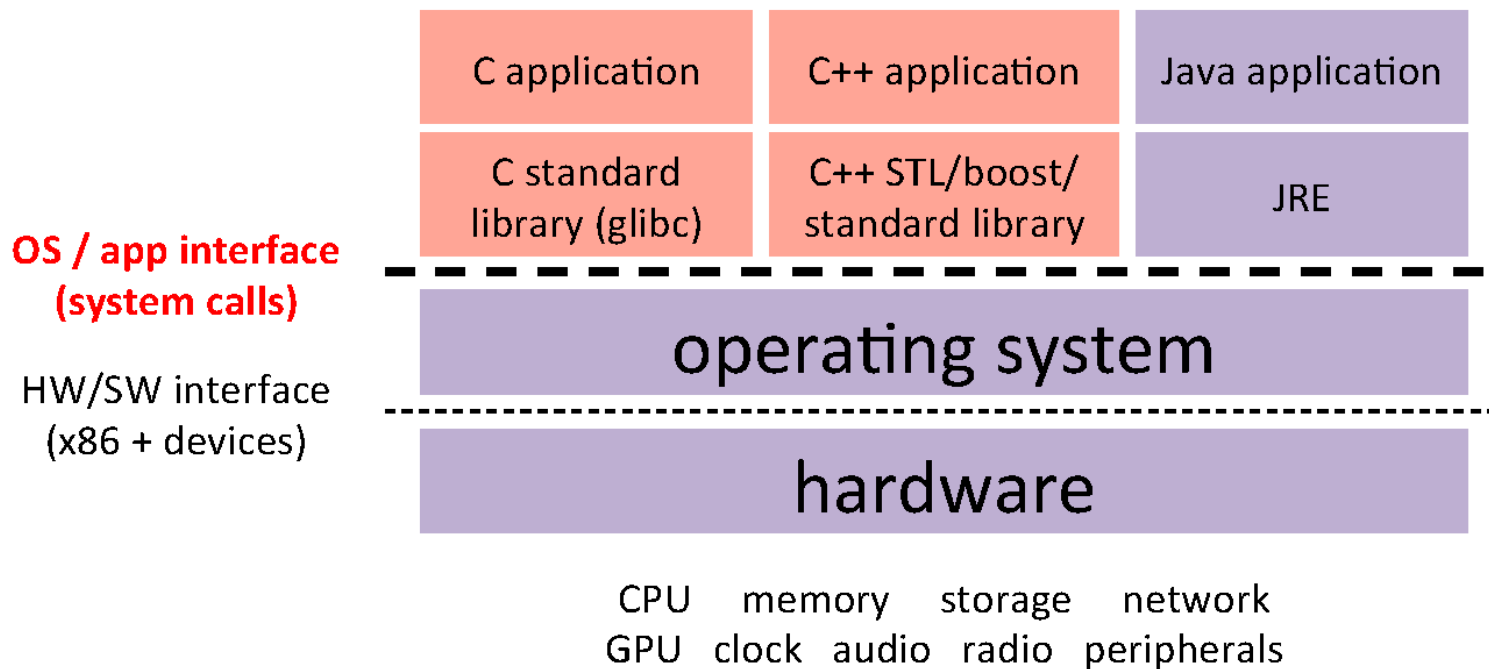


introduction to System Programming

System Programming using C

Course Map: 100,000 foot view



System Programming

A **system** is a set of interacting or interdependent entities

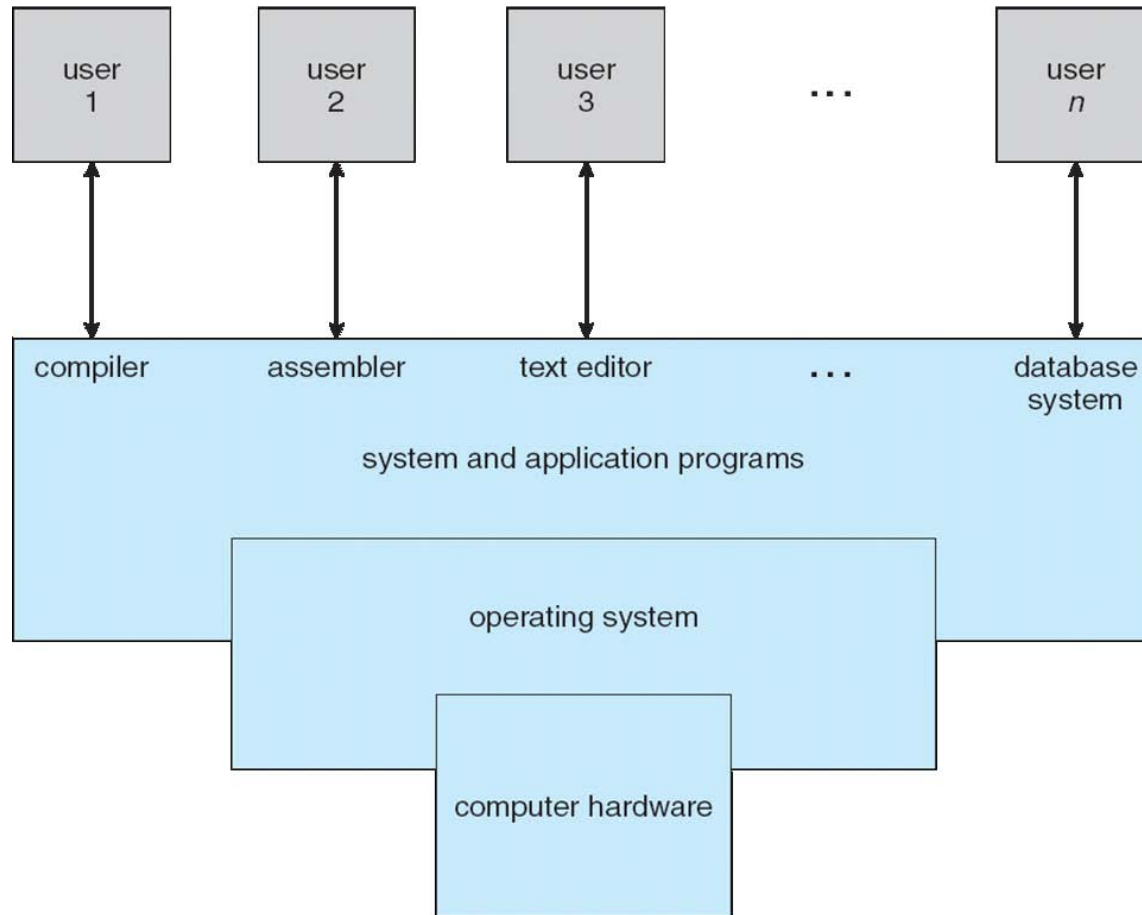
- Managing complexity is the key problem in systems engineering

System Programming: is the writing System Software.

- Systems software is software supporting application software.

Examples of Systems Software: Operating systems / Firmware, Programming tools (compilers, linkers, loaders), (Essential) runtime libraries of programming languages

System Programs



System Programs

- **System programs provide a convenient environment for program development and execution.** They can be divided into:
 - File manipulation
 - Status information sometimes stored in a File modification
 - Programming language support
 - Program loading and execution
 - Communications
 - Background services
- Most users' view of the operation system is defined by system programs, not the actual system calls

System Programs

- Provide a convenient environment for program development and execution
 - Some of them are simply user interfaces to system calls; others are considerably more complex
- **File management** - Create, delete, copy, rename, print, dump, list, and generally manipulate files and directories
- **Status information**
 - Some ask the system for info - date, time, amount of available memory, disk space, number of users
 - Others provide detailed performance, logging, and debugging information
 - Typically, these programs format and print the output to the terminal or other output devices
 - Some systems implement a **registry** - used to store and retrieve configuration information

System Programs (Cont.)

- **File modification**
 - Text editors to create and modify files
 - Special commands to search contents of files or perform transformations of the text
- **Programming-language support** - Compilers, assemblers, debuggers and interpreters sometimes provided
- **Program loading and execution**- Absolute loaders, relocatable loaders, linkage editors, and overlay-loaders, debugging systems for higher-level and machine language
- **Communications** - Provide the mechanism for creating virtual connections among processes, users, and computer systems
 - Allow users to send messages to one another's screens, browse web pages, send electronic-mail messages, log in remotely, transfer files from one machine to another

System Programs (Cont.)

□ Background Services

- Launch at boot time
 - Some for system startup, then terminate
 - Some from system boot to shutdown
- Provide facilities like disk checking, process scheduling, error logging, printing
- Run in user context not kernel context
- Known as **services**, **subsystems**, **daemons**

□ Application programs

- Don't pertain to system
- Run by users
- Not typically considered part of OS
- Launched by command line, mouse click, finger poke

Systems Programming

- ❖ The programming skills, engineering discipline, and knowledge you need to build a system
 - **Programming:** C / C++
 - **Discipline:** testing, debugging, performance analysis
 - **Knowledge:** long list of interesting topics
 - Concurrency, OS interfaces and semantics, techniques for consistent data management, distributed systems algorithms, ...
 - Most important: a deep(er) understanding of the “layer below”

Why Study C?

- C is the language of the systems programmer
- All major Operating Systems are written in C (BSD, Linux, OS X, Windows)
- C is a subset of C++ (so you cannot know C++ and not know C)
- C and C++ together are used for more than $\frac{1}{3}$ of all open-source projects at SourceForge¹

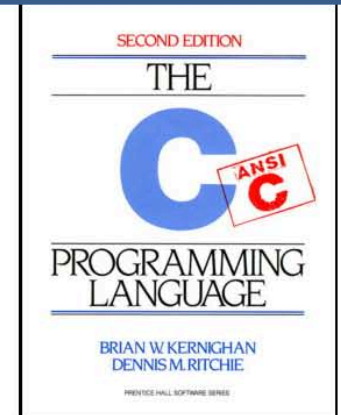
¹<http://www.cs.berkeley.edu/~flab/languages.html>

Discipline?!?

- ❖ Cultivate good habits, encourage clean code
 - Coding style conventions
 - Unit testing, code coverage testing, regression testing
 - Documentation (code comments, design docs)
 - Code reviews
- ❖ Will take you a lifetime to learn
 - But oh-so-important, especially for systems code
 - Avoid write-once, read-never code

C

- ❖ Created in 1972 by Dennis Ritchie
 - Designed for creating system software
 - Portable across machine architectures
 - Most recently updated in 1999 (C99) and 2011 (C11)
- ❖ Characteristics
 - “Low-level” language that allows us to exploit underlying features of the architecture – **but easy to fail spectacularly (!)**
 - Procedural (not object-oriented)
 - Typed but unsafe (possible to bypass the type system)
 - Small, basic library compared to Java, C++, most others....



Generic C Program Layout

```
#include <system_files>
#include "local_files"

#define macro_name macro_expr

/* declare functions */
/* declare external variables & structs */

int main(int argc, char* argv[]) {
    /* the innards */
}

/* define other functions */
```

C Syntax: main

- ❖ To get command-line arguments in `main`, use:

```
int main(int argc, char* argv[])
```

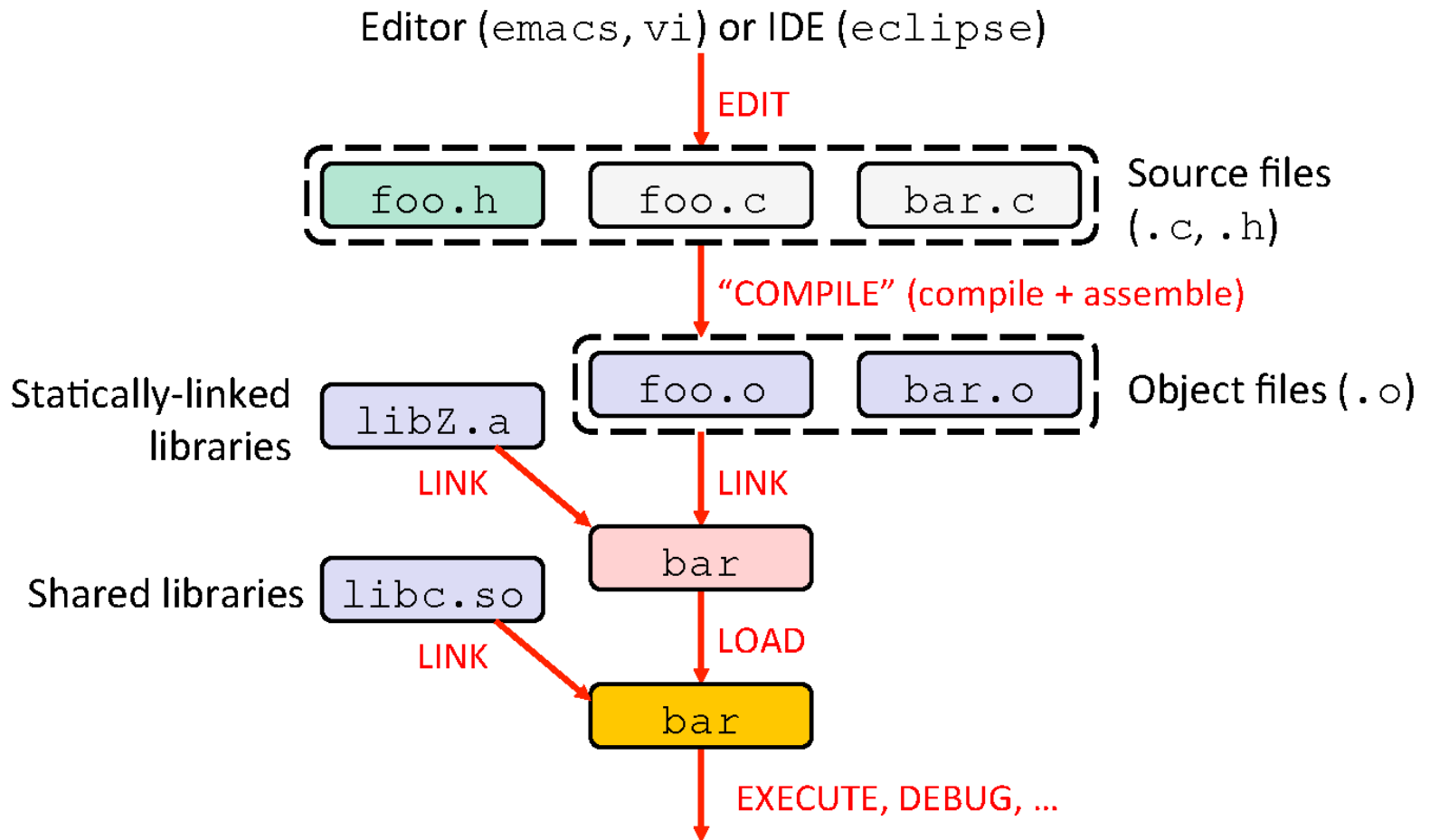
- ❖ What does this mean?

- `argc` contains the number of strings on the command line (the executable name counts as one, plus one for each argument).
- `argv` is an array containing *pointers* to the arguments as strings (more on pointers later)

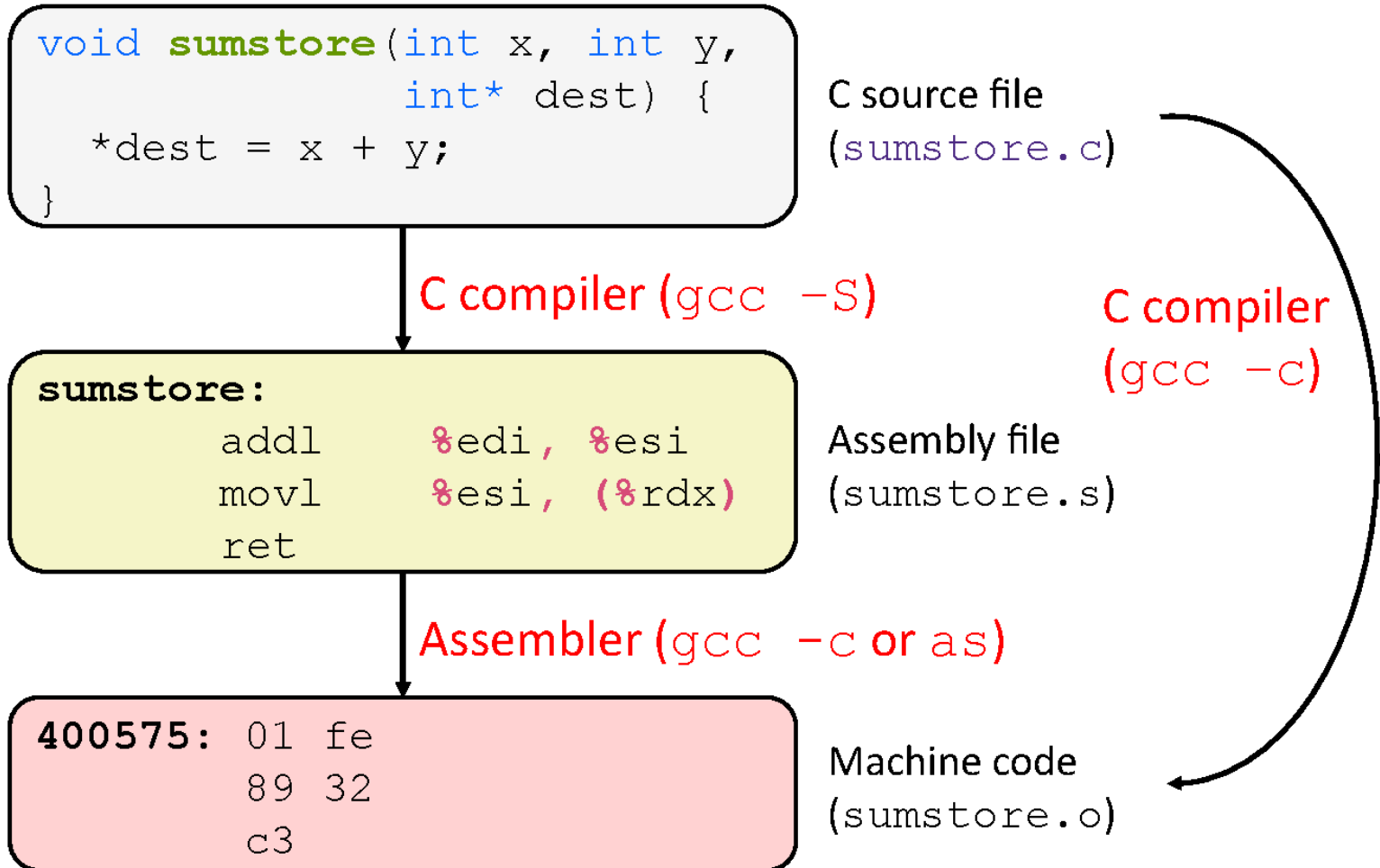
- ❖ Example: `$ foo hello 87`

- `argc = 3`
- `argv[0] = "foo", argv[1] = "hello", argv[2] = "87"`

C Workflow



C to Machine Code



When Things Go South...

- ❖ Errors and Exceptions
 - C does not have exception handling (no `try/catch`)
 - Errors are returned as integer error codes from functions
 - Because of this, error handling is ugly and inelegant
- ❖ Crashes
 - If you do something bad, you hope to get a “segmentation fault” (believe it or not, this is the “good” option)

Java vs. C (351 refresher)

- ❖ Are Java and C mostly similar (S) or significantly different (D) in the following categories?
 - List any differences you can recall (even if you put 'S')

Language Feature	S/D	Differences in C
Control structures		
Primitive datatypes		
Operators		
Casting		
Arrays		
Memory management		

Java vs. C (351 refresher)

- ❖ Are Java and C mostly similar (S) or significantly different (D) in the following categories?
 - List any differences you can recall (even if you put 'S')

Language Feature	S/D	Differences in C
Control structures	S	
Primitive datatypes	S/D	Similar but sizes can differ (char, esp.), unsigned, no boolean, uninitialized data, ...
Operators	S	Java has >>>, C has ->
Casting	D	Java enforces type safety, C does not
Arrays	D	Not objects, don't know their own length, no bounds checking
Memory management	D	Manual (malloc/free), no garbage collection

Primitive Types in C

❖ Integer types

- `char, int`

❖ Floating point

- `float, double`

❖ Modifiers

- `short [int]`
- `long [int, double]`
- `signed [char, int]`
- `unsigned [char, int]`

C Data Type	32-bit	64-bit	printf
<code>char</code>	1	1	<code>%c</code>
<code>short int</code>	2	2	<code>%hd</code>
<code>unsigned short int</code>	2	2	<code>%hu</code>
<code>int</code>	4	4	<code>%d/%i</code>
<code>unsigned int</code>	4	4	<code>%u</code>
<code>long int</code>	4	8	<code>%ld</code>
<code>long long int</code>	8	8	<code>%lld</code>
<code>float</code>	4	4	<code>%f</code>
<code>double</code>	8	8	<code>%lf</code>
<code>long double</code>	12	16	<code>%Lf</code>
<code>pointer</code>	4	8	<code>%p</code>

Typical sizes – see `sizeofs.c`

C99 Extended Integer Types

- ❖ Solves the conundrum of “how big is an `long int`?”

```
#include <stdint.h>

void foo(void) {
    int8_t  a; // exactly 8 bits, signed
    int16_t b; // exactly 16 bits, signed
    int32_t c; // exactly 32 bits, signed
    int64_t d; // exactly 64 bits, signed
    uint8_t w; // exactly 8 bits, unsigned
    ...
}
```

Use extended types in cse333 code

```
void sumstore(int x, int y, int* dest) {
```

```
void sumstore(int32_t x, int32_t y, int32_t* dest) {
```

Basic Data Structures

- ❖ C does not support objects!!!
- ❖ **Arrays** are contiguous chunks of memory
 - Arrays have no methods and do not know their own length
 - Can easily run off ends of arrays in C – **security bugs!!!**
- ❖ **Strings** are null-terminated char arrays
 - Strings have no methods, but `string.h` has helpful utilities

```
char* x = "hello\n";
```

x	h	e	l	l	o	\n	\0
---	---	---	---	---	---	----	----

- ❖ **Structs** are the most object-like feature, but are just collections of fields

Function Definitions

❖ Generic format:

```
returnType fName(type param1, ..., type paramN) {  
    // statements  
}
```

```
// sum of integers from 1 to max  
int sumTo(int max) {  
    int i, sum = 0;  
  
    for (i = 1; i <= max; i++) {  
        sum += 1;  
    }  
  
    return sum;  
}
```

Function Ordering

- ❖ You *shouldn't* call a function that hasn't been declared yet

sum_badorder.c

```
#include <stdio.h>

int main(int argc, char** argv) {
    printf("sumTo(5) is: %d\n", sumTo(5));
    return 0;
}

// sum of integers from 1 to max
int sumTo(int max) {
    int i, sum = 0;

    for (i = 1; i <= max; i++) {
        sum += 1;
    }
    return sum;
}
```


Solution 1: Reverse Ordering

- ❖ Simple solution; however, imposes ordering restriction on writing functions (who-calls-what?)

sum_betterorder.c

```
#include <stdio.h>

// sum of integers from 1 to max
int sumTo(int max) {
    int i, sum = 0;

    for (i = 1; i <= max; i++) {
        sum += 1;
    }
    return sum;
}

int main(int argc, char** argv) {
    printf("sumTo(5) is: %d\n", sumTo(5));
    return 0;
}
```

Solution 2: Function Declaration

- ❖ Teaches the compiler arguments and return types; function definitions can then be in a logical order

sum_declared.c

Hint: code examples from slides are on the course web for you to experiment with

```
#include <stdio.h>

int sumTo(int); // func prototype

int main(int argc, char** argv) {
    printf("sumTo(5) is: %d\n", sumTo(5));
    return 0;
}

// sum of integers from 1 to max
int sumTo(int max) {
    int i, sum = 0;
    for (i = 1; i <= max; i++) {
        sum += 1;
    }
    return sum;
}
```

Function Declaration vs. Definition

- ❖ C/C++ make a careful distinction between these two
- ❖ **Definition:** the thing itself
 - *e.g.* code for function, variable definition that creates storage
 - Must be **exactly one** definition of each thing (no duplicates)
- ❖ **Declaration:** description of a thing
 - *e.g.* function prototype, external variable declaration
 - Often in header files and incorporated via `#include`
 - Should also `#include` declaration in the file with the actual definition to check for consistency
 - Needs to appear in **all files** that use that thing
 - Should appear before first use

Multi-file C Programs

C source file 1
(sumstore.c)

```
void sumstore(int x, int y, int* dest) {  
    *dest = x + y;  
}
```

C source file 2
(sumnum.c)

```
#include <stdio.h>  
  
void sumstore(int x, int y, int* dest);  
  
int main(int argc, char** argv) {  
    int z, x = 351, y = 333;  
    sumstore(x, y, &z);  
    printf("%d + %d = %d\n", x, y, z);  
    return 0;  
}
```

Compile together:

```
$ gcc -o sumnum sumnum.c sumstore.c
```

Compiling Multi-file Programs

- ❖ The **linker** combines multiple object files plus statically-linked libraries to produce an executable
 - Includes many standard libraries (e.g. `libc`, `crt1`)
 - A *library* is just a pre-assembled collection of `.o` files

