CS56 - C++ Review

Basic program

header file

```
#include <iostream>

int main(int argc, char** argv)
{
  std::cout << "Hello World!\n";
  return 0;
}
```

Basic program

```
#include <iostream>

int main(int argc, char** argv)
{
  std::cout << "Hello World!\n";
  return 0;
}
```

Function
Return type
Parameters
Print statement

C++ stuff to know: built-in types

```
#include <iostream>

int main(int argc, char** argv)
{
	int a = 1;
	bool b = true;
	char letter = ‘s’;
	double pi = 3.14;
	float t = 0.5;

	std::cout << "Hello World! " << a << “ “ << b << “ “ << t << std::endl;
}
```

C++ stuff to know: string types

```
#include <iostream>
#include <string>

int main(int argc, char** argv)
{
	const char* title = “a hard-coded string”;
	std::string s = title;
	s += “ we can put more words!”;
	std::string codeWord = s.substr(7,4);
	const char* cs = codeWord.c_str();
	std::cout << title << “ “ << codeWord << “ “ << cs << std::endl;
}
```

C++ stuff to know: control

```
#include <iostream>
#include <string>

int main(int argc, char** argv)
{
	std::cout << “Do you like jokes? (Y/N): ”;

	std::string response;
	std::getline(std::cin, response);

	if (response == “Y”) {std::cout << “Me too!\n”;}
	else {std::cout << “Ok, see you later\n”; }
}
```

C++ stuff to know: arithmetic operators

```
#include <iostream>
#include <cmath>

int main(int argc, char** argv)
{
	int a = 10;
	int b = 5;
	std::cout << “a/b = “ << a/b << std::endl;
	std::cout << “b/a = “ << b/a << std::endl;
	std::cout << “asin(-1) = ” << asin(-1) * 180/M_PI << std::endl;
	std::cout << “asin(-1) = ” << asin(-1.0000001) * 180/M_PI << std::endl;
}
// what is the output of this program?
```

C++ stuff to know: std::vector

```
#include <iostream>
#include <vector>
int main(int argc, char** argv)
{
	std::vector<int> values;
	for (int i = 0; i < 10; i++)
	{
		values.push_back(i);
	}
	int sum = 0;
	for (int i = 0; i < values.size(); i++)
	{
		sum += values[i];
	}
	std::cout << “Sum = “ << sum << std::endl;
}
```

```
#include <iostream>
#include <vector>

int main(int argc, char** argv)
{
	std::vector<int> values = {0,1,2,3,4,5,6,7,8,9};
	int sum = 0;
	for (int v : values) // needs -std=c++11
	{
		sum += v;
	}
	std::cout << “Sum = “ << sum << std::endl;
}
```

C++ stuff to know: classes

```
#include <iostream>

class A
{
public:
	A(int a) : _a(a)	{}
	virtual ~A()
	{
		std::cout << _a << “is deleted\n”;
	}
	void hello()
	{
		std::cout << “I am “ << _a << std::endl;
	}
private:
	int _a;
};
```

```
int main(int argc, char** argv)
{
	A a1(10);
	a1.hello();

	A a2(-4);
	a2.hello();
}
```

C++ stuff to know: classes in their own files

```
// header: A.h
#ifndef A_H_ // header sentry
#define A_H_
// can also write #pragma once

// Class definition
class A
{
public:
	A(int a);
	virtual ~A();
	void hello();
private:
	int _a;
};
#endif
```

```
// A.cpp, file with implementations!
#include “A.h” // include definitions!
#include <iostream>
A::A(int a)
{
	_a = a;
}
A::~A()
{
	std::cout << _a << “ is deleted\n”;
}

void A::hello()
{
	std::cout << “I am “ << _a << std::endl;
}
```

```
// main.cpp

#include “A.h”
int main(int argc, char** argv)
{
	A a1(10);
	a1.hello();

	A a2(-4);
	a2.hello();
}
```

C++ stuff to know: class inheritance

```
#include <iostream>
#include “A.h”

class B : public A
{
public:
	B(int a) : A(a) {}
	virtual ~B()
	{
		std::cout << “B is deleted\n”;
	}
	void hello()
	{
		std::cout << “Hello from B\n“;
	}
};
```

```
int main(int argc, char** argv)
{
	A a(10);
	a.hello();

	B b(-4);
	b.hello();
}

// What is the output of this program?
```

C++ stuff to know:
pass-by-reference vs pass-by-value

```
#include <iostream>
void foo(std::string s)
{
	s += “***”;
	std::cout << s << std::endl;
}

int main(int argc, char** argv)
{
	std::string apple = “apple”;
	foo(apple);
	std::cout << apple << std::endl;
}
// what is the output?
```

```
#include <iostream>
void foo(std::string& s)
{
	s += “***”;
	std::cout << s << std::endl;
}

int main(int argc, char** argv)
{
	std::string apple = “apple”;
	foo(apple);
	std::cout << apple << std::endl;
}
// what is the output?
```

C++ stuff to know: pass-by-reference vs pass-by-value

```
#include <iostream>
void foo(std::string s)
{
        s += “***”;
        std::cout << s << std::endl;
}

int main(int argc, char** argv)
{
        std::string apple = “apple”;
        foo(apple);
        std::cout << apple << std::;endl;
}
```

Makes a copy!

Modifies ‘apple’ !

```
#include <iostream>
void foo(std::string& s)
{
        s += “***”;
        std::cout << s << std::endl;
}

int main(int argc, char** argv)
{
        std::string apple = “apple”;
        foo(apple);
        std::cout << apple << std::endl;
}
```

C++ stuff to know:
pass-by-reference vs pass-by-value

```
#include <iostream>
void foo(std::string s)
{
	s += “***”;
	std::cout << s << std::endl;
}

int main(int argc, char** argv)
{
	std::string apple = “apple”;
	foo(apple);
	std::cout << apple << std::;endl;
}
```

```
#include <iostream>
void foo(const std::string& s)
{
	s += “***”;
	std::cout << s << std::endl;
}

int main(int argc, char** argv)
{
	std::string apple = “apple”;
	foo(apple);
	std::cout << apple << std::endl;
}
```

C++ stuff to know: pass-by-reference vs pass-by-value

Compiler says NO!

```
#include <iostream>
void foo(std::string s)
{
        s += “***”;
        std::cout << s << std::endl;
}

int main(int argc, char** argv)
{
        std::string apple = “apple”;
        foo(apple);
        std::cout << apple << std::endl;
```

```
#include <iostream>
void foo(const std::string& s)
{
        s += “***”;
        std::cout << s << std::endl;
}

int main(int argc, char** argv)
{
        std::string apple = “apple”;
        foo(apple);
        std::cout << apple << std::endl;
```

```
hello-ref2.cpp:5:10: error: passing ‘const string {aka const std::__cxx11::basic_string<char>}’ as ‘this’ argument discards qualifiers [-fpermissive]
        s += "***";
```

C++ stuff to know:
pass-by-reference vs pass-by-value

```
#include <iostream>
void foo(std::string& s)
{
	std::cout << s << “***” << std::endl;
}

int main(int argc, char** argv)
{
	foo(“apple”);
}
```

```
#include <iostream>
void foo(const std::string& s)
{
	std::cout << s << “***” << std::endl;
}

int main(int argc, char** argv)
{
	foo(“apple”);
}
```

C++ stuff to know:
pass-by-reference vs pass-by-value

```
#include <iostream>
void foo(std::string& s)
{
        std::cout << s << "***" << std::endl;
}

int main(int argc, char** argv)
{
        foo("apple");
}
```

Compiler says NO!

```
#include <iostream>
void foo(const std::string& s)
{
        std::cout << s << "***" << std::endl;
}

int main(int argc, char** argv)
{
        foo("apple");
}
```

```
hello-val2.cpp:10:15: error: cannot bind non-const lvalue reference of type 'std::__cxx11::string& {aka
std::__cxx11::basic_string<char>&}' to an rvalue of type 'std::__cxx11::string {aka
std::__cxx11::basic_string<char>}'
        foo("apple");
```

C++ stuff to know: pass-by-reference vs pass-by-value

```
#include <iostream>
void foo(std::string& s)
{
        std::cout << s << "***" << std::endl;
}

int main(int argc, char** argv)
{
        foo("apple");
}
```

Compiler says YES!

```
#include <iostream>
void foo(const std::string& s)
{
        std::cout << s << "***" << std::endl;
}

int main(int argc, char** argv)
{
        foo("apple");
}
```

C++ stuff to know: const

const -> constant -> “cannot change”

```
#pragma once

class A
{
public:
	A(int a);
	A(const A& a);
	virtual ~A();
	void hello() const;
	const C& getC() const;

private:
	int _a;
	C _c;
};
```

C++ stuff to know: const

const -> constant -> “cannot change”

```
#pragma once

class A
{
public:
        A(int a);
        A(const A& a);
        virtual ~A();
        void hello() const;
        const C& getC() const;

private:
        int _a;
        C _c;
};
```

Promise that method does not change parameter a

C++ stuff to know: const

const -> constant -> “cannot change”

```
#pragma once

class A
{
public:
        A(int a);
        A(const A& a);
        virtual ~A();
        void hello() const;
        const C& getC() const;

private:
        int _a;
        C _c;
};
```

Promise that method does not change instance of A on which method is called

C++ stuff to know: const

const -> constant -> “cannot change”

```
#pragma once

class A
{
public:
        A(int a);
        A(const A& a);
        virtual ~A();
        void hello() const;
        const C& getC() const;

private:
        int _a;
        C _c;
};
```

Callers of getC() are not allowed to change the returned value!

C++ stuff to know: random number generation

```
#include <random> // the new way using C++11
#include <iostream>
#include <functional>

int main()
{
	std::default_random_engine generator(time(0));
	std::uniform_real_distribution<double> distribution(0,1);
	auto dice = std::bind(distribution, generator);

	for (int i = 0; i < 10; i++) // print 10 uniform random numbers [0,1]
	{
		std::cout << dice() << std::endl;
	}
	return 0;
}
```

C++ stuff to know: random number generation

```
#include <cstdlib> // the old way
#include <iostream>

int main()
{
	for (int i = 0; i < 10; i++) // print 10 uniform random numbers [0,1]
	{
		// watch out for integer divides!!
		double v = ((float)rand())/RAND_MAX;
		std::cout << v << std::endl;
	}
	return 0;
}
```

Summary

Regarding C++, you will be working with the following:

- built-in types: int, bool, float, double, char. (We will use double by default).
- string types: char*, const char*, std::string
- control: if/else, for, while
- relationship operators: >, <, \>=, <=, ==, !=
- arithmetic: +, -, ++, --, /, *, %, NaN, sin, cos, asin, acos, tan, atan2
- input/output using iostreams (std::cout, std::cin)
- std::vector (and arrays sometimes)
- classes: methods and member variables, inheritance (public, private, protected keywords), virtual methods, destructors, constructors
- pointers: new/delete (pointers will be rare in this course)
- functions: return types and parameters. (99% of your code for this class should be uses classes!), all logic will be in classes)
- header and source files
- pass-by-reference vs pass-by-value
- const parameters, const return types and const methods
- random number generation