

CMSC 216

Introduction to Computer Systems



Ekesh Kumar

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

<http://www.cs.umd.edu/class/summer2019/cmssc216/>

Last Revision: July 13, 2019

Contents

1	Tuesday, May 28, 2019	5
	Logistics	5
	Basic Unix Commands	5
	Introduction to C Programming	5
2	Wednesday, May 29, 2019	7
	More Unix Commands	7
	Compilation Stages of a C Program	7
	Variables in C	8
3	Friday, May 31, 2019	9
	printf() and scanf()	9
	Control Statements	10
	Functions	10
4	Monday, June 3, 2019	11
	The sizeof Operator	11
	Introduction to Pointers	11
	Pointers as Parameters	12
5	Tuesday, June 4, 2019	14
	Identifier Scopes	14

6	Wednesday, June 5, 2019	15
	Invalid Uses of Pointers	15
	Null Pointers	16
	Introduction to Arrays	16
	Arrays as Parameters	16
7	Monday, June 10, 2019	17
	Pointing to a Local Variable	17
	String Comparison	17
	Copying Strings	18
	String Literals	19
	Void Pointers	19
	Pointers to Pointers	20
8	Wednesday, June 12, 2019	21
	Command Line Parameters	21
	Two-Dimensional Arrays	22
	Two-Dimensional Character Arrays	22
	The typedef Keyword	22
	An Exception to Typedef	22
	Structures	23
	Combining Typedefs with Structs	24
	Pointers to Structures	24
9	Friday, June 14, 2019	25
	Size of Structures	25
	Unions	25
	More on Structures	25
10	Monday, June 17, 2019	27
	Exit Codes	27
	Text and Binary Streams	27
	Standard Input/Output	28
11	Wednesday, June 19, 2019	30
	The scanf() Family	30
	The printf() Family	30
	Dynamic Memory Allocation	30

12 Friday, June 21, 2019	33
Recap of Dynamic Memory Allocation	33
Dynamically Allocated Structures	33
Pointer Aliases	34
Common Errors	35
13 Monday, June 24, 2019	36
Linked Lists	36
14 Wednesday, June 26, 2019	38
Operating on Memory Blocks	38
Function Pointers	39
15 Thursday, June 27, 2019	40
Memcpy and Memset	40
Searching Files with Grep	41
Data Representation	41
Character Representation	41
Integers	41
Floats and Doubles	42
Imprecision with Real Numbers	42
16 Friday, June 28, 2019	43
Review of Numeric Base Systems	43
Bitwise Operations	44
Bitmasking	45
Big Endian and Little Endian	45
17 Monday, July 1, 2019	46
Unix File Permissions	46
Introduction to Assembly Language	46
An Illustrative Example	47
18 Tuesday, July 2, 2019	49
Data Space Instructions	49
Instructions List	50
Caller/Callee Saving	50
Arguments and Return Values	51
Accessing Memory	52

19 Wednesday, July 3, 2019	54
More on Register Pointers	54
Instruction Encoding and the Status Register	56
Branch Instructions	56
20 Monday, July 8, 2019	59

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: cmisc216/sprcoredump.
4. Office hours are immediately after class in IRB 2210.
5. Everybody will get an Arduino to 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

```
1 #include <stdio.h>
2 int main() {
3     printf("Fear the turtle\n");
4     return 0;
5 }
```

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/callee 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:

$$\begin{aligned}\text{sizeof}(\text{short}) &\leq \text{sizeof}(\text{int}) \leq \text{sizeof}(\text{long}) \\ \text{sizeof}(\text{float}) &\leq \text{sizeof}(\text{double}) \leq \text{sizeof}(\text{long double})\end{aligned}$$

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

```
1 #include <stdio.h>
2 int main() {
3     if (100) {
4         printf("Fear the turtle\n");
5     }
6     return 0;
7 }
```

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

```
1 #include <stdio.h>
2 int main() {
3     int i = 10;
4     printf("i = %d\n", i);
5     return 0;
6 }
```

`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

```
1 #include <stdio.h>
2 int main() {
3     int i, j;
4     float x, y;
5     scanf("%d%d%f%f", &i, &j, &x, &y);
6 }
```

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 the same as Java. Some other things that are similar/different from Java include:

- C supports recursion.
- C does **not** support 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

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

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

```
1 #include <stdio.h>
2 int main() {
3     int y = 5;
4     int *p;
5     p = &y;
6 }
```

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

```
1 #include <stdio.h>
2 int main() {
3     int age, values_read;
4     int *age_ptr = &age;
5     printf("Enter your age and salary");
6     scanf("%d %f", age_ptr, &salary);
7 }
```

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

```
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

```
1 void wrong_swap(int a, int b) {
2     int temp = a;
3     a = b;
4     b = temp;
5 }
6
7 int main() {
8     int x = 2, y = 3;
9     wrong_swap(x, y);
10 }
```

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

```
1 void swap(int *a, int *b) {  
2     int temp = *a;  
3     *a = *b;  
4     *b = temp;  
5 }  
6  
7 int main() {  
8     int x = 2, y = 3;  
9     int p = &x;  
10    int q = &y;  
11    swap(p, q);  
12 }
```

This is the same idea, but why does it work? Because we can dereference the pointer. We're not 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 at the cost of running slower.

6 Wednesday, June 5, 2019

Invalid Uses of Pointers

Consider the following code segment:

Listing 11: Incorrect Pointer Usage

```
1 #include <stdio.h>
2 int main() {
3     int *p;
4     *p = 200; /* This is wrong! */
5     printf("The value is %d\n", *p);
6     return 0;
7 }
```

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

```
1 #include <stdio.h>
2 int main() {
3     int *p;
4     int x;
5     p = &x; /* This is correct! */
6     *p = 200;
7     printf("The value is %d\n", *p);
8     return 0;
9 }
```

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

```
1 #include <stdio.h>
2 int* process() {
3     int x = 10;
4     int *p = &x;
5     return p; /* This is bad - we're returning a
6               pointer to some area that no longer exists! */
7 }
```

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 dereferencing 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

```
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

```
1 int strcmp(const char *s1, const char *s2) {  
2     int i;  
3     for (i = 0; s1[i] && s2[i]; i++) {  
4         if (s1[i] != s2[i]) {  
5             break; /* Gets us out of the for loop */  
6         }  
7     }  
8     return s1[i] - s2[i];  
9 }
```

- 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()` function, 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

```
1 int strcpy(const *dest, const char *src) {  
2     int i = 0;  
3 }
```

```
4     while (src[i]) {  
5         dest[i] = src[i];  
6         i++;  
7     }  
8     dest[i] = '\0';  
9 }
```

- 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 pointers 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

```
1 int one_d[5] = {12, 19, 25, 34, 46}, i;  
2 void *vp = one_d;  
3  
4 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, 2019

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

```
1 struct pixel {  
2     int x, y;  
3     char color;  
4 };
```

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 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

```
1 struct pixel {  
2     int x, y;  
3     char color;  
4 };
```

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

```
1 typedef struct pixel {  
2     int x, y;  
3     char color;  
4 } 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 append a forward slash to directory names and append an asterisk 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.

The declaration of a union is exactly the same as a structure (just replace every instance of `struct` with `union`), so I won't include any code examples of them.

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

```
1 struct {  
2     int id_number;  
3     char last_name[10];  
4     char first_name[10];  
5     double salary;  
6 } 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

```
1 #include <stdlib.h>
2
3 #define MAXLEN 80
4
5 int main() {
6     FILE *input; /* does not need to be named input */
7     char line[MAXLEN + 1], filename[MAXLEN + 1];
8
9     printf("Input file name (e.g., data.txt): ");
10    scanf("%s", filename);
11    if ((input = fopen(filename, "r")) == NULL) {
12        perror("error opening file");
13        exit(EXIT_FAILURE);
14    } else {
15        while (fgets(line, MAXLEN + 1, input) != NULL) {
16            printf("%s", line);
17        }
18        fclose(input);
19        exit(EXIT_SUCCESS);
20    }
21 }
```

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 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 `stderr`, 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 be open. We can manually enter the end-of-file character with our keyboard by entering CTRL + D. Also, EOF is a preprocessor directive, so we can use that in our conditionals.

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

11 Wednesday, June 19, 2019

In addition, it's important to remember that `fgets()` returns `NULL` after we've reached the end of file (i.e. we shouldn't be using `EOF` with it. On the other hand, it's fine to use `EOF` with `scanf` statements.

The `scanf()` Family

There are three functions in the `scanf()` family: `scanf()`, `fscanf()`, and `sscanf()`.

First, `fscanf()` has the header `int fscanf (FILE * stream, const char * format, [address of variables])`. Pretty much, we take in a file pointer along with some format, and we read it into some variables. When processing files, we want to keep reading until we hit `EOF`.

The loop conditional `while(fscanf(input_stream, "%s%d", students_name, &id) != EOF)` allows us to process the lines in a file one-by-one until we hit the end of the file. Note, however, that there's an assumption that the lines of the file are formatted in the same way. It's a good idea to use `fscanf()` when the lines are inputted in a uniform manner.

The `sscanf()` function has the header `int sscanf (const char * s, const char * format, ...)`, which returns the number of variables successfully read from the input string `s`. This can be helpful when we've already stored the string (e.g. a line) to be processed.

Sometimes, it's really helpful to combine the use of `fgets()` and `sscanf()`. The former allows us to store the entire line, and the latter allows us to make sure that everything is formatted properly (by using its return value).

The `printf()` Family

- `printf()`, we've already seen.
- `fprintf()` has the header `int fprintf (FILE * stream, const char * format, ...);`. It takes a stream, and it's analogous to `fscanf()`.
- `int sprintf(char *str, const char *format, ...)` prints into the string variable `str`; it's the analogue of `sscanf()`.

Dynamic Memory Allocation

Dynamic memory allocation allows us to allocate storage space while the program is running. Once we're done using this allocated memory, it's important to call the `free()` function to make that space available again. There are a few other important functions that help us with dynamic memory allocation, the first of which is `malloc()` (which is short for memory allocation).

The `malloc` function has the header `void* malloc(size_t size)`. The function takes in a size parameter, specifying how much space to allocate. It returns a void pointer pointing to where that space begins. The function returns `NULL` if the memory allocation fails.

As an example, consider the following code segment:

Listing 23: Malloc Example 1

```
1 #include <stdlib.h> /* For malloc, EXIT_FAILURE,  
   EXIT_SUCCESS */  
2  
3 int main(){  
4     int *ip, i, array_length = 3;  
5  
6     /* Allocating space for an integer */  
7     ip = malloc(sizeof(int)); /* notice casting is not  
   necessary */  
8     if (ip == NULL) {  
9         exit(EXIT_FAILURE);  
10    }  
11    *ip = 104;  
12    printf("Value assigned is %d\n", *ip);  
13    free(ip); /* deallocating memory */  
14 }
```

We start with an integer pointer `ip`, and we make it point to some space of memory using `malloc`. Note how we've specified that this memory has enough space to store one integer. Thus, we can dereference the pointer and assign it to an integer, and everything works fine. Further, observe that there is no need to cast the void pointer to an integer pointer.

Once we've called `free()` on a dynamically allocated memory address, it's important that we don't access that memory location again. When a memory location has been freed, any pointers that used to point to it become **dangling pointers**, which shouldn't be used. Doing so could lead to a segmentation fault, so it can be helpful to set the previously used pointer equal to null.

So, what happens internally when we call `free()`? Essentially, the heap manager marks the bytes in the memory specified as available for use. We don't actually care about what the `free()` returns – to us, it just means that the memory is free again. When we don't free the memory we've used, it's called a **memory leak**, which is bad.

We can also allocate memory for an entire array using the following code segment:

Listing 24: Malloc Example 2

```
1 int main() {  
2     /* Allocating space for array */  
3     int *ip = malloc(sizeof(int) * array_length);  
4     int array_length = 3;  
5     if (ip == NULL) {  
6         exit(EXIT_FAILURE);  
7     }  
8     for (i = 0; i < array_length; i++){  
9         ip[i] = i * 3;  
10    }  
11    for (i = 0; i < array_length; i++){  
12        printf("%d ", ip[i]); /* notice using array  
   indexing */  
13    }  
14    printf("\n");  
15    free(ip); /* deallocating memory */  
16 }
```

There aren't really any new concepts in this code segment. It should be carefully noted, however, that we check if the pointer returned from `malloc` and `calloc` is null after each call. The output of the program is 0 3

6, and the memory allocated for the array is freed after this is printed. It's important to call `free()` on a pointer that points to the *start* of the array that we've allocated, otherwise freeing won't work.

When we assign `malloc()` to a pointer, the value at the assigned memory location is garbage.

There's an alternative way to allocate memory: with the `calloc()` function. Unlike the `malloc()` function, `calloc()` takes in two parameters: its header is `void *calloc(size_t count, size_t obj_size)`, and it allocates `count` objects of size `obj_size` each. The function returns a pointer to the beginning of the memory address created. Also unlike `malloc`, the `calloc` function initializes all spaces to zero, which can save time depending on what we're doing.

Having introduced `calloc`, there are a lot of shortcuts we can take. For example, consider the statement `int **q = calloc(4, sizeof(int *))`, which allocates space for an array of four integer pointers. Also, since `calloc` automatically initializes its spaces to zero, all of these pointers are automatically set to null for us.

12 Friday, June 21, 2019

Recap of Dynamic Memory Allocation

Today, we will continue discussing dynamically allocated memory (i.e. memory that isn't allocated until the program starts running). What's another reason we 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.

To recap, there are two memory management 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. Something to take note of is that `free()` has to be the same pointer that was returned from `malloc()` or `calloc()` – we can't call `free()` in the middle of the area that we allocated. Also, after the pointer is freed, the pointer becomes a dangling pointer, so we shouldn't dereference it.

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()`. It's also good to know that calling `free()` on null is harmless – you don't need any null checks for calling `free()`. Doing `free(NULL)` is completely harmless. Also, as one would expect, we can't free pointers whose data is constant (i.e. we can't free a pointer declared as `const char *p`).

We should only call `free()` on a pointer once. Why? When we call `malloc` or `calloc`, we're telling our computer that we want to reserve some memory just for that pointer. When we subsequently call `free`, we're telling the computer that we don't need that space anymore; however, the pointer still points to that memory address. If we invoke `free()` a second time, we're not freeing the previous data, but possibly some new data that resides at that memory address.

Dynamically Allocated Structures

Consider the following lecture example:

Listing 25: Dynamically Allocated Structure

```
1
2 #include <stdio.h>
3 #include <stdlib.h> /* For malloc, EXIT_FAILURE,
   EXIT_SUCCESS */
4
5 /* Notice tag and typedef identifier can be the same */
6 typedef struct Student {
7     char *name;
8     int age;
9 } Student;
```

```
10
11 int main(){
12     Student *student;
13     int length;
14
15     /* Allocating space for a Student structure */
16     student = malloc(sizeof(Student));
17     if (student == NULL) { exit(EXIT_FAILURE); }
18
19     /* Allocating space for name */
20     printf("Enter number of characters in your name: ");
21     scanf("%d", &length);
22     student->name = malloc(length + 1);
23
24     /* Reading name and age */
25     printf("Enter your name: ");
26     scanf("%s", student->name);
27     printf("Enter your age: ");
28     scanf("%d", &student->age);
29
30     /* Feedback */
31     printf("Your name is %s and your age is %d\n",
32           student->name, student->age);
33
34     /* Freeing memory */
35     /* We must free name first */
36     free(student->name);
37     free(student);
38
39     return EXIT_SUCCESS;
40 }
```

On Line 16, we dynamically allocate space for `student` to become a `Student` type. Immediately after, we check if this allocation was successful (i.e. check if `student == NULL` holds), and we continue if it was. Next, we allocate space for `name`, depending on how many characters the user requires (which is provided through `stdin`). Finally, we store the student's age (also from `stdin`), we print the student's information, and we free the space we allocated.

Some key things to note:

- Why did we allocate space for `student` and `name` but not for `age`? Because `age` isn't a pointer, so we already get space for `age` after we dynamically allocate space for `student`.
- Why do we free `name` before `student`? If we freed `student` first, then `student` becomes a dangling pointer. So, we shouldn't be accessing `student->name` afterwards (accessing the name depends on the existence of a student, but the student's existence doesn't depend on its name).

Pointer Aliases

We can have two pointers point to the same dynamically allocated memory area. For instance, consider the following code segment:

Listing 26: Pointer Aliases and Dynamic Memory Allocation

```
1 int *p, *q; /* Declare two integer pointers */
2 p = malloc(sizeof(int)); /* Dynamically allocate
   memory */
3 if (p != NULL) { /* Make sure allocation succeeded. */
```

```
4      *p = 99; /* Dereference p; set its entity to 99. */
5      q = p; /* q points to the same memory address as p
6             */
7      free(p) /* BOTH p and q are dangling pointers now
8             */
9      *q = 42; /* This is WRONG. */
10 }
```

In summary, if we free one pointer pointing to a memory address, *every* pointer pointing to that same memory address becomes a dangling pointer.

Common Errors

The common errors of dynamically allocating memory comes are summarized below:

1. Dereferencing pointers to freed space. We've already discussed this.
2. Forgetting to check if malloc or calloc returned null (i.e. the dynamic memory allocation was unsuccessful).
3. Forgetting to initialize the memory malloc() returns (calloc() automatically does this for us).
4. Attempting to free non-heap memory (i.e. we should *only* be calling free() on dynamically allocated memory).
5. Memory leaks: forgetting to call free() on dynamically allocated memory.
6. Calling free() twice on a pointer.

Using valgrind can help identify these errors.

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 27: Linked List Node

```
1 typedef struct node {  
2     int data;  
3     struct node *next;  
4 } Node;
```

Note that the structure tag inside of the typedef 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. So, functions that modify the actual Linked List have function prototypes that take in a double pointer `Node` type.

For instance, a function which instantiates the Linked List might have header `void create_list(Node **head)`, and its body would simply be `*head = NULL`.

There are two noteworthy types of Linked Lists traversal:

- The “**print traversal**,” which works by moving a current pointer forward after some processing:

Listing 28: Print Traversal

```
1 /* The print traversal */  
2 Node curr = head;  
3 while (curr != NULL) {  
4     /* Processing */  
5     curr = curr->next;  
6 }
```

If we need to visit every node in the list and do some processing, we perform the print traversal.

- The “**Tom and Jerry traversal**”, which works with two adjacent pointers. The left-most pointer allows us to look back and access previous elements. The two pointers move in parallel.

Listing 29: Tom and Jerry Traversal

```
1 /* The Tom and Jerry Traversal */  
2 prev = NULL;  
3 curr = head;  
4 while (curr != NULL) {  
5     /* Processing */  
6     prev = curr;  
7     curr = curr->next;  
8 }
```

This traversal allows us to add a node before or after any element.

When we construct a Linked List, we will want to add nodes one by one. Creating a node requires three key steps:

1. Allocate memory for the node.
2. Store data in the node.
3. Insert the node into the list.

When we create a node, we write something like `struct node *new_node` followed by the command `new_node = malloc(sizeof(struct node))`. Note that something like `new_node = malloc(sizeof(new_node))` would be incorrect as `new_node` is a pointer variable (this command would allocate 8 bytes rather than the actual amount of space needed).

To change the data of a node, we can use the arrow operator by doing something like `new_node->data = 5`.

14 Wednesday, June 26, 2019

Midterm 1 will be graded by sometime next week.

Operating on Memory Blocks

Like `strcpy()` operates on strings, there is an analogous function that operates on entire blocks of memory. Namely, this function is `memcpy`, which has the following declaration: `void *memcpy(void *dst, void *src, size_t n)`. It works pretty much the same with `strncpy()` works: it takes a destination, source, and the number of bytes to copy. A prerequisite to using `memcpy()` is that `dst` and `src` cannot be overlapping pieces of memory (for instance, they shouldn't be pointing to different portions of the same array).

So, what do we do if we need to quickly copy overlapping memory areas? We can use the `memmove()` function, which has the declaration `void *memmove(void *dst, void *src, size_t n)`, but it's not as efficient.

A possible implementation of `memcpy()` is presented below:

Listing 30: Memset and Malloc

```
1 void print_decimal(unsigned int i) {
2     printf("%u\n", i);
3 }
4
5 void print_hex(unsigned int i) {
6     printf("%x\n", i);
7 }
8
9 void print_octal(unsigned int i) {
10    printf("%o\n", i);
11 }
12
13 int main() {
14     /* The following is the declaration of */
15     /* fp as a function pointer variable */
16     void (*fp)(unsigned int);
17
18     /* Below both & and * are optional */
19     /* due to the use of the function */
20     /* call operator */
21
22     fp = print_hex;
23     fp(16); /* prints "10" */
24     fp = &print_octal;
25     fp(16); /* prints "20" */
26     fp = print_decimal;
27     (*fp)(16); /* prints "16" */
28
29     return 0;
30 }
```

Note how the function utilizes void pointers, which means that we can pass any type of pointer into the function. We cast to `char *` so that we can iterate byte-by-byte.

The function `memcpy()` is really helpful. So far, to make copies of arrays of structures, we've been iterating through every index with a for loop. But now, we can do all of this with a single statement. Nelson says that this is important to know for exams.

Function Pointers

In the same way that we can have pointers to variables, we can also have pointers to functions. **Function pointers** store the memory address of a function, which is possible as each function is located somewhere in memory. Similar to an array, the name of a function can be seen as a variable that stores the function's address.

We can use function pointers in pretty much the same way that we use normal pointers.

A procedure to write the declaration of a function pointer is to first just write the function prototype, add parentheses around the function name ("hug the function"), and add an asterisk to the start of the newly added parentheses ("kiss the function"). The following example illustrates the basics:

Listing 31: Introduction to Function Pointers

```
1 void print_decimal(unsigned int i) {
2     printf("%u\n", i);
3 }
4
5 void print_hex(unsigned int i) {
6     printf("%x\n", i);
7 }
8
9 void print_octal(unsigned int i) {
10    printf("%o\n", i);
11 }
12
13 int main() {
14     /* The following is the declaration of */
15     /* fp as a function pointer variable */
16     void (*fp)(unsigned int);
17
18     /* Below both & and * are optional */
19     /* due to the use of the function */
20     /* call operator */
21
22     fp = print_hex;
23     fp(16); /* prints "10" */
24     fp = &print_octal;
25     fp(16); /* prints "20" */
26     fp = print_decimal;
27     (*fp)(16); /* prints "16" */
28
29     return 0;
30 }
```

On Line 16, we declare a function pointer `fp` using the exact procedure previously described. The key thing to note there is that the parentheses around `*fp` indicates that `fp` is a pointer to a function, not a function that returns a pointer.

Once we've got a function pointer, we can assign it to the name of a function (just as we could do with arrays), and everything works fine. In fact, the C compiler even allows us to assign a function pointer to the *address* of a function (like on Line 24), or even dereference while calling the function (like Line 27), and everything will still work fine. However, the statements on Line 22 and 23 illustrate standard way to do this.

15 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, whereas `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 in the block of memory pointed to by `ptr`, and sets them all to `value`. For instance, if you have a character array `arr`, the statement `memset(arr, 'a', 8)` would set `arr` to have eight characters, each of which are `a`.

A more comprehensive example is provided below:

Listing 32: Memset and Malloc

```
1 #include <stdio.h>
2 #include <string.h>
3
4 #define MAXLEN 80
5 #define ROSTER_MAXLEN 2
6
7 typedef struct student {
8     int id;
9     char name[MAXLEN + 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,
17             roster[i].name);
18     }
19 }
20
21 int main() {
22     Student roster[ROSTER_MAXLEN] = {{10, "Kelly"},
23         {20, "John"}};
24     Student copy[ROSTER_MAXLEN];
25     char name[MAXLEN + 1];
26
27     print_roster(roster, ROSTER_MAXLEN);
28     memcpy(copy, roster, ROSTER_MAXLEN *
29         sizeof(Student));
30     print_roster(copy, ROSTER_MAXLEN);
31
32     /* memset */
33     memset(name, 'a', MAXLEN / 2);
34     name[MAXLEN / 2] = '\0';
35     printf("%s\n", name);
36
37     return 0;
38 }
```

Upon running the code, Line 27 copies the contents of `roster` to the destination `copy`. It's important to note that this isn't the same as making the two array pointers point to the same block of memory. Using `memcpy()`, we've created two distinct blocks of memory with the same contents. Hence, the print statements on Lines 26 and 28 print out the exact same thing. Subsequently, the `memset` call on Line 31 sets the first half of the elements of `name` equal to `'a'`. Consequently, the print statement on Line 33 prints forty `a`'s.

We cannot copy overlapping memory areas when using `memcpy()` or `memset()`.

Searching Files with Grep

`grep` is a command-line utility that can be used in Unix to search for patterns of characters in a file. The general format for `grep` is `grep [target_string] [file_name]`. Consider the following text file, called `information.txt`:

Listing 33: Text Sample

```
1 This project is about hashing ,  
2 files , structures ,  
3 pointers  
4 and dynamic memory allocation (and more pointers).
```

If we were to type `grep point information.txt`, the shell would return the lines in which the string `point` is found (namely, the third and fourth lines). A useful flag that we can add is the `-n` flag, which returns the line numbers alongside the lines that are found.

Now, what if we want to search multiple files? Like other Unix commands, we can use the `*` wildcard. For instance, `grep -n point *` would print the lines and line numbers of every file in the current directory that contains the string `point`. To search recursively (in subdirectories), we need the `-r` flag.

Data Representation

Character Representation

The two 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 most popular, and UTF-8 provides backwards compatibility with ASCII.

Integers

When we're representing unsigned integers, the representation is more straightforward - all of the numbers are stored using binary. Now, this works pretty easily for positive integers, but what if we want to represent a negative number? It turns out that the solution comes using a convention called **two's complement**.

Under two's complement, the positive value of a number is just its binary representation with its leftmost bit equal to 0. To obtain a negative value, we invert all of the bits of the corresponding positive value, and we add 1. The eight bits `0000101` typically correspond to the decimal number `+5`. Under the two's complement convention, the number `-5` can be represented by `11111011`.

Why do we use two's complement instead of, say, just adding an extra bit at the start or end to indicate the sign? The reason why we use two's complement over just having an additional sign-indicating bit is mostly for math-simplifying reasons¹.

With two's complement, The range of values that we can store with n bits ranges from -2^{n-1} to $2^{n-1} - 1$, inclusive.

¹<https://stackoverflow.com/questions/1125304/why-prefer-twos-complement-over-sign-and-magnitude-for-signed-numbers>

Floats and Doubles

We can store floats and doubles by writing the value we're storing in the form

$$(-1)^s \cdot m \cdot r^e, \tag{1}$$

where s is a bit (either 0 if we're representing a positive number or 1 if we're representing a negative number) representing the sign of the number, m is a **mantissa**, r is the **radix** (base) we're working in, and e is an exponent to be determined.

This might seem confusing at first, but we can break this process down into one simple step: writing the number in scientific notation.

All of the variables in Equation (1) comes from writing our number in scientific notation. For example, if we want to store the decimal number 51.432, we can write it as $5.1432 \cdot 10^1$, and we can examine this expression to see that $m = 5.1432$, $r = 10$, $e = 1$, and $s = 0$.

Similarly, if we're storing the binary number 1001.1101, we would be able to write this as $1.0011101 \cdot 2^3$ and find $m = 1001.1101$, $r = 2$, $e = 3$, and $s = 0$.

The number of bits allocated for the radix, exponent, and mantissa are specified by the IEEE 754 floating point standard. To store a negative exponent, we add an **exponent bias** to the exponent e , which normalizes the number zero to all zeroes.

Imprecision with Real Numbers

Some real numbers, like $1/3 \approx 0.33333$, have infinitely long decimal representations. This can create complications when we're storing these numbers because we are trying to store an infinite decimal with finite space. In summary, some of these bits get cutoff, which can cause some small imprecisions when dealing with real numbers.

16 Friday, June 28, 2019

Today, we're going to discuss bits and bitwise operations.

Recall from last lecture that the negative representation of an integer can be obtained by inverting all of the bits and adding 1. As an example, consider the number `01110`, which corresponds to the decimal number 15. By inverting the bits and adding one, we see the number -15 can be represented as `10010`, and this can easily be verified for correctness by noting that $01110 + 10010 = 00000$.

Review of Numeric Base Systems

We should already be familiar with how to convert between two different bases. However, we can briefly recap:

- To convert from base 10 (what we are accustomed to) to binary, repeatedly divide the number by 2, and keep track of the remainders. The binary representation of the number will be the remainders concatenated together in reverse-chronological order.
- To convert from binary to base 10, multiply the i^{th} bit from the right (starting from 0) by 2^i . The desired number in base 10 will be the sum of these powers.

Consider the following conversion table:

decimal	hexadecimal	binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

From the table above, it becomes clear that a single hexadecimal digit requires four bits for its representation. This matters because it allows us to quickly convert between hexadecimal and binary. Consider, for example, the hexadecimal number 55. We can look at each hexadecimal digit, and we can write down the corresponding binary number with four bits to obtain 01010101 as the equivalent binary representation.

The `printf()` command has a format specifier to print a number in hexadecimal (with the `%X` specifier) or octal (with the `%o` specifier). Moreover, the `%08x` specifier can be used for printing unsigned integers as zero-padded, eight-digit hexadecimal numbers. However, there is no format specifier to print a number in binary. As an example, consider the following code segment:

Listing 34: Hexadecimal Format Specifier

```
1 #include <stdio.h>
2
3 int main() {
4     unsigned int k = 0, limit = 32;
5
6     for (k = 0; k <= limit; k++) {
7         printf("%d —> %08x\n", k, k);
8     }
9
10    return 0;
11 }
```

The output of this program will be all hexadecimal numbers between 0 and 32, inclusive. There are additional zeroes added to the start of each number to ensure that each output is eight digits long.

So, why do we specify that we want eight bytes, rather than some other number? In our system, an unsigned integer is 4 bytes, or equivalently, 32 bits (there are 8 bits in a byte). Now since each hexadecimal number requires four bits, we require $32/4 = 8$ hexadecimal digits.

As another example, suppose we're printing out a single character. There is 1 byte, or equivalently, 8 bits in a character. Hence, we require $8/4 = 2$ hexadecimal digits to print a character.

We can utilize the preceding fact and store a two-digit hexadecimal number as an unsigned character type.

Bitwise Operations

First, we discuss bit shifting.

A bit shift moves each digit in a number's binary representation a specified number of places to the left or right. The left shift operator is written as `<<`, whereas the right shift operator is written as `>>`. The number that directly follows this operator specifies the number of places we wish to shift.

As a basic example, consider the number 2, which has binary representation `0011`. If the variable x were storing the value 2, the statement `x << 1` would return the value 4 since the binary representation of `x << 1` is `0110` (note that an additional zero has been added at the end). The variable x itself won't be updated from a bitshift. If we want to update the value of x , we would have to instead write either `x = x << 1` or `x <<= 1`. Likewise, the statement `x >> 1` would return the value 1. Performing left and right shifts are equivalent to multiplying or dividing by powers of 2. That is, a left shift by one multiplies by 2, and the right shift divides by 2 (note, however, that they are much faster than multiplying or dividing by 2).

Some other bitwise operators include the logical operators of AND, OR, NOT, and XOR. These can be used in C with the binary operators `&`, `|`, `~`, and `^`, respectively. We should already be familiar with these, but to recap, consider the numbers 5 and 7. The binary representation of these two numbers are `0101` and `0111`. Consequently, we see that `5 & 7` returns 5, `5 | 7` returns 7, `~5` returns 10, and `5 ^ 7` returns 2.

The identity for the AND operator is 0, and the identity for the OR operator is 1 (i.e. `x & 0 = x`, and `x | 1 = x`)

Note that all of these bitwise operations can also be applied to hexadecimal numbers (or rather, numbers in any base). It's just important to convert everything to binary first.

Bitmasking

Why are bitwise operations important? We can use bitwise operations to perform **bitmasking**, ultimately allowing us to “toggle” or “retrieve” bits on and off.

Consider a function that takes in an integer; say the input is `1010`. If we want to always clear the second bit from the right, this function can simply return the user’s input AND’d with `1101`, which will always result in a `0` in the second bit. As another example, suppose we want to extract the second bit from the right of a given integer. Again, suppose the integer we’re working with is `1010`. We can AND this number with `0010`, which will clear everything except for our digit of interest. If the result is `0`, the value of that digit is `0`; otherwise, it is `1`.

Using AND in conjunction with right shifts, we can write a program that counts the number of zeroes and ones in the bitwise representation of an integer.

In summary,

- We can clear bits by AND’ing with `0`.
- We can check bits by AND’ing with `1`.
- We can set bits by OR’ing with `0`
- We can flip bits by XOR’ing with `1`.

Consider the following problem that can be solved with bitmasking: Given the hexadecimal number `0xab` (which has binary representation `1010 1011`), perform bitwise operations to make bits in positions 2 to 4 (from the left) equal to `110`.

The solution is to AND with `1000 1111` to clear the targeted bit positions, and finally OR with `0110 0000` to set them to what we desire.

Big Endian and Little Endian

Endianness refers to how data is arranged in memory. It is dependent on the hardware we are working on (not the operating system). Endianness is important when we’re working at the bit level.

There are two types of endianness that we are concerned with:

- In **Big Endian** refers to how we typically visualize data. In this representation, there’s a pointer that points to the most significant byte.
- In **Little Endian**, the data is laid out in reverse-chronological order, and the pointer points to the least-significant byte.

Grace runs using little Endian.

17 Monday, July 1, 2019

A **nibble** is a term used to describe four bits. A very important thing to remember from last lecture is that each hexadecimal digit is represented by a nibble.

Unix File Permissions

Unix file permissions are based on the octal system. Every file is associated with three entities: the user entity, the group entity, and the “other” entity. The user entity describes the file permission of yourself. The group entity describes a set of users on the same system. The “other” section describes everyone else.

The `chmod` command can be used in Unix to change permissions of a file. The general format of the command is `chmod [settings] [file_name]`. What we enter for the `[settings]` portion of the command is a three-digit octal number, specifying the permissions for each entity. The first digit corresponds to the specifications for the user, the second digit corresponds to the settings of the group, and the last digit is for the “other” entity.

So, how does it work? We convert each octal digit in `[settings]` to a three-digit binary number, which specifies whether the entity has read, write, and/or executing permissions, in that order.

As an example, suppose we execute `chmod 400 a.out`. The corresponding binary representation of 4 is 100, meaning that the user (the person executing the command) will be able to read the file, but they will not be able to write to the file or execute it. Since the other two octal digits are 0, the permissions of the group and “other” categories aren’t modified.

How do we determine what the `[settings]` number should be? This is easy - just work backwards. Suppose we want the user and the group to be able to read and execute the file but not write to it. This corresponds to the three-digit binary number 101, which has octal representation 5. So, `chmod 550 a.out` does exactly what we want.

We can call `chmod` recursively on a directory using the `-r` flag.

Introduction to Assembly Language

Assembly is a low-level, readable translation of machine language. It is a programming language that we work with when we want to see what the computer is doing. There are many assembly languages out there - in this class, we will use AVR Assembly. We can generate the `.S` file corresponding to the assembly instructions of a `.c` file by compiling with the `-S` flag when using `avr-gcc`. This is not allowed for projects/exercises.

A computer consists of some memory (RAM), and a CPU. Inside of the CPU, there is an **arithmetic logic unit**, which is responsible for performing computations. Additionally, inside of the CPU, there are **registers**, which are fast-access locations that instructions use instead of storing all values in memory. Registers are all one byte. In AVR, there are 32 registers, labeled `r0`, `r1`, ..., `r31`. A computer also has a **program counter**, which is a register that specifies the next instruction to be executed. Finally, the computer has an **immediate**, which consists of constants that are in the instruction itself.

A program written in AVR assembly has four components to it. Firstly, there are **instructions**, which specify what the processor will execute. These instructions typically consist of a name, a list of registers, and sometimes a constant value. In addition, there are **labels**, which represent an address. Labels are typically denoted by some text, followed by a colon. They are also used to define functions. Finally, there are **assembler directives**, which controls where code and data are placed, as well as **comments**.

How does our assembly code become machine language? We use an **assembler** (analogous to a compiler) to produce the zeroes and ones associated with our instructions.

An Illustrative Example

To illustrate how a basic assembly program works, we'll first consider some logically equivalent code written in C:

Listing 35: C Program for Assembly Example

```
1 #include <stdio.h>
2
3 #define LETTER_A 65
4 #define NEWLINE 10
5
6 char x = 6; /* We are not using it */
7
8 int main() {
9
10     putchar(LETTER_A);
11     putchar(NEWLINE);
12
13     return 0;
14 }
```

Note that we don't actually use the variable `x` - it's just there so that we can see how to create global variables in assembly. What does this code segment do? It prints the letter A (which has ASCII value 65), and it subsequently prints a newline character (which has ASCII value 10).

The corresponding assembly program is presented below:

Listing 36: Assembly Example

```
1 ;;; Organization of typical avr program (; used for
   comments (don't use #))
2
3 ;;; Symbolic constants
4     .set LETTER_A, 65      ; Constant for A
5     .set NEWLINE, 10      ; Constant for \n
6
7 ;;; Global data
8     .data                 ; Begins data section.
9 x:     .byte 6             ; Label "x" stores 6
10
11 ;;; Program code
12     .text                 ; Directive to start code
13
14 .global main              ; Makes main label
   externally available
15 main:
16     ;; main function. Task is defined at this
   point. This function prints letter A
   followed by newline character. We need the
   newline to force the flushing of
   character A.
17
18     call init_serial_stdio ; To call a function we
   use the call instruction
19
20     ldi r24, LETTER_A      ; The ldi instruction
   loads a value into a register.
21     clr r25 ; Sets high byte of putchar's integer
   argument to 0.
22     call putchar ; Calls putchar to print the
   character
```

```
23
24     ldi r24, NEWLINE ; Initializes r24 with '\n'
                        character ascii value
25     clr r25 ; Sets high byte of putchar's integer
                        argument to 0
26     call putchar
27
28     cli                ; We need to stop the
                        program. Relying on cli and sleep.
29     sleep
30
31     ret ; Adding this ret to show functions end
                        with ret but this ret is unreachable
                        (program already stopped)
```

Before we start analyzing this code, it should first be noted that lines beginning with dots are directives, lines ending with colons are labels, lines beginning with a semicolon are comments, and everything else is an instruction.

What observations can we make?

- Comments begin with a single semicolon, but we sometimes prefer to use more semicolons for stylistic reasons.
- Symbolic constants are defined with `.set` directive followed by a target text, a comma, and a replacement text. The target text never actually makes it to the machine code; it is processed by the assembler.
- By writing the `.data` directive, we indicate that what follows is a data section. We create labels by having some text, followed by a colon. Here, our label is `x`, which stores the memory address of the value 6. What follows `.byte` indicates the entity being stored at the memory address. This can be written in decimal (as it is), hexadecimal, or even binary.
- The `.text` directive indicates that we're done with our initial setup, and everything that follows is actual code. This is a very important directive to have.
- The `.global main` directive allows `main` to be called outside of the current file. Functions, including `main`, begin with labels. Also, functions end with `ret`.
- To call a function, we use the call instruction `call init_serial_stdio`. We will always call this function to permit us to use input and output.
- To print the ASCII value of 'A,' we need to first load a value into a register using `ldi`. In our program, we move 65 to register 24.
- After we load into register 24, we clear register 25 with `clr`. Why do we clear register 25? Because `putchar()` assumes that the value it will display can be found in registers 24 and 25. This is a rule defined by gcc. So we clear register 25 to contain no data.
- To flush the buffer, we load in the new line character to register 24, and we re-clear register 25 (in case `putchar` messed it up). Finally, we call `putchar` again, and we get our desired output.
- The `cli` and `sleep` functions are necessary to stop the program.

Side-note: we can't have floating-point types in AVR Assembly, but we can have integers, characters, and strings.

18 Tuesday, July 2, 2019

No discussion today. Today is day two of Assembly.

Data Space Instructions

In order to transfer the contents of a register back into memory (like a variable), we can use the `sts` instruction, which is short for “store to data space.” Using `sts` in assembly is analogous to assigning a variable some value. On the other hand, we can use stored memory to write to a register using the `lds` instruction, which is short for “load direct from data space.” The general syntax for `sts` is `sts [data destination], [register source]`. The general syntax for `lds` is `lds [register destination], [data source]`.

Another useful instruction is `inc`, which simply increments the contents of a register. As we’d expect, the syntax for this instruction is just `inc [register name]`.

Consider the following code segment, which illustrates all three of these instructions:

Listing 37: Storing and Loading to Data Spaces

```
1  ;;; Global data
2  .data
3  a: .byte 0x2
4  b: .byte 0b00000100
5  c: .byte 0
6
7  ;;; Program code
8  .text
9
10 .global main
11 main:
12
13     call init_serial_stdio
14     lds    r18, a ; stores contents of a into r18.
15     lds    r19, b ; stores contents of b into r19.
16     push   r19 ; saves value of r19 on the stack.
17     add    r19, r18 ; adds contents of registers.
18     sts    c, r19 ; stores contents of r19 into c.
19
20     lds    r18, c ; stores contents of c into r18.
21     inc    r18 ; increments r18
22     pop    r19 ; restores r19
23     inc    r19 ; increments r19
24
25     cli    ; stops the program
```

What’s happening here?

On Lines 3, 4, and 5, we’re just declaring three global variables: `a`, `b`, and `c`. Like we saw yesterday, the `.data` directive indicates that we’re starting our data section, and the `.byte` directive indicates that the value that follows should be stored in the specified variable. In this case, we store `0x2` (hexadecimal for 2) in `a`, 100 (binary for 4) in `b`, and 0 in `c`. The reason why we’re using different base systems is just to clearly convey that it is allowed.

All of our variables have been initialized, so we can begin writing our code (officially, this is indicated by the `.text` directive on Line 8). In our `main`, we see an example of `lds` in action: the instruction takes a register and some data, and it loads the contents of the data into the register. In this case, `r18` stores 2, and `r19` stores 4. The contents of `r19` are pushed onto the stack for safekeeping.

The `add` instruction on Line 17 stores the contents of `r19` and `r18` and stores the sum in `r19` (it overwrites the previous value, which is precisely why we pushed the original value onto the stack). On Line 18, the `sts` instruction is used to store the sum into variable `c`. This value is incremented by one, the original contents of `r19` are restored, and the contents of `r19` are incremented by one. The final value stored in `r18` is 7.

Instructions List

The list below summarizes some important instructions that we should become familiar with (we've already seen some of these):

- `ldi` initializes a register with a constant value. Its general syntax is `ldi [register] [constant]`. For instance, `ldi r24, 5` would set the contents of `r24` to 5. We can only load constants to the registers `r16`, ..., `r31` (with the exception of `clr`, which loads in 0).
- `lds` loads data from memory into a register — we saw this in the previous example.
- `sts` stores data from a register into memory — we also saw this in the previous example.
- `clr` clears the contents of a register. Its general syntax is `clr [register]`. After this instruction is executed, the contents of the register it was performed on becomes 0.
- `add` is used to add the contents of two registers. Its general syntax is `add [register1], [register2]`. After this instruction is executed, the contents of `register1` are replaced with `register1 + register2`.
- `inc` is used to increment the value of a register by one. We saw this in the previous example.
- `push` is used to push a register value onto the stack. Its syntax is `push [register]`.
- `pop` is used to move data from the top of the stack into another register. Its syntax is `pop [register]`. Note that you don't have to pop to the same register that was pushed. There should be a one-to-one correspondence between `pop` and `push` instructions.
- `call` is used to call a function, as we saw with the `putchar` example yesterday.
- `ret` is used to return from a function.
- `nop` is short for “no operation,” and it does nothing. Why does it exist? To make our processor wait and do nothing.
- `mov` has syntax `mov [destination register], [source register]`. It copies the contents of `source register` into `destination register`. Note that the name “move” is slightly misleading here - the contents of `source register` aren't moving anywhere - they are being copied, not moved.

One way to remember the syntax of some of these instructions is to note that, when we're writing to a register, the first register is always the target register where the result is being written to.

Caller/Callee Saving

The 32 registers that we use in Assembly are all global registers. What this means is that these registers are shared among every function, including the main. To illustrate why this matters, suppose we were to store 20 in register `r19` in the main. We then decide to call some function, which stores 100 in the register `r19` as a part of its processing. This change will also be reflected in the main (and everywhere else in the program).

Registers are shared across the entire program, which makes things more complicated. How can we avoid overwriting registers used by functions? The solution to this problem comes from two protocols that we use:

1. **Caller-saved protocol:** When writing a function, we assume the programmer who called the function has already saved the contents of registers from `r18` to `r27`, `r30`, and `r31`. So, the function writer should be operating on only these registers, and it's up to the programmer to have already saved anything of importance in these registers. We are expected to only operate on these registers when writing our functions as well.

2. **Callee-saved protocol:** If a function writer wants to use registers `r2` to `r17`, `r28`, or `r29` inside of their function, they need to be saving the registers prior to using them. Furthermore, the contents of these registers need to be restored back to their original values before leaving the function.

What do these protocols mean to us? It means that there is less work for both the function writer and the function caller. On one hand, the function writer can use several registers without worrying about overwriting important data. On the other hand, the function caller has a set of registers where they can store data and call a function with certainty that their data will not be overridden.

To better understand these two protocols, consider a blackboard that is shared among different professors. The blackboard follows the caller-saved protocol: a professor who enters the classroom to teach is allowed to erase whatever is on the blackboard. It is assumed that any important information on the blackboard has already been recorded by the previous user. On the other hand, if the blackboard were callee-saved, the most recent user would have to restore the blackboard to however it originally was, prior to their use.

We can use the fact that registers are shared across the entire program to pass and return values to functions.

Arguments and Return Values

Unlike C, Assembly doesn't have function headers: a function declaration is just a label. So, how do we take in arguments and/or return values? Both of these tasks are accomplished through registers. If we have a lot of arguments or return values and we run out of registers, we can use the stack (but we won't need to worry about that).

When we're passing arguments to a function, we can just load the argument into a register for the function to use. Arguments are aligned to start in even-numbered registers, beginning at `r24` and going downwards. Also, if we're passing an odd number of parameters, there will always be one free register above them. So, for instance, if we're passing in a single `char` to a function, we would load the value of that character into `r24`, and we would leave `r25` empty as there is an odd number of parameters.

The conventions for passing arguments to functions are illustrated below:

- If we're passing an argument that's just one byte, the argument will go in `r24` and `r25` will be cleared.
- If we're passing an argument that's two bytes, the arguments will go in `r24` and `r25` (note how there is no free register since there are two arguments).
- If we're passing four bytes of arguments, the arguments will go in `r25`, `r24`, `r23`, `r22`, and there won't be any empty registers.

Once these registers have been loaded with the arguments, we can use the `call` instruction to go inside of the function. The function will share the contents of these registers, and it will be able to perform any necessary processing.

How do we return values from functions? It's the same idea as passing arguments. The exact same convention for passing parameters. For instance, if the function returns one byte, the return value is loaded into register `r24`.

Note that the conventions for passing arguments and returning values align with the caller/callee-saved protocol. Particularly, we are passing parameters through caller-saved registers. The function caller is expected to have already saved any important data in these registers, as the function will be changing the contents of these registers after processing.

After introducing these concepts, let's look at an example that's very similar to what we saw yesterday:

Listing 38: Assembly: Arguments and Return Values

```
1  .text
2
3  .global main
4  main:
5      call init_serial_stdio
6
7      ; Printing 'A'
8      clr r25
9      ldi r24, 65
10     call putchar
11
12     cli
13     sleep
```

The function `putchar` from C takes in a single character as an argument. To pass in the argument ‘A,’ we clear register `r25` and load the ASCII value of ‘A’ into `r24`. This aligns with the convention we’ve previously discussed. Finally, we call `putchar`, which is now free to do whatever it desires with these parameters. It will no longer be guaranteed that register `r24` and `r25` are how they were prior to the function call.

Accessing Memory

In Assembly, `lo8` and `hi8` can be used to extract the lower and higher 8 bits of data (if we are passing in 2 bytes of data, there will be two 8 bit blocks). The instruction `ldi r24, lo8(x)` would load the lower byte of variable `x` into register `r24`.

We have already seen that `lds` and `sts` are instructions that allow us to read and write to memory. Assembly has three pairs of registers that allow us to access memory. These pairs of registers are called `X` (resides in registers `r26` and `r27`), `Y` (resides in registers `r28` and `r29`), and `Z` (resides in registers `r30` and `r31`).

`X`, `Y`, and `Z` represent addresses in memory (like pointers). Conventionally, the low byte is stored in the even-numbered register, and the high byte is stored in the odd-numbered register.

Consider the following example:

Listing 39: Assembly: Arguments and Return Values

```
1  .data
2  pctd:
3      .asciz  "%d "
4
5  values:
6      .byte  15
7      .byte  16
8      .byte  17
9      .byte  18
10
11     .text
12
13     .global main
14     main:
15
16         call init_serial_stdio
17
18         lds r24, values
19         clr r25
20         call pint
21         ldi r24, 10
22         clr r25
23         call putchar
24
25         ldi r26, lo8(values)
```

```
26     ldi r27, hi8(values)
27     ld r24, X
28     push r26 ; Caller-save
29     push r27
30     clr r25
31     ldi r24, 10
32     clr r25
33     call putchar
34
35     pop r27
36     pop r26
37     adiw r26, 1 ; Move the pointer
38
39     ld r24, X
40     clr r25
41     call pint
42     ldi r24, 10
43     clr r25
44     call putchar
45
46     cli ; Terminate program
47     sleep
48     ret
```

First off, we see a new directive: `.asciz`. This just indicates that we are declaring a string literal.

Next, we note the label `values` is defined differently than what we've seen so far: there are multiple `.byte` directives in its definition. The key thing to remember here is that a label is just an address in memory. Hence, one way we can interpret `values` is the value 15: if we were to load this value somewhere, the value 15 would be loaded. We can alternatively interpret `values` as the name of an array with contents 15, 16, 17, and 18.

When we call `pint` (the function for printing), 15 is outputted. We then load the ASCII value for a new line, and we call `putchar` to print a new line.

On Lines 25 and 26, we load the low byte and high bytes of `values` into the registers `r26`, and `r27`. The registers `r26` and `r27` are special: they represent a register pointer, denoted by `X`. Hence, on the subsequent line, we can use `ld` to initialize `r24` to whatever `X` points to (note that we use `ld` for register pointers instead of `lds`). Finally, when we call `pint` again, 15 is printed again (as `r24` now points to `X`).

Okay, now what if we want to print the other values in the array? We use pointer arithmetic, just like in C. The `adiw` instruction is short for “add immediate to word”, and it has syntax `adiw [register], [number]`, and it increments the contents of `[register]` by `[number]`. This is exactly what's happening from Lines 37 to 44. These lines increment `X` and print 16 along with a new line character.

Tomorrow, we will pick up from this point and do some more Assembly.

19 Wednesday, July 3, 2019

Today is the last day of Assembly. Before we start, here's some quick review from the past two classes:

- Assembly programs are written in a `.S` file.
- There is a `.data` directive, which denotes the portion of the code where we define data (like variables).
- A label represents a memory address (it can be viewed as a pointer to a variable or a function).
- A value in memory is defined using the `.byte` directive. Hexadecimal, decimal, or binary are all permitted.
- The `.global main` directive is like the `extern` keyword in C—it indicates that the function can be accessed from outside of the current file.
- The `lds [register], [data]` instruction stores `data` into the contents of `register`. Similarly, `ldi [register], [constant]` allows us to load a constant value to `register`.
- The `clr [register]` instruction clears the contents of `register` to zero.
- Assembly has a built-in pointer register, denoted `X`, which represents the combination of registers `r26` and `r27`. (Why do we need two registers? Memory addresses are 16 bits, or 2 bytes, so they need two bytes).
- To initialize a pointer register like `X`, we use `ldi` along with `lo8` and `hi8` on `r26` and `r27`, respectively. We use these directives on whatever value we want `X` to point to.
- To load the contents of a register pointer into another register, we can use the `ld` instruction in the form `ld [destination register], [register pointer]`. This is the C-equivalent of dereferencing a pointer. If we now increment `r26`, we can write `adiw r26, 1` to move the pointer `X` by one. Using `inc` would also work, but it's not great to use since with register pointers as it only operates on one register. Assembly also supports a `+` operator. The instruction `ld r24, X+` would load the contents of `X` to `r24` and move the pointer by one.
- To save a value in a register, we use `push` in the form `push [register]` to push the contents of the register onto the stack. You need to have one `pop` for every `push`.

More on Register Pointers

Consider the following code segment, which is an example of writing to memory:

Listing 40: Assembly: Register Pointers

```
1
2 ;;; Example — writes the values 77 and 99 to memory
   using sts and st;
3 ;;; Also using X, Y, Z register pointers
4
5     .data
6 pctd:
7     .asciz "%d "
8
9 values: ; represents data memory area where we will
   write
10        ; note we are not using any .byte
   directive
11
12     .text
```

```

13
14 .global main
15 main:
16     call init_serial_stdio
17
18     push r29                ; needs to save
19     push r28                ; r29:r28 (callee-saved)
20
21     ldi r18, 77
22     sts values, r18         ; assigning 77 to
23                             ; location values (using sts)
24
25     ldi r28, lo8(values)    ; reading first value
26     ldi r29, hi8(values)    ; using Y (r29:r28)
27     ld r24, Y+              ; r29:r28 = values
28     ; using Y+ (increases
29     ; pointer by one location)
30     clr r25                 ; printing the value
31     call pint
32
33     ldi r18, 99              ; writing location
34     ; after first entry
35     st Y, r18               ; using st (NOT sts)
36     ldi r30, lo8(values)    ; using Z pointer
37     ldi r31, hi8(values)    ; register to read value written
38     adiw r30, 1             ; moving forward Z
39     ; pointer one position
40     ld r24, Z               ; reading value written
41
42     clr r25                 ; printing value
43     call pint
44
45     clr r25                 ; newline
46     ldi r24, 0xa
47     call putchar
48
49     pop r28
50     pop r29
51
52     cli ; stopping program
53     sleep
54
55     ret
56
57 pint:
58     ;; prints an integer value, r22/r23 have the
59     ; format string
60     ldi r22, lo8(pctd)      ; lower byte of the
61     ; string address
62     ldi r23, hi8(pctd)      ; higher byte of the
63     ; string address
64     push r25
65     push r24
66     push r23
67     push r22
68     call printf
69     pop r22
70     pop r23
71     pop r24
72     pop r25
73
74     ret

```

Everything up to the main is what we're used to. Note, however, that the values label doesn't have any

.byte directives—all this means is that the memory address corresponding to `values` hasn't been initialized.

In this example, we'll be using the `Y` register pointer (there's no particular reason why we're using `Y`—we could have used `X` or `Z` instead), which corresponds to the registers `r28` and `r29`. So, we save the contents of these registers, and we load the low and high byte of `r18` (which stores 77) into them. On Line 26, we load what `Y` is pointing to into `r24`, and increment `Y` by one (so `Y` is now pointing to a new uninitialized area of memory). Next, the value 77 is printed by calling `pint`.

Line 30 updates the contents of `r18` to 99, and line 31 stores 99 into `Y`. Note that we use `st`, which works with register pointers, instead of `sts`. Lines 32 to 33 initialize the register pointer `Z`, and Line 34 moves `Z` forward by one (`Z` is now pointing to 99). Thus, loading this value into `r24` and calling `pint` prints out 99.

Instruction Encoding and the Status Register

In Assembly, an instruction is represented by a set of bits which are assembled into a set of zeros and ones. But, not all of these bits are telling the assembler what operation to perform. The portion of the bits that encode what operation should be performed is known as the **opcode**.

So, what do the rest of the bits represent? The short answer is that it is dependent on the instruction we're considering. For instance, if we're dealing with `ldi`, we'd have the opcode, another portion to store the registers, and another portion to store the values of the registers.

The number of bits necessary to encode an instruction also varies. In AVR, however, we're guaranteed that instructions are either two bytes or four bytes. When memory is scarce, choosing the correct instruction is vital to saving memory.

In an Assembly program, there's a register—known as the **status register**—that keeps track of recent operations (particularly mathematical operations). We can view what's inside of the status register by typing `info r` in `gdb`. Why is the status register important? It allows us to perform conditional executions, known as **branching**.

Branch Instructions

We can use the status register to perform conditional execution of statements. This is done with the `cp` instruction, which has syntax `cp [register1] [register2]`. This is used to compare the contents of two registers.

Consider the following example:

Listing 41: Assembly: Register Pointers

```
1  ;; Example - if a == b prints 'Y' else prints 'N'
2  ;; Change a and b to see different outputs
3
4      .set LETTER_N, 'N'
5      .set LETTER_Y, 'Y'
6
7  ;; Global data
8      .data
9
10 a:      .byte 0x6
11 b:      .byte 0x5
12
13 ;; Program code
14      .text
15
16 .global main
17 main:
18      call init_serial_stdio
19
```



```

20      lds r18, a           ; reading a from memory
21      lds r19, b           ; reading b from memory
22      cp r18, r19          ; comparing registers
23      breq 1f              ; 1 is the target: f
                          represents forward
24      ldi r24, LETTER_N    ; store N
25      jmp 2f
26 1:    ldi r24, LETTER_Y    ; store Y
27
28 2:    clr r25              ; printing result
29      call putchar
30
31      clr r25              ; print newline
32      ldi r24, 10
33      call putchar
34
35      cli                  ; stopping program
36      sleep
37
38      ret

```

In this program, we read the value of *a* and *b* into registers *r18* and *r19*. These values are compared with *cp* on Line 22, which has syntax *cp* [*register1*], [*register2*]. The instruction *breq* is short for “branch if equal,” and its syntax is *breq* [*label*]. Pretty much, this instruction utilizes the status register to check the value of the preceding comparison. If the comparison indicated that the two variables were equal, the program jumps to the target label (here, the *f* indicates that we’re jumping forward).

Suppose $a \neq b$. In this case, we don’t jump to label 1. Instead, we’ll store the letter *N* in *r24*. The next instruction, *jmp*, indicates an unconditional jump (it always gets executed). Hence, once *N* is loaded into *r24*, we’ll jump forward to label 2, and we’ll print the result along with a new line.

Now suppose $a = b$ holds. In this case, we’ll jump over the statement that loads *N* into *r24*. Instead, we’ll load *Y* into *r24*, and we’ll continue from there and print the character along with a new line.

The advantage of using *cp* is that we don’t have to modify the registers.

Some other branch instructions are listed below:

- *cpi* [*register*], [*constant*] compares the contents of a register to a constant.
- *tst* [*register*] tests whether a register is non-positive.
- *breq* [*label*] branches to *label* if the previous comparison indicated an equality.
- *brne* [*label*] branches to *label* if the previous comparison did not indicate an equality.
- *brge* [*label*] branches to *label* if the previous comparison indicated *register1* was greater than or equal to *register2*. This should be used on signed integers.
- *brlt* [*label*] branches if *register1* is strictly less than *register2*. This instruction should also be used for signed integer.

It’s important to remember that branching instructions always look at the last result with the status register. Thus, comparison instructions need to come immediately before the branch instructions.

Here’s another example.

Listing 42: Assembly: Do-While Loop

```

1  ;;; Example – prints values 1 to 5 (do while)
2
3  ;;; Global data
4  .data

```

```

5
6 pctd:
7     .asciz "%d "
8
9 upper_limit: .byte 0x5
10
11 ;;; Program code
12     .text
13
14 .global main
15 main:
16     call init_serial_stdio
17
18     push r15                ; callee-save
19     push r16
20
21     lds r15, upper_limit    ; upper limit
22     ldi r16,1               ; loop starts, r16 is
                           the iteration variable
23
24 1:   mov r24, r16           ; printing value
25     clr r25
26     call pint
27
28     inc r16                ; increasing iteration
                           variable
29     cp r15, r16            ; checking whether we
                           reach limit
30     brge 1b               ; go back as long as
                           r15 >= r16
31
32     clr r25                ; newline
33     ldi r24, 10
34     call putchar
35
36     pop r16                ; restoring registers
37     pop r15
38     cli                   ; stopping program
39     sleep
40
41     ret
42
43 pint:
44     ;; prints an integer value, r22/r23 have the
                           format string
45     ldi r22, lo8(pctd)     ; lower byte of the
                           string address
46     ldi r23, hi8(pctd)    ; higher byte of the
                           string address
47     push r25
48     push r24
49     push r23
50     push r22
51     call printf
52     pop r22
53     pop r23
54     pop r24
55     pop r25
56
57     ret

```

There isn't anything confusing here. Register r15 stores the upper limit of loop. After each iteration, we compare the iteration variable, stored in r16, to the upper limit. If we haven't hit the limit, we branch back to the start of the loop. Similarly, we could implement a while loop by performing an initial comparison.

20 Monday, July 8, 2019