Introduction to Programming (Adv)

School of Computer Science, University of Sydney

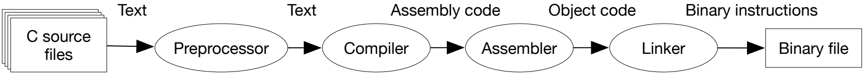
Copyright Warning

Compilation pipeline

From source code to program

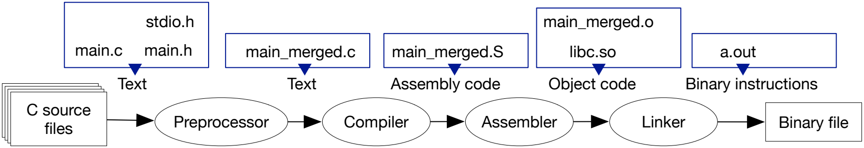
Compilation pipeline

Beginning to end



Compilation pipeline (cont.)

Example with main.c and main.h



Compiler

Check syntax of all statements

Generate, or reorganise code

Decide and also reorganise memory layout

Produce architecture dependent instructions

Produce final program

Can be only one, or all of these steps

The preprocessor

A text manipulation program

Compiling source code into a program binary

A compiler will take human readable source code and produce machine executable code

```c
int main() {
    fputs("Hello world!\n", stdout);
    return 0;
}
```

```
gcc -o program source.c
```

Compiling source code into a program binary (cont.)

Syntax error.

```c
int main() {
    fputs("Hello world!\n", stdout);
    return 0;
}
```

What is `stdout`?

What is `fputs`?

When you use anything outside the standard programming language. The compiler needs to include that code in the final binary file. How will the computer know how to execute `fputs`?

Preprocessor

`stdout` is a pointer to a file structure type. This is available from the C runtime library

```
#define stdout __stdoutp
extern FILE *__stdoutp;
```

`fputs` is a function. This is available from the C runtime library

```
int fputs(const char *, FILE *);
```

During the compilation process the unidentified symbols need to be declared before use.

Preprocessor (cont.)

```c
// from outside this module
extern FILE *stdout;

// from outside this module
extern int fputs(const char *, FILE *);

int main() {
    fputs("Hello world!\n", stdout);
    return 0;
}
```

Oh no! what is FILE? we need to declare that too!
`typedef struct _IO_FILE FILE;`

Oh no! what is `struct _IO_FILE` ...

Looking at `struct _IO_FILE`, there are many more definitions required. `_IO_file_flags _IO_read_ptr _IO_read_end` ...

Preprocessor (cont.)

To write the simple program, the programmer needs to effectively redefine all the declarations (not implementation) of the functions used from the C library.

The preprocessor serves to alleviate this work for the programmer by performing text processing.

`#include` is a directive to the *preprocessor* asking: please copy all the text from this file and put it here.

Preprocessor (cont.)

```
// bring in text for declarations "stdout" and "fgets"
#include <stdio.h>

// bring in text for "int"
#include "int.txt"
main()
{
   fputs("Hello world!\n", stdout);
// bring in text for "return 0"
#include "return.txt"
}
```

file contents of int.txt:

```
int
```

file contents of return.txt:

```
return 0;
```

Preprocessor (cont.)

How many lines of code are produced by the preprocessor for this program? `gcc -E source.c | wc`

H/W	System	lines of code
Macbook 2015	Mac OS X Catalina 10.15.3	549
Macbook 2010	Linux 4.19.0-8-amd64 x86_64	734
SBC Raspberry Pi 3	Linux 4.19.75-v7+ armv7l	737
Cloud VPS	Linux 4.15.0-74-generic x86_64	813
Mini PC	Linux 4.15.0-76-generic x86_64	813
SBC ODROID N2	Linux 4.9.210-66 aarch64	820
Dell notebook	Linux 4.9.0-12-amd64 x86_64	869
SBC ODROID C2	Linux 3.16.78+aarch64	876
Dell notebook	Windows 10 Pro CYGWIN_NT-10.0	1486

Standardisation is most important for portability. Emphasise the difference in output. The amount of code is not important here. [1]

Preprocessor defines are also recursive!

[1] less code does not mean better!

Preprocessor define

Another text replacement directive is *define*

Equivalent to find and replace

Example 1:

```
#define NUMPEOPLE 400
int ages[NUMPEOPLE];
```

Example 2:

```
#define ERRORMSG "Oh no something went wrong!"

if (x < 0) {
   fprintf(stderr, "%s\n", ERRORMSG);
}
```

Preprocessor define (cont.)

Example 3:

```c
#define CHECK_STATUS { if (x < 0) { fprintf(stderr, "%s\n", "Error"); } }

x = get_positive_number();
CHECK_STATUS(30)
...
x = smooth_number(x);
CHECK_STATUS(63)

while (x > 0) {
    ...
    x -= timestep;
}
CHECK_STATUS(54)

printf("Height in cm after %d months is: %d\n", months, x);
```

Preprocessor define (cont.)

define can also be defined as a *macro*. It accept parameters, similar to a function. The parameter is also treated as a piece of text

```c
#define CHECK_STATUS(errorcode) { if (x < 0) { fprintf(stderr, "%s:%d\n", "Error", errorcode); } }

x = get_positive_number();
CHECK_STATUS(30)
...
x = smooth_number(x);
CHECK_STATUS(63)

while (x > 0) {
    ...
    x -= timestep;
}
CHECK_STATUS(54)

printf("Height in cm after %d months is: %d\n", months, x);
```

Macros are not functions!

Modules: Header files

Programs consist of *modules*

A module is a file. It is source code for a particular purpose within the program, a subsystem, and it includes all related declarations and definitions to function as the subsystem.

Think of a single purpose that has limited dependencies.

- Steering function in a vehicle is its own module within the program Car
- Discount code calculation for online e-commerce system
- Weight sensor and alarm for elevators

Modules: Header files (cont.)

A module is a file, i.e hello.c
Modules consist of:

- Function declarations
- Function definitions
- Global variables

Modules are translated to object files

Object files are linked by linker with other object files and standard libraries

Modules: Header files (cont.)

A module can refer to global variables and functions of other modules - use the `extern` qualifier for global variables

Symbols can only be defined in one module

Data structures definitions and declarations, macro definitions and external function declarations are found in modules

These are commonly found in header files

Modules: Header files (cont.)

Header files give a public facing interface to the modules functionality

Using the header file, a programmer can include all the necessary definitions and extern functions to reference in another module.

e.g. simulation.c uses math functions in mypoly.c

```c
/* mypoly.h */
// declaration of related data structure here
struct poly2 {
   float a, b, c;
};
// declaration of function here
extern float poly2_evaluate(struct poly2 *, float);

// no actual math logic/solving code in this file!
```

Modules: Header files (cont.)

```c
/* mypoly.c */
// needs to include the declaration of related data structure
#include "mypoly.h"

// definition of function here
float poly2_evaluate(struct poly2 *p, float x) {
    ...
    // code here
    return ...
}
```

Modules: Header files (cont.)

```c
/* simulation.c */
// needs declaration of related data structure
// needs declaration of function (not definition)
#include "mypoly.h"

int main() {
    ...
    float answer = poly2_evaluate(p, x);
    ...
    return 0;
}
```

After the preprocessor has duplicated the text for all the necessary declarations and definitions, the linker can not build the final program binary among modules.

Preprocessor guards

C language looks at the name of a symbol.

Cannot have the same named symbol defined twice.

```c
int main() {
    int x;
    ...
    int x;
    ...
    return 0;
}
```

Only one local variable and also only one global variable.

If two modules declare the same symbol name, the linker cannot resolve which definition should be referred to. Program cannot be compiled.

Preprocessor guards (cont.)

```c
/* simulation.c */
// needs declaration of related data structure
// needs declaration of function (not definition)
#include "mypoly.h"

// oops twice!
#include "mypoly.h"

int main() {
    ...
    float answer = poly2_evaluate(p, x);
    ...
    return 0;
}
```

It is possible that modules have a circular dependency with other modules. A needs B, B needs C, C needs A etc.

However, it really only needs to be defined once *per module*

Preprocessor guards (cont.)

Conditional text processing.

Preprocessor can look for previously defined symbols and continue processing within that flow control using if/else/endif syntax

```c
/* mypoly.h */
#ifndef MYMATH_H
#define MYMATH_H

// declaration of related data structure here
struct poly2 {
   float a, b, c;
};
// declaration of function here
extern float poly2_evaluate(struct poly2 *, float);

#endif
```

Preprocessor guards (cont.)

```c
/* mypoly.c */
// needs to include the declaration of related data structure
#include "mypoly.h"
#include "mypoly.h"
#include "mypoly.h"

// definition of function here
float poly2_evaluate(struct poly2 *p, float x) {
    ...
    // code here
    return ...
}
```

Compiles!

Conditional inclusion, use of header guards, only allow the first include to be processed. The other two are skipped.

Summary

The compilation pipeline requires many steps

Preprocessor is built into the C language

Preprocessor serves as a text program for the programmers benefit. Duplicating the definitions in each module as needed.

Preprocessor used incorrectly will introduce side-effects (it is not a compiler!)

Compiler will perform many steps to generate assembly code of the C language. Errors here are mainly *intra-module*.

Linker will finally put all modules together in a program. Remember that it is also bringing in the C functions from the standard library. Errors here are mainly *inter-module*.