C Course Material

May 11, 2014

1 Programming Environment

It's easier for all parties involved if everyone uses the same programming environment. We have decided to provide instructions for setting up a linux virtual machine. If you already use a modern linux system, this step is obviously not required and you can just install the required packages from your package manager.

1.1 Windows users

1.1.1 Installing VirtualBox

Using virtualbox you can install and run a linux environment just like a normal program, without having to change the operating system you are using currently. You can download and install VirtualBox 4.3 for windows hosts from https://www.virtualbox.org/wiki/Downloads.

1.1.2 Creating a virtual machine

1. Download *one* of the following Debian Wheezy ISO files.
   - Debian Wheezy 7.2.0 i386 (32 bit) (download via torrent or via http)
   - Debian Wheezy 7.2.0 amd64 (64 bit) (download via torrent or via http)
2. Open Virtualbox
3. In the top menu choose: *Machines→New*
4. Pick any name for your machine, set the type to *Linux* and set the version to *Debian* or *Debian (64 bit)* depending on what ISO file you downloaded before. Go to the next screen.
5. You are asked how much memory should be available to the virtual machine. Increasing it to 512MB or 1024MB is probably not a bad idea, but the default should work too. Go to the next screen.
6. You are asked to assign a hard drive to the virtual machine. Pick the default *Create a virtual hard drive now* option and go to the next screen.
7. You are asked for the hard drive format. Pick the default *VDI (VirtualBox Disk Image)* format and go to the next screen.
8. You are asked how the storage for this harddrive should be allocated. Pick the default *Dynamically allocated* option and go to the next screen.
9. You are asked for the filename and size of the new virtual hard drive. You can leave these options default and go to the next screen.

1.1.3 Installing Debian

At this point the virtual machine you have just created should be visible in the list of virtual machines when you have the VirtualBox program open. Next we will install the Debian operating system inside the virtual machine:

1. *Right click* the virtual machine you just created and choose *start*.
2. A box will pop up asking you to select a start-up disk. Point it towards the Debian ISO file you downloaded and go to the next screen.
3. When you click inside the debian virtual machine window a popup *might* show up to explain that the virtual machine will capture your mouse and keyboard focus. To get out of this state press *Right Ctrl*.
4. Once inside the Debian Installer window you will no longer be able to use the mouse. Use the arrow keys to select *Graphical Install* and press enter to confirm. This should bring you into a more familiar environment where you can use the mouse. Virtualbox *might* give a warning about mouse integration, you can just ignore that.
5. Pick English (the default) as a language and press continue.
6. Select your region (European countries are listed under Other).
7. After selecting your region the installer will ask you to select the locale. Whichever is recommended for your region is fine. If in doubt, you can go for *United states (en_US.UTF-8)*.
8. Next the installer asks you what type of keyboard you use. Select what applies to you. If in doubt pick American English which will select the default QWERTY layout.
9. The installer should start loading things at this point. Just sit back and enjoy the show until it comes nagging again.
10. The installer will ask for the hostname of the machine. You can pick whatever as long as it's a single word, or you can just leave it at the default *debian*.
11. When asked for the domain name, leave that blank and go to the next screen.
12. You are now asked to choose a root password. This is the password for the administration account of the linux environment. Enter the password and go to the next screen.
13. You are now asked to enter your name. Enter something and go to the next screen.
14. You are now asked to enter the username you want to use. Pick something and go to the next screen.

15. You are now asked to choose a password for the newly created user. Pick something and go to the next screen.

16. Again the installer should start loading things. As before, just sit back and wait.

17. In the next screen the installer asks how to partition the virtual hard disk that you created before. Just choose the default *Guided - use entire disk* and press continue.

18. The installer will ask what hard disk to install on. There will only be one option, the disk you have created before. Just select that and press continue.

19. Now the installer will ask you how the files should be structured. Just choose the default *All files in one partition (recommended for new users)* and press continue.

20. The installer will ask you to review the changes. Pick the default *Finish partitioning and write changes to disk* and press continue.

21. One final time the installer will ask you to confirm. Select *Yes* and press continue.

22. The installer will now start installing the base system. Again, just sit back and wait for the next nag.

23. The installer will ask to select a nearby mirror so the package manager can download updates properly. The default choice is based on your region and should be fine. Pick a country and then pick a mirror and press continue.

24. Next you will be asked to enter a http proxy. Leave this blank and press continue.

25. The installer will configure apt and start installing the updates it downloads.

26. Next the installer asks if you want to send information about packages you install to Debian. You can make that choice yourself. Pick whichever and press continue.

27. You're almost done now. The installer asks you which extra packages you want to install. Keep the *Debian desktop environment* and *Standard system utilities* selected but disable *Print server*. Press continue.

28. The installer will now download and install these extra packages. Again it's time to sit back and relax until it starts nagging once again.

29. And for the final step, the installer will ask you to install GRUB. Pick the default *yes* and press continue.

30. The installer will warn you to remove the CD before restarting, but the virtual machine will take care of that so you can just press continue.

1.2 Linux users

Just make sure you have a recent GCC version. We are using GCC 4.7.2 but an older version should not really be a big issue.

```
user@debian:~/$ gcc -version
gcc (Debian 4.7.2-5) 4.7.2
Copyright (C) 2012 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

1.3 OSX users

We can't provide any OSX Support but you should be able to find some information on getting a recent GCC version running on OSX by checking out the following page: http://railsapps.github.io/xcode-command-line-tools.html

Alternatively you can install VirtualBox for OSX and follow the installation instructions for Windows.

1.4 First steps using linux

After your virtual machine is done rebooting it will greet you with a login screen and ask you for the password you have provided during installation. Log in and you'll find yourself in a desktop environment that shouldn't look too unfamiliar to you.

In the top left corner you'll find the *Applications* menu, which is similar to the *Start* menu you're used to from windows. You'll also find the *Places* → *Home Folder* menu item in the top left corner. This will open a file browser in your home folder, you can compare that to *My documents* on windows.

1.4.1 The text editor

One of the two applications you'll use the most when programming is the text editor. You can open the text editor by using the menu in the top left corner: *Applications* → *Accessories* → *gedit Text Editor*. You could try playing around with the text editor, saving some files to the home directory and then deleting them using the file browser.

Try playing around and saving a text file to your home directory, then deleting it using the file browser to get familiar with the whole process. You might notice that you're not allowed to save and delete files outside your home directory (a few exceptions do apply), so you don't need to wory about causing damage to the system.

1.4.2 The terminal emulator

The most effective way to compile your source code is by invoking the compiler commands on the command line. This requires you to be somewhat familiar with navigating on the command line. It might look a bit intimidating, but in reality it's fairly simple.

Start off by opening the terminal emulator: *Applications* → *Accessories* → *Terminal*. Make sure you *don't* pick the *Root terminal* from the list instead.

```
user@debian:~/$
```

Once inside the terminal you should see something like the above. Between the colon (:) and dollar ($) the current directory will be printed. ~/ is short for your home directory. After the $ sign you can type commands. Try typing the *ls* command to list all files in the current directory.

```
user@debian:~/$ ls
Desktop Documents Music Pictures Public Templates Videos
user@debian:~/$
```

All files listed in blue are themselves directories. You can move into other directories using the *cd* (change directory) command. Let's move into one of the directories the *ls* command showed to us earlier.

```
user@debian:~/$ cd Documents
user@debian:~/Documents$ ls
user@debian:~/Documents$ cd ~/
user@debian:~/$
```

Using *mkdir* and *rmdir* we can create and remove directories. *rm* can be used to remove files. The touch command can be used to create new regular files, but you probably won't need that very often.

```
user@debian:~/$ mkdir programming
user@debian:~/$ cd programming
user@debian:~/programming$ mkdir deleteme
user@debian:~/programming$ touch metoo
user@debian:~/programming$ ls
deleteme metoo
user@debian:~/programming$ rmdir deleteme
user@debian:~/programming$ rm metoo
user@debian:~/programming$ ls
user@debian:~/programming$
```

There are two special directories that are available everywhere. by default *ls* hides these two special directories but you can show them using the *-a* flag. The current directory is represented by a single dot (.) and the parent directory is represented by a two dots (..).

```
user@debian:~/programming$ cd ../..
user@debian:/home$ ls -a
.  ..  user
user@debian:/home$ cd ./user/programming
user@debian:~/programming$
```

One more important command is the *man* command that will show you a manual file on how to use other commands. Once in the manual file you can quit it by pressing *q* and navigate it using *page up* and *page down*.

```
user@debian:~/programming$ man ls
user@debian:~/programming$ man mkdir
user@debian:~/programming$ man man
user@debian:~/programming$
```

1.4.3 Installing the compiler

We'll need to install the compiler before we can go on with the next section of the course. Installing the compiler is a fairly straightforward process. Start off by opening the root terminal by going to *Applications* → *Accessories* → *Root Terminal*. A popup will ask you for your administrative password, this is the root password you entered during the Debian installation.

Once in the root terminal all it takes to install the gcc compiler is typing the command `apt-get install gcc` and pressing enter.

```
root@debian:/home/user# apt-get install gcc
Reading package lists...  Done
Building dependency tree
Reading state information...  Done
More output removed...
```

2 Lesson One

In this lesson we're going to take a first look at writing C code and how to turn it into an executable file that we can run. We're also going to meet two programming concepts called variables and loops.

What you should do before starting this Lesson:

- Set up your Programming Environment.
- Learn how to navigate and create files in this environment.
- (Optional) Read the introduction to the accompanying book; The C Programming Lanaguage.

2.1 Hello, World

Open your text editor of choice and create a new file. Give it an appropriate name with the file-ending ".c". We're going to assume the file was named "hello_world.c" from now on. Once the file is open copy the content below into it (preferably by typing it over yourself).

```
#include <stdio.h>

int main(void)
{
    printf("Hello, World!");

    return 0;
}
```

To compile and run the code type the following in your terminal:

gcc hello_world.c
: Running this command will result in your file, hello_world.c, being compiled turned into executable output by our C compiler — the GNU Compiler Collection. The output will be written to a file called "a.out".

./a.out
: We run the file generated by our previous command by writing ./ followed by the name of our file.

Now, let us analyze this piece of software line by line:

#include <stdio.h>
: This line provides the compiler with the necessary information to use the input/output functions of the standard library. In this case we needed the printf function.

int main(void)

This line serves as the entry point for the Operating System when you run your software. We call this a function. The logic you wish your software to perform needs to be placed between the opening bracket ({) and the closing bracket (}) of this particular function. This function must always be named main.

printf("Hello, World!");

Here we see our program's first statement in the form of calling a function. In this case the function is one that will print the string inside the parentheses to the standard output, a concept we will meet later on. After the function call we can see a semi-colon, which serves to mark the end of a statement in C.

return 0;

Our second and last statement serves to return the result of the execution of our software. In this case our software finished its execution successfully; which we denote by return 0 to the Operating System.

2.1.1 Exercises

1. Try adding another printf statement below the current one that prints the string "Hello, Programmer!". What happens if you instead add it before?

2. Enter the following source code into your editor. Before running it try to predict what the output will be.

```
#include <stdio.h>

int main(void)
{
    printf("1");
    printf("2");
    printf("3");
    return 0;
}
```

   Was your prediction right? If not, how did it differ?

3. A student who's new to C is having problem making his code compile and run. See if you can help him interpret the error that GCC gives him by attempting to compile his code (below). Help him out by attempting to solve his error and making his code compile.

```
#include <stdio.h>

int main(void)
{
    printf("I'm the best C programmer in the World!")

    return 0;
}
```

2.2 Formatting Output

2.2.1 Newline

In the previous section we noticed that multiple calls to printf does not result in multiple lines being printed. This is because printf does not add a newline character to your output. Adding a line break in the middle of the string will not result in a newline either. What we need is the ability to print a newline character. C provides this possiblity through what is referred to as escape characters. They allow us to alter the meaning of a character, and in this case print characters that we cannot write in the source code.

This character looks like this: '\n', and despite involving two keystrokes — \and n — it's only a single character.
Compile the program below and run it:

```
#include <stdio.h>

int main(void)
{
    printf("This text is going \nto be "
        "broken up by a newline.\n");
    printf("And this line too is preceeded by a newline.");
    return 0;
}
```

Notice how there's no newline between "be" and "broken" despite there being one in the code! This technique we see here allows us to split a string over multiple lines by closing and reopening the double-quotes on the next line. It's identical to writing it all out on one line.

2.2.2 Expression

So far we've met code statements. In the form of:

```
printf("Hello");
return 0;
```

Both these examples are examples of statements. They're statements because they do something. In the case of printf; it prints a string to the standard output, and in the case of return 0; it returns a result to the OS.

```
1 + 2;
```

Above is an example of what we call an expression. An expression differs from a statement in the sense that it produces a result (in this case 3).
If you try compiling a program with the above expression you will notice that it yields no visible output. We'll get to how to do that in the next section.

2.2.3 Format Strings

```
/* The rest of this program has been excluded,
   we're currently inside main(). And this, by
   the way is a comment. A way to include a note
   to other programmers. It's invisible to the
   compiler */
printf("1 + 2");
```

When we run the above code we'll notice that the output is:

```
1 + 2
```

We might've expected it to be 3. However, "1 + 2" is a string, not an expression. So, how would we be able to print the result of an expression? Well, first we need to step back a bit and take a look at what the function call to printf actually looks like:

```
printf("Hello!");
```

The string "Hello!" inside the parentheses is what we call an argument. The first argument to this function is always a string. However, printf also accepts an undetermined amount of arguments that can follow this string. We seperate them with a comma:

```
printf("Hello!", 10, 5, 8);
```

This might seem nonsensical at this point. But there's an extra attribute associated to printf that we haven't discussed yet. And that is that the first argument — the string — is a *format string*. This means that printf will go through the string and look for special patterns that indicate a substitution. We call those patterns format specifiers. When printf comes across the first format specifier in a string, it will take the second argument to printf and substitute it. The second format specifier will be substituted with the third argument, and so on.

The format specifier explains how the argument is to be printed. For now we're going to introduce only one format specifier, the format specifier used to print integers. It looks like this:

```
%d
```

Taking this knowledge of format strings into account, we can print the result of our expression (1 + 2) with printf like such:

```
#include <stdio.h>

int main(void)
{
    printf("1 + 2 = %d", 1 + 2);
    return 0;
}
```

The %d in the above code will be substituted with the result of the expression in the second argument (1 + 2). This result happens to be an integer, so the output will be properly formatted and read:

```
1 + 2 = 3
```

For priting multiple expression results we might do:

```
printf("Printing %d some %d numbers %d", 8 * 8, 5 + 5, 25 - 5);
```

The first occurance of %d will be replaced with the result of 8x8, the second %d with the result of 5+5, and so on. The final output will be:

```
Printing 64 some 10 numbers 20
```

2.2.4 Exercises

1. Make a program that prints the following output:
   Hey, this
   string has 2 numbers
   and 2 newlines
   Did you use %d for the numbers? If so, could you have done without it?

2. Write a program that solves the following equations and prints the result to the standard output:
   1 + 5
   10 - 4
   5 * 4
   20 / 5
   Did you use %d for the numbers? If so, could you have done without it?

3. What happens if you try to print an expression that does a divison by zero? Do you think the result you got could ever be a problem in real-world software?

2.3 Variables

We previously said that expressions differ from statements in that they yield a result. This must mean that we're able to save this result, and further manipulate it somehow. And in fact it does. We do so using variables. A variable represents a place in the computer's memory, and we use a symbolic name to manipulate this piece of memory. The type of variables vary depending on their purpose — you're the one that decides if your variable is an integer or if it's something else; like the string "Hello!".
To *declare* a variable (make its symbolic name known) and *define* it (associate it to a memory location) we write a statement where we first specify the type, and then the name, and lastly we follow that up with a semi-colon, which we will remember ends a statement. It looks like this:

```
type name;
```

Our first type that we'll get to know is going to be the type that we use to store our average integer. This type is conveniently named "int", which is, you guessed it, a short for integer. Let's look at how we'd create an integer and give it the name "my_integer":

```
int my_integer;
```

To be able to save the result of an expression we use what is called the assignment operator in C. And a statement using the assignment operator looks like:

```
my_integer = 1 + 2;
```

The assignment operator is the same character that we use for equality in math, but its intent should not be mixed with the math intent. Instead, the C assignment operator takes the right hand expression (1 + 2) and stores the result into the left hand (my_integer). This means that my_integer is now holding the value 3. We call this process assigning a value to a variable.

We also note that using the format specifier %d from last section, we can print our integer variables. Let's write a test program that does this:

```
#include <stdio.h>

int main(void)
{
    int my_int;
    my_int = 10;
    printf("my_int = %d\n", my_int);
    my_int = my_int * 10; /* 10x10 */
    printf("my_int = %d\n", my_int);

    return 0;
}
```

At the first line of main we define an integer with the name my_int. We then proceed to assign it the value of 10 using our assignment operator on the next line. After that we print it, to make sure it is indeed equal to 10. Then we multiply my_int with 10, and assign the result of our expression to my_int. This will result in my_int now being equal to a hundred. We print it to make sure it is indeed so.

2.3.1 Exercises

1. Write a program in which you define an integer. Perform mathematical operations, and print it in between each operation. The output should be:
   2
   4
   8
   16
   32

2. Declare two integers. Assign 50 to one of them. Then assign that variable to the other one (int_two = int_one). Print both of them.

3. Carl keeps running his software, which is just a simple math operation, yet he keeps getting different results each time. Copy his code and run it repeatedly — are you getting different results as well? Why is this, and how do we fix it? Carl's code:

```c
#include <stdio.h>

int main(void)
{
    /* My program calculates the result of 1000 * 1000,
       but it doesn't work?! What is going on? */

    int my_var; /* my_var is 1000 */
    my_var = my_var * 1000; /* 1000x1000 */
    printf("%d", my_var); /* doesn't work :( */

    return 0;
}
```

2.4 Loops

With our current knowledge, in order to run a statement multiple times we would have to duplicate it, as such:

```c
/* A program that counts from 1 to 3
   using a variable. */
#include <stdio.h>

int main(void)
{
    int i;
    i = 1;
    printf("i = %d\n", i);
    i = i + 1;
    printf("i = %d\n", i);
    i = i + 1;
    printf("i = %d\n", i);

    return 0;
}
```

Copying over this program you're probably already feeling as if we're doing an exercise in futility. For example, let's imagine we wanted to count all the way from 1 to 10, or even 1 to 100? Granted, there has to be an easier solution

We call the solution loops. A loop is what we call an *iteration statement*. As the name suggests this allows us to execute a statement, or a block of statements, multiple times. Each time is a new iteration of that loop. Loops also have an associated condition; something that determines if another iteration of that statement needs to be performed. Our most simple loop is called a while loop, and it executes the statement while the condition is true. This is how a while loop looks:

```c
/* One statement: */
while (condition)
    statement;
/* Block of statements: */
while (condition)
{
    statement1;
    statement2;
}
```

Exactly what qualifies as a condition we are going to leave for a subsequent lesson.
So, let us look at how we might turn the above program that counts from 1 to 3 into a program that uses a while loop instead:

```
#include <stdio.h>

int main(void)
{
    int my_int;
    my_int = 1;
    while (my_int < 4)
    {
        printf("i = %d\n", my_int);
        my_int = my_int + 1;
    }

    return 0;
}
```

This particular while loop's condition should be interpreted as: WHILE my_int is LESS than 4 PERFORM the statements inside { and }.

2.4.1 Exercises

1. Rewrite the program that uses the while loop in this section to count from 1 to 10 instead.
2. See if you can make it count from 5 to 10, as well.
3. Write a program that counts from 1 to 100, and prints in like such:
   1 2 3 ... 98 99 100

2.5 Summary

This lesson did briefly touch on some core subjects that are used in most software that gets written. It does not by any means exhaustively explains any of the topics introduced. Instead it is meant to serve as a brief introduction that gets you familiar with some of the lingo, and terminology we use in programming. Everything that has been mentioned in this lesson is going to return in much more detail in future lessons.

2.6 Chapter Exercises

2.6.1 Easy

1. Write a program that prints your name and age to the standard output.
2. Write a program where you create an integer, and then print it three times to the standard output. Between each time you should change its value.
3. Write a program that counts from 1 to 5 and prints it to the standard output.

2.6.2 Intermediate

1. Write a program that counts from 50 to 100 and prints it to the standard output.

2. Write a program that counts from 10 to 1 and prints it to the standard output.

2.6.3 Challenging

1. Write a program that prints all numbers between 1 and 100 that are evenly divisible by 5. In other words:
   5 10 15 ... 90 95 100

2.7 Homework

This section contains work that you're expected to do after attending lesson one and before attending lesson two.

2.7.1 The Book

From now on we're going to be using the accompanying book — The C Programming Language 2nd Edition by Brian Kernighan and Dennis Ritchie. This book's code has some differences to the code we've met so far that we need to point out before we start. Please open the book and look at page 7 which contains a listing titled "The First C Program".

As you'll undoubtedly see it strongly resembles the Hello World program we saw in our first section. There are two key differences:

```
int main(void)
```

Has been replaced with:

```
main()
```

And:

```
return 0;
```

Is missing.

This will be the case in all programs we see in this book. The C90 standard specifies the version of main() that we're using as the correct one, and as such when you're copying code over from this book, be sure to replace it with our way of saying main().

In C90 reaching the end of the main function (}) without a return statement will make an undefined value be returned to the host environment. Undefined is a term we'll learn to interpret as meaning very bad as we delve deeper into the realm of programming. All you really need to know is that you should just stick returning 0 at the end of your main function, and then you don't need to worry anymore.

2.7.2 What to read

At page 5 starts the first section — "Getting Started". Most of this will be a repetition so reading it should go smoothly. But start from there and read down to the exercises at page 8.

Having completed them you should read the section about "Variables and Arithmetic Expressions" as well. Which ends at page 13 where you'll have some more exercises to do.

In this section we'll encounter something called a "float". For now we can realize that like "int" it's a *variable type*. Unlike int which holds integers (1, 2, 3, etc) this type represents numbers with a fractional part to them — such as: 1.5.

There will also be talk about *bits* and other variable types. You can note the terms for now, but it's not something you need to focus on. We'll introduce all of that more formally in the next lesson.

If you encounter anything that you're unable to figure out — whether that be in the text or an exercise — check the mailing list and see if anyone has asked your question already. If not, ask it on the mailing list!

3 Lesson Two

In this lesson we're going to take a step back from C and instead look at computer architecture. While we'll get to do some C towards the end of the lesson, the lesson will mainly be about how the computer works.

A lot of this stuff here is very detailed oriented. What I mean about that is that it's important for you to understand all of it, reading it through. But it's not important to remember all the details a few days from now. As long as you understand the concept, you can look up the details once you really need them.

IMPORTANT While the majority of this lesson is generally applicable, some of it is not generally true, but rather specific to certain architectures. Just to be safe, keep in mind that all that is said in this lesson applies to the x86 or x86-64 CPU architecture, but not necessarily all architectures.

3.1 Numeral Systems

3.1.1 Decimal — Base-10

In day to day life, most of us do mathematics using base-10, the decimal numeral system. Base-10 has, as the name suggests, 10 different symbols, namely:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9

This means that if we count from 0 and all the way to 9, once we pass nine a *positional increment* happens. This is because the *radix* is 10 in base-10 notation. Let's express a number, let's say 2704, using positional notation to emphasize the point:
$2704 = 2 * 10^3 + 7 * 10^2 + 0 * 10^1 + 4 * 10^0$

3.1.2 Binary — Base-2

When we're using base-2, we only have 2 available symbols. Namely:

0, 1

This means that our radix is 2, and our positional increment happens after we pass 1. So, counting from 0 to 10 using binary would be:

$$
\begin{array}{rcrl}
0 & \rightarrow & 0 \cdot 2^0 & = 0 \\
1 & \rightarrow & 1 \cdot 2^0 & = 1 \\
10 & \rightarrow & 1 \cdot 2^1 + 0 \cdot 2^0 & = 2 \\
11 & \rightarrow & 1 \cdot 2^1 + 1 \cdot 2^0 & = 3 \\
100 & \rightarrow & 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 & = 4 \\
101 & \rightarrow & 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 & = 5 \\
110 & \rightarrow & 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 & = 6 \\
111 & \rightarrow & 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 & = 7 \\
1000 & \rightarrow & 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 & = 8 \\
1001 & \rightarrow & 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 & = 9 \\
1010 & \rightarrow & 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 & = 10
\end{array}
$$

While it might not feel familiar at the moment, if you look closer you will notice that just as with decimal, when we're about to "hit" the radix (going from 9 to 10 in decimal, or from 1 to 2 in binary) we get a positional increment. For example:

Decimal: (decimal 99 to 100) 99, 100
Binary: (decimal 7 to 8) 111, 1000

3.1.3 Hexadecimal — Base-16

Hexadecimal is base-16, which means we have the following symbols:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

Just as with binary and decimal, going above F means we get a positional increment, in other words:

$$F + 1 = 10$$

3.1.4 Converting

With the knowledge that the different bases are just different notations, and that each base can express the same mathematical value just using different symbols, we realize that we can express the same amount in any base we choose. Knowing that we take a look at how we can convert between the bases, so that we end up representing the same number with our bases' symbols.

Converting to a base with less symbols than ours is easy. We simply divide our number with the radix of our target base, saving the remainder as our least significant digit. Then we take the result of our division and divide it with the radix again, saving the remainder as our second least significant digit. And so on while our number is bigger than 0.

Let's try converting 157 decimal to binary:

$$
\begin{array}{rlll}
157/2 & = 78 & \text{R } 1 & \rightarrow 1 \\
78/2 & = 39 & \text{R } 0 & \rightarrow 01 \\
39/2 & = 19 & \text{R } 1 & \rightarrow 101 \\
19/2 & = 9 & \text{R } 1 & \rightarrow 1101 \\
9/2 & = 4 & \text{R } 1 & \rightarrow 1\ 1101 \\
4/2 & = 2 & \text{R } 0 & \rightarrow 01\ 1101 \\
2/2 & = 1 & \text{R } 0 & \rightarrow 001\ 1101 \\
1/2 & = 0 & \text{R } 1 & \rightarrow 1001\ 1101
\end{array}
$$

And with that we can see that 157 decimal is the same as 10011101 binary.

As another example, Let's look at how to convert the number 61870 from decimal to hexadecimal:

$$
\begin{array}{rlll}
61870/16 & = 3866 & \text{R } 14 & \rightarrow \text{E} \\
3866/16 & = 241 & \text{R } 10 & \rightarrow \text{AE} \\
241/16 & = 15 & \text{R } 1 & \rightarrow 1\text{AE} \\
15/16 & = 0 & \text{R } 15 & \rightarrow \text{F1AE}
\end{array}
$$

As we can see converting to hexadecimal is very similar to converting to binary with the slight difference that we replace remainders bigger than 9 with the corresponding hexadecimal symbol.

3.1.5 Exercises

1. Convert 87563 decimal into hexadecimal.
2. Convert 16ED base-16 into base-10.
3. Convert 42 decimal into binary.
4. Convert 1011010 base-2 into base-10.
5. If you're still feeling unsure about the conversions, make up some numbers yourself and convert between the bases.

3.2 Computer Memory

In our computer's memory there are many, many tiny memory cells with transistors inside of them. These tiny memory cells can each hold either a value of 1, or a value of 0 depending on their voltage.

We can already see that using just one memory cell is going to allow the information we store to be very limited — we can after all just store a one or a zero. However, what if we had multiple memory cells?

Each memory cell can hold one *bit* — a binary digit. A group of 8 bits (8 memory cells) is what we call a *byte*. In other words:

8 bits = 1 byte

If we have 8 binary digits, 1 byte, we can represent the following binary numbers:

0000 0000 and 1111 1111

And any number in between. Using our knowledge of the bases from last section we can see that a byte allows us to represent any decimal number from 0 to 255. In other words, we can represent 2^8 different numbers.

If we have 2 bytes, we can see that the available numbers are $0...2^{16} - 1$ using the same logic as before.

Everything we store in our computer memory is stored in binary — it is how we interpret these binary digits that allow us to extract the information in the way we intended. E.g., are the binary digits part of an image, or a text file?

3.2.1 Exercises

1. If we have 4 bytes, what range of integers can we store?
2. If we have 8 bytes, what range of integers can we store?

3.3 Signed & Unsigned

If I gave you 1 byte, you could store integers from 0 to 255. Which seems all well and good for now, but how exactly would we go about storing something less than 0? A negative number, in other words. We already know that we can ONLY store 0 and 1 in a memory cell — so how exactly do we know that the number is negative?

3.3.1 Unsigned

If we use all our available bits to represent the number, we can represent any number from 0 to $2^{bits} - 1$. When we have discarded the posibility to represent negative numbers, we call that unsigned. That, as the name suggest, means that there's no sign (+/-).

3.3.2 Signed

To represent a signed number — a number where there is a sign (+/-) and we can express negative numbers — there exists some different systems. The most common one is called *Two's Complement*. This system is a derivation of *Ones' Complement*. Both these systems are dependent on the amount of bits we have available to represent our number. We say that we're using N-bit ones' or two's complement, where N is the amount of bits we are using. For simplicity let's use

3-bit systems for this part so that we can type out all the numbers in a small table. The unsigned table would look like:

Binary	Unsigned value
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

3.3.3 Ones' complement

In ones' complement the negative counterpart is obtained by taking the positive and inverting all the bits (0 becomes 1, and 1 becomes 0). Let's do that with 2:

$$\begin{array}{r} 010 \\ \hline 101 \end{array}$$

Which gives us that -2 is 101 in ones' complement. Let's build a table of the entire 3-bit ones' complement system:

Binary	Ones' complement value
000	0
001	1
010	2
011	3
100	-3
101	-2
110	-1
111	-0

Ones' complement is mainly interesting because two's complement builds on it. It has some problems that we're not going to go into as they're solved by two's complement.

3.3.4 Two's complement

Two's complement is the process of first applying Ones' complement (invert all the bits) and then adding 1 to the result. So 2 as we saw before would be:

$$\begin{array}{r} 010 \\ \hline 101 \end{array}$$

$$\begin{array}{r} 101 \\ +\quad 1 \\ \hline 110 \end{array}$$

When we get a carry (an extra bit needed to represent the result), we simply discard that carry. With that knowledge in mind we can see what applying two's complement to 0 will result in there not being a negative zero (as opposed to ones' complement):

$$\begin{array}{r} 000 \\ \hline 111 \end{array}$$

$$\begin{array}{r} 111 \\ +\quad 1 \\ \hline 1000 \\ \hline 000 \end{array}$$

The two's complement for 0 is for that reason always requiring one bit more than the bit-count of the system itself, which means the 1 is discarded and we're back at all zeros.

Let's look at the two's complement table for the 3-bit variant of the system:

Binary	Two's complement value
000	0
001	1
010	2
011	3
100	-4
101	-3
110	-2
111	-1

Doing math operations in a two's complement system is simple (red = decimal, green = binary)

$$1 + 2 = 001 + 010 = 011 = 3 \tag{1}$$

$$3 - 4 = 3 + (-4) = 011 + 100 = 111 = -1 \tag{2}$$

Some observations:

1. In both systems the left-most digit is 1 if it's a negative number, and 0 if it's a positive one. This digit is called the *Most Significant Bit* or MSB for short.

2. In a N-bit system N amount of ones (4-bit: 1111, 8-bit: 1111 1111) is equal to -1.

3. In a N-bit system a zero and N-1 following ones is always the biggest positive number in that system. (4-bit: 0111, 8-bit: 0111 1111).

4. In a N-bit system a one and N-1 following zeros is always the smallest negative number (biggest absolute value) in that system. (4-bit: 1000, 8-bit: 1000 0000).

3.3.5 Other signed sytems

We looked at ones' complement mainly to providee context for two's complement. But there are other systems to deal with this problem as well. If you're interested you can read more at: http://en.wikipedia.org/wiki/Signed_number_representations.

3.3.6 Counting the bits

Just as MSB means most significant bit and indicates the left-most bit, LSB means least significant bit and indicates the right-most bit.

0000 0000

Here, red is the MSB and green is the LSB.

3.3.7 Memory addresses

As we noted before a group of 8 bits is a byte. This is also the smallest individual piece of memory we can *address* (address here means to either write to or read from). This means that our memory can be thought about like this:

a: byte
a+1: byte
a+2: byte
a+3: byte
...

Where a is an address. An address is a number that we use to refer to individual bytes in our computer's memory. Each increment in the address number is one extra byte we go over.

3.3.8 Endianness

The smallest piece of memory we can read from or write to is a byte. This means that there's no disorder that can happen inside the individual bits. But once we start using multiple bytes for our information, the question becomes how the bytes are ordered.

Before we look at an example, let's note that an unsigned byte goes from the decimal numbers 0 to 255, which is the hexadecimal numbers 0 and FF. This means that a byte can be conveniently expressed with two hexadecimal symbols, ranging from 00 to FF. Now, let's consider that at address "a" we want to store the number AABBCCDD. We can see that this number does not fit into one byte — in fact, there's 4 groups of 2 hexadecimal symbols, which means we need 4 bytes to store it.

The question that arises is do we store DD at address "a", and AA at address "a+3", or do we do it the other way around?

In *little-endian* the smallest address number (a) gets the least significant byte (not to be confused with least significant *bit*) — DD. In othe words, if we use a CPU architecture that uses little-endian our number will be stored like this:

a+0: DD
a+1: CC
a+2: BB
a+3: AA

In *big-endian* the smallest address number (a) gets the most significant byte (again don't confuse bit with byte) — AA. In othe words, if our CPU architecture uses big-endian, our number is stored like this:

a+0: AA
a+1: BB
a+2: CC
a+3: DD

When writing programs in C local to our own system, there's not much to worry about — our compiler knows our endianness and when we use our integer types, the order of the bytes are handled for us. The times when we as programmers need to remember the difference is when we transfer data from one architecture to the other — this most commonly happens in networking scenarios.

3.3.9 Exercises

1. Make a table of 4-bit two's complement system. Fill in all binary patterns and resulting two's complement values in that table.

2. Using your table, solve the following arithemtic operations by replacing both numbers with their 4-bit two's complement binary pattern, and then answering using binary:

(a) 2+3

(b) 6-4

(c) 5-6

(d) 1+5

3. At address "a" you have 4 bytes you intend to read. Read them first using little-endian, then using big-endian. Write down the two numbers in base-10.

   a+0: 00
   a+1: 00
   a+2: 4D
   a+3: FA

3.4 Floating point

So far we've not examined how we may represent anything but integers. When talking about real numbers, we're not only speaking about integers, but also about fractions and irrational numbers. For example, $4/3$ is a rational number, but it's not an integer. Floating point is a method in computing aimed at representing approximations of real numbers. There are different ways of doing this. We're going to be looking at a standard, IEEE 754, which is a common way to represent floating point numbers.

3.4.1 IEEE 754

The IEEE 754 standard describes multiple formats, but we're going to put all our focus on two of those. Namely single precision and double precision.

3.4.2 Single precision

This precision uses 32 bits (4 bytes) of information. The left-most bit is the sign, just as with signed integers. After that follows 8 bits for a biased exponent. And lastly 23 bits for the fraction. Like this:

0000 0000 0000 0000 0000 0000 0000 0000.

Where blue is the sign, red the biased exponent and green the fraction. Let's look at how these bits are used to express a value:

$$Value = (-1)^s * 1.f * 2^e$$

Where s is the sign bit, f is the fraction and e the biased exponent.

If we first look at the biased exponent. The biased exponent does not use two's complement, but it can still accept negative numbers. It solves the negative number problem using a *bias*. In the case of single precision the bias is 127. What this means is that you take your actual exponent and add 127 to get the biased exponent. Vice

versa, you get the actual exponent from the biased exponent by subtracting 127. As you can see this allows us to represent negative numbers in the exponent as well.

The "1." is not part of the 23 bits you use for your fraction, it's instead implied. Which means that if you store 101 in your 23-bit fraction component, your fraction is implicitly 1.101.

Let's look at an example. Let's say we have the decimal number: 5.125. First step is to convert 5.125 to binary.

5 we will remember is 101.
Knowing that 0.125 = 1/8,
we are able to write that
as 1/1000 in binary. Which gives
us that 0.125 decimal is 0.001
binary.
Our number is: 101.001.
$101.001 = 1.01001 * 2^2$

We see that our fraction is 1.01001, and the part we need to store in the binary digits that represent our fraction is 01001.

We also have an exponent of 2, which if we apply our bias to becomes:

$$2^2 = 2^{2+127} = 2^{129} = 2^{10000001}$$

Our biased exponent is, as we can see, 1000 0001. This means that our 5.125 is stored like this in our memory:

0 1000 0001 0100 1000 0000 0000 0000 000

3.4.3 Double Precision

For double precision we have 8 bytes (64 bits) available. The MSB is still the sign. The biased exponent is 11 bits, and the fraction the remaining 52. The bias for this exponent is 1023.

3.4.4 Non-numbers

The biased exponent cannot be all zeros, or all ones. All ones carry special meaning, and all zeros is used for 0. See table below.

Sign	Fraction	Exponent	Meaning
0	all zeros	all zeros	0
1	all zeros	all zeros	-0
0	all ones	all zeros	infinity
1	all ones	all zeros	negative infinity
0 or 1	all ones	non-zero	NaN (Not a Number)

3.4.5 Exercises

1. Can $\sqrt{2}$ be represented exactly using floating point? Either with single precision or double precision?

2. Given that floating point triest to express real numbers using a finite amount of bits, do you think it's common to have "rounding errors"?

3. Following the previous question. Would you think exact comparison between floating points is a good idea, or would you rather use relative comparison (f1 $<$ f $<$ f2)?

3.5 Character encoding

Since we are only able to store patterns of 1s and 0s, we need some unified way to map letters & other characters to numbers. So that when reading the string "Hello, World", we are sure to see the letters intended.

The solutions to this problem is called character encoding. It's simply a "map" that maps integers, which are storable as bit-patterns, to characters. If both parties of a conversation has the same "map" they are able to properly communicate using only binary transmissions.

A common character encoding that is based on the English alphabet is called the American Standard Code for Information Interchange, or ASCII for short. The ASCII character encoding scheme has in total 128 specified characters — among those are 0-9, a-z, A-Z, punctuations symbols, space and a number of control characters (one being the newline character we met last lesson).

3.5.1 Exercises

1. On http://www.asciitable.com/ you can see the 128 characters that make up the ASCII character encoding. Go check it out. See what your name would be if you wrote it in a binary-pattern using ASCII encoding.

3.6 Variable Types

So far we've only used int as our variable type. In K&R we got to look a bit at float as well. In this section we'll encounter the various types. Let's look at a table below of the types. This table does not account for signedness.

Keyword	Purpose
char	character — always a single byte in size
short	short integer
int	integer
long	long integer
float	single-precision floating point
double	double-precision floating point

As you can see from the table, only char has a defined size. There's also the

addition that short is always *at least* 2 bytes. Integer sizes that come later in the list are never smaller than those before it, but they can be of equal size. Long is always *at least* 4 bytes.

Short, int and long are signed by default. Char is system-dependent and can be either signed or unsigned. All integer types (char, short & int) can be preceded with the keywords "signed" or "unsigned" to make them just that. Such as:

```
unsigned char;
signed int;
```

While char is just an integer type like the rest, its size is guaranteed to be big enough to hold any ASCII character (0 through 127, 128 characters in total).

3.6.1 Format Specifiers

Some notable format specifiers:

Type	Specifier
int	%d
unsigned int	%u
char	%c
double	%f

3.6.2 Constants

The way we write a constant in our code, affects its type. For example:

```
printf("%f", 1/8);
```

Prints "0", not "0.125". The reason is that 1 and 8 are both integers, and as such any mathematical operation between the two will produce an integer (why this is and how it works exactly will be covered more later on in the K&R book). The solution is to write:

```
printf("%f", 1.0/8.0);
```

Which means 1 and 8 are both of type double (if either of them is a double they are both promoted to a double, and the result is a double as well). Below is a table that shows some notable constants and their resulting type:

<nr>	integer (short, int long)
<nr>u	unsigned integer (short, int long)
'a'	char
1.0f	float
1.0	double

For integers, the underlying type depends on the size of the integer. What is guaranteed is that it's one that is big enough to hold your number, as long as your number can be expressed using the guaranteed 4 bytes your long type is. For example, if your number is let's say 8 bytes, your code is no longer portable as the C90 standard's biggest *guaranteed* type is 4 bytes (it can still be bigger depending on your system). All in all, we need not worry too much about this for now.

3.6.3 Exercises

1.
```c
#include <stdio.h>

int main(void)
{
    X var;
    var = (exercise);
    printf("%Y", var);

    return 0;
}
```

   In each of the following exercises, replace (exercise) with the constant and then fix X so that you use the same type as (exercise) yields. Make sure to change Y so that you print it properly as well.

   (a) 5.5

   (b) 25

   (c) 'f'

   (d) 1.0/4.0

   (e) 25.0f

   (f) 's'

2. The following code snippet yields output its creator didn't expect. Fix the code for him:

```c
int f = 1.0/2.0;
printf("%f", f);
```

3. Create 5 character variables. Assign 'h', 'e', 'l', 'l', 'o', to them and print them one by one next to eachother.

3.7 Summary

In this chapter we took a very theoretical detour that allowed us to look at a lot of new concepts that are useful to know for us as programmers. Given that you have once understood these concepts, you can easily look up the details once you are in need of knowing the specifics. However, for most of what has been covered in this lesson, remembering the specifics is not all that important.

3.8 Chapter Exercises

Due to the theoretical nature of this lesson, there will be no chapter exercises. Instead the homework section will have a lot of exercises for you to work on.

3.9 Homework

The exercises early on in K&R are fairly complicated, and do not simply test if you understood the concept, but rather challenges you as a programmer. This is mainly because the intended audience are programmers already. For that reason, for each section you will read here in the homework department I will give you some exercises to do instead. Of course, if you're feeling adventurous you're welcome to try the book exercises as well.

As always, if you're having problems with the homeworks, mail the mailing list.

3.9.1 1.3

Open the C Programming Language and go to section 1.3 about for-loops. Read through it and come back here to do the exercises.

3.9.2 Exercises 1.3

1. Write a while loop that counts from 1 to 100 and prints the process. Then write a for loop that does the same. Which one do you think is the nicest way to solve the given problem?

2. Make a for loop that counts down from 100 to 0, printing while counting.

3. Make a for loop that prints the 4th multiplication table (up to & including 40).

4. The code below has a loop inside of a loop. Look at the code below and try to predic its output. Run the code and see if you were right. Why does it work the way it does?

```
#include <stdio.h>

int main(void)
{
    int i, j;

    for (i = 0; i <= 5; i = i + 1)
    {
        for (j = 0; j <= 5; j = j + 1)
            printf("i = %d, j = %d\n", i, j);
    }
}
```

5. Write a loop inside a loop to print the entire multiplication table (up to 10x10).

3.9.3 1.4

Open the C Programming Language and go to section 1.4 about symbolic constants. Read through it and come back here to do the exercises.

3.9.4 Exercises 1.4

1. Writea a program that calculates the area of all circles with an integer radius between 1 and 10. Define π as a constant. Print the area to the standard ouput. (Circle Area $= \pi * r^2$.)

3.9.5 1.5

Open the C Programming Language and go to section 1.5 about character input and output. To type an EOF character (a concept you will meet while reading) you can press ctrl+d on your keyboard. The books exercises for 1.5.1 are simple and straightforward, so make sure to do those while reading. Then read 1.5.2 and 1.5.3 and come back and do the following exercises:

3.9.6 Exercises 1.5.2 & 1.5.3

1. We see "if" for the first time in this section. Unlike loops, which iterate their statement while the condition is true, if simply does the statement once if the condition is true, otherwise it skips it. Prove this by writing an if that prints "Always True" for a condtion that can never be false (let's say $0 < 1$). Then an if that would print "Never True" for a condition that is always false (let's say $0 > 1$).

2. Write a program that reads input until we encounter EOF. For each character read, print the same character to the standard output using printf.

3. Change the line counting program in 1.5.3 to count spaces instead of newlines.

4. Write a program that counts newlines and spaces, and then prints how many of each was read.

5. Make your program from last exercise also print how many tabs were read ('\t').

6. Write a program that exploits your knowledge of ASCII by checking if a character is in the range 'a'-'z', and if so make it a capital letter (you can do this with a simple addition, since you know characters are just integers).

7. What would happen if you ran your program on a system that did not use ASCII character encoding for input? With that in mind, do you think the program from last exercise is universally portable (works the same on all systems)?

8. Make a program that increases a counter IF you encounter an 'a' OR an 'E' or an I, or an 'o', or a 'u'. After EOF has been encountered, print out how many TOTAL vowels you encountered.

Read 1.5.4 and do the two exercises that it contains, as they too are fairly straightforward.

3.9.7 1.6

Now comes a rather difficult section, that we're going to spend future lesson time on to get better understanding of. So finding this section a bit confusing is only natural. But try to read it through carefully and do the exercises to the best of your abilities. If you get stuck, remember to use the mailing list to ask questions.

Read 1.6 and then return here to do the exercises.

3.9.8 Exercises 1.6

1. An array is a convenient way for us to group variables as one symoblic name. Let's write a program that reads input until EOF and counts all vowels (a, e, i, o u, excluding y), but instead of printing the total of vowels, let's print how many of each vowel there is. Declare an array for that purpose, like such:

   ```
   #define MAX_VOWELS 5
   ...
   int vowels[MAX_VOWELS]; // subscript: 0...4
   ```

   After encounterinf EOF, print the amount of 'a's, 'e's, etc.

2. Modify the above program to also print the total amount of vowels without adding new variables.

3. We learned the following structure:

   ```
   if (condition)
       statement;
   else if (condition)
       statement;
   else
       statement;
   ```

   Using this strucutre, write a program that keeps counters of how many lower-case characters, upper-case characters and digits you encounter. Print the result.

4. Add an extra else if to the previous program and also print the number of spaces you encounter.

5. Write a program with an array and a long if, else if, else statement where you count the amount of digits 0 through 9 that you receive. Then print it as:
   Found X 0s:
   Found X 1s:
   Found X 2s:
   ... Found X 9s:

3.9.9 1.7

In this chapter we will see a brief introduction of functions. Along with the previous concept — arrays — this tends to be a confusing topic for new programmers, as well. The best way to get a hang of confusing topics is not to worry too much, and just keep experimenting until the logic behind it starts getting clearer.

Read 1.7 and come back here to do the exercises. In 1.7 you will find what they call "A note of history", you're free to ignore that since its relevance is close to zero 25 years later.

3.9.10 Exercises 1.7

1. Complete the following program by using the function mul(int x, int y) to print the 5th multiplication table (up to 5x10).

```c
#include <stdio.h>

int mul(int x, int y);

int main(void)
{
    // ...
    return 0;
}

int mul(int x, int y)
{
    return x * y;
}
```

2. Change the above function to work with floating point numbers instead. Test it by multiplying some real numbers (e.g. mul(2.5, 0.5)).

3. Complete the following program by defining the function div. Print an error and return a symbolic constant -1 if someone tries to divide by zero.

```c
#include <stdio.h>

#define DIV_BY_ZERO -1

int div(int x, int y);

int main(void)
{
    printf("10/5 = %d\n", div(10, 5));
    printf("Testing division by zero:\n");
    div(5, 0);
    return 0;
}
```

4. Write a function that prints all numbers between x and y, and also returns the accumulated value ((x+1)+(x+2) + ... + y). Test the function a few times in your main().

3.9.11 1.8

Read 1.8 and prove that changing the value of a parameter in a function does not change the value of the argument by writing a test program.

4 Lesson Three

4.1 The Compiler

In order to do anything on a computer we need to tell the CPU (or Central Processing Unit) what to do. How this happens exactly is not that important at this stage, but having at least a rough idea is useful for introducing the concept of compiling.

The CPU gets its intstructions in the form of "machine code", this machine code is read from the memory and is represented by a stream of bits. Essentially this code consists simply of numbers that the CPU knows how to interpret and it will perform simplistic tasks (add two numbers, read a number from memory, write a number to memory). Computer programs consist of large sequences of these simplistic instructions.

Writing machine code is hard and error prone, and while nowadays no one does that by hand anymore in the early days of computers that was the only way to program a computer. Over time people have developed ways to make it easier to program computers. The C programming language is a third generation language and is a high level language that needs to get translated into machine code before the computer can do anything useful with it. If you're interested reading more about the different generations of programming languages I encourage you to check out http://en.wikipedia.org/wiki/Programming_language_generations.

This is where our compiler comes into the picture. The compiler is a program that will translate the high level language we as humans write into the machine language your CPU can understand. This happens in multiple steps, but what we care about for our purposes are the compiling and the linking steps.

In the compiling step you feed the compiler with source code files (the files we've ended in .c) which it will then translate into machine code. The compiler outputs an object file for every source file it was given. Object files usually contain machine code, or sometimes an intermediate language that's close to machine code but not dependent on the current architecture. They also contain other data the linker will need but we don't really need to know much about that right now. During the linking stage we then "link" all the different object files and other system libraries together into a single executable. The compilation commands we've been using so far combine the compiling and linking step into one, but more complex programs will often split up those steps.

4.2 What we covered from the book

During the lesson we covered from 1.8 to and including 2.12 in the K&R book.

4.3 Homework

4.3.1 If you attenended the lesson

If you attended the lesson we skipped some sections that we felt were easier to read on your own. Below is a list of which sections you need to read:

1. 2.3
2. 2.7
3. 2.11
4. 2.12

4.3.2 Book Exercises

Do all the book's exercises between 1.8 and 2.12. If you get stuck, make sure to mail the mailing list.

4.3.3 Chapter 3

Chapter 3 in K&R is a very short chapter that just formally reintroduces concepts we've already learned. It's only 11 pages. Read it as a refreshment (it has some new info as well!) of the control structures we've used up until this point.

4.3.4 Exercises

Here below are some real-world exercises of problems to solve. They can be solved in multiple ways, but try to use as many of the new concepts you've learned as possible.

The vending machine Write a program which repeatedly queries the user for various bevereges. Make sure to handle someone inputting an invalid beverage correctly (tip: you can use integers; 0 to n to let them order beverages). Once EOF has been encountered, print how many of each beverage was ordered.

Power of N Write a function that is able to calculate power of. Exercise the function with some test cases.

String copy a) Write a function that copies a C-string from one character array into another character array. Assume that the destination character array is big enough to hold the source for now.

b) We have a problem in our function – if the source string is bigger than the size of the destination character array, we're going to be writing to memory that's not part of the destination! Oops! Figure out a way to solve this case!

Caesar Cipher Going for a trip in history, let's learn about the Caesar Cipher, which is an encryption technique named after Julis Caesar. The cipher is simply taking the letter and shifting it 3 steps to the right in the alphabet, so: A becomes D, E becomes H, and so on.

Write a function to encrypt a C-string with this cipher, and one to decrypt it. Exercise your functions by accepting user input, a line at a time, then first encrypting it, then decrypting it again, making sure you get the same string you entered printed back.

5 Lesson 4

5.1 Pre Lesson

The assignment to read Chapter 4 in the K&R book and do the exercises was sent out to the mailing list, so hopefully most people have don that by now. Along with that was the assignment to do and submit a quiz. The quiz is still available if you want to complete it at: https://www.worldofcorecraft.com/question.html.

5.2 Lesson

In this lesson we covered chapter 5. Some parts were left to the homework section.

5.3 Homework

5.3.1 If you attended

These are the sections we left for you to cover on your own:

1. 5.6. We sort of covered this, but the example code in here is useful, so check it out.
2. 5.7.
3. 5.9.
4. 5.10.
5. 5.11. We covered this subject, but we never saw an example of when to use the concept. This section ties nicely back to the code in section 5.6, and you should compare them for clearer insight.
6. 5.12.

5.3.2 For Everyone

All the exercises in chapter 5 are valuable, so you should do them and make sure you understand the entire chapter's content clearly. The content within is very important and required for the following lessons.

6 Lesson Five

In this lesson we're going to meet a new concept called structures, we're also going to find out about different segments of memory. This chapter is going to conclude with a pretty big exercise, that you will also turn in and have reviewed by us (to get valuable feedback).

6.1 Structures

As K&R puts it: a structure is a collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling.

Let's use a World of Warcraft example. The position of a player, the X, Y and Z, could be conveniently grouped into a structure that we could name something descriptive such as position. Let's see how we'd do that:

```
struct position
{
    float x;
    float y;
    float z;
};
```

A structure declaration defines a type. And just as with any type it can be followed by a list of variables. For example:

```
struct { ... } a, b, c;
```

Creates the variables a, b and c that are of the type our nameless structure defines. Which is syntatically analogous to what we'd do for a built-in type:

```
int a, b, c;
```

Given a structure with a name, we'd create a variable of that structure type like this:

```
struct position { /* x, y z from before */ };
struct position p; /* p: variable of type struct position */
```

Just as with any other variable, variables of a structure type can be initialized:

```
struct position p = { 10.0f, 5.0f, 1.0f };
```

Just as with arrays; if an incomplete initialization list is provided, then the remaining elements will be initialized to zero. Such as:

```
struct position p = { 10.0f }; /* y == 0 && z == 0 */
```

We call the variables that are part of the structure members of that structure. To denote a particular member, we write the name of our structure type variable, followed by a period and then the name of the member variable. Like this:

```
p.x = 5.0f;
```

Structures can also be nested, such as:

```
struct line
{
    struct position p1;
    struct { int a; int b } my_ints;
};
```

The names of the members are local to the structue, and the following code has no name conflicts:

```
void f(void)
{
    struct position p;
    float x, y z;
    /* ... */
    p.x = x;
    /* ... */
}
```

6.2 Using structures

There are only a few legal operations on a variable of a structure type: copying it, assigning using it as a whole unit, takings its address with the address of operator (&), and accessing its members. This means that structures may not be compared, for example. Let's look at some example code:

```
struct position p1, p2;
/*...*/
p1 = p2;        /* okay, same as doing:
                   p1.x=p2.x; p1.y=p2.y; p1.z=p2.z */
void f1(struct position p);
f1(p1);         /* okay: f1's prototype is:
                   void f1(struct position p); */
void f2(struct position* p);
f2(&p2);        /* okay: f2's prototype is:
                   void f2(struct position* p); */
p1.x = p2.x     /* okay */
if (p1 == p2)   /* error! */
```

Let's look at a function we call make_position, that constructs a position from 3 floats:

```
struct position make_position(float x, float y, float z)
{
    struct position temp = { x, y, z };
    return temp; /* a copy of temp is returned */
}
```

An important comment has been added here to the return statement. To further investigate what this means, let's look at the following function:

```
struct position increase_x(struct position p, float x)
{
    p.x += x;
    return p;
}
```

```
int main(void)
{
    struct position pos = { 10.0f, 5.0f, 0.0f };
    pos = increase_x(pos, 5);
    return 0;
}
```

When we call increase_x, the parameter p gets a copy of the argument pos, and when the function returns the value of the function's parameter p is returned, and copied back into main's pos. This means that the entire structure is copied around; which seems unncessary for our purpose – after all we just wanted to modify our position. As you can imagine, copying around big structures can be quite inefficient. So, instead, we could pass a pointer to struct position:

```
/* improved increase_x */
void increase_x(struct position* pp, float x)
{
    pp->x += x;
}
```

As you may or may not have noticed, we no longer use a period to refer to our structure's member. That is because we have a pointer to struct, so we first need to dereference it, and then access the member. This can be done using what we have learned so far:

```
(*pp).x
```

But there's also an operator, which we used in our function, that does these two steps in a syntactically cleaner way, and can be used as a shorthand. It simply looks like an arrow:

```
pp->x
```

-> and . binds very tightly, which means that given a structure:

```
struct
{
    int len;
    char* str;
} p;
```

The following statement:

```
++p->len;
```

Increases len and not p, because of the precedence of the -> and . operators. Enough said about structures for now.

6.3 Memory Segments

The programs we have compiled up until this point have been organized into what we refer to as memory segments (sections). Different segments have different attributes and purposes. For example, for one memory segment we might say that what's inside cannot change. And an attempt to change it would make the OS kill the program. Your programs written so far, as loaded into memory, have the following

memory segments: text, data, bss, stack and heap. The text segment contains all our executable code, and does not change when we're running our program. In the data segment are our initialized global and static variables (non-automatic). In the bss segment are our uninitialized global and static variables (non-automatic). The stack holds all our variables that are of automatic storage.

Let's look at a program that hopefully explains this well enough:

```
float data_variable = 50.0f;
int bss_variable;

void f(void)
{
    int stack_variable;
    static int bss_variable;
    static char data_variable = 'A';
}

int main(void)
{
    float stack_variable;
    int i; /* stack as well */

    for (i = 0; i < 10; ++i) /* text segment */
        /* ... */;
}
```

The names of everything should make it fairly clear where the different variables reside. The for loop is added for emphasis that all code is in the text segment (our main() and f() included).

Text, data and bss are all going to remain the same size throughout the program's execution. Our stack is not, however. Our stack can grow or shrink on a need basis. The stack grows toward *lower* memory addresses.

The only segment we mentioned earlier that we have not yet seen is called the heap. This is actually where we put the majority of our data, especially data that are going to persist for longer than a single function's scope. The heap grows towards *higher* memory addresses. This is the opposite of the stack, and the following image demonstrates how it all fits together:

http://i.imgur.com/2tTfZaU.png

Note that the memory model we've described here is what we might call a common praxis, based in history, but it is in no way the only way to arrange it.

6.4 The Stack

The stack is an interesting construct that I'd like to take a moment to discuss further. The stack is what we call a LIFO (Last in First Out) construct. This means that the last object you put on the stack, is the first one you need to retrieve. So if you put 3 objects on the stack, and intend to get the first one out, you'll need to remove the other two as well. Where object is an arbitrary bundle of bytes. We call the process of putting something on the top of the stack *pushing*, and retrieving something from the top of the stack we call *popping*.

The stack's purpose is to store any variables whose storage is temporary. It's also used to store a sort of "breadcrumb trail" when we call functions, so that we easily can find our way back to where we came from. (And other stuff we don't care about at the moment.)

Basically, when a function is called, the arguments are pushed onto the stack. A return address (so we can find our way back from the function) is also pushed onto the stack. Then the function itself is called. Inside the scope of the new function, the local variables' initialization values will be pushed onto the stack.

To fully understand the purpose of the stack, we'll have to make a rather concrete example. And to do that we will have to look at actual x86 assembly! But don't get too scared, we'll take a very brief peak with much assitance. The point being that we want you to *really* understand the basics of the stack.

We need to quickly and vaguely introduce two concepts before we move forward.

One is called *registers*. We can think of registers sort of like variables for the CPU. Each register is the same amount of bits as your CPU is; so for a 32 bit processor, your registers are 4 bytes wide, and for a 64 bit processor they're 8 bytes wide (reminder: all our discussion is about the x86 and x86-64 architecture).

The other is what is called a *stack frame*. All your functions get their own "frame" on the stack, that contains what is important for your function (like its local variables). Every time you call a function, that functions frame goes on the top of the stack.

Two registers that we will meet soon, are called *esp* and *ebp*. The first one, esp, is a "pointer" (holds the address) that points to the top of the stack. The ebp register will point to the beginning of the current active stack frame. Together they make up the scope of the current stack frame.

Take a minute to look at the following program:

```
/* our function: */
void func(int arg)
{
    int local = 5;
    /* more code here that's not interesting to our example */
}

/* some code that calls our function, we don't care where this
    is */
func(8);
```

Now, let's meet the x86 assembly for this code. Let's first just point out that assembly is a human like representation of machine code. One assembly instruction translates directly to one machine instruction for the CPU.

First let's look at the assembly that corresponds to the call to func:

```
0x08048063:      push   0x8  ; push our argument on the stack
0x08048065:      call   0x8048073 <func> ; this does two things:
                 ; 1: pushes return address onto stack, so our
                 ;    ret instruction knows how to get back
                 ; 2: transfers execution to func() (0x8048073)
```

Notice that addresses to the left are the addresses inside the .text segment that those instructions reside on.

Then, let's now look at the assembly for our function:

```
; func() starts here:
0x08048073:      push    ebp      ; these two instructions are
0x08048074:      mov     ebp,esp  ; used to remember the start
                                  ; of the current stack frame
0x08048076:      push    0x5      ; push the value of our
                                  ; local variable
; more code here that's not interesting to our example
0x08048078:      mov     esp,ebp  ; unwind stack frame (remove our
0x0804807a:      pop     ebp      ; stack frame by pointing esp to
                                  ; what it pointed to before and
                                  ; restore ebp with pop)
0x0804807b:      ret              ; return to the address pushed
                                  ; onto the stack by call
```

The comments should be clear enough that you can see how the stack is manipulated when we call a function.

As always, there's important steps left out to keep the example simpler. But this should be enough to make you understand the stack and its purpose.

Given our knowledge of the stack, we can see how no matter how many functions we call from functions, and so on, creating a very nested path, we are still going to go the same route back as we arrived with, assuming we do indeed return from each function (as opposed to using setjmp() and longjmp() – which we will not introduce; you can read more at http://en.wikipedia.org/wiki/Setjmp.h#Example_usage if you're interested).

6.5 The Heap

A question has probably popped up at this point: how do we create variables on the heap? The answer lies in a function named "malloc", which takes a size in bytes indicating how much memory you want to allocate, and then returns a pointer (or a null pointer on failure) to the allocated memory. The pointer returned is a pointer to void, and how we use and interpret the memory is up to us. Once we're done with the memory, and are no longer in need of it, we can hand it back by using a function named "free". Both of these functions are found in stdlib.h. Let's look at an example:

```
#include <stdlib.h>

int main(void)
{
    void* vp;

    /* allocate 50 bytes */
    vp = malloc(50);
    if (vp)
    {
        /* do something with our 50 newly allocated bytes */
    }

    /* return the 50 bytes (so that they potentially
       can be reused for other purposes) */
```

```
    free(vp);

    return 0;
}
```

We have already seen implicit conversion. For example, when multiplying an integer and a float (i * f), our integer is implicitly converted to a float, and the result too is of type float. There is also a way for us to do explicit conversions. We call this process "casting" to another type. A cast looks like this:

```
(target-type)variable;
```

For example, given an int named i, we could explicitly cast it to a float like this:

```
(float)i;
```

The same is true for pointers. For example, given the return result of malloc, a pointer to void, we could explicitly cast it to a pointer of char, for example. Like this:

```
(char*)vp; /* our pointer to void from before */
```

Explicit conversions can be made in places where implicit conversions will happen, as a form of documentation towards other programmers reading your code. For example:

```
char *p1, *p2;
p1 = malloc(50);         /* void* implicitly converted to char*
    */
p2 = (char*)malloc(50); /* void* explicitly converted to char*
    */
```

As we will also remember, to dereference the pointer (use the memory at the address we're pointing to) we need to have a pointer of a defined type. In other words, our pointer to void cannot be used to manipulate the memory we allocated unless converted to a pointer to a defined type.

Let's write a function that takes a length as its only parameter, and as a return value it will return a pointer to char that will point to a C string that are given the characters of the alphabet. For example, calling our function with the argument 10, we would get a C string that looks like: "ABCDEFGHIJ"

The following program implements this idea:

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

char* alphabet_string(int len);

int main(void)
{
    char* p1;
    char* p2;

    p1 = alphabet_string(10);
    p2 = alphabet_string(40);

    printf("str1: %s\n", p1);
```

```
    printf("str2: %s\n", p2);

    free(p1);
    free(p2);

    return 0;
}

/* returns pointer to allocated memory, freeing the memory is up
     to the caller */
char* alphabet_string(int len)
{
    int i;
    char c;
    char* cp;

    cp = malloc(len + 1); /* +1 for the null character */
    if (!cp)
        return 0; /* memory allocation failed, return null
            pointer */

    c = 'A';
    for (i = 0; i < len; ++i)
    {
        cp[i] = c++;
        if (c > 'Z')
            c = 'A'; /* start from A again if we passed Z */
    }

    cp[len] = '\0'; /* null character */

    return cp;
}
```

6.6 sizeof operator

When we're using malloc, there's a useful operator that comes in play, and that's called the sizeof operator. What it does is simply, at compile time, insert the amount of bytes needed for a variable of the given data type, or occupied by a variable if the symoblic name of a variable was given. It looks like this:

```
sizeof(int); /* type */
/* or: */
sizeof int;
/* or: */
sizeof(i); /* variable */
/* or: */
sizeof i;
/* etc */
```

Where the two variations of syntax above result in the same operation.
This operator is very useful in conjunction with malloc, for example:

```
int* ip;
struct position* pp;
ip = malloc(sizeof(int));
```

```
pp = malloc(sizeof(struct position));
```

Just two quick notes: We cannot use the sizeof operator in control code for the preprocessor. The following is invalid:

```
#if sizeof(int) == 2 /* error */
    ...
#endif
```

And the second note: When using the sizeof operator on the symbolic name of an array (not a pointer, but an array) it yields the size of all the elements in that array. For example:

```
int ia[10];
printf("%d\n", sizeof(ia));
```

Will print 40, assuming sizeof(int)==4. This does not hold true if you give it the symbolic name of a pointer, since that is just a 4 byte (on our 32 bit system) address.

6.7 string.h

Let's take a quick detour and look a bit at the standard library; <string.h> in particular. <string.h> gives us some convenient functions to manipulate C strings. For reasons that should be clear at this point, C strings obviously don't behave as simplisticly as they do in a higher level language, such as a scripting language. And doing things like:

```
"str1" + "str2"
```

Is not possible. After all, what you're trying to do is add two char* together. Instead we have the <string.h> header which are simple functions that allow us to manipulate C strings. These are all functions you can very easily write yourself at this point, but why write them ourselves if the standard library already has them?

Some interesting functions from string.h are:

Team sheet	
strcat(destination, source)	concatenates (appends) source to the end of destination
strncat(destination, source, num)	concatenates (appends) source to the end of destination, with num as the maximum number of characters
strlen(str)	returns length of string (up to but not including the null character)
strcmp(str1, str2)	compares two strings, return 0 if equal, >0 if str1's first different character has a character earlier in the alphabet, and <0 if the opposite is true

To see all functions available in string.h on your system, open a terminal window and type:

```
shiro@debian:~$ man string
```

To see the documentation of a particular function listed in there, type "man <name of function>", like this:

```
shiro@debian:~$ man strcpy
```

This will show you the documentation for "strcpy" as well as "strncpy" as it appears in the Linux Programmer's Manual. For some functions you might even see a sample implementation presented, which can be interesting to look at sometimes.

6.8 Heap part two

Back to the heap. There's another function that's useful if we realize that the allocated memory we have needs to change in size. For example, let's say we have a "message" that's a certain length, and we wish to concatenate another string to that message, but we don't have enough space in our allocated memory. I.e.:

```
#define MEM_ALLOC 20

const char* msg_concat = " append this message"

void f()
{
    char* msg = malloc(MEM_ALLOC);
    if (!msg)
        return;
    fill_msg(msg);

    if (strlen(msg) + strlen(msg_concat) + 1 > MEM_ALLOC)
        /* allocate more memory */

    strcat(msg, msg_concat);
}
```

Then, in the if case where our allocated memory is not sufficient, we need to allocate more memory. We can do this with a function called "realloc". Realloc takes a pointer to the memory we want to reallocate as well as a size of how big we want the new memory block to be. Realloc will either grow in place, or if it can't, allocate a new place on the heap, and copy the content of our memory block there. If realloc fails, it will return a null pointer, otherwise it will return a pointer to the new memory. This is how we would use it:

```
msg_new = realloc(msg, new_size);
if (msg_new)
    /* use memory */
```

If a null pointer is returned, what happens to your memory depends on the C standard you're using:
C90 or earlier: the memory is left as it is, if you do not want it you need to free it.
C99 or later: the memory is freed for you, you can no longer use your old memory.
We're using C90, so for us the memory would still be allocated, and up to us to free.

An important property that we have made use of, but not pointed out is that, as opposed to an array, where the size must be known at compile time, malloc takes a size_t as its size. Let's illustrate this with a simple example:

```
void f(int sz)
{
    char a1[128]; /* ok: size known at compile time */
    char a2[sz];  /* error: size not known at compile time */
    char *p1, *p2;

    p1 = malloc(128); /* ok */
    p2 = malloc(sz);  /* ok */
}
```

And that concludes the introduction of the heap.

6.9 Homework

In this homework section we will implement our own string library. The most important property of our string library is that it will give us a type that can grow in size when more memory is needed to represent our string, without us as users of the library having to worry about it. So if you concatenate to a string that's too small, your string will automatically expand for you.

I have provided the header file, your job is to implemenet the source file.

```
/*
 * file:                shirostr.h
 * comment:             the shiro string library
 * prefix on functions: ss_
 * extra note:
 *     implementations of all functions should be in
 *     shirostr.c, which is for you to provide!
 */

#ifndef SHIRO_STR_H
#define SHIRO_STR_H

#include <stddef.h>

#define GROWTH_FACTOR /* TODO: figure out how much
                         you want your string to grow,
                         try to keep a balance that assures
                         not too many reallocs, but not
                         too much wasted memory */

struct sstr { /* TODO: you fill this in! */ };

/*
 * creating and destroying:
 */

/* empty shiro string */
struct sstr* ss_alloc();
/* shiro string from c string */
struct sstr* ss_alloc_cstring(const char*);
/* allocate a new shiro str that is a copy of
   the parameter */
struct sstr* ss_alloc_ss(const struct sstr*);
/* free shiro string */
void ss_free(struct sstr*);
```

```
/*
 * everything else:
 */

/* Note: most functions here return an int, to indicate
   success or error. See the test code for usage. */

/* intuitively: shiro_str = c_string */
int ss_set_cstring(struct sstr*, const char*);

/* intuitively: shiro_str = shiro_str */
int ss_set(struct sstr*, const struct sstr*);

/* intuitively: shiro_str += shiro_str */
int ss_concat(struct sstr*, const struct sstr*);

/* intuitively: shiro_str += c_string */
int ss_concat_cstring(struct sstr*, const char*);

/* should work the same as strcmp for c strings */
int ss_cmp(const struct sstr*, const struct sstr*);

/* deletes from pos to pos+len, shifting what's behind
   pos+len (the tail) down to pos */
int ss_delete(struct sstr*, size_t pos, size_t len);

/* get a pointer to the raw c string */
const char* ss_cstring(const struct sstr*);

/* get the length of the string */
size_t ss_strlen(const struct sstr*);

#endif
```

Here's a program to help you test your library. Make sure it works as it should before you submit the code!

```
#include "shirostr.h"
#include <stdio.h>

int main(void)
{
    struct sstr *a;
    struct sstr *b;

    a = ss_alloc_cstring("The shiro string");

    b = ss_alloc();
    ss_set_cstring(b, " works!");

    ss_concat_cstring(a, " library");
    ss_concat(a, b);
    ss_delete(a, 4, 6);

    printf("%s\n", ss_cstring(a));
    return 0;
}
```

7 Lesson Six

This lesson is going to be very short. I will present three new concepts, as well as discuss the last homework. The reason it's short is because we're going to give you a bunch of stuff to code, but not introduce that many new things.

7.1 Data structures

The string library we wrote in last chapter was what we call a data structure. A data structure is, in general, a method or idea to store and orginize data so it can be used efficiently. Our basic arrays are even a type of data structure. Our string library that we created was a *dynamic array*, in other words an array that is able to grow in size on a per need basis.

7.2 Big O Notation

Big O Notation is a concept that we commonly use to describe the performance aspect of an algorithm. The best way to understand big O notation is to see it used. So in this chapter we will discuss big O for our different algorithms. The definition of it, and the most important thing to keep in mind is:

Big O notation describes how an algorithm changes (usually in terms of speed or memory requirement) based on the size of its input.

Let's say we have the following algorithm, which searches a C string for a given character, and returns a pointer to the first occurence of that character.

```
char* find(const char* s, char c)
{
    for ( ; *s; ++s)
        if (*s == c)
            return s;
    return 0;
}
```

Let's try to describe the speed aspect of our find algorithm using big O.

We can already see that if our sought character is not present, our algorithm will go through all available characters. If the size of our input is denoted as n, then the big O would be:

Worst Case: O(n)

O(n) is simply us saying that the worst case performance is equal to the size of the input.

Our best case is finding the character right away, which we denote as:

Best Case: O(1)

Which means the best case speed for our algorithm is irrelevant to the size of the input.

Big O gives us the upper bound, so we say that the average complexity is O(n) as well. Unless the worst case describes some extremity that can happen in theory, but likely won't in practice, average and worst is the same.

Average Case: O(n)

Our function is always O(1) for memory space requirement, as it does not use any more memory when searching a ten elements C string, or a ten thousand elements C string.

7.3 Dynamic Array

The C library we created was a dynamic array. Obviously we can make dynamic arrays that store any type of memory. We could make a general dynamic array structure that stores pointers to void, like this:

```
struct dyn_array
{
    void** array; /* array[0] ... array[size-1] gives us
                     the pointer to void at that location
                     in our dynamic array */
    size_t size;
};
```

There's two potential performance problems with a dyanmic array.

One: If we need to grow, and there's not enough space for realloc to grow in place, the entire memory needs to be copied to a new location. For a large amount of elements a lot of memory might need copying.

Two: When deleting an element, all elements after it needs to be moved so that there's no gaps in the array. If a deletion happens towards the end, this is a non-issue no matter the size, but if a lot of elements follow the deleted one, a lot of memory needs to be shifted.

Especially two can be problematic, and if we're dealing with big amounts of data and a lot of deletions, then an array is probably not the data structure we want.

We're going to meet what is called a linked list soon. It's likely you will initially think that it solves both problems, and that it has to always be superior to a dynamic array. But the truth is, for reasons we will go into later, that an array usually outperforms a list, unless the data set is substantial and we delete a lot. This is due to how our CPU works. But more about that later.

One final note about performance: When judging what data structure you want for a specific case, you should always test the difference in performance, rather than assume it.

7.4 Linked List

Unlike an array that needs to appear sequential in memory, a linked list can have its elements scattered about. The linked list data structure instead solves the problem of knowing order by having a pointer associated with each element that points to the next element. We call this collection of data and a pointer a "node" in the linked list. Together the nodes make up a form of chain, and that's why we call the list a *linked* list.

The data it manages is irrelevant to the linked list, but let's say we store pointers to void. Then a node structure could be defined like this:

```
struct node
{
    struct node* next; /* next node in the list */

    void* data;
};
```

The first node of the list points to the second, which points to the third, and so on. All the way until the end, where the last node points to nothing (next is a null pointer).

Let's visualize a linked list:

http://i.imgur.com/EN8ssc5.png

To use a linked list, all we need to remember is the first node of the list. The first node is commonly called the "head". We always use the next pointer to traverse the list, which could be done like this:

```
/* node* head points to the first element in our list */
node* p = head;
do
{
    /* use data */
} while ((p = p->next) != 0);
```

7.4.1 Deleting from a list

Deleting from a list is not too hard, either. If there's only one node, the head, then the list is gone. We simply free the memory and we're done.

Otherwise we traverse (go through) the list until we find our node again. While doing this we will have saved a pointer to each previous node, so that when we reach the current, we have a pointer to the node before it. Then we just repoint the previous node's next to our next (which is 0 if we're the tail):

```
/* node* current: we wish to delete this one
   node* prev: this is the one before us */
prev->next = current->next;
```

And then we free the resources.

7.4.2 Doubly linked list

A problem with the linked list is that when we want to delete a node, we need to traverse the entire list to find the node just before us. We can solve that by having a list that's doubly linked, which means that we have a pointer both to the next and the previous element, and traversing can be done in either direction. A node now looks like this:

```
struct node
{
    struct node* next;
    struct node* prev;

    void* data;
};
```

Let's look at an image that visualizes this:
http://i.imgur.com/T89sdih.png
To remove a node we simply connect the node before and after (assuming those do indeed exist) and free our resources.

Since we can traverse in both directions, it would make sense that our list both holds a pointer to the first and the last element. So our doubly linked structure would look like this:

```
struct dbl_link_list
{
    struct node* head; /* first */
    struct node* tail; /* last */
    size_t size; /* might be an interesting attribute,
                    but not necessary of course */
};
```

Your first homework this lesson will be to implement a doubly linked list.

7.5 Big O

Searching a linked list too is O(n) complexity, as you'll have to start in one end and traverse until you find it.

Deleting from it is O(1) given a doubly linked list, and O(n) a singly linked one, assuming you have a pointer to the node you wish to delete.

Deletion if you only have a value is O(n) for both type of lists (again traversing the list from head to tail).

Deletion if you have a pointer to the element before the one you wish to delete is O(1) for both lists.

Insertion at the beginning is O(1) for both lists. At the end it's O(1) for the doubly linked, and O(n), assuming you do indeed *not* keep a tail pointer, for the singly linked list.

A conclussion we could draw, is that given a set of data which we intend to delete frequently from, a linked list might outperform a dynamic array.

7.6 Binary Search Tree

The main problem of both a list and an array (or dynamic array) is if we frequently need to locate a single element based on a property of that element (i.e. for arrays we don't know the index).

Let's look at a case where this might happen. Let's say we have been presented the task of making a phone book with the number and address of *everyone in the United States*, in other words a *huge* set.

The first thing we'd do is design a structure with the relevant information:

```
struct person
{
    /* size of the arrays could be argued */
    char name[128];
    char address[256];
    char phonenr[32];
```

```
    char ssn[16];
}
```

Let's assume we stored this in a linked list. We're then tasked with finding the phone number given a person's name. We'd do something like this:

```
/* find person with name in list */
struct person* find_name(struct dbl_linked_list* lst, const char
    * name)
{
    struct node* n;
    struct person* p;

    for (n = lst->head; n != 0; n = n->next)
    {
        p = n->data; /* cast from void* to struct person* */
        if (strcmp(p->name, name) == 0)
            return p; /* ignore potential name collisions */
    }

    return 0;
}

/* find phone number matching name */
const char* phone_nr(struct dbl_linked_list* lst, const char*
    name)
{
    struct person* p;
    if ((p = find_name(lst, name)) != 0)
        return p->phonenr;
    return 0;
}
```

As we can see, our algorithm to find the pointer to struct person with the given name, is a loop that potentially goes through the entire set of our data, in other words the big O is:

Average & Worst Case: O(n)

And the problem is quite clear when we realize that n (the algorithm's input) is the amount of people in the *entire* US. In other words, there's more than 300 million entries in our list.

Given a program that fetches data based on a person's name, you can see that this will be *extremely inefficient*. On average, we'll have to go through over 150 million entries per query we want to do.

7.6.1 A tree structure

One way to solve this problem, is to look for another data structure. One in particular we'll discuss here (not necessarily the best one) is called a binary search tree. It's called a tree, because the structure looks like a tree. If we stored 0 through 9 as a binary tree, it'd look something like this:

http://i.imgur.com/Kz7nkms.png

Hopefully you see the resemblance to a tree here. The top is what we call the root, and from the root the tree branches in two directions. And from each node

two new branches are born. This means that one node has *one* "parent" and *two* "children".

As you probably noticed, the element in the middle of the set (0-9) is placed at the root. And for all nodes it's true that the right child has a bigger value, and the left child a smaller.

Given this knowledge, we see how traversing to 7, for example, has us starting at the root and realizing 5 is less, which leads us to the right child. Then when we compare 8, which is bigger than 7, we go to the left child, arriving at 7.

The time complexity for searching a binary search tree is, on average:

Average Case: O(log n)

Which is *a lot* quicker than O(n). You might want to graph n and log n on a graph calculator just to see how big the difference really is.

7.6.2 A node in our tree

We know that each node has a right and left child and a parent. We also know that the node has a value we can compare for (integers in our previous case), we commonly refer to this value as the key.

Now, in our last example the key was the only thing present in the list, but more often than not when we're looking at a structure like this, it's more useful to have a key and some underlying data.

For example, in our phone book program we searched on the name of a person, and returned a pointer to struct person. And similarly here, it seems like the name of a person would be a good key. And the data we store (associate to that name), would be a pointer to the struct person associated to that name. (Again we don't worry about overlapping names for now.)

A node in our tree might be defined as:

```
struct node
{
    char key[128]; /* the name of the person is our key */
    struct person* value; /* pointer to person struct */

    struct node* left_child; /* if no left child this is NULL
        (0) */
    struct node* right_child; /* if no right child this is NULL
        (0) */
};
```

7.6.3 Searching

Searching a tree is very simple and explained previously. The code to do it would be something like:

```
struct person* find(struct node* n, const char* name)
{
    int cmp = strcmp(n->key, name);
    if (n == 0 || cmp == 0)
        return n->value; /* either key is equal or NULL */
    else if (cmp < 0)
```

```
        return find(n->left, name); /* go left */
    else
        return find(n->right, name); /* go right */
}
```

Which should be simple and straightforward at this point.

7.6.4 Insertion

Inserting is just the same as searching. We search until we find a node with a null pointer as child, and insert at that point. If you reach a key that matches yours, then you'll get some extra complexity, but nothing that's too difficult. For simplicity, we have left out this case, and when you implement the structure for the homework, you can just leave this case out (unless you want an extra challenge, of course).

7.6.5 Deletion, traversal and destruction

Deleting a node in the tree, traversing the entire tree or indeed freeing all allocated resources (destroying the tree) are fairly complex operations. And since all we really wanted to show was the basic tree structure, you will not be required to add these operations to your binary tree (for a real tree you want to use in production code these are obviously necessary operations). If you feel you want an extra challenge, you're free to try your hand at these operations, but it's not something we're expecting you to do.

7.6.6 Self-balancing binary search tree

The main problem with a binary search tree is that the order in which elements are inserted, defines the search complexity. For example, if we inserted our 0-9 starting at 0, then 1, and so on, our resulting binary search tree would look like this:

http://i.imgur.com/p5vSxd3.png

And as you can see, this is just a fancy linked list, as all children are to the right. Searching has now reached the worst case for a binary search tree:

Worst Case: O(n)

There are data structures that have the attribute that they're *self-balancing*. That means whenever you insert an element, the structure will use algorithms to try to efficiently reorder itself to keep the O(log n) search speed complexity.

These are fairly complex constructs, and we won't be looking at them. However, if you're interested, one interesting such structure is called a red black tree, and you can read more about it here: http://en.wikipedia.org/wiki/Red%E2%80%93black_tree.

7.7 Homework

Your homework is to implement the doubly linked list described in this chapter, as well as the binary search tree (quite a bit of the functionality you might expect for this structure you do not need to implement; check the lesson itself).

8 Lesson Seven

8.1 Pre-lesson

We finished the K&R book leading up to this lesson in class. The sections that were mandatory were:

1. 6.7, 6.8, 6.9
2. 7.1, 7.2, 7.3, 7.4
3. 8.1, 8.2, 8.3, 8.4

Any other section was left as optional reading.

8.2 From here on

We have learned the language C. While we could not yet call us proficient C programmers, we know enough to set out and make useful applications.

But more than anything, we've learned about computers. We have studied how they work, and our tool for learning about them at a low level have been C. C has been our stepping block, and it's now time to take a step to a higher level. It's time to look at C++.

Important to keep in mind from here on out, is that language that have high level features, must somehow implement them in terms of the low levels we've already studied. All of these features we will see are abstractions to make it easier for the programmers, and many of them come with a cost – a cost that it's vital that we as programmers know and understand, so that we can make educated decisions.

Far too many programmers that write code in high level languages have no idea about the lower level, and simply write code as they are capable in their simplified and abstract world. This, no surprisingly, is a recipe for utter disaster. Only a programmer who truly understands what happens at the lowest level when he writes his code, can ever be a programmer that consistently makes acceptable decisions.

A programmer exposed to these powerful features, but without the actual understanding, will tend to write both inefficient and poor code. Do not make that mistake – make sure you *only* use features you understand truly and fully.

Now, C++ is a good continuation language, because most that we've learned in C transitions over to C++. Granted, there are detail differences between the languages; but the attained knowledge we have remains the same.

With all this in mind, we will make sure to always explain new features in terms of a lower level, so that you can fully understand the abstractions and become a decent programmer.

8.3 Compiling with C++

In C++ there's a bunch of valid file extensions we can use, but in general our .c files should now be called .cpp, while our .h files remain as such. To compile a C++ file we'd do something like this:

1. Create a file named "test.cpp".
2. Open a terminal and navigate to the file.
3. Type: "g++ -std=c++11 test.cpp", to compile your file with gcc using the C++11 standard.

8.4 C++ and C; differences

C++ not too long ago released a new version, called C++11. This version was the most significant upgrade to the language in over a decade. It adds many conveniences that have been found in other languages, but still yet missing in C++.

There are important differences between C and C++, some of which we will introduce right away, some of which we will introduce as we go.

8.4.1 Some quick differences

C++ comes with its own standard library, but you can still use the headers we familiarized ourselves with in C, the way you include them has just changed slightly:

```
#include <cstdio>
```

All headers have had a c prepended, and the ".h." part has been removed, so stdio.h becomes cstdio, string.h becomes cstring, and so on.

There's also a new type of comments:

```
// C++ style comment: comments one line
/* C style comment */
```

In C we specify void as a parameter list if our function intends to take no arguments; as the following code is valid in C:

```
void f();
f(10); /* valid in C */
```

However, it's not valid in C++, so in C++ we leave out void and just leave our parameter list empty.

```
void f();
f(10); // error in C++
```

Another trivial change is that when we make an instance of a struct, we do not need to restate the "struct" part of the name. In other words:

```
struct my_data { /* ... */ };
/* C way of creating an instance of my_data: */
struct my_data instance;
// C++ way of creating an instance of my_data:
my_data instance;
```

8.4.2 Classes and objects

There are two keywords to define new types in C++. One is struct, whose keyword is carried over from C. The other is class, which we will meet soon. For now just think of class and struct as being synonyms.

When we encounter the declaration and definition of a class (or struct), we say that provides the blueprint, if you may. When we make a variable of that type, we say we create an object of our type; an instantiation.

It is important to realize that the class does *not* set aside any memory for its data members, it simply serves as a description. However, an object of that class *does* set aside the memory needed.

If we should make a real world example, we might say that the class is the blueprints of an object, let's say a car. The blueprints does not a car make, but from the blueprints we can build a car. And just like that, from the class we build an object (an instance).

Let's see the difference in code:

```
struct car
{
    int horse_power;
    int cylinders;
    // ...
};

car ferrari;
car mercedes;
```

In the above example, the struct car is the blueprint of any car objects. Ferrari and Mercedes are objects of this class.

8.5 Member functions: methods

In C when we're building a library, such as the list or string libraries we made, we're likely to make functions that operate on the structure(s) that our library uses — such as the dbl_linked_list structure. Such functions commonly take a pointer to an instance of the structure it operates on. In other words, something like this:

```
struct player
{
    /* array of equipped items */
    struct item* equipment[MAX_ITEMS];
};

void player_equip(struct player* p, struct item* i)
{
    int s = i->equipment_slot();

    if (p->equipment[s])
        unequip_slot(p, s);

    p->equipment[s] = i;
}
```

In C++ we have been given the convenience of what we call member functions; or methods. Methods allow us to put a function declaration inside of a structure, and have it be part of that structure's scope. This both means that it cannot clash with names outside of the structure – it's local to its scope, and it also allows us to use the syntax we use in C for member variables, but for member functions as well. In other words:

```
struct my_type
{
    void func();
};

// this is NOT my_type's func,
// nor does it give a name collision
void func()
{
    my_type t, *pt;
    pt = &t;
    t.func();
    pt->func();
}

// this is the implementation of my_type's func()
void my_type::func() // notice the my_type::,
                     // which specify that the scope is my_type
{
}
```

When we call a method, an implicit argument is passed into the function. This argument is called the *this pointer*. It's a pointer that points to the structure we called the function from. In comparison to C where we usually make the first parameter of the function be a pointer to the structure, a method in C++ has that parameter made implicit. This implicit pointer can be referred to by typing "this" inside of a method. Let's look at an exmaple:

```
struct my_c_struct { int i; };
/* In C we don't have methods,
   but instead pass a pointer along */
void operation(struct my_c_struct* inst, int i)
{
    inst->i += i;
}

// In C++:
struct my_cpp_struct
{
    int i;
    void operation(int i);
}
void my_cpp_struct::operation(int i)
{
    this->i += i;
}
```

If we are referring to a member inside of our structure, we actually do not need to specify the this pointer; as it will be implicitly assumed, unless it's hidden by a

local name. Namely:

```
struct data
{
    int my_int;
    float my_float;
    void method();
};

void data::method()
{
    float my_float;       // hides data::my_float
    my_int = 10;          // changes data::my_int
    my_float = 5;         // changes our local my_float;
                          // because of the hiding
    this->my_float = 15; // changes data::my_float
}
```

Whether you use this or rely on the implicity when no hiding is going on is up to you, although from experience I can say that code is generally more readable when *not* using this explicitly.

Let's look at our player equipment example from before, but instead with methods.

```
struct player
{
    // array of equipped items
    struct item* equipment[MAX_ITEMS];
    // method to equip items
    void equip(struct item* i);
};

void player::equip(struct item* i)
{
    int s = i->equipment_slot();

    if (equipment[s])
        unequip_slot(s);

    equipment[s] = i;
}
```

As we mentioned before, the compiler implements this feature of methods by passing the address of the object you called the method on as an implicit parameter to the function. The compiler then assumes the implicit dereference of the this pointer if the name can be found inside the scope of your structure. There's nothing special going on. All you get is syntactic sugar that makes it easier for you to work with structures and member functions, partially by a cleaner and more intuitive syntax, but even more so by avoiding name collisions.

8.6 public and private

In C++ we can assign access control to members (both variables and functions), where we can restrict who can change or use them. Two of these are *public* and *private*. By default anything that goes into a structure has public accessability,

which means both methods and code outside the scope of your class can change or use the members. If we specify private, however, only methods local to our structure's scope can manipulate or use the member. We specify access inside a structure like this:

```
struct access_controlled
{
    int g;       // g is public
private: // Anything below this point will be
         // private (until another modifier is encountered)
    int i;       // is is private
    void f();    // f is private
public:
    int a;       // a is public
    void b();    // b is public
private:
    int j;       // j is private
};

void use()
{
    access_controlled a;
    a.g = 10; // ok: public
    a.b();    // ok: public
    a.i = 5;  // error: private
    a.f();    // error: private
}
```

C++ introduces another keyword for declaring new types — besides struct — that is called *class*. There is only *one* difference between a struct and a class — namely, a struct is by default public (i.e., the g in our previous example was public), whereas a class is by default private (replace struct with class in our previous example, and g would be private).

There is no actual difference besides this. However, programmers generally do associate a difference, and say that class is to be used for any type that implements an abstraction, whereas struct should be used only for "raw" bundles of data — where no abstracting interface is applied. This is not a difference enforced by the language, and far from every programmer even uses it. We do recommend the distinction though. Exactly what abstraction means and how we implement it will be seen later.

8.7 Constructors and destructors

When we make a library in C it's common we have functions that fill in our structure, and allocate the necessary resources. It's also common we have a function that frees our resources. Something like this:

```
struct str
{
    /* ... */
};

struct str* alloc_str(void);
void free_str(struct str* s);
```

In fact it's so common that C++ made it part of every class or structure (henceforth only referred to as class). There's special functions for this purpose. There's one type of function that's called when an instance of your class is created, called a *constructor*, and there's one called when your instance comes to an end (e.g. by going out of scope), called a *destructor*.

Let's look at an example that uses the constructor and destructor:

```
#include <cstdio>

class test
{
public: // try what happens if this is private!
    test() { printf("Inside of test's constructor!\n"); }
    ~test() { printf("Inside of test's destructor!\n"); }
};

int main()
{
    test t;
    return 0;
}
```

C++ has its own library for doing I/O (input/output), but we have not yet learned enough to use it, so we're sticking to printf for now.

As we can see the constructor is a function with the same name as the class, and a destructor is the function with the same name as the class, no parameters and a ~sign infront.

A constructor can also take parameters, for example:

```
class abc
{
public:
    abc(int i);
};

abc::abc(int i) { printf("abc::abc: i = %d\n", i); }

int main()
{
    abc a(10);
    return 0;
}
```

8.8 Default constructor & destructor

If we do not provide a constructor or destructor, we get default versions of those. They are equivalent to parameterless and body-less versions that we would write ourselves. If we provide a constructor with a parameter list, our default constructor is deleted; which means that we can no longer construct an object with no argument list, unless we define our own default constructor. Example:

```
class no_def
{
public:
```

```
    no_def(int i);
};

class def
{
};

class explicit_def
{
public:
    explicit_def() { }
    explicit_def(int i);
};

int main()
{
    no_def        a;     // error
    no_def        b(10); // ok
    def           c;     // ok
    explicit_def  d;     // ok
    explicit_def  e(5);  // ok
    return 0;
}
```

8.9 C++ overloading

In C++ we have a concept called overloading. In C when we needed a function to do the same functionality, but take different parameters, we needed to use different names for the functions. In C++ we have the option to *overload* it, in other words use the same name but with a different set of parameters. We will see an example of overloading used in conjunction with constructors in a later code listing.

8.10 Namespaces

There's a feature added in C++ called namespaces. This allows you to open a scope to put functions, data, classes etc inside, to avoid name collisions. There's a lot to be said about namespaces for the future, but for now it's sufficient to realize that they prevent name clashes. Let's look at a quick example:

```
namespace shiro
{
class my_class
{
    /* ... */
};
};

namespace bab
{
class my_class
{
    /* ... */
};
```

```
int main()
{
    shiro::my_class a; // create instance of shiro's my_class
    bab::my_class b;   // create instance of bab's my_class
    return 0;
}
};
```

In the above examples, the two classes with the same name describe types unrelated to one another, with no name clashes happening; that is because they're encapsulated in a namespace and their fully qualified name is namespace::class_name, e.g. shiro::my_class.

There's a lot more to be said about namespaces, which is probably one of the most important features in C++ for big projects. There is one special namespace called "std", inside of this namespace all of C++'s standard library resides. The next code listing will show an example of this.

8.11 C++ string library

Remember the string library we wrote in C? C++ comes with a string library as part of the standard libray. It's also, not surprisingly, more feature complete than the one we wrote. You can find it in the <string> header. Below is a listing of some (not all) overloaded constructors we can use:

```
namespace std
{
class string
{
public:
    // construct empty string
    string();
    // construct from C string
    string(const char* s);
    // construct by filling with n amount of c characters
    string(size_t n, char c);
    /* ... */
}
}
```

Let's construct some strings and print their content:

```
#include <string> // C++'s string library
#include <cstdio>

int main()
{
    // "std" is a namespace which string resides in
    std::string s1; // empty
    std::string s2("hello there"); // from C string
    std::string s3(10, 'A'); // fill with 10x'A'

    printf("s1 = %s\n", s1.c_str());
    printf("s2 = %s\n", s2.c_str());
    printf("s3 = %s\n", s3.c_str());
```

```
    return 0;
}
```

To print a C++ string using the %s format specifier we have to use the method string::c_str() which returns a C string representation of the string object.

8.12 new and delete

You might have already figured out that malloc and free will not work very well with C++; after all when you allocate raw bytes, no constructor will be called for your type. C++ introduces two new keywords called *new* and *delete*, which is the counterpart of C's malloc and free. In other words they allocate enough memory on the heap to hold an instance of your type. They also assure that your constructor and destructor is called appropriately. Let's see an example of how we use them:

```
class my_type
{
    my_type(const char* s);
};

void f()
{
    my_type* tp = new my_type("hello");
    /* ... */
    delete tp;
}
```

If we're dealing with allocating more than one instance, we need to use a slightly different syntax:

```
void f()
{
    char* heap_arr = new char[128]; // 128 char elements
    /* ... */
    delete[] heap_arr;
}
```

8.13 References

Pointers — variables that hold the address of another variable — is a construct that tightly relates to what happens at the CPU level. References, however, is an abstraction made to solve some common problems with pointers in C.

Let's first look at the high level aspect of references; in other words what abstraction do they provide us with. Let's begin that discussion with two common problems with pointers and functions:

```
void func(int* a)
{
    // Problem one: Are we sure "a" points to something valid
    // and isn't a null pointer, for example?
    if (!a)
        return;
```

```
    // Special syntax; as opposed to standard variables
    // If we have the following line:
    *a = 10;
    // And we'd like to change to use pass by value,
    // we'd need to rewrite the code that deals
    // with our pointer, e.g.:
    a = 10; // if a was just "int a"
}
```

These two problems are solved by what we call a reference. From a high-level point of view references are just a *synonym* for a variable. This synonym can be used to change the original variable without the usage of any special syntax. A reference is not a pointer — it's not an object of its own; it's just a synonym for another variable.

A reference is declared by specifying the type followed by an ampersand; do not confuse this ampersand with the address-of operator — they are not the same thing.

```
int i;
int& ref = i; // ref is a synonym for i
int& err_ref; // error: a reference is not its own object, we
              // must refer it to something when we create it
ref = 10;     // i is now equal to 10
```

As you can see, with references we can use the same syntax as with variables, since it's just a synonym. We also cannot have "null references", which means we need not check validity before using a reference. Let's see how that changes our function from before:

```
void func(int& a)
{
    a = 10;
}
```

8.13.1 Reference to const

Just as we can make pointers to const, we can make references to const. Let's see an example that illustrates this:

```
int i;
const int j = 15;
const int& r1 = i; // ok: i cannot be manipulated through r1
const int& r2 = j; // ok
int& r3 = i;       // ok
int& r4 = j;       // error: j is const
```

As we noted in C; when we pass objects with significant amounts of data, we tend to pass a pointer, as to not have to copy the object around. The same can be done with references. And just as we could with pointers, we can make const references — in other words references to variables we do not intend to change. Let's see an example:

```
class large_object { /* ... */ };
void important_procedure(const large_object& ref)
```

```
{
    // ....
}
```

A temporary variable — a variable that's the result of an expression, for example, cannot be referenced to, *unless* with a const reference. For example:

```
int i = 42;
int& r1 = 10;           // error: non-const reference
const int& r2 = 5 * i; // ok
const int& r3 = 10;     // ok
```

This becomes particularly useful for functions that take const references. For example, let's imagine the following scenario:

```
void f1(std::string& ref);
void f2(const std::string& ref);

void f()
{
    f1("hello"); // error: cannot bind reference
                 //        to temporary object
    f2("hello"); // ok: we can with a reference
                 //     to const, though!
}
```

8.13.2 Low-level

As we said earlier, we should always be aware of the underlying workings of an abstraction. So, let's take a moment to consider how the compiler might implement references — after all, we know that they're not a concept at the hardware level.

```
void f2(int ref&);

void f()
{
    int i = 10;
    // When f() is entered, we know that 10 is pushed
    // onto the top of the stack. "i" is our symbolic
    // name to refer to this place on the stack; the
    // compiler knows the difference from the top of
    // the stack to our location of i, and can
    // substract the difference from the stack
    // pointer (as we will remember is the top of our
    // stack) to read/write to i.

    f2(i);
    // The only way for us to access i is to know its
    // place on the stack. So, when calling another
    // function and passing it a reference to i, we
    // can see how it must be the case that, on the
    // lower level, an address is actually passed.
    // In other words, our reference parameter is
    // implemented as a pointer.
}
```

Apart from being a useful reasoning to arrive at, we can see that passing const references to built-in types (int, float, etc) in the purpose of avoiding to make a copy to, pressumably speed things up, is actually likely to speed it down — after all a pointer needs to be passed, and access happens through that pointer, which is going to be slower than copying the value to our new function's stack frame.

In other words:

```
// bad idea: pass by value instead of reference
// to const for built-in types
void f(const int& ref);
```

8.14 Summary

This is our crash-course in C++. It builds on the knowledge we have about C, and allows us to quickly get some familiarity with C++, so that we soon can move to more interesting stuff.

9 Lesson Eight

9.1 Our Goal

Our goal this lesson is to teach you enough about what is called Object Oriented Programming (OOP) so that we can look at some of MaNGOS code and try to play around with that code base some.

9.2 OOP

Object Oriented Programming (OOP) is a programming paradigm. As the name suggests, it makes us think in terms of Objects, with the intent to make modular (stands on its own) and reusable code. This dividing into objects is supposed to make code easier to understand, maintain and develop further.

Last lesson we described the difference between a class and an object, where a class implements the interface (blueprint) of an idea, and the objects are the instantiation of these classes. This is an important difference to keep in mind.

9.3 Composition

Our classes can be composed of objects of other classes. If we think of a concrete example, then we could say that a car is composed of many other objects — for example the engine or the gas tank. Let's look at how this might be turned into code:

```
class engine
{
public:
    void start();
    void stop();
    // Other stuff...
private:
    int cylinders;
    // ...
};

class gas_tank
{
public:
    // interface
private:
    // private data
};

class car
{
public:
    // interface
    void method();
private:
    // A car is composed of an engine and gas tank...
    engine engine_;
    gas_tank tank_;
```

```
    // ...and other stuf...
};

void car::method()
{
    // How to use our instance of the engine class:
    engine_.start();
}
```

Composition is one of the most useful tools in OOP, and allows for great potential for modularity. After all, if our engine class needs to change internally, our car class doesn't really need to care, as long as the interface to the engine class remains the same (the public methods). This allows us to split up objects into components, that each function on their own — allowing them to both be reused in other code, and to be able to change without affecting classes that use them (or that's the idea, anyway).

9.3.1 Low-level

From a low-level approach this is pretty straight-forward. All that happens is that when we construct our object, all our composed objects are constructed as well. So the total amount of memory that's set aside when an instance of our class is created is the sum of the memory our member variables make up, as well as the memory of all our composed classes' member variables.

9.4 Inheritance

Another way to reuse code is a concept called inheritance. Again, using a concrete example: An apple is a fruit, a banana is a fruit, a mango is a fruit, and so on. They all share a "is a" relationship, with common attributes. In other words, they all *inherit* the properties of being a fruit.

Inheritance allows us to say that a class *is a* subset of another class. Another concrete example: A ferrari is a car, so a ferrari might inherit from a car class, and likewise a car is a vehicle so a car might inheirt from a vehicle class. A bus, too, might inherit from a vehicle.

In C++ there are three different types of inheritance, that relates back to accessability that we saw in last lesson. The most common one is public inheritance, which we will see now. The other two forms we will see later in this lesson.

```
class bird
{
    /* ... */
};

// crow inherits from bird (public inheritance)
class crow : public bird
{
    /* ... */
};

// sparrow inherits from bird (public inheritance)
```

```
class sparrow : public bird
{
    /* ... */
};

// rock_sparrow inherits from sparrow (public),
// which inherits from bird
class rock_sparrow : public sparrow
{
    /* ... */
};
```

In the example above both crow and sparrow are birds. rock_sparrow is a sparrow, which is a bird. So rock_sparrow is both a sparrow and a bird.

Click at the following link which is an image of a UML Diagram, showing the relation between the classes: http://i.imgur.com/LppvHhm.png. The arrows point from a class to another, specifying that the pointer inherits from the pointee. As the diagram shows; rock_sparrow inherits from sparrow, whom in turn inherits from bird.

UML Diagrams are useful for visualizing an OOP idea, and we will use it throughout the remaining parts of the course. The diagram was drawn with UMLet, which is an open-source GNU licensed UML drawing tool that's very simple to use. Here's a link: http://www.umlet.com/. I recommend you to download that and use it to document any OOP designs you implement, to make maintaining and evolving the designs easier.

9.5 Pointers

Since inheritance specify a "is a" relationship, a pointer of a class higher up in the hiarchy can be made to point to a class inheriting from it. Such as:

```
class A {};
class B : public A {};
class C : public A {};

void f()
{
    B b;
    A* ap = b;  // ok: B is an A
    C* cp = b;  // error: B is not a C
    B* bp = ap; // error: A is not necessarily a B
    bp = (B*)ap; // ok: we suppress the error by telling
                 // the compiler we know what we're doing
                 // (See note below code snippet)
}
```

Notice that the above is the type of cast we learned in C. This is a possible cast in C++ as well, but it's one that is best avoided — we will meet a C++-alternative later on.

9.6 Low-level

At the low level, the same as what happens with composition happens here. All the data of the classes you inherit from are part of your data as well. So if we added 4 bytes of data to the bird class and 8 to the sparrow, but no to the rock_sparrow, then a rock_sparrow instance would still be 12 bytes wide.

9.7 Composition vs inheritance

To keep composition and inheritance seperate, remember the following:

Composition implements the idea that an object "has a <something>". For example, a car has a wheel.

Inheritance implement the idea that an object "is a <something>". For example, an apple is a fruit.

It is easy to go overboard with inheritance as a programmer new to OOP, after all a lot of things can be said to be something. In general, we tend to say that it's better to prefer composition over inheritance. This is mainly because code tends to be easier to alter later down the road if you don't overuse inheritance. That being said, there are applications when inheritance is very useful, as we will see shortly.

9.8 Protected

There's another accessability modifier besides public and private. Its purpose is to moderate what can be shared from a *parent class* to a *child class* (terminology commonly used to specify one class inheriting from another). It's called protected. The rules are as follows:

1. public: anyone can access this
2. private: no one but myself can access this
3. protected: I, and classes deriving from me, can access this.

Protected is, as you can see, private but with an exception for classes that derive from ours. Let's look at an example:

```
class A
{
public:
    int pub;
protected:
    int prot;
private:
    int priv;
};
class B : public A
{
public:
    void method()
    {
        pub = 10; // ok: modifies A::pub
```

```
        prot = 5; // ok: modifies A::prot
        priv = 1; // error: cannot modify A::priv
    }
};

int main()
{
    A a;
    a.pub = 10; // ok: public
    a.prot = 5; // error: protected
    a.priv = 1; // error: private
}
```

These access restrictions apply for member variables and member functions (methods) alike.

9.8.1 Low-level

All of these restrictions are verified on compile time, and the resulting binary will not be affected in speed nor size due to these access restrictions.

9.9 Different types of inheritance

As we mentioned before there's three types of inheritance. We can either do a public, private or protected derivation (the action of inheriting from another class can also be referred to as *deriving* from it).

The difference is most easily showcased with a code example:

```
class A
{
public:
    int a;
private:
    int b;
protected:
    int c;
};

class B : public A
{
    // A::a is public
    // A::b is not accessible
    // A::c is protected
};

class C : private A
{
    // A::a is private (*)
    // A::b is not accessible
    // A::c is private (*)
};

class D : protected A
{
    // A::a is protected
```

```
    // A::b is not accessible
    // A::c is protected
};
// * The current class can access the
//   variable, but further derivatives cannot.
```

Notice how private and protected inheritance affect classes that come after in the inheritance scheme. For example, a class deriving from C cannot see anything of A, not even its public member variables or methods.

9.10 Base and Derived

Any class that is inheriting from another class is said to be derived from that class. A base class is a class that's at the base of an inheritance scheme. We can use the scope operator to refer to a class we derived from explicitly or implicitly, in case of hidden names (looks like: class_name::), Let's look at an example when that might be useful:

```
class base
{
public:
    void f();
protected:
    float f;
    int j;
};

class derived : public base
{
public:
    void f();
private:
    int j;
};

void derived::f()
{
    j = 10;       // derived::j (base::j hidden)
    base::j = 5; // accesses base::j
    f = 5;        // base::f
}

int main()
{
    derived d;
    base* b;

    d.f();    // derived::f()
    b = d;
    b->f();   // base::f(), NOT derived::f()
}
```

There's another important thing to notice about pointers in this example. Namely, that a pointer of a type higher up in our inheritance hierarchy will *not* dereference to a call on the actual type. The compiler has no way of knowing that the memory

the pointer is pointing to is actually of a sub-type. There exists an OOP feature in C++ that will allow us to change this behavior that we will meet shortly.

9.11 Constructor initialization lists

As we briefly mentioned before. Whenever a class is constructed, its parent's constructor is also called, and so is the constructor of each class it's composed of. This happens in a predetermined order:

1. The parent constructor is called.
2. Objects we contain's constructors are called (in the order they appear in our class definition).
3. Our own constructor is called.

For a destructor the child is destroyed before the parent, which is reversed.

The order of composed objects only matters when we use something called an initialization list. Let's first look at what an initalization list is for built-in types:

```
class A
{
public:
    A();
private:
    int a;
    float f;
    char c;
};

A::A() : a(0), f(0) // initialization list
{
    c = 'a'; // assignment, not initialization
}
```

In the above example a_ and f_ are both initialized, whereas c is assigned to. When we deal with composition or inheritance, we can specify arguments to the constructor of composed classes or parent classes in the initalization list. If they do not provide a default constructor, we *have* to do this. Let's see an example:

```
class composed
{
public:
    composed(int a);
};

class base
{
public:
    base(std::string s);
};

class derived : public base
{
```

```
public:
    derived() : base("hello"), a_(10), b_(20) { }

private:
    composed a_;
    composed b_;
};
```

Remember what we said before about the order of composition. Had we used the value of b_ to initalize a_, we would've gotten an undetermined value even if we reversed the order in the initalization list — it's the order in the class definition that matters!

9.12 Polymorphism

9.12.1 The problem

Through inheritance it's easy to see how we could store pointers to a base class in a data structure (let's use an array for simplicity), but have them actually be different classes. Let's use a WoW example for this.

Let's say we have a class for players, and one for NPCs. It's not a far stretch to see that they most likely share a lot of functionality — and quite quickly we might start to think that they should share a common base class; let's call this class a unit; it's not an object in the world by itself, but it's used to implement functionality that players and NPCs share. Let's see the UML diagram for this idea: http://i.imgur.com/3Y91hL2.png.

And the code:

```
class unit
{
public:
    void walk_to(float x, float y, float z);
    // other methods...
private:
    float x_;
    float y_;
    float z_;
};

class player : public unit
{
public:
    void walk_to(float x, float y, float z)
    {
        // player specific walk stuf...

        // finish up by invoking shared functionality:
        unit::walk_to(x, y, z);
    }
};

class npc : public unit
{
public:
```

```
    void walk_to(float x, float y, float z)
    {
        // npc specific walk stuff...

        // finish up by invoking shared funcionality:
        unit::walk_to(x, y, z);
    }
};

// takes an array of unit pointers;
// includes both players and npcs
void make_units_walk(float x, float y, float z, unit* array[],
    int size)
{
    for (int i = 0; i < size; ++i)
    {
        unit* u = array[i];
        u->walk_to(x, y, z);
    }
}
```

This code above might look like it should invoke the player::walk_to and npc::walk_to correctly. But if we remember what we said before, the compiler does not actually know whether a unit is a player or an npc; so the function that's invoked is unit::walk_to, not as we intended: npc::walk_to or player::walk_to.

9.12.2 The solution

The solution comes in what is referred to as *polymorphism*. In the simple high-level understanding of how it works, it allows us to mark certain methods as *virtual*. This mean that extra code will be generated to *dynamically* (while the programming is running) determine what type the instance is, and then call the corresponding type's function.

Let's see how this works in code:

```
class A
{
public:
    virtual void method_one();
    void method_two();
};

class B : public A
{
public:
    // Override functionality of both of A's method:
    void method_one();
    void method_two();
};

class C : public B
{
    // Override no functionality of B
};:
```

```
int main()
{
    A a;
    B b;
    C c;
    A *ap1 = &a, *ap2 = &b, *ap3 = &c;

    ap1->method_one(); // A::method_one()
    ap1->method_two(); // A::method_two()

    ap2->method_one(); // B::method_one(): because of virtual
    ap2->method_two(); // A::method_two(): not virtual

    ap3->method_one(); // B::method_one()
    ap3->method_two(); // A::method_two();
}
```

As you can see, specifying virtual in the A causes the "natural" behavior to happen. The underlying object's overriden implementation is called, or if it specifies none, the closest parent in our inheritance scheme's.

9.12.3 Low-level

As you hopefully already realized this comes with a trade-off. After all, the CPU is not aware of our high level types, so the compiler needs to implement this dynamic lookup of types and what method to invoke as more instructions in our compiled binary.

This is commonly done through what is called a virtual table (vtable for short). A standard way to implement this for compiler vendors is by adding another pointer to each instance of our class, and then generate code that's executed when the construction happens, that make this pointer point to what is called a virtual table.

This virtual table is unique to your class. For example, if we have three classes and we use virtual methods in our inheritance scheme, then 3 different virtual tables will be generated. In other words, each instance of that specific class will be pointing to the same virtual table.

The virtual table in turn is an array of function pointers — one for each virtual method in your class — that are pointing to the function that needs to be invoked for an instance of this class.

The vtable itself is built during compilation, which increases your binary size (something you generally don't care about, but still interesting to note). There's also the size of the extra pointer of each instance of your class. As well as the instructions needed to lookup the correct function to call in the vtable when a method invocation happens on a pointer or reference.

Remember that this trade-off is not generally going to be a speed-factor you will care about. When in doubt about performance, remember to always benchmark your specific concern (in other words, a general benchmark of virtual methods make no sense).

9.12.4 Virtual destructor

If we delete an object using a base pointer, our derived's class destructor will *not* be called unless the destructor too is virtual. This is an important thing to remember if your sub-class allocates new memory. If it does, and the destructor is not virtual, then deleting an object of that type using a base pointer will cause a memory leakage. Let's see an example:

```
class base
{
public:
    base() {}
    ~base() {} // no virtual destructor
};

class mem_leak : public base
{
    int* ptr_;
public:
    mem_leak() ptr_(new int(10)) {}
    ~mem_leak() { delete ptr_; }
};

int main()
{
    base* bp = new mem_leak;
    delete bp; // the memory of mem_leak::ptr_ is not freed
}
```

The solution would be to add the virtual keyword in front of ~ base(), causing mem_leak's destructor to be called as well (as opposed to just base's).

9.13 Override and final

There are two keywords we can use with inheritance, to help the compiler enforce certain design choices. One is override — this one helps us assure that we're actually overriding a method in a parent class. If for example we named the parameters wrong, the compiler will tell us that we're in fact not overriding but creating a completely new function prototype.

There's another keyword called final. This one indidcates that no further overrides of this method can be done in classes following ours. It can also be used for classes to specify that it cannot be derived from.

Let's see a code example:

```
// A cannot be inherited from because it's marked as final
class A final { };

class B
{
public:
    virtual void f();
};

class C : public B
```

```
{
public:
    void f() override; // overrides B::f()
};

class D : public B
{
public:
    void f() final; // cannot be further overriden
};

class E : public D
{
public:
    void f() override; // error: D::f() is final
};
```

9.14 Summary

Today we learned the founding blocks of OOP, there's still a lot to be learned, but now we know enough to delve into the simpler parts of MaNGOS, which we will do next lesson.

9.15 Homework

Make sure you understand this lesson perfectly — all the way to the details. The following lessons will require good understanding of what's been covered here.

10 Lesson Nine

The first part of this lesson in setting up mangos. I'll run through how that works.

10.1 Setting up MaNGOS

1. Open a root terminal, and run the following command: apt-get install git make cmake g++ libmysqlclient-dev libssl-dev
2. Open a non-root terminal. Run the following commands:
3. git clone https://github.com/mangosone/server.git
4. cd server/src/bindings/
5. git clone https://github.com/mangosone/scripts.git
6. Open the "CMakeLists.txt" file and remove the # in front of the last line, so the last line reads: "add_subdirectory(scripts)".
7. cd
8. mkdir build
9. cd build
10. cmake ../server -DCMAKE_INSTALL_PREFIX=$HOME/local
11. make install
12. cd

What this above does: first we clone the git repositories (git is a revision control software), make a build directory, run cmake which generate make files. We then invoke make which will process these make files and invoke our compiler to compile the source code and link it all together.

1. Go back to the root terminal and run: apt-get install mysql-server
2. Go back to the normal terminal.
3. cp local/etc/mangosd.conf.dist local/etc/mangosd.conf
4. cp local/etc/realmd.conf.dist local/etc/realmd.conf
5. cp local/etc/.conf.dist local/etc/.conf
6. Open these files mangosd.conf, realmd.conf and scriptdev2.conf and edit the DatabaseConnection attributes so that they contain the correct login info (e.g. the root password to mysql).

1. Clone the database repository by typing: git clone https://github.com/mangosone/database.git

2. Open the file README.md and follow the instructions.

1. Open the VirtualBox manager and go into settings for your virtual machine.
2. Select the "Shared Folders" tab.
3. Add your World of Warcraft TBC directory as a shared folder.
4. Open a terminal.
5. mkdir /build-extractor
6. cd /build-extractor
7. cmake /server/contrib/extractor
8. make
9. cd /path/to/wow
10. /extractor-build/ad ; creates dirs: dbc and maps
11. mkdir /home/user_name/Data
12. mount -t vboxsf share_name /home/user_name/Data
13. cd /
14. build-extractor/ad

To add an account and realm you do:

1. /sbin/ifconfig : use your IP for the following command
2. echo 'INSERT INTO realmd.realmlist(id, name, address, realmflags) VALUES(1, "mangos", "YOUR IP GOES HERE", 0x2);' | mysql -u root -p
3. echo 'INSERT INTO account (username, sha_pass_hash, gmlevel, expansion) VALUES ("ACCOUNT", SHA1("ACCOUNT:PASSWORD"), 3, 1);' | mysql -u root -p

To start the server you do:

1. Open two terminals and do in both: cd /local
2. Run in one of them: ./bin/mangosd
3. And in the other: ./bin/realmd

Point your realmlist.wtf to the IP from before, and you should be able to log in with ACCOUNT/PASSWORD.

10.2 Const methods

A method can be marked as const. When you're dealing with a const object or a const pointer or reference to an object you can only call const methods. These methods are const because they do not modify the object that's pointed to by the this pointer. An object or pointer or reference to object that's not const can still invoke const methods. When you don't change any data, it's a good practice to always mark your methods as const. You do it like this:

```
class C
{
public:
    // Non-const method, can change data
    void m() { }
    // Const method, cannot change data
    void cm() const { }
};
```

10.3 Containers

C++'s standard library comes with a handful of data structures, that are referred to as containers in the standard library. One of them is a dynamic array — the same type of data structure we implemented for our C string. C++ containers come with an added feature where it allows us to specify the type it will contain. In other words, we can use these containers no matter what we want to contain, and we don't have to use the void pointer approach we did in C.

These containers do that by using *templates*, a concept we will save for later. Let's just for now look at how we'd use the most common container in the C++ standard library — the dynamic array, or as it's called in C++ the *vector*.

```
#include <cstdio>
#incluce <vector> // for std::vector

int main()
{
    // Create a vector that contain ints
    std::vector<int> vi;
    // Add some ints to it
    for (int i = 0; i < 10; ++i)
        vi.push_back(i);
    // Print out the elements in our vector
    for (std::size_t i = 0; i < vi.size(); ++i)
        printf("%d ", vi[i]);
}
```

The example above should hopefully be clear enough how we use the std::vector. Inside of the <> goes the data type we intend to contain.

10.4 Event-driven programming

Event-driven programming is a *programming paradigm* where we register objects to what is commonly referred to as an *event handler* and when the event happens,

everyone that's registered to the handler has their *callback function* invoked.

This is more easily explainable using code.

```
#include <cstdio>
#include <vector>

enum chest_events
{
    EVENT_CHEST_OPENED,
    EVENT_CHEST_RESPAWNED
};

// class to handle events with chests in the world
// this is not a part of actual mangos, but just
// an example to show this paradigm
class chest_event_handler
{
    // registered is used for callback functions that
    // are registered to a particular event
    struct registered
    {
        registered(chest_events e, void (*c)())
            : event(e), callback(c)
        { }
        chest_events event; // event we're registered to
        void (*callback)(); // our callback function
    };

public:
    void subscribe(chest_events event, void (*callback)())
    {
        registered r(event, callback);
        callbacks_.push_back(r);
    }

    void invoke(chest_events event)
    {
        for (std::size_t i = 0; i < callbacks_.size(); ++i)
            if (callbacks_[i].event == event)
                callbacks_[i].callback();
    }

private:
    std::vector<registered> callbacks_;
};

void callback_one() { printf("callback one\n"); }
void callback_two() { printf("callback two\n"); }
void callback_three() { printf("callback three\n"); }

void register_callbacks(chest_event_handler& h)
{
    h.subscribe(EVENT_CHEST_OPENED, callback_one);
    h.subscribe(EVENT_CHEST_OPENED, callback_two);
    h.subscribe(EVENT_CHEST_RESPAWNED, callback_three);
}

int main()
```

```
{
    chest_event_handler h;

    register_callbacks();

    printf("Chest opened...\n");
    h.invoke(EVENT_CHEST_OPENED);

    printf("Chest respawned...\n");
    h.invoke(EVENT_CHEST_RESPAWNED);

    printf("Chest opened again...\n");
    h.invoke(EVENT_CHEST_OPENED);

    return 0;
}
```

In this above example we make a class that allows users to register callbacks for chest related events — such as opening and respawning. Objects that care about these events can then register to be notified when the event they care about triggers.

Event-driven design is a powerful paradigm and is used for various purposes.

10.5 Mangos AI

Given how simple the WoW AI is (apart from bosses having scripted behavior, the AI does little more than attack and use abilities on a timer), it makes sense to not delve into complex territories of AI design.

Mangos chose to use an event-driven design for their AI. What they did was create a common base class (called CreatureAI) that has a set amount of methods, such as for when the creature is attacked, when it takes damage, when it dies, etc.

They then derive from this common base, for example you can derive a GuardAI, to define behaviors for guards, or you can derive a PetAI, to define behaviors for how pets react to events, or even a RagnarosAI that defines how Ragnaros behaves to events.

The main event of mangos AI classes is called "Update", and it's invoked each "tick" of the server. A tick is, assuming the server is default configured and not under heavy load, executed every 100 milliseconds. So our AI will have its Update event invoked every 100 milliseconds.

In mangos they did not apply a subscribe-based event AI, instead *all* creatures have an AI, and the callbacks are always triggered. If you don't override the callback, then CreatureAI's default behavior will be executed — which is to do nothing at all (all default callbacks have an empty body).

Let's look at an UML diagram that describes how these classes relate to eachother (not every AI derived from CreatureAI is included):

http://i.imgur.com/H1y9DYG.png

This above is the gist of the relation of the AI classes in mangos. CreatureAI implements all its methods as virtual functions with an empty body. Then a CreatureAI pointer is saved inside of every Creature and when the game state changes

in a way that the AI needs to call, the CreatureAI's method is invoked, and through the use of virtual tables, the actual AI that creature is using has its method invoked.

An isolated example of this design can be seen in the following code listing:

```
// Note: This code is exhibits of the MaNGOS code base
// taken out of context. Not everything in here is good
// design, but the code is presented to allow you to
// get a feeling for the code base and how it deals with
// AI.
// General rule of thumb: Don't learn design by looking
// at the MaNGOS code base.

#include <cstdio>

enum TypeID
{
    // ...
    TYPEID_UNIT   = 4, // This means NPC
    TYPEID_PLAYER = 5,
    // ...
};

class Object
{
public:
    TypeID GetTypeID() const { return m_objectTypeId; }
protected:
    TypeID m_objectTypeId;
};

class WorldObject : public Object { /* ... */ };

class Unit : public WorldObject
{
public:
    void Kill(Unit* target);
};

class CreatureAI
{
public:
    virtual void JustDied(Unit* killer) { }
};

class ScriptedAI : public CreatureAI { /* ... */ };

class boss_ragnarosAI : public CreatureAI
{
public:
    void JustDied(Unit* killer) override
    {
        printf("boss_ragnarosAI::JustDied\n");
    }
};

class Creature : public Unit
{
```

```
public:
    Creature() : i_AI(NULL)
    {
        m_objectTypeId = TYPEID_UNIT;
        AIM_Initialize();
    }
    ~Creature() { delete i_AI; }

    // Function that selects AI for this creature
    void AIM_Initialize()
    {
        // Simplified for this example
        if (i_AI)
            return;

        i_AI = new boss_ragnarosAI;
    }

    CreatureAI* AI() { return i_AI; }

private:
    CreatureAI* i_AI;
};

class Player : public Unit
{
public:
    Player() { m_objectTypeId = TYPEID_PLAYER; }
};

void Unit::Kill(Unit* target)
{
    if (target->GetTypeID() == TYPEID_UNIT)
    {
        // If it's a creature we cast it
        // and invoke the
        // CreatureAI::JustDied method
        ((Creature*)target)->AI()->JustDied(this);
    }
    // ...
}

int main()
{
    Creature ragnaros;
    Player shiro;
    shiro.Kill(&ragnaros);
}
```

Just to make the above a bit more visual I've made a UML diagram that shows the relation of the different Object classes in the core:

http://i.imgur.com/RBfPLjG.png

Note that not all of these were included in the extracted code example above.

Also, if you go into the mangos code base and open src/game/CreatureAI.h you can see which callbacks are available.

10.6 Homework

Go into src/bindings/scriptdev2 and locate the script of a boss you find interesting. Test him in game and figure out if there's anything wrong with the script (if there isn't choose another boss). Then proceed to fix the script and test your fixes. Remember to check other scripts for how to solve certain problems. The best resource is seeing code of people more experienced than you are.

We will later look into how we could add scripts to bosses that aren't using scripts already. But that requires us understanding how scripts are loaded and some of the database structure, so we will wait with that for now. Make sure the boss you pick has a script it's using already, in other words.

11 Lesson Ten

In the following lessons we will start using another book. If you're unable to get the book, you could probably follow along by finding texts through google on the subjects the book covered (they will be listed in this pdf), but it is recommended you get the book if you can — it contains invaluable exercises, and you're also guaranteed that the information in there is accurate and up-to-date.

The book is called C++ Primer (5th edition) and is authored by Stanley B. Lippman, Josee Lajoie and Barbara E. Moo. Link to amazon: http://www.amazon.com/Primer-5th-Edition-Stanley-Lippman/dp/0321714113.

11.1 What to read

Below is the listing of what you should read, we pick and skip quite a bit as we know some of these subjects already. Remember to do all exercises you encounter in the listed chapters and sections:

1. 1.2 — <iostream>, istream and ostream, std::cout, std::cin, std::cerr, std::clog (basic input and output using the C++ library <iostream>).
2. 3.1 — Namespace using declarations
3. 3.2 — C++ <string>
4. 3.3 — C++ <vector>
5. 3.4 — iterators; iterator with C++ standard library containers, iterator arithmetic
6. 5.6 — exceptions, try throw and catch keyword, exception safety, <stdexcept> header
7. 6.4 — function overloading
8. 6.5 — default arguments, inline and constexpr functions, <cassert> header
9. 6.6 — functions argument matching
10. 7.5.2 — delegating constructors
11. 7.5.3 — default constructor
12. 7.5.4 — conversion constructors, explicit constructors
13. 7.6 — static methods, static data members

For those of you not using the book, the topic is listed as well so you'll be able to google information about it.

And that's the entire first part covered in a single lesson. Mainly because we were able to skip the vast majority of the first part due to us already knowing most things covered.

11.2 Exceptions at a lower level

Let's continue our approach of always thinking about features in a lower-level, and let's apply that to exceptions as well.

11.2.1 When an exception is thrown

Let's first look at what logically happens when an exception is thrown. An exception will follow the stack until it reaches a try-block with a catch that is able to handle that exception type, in other words an appropriate exception handler. That exception handler can then choose to rethrow the exceptions, upon which this process is repeated.

If you remember how the stack works, you'll remember that returning from a function is popping our stack frame and returning to the address that was pushed onto the stack. We have not discussed how this works with C++ quite yet, in particular when a function exists and the stack is unwound, destructors must be appropriately called. This entire process is referred to as *stack unwinding*. Stack unwinding does not only happen at return statements, but also when exceptions are thrown. This means that any objects that would've been destructed on a return path, will be destructed if a throw happens.

When an exception is thrown the stack will be unwound until an appropriate exception handler is reached.

11.2.2 How are exceptions implemented

There's different ways to implement exceptions. The main way of doing it today is called the zero-cost exception model. This method is aimed at not adding any run-time cost when exceptions do *not* occur.

It works by, at compile time, creating a table of exception handlers that are appropriate at that point in the code. When the exception then happens during run-time, the code consult that table and pick the handler it needs and begin unwinding the stack.

This means no extra computation needs to be done when no exception happens, but instead the logic will only execute when an actual exception occurs.

However, since the table is not often used, it's unlikely to be present in the CPU's cache, and it will need to be loaded from memory. This means that actually throwing an exception is generally quite expensive with this model.

This model of implementing exceptions essentially means that you should indeed prefer to only use exceptions when an exceptional condition occurs, as opposed to a way to control normal program flow. Which of course would make sense no matter the performance concerns.

11.2.3 To use or not to use exceptions

Let's take a quick example of when you might not want an exception versus when you might want one.

Let's imagine we have a function that verifies account-name and password. Now, if the combination was incorrect, in other words the user entered an invalid password and username combo, then that should probably *not* be an exception. Instead our authenticate function might return a boolean indicating success or failure.

Now, on the other hand, if the function is querying the back-end database, and mid-query the database server interrupts the connection, we might be looking at a proper time to use an exception. After all, our function — which purpose is to determine if a username and password combination is valid — is unable to fullfill its purpose. While one could argue that we'd still want to deny the user to log-in if the back-end went missing, that's a decision our function should not make (our function's purpose is to verify the username and password combination, not to log-in the user, after all). Therefore, when this condition happens, our function's only option is to throw an exception.

11.3 Homework

As a homework we will look at the mangos spell system. The spell system is potentially the least elegant system in mangos, so we'll only take a quick dip into it. But that is still our homework.

Your task is to download the following program, called spellworks, which you can obtain here: `https://github.com/ValkyrieProject/QSpellWork` (the release tab has a setup file). This program works on Windows, and probably should work on Linux as well, although I haven't tried it. It allows you to visually browse the data of Spell.dbc, which is a data-file that mangos uses that describes how all spells work.

Once you've downloaded that program take a spell, be it charge, moonfire, swiftmend, or what have you, and look at its data in spellworks.

When you've done that your next task is to look at the spellsystem in mangos (which files you should check out is mentioned below) and figure out enough about how your spell works so that you can change something about its behavior. It does not have to be something significant — just make it do *something* different.

Relevant files:

1. src/game/Spell.h — contains the Spell class definiton amongst others, it describes the working of spells.
2. src/game/Spell.cpp — Implementation file of Spell.h, contains core functionality of spells.
3. src/game/SpellEffects.cpp — Implementation file of Spell.h, contains implementation of all individual spell effects (corresponding to the Effect Ids you see in spellworks, for example).
4. src/game/SpellAuras.h — describes the different Aura classes
5. src/game/SpellAuras.cpp — provides implementation, also contains the "effect" code of auras (what happens when that aura is applied or removed).

12 Lesson Eleven

Below will be listed what to read about in the book, as well as what's covered; in case someone is not using the book. Remember to ask questions if you have any problems with the covered subjects.

1. Read the intro to part 2 — skip if you don't have the book
2. All of Chapter 8 — relationship of stream-classes, read up on istream and ostream, which provide the basic input stream and output stream objects of the standard library. Then read up on the details of fstream, ifstream and ofstream, all included from the <fstream> header. Lastly look at stringstreams, which come from the <sstream> header.
3. All of Chapter 9 — the sequential containers in C++ are: vector, deque, list, forward_list, array and string. Their interfaces are similar so they should be easy to read up on in a group. Check how iterators work with these sequential containers, as well as what members they have. Make sure your resources are written for C++11. You should also read up on how std::vector and std::string grows (increases in size when you add elements to it). Then take a quick peak at container adaptors — "new" containers built of the ones we discussed, such as the std::stack, std::queue and std::priority_queue.

As always I recommend you to use the actual book, rather than internet-found resources, but given the above information both options should be doable.

12.1 Homework

Make sure you understand the covered topics — do as many exercises as required until you're fully confident that you know the topics that have been covered inside out.

13 Lesson Twelve

What to read:

1. All of Chapter 10 — <algorithm> header, check out some of the functions, read up on what predicates are and how you can use them with the std algorithms. Also read up on a topic called lambdas, and how you can use them as predicates. There are different types of iterators (beyond those we've seen already), some of them you should read up on, namely: insert iterators, stream iterators and reverse iterators. Iterators also come in different categories, which define the operations they can do, read up on the different iterator categories: input iterator, output iterator, forward iterator, bidirectional iterator, random-access iterator. Some containers define algorithms as members, check std::list for some examples (list::merge, list::sort, list::splice, etc).

2. All of Chapter 11 — Associative containers, read up on how to use std::map from the <map> header and std::set from the <set> header. Check out list initalization of maps, for example, a new feature in C++11. Check out the std::multimap and std::multiset, as well. Read up on how the key for maps and sets work, and how you can use a comparison function for the key-type. Lookup the std::pair type. As opposed to the ordered associated containers, there also comes unordered ones — look up std::unordered_map and std::unordered_set, see how you can define your own comparison operation using the std::hash function.

13.1 Container implementations

The reason we started with C and not C++, was that so we'd always be able to make parallels to how things work under the hood, so that when we use containers and algorithms from the standard library we have a knowledge of their inner workings.

An important notion before we start our discussion. The standard most often does not specify the details of how something must be implemented, instead it puts up conditions that an implementation must meet.

Looking back at our chapters on data structures for C, it should hopefully be fairly clear to you already how most C++ containers work under the hood. Spend a moment before you read on and try to realize how the following containers would be implemented:

1. std::forward_list
2. std::list
3. std::vector
4. std::string
5. std::set

6. std::map

You should be able to see which underlying data structure they use, just by having used them. But just for thoroughness I'll go through all of them, and some more.

13.1.1 std::forward_list and std::list

These, as the name probably already tipped you off to, are implemented as linked lists, the forward_list as a singly linked list, and the list as a doubly one.

As a reminder look at the following image which shows a visual representation of a singly linked list: http://upload.wikimedia.org/wikipedia/commons/6/6d/Singly-linked-list.svg And for a doubly linked list: http://upload.wikimedia.org/wikipedia/commons/5/5e/Doubly-linked-list.svg

The forward_list exists as a container that should have no space or time overhead compared to a hand-written singly linked list in C, therefore any convenience features that go against this requirement has been left out.

13.1.2 std::vector and std::string

The std::vector container provide us with a dynamic array, in other words a data-structure that will grow in size as elements are inserted into it (a common growth factor being 2, meaning the container doubles in allocated memory when it needs to grow — note that this is an implementation detail and the exact growth strategy varies with your compiler). The vector being a dynamic array means that all its elements are guaranteed to be contiguous in memory.

The std::string library too is an array allocated at the heap that grows in size when need be, as the vector. There's also a strategy called the Short String Optimization. To showcase what SSO is here's some code:

```
class string
{
public:
    string();
    string(const char* str);

    // ...

private:
    static const size_t sso_size = 15;
    size_t size_;                    // string::size()
    char* ptr_;
    union
    {
        size_t alloc_size_;          // allocated memory
        char sso_[sso_size + 1];     // +1 for null char
    };
};


string::string() :
```

```
    size_{0}, ptr_{sso_}
{
    sso_[0] = '\0';
}

string(const char* str) :
    size_{strlen(str)},
    ptr_{size_ <= sso_size ? sso_ : new char[size_ + 1]},
    alloc_size_{size_ > sso_size ? size_ + 1 : 0}
{
    strcpy(ptr_, str);
}
```

As you can see, the SSO is simply using our sso_ array if the size remains below our constant, if it goes above we begin using the heap. It's a simple optimization, but research has shown that it's quite effective — it's quite common to have short strings with a short life-time. For example:

```
void f()
{
    string name{"john"}; // does not go on the heap
    // ...
}
```

13.1.3 std::map and std::set

Map and set are the same type of container. The only difference being that the latter does not have a value, but only a key. The time requirements for map and set means that their underlying implementation has to be a self-balancing binary search tree.

Here's a picture showing a balanced binary tree where the value is the key, in other words what an std::set might be depicted as: http://en.wikipedia.org/wiki/File:AVLtreef.svg

13.1.4 std::unoreded_map and std::unordered_set

Both of these containers are hash-based containers. A hash-based container is called so because it depends on a hash function. The hash function will take our key-value and convert it into a *bucket* index. The bucket then contains the value that maps to that key. In an ideal world each key would mad to its unique bucket, which only contains one value, but in practice this is not true and a bucket is a list of key, values pair.

In other words, our hashing function brings us to a bucket, where we need to compare our key against the keys in the bucket to find the right one. This is a useful data structure because if the buckets are kept small, then the amount of looking we need to do is low, even if the total data set is enormous.

Here's an image that describes this mapping: http://upload.wikimedia.org/wikipedia/commons/7/7d/Hash_table_3_1_1_0_1_0_0_SP.svg

A concrete example:

1. The key "john" is looked up in our hash map, we use std::hash as our hashing function which yielded the following hash: 6667620739439280681.
2. This value is the index to our buckets, but our amount of buckets is much smaller, in fact we only have 256 buckets, so we apply modulo to this number: 6667620739439280681 % 256 = 41.
3. We go to bucket number 41, which like all buckets in this concrete example happen to be a linked list of std::pair<key, value>.
4. We iterate this linked list until we find our exact key ("john"), then we return the value.

As you can see, the hash map can be a very fast container if the data is evenly distributed amongst the buckets, and the amount of buckets is sufficiently big.

The drawback with hash tables being that when we insert elements and our implementation realizes we need more buckets, then the entire hash table needs to be rebuilt — which can be a very costy operation. This effect can usually be worked around by reserving more space in parts of the program's execution that are less time-critical, so that our insertions during a tight loop (for example), will not cause the hash table to be rebuilt.

Another, and quite dangerous side-effect, is what happens if our hashing function were to map every value to the same bucket — if that happens our data structure becomes nothing more than an expensive linked list. In fact, there are Denial of Service attacks that make use of knowing how to produce hash collisions within the hashing algorithm that the target uses, to fill up its internal hash tables so that each query to the hash table becomes closer to a O(n) operation, as opposed to the wanted speed of O(1). Given an efficient attack, this can bring down a targets operation all together, making it a Denial of Service attack that your hash map was vulnerable to.

To defend against such attacks programmers will need to make it so users cannot manipulate the hash map. If that is not an option, which is common for real-world cases, then other solutions need to be sought. One aspect of a secure hashing function is being collision resistant. A hash function is considered collision resistant if the probability that an adversary can output collisions for said hash function is negligible. The hashing function in C++ is not enforced to be secure, which means we should assume it to be of the non-cryptographic type and there not providing collision resistance. Which means collisions can usually be reliably found. The problem being that most secure hash functions are marginally slower.

A common approach has been to randomize the hash function on initialization, trying to make its output seem less deterministic. However, if the attacker learns the seed he will be able to attack the hashing function all the same, and this technique serves mainly as mitigation to make it harder for the attacker.

However, a cryptographically secure hash function that stays competitive in performance, and is meant to solve just this issue, has been developed called SipHash. If you're interested, or need to use a hash map in a scenario where the security

is important, you should check this talk from CCC: http://www.youtube.com/watch?v=wGYj8fhhUVA.

14 Lesson Thirteen

The following sections in the book should be read:

1. Chapter 12 — std::shared_ptr, std::make_shared, new and delete, new and delete with arrays. std::unique_ptr, std::make_unique, std::allocator.
2. 13.1 — Constructors, destuctors, copy constructors.
3. 13.2 — Copy assignment operator, reference counting for resource management
4. 13.3 — std::swap

15 Lesson Fourteen

The following sections in the book should be read:

1. 13.4 — Simply shows an example of what has been presented so far in the chapter.
2. 13.5 — In this section the authors show the student what classes that manage dynamic memory means in practice, and they implement their own string vector class (works like a vector, but can only store strings).
3. 13.6 — Rvalue references and move semantics. Also introduces the move constructor and move assignment.
4. All of Chapter 14 — Showcases operator overloading and the various conventions for different operators. Operators you should look up how to overload are: input and output operator (« and »), arithmetic and relational operators (==, >, !=, +, -, etc.), compound assignment (+=, -=, etc), subscript operator ([]), increment and decrement (++, –), member-access operator (-> and *), function call operator (operator()), conversion operator (operator type()), make sure to read up on explicit conversion operators.

16 Lesson Fifteen

You should read the chapter listed below. The amount of pages is a bit less this week, because I've added an excerpt of a C++ talk I'd like you all to watch after having read the book material.

1. Chapter 15 — Inheritance, virtual functions, abstract base classes, access control, name lookup in regards to class scope, constructors and copy-control, containers and inheritance (store pointers, read up on splicing which explains why).

We have discussed virtual functions previously in this course. If you don't remember how virtual functions are implemented at a hardware level, go back in this pdf and read up on it. When you do remember, there's a section of a talk I'd like you to watch that explains why function pointers, and consequently virtual functions are bad for performance on modern CPUs.

This is a talk by Eric Brumer, which is one of the developers on the Visual C++ compiler (Microsoft's compiler), who specializes in performance of native code. It's an hour long talk, but I do not intend for you to watch all of it. Below is the link and the timestamps you should watch between:

https://www.youtube.com/watch?v=3MRxucTXPdw Start at 22:25, and end at 37:20.

Make sure to pause during the talk and look up terminology unfamiliar to you, and to also rewatch it if the content presented felt a bit foreign. It's an important subject, and understanding it will help you make better decisions about when to use virtual function dispatching and when to not.

17 Lesson Sixteen

Chapter 16 is what you should read this week, below is the list of what is covered in the chapter, if someone still doesn't have the book.

1. 16.1 — Function templates. Template parameters: type-based (typename) and non-type based (e.g. int).
2. 16.2 — Template argument deduction.
3. 16.3 — How overloading (both of template and non-template functions) work in conjunction with templates.
4. 16.4 — Variadic templates.
5. 16.5 — Template specializations.

It's important to realize that templates are a compile-time feature. This means that you are paying no additional run-time cost. You are however paying in terms of slower compile time, (potentially) a bigger resulting binary, and usually harder to understand compile-time errors (granted something you get used to through trial and error).

Let's consider the following code:

```
/* poor example, avoid this type of code */
class base_obj
{
public:
    virtual bool func(base_obj* rhs) const;
};

void some_algorithm(base_obj* lhs, base_obj* rhs)
{
    if (lhs->func(rhs)
    {
        /* do stuff */
    }
    else
    {
        /* do stuff */
    }
}
```

Given the above code and you wanting to reuse some_algorithm for your program, you would need to inherit from base_obj and define an overload of the virtual function. Not only is this code inefficient (for reasons we looked at in last lesson), it's also incredibly hard to extend and maintain. This is the type of code you do not want to write.

Let's look at a potential, but still poor, solution.

```
/* poor solution, avoid this type of code */
void some_algorithm(void* a, void* b, bool (*func_ptr)(void* a,
    void* b))
{
```

```
    if (func_ptr(a, b))
    {
        /* do stuff */
    }
    else
    {
        /* do stuff */
    }
}
```

This is a solution that's more general in that you don't need to extend a class to use it, but it's still a poor one. Not only does it provide no type knowledge (we're passing void pointers), but it also incurrs the same overhead as a virtual function call would (our function pointer).

Now, you probably already see where this is going. But, indeed, templates would be the ideal solution here. Not only does that retain type-information that the compiler can use to verify the legality of using our type for some_algorithm, but it will also not incurr the runtime over-head that virtual functions or function pointers would, as templates are resolved on compile-time.

The solution would look like this:

```
/* the correct solution */
template<typename T>
void some_algorithm(T& lhs, T& rhs)
{
    if (a.func(b))
    {
        /* do stuff */
    }
    else
    {
        /* do stuff */
    }
}
```

Notice that using this function with a type that does not have the function func does not work, so we're covered for that case. It also incurrs no runtime over-head, and all around is a much better solution.

Templates are an amazing feature, learn to love them and your productivity and code-quality will benefit significantly (of course, like any feature they can be over-used, and there are times when templates are the wrong solution).

18 Lesson Seventeen

This week won't be about reading, instead there's a decently big exercise that you'll get to do this week. This exercise should be done by next lesson opportunity, and handed in.

18.1 Exercise part one

Your task is to implement your own variation of the std::vector. You will not have to implement it to be standard-compliant, nor full-featured. Your main task is to make sure the test example presented below compiles with your vector, and that the output says your vector passed all the tests. After you have achieved this, you should submit it to Shiro for a code-review and feedback.

Here's a paste version of this source for easier copy: https://thunked.org/p/view/pub/ghlyiiz5e

```
/* Write a Vector Exercise
 * This is the test-code your Vector must conform to.
 * When you compile and run this code, the output should be
 * that your vector passed the test.
 *
 */

#include "Vector.h" // Your vector header.
#include <iostream>

class test_class { };

Vector<int> move_test(Vector<int> v)
{
    // Test copy or move construction
    return v;
}

int main()
{
    Vector<int> vi {1,4,7,12};
    Vector<test_class> vtc;

    // Verify that the vectors are of expected size
    if (vi.size() != 4 || !vtc.empty())
    {
        std::cerr << "size() returned unexepected result" <<
            std::endl;
        return 1;
    }

    // Equivalent to calling std::accumulate
    int sum = 0;
    for (auto itr = vi.begin(); itr != vi.end(); ++itr)
        sum += *itr;
    if (sum != 24)
    {
        std::cerr << "Iterating over the range does not yield "
            "the inserted range" << std::endl;
```

```
        return 2;
    }

    // Capacity and push back
    std::size_t capacity = vi.capacity();
    for (int i = 0; i < 1024; ++i)
        vi.push_back(i);

    // Push back and subscript
    if (vi.size() != 1028 || vi[512+4] != 512)
    {
        std::cerr << "push_back() or subscript not working" <<
            std::endl;
        return 3;
    }

    // Verify that a reallocation has happened
    if (capacity == vi.capacity())
    {
        std::cerr << "The growth factor is chosen poorly" <<
            std::endl;
        return 4;
    }

    // Test copy construction
    auto v2 = move_test(vi);

    v2.erase(v2.begin() + 10, v2.begin() + 20);
    if (v2[11] != 17)
    {
        std::cerr << "Erase not working correctly" <<
            std::endl;
        return 5;
    }

    if (Vector<float>{7.5,12.5} != Vector<float>{7.5,12.5})
    {
        std::cerr << "Comparison working incorrectly" <<
            std::endl;
        return 6;
    }

    std::cout << "Your vector passed all tests" << std::endl;
    return 0;
}
```

18.2 Exercise part two

The next part of the exercise is much smaller, and it requires you to implement a generic algorithm that works on all containers conforming to its interface. This means that it should work for your vector, and std::vector alike. The C++ Standard Library solves this by using iterators, and as long as a container implements the needed iterator type, the algorithm can be used for it.

The algorithm we chose for you to implement is std::accumulate. The following

code should work with your algorithm implementation:

```
#include <cassert>
#include <vector>
#include <algorithm>
#include "Vector.h"     // Your vector class from part one.
#include "Algorithm.h" // Your algorithms header, containing
                       // Accumulate

int main()
{
    std::vector<int> vi{5, 10, 15};
    Vector<int> vi2{15, 5, 10};

    assert(Accumulate(vi.begin(), vi.end(), 0) ==
        std::accumulate(vi2.begin(), vi2.end(), 0));
    assert(Accumulate(vi2.begin(), vi2.end(), 10) ==
        std::accumulate(vi.begin(), vi.end(), 10));
}
```

19 Lesson Eighteen

This week is dedicated to reading Chapter 17 and 18. The topics in these chapters should be fairly straightforward, so although the combined amount of pages is significant, it should be simple to read. What's covered is:

1. 17.1 — std::tuple.
2. 17.2 — std::bitset.
3. 17.3 — the <regex> header.
4. 17.4 — the <random> header.
5. 17.5 — IO manipulators defines in <iostream>, as well as the <iomanip> header.
6. 18.1 — exception handling (in-depth)
7. 18.2 — namespaces (in-depth)
8. 18.3 — multiple and virtual inheritance (the latter of which we've investigated thoroughly before).

In two weeks time we will have finished the C++ Primer 5th edition book, and covered a great deal of C++. While of course optional, I intend to propose a project of significant proportion (and hopefully fun) to try your hands at. After all, while you might have learned a lot of new concepts, it's not until you get to use them to solve problems that they will truly be a part of your knowledge repertoire.

20 Lesson Nineteen

This week is the last week of reading in the book. Next week I will write about an exercise that I hope those of you still with us attempt. This week, however, it's all about reading Chapter 19. Namely:

1. 19.1 — Overloading new and delete. Placement new.
2. 19.2 — RTTI. The typeid operator, and dynamic_cast.
3. 19.3 — Scoped enums (extends upon the enums we learned in C).
4. 19.4 — Pointer to class member.
5. 19.5 — Nesting classes.
6. 19.6 — Union (we looked at unions briefly while doing C).
7. 19.7 — Local classes. These are classes defined inside a function's scope.
8. 19.8 — Bitfields, the volatile keyword. Linkage specification using extern "C".

21 Final Exercise

As I've mentioned before I wanted the final exercise to be a bigger project, preferably something enjoyable. Therefore I have a suggestion to those of you still reading. Obviously, you're free to choose any project to work on, but the next step in your path to mastering C++ *should be* working on a sizable project. This could benficially be done by checking out some open-source project, and doing some work on that (for example by fixing some of their bugs, or resolving some of their issues). However, the suggestion I have for you is different from that.

21.1 A game

My suggestion is to use a framework called openFrameworks, which is meant for "creative coding", to create a game. This could be something simple such as a pacman game, or something big such as a sprite-based 2D RPG. Your fantasy is the limit, as they say. Below are some links to check out (in order) to get openFrameworks set up and ready for some creative coding:

1. http://openframeworks.cc/download/ (download the linux code::blocks version)
2. http://openframeworks.cc/setup/linux-codeblocks/ (follow the instructions under the section called Debian)
3. http://openframeworks.cc/tutorials/introduction/001_chapter1.html (complete this tutorial to gain some familiarity with openFrameworks)
4. http://ofxaddons.com/ (here are some addons to extend the functionality of openFrameworks, one you might want to check out is ofxSpriteSheetRenderer if you plan on rendering sprites).

This is not an exercise with a deadline, or indeed something you even need to submit to me. It's just an idea of something that sounds fun to me, and hopefully to you as well. Just go for something that captivates and pulls you back in to wanting to code more, and you'll become a proficient programmer as a by-effect. Good luck, and thanks for sticking with us for this course!