CS 309: Autonomous Robots
FRI I

C++ Primer

Instructor: Justin Hart

http://justinhart.net/teaching/2020_spring_cs309/

Hello World!

Exercises	ex01
	ex02
Code Objectives	#include
	main()
	printf
	std::cout
Compiling	g++
	make

Hello World!

Writing a “Hello World!” program is a traditional way to quickly familiarize oneself with the basics of a new programming language.

```
#include <stdio.h>     //Include statement

//"Main" function
//Parameters
//              int argc
//              char **argv
int main(int argc, char **argv) {
              //Function call
              //"Hello world" - Text literal
              //\n - Escape sequence
              printf("Hello world\n");

              //Return statement
              //0 return value
              return 0;
}
```

Hello World!

```
#include <stdio.h>   //Include statement

//"Main" function
//Parameters
//            int argc
//            char **argv
int main(int argc, char **argv) {
            //Function call
            //"Hello world" - Text literal
            //\n - Escape sequence
            printf("Hello world\n");

            //Return statement
            //0 return value
            return 0;
}
```

```
#include <--file-->
```

Copies a file, verbatim, into this file

Used for “header files”

Header files contain functions and other code that you want to use but not include in this file

Headers are generally used to define the interface to “libraries” of functions and classes written by other people for use in your programs

The also contain the definitions for Application Programmer Interfaces (APIs) or standard or widely-used libraries

Hello World!

```
#include <stdio.h>   //Include statement

//"Main" function
//Parameters
//            int argc
//            char **argv
int main(int argc, char **argv) {
            //Function call
            //"Hello world" - Text literal
            //\n - Escape sequence
            printf("Hello world\n");

            //Return statement
            //0 return value
            return 0;
}
```

int main()

- In C++ runnable code is written in functions
- main() is the function that is run when the OS starts your program

int argc, char **argv

- We won't be using this right now
- This is how the OS provides arguments to your program
- If you wrote a program that was run akin to:
  - # add 5 6 7
- 5, 6, and 7 would appear as text in argv
- argc would tell you how many arguments are in argv

Hello World!

```
#include <stdio.h>   //Include statement

//"Main" function
//Parameters
//            int argc
//            char **argv
int main(int argc, char **argv) {
            //Function call
            //"Hello world" - Text literal
            //\n - Escape sequence
            printf("Hello world\n");

            //Return statement
            //0 return value
            return 0;
}
```

printf()

C-style function for printing formatted text to the terminal

\n inserts a carriage return and a newline

return 0

Returns the number 0

0 tells the OS that the program ran and ended successfully

Hello World!

```
#include <stdio.h>   //Include statement

//"Main" function
//Parameters
//            int argc
//            char **argv
int main(int argc, char **argv) {
            //Function call
            //"Hello world" - Text literal
            //\n - Escape sequence
            printf("Hello world\n");

            //Return statement
            //0 return value
            return 0;
}
```

Usually the contents of a function appear inside curly braces

This is called a “block”

argc & argv are called “formal parameters”

Formal parameters define where data can be passed into a function

In calling printf “Hello world\n” is called an “actual parameter.”

Actual parameters are data passed into the function.

“Lines” in C++ end in ;

Compiling & Running

To compile (build the program from source code):

g++ ex01.cpp –o ex01

To run:

./ex01

Hello World!

```
#include <iostream> //C++ - style include file, no .h at the end

int main(int argc, char **argv) {

	//std - Namespace
	//		Namespaces allow us to declare functions and variables separately, so if
	//		std - the standard namespace defines cout, we may have another namespace
	//		that defines it differently. Both can co-exist because of the namespace
	//		<< - Insertion operator, says "Insert 'Hello world' into cout."
	//		endl - Endline, serves the same purpose as \n in the previous example
	std::cout << "Hello world" << std::endl;

	return 0;
}
```

Hello World!

```
#include <iostream>

int main(int argc, char **argv) {
        std::cout << "Hello world" << std::endl;
        return 0;
}
```

C++ standard headers generally exclude the .h

std::cout is a “stream”

The parameter passed to cout is passed using the << operator.

This is a slightly complicated point, but “<<“ returns std::cout in this case

Don’t worry if you don’t get it, but that means that you can send more data to cout just by chaining <<‘s

Makefiles

```
CXX=g++

ex01: ex02.cpp
	$(CXX) ex02.cpp -o ex02

clean:
	rm ex02 *~
```

Makefiles

```
CXX=g++

ex02: ex02.cpp
	$(CXX) ex02.cpp -o ex02

clean:
	rm ex02 *~
```

CXX is a variable, we assign g++ to it

Our compiler

ex02 and clean are “targets”

Things that can be “made”

ex02 is our program

clean tells the system how to “clean” our program

Which means get rid of the compiled version and any “intermediate build products” so we can build it again

Makefiles

```
CXX=g++

ex02: ex02.cpp
	$(CXX) ex02.cpp -o ex02

clean:
	rm ex02 *~
```

ex02: ex02.cpp

The left hand side tells “make” what is built

Examples:

- Executables
- .o files
  - (compiled C++ code that isn’t a whole program)

The right hand side tells “make” what is needed in order to build the left hand side.

This way, “make” can do dependency checking

- It checks what has been changed so it only needs to build what has been updated.

- This speeds up compile times on large projects

Makefiles

```
CXX=g++

ex02: ex02.cpp
	$(CXX) ex02.cpp -o ex02

clean:
	rm ex02 *~
```

The line under the “target” is what is run.

$(CXX) ex02.cpp -o ex02 becomes

g++ ex02.cpp –o ex02

And is run in the shell

(what you type into in the terminal)

All of this goes into a file called “Makefile”

If you type “make” in a directory, that will call its makefile.

Variables

```
#include <iostream>
int main(int argc, char **argv) {
        int a;    //Variable declaration
        int b;    //Variable declaration
        std::cout << "Declared a: " << a << std::endl;
        std::cout << "Declared b: " << b << std::endl;

        a = 0;   //Assignment
        b = 1;   //Assignment
        std::cout << "Assigned a:                " << a << std::endl;
        std::cout << "Assigned b:                " << b << std::endl;

        a = a + b;                  //Addition
        std::cout << "Added a + b: " << a << std::endl;

        a++;     //Increment
        std::cout << "Increment a: " << a << std::endl;

        //Post-Increment
        std::cout << "Post-Increment a: " << a++ << std::endl;
        std::cout << "Post-Increment a: " << a << std::endl;

        //Pre-Increment
        std::cout << "Pre-Increment a: " << ++a << std::endl;
        std::cout << "Pre-Increment a: " << a << std::endl;

        a = a + 10;                 //Addition
        std::cout << "a + 10:    " << a << std::endl;


        return 0;
}
```

Variables in C++ must be declared

Variables in C++ are strongly-typed.

Meaning they always have a type.

C++ deals a lot in low-level handling of raw memory.

As such,

Variables which have not been initialized have whatever was previously in memory in them.

Here are a few additions & increments just to demonstrate syntax

Note the difference between pre-increment and post-increment

Post-increment returns the value BEFORE the variable is incremented,

but the variable stores the incremented value.

Pre-increment works the same, but returns the value AFTER the variable is

incremented.

Loops - for

```
#include <iostream>

using namespace std;

int main(int argc, char **argv) {
	//Loops, our first control structure!!

	//for loop
	//	i = 0	- Initializer
	//	i < 10	- Comparison
	//	i++	- Increment
	for(int i = 0; i < 10; i++) {
		cout << "Inside for loop i: " << i << endl;
	}
}
```

Loops repeat a chunk of code until a stopping criterion is met

for has

- Initializer
  - Set up a variable used to count times through the loop
- Comparison
  - Compare if the variable has met the stopping condition
- Increment
  - Update the variable each time you go through the loop

for(;;) is often pronounced “forever”

Loops - while

```
#include <iostream>

using namespace std;

int main(int argc, char **argv) {
	//while loop
	int i = 0;
	//	i < 10		- Comparison
	while(i < 10) {
		cout << "Inside while loop i: “
				<< i << endl;
		i++;
	}
}
```

Loops repeat a chunk of code until a stopping criterion is met

while has

- Comparison
  - Compare if the variable has met the stopping condition

It is sort of similar to “for,” but you can think of “for” as often counting something.

While on the other hand waits for the condition to be met with no implication of counting

Loops - do

```
#include <iostream>

using namespace std;

int main(int argc, char **argv) {
	//do loop
	i = 0;
	//	i < 10		- Comparison
	do {
		cout << "Inside do loop  i: " << i << endl;
		i++;
	} while(i < 10);
}
```

While performs comparison before doing what is inside the loop

Do performs comparison after doing what is inside the loop

Loops – Are the braces needed?

```
#include <iostream>

using namespace std;

int main(int argc, char **argv) {
        for(int i = 0; i < 10; i++)
                cout << "Inside for loop i: " << i << endl;
}
```

The braces create a “block”

In C++ a “block” makes several lines of code syntactically equivalent to one line of code

This means that you can do things like this

You can totally get through this class without understanding this.

It’s just kind of neat to know.

Ex01 & Ex02 – Hello World!

- “Hello world!” is a traditional exercise to learn the basics of a new programming language.
  - C/C++ versions look similar

- Objectives
  - #include
  - main()
  - printf/std::cout
  - Return
  - Basic syntax
  - Invoking the compiler

Ex03 - Variables

- Variable declarations
- Uninitialized variables have unknown values
- Assignment
- Addition
- Pre/post-increment

Ex04 - Namespaces

- C++ uses namespaces
- You can “use” a namespace to remove “std::” (or similar)
- Namespaces prevent name collisions
  - There could be std::cout & special::cout or others. Which do you mean?
    - The namespace resolves this conflict
- In ROS we will use the ros:: namespace

Ex05 – Loops

- for
  - Generally counting up or iterating through lists
- while
  - While a condition is true
- do
  - Same as while, but comparison is evaluated at the end

- for(intializer, comparison, increment)
- while(comparison)

- {} forms a “block”
  - You can use 1 line after your loop (or if) syntax if not enclosed in block
  - A block really makes more than one line the same as 1 line

Ex06 – Functions

- A function with/without parameters
  - Formal parameters go onto the function
  - Actual parameters are used when a parameter is called

- Functions have return values
  - void means it returns nothing

Ex07 – Scoping

- In general, changing a variable's value inside a function does not change its value outside of the function
  - Even if they share a name
  - Even for parameters
  - References and pointers are special cases
    - A reference parameter **will** change
    - A pointer points to memory (by value), so if the contents of the memory change, they change everywhere in the program
- Globals are possible, but in general should be avoided

Ex08 – Header Files

- #include copies a header file into your file
- A function must have a prototype before it is used
  - But the full body is not needed before the usage
- Header files have #ifndef/#define macros to prevent you from copying their contents into your program twice
  - Causing problems like circular self-inclusion
- Header files normally have function/class prototypes in them

Ex09 – Implementation Files

- The implementations from a header (or for any function) can go into a .cpp file
- Prototypes in the header tell other files that the function exists
- You link your implementations together with g++
  - g++ a.cpp b.cpp c.cpp -o program

Ex10 – if / else if / else

- if(boolean) {
  thing
  }
- if(boolean) {
  thing
  } else {
  other thing
  }
- if(boolean) {
  thing
  } else if {
  other thing
  } else {
  other thing
  }

Ex11 – Types

- int – whole numbers
- float – decimal numbers
- C++ truncates floating-point numbers
  - Rounding functions are available
- Casting looks like this
  - (int) number
  - (float) number

Ex12 – Types

- Pointers start with * and point to memory

- You can point to the memory storing a variable
  - Prefix the variable name with & to get the pointer to that variable
  - Prefix the pointer with * to dereference the pointer, returning the value stored rather than the memory address

- The *new* keyword will allocate memory (dynamically)
  - It returns a pointer to data of the chosen type

- The *delete* keyword frees memory
  - You should definitely not delete pointers to memory that you still need!
  - You should definitely not delete pointers to statically allocated memory!
    - (As in when you say int *a = &b;)

Ex13 – Arrays

- Arrays are declared *type name[number];*
- Arrays are indexed *name[index];*
  - Indices start at 0 in c++
- Arrays have n items of type
  - The array index operator is actually just a special type of pointer syntax
  - You can allocate an array with *malloc* or *new[]*
    - You should use delete/free accordingly

Ex14 – Vectors

- Vectors are an STL (Standard Template Library) template
  - push_back/pop_back
- Templates use *iterators* for access
- *erase* is generally used for templates
- Vectors have a dynamic size, and do not need to be allocated all at once like arrays

Ex15 – Classes

- Classes hold related data together
- Initializers describe what data should be initialized to
  - (The part after the colon in the *constructor*)
- *Constructors* set up a class when an *object* is created
- Classes have *methods* which are like functions
- Classes have *access control*
  - Public
  - Private
  - Protected

Ex16 – Pulling it Together

- Class with a header and an implementation

Ex17 – Inheritance and Abstract Classes

- *Abstract* classes tell us what needs to be in *child* classes
- *Child* classes *inherit* things from *parent* classes
- Imaging this:
  - You have a class for a sensor that returns pictures
    - The parent class says that a picture is returned by a function
    - The child class may implement this for different cameras

Ex18 – Runtime Errors & Signed Variables

- The syntax in this example is correct, but there is an obvious, predictable, understandable flaw in the implementation.

Makefiles

- Run with the command “make”
- Named “Makefile”

- Variables
  - CXX, $(CXX)
- Targets
  - ex14, clean
- Dependencies
  - ex14.cpp
- Commands
  - $(CXX) ex14.cpp –o ex14
  - rm –f ex14 *~

```
1 CXX=g++
2
3 ex14: ex14.cpp
4   $(CXX) ex14.cpp -o ex14
5
6 clean:
7   rm ex14 *~
```