http://bit.ly/gwsofteng16

CS 2113 Software Engineering

Lecture 2: Arrays, Pointers, and Memory in C

Professor Tim Wood - The George Washington University

Administrative

- Professor T. Wood
  - timwood@gwu.edu
  - Office Hours: **Tues 11-12:30 — does this work??**
    - SEH 4580
- TA: Bo Mei
  - bomei@gwu.edu
  - Office Hours: **Mondays 10:10-11AM and 1:30-2:10PM**
    - Outside of SEH 4900
- Are you:
  - Registered for the class?
  - **Registered for Piazza?**

Exercise 1?

- Should have been a challenge
  - (but possible)

- I expect you:
  - to be resourceful
  - to persevere
  - to ask for help

- Use:
  - assignment description
  - lecture slides
  - textbook
  - your classmates (within limits)

Part 2 will be due Tuesday 9/13

Also will have a worksheet to complete

Participation…

- Is important! Why???
- Every 2 weeks you can get 0-3 points
  - +1 whenever you say something in class
  - +1 whenever you post something to piazza
  - +1 whenever you attend office hours

  - If you post to piazza twice you get 2 points
  - If you speak in class 50 times you get 3 points
  - If you attend office hours once, post once, and speak up once you get 3 points
  - etc.

Programming Tips

- Try it out on paper first
  - Make sure the algorithm is right before worrying about syntax
- Test your code as you write it
  - Especially when you are first learning a language
  - Give some simple inputs that you can hand verify
  - "Correctly" running programs that give incorrect results are worse than programs that don't compile!
- Print out lots of debugging information
  - **`printf`** is your best friend!
  - Check that functions are being called, vars are being set properly, etc
  - Easy to just comment out once you know things are working

Additional Resources

- "Essential C" & Common Mistakes
  - **http://faculty.cs.gwu.edu/~timwood/wiki/doku.php/teaching:cmistakes**
  - Linked PDF is a great reference
- Prof. Simha's course notes:
  - **http://www.seas.gwu.edu/~simhaweb/cs143**
  - See Modules 1-4 (slightly different ordering)
- C Library Reference
  - **http://www.cplusplus.com/reference/clibrary/**
  - (Only look within the "C Library" section, the rest is C++)
  - Docs and examples for funcs in `stdio.h, stdlib.h, math.h`, etc

Git

- Who had used git before? Where?

- Basic steps will always be:
  - Follow link in assignment to create your own private repository
  - Use **git clone** to download that repo to CodeAnywhere
  - Use **git add/commit** to log your progress locally on CodeAnywhere
  - Use **git push** to send your commits to GitHub for grading

- Can also use **git pull** to get changes made on Github

This time...

- Understand memory in C
  - Arrays
  - Program memory layout
  - Pointers
  - Dynamic arrays
- Memory Worksheet
- **Share a computer with your neighbor**

Arrays

- Use arrays to store a “list” of variables

```
int main ()
{
   int profits[52];
   int w;
   int sum = 0;

   for(w=0; w < 52; w++)
   {
     profits[w] = w*10;
     sum += profits[w];
   }

   printf("Profits in third week: %d\n",
          profits[2]);
   printf("Total profit: %d\n", sum);

   return 0;
}
```

array size must be a constant

array indexes start at 0!

Arrays can be of any type

- Can make an array of any type

```
int profits[52];
float temps[100];
char letters[26];
```

- The array size needs to be a constant

```
int weeks = 52;
int profits[weeks];
// WILL NOT WORK!
```

- For best results, declare at top of a code block

Hint: Use your C reference sheet!

Simple array

- **WITH YOUR NEIGHBOR**

- Write a program that declares an array of 10 ints
  - Use a for loop to set array entry **i** to **i*10**
  - Use a for loop to print out the array
- Increase the size of your array to 20
  - but only fill in the first 10 entries... what happens when you print all 20?
  - What default value does C use for an int? What did Java do?
- Leave the size at 20, but print out the contents of array at indices 21...30
  - What happens? What did Java do?

Buffer Overflows

- What happens in Java? What happens in C?

```java
// bad Java code
int myArray[12];
myArray[99] = 666;
```

```c
// bad C code
int myArray[12];
myArray[99] = 666;
```

- Java tracks the size of an array... C does not
- What is the cost/benefit?

Simple array

- **WITH YOUR NEIGHBOR**

- Leave the size at 20, but print out the contents of array at indices 21...30000
  - What happens now?
  - Is your result the same as other people at your table?

Arrays in C

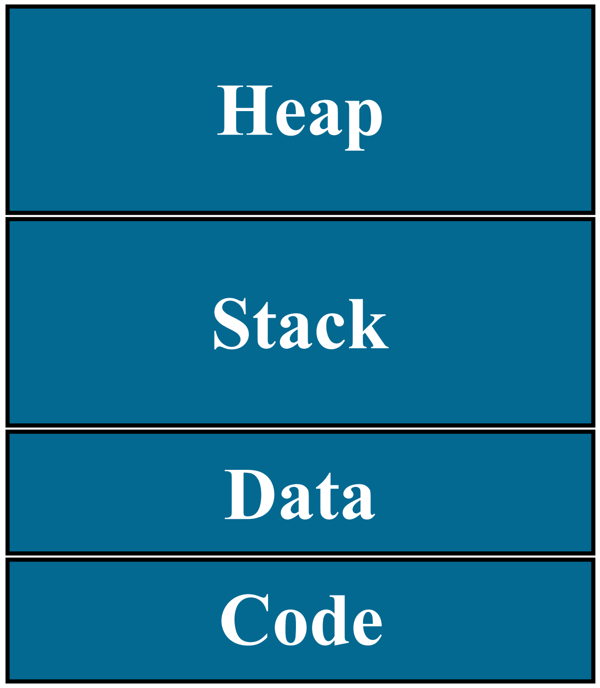
- Size must be a constant

```
int myArray[100]; // create an array with 100 integer entries
int hundred=100;
int illegalArray[hundred]; // WILL NOT COMPILE
```

- What if we want to change the array size as the program is running?
  - Suppose we want enough days in our array for a leap year?
  - What if we move to Jupiter (10,563 sunrises/rotation)? Or Venus (about 2 sunrises/rotation)?

Memory Management

- Code segment
  - Stores the binary code of your program
  - Read only
- Data segment
  - Stores global variables, constants, etc
- Stack segment
  - Stores temporary variables
  - New section added to stack with each function call
- Heap segment
  - Holds dynamically allocated memory

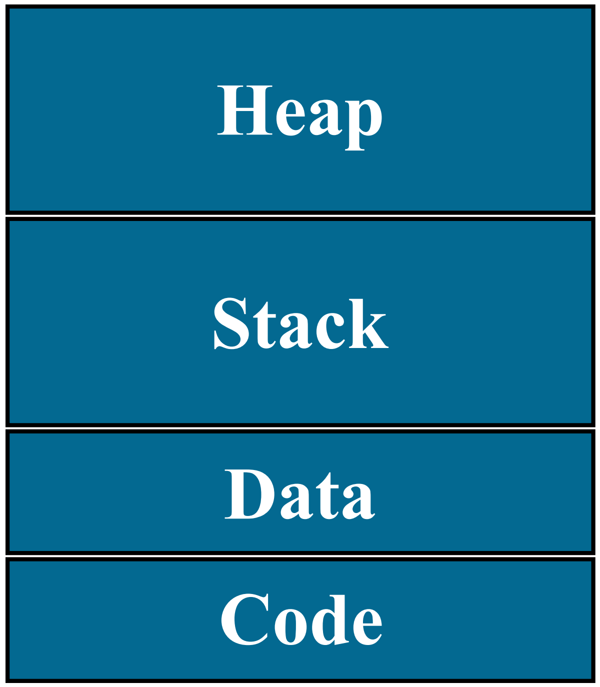


Program Memory

Memory Management

- Memory is allocated by the OS
  - Ask for a chunk of memory at start
  - Can ask for more as you run

- Why split up memory?



* actual memory layout varies by CPU architecture

Code Area

- Read only memory region to store binary instructions that make up a program

- Why load program into memory at all?
- Why keep it read only?

```
#include <stdio.h>

int main ()
{
  printf ("Hello Class!\n
  return 0;
}
```

C code

```
        .cstring
LC0:
        .ascii "Hello Class!\0"
        .text
.globl _main
_main:
LFB3:
        pushq   %rbp
LCFI0:
        movq    %rsp, %rbp
LCFI1:
        leaq    LC0(%rip),
        call    _puts
        movl    $0, %eax
        leave
```

assembly code

memory address

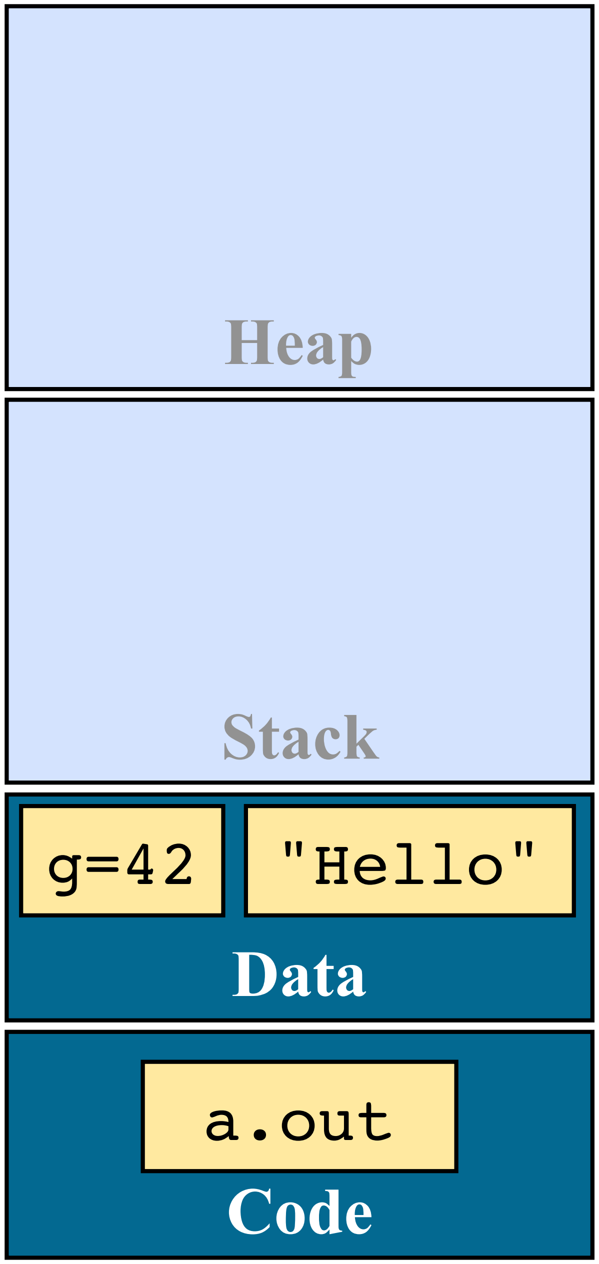
```
0000000 cf fa ed fe 07 00 00 01 03 00 00 80 02 00 00 00
0000010 0b 00 00 00 06 00 00 85 00 00 00 00 00 00 00
0000020 19 00 00 00 48 00 00 00 5f 5f 50 41 47 45 5a 45
0000030 52 4f 00 00 00 00 00 00 00 00 00 00 00 00 00
0000040 00 00 00 00 01 00 00 00 00 00 00 00 00 00 00
0000050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000060 00 00 00 00 00 00 00 00 19 00 00 00 28 02 00 00
0000070 5f 5f 54 45 58 54 00 00 00 00 00 00 00 00 00
0000080 00 00 00 00 01 00 00 00 00 10 00 00 00 00 00 00
0000090 00 00 00 00 00 00 00 00 10 00 00 00 00 00 00
00000a0 07 00 00 00 05 00 00 00 06 00 00 00 00 00 00
00000b0 5f 5f 74 65 78 74 00 00 00 00 00 00 00 00 00
```

binary code

Data Segment

- Need an easily accessed area for global data
  - Global variables (declared outside a function)
  - Constants (usually strings)

```
// gcc file.c -o a.out

int g=42;

int main()
{
    int x=100*g;
    char s[] = "Hello";
    // ....
}
```



Variables in Functions

- How could we store these?

Memory

```
int someFunc(int x)
{
   float a=2.25;
   double b=9.123456789;
   ...
   return round(b)
}
int round(double d)
{
   int r;
   ...
   return r;
}
int main()
{
   int x, y;
   ...
   x=23;
   y = someFunc(x);
   ...
```

Variables in Functions

- How could we store these?

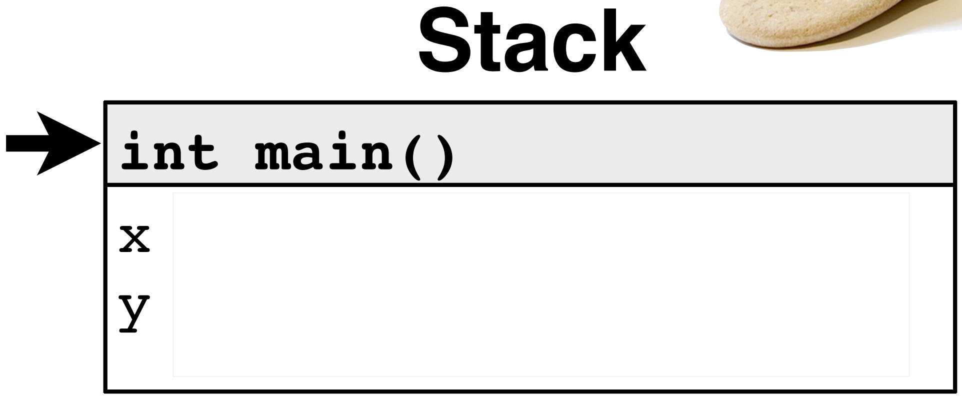
- Not randomly…
- Ordered by function
- Only need memory for functions that are active
- Free memory for vars in functions when they return

```
int someFunc(int x)
{
  float a=2.25;
  double b=9.123456789;
  ...
  return round(b)
}
int round(double d)
{
  int r;
  ...
  return r;
}
int main()
{
  int x, y;
  ...
  x=23;
  y = someFunc(x);
  ...
```

The Stack

- For predictably sized memory regions

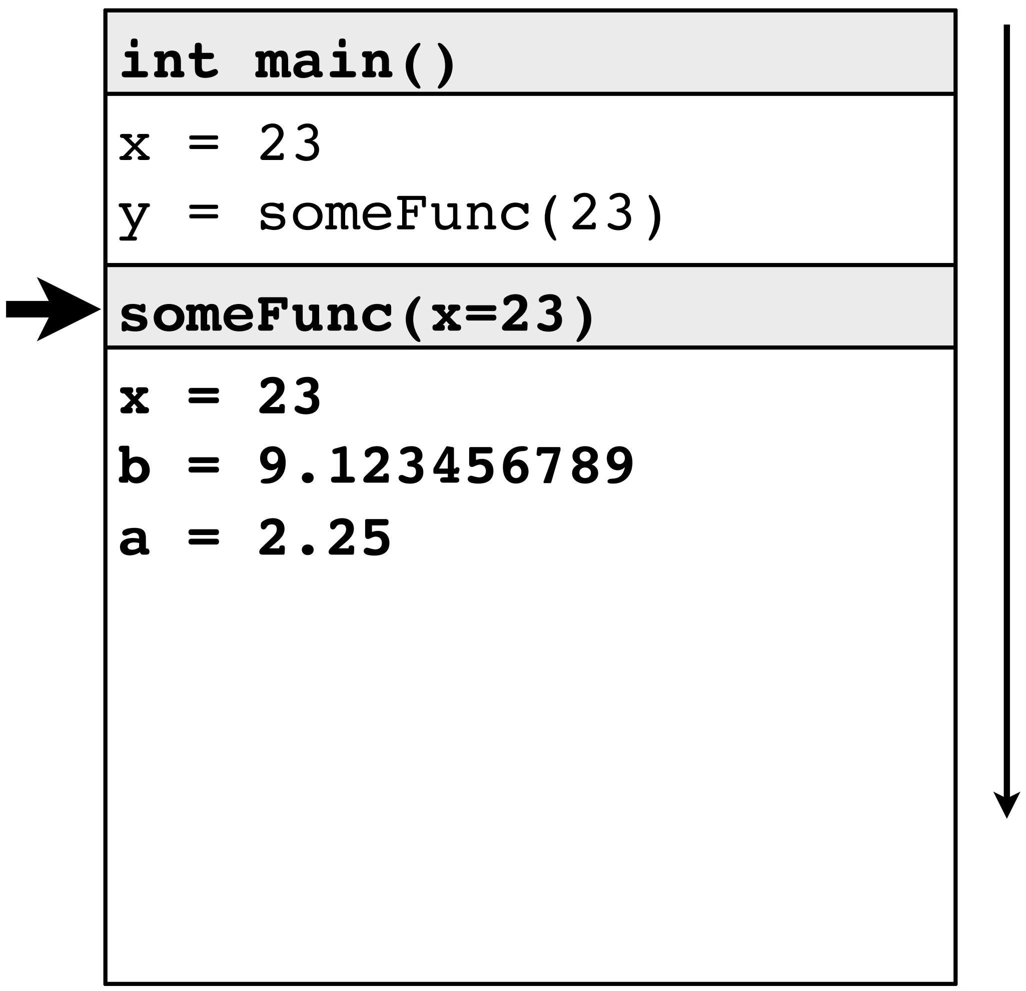
```
int someFunc(int x)
{
  float a=2.25;
  double b=9.123456789;
  ...
  return round(b)
}
int round(double d)
{
  int r;
  ...
  return r;
}
int main()
{
  int x, y;
  ...
  x=23;
  y = someFunc(x);
  ...
```



The Stack

- Adds new section for each function call

```
int someFunc(int x)
{
  float a=2.25;
  double b=9.123456789;
  ...
  return round(b)
}
int round(double d)
{
  int r;
  ...
  return r;
}
int main()
{
  int x, y;
  ...
  x=23;
  y = someFunc(x);
  ...
```



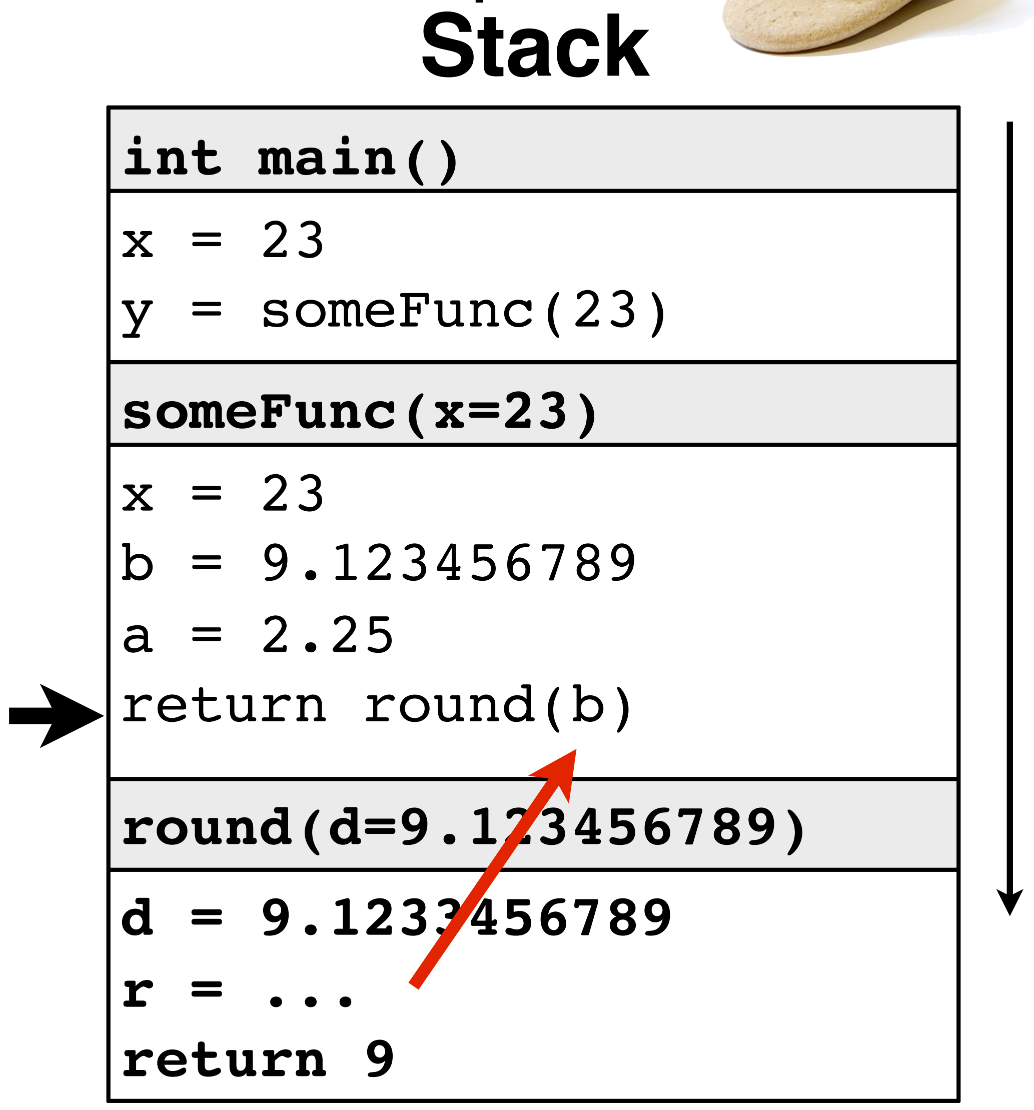
Stack

```
int main()
x = 23
y = someFunc(23)
someFunc(x=23)
x = 23
b = 9.123456789
a = 2.25
```

The Stack

- "Pop" section when function completes

```
int someFunc(int x)
{
   float a=2.25;
   double b=9.123456789;
   ...
   return round(b)
}
int round(double d)
{
   int r;
   ...
   return r;
}
int main()
{
   int x, y;
   ...
   x=23;
   y = someFunc(x);
   ...
```

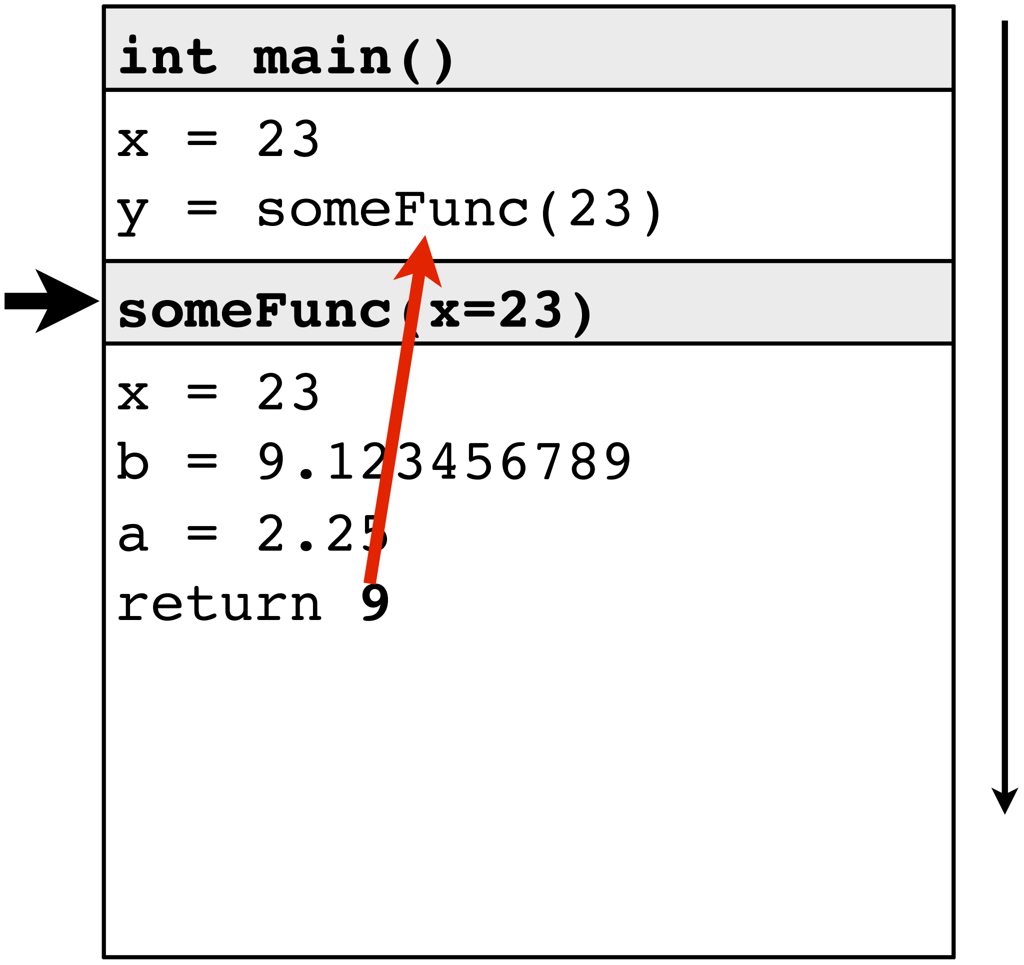


The Stack

- "Pop" section when function completes

```
int someFunc(int x)
{
  float a=2.25;
  double b=9.123456789;
  ...
  return round(b)
}
int round(double d)
{
  int r;
  ...
  return r;
}
int main()
{
  int x, y;
  ...
  x=23;
  y = someFunc(x);
  ...
```

Stack

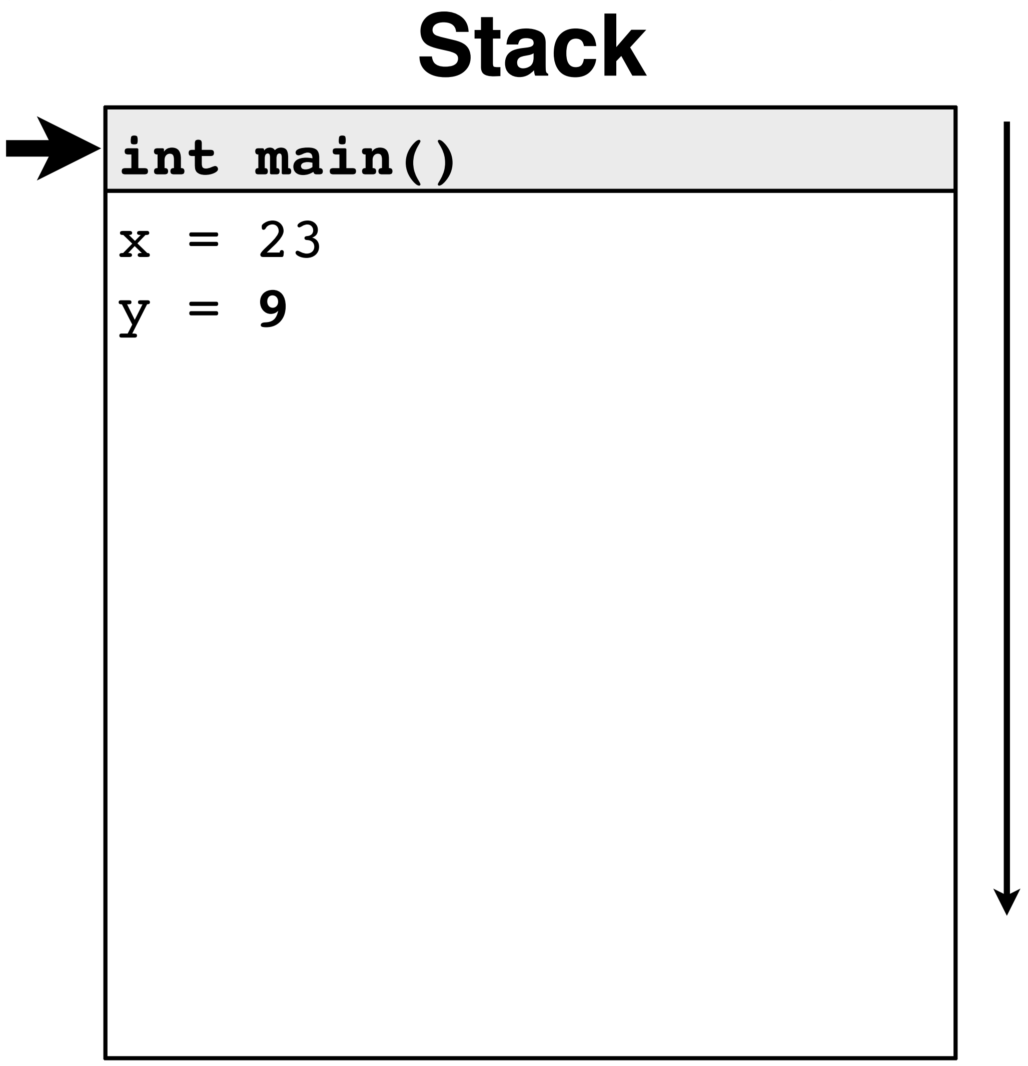


```
int main()
x = 23
y = someFunc(23)
someFunc(x=23)
x = 23
b = 9.123456789
a = 2.25
return 9
```

The Stack

- Subsequent calls may reuse the old memory

```
int someFunc(int x)
{
  float a=2.25;
  double b=9.123456789;
  ...
  return round(b)
}
int round(double d)
{
  int r;
  ...
  return r;
}
int main()
{
  int x, y;
  ...
  x=23;
  y = someFunc(x);
  ...
```

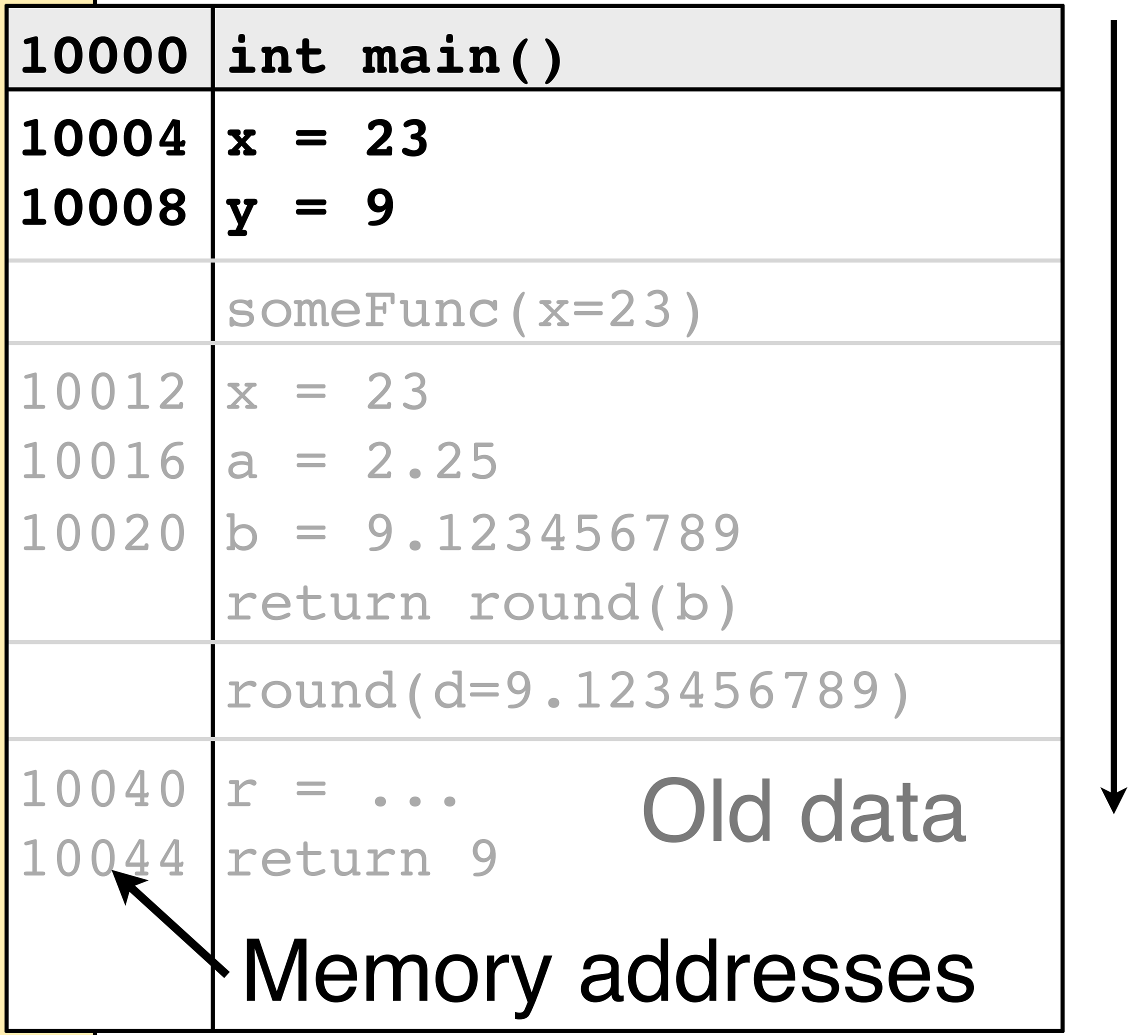


The Stack

- Old data may still be left on stack!

```
int someFunc(int x)
{
  float a=2.25;
  double b=9.123456789;
  ...
  return round(b)
}
int round(double d)
{
  int r;
  ...
  return r;
}
int main()
{
  int x, y;
  ...
  x=23;
  y = someFunc(x);
  ...
```

Stack



10000	int main()
10004	x = 23
10008	y = 9
	someFunc(x=23)
10012	x = 23
10016	a = 2.25
10020	b = 9.123456789
	return round(b)
	round(d=9.123456789)
10040	r = ...
10044	return 9

Worksheet #1

- Solve at your table

Stack Dump 1		
Address	*Name*	*Contents*
10000		
10004		
10008		
10012		
10016		
10020		
10024		
10028		
10032		

Stack Dump 2		
Address	*Name*	*Contents*
10000		
10004		
10008		
10012		
10016		
10020		
10024		
10028		
10032		

```
int main(void) {
  int a, b, c;
  a = 123;
  b = 456;
  c = func_2(a);
  c = func_3();
  // STACK DUMP 2
}

int func_2(int x) {
  int y = 15;
  x = 100;
  func_3();
  // STACK DUMP 1
  return x + y;
}

int func_3(void) {
  int array[3];
  array[0] = -5;
}
```

You've seen this before...

- In Java, you probably saw "Stack Traces" printed out when you hit an error / exception

```
Exception in thread "main" java.lang.NullPointerException
        at com.example.myproject.Book.getTitle(Book.java:16)
        at com.example.myproject.Author.getBookTitles(Author.java:25)
        at com.example.myproject.Bootstrap.main(Bootstrap.java:14)
```

- Each "at" line is a level of the stack
- Can get very deep with nested or recursive functions!

Function Variables

- Consider this code:

```
void moveNE(int a, int b)
{
  a++; b++;
}
int main()
{
  int x = 1; int y=10;
  printf("XY is: %d, %d\n", x, y);
  moveNE(x, y);
  printf("XY is: %d, %d\n", x, y);
}
```

???

- What will this print?
- Get the code: git clone
- What if the func arguments are renamed **x** and **y**?

Where is X?

- We wanted the code to modify x and y!
- Need to tell **where** they are stored
- Use the "address of" operator: &

```
int main()
{
  int x = 1; int y=10;
  printf("XY is: %d, %d\n", x, y);
  printf("The address of X is %p\n", &x);
  printf("The address of y is %p\n", &y);
  // ...
}
```

Use %p to format an address for printing

Use &VAR to get the address of VAR

```
The address of X is: 0x7fff5fbff74c
The address of y is: 0x7fff5fbff748

# addresses in hexadecimal format
```

How far apart?
What about a global?

So....

- We know where the data we want the function to modify is
  - Can use &x and &y
- How can we tell moveNE() to modify that data?
  - Can we just use

```
void moveNE(int a, int b)
{
  a++; b++;
}
int main()
{
  // ... will this work?
  moveNE(&x, &y);
  // ...
}
```

?

No...

Pointers

- Pointers are special variables for **storing memory addresses**
- A pointer has an associated type
  - e.g., int pointer vs float pointer vs char pointer
- Pointers can be accessed **two ways:**
  - to read/write the **address** stored in them
  - to read/write the **value** stored at the address

```
int main()
{
   int a = 10;
   int *ptr;   // declare a pointer
   ptr = &a;   // set the ADDRESS
   *ptr = 20;  // set the VALUE
}
```

be very careful with your *!

Worksheet #2 and 3

```
int i;
int j;
int *p;
```

Given the variable declarations above, what syntax would you use to access the following pieces of information in C code?

The value of i:

The address that p points to:

The address of j:

The address of p:

The value that p points to:

Worksheet #2 and 3

```
int main()
{
   int a = 10;
   int *ptr;
   ptr = &a;
   *ptr = 20;
   // STACK DUMP 3
   …
}
```

Stack Dump 3		
Address	*Name*	*Contents*
10000		
10004		
10008		

What about...?

```
void moveNE(int *a, int *b)
{
  // what will go here?
}
int main()
{
  // ...
  moveNE(&x, &y);
  // ...
}
```

Magic of Pointers

- Pointers are a 2-for-1 deal
  - Can easily work with memory address **and** the value stored in the memory address
- "Dereference" a pointer using the asterisk *
  - Returns the contents of the memory address referenced by the pointer
  - Can both set and get value

```
void moveNE(int *a, int *b)
{
  *a = *a+1; // change VALUE of pointer a
  (*b)++; // different syntax, same result
}
int main()
{
  // ...
  moveNE(&x, &y);
  // ...
}
```

More Pointing

- Pointers can be used for more than just passing references as function arguments

```
int i = 100;
int *ptr;
ptr = &i;

printf("The ADDRESS of i is the value of ptr: %p\n", ptr);
printf("which is the same as &i: %p\n", &i);
printf("The VALUE of i is *ptr: %d\n", *ptr);
printf("which is the same as i: %p\n", i);

// use * to dereference pointer->memory value
printf("Changing *ptr to 50...\n");
*ptr = 50; // or to set value of memory pointed at

printf("The VALUE of *ptr is now: %d\n", *ptr);
printf("which is the same as i: %p\n", i);
```

Attendance Quiz!

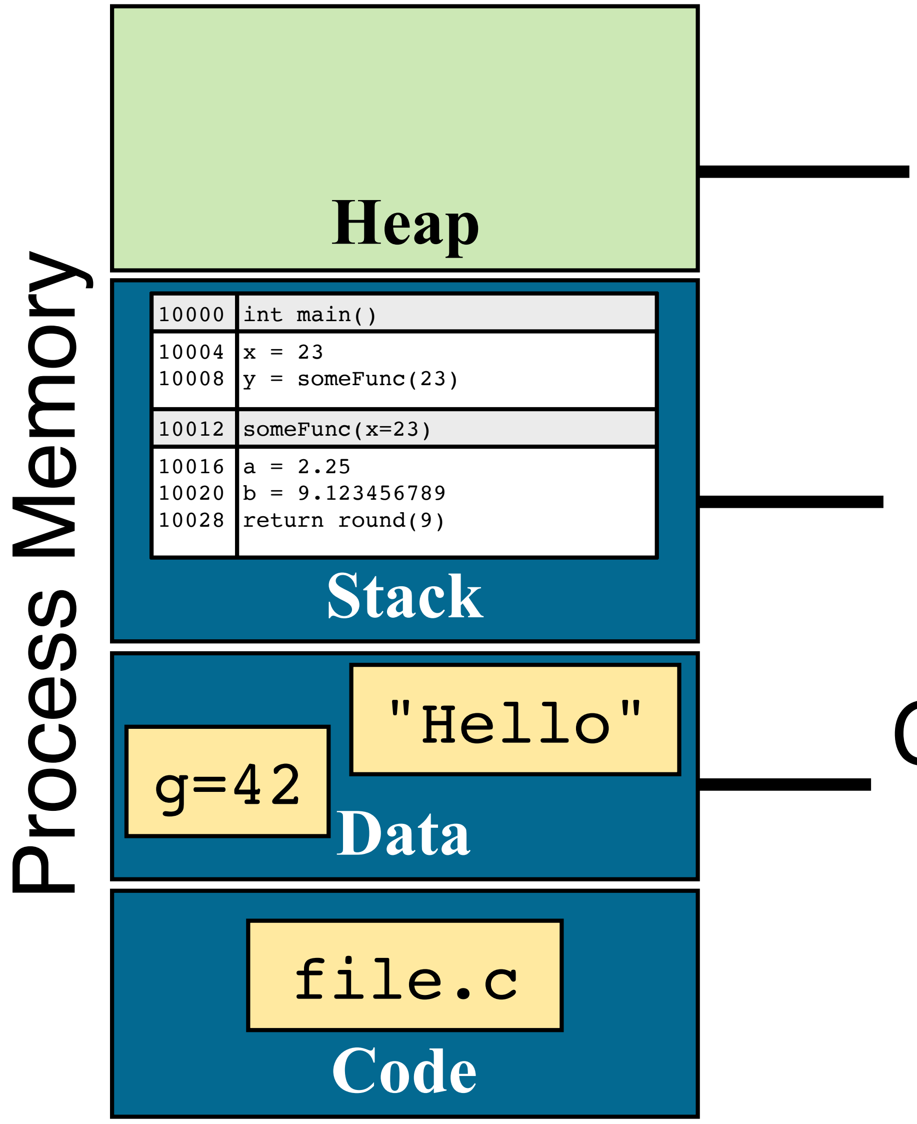
- What are the values of i, j, and *ptr?

```
int i = 100;
int j;
int *ptr;

ptr = &i;
j = *ptr;
*ptr = 200;

printf("i=%d, j=%d, *ptr=%d\n", i, j, *ptr);
```

- Fill out the quiz on **blackboard**
  - In the "Testing" category

•

Heap Segment



Dynamically created memory

Structured area for local variables

Global variables, constants
Easily referenced

The Heap / Memory Pool

- The stack enforces very specific ordering
  - "Local variables" in a function must be on stack!
  - Stack changes with every function call
- Heap is a bit... messier
  - Gives access to variable size chunks of memory
  - Structure is not affected by function calls
  - Required for dynamically sized objects

vs

Aside:
"The heap" and "a heap" (data structure) are not particularly related to one another

images from wikimedia commons

Dynamic Memory in C

- Heap acts as a pool of memory for data structures
    - **`ptr=malloc(size);`**
        - Ask the heap for **`size`** bytes of memory
        - Returns a memory address (so assign it to a pointer!) or 0 if the allocation failed (so check it!)
    - **`free(ptr);`**

```
#include <stdlib.h> // include to use malloc without warnings
#include <stdio.h>
int main()
{
  int *myInt;
  myInt = (int*) malloc(4); // request 4 bytes for an integer
  if(myInt == NULL) { return -1;}
  *myInt = 1234567;
  printf("Value of myInt = %d\n", *myInt);
  free(myInt); // release memory when done
  return 0;
}
```

Stack + Heap

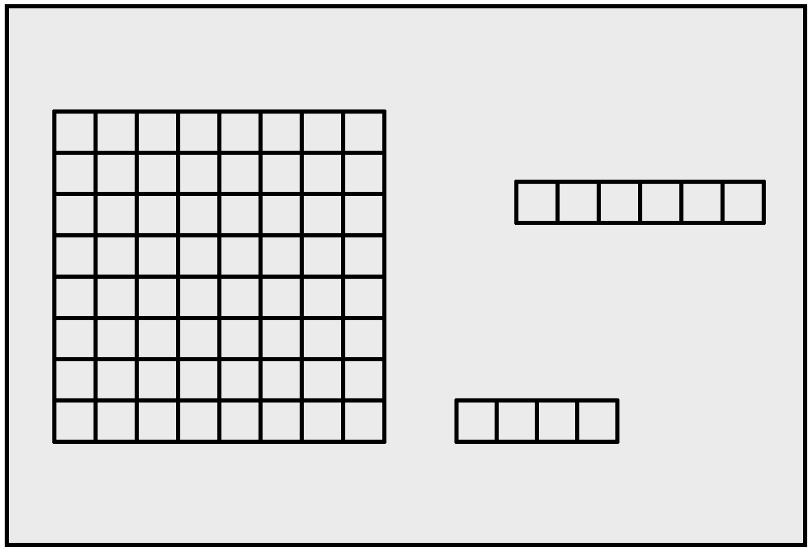
- Need a pointer to reach the heap

```
int main()
{
  int *myInt;
  double *myDouble;
  char *myString;
  myInt = (int*) malloc(4);
  myDouble = (double*)malloc(8);
  myString = (char*)malloc(64)
  ...
}
```

Stack

10000	int main()
10004	*myInt --> address 99048
10008	*myDouble --> address 99743
10012	*myString --> address 98321

Heap



- Pointer addresses stored in the stack
  - "Value" of pointer is kept on the heap

(Simplified) Memory Layout

```
void firstFunc() {
    int a = 20;
    int b = 30;
    int *p = malloc(sizeof(int));
    *p = 5;
    b = *p;
    free(p);
    secondFunc();
    p = malloc(sizeof(int));
    *p = 50;
}
void secondFunc() {
    int *q = (int*) malloc(sizeof(int));
    int *r = (int*) malloc(sizeof(int));

    *q = 45;
    *r = 100;
    /* DRAW MEMORY @ THIS LINE */
    free(q);
    return;
}
int main() {
    int x = 10;
    int *y = &x;
    firstFunc();
    return 0;
}
```

Assume:
- int and int* use 4 bytes
- No extra space needed for function headers

Stack

Address	*Name*	*Contents*
10000		
10004		
10008		
10012		
10016		
10020		
10024		
10028		
10032		

(addresses grow up)
start: 10,000
Stack

Heap

Address	*Name*	*Contents*
50000		
49996		
49992		
49988		
49984		
49980		
49976		
49972		
49968		

start: 50,000
(addresses grow down)
Heap

(Simplified) Memory Layout

```
void firstFunc() {
    int a = 20;
    int b = 30;
    int *p = malloc(sizeof(int));
    *p = 5;
    b = *p;
    free(p);
    secondFunc();
    p = malloc(sizeof(int));
    *p = 50;
}
void secondFunc() {
    int *q = (int*) malloc(sizeof(int));
    int *r = (int*) malloc(sizeof(int));

    *q = 45;
    *r = 100;
    /* DRAW MEMORY @ THIS LINE */
    free(q);
    return;
}
int main() {
    int x = 10;
    int *y = &x;
    firstFunc();
    return 0;
}
```

Stack

Address	*Name*	*Contents*
10000		
10004		
10008		
10012		
10016		
10020		
10024		
10028		
10032		

Heap

Address	*Name*	*Contents*
50000		
49996		
49992		
49988		
49984		
49980		
49976		
49972		
49968		

The Heap Sticks Around

- If you call **`malloc`**, you must also call **`free`** at some point
  - Need to clean up data structures you are no longer using
- **Memory Leak**: wasting resources by not freeing data after the program is done with it

```
for(i=0; i < 1000000; i++)
{
  myArray = (int*) malloc(4*1000);
  // do something with array
}
free(myArray);
```

- Be sure to balance **`malloc`** / **`free`** calls!

Dynamic Arrays

- Pointers can also be used like arrays
  - Remember: allocate space in bytes
    - Use **`sizeof(TYPE)`** to find size of a var type in bytes
  - Can index into pointer using array[] syntax
    - Do not need to use dereference operand!

```
int earthDays = 365;
int jupiterDays = 10563;

int* dayArray = (int*) malloc(sizeof(int)*earthDays);
dayArray[32] = 1; // set my birthday
// ...
free(dayArray);

dayArray = (int*) malloc(sizeof(int)*jupiterDays);
dayArray[4632] = 1;
```

Pointer Math

- Can treat arrays as pointers and vice versa

```
int nums[] = {1, 3, 5, 7, 9};
printf("%d ", nums[2]);
printf("%d ", *(nums+2);
printf("%d ", 2[nums];
```

never use this

- C lets us access memory at arbitrary locations
  - It is easy to mistakenly write syntactically correct, but logically incorrect C code

```
int* dayArray = (int*) malloc(sizeof(int)*365);
*dayArray = 1; // set first entry in array
dayArray += 1;  // adjust address by 1 entry
*dayArray = 2; // set second entry in array
dayArray += 500; // beyond the end of the array!
```

Segmentation Faults

- Probably the most common type of error in C
- Caused by trying to access a memory address which is invalid

```
// world's stupidest seg fault:
int *dumb = 0;
*dumb = 123;
```

```
$ ./a.out
Segmentation fault
```

- Compiler will not detect
- Common cause: pointer arithmetic
  - Can treat pointer addresses as regular numbers

Arrays as Arguments

- Before we saw that C is pass-by-value
  - Stops our ship from sailing :(

- But what about array arguments

```
#include <stdio.h>

void moveNE(int xy[])
{
    xy[0]++;
    xy[1]++;
}
int main()
{
   int pos[2];
   pos[0] = 1; pos[1] = 10;
   printf("XY is: %d, %d\n", pos[0], pos[1]);
   moveNE(pos);
   printf("XY is: %d, %d\n", pos[0], pos[1]);
   return 0;
}
```

This code **will work** because the **array address** is sent to the function

Finding Addresses

- Write a program that prints out:
  - A global variable's address
  - A variable defined inside main()
  - A variable defined inside func1() called from main()
  - A variable defined inside func1b() called from func1()
  - A variable defined inside func2() called from main after func1() returns
  - The address of the function main()