CMSC 216

Introduction to Computer Systems

Ekesh Kumar
Dr. Nelson Padua-Perez • Summer 2019 • University of Maryland
http://www.cs.umd.edu/class/summer2019/cmsc216/

Last Revision: June 9, 2019

Contents

1 Tuesday, May 28, 2019

Logistics

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

Basic Unix Commands

Unix has lots of commands, but we want to 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
  - `ls -F` → identifies directories by listing them with a /.
- `cd` → change directory to the inputted parameter.

Introduction to C Programming

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

Listing 1: A First Program

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

How does this program work?

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

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

Some more compilation options are summarized below:

- `-g` enables debugging
- `-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. That is, even if our header is `int main(void)`, *we will still return* 0 *at the end of the function.*

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 `.alias` file in the home directory of Grace.

Compilation Stages of a C Program

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

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

Variables in C

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

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

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

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

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

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

Consider the following code example:

Listing 2: Conditional Example

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

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

3 Friday, May 31, 2019

printf() and scanf()

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

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

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

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

Listing 3: Printing a Variable

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

`scanf()` is used for user input and it works similarly. The introduce the **address** operator, which is denoted by a `&`. The address operator is a unitary operator which, as its name specifies, returns the memory address of the variable on which it is acting on. Here's an example:

Listing 4: Reading Variables

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

Now suppose that the user enters the line

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

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

There are two important things that the programmer should check when using `scanf()`:

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.**

There are some more problems that can occur with `scanf()` and `printf()`.

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.

Control Statements

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

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

Functions

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

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

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

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

4 Monday, June 3, 2019

The sizeof Operator

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

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

Introduction to Pointers

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

Listing 5: Pointers Example

```
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 note actually changing x and y – we're changing their memory addresses. So, this works.

5 Tuesday, June 4, 2019

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

Identifier Scopes

There are two main types of scopes in C:

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

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

There are two types of storage types:

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

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

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

6 Wednesday, June 5, 2019

Invalid Uses of Pointers

Consider the following code segment:

Listing 11: Incorrect Pointer Usage

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

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

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

Listing 12: Correct Pointer Usage

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

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

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

Listing 13: Incorrect Pointer Usage

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

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

Null Pointers

The **null pointer** is a special pointer that points to the address 0, where nothing is allowed to be accessed.