CMSC 216

Introduction to Computer Systems

Ekesh Kumar

Dr. Nelson Padua-Perez • Summer 2019 • University of Maryland

http://www.cs.umd.edu/class/summer2019/cmsc216/

Last Revision: June 28, 2019

Contents

1 Tuesday, May 28, 2019

Logistics

1. All lectures are recorded and posted online.
2. No pop quizzes, no collaboration on projects.
3. Website sign-in: cmsc216/sprcoredump.
4. Office hours are immediately after class in IRB 2210.
5. Everybody will get an Arduino – this will be used later in the course.
6. This class isn't curved.

Basic Unix Commands

Unix has lots of commands, but we want to first focus first on the ones that'll let us write and execute C programs.

- `pwd` → displays your current directory.
- `ls` → displays the files/directories in the current directory.
  - `ls -al` → lists all of the files and directories, including hidden ones (Here, the `a` flag functions to show hidden files, whereas the `l` flag functions to list all entries with detailed information, like last date accessed).
  - `ls -F` → identifies directories by listing them with a /.
- `cd` → change directory to the inputted parameter.

Introduction to C Programming

In CMSC131 and 132, we learned Java. Unlike Java, C is not object-oriented; it has no concept of classes, objects, polymorphism, or inheritance. However, C can be used to implement some object-oriented concepts, like polymorphism or encapsulation. Consider the following program:

Listing 1: A First Program

```c
#include <stdio.h>
int main() {
    printf("Fear the turtle\n");
    return 0;
}
```

How does this program work?

- The `#include` allows the compiler to check argument types. It can compile without declaration, though the compiler will warn you.
- Like Java, C provides a definition of the `main()` function, where all C programs begin.
- We return from `main()` to end the program. For standard practice, we return 0 to signal that everything worked out fine.

Now, let's say we want to run this program. How can we do this? C programs need to be compiled before they can be executed. With the `gcc` compiler, a very simple compilation command is `gcc file.c`, from which we can run the executable by just typing `./file`.

Some more compilation options are summarized below (these are called **flags**):

- `-g` enables debugging by generating and maintaining necessary symbols (e.g. line numbers) upon compilation.
- `-Wall` warns about common things that might be a problem.
- `-o filename` places an executable in the file name.

2 Wednesday, May 29, 2019

Last time, we analyzed a sample C program. It's important to know that returning 0 in the `main()` function is independent of the `void` that appears in the main's header. That is, even if our header is `int main(void)` instead of `int main()`, we will still return 0 at the end of the function. The `void` is just an explicit way of telling our compiler that we shouldn't be passing any parameters in.

More Unix Commands

Some more Unix:

- The `cp` command makes a copy of a file from a source to a destination. Some options are `-a`, which allows us to preserve attributes, like timestamp modified. Also, `-v` explains what's being done, while `-r` copies recursively.
- The `-rm` command removes a file.
- The `-mv` command renames a file or moves a file/directory to another directory. For example...
  - `-mv f1 f2` renames file `f1` to `f2`.
  - `-mv f1 d1` moves the file `f1` to the directory `d1`.
  - Finally, `-mv d1 d2` moves the directory `d1` to `d2`.
  - The `-cat` command displays the contents of a file.

In Unix, we can create **aliases**, which are shortcut commands to use a longer command. Users can use the alias name to run the longer command while typing less. Without any arguments, the `alias` command prints a list of defined aliases. A new alias is defined by assigning a string with the command to a name. We can add an alias by modifying the `.aliases` file in the home directory of Grace.

The general format for defining an alias is `alias [alias name] 'command'`. So adding the line `alias cookies 'ls'` would define the command `cookies` to do the same thing as `ls`

Compilation Stages of a C Program

C programs need to be compiled before they can be executed. What happens when we compile a C program? There are **three compilation stages**:

1. **Preprocessor Stage:** This stage is used to verify that program parts sees declarations that they need. Also, statements starting with a # are called **directives** (for example,
2. **Translation:** In this stage, an object (`.o`) file is created. In addition, the compiler checks to make sure that individual files are consistent with themselves.
3. **Linkage:** Finally, this stage brings together one or more object files. It makes sure that the caller/calee to functions are consistent. The result is an executable file (by default, it's named `a.out`),

Variables in C

There are a lot of data types in C, some of which include `char, short, int, long int, float, double`, etc. In Java, data types take up the same amount of space, independent of the system they're run on. This is not true in C; the minimum size of various data types are not necessarily the same size on grace. We do not need to memorize the sizes of various data types; however, it is important to know that a `char` data type is an exception to this rule: it always takes one byte.

Also unlike Java, there is no maximum size for a type; however, the following inequalities hold:

$$\texttt{sizeof(short)} \leq \texttt{sizeof(int)} \leq \texttt{sizeof(long)}$$

$$\texttt{sizeof(float)} \leq \texttt{sizeof(double)} \leq \texttt{sizeof(long double)}$$

Suffixes allow us to specify a number of a given type. For instance, `30000` is of type `int`, whereas `30000L` is of type `long`.

In C, there is no default `boolean` data types; anything with value 0 is considered false, whereas any other value is considered true. However, we can use integers to represent booleans with `true` mapping to 1 and `false` to 0.

Consider the following code example:

Listing 2: Conditional Example

```c
#include <stdio.h>
int main() {
    if (100) {
    printf("Fear the turtle\n");
    }
    return 0;
}
```

The print statement in conditional executes successfully for reasons described above.

3 Friday, May 31, 2019

printf() and scanf()

When we're using `printf()` to print something, we just print anything that's in the quotations. For instance, the line `printf("Hello");` would print "Hello," as we desire.

To print variables, we use **conversion specifications**, which begin with the `%`. These are just placeholders representing a value to be filled in during printed. More specifically, the `%` *specifies* how the value is *converted* from its internal binary form to characters. For instance, the conversion specification `%d` specifies that `printf` is to convert an `int` value from binary to a string of decimal digits. In summary,

- `%d` for integers,
- `%c` for chars,
- `%f` for floats,
- `%s` for strings (null-terminated char array)
- `%x` for hexadecimal form
- `%e` for exponential form
- `%u` for unsigned integer.

For example, the following code segment will print `i = 10`.

Listing 3: Printing a Variable

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

`scanf()` is used for user input and it works similarly. The introduce the **address** operator, which is denoted by a `&`. The address operator is a unitary operator which, as its name specifies, returns the memory address of the variable on which it is acting on. When `scanf()` is called, it starts processing the information in the inputted string, from left to right. For each conversion specification in the format string, `scanf()` attempts to locate an item of the appropriate type in the input data, skipping any blank space if necessary.

Here's an example:

Listing 4: Reading Variables

```c
#include <stdio.h>
int main() {
    int i, j;
    float x, y;
    scanf("%d%d%f%f, &i, &j, &x, &y");
}
```

Now suppose that the user enters the line

```
1 -20 .3 -4.0e3.
```

The code above will convert its characters to the numbers they represent and assign the values $1, -20, 0.3, -4000.0$ to the four variables.

Finally, one should keep in mind that `char` variables are actually just integers that map to an ASCII character. So something like `printf("%c", 65)` works completely fine; it prints the character `A`.

There are two important things that one should check when using `scanf()` and `printf()`:

1. Check that the conversion specifications match the number of input variables and that each conversion is appropriate for the corresponding variable (as should also be done with `printf()`. Since the compiler doesn't necessarily have to check for mismatches, there won't be any warning.

2. Another trap involves the `&` symbol, which should precede each variable in a `scanf` call. Forgetting to put it can lead to unpredictable results. It is wrong to use the address-of operator in a printf() statement.

A **segmentation fault** error occurs when the program attempts to access an area of memory that it should not be accessing. Why is it called a segmentation fault? Because the content of memory at the time of crash is stored into a **core file.**

We can get segmentation faults when using `scanf()` or `printf()` if we try to read into or print some variable that we don't have access to.

Control Statements

C has `if/else, for, do-while,` and `switch` statements, just like in Java. But due to the compiler flags in our submit server, we won't be allowed to declare variables in the `for loop` header.

There's also `break` and `continue`, but they are bad practice and shouldn't be used often.

Functions

C functions have the following format to create a function `returnType functionName(parameter list) { ... }` Just like in Java, to call a function, we just write `functionName(argument list);`.

However, if the function appears after the main, then we need to do something called **function prototyping**, which is just declaring the function before the main. This isn't necessary if we implement the function before the main, though. Function prototypes don't actually need the name of the variable, but it's easier to read with them.

In C, variables are passed in **by value**. This is different from Java, in which variables are passed in by reference. Some other things that are similar/different are:

- We can have recursion in C.
- We can **not** have function overloading. In particular, we can point out that `printf` and `scanf` are not overloaded functions; they refer to the same function!

4 Monday, June 3, 2019

The sizeof Operator

Before we talk about pointers, first we need to talk about the `sizeof` operator. The `sizeof` is a unitary operator tells us how many bytes are associated with a particular entity. This is an important operator when we're doing dynamic memory allocation. For instance, suppose we don't know how much memory to allocate when we're storing 10 integers. Then, we can do something like 10·`sizeof`

It is important to note that the `sizeof` operator does **not** evaluate the expression; for instance, doing something like `sizeof(x++)` will not increment `x`. It just looks at the type of what's inside.

Introduction to Pointers

A pointer is declared using the `*` symbol, right before the variable name. Consider the following code example:

Listing 5: Pointers Example

```c
#include <stdio.h>
int main() {
    int y = 5;
    int *p;
}
```

Here, y is a standard integer variable, holding the value 5. By contrast, p is a pointer variable whose value is garbage. But each of these don't only have a value – they also have a **memory address**, which can also be represented by an integer. For example, the memory address of y might be 2000; p doesn't have a memory address yet. A program refers to a block of memory using the address of the first byte in the block.

Now let's say we add another line of code:

Listing 6: Pointers Example

```c
#include <stdio.h>
int main() {
    int y = 5;
    int *p;
    p = &y;
}
```

Recall that the `&` symbol is the address-of operator. So at this point, p stores the memory address of y, namely, 2000. Now, we can do something like `printf("%d", *p)`, like we're used to. Also, whenever we change `*p`, we also change the value of y. **In summary, a pointer is a variable that stores a memory address.**

Why do we need the type when we declare a pointer variable? We need to know the number of bytes to grab. Since it's an integer here, we know to grab four bytes.

Now what if we want to read in the pointer using `scanf`? Then we don't need to use the `&` operator on the pointer – the pointer already refers to a memory address! To make this more clear, consider the following code:

Listing 7: Pointers Example

```c
#include <stdio.h>
int main() {
    int age, values_read;
    int *age_ptr = &age;
    printf("Enter your age and salary");
    scanf("%d %f", age_ptr, &salary);
}
```

When we're taking in `salary`, we need to use the `&` operator since we want to retrieve the address. By contrast, we don't need the `&` operator for `age_ptr` since it already stores a memory address.

Pointers as Parameters

Recall that parameters in C are passed **by value**. To demonstrate this, consider the following code example:

Listing 8: Variables Passed by Value

```c
1 #include <stdio.h>
2 int main() {
3     int y = 7;
4     f(y);
5 }
6
7 void f(int x) {
8     x = 200;
9 }
```

When the code above is executed, the value of y doesn't change – we're passing a copy of y into the function. That is, the value of y is 7 even after Line 4 executes.

Now consider the following function `wrong_swap` below:

Listing 9: Variables Passed by Value

```c
void wrong_swap(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}

int main() {
    int x = 2, y = 3;
    wrong_swap(x, y);
}
```

When the function terminates, the variables a and b are destroyed. The variables x and y **are not** swapped. The reason why is, again, because parameters are passed by value in C. So how can we swap variables, if we're only returning one value? This can be done with pointers, where the same idea of passing-by-value holds. Here's the correct way to swap —

Listing 10: Variables Passed by Value

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

int main() {
    int x = 2, y = 3;
    int p = &x;
    int q = &y;
    swap(p, q);
}
```

This is the same idea, but why does it work? Because we can dereference the pointer. We're note actually changing x and y – we're changing their memory addresses. So, this works.

5 Tuesday, June 4, 2019

The **comma operator** in C is used to separate expressions. It's a binary operator that evaluates its first operand and discards the result. It then evaluates the second operand and returns this value. For instance, `y = (3, 4);` is a valid expression, which assigns the value 4 to `y`.

Identifier Scopes

There are two main types of scopes in C:

- The **block scope** contains variables declared inside a block, and it is only visible within the block. They do not exist outside of the block.
- The **file scope** contains identifiers declared outside of any block; it is visible everywhere in the file **after** the declaration.

In the heap segment, text and data are constant from start to the end of the program. Execution follows the text segment of the memory. The data section contains global and static variables. Finally, the stack stores local variables and function parameters. There's some extra space in the heap which is used for dynamic memory allocation. The stack and heap grow in opposite directions, which is convenient to prevent overlapping. The heap goes up, and the stack goes down.

There are two types of storage types:

- **Automatic storage** occurs when the variable is transient. That is, after some time, it is no longer returned (e.g. when a function returns).
- **Static storage** occurs when the variable exists throughout the entire life of the program. Global variables have this kind of storage, and initialization to static variables only occur once.

You can make a block-scoped variable static, which would be important when you're counting the number of times a function executes.

A **linkage** is a property of an identifier that determines if multiple declarations of that identifier refer to the same object.

There are two main types of linkage that we should know about:

1. **Static linkage** is performed in the final step of compilation; it is fast, and it can be referenced from anywhere within the same file.
2. **Dynamic linkage** is performed during runtime; it is slower; however, it can

6 Wednesday, June 5, 2019

Invalid Uses of Pointers

Consider the following code segment:

Listing 11: Incorrect Pointer Usage

```c
#include <stdio.h>
int main() {
    int *p;
    *p = 200; /* This is wrong! */
    printf("The value is %d\n", *p);
    return 0;
}
```

This is wrong, and it might generate a segmentation fault error. Why? We need p to be associated with an area of memory that is valid.

A quick fix is to initialize a variable, and assign p to the memory address of that variable. For example, the code segment below is correct, and it will print 200.

Listing 12: Correct Pointer Usage

```c
#include <stdio.h>
int main() {
    int *p;
    int x;
    p = &x; /* This is correct! */
    *p = 200;
    printf("The value is %d\n", *p);
    return 0;
}
```

The first code segment doesn't work correctly because the pointer is not initialized. Pretty much, we've created a pointer to "anywhere you want," which can be the address of some other variable, or some nonexistent memory.

When you have a program in C, there are four areas of memory: the **stack**, **heap**, **data**, and **code**. If some amount of memory is allocated for a function process, that memory becomes deallocated after the function is finished. So, we don't want to be messing with memory that no longer exists. For instance, the following code example is bad:

Listing 13: Incorrect Pointer Usage

```c
#include <stdio.h>
int* process() {
    int x = 10;
    int *p = &x;
    return p; /* This is bad - we're returning a
        pointer to some area that no longer exists! */
}
```

Even if the program seems to work, the local variable disappears – the space for it is gone, and we're not supposed to be messing with the memory that it used to be in.

Remark 6.1. We can print the memory address of a pointer using `printf` with the format specifier `%p`.

Null Pointers

The **null pointer** is a special pointer that points to the address 0, where nothing is allowed to be accessed. It's analogous to Java's null, except we use `NULL` rather than `null`.

You can assign null to any kind of pointer variable, and we also need to check if they're null prior to derefering them; using a simple `if (p != null)` conditional works.

Also, `null`'s numeric value is equal to zero, so a conditional statement with them will not execute.

Introduction to Arrays

Arrays are a bit different in C when compared to arrays in Java. In C, an **array** is just a chunk of bytes, one after another. We can declare an array of integers doing something like `int a[3]`, and indexing works the same as Java (starting at zero). Note that when we make the declaration `int a[3]`, the default elements are **not** zero (like in Java); instead, they are all garbage values. Also, you can't use a variable to declare the size of an array, but you can use it for indexing.

Note that an array is **not** an object, meaning that things like `a.length` don't exist. We need to keep track of the length ourselves. This can often be done with **constants**, which start with the `const` keyword. For now, we assume arrays are not dynamic in terms of their space.

If the array has three elements, then the size of the array is actually 12 bytes (four bytes per integer). We can use the operator `sizeof`, and something like `sizeof(a)` will return 12.

Some examples of array declarations are as follows:

- `int a[3] = {10, 20, 30};` will declare an array `a` of length three, with the three elements listed.
- `char b[] = {'A', 'B', 'C'}` will declare an array with size 3 with the provided elements. Note that we don't need to specify the length when we're initializing by list.
- `float c[4] = {1.5}` will declare an array of size 4 with first element equal to 1.5. The other elements will equal 0. This is really convenient because we can do something like `int a[3] = {0};` to initialize our array of length three to have all zero elements.

Arrays as Parameters

Recall that everything in C is passed by value.

7 Monday, June 10, 2019

The `pushd` and `popd` commands in Unix can be used to work with a directory stack. The command `pushd` pushes a directory on top of the directory stack, and the `popd` command returns to the path at the top of the stack.

The `history` command prints your most recent commands.

Pointing to a Local Variable

When working with pointers, you shouldn't return the address of a local variable. Consider the following code segment:

Listing 14: Incorrect Pointer Usage

```c
1  int* get_value-wrong() {
2      int x = 20;
3      return &x;
4  }
5
6  int add_value(int y) {
7      int a = 99;
8      return a + y;
9  }
10
11 int main(void) {
12     int *a, b;
13
14     a = get_value_wrong();
15     printf("First result %d\n", *a);
16
17     b = add_value(7);
18     printf("Second value %d\n", b);
19
20     printf("First result (changed?) %d\n", *a);
21 }
```

The first print statement on Line 15 will work fine; it'll print out 20 as we'd want it to. This first part might seem counterintuitive because we usually think that the memory gets "thrown away" after the function finishes execution. In reality, this isn't what happens – the stack pointer just moves down, below the local variable. This tells our computer that the previously occupied area of memory is now available for reuse. In our case, the space occupied by the integer `x` will now be available for reuse.

But then, after we call the `add_value` function, the first result will have changed. Since we're declaring another local variable of the same type (integer), the same space that was previously being used will be filled for the second function call. The space that was previously holding the number 20 will now hold 99. Once again, after the function finishes execution, the stack pointer moves below the 99 again (but it does not disappear!).

And so, the print statement on Line 18 prints 106, and the print statement on Line 20 will print out 99.

String Comparison

To compare strings, we use the `strcmp` function, which is built in `string.h` library. The function header is as follows:

`int strcmp(const char *s1, const char *s2);`

Pretty much, it takes in two strings `s1` and `s2`, and it returns a negative number if `s1` is (lexicographically) less than `s2`, the integer 0 if they're (lexicographically) equal, and a positive number otherwise. Note that the `const` in the parameter list of `strcmp` indicates that the data of `s1` and `s2` can't be changed (this makes sense because changing the strings isn't necessary for comparison).

Remark 7.1. This functionality is pretty much the same as `compareTo` in Java

Here's one way to implement the `strcmp` function. Nelson says we should know this implementation for the exam.

Listing 15: String Comparison Implementation

```c
int strcmp(const char *s1, const char *s2) {
    int i;
    for (i = 0; s1[i] && s2[i]; i++) {
        if (s1[i] != s2[i]) {
            break; /* Gets us out of the for loop */
        }
    }
    return s1[i] - s2[i];
}
```

- The for loop iterates until we reach the end of either string (the Boolean expression `s1[i] && s2[i]` is false only if we hit a null character in either string) or if we hit two characters that are different (this is what the conditional inside the loop does).
- Once we're at the differing character, we can just return their difference.

Note that the above implementation uses the fact that the null character has numeric value 0. This allows the code above to take care of the cases in which one string is shorter than the other.

Copying Strings

To copy a string, we use the `strcpy()` funcion, which is built into the `string.h` library. The header is as follows:

`char* strcpy(char *dest, const char *src).`

The function copies the string in `src` to the string in `dest`. After successful completion, the function returns a pointer to the destination string.

The danger with `strcpy()` is that it doesn't specify the size of the destination array, which can lead to a **buffer overflow error**. This type of error occurs when you put more data into a fixed-length buffer. The extra information has to go somewhere, and it can overflow into adjacent memory space, which corrupts other data.

Here's an implementation of the function `strcpy()` function:

Listing 16: String Copy Implementation

```c
int strcpy(const *dest, const char *src) {
    int i = 0;

    while (src[i]) {
        dest[i] = src[i];
        i++;
    }
    dest[i] = '\0';
}
```

- We are copying the source into the destination.
- The while loop executes until we hit the null character in the source.
- We copy the first, second, third.... character into the destination.
- When the while loop executes, i is equal to the position where source has a null character, so we need to add that into the destination string.

Again, take note of the parameters. The string `dest` isn't constant because we're modifying it.

String Literals

The declarations `char name[] = "Mary"` and `char *name = "Mary"` are not the same. The first declaration is an array, which is what one should use if they're planning to change the value of the name from Mary to something else. On the other hand, the second declaration declares name as a pointer to a string literal. This should instead be declared as `const char* name = "Mary"`.

Void Pointers

A **void pointer** or **generic pointer** (they are the same) is a special pointer that's used to hold memory addresses of data when you don't know its type. They are used by functions (like C's built-in quicksort function) when you don't know the type of data you're dealing with. Void points are declared by simply replacing the type of a normal pointer with the void keyword. So, `void *ptr` would declare `ptr` to be a void pointer.

A void pointer can point to any type. So, you would be able to do `ptr = &someInt` or `ptr = &someChar`, etc. But integers, floats, and characters occupy different amount of space. So how does this work? The key is to note that pointer variables store the address of the first byte.

Note that **a void pointer cannot be dereference directly**. Dereferencing requires casting because the pointer needs to know how many bytes to grab. It's up to the user to make sure that the void pointer is casted right. So if you read in a float value into `ptr`, the statement `printf("V1: %f\n", * (float *) v_ptr)` would print the entity at the address stored in `v_ptr`. Note how this print statement has two asterick symbols: one is used in the cast, whereas the other is used for dereferencing.

It's also a good idea to use type casting when you're doing pointer arithmetic with void pointers. For example, consider the following code segment:

Listing 17: Bad Void Pointer Arithmetic

```
int one_d[5] = {12, 19, 25, 34, 46}, i;
void *vp = one_d;

printf("%d", one_d + 1); // bad
```

You might want to print the second element of the array with the above code. But this won't work. Adding a number to a memory address works by shifting logical units. However, these logical units are dependent on the type being worked with. Changing the fourth line to `printf("%d", (int *) one_d + 1)` would fix this.

The value of a void pointer can be assigned to integer/float/other pointer variables **without a cast**. For example, if we have a float pointer `f_ptr` and a void pointer `v_ptr`, the statement `f_ptr = v_ptr` works perfectly fine because it's just specifying how many bytes to grab.

Pointers to Pointers

You can have pointers to pointers (and even pointers to those pointers). The number of astericks indicates the degree-of-separation from the original variable. For instance, `int **p2` is a pointer to a pointer. Once `p2` has properly been initialized, we can do double dereferencing by typing `** p_{2}` to get the value of the original variable.

When do we use pointers to pointers? Consider a function we're writing that needs to modify a pointer. This would need to be implemented by taking in a pointer to a pointer;

8 Wednesday, June 12

The `grep` command in Unix looks for a pattern in a file. The general syntax is `grep [pattern] [file_name]`. So for example, if you wanted to find all instances of ”cheese” in ”homework.c,” you could execute `grep cheese homework.c` to get this result (note how there aren't any quotes).

Command Line Parameters

So far, our main function's header has always been `int main(void)`, which indicates that the main method doesn't take in any parameters. However, it is possible to accept command line parameters into the main by instead using the header `int main(int argc, char **argv)`. This second form allows us to access command line arguments as well as the number of arguments specified (arguments will be separated by spaces).

In summary, the two arguments that the `main` function accepts in this second formulation are

- `int argc`, which represents the number of arguments passed into the program when it's run. This number needs to be at least 1.
- `char **argv`, which is a pointer to a character pointer. We can alternatively replace `char **argv` with `char *argv[]`, which is an array of character pointers.

For instance, consider the following program:

Listing 18: Command Line Parameters

```c
struct pixel {
    int x, y;
    char color;
};
```

We can pass in parameters through command line by typing, for example,

`./a.out hello my name is ekesh`.

The output is presented below:

```
argv[0]: ./a.out
argv[1]: hello
argv[2]: my
argv[3]: name
argv[4]: is
argv[5]: ekesh
```

Note how even `./a.out` counts as one of the strings processed. If we don't want this to happen, we can just treat the 0th index in the array as a sentinel. Also, keep note that !argv[i]! is a string. If we want to use a passed in value in, say, a loop, then we need to use the `atoi()` function, which converts a string argument into an integer.

Two-Dimensional Arrays

When you're passing in a two-dimensional array into a function, the first array dimension (i.e. the number of rows) does not have to be specified. The second (and any subsequent parameters, if we're working with more than two dimensions) need to be specified.

So, for example, a function with header `void print(int arr[][n], int m)` would be fine, whereas something like `void print(int arr[][], int m)` wouldn't work.

Obviously, this would only work if the second dimension is fixed and isn't user-specified; this is a clear drawback.

Two-Dimensional Character Arrays

Consider a two-dimensional array of characters declared as follows: `char friends[100][81]`.

Typically, we can view a two-dimensional character array as a one-dimensional array of strings. For example, `char friends[100][81]` would store 100 strings, each of which have a maximum length of 80. We can then access the i^{th} friend stored in the array by standard one-dimensional array indexing, like `friends[0]`. However, it's up to the programmer to verify that the null character is present at the end of each row in the array.

Since two-dimensional arrays are stored in row-major order, executing `strcpy(a[0], "12345")` to copy a five-character-long string into an array with column-length less than 5 will still work; however, the "trailing characters" will go into the next row. There is no compilation error here, though.

The typedef Keyword

The **typedef** keyword is used in C to create an alias for another data type. The general syntax for declaring a typedef is `typedef [data_type] [new_name];` By convention, the `new_name` of a data type usually starts with a capital letter.

The main reasons why we use typedefs are to improve code readability and maintainability. As per convention, it's good to start `new_name` with a capital letter so that we can distinguish it from other types.

It is important to note that the `typedef` and `#define` preprocessor are not the same: the `#define` preprocessor works by by blindly substituting what we're defining, whereas `typedef` actually defines a new type. In fact, a typedef is not a preprocessor directive; **typedef is a compiler token**, and the preprocessor doesn't care about it at all.

An Exception to Typedef

An exception to the standard `typedef [data_type] [new_name]` syntax for defining a typedef is when we're dealing with arrays. Pretty much, if we're typedef'ing something to become an array, from what we've learned, we would expect to write something like `typedef int[30] MyArray`. However, **this is wrong**. The correct way to do this would be to write `typedef int MyArray[30]`; the size of the array comes after the `new_name` identifier. This is an exception, and `MyArray` will now represent an array of 30 integer elements.

This exception also applies to multi-dimensional arrays.

Structures

Defined in terms of Java, a structure is a class without methods and without private fields.

More formally, a **structure** is a user-defined data type which allows one to group items of possibly differing types into one single type.

The basic syntax for declaring a structure type is `struct [struct_tag] { [member_list] };` (note how there is a semicolon at the end). Conventionally, structures are typically declared at the top of a program, before the main. Conventionally, structure tags begin with a lowercase letter.

Suppose we are writing a program that involves computer graphics. We might want to have a structure to represent a pixel. This structure should abstract the basic details about a pixel, like its x and y-coordinates and its color. This can be done with the following code:

Listing 19: Structure Example

```c
struct pixel {
    int x, y;
    char color;
};
```

Here, we've declared a structure with the tag "pixel," which contains an integer `x`, an integer `y`, and a character `color`.

Fields in structures cannot be initialized (so, it would be invalid to set `x` and `y` to 0 by default in the above example). Why can't we initialize fields in structs? Basically, when the structure is declared, there isn't any memory allocated for it (there's no reason to allocate memory yet – we don't even know if the program will ever use the structure). Memory is allocated only when variables are created, so there isn't any space to actually declare a variable yet.

Once we've declared the `pixel` struct, we can declare a variable `p1` of its type by typing `struct pixel p1;`. The members of `p1` can be accessed by using the period: `p1.x = 50` would set the x variable associated with `p1` equal to 50.

C also supports using an initialization list to initialize a structure. For example, we could write `p1 = {1, 2, 'r'}` in order to declare `x, y,` and `color` to $1, 2,$ and 'r' respectively. The order in which the variables are provided is the same order in which these variables are assigned values. If we don't assign all of the values, their default values will be assigned.

There aren't any conversion specifiers that allow us to directly print out all of the variables associated with a structure (side-note: this is called a **reflection**). If we want to do this, we need a conversion specifier for every variable in the structure.

A couple of other things to remember:

1. Structures can be assigned to each other. For instance, $a = b$ will compile, and it will assign all of the field values of b to the corresponding fields in a. This performs a shallow copy.
2. Structures cannot be compared. The line $a == b$ will not even compile. Even structures with the same fields in the same order aren't compatible.

Combining Typedefs with Structs

Following the `typedef [data_type] [new_name]` syntax for declaring a new data type, we can typedef a structure in order to get rid of the `struct` that's usually necessary when declaring a structure.

For example, consider the pixel example from above. It was necessary to write `struct pixel p1;` to declare a pixel. However, if we modify the code to what follows, we can instead write `Pixel p1;`.

Listing 20: Typedef'ing a Structure

```c
typedef struct pixel {
    int x, y;
    char color;
} Pixel;
```

We usually typedef a structure for brevity and readability.

Pointers to Structures

Since everything is pass-by-value in C, when we pass in a struct as a parameter to a function, we'll have a copy of the structure with every value equal to the original value's corresponding fields. Like we'd expect, this would mean that changing the structure inside the function doesn't change the original structure outside of the function (i.e. a shallow copy is performed). Like always, if we want to modify the actual structure, we need a pointer to the structure.

When we're dealing with pointers to structures, there's an **arrow operator**, which is used to dereference a pointer to a structure. Going back to our pixel example, for instance, if we have the pointer `p1` defined as `Pixel * p1`, we can set the structure's associated value of x equal to 50 by writing `p1->x = 50`.

Why do we need the arrow operator? There's nothing wrong with writing `(*p1).x = 50` – it does the same thing. But, something like `*p1.x = 50` **does not work** for precedence reasons. Hence, having the arrow operator improves readability instead of having a lot of parentheses and astericks.

9 Friday, June 14, 2019

The `touch` command in Unix is used to make files on the fly. For instance, `touch bla` would create a 0-byte file named "bla."

The `-F` flag can be used with `ls` in order to appends a forward slash to directory names and appends an asterick to executable files. This can help when identifying different types of files. The `-t` flag can be used with `ls` to list files based on modification time. The `-R` flag can be used to list the contents of every directory recursively. Finally, the `-h` flag displays file sizes in a human-readable format.

Size of Structures

We shouldn't be worried about the size of a structure. Within a structure, the fields aren't necessarily laid out continuously. There's usually some deadweight loss in the memory that a structure is using. Hence, in some cases, adding a field might not even change the size of the structure if the space can be capitalized on. In order to minimize memory loss, one should order fields from longest to shortest. Also due to this lack of memory continuity, you should not perform pointer arithmetic to access fields of structures.

The only thing that we can conclusively say about the size of the structure is that it is *at least* the size of all of its data types combined. Despite these restrictions, we can still have arrays of structures and perform pointer arithmetic within this array.

Unions

A **union** is like a structure, except the memory is shared. That is, all of the fields share the same memory space (so you can only access one field at any given time). Consequently, the size of a union is *always* the size of its largest field.

Unions are particularly used when memory is scarce.

More on Structures

For the first midterm, it's important to remember that assigning one structure to another initializes corresponding values to each other. If there's an array within that structure, both structures will subsequently be pointing to the same array (i.e. changes will be reflected in both structures). However, it's still perfectly valid (and saves time) to just make an assignment whenever possible.

Although the assignment operator doesn't work with arrays, the assignment operator would work with two structures which contains an array.

The tag on a structure is **optional**. We can declare structures without a tag or a typedef such as in the following example:

Listing 21: Tagless Structure

```c
struct {
    int id_number;
    char last_name[10];
    char first_name[10];
    double salary;
} emp1, emp2;
```

The expression in the code above declares `emp1` and `emp2` to be a structure with the fields specified between Lines 2 and 5. However, since the structure is tagless, it's impossible to declare another variable of this type — this means that `emp1` and `emp2` have a unique type.

When do we want a tagless structure?

- If we want a relatively small number of the structures (and the structure becomes useless afterwards)
- We don't want the variables to be passed in as function arguments (there is no type specified, so we can't use them in a function).

This is sort of like a singleton design pattern in Java, where we can enforce only one (or in this case, two) initialization of a structure.

10 Monday, June 17, 2019

Exit Codes

An **exit code** is a value that is returned to the **shell**, which is responsible for reading and executing your code. By convention, when everything goes well, we return 0 (as we have been doing in all of our programs).

The header file `stdlib.h` contains a lot of preprocessor directives, which represent exit codes. For example, `EXIT_SUCCESS` and `EXIT_FAILURE` can be used when the program successfully executes or unsuccessfully executes (these would be used instead of having a line that says `return 0`). It turns out that `EXIT_SUCCESS` is actually a preprocessor directive for 0.

In order to use an exit code, we use the built-in `void exit(int status)` function. So, for example, we could replace `return 0` in the main with `exit(EXIT_SUCCESS)`, and it would mean the same thing. On the other hand, if `exit()` is used *outside* of the main, the program will terminate once it reaches that statement, while a return statement would bring us back to the main.

How do exit codes help us? After executing a program, we can type `echo $?` to check the previous command's exit code. This can be used in shell programming, where we are telling the actual shell what to do.

In addition to exit codes and return values, there are a few important functions that are used to produce **error messages**:

1. The function `void perror(const char *str)` is used to describe the last error encountered during a library function or system call. If a string is provided, that string will be printed prior to the default error description. The default description is generated by a global variable called `errno`, which comes from the `errno.h` header file (i.e. it is an integer-to-string mapping).

2. The `char *strerror(int errnum)` function returns a pointer to the textual representation of the current errno value.

Note that neither of these functions kill the program.

Text and Binary Streams

In C, most input and output is provided in the sequence of bytes, which is more commonly known as a **stream**. There are two types of streams: **text streams** and **binary streams**.

- Text streams consist of lines of text, each of which are terminated by the `\n` character. They can be opened in text editors.

- Binary streams consist of raw data; they require a special editor to open.

What are the advantages of one type of stream over another? When we're using text streams, we can easily debug the program (it's human-readable and doesn't require additional tools, while binary files do). On the other hand, text streams might not be a great idea for when we're modifying files a lot: changing even a single character requires re-reading the entire file. If we were using a binary stream, however, we could (in most cases) just change the relevant bytes.

Standard Input/Output

Now, we'll discuss how to read and write to files.

For files you want to read or write, we need a **file pointer**, declared like `FILE *fp`. Realistically, it isn't really important what the type `FILE` actually is – we can just think of it as some abstract data structure which permits us to perform file I/O operations.

Performing file I/O operations has three key steps:

1. Open the file
2. Perform any processing
3. Close the file

To open the file, we use the `fopen` command, whose declaration is as follows: `FILE *fopen(const char *filename, const char *mode)` . Note that the function returns a file pointer, which we'll set our pointer equal to. If there's any error in opening the file, `fopen` will return `NULL`.

The `filename` parameter is a string, which holds the name of the file on the disk (including a path if necessary), and the `mode` is another string, which represents *how* we want to open the file. In this class, the file will be opened with mode equal to "r" (for reading) or "w" (for writing). Another mode is "a", which lets us append to a file, without losing the rest of its contents.

Once we've opened the file, we're ready for processing. If we're reading the file, we can use the `fgets()` function, whose declaration is specified as follows: `char *fgets(char *str, int n, FILE *stream)`. The parameter `str` in `fgets` stores the line read by the function, and it stops reading until either `n` characters have been read, or a `\n` character is encountered. Note that this `\n` character is also stored as a part of the out parameter. If there are any errors, the function returns `NULL`.

If we're writing the file, we can use the `fputs()` function, whose declaration is the following: `int fputs(const char *str, FILE *stream)`. It places the string `str` into the file `stream`. The function returns a non-negative integer upon success.

Finally, we need to close the file. This requires use of the `fclose()` function, whose declaration is as follows: `int fclose(FILE *stream)`. The function returns 0 upon success, and it signals that we are done processing the file.

The three key steps of file I/O operations are captured with the following code segment:

Listing 22: Processing a File

```c
#include <stdlib.h>

#define MAX_LEN 80

int main() {
   FILE *input; /* does not need to be named input */
   char line[MAX_LEN + 1], filename[MAX_LEN + 1];

   printf("Input file name (e.g., data.txt): ");
   scanf("%s", filename);
   if ((input = fopen(filename, "r")) == NULL) {
      perror("error opening file");
      exit(EXIT_FAILURE);
   } else {
      while (fgets(line, MAX_LEN + 1, input) != NULL) {
         printf("%s", line);
      }
      fclose(input);
      exit(EXIT_SUCCESS);
   }
}
```

On Lines 9 and 10, the program prompts a file name, which is subsequently stored. Line 11 attempts to open the file; upon success, each line is processed and printed. If the file cannot be opened, an error message is printed, and an exit code is returned. Line 18 closes the input stream, and Line 19 returns a successful exit code. Note that when we're printing on Line 16, there's no `\n` necessary. When we perform `fgets()`, we've already stored the new-line character, so adding an additional `\n` will put two spaces between lines.

If we want formatted input and output, we can similarly use `fprintf()` and `fscanf()`.

Nelson says that, at this point, we should be able to write a C program that copies one file to another using command line arguments.

Every program has three defined streams: **standard input**, **standard output**, and **standard error**. We can use the the keyword `stdin` in place of a file pointer to read from the user's keyboard.

Like standard input, standard error is also printed to the screen. It is denoted by the built-in file pointer `sterr`, and it is helpful since it allows us to sort out our print statements, depending on whether a program executed successfully or not.

So, standard input and standard error are different files; however, they both map to the screen. To direct standard error, we can use `> &` in Unix.

The end of a file is denoted by an invisible **end of file** (EOF) character. There's a function with the header `int feof(FILE *fstream)` that checks whether the EOF file has been reached, after the file has been attempted to been open. We can manually enter the end-of-file character with our keyboard by entering `CTRL + D`.

It is important to note that we need to first attempt to read the file before using `feof()`.

11 June 19, 2019

12 Friday, June 21, 2019

Today, we will discuss dynamically allocated memory (i.e. memory that isn't allocated until the program starts running). Why use it? Sometimes, the size of a data structure isn't known until runtime (for example, suppose we want to initialize an array of size N, where N is a positive integer provided by the user). Also, Linked Lists will use dynamic memory allocation everytime we make a new node.

There are two memory management are library functions that are used to allocate memory dynamically: `malloc()` and `calloc()`.

1. The `void *malloc(size_t amount);` function allocates `amount` bytes (if available) from the heap and returns a void pointer to the beginning of it. Note that there cannot be any initialization of this space.
2. The `void *calloc(size_t count, size_t obj_size);` function allocates `count` objects of size `obj_size` each (if memory is available), and it returns a void pointer to the beginning of it. By default, all the space is initialized to zero.

Both `malloc()` and `calloc()` return `NULL` if the allocation fails.

A third memory management function is `void free(void * ptr)` – after this function is called, the memory pointed to by `ptr` is now available for reuse by the memory allocator.

Good programming practice should exhibit a one-to-one mapping between the number of calls to `malloc()` and `calloc()` and the number of calls to `free()`. This should prefereably occur ni the

13 Monday, June 24, 2019

Linked Lists

Like inner classes in Java, C structures can have pointers to structures of the same type. This allows us to define a Linked List's node as follows:

Listing 23: Linked List Node

```c
typedef struct node {
    int data;
    struct node *next;
} Node;
```

Note that the structure tag here is necessary here since we have a self-reference inside our definition of a node.

In order to represent a Linked List, we declare a pointer to the head by typing something like `Node *head`. The pointer allows us to modify the Linked List inside of various functions.

There are two noteworthy types of Linked Lists traversal:

- The "print traversal," which works by moving a current pointer forward after some processing

Listing 24: Print Traversal

```c
/* The print traversal */
Node curr = head;
while (curr != NULL) {
    /* Processing */
    curr = curr->next;
}
```

- The "Tom and Jerry" traversal, which works with two adjacent pointers. The left-most pointer allows us to look back and access previous elements

Listing 25: Tom and Jerry Traversal

```c
/* The Tom and Jerry Traversal */
prev = NULL;
curr = head;
while (curr != NULL) {
    /* Processing */
    prev = curr;
    curr = curr->next;
}
```

14 Thursday, June 27, 2019

Memcpy and Memset

The `void *memcpy(void * destination, const void * source, size_t num)` function is really similar to the `strcpy()` function. The function `memcpy()`— is used to copy a specified number of bytes from one memory to another, whereas `strcpy()` copies the contents of one string into another. Also, `strcpy()` works exclusively with strings, but `memcpy()` works with any type of data.

Another function is `void * memset (void * ptr, int value, size_t num)`, which sets the first *num* bytes to the block of memory pointed to by *ptr*.

For example, consider the following code:

Listing 26: Memset and Memcpy

```
 1 #include <stdio.h>
 2 #include <string.h>
 3
 4 #define MAX_LEN 80
 5 #define ROSTER_MAX_LEN 2
 6
 7 typedef struct student {
 8    int id;
 9    char name[MAX_LEN + 1];
10 } Student;
11
12 void print_roster(Student *roster, int length) {
13    int i;
14
15    for (i = 0; i < length; i++) {
16       printf("%d - %s\n", roster[i].id,
             roster[i].name);
17    }
18 }
19
20 int main() {
21    Student roster[ROSTER_MAX_LEN] = {{10, "Kelly"},
22                                      {20, "John"}};
23    Student copy[ROSTER_MAX_LEN];
24    char name[MAX_LEN + 1];
25
26    print_roster(roster, ROSTER_MAX_LEN);
27    memcpy(copy, roster, ROSTER_MAX_LEN *
          sizeof(Student));
28    print_roster(copy, ROSTER_MAX_LEN);
29
30    /* memset */
31    memset(name, 'a', MAX_LEN / 2);
32    name[MAX_LEN / 2] = '\0';
33    printf("%s\n", name);
34
35    return 0;
36 }
```

Upon running the code, Line 27 copies the contents of `roster` to the destination `copy`. Hence, the print statements on Lines 26 and 28 print out the exact same thing. Also, the `memset` cal on Line 31 sets the first half of the elements of `name` equal to `'a'`. Consequently, the print statement on Line 33 prints fourty `a`'s.

Representing Characters

The most common formats of representing characters are listed below:

1. **ASCII** is the most commonly used format; the capital letters are assigned numbers from $65 - 90$, and the lowercase letters are assigned letters from $97 - 122$.

2. **Unicode** is another common format. It stands for Unicode Transformation Format, and there are a few different versions. UTF-32 allows us to represent any character in any language (used by the Government), UTF-16 is the more popular, and UTF-8 provides backwards compatibility with ASCII (popular on the web).