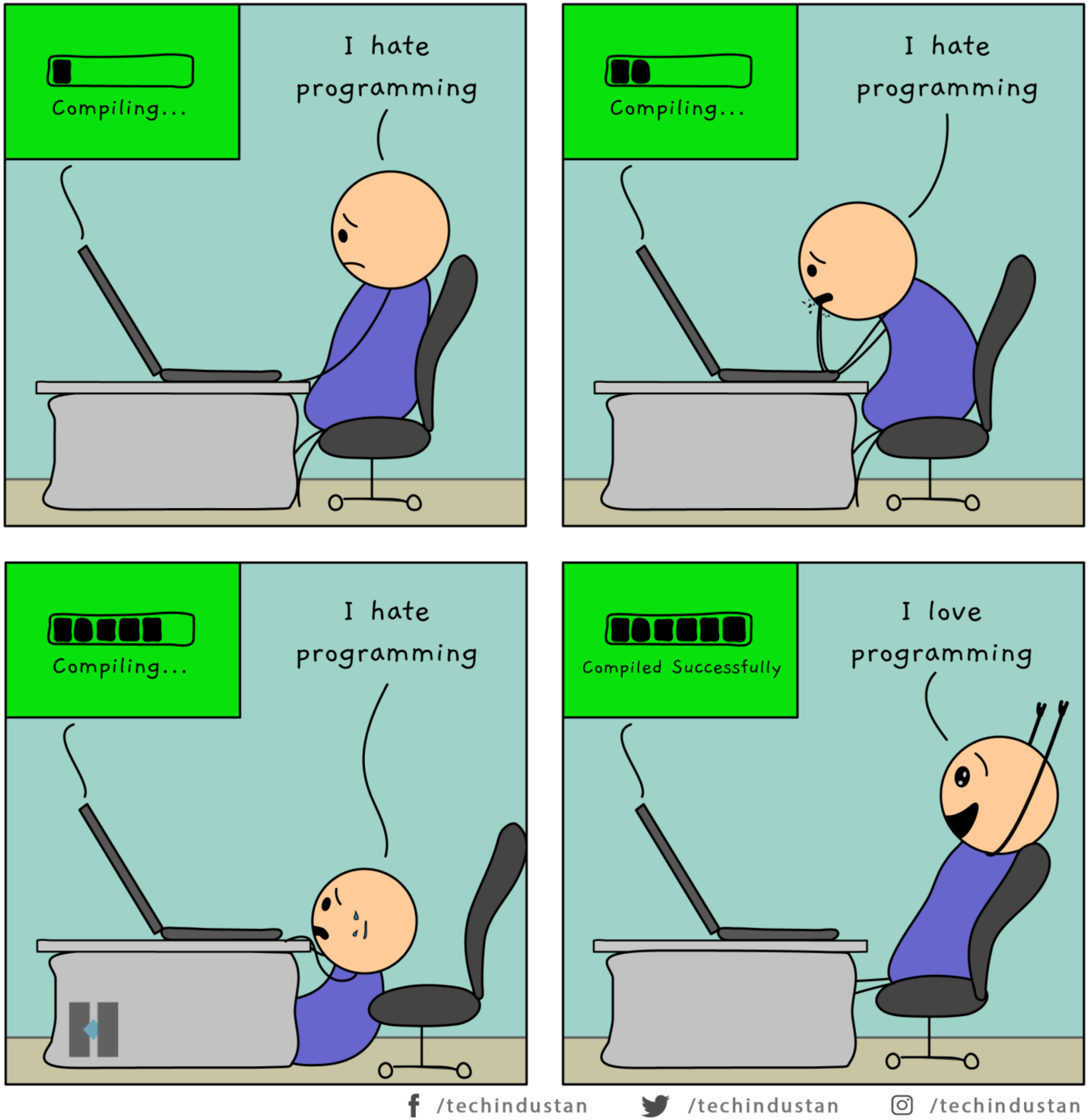
C Program Compilation

Computer Science Practice and Experience: Development Basics

CS1XC3

Professor: Kevin Browne

E-mail: brownek@mcmaster.ca



Source: https://www.reddit.com/r/ProgrammerHumor/comments/aknkqr/what_do_you_do_while_compiling_your_code/

C Program Compilation

- Thus far we have kept our programs to a single file
  - But we know C programs are made up of multiple files, as we are already including libraries like stdlib.h, etc.
- Our latest programs are starting to get pretty big!
  - The linked list and binary search tree code examples could really be a library for each data structure...
- Why do we split our programs across multiple files?
  - It's not just because it takes long to scroll through a file!

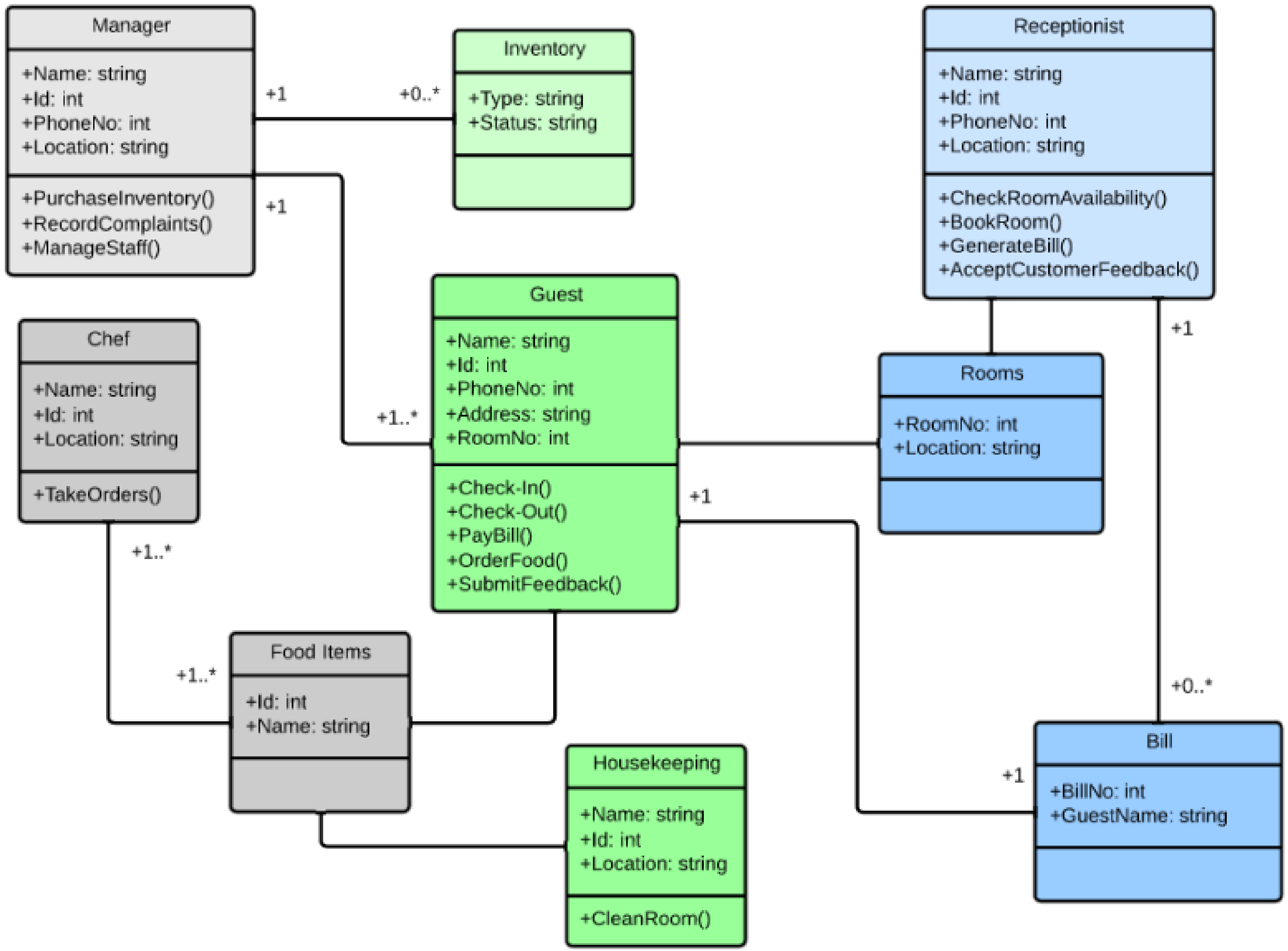
Software architecture

- **Software architecture** is about the structure of software systems and the processes to create them
  - If you keep studying CS you'll take courses on software architecture

- Software architecture arises from the need to build large software systems that are maintainable, modifiable, and extensible, among other traits
  - These criteria by which we can judge a system are called **non-functional requirements** or **quality attributes**
  - In contrats to functional requirements about behaviours

Component-based architecture

- A software **component** is a section of code that encapsulates a set of related functions and/or data
  - Exactly what a component is varies from one technology to another... a package, a library, a module, a file, a class in an OO langauge, etc.
- Writing programs as a series of interacting components is very important to good software arhcitecture
  - Closely related to the idea of modular programming and modularity



Source: https://d2slcw3kip6qmk.cloudfront.net/marketing/pages/chart/class-diagram-for-hotel-management-system-UML/UML_class_diagram_hotel-949x683.PNG

Component-based architecture

- We can replace a component with another and expect the system to still work
  - When a car gets new brakes, they'll work work with the existing muffler and engine
  - This makes a system more maintanble

- Developers can work in parallel on different components
  - As long as the ***interface*** (e.g. functions) for components is respected and agreed upon, this generally isn't a problem!
  - This makes a system more modifiable and extensible

Component-based architecture

- Components can be re-used in different projects
- If only some components are changed, only those components need to be re-compiled
- Components can be tested individually, which lead to easier debugging

Components in C

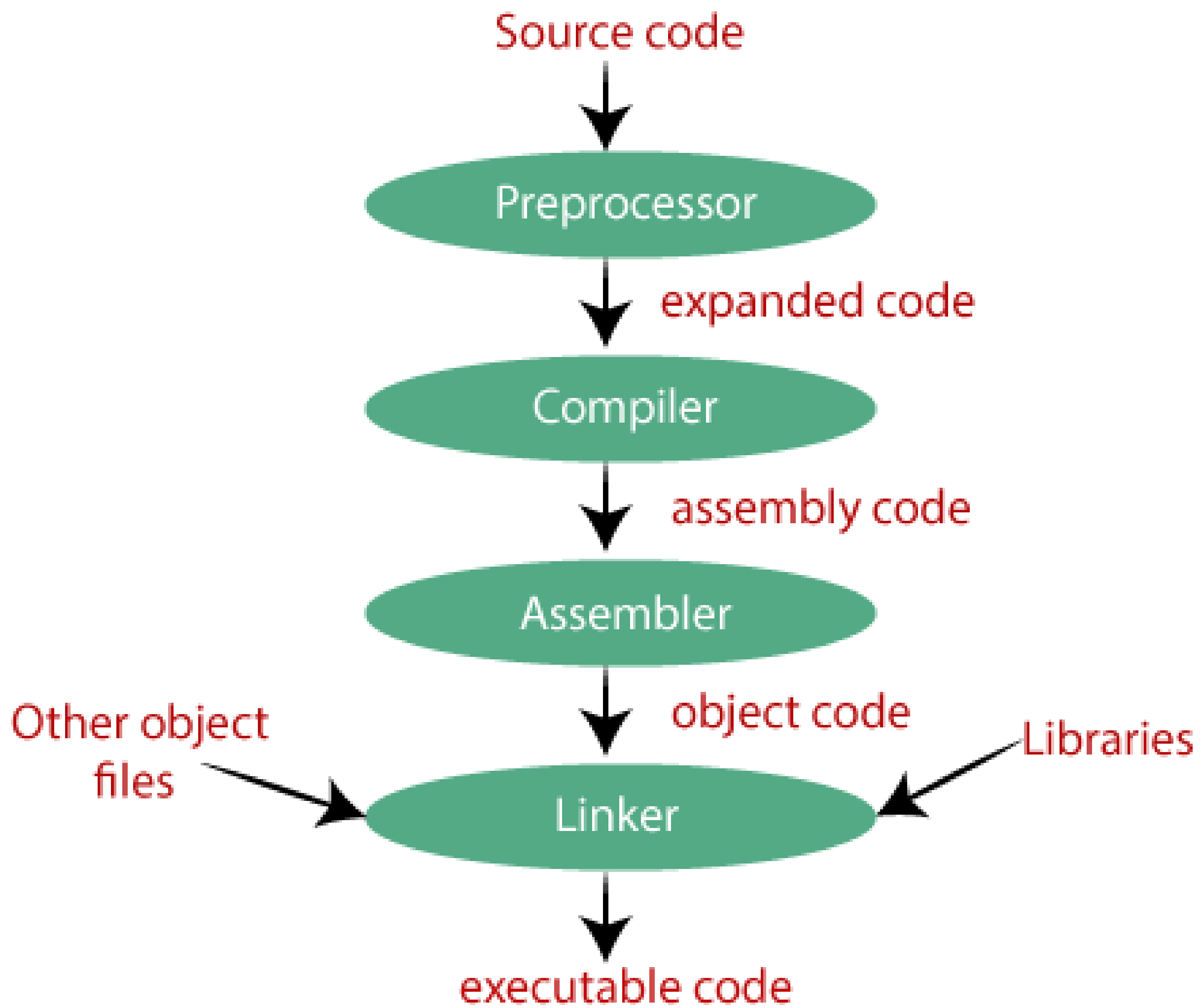
- As with many other things, C does not support the creation of components with the ease of other languages!

- We can create our own "components", or "modules" or "libraries" in C
  - We'll use the term library to describe our components, but the other terms wouldn't be wrong to use either

- How can we make libraries, and how do we use and compile them as part of larger programs?

C Compilation

- C Compilation takes place in 4 phases:
  - Preprocessor
  - Compiler
  - Assembler
  - Linker
- Each phase has an input and an output



Source: https://static.javatpoint.com/cpages/images/compilation-process-in-c2.png

Preprocessor

- The input to the preprocessor is the source code, and the output is an exapnded version of the source code
- #include statements are replaced with the text content of the header file
  - e.g. # include <stdio.h> is replaced with text that includes the function declaration for printf is replaced
- #define constants are replaced with their respective values throughout the source code
- Though we haven't covered them, **macros** and **conditional compilation directives** can be used to alter compilation, these are also processed during this stage
  - https://en.wikipedia.org/wiki/C_preprocessor

Compilation stage

- The compiler stage uses the expanded source code as input and produces assembly code as output
- The compiler stage parses the C program into a (potentially massive) tree data structure
  - Compilation algorithms are then used to produce assembly code from this tree data structure
  - Compilation algorithms attempt to optimize the performance of the assembly code produced
    - An entire field of study is dedicated to doing this, including researchers at McMaster

C code to assembly code example

```
int  dotp( short a [ ],  short  b [ ] )
{
      int  sum, i;
      int  sum1 = 0 ;
      int  sum2 = 0 ;

      for( i = 0; i < 100/2; i+2 )
      {
            sum1 += a[i] * b[i];
            sum2 += a[i+1] * b[i+1];
      }
      return  sum1  + sum2;
}
```

(a) C Code for Dot Product.

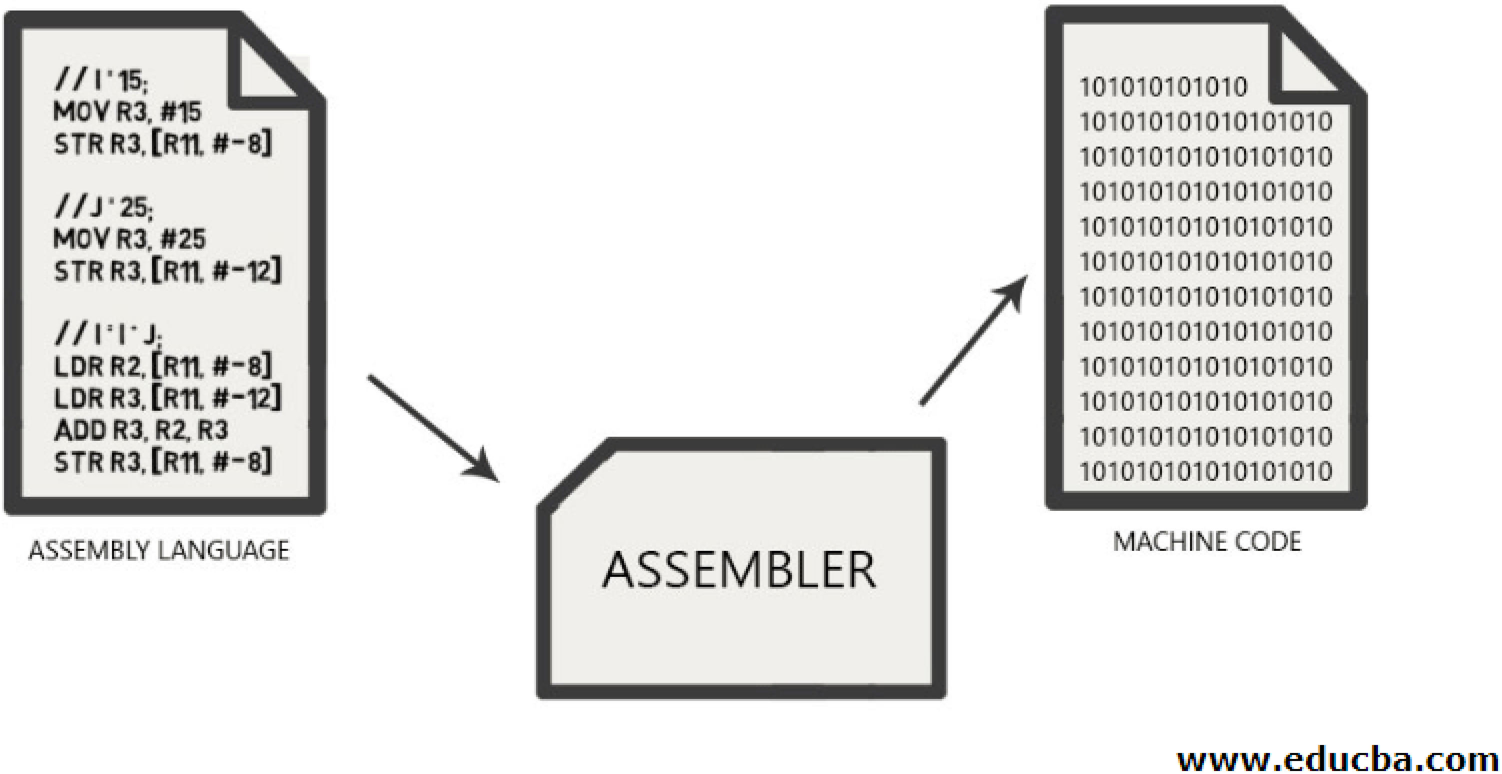
```
_dotp    .cproc   a,  b
     .reg  sum1, sum2,  i
     .reg  val_1, val_2, prod_1, prod_2
     mvk    50, i                ; i = 100/2
     zero   sum1                 ;  Set sum1 = 0
     zero   sum2                 ;  Set sum2 = 0
loop:                                                        Loop Body
     ldw     * a++,  val_1              ; load a[0, 1] and add a by 1
     ldw     *b++,  val_2               ; load b[0, 1] and add b by 1
     mpy    val_1, val_2, prod_1        ; a[0] * b[0]
     mpyh   val_1, val_2, prod_2        ; a[1] * b[1]
     add     prod_1, sum1, sum1         ; sum1 += a[0] * b[0]
     add     prod_2, sum2, sum2         ; sum2 += a[1] * b[1]
     add     −1, i, i                   ; i−−
     [i] b   loop                       ; if i>0, goto loop
     add     sum1, sum2, A4   ; get finial result
     .return  A4
     .endproc
```

(b) Assemly Code for Dot Product.

Source: https://www.researchgate.net/profile/Edwin-Sha/publication/220305740/figure/fig2/AS:670708619284480@1536920804237/A-dot-product-C-code-and-its-corresponding-assembly-code-from-TI-C6000-25.png

Assembler

- The assembler takes as input the assembly code and produces as output **machine code** (also known as **object code**)
  - Unlike compilation, nothing too complicated is happening here in terms of algorithms for optimization
  - Assembly code instructions like 'add' have a numerical representation in binary like 110011 and a translation is being made from one to the other
  - The processor can execute machine code directly
  - Assembly code is still somewhat comprehensible by a human, machine code is almost totally incomprehensible



Source: https://cdn.educba.com/academy/wp-content/uploads/2019/10/Assembly-Language-vs-Machine-Language-1.png

Linker

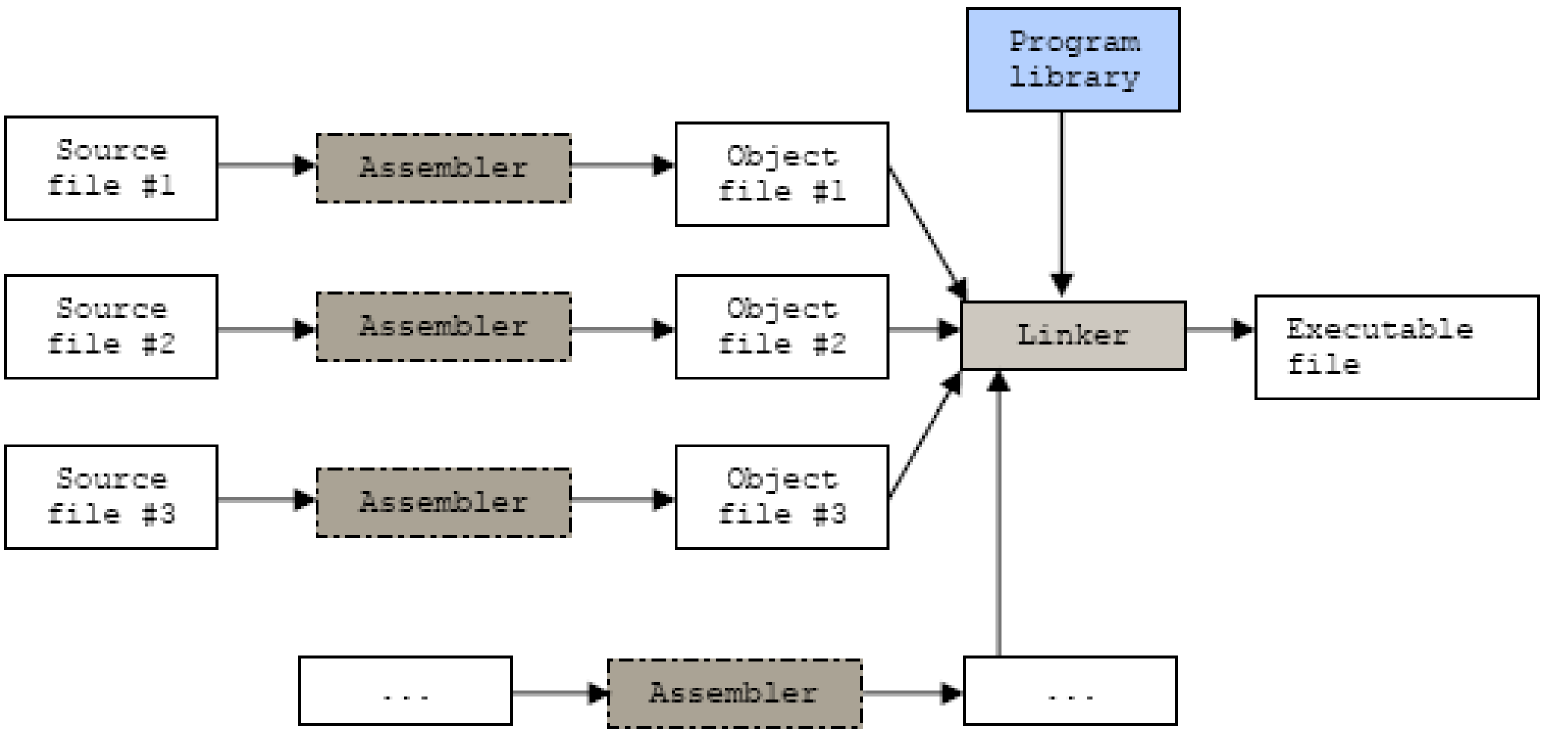
- If we use a program that includes stdio.h and uses printf, notably up until this point our code does not include the printf function!

- It has a function declaration for printf that is provided in stdio.h, but the actual function defintion is **not** provided in stdio.h

- The function definition is in another precompiled object code file that the linker combined together with our program
    - We can just call these files **object files**

Linker

- The linker takes as input the various object files required for the program and produces an executable file as output
- Object files could include the object files for standard C libraries (e.g. stdio, stdlib, etc.)
  - These have already been compiled to save time, they just need to be linked at this stage
- Object files could also include object files for libraries we have defined ourselves
  - We need to make sure the compiler knows about these files in order to include them

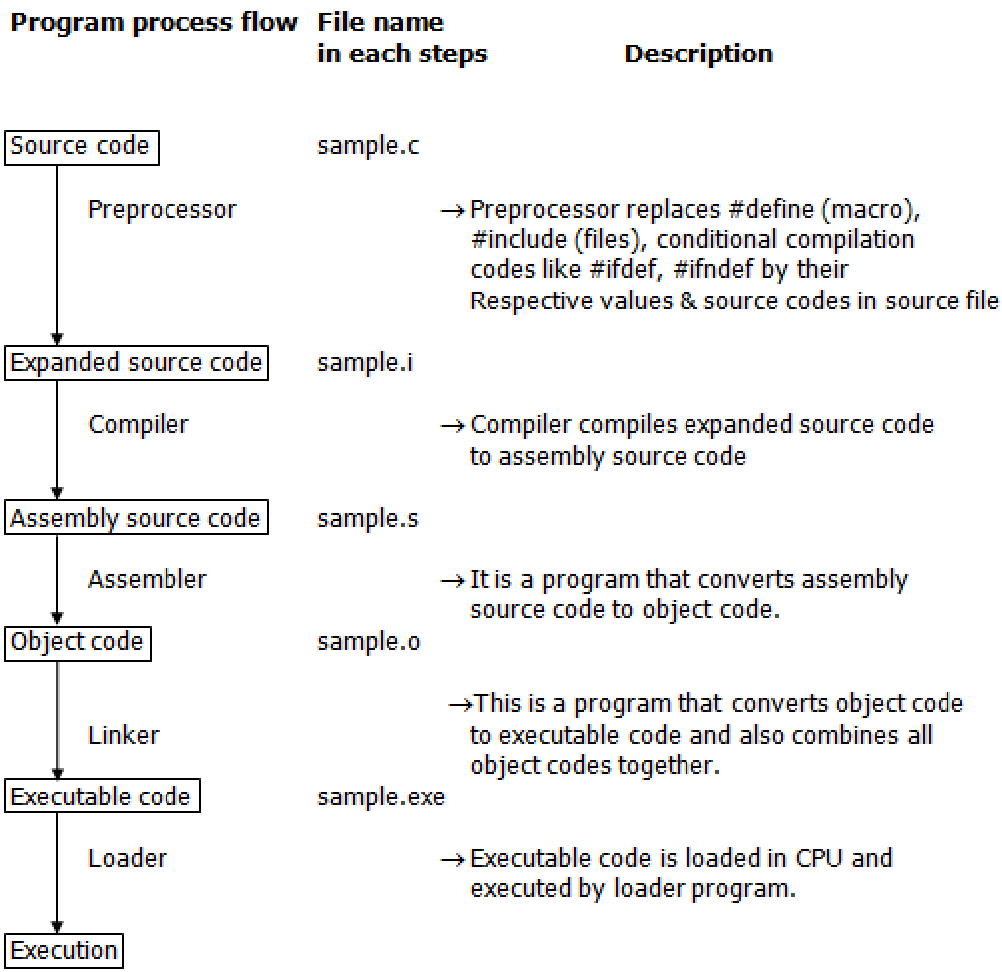


Source: https://www.tenouk.com/Bufferoverflowc/Bufferoverflow1c.html

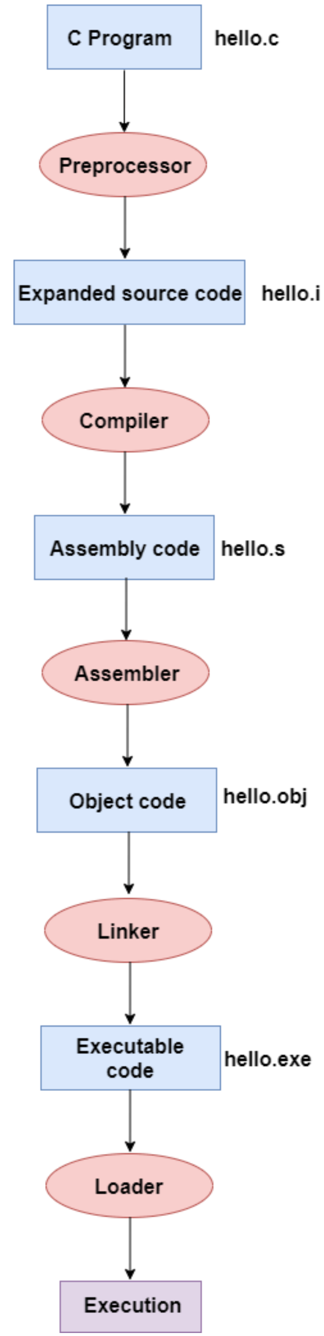
Executable file

- The executable file contains everything necessary to execute the program
- Remember that unlike Python or Java, C compilation is specific to a type of machine code and thus a type of machine
  - An executable file might not work on a different computer with a different set of machine code instructions!
- A loader program is responsible for executing an executable file

Compilation and execution



Source: http://fresh2refresh.com/wp-content/uploads/object%20and%20exe.png



Source: https://static.javatpoint.com/cpages/images/compilation-process-in-c3.png

Creating C libraries

- In order to create our own C libraries and include them in our program, we need to:
  - Create a .h file that specifies any function declarations, constant values, etc.
  - Create a .c file that contains the function definitions for each function declaration
  - Include the .h file in our .c file containing the main function
  - Compile **both** the .c file containing our main function AND the .c file containing the function definitions for our library

Creating C libraries

- When we create the .h file we are specifiying the **interface** for our library
- The interface tells source code files that include our library's .h file what functions are available
  - It does not tell them how they are implemented, only how they work in terms of parameters/return values
  - This is important for a component-based architecture, as code in one source file does not depend on code in the other beyond the interface that is defined
  - By specifying exactly how components interact, we make it possible to separate them, and change them independently without breaking things

Creating C libraries

- C libraries can include other C libraries

- They are not limited to only being included by the file that provides the main function definition

- For our first examples though, we'll assume our library is being include in the same file with our main function definition

Let's create a C library!

```c
#include <stdio.h>

int my_add_function(int a, int b);

int main()
{
  my_add_function(10,20);

  return 0;
}

int my_add_function(int a, int b)
{
  int result = a + b;
  printf("\nmy_add_function does some adding: %d\n\n", result);
  return result;
}
```

main.c

Program output

```
[brownek@pascal ~]$ ./main
my_add_function does some adding: 30
```

Creating a library

- Let's put the my_add_function() function definition in its own add.c file
  - We'll need to include stdio.h in this file now because it uses printf!

- And we'll put its function declaration in its own add.h file

- And then we'll include the add.h file in main.c

```c
#include "add.h"

int main()
{
  my_add_function(10,20);

  return 0;
}
```

main.c

```c
int my_add_function(int a, int b);
```

add.h

```c
#include <stdio.h>

int my_add_function(int a, int b)
{
  int result = a + b;
  printf("\nmy_add_function does some adding: %d\n\n", result);
  return result;
}
```

add.c

Notice the #include in main.c!

- We used **#include "add.h"** instead of **#include <add.h>**

- The **#include <something.h>** syntax is used for system library headers

- The **#include "something.h"** syntax is used for our own libraries created for our program

How do we compile this now?

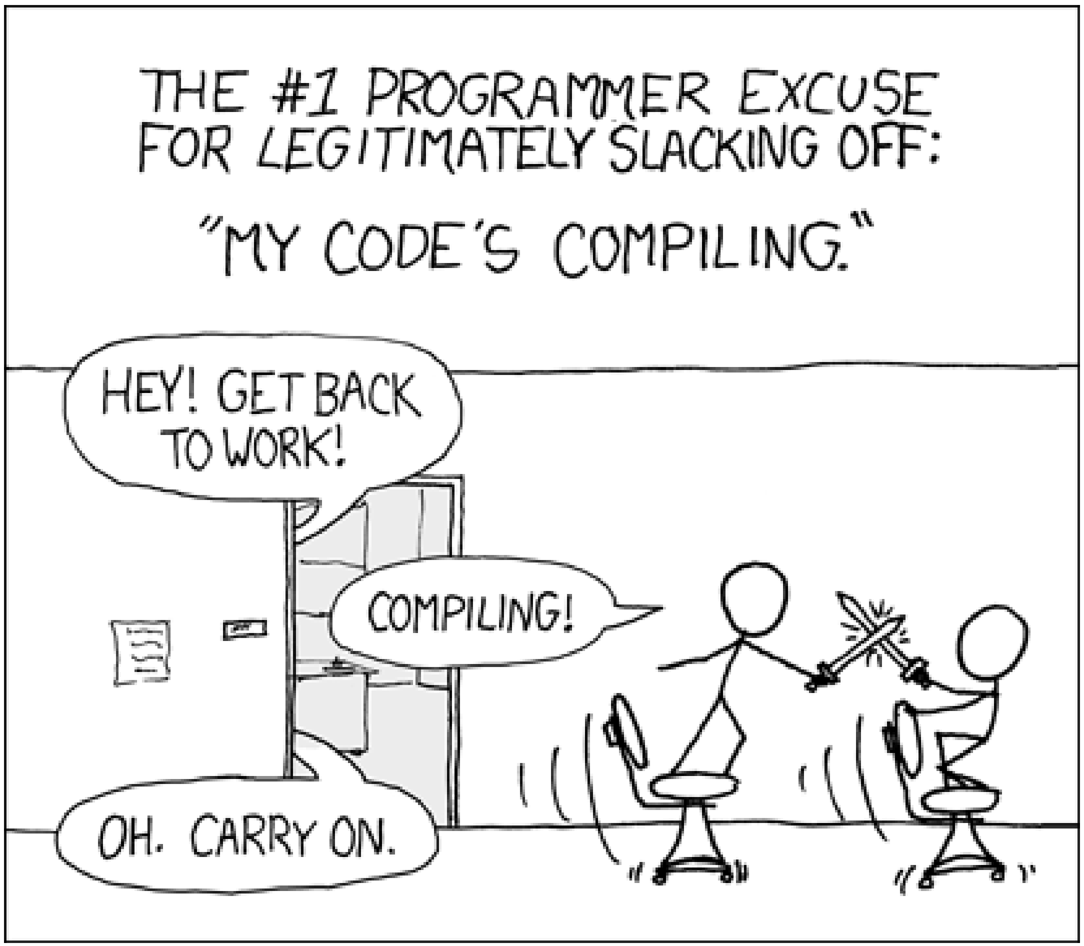
- We can run the command:
  - gcc –o main main.c add.c
- We compile ***both*** main.c and add.c
  - The linker will know how to put them together into one executable
  - The order of the arguments here won't matter
- add.h is included into main.c during the pre-compilation phase, we don't need to do anything here

Compilation and output

```
[brownek@pascal ~]$ gcc –o main main.c add.c
[brownek@pascal ~]$ ./main

my_add_function does some adding: 30
```

This famous xkcd comic really does have some truth to it...



Source: https://xkcd.com/303/

Compilation

- Compilation of very large programs can take a long time in practice
  - At a company I worked for in 2005 it would take 5-10 minutes to do a complete build of a very large program

- But developers typically work on their code with a write -> compile -> test cycle

- We need a faster way of compiling large programs

Compilation

- We can actually split up our compilation into multiple stages
  - Once created, libraries often won't need to be compiled again until they are changed again, which may be rare
  - We can compile them once into object files and only re-compile them when necessary due to cahnges
  - The creation of object files is the complex, time consuming step (with the tree data structures and algorithms)
  - Assembly and linker steps are much faster

- We can then compile only what we need to compile in order to save time

Compilation

- We can produce object files with –c in gcc
- Example:
  - gcc –c main.c
  - gcc –c add.c
- This will produce object files **main.o** and **add.o**

Creating object files example

```
[brownek@pascal ~]$ ls
add.c  add.h  main.c  public_html
[brownek@pascal ~]$ gcc -c main.c
[brownek@pascal ~]$ ls
add.c  add.h  main.c  main.o  public_html
[brownek@pascal ~]$ gcc -c add.c
[brownek@pascal ~]$ ls
add.c  add.h  add.o  main.c  main.o  public_html
```

Linking

- We can then link these files together with gcc to produce an exectuable

- Example:
  - gcc –o main main.o add.o

- This will produce an executable main that we can then run

Linking object files example

```
[brownek@pascal ~]$ ls
add.c  add.h  add.o  main.c  main.o  public_html
[brownek@pascal ~]$ gcc -o main main.o add.o
[brownek@pascal ~]$ ./main

my_add_function does some adding: 30
```

Now let's say we modify main.c...

```c
#include "add.h"

int main()
{
  my_add_function(10,20);
  my_add_function(5,5);
  my_add_function(10,-5);

  return 0;
}
```

Compilation

- After we modify main, there is no need to re-compile add.c
  - We can recompile main.c, and then link the object files again

- Example:
  - gcc –c main.c
  - gcc –o main main.o add.o

- This will re-compile main.c into a new object file and then link them together in a new executable

Compilation and linking example

```
[brownek@pascal ~]$ gcc –c main.c
[brownek@pascal ~]$ gcc –o main main.o add.o
[brownek@pascal ~]$ ./main

my_add_function does some adding: 30


my_add_function does some adding: 10


my_add_function does some adding: 5
```

The same is true if we make modifications to add.c...

```c
#include <stdio.h>

int my_add_function(int a, int b)
{
  int result = a + b;
  printf("\na: %d", a);
  printf("\nb: %d", b);
  printf("\nmy_add_function does some adding: %d\n\n", result);
  return result;
}
```

We only need to re-compile add.c and then perform another link..

Compilation and linking example

```
[brownek@pascal ~]$ gcc -c add.c
[brownek@pascal ~]$ gcc -o main main.o add.o
[brownek@pascal ~]$ ./main

a: 10
b: 20
my_add_function does some adding: 30


a: 5
b: 5
my_add_function does some adding: 10


a: 10
b: -5
my_add_function does some adding: 5
```

make and make files

- This process of compiling files will save compilation time, but it will get complex if we try to manage it manually
  - Imagine we have dozens or even hundreds of libraries in a complex program

- **make** is a program that automates this process for us using **makefiles** that contain directives for compilation of our program
  - By checking file save timestamps make is able to build only what is necessary to produce the executable

Example makefile

```
CC=gcc
main: main.o add.o

clean:
	rm -f main main.o add.o
```

Makefiles

- This example makefile:
  - Explains which compiler to use (gcc)
  - What to build (main) and how to build it (using main.o and add.o)
  - Explains how to clean up the result of a build (removing main, main.o, add.o)

- We'll talk more about makefiles tomorrow!