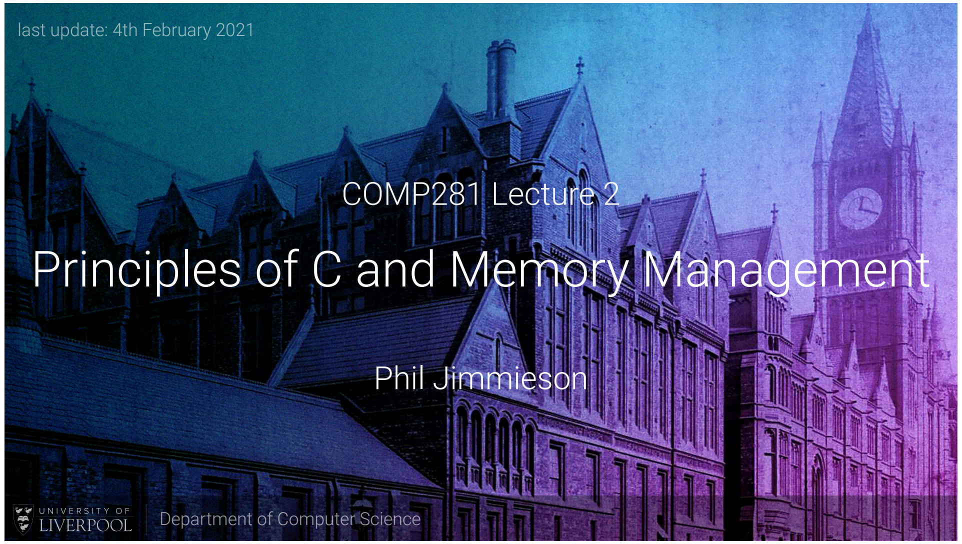


last update: 4th February 2021

COMP281 Lecture 2

Principles of C and Memory Management

Phil Jimmieson

UNIVERSITY OF LIVERPOOL

Department of Computer Science

Last Lecture

- Principles of C and Memory Management?
  what this module is about
- General module information.

Last Lecture

- Principles of C and Memory Management?
  what this module is about
- General module information.

Recap

Last Lecture

hello.c

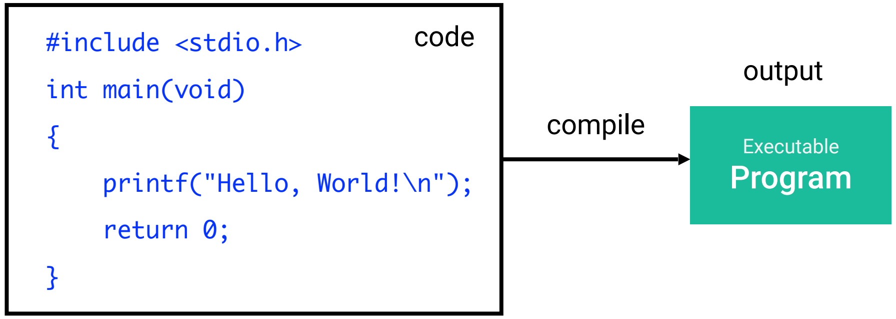
```
#include <stdio.h>
int main(void)
{
    printf("Hello, World!\n");
    return 0;
}
```

===

Compiler via the terminal

```
% gcc hello.c
% ./a.out
Hello, world!
%
```

Last Lecture



code

```
#include <stdio.h>
int main(void)
{
    printf("Hello, World!\n");
    return 0;
}
```

compile

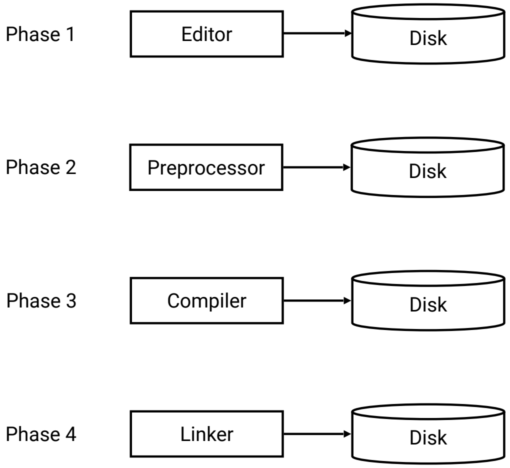
output

Executable
Program

This time:

- Compiling and Running C Programs
- C Language Basics

Compiling and Running C Programs



Phase 1 Editor → Disk

Program is created using the **Editor** and stored on Disk

Phase 2 Preprocessor → Disk

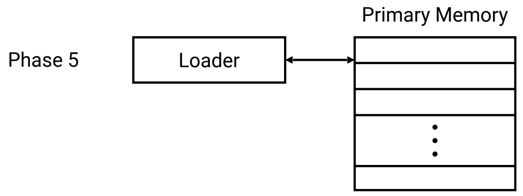
Preprocessor program processes the code

Phase 3 Compiler → Disk

Compiler creates object code and stores it on Disk

Phase 4 Linker → Disk

Linker links object code with libraries, create .out and stores on Disk

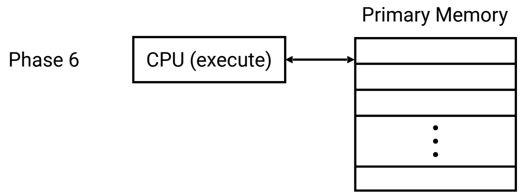


Phase 5

Loader

Primary Memory

Loader puts Program in Memory



Phase 6

CPU (execute)

Primary Memory

CPU takes each instruction and executes it, storing new data values as the program executes

Compiling C Programs

- 4 kinds of files to work with
- The Preprocessor
- The Compiler
- The Linker

Compiling C Programs

- **4 kinds of files to work with**
- The Preprocessor
- The Compiler
- The Linker

4 kinds of files to work with

1. **Source Code** files

- *.c files
- Contain function ***definitions***

4 kinds of files to work with

2. **Header** files

- *.h files
- Contain function ***declarations*** (function prototypes)
- Contain various preprocessor statements
- Allow source code files to access externally-defined functions

4 kinds of files to work with

3. **Object** files

- *.o files (or *.obj on Windows)
- The *output* of the **compiler**
- Contain function ***definitions in binary form***
- Not executable by themselves

4 kinds of files to work with

4. **Binary executables**

- No suffix on Unix OS (or *.exe on Windows)
- The *output* of the **Linker**
- Made from a few object files
- Can be directly executed

Compiling C Programs

- 4 kinds of files to work with
- **The Preprocessor**
- The Compiler
- The Linker

Compiling C Programs – The Preprocessor

Before the C compiler starts compiling a source code file, the file is processed by the preprocessor.

- It is a separate program, normally called “cpp” for “c preprocessor”.
- It **is invoked automatically** by the compiler **before compilation** proper begins.
- It **converts** source code (*.c) files, which may exist as a real file or be stored in memory for a short time before being sent to the Compiler.
- Preprocessor commands start with “#”. There are several preprocessor commands; the most important ones are:

  `#include #define`

`#include`

To access function definitions defined outside of a source code file, e.g.,

```
=====================
#include <stdio.h>

====================
```

causes the preprocessor to paste the contents of `<stdio.h>` into the source code at the location of the `#include` statement before it get compiled.

- C compilers do not allow using a function unless it has previously been **declared** or **defined** in the file.

  #include statements are thus the way to *re-use previously-written code* in C programs.

- To include **header** files, which mainly contain **function declarations** and `#define` statements, e.g.,

  `#include <stdio.h>` for using functions such as `printf`, whose declarations are located in the file `stdio.h`.

#define

Mainly to define constants, e.g.,

```
#define MAXNUM 999999
```

specifies wherever the character string MAXNUM is found in the rest of the program, 999999 should be substituted for it, e.g.,

```
int i = MAXNUM;
```

becomes

```
int i = 999999;
```

#define

Mainly to define constants, e.g.,

```
#define MAXNUM 999999
```

specifies wherever the character string MAXNUM is found in the rest of the program, 999999 should be substituted for it, e.g.,

```
int i = MAXNUM;
```

becomes

```
int i = 999999;
```

Why is it useful?

- To avoid having to explicitly write out some constant value in many different places in a source code file.
- This is important if the constant value needs to be changed later; it's much less bug-prone to change it once, in the `#define`, than to have to change it in multiple places scattered all over the source code.

Some preprocessors commands

```
#define
#include
#undef
#ifdef
#ifundef
#error
#if
#else
#elif
#endif
#pragma
```

Compiling C Programs

- 4 kinds of files to work with
- The Preprocessor
- **The Compiler**
- The Linker

Compiling C Programs – The Compiler

- After the Preprocessor has included all header files and expanded out all the `#define` and `#include` statements (and any other preprocessor commands that may be in the original file), the compiler compiles the program.
- It turns the source code into an **object code** file, which contains the binary version of the source code (not *executable* yet).

Compiling C Programs – The Compiler

- The **Compiler** may be invoked as:

```
% gcc foo.c
```

or

```
% gcc –c foo.c
```

This tells the compiler to run the preprocessor on the file `foo.c`, and then compile it into the **object** file `foo.o`. The **–c** option means to compile the source code file into an **object** file but NOT to invoke the Linker.

Compiling C Programs – The Compiler

- If the program is in one **source code** file

```
% gcc foo.c –o foo
```

This tells the Compiler to run the Preprocessor on the file `foo.c`, *compile* it and then *link* it to create an **executable** called `foo`.

The **–o** option states the name of the *output* binary **executable** file

Compiling C Programs

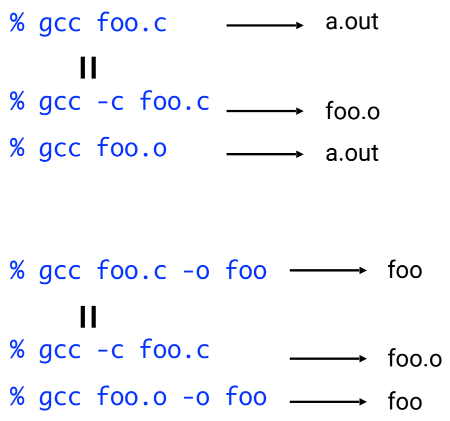
- 4 kinds of files to work with
- The Preprocessor
- The Compiler
- **The Linker**

Compiling C Programs – The Linker

- It links together **object** files (.o files) into a binary **executable**.
- It is a separate program called `ld`.
- It is invoked *automatically* when using the **Compiler**.
- The normal way of using the linker is as follows:

```
% gcc foo.o bar.o baz.o –o myprogram
```

This tells the compiler to link together 3 **object** files (`foo.o`, `bar.o` and `baz.o`) into a binary **executable** file named `myprogram`.



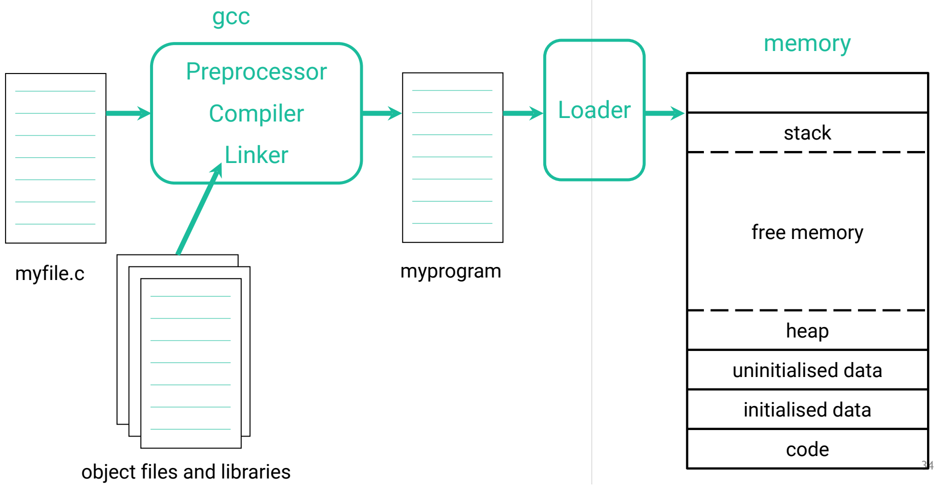
```
% gcc foo.c          ⟶  a.out
    ||
% gcc -c foo.c       ⟶  foo.o
% gcc foo.o          ⟶  a.out

% gcc foo.c -o foo   ⟶  foo
    ||
% gcc -c foo.c       ⟶  foo.o
% gcc foo.o -o foo   ⟶  foo
```

Now you have a file called `myprogram` that you can **run** and which will hopefully do something cool and/or useful.

```
% ./myprogram
```

Compiling

Running



gcc

Preprocessor

Compiler

Linker

myfile.c

object files and libraries

myprogram

Loader

memory

stack

free memory

heap

uninitialised data

initialised data

code