Lecture Participation Poll #29

Log onto pollev.com/cse374
Or
Text CSE374 to 22333

Lecture 29: Final Exam Review & Careers in tech

CSE 374: Intermediate Programming Concepts and Tools

Administrivia

- Reminders:
  - Final exam Wednesday 12/15 8:30am to 10:20am in CS2 G10
  - HW6 due Monday
  - Individual HWs 4, 5 & 6 due
  - No office hours next week (we will respond to Ed posts)
  - HW5 grades posting Monday 10/13 – Regrade requests will be due Thursday 10/16
- Note: Klaatu gets cleaned off at the start of each quarter
- Career Stuff:
  - https://bit.ly/csecareerguide https://bit.ly/cseresumeguide
  - 492J – "Careers in Tech" seminar 1 cr no hw lectures on how to get an interview & pass it
  - Join Husky Coding Project – student founded and run personal coding project club
  - Feel free to set up time with Kasey anytime via calendly! once a student always a student :)

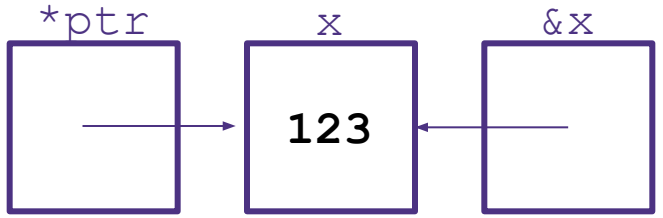
Final Exam Outline:
- C Pointer mystery
- C Errors
- C dynamic memory programming
- C Make Files
- C++ inheritance mystery
- C++ Programming
- Assembly

Final Review

Pointer and Address Syntax in C

`int *ptr;` also works! Programmer preference

```
int* ptr; // a variable of type "pointer to int" without assignment

int x = 123; //an int variable called "x" that stores "123"

ptr = &x; // store the address of "x" in "ptr"
```



* Means "pointer to type"
  - `*` placed after type indicates a pointer data type
    - Similar in java if you add [] after type you declare an array of that type
    - `int*` means "pointer to int"

`&` means "address variable"
  - Placing an & before a variable name will give you the address in memory of that variable

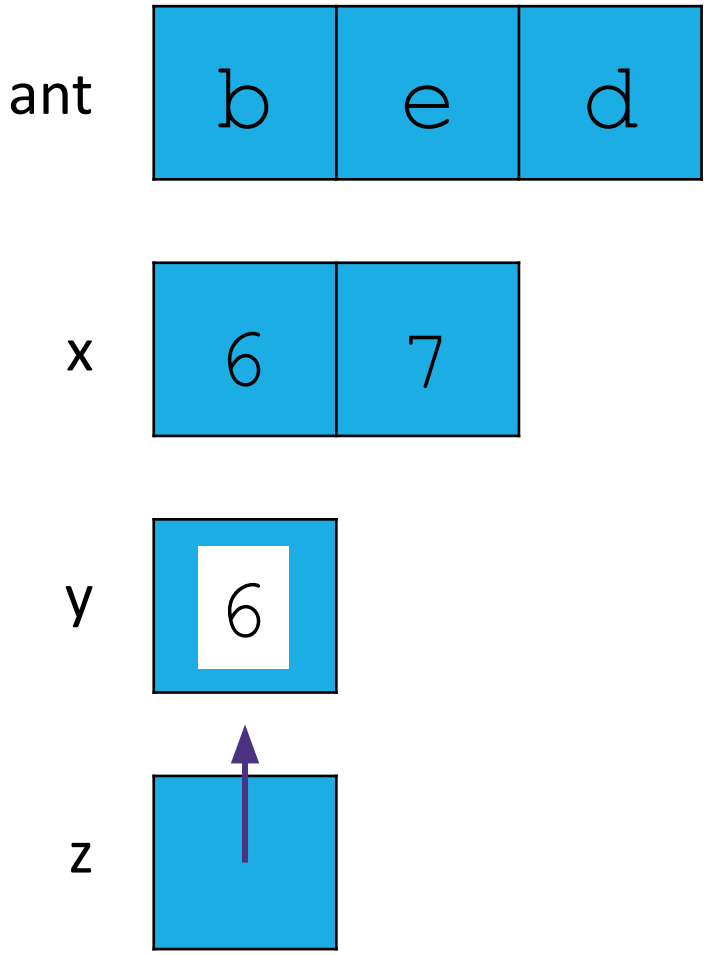
Dereferencing Pointers

```
int x = 123;
int* ptr = &x;
*ptr = 456;
printf("new value of y:%d\n", *ptr);
```

- Placing a * before a pointer **dereferences** the pointer
  - Means "follow this pointer" to the actual data
  - `*ptr = <data>` will update the data stored at the address the pointer is referring to ie 'write to memory'
  - `*ptr` will read the data stored at the address indicated by the pointer
  - Accessing unused addresses causes a 'segmentation fault'
- A **dangling pointer** is one that points to a dead local variable
  - Data that is no longer in use
  - Dereferencing a dangling pointer is "undefined behavior" (UB)
  - UB means ANYTHING could happen
    - Program could crash(best case), silently fail(worst case)
    - GCC can catch this kind of error with a warning, but not always

Pointer Mystery

```
#include <stdio.h>
// What does the program print?

void mystery(char *a, int *b, int c)
{
    int *d = b - 1;
    c = *b + c;
    *b = c - *d;
    *d = *b - *d;
    a[2] = a[b - d];
}

int main(int argc, char **argv)
{
    char ant[4] = "bed";
    int x[2];
    *x = 6;
    x[1] = 7;
    int y = 4;
    int *z = &y;
    *z = *x;
    printf("%d %d %d %s\n", *x, x[1], y, ant);
    mystery(ant, x + 1, y);
    printf("%d %d %d %s\n", *x, x[1], y, ant);
}
```



Output:
6 7 6 bed
1 7 6 bee

Memory Allocation

- **Allocation** refers to any way of asking for the operating system to set aside space in memory
- How much space? Based on variable type & your system
  - to get specific sizes for your system use "sizeof(<datatype>)" function in stdlib.h
- Global Variables – **static** memory allocation
  - space for global variables is set aside at compile time, stored in RAM next to program data, not stack
  - space set aside for global variables is determined by C based on data type
  - space is preserved for entire lifetime of program, never freed
- Local variables – **automatic** memory allocation
  - space for local variables is set aside at start of function, stored in stack
  - space set aside for local variables is determined by C based on data type
  - space is deallocated on return

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	-32,786 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	-9223372036854775808 to 9223372036854775807
unsigned long	8 bytes	0 to 18446744073709551615
float	4 bytes	1.2E-38 to 3.4E+38
double	8 bytes	2.3E-308 to 1.7E+308
long double	10 bytes	3.4E-4932 to 1.1E+4932

* pointers require space needed for an address – dependent on your system – 4 bytes for 32-bit, 8 bytes for 64-bit

https://www.gnu.org/software/libc/manual/html_node/Memory-Allocation-and-C.html

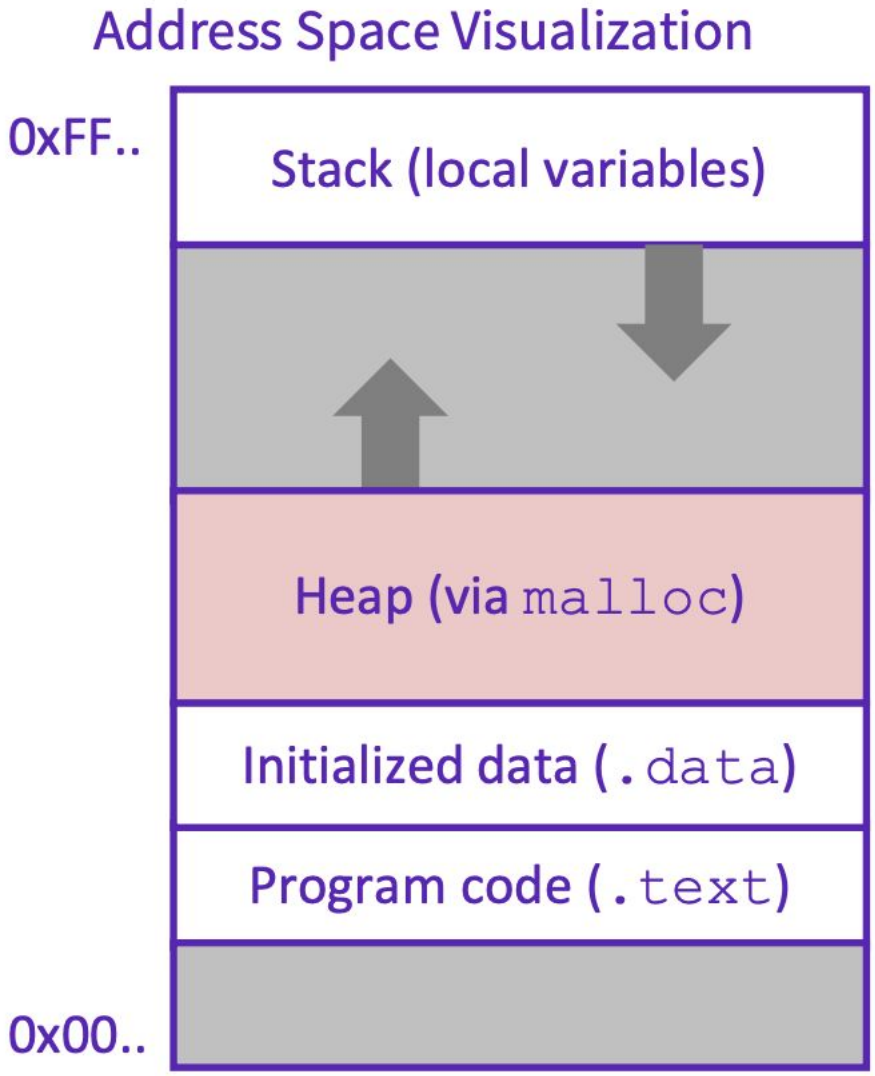
Dynamic Allocation

- Situations where static and automatic allocation aren't sufficient
  - Need memory that persists across multiple function calls
    - Lifetime is known only at runtime (long-lived data structures)
  - Memory size is not known in advance to the caller
    - Size is known only at runtime (ie based on user input)
- Dynamically allocated memory persists until:
  - A garbage collector releases it (automatic memory management)
    - Implicit memory allocator, programmer only allocates space, doesn't free it
    - "new" in Java, memory is cleaned up after program finishes <HOW DOES THIS WORK?
  - Your code explicitly deallocates it (manual memory management)
    - C requires you manually manage memory
    - Explicit memory allocation requires the programmer to both allocate space and free it up when finished
    - "malloc" and "free" in C
- Memory is allocated from the heap, not the stack
  - Dynamic memory allocators acquire memory at runtime

Storing Program Data in the RAM

- When you trigger a new program the operating system starts to allocate space in the RAM
  - Operating System will default to keeping all memory for a program as close together within the ram addresses as possible
  - Operating system manages where exactly in the RAM your data is stored
    - Space is first set aside for program code (lowest available addresses)
    - Then space is set side for initialized data (global variables, constants, string literals)
    - As program runs…
      - When the programmer manually allocates memory for data it is stored in the next available addresses on top of the initialized data, building upwards as space is needed
      - When the program requires local variables they are stored in the empty space at top of RAM, leaving space between stack and heap
      - When the space between the stack and heap is full – crash (out of memory)

The heap is a large pool of available memory set aside specifically for dynamically allocated data



Allocating Memory in C with malloc()

- `void* malloc(size_t size)`
  - allocates a continuous block of "`size`" bytes of **uninitialized** memory
  - Returns `null` if allocation fails or if `size == 0`
    - Allocation fails if out of memory, very rare but always check allocation was successful before using pointer
  - `void*` means a pointer to any type (int, char, float)
    - `malloc` returns a pointer to the beginning of the allocated block
- `var = (type*) malloc(sizeInBytes)`
  - Cast void* pointer to known type
  - Use `sizeof(type)` to make code portable to different machines
- `free` deallocates data allocated by `malloc`
- Must add `#include <stdlib.h>`
- Variables in C are uninitialized by default
  - No default "0" values like Java
  - Invalid read – reading from memory before you have written to it

```
//allocate an array to store 10 floats
float* arr = (float*) malloc(10*sizeof(float));
if (arr == NULL)
{
    return ERROR;
}
printf("%f\n", *arr) // Invalid read!
<add something to array>
<print f again, now it's ok>
```

Freeing Memory in C with free()

- `void free(void* ptr)`
  - Released whole block of memory stored at location ptr to pool of available memory
  - ptr must be the address originally returned by malloc (the beginning of the block) otherwise system exception raised
  - ptr is unaffected by free
    - Set pointer to NULL after freeing it to deallocate that space too
  - Calling free on an already released block (double free) is undefined behavior – best case program crashes
  - Rule of thumb: for every runtime call to malloc there should be one runtime call to free
  - if you lose all pointers to an object you can no longer free it – memory leak!
    - be careful when reassigning pointers
    - this is usually the cause of running out of memory- unreachable data that cannot be freed
  - if you attempt to use an object that has been freed you hit a dangling pointer
  - all memory is freed once a process exits, and it is ok to rely on this in many cases

```
//allocate an array to store 10 floats
float* arr = (float*)
malloc(10*sizeof(float));
if (arr == NULL)
{
    return ERROR;
}
for (int i = 0; i < size*num; i++)
{
    arr[i] = 0;
}
free(arr);
arr = NULL; // Optional
```

Common Memory Errors

```
x = (int*)malloc(M*sizeof(int));
free(x);
y = (int*)malloc(M*sizeof(int));
free(x);
```

Double free and Forgetting to free memory “memory leak”

```
int x[] = {1, 2, 3};
free(x);
```

x is a local variable stored in stack, cannot be freed

```
char** strings = (char**)malloc(sizeof(char)*5);
free(strings);
```

Mismatch of type - wrong allocation size

```
x = (int*)malloc(M*sizeof(int));
free(x);
y = (int*)malloc(M*sizeof(int));
for (i=0; i<M; i++)
   y[i] = x[i];
```

Accessing freed memory

Common Memory Errors

```
#define LEN 8
int arr[LEN];
for (int i = 0; i <= LEN; i++)
   arr[i] = 0;
```

Out of bounds access

```
int* foo()
{
   int val = 0;
   return &val;
}
```

dangling pointer

```
long val;
printf("%d", &val);
```

Dereferencing a non-pointer

```
int foo()
{
   int* arr = (int*)malloc(sizeof(int)*N);
   read_n_ints(N, arr);
   int sum = 0;
   for (int i = 0; i < N; i++)
      sum += arr[i];
   return sum;
}
```

memory leak – failing to free memory

```
int sum_int(int* arr, int len)
{
   int sum;
   for (int i = 0; i < len; i++)
      sum += arr[i];
   return sum;
}
```

Reading memory before allocation

Structs

- **structs** are a method of constructing new datatypes
  - store a collection of values together in memory, fields
  - similar to a Java class, but no methods
  - individual values are referred to using the "." operator
  - can use typedef to rename and turn struct tag into a "type"

```
typedef struct Cat Cat;
```

or

```
typedef struct Cat {
    …
} Cat;
```

Then you don't need keyword "struct"

`Cat mercy;` instead of `struct Cat mercy;`

```
struct Cat
{
    char *name;
    int age;
    char *breed;
}
int main()
{
    struct Cat mercy;
    mercy.name = "Iron Fist No Mercy";
    mercy.age = 6;
    mercy.breed = "Pixie Bob";
}
```

Parameters / Arguments

- Function parameters are initialized with a copy of corresponding argument
  - If the argument is a pointer, the parameter value will point to the same thing (pointer is copied)
  - arrays are passed as pointers
  - Structs are passed as a copy by default, so it is more common to intentionally pass as pointers
    - avoids copying large objects
    - allows manipulation of original struct <- allows creation of methods that manipulate new type, like Java
    - to access members you must dereference the pointer (*) and access the field (.) – use parentheses to ensure dereference happens first
    - `(*ptr).` has a shortcut: `ptr->`

```
Cat (*ptr) = (Cat*)malloc(sizeof(Cat));
(*ptr).age = 6;
…
(*ptr).age++;
ptr->age;
```

Common C Bugs

- forget to free -> program uses more memory than needed
- memory leak -> lose pointer to start of dynamically allocated memory, can't free
- keep using after free -> later calls to malloc may reuse freed memory
- double free -> can corrupt internal data structures of malloc
- dangling pointer -> lose memory that pointer referenced, dereferencing dangling pointer, undefined behavior

Segmentation Fault

- attempt to access memory that "does not belong to you"
- indicates memory corruption
- Can be caused by:
  - array index out of bounds
  - accessing freed memory
  - dereferencing null pointer
  - changing String(char*) literal

https://en.wikipedia.org/wiki/Segmentation_fault

C Debugger

- A debugger is a tool that lets you stop running programs, inspect values etc…
  - instead of relying on changing code (commenting out, printf) interactively examine variable values, pause and progress set-by-step
  - don't expect the debugger to do the work, use it as a tool to test theories
  - Most modern IDEs have built in debugging functionality
- 'gdb' -> gnu debugger, standard part of linux development, supports many languages
  - techniques are the same as in most debugging tools
  - can examine a running file
  - can also examine core files of previous crashed programs
- Want to know which line we crashed at (backtrace)
- Inspect variables during run time
- Want to know which functions were called to get to this point (backtrace)

Meet gdb

- Compile code with '`-g`' flag
  - `gcc -g program.c`
  - saves human readable info
- Open program with `gdb <executable file>`
  - `gdb a.out`
- start or restart the program: `run <program args>`
  - quit the program: `kill`
  - quit gdb: `quit`
- Reference information: help
  - Most commands have short abbreviations
    - `bt` = backtrace
    - `n` = next
    - `s` = step
    - `q` = quit
  - <return> often repeats the last command

```
Breakpoint 1, factorial (x=10) at factorial.c:18
18        if (x == 0) {
(gdb) n
21          return x * factorial(x-1);
(gdb) n

Breakpoint 1, factorial (x=9) at factorial.c:18
18        if (x == 0) {
(gdb) n
21          return x * factorial(x-1);
(gdb) n

Breakpoint 1, factorial (x=8) at factorial.c:18
18        if (x == 0) {
(gdb) n
21          return x * factorial(x-1);
(gdb) n

Breakpoint 1, factorial (x=7) at factorial.c:18
18        if (x == 0) {
(gdb) n
21          return x * factorial(x-1);
(gdb) n

Breakpoint 1, factorial (x=6) at factorial.c:18
18        if (x == 0) {
(gdb)
```

[Video] gdb debugger demo

Valgrind

- Valgrind is a tool that simulates your program to find memory errors
  - catches pointer errors during execution
  - prints summary of heap usage, including details of memory leaks

```
gcc -g -o myprogram myprogram.c
valgrind --leak-check=full myprogram arg1 ag
```

- Can show:
  - Use of uninitialized memory
  - Reading/writing memory after it has been free'd
  - Reading/writing off the end of malloc'd blocks
  - Reading/writing inappropriate areas on the stack
  - Memory leaks -- where pointers to malloc'd blocks are lost forever
  - Mismatched use of malloc/new/new [] vs free/delete/delete []
  - Overlapping src and dst pointers in memcpy() and related functions

[Video] Valgrind Demo

Valgrind Example

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

int main(int argc, char** argv){
  int i;
  int *a = malloc(sizeof(int) * 10);
  if (!a) return -1; /*malloc failed*/
  for (i = 0; i < 11; i++){
    a[i] = i;
  }
  free(a);
  return 0;
}
```

example1.c

Attempt to write 4 bytes to an invalid location in memory (sizeof(int))
a[10] -> index out of bounds

```
$ gcc -Wall -pedantic -g example1.c -o example
$ valgrind ./example
==23779== Memcheck, a memory error detector
==23779== Copyright (C) 2002-2009, and GNU GPL'd, by Julian Seward et al.
==23779== Using Valgrind-3.5.0 and LibVEX; rerun with -h for copyright info
==23779== Command: ./example
==23779==
==23779== Invalid write of size 4
==23779==    at 0x400548: main (example1.c:9)
==23779==  Address 0x4c30068 is 0 bytes after a block of size 40 alloc'd
==23779==    at 0x4A05E46: malloc (vg_replace_malloc.c:195)
==23779==    by 0x40051C: main (example1.c:6)
==23779==
==23779==
==23779== HEAP SUMMARY:
==23779==     in use at exit: 0 bytes in 0 blocks
==23779==   total heap usage: 1 allocs, 1 frees, 40 bytes allocated
==23779==
==23779== All heap blocks were freed -- no leaks are possible
==23779==
==23779== For counts of detected and suppressed errors, rerun with: -v
==23779== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 6 from 6)
```

terminal

Valgrind EX2

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

int main(int argc, char** argv){
  int i;
  int a[10];
  for (i = 0; i < 9; i++)
    a[i] = i;

  for (i = 0; i < 10; i++){
    printf("%d ", a[i]);
  }
  printf("\n");
  return 0;
}
```

example2.c

attempting to print a[10] which is not an initialized value (array index out of bounds)

```
$ gcc -Wall -pedantic -g example2.c -o example2
$ valgrind ./example2
==24599== Memcheck, a memory error detector
==24599== Copyright (C) 2002-2009, and GNU GPL'd, by Julian Seward et al.
==24599== Using Valgrind-3.5.0 and LibVEX; rerun with -h for copyright info
==24599== Command: ./example2
==24599==
==24599== Conditional jump or move depends on uninitialised value(s)
==24599==    at 0x33A8648196: vfprintf (in /lib64/libc-2.13.so)
==24599==    by 0x33A864FB59: printf (in /lib64/libc-2.13.so)
==24599==    by 0x400567: main (example2.c:11)
==24599==
==24599== Use of uninitialised value of size 8
==24599==    at 0x33A864484B: _itoa_word (in /lib64/libc-2.13.so)
==24599==    by 0x33A8646D50: vfprintf (in /lib64/libc-2.13.so)
==24599==    by 0x33A864FB59: printf (in /lib64/libc-2.13.so)
==24599==    by 0x400567: main (example2.c:11)
==24599==
==24599== Conditional jump or move depends on uninitialised value(s)
==24599==    at 0x33A8644855: _itoa_word (in /lib64/libc-2.13.so)
==24599==    by 0x33A8646D50: vfprintf (in /lib64/libc-2.13.so)
==24599==    by 0x33A864FB59: printf (in /lib64/libc-2.13.so)
==24599==    by 0x400567: main (example2.c:11)
==24599==
0 1 2 3 4 5 6 7 8 7
==24599==
==24599== HEAP SUMMARY:
==24599==     in use at exit: 0 bytes in 0 blocks
==24599==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==24599==
==24599== All heap blocks were freed -- no leaks are possible
==24599==
==24599== For counts of detected and suppressed errors, rerun with: -v
==24599== Use --track-origins=yes to see where uninitialised values come from
==24599== ERROR SUMMARY: 3 errors from 3 contexts (suppressed: 6 from 6)
```

terminal

Multi-File C Programming

- You can split C into multiple files!
  - What if we wanted to use Linked List code in a different project?
  - If the linked list code is long, it can make files unwieldy
  - What if we want to separate our "main" from the struct definitions
- Pass all ".c" files into gcc:

```
gcc -o try_lists ll.c main.c
```

Must include code header files to enable one file to see the other, otherwise you have linking errors

```
$ gcc -g -Wall -o try_lists ll.c main.c
main.c: In function 'main':
main.c:5:5: error: unknown type name 'Node'
    5 |     Node *n1 = make_node(4, NULL);
      |     ^~~~
main.c:5:16: warning: implicit declaration of function 'make_node' [-Wimplicit-function-declaration]
    5 |     Node *n1 = make_node(4, NULL);
      |                ^~~~~~~~~
```

Sharing code across files

- Must always declare a function or struct in every file it's used in
  - Thank goodness C lets us separate declarations and definitions ;)
  - Include function header as definition

```
Node *make_node (int value, Node *next);
```

  - Include struct type definition

```
typedef struct Node
{
    int value;
    struct Node *next;
} Node;
```

```c
#include <stdlib.h>

typedef struct Node {
    int value;
    struct Node *next;
} Node;

Node *make_node(int value, Node *next);

Node *make_node(int value, Node *next) {
    Node *node = (Node*) malloc(sizeof(Node));
    node->value = value;
    node->next = next;
    return node;
}
```

ll.c

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

typedef struct Node {
    int value;
    struct Node *next;
} Node;

Node *make_node(int value, Node
*next);

int main() {
    Node *n1 = make_node(4, NULL);
    Node *n2 = make_node(7, n1);
    Node *n3 = make_node(3, n2);

    // rest of main…
}
```

main.c

Header Files

- Copying your function declarations to every file you want to use them is not fun
  - If you forget to make a change to all of them, confusing errors occur!
- A **header file** (.h) is a file which contains just *declarations*
- #include inserts the contents of a header file into your .c file
  - Put declarations in a header, then include it in all other files
  - Two types of #include

`#include <stdio.h>`

  - Used to include external libraries. Does not look for other files that you created.

`#include "myfile.h"`

  - Used to include your own headers. Searches in the same folder as the rest of your code.

```c
typedef struct Node {
    int value;
    struct Node *next;
} Node;

Node *make_node(int value, Node *next);
```

ll.h

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

#include "ll.h"

Node *make_node(int value, Node *next) {
    Node *node = (Node*) malloc(sizeof(Node));
    node->value = value;
    node->next = next;
    return node;
}
```

ll.c

```c
#include "ll.h"

int main() {
    Node *n1 = make_node(4, NULL);
    Node *n2 = make_node(7, n1);
    Node *n3 = make_node(3, n2);

    // rest of main…
}
```

main.c

Header Guards

- Consider the following header structure:
  - Header A includes header B.
  - Header C includes header B.
  - A source code file includes headers A and C.
  - The code now includes two copies of header B!
  - Solution: "**header guard**"
    - Uses ifndef to check if header is already defined for this file

```c
#ifndef LL_H
#define LL_H

typedef struct Node {
    int value;
    struct Node *next;
} Node;

Node *make_node(int value, Node *next);
#endif
```

ll.h

```c
#include "ll.h"

int main() {
    Node *n1 = make_node(4, NULL);
    Node *n2 = make_node(7, n1);
    Node *n3 = make_node(3, n2);

    // rest of main…
}
```

main.c

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

#include "ll.h"

Node *make_node(int value, Node *next) {
    Node *node =
(Node*)malloc(sizeof(Node));
    node->value = value;
    node->next = next;
    return node;
}
```

ll.c

Make Files

- **Make** is a program which automates building **dependency trees**
  - List of **rules** written in a **Makefile** declares the commands which build each intermediate part
  - Helps you avoid manually typing gcc commands, easier and less prone to typos
  - Automates build process
- Makefiles are a list of with **Make rules** which include:
  - **Target** - An output file to be generated, dependent on one or more sources
  - **Source** – Input source code to be built
  - **Recipe** - command to generate target

Make Rule Syntax:

```
target: source
    recipe
```

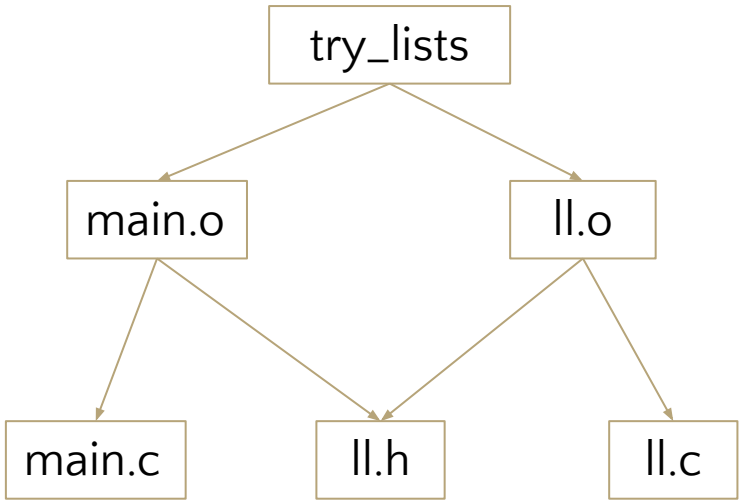
tab not spaces!

```
ll.o: ll.c ll.h
    gcc -c ll.c
```

- Makefile logic
  - Make builds based on structural organization of how code depends on other code as defined by includes
  - Recursive – if a source is also a target for other sources, must also evaluate its dependencies and remake as required
  - Make can check when you've last edited each file, and only build what is needed!
    - Files have "last modification date". make can check whether the sources are more recent than the target
  - Make isn't language specific: recipe can be any valid shell command
- run `make` command from within same folder
  - `$ make [ -f makefile] [ options ] … [ targets ] ../`
  - Starts with first rule in file then follows dependency tree
  - -f specifies makefile name, if non provided will default to "Makefile"
  - if no target is specified will default to first listed in file

https://www.gnu.org/software/make/manual/make.html#Introduction

Example Makefile



variable definitions

must include rules for each non .h file

rules define dependency hierarchy

```
CC = gcc
CGLAGS = -g -Wall -std=c11

try_lists: main.o ll.o
    $(CC) $(CFLAGS) -o try_lists main.o ll.o

main.o: main.c ll.h
    $(CC) $(CFLAGS) -c main.c

ll.o: ll.c ll.h
    $(CC) $(CFILES) -c ll.c
```

Makefile

Example

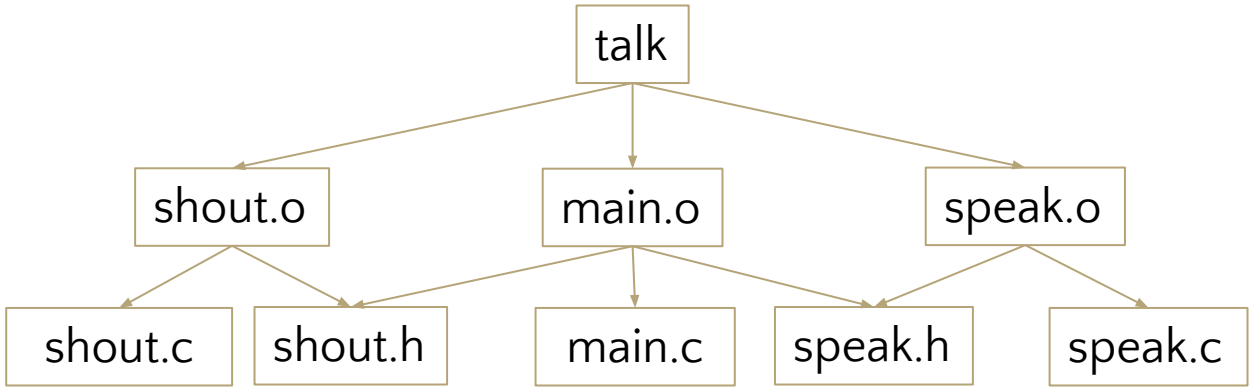
```
#ifndef SHOUT_H
#define SHOUT_H
/* Write message m in uppercase to stdout */
void shout(char m[]);
#endif /* ifndef SHOUT_H */
```
shout.h

```
#ifndef SPEAK_H
#define SPEAK_H
/* Write message m to stdout */
void speak(char m[]);
#endif /* ifndef SPEAK_H */
```
speak.h

```
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include "speak.h"
#include "shout.h"
/* Write message m in uppercase to stdout */
void shout(char m[])
{
   int len; /* message length */
   char *mcopy; /* copy of original message */
   int i;
   len = strlen(m);
   mcopy = (char *)malloc(len*sizeof(char)+1);
   strcpy(mcopy,m);
   for (i = 0; i < len; i++)
      mcopy[i] = toupper(mcopy[i]);
   speak(mcopy); free(mcopy);
}
```
shout.c

```
#include "speak.h"
#include "shout.h"
/* Say HELLO and goodbye */
int main(int argc, char* argv[])
{
   shout("hello");
   speak("goodbye");
   return 0;
}
```
main.c

```
#include <stdio.h>
#include "speak.h"
/* Write message m to stdout */
void speak(char m[])
{
   printf("%s\n", m);
}
```
speak.c




```
all: talk
# The executable
talk: main.o speak.o shout.o
   gcc -Wall -std=c11 -g -o talk main.o speak.o shout.o

# Individual source files
speak.o: speak.c speak.h
   gcc -Wall -std=c11 -g -c speak.c
shout.o: shout.c shout.h speak.h
   gcc -Wall -std=c11 -g -c shout.c
main.o: main.c speak.h shout.h
   gcc -Wall -std=c11 -g -c main.c

# A "phony" target to remove built files and backups
clean: rm -f *.o talk *~
```
Makefile

Hello World C++ ostrem

helloworld.cc

```
#include <iostream>    // for cout, endl
#include <cstdlib>     // for EXIT_SUCCESS

int main(int argc, char** argv) {
  std::cout << "Hello, World!" << std::endl;
  return EXIT_SUCCESS;
}
```

- `ostream` has many different methods to handle <<
  - The functions differ in the type of the right-hand side (RHS) of <<

e.g. if you do `std::cout << "foo";` , then C++ invokes `cout`'s function to handle << with RHS `char*`

- The `ostream` class' member functions that handle << return *a reference to themselves*
  - When `std::cout << "Hello, World!";` is evaluated:
    - A member function of the `std::cout` object is invoked
    - It buffers the string `"Hello, World!"` for the console

And it returns a reference to `std::cout`

Namespaces

A **namespace** is a declarative region that provides a scope to the identifiers (the names of types, functions, variables, etc) inside it.

- used to organize code into logical groups and to prevent name collisions that can occur especially when your code base includes multiple libraries

- allow us to group named entities that otherwise would have global scope into narrower scopes, giving them namespace scope. This allows organizing the elements of programs into different logical scopes referred to by names.

```
// A program to demonstrate need of
namespace
int main()
{
    int value;
    value = 0;
    double value; // Error here
    value = 0.0;
}
```

```
Compiler Error:
  'value' has a previous declaration as 'int value'
```

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

// Variable created inside namespace
namespace first
{
    int val = 500;
}
// Global variable
int val = 100;

int main()
{
    // Local variable
    int val = 200;

    // These variables can be accessed from outside the namespace
    // using the scope operator ::
    cout << first::val << '\n';

    return 0;
}
```

https://www.geeksforgeeks.org/namespace-in-c/

Declaring Namespaces

- Namespace declarations appear only at global scope.
- Namespace declarations can be nested within another namespace.
- Namespace declarations don't have access specifiers. (Public or private)
- No need to give semicolon after the closing brace of definition of namespace.
- We can split the definition of namespace over several units.

```
namespace namespace_name
{
   int x, y; // code declarations where
             // x and y are declared in
             // namespace_name's scope
}
```

```
// Creating namespaces
#include <iostream>
using namespace std;
namespace ns1
{
    int value()  { return 5; }
}
namespace ns2
{
    const double x = 100;
    double value() {  return 2*x; }
}

int main()
{
    // Access value function within ns1
    cout << ns1::value() << '\n';

    // Access value function within ns2
    cout << ns2::value() << '\n';

    // Access variable x directly
    cout << ns2::x << '\n';

    return 0;
}
```

```
Output:
5
200
100
```

new / delete

- To allocate on the heap using C++, you use the `new` keyword instead of `malloc()` from `stdlib.h`
  - You can use new to allocate an object (*e.g.* new Point)
  - You can use new to allocate a primitive type (*e.g.* new int)
- To deallocate a heap-allocated object or primitive, use the `delete` keyword instead of `free()` from `stdlib.h`
  - Don't mix and match!
    - *Never* **free**() something allocated with new
    - *Never* delete something allocated with **malloc**()
    - Careful if you're using a legacy C code library or module in C++

```
int* AllocateInt(int x) {
  int* heapy_int = new int;
  *heapy_int = x;
  return heapy_int;
}
```

```
Point* AllocatePoint(int x, int y) {
  Point* heapy_pt = new Point(x,y);
  return heapy_pt;
}
```

heappoint.cpp

```
#include "Point.h"

// definitions of AllocateInt() and AllocatePoint()

int main() {
  Point* x = AllocatePoint(1, 2);
  int* y = AllocateInt(3);

  cout << "x's x_ coord: " << x->get_x() << endl;
  cout << "y: " << y << ", *y: " << *y << endl;

  delete x;
  delete y;
  return EXIT_SUCCESS;
}
```

Pointers in C++

- Work the same as in C, hooray!
- A **pointer** is a variable containing an address
  - Modifying the pointer *doesn't* modify what it points to, but you can access/modify what it points to by *dereferencing*

```
int main(int argc, char** argv) {
  int x = 5, y = 10;
  int* z = &x;

  *z += 1;  // sets x to 6
   x += 1;  // sets x (and *z) to 7

   z = &y;  // sets z to the address of y
  *z += 1;  // sets y (and *z) to 11

  return EXIT_SUCCESS;
}
```

References in C++

- A **reference** is an alias for another variable
  - *Alias*: another name that is bound to the aliased variable
  - Mutating a reference ***is*** mutating the aliased variable
  - Introduced in C++ as part of the language

```
int main(int argc, char** argv) {
  int x = 5, y = 10;
  int& z = x;  // binds the name "z" to x

  z += 1;  // sets z (and x) to 6
  x += 1;  // sets x (and z) to 7

  z  = y;  // sets z (and x) to the value of y
  z += 1;  // sets z (and x) to 11

  return EXIT_SUCCESS;
}
```

Pass by Reference

C++ allows you to use real **pass-by-reference**

- Client passes in an argument with normal syntax
  - Function uses reference parameters with normal syntax
  - Modifying a reference parameter modifies the caller's argument!

```
void swap(int& x, int& y) {
  int tmp = x;
  x = y;
  y = tmp;
}

int main(int argc, char** argv) {
  int a = 5, b = 10;

  swap(a, b);
  cout << "a: " << a << "; b: " << b << endl;
  return EXIT_SUCCESS;
}
```

- In C all function arguments are copies
- pointer arguments pass a copy of the address value, original values will be unaffected by changes to parameter

- A stylistic choice, not mandated by the C++ language
- Google C++ style guide suggests:
  - Input parameters:
    - Either use values (for primitive types like int or small structs/objects)
    - Or use const references (for complex struct/object instances)
  - Output parameters:
    - Use unchangeable pointers referencing changeable data
  - Ordering:
    - List input parameters first, then output parameters last

Classes in C++

- Unlike C structs
  - Class definition is part of interface and should go in .h file
    - Private members still must be included in definition (!)
  - Typically put member function definitions into companion .cpp file with implementation details
    - Common exception: setter and getter methods
  - These files can also include non-member functions that use the class
- Like java
  - Fields & methods, static vs instance, constructors
  - method overloading (functions, operators and constructors)
- Not quite like Java
  - access-modifier (eg private) syntax
  - declaration separate from implementation (like C)
  - funny constructor syntax, default parameters (eg, …=0)
- Not at all like Java
  - you can name files anything you want
    - Typically a combination of Name.cpp and Name.h for class Name
  - destructors and copy constructors
  - virtual vs non-virtual

MyClass.h

```
namespace mynamespace {
   class MyClass {
      private:
         type fieldOne;
         type fieldTwo;

      public:
         MyClass();
         MyClass(type, type);

      public:
         type functionOne() {
            // function definition
         }
         type functionTwo() {
            // function definition
         }
   };
}
```

Defining Classes in C++

- Class Definition (in a .h file)

Name.h

```
class Name {
 public:
  // public member definitions & declarations go here

 private:
  // private member definitions & declarations go here
};  // close class Name
```

- Class Member Definition (in a .cpp file)

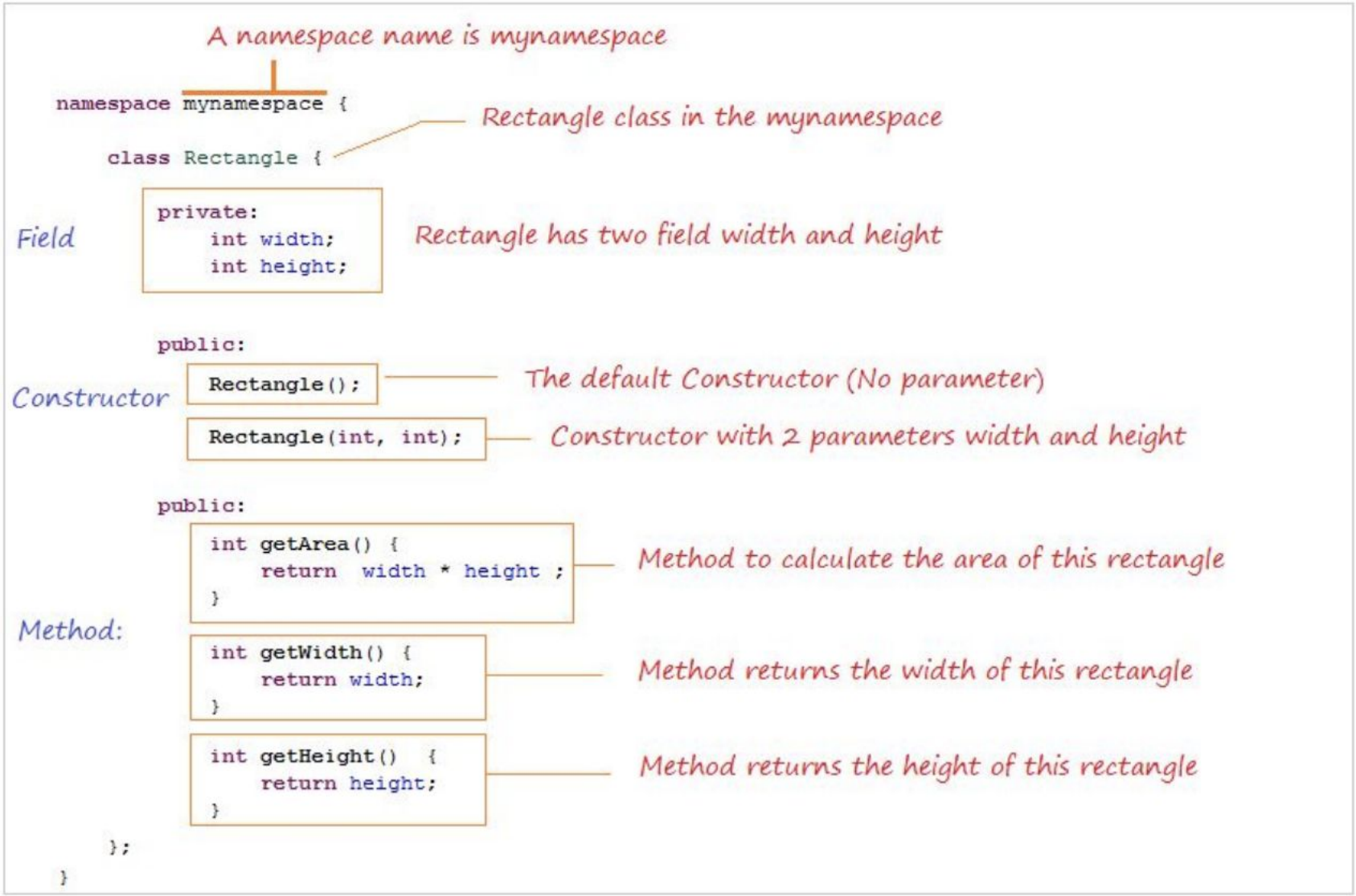
Name.cpp

```
returnType ClassName::MethodName(type1 param1, …, typeN paramN) {
  // body statements
}
```

- Members can be functions (methods) or data (variables)
- (1) *define* within the class definition OR (2) *declare* within the class definition and then *define* elsewhere

Anatomy of C++ Class

Rectangle.h



Constructors in C++

- A constructor (ctor) initializes a newly-instantiated object
  - A class can have multiple constructors that differ in parameters
    - Which one is invoked depends on *how* the object is instantiated
- Written with the class name as the method name:

```
Point(const int x, const int y);
```

  - C++ will automatically create a synthesized default constructor if you have *no* user-defined constructors
    - Takes no arguments and calls the default ctor on all non-"plain old data" (non-POD) member variables
    - Synthesized default ctor will fail if you have non-initialized const or reference data members
- 4 different types of constructors
  - default constructor – takes zero arguments. If you don't define any constructors the compiler will generate one of these for you (just like Java)
  - copy constructor – takes a single parameter which is a *const reference*(`const T&`) to another object of the same type, and initializes the fields of the new object as a *copy* of the fields in the referenced object
  - user-defined constructors – initialize fields and take whatever arguments you specify
  - conversion constructors – implicit, take a single argument. If you want a single argument constructor that is not implicit must use the keyword "explicit" like: `explicit String(const char* raw);`

Class Derivation List

- Comma-separated list of classes to inherit from:

```
#include "BaseClass.h"

class Name : public BaseClass {
  ...
};
```

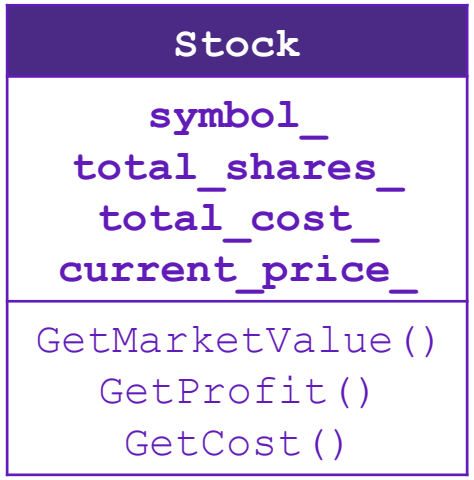
  - Focus on single inheritance, but *multiple inheritance* possible

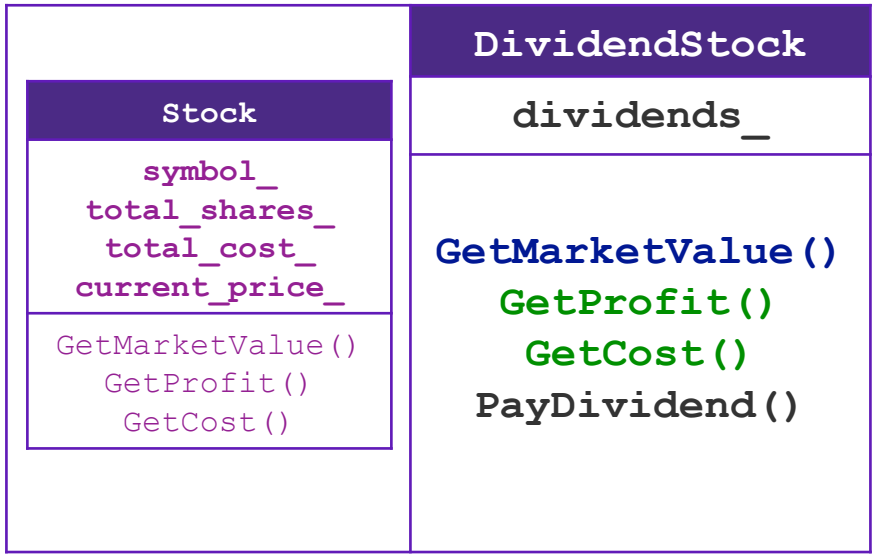
```
#include "BaseClass.h"
#include "BaseClass2.h"
class Name : public BaseClass, public BaseClass2 {
  ...
};
```

- Almost always use "public" inheritance
  - Acts like extends does in Java
  - Any member that is non-private in the base class is the same in the derived class; both *interface and implementation inheritance*
    - Except that constructors, destructors, copy constructor, and assignment operator are *never* inherited

- `public`: visible to all other classes
- `protected`: visible to current class and its derived classes
- `private`: visible only to the current class
- Use `protected` for class members only when:
  - Class is designed to be extended by derived classes
  - Derived classes must have access but clients should not be allowed

Inheritance Design Example: Stock Portfolio





A derived class:

- **Inherits** the behavior and state (specification) of the base class
- **Overrides** some of the base class' member functions (opt.)
- **Extends** the base class with new member functions, variables (opt.)

Polymorphism in C++

- In Java: `PromisedType var = new ActualType();`
  - var is a reference (different term than C++ reference) to an object of ActualType on the Heap
  - ActualType must be the same class or a subclass of PromisedType
- In C++: `PromisedType* var_p = new ActualType();`
  - var_p is a *pointer* to an object of ActualType on the Heap
  - ActualType must be the same or a derived class of PromisedType
  - (also works with references)
  - PromisedType defines the *interface* (*i.e.* what can be called on var_p), but ActualType may determine which *version* gets invoked

```
#ifndef BANKACCOUNT_H
#define BANKACCOUNT_H

#include <iostream>

namespace bank {

  class BankAccount {
    public:
      explicit BankAccount(const std::string& accountHolder);
      BankAccount(const BankAccount& other) = delete;

      // Accessors
      int getBalance() const;
      int getAccountId() const;
      const std::string& getAccountHolder() const;

      // Modifier - add money.
      void deposit(int amount);

      // different for every type of account,
      // require derived classes to implement
      virtual void withdraw(int amount) = 0;

    protected:
      // derived classes can modify the balance.
      void setBalance(int balance);

    private:
      const std::string accountHolder_;
      const int accountId_;
      int balance_;

      static int accountCount_;
  };
}
#endif
```

BankAccount.cc

```
#ifndef SAVINGSACCOUNT_H
#define SAVINGSACCOUNT_H

#include "BankAccount.h"

namespace bank {

  class SavingsAccount : public BankAccount {
    public:
      SavingsAccount(double interestRate, std::string name);

       double getInterestRate() const;

       virtual void withdraw(int amount) override;

    private:
      bool isNewMonth(time_t* curTime);

      double interestRate_;
      time_t lastMonth_;
      int numTransactionsInMonth_;
  };
}

#endif
```

SavingsAccount.cc

Self Check

```
#include <iostream>

using namespace std;

class A {
   public:
     A() { cout << "a()" << endl; }
     ~A() { cout << "~a" << endl; }
     void m1() { cout << "a1" << endl; }
     void m2() { cout << "a2" << endl; }
};

// class B inherits from class A
class B : public A {
   public:
     B() { cout << "b()" << endl; }
     ~B() { cout << "~b" << endl; }
     void m2() { cout << A::m2() << "b2" << endl; }
     void m3() { cout << "b3" << endl; }
};

int main() {
  A* x = new B();
  x->m1();
  x->m2();
  x->m3();
  delete x;
}
```

b()

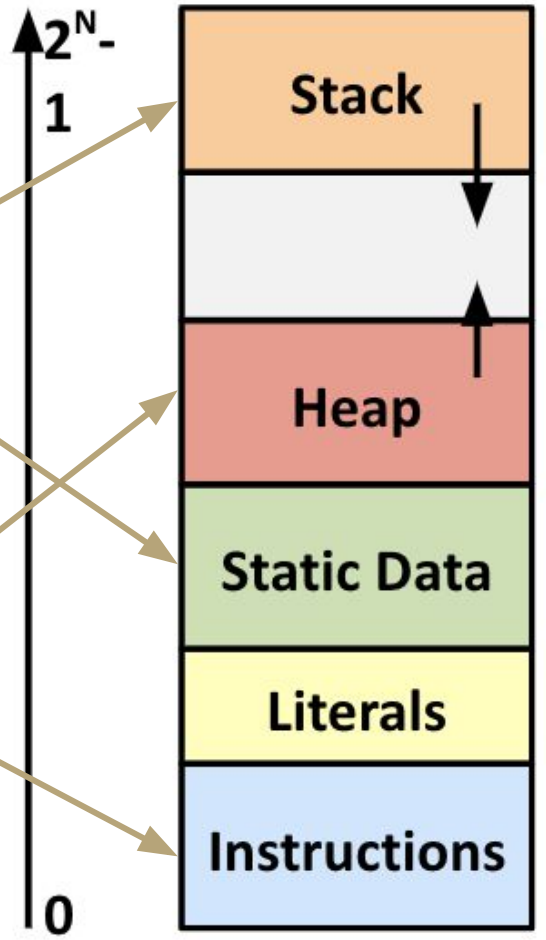
m1. a1

m2. a2
b2

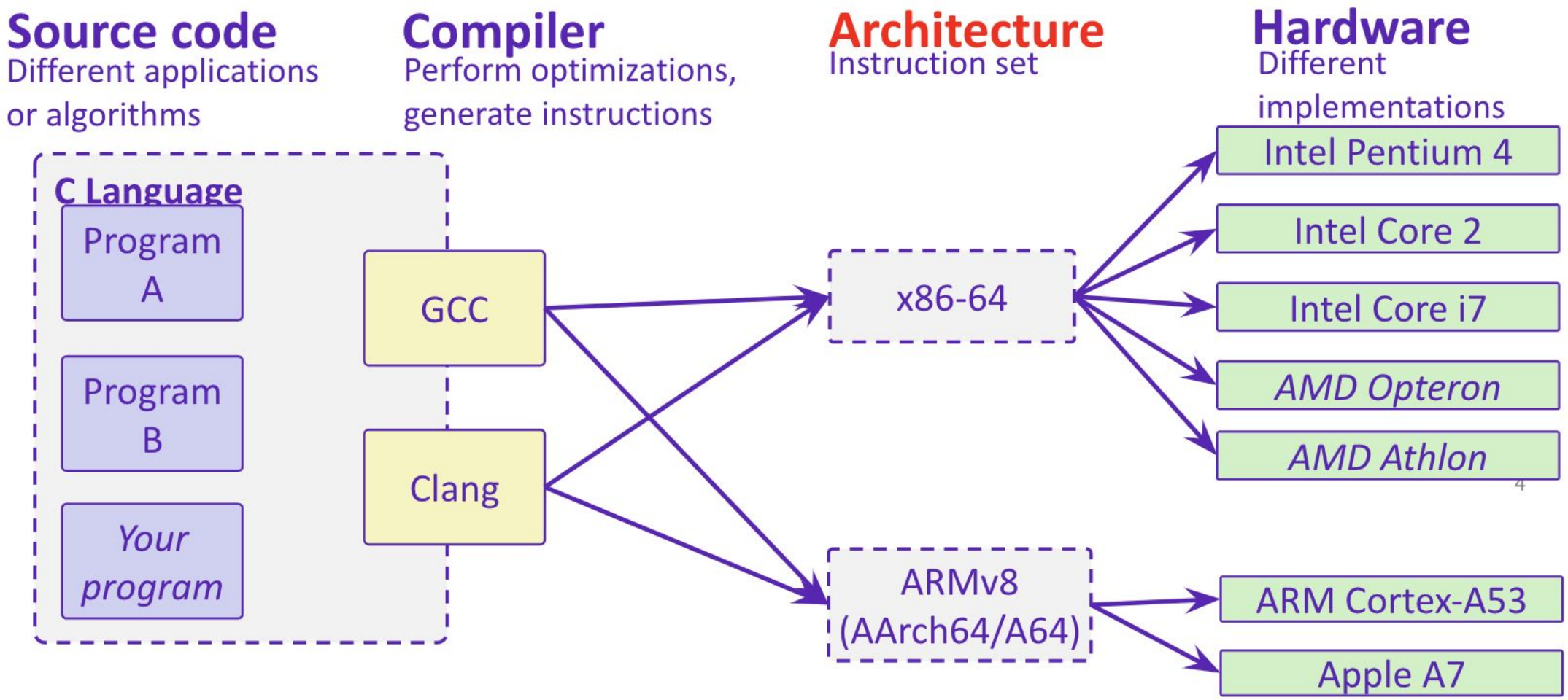
m3.
b3

Where does everything go?

```
char big_array[1L<<24];  /* 16 MB */
char huge_array[1L<<31]; /*  2 GB */

int global = 0;

int useless() { return 0; }

int main()
{
    void *p1, *p2, *p3, *p4;
    int local = 0;
    p1 = malloc(1L << 28); /* 256 MB */
    p2 = malloc(1L << 8);  /* 256  B */
    p3 = malloc(1L << 32); /*   4 GB */
    p4 = malloc(1L << 8);  /* 256  B */
    /* Some print statements ... */
}
```



Hardware Software Interface



Registers

- A location in the CPU that stores a small amount of data, which can be accessed very quickly (once every clock cycle)
- Registers have *names*, not *addresses*
  - In assembly, they start with % (*e.g.* %rsi)
- Registers are at the heart of assembly programming
  - They are a precious commodity in all architectures, but *especially* x86

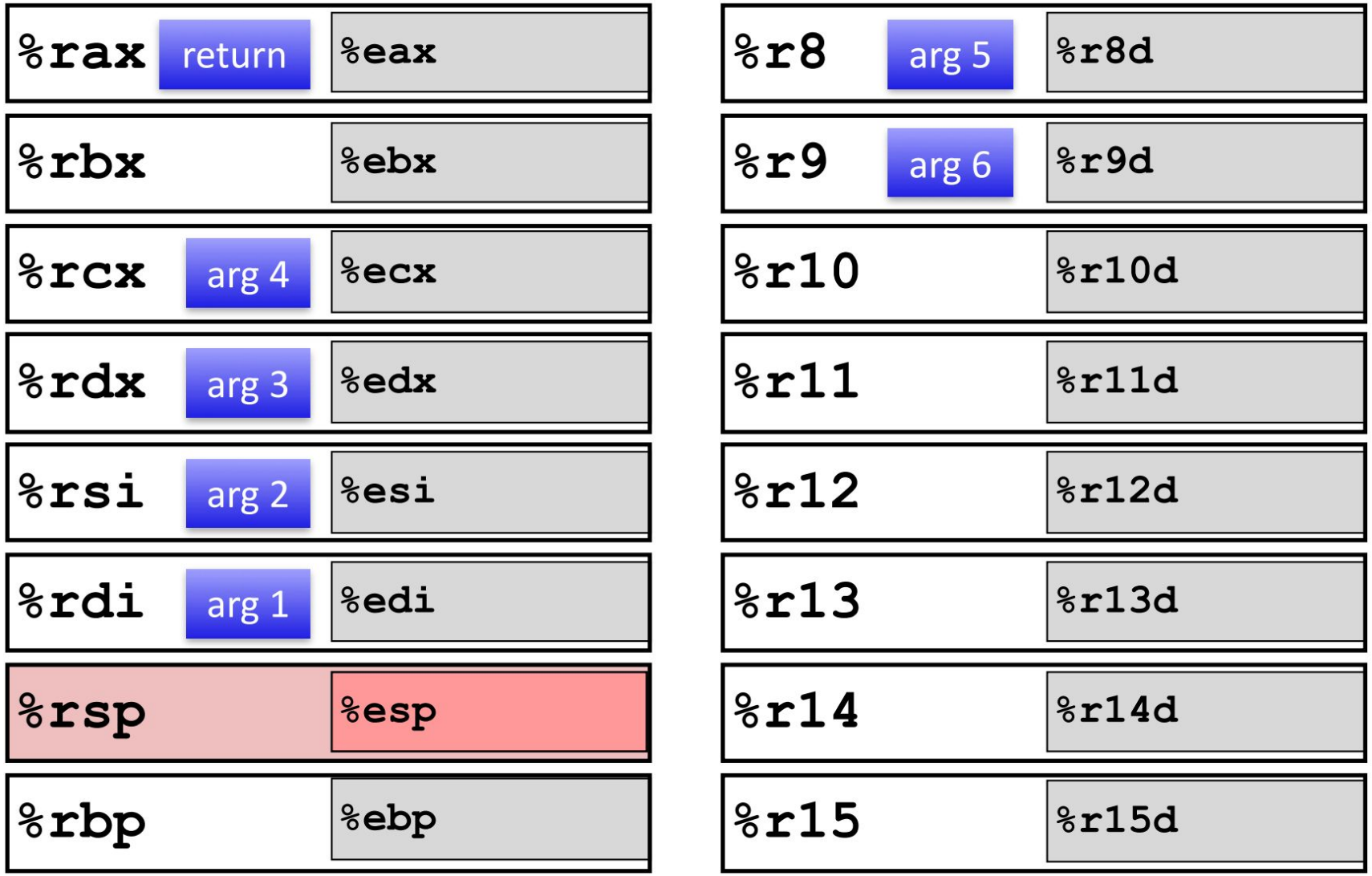
Memory

- Addresses (EX: 0x7FFFD024C3DC)
- Big ~ 8 GiB
- Slow ~50-100 ns
- Dynamic – Can "grow" as needed while program runs

Registers

- Names (EX: %rdi)
- Small – (16 x 8 B) = 128 B
- Fast – sub-nanosecond timescale
- Static – fixed number in hardware

Special Registers for Intel x86-64



Assembly Instruction Basics

Assembly instructions fall into one of 3 categories:

- **Transfer data** between memory and register
  - Load data from memory into register
    - %reg = Mem[address]
  - Store register data into memory
    - Mem[address] = %reg
- **Perform arithmetic** operation on register or memory data
  - c = a + b; z = x << y; i = h & g;
- **Control flow**: what instruction to execute next
  - Unconditional jumps to/from procedures
  - Conditional branches

Items in Assembly fall into one of 3 operand categories:

- **Immediate**: Constant integer data
  - Examples: $0x400, $-533
  - Like C literal, but prefixed with '$'
  - Encoded with 1, 2, 4, or 8 bytes
- **Register**: 1 of 16 integer registers
  - Examples: %rax, %r13
- **Memory**: Consecutive bytes of memory at a computed address
  - Simplest example: (%rax)

Example: Moving Data

- General form: `mov_ source, destination`
  - Missing letter (_) specifies size of operands
  - Lots of these in typical code

Examples:

- `movb src, dst`
  - Move 1-byte "byte"
- `movw src, dst`
  - Move 2-byte "word"
- `movl src, dst`
  - Move 4-byte "long word"
- `movq src, dst`
  - Move 8-byte "quad word"

Assume we have two variables called rax and rdx.

Which assembly instruction does `*rdx = rax`?

```
movq %rdx, %rax
movq (%rdx), %rax
movq %rax, (%rdx)
movq (%rax), %rdx
```

	Source	Dest	Src, Dest	C Analog
movq	Imm	Reg	`movq $0x4, %rax`	`rax = 4;`
		Mem	`movq $-147, (%rax)`	`*rax = -147;`
	Reg	Reg	`movq %rax, %rdx`	`rdx = rax;`
		Mem	`movq %rax, (%rdx)`	`*rdx = rax;`
	Mem	Reg	`movq (%rax), %rdx`	`rdx = *rax;`

* parentheses around a register dereference

Example: Arithmetic Operations

```c
long simple_arith(long x, long y)
{
  long t1 = x + y;
  long t2 = t1 * 3;
  return t2;
}
```

C

Register	Use(s)
%rdi	1st argument (x)
%rsi	2nd argument (y)
%rax	return value

```c
y += x;
y *= 3;
long r = y;
return r;
```

C

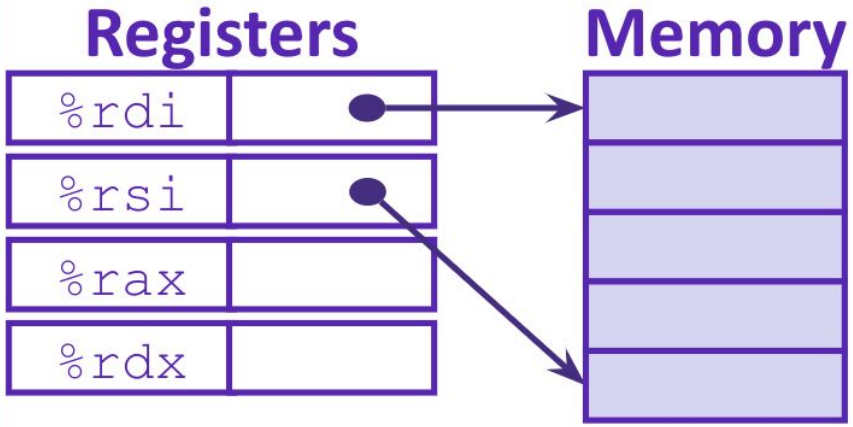
```
simple_arith:
  addq    %rdi, %rsi
  imulq     $3, %rsi
  movq    %rsi, %rax
  ret
```

Assembly

Example: swap()

Register	Use(s)
%rdi	1st argument (x)
%rsi	2nd argument (y)
%rax	return value

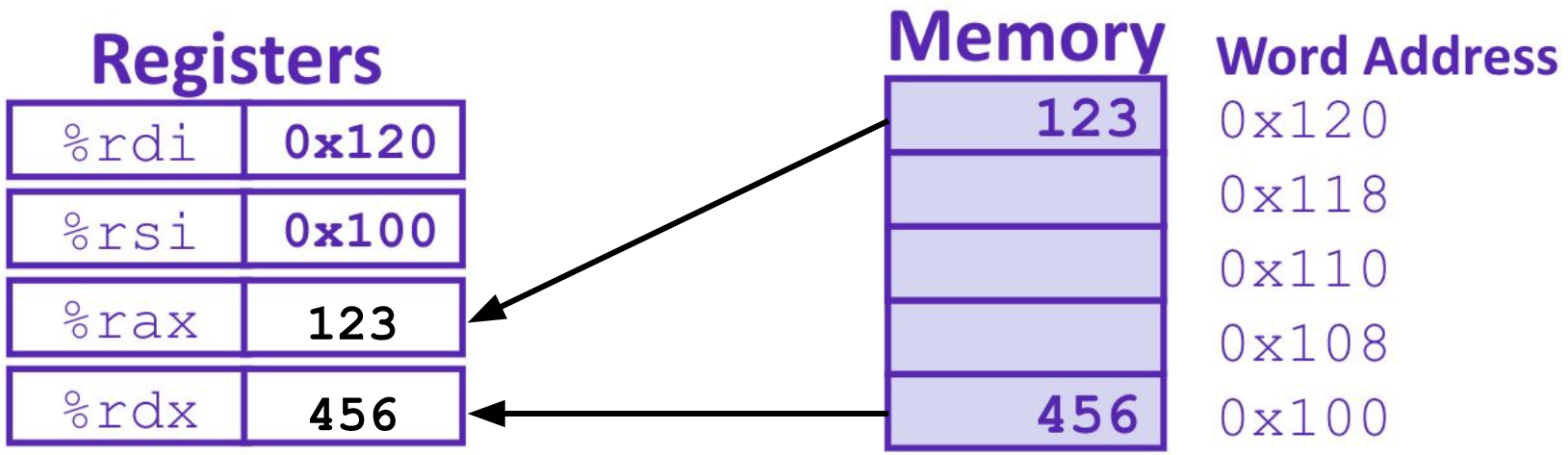
```
void swap(long *xp, long *yp){
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```




```
swap:
    movq   (%rdi), %rax
    movq   (%rsi), %rdx
    movq   %rdx, (%rdi)
    movq   %rax, (%rsi)
    ret
```

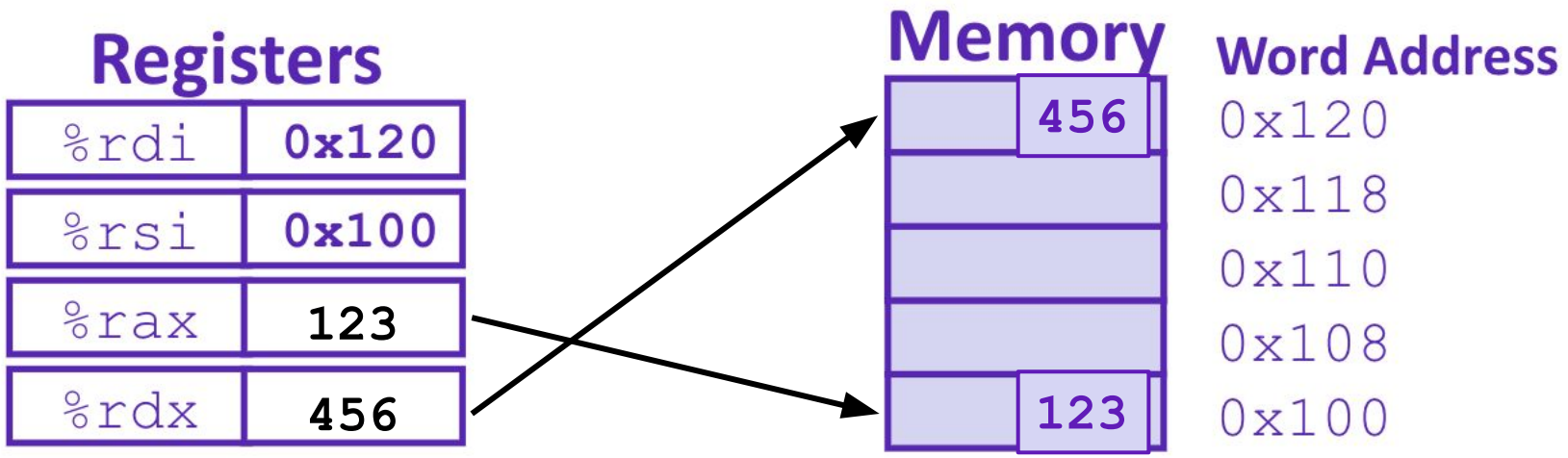
Register		Variable
%rdi	⇔	xp
%rsi	⇔	yp
%rax	⇔	t0
%rdx	⇔	t1

Example: swap()




```
swap:
    movq   (%rdi), %rax   #  t0 = *xp
    movq   (%rsi), %rdx   #  t1 = *yp
    movq   %rdx, (%rdi)   # *xp =  t1
    movq   %rax, (%rsi)   # *yp =  t0
    ret
```

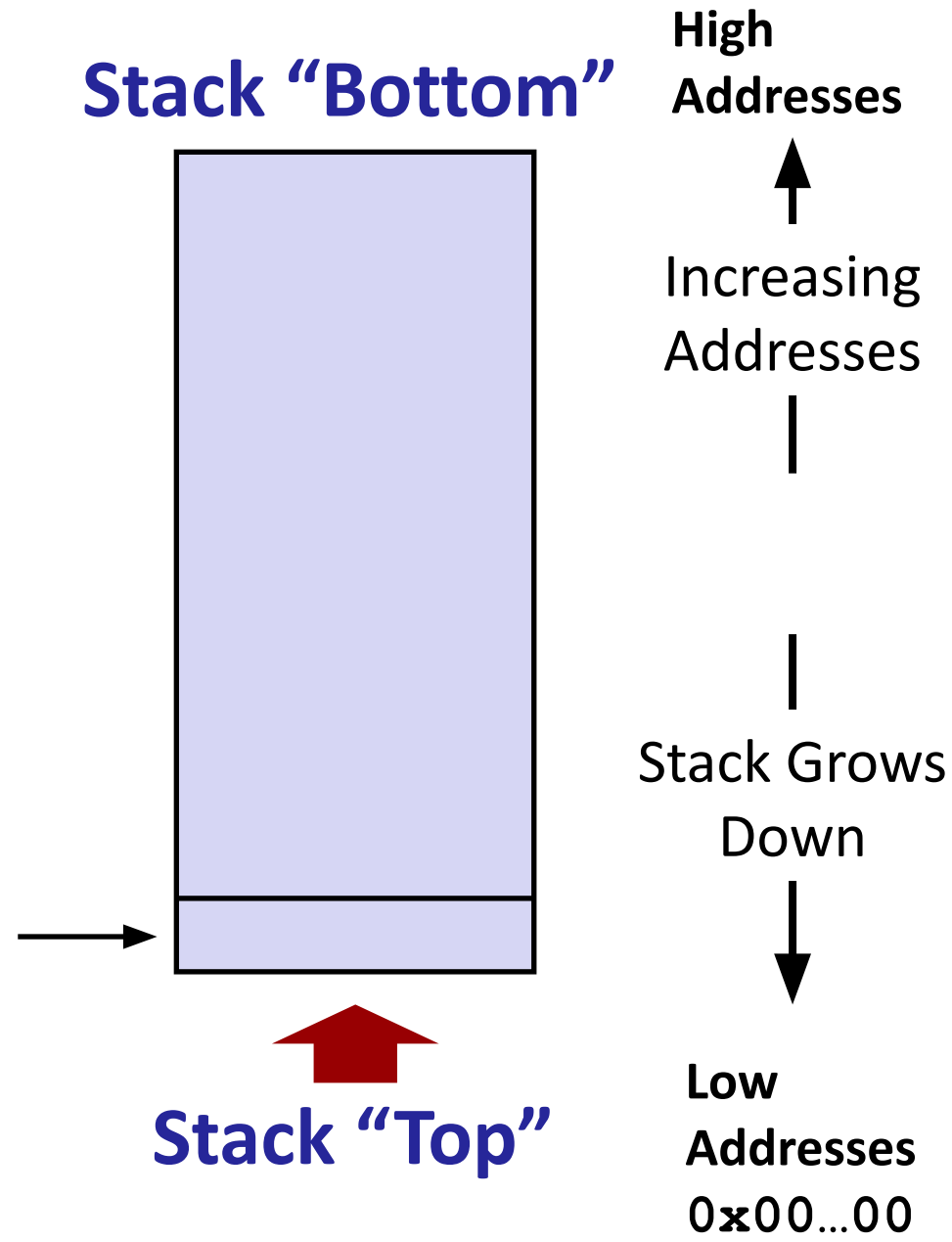
Example: swap()




```
swap:
    movq   (%rdi), %rax   #  t0 = *xp
    movq   (%rsi), %rdx   #  t1 = *yp
    movq   %rdx, (%rdi)   # *xp =  t1
    movq   %rax, (%rsi)   # *yp =  t0
    ret
```

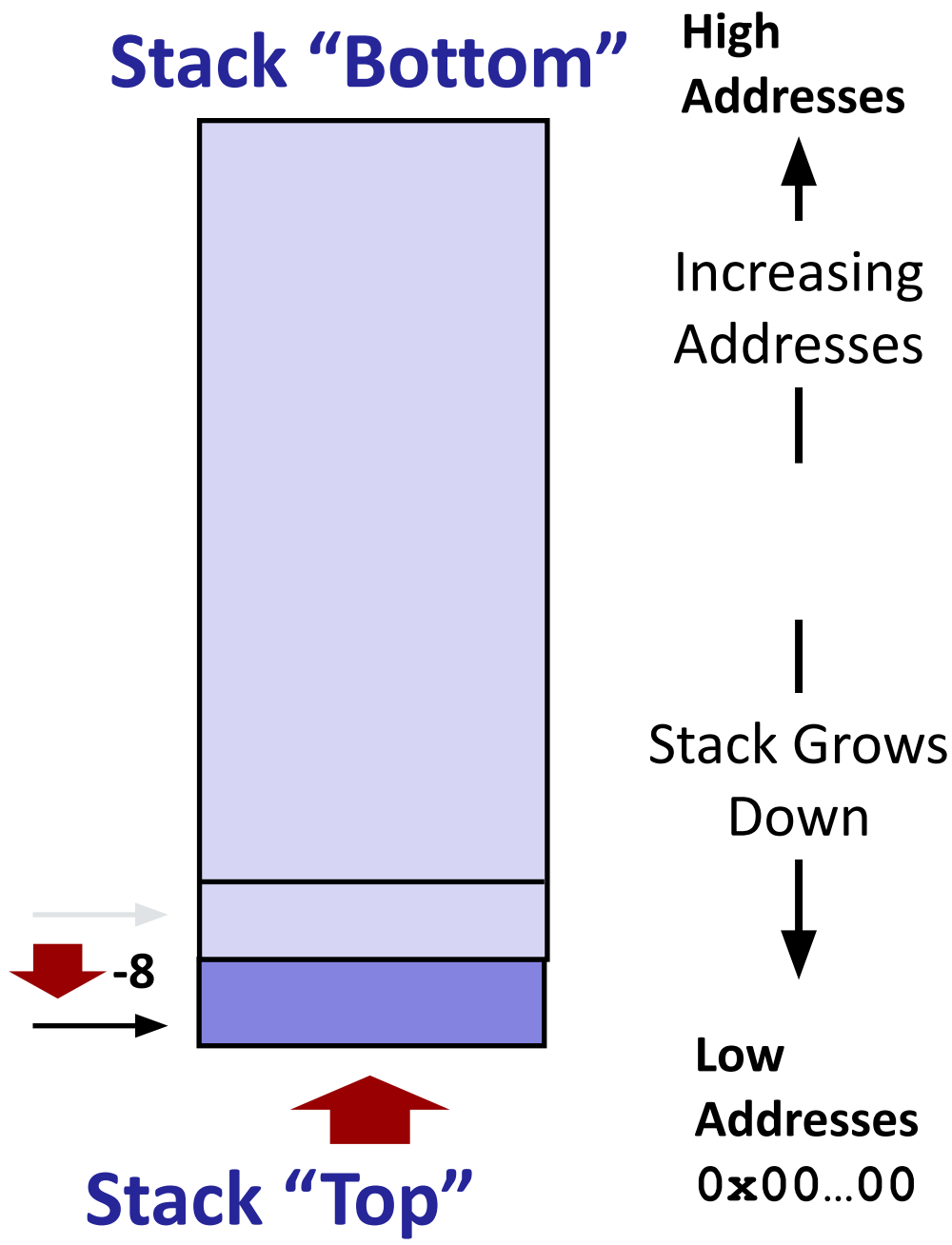
x86-64 Stack

- Region of memory managed with stack "discipline"
  - Grows toward lower addresses
  - Customarily shown "upside-down"
- Register `%rsp` contains *lowest* stack address
  - `%rsp` = address of *top* element, the most-recently-pushed item that is not-yet-popped



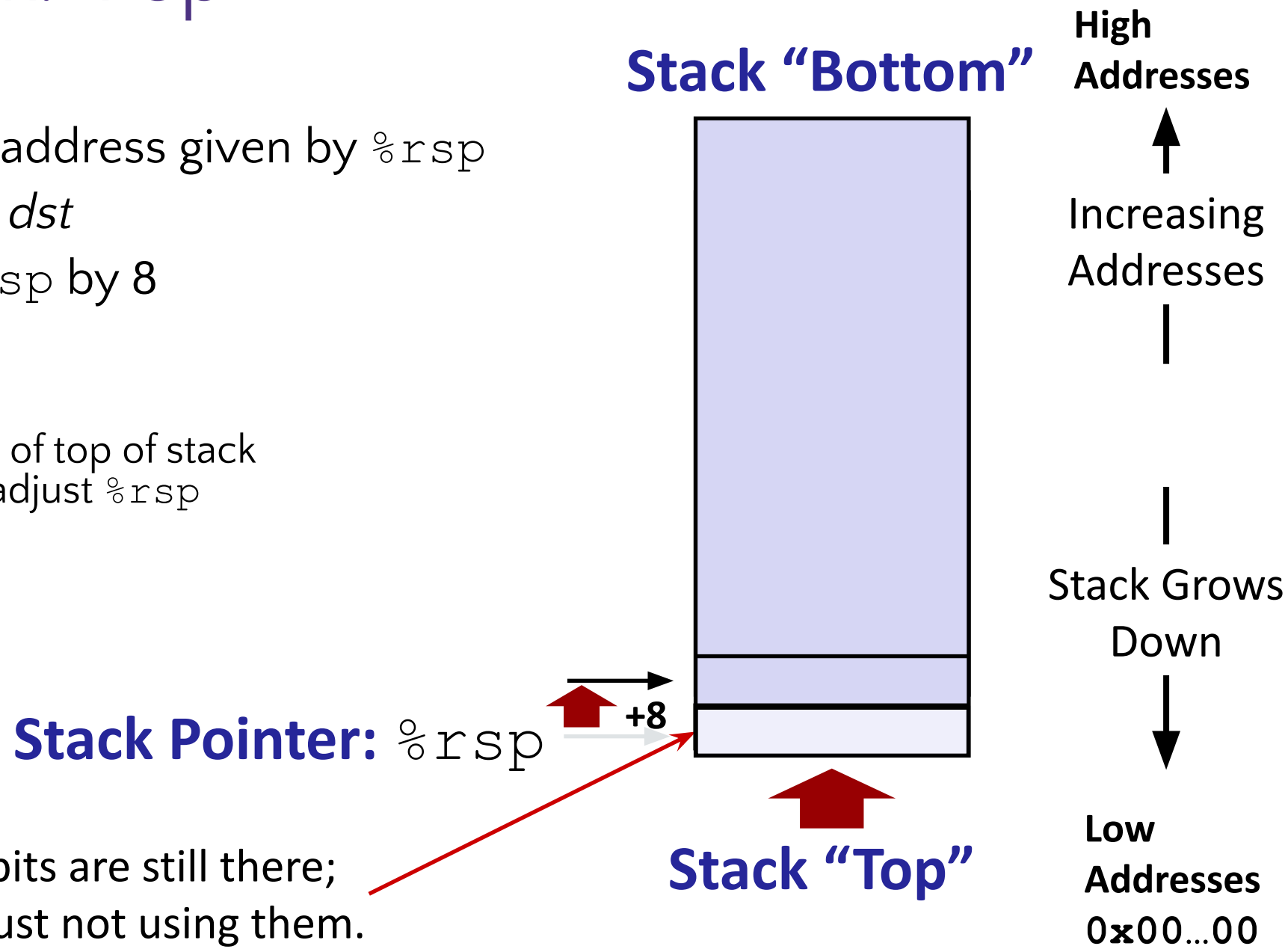
x86-64 Stack: Push

- `pushq` *src*
  - Fetch operand at *src*
    - *Src* can be reg, memory, immediate
  - ***Decrement*** `%rsp` by 8
  - Store value at address given by `%rsp`
- Example:
  - **`pushq %rcx`**
  - Adjust `%rsp` and store contents of `%rcx` on the stack



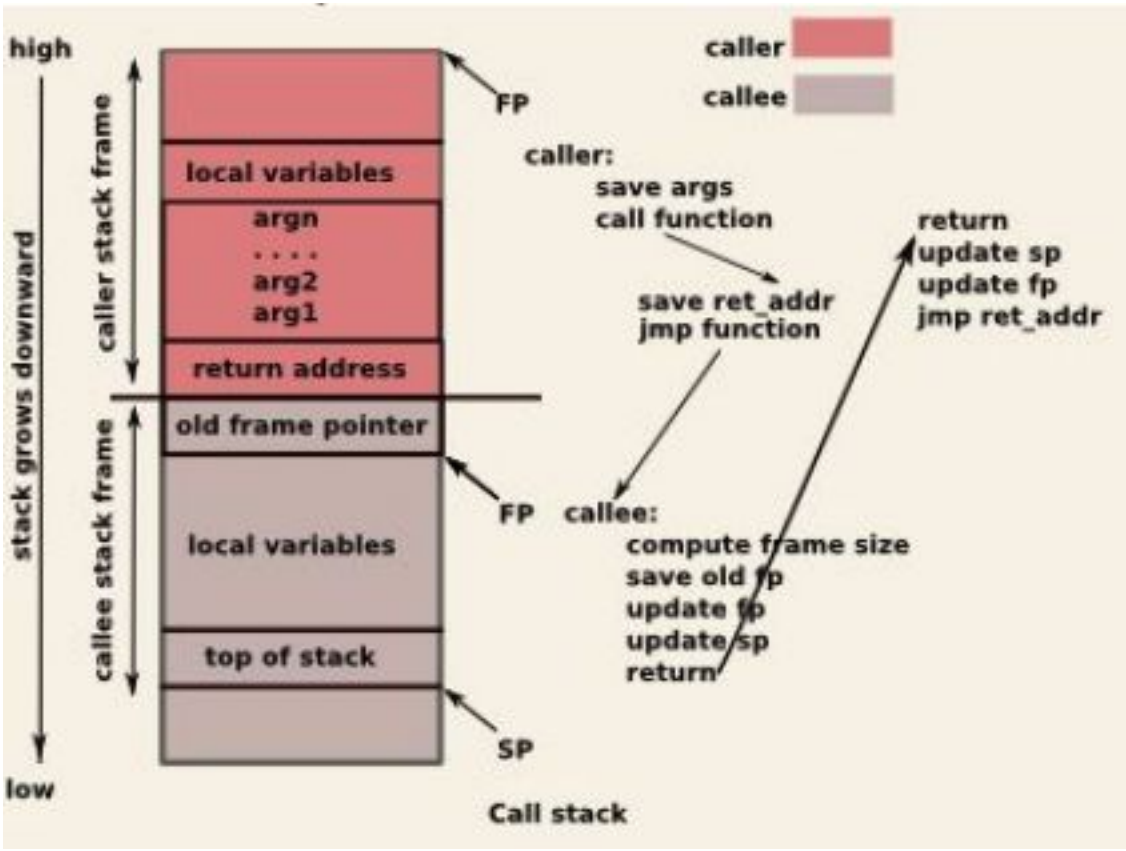
x86-64 Stack: Pop

- `popq` *dst*
  - Load value at address given by `%rsp`
  - Store value at *dst*
  - ***Increment*** `%rsp` by 8
- Example:
  - **`popq %rcx`**
  - Stores contents of top of stack into `%rcx` and adjust `%rsp`



Function Pointers & Frames

- Coded instructions are translated into numerical values stored in memory and fed into the processor for execution
- **function pointer** – address of a function stored in memory, pointing to the start of the block of memory storing the set of instructions expressed by the function.
- **stack frames** – section of the stack that is set aside for each function call
  - frame pushed onto the stack when the function is called and popped off when the function returns.
  - each frame contains: arguments, return address, pointer to last frame, local variables



Assembly example question

Consider the following x86-64 assembly instructions:

```
mystery:
    movl $0, %edx
    movl $0, %eax
.L3
    cmpl %esi, %edx
    jge .L1
    movslq %edx, %rcx
    addl (%rdi, %rcx, 4), %eax
    addl $1, %edx
    jmp .L3
.L1
    rep ret
```

what variable type would %rdi be in the corresponding C program?

```
int*
```

what variable type would %edx be in the corresponding C program?

```
int
```

Careers in Tech

College Career Timeline

Freshman	**Things To Think About** Pay attention to what you like and don't like.	**Actions to Take** Be kind to yourself.
Sophomore	**Things To Think About** How are you incorporating your interests into your extracurricular activities?	**Actions to Take** Take Data Structures. Find a technical experience outside of taking class.
Junior	**Things To Think About** What type of role and what type of company do you want after college?	**Actions to Take** Find a summer internship.
Senior	**Things To Think About** What is life after college going to look like for you? How are you going to transition from the UW into the real world?	**Actions to Take** Get that offer. Negotiate. Live your life.

Pick at least one.

- Volunteer for a cause you care about
- Personal project (outside of class) to solve a problem
- Author a tech blog (or a fashion blog or a cat blog)
- Undergraduate research
- Teaching Assistant/Tutor
- Create a community for your peers.
- Join a consulting club on campus.
- Programming Contests
- Hackathons
- Leadership in student organizations
- Get good grades (if the only thing you are doing is studying, your GPA better be super high > 3.8)

Lies people tell you

Your GPA is the only thing that matters.

Bullshit.

You must have a CS degree to work in tech.

Bullshit.

Everyone but you knows what they're doing.

Bullshit.

Happy Hustling!

Kim Nguyen |  @hellokimwin | http://bit.ly/kimLinkedIn

Kasey Champion |  @techie4good | http://bit.ly/kaseyLinkedIn

Kim’s Ultimate Resume Guide: http://bit.ly/cseresumeguide

Kim’s College Recruiting Guide for Tech Roles: http://bit.ly/csecareerguide

The typical (tech) recruiting process

Step 1	Express initial interest *Career fairs, events, or via email*
Step 2	First round of the technical interview *Coding challenge, 30 - 60 minute technical screen*
Step 3	Final round of interviews *In person, 3 - 6 interview rounds lasting ~60 minutes each*
Step 4	Decision & offer *Good luck!*

Components of a technical interview

1. Introductions
2. Project discussion
3. Coding exercise
4. Your questions

Introductions

- Develop your pitch
  - Who are you?
  - What are your interests? Goals?
  - Why are you interested in the position?
  - 30 seconds – 1 minute
- Know SOMETHING about the company and why you're interviewing with them

Project Discussion

- Pick 1-2 projects off your resume you can speak in depth about
  - Pick your biggest or most technically interesting project
  - It’s ok to talk about school projects
- Don't assume subject domain expertise, but be able to go into detail when asked
- Avoid “we”

Project Discussion

- "I spent this summer working at an advertising network, specifically trying to drive engagement on our video ads by A/B testing new ad content and formats. I worked primarily in the backend and used Python and R for data analysis. I produced a 8% improvement in click-through rates across the board over six weeks of testing."

Follow up questions:

- How long did you work on this project?
- How big was the team working on this, what was your role specifically?
- Why did you choose that technology stack?
- What was the biggest bug you encountered and how did you fix it?
- If you redid the project what would you do differently?

Coding Exercise – Before

Practice Practice Practice

- Treat the interview like a standardized test
- Practice coding without an IDE/Compiler/Computer
- Practice coding and talking aloud at the same time
- Essential Practice Resources:
  - Cracking the Coding Interview
  - LeetCode
  - Data Structures and Algorithms (edX course)

Picking Your Language

- Strongly recommended: Pick something OOP
- Syntax typically doesn't matter
- Review helpful APIs
  - String -> Int
  - String manipulation
  - Popular data structures
  - Searching and Sorting algorithms
- Be able to talk about why you picked that language

Coding Exercise – In the Interview

Question Patterns

- **String or Array manipulation** - Great for tech screens, shorter, sometimes mathy
- **Linked Lists** - Often used in whiteboard interviews because they *expect* you to draw pictures
- **Trees** – BSTs, self balancing. Often used when building up directories or searching for something ie phone trees
- Sorting & Heaps -
- **Hash Tables** - If you are organizing data for lookups… chances are the answer is a hash table

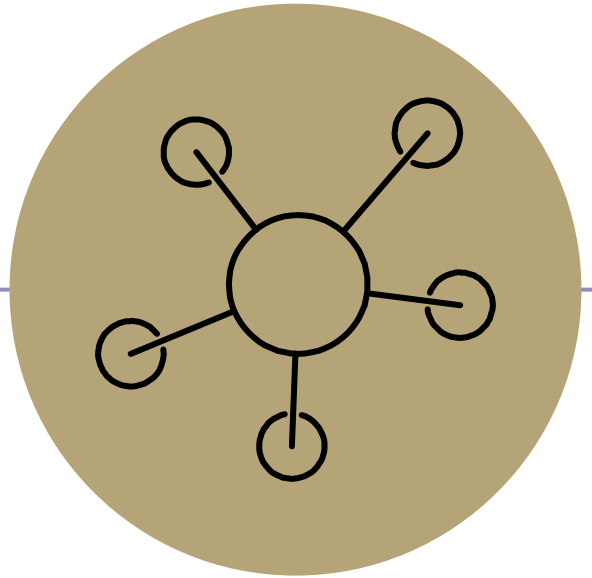
Structure Your Thoughts

- **Talk** - clarify the question
- **Example** - talk through sample input and expected output
- **Brute Force** - what is the simplest way to solve this?
- **Optimize** - can you save run time or memory?
- **Walk Through** - clarify your algorithm
- **Implement** - write the code!
- **Test** - list test cases, does your code address these?

Nervous during your technical coding interview? TEBOW IT!

Your Questions

- Don’t drop the ball!
  - You better have some questions
    - “What is your favorite part about working for X?”
    - “What are some projects you’ve worked on at X?”
    - “Where do you see X in 5 years?”
  - Don’t ask rude questions
    - “Did I pass?”
    - “How much do you make?”
- Show your interest
- Actually look for a good fit



Questions