CSCI 210 Systems Programming

Week 8

Pointers

Dynamic memory management

Strings, Structs, Unions, Function pointers

Slides adapted from courses: Cornell CS4414, U of Buffalo CSE220, METU CENG140

Overview

- Overview of the memory space of a program
- Pointers
- Strings, i.e., char arrays, and the string library functions
- Structs and Unions
- Function pointers

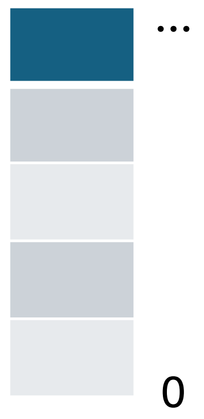
What happens when you run a compiled program?

- Bash sees that you are trying to execute a program – it finds the file and checks, and learns that it is an executable (and remembers this, for quicker future responses)

- Bash uses Linux to prepare an address space and then load and execute the program in the new address space. This is done with the **fork** and **exec** systems calls. (Both have several variants).

Address space?

A page is a 4K block of memory



- Every program runs in an isolated address context.

- Within it, the addresses your program sees are “virtual”. They don’t match directly to addresses in physical memory.

- A “page table” managed by Linux maps virtual to physical, at a granularity that would usually be 4k (4096) bytes per page.

… pages are grouped into segments

- Rather than just treating memory as one range of pages from address 0 to whatever the current size needed might be, Linux is segmented. There are often **gaps** between them.

- Definition: A segment is just a range of virtual memory with some base address, and some total size, and access rules.

- One segment might be as small as a single page, or could be huge with many pages. We don’t normally worry about page boundaries

A few segment types Linux supports

- **Code:** This kind of segment holds compiled machine instructions
- **Data:** Uses for constants and initialized global objects
- **Stack:** A stack segment is used for memory needed to call methods, or for inline variables.
- **Heap:** A heap segment is used for dynamically allocated memory that will be used for longer periods
- **Mapped files:** The file can be accessed as a byte array (mmap)

Different processes have distinct address spaces

- Each distinct process has its own address space mapping.

- Thus an address can mean different things: my 0x10000 might contain code for fast-wc, but your 0x10000 could be part of a data segment.

- The hardware knows which process is running, so it can use the proper page table mapping to know which memory it wants.

Gaps

- The address space will often have “holes” in it.
- These are ranges of memory that don’t correspond to any allocated page.
- If you try and access those regions, you’ll get a segmentation fault and your process will crash.

Stacks, Heaps

- Our programs often need to dynamically allocate memory to hold new data. Later they might free that memory.

- The stack and the heap are two resources for doing this.

… Heaps

- A heap is a memory region allocated via **malloc(size)/free.** Access the memory via *pointers*. Use a “static cast” to tell C what type of data your region will hold.
- C trusts you to allocate the proper number of bytes and stay within the bounds of the allocated region
- We will see details of pointers in the upcoming slides

Initialization is very important!

- Malloc doesn’t zero or initialize the region
- You can zero a memory region “by hand”
  - Example: If ptr points to a memory of 1000 bytes, use
    - memset(ptr, 0, 1000)

memset

memset is a function that sets a specified amount of bytes at one address to a certain value.

```
void *memset(void *s, int c, size_t n);
```

It fills n bytes starting at memory location **s** with the byte **c**. (It also returns **s**).

```
int counts[5];
memset(counts, 0, 3); // zero out first 3 bytes at counts
memset(counts + 3, 0xff, 4)// set 3rd entry’s bytes to 1s
```

Malloc is “inexpensive” but not free

- It maintains a big pool of memory and uses various techniques to keep track of the allocated versus free chunks of memory and minimize the calls to increase the program break.

  - See TLPI 7.1.3 Implementation of malloc() and free() for details

- C does not have **garbage collection**, which refers to mechanisms that automatically free an object that no longer has any references to it.

How segments grow

- Heaps and stacks are the two kinds of segments that can grow as needed, or shrink.
- A stack has a limited maximum size, but Linux initially makes it small. As methods call each other and stack space is needed, Linux finds out and quietly grows the “top” of the stack.
- If you use up the limit, then you get a “stack overflow” error, and a crash. E.g., a huge recursion depth may cause this

How segments grow

- The heap has an initial size, but can be expanded by calling the “sbrk” Linux system call.

- You don’t need to explicitly call sbrk(). Calls to malloc() will increase heap size when needed by calling sbrk() for you.

What if you access a segment illegally?

- The most notorious way for a process to crash in Linux is a “segmentation fault”
- This means it tried to read from an address that isn’t mapped into its address space, or from an “unreadable” region (or write, or execute).
- Linux terminates the whole process and might also save a “core” file for you to study using gdb to understand what crashed.

Pointers

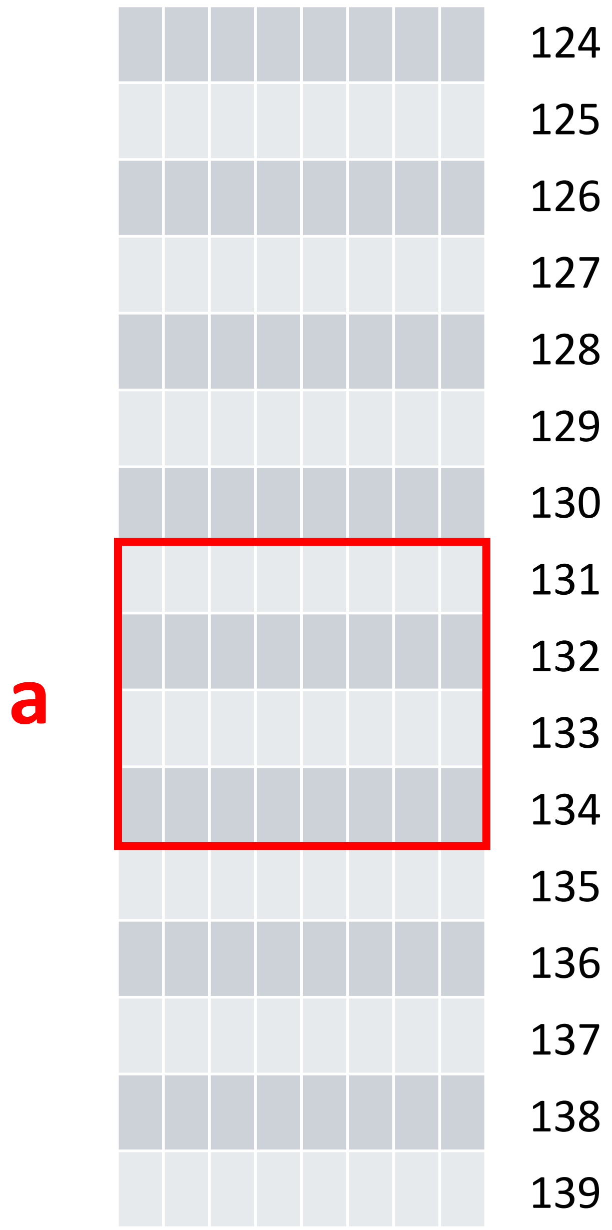
- C pointers are variables that hold memory addresses
- Pointers allow your program interact with memory explicitly
- Pointers are very powerful, but potentially unsafe tools
- The C compiler doesn’t know which pointers are valid!
- Most non-trivial data structures in C use pointers
    - Examples: linked lists, trees, graphs (covered in CSCI 220)

How to get addresses in C?

```
int a;
```

`&a` $\longrightarrow$ Address of a

address operator



A variable and its address

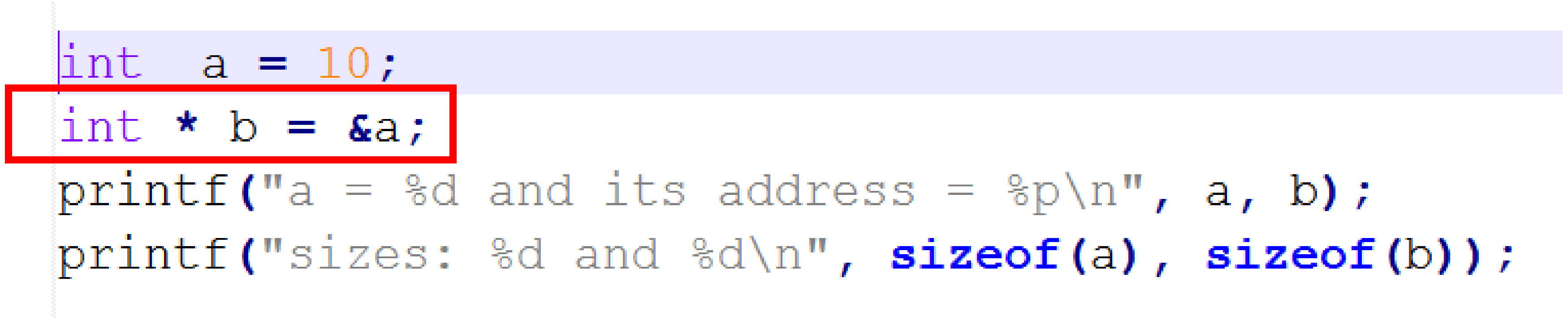
```
 int a = 10;

printf("a is %d and its address %p\n",a,&a);
printf("sizes: %lu and %lu\n",sizeof(a),sizeof(&a));
```

- This is the output:

```
a = 10 and its address = 0x7ffff90bdb34
sizes: 4 and 8
```

- The output depends on the architecture!
  - sizeof(a) → depends on how much space an int takes
  - sizeof(&a) → depends on the how large an address can be (therefore how much space and “address” needs)

Variables, addresses and pointers



```
int  a = 10;
int * b = &a;
printf("a = %d and its address = %p\n", a, b);
printf("sizes: %d and %d\n", sizeof(a), sizeof(b));
```

- The data type storing addresses are called pointers!
    - int * , char * , float * , double *

Pointers can point to different variables

```
int a = 10;
int c = 20;
int *b;

b = &a;
printf("a is %d and its address %p\n",*b,b);

b = &c;
printf("c is %d and its address %p\n",*b,b);
```

```
a = 10 and its address = 0x7fffe1e22aa8
c = 20 and its address = 0x7fffe1e22aac
```

Addresses are data, too.
They are integers (long) and
usually printed in hexadecimal

Pointers

```
int a = 10;
int * b = &a;

*b = 20;

a = 2 / *b + 25;
```

- Initialization is important since a pointer initially points to an arbitrary memory position, which may not belong to your program!
- A good practice:
  - `int *a = NULL;`

*: dereferencing operator (dereference expression can both be l-value and r-value)

Pointer arithmetic

```
int a = 10;
char c = 20;
int *b;
char *d;

b = &a;
d = &c;
printf("b is %p and b+1 is %p\n",b,b+1);
printf("d is %p and d+1 is %p\n",d,d+1);
```

```
b is 0x7fffc9c17ea4 and b+1 is 0x7fffc9c17ea8
d is 0x7fffc9c17ea3 and d+1 is 0x7fffc9c17ea4
```

So, the difference depends on the data type!

Pointer arithmetic

- Pointer arithmetic is based on the ***type*** of the pointer.
- In other words, if p is a pointer, p+1 points to the next object (next int, char, float or double).
- So, p+1 is not necessarily the next byte in the memory!!!

Pointer arithmetic

- Increment, decrement operators work on pointers:

```
int *a = &b;

a++,   ++a
a--,    --a
```

- Pointer operators (&, *) have the same precedence with ++, -- (and unary +, -)!
- They are right-to-left associative!
  - `*++cp` → `*(++cp)`
  - `*cp++` → `*(cp++)`

void* type

- void *
  - → Generic pointer
  - → Useful especially in cases where we don’t know the type of the data beforehand!

Pointer Comparison

- Equality comparison is meaningful between:
  - Pointers of the same type
  - A pointer with a void pointer
  - A pointer and a NULL pointer
- The result is true if the operands point to the same *object*

- For other relational operators (<, <=, >, >=):
  - Result is based on the relative addresses of the objects pointed to.

Pointer Type Casting

- similar to explicit casting of regular data types (i.e., int, float, double, char, ..), we can type cast pointers:

```
int * a;
float *d;
a = (int *) d;
```

- Type casting a pointer is considered an unsafe operation
- According to the C standard, reinterpreting a floating point value as an integer by casting then dereferencing the pointer to it is not valid (undefined behavior).

Pointers Summary - I

- A pointer: Contains an address
- Allows the memory at that address to be manipulated
    - Dereferencing (*) and address (&) operators
- Associates a type with the manipulated memory
- To the computer, memory is just bits
    - Programmers supply the meaning
- The special pointer value NULL represents an invalid address
- Pointer arithmetic using +, -, ++, --. Comparison of pointers is OK.

Pointers Summary - II

- Pointers must store a valid address
- There are limited opportunities to create valid addresses:
  - Acquire the address of a variable
  - Request new memory from the system
  - Create a string or array constant
  - Calculation from other addresses
- Pointers created in other manners probably are not valid

Additional ways to dereference a pointer

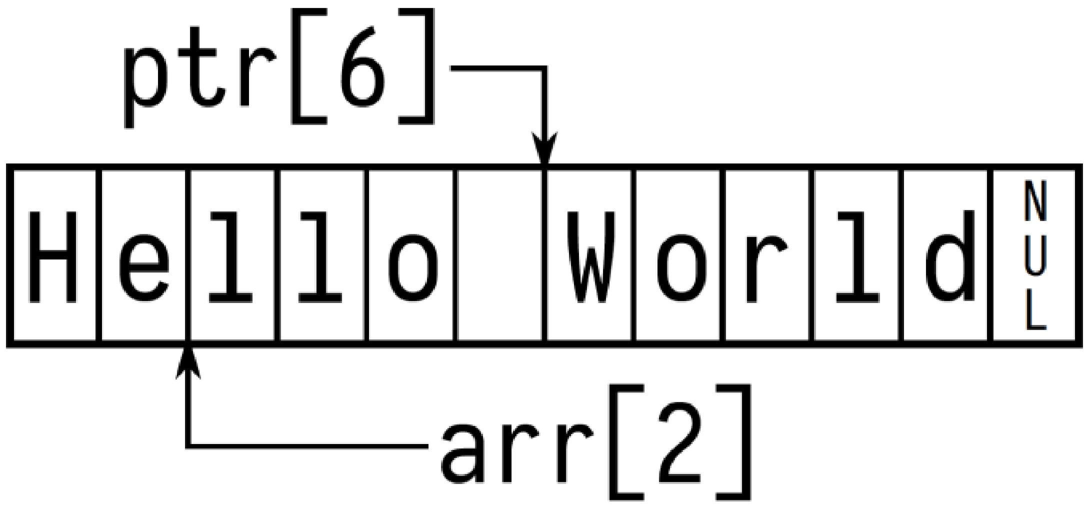
- When accessing struct fields and if p is a pointer to a struct, (*p).field can be written as p->field
  - We will see structs shortly
- We can use the array indexing operator on pointers:
  - *(p+10) is the same as p[10]

Pointers and arrays

- Arrays and pointers are closely related in C
- You can often think of an array variable as a pointer to the first array element, and a pointer variable as an array
- However, they are not the same.
- In both cases, dereferencing with [i] says
  - …add i times the size of the type of this variable to the base address (first element of the array or pointer value), then treat the memory at that location as if it is of the type of this variable.

Pointers and arrays

```
char arr [] = "Hello World";
char *ptr = arr;
```



malloc() and free()

- Check the first page of “man malloc” to learn about these functions.
- Yes, C functions also have man pages
    - Sections 2 and 3 of the manual pages contain system and library calls
- Additional resource:
    - https://www.geeksforgeeks.org/dynamic-memory-allocation-in-c-using-malloc-calloc-free-and-realloc/

Exercise

- Which lines have errors?

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

  int main(int argc, char** argv) {
    int a[2];
    int* b = malloc(2*sizeof(int));
    int* c;

1   a[2] = 5;
2   b[0] += 2;
3   c = b+3;
4   free(&(a[0]));
5   free(b);
6   free(b);
7   b[0] = 5;

    return EXIT_SUCCESS;
  }
```

Memory Leak

- A memory leak occurs when code fails to deallocate dynamically allocated memory that is no longer used
  - e.g. forget to free malloc-ed block, lose/change pointer to malloc-ed block
- What happens: program's VM footprint will keep growing
  - This might be OK for short-lived program, since all memory is deallocated when program ends
- Usually has bad repercussions for long-lived programs
  - Might slow down over time (e.g. lead to VM thrashing)
  - Might exhaust all available memory and crash
  - Other programs might get starved of memory
- Best tool for finding leaks: **valgrind**
  - **We will learn about gdb and valgrind next week**

Arrays and strings

- C arrays are a series of contiguous memory locations
- Arrays are declared with []. The size is between []
- Arrays can have three “sizes”, depending on what’s in the []
  - Unknown size: Nothing is specified
    - We can make it known by providing an “initializer”
  - Constant size: A constant expression is specified
  - Variable size: A run-time computed expression is specified
- Array access violations due to their sizes are not caught in compile time

Static initializers

- An array can be initialized all at once at declaration

```
int array [10] = { 0, 3, 5, 0, 0, 1, 0, 0, 2, 0};
```

- This is called a static initializer
- Static initializers can only be used at declaration

```
int array [3];
array = { 1, 3, 5 }; /* syntax error */
```

- Can provide a smaller number of initializers than the size. The rest is filled with zeros.
- Shorthand for initializing an array to all zeros:

```
int array [10] = {};
```

C strings

- C strings are arrays of chars
- A C string consists of:
  - the characters in the string, followed by a zero byte (the ASCII NULL character) (NULL terminator), i.e., **0**, `'\0'`, **NOT `'0'`**
- Strings, like arrays, do not have an associated length
- You can count the number of chars until the terminator to know how long the string is

String examples

- Strings are represented as a series of characters between double quotes
- Strings can be declared as follows

```
char str [] = "Hello";
/* same as str = { 'H', 'e', 'l', 'l', 'o', '\0' } */
```

- Like arrays, such an assignment is possible only at declaration
- After declaration, strings must be copied into arrays

```
char str [32];
strncpy(str , 32, "Hello"); /* See man 3 strncpy */
```

What about?

```
char *str;
str = "Hello";
```

Are the following a valid operations?

```
str[0] = 'Y'
free(str);
```

String constants

- In general, assume that C string constants are not modifiable
  - There are compile options to make them modifiable
  - But the many C standards on this issue advises not attempting this
    - https://wiki.sei.cmu.edu/confluence/display/c/STR30-C.+Do+not+attempt+to+modify+string+literals
- See also:
  - https://www.gnu.org/software/c-intro-and-ref/manual/html_node/String-Constants.html

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main() {
    char *str = malloc(sizeof(char) * 14);
    // Allocate memory for 13 characters + null terminator
    strcpy(str, "Hello, World!");
    str[0] = 'J';
    printf("%s\n", str); // Output: Jello, World!
    free(str); // Free allocated memory    return 0;
}
```

String functions in C

- There are many string functions in the C library.
- Most of them are defined in **`<string.h>`**
- Some useful examples:
  - **`strlen()`**: Compute the length of a string by counting bytes
  - **`strncpy()`**: Copy a string until its NUL character
  - **`strncat()`**: Concatenate one string to another
  - **`strstr()`**: Search for one string inside another
  - **`strtok()`**: Splits the string into tokens based on a delimiter
    - It does not return an array of tokens, but the next single token
    - It has an interesting usage, where it remembers where it is on the current parsed string. Check out the manual page for **`strktok()`**.

structs and unions

- They are derived types
  - structs are “product” types, e.g., the values of a struct type are elements of the set, AxBxC, where are A, B, and C are the sets of values of the primitive types that comprise the struct
  - unions are “sum” types, e.g., the values of a union type are elements of the set, A+B+C, where are A, B, and C are the sets of values of the primitive types that comprise the struct
- Example: a value of a struct of int and float is (3,4.5), where a value of union of int and float is 3 (another value of that union could be 4.5).

struct

- A struct is a C datatype that contains a set of fields
- It is similar to a Java/C++ class
  - but with no methods or constructors
- Useful for defining new structured types of data
  - For returning multiple values from a function for example
  - Or for defining higher order types like a “point_3d”
- Behave similarly to primitive variables
  - They can be assigned to each other and passed to functions as copies (unlike strings or arrays)

struct example

```c
// the following defines a new
// structured datatype called
// a "struct Point"
struct Point {
  float x, y;
};

// declare and initialize a
// struct Point variable
struct Point origin = {0.0,0.0};
```

Type definition versus variable declaration

```c
// the following defines a new
// structured datatype called
// a "struct Point"
struct Point {
  float x, y;
};
```

```c
// the following defines a new
// structured datatype called
// a "struct Point" and declares
// a variable "origin" of type
// struct Point
struct Point {
  float x, y;
} origin;
```

Using structs

- Use “.” to refer to a field in a struct
- Use “->” to refer to a field from a struct pointer
  - Dereferences pointer first, then accesses field
- You can assign the value of a struct from a struct of the same type
  - this copies the entire contents! E.g., p2 = p1

```
struct Point {
  float x, y;
};

int main(int argc, char** argv) {
  struct Point p1 = {0.0, 0.0};  // p1 is stack allocated
  struct Point* p1_ptr = &p1;

  p1.x = 1.0;
  p1_ptr->y = 2.0;  // equivalent to (*p1_ptr).y = 2.0;
  return EXIT_SUCCESS;
}
```

`typedef`

- Generic format: `typedef type name;`
- Allows you to define new data type names/synonyms
- Both type and name are usable and refer to the same type
- Be careful with pointers – * before name is part of type!
- Especially useful with structs as it allows you to get rid of the struct keyword when declaring struct type variables.

```
// make "superlong" a synonym for "unsigned long long"
typedef unsigned long long superlong;

// make "str" a synonym for "char*"
typedef char *str;

// make "Point" a synonym for "struct point_st { ... }"
// make "PointPtr" a synonym for "struct point_st*"
typedef struct point_st {
  superlong x;
  superlong y;
} Point, *PointPtr;  // similar syntax to "int n, *p;"

Point origin = {0, 0};
```

Dynamically allocated structs

- You can malloc and free structs, just like other data type
  - `sizeof` is particularly helpful here

```c
// a complex number is a + bi
typedef struct complex_st {
  double real;    // real component
  double imag;    // imaginary component
} Complex, *ComplexPtr;

// note that ComplexPtr is equivalent to Complex*
ComplexPtr AllocComplex(double real, double imag) {
  Complex* retval = (Complex*) malloc(sizeof(Complex));
  if (retval != NULL) {
    retval->real = real;
    retval->imag = imag;
  }
  return retval;
}
```

Structs as function arguments

- Structs are passed by value, like everything else in C
  - Entire struct is copied
- To manipulate a struct argument, pass a pointer instead

```c
typedef struct point_st {
  int x, y;
} Point, *PointPtr;

void DoubleXBroken(Point p)   {  p.x *= 2; }

void DoubleXWorks(PointPtr p) { p->x *= 2; }

int main(int argc, char** argv) {
  Point a = {1,1};
  DoubleXBroken(a);
  printf("(%d,%d)\n", a.x, a.y);   // prints: (  ,  )
  DoubleXWorks(&a);
  printf("(%d,%d)\n", a.x, a.y);   // prints: (  ,  )
  return EXIT_SUCCESS;
}
```

Pass a copy or a pointer?

- Value passed: passing a pointer is cheaper and takes less space unless struct is small
- Field access: indirect accesses through pointers are a bit more expensive and can be harder for compiler to optimize
- For small structs (like struct point), passing a copy of the struct can be faster and often preferred if function only reads data; for large structs use pointers

Unions

- Similar usage to struct. However:
  - The value stored at a given time should be **one** of the constituent types only
  - Storing as one type and retrieving as another type has undefined behavior
  - The size of a union is usually smaller than a struct and determined as the size of its largest field.

Union example

```
union u {
      int i;
      float f;
};
union u myunion;

myunion.i = 3;
printf("as int: %d\n",myunion.i);
myunion.f = 3.14;
printf("as int: %f\n",myunion.f);

printf("size of the union: %lu\n",sizeof(myunion));
```

Functions

Function **definition**

```
return_type function_name(parameter declarations) {
  statement-1;
  statement-2;
...
}
```

- if is *return_type* not void, “return” statement must be used:

```
    return expression;
```

Function **declaration**

- *return_type* function_name(list-of-params);
- The parameters have to have the same types as in the function definition although the names of the parameters may differ (or omitted altogether).
- Example:
  - `int factorial(int N);`
  - `void print_matrix(int matrix[N][M]);`
- If a function is used before it is defined, it must be declared first.

Function **call**

function_name(list of arguments)

- Example:
  - Function declaration:

```
int greatest(int A, int B, int C);
```

  - Example function call:

```
printf("%d\n", greatest(10, 20, -10));
```

Exercises

- Write a function that finds the length of a string.
- Write two functions that finds the minimum and the maximum values in an array.
- Write a function that checks whether two arrays are equal.

Call by Value

- The arguments of the function are just copies of the passed data!

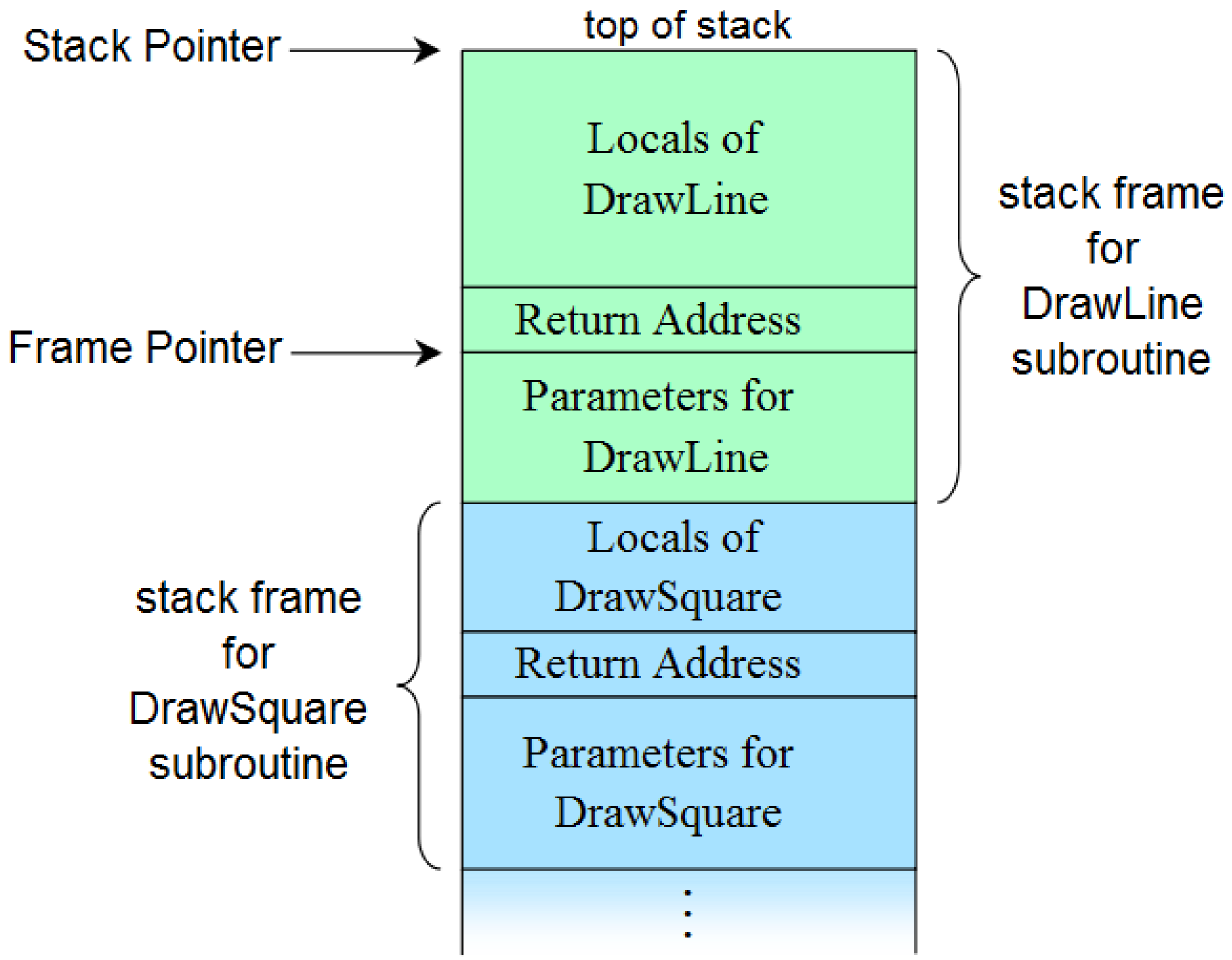
```
void f(int a){
 a = 10 * a;
}
void g(int b){
 b = 10;
 f(b);
 printf("%d",  b);
}
```

Call by Value

- So, what do we do? How can I get the new updated value?
  - You can use the “return” statement and assign it to a variable or directly print it.
  - If you have more than one variable, you can use global variables.
  - Or, you can use pointers!

Tracking Function Calls

- Function calls are “traced” using the call stack.



Function Pointers

- In C, there is a variable type for functions!
- We can pass functions as parameters, store functions in variables, etc.
- Why is this useful?

Need for generic functionality

Sometimes, there is functionality that *cannot* be made generic.

```
void bubble_sort(void *arr, int n, int elem_size_bytes) {
    while (true) {
        bool swapped = false;
        for (int i = 1; i < n; i++) {
            void *prev_elem = (char *)arr + (i-1)*elem_size_bytes;
            void *curr_elem = (char *)arr + i*elem_size_bytes;
            if (curr_elem should come before prev_elem) {
                swapped = true;
                swap(prev_elem, curr_elem, elem_size_bytes);
            }
        }

        if (!swapped) {
            return;
        }
    }
}
```

Need for generic functionality

The caller can pass in a function to perform that functionality for the data they are providing.

```
void bubble_sort(void *arr, int n, int elem_size_bytes,
        bool (*cmp_fn)(const void *, const void *)) {
    while (true) {
        bool swapped = false;
        for (int i = 1; i < n; i++) {
            void *prev_elem = (char *)arr + (i-1)*elem_size_bytes;
            void *curr_elem = (char *)arr + i*elem_size_bytes;
            if (cmp_fn(prev_elem, curr_elem) > 0)) {
                swapped = true;
                swap(prev_elem, curr_elem, elem_size_bytes);
            }
        }

        if (!swapped) {
            return;
        }
    }
}
```

Generic C Standard Library Functions

- **qsort** – I can sort an array of any type! To do that, I need you to provide me a function that can compare two elements of the kind you are asking me to sort.
- **bsearch** – I can use binary search to search for a key in an array of any type! To do that, I need you to provide me a function that can compare two elements of the kind you are asking me to search.
- **lfind** – I can use linear search to search for a key in an array of any type! To do that, I need you to provide me a function that can compare two elements of the kind you are asking me to search.
- **lsearch** - I can use linear search to search for a key in an array of any type! I will also add the key for you if I can't find it. In order to do that, I need you to provide me a function that can compare two elements of the kind you are asking me to search.

Generic C Standard Library Functions

- **scandir** – I can create a directory listing with any order and contents! To do that, I need you to provide me a function that tells me whether or not you want me to include a given directory entry in the listing. I also need you to provide me a function that tells me the correct ordering of two given directory entries.

Function Pointers

Here’s the variable type syntax for a function:

```
[return type] (*[name])([parameters])
```

Function Pointers

```
int do_something(char *str) {
    …
}

int main(int argc, char *argv[]) {
    …
    int (*func_var)(char *) = do_something;
    …
    func_var("testing");
    return 0;
}
```

Function Pointers

```
void bubble_sort(void *arr, int n, int elem_size_bytes,
        int (*cmp_fn)(const void *, const void *)) {

…
}

int cmp_double(const void *, const void *) {…}

int main(int argc, char *argv[]) {
    …
    double values[] = {1.2, 3.5, 12.2};
    int n = sizeof(values) / sizeof(values[0]);
    bubble_sort(values, n, sizeof(*values), cmp_double);
    …
}
```

Comparison Functions

- Comparison functions are a common use of function parameters, because many generic functions must know how to compare elements of your type.
- Comparison functions always take *pointers to the data they care about,* since the data could be any size!

When writing a comparison function callback, use the following pattern:

1) Cast the void *argument(s) and set typed pointers equal to them.
2) Dereference the typed pointer(s) to access the values.
3) Perform the necessary operation.

(steps 1 and 2 can often be combined into a single step)

Comparison Functions

- It should return:
  - < 0 if first value should come before second value
  - > 0 if first value should come after second value
  - 0 if first value and second value are equivalent
- This is the same return value format as **strcmp**!

```
int (*compare_fn)(const void *a, const void *b)
```

Integer Comparison Function

```
int integer_compare(void *ptr1, void *ptr2) {
    // cast arguments to int *s and dereference
    int num1 = *(int *)ptr1;
    int num2 = *(int *)ptr2;

    // perform operation
    return num1 - num2;
}

…
qsort(mynums, count, sizeof(*mynums), integer_compare);
```

String Comparison Function

```
int string_compare(void *ptr1, void *ptr2) {
     // cast arguments and dereference
     char *str1 = *(char **)ptr1;
     char *str2 = *(char **)ptr2;

     // perform operation
     return strcmp(str1, str2);
}

…
qsort(mystrs, count, sizeof(*mystrs), string_compare);
```