CS 137 Week 2

Loops, Functions, Recursion, Arrays

September 18th, 2017

Loops

- We will finish this week with looping statements
- We already discussed one such structure, namely while loops.
- `while (expr) statement` can run 0 or more times
- A `do while` loop will execute at least once
- `do {statements} while (expr);`

Example

```c
#include <stdio.h>
int main(void) {
  int a=1920, b = 1080;
  int r; //Why is this declaration here?
  do{
    r = a % b;
    a = b;
    b = r;
  } while (r != 0);
  printf("%d", a);
  return 0;
}
```

For Loop

Syntax: `for(expr1; expr2; expr3) { statements }`

- Expression 1 is usually an initializer before the loop is executed
- Expression 2 is usually a condition
- Expression 3 is usually an incrementer/decrementer.
- In C99 and beyond, Expression 1 can be a initialization of a new variable whose scope is only the for loop.

Example

Print the numbers from 0 to 9

```
#include <stdio.h>
int main(void) {
  for(int i = 0; i < 10; i = i+1){
    printf("%d\n", i);
  }
  //Note: i=2 here would result in an error.
  return 0;
}
```

Breaks and Continues

- Like with switch statements, you can use the break; command to end the termination of a while, do or for statement early.
- C also has continue statements which allows you to exit the loop body early and go to just before the end of the loop body
- There is also a goto command that can jump to any labelled part of the program.
- Using these statements almost always makes code harder to read and so most programmers avoid or limit the use of these as much as possible.

Example - Sum Three Nonzero Integers

```
#include <stdio.h>
int main(void) {
   int n=0, sum=0, i=0;
   while (n < 3){
      scanf("%d",&i);
      if (i==0) continue;
      sum += i;
      n++;
   }
   printf("%d", sum);
}
```

Example

Write a program that asks the user for an integer and determines if it is a prime number. Print "Not Prime" if it is not prime and "Prime" otherwise.

- Recall: A prime number is an integer greater than 1 whose only positive divisors are 1 and itself.

Idea 1

Count all the divisors of the given number and if it is 2, we have a prime number.

```
#include <stdio.h>
int main(void) {
  int n, div=1, c=0;
  scanf("%d",&n);
  if (n<=1) printf("Not Prime");
  else {
    while(div <= n){
      if (n%div == 0) c++;
      div++;
    }
    if (c==2) printf("Prime");
    else printf("Not Prime");
  }
  return 0;
}
```

Optimization

There are a few ways we can make this code run faster

- If we have a non-prime number, then it has a divisor between the number itself and 1 (not including endpoints). If we find this, we can immediately break the code and print not prime
- If it has such a divisor d, there must be one less than $\sqrt{n}$ where n is the given number for one of d or n/d is bounded above by $\sqrt{n}$.
- Let's rewrite the code to account for these changes.

Idea 2

Search for a single divisor not 1 or n and if we find one return false.

```
#include <stdio.h>
int main(void) {
  int n, div=2;
  scanf("%d",&n);
  if (n<=1) printf("Not Prime");
  else {
    while(div*div <= n){
      if (n%div == 0) break;
      div++;
    }
    if (div*div <= n) printf("Not Prime");
    else printf("Prime");
  }
  return 0;
}
```

Comments

- This code now works but isn't very reuseable.
- You would have to embed this code every time inside the main function if you wanted to determine if a number is prime.
- It would be nice if we could create a function that would return true or false depending on if the given number is prime or not.

Functions

- We've actually already seen functions;

  ```
  int main(void)
  ```

  is a function!
- Syntax:

  ```
  return-type fcn_name (parameter[s]) {}
  ```
- The return-type must be specified in C99 and later; it is `void` if the function does not return anything
- `fcn_name` follows the same rules as variable names
- `parameters` must have their types declared per variable, eg. `int fun(int a, int b)` vs `int fun(int a, b)`
- Calling a function with no parameters: `fcn_name();`
- `return` ends the function and returns the value after it.

Basic Example

```
#include <stdio.h>
int max(int a, int b){
  return a > b ? a : b;
}
int main(void) {
  printf("%d", max(5,10));
  return 0;
}
```

Boolean Variables

- One more quirk about C is that there are no boolean variables.
- In C99 and later, there is a library `<stdbool.h>` that gives you boolean variables.
- It turns out that these `bool` variables are secretly unsigned integers in disguise (so even with this really aren't boolean variables!) and these take up a full byte like a `char`.
- These types can only take 0 and 1 as values (all non-zero values are just 1). You can also use the words `true` and `false` with this library.

Code Page 1

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

bool isPrime(int n){
  int div=2;
  if (n<=1) return false;
  while(div*div <= n){
    if (n%div == 0) return false;
    div++;
  }
  return true;
}
```

Code Page 2

```
int main(void) {
   int n;
   scanf("%d",&n);
   bool is_prime = isPrime(n);
   if (is_prime) printf("Prime\n");
   else printf("Not Prime\n");
   return 0;
}
```

A Bit About Local Variables

- The variable `n` inside isPrime is what we call a local variable.
- The value from main is copied into the local variable in the `isPrime` function.
- Changing the local variable inside `isPrime` does not affect the variable in main (Thankfully!)
- Returning a local variable returns a copy of the value.

Passing By Value

Functions in C pass by value. What is printed below?

```
#include <stdio.h>
void foo(int n){
  n = 10;
}
int main(void) {
  int n = 4;
  foo(n);
  printf("%d\n",n);
  return 0;
}
```

Function Declarations

- In our example, we defined the function before using it.
- Strictly speaking, C doesn't force us to do this.
- However, C will be guessing at what the return type is of the function which it will default to `int`.
- To correct this, we can include a function declaration - a sort of promise to C that eventually we'll be defining this function that has this given return type.
- The declaration is the first line of the function but ends with a semicolon.
- The declaration needn't include the parameters, though it is advised to do so.

Rewritten

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

bool isPrime(int num);

int main(void) {
  //Same Code
}

bool isPrime(int num){
  //Same Code
}
```

New Topic - Recursion

- Simply put, recursion occurs when a function calls itself.
- Two golden rules of recursion:
  1. Must have a base case.
  2. Must make progress towards that base case.

GCD

Below is the recursive version of computing the GCD of two numbers.

GCD

Below is the recursive version of computing the GCD of two numbers.

```
int gcd(int a, int b){
  if (b==0)
    return a;
  return gcd(b, a%b);
}
```

How This Looks on the Stack

gcd	a = 26, b=0	returns 26
gcd	a = 52, b=26	returns gcd(26, 0)
gcd	a = 78, b=52	returns gcd(52, 26)
gcd	a = 130, b=78	returns gcd(78, 52)
gcd	a = 338, b=130	returns gcd(130, 78)
gcd	a = 806, b=338	returns gcd(338, 130)
main	...	

Sample

- Write a recursive function `power` that takes two non-negative integer parameters `b` and `n` and returns b^n.
- Write a recursive function `geometric_sum` that takes two non-negative integer parameters `b` and `n` and returns the sum

$$1 + b + b^2 + b^3 + ... + b^n.$$

If you finish quickly, also do this iteratively.

Example - Leap Years

- Gregorian calendar replaced the Julian calendar in most of Europe in 1582.
- North America (ie. England and its colonies) adopted this calendar in September of 1752.
- A leap year, that is, a year containing an extra day occurs every 4 years that are not multiples of 100 unless they are also multiples of 400.
- Examples: 2016, 2000, 1804 were all leap years.
- Non-Examples: 2017, 1900, 1950 were not leap years.
- Write a function `is_leap_year` that returns `true` if a given year was a leap year and returns `false` otherwise.

Exceptions

- This works well but gives us some problems if the year were say negative.
- In fact, since we are in North America, we probably want the year to be at least 1752.
- We can accomplish this by using `assert` statements.
- First, add `#include <assert.h>` to the beginning then include `assert(year > 1752);`

Assert

- In general, `assert(expr);`
- If `expr` is true, this line does nothing
- Otherwise, it terminates the program with a message containing the filename, line number, function and expression.
- This is great for debugging.
- Also great to leave it in (so long as `expr` is not computationally expensive). Helps to remember assumptions, causes program to fail "loudly" vs "quietly", advises other programmers if code undergoes modifications.
- Good for regression testing as well, that is checking that changes haven't broken anything in another part of the code.

Separate Compilation

- In the "real world", programs are coded by many programmers.
- It is often inefficient to all be working on the same file, not to mention it can get very confusing when you have millions of lines of code.
- We would like to modularize the design and reduce compile time.

powers.h

Below are the contents of `powers.h` - a header file with some function definitions.

```
#ifndef POWERS_H //Prevents multiple inclusion
#define POWERS_H

/* Pre: num is a valid integer
   Post: returns the square of num.
*/
int square(int num);
int cube(int num);
int quartic(int num);
int quintic(int num);
#endif
```

powers.c

Below are the contents of `powers.c` - the file defining the header functions.

```
#include "powers.h" //notice the quotes!
int square(int num) {return num * num;}
int cube(int num) {return num*square(num);}
int quartic(int num) {
  return square(num)*square(num);}
int quintic(int num) {
  return square(num)*cube(num);}
```

Note - we separate these to help reduce compile time.

main.c

Below are the contents of `main.c` - the file defining the header functions.

```
#include <stdio.h>
#include "powers.h"
int main(void){
   int num=3;
   printf("%d^4 = %d\n", num, quartic(num));
   num = 2;
   printf("%d^5 = %d\n", num, quintic(num));
   return 0;
}
```

Note - we separate these to help reduce compile time.

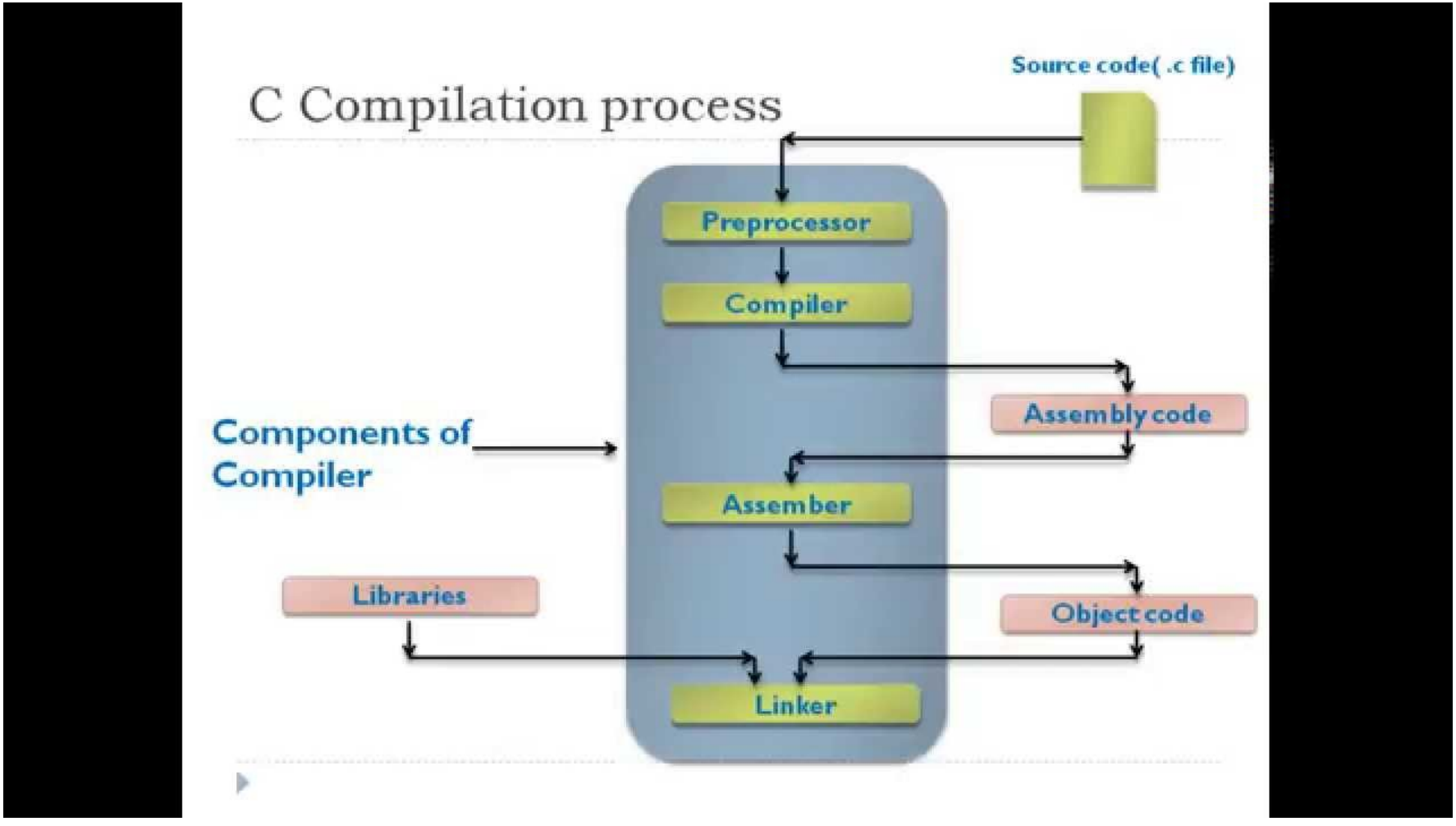
Compiling

- To compile, use

```
gcc -o powers powers.c main.c
```

- This compiles the code into assembly code then to object code and these files are automatically passed to the linker merging them to a single file.
- Reminder the `-o` command means to send the output to the filename immediately following.
- If you don't want to link the files (ie you just want to compile), use the command `-c`.

C Compiler



Source:
`https://i.ytimg.com/vi/VDslRumKvRA/maxresdefault.jpg`

More on Macros

- We've already seen three macros, namely `#include`, `#ifndef` and `#define`.
- In fact we can use the `#define` to define constants that we use in our code.
- Syntax: `#define identifier replacement_list`
- Notice that you don't need an equal sign or a semi colon - this is a straight replacement.
- This can be useful for constants in your code.

Example

Preprocessing turns the left into the right. (Mention floats).

```c
#include <stdio.h>
#define PI 3.1415
int main(void) {
  int r = 3;
  printf("Area:
    %f", PI*r*r);
  return 0;
}
```

```c
#include <stdio.h>
int main(void) {
  int r = 3;
  printf("Area:
    %f", 3.1415*r*r);
  return 0;
}
```

Arrays

- Up to this point we've seen ways to create variables to store data.
- What if wanted lots of variables? For example, suppose you wanted to store an office directory. Should we create a variable for each person?
- This is solved by arrays.
- Think of an array like a filing cabinet.
- Arrays start indexed at 0 (often why in CS natural numbers start at 0).

How Arrays look in Memory

`int a[5] = {10,-7,3,8,42}`

Variable Name	Memory Address	Value
a	10856	10
	10860	-7
	10864	3
	10868	8
	10872	42

We access the values via `a[0]`, `a[1]` and so on.

Creating Arrays

- `int a[5] = {10,-7,3,8,42}`
- `int a[] = {10,-7,3,8,42}` (size is inferred)
- `int a[5] = {1,2,3}` (last two entries are 0 by default)
- `int a[5] = {0}` (creates an all 0 array)
- `int a[5]` (uninitialized array contains garbage entries)
- `int a[5] = {[2] = 2, [4] = 3123}` (specified the third and fifth entries - rest are 0)

Warning

This is not allowed:

```
int a[5]; a = {1,2,3,4,5};
```

Nor is

```
int a[5]; a[5] = {1,2,3,4,5};
```

initialization is done at compile time and must be done at once (otherwise you need to manually enter entries)

Summing an Array

```
#include <stdio.h>
int main(void) {
   int a[] = {10,-7,3,8,42};
   int sum = 0;
   for(int i=0; i< 5; i++){
      sum += a[i];
   }
   printf("%d",sum);
   return 0;
}
```

Sieve of Eratosthenes

- Question: Find all prime numbers up to n.
- Idea:
  - List all the numbers from 2 to n.
  - Start at 2 and cross out all the multiples of 2.
  - Choose the next highest number not stricken out and repeat.
  - Stop when next number is greater than $\sqrt{n}$.
  - The remaining numbers are prime.
- For example, iteration 1:

	2	3	~~4~~	5	~~6~~	7	~~8~~	9	~~10~~
11	~~12~~	13	~~14~~	15	~~16~~	17	~~18~~	19	~~20~~

Sieve of Eratosthenes

- Question: Find all prime numbers up to n.
- Idea:
  - List all the numbers from 2 to n.
  - Start at 2 and cross out all the multiples of 2.
  - Choose the next highest number not stricken out and repeat.
  - Stop when next number is greater than $\sqrt{n}$.
  - The remaining numbers are prime.
- For example, iteration 2:

	2	3	~~4~~	5	~~6~~	7	~~8~~	~~9~~	~~10~~
11	~~12~~	13	~~14~~	~~15~~	~~16~~	17	~~18~~	19	~~20~~

main.c

```
#include <stdio.h>
#include <assert.h>
void sieve(int a[], int n);
int main(void) {
  int n; scanf("%d", &n);
  assert(n > 0);
  int a[n+1];
  sieve(a,n+1);
  for(int i=0; i<n; i++){
    if (a[i]) printf("%d\n",i);
  }
  return 0;
}
```

sieve.c

```
void sieve(int a[], int m){
  a[0] = 0;
  if (m==1) return;
  a[1] = 0;
  if (m==2) return;
  //Set potential primes
  for(int i=2; i < m; i++) a[i] = 1;
  for(int i=2; i*i <= m-1; i++){
    if (a[i]){
      //strike out multiples
      for(int j = 2*i; j < m; j += i) a[j] = 0;
    }
  }
}
```

Important Note

- Notice in the code that we passed an array and our original array was actually modified.
- What's really being passed to the function is not the array, but rather where the array exists in memory.
- Thus, when you make changes inside the array, it is actually changing the memory location that our array was specified to live in.
- Think back to the filing cabinet analogy.
- More on this later.

Exercise

Write a function `missing` that consumes an array of 99 distinct integers from 1 to 100 with one exception and returns the integer that is missing.

Array Sizes

- In all of our examples we know the size of our arrays.
- Ideally we wouldn't have to pass this information to our functions - the array itself should encode how large it is.
- We can access this information using the `sizeof()` operator, which takes as a parameter either a type or a variable and returns the number of bytes used to store a type or variable.
- Technically, it returns an unsigned integer of type `size_t`, which is an unsigned type.
- You might need to typecast this in certain situations to convince C that this is an integer (more on this later).

sizeof() Examples

Below %lu refers to "unsigned long". The %zu is preferred for cross-platform compatibility.

```
#include <stdio.h>
int main(void) {
  int a[] = {-1, 2, -1, 2, -4, 1};
  printf("%d\n", sizeof(a)/sizeof(a[0]));
  printf("%d %d %d %d\n",
    sizeof(char), sizeof(int),
      sizeof(a[0]), sizeof(a));
  printf("%lu %lu %lu %lu\n",
    sizeof(char), sizeof(int),
      sizeof(a[0]), sizeof(a));
  printf("%zu %zu %zu %zu",
    sizeof(char), sizeof(int),
      sizeof(a[0]), sizeof(a));
  return 0;
}
```

Passing Arrays to Functions

Let's take a closer look at passing an array into a function

```
#include <stdio.h>
void plusOne(int a[], int n){
  for(int i=0; i<n; i++) a[i]++;
}
int main(void) {
  int a[10] = {0,1,2,3,4,5,6,7,8,9};
  //Below, a is really a memory address.
  plusOne(a, sizeof(a)/sizeof(a[0]));
  printf("%p\n", a); //mem address
  return 0;
}
```

Pointers

- Key note: An array (by name) is a pointer to something in memory.
- `int *a;` - a pointer to an integer.
- So for example, in the function definition before,

  ```
  void plusOne(int a[], int n);
  ```

  is equivalent to

  ```
  void plusOne(int *a, int n);
  ```

Caution!

Watch out! It's a trap! What does this code print?

```
#include <stdio.h>
void sizeofArray(int a[]){
  printf("%zu\n", sizeof(a));
}
int main(void) {
  int a[] = {1,2,3,4,5,6,7,8,9,10};
  printf("%zu\n",sizeof(a));
  sizeofArray(a);
  return 0;
}
```

Probably...

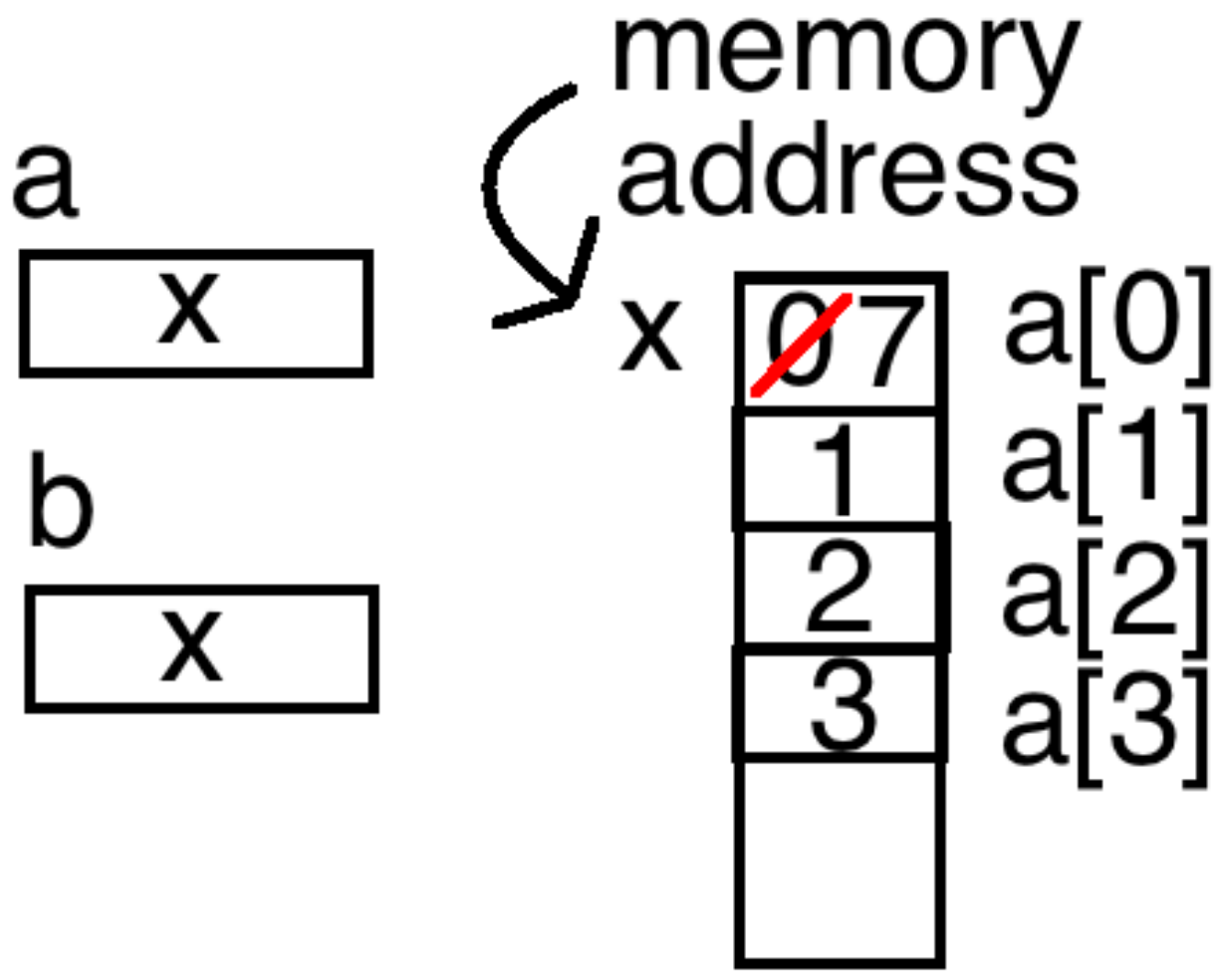
- ... not what you expected. Remember that the a in the function is really just a memory address and so has the size of a memory address location (which in this case is 8 bytes).
- The sizeof(a) inside `main` is actually the size of the array (so the size of all the elements in the array). In this case, this is 10 times `sizeof(int)`
- This awkwardness is why we need to pass the size of the array to functions.
- This is called **pointer decay**.

More Fun - References

What about this code?

```
#include <stdio.h>
int main(void) {
   int a[] = {0,1,2,3};
   int *b = a;
   printf("%d\n", a[0]);
   b[0] = 7;
   printf("%d\n", a[0]);
   return 0;
}
```

This Actually is Reasonable:



Copying Arrays

- So what if you wanted an exact but distinct copy of an array?
- For now, one way to achieve this is to create a new array of the same size and then manually copy everything into it. (Note: C99 and beyond allowed for variable length arrays)

```
#include <stdio.h>
int main(void) {
   int a[3] = {2,4,6};
   int len = sizeof(a)/sizeof(a[0]);
   int b[len];
   for(int i=0; i<len; i++) b[i] = a[i];
   a[0] = 19;
   printf("%d",b[0]);
   return 0;
}
```

Multi-Dimentional Arrays

C can also deal with arrays of arrays:

```
#include <stdio.h>
int main(void) {
  int matrix[4][3] = {{0,1,2},
      {10,11,12},{20,21,22},{30,31,32}};
  return 0;
}
```

We say this array `martix` has four **rows** and three **columns**.

Conceptually:

0	1	2
10	11	12
20	21	22
30	31	32

However...

In memory, this is stored in **row major** order:

0	$a[0][0]$
1	$a[0][1]$
2	$a[0][2]$
10	$a[1][0]$
11	$a[1][1]$
12	$a[1][2]$
20	$a[2][0]$

Example

Let's sum all the elements in a two dimensional array

```
#include <stdio.h>
int sumMulti(int a[4][3]) {
   int sum = 0;
   for (int i=0; i < 4; i++){
      for (int j=0; j < 3; j++){
         sum += a[i][j];
      }
   }
   return sum;
}
```

Final Notes

- The line

  ```
  sum += a[i][j];
  ```

  can be replaced with

  ```
  sum += a[0][i*3+j];
  ```

  (Think back to how the memory model works!
- When initializing, all dimension except possibly the first must be explicit. This holds also in function definitions.
- Example: `int a[][2] = {{1,2},{3,4}};`
- We will return in a few weeks to discuss more about pointers in more detail.