

C/C++ Compilation and Linkage



한양대학교 컴퓨터소프트웨어학부
2017년 2학기

C/C++ Build Stages

example.c

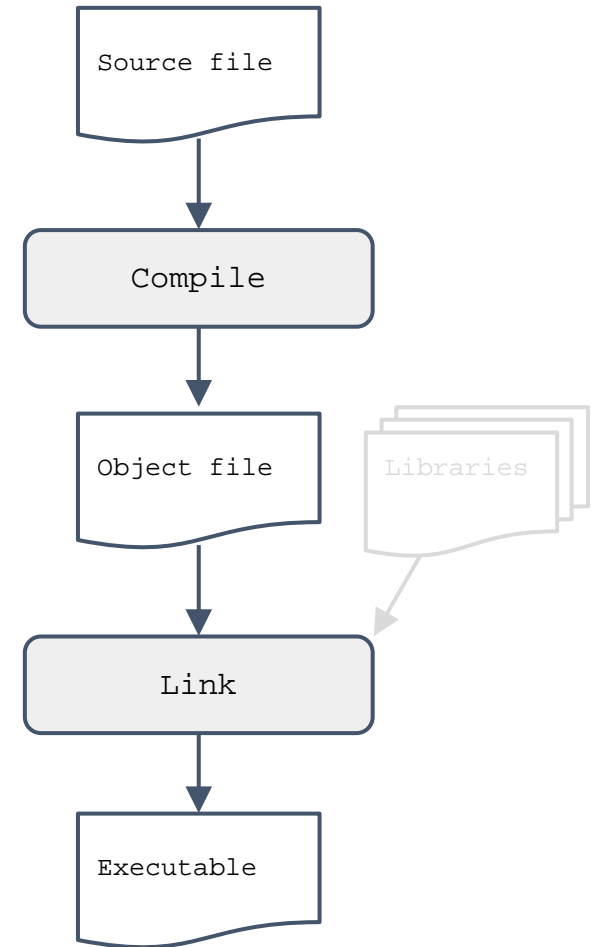
```
int FuncInt(int a, int b) {  
    ...  
}  
  
int FuncDouble(double a, double b, double c) {  
    ...  
}  
  
int main() { ... }
```

example.o

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

example (example.exe)

.....



C/C++ Build Stages

example.c

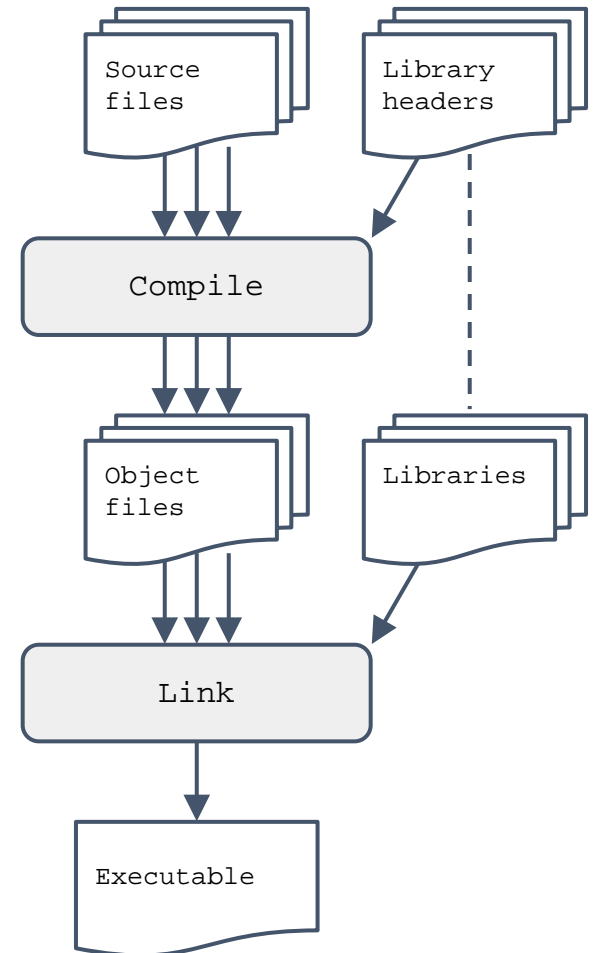
```
#include <math.h>

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

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

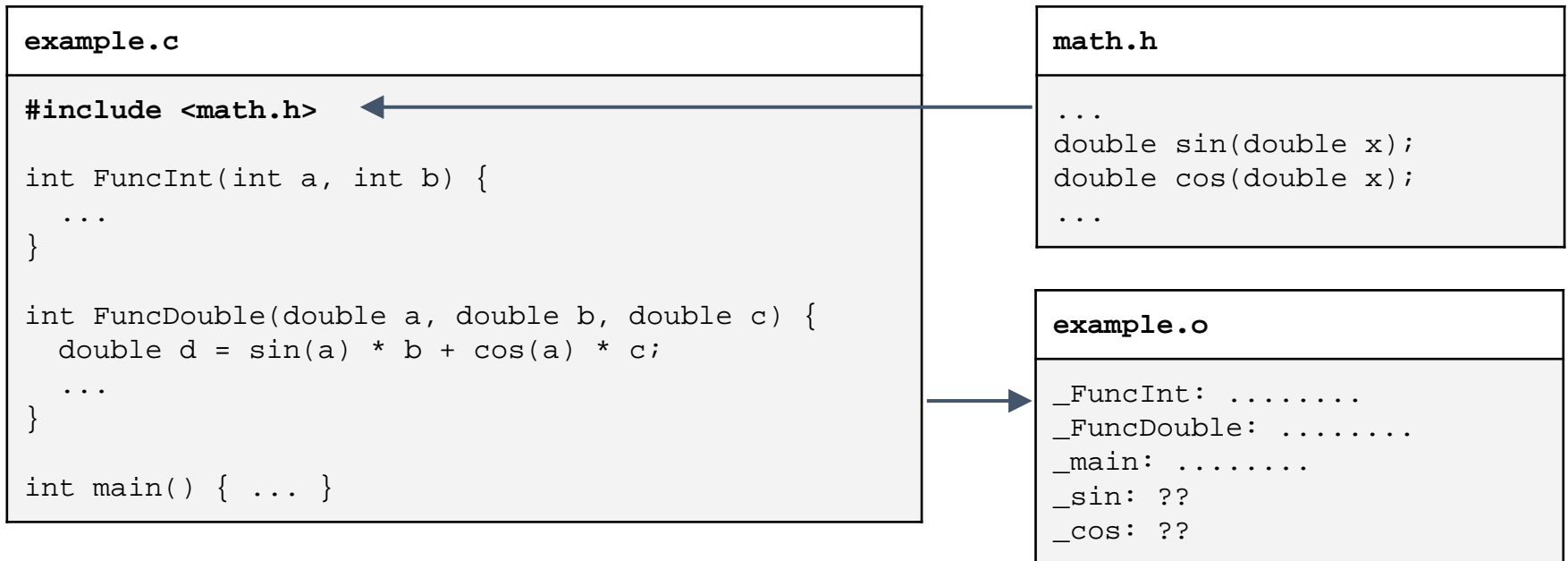
int main() { ... }
```

How do we know the signature of the function `sin` and `cos`?
E.g. how can the compiler find syntax errors?



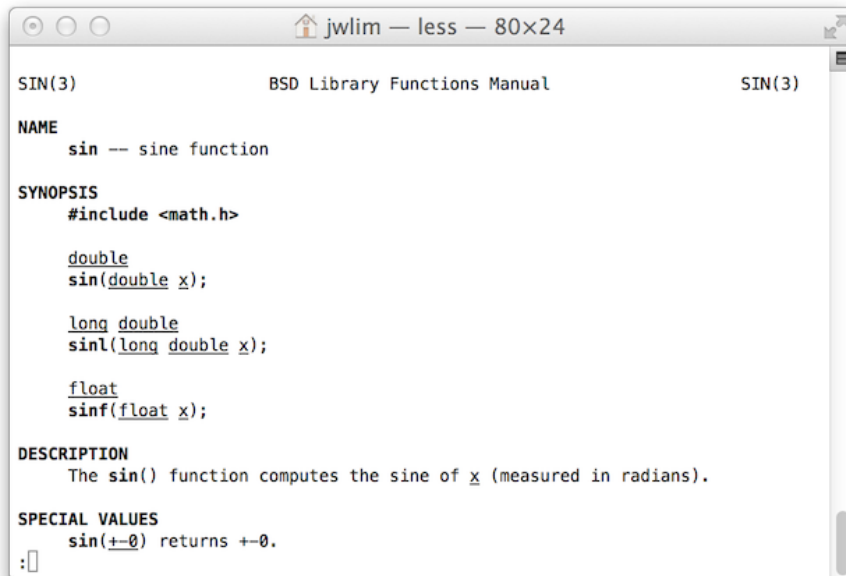
C/C++ Compilation

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



C/C++ Standard Library Header

- You don't need to find the actual header file to check the function signatures while you are programming.
- There are **man** pages for all functions in C standard library.

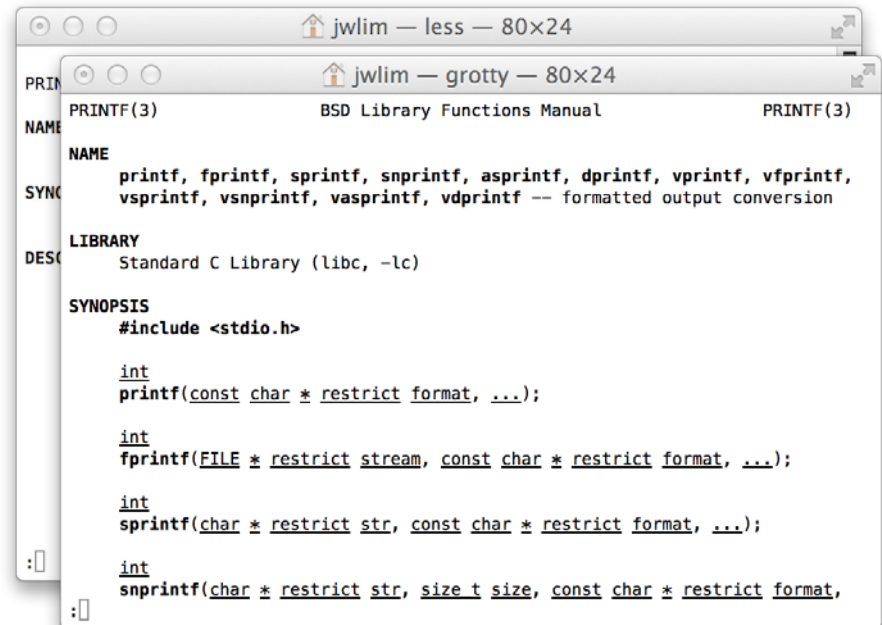


```
jwlim — less — 80x24
SIN(3)          BSD Library Functions Manual          SIN(3)
NAME
    sin -- sine function
SYNOPSIS
    #include <math.h>

    double
    sin(double x);

    long double
    sinl(long double x);

    float
    sinf(float x);
DESCRIPTION
    The sin() function computes the sine of x (measured in radians).
SPECIAL VALUES
    sin(+0) returns +0.
:
```



```
jwlim — grotty — 80x24
PRINTF(3)       BSD Library Functions Manual          PRINTF(3)
NAME
    printf, fprintf, sprintf, snprintf, asprintf, dprintf, vprintf, vfprintf,
    vsprintf, vsnprintf, vasprintf, vdprintf -- formatted output conversion
LIBRARY
    Standard C Library (libc, -lc)
SYNOPSIS
    #include <stdio.h>

    int
    printf(const char * restrict format, ...);

    int
    fprintf(FILE * restrict stream, const char * restrict format, ...);

    int
    sprintf(char * restrict str, const char * restrict format, ...);

    int
    snprintf(char * restrict str, size_t size, const char * restrict format,
```

C/C++ Build Stages

example.c

```
#include <math.h>

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

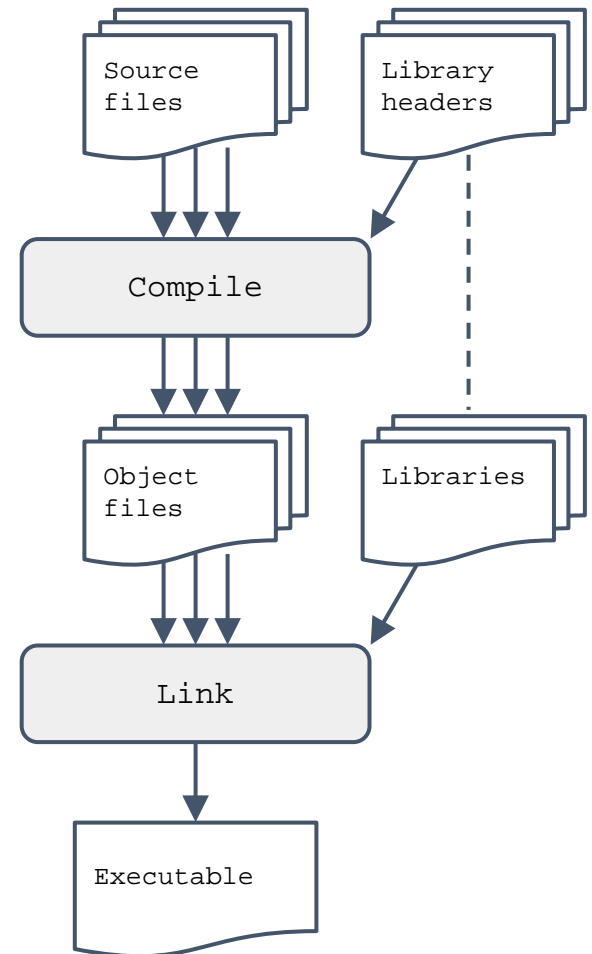
int FuncDouble(double a, double b, double c) {
    double d = sin(a) * b + cos(a) * c;
    ...
}

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

example.o

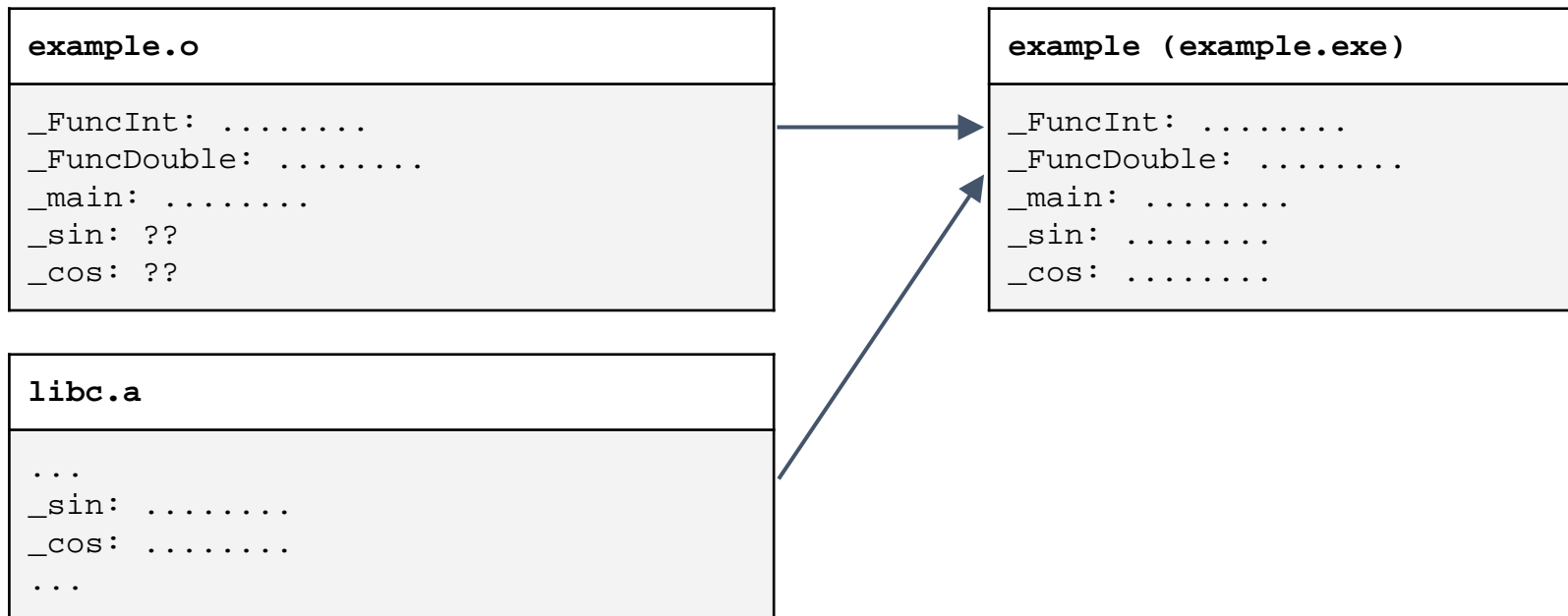
```
_FuncInt: .....
_FuncDouble: .....
_main: .....
_sin: ??
_cos: ??
```

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



C/C++ Linking

- Linker tries to find all unknown symbols in the object files and the libraries.
- A library is just a collection of object files.



Header and Source Files

Header file's extension is '.h' and source file's is '.cc' or '.cpp'.

C/C++ header files contain

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

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

Function Declaration and Definition

- Function declaration only specifies the function name, parameter profile, and the return type.
- Function definition provides the actual implementation of the function body.

```
#include <math.h>

int FuncInt(int a, int b);

double MyFunc(const int* array, int n, const char* command);

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);
}
```

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;
}
```

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

- Function definitions in a class definition are inline functions.
- Otherwise specify with the keyword `inline`.

```
class SimpleIntSet {
public:
    SimpleIntSet() : values_(NULL), size_(0) {}
    ~SimpleIntSet() { delete[] values; }

    inline void Set(const int* values, size_t size);
    const int* values() const { return values_; }
    size_t size() const { return size_; }

private:
    int* values_;
    size_t size_;
};

void SimpleIntSet::Set(const int* values, size_t size) {
    ...
}

int main() {
    SimpleIntSet int_set;
    int_set.Set(...);
    return 0;
}
```

Building Multi-file Project

- Give all source files to the compiler.

```
$ g++ -o my_example my_example.cc main.cc
```

- Compile the source files first, then link the object files.

```
$ g++ -c my_example.cc main.cc  
$ g++ -o my_example my_example.o main.o
```

- Make a library with the source files, then link the library.

```
$ g++ -c my_example.cc main.cc  
$ ar rvs libmyex.a my_example.o  
$ g++ -o my_example main.o libmyex.a    # OR -lmyex -L.
```

Example Header and Source

my_example.h

```
// my_example.h
// Author: jwlim

#ifndef MY_EXAMPLE_H_
#define MY_EXAMPLE_H_

#define MIN(a, b) ((a) < (b) ? (a) : (b))
extern int my_error_no;

enum {
    ERROR = 0, OK = 1, WARNING = 2
};

// Returns the squared sum of the array.
double SquaredSum(const double* array,
                  int n);
// Computes the norm of the vector.
double Norm(const double* array, int n);

#endif // MY_EXAMPLE_H_
```

my_example.cc

```
// my_example.cc
// Author: jwlim

#include "my_example.h"
#include <math.h>

int my_error_no = 0;

double SquaredSum(const double* array,
                  int n) {
    int sqsum = 0;
    for (int i = 0; i < n; ++i) {
        sqsum += array[i] * array[i];
    }
    return sqsum;
}

double Norm(const double* array, int n) {
    return sqrt(SquaredSum(array, n));
}
```

Example Header and Source

my_example.h

```
// my_example.h
// Author: jwlim

#ifndef MY_EXAMPLE_H_
#define MY_EXAMPLE_H_

#define MIN(a, b) ((a) < (b) ? (a) : (b))
extern int my_error_no;

enum {
    ERROR = 0, OK = 1, WARNING = 2
};

// Returns the squared sum of the array.
double SquaredSum(const double* array,
                  int n);
// Computes the norm of the vector.
double Norm(const double* array, int n);

#endif // MY_EXAMPLE_H_
```

my_example_main.cc

```
// main.cc
// Author: jwlim

#include <stdio.h>
#include <stdlib.h>
#include "my_example.h"

int main(int argc, const char** argv) {
    if (argc < 2) return 0;
    const int buflen = 10;

    // Q: will MIN(--argc, buflen) work?
    const int n = MIN(argc - 1, buflen);
    // Q: will 'double val[n];' compile?
    double val[buflen];
    for (int i = 0; i < n; ++i) {
        val[i] = atof(argv[i + 1]);
    }

    double norm = Norm(val, n);
    printf("norm = %.3f, error = %d\n",
           norm, my_error_no);
    return 0;
}
```

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 arguments are passed.

```
$ ./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
```

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

Command-line Arguments

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

```
-> argc: 5
```

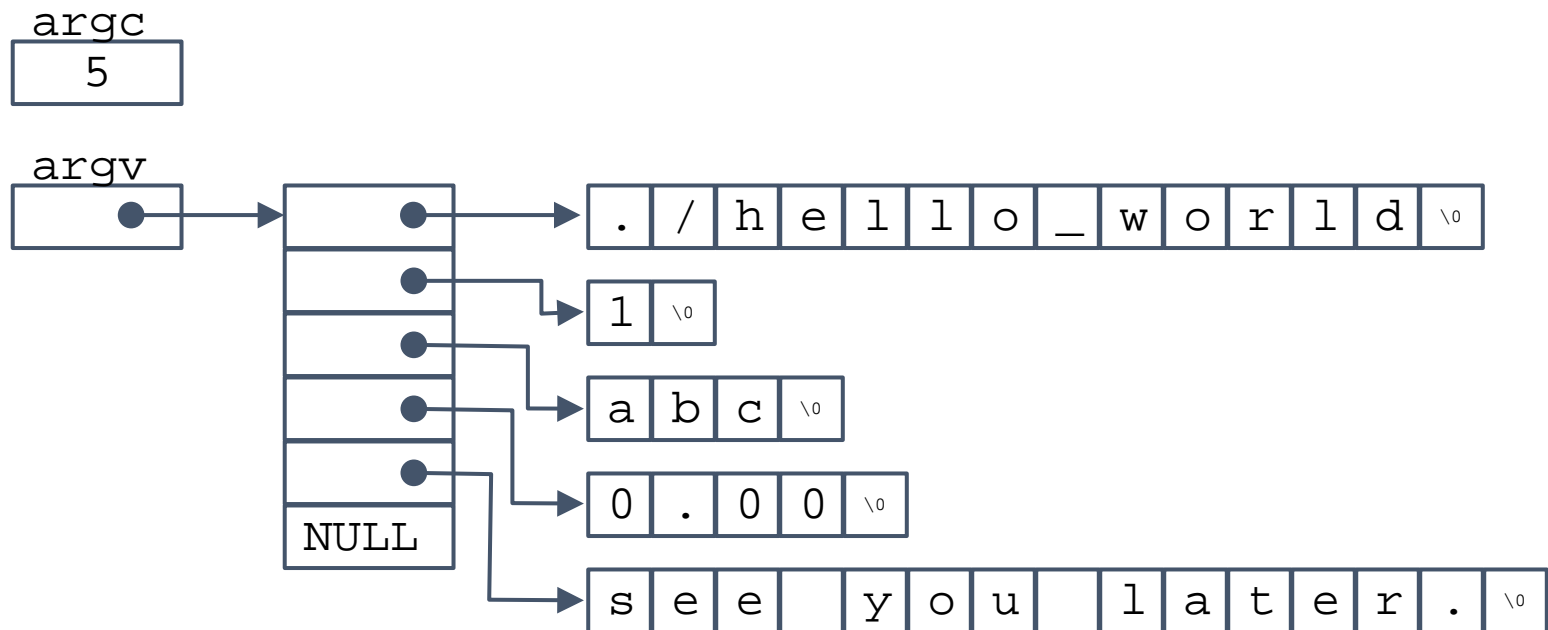
```
    argv[0]: "./hello_world"    argv[3] = "0.00"
```

```
    argv[1]: "1"
```

```
    argv[2]: "abc"
```

```
    argv[4] = "see you later."
```

```
    argv[5] = NULL
```



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.
```

Summary

- ◆ Function declaration vs. definition
- ◆ Header files and source files
- ◆ Compiler, linker, preprocessor
- ◆ Command-line arguments

