C++ Exam Review

Fall 2021

Topics Covered: C++ examples, functions, pass by value vs reference, loops and conditionals, file I/O, data types, compilation and linking.

Contents

1 C++ Examples and Key Concepts

1.1 Variable Declaration and Initialization

When an object is declared as a certain type we allocate storage for it in memory. If the object is atomic, the default initialization only *reserves the memory*, it doesn't set the value:

```cpp
#include <iostream>

int main() {
  int a;
  int b, c;
  c = a + b;
  std::cout << "c = " << c << std::endl;
}
```

```
$ g++ -std=c++11 variables.cpp -o variables
$ ./variables
c = 32767
```

You can declare several variables at once, while initializing one.

```cpp
#include <iostream>

int main() {
  int a = 1;
  int b, c, d, e = 10;
  b = 2;
  c = a + b;
  std::cout << "c = " << c << std::endl;
  std::cout << "d = " << d << std::endl;
  std::cout << "e = " << e << std::endl;
}
```

```
c = 3
d = 3540645
e = 10
```

When we initialize a static array, each element of the array is **default-initialized**.

1.2 Truncation and Casting

Although C++ is strongly typed, it allows implicit conversions between similar types:

```cpp
#include <iostream>

int main() {
  int a, b;
  a = 2.7;
  b = 3;
  int c = a + b;

  std::cout << "c = " << c << std::endl;
}
```

```
$ g++ -std=c++11 -Wall -Wconversion -Wextra variables.cpp -o variables
variables.cpp: In function ‘int main()’:
variables.cpp:6:5: warning: conversion to ‘int’ alters ‘double’ constant value [-Wfloat-conversion]
   a = 2.7;
     ^
$ ./variables
c = 5
```

What is happening here? When calling `a = 2.7;` we get the same result as if calling `a = (int) 2.7;`. This works the other way around as well:

```cpp
#include <iostream>

int main() {
  int a = 2;
  float b = 3.1415;
  double c = 1.61803398874989484820458683436563;

  int int_abc = a*b*c;
  float fl_abc = a*b*c;
  double db_abc = a*b*c;

  std::cout.precision(17)
  std::cout << "int:    " << int_abc << std::endl;
  std::cout << "float:  " << db_abc << std::endl;
  std::cout << "double: " << fl_abc << std::endl;

}
```

```
int:    10
float:  10.166107538970969
double: 10.166107177734375
```

Why are the float and double outputs different?

1.3 Undefined Behavior

There are a few instances where something *seems* to run correctly, but can break suddenly.

```
#include <iostream>

int main() {
  int a[3]; // Array has 3 elements

  a[0] = 0;
  a[1] = a[0] + 1;
  a[2] = a[1] + 1;
  a[3] = a[2] + 1; // Out of bounds access

  std::cout << "a[0] = " << a[0] << std::endl;
  std::cout << "a[1] = " << a[1] << std::endl;
  std::cout << "a[2] = " << a[2] << std::endl;
  std::cout << "a[3] = " << a[3] << std::endl;

  a[10000] = 5;
  std::cout << "a[10000] = " << a[10000] << std::endl;
}
```

```
a[0] = 0
a[1] = 1
a[2] = 2
a[3] = 3
Segmentation fault (core dumped)
```

Compilers are not required to diagnose undefined behavior... and the compiled program is not required to do anything meaningful.

1.4 Pre vs Post Operators

In C++, the calls `a++` and `++a` return different values, even though they modify a in the same way. A pre-increment (`++a`) increments a and then returns the value. A post-increment (`a++`), returns the original a before incrementing.

```
#include <iostream>

int main() {
  int a = 1;
  std::cout << "             a: " << a   << std::endl;
  std::cout << "return of a++: " << a++ << std::endl;
  std::cout << "             a: " << a   << std::endl;
  std::cout << "return of ++a: " << ++a << std::endl;
  std::cout << "             a: " << a   << std::endl;
  return 0;
}
```

Output:

```
            a: 1
return of a++: 1
            a: 2
return of ++a: 3
            a: 3
```

1.5 Conditionals

Conditionals evaluate a boolean expression:

```cpp
#include <iostream>

int main() {
  int cups_of_coffee_today = 3;
  int cur_hour = 10;
  int unanswered_piazza_questions = 25;

  bool make_more_coffee = false;
  bool make_even_more_coffee = false;

  if(cups_of_coffee_today < 5 && cur_hour < 12 && unanswered_piazza_questions > 10){
    make_more_coffee = true;
  }
  if(make_more_coffee && unanswered_piazza_questions > 24){
    make_even_more_coffee = true;
  }
  std::cout << "More coffee? " << (make_more_coffee ? "Yes" : "No") << std::endl;
  std::cout << "Even more coffee? " << (make_even_more_coffee ? "Yes" : "No") << std::endl;
}
```

```
More coffee? Yes
Even more coffee? Yes
```

Be careful with brackets:

```cpp
#include <iostream>

int main() {
  float grade = 0.75;

  if(grade > 0.90);
    std::cout << "Your grade is above 0.90" << std::endl;
    std::cout << "Congrats on the A!" << std::endl;

  if(grade > 0.80)
    std::cout << "Your grade is above 0.80" << std::endl;
    std::cout << "Congrats on the B!" << std::endl;

  if(grade > 0.70){
    std::cout << "Your grade is above 0.70" << std::endl;
    std::cout << "You passed!" << std::endl;
  }
}
```

```
Your grade is above 0.90
Congrats on the A!
Congrats on the B!
Your grade is above 0.70
You passed!
```

Be careful with paranthesis:

```cpp
#include <iostream>

int main() {
  std::cout << (true && !(false || true)) << std::endl; // False
  std::cout << (true && !false || true)  << std::endl;  // True
}
```

Remember switch statements (from the maze homework)? The argument of switch must be an integral type (`int`, `char`, etc). However, it does work with `enum`'s, like `enum directionleft, right, up, down;`. Remember that the switch will jump immediately to the corresponding `case`/`default`. However, you need to be careful to remember your `break` statements!

```
#include <iostream>

int main() {
  char grade = 'D';
  switch (grade) {
    case 'A': std::cout << "Fantastic!\n"          ; break;
    case 'B': std::cout << "Great work!\n"         ; // Missing break;
    std::cout << "Jumping to a case should skip me" << std::endl;
    case 'C': std::cout << "Nice!\n"               ; break;
    case 'F': std::cout << "See me after class.\n" ; break;
    default: std::cout << "Unknown grade\n";
  }
}
```

```
Unknown grade
```

If we set `char grade = 'B'` note what happens:

```
Great work!
Jumping to a case should skip me
Nice!
```

1.6 Loops

You don't need all of the parts of a for loop:

```
#include <iostream>

int main() {
  for(int i = 1; i < 100;){
    std::cout << i << std::endl;
    i = i * 3 - 1;
  }
}
```

```
1
2
5
14
41
```

Modifying the loop variable can cause bugs:

```
#include <iostream>

int main() {
  int i = 0;
  int j = 0;
  for(i = 1; i < 10; i++){
    if ( j++ > 2) break;
    i = 5;
  }
  std::cout << i << std::endl;
}
```

```
6
```

1.7 Scope

We *can't* access variables from narrower scopes, but we *can* access variables from wider scopes.

```
#include <iostream>

int main() {
  int n = 0;
  { int n = 1; }                             // Narrow scope
  std::cout << "n = " << n << std::endl;
  std::string n = 2;
  std::cout << "n = " << n << std::endl;
  {
    int n = 3;                               // Wide scope
    { std::cout << "n = " << n << std::endl; }
  }
}
```

```
n = 0
n = 2
n = 3
```

If we try the following:

```
#include <iostream>

int main() {
  for(int i = 0; i < 3; i++){
    int j = i;
  }
  std::cout << "i = " << i << std::endl;
  std::cout << "j = " << j << std::endl;
}
```

```
maxfit@rice03:~/Documents/scratch$ g++ test.cpp -o a.out
test.cpp: In function ‘int main()’:
test.cpp:7:26: error: name lookup of ‘i’ changed for ISO ‘for’ scoping [-fpermissive]
   std::cout << "i = " << i << std::endl;
                          ^
test.cpp:7:26: note: (if you use ‘-fpermissive’ G++ will accept your code)
test.cpp:8:26: error: ‘j’ was not declared in this scope
   std::cout << "j = " << j << std::endl;
```

However, if we declare our variables before the loop (wider scope):

```
#include <iostream>

int main() {
  int i,j;
  for(i = 0; i < 3; i++){
    j = i;
  }
  std::cout << "i = " << i << std::endl;
  std::cout << "j = " << j << std::endl;
}
```

```
i = 3
j = 2
```

What happens if we re-declare i and j in a narrower scope?

```
#include <iostream>

int main() {
  int i = -1;
  int j = -2;
  for(int i = 0; i < 3; i++){
    int j = i;
  }
  std::cout << "i = " << i << std::endl;
  std::cout << "j = " << j << std::endl;
}
```

```
i = -1
j = -2
```

Don't forget about do-while loops, they will **always execute at least once**.

```
do {
  // loop body - this will always get run at least once!
} while (expression);
```

2 Functions

Void functions do not need a `return` call, but can have one.

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

void print(std::string str){
  std::cout << str << std::endl;
}

void dont_print(std::string str){
  bool sky_is_blue = true;
  if(sky_is_blue) return;
  std::cout << str << std::endl;
}

int main() {
  print("Wow! Just like Python!");
  dont_print("Don't print this please!");
}
```

```
Wow! Just like Python!
```

The order of functions in your file matters:

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

int main() {
  this_function_hasnt_been_defined_yet();
}

void this_function_hasnt_been_defined_yet(){
  std::cout << "Or has it..." << std::endl;
}
```

```
t.cpp: In function ‘int main()’:
t.cpp:5:40: error: ‘this_function_hasnt_been_defined_yet’ was not declared in this scope
   this_function_hasnt_been_defined_yet();
```

To fix this, declare a function prototype:

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

void this_function_hasnt_been_defined_yet();

int main() {
  this_function_hasnt_been_defined_yet();
}

void this_function_hasnt_been_defined_yet(){
  std::cout << "Or has it..." << std::endl;
}
```

```
Or has it...
```

Now, if we define these function prototypes/implementations in `.hpp`/`.cpp` files we need to take care when linking/compiling, see Section 6.

Make sure you respect the arguments and their types:

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

void this_function_returns_nothing(){
  return 1;
}

int this_function_returns_an_int(){
  std::string a = "3.14159265358979";
  return a;
}

void this_function_takes_an_int(int arg){
  return;
}

int main() {
  this_function_returns_nothing();

  int a = this_function_returns_an_int();

  std::string b = "This is a string";
  this_function_takes_an_int(b);
}
```

```
t.cpp:12:31: error: cannot convert ‘std::__cxx11::string {aka std::__cxx11::basic_string<char>}’
to ‘int’ for argument ‘1’ to ‘void this_function_takes_an_int(int)’
   this_function_takes_an_int(b);
                               ^
t.cpp: In function ‘void this_function_returns_nothing()’:
t.cpp:16:10: error: return-statement with a value, in function returning ’void’ [-fpermissive]
   return 1;
          ^
t.cpp: In function ‘int this_function_returns_an_int()’:
t.cpp:21:10: error: cannot convert ‘std::__cxx11::string {aka std::__cxx11::basic_string<char>}’
to ‘int’ in return
   return a;
```

However, implicit casting can occur when values are of compatible types:

```
#include <iostream>

double cast_to_double(int a){
  return a;
}

int cast_to_int(double b){
  return b;
}

int main() {
  int b = 5;
  double c = cast_to_double(b);
  std::cout << c << std::endl;

  //                          1111111
  //                 1234567890123456
  double d = c - 0.000000000000001;
  double e = c - 0.0000000000000001;
  std::cout << "d " << cast_to_int(d) << std::endl;
  std::cout << "e " << cast_to_int(e) << std::endl;
}
```

```
5
d 4
e 5
```

2.1 Pass by Value vs Reference

Be careful of pass by value vs reference. Pass be reference is faster (doesn't have to copy the object) but may have unintended consequences if you didn't mean to modify the argument.

```
#include <iostream>

void pass_by_ref(int& a, int b){
  a += b;
  std::cout<< "a in ref = " << a << std::endl;
}

void pass_by_val(int a, int b){
  a += b;
  std::cout << "a in val = " << a << std::endl;
}

int main() {
  int a = 1;
  int b = -1;
  std::cout << "a before = " << a << std::endl;
  pass_by_val(a,b);
  std::cout << "a middle = " << a << std::endl;
  pass_by_ref(a,b);
  std::cout << "a after  = " << a << std::endl;
}
```

```
a before = 1
a in val = 0
a middle = 1
a in ref = 0
a after  = 0
```

2.2 Constant Reference

Marking an argument as constant makes it so the compiler throws an error when a function modifies that argument. Passing by constant reference gives us the benefit of not having to copy without unintended side effects of accidental modification.

```cpp
#include <iostream>

void pass_by_const_ref(const int& a, int b){
  a += b;
  std::cout<< "a in ref = " << a << std::endl;
}

int main() {
  int a = 1;
  int b = -1;
  std::cout << "a before = " << a << std::endl;
  pass_by_const_ref(a,b);
  std::cout << "a after  = " << a << std::endl;
}
```

```
main.cpp: In function ‘void pass_by_const_ref(const int&, int)’:
main.cpp:4:4: error: assignment of read-only reference ‘a’
  a += b;
```

2.3 Trickier Example

Watch the pass by reference and post-increment:

```cpp
#include <iostream>

int func_a(int& foo){
  foo++;
  return foo + 10;
}

int func_b(int bar){
  return bar++/2;
}

int func_c(int& inp){
  return func_b(func_a(inp));
}

int main() {
  int a = 5;
  std::cout << "Returns: " << func_c(a) << std::endl;
  std::cout << "a = " << a << std::endl;
}
```

```
8
6
```

3 File I/O

Writing to a file:

```cpp
#include <iostream>
#include <fstream>

int main() {
  std::string file_name = "hello.txt";
  std::ofstream f;
  f.open(file_name.c_str()); // Don't need c_str() if using -std=c++11 in compilation
  if (f.is_open()){
    f << "Hello!" << std::endl;
    f.close();
    std::cout << "Wrote to the file" << std::endl;
  }
  else {
    std::cout << "Failed to open file" << std::endl;
  }
}
```

```
Wrote to the file
```

Reading from a file `array.txt`:

```
2 2
0 0 1
0 1 2
1 0 3
1 1 4
```

```cpp
#include <iostream>
#include <fstream>

int main() {
  std::ifstream f("array.txt");
  int arr[5][5];

  if (f.is_open()) {
    int nif, njf;
    f >> nif >> njf;
    if (nif > 5 or njf > 5) {
      std::cout << "Not enough storage available" << std::endl;
      return 0;
    }
    int ni, nj, v;
    while ( f >> ni >> nj >> v){
      arr[ni][nj] = v;
    }
    f.close();
  }
 }
```

4 Types

Integer types:

Type	Memory (bits)	Range	# of Values Represented
Char	8	-127 to 127	256
Unsigned Char	8	0 to 255	256
Short	16	-32768 to 32767	O(65k)
Unsigned Short	16	0 to 65,535	O(65k)
Int	32	-2,147,483,648 to 2,147,483,647	O(4B)
Unsigned Int	32	0 to 4,294,967,295	O(4B)
Long	64	-(2^{63}) to (2^{63})-1	O(1.8×10^{19})
Unsigned Long	64	0 to 18,446,744,073,709,551,615	O(1.8×10^{19})

Note that you **do not** need to memorize these exact values, rather, learn the number of bits and the approximate size (e.g. an unsigned int has a maximum value of around 4 billion, signed int 2 billion, a short 32 thousand, etc). If you forget these values, you can always use the number of bits! If an integer datatype has n bits then the signed type will cover $[-2^{n-1}, 2^{n-1} - 1]$ and the unsigned type will cover $[0, 2^n - 1]$.

Floating point types:

Type	Memory (bits)	Precision	# of Accurate Digits
Float	32	$\sim 10^{-8}$	~ 8
Double	64	$\sim 10^{-16}$	~ 16

Remember, when working with floating point types we need to be careful with comparisons:

```cpp
#include <iostream>

int main() {
  float a = 0.0415926
  float b = 3.1
  if(a+b == 3.1415926){
    std::cout << "Have some pi!" << std::endl;
  }
  else{
    std::cout << "No pi ):" << std::endl;
  }
}
```

```
No pi ):
```

How would you represent a:

- Greyscale pixel? (Char)
- Cost of the Apollo space program, $25B? (Long)
- 28 factorial (28!) ? (None of the above)
- Speeds in the range [-100,100] m/s, to 5 decimal places of accuracy? (Float)
- The speed of light (2.9979×10^8), accurate to 1 decimal place? (Double)

5 Static Arrays

Static arrays are *not* the same as a single value of that type:

```cpp
#include <iostream>

int main() {
  double a[10];
  double b = 5.5;
  a = b;
}
```

```
test1.cpp: In function ‘int main()’:
test1.cpp:6:5: error: incompatible types in assignment of ‘double’ to ‘double [10]’
   a = b;
```

Static arrays pass the *memory address* by value:

```cpp
#include <iostream>

void zero(int a[10], int c[10]){
  a = c;
  std::cout << "a[0] in zero = " << a[0] << std::endl;
}

void zero_zero(int a[10]){
  a[0] = 0;
}

void zero_zero_zero(int b[10][10]){
  b[0][0] = 0;
}

int main() {
  int a[10], c[10];
  a[0] = 1000;
  c[0] = -1;

  zero(a,c);
  std::cout << "a[0] after zero = " << a[0] << std::endl;

  zero_zero(a)
  std::cout << "a[0] after zero zero = " << a[0] << std::endl;

  int b[10][10];
  b[0][0] = 1000;
  zero_zero_zero(b);
  std::cout << "b[0][0] after zero zero zero = " << b[0][0] << std::endl;
}
```

```
a[0] in zero = -1
a[0] after zero = 1000
a[0] after zero zero = 0
b[0][0] after zero zero zero = 0
```

6 Compilation

`main.cpp`:

```cpp
#include <iostream>
#include "sum.hpp"
 int main() {
  double a = 2, b = 3;
  double c = sum(a,b);
  std::cout << "c = " << c << std::endl;
}
```

`sum.hpp`

```cpp
double sum(double a, double b);
```

`sum.cpp`

```cpp
#include "sum.hpp"

double sum(double a, double b) {
  double c = a + b;
  return c;
}
```

Don't forget that we have to compile with *all* `.cpp` files! If we just compile `main.cpp` then the preprocessor will go and find `sum.hpp`, but won't be able to find the definitions!

```
$ g++ -Wall -Wextra -Wconversion main.cpp -o sum
/tmp/ccc0E8Sd.o: In function ‘main’:
main.cpp:(.text+0x39): undefined reference to ‘sum(double, double)’
collect2: error: ld returned 1 exit status
```

```
$ g++ -Wall -Wextra -Wconversion main.cpp sum.cpp -o sum
$ ./sum
c = 5
```

7 Review Questions

These are for you to practice your C++, they are **not** an indicator of difficultly or length of the actual exam.

1. What is happening here?

```cpp
#include <iostream>

unsigned char prod(unsigned char first, unsigned char second){
  return first * second;
}

int main() {
  unsigned char a = 24;
  unsigned char b = 11;
  unsigned char c = prod(a,b);
  if(c > 200){
    std::cout << "24*11 > 200? Well, you're not wrong!" << std::endl;
  }
  else{
    std::cout << "24*11 < 200? Math seems a little off today!" << std::endl;
  }
}
```

```
24*11 < 200? Math seems a little off today!
```

2. What value of n would you choose?

```cpp
#include <iostream>

int main() {
  int n = ?
  if(n%2 == 0 && n%3 == 0 && (n > 13 || n < 8) && n < 15){
    std::cout << "You win - no more 211 homeworks!" << std::endl;
  }
  else{
    std::cout << "You lose - an even harder trusses homework has been assigned!" << std::endl;
  }
}
```

3. The following code has some undefined behavior, can you find it? Once you've fixed that you should be seeing a tie. How can you modify the function to get the other two outputs (one at a time)?

```cpp
#include <iostream>

int what_is_the_best_programming_language(int c){
  return c++;
}

int main() {
  int c, python = 0;
  python = what_is_the_best_programming_language(c);
  if(python > c){
    std::cout << "Python!" << std::endl;
  }
  else if(python == c){
    std::cout << "It's a tie!" << std::endl;
  }
  else{
    std::cout << "C++!" << std::endl;
  }
}
```

4. Write a simple C++ program that takes in an input filename and an integer value x from the command line. The input file consists of a series of floating point values (one per line). Find the largest entry in the file that is less than x, the smallest entry in the file that is greater than x, and the absolute difference between these two values. I.e. if the input file has values $[5.001, -1.14, 2.7, 11.0]$ and $x = 5$ then the output would be:

```
x = 5 is in the range [2.7, 5.001], width = 2.301
```

But with $x = 20$ or $x = -2$ then:

```
x = 5 is not in the range of the input data.
```

A Extra Review Material

If you'd like more to review, feel free to check out the notes from 2019. Note that the material covered on that year's exam was slightly different (see Appendix B for a note on what is not being covered this year).

B Not Included on the Exam

The following material was covered in the lectures, but will not be on the exam. We have plenty of C++ "basics" to test you on without going into the following topics:

- Heap and memory allocation. Covered in lectures 11 and 17.
- Vectors, Sets, and Maps (std library containers). These were covered in lecture 13.
- Makefiles. These were covered in lecture 14.
- The Boost libaray. This was covered in lecture 15.
- Object oriented programming (OOP), or classes in C++. This was covered in lectures 16/17.

Note that these lectures built upon the C++ fundamentals that we will be testing you on, but the above explicitly listed topics will not be on the exam.