Creative Software Programming

5 – Compilation and Linkage, CMD Args

Today's Topics

- Compilation and Linkage
  - C/C++ Build Stages
  - Header and Source Files
    - Function / Class Declaration and Definition
    - Include Guards
    - Inline Function
  - Preprocessor
- Command-line Arguments
- Building a Multi-file Project
  - Introduction to CMake

Compilation and Linkage

C/C++ Build Stages

example.c

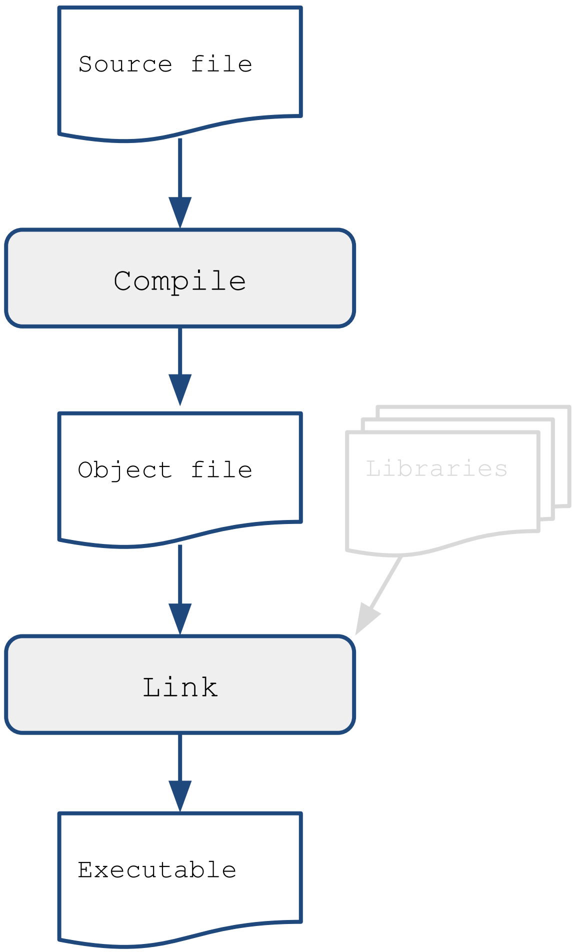
```
int FuncInt(int a, int b) {
  ...
}

int FuncDouble(double a, double b, double c) {
  ...
}

int main() { ... }
```

example.o

```
_FuncInt: ........
_FuncDouble: ........
_main: ........
```

example (example.exe)

```
........
```

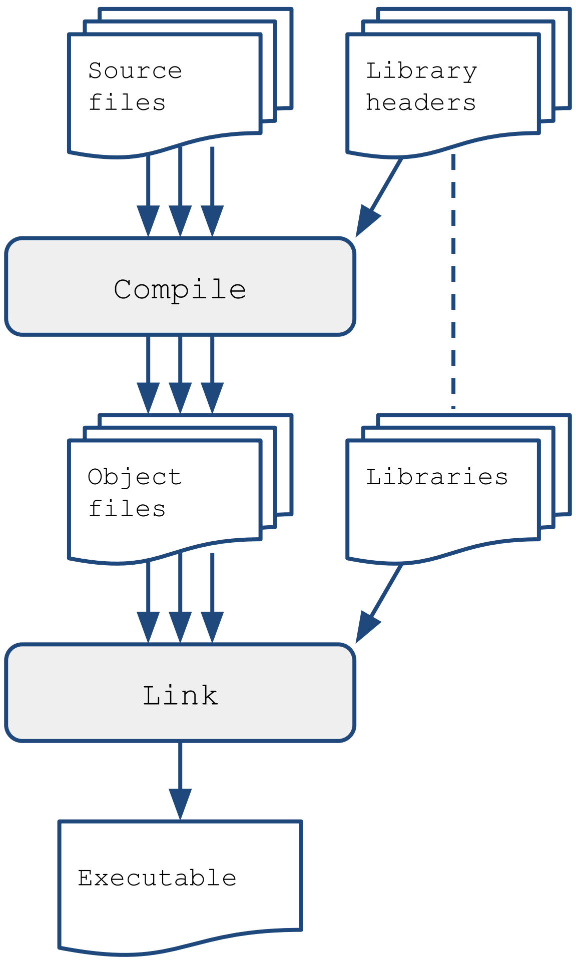


C/C++ Build Stages

example.c

```
#include <math.h>

int FuncInt(int a, int b) {
  ...
}

int FuncDouble(double a, double b, double c) {
  double d = sin(a) * b + cos(a) * c;
  ...
}

int main() { ... }
```

How can the compiler know the type of the function `sin` and `cos`?



C/C++ Compilation

- Compilers only need to know the declarations (types) of the functions or external variables.
- The preprocessor just replaces `#include` statements with their file content.

example.c

```
#include <math.h>

int FuncInt(int a, int b) {
  ...
}

int FuncDouble(double a, double b, double c) {
  double d = sin(a) * b + cos(a) * c;
  ...
}

int main() { ... }
```

math.h

```
...
double sin(double x);
double cos(double x);
...
```

example.o

```
_FuncInt: ........
_FuncDouble: ........
_main: ........
_sin: ??
_cos: ??
```

C/C++ Build Stages

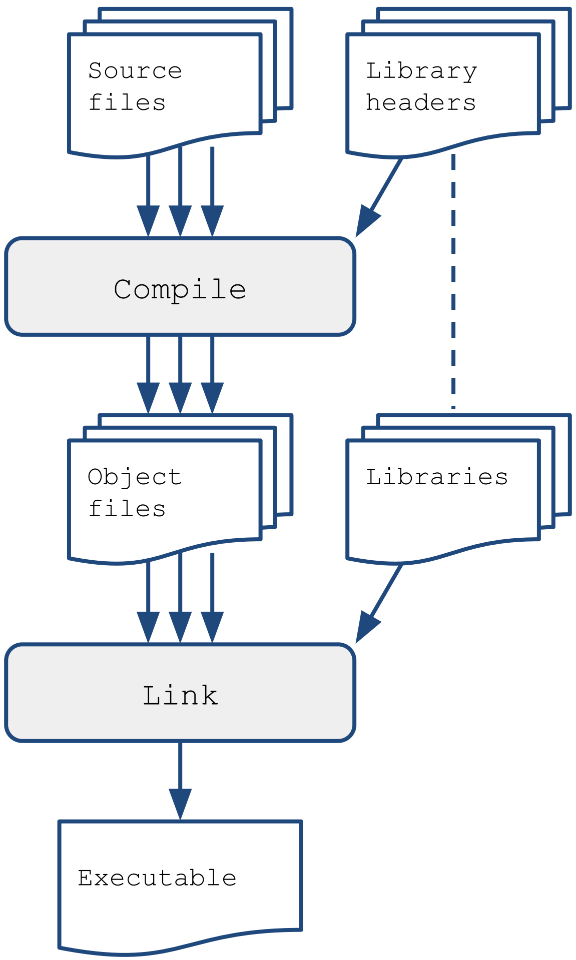
example.c

```
#include <math.h>

int FuncInt(int a, int b) {
  ...
}

int FuncDouble(double a, double b, double c) {
  double d = sin(a) * b + cos(a) * c;
  ...
}

int main() { ... }
```

example.o

```
_FuncInt: ........
_FuncDouble: ........
_main: ........
_sin: ??
_cos: ??
```

Where can we find the definition of the function `sin` and `cos`?



C/C++ Linking

- A library is just a collection of object files.
  - sin() and cos() are defined in C standard library (libc)
- Linker tries to find all unknown symbols in the object files and the libraries.

example.o

```
_FuncInt: ........
_FuncDouble: ........
_main: ........
_sin: ??
_cos: ??
```

libc.a

```
...
_sin: ........
_cos: ........
...
```

example (example.exe)

```
_FuncInt: ........
_FuncDouble: ........
_main: ........
_sin: ........
_cos: ........
```

Header and Source Files

In C++, a header file's extension is **'.h'** or '.hpp', and a source file's is **'.cpp'** or '.cc'.

C/C++ header files contain

- function and external variable declarations.
- struct and class (type) definition.
- enumeration definitions.
- macro definitions.
- inline function definitions (C++).
- ...

Headers show the interface of the entities in the source files.

Header & Source Files for Functions

- ***Function declaration*** only specifies the function name, parameter profile, and the return type → in a **header file**

- ***Function definition*** provides the actual implementation of the function body → in a **source file**

```
// myfunc.h - header file
int FuncInt(int a, int b);
double MyFunc(const int* array, int n, const char* command);
```

```
// myfunc.cpp - source file
#include <math.h>
#include "myfunc.h"

int FuncInt(int a, int b) {
  return a * 10 + b * b;
}
double Norm(const double* array, int n) {
  double sqsum = 0;
  for (int i = 0; i < n; ++i) sqsum += array[i] * array[i];
  return sqrt(sqsum);
}
```

Header & Source Files for Classes

- ***Class definition*** which contains member variables and member functions declarations → in a **header file**
- **Actual implementations of the class member functions** → in a **source file**
- Separating a class code into header & source files is important!
- If you do not understand, skip it. Classes will be covered in the next class.

```
// rectangle.h - header file
class Rectangle {
 private:
  int width, height;
 public:
  void SetValues(int x, int y);
};
```

```
// rectangle.cpp - source file
#include "rectangle.h"

void Rectangle::SetValues(int x, int y)
{
   width = x;
   height = y;
}
```

C/C++ Preprocessor

- When compilation begins, the preprocessor replaces the # directives in the source.

```
#include <math.h>
#include <iostream>
#include "my_header.h"

#pragma once

#define PI 3.141592
#define PI_2 (PI/2)

#define MAX(a, b) ((a) > (b) ? (a) : (b))

int main() {
  const double angle = PI / 3;
  int n, min_iter = 10;
  std::cin >> n;
  const int num_iter = MAX(n, min_iter);
  // What happens if we use MAX(++n, min_iter);
  for (int i = 0; i < n; ++i) {
    ...
  }
  return 0;
}
```

Include Guard: Will this code compile?

```c
// point.h
typedef struct {
  double x;
  double y;
} Point;
```

```c
// pointfunc.h
#include "point.h"
double CalcDist(Point p1, Point p2);
```

```c
// main.c
#include <stdio.h>
#include "point.h"
#include "pointfunc.h"

int main() {
  Point p1 = { 0, 0 };
  Point p2 = { 1, 1 };

  // print distance btwn two points
  printf("distance: %f\n", CalcDist(p1, p2));
  return 0;
}
```

```c
// pointfunc.c
#include <math.h>
#include "pointfunc.h"

double CalcDist(Point p1, Point p2) {
  double xdiff = p2.x - p1.x;
  double ydiff = p2.y - p1.y;
  return sqrt(xdiff * xdiff + ydiff * ydiff);
}
```

No, because of double inclusion of point.h

```c
// main.c
#include <stdio.h>
#include "point.h"
#include "pointfunc.h"
…
```

```c
// pointfunc.h
#include "point.h"
…
```

```c
// point.h
typedef struct {
  double x;
  double y;
} Point;
```

- As a result, the definition of Point appears twice in main.c. → Generates a compile error
- Deleting `#include "point.h"` from main.c solves the problem, but
- The more files, the more complicated include dependencies, so it's not easy to check all the inclusions.
- We have a better way to handle this!

Include Guard: #pragma once

- Add **#pragma once** at the top of header files
  - Preprocessor directive to instruct that the file to be included only once

- Although it is not an official C / C++ standard, it is a de facto standard that is supported by most compilers.

Include Guard: #pragma once

```c
// point.h
#pragma once

typedef struct {
  double x;
  double y;
} Point;
```

```c
// pointfunc.h
#pragma once

#include "point.h"
double CalcDist(Point p1, Point p2);
```

```c
// main.c
#include <stdio.h>
#include "point.h"
#include "pointfunc.h"

int main() {
  Point p1 = { 0, 0 };
  Point p2 = { 1, 1 };

  // print distance btwn two points
  printf("distance: %f\n", CalcDist(p1, p2));

  return 0;
}
```

```c
// pointfunc.c
#include <math.h>
#include "pointfunc.h"

double CalcDist(Point p1, Point p2) {
  double xdiff = p2.x - p1.x;
  double ydiff = p2.y - p1.y;
  return sqrt(xdiff * xdiff + ydiff * ydiff);
}
```

Another Include Guard: #ifndef

```
// point.h
#ifndef __POINT_H__
#define __POINT_H__

typedef struct {
  double x;
  double y;
} Point;

#endif
```

- If the name __POINT_H__ is not already defined, define __POINT_H__ and include the later part in the compilation.
- If __POINT_H__ is defined, the entire file is not included in the compilation.
- When point.h is included a second time, __POINT_H__ is already defined. Therefore, entire point.h is not included in the compilation.
- Still used a lot.

Quiz #1

- Will the following code compile? Answer yes or no.

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

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

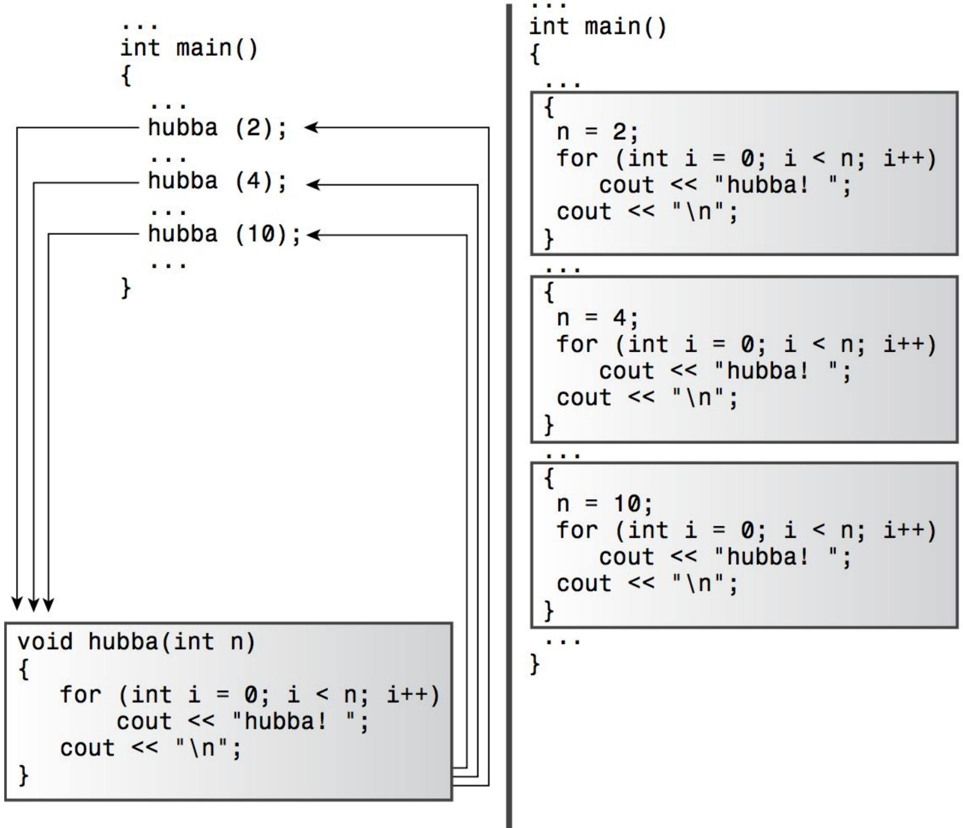
Inline Function

- Function definitions should not be in header files, except inline functions.
- Inline expansion : an inline function works as if the function call is replaced with the function body.
- Use with care : often executes faster but bloats the code.

```
#include <iostream>

#define MAX(a, b) ((a) > (b) ? (a) : (b))

inline int max(int a, int b) {
  return a > b ? a : b;
}

int main() {
  const int size = 5;
  int array[size] = { 2 3 1 5 3 };
  for (int i = 1; i < size; ++i)
    std::cout << max(array[i - 1], array[i]) << std::endl;
  return 0;
}
```

Inline Function

- The difference between normal functions and inline functions is how the compiler incorporates them into a program.



A regular function transfers program execution to a separate function.

An inline function replaces a function call with inline code.

Inline Function in Classes

- Member functions defined in a class definition (in a header file) are inline functions.

- Again if you do not understand, skip it. Classes will be covered in the next class.

```
// rectangle.h - header file

class Rectangle {
 private:
  int width, height;

 public:
  void setValues(int x, int y) {
    width = x;
    height = y;
  }
};
```

Command-line Arguments

Command-line Arguments

- C/C++ main function may take additional input parameters.

```
int main();          // OR int main(void);
int main(int argc, char **argv);
int main(int argc, char *argv[]);
int main(int argc, char **argv, char **env); // UNIX
```

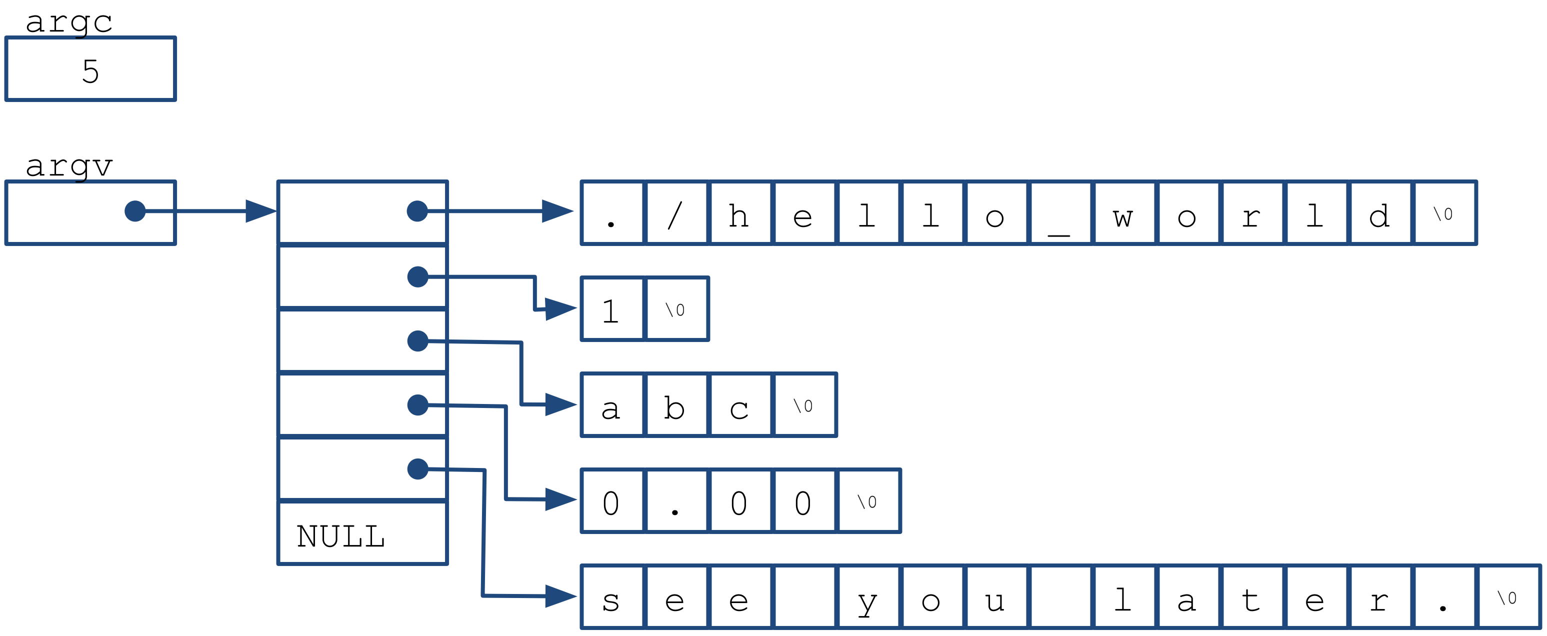
- When the program is executed, the *command-line arguments* are

```
$ ./hello_world 1 abc 0.00 "see you later."

-> argc: 5
   argv[0]: "./hello_world"   argv[3] = "0.00"
   argv[1]: "1"          argv[4] = "see you later."
   argv[2]: "abc"        argv[5] = NULL
```

Command-line Arguments

```
int main(int argc, char **argv);
```

```
$ ./hello_world 1 abc 0.00 "see you later."

-> argc: 5
   argv[0]: "./hello_world"   argv[3] = "0.00"
   argv[1]: "1"          argv[4] = "see you later."
   argv[2]: "abc"        argv[5] = NULL
```



(FYI) Double Pointer (Pointer to Pointer)

- A string array: char* strArr[] = {"aaa", "bbb", "ccc"};

- Recall: Passing an Array to a Function:
  - Pass the **start address** of the array as a pointer parameter

Example 1: A function to print an int array:

```
void printArray(int* arr, int len)
```

Example 2: A function to print an char* array:

```
void printArray(char** strArr, int len)
```

Command-line Arguments

- A simple program to print all command-line arguments.

```
#include <stdio.h>

int main(int argc, const char **argv) {
  for (int i = 0; i < argc; ++i) printf("%s\n", argv[i]);
  return 0;
}
```

- You may need string-to-number conversion.

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

int main(int argc, const char **argv) {
  for (int i = 1; i < argc; ++i) printf("%d\n", atoi(argv[i]));
  return 0;
}
```

Command-line Arguments

- The return value of the main function is the program's exit status.
  - EXIT_SUCCESS (typically 0) or EXIT_FAILURE.

- Where is this return value used?

```
$ command_a ; command_b     # Execute command_a then command_b.

$ command_a && command_b    # Execute command_a AND IF IT IS SUCCESSFUL
                            # execute command_b.

$ command_a || command_b    # Execute command_a AND IF IT FAILS
                            # execute command_b.
```

Quiz #2

```
#include <iostream>
#include <string>
using namespace std;

int main(int argc, const char **argv) {
  cout << argv[3] << endl;
}
```

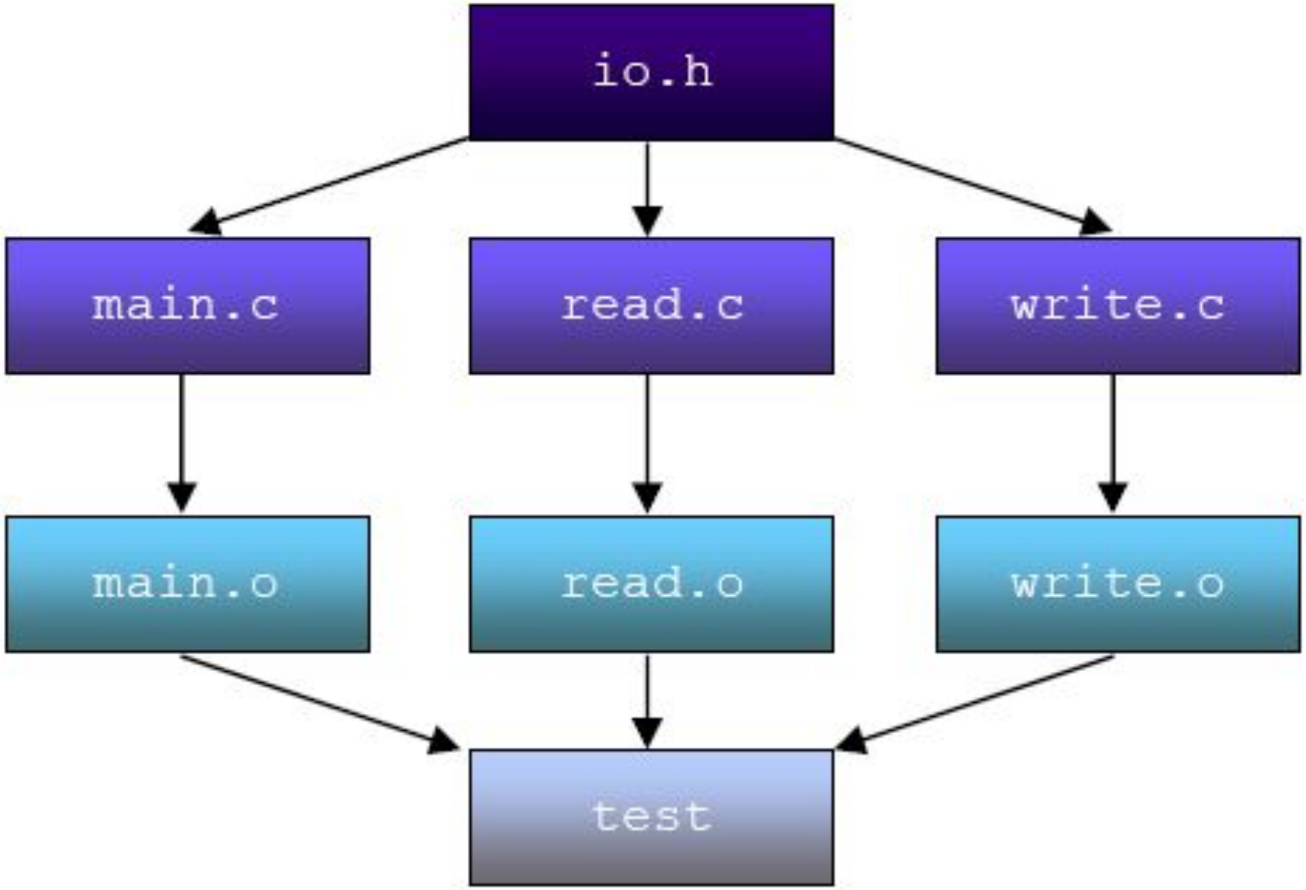
```
$ ./test aa bb cc dd
```

- What is the expected output of this program when you run it as shown in the lower left? (the executable name is test)
  - 1) test
  - 2) aa
  - 3) bb
  - 4) cc
  - 5) dd

Building a Multi-file Project

Building a Multi-file Project

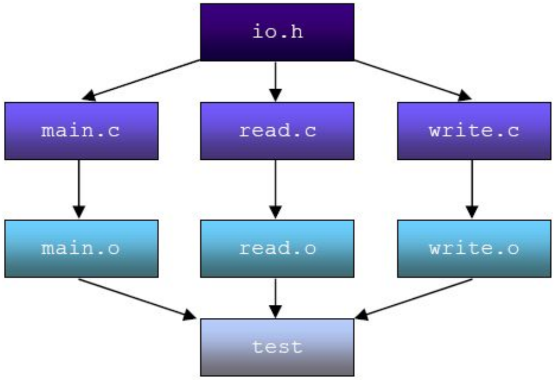
- How to build this project effectively?



1) Using g++ directly

(Shell)

```
g++ -c test read.c write.c main.c  # compile and link

# or
g++ -c read.c write.c main.c       # compile
g++ -o test read.o write.o main.o  # link
```



- Typing these lines every time is cumbersome!

- How about put these commands into a shell script?
  → Cannot use dependency information
  - It means you need to recompile main.c and write.c even if you only modify read.c

- Using dependency information is essential for building large projects
  - Because it takes too long to compile and link all files every time

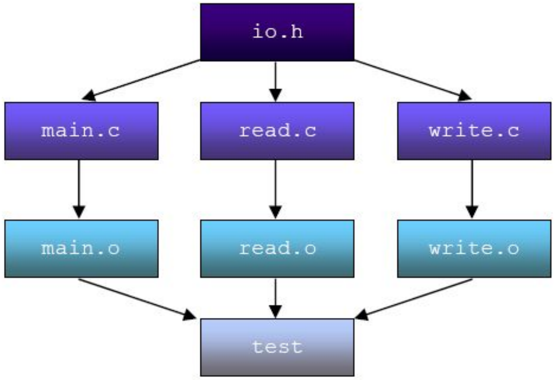
2) Makefile

- A Makefile contains dependency information

Makefile

```
test : read.o write.o main.o
     gcc -o test read.o write.o main.o

main.o : io.h main.c
     gcc -c main.c

read.o : io.h read.c
     gcc -c read.c

write.o: io.h write.c
     gcc -c write.c
```

Dependency information



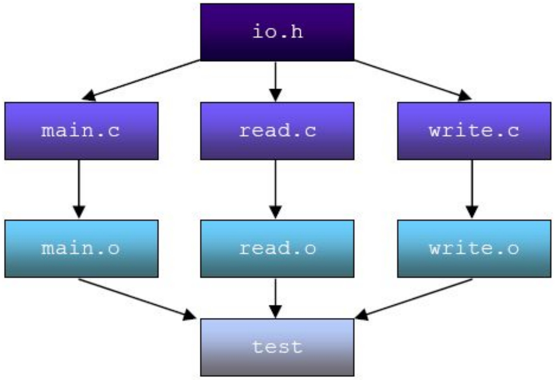
2) Makefile

- More sophisticated one

Makefile

```
CC=g++
SRCS=main.c read.c write.c
OBJS=$(SRCS:%.c=%.o)
TARGET=test

.SUFFIXES : .c .o

$(TARGET): $(OBJS)
	$(CC) -o $(TARGET) $(OBJS)

main.o: io.h main.c
read.o: io.h read.c
write.o: io.h write.c
```

Dependency information



Quiz #3

- After running make once using the Makefile below, you modified the read.c file. What are newly created files when you run make again?

```
test : read.o write.o main.o
   gcc -o test read.o write.o main.o

main.o : io.h main.c
   gcc -c main.c

read.o : io.h read.c
   gcc -c read.c

write.o: io.h write.c
   gcc -c write.c
```

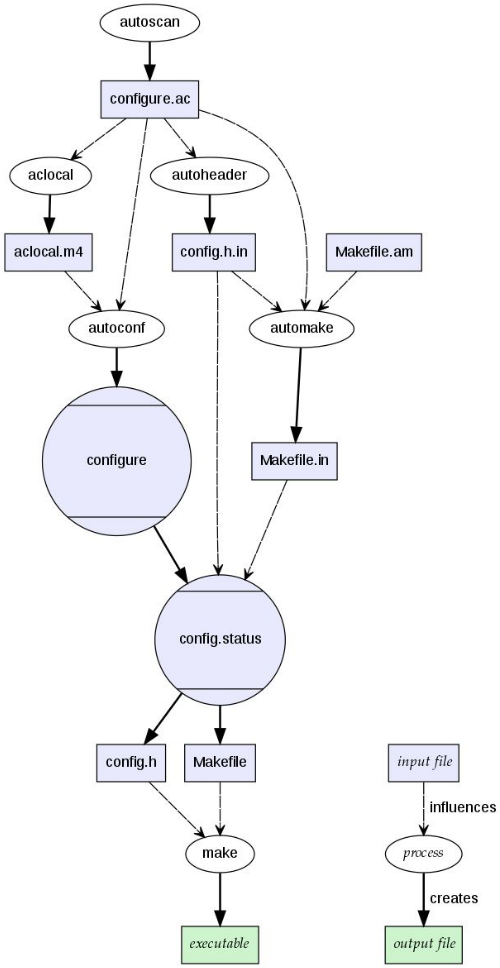
2) Makefile

- The larger and more complex the project, the more difficult it is to...
  - Keep track of vast dependency information
  - Specify additional tasks before / after build
  - Adjust build options for different target platforms

- So, pure Makefiles are rarely used in the field. All serious projects use “Makefile generators” or alternatives.

3) Autotools

- Traditional Makefile generator
  – Many GNU tools are built using it
- Too complicated!
  – Main tools (autoconf, automake, libtool) are separate but highly dependent on each other
  – Need to know how to use other languages: bash script, m4
  – “autohell”



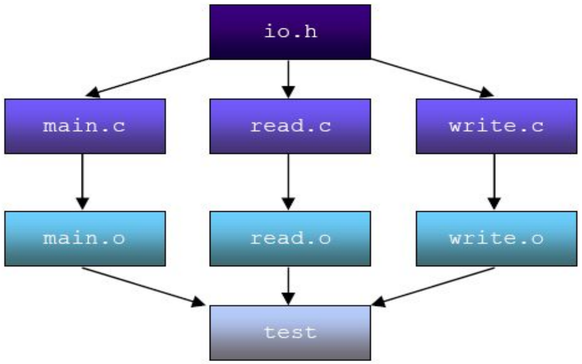
4) CMake

- Much easier to use with relatively simple syntax

- Cross-platform
  - On Unix/Linux: Generates Makefile
  - On Windows: Generates Visual Studio project file (.vcxproj)

- Some large open source projects has moved to CMake
  - KDE, https://lwn.net/Articles/188693/
  - https://gitlab.kitware.com/cmake/community/wikis/doc/cmake/Projects

Example using Makefile



Makefile

```
test : read.o write.o main.o
	gcc -o test read.o write.o main.o

main.o : io.h main.c
	gcc -c main.c

read.o : io.h read.c
	gcc -c read.c

write.o: io.h write.c
	gcc -c write.c
```

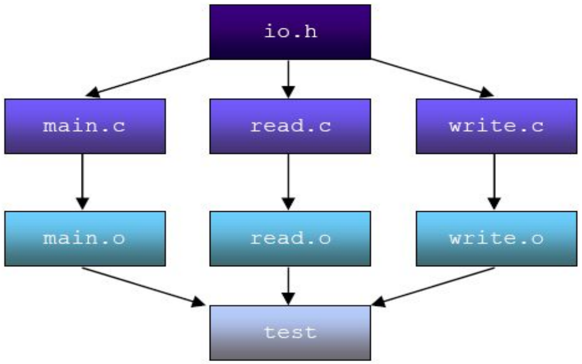
Makefile

```
CC=g++
SRCS=main.c read.c write.c
OBJS=$(SRCS:%.c=%.o)
TARGET=test

.SUFFIXES : .c .o

$(TARGET): $(OBJS)
	$(CC) -o $(TARGET) $(OBJS)

main.o: io.h main.c
read.o: io.h read.c
write.o: io.h write.c
```

(Shell)

```
make
```

Example using CMake



CMakeLists.txt

```
add_executable( test main.c read.c write.c )
```

(Shell)

```
cmake
make
```

Build Process using CMake

CMake Project File	CMakeLists.txt
	↓ **CMake**
Native Project File	Makefile, .vcxproj, … (+source files)
	↓ Compiler (gcc, g++, Visual C++, …)
Object files	.o, .obj
	↓ Linker (gcc, g++, Visual C++, …)
Executables or Libraries	a.out, .a, .so, .exe, .lib, .dll

[Practice] CMake

- Install CMake

```
(Shell)
sudo apt-get install cmake
```

[Practice] CMake

- Create these files somewhere

my_print.h

```
#pragma once
void MyPrint(
    const std::string& s, int n);
```

main.cpp

```
#include <string>
#include "my_print.h"

int main() {
  MyPrint("hello world", 5);
  return 0;
}
```

my_print.cpp

```
#include <iostream>
#include <string>

void MyPrint(const std::string& s, int n) {
  for (int i = 0; i < n; ++i)
    std::cout << s << std::endl;
}
```

CMakeLists.txt

```
add_executable(test main.cpp myprint.cpp)
```

[Practice] CMake

- Create a build directory & cd
  - The name does not have to be “build”.

(Shell)

```
mkdir build
cd build
```

```
▼ test/
      build/
      CMakeLists.txt
      main.cpp
      myprint.h
      myprint.cpp
```

[Practice] CMake

- Run CMake
  - "Generate Makefile using CMakeLists.txt in the parent directory(../)"

(Shell)

```
cmake ../
```

```
▼ build/
  ▸ CMakeFiles/
    cmake_install.cmake
    CMakeCache.txt
    Makefile
```

Final output

Intermediate output

- Run Make
  - "Compile & link the project using Makefile in the current directory(./)"

(Shell)

```
make
```

(Shell)

```
./test # run the final executable
```

More about CMake

- We've just covered very basic usage of CMake
- The real power of CMake comes from more complicated projects using a bunch of libraries, subdirectories, etc
  - add_library(), target_link_libraries(), add_subdirectory(), target_include_directories(), find_package(), ...
- More resource
  - https://cmake.org/cmake-tutorial/
  - https://cmake.org/cmake/help/v3.12/#reference-manuals

Next Time

- Next lecture:
  - 6. Class