The background of the slide features a photograph of a large, historic stone building with multiple gables, arched windows, and a prominent clock tower on the right side. The sky is overcast.

COMP281 Lecture 2

Principles of C and Memory Management

Dr SHI Lei

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) only one main function
{
    printf("Hello, World!\n");
    return 0; return the signal
}
```

=====

**Compiler
on terminal**

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

Last Lecture

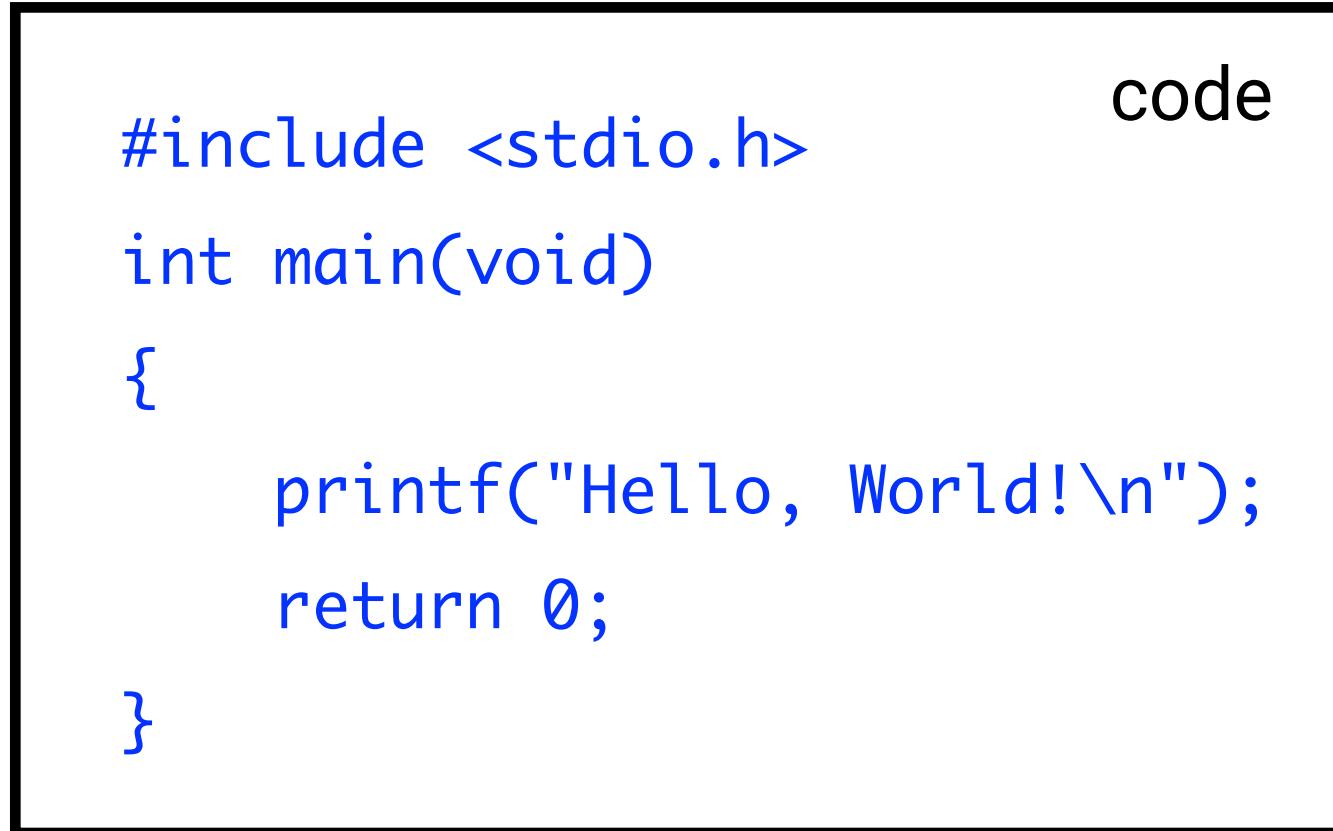
```
#include <stdio.h>
int main(void)
{
    printf("Hello, World!\n");
    return 0;
}
```

code

compile

output

Executable
Program

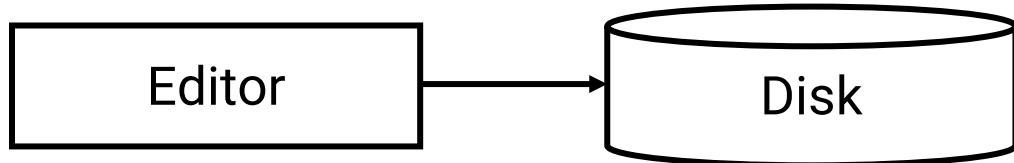


Today

- Compiling and Running C Programs
- C Language Basics

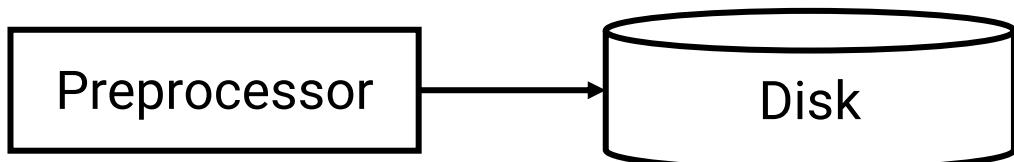
Compiling and Running C Programs

Phase 1



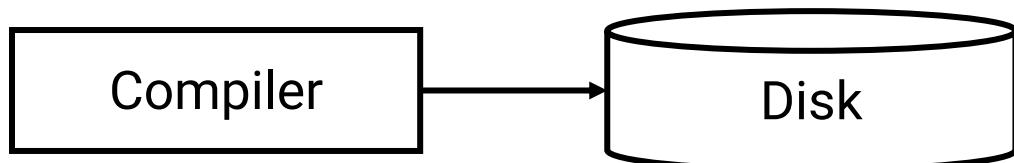
Program is created using the **Editor** and stored on Disk

Phase 2



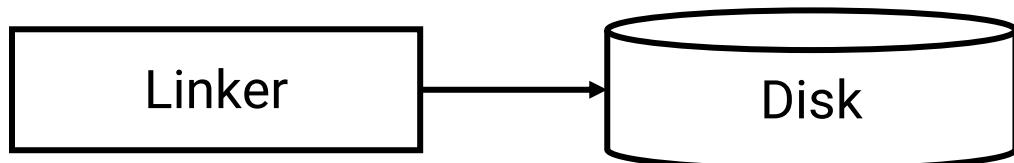
Preprocessor program processes the code

Phase 3



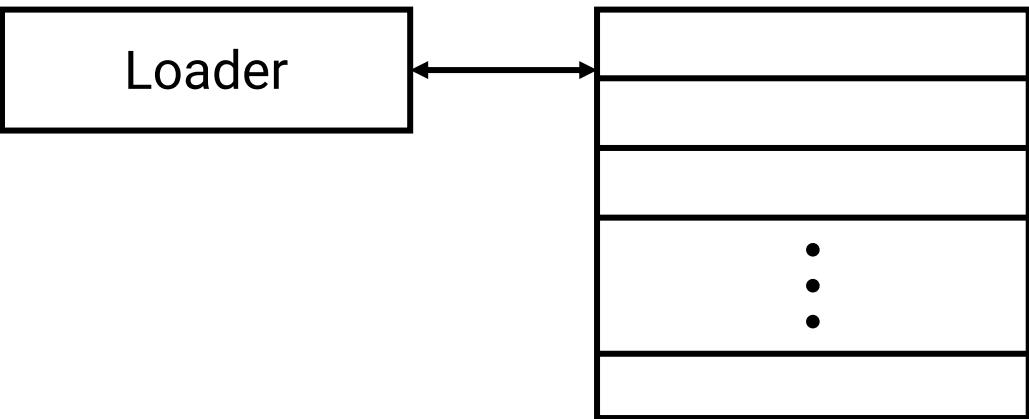
translate the code into the computer language
Compiler creates object code and stores it on Disk

Phase 4



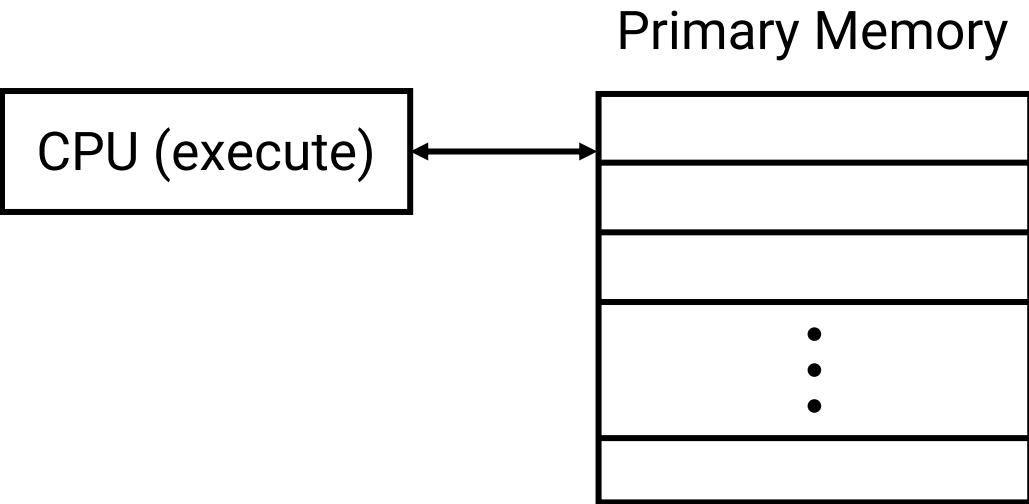
get a real code program to run
Linker links object code with libraries, create .out and stores on Disk

Phase 5



Loader puts Program in Memory

Phase 6



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 *processor*
- 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

it is translate , form human language into the computer bytes code

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

#if

#include

#else

#undef

#elif

#ifdef

#endif

#ifndef

#pragma

#error

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 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 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
```

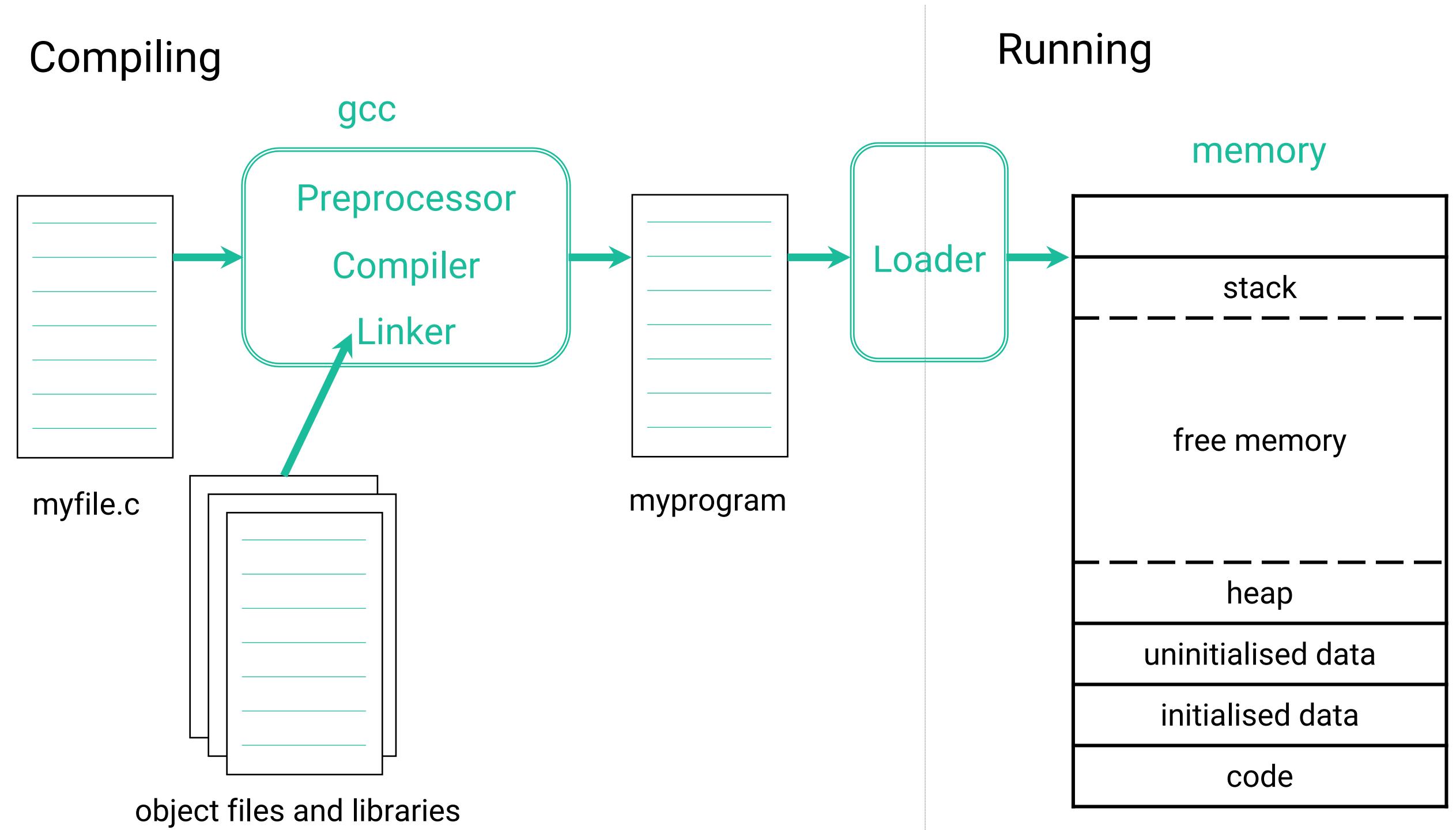
```
% gcc foo.o -o foo → foo
```

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



% ./myprogram

Compiling



C Language Basics

```
#include <stdio.h>

int main(void)

{
    printf("Hello, World!\n");

    return 0;
}
```

- The main () function
- Statements
- C Skeleton
- Identifiers
- Keywords
- Basic data variables and types
- Constants

The `main()` Function

- Identify the start of the program
- Every C program has a `main()` one and only one
- “main” is a C keyword. We **must not** use it for any other variable.

just for the entrance of program

Statements

- A specification of an action to be taken by the machine as the program executes.
- Each statement in C needs to be terminated with semicolon (`;`).

Statements

- A specification of an action to be taken by the machine as the program executes.
- Each statement in C needs to be terminated with semicolon (;).

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

Text

a segment

another segment

C Program Skeleton

In short, the basic skeleton of a C program looks like this:

```
#include <stdio.h>

int main(void)
{
    printf("Hello, World!\n");
    return 0;
}
```

C Program Skeleton

In short, the basic skeleton of a C program looks like this:

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

The diagram illustrates the basic skeleton of a C program with the following annotations:

- preprocessor directives**: Points to the line `#include <stdio.h>`.
- function main**: Points to the line `int main(void)`.
- start of segment**: Points to the opening brace `{`.
- segment (s)**: Points to the line `printf("Hello, World!\n");`.
- end of segment**: Points to the closing brace `}`.

Identifiers

Words used to represent certain program entities

(variables, function names, etc.)

E.g.,

- `int bar_baz;` Text
 - `foo_bar` is an identifier used as a program variable.
 - `Void CalculateArea (int radius)`
 - `CalculateArea` is an identifier used as a function name

Identifiers

Rules for naming identifiers:

Rule	Example
Can contain a mix of characters and numbers	W3c
Cannot start with a number	2assignments
Must start with a letter or underscore	Number1 _area
Can be of mixed cases	whoAmI
Cannot contain any arithmetic operators	Sm*il
Cannot be any other punctuation marks (separators)	!@#\$%^&*(){} main printf
Cannot contain a space	Oh yay
Identifiers are case sensitive	Happy ≠ happy

Keywords

These are reserved words in C. They may not be used as constants or variables or any other identifier names.

auto	else	long	switch
break	enum	register	typedef
case	extern	return	union
char	float	short	unsigned
const	for	signed	void
continue	goto	sizeof	volatile
default	if	static	while
do	int	struct	_Packed
double			

Basic Data Variables and Types

- All data in C has to have a specified type
- C has several types of variables, but there are a few basic types:
 - Integers
 - Unsigned integers
 - Floating point numbers
 - Chars
- Variables hold data of a particular type only
- Variables must be declared before use

Integers

have to give the data type of the variable

Whole numbers which can be both ***positive*** and ***negative***.

Defined using:

- `char`
- `int`
- `short`
- `long`
- `long long`

Unsigned integers

Whole numbers which can only be *positive*.

Defined using:

- `unsigned char`
- `unsigned int`
- `unsigned short`
- `unsigned long`
- `unsigned long long`

Floating point numbers

Real numbers (numbers with fractions)

Defined using:

- `float` short
- `double` longer version

Chars

Equivalent to ‘letters’ in English.

Examples:

- Numeric digits: 0 – 9
- Letters: a – z and A – Z
- Space (blank)
- Special characters: !@£\$%^&*()

Single character

e.g., `char my_letter = 'E';`

(the declared character must be enclosed within a single quote!)

Constants

- Entities that appear in the program code as fixed values.
- Any attempt to modify a CONSTANT will result in error.
- There are 4 types of constants:
 - Integer constants, e.g., `const int MAX_NUM = 9999;`
 - Floating point constants, e.g., `const double VAL = 1.23e4;` (1.23×10^4)
 - Character constants, e.g., `const char letter = 'l';`
 - Enumeration, e.g., `enum City { Manchester, Liverpool, Leeds };`
- **The other way to define "constants is to use" `#define`**

Constants

Example 1

```
#include <stdio.h>
void main()
{
    int i = 1;
    const int x = 2;
    i = 3;
    x = 4;
    printf("i = %d\nx = %d", i, x);
}
```

Constants

Example 1

```
#include <stdio.h>
void main()
{
    int i = 1;
    const int x = 2;
    i = 3;
    x = 4; ← This creates an error
    printf("i = %d\nx = %d", i, x);
}
```

Constants

Example 2

```
#include <stdio.h>
#define PI 3.14 ← Define a constant using #define preprocessor directive
int main()
{
    int r = 2, area ;
    printf("The radius of circle is %d.\n",r) ;
    area = PI * (r * r) ;
    printf("Area of the circle = %d.\n", area) ;
    return 0;
}
```

Summary

Today

- Compiling C Programs
 - 4 kinds of files to work with: **source code files**, **header files**, **object files**, and binary **executables**
 - The Preprocessor -> `#define, #include`
 - The Compiler -> `% gcc foo.c -o foo`
 - The Linker -> `% gcc foo.o bar.o baz.o -o myprogram`

Today

- C Language Basics
 - The `main()` function
 - C Program Skeleton -> segments
 - Identifiers -> naming variables, functions, etc.
 - Keywords -> reserved words, may not be used as identifier names
 - Basic Data Variables and Types
 - Integers, unsigned integers, floating point numbers, chars
 - Constants (fixed values) -> `const int x = 2;` `#define PI 3.14`

Next

- C Language Basics

- ...
- Basic I/O
- Operators
- Decision Making