Week 1 Lecture 2

XMUT-NWEN 241 - 2026 T2

Systems Programming

Agatha Rachmat

School of Engineering and Computer Science

Victoria University of Wellington

Content

- Systems Programming
- C Program Design
- C Compilation Process

What is Systems Programming?

Systems Programming

- Systems programming refers to the implementation of systems programs or software

- Systems program / software:
  - Programs that support the operation and use of the computer system itself
  - Maybe used to support other software and application programs
  - May contain low-level or architecture-dependent code

- Low-level or architecture-dependent code:
  - Program that directly accesses registers or memory locations
  - Program that uses instructions specific to a computer architecture

Systems Programming

writing software that controls, manages, or supports the computer system and provides a foundation for other programs to run.

For example: open a browser, the browser needs many things from the operating system:

- memory
- CPU time
- files
- network access
- keyboard/mouse input
- display/screen access

System software, such as the operating system, device drivers, and system libraries, helps provide these services.

Systems Programming

Ex: Windows

Imagine you double-click Microsoft Word.

Microsoft Word is an **application program**.

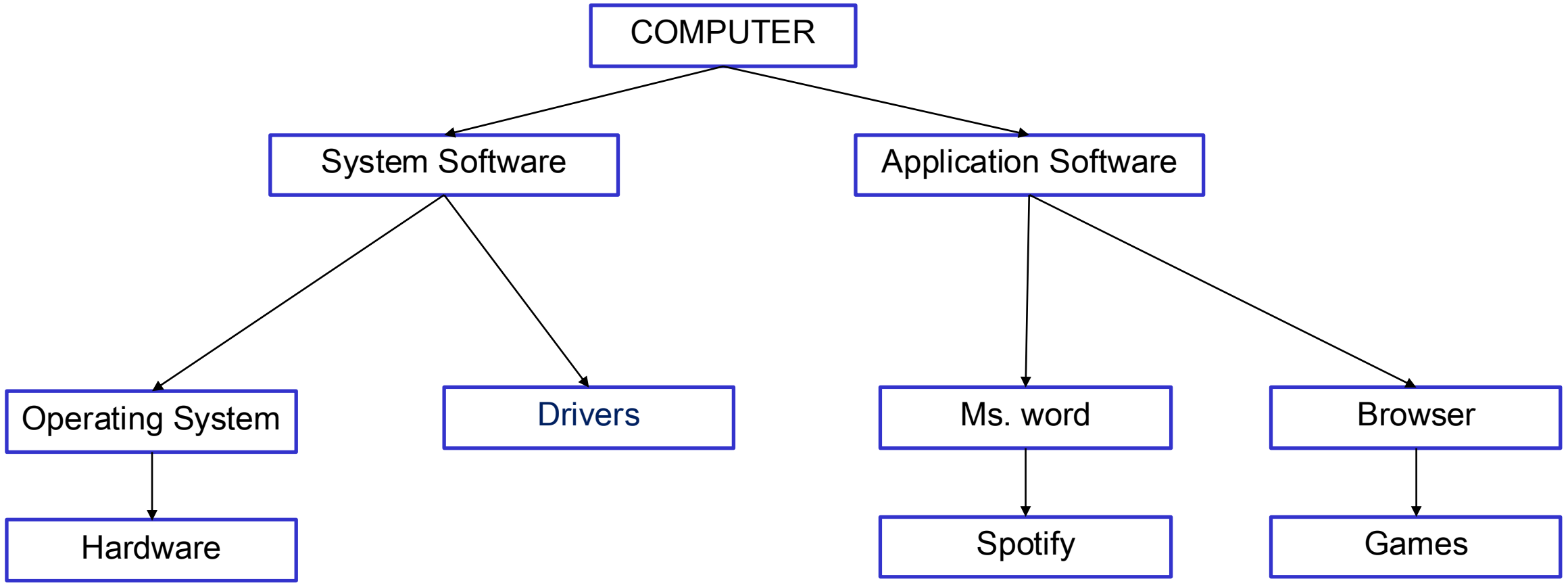
Windows contains **system software**.

Programs that implement parts of this system software are examples of system **programming**.

The process is roughly:

You
↓
Microsoft Word
↓
Operating System (Windows)
↓
CPU / Memory / Disk / Devices
↓
Hardware

Systems Programming



Primarily to **operate, manage, or support the computer**.

Primarily to help the **user perform a task**.

Example Systems Programs

- Operating system
- Embedded system software (firmware)
- Device drivers
- Text editors, compilers, assemblers
- Virtual machines
- Server programs
  - Database systems
  - Network protocols

Systems Programming

Domain	Purpose / What it Does	Core System Interaction	Typical Languages
Operating Systems	Manage core system resources (memory, CPU).	Kernel calls, Interrupts, Process IPC.	C, Assembly, Rust.
Device Drivers	Speak the language of physical hardware.	Hardware registers, Memory-Mapped I/O.	C, C++.
Compilers	Convert human code into machine instructions.	Parsing theory, Optimization algorithms.	C++, Python (for toolbuilding).
Embedded Systems	Control minimal, specialised hardware components.	Bare metal access, Direct GPIO manipulation.	C, Assembly, Rust.
Networking Stacks	Manage and route data across protocols.	Sockets API, Packet headers (IP/TCP).	C, C++, Go.

Why C?

- C supports high-level abstractions and low-level access to hardware at the same time
- High-level abstractions:
  - User-defined types (structures)
  - Data structures (stacks, queues, lists)
  - Functions
- Low-level access to hardware:
  - Possible access to registers
  - Dynamic memory allocation
  - Inclusion of assembly code

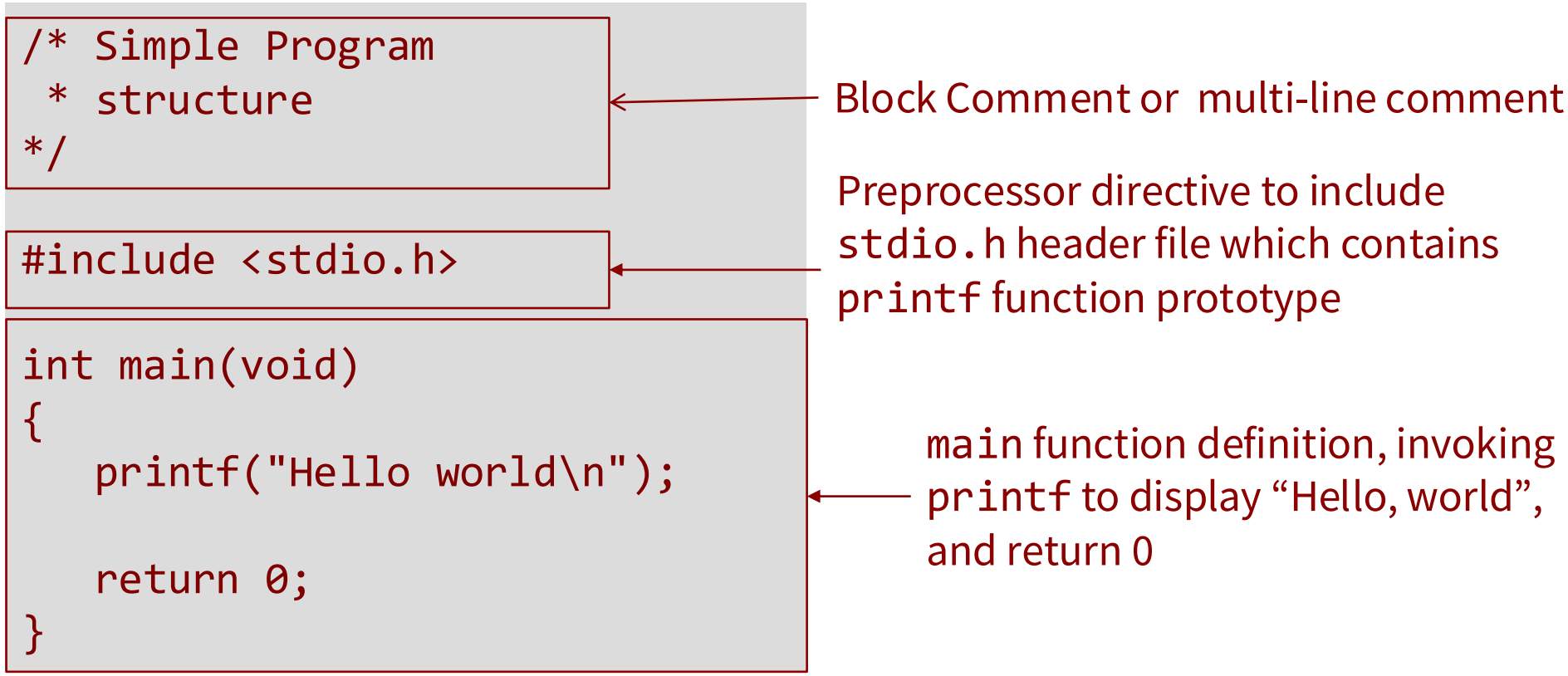
Comparing C, C++ and Java

- C is the basis for C++ and Java
  - C evolved into C++
  - C++ change into Java
  - The “class” is an extension of “struct” in C
- Similarities
  - Java uses a syntax similar to C++ (for, while, …)
  - Java supports OOP as C++ does (class, inheritance, …)
- Differences
  - Java does not support **pointers**
  - Java frees memory by **garbage collection**
  - Java is more portable by using bytecode and **a virtual machine**
  - Java does not support **operator overloading**

C Program Design

Program Structure

- A typical C program consists of
  - 1 or more header files
  - 1 or more C source files



Program Structure

```
/* Simple Program
 * structure
*/

#include <stdio.h>

int main(void)
{
   printf("Hello world\n");

   return 0;
}
```

Your C program

↓

printf()

↓

C standard library

↓

Operating system

↓

Device driver

↓

Display hardware

Comments

```
/* Comment text */
```

- Compiler ignores everything from /* to */

```
// Comment text
```

- Compiler ignores everything from // to the end of the line
- This commenting style originated from C++ and was adopted by C (C99 standard)

Main Function

- Generally, a C program has exactly one `main` function
- Execution begins with the `main` function

```
int main(void)
{
    /* Main function body */
}
```

Header File Inclusion

```
#include <filename>
```

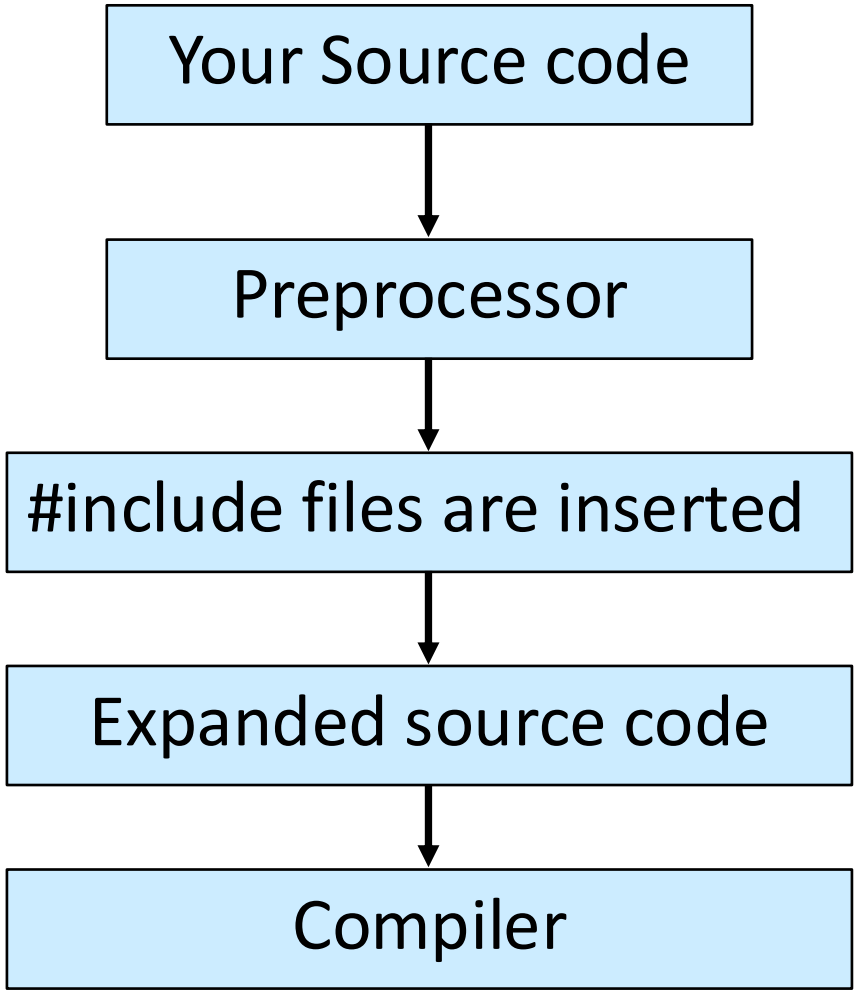
```
#include <stdio.h>
```

What does #include **do?**

Before your C program is actually compiled, a program called the **preprocessor processes** directives beginning with #

Header File Inclusion

Conceptually:



Header File Inclusion

```
#include <filename>
```

- Include file named `filename`
- Preprocessor searches for file in pre-defined locations

```
#include "filename"
```

- Include file named `filename`
- Preprocessor searches for file in current directory first, then in locations specified by programmer

Header File Inclusion

```
#include "filename"
```

```
myproject/
|
 ├── main.c
 ├── file_one.c
 └── file_two.h
```

Header Files

- A header file usually contains function prototypes, constant definitions, type definitions, etc.

- Which header file to include?
  - Include header files that contain the function prototype, constant definition, type definition, etc., used in your program

Standard C Library Header Files

Header	Main purpose	Examples
<stdio.h>	Input/output	printf, scanf, fopen
<stdlib.h>	General utilities/memory	malloc, free, exit
<string.h>	Strings/memory	strlen, strcpy, strcmp
<math.h>	Mathematics	sqrt, pow, sin
<ctype.h>	Character operations	isdigit, isalpha, toupper
<time.h>	Date/time	time, localtime
<stdint.h>	Fixed-width integers	uint32_t, int64_t
<stdbool.h>	Boolean values	bool, true, false
<limits.h>	Integer limits	INT_MAX, INT_MIN
<assert.h>	Assertions/debugging	assert

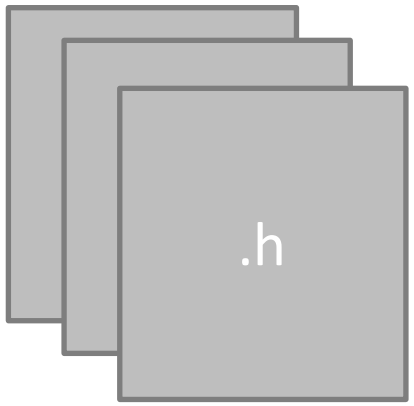
Standard C Library Header Files

C provides a standard library* which consists of the following headers:

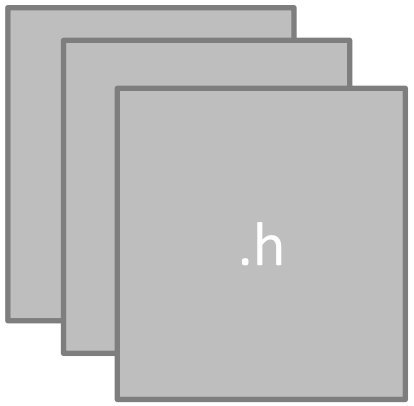
`assert.h`	`float.h`	`math.h`	`stdarg.h`	`stdlib.h`
`ctype.h`	`limits.h`	`setjmp.h`	`stddef.h`	`string.h`
`errno.h`	`locale.h`	`signal.h`	`stdio.h`	`time.h`

- To know more about the C standard library, visit
  `https://www.tutorialspoint.com/c_standard_library/index.htm`

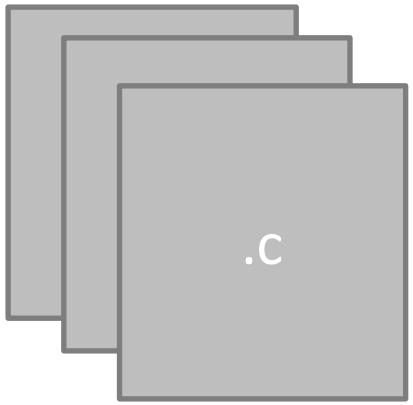
Large C Program



Header files from standard C library



Own header files



Source files

C Compilation Process

Compilation Process At A Glance

1) Preprocessing Phase

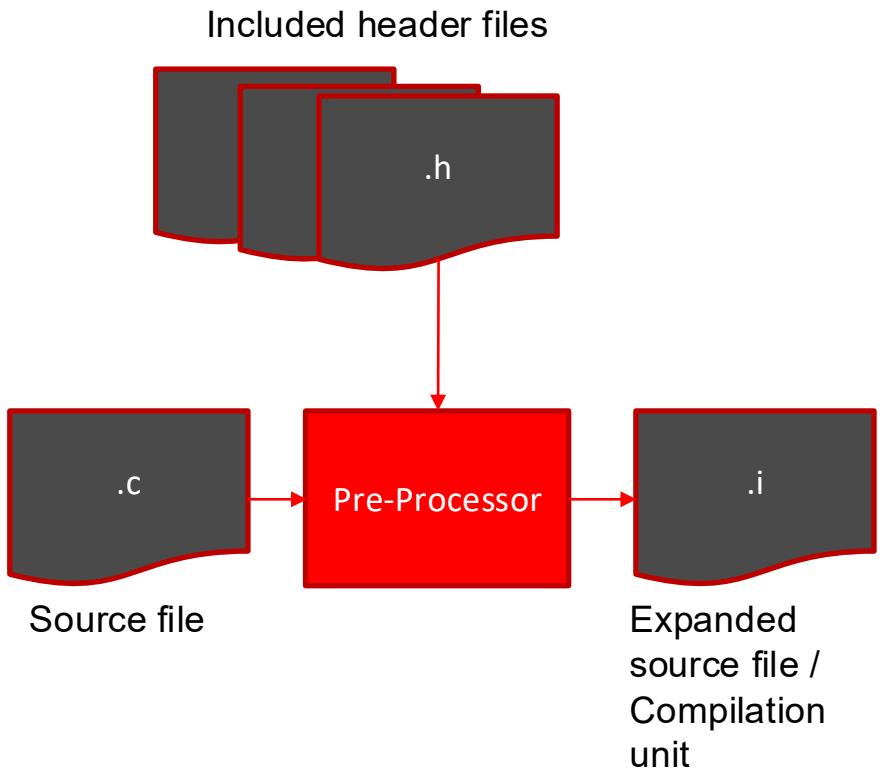
2) Compilation Phase

3) Assembly Phase

4) Linking Phase

Preprocessing Phase

- The preprocessor modifies the original C program according to directives that begin with the '#' character
  - Example: `#include <stdio.h>` command tells the preprocessor to read the contents of the system header file `stdio.h` and insert it directly into the program text.
- The result is another C program, typically with the **.i** suffix.

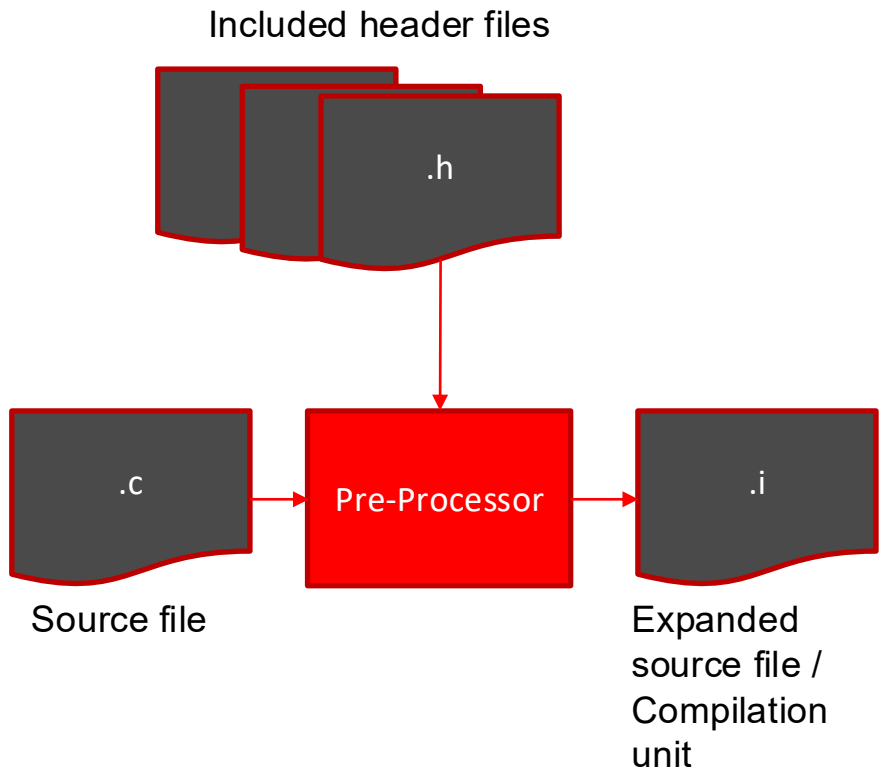


Preprocessing Phase

hello.c

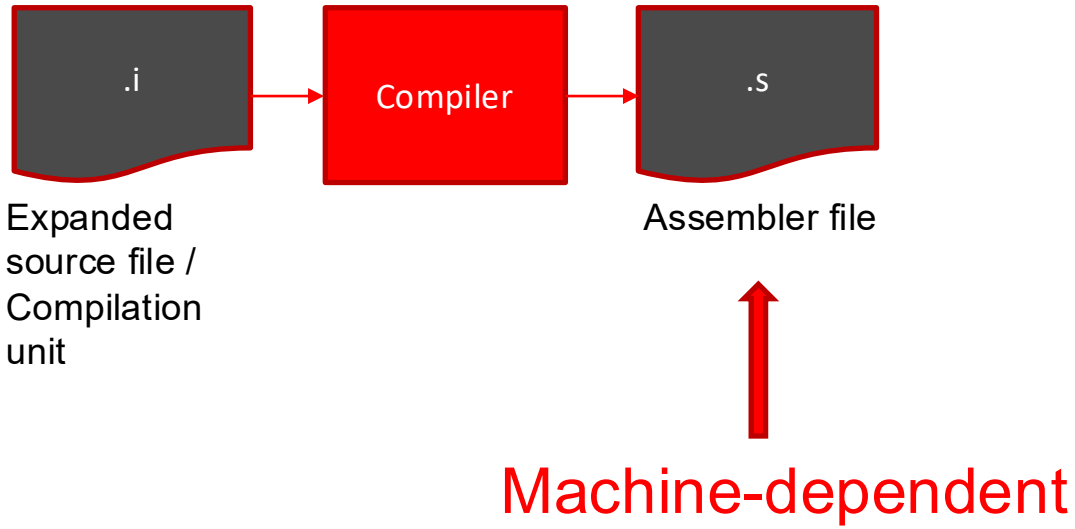
```
/* Simple Program
 * structure
*/

#include <stdio.h>

int main(void)
{
   printf("Hello world\n");

   return 0;
}
```

hello.i



Compilation Phase

- The compiler translates the text file (**.i**) into the text file (**.s**), which contains an assembly-language program.



Compilation Phase

- hello.i -> hello.s

For example, you might have C code like:

```
int add(int a, int b) {
    return a + b;
}
```

The compiler might produce assembly resembling:

```
add:
    push    ...
    mov     ...
    add     ...
    ret
```

The exact assembly *depends on the CPU architecture and compiler.*

The .s file contains **assembly language**.
So:
C code
↓
Compiler
↓
Assembly code

Compilation Phase

Why does the compiler produce assembly? Why not go directly from C to machine code?

The compiler **could conceptually produce machine code**, but assembly code provides a useful intermediate representation.

C is relatively easy for humans to write.

Assembly is much closer to what the CPU understands.

Machine code is the binary instructions the CPU executes.

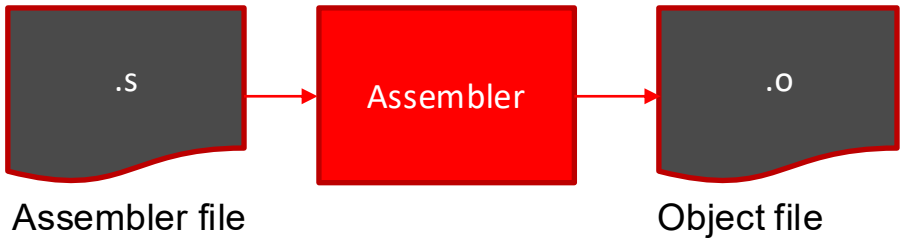
Think of it as a translation:
Human-friendly
↓
C
↓
Assembly
↓
Machine code
↓
CPU

Assembly Phase

- The assembler translates assembler file (**.s**) into machine-language instructions, packages them in a form known as a relocatable object program, and stores the result in the object file (**.o**).

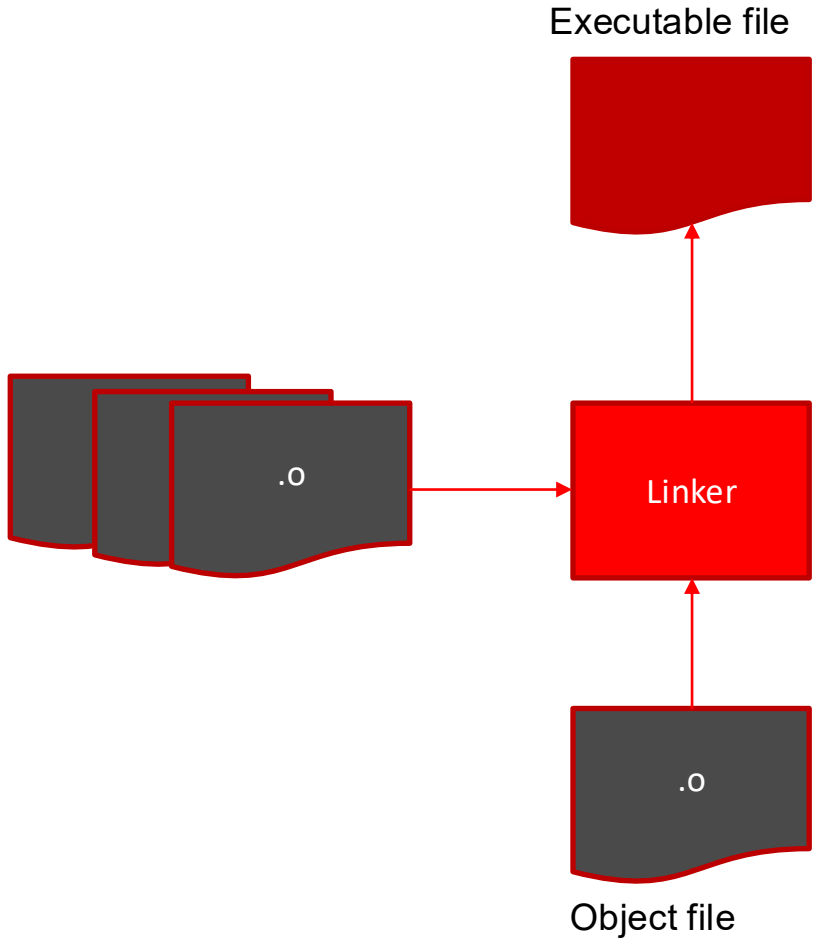
- Object files are binary - if you try to open one with a text editor, it would appear to be gibberish.

hello.s to hello.o



Linking Phase

- The linker looks for external object files needed by the program and merges these with the object file generated in the assembly phase, creating an executable object file (or simply executable) that is ready to be loaded into memory and executed by the system.



Summary

File	Contains	Produced by
.c	C source code	You
.i	Preprocessed C source code	Preprocessor
.s	Assembly-language source code	Compiler
.o	Object/machine code	Assembler
executable	Complete runnable program	Linker

In Practice- GCC compiler

- All the phases can be done in one step using the GNU C Compiler (GCC)
- Usually, we use gcc to do all the phases and directly generate the binary executable
- We can also ask gcc to do certain phases

`hello.c`

```
#include <stdio.h>
int main(void)
{
   printf("Hello world\n");

   return 0;
}
```

```
gcc hello.c
```

Generates executable file `a.out`

```
gcc hello.c -o hello
```

Generates executable file `hello`

gcc Options

Phase	Gcc **Option**	Result	Output File
Preprocessing	-E	Compilation unit	.i .ii
Compilation	-S	Assembler file	.s
Assembly	-c	Object file	.o .obj
Linking		Executable	Binary excutable (.exe in Windows)

What You Need to Program in C

- Text editor to type in code
  - Any text editor will do (even notepad)
  - Suggested editors: Sublime Text, Atom, Kate (Linux only)
- C toolchain (pre-processor, compiler, assembler, linker, debugger)
- Terminal to run compilation commands and execute program

OR

- IDE