoryspace$$



How to handle memory in C++

- Pointers
- Pass by value
- Pass by reference

Before we start actually looking at any code. . . why do we use pointers? Well we know that it **saves** time but how exactly? We saw that in order to get data from the free storage we must use pointers, but is that the only benefit? Much to our **"surprise"**, there are a lot of awesome advantages to using pointers, and with a little practice we can enjoy them.

So first things first, why do we save time and space with them? Because C++, by default, **copies** every parameter we pass in to a function, which means that if we have large object (or large variables for that matter) which we must pass to a lot of functions, that object, and all of its data, will be copied to the stack frame of that function and then, when we leave the function, the copied data will be discarded. This means that when we are passing by value, a copy of **every** parameter is made and we make changes to the **copy**, **not** on the originally passed data. Not only are we doing unnecessary copy work, but we might also fall into the trap that we are transforming the object and changing its state, but in reality, we are doing only work with the copied object. Here is a practical example - two years ago, I wrote a C++ program that passed objects like crazy in order to render an image at the end. The program, depending on the level quality it rendered, took between 30 secs to 30 minutes for the highest quality. The same program, when it used the power of passing by reference, took from 10 secs to 5 minutes max. Ain't optimization beautiful.

Lets get back to the world of C++ and go over how we pass arguments to functions. In particular we will try to review the difference between passing by value and passing by reference and dealing with pointers.

Lets take this example now:

Pass by reference in C

```
 0  #include "stdio.h"
 1
 2  void f(int *j)
 3  {
 4      (*j)++;
 5  }
 6
 7  int main()
 8  {
 9      int i = 20;
10      int *p = &i;
11      f(p); // or f(&i); will yield the same result
12      printf("i = %d\n", i);
13
14      return 0;
15  }
```

With this example we can see how, although we are passing a pointer, we are still employing the pass-by-value idiom. This is normal for all C code (this will also work for C++ as well). This type of passing is called **"C-style pass-by-value imitation"** . The code does what we expect, because after the function call we **de-reference** the pointer which we have passed to the function in order to get the integer it was pointing to. **De-referencing** means that we are evaluating what the pointer points to. In our case, when the function call finishes and we de-reference the pointer, we are just looking at the value to which the pointer is pointing. So here is what's happening, when we pass a pointer to the function, that pointer gets copied, and that copy is used in the function. At this point we have two pointers, which are pointing to the same value. When we go through the function body, the **copied** pointer **changes** the value it points to (in our case it changes the `i` variable declared in `main`). In other words, the copy of the pointer in the function body changes the same `int` we have in memory, regardless of the fact that a copy pointer is doing that (the integer is still changed because we are pointing to that integer). This is how we **"imitate"** pass-by-reference, the old way.

Let's look at the same example, but in C++ code.

Pass by reference in C++

```
 0  #include <iostream>
 1
 2  void f(int& a)
 3  {
 4       a += 1;
 5  }
 6
 7  int main()
 8  {
 9      int a = 5;
10      f(a);
11      std::cout << "a = " << a << std::endl;
12
13      return 0;
14  }
```

Doing the exact same thing, the difference is that we have no pointer declared, because we don't need one, and we changed the sign of parameter

we pass, from an `*` to an `&`. Here C++ and it's awesome new feature allows us to actually pass a real reference to the function. This means that the function is working with the real data we have passed, and **not** with a copy of it. If we make changes to the variable inside the function, those changes will **stick** once we leave the function!

Hopefully this cleared some things about pointers, but the question now is: when should I pass by reference and when by value, and when I am passing by reference, which of the two ways should I employ? The general answer to this is - it depends on the code you are writing and the problem you are solving, but because this is too generic, here is some better advise - when writing C, imitate the pass by reference when you **want** the data to be changed, other wide, pass it by value. When writing C++, do the same, use the C++ style with real reference passing, and also, as a bonus rule, pass **objects** by reference (this minimizes the situation where you have to copy large amounts of data to functions that only read from the passed objects).

References

Preprocessing, Compilation, and Linking

- Objects Files
- Understanding compilation in C++
- Preprocessor
- Compilation and Linking
- Linking in C
- Linker
- Linkers part 1
- Working with ELF files
- What do linkers do?
- C++ compilation process
- Compiling and Linking
- Compiling and Linking in C++

- COMPILER, ASSEMBLER, LINKER AND LOADER: A BRIEF STORY
- Buffer Overflow
- Object files and symbols
- Static and dynamic libraries
- How to view symbols in object files
- What is a symbol table
- How to list symbol tables

Memory management

- What is memory fragmentation
- Pass by reference in C
- Difference between pass by reference in C/C++
- Array memory allocation
- Memory storage in C/C++
- References in C/C++

File Structure

- How to use header files
- Header files
- Why do we have header and cpp files
- Declaration vs Definition
- Difference between declaration and definition
- The PIMPL idiom
- Forward declarations