**Compile & Execute**

1. **Compile and Execute, with separate compilation**

**Building and Running file**

There are 4 steps that needs to be taken:

|  |  |
| --- | --- |
| main.cpp | header.hpp |
| Preprosessing 🡪 Compilation 🡪 Assembly  🡪 main.o | Preprosessing 🡪 Compilation 🡪 Assembly  🡪 header.o |
| Linking all .o files 🡪 Executable .exe file | |

**Preprocessing**: Resolve all '#', to copy paste every files and libraries we included; remove all comments

g++ -E main.cpp -o main\_preprocessed.i

* This will generate .i file, the output of the preprocessing function.
* Rule: Only include header file 🡪 to make objects interact with each other, we include the class declaration in the header file.

**Compilation**: Input c++; Output: **assembly code**

g++ -s main\_preprocessed.i -o main.s

**Assembly**: Input: assembly code; Output: **binary code** (machine code, object file)

g++ -c main.s -o main.o

Note: we can merge these 3 actions into 1 command, for compilation of each file into binary file

g++ -c hello.cpp -o hello

**Linking**: The machine code file is stored on the disk with extension <.o> but this is not executable. A linker then links the object code with **standard library routines** that the program may use (give instructions to run the program) and creates an executable file, usually as a file with the file name without any extension (program)

* Input: object file(s); Output: executable file
* Symbol resolution
* Relocation

To link multiple compiled files, we do

g++ <object\_file\_1.o> <object\_file\_2.o> -o program

**g++ \*.o -o program**

Now, to execute the new machine code file, we type ./program

The executable is loaded from the disk to memory and the computer’s CPU (Central Processing Unit) executes the program one instruction at a time.

In summary, to compile and link multiple files, we do:

g++ -c Linked\_list.cpp

g++ -c Node.cpp

g++ -c main.cpp

g++ Linked\_list.o main.o Node.o -o main

**Header File \*\***

It is convenient for bug fixing, and seeing the main() function not at the end of file, by using a **header file for all** **function declarations** – **with extension .hpp or .h**

We link the header file by using:

#include "header\_file.hpp"

-- > This action "copies" the header\_file.hpp content. However, this may lead to multiple declations of the functions (multiple .cpp files do #include).

Therefore, we need to include placeholders at the beginning and the end of header files

**#ifndef HEADER\_FILE\_NAME**

**#define HEADER\_FILE\_NAME**

… (header file content)

**#endif**

1. **\*\*Separate Compilation**

- ADTs: abstract data type 🡪 a class that has been defined so as to separate the **interface** and the **implementation** of the class. All class definitions should be ADTs. This has benefits in program compilations & fixing.

- The point of separate compilation is that we have **generated object files .o for all files, separately**.

If we instead used the command

g++ main.cpp func1.cpp func2.cpp -o linkedlist

🡪 Generates 1 object file, and linked them. Then if we make any changes to the files, we will have to **compile and link everything again**

Therefore, we would want to compile separately, and then link. If we change 1 file, we only compile THAT file, and then link them together.

***Compiling 1 file changed + Linking all files again is better than Recompile all files***

? What if we update the .h file?

**YOU WILL NEED TO RECOMPILE EVERY FILES #include THAT .h FILE**

1. **\*\*Makefile**

A utility that executes the necessary compilation (specifies what compilations are necessary)

Makefile is intelligent enough so that we can:

* Do a shortcut for separate compilation & linking: by just pressing **make**
* **make** recognize only the files changed, and compile only that file again.

Makefile format:

Text

Description automatically generated with medium confidence

"If you want to create this **target**, we need these **dependencies**. We do this by executing **command**"

Text

Description automatically generated with medium confidence

**make target** (= first\_target\_by\_default)

* First compilation: make can recursively find the commands to get the dependencies needed to get the target in first line
* Next compilations: make will check the timestamps (last modification time) of the dependencies, to see whether the dependencies have newer timestamps than the target (whether they are updated (changed). If so, the dependencies will need to be remake.

🡪 Automatically recompile the files changed. **We only need to type in "make" even in the next times.**