Today’s topics

- Introductions
- Learning goals and objectives
- Course syllabus
- Exploring the classroom/C programming

I bit about myself

- My name is Taha Khan
  - It's my fourth year at Washington and Lee University (W&L)
  - You may call me Taha or Professor/Dr. Khan
- My research focuses on how can we make the Internet more secure, robust, and usable.
- How to contact me?
  - Office: Parmly 406
  - Slack for Online Communications
  - Office Hours: M/W/F from 2:30 pm – 3:30 pm

Class introductions

- What is your preferred name?
- What do you plan to get out of this course?
- What is the lowest level computer language you've programmed in?
- Tell us all something fun you did over the summers?

Systems programming

- What is a computer system?

- Themes of this course:
  - End to end process of compiling and executing programs
  - Inspect running programs and low-level debugging
  - Memory management and virtual memory
  - The details of I/O
  - Coordinating with other processes

Why C programming?

- C is the primary language that is used today for UNIX based systems
  - Standardization and portability
  - Low level capabilities, yet a human readable syntax
  - Generates very efficient code
- It has stood the test of time
- Roughly 80% of the computers out there are running UNIX systems

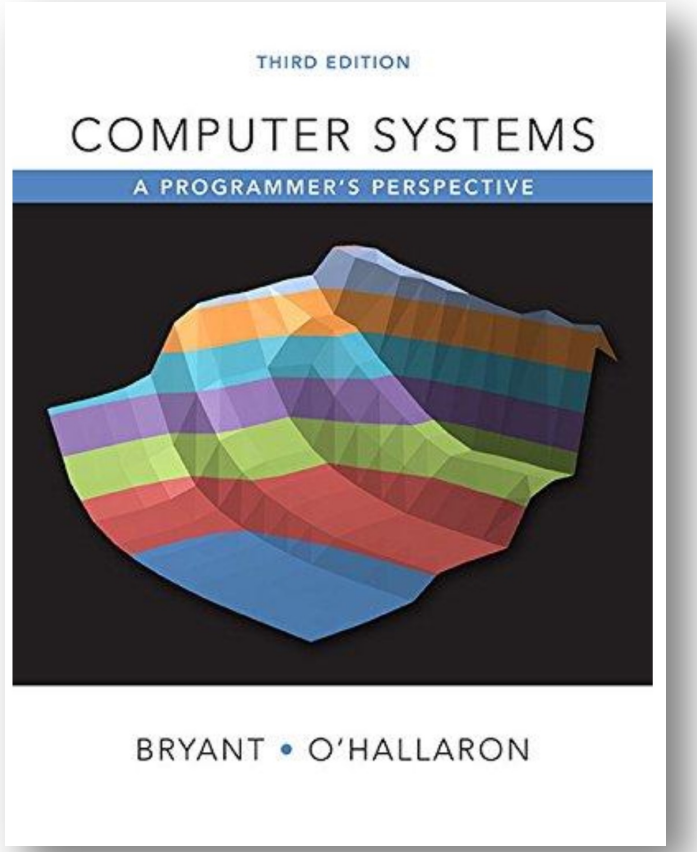
- Other benefits
  - Opens doors to many other avenues in computer science
  - A highly desired skill in the CS job market

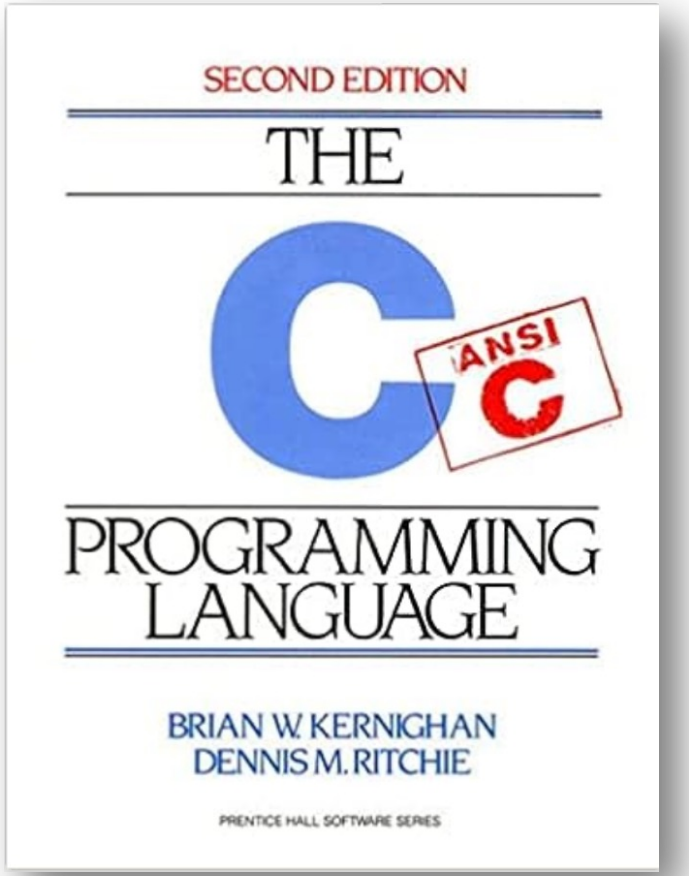
Course components and grading

- Homework assignments (35%)
- Midterm exam (20%)
- Final exam (30%)
- Class presentations (10%)
- In class activities (5%)
- Bonus points (up to 2%)

- Grades will be assigned on a relative curve
- You have a quota of 5 late days that you may use for homework assignments

Textbooks


THIRD EDITION
COMPUTER SYSTEMS
A PROGRAMMER'S PERSPECTIVE
BRYANT • O'HALLARON

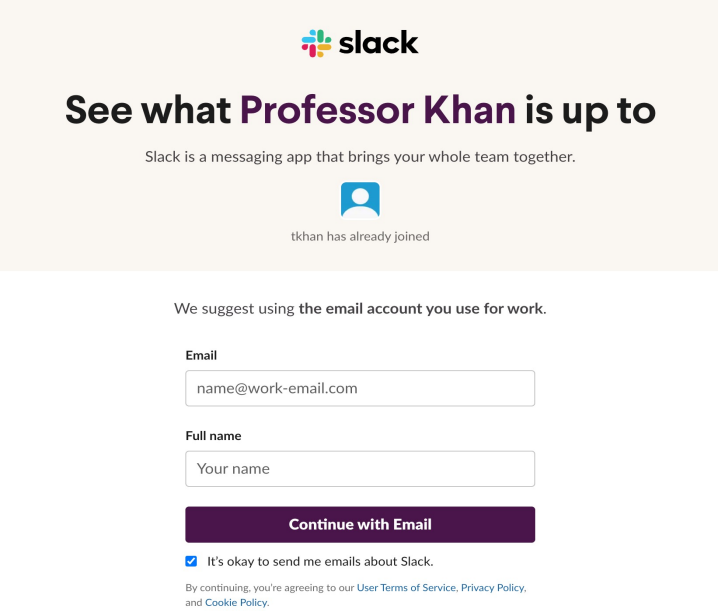

SECOND EDITION
THE
C
ANSI
C
PROGRAMMING
LANGUAGE
BRIAN W. KERNIGHAN
DENNIS M. RITCHIE
PRENTICE HALL SOFTWARE SERIES

Academic integrity

- Collaboration
  - You may discuss approaches, not solutions
  - You must submit your own work
  - Exams may include questions on programming
- Cheating
  - Zero tolerance for cheating, don’t do it!
  - Potential violations of the W&L Honor System will be reported to the EC
  - All cheating results in AT LEAST one letter grade reduction
- You may use generative AI for studying concepts, but not to generate code

Resources and communications

- W&L Canvas
  - Course materials
  - Homework assignment submissions
  - Grades
- Gradescope
  - Exams
- Slack
  - Communications
  - Please use your **Full Name** in **Title Case** when signing up



Bring your laptops to class

- In class programming exercises
- Exploration of tools
- AWS for homework assignments

C programming exercises

1. Write a simple C function that swaps two integer variables using pointers
   - Test your function in **`main()`**
   - Compile and run your program using **`gcc`**

2. Next, modify your function (above), so that it is able to swap variables of any data type
   - Make sure to test it as well

3. Write a function that converts Celsius temperature to Fahrenheit
   - The program should prompt the user for input temperature in Celsius
   - You may compute everything in the main function or write a helper

Next Class

- **Topic:** C programming basics
- **Reading:** Chapter 1, 2 and 3 of the C Book (K&R)

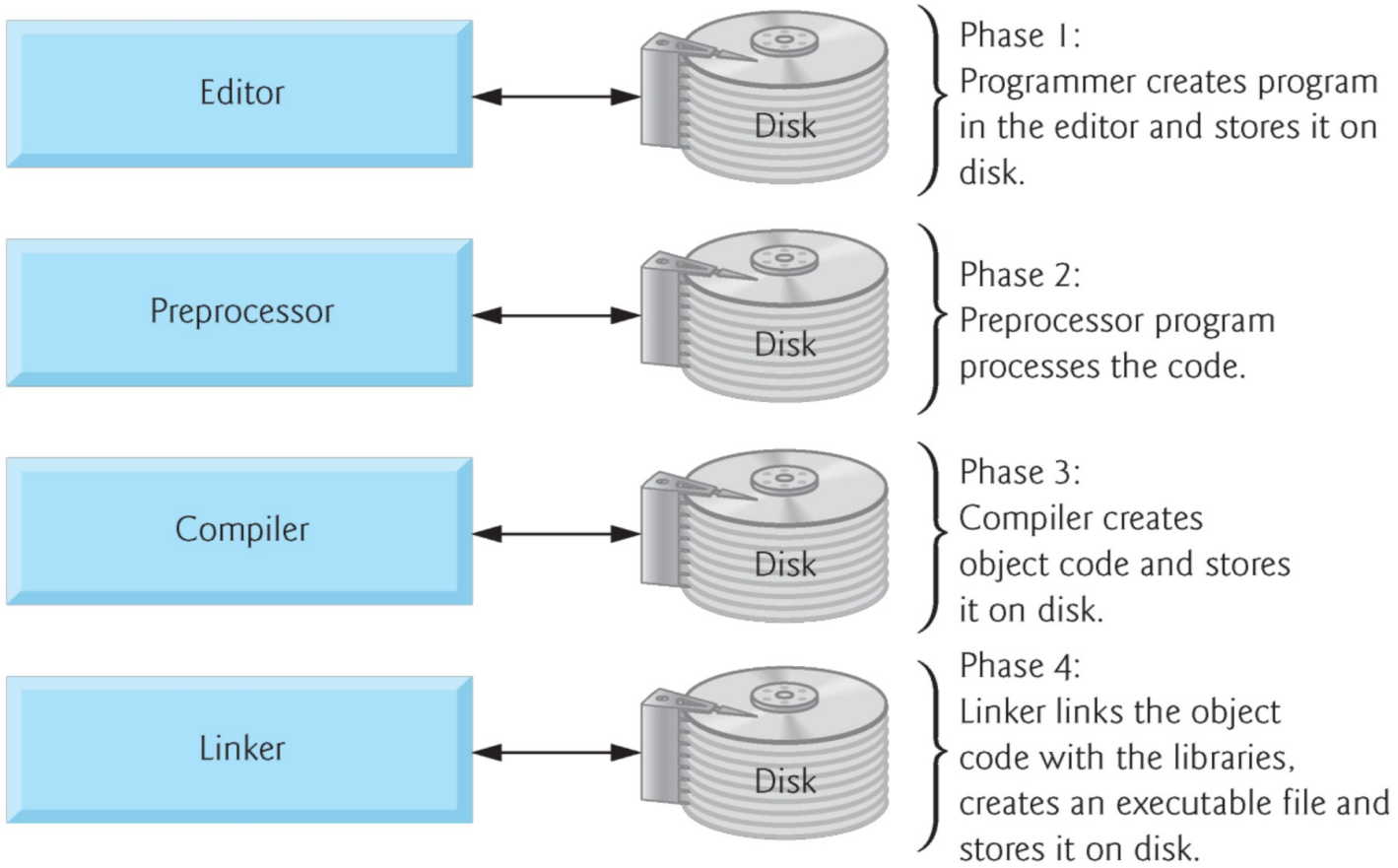
Next Class

- C programming
- The `gcc` compiler
- Datatypes
- User input/output

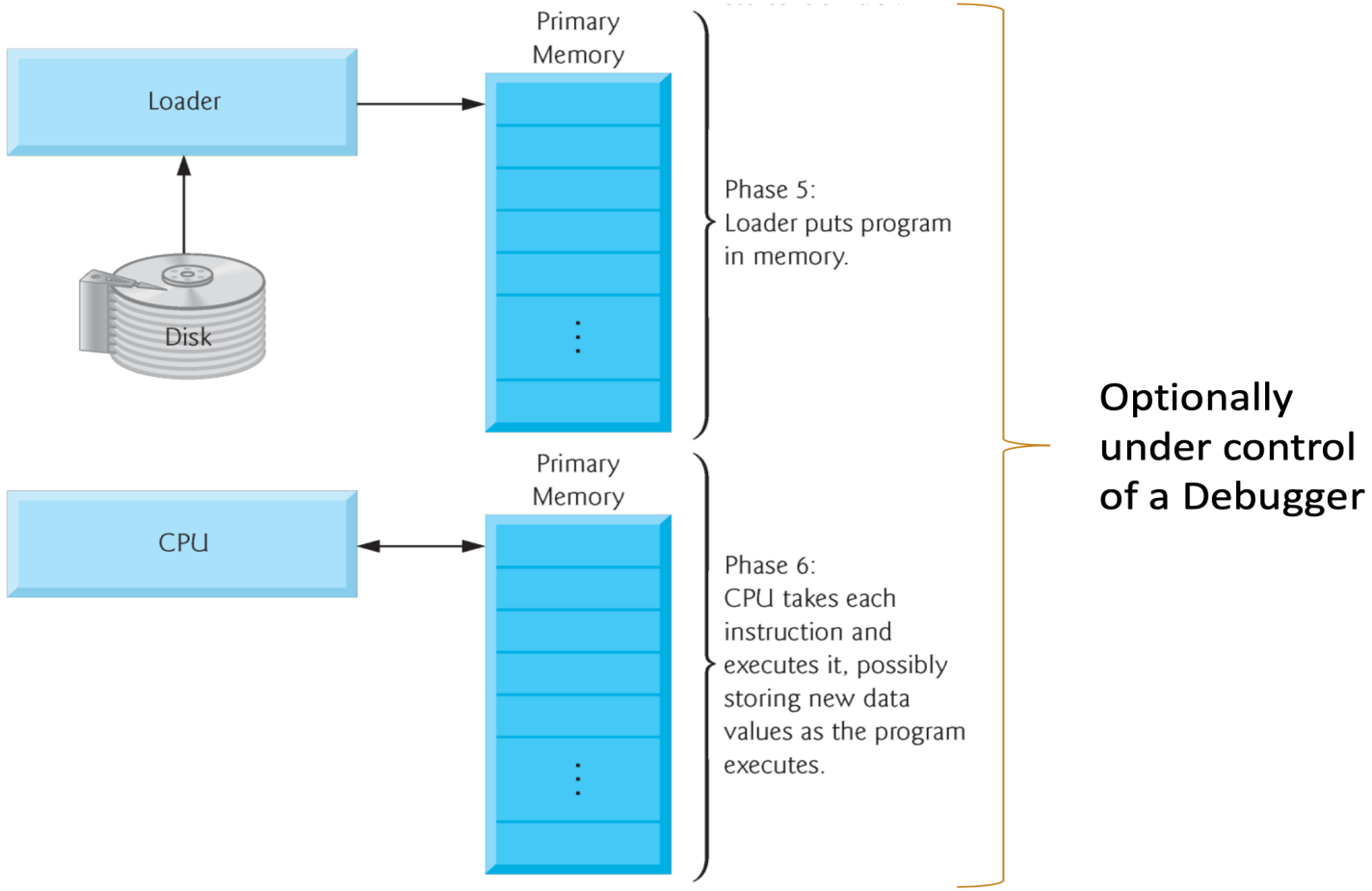
History of C

- Developed at AT&T Bell Labs in early 1970s
- Unix also developed at Bell Labs
- All but core of Unix is in C
  - Linux and BSD are very similar to Unix
- Standardized by American National Standards Institute (ANSI)

Phases of a C program



Next Class


Primary Memory
Loader
Disk
Phase 5:
Loader puts program in memory.
Primary Memory
CPU
Phase 6:
CPU takes each instruction and executes it, possibly storing new data values as the program executes.
Optionally under control of a Debugger

Compiling a C program

- To compile a C program we use **`gcc`**
  - **`gcc hello_world.c`**
  - **`gcc -o hello_exec hello_world.c`**

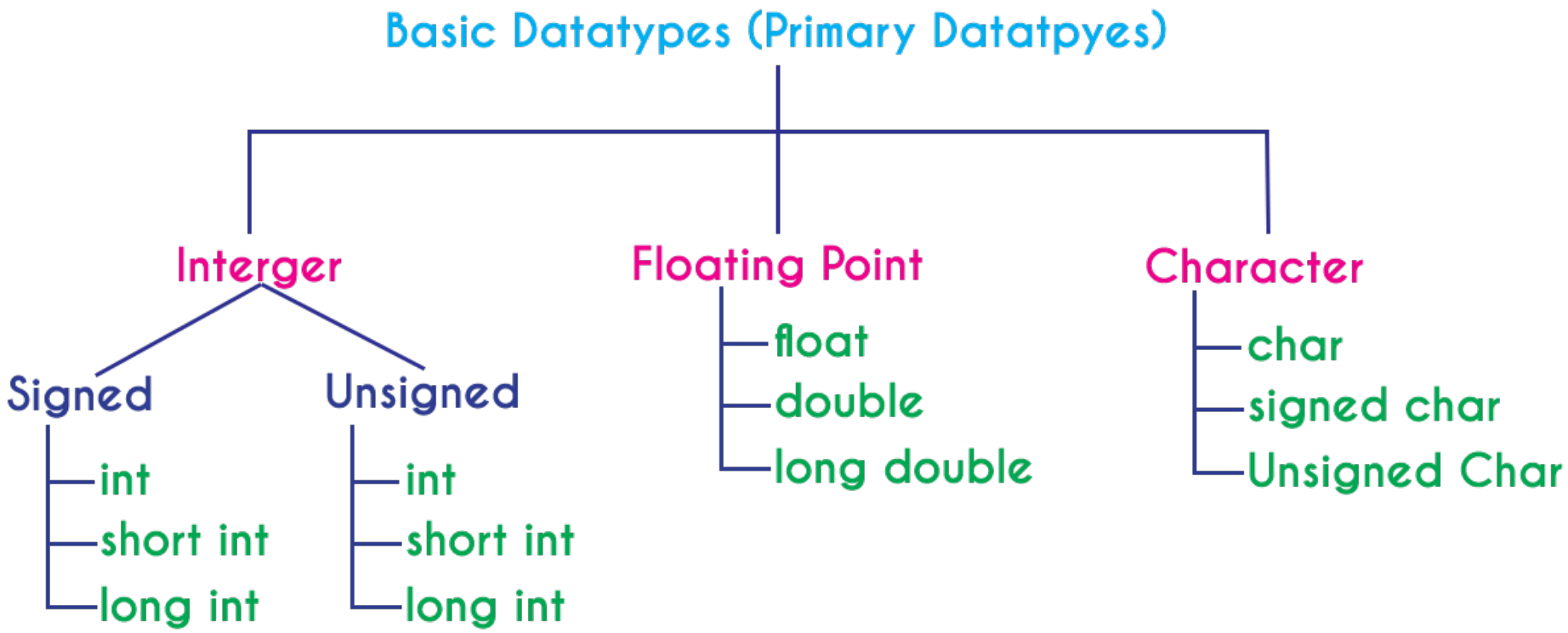
- Extra flags to see warnings:
  - **`-Wall`** – check for warnings
  - **`-Wextra`** – check for even more warnings
  - **`-Werror`** – treat warnings as errors

- Additional flags (not comprehensive):
  - **`-S`** (assembly), **`-E`** (preprocess only), **`-c`** (compile only), **`-m32`** (compile for 32 bit architecture)

Hello World

```c
#include <stdio.h>

int main()
{

        printf("Hello World!\n");

        return 0;
}
```

Basic datatypes



Basic datatypes

Type	Storage size	Value range
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	8 bytes or (4bytes for 32 bit OS)	-9223372036854775808 to 9223372036854775807
unsigned long	8 bytes	0 to 18446744073709551615

Basic datatypes

Type	Storage size	Value range	Precision
float	4 byte	1.2E-38 to 3.4E+38	6 decimal places
double	8 byte	2.3E-308 to 1.7E+308	15 decimal places
long double	10 byte	3.4E-4932 to 1.1E+4932	19 decimal places

Variables and datatypes examples

```c
#include <stdio.h>

int integerVar = 42;
float floatVar = 3.14;

int main()
{

    double doubleVar = 2.71828;
    char charVar = 'A';


    printf("Integer: %d\n", integerVar);
    printf("Float: %f\n", floatVar);
    printf("Double: %lf\n", doubleVar);
    printf("Character: %c\n", charVar);

    return 0;

}
```

Specifiers

specifier	Output	Example
`d` *or* `i`	Signed decimal integer	`392`
`u`	Unsigned decimal integer	`7235`
`o`	Unsigned octal	`610`
`x`	Unsigned hexadecimal integer	`7fa`
`X`	Unsigned hexadecimal integer (uppercase)	`7FA`
`f`	Decimal floating point, lowercase	`392.65`
`F`	Decimal floating point, uppercase	`392.65`
`e`	Scientific notation (mantissa/exponent), lowercase	`3.9265e+2`
`E`	Scientific notation (mantissa/exponent), uppercase	`3.9265E+2`
`g`	Use the shortest representation: `%e` or `%f`	`392.65`
`G`	Use the shortest representation: `%E` or `%F`	`392.65`
`a`	Hexadecimal floating point, lowercase	`-0xc.90fep-2`
`A`	Hexadecimal floating point, uppercase	`-0XC.90FEP-2`
`c`	Character	`a`
`s`	String of characters	`sample`
`p`	Pointer address	`b8000000`
`n`	Nothing printed. The corresponding argument must be a pointer to a `signed int`. The number of characters written so far is stored in the pointed location.	
`%`	A `%` followed by another `%` character will write a single `%` to the stream.	`%`

Constant variables

- C has two distinct ways of creating constants
  - **`#define`** preprocessor directive
  - **`const`** keyword

- Modern programs programs prefer **`const`**
  - Better code readability
  - Easier to debug
  - Follow the scope bounds

Today’s lecture

- Setting up SSH keys
- VS Code remote development

SSH key generation

- AWS will generate a fresh key pair when you create a new instance
- You can add additional key pairs if you'd like to give other individuals access to your instance
- This generates a key pair, a public and a private key.
- The key must be in the **`~/.ssh/`** directory
  - For windows it should be **`C:\User\.ssh\`**
- The private ley must have correct permissions:
  - **`chmod 400 <path to private key>`**

VS Code remote development

- Press **Cmd (Ctrl for windows) + Shift + P**
- Search for Remote SSH add new agent
- Type in **ssh <username>@<host>** -- default username is ubuntu

- Open config file and add the following lines:

```
Host <IP of Instance>
    HostName <IP of Instance>
    User <username>
    IdentityFile <Path to private key> (~/.ssh/key_name.pem)
```

Today's lecture

- Continuation of C discussion
  - Standard libraries
  - Operators
  - Loops and Conditionals
  - Functions
  - Libraries
  - Scope

Standard libraries, and headers

- A header file contains function headers that allow your program to reference the library's functions
- The preprocessor combines these headers with your source program
- The linker hooks these references to the compiled object code of the library

Library Header File	Functions
`stdio.h`	Input and output, including file I/O
`stdlib.h`	Arithmetic, random numbers, terminating execution, sorting and searching
`math.h`	Advanced math (trig)
`string.h`	String processing

User input

- When we call **`scanf`** to read input from the user, we specify:
  - what type of data is being read,
  - a variable pointer into which we store the value.

```c
#include <stdio.h>

int main()
{
        int num;

        printf("Enter an integer: ");
        scanf("%d", &num);
        printf("You entered: %x\n", num);

return 0;
}
```

Operators in C

Operator symbol	Operation	Example usage
*	multiplication	x * y
/	division	x / y
%	integer remainder	x % y
+	addition	x + y
-	subtraction	x - y

Operator symbol	Operation	Example usage
~	bitwise NOT	~x
&	bitwise AND	x & y
|	bitwise OR	x | y
^	bitwise XOR	x ^y
«	left shift	x « y
»	right shift	x » y

Operator symbol	Operation	Example usage
>	greater than	x > y
>=	greater than or equal	x >= y
<	less than	x < y
<=	less than or equal	x <= y
==	equal	x == y
!=	not equal	x != y

Operator symbol	Operation	Example usage
!	logical NOT	!x
&&	logical AND	x && y
||	logical OR	x || y

Conditionals and loops

Conditional	`if (a < 1)`
	`{`
	`  // do something`
	`}`
	`else`
	`{`
	`  // do something else`
	`}`
		A condition determines which alternative to execute.

Iterative	`for (int i=1; i<10; i++)`
	`{`
	`  // iterate`
	`}`
		Repeats statement as long as a condition is true.

Class exercise

- Write a C program that calculate the area of a circle by prompting the user to input a radius value

- Your program should have the following requirements:
  - It must define pi as a constant with global scope.
  - Make sure to pick appropriate data types for your radius and area
  - Make use of the pow function to square the radius
  - If they user enters a negative radius value the program prints out an error and quits

  - To install GCC on your VMs run the following command

```
sudo apt update
sudo apt install gcc
Check gcc --version (Should be 11.04)
```

bit.ly/297-1

Today’s lecture

- Functions
- Scope
- Libraries

Functions

- Header: Everything before the first brace.
- Body: Everything between the braces.
- Type: Type of the value returned by the function.
- Parameter List: A list of identifiers that provide information for use within the body of the function.

```
type function_name ( parameter type list )
{
    declarations
    statements

    return type
}
```

Examples

```
long long factorial(int n)
{
    if(n == 0 || n == 1)
    {
    return 1;
    }

     return n * factorial(n-1);
}
```

```
bool isPrime(int n)
{
     if (n <= 1) {
          return false;
      }

     for(int i=2; i*i <= n; i++)
     {
          if (n % i == 0) {return false;}
     }
     return true;
}
```

Scope

- A block is code within enclosing braces { ... }
- A variable is declared within a block is local to that block, and is visible until the end of that block.
- Typically, a local variable will be declared at the beginning of a function block and used within that function.
- A variable declared outside any block is global. It is visible throughout the entire program.

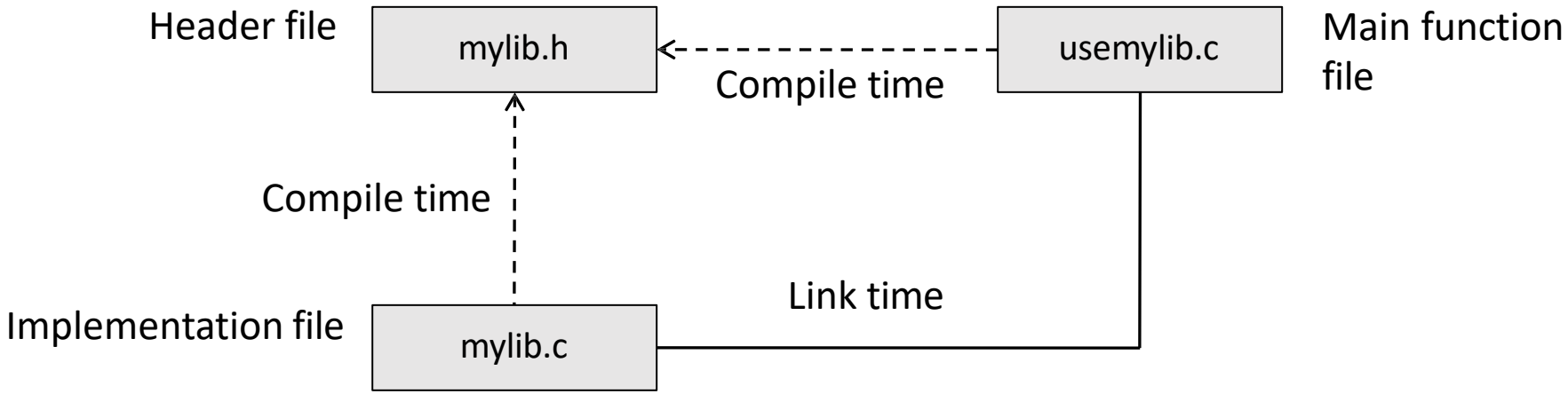
```
int count = 10;

int main(void)
{
   int echo;
   scanf("%d", &echo);

   for (int i =0; i<count; i++)
   {
       printf("%d\n", echo);
   }
}
```

Placement of functions

- We can put them all in one file with the application; prototypes are at the beginning of the file, and we work top down
- But this requires recompiling the whole file after one small change
- Work can’t be divided among many developers
- Functions can’t be maintained independently of their clients

Libraries

- Place the function prototypes for a module in a header file (.h extension)
- Place the completed function definitions for that module in an implementation file (.c)
- Compile the c. file, link it to a test driver, test it, and store it away



Separate compilation and linking

- Save each implementation file with a .c extension
- Work bottom up, compile each .c file
  - **`gcc -c <filename>.c`**
- If no errors, link the object files
  - **`gcc -o <main filename> <filename>.o ... <filename>.o`**
- If no errors, run the program with **`./<main filename>`**

An alternate approach – library archives

- An alternate approach is to create a library archive file (.a extension)
  - **`ar rcs <output_file> <list of object files>`**
- The convention is to prefix **`lib`** before the name of the library, and and the **`.a`** extension
- This can then be linked using the –l flag in gcc
  - **`gcc <main_file> -L<path_to_lib> -l<name_of_lib> -o <output_file>`**
  - When providing the name of the library, we do not need to provide the lib prefix or the .a extension

Makefiles

- We use make files when there are many modules to keep track of
  - Which ones are up to date, which ones need to be recompiled?
- The UNIX **`make`** tool and makefiles help to manage this problem
  - When you run make, UNIX looks for a file named **`makefile`** or **`Makefile`** in the current working directory
- A makefile contains info about the dependencies among program modules and commands to compile and/or link them

Makefile entries

```
targets: dependencies
    commands
```

A tab must go here

- Targets have specific dependencies
- Targets and dependencies are file names
- Commands are similar to those we've seen already e.g. (gcc ….)

Checks for timestamps

- Makefile also checks for timestamps files are modified
- Compare time stamps of files
- If target file is older than any dependency file then execute commands

Next class

- **Topic:** Pointers
- **Reading:** Chapter 5 of K&R

Today’s lecture

- Arrays
- Pointers

An alternate approach – library archives

- An alternate approach is to create a library archive file (.a extension)
  - **`ar rcs <output_file> <list of object files>`**
- The convention is to prefix **`lib`** before the name of the library, and and the **`.a`** extension
- This can then be linked using the –l flag in gcc
  - **`gcc <main_file> -L<path_to_lib> -l<name_of_lib> -o <output_file>`**
  - When providing the name of the library, we do not need to provide the lib prefix or the .a extension

Arrays

- A linear sequence of memory cells, each accessible by an an integer index, ranging from 0 to the number of cells minus 1
- Like Python lists and Java array lists, C arrays support random (constant-time) access to the cells

Declaring array variables

```
int integers[10]; // Cells for 10 ints
char letters[5]; // Cells for 5 characters
float matrix[3][2]; // A 3 by 2 matrix of floats

int max = 100; // The number of cells in the next array
double doubles[max]; // Cells for max doubles
```

<element type> <variable> [<integer expression>]

- Declares an array variable and reserves memory for a definite number of cells
  - use more [] for more dimensions
- Each of which can store an element of the given type (arrays are statically typed)
- Each cell initially contains garbage!

Accessing array values

```
int i, max = 10, array[max] //Declare variables

for (i = 0; i < max; i++)
       array[i] = i + 1; //Store 1..max in the array

for (i = 0; i < max; i++)
       printf("%d\n", array[i]); //Print the contents
```

- Here we initialize all of the array's cells and visit them all
- The loop is pretty standard

Arrays and functions

```
int minIndex(int array[], int length)
{
    int probe, minPos = 0;

    for (probe = 1; probe < length; probe++)
    {
        if (array[probe] < array[minPos])
                minPos = probe;
    }

   return minPos;
```

- An array can be passed as an argument to a function
- The function can access or replace values in the array cells

Class exercise

```
int number, length = 0, max = 10, array[max];

printf("Enter a number or 0 to stop: ");
scanf("%d", &number);

while (number)
{
        array[length] = number;
        length++;

        printf("Enter a number or 0 to stop: ");
        scanf("%d", &number);
}
```

Observe this code, and identify any errors/potential correctness issues.

bit.ly/csci297-f23-2

Parameter passing

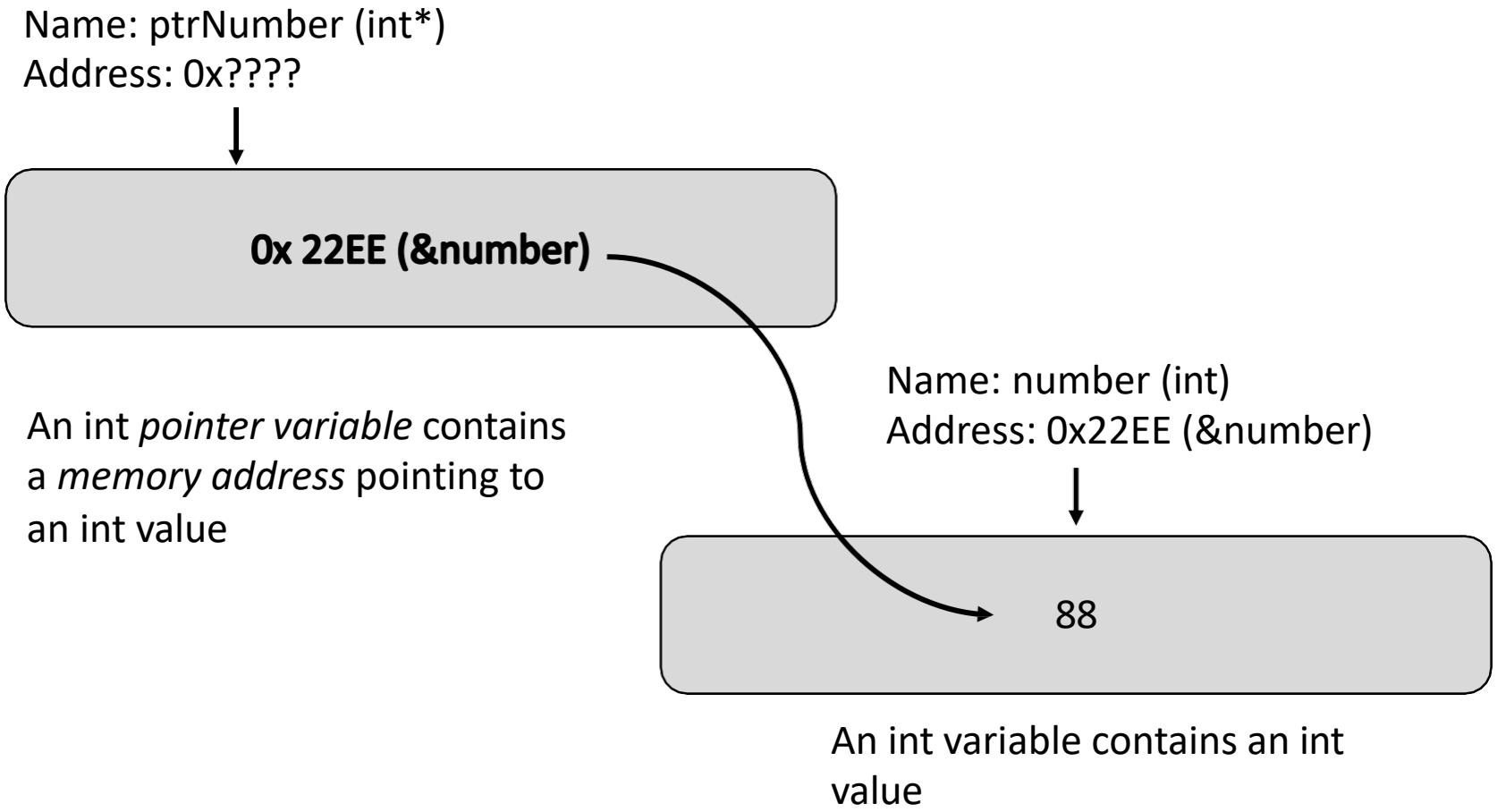
- Parameters of basic types (char, int, float, and double) are passed by value (a copy of the value of the argument is placed in temporary storage on the runtime stack)

- A copy of an array's base address is also placed in temporary storage, so its cells can still be accessed or modified

Pointers

- C allows the programmer to access the address of a memory location and use it as a data value
- Pointer = address of a variable (or any memory location)
  - Allows indirect access to variables – for example, allow a function to change a variable in the caller.
  - Pointer-based data structures that can grow or shrink to meet needs.
  - Pointers are also used to refer to dynamically allocated storage (from the system heap)

Visualization



Declaring pointers

- C allows us to declare a variable that holds a pointer value.
- We need to specify the type of data to which the pointer points to.
- Add `*` operator to the type name to declare a pointer to that type.
  - `int * P;`
  - `float *Q;`
  - `int *A[5];`

- P is a pointer of type int, Q is a pointer of type float, A is an array of 5 pointers of int
- **Common Confusion:** The use of whitespaces in declaration is irrelevant

The address operator

- The `&` operator can be used before any variable to get its memory address

```c
int main()
{
    int V = 5;

    printf ("The value of V: %d\n", V);
    printf ("The address of V: %p\n", &V);

}
```

Assigning pointer values

```
int * P; //declare a pointer P of type int, int V = 5;

printf ("The value of V: %d\n", V);

printf ("The address of V: %p\n", &V);

P = &V; //Pointer now points to the memory location of V

printf ("The memory location P points to: %p\n", P);
```

Dereferencing pointers

- The dereference operator (*) is used to obtain the value of the variable a pointer points to.
- Done by adding * before the pointer variable

```
int * P;
int V = 5;
P = &V;

printf ("The value stored at P: %d\n", *P);
```

- Common Confusion: The dereference operator (*) is different from the asterisk used to declare a pointer.

Next class

- **Topic:** Dynamic memory / Heap
- **Reading:** Chapter 5 of K&R

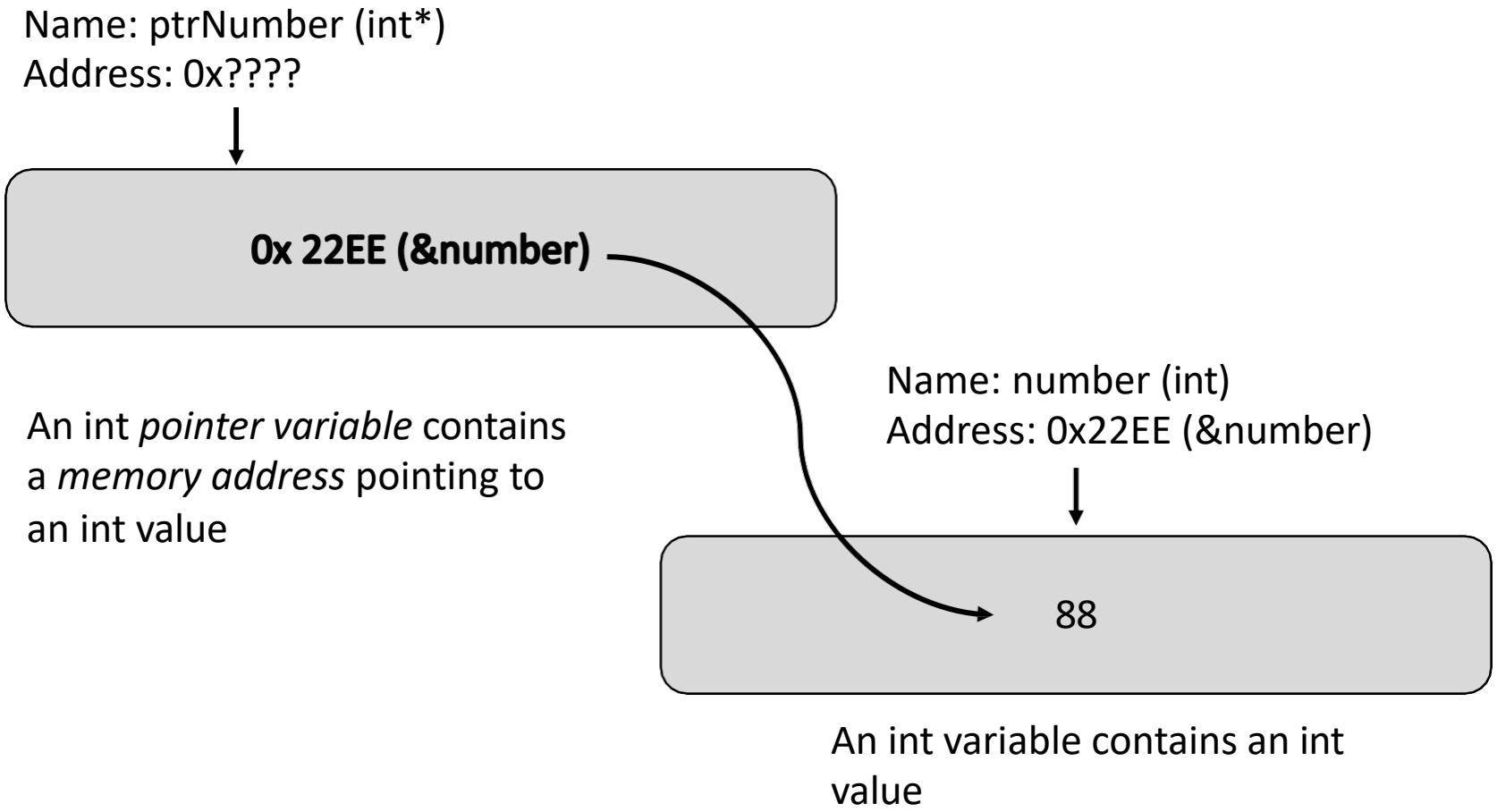
Today's lecture

- Assignment 1
- Pointers (Recap)
- Dynamic Memory

Pointers

- C allows the programmer to access the address of a memory location and use it as a data value
- Pointer = address of a variable (or any memory location)
  - Allows indirect access to variables – for example, allow a function to change a variable in the caller.
  - Pointer-based data structures that can grow or shrink to meet needs.
  - Pointers are also used to refer to dynamically allocated storage (from the system heap)

Visualization



Declaring pointers

- C allows us to declare a variable that holds a pointer value.
- We need to specify the type of data to which the pointer points to.
- Add `*` operator to the type name to declare a pointer to that type.
  - `int * P;`
  - `float *Q;`
  - `int *A[5];`

- P is a pointer of type int, Q is a pointer of type float, A is an array of 5 pointers of int
- **Common Confusion:** The use of whitespaces in declaration is irrelevant

The address operator

- The `&` operator can be used before any variable to get its memory address

```c
int main()
{
    int V = 5;

    printf ("The value of V: %d\n", V);
    printf ("The address of V: %p\n", &V);

}
```

Assigning pointer values

```
int * P; //declare a pointer P of type int, int V = 5;

printf ("The value of V: %d\n", V);

printf ("The address of V: %p\n", &V);

P = &V; //Pointer now points to the memory location of V

printf ("The memory location P points to: %p\n", P);
```

Dereferencing pointers

- The dereference operator (*) is used to obtain the value of the variable a pointer points to.
- Done by adding * before the pointer variable

```
int * P;
int V = 5;
P = &V;

printf ("The value stored at P: %d\n", *P);
```

- Common Confusion: The dereference operator (*) is different from the asterisk used to declare a pointer.

NULL Pointer

- Sometimes we want to create a pointer that points to nothing.
- This is known as a null pointer
  - `int *ptr = NULL;`

- Why would we want to do this?
- Initialize unused pointers to NULL. If you don't initialize, the pointer will point to a random memory location, which can lead to strange and interesting bugs.

Pointers and functions

```c
#include <stdio.h>

void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main()
 {
    int x = 10, y = 20;

    printf("Before swap: x = %d, y = %d\n", x, y);
    swap(&x, &y);
    printf("After swap: x = %d, y = %d\n", x, y);

   return 0;
}
```

Dynamic memory

- C allows the program to allocate memory in real time on the heap
- These memory locations are referenced by pointer
- Benefits
  - No need to decide beforehand the size of data structures e.g (arrays)
  - Allows working with variable input data
  - Also efficient and saves memory as we only allocate only what is needed.
- C functions for dynamic memory allocation:
  - `malloc` allocates a block of storage from the heap
  - `free` returns storage to the heap

Malloc

- Allocates requested number of bytes
- Returns a void pointer to the first byte of the allocated space
- Malloc allocates a contiguous chunk of memory
  - `ptr= (int*) malloc(100 *sizeof(int));`
- int is of 4 bytes, so the total bytes allocated are 400

Free

- Use the free command to free the allocated memory
  - `free(ptr);`
- With C we have to manually clear memory we allocate
- Question: What if we don't free the memory?

Next class

- **Topic:** File I/O and Structs
- **Reading:** Chapter 6 and 7 of K&R

Today’s lecture

- Dynamic Memory

Dynamic memory

- C allows the program to allocate memory in real time on the heap
- These memory locations are referenced by pointer
- Benefits
  - No need to decide beforehand the size of data structures e.g (arrays)
  - Allows working with variable input data
  - Also efficient and saves memory as we only allocate only what is needed.
- C functions for dynamic memory allocation:
  - `malloc` allocates a block of storage from the heap
  - `free` returns storage to the heap

Malloc

- Allocates requested number of bytes
- Returns a void pointer to the first byte of the allocated space
- Malloc allocates a contiguous chunk of memory
  - `ptr= (int*) malloc(100 *sizeof(int));`
- int is of 4 bytes, so the total bytes allocated are 400

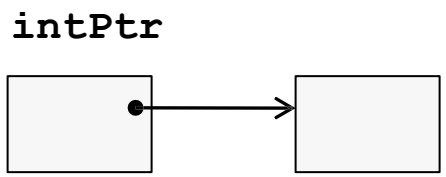
Free

- Use the free command to free the allocated memory
  - `free(ptr);`
- With C we have to manually clear memory we allocate
- Question: What if we don't free the memory?

Declare and Initialize with malloc

- **`malloc`** expects an integer representing the size of the storage needed
- The **`sizeof`** function takes a type as an argument and returns the number of bytes needed for a value of that type
- **`malloc`** returns a pointer to the first byte of the storage

```
#include <stdlib.h>

int *intPtr = malloc(sizeof(int));
```

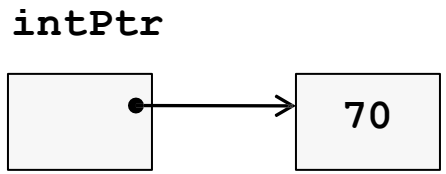


The variable now points to a new storage area for an **`int`**, but this storage contains garbage

Access via Dereference with *

```
#include <stdlib.h>

int *intPtr = malloc(sizeof(int));
(*intPtr) = 70;
printf("%d\n", *intPtr);
```

- The * operator accesses the cell pointed to
- The pointer must point to a cell
- * has a lower precedence than =

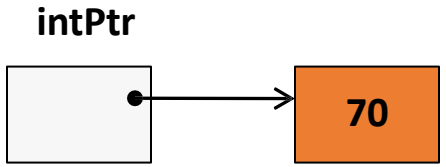


The variable now points to storage that has a program-supplied value

Return the Storage with free

```
#include <stdlib.h>

int *intPtr = malloc(sizeof(int));
(*intPtr) = 70;
printf("%d\n", *intPtr);
free(intPtr);
```

- The storage is still accessible, but might be taken for other applications (also called a dangling pointer reference)



The variable still points to storage, which might be used by others

Reset the Pointer to NULL

```
#include <stdlib.h>

int *intPtr = malloc(sizeof(int));
(*intPtr) = 70;
printf("%d\n", *intPtr);
free(intPtr);
intPtr = NULL;
```

- Clean up by setting the pointer to **NULL** or to other storage

The variable now has no dangling pointer

Dynamic memory as arrays

```c
int *arr = (int*)malloc(10 * sizeof(int));


// Populate the array with numbers 1 through 10.
for (int i = 0; i < 10; i++)
{
        arr[i] = i + 1;
}

int *temp = (int*)realloc(arr, 20 * sizeof(int));

// Check if the memory reallocation was successful.
if (temp == NULL) {return 1;}

arr = temp; // Update the pointer to the new memory location.

// Don't forget to free the memory when done.
free(arr);

return 0;
```

Structs

- An issue with arrays is you can only store one data type.
- C supports data structures called `structs` that can support a combination of data types such as `ints, floats` and `char` types.
- A `struct` is a data type that includes one or more named data fields
- Good for containing data of different types
- Like classes in Python or Java, but without the methods

Typedef

- C allows in you to give data types custom names or aliases
- This is done using the typedef keyword. Syntax:
- MYINT is the new alias. It is convention to use uppercase letters, but lowercase is allowed
- typedef can also be used with custom data types (eg. Structs)

```
typedef int typedef int MYINT;

MYINT a; // a is of an integer type MYINT

b[10] // array of 10 ints
```

Next class

- **Topic:** Valgrind and GDB – Bring Laptops

Today's lecture

- Structs wrap-up
- Valgrind

Struct Example

```c
// Define a struct to hold a dynamic array and its size

typedef struct {

int *array; // Pointer to the first element of the dynamic array

int size, count; // Number of elements in the array

} DynamicArray;
```

Initializing the array struct

```c
// Function to initialize the dynamic array with a given size
DynamicArray initializeArray(int size) {

    DynamicArray arr;

    arr.array = (int *)malloc(size * sizeof(int));

    if (arr.array == NULL)
    {
        fprintf(stderr, "Memory allocation failed!\n");
        exit(EXIT_FAILURE);
    }
    arr.size = size;
    arr.count = 0; // Initialize the count to zero

    return arr;
}
```

Adding an element

```c
// Function to add an element to the dynamic array
int addElement(DynamicArray *arr, int element) {

    if (arr->count < arr->size) {

        arr->array[arr->count] = element;
        arr->count++;

        return 0; // Success
    }
    return -1; // Failure: array is full
}
```

Freeing the array

```c
// Function to free the dynamic array
void freeArray(DynamicArray *arr) {

    free(arr->array);

    arr->array = NULL;

    arr->size = 0;

    arr->count = 0;
}
```

Main function argument vector

- The argument vector is an array of strings that are gathered up when a C program is run at the command prompt
- The first string is the name of the command
- The remaining strings are the arguments, if any
- The argument vector and its length are optional formal parameters of the main function

Example

```c
/*
Prints contents of the argument vector (the name of the program
and any command-line arguments).
*/

#include <stdio.h>

int main(int length, char *argVec[])
{
    int i;
    for (i = 0; i < length; i++)
    printf("Argument %d: %s\n", i, argVec[i]);
}
```

Valgrind

- Memory management, leak detection, and analysis tool.
- Detect memory leaks and errors in C/C++ programs.
- Valgrind flags errors that don't appear without valgrind

- Installing Valgrind:

```
sudo apt-get install valgrind
valgrind --version
```

- Running Valgrind
  - `valgrind ./program`

Valgrind

- When to use?
  - Dealing with memory (especially dynamic memory allocation)
  - Whenever bugs occur. Get instant feedback about what the bug is, where it occurred, and why.

- When not to use?
  - Program contains no invalid reads and writes and no leaked memory
  - If the test case is inherently slow, then this is not a good choice

Example run

```c
#include <stdlib.h>

int main() {
        int *ptr = malloc(sizeof(int));
return 0;
}
```

```
ubuntu@ip-172-31-29-123:~/vg$ valgrind ./a.out
==2081== Memcheck, a memory error detector
==2081== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==2081== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==2081== Command: ./a.out
==2081==
==2081==
==2081== HEAP SUMMARY:
==2081==     in use at exit: 4 bytes in 1 blocks
==2081==   total heap usage: 1 allocs, 0 frees, 4 bytes allocated
==2081==
==2081== LEAK SUMMARY:
==2081==    definitely lost: 4 bytes in 1 blocks
==2081==    indirectly lost: 0 bytes in 0 blocks
==2081==      possibly lost: 0 bytes in 0 blocks
==2081==    still reachable: 0 bytes in 0 blocks
==2081==         suppressed: 0 bytes in 0 blocks
==2081== Rerun with --leak-check=full to see details of leaked memory
==2081==
==2081== For lists of detected and suppressed errors, rerun with: -s
==2081== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Recommended flags

- Recommended flags

```
valgrind --leak-resolution=high --leak-check=full --show-reachable=yes --
track-fds=yes -track-origins=yes ./myProgram
```

- Feel free to lookup **`man valgrind`**

Types of memory leaks

- Still Reachable
  - Block is still pointed at, programmer could go back and free it before exiting
- Definitely Lost
  - No pointer to the block can be found
- Indirectly Lost
  - Block is “lost” because the blocks that point to it are themselves lost
- Possibly Lost
  - Pointer exists but it points to an internal part of the memory block

Class exercise

- View the code provided in slack, run is using valgrind and post your observations at this link:

https://bit.ly/csci297-f23-3

Today's lecture

- File I/O
- Introduction to linking

Interacting with files

```c
#include <stdio.h>

int main(){
    FILE *infile = fopen("test_file.c", "r");

    FILE *outfile = fopen("test.bin", "w");

    if (infile == NULL)
        {printf("The input file does not exist.\n");}
    else
    {
        processFile(infile, outfile);
        fclose(infile);
        fclose(outfile);
    }
}
```

File I/O functions

Function	What it does
`char* fgets(char* array, int arraySize, FILE *infile)`	Returns **`NULL`** if end of file (**`EOF`**). Otherwise, loads the next line, including the newline and **`null`** characters, into **`array`**. ***Note***: there must be storage for **`array`**!
`int fputs(char* array, FILE *outfile)`	Returns **`EOF`** if the string cannot be written to **`outfile`**. Otherwise, writes the string to **`outfile`**. A newline is not automatically written.
`fprintf(FILE *outfile, char* formatString, arg1, . . ., argn)`	Works just like **`printf`** with the standard output file (the terminal)

stdin and stdout

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

int main() {
      char filename[256]; // Buffer to hold the filename
      // Prompt the user to enter the filename

      printf("Enter the filename: ");
      if (fgets(filename, sizeof(filename), stdin) == NULL)
      {
            perror("Error reading filename");
            return 1;
      }

      // Open the file in read mode
      FILE *file = fopen(filename, "r");

      char buffer[256]; // Buffer to hold lines read from the file
      // Read lines from the file and write them to stdout

      while (fgets(buffer, sizeof(buffer), file) != NULL)
      {
            fputs(buffer, stdout);
      }

      // Close the file after reading
      fclose(file);

      return 0;
}
```

Example C Program

main.c

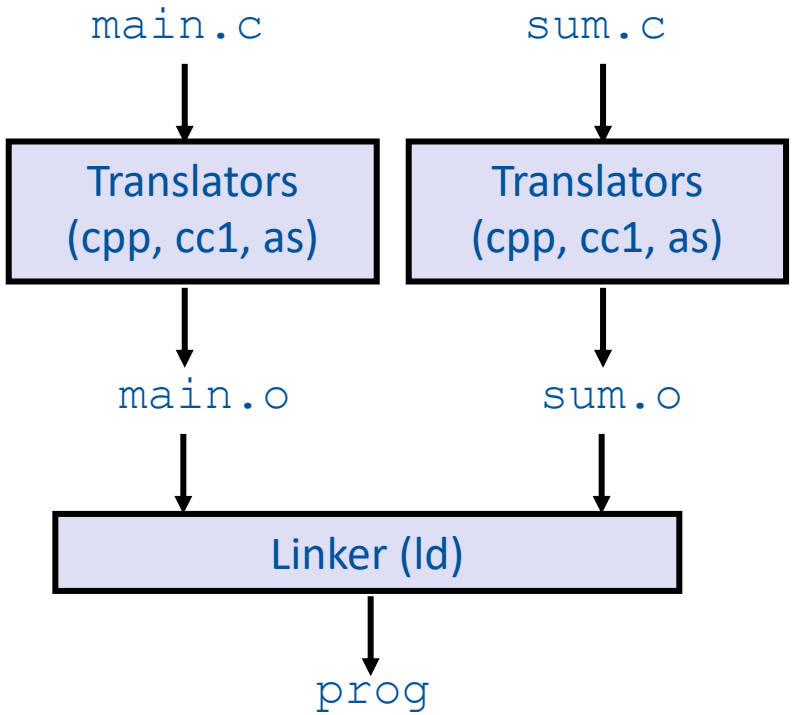
```c
int sum(int *a, int n);

int array[2] = {1, 2};


int main(int argc, char** argv)
{

    int val = sum(array, 2);

    return val;
}
```

sum.c

```c
int sum(int *a, int n)
{
        int i, s = 0;

        for (i = 0; i < n; i++)
        {
                s += a[i];
        }

        return s;
}
```

Linking

- Programs are translated and linked using a *compiler driver*:
  - **`linux>` *`gcc -Og -o prog main.c sum.c`***
  - **`linux>` *`./prog`***



Source files

Separately compiled
relocatable object files

Fully linked executable object file
(contains code and data for all functions
defined in main.c and sum.c)

Next Class:

- The ELF format and relocation
- Readings: 7.1 -7.7

Today’s lecture

- Linking steps
- The ELF format

Why use linkers?

- Reason 1: Modularity
  - Program can be written as a collection of smaller source files, rather than one monolithic mass.
  - Can build libraries of common functions

- Reason 2: Time Efficiency
  - Time: Separate compilation
  - Change one source file, compile, and then relink
  - No need to recompile other source files, and can compile multiple files concurrently

Why use linkers?

- Reason 3: Space Efficiency
  - Common functions can be aggregated into a single file
  - Option 1: Static Linking
    - Executable files and running memory images contain only the library code they actually use
  - Option 2: Dynamic Linking
    - Executable files contain no library code
    - During execution, single copy of library code can be shared across all executing processes

Linker steps

- Step 1: Symbol resolution
  - Programs define and reference symbols (globals, functions, and some locals)
    - `void swap() {…}  /* define symbol swap */`
    - `swap();  /* reference symbol swap */`
    - `int *xp = &x;  /* define symbol xp, reference x */`
  - Symbol definitions are stored in object file (by assembler) in *symbol table*.
    - Symbol table is an array of entries
    - Each entry includes name, size, and location of symbol.
  - **During symbol resolution step, the linker associates each symbol reference with exactly one symbol definition.**

Identify the symbols

main.c

```c
int sum(int *a, int n);

int array[2] = {1, 2};


int main(int argc, char** argv)
{

    int val = sum(array, 2);

    return val;
}
```

sum.c

```c
int sum(int *a, int n)
{
        int i, s = 0;

        for (i = 0; i < n; i++)
        {
                s += a[i];
        }

        return s;
}
```

Linker steps

- Step 2: Relocation
  - Merges separate code and data sections into single sections
  - Relocates symbols from their relative locations in the `.o` files to their final absolute memory locations in the executable.
  - Updates all references to these symbols to reflect their new positions.

Three kinds of object files

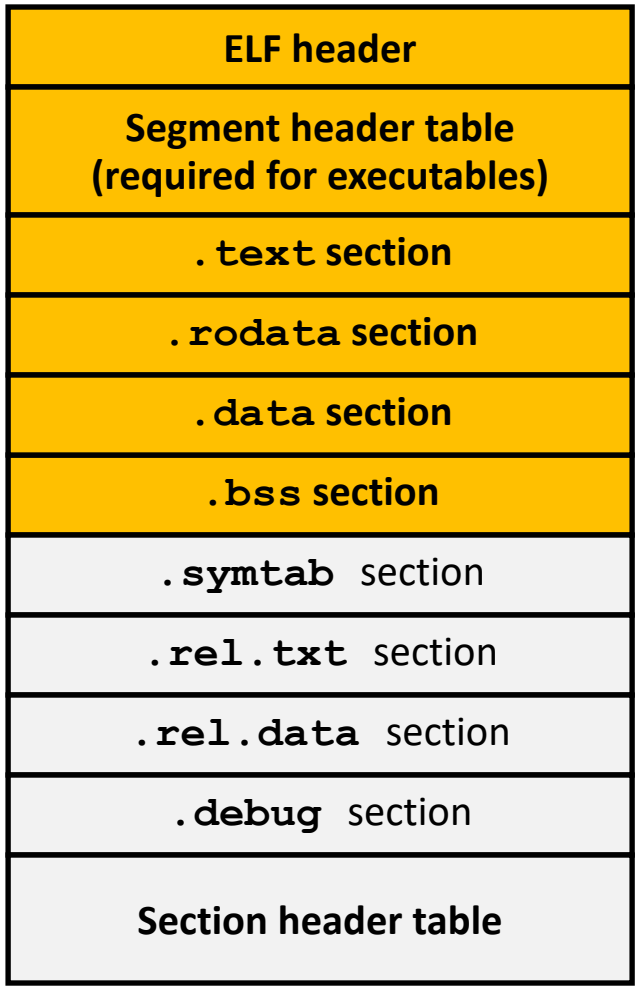
- Relocatable object file (.o file)
  - Contains code and data in a form that can be combined with other relocatable object files to form executable file.
  - Each .o file is produced from exactly one source (.c) file
- Executable object file (a.out file)
  - Contains code and data in a form that can be copied directly into memory and then executed.
- Shared object file (.so file)
  - Special type of relocatable object file that can be loaded into memory and linked dynamically, at either load time or run-time.
  - Called Dynamic Link Libraries (DLLs) by Windows

Executable and linkable format

- Standard binary format for object files
- One unified format for
  - Relocatable object files (.o),
  - Executable object files (a.out)
  - Shared object files (.so)
- Generic name: ELF binaries

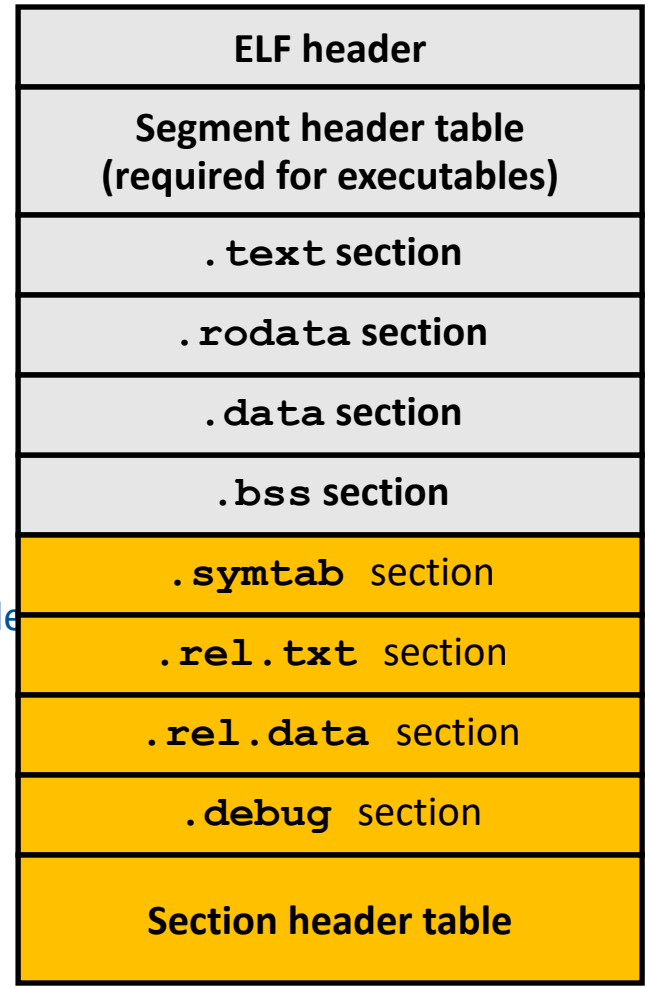
ELF object file format

- **Elf header**
  - Word size, byte ordering, file type (.o, exec, .so), machine type, etc.
- **Segment header table**
  - Page size, virtual address memory segments (sections), segment sizes.
- `.text` section
  - Code
- `.rodata` section
  - Read only data: jump tables, string constants, ...
- `.data` section
  - Initialized global variables
- `.bss` section
  - Uninitialized global variables
  - “Block Started by Symbol”
  - “Better Save Space”
  - Has section header but occupies no space



ELF object file format

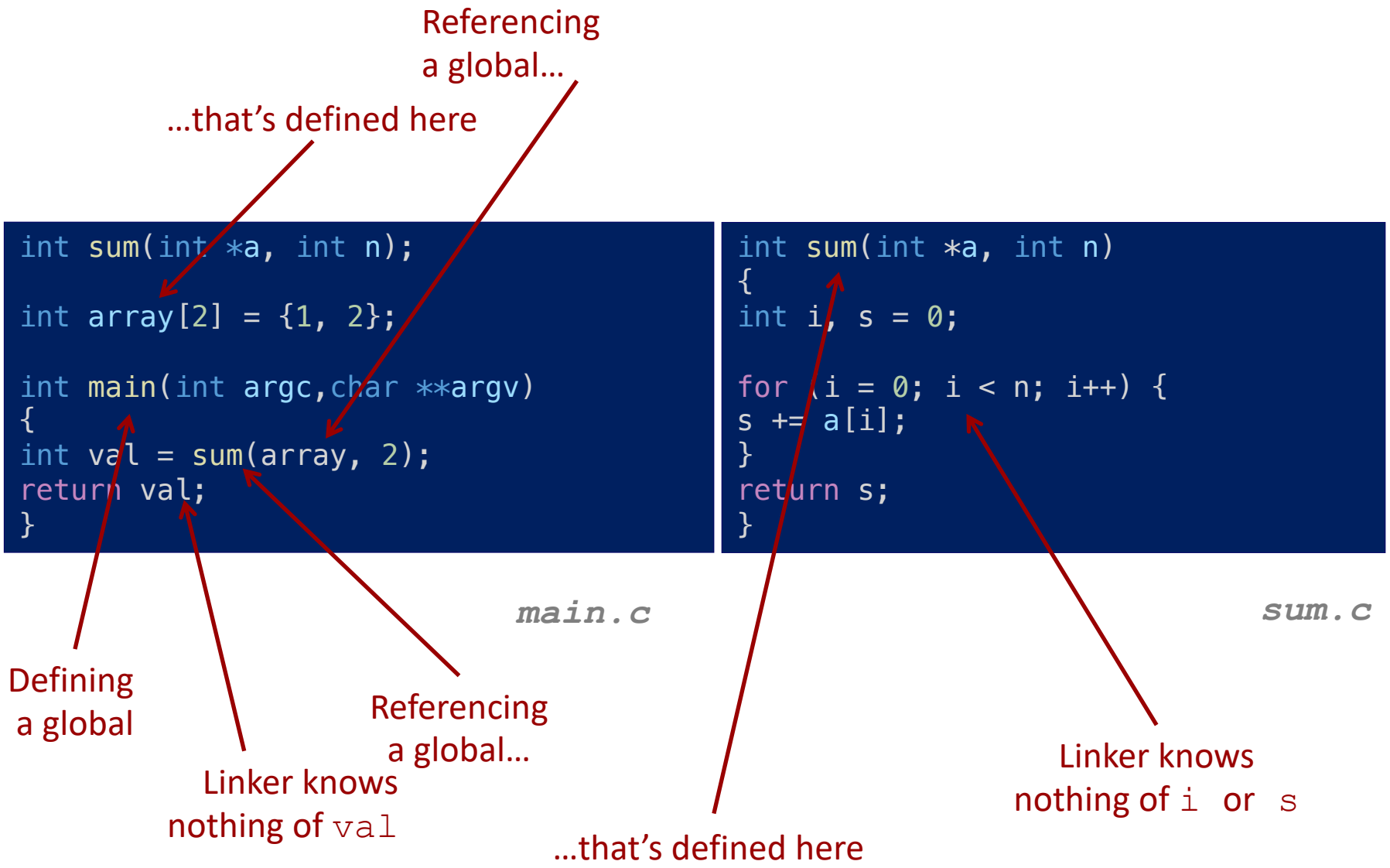
- `.symtab` section
  - Symbol table
  - Procedure and static variable names
  - Section names and locations
- `.rel.text` section
  - Relocation info for `.text` section
  - Addresses of instructions that will need to be modified in the executable
  - Instructions for modifying
- `.rel.data` section
  - Relocation info for `.data` section
  - Addresses of pointer data that will need to be modified in the merged executable
- `.debug` section
  - Info for symbolic debugging (`gcc -g`)
- **Section header table**
  - Offsets and sizes of each section



Linker symbols

- Global symbols
  - Symbols defined by module m that can be referenced by other modules.
  - e.g., non-static C functions and non-static global variables.
- External symbols
  - Global symbols that are referenced by module m but defined by some other module.
- Local symbols
  - Symbols that are defined and referenced exclusively by module m.
  - e.g, C functions and global variables defined with the static attribute, and local static variables
  - **Local linker symbols are not local program variables**

Symbol identification



Symbol identification

- Which of the following names will be in the symbol table of symbols.o?

```c
int incr = 1;
static int foo(int a)
{
    int b = a + incr;
    return b;
}

int main(int argc, char* argv[])
{
        printf("%d\n", foo(5));
        return 0;
}
```

- **incr**
- **foo**
- **a**
- **b**
- **argc**
- **argv**
- **main**
- **printf**
- **"%d\n"**

Local symbols

- Local non-static C variables vs. local static C variables
  - Local non-static C variables: stored on the stack
  - Local static C variables: stored in either **`.bss`** or **`.data`**

```
static int x = 15;

int f() {
    static int x = 17;
    return x++;
}

int g() {
    static int x = 19;
    return x += 14;
}
```

Compiler allocates space in **`.data`** for each definition of `x`

Creates local symbols in the symbol table with unique names, e.g., `x`, `x.1721` and `x.1724`.

Class exercise

```
int time;

int foo(int x, int y);

int bar(int a) {
        int b = a + 1;
        return b;
}

int main() {
        printf("%d", bar(5));
}
```

How many entries do we expect to see in the symbol table?

A. 1
B. 2
C. 3
D. 4
E. > 4

https://bit.ly/csci-297-f23-4

Next Class:

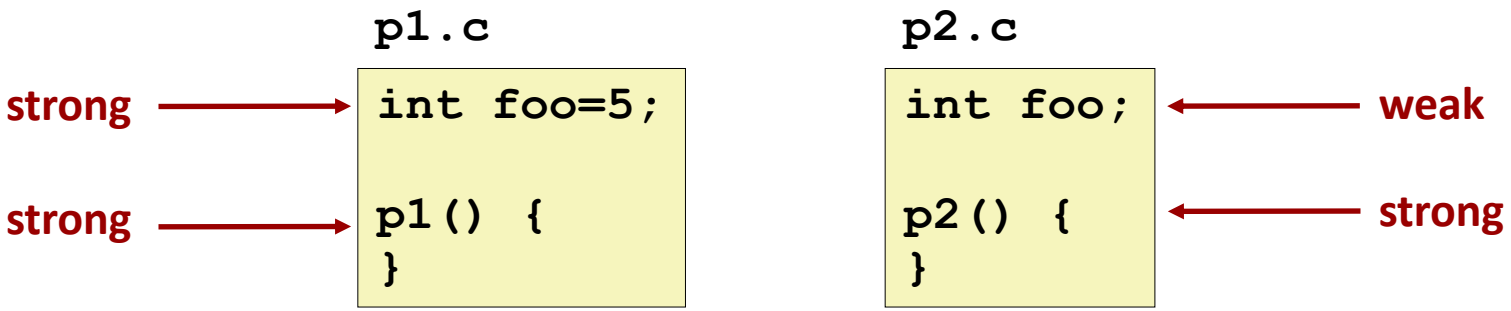
- Symbols resolution and relocation
- Readings: 7.1 -7.7

Today's lecture

- Symbol resolution
- Linking with static libraries

Duplicate symbols

- Program symbols are either strong or weak
  - Strong: procedures and initialized globals
  - Weak: uninitialized globals, or ones declared with specifier **`extern`**



Symbol rules

- Rule 1: Multiple strong symbols are not allowed
  - Each item can be defined only once
  - Otherwise: Linker error
- Rule 2: Given a strong symbol and multiple weak symbols, choose the strong symbol
  - References to the weak symbol resolve to the strong symbol
- Rule 3: If there are multiple weak symbols, pick an arbitrary one
- DEMO!

Practice questions

```
int x;
p1() {}
```

```
p1() {}
```

Link time error: two strong symbols (p1)

```
int x;
p1() {}
```

```
int x;
p2() {}
```

References to x will refer to the same uninitialized int. Is this what you really want?

```
int x;
int y;
p1() {}
```

```
double x;
p2() {}
```

Writes to x in p2 might overwrite y!
Evil!

```
int x=7;
int y=5;
p1() {}
```

```
double x;
p2() {}
```

Writes to x in p2 might overwrite y!
Nasty!

```
int x=7;
p1() {}
```

```
int x;
p2() {}
```

References to x will refer to the same initialized variable.

Global variables

- Avoid if you can

- Otherwise
  - Use **`static`** if you can
  - Initialize if you define a global variable
  - Use **`extern`** if you reference an external global variable
    - Treated as weak symbol
    - But also causes linker error if not defined in some file

Use of extern

c1.c

```
#include "global.h"

int f() {
  return g+1;
}
```

global.h

```
extern int g;
int f();
```

c2.c

```
#include <stdio.h>
#include "global.h”

int g = 0;

int main(int argc, char argv[]) {
  int t = f();
  printf("Calling f yields %d\n", t);
  return 0;
}
```

Packaging functions

- How to package functions commonly used by programmers?
  - Math, I/O, memory management, string manipulation, etc.

- Awkward, given the linker framework so far:
  - **Option 1:** Put all functions into a single source file
    - Programmers link big object file into their programs
    - Space and time inefficient
  - **Option 2:** Put each function in a separate source file
    - Programmers explicitly link appropriate binaries into their programs
    - More efficient, but burdensome on the programmer

Static libraries

- Static libraries (`.a` archive files)
  - Concatenate related relocatable object files into a single file with an index (called an *archive*).
  - Enhance linker so that it tries to resolve unresolved external references by looking for the symbols in one or more archives.
  - If an archive member file resolves reference, link it into the executable.

Next Class

- Relocation, Dynamic libraries
- **Readings:** 7.7 – 7.14

Today's lecture

- Linking with static libraries
- Relocation
- Executables

Packaging functions

- How to package functions commonly used by programmers?
  - Math, I/O, memory management, string manipulation, etc.

- Awkward, given the linker framework so far:
  - **Option 1:** Put all functions into a single source file
    - Programmers link big object file into their programs
    - Space and time inefficient
  - **Option 2:** Put each function in a separate source file
    - Programmers explicitly link appropriate binaries into their programs
    - More efficient, but burdensome on the programmer

Static libraries

- Static libraries (`.a` archive files)
  - Concatenate related relocatable object files into a single file with an index (called an *archive*).
  - Enhance linker so that it tries to resolve unresolved external references by looking for the symbols in one or more archives.
  - If an archive member file resolves reference, link it into the executable.

Common C libraries

`libc.a` (the C standard library)

- 4.6 MB archive of 1496 object files.
- I/O, memory allocation, signal handling, string handling, data and time, random numbers, integer math

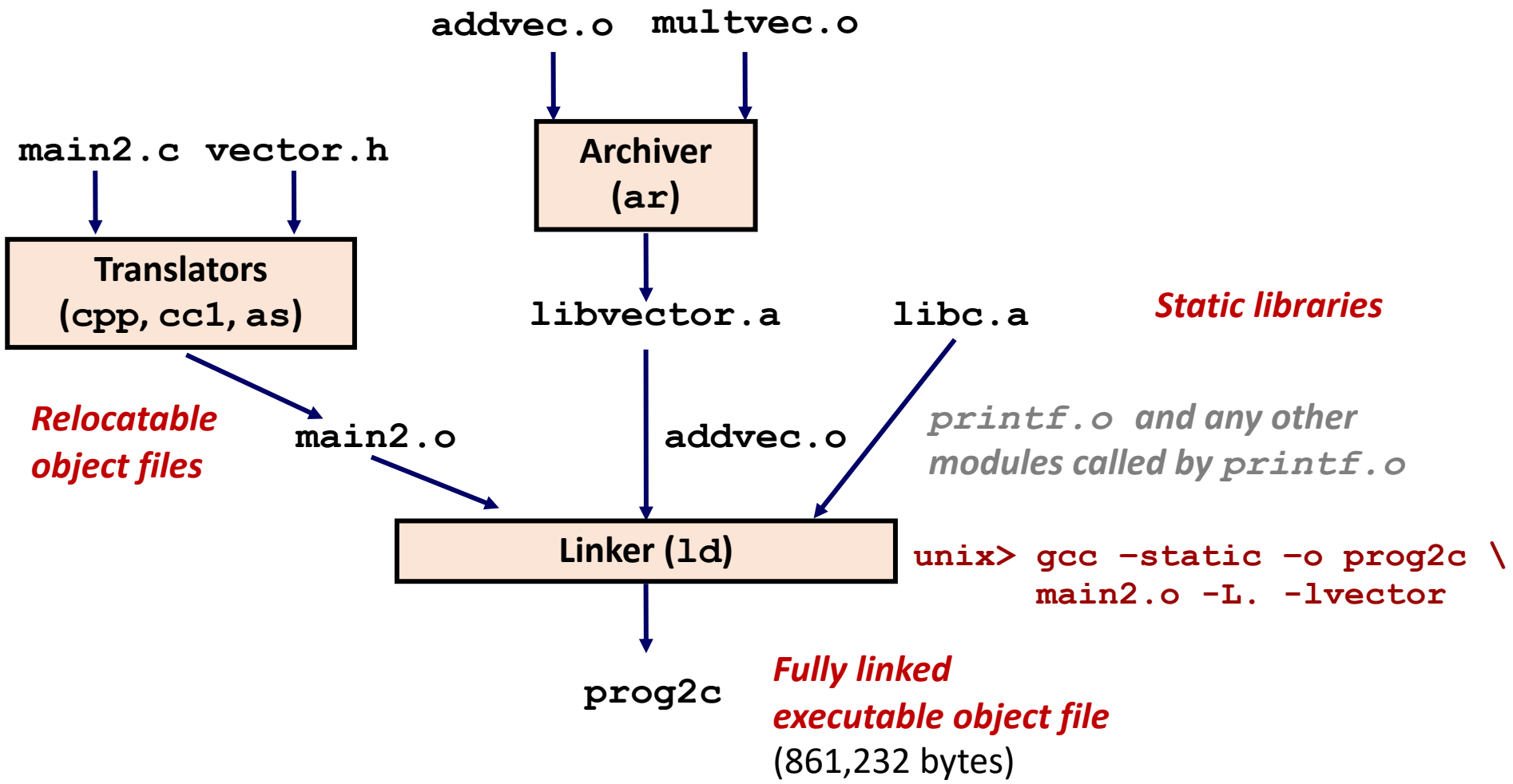
`libm.a` (the C math library)

- 2 MB archive of 444 object files.
- floating point math (sin, cos, tan, log, exp, sqrt, …)

```
% ar -t /usr/lib/libc.a | sort
…
fork.o
…
fprintf.o
fpu_control.o
fputc.o
freopen.o
fstab.o
…
```

```
% ar -t /usr/lib/libm.a | sort
…
e_acos.o
e_acosf.o
e_acosh.o
e_acoshf.o
e_acoshl.o
e_asinf.o
e_asinl.o
…
```

Linking with static libraries



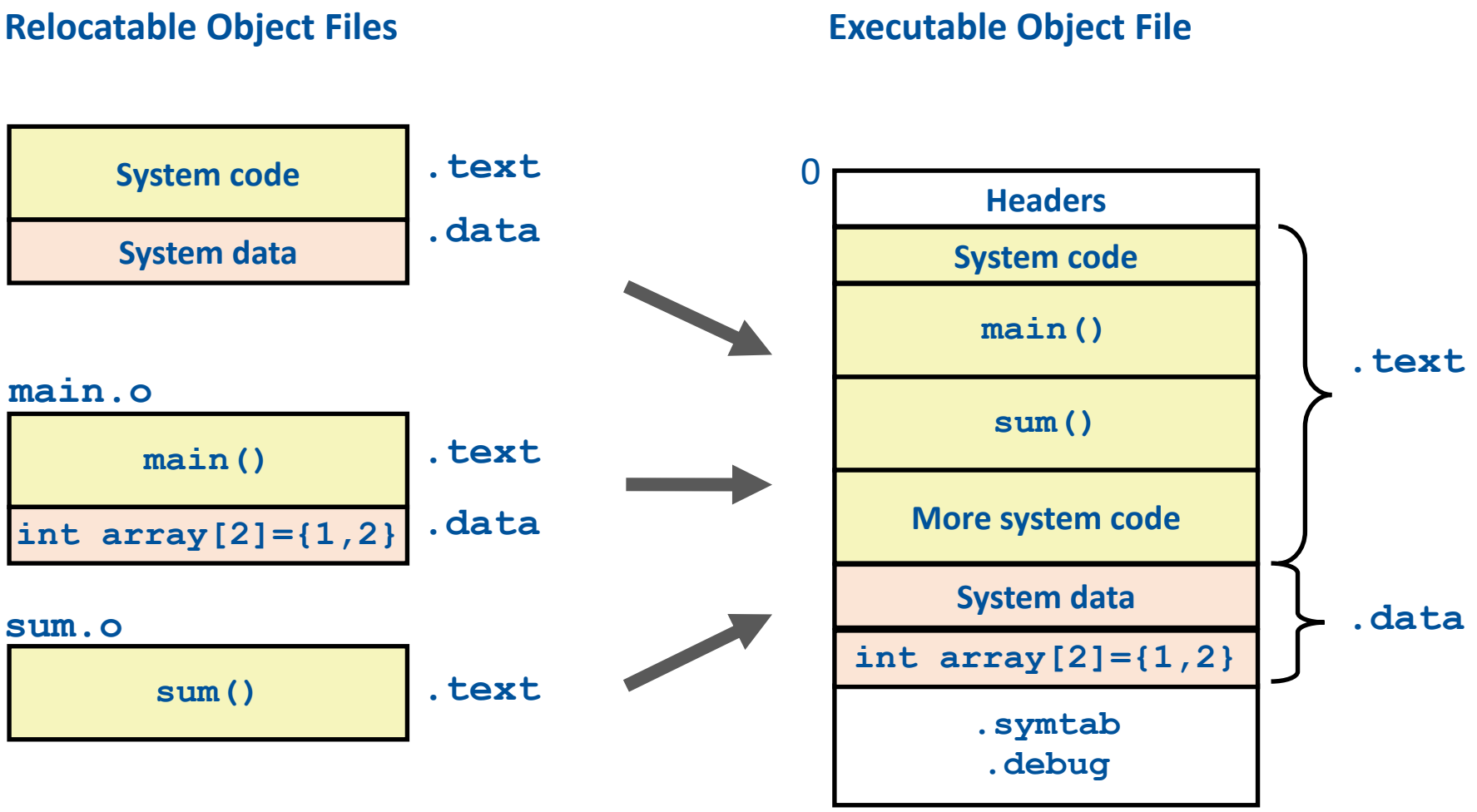
Static libraries algorithm

- Linker's algorithm for resolving external references:
  - Scan `.o` files and `.a` files in the command line order.
  - During the scan, keep a list of the current unresolved references.
  - As each new `.o` or `.a` file, *obj*, is encountered, try to resolve each unresolved reference in the list against the symbols defined in *obj*.
  - If any entries in the unresolved list at end of scan, then error.

```
unix> gcc -static -o prog2c -L. -lvector main2.o
main2.o: In function `main':
main2.c:(.text+0x19): undefined reference to `addvec'
collect2: error: ld returned 1 exit status
```

  - Order matters! -- put libraries at the end of the command line.

Linker relocation



Relocation entries

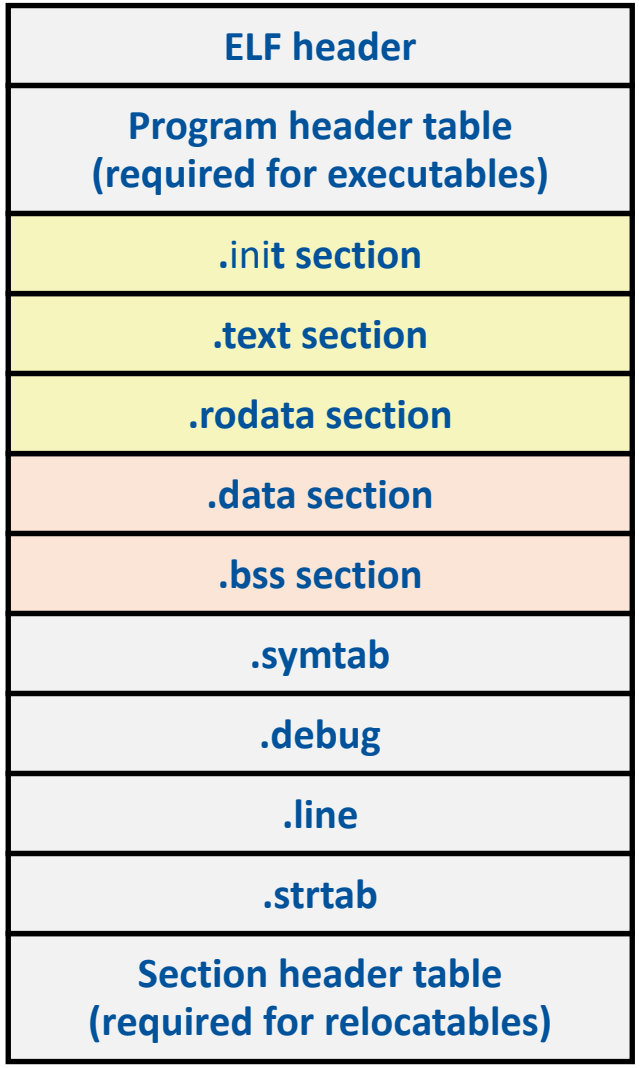
```c
int array[2] = {1, 2};

int main(int argc, char**
argv)
{
    int val = sum(array, 2);
    return val;
}
```

```
0000000000000000 <main>:
   0:   48 83 ec 08             sub    $0x8,%rsp
   4:   be 02 00 00 00          mov    $0x2,%esi
   9:   bf 00 00 00 00          mov    $0x0,%edi       # %edi = &array
                        a: R_X86_64_32 array          # Relocation entry

   e:   e8 00 00 00 00          callq  13 <main+0x13> # sum()
                        f: R_X86_64_PC32 sum-0x4      # Relocation entry
  13:   48 83 c4 08             add    $0x8,%rsp
  17:   c3                      retq
```

Relocated .text section

```
00000000004004d0 <main>:
  4004d0:       48 83 ec 08          sub    $0x8,%rsp
  4004d4:       be 02 00 00 00       mov    $0x2,%esi
  4004d9:       bf 18 10 60 00       mov    $0x601018,%edi   # %edi = &array
  4004de:       e8 05 00 00 00       callq  4004e8 <sum>     # sum()
  4004e3:       48 83 c4 08          add    $0x8,%rsp
  4004e7:       c3                   retq

00000000004004e8 <sum>:
  4004e8:       b8 00 00 00 00           mov    $0x0,%eax
  4004ed:       ba 00 00 00 00           mov    $0x0,%edx
  4004f2:       eb 09                    jmp    4004fd <sum+0x15>
  4004f4:       48 63 ca                 movslq %edx,%rcx
  4004f7:       03 04 8f                 add    (%rdi,%rcx,4),%eax
  4004fa:       83 c2 01                 add    $0x1,%edx
  4004fd:       39 f2                    cmp    %esi,%edx
  4004ff:       7c f3                    jl     4004f4 <sum+0xc>
  400501:       f3 c3                    repz retq
```

Executable object file



Next Class

- Dynamic linking
- **Readings:** 7.12 – 7.14