Survival C++ for Java Programmers

This guide assumes you have AP CS Java experience or CS161 credit in the Java programming language. It covers the biggest differences between Java and C++ and should be sufficient to help you get up and running in CS162 at Chemeketa Community College.

It is NOT exhaustively coverage of C++, just a guide to getting started and avoiding the most common issues for newcomers to the language. It focuses on material covered in CS161 - it does not detail differences between C++ and Java Object Oriented Programming (which is typically covered in the first part of CS162).

Make sure to open the document outline for easier navigation.
View** menu**, Show Document Outline

If you have time and want a more full tour, work through the content of a recent version of 161:
https://computerscience.chemeketa.edu/courses/cs161/202220/online/

If you can't find something in this guide, try:
https://www.learncpp.com/
https://stackoverflow.com/

General Advice

The assumption in C++ is generally to trust the programmer and to not complain when the programmer asks to do something that may not make sense. Things that would produce run time errors in Java (you asked for index 10 in a 5 element array) will silently produce weird results in C++.

Compile your code with as many warnings on as possible. Warnings are how C++ compilers tell you, "I am pretty sure you are doing something dumb, but I will let you try it." Treat all warnings as serious issues and fix them immediately. If you use the Chemeketa project template you will automatically have most warnings turned on.

If you are compiling from the command prompt, use -Wall and -Wextra to turn on many of these warnings:

```
> g++ file.cpp -Wall -Wextra -o program.exe
```

Tools

Visual Studio Code

VSCode is the IDE we use at Chememeta. There are many good C++ IDEs, but I encourage you to setup VSCode as provided code samples will be set up assuming you are using it.

https://computerscience.chemeketa.edu/guides/vscode-setup/

Command Prompt

You should be familiar with how to navigate at the command prompt (terminal) and build and run code from the command prompt.

Command Line Quick Reference

Demonstration:
https://www.youtube.com/watch?v=BClTBmTFgk8

Basic Types And Assignment

int, double, char and bool are what you will use for most jobs

C++ uses the same primitive types as Java: int, char, bool (instead of boolean), float. Integers can also be specified in different sizes (number of bits, which determines the min/max value you can store) and as unsigned.

```
int x;              //x is at least 16, probably 32 bits
double y;           //y is a 64 bit representation of a decimal number
bool b;             //b holds true or false
```

```
//Examples of others
unsigned int x;     //x is 32 or 64 bits and only represents positive values
short x;            //x is probably 16 bits
```

ALWAYS initialize variables

C++ does not automatically initialize variables for you. Instead, they have the value of whatever bits were in memory where the variable was placed. You are probably very used to Java setting all int's and double's to be 0 if you don't bother initializing them.

```
int x;              //x is who knows what?!?!?!? BAD BAD BAD
int x = 0;          //correct
double x = 0;       //correct
```

Constants

Constants are declared with **const** as part of the data type instead of final. They must be initialized when declared:

```
const int MAX_SCORE = 100;
MAX_SCORE = 20;             //illegal, it is const and can't be modified
```

Conversions

C++ is perfectly happy to take a wider type and convert it to a more narrow type. E.g. to take a double and store it as an int. Extra information (like decimal values) will simply be discarded. This happens when you do assignments, or if you pass a value to a function.

If you have warnings on, doing a narrowing conversion will produce a warning. To get rid of that warning, you can do a cast to announce that yes, you know what you are doing and want the conversion. The C++ syntax for that is **`static_cast<`*`NewType`*`>(`*`value`*`)`**. You can also use static_cast to force math to be done as integer or floating point.

```
double x = 5.9;
```

```
int i = x;              //i is now 5 - maybe a warning

someFunction(x);        //someFunction expects an int - will be passed 5
                        // may give warning

int i = static_cast<int>(x);
//i is 5, but no warnings as we requested conversion

int j = 2;
double e = j;           //no problem - e is 2.0

//Use cast to ensure floating point math even though j is an int
double result = static_cast<double>(j) / 3;
```

Input / Output

C++'s standard tools for reading and printing from the console are **std::cout** and **std::cin**. The are in the <iostream> library.

Cout should be followed by one or more things to print, each separated with **<<**. **std::endl** inserts a new line. Cin should be followed by one or more variables to be read into , each separated with **>>**.

```
#include <iostream>

int main() {
   int x;

   std::cout << "Enter a number: ";
   std::cin >> x;

   std::cout << "You entered " << x << std::endl;
}
```

Rather than use the full names, it is common to include the std namespace. This allows us to skip std:: in front of cout, cin and endl.

```
#include <iostream>

using namespace std;

int main() {
   int x, y;

   cout << "Enter two numbers: ";
   cin >> x >> y;

   cout << "Their sum is " << (x + y) << endl;
}
```

When we start reading into a variable, any whitespace in the buffer is skipped (think of >> as “fast forward past whitespace). Then cin tries to read into the variable. It will read as much input as makes sense, stopping at whitespace or a character that is not appropriate for the type.

```
int x;
cin >> x;
//Input "  12 3" => x is 12; " 3" is left in buffer
//Input "  12x" => x is 12; "x" is left in buffer
//Input "  12.2" => x is 12; ".2" is left in buffer
```

IO failure

If there is no valid input to read, cin silently fails and will not read anything else. Any variables are left unmodified. Unless you reset the input stream, any further attempts to use cin are ignored.

```
int x, y;
cin >> x;
cin >> y;
//Input "  12 q" => x is 12; y fails and is left alone
//Input "  q 12" => x fails; we do not even try to read in y!!!
```

IO Formatting

If you need to make something pretty, watch the Formatting video from week 5 of CS161.
http://faculty.chemeketa.edu/ascholer/cs161/content05.html

FileIO

File input/output works just like reading/writing with cin and cout. If you need to do FileIO, see:
http://faculty.chemeketa.edu/ascholer/cs161/content05.html

Strings

C++ has a **string** class in the <string> library. It is similar to a Java string with some key differences detailed below.

Like cin and cout, you either need to call it **std::string** or have “using namespace std;” in your code file.

For more examples of using string functions, see:
http://faculty.chemeketa.edu/ascholer/cs161/content05.html

[ ] is dangerous

If you use **[i]** and ask for a bad index in C++, you will get unpredictable behavior as you read/write memory outside the string. If you use **.at(i)** you get an exception when you ask for a bad index.

```
#include <string>

using namespace std;

int main() {
   string s = "hello";

   char c = s[50];          //Who knows what c will be???
   char c = s.at(50);       //An exception will be thrown
}
```

C++ strings use beginIndex, length

In Java, string functions tend to take (beginIndex, endIndex). C++ uses (beginIndex, length).

```
string s = "hello there";

string s2 = s.substr(6);        //s2 is "there"
string s3 = s.substr(0,2);      //s3 is "he"
```

```
string s4 = s.substr(6,3);      //s4 is "the”
```

C++ strings are mutable

In Java, a string is immutable. Any function you call on a string returns a new string with the modifications made. In C++, most string functions (other than substr) modify the string.

```
string s = "hello there";
s.erase(4);                 //s = "hell"
s = "AE";
s.insert(1, "BCD");         //s = "ABCDE"
```

You can’t concatenate numbers and strings

Use to_string( ) to turn ints or doubles into strings. Use stoi( ) or stod( ) to convert to numbers.

```
double c = 12.5;
string s = "Cost is $" + to_string(c);

string ageString = "23";
int age = stoi(ageString);
```

If you need more powerful formatting as you turn numbers into strings, check out **stringstream.** It is like Java’s StringBuilder.

Functions

In C++ functions do not have to be members of a class. **int main()** is the standard entry point for a program.

Declaration vs Definition

In C++, the **declaration** that says “this function will exist” and the **definition** that has the code for the function can and often will be seperate. We MUST declare a function before trying to use it. Then somewhere in the code we need to define it to actually provide the code.

To declare a function, you simply write the prototype with a ; behind it and don’t provide a body:

returnType name(parameters);

You do not have to separately declare functions, when you define a function you are also declaring it.

```
//declare the function
int doubleIt(int n);

int main() {
   //Must have already declared doubleIt - definition can come later
   int x = doubleIt(10);
}

//define the function we declared earlier
int doubleIt(int n) {
   return 2 * n;
}

//declare and define the function - could not use in main as it was not
//  declared at that point
int bar() {
   return 42;
}
```

Pass By Value vs Reference

In Java, primitives are stored as values while object variables store a **reference** - a link to the object's data. When you pass a primitive into a function, we copy its value and work with the copy. When you pass an object to a function we copy the reference and are thus working with the same object.

In C++ both objects can be stored as references or as values. And when you go to call a function, you have to specify for each parameter if you want to pass by **value** (a copy of the data) or pass by **reference** (a link to the original data).

The default is pass by value . To pass a parameter by reference, use **&** next to the data type, like "int& x". The way to read that is "x is a reference to an int". A reference type variable does not represent a new value - it represents a link to the variable that was passed into the function.

```
void foo(int x) {  //pass by value
   x = 0;    //no real effect... working with a copy
}

void bar(int& x) {  //pass by reference
   x = 0;    //we just set temp in main to 0
}

int main() {
   int temp = 10;
   foo(temp);      //temp passed by copy
   bar(temp);      //temp passed by reference
}
```

To see an animation of how this works, see:
https://goo.gl/XwEfLZ

Objects work the same way - we have to specify whether to pass by value or reference. If you don't use & your function gets a copy of the object.

```
void foo(string x) {  //pass by value
   x = "";    //no real effect... working with a copy
```

```
}

void bar(string& x) {  //pass by reference
   x = "";    //we just set temp in main to an empty string
}

int main() {
   string temp = "hello";
   foo(temp);      //temp passed by copy
   bar(temp);      //temp passed by reference
}
```

In general, you should pass primitives by value unless the function really needs to work with a reference to the original data. Objects, like strings, should be passed by reference to avoid the work of making a copy. If you want to promise that the reference will not be used to modify the original data, you can declare the parameter as **const**. This will make any attempt to modify it in the function a compile error. In the sample below, bar takes x as a **const string&** - any attempt to change x will be an error.

```
void foo(string& x) {
  char c = c[0];       //reads from x
  x = "";              //modifies x
}

void bar(const string& x) {
  char c = c[0];       //reads from x
  x = "";              //modifies x - compile error as x is const
}
```

Arrays

C++ arrays are simply blocks of contiguous memory. We can index into them like in Java with array[i], but they lack most other nice features of Java arrays:

- They do not know their size - no **.length**

- Nothing prevents you from going past either end of the array. You just end up reading or writing to memory that is not part of the array and get weird results/crash your program.
- You can't copy an array using =
- You have to put the [ ] after the name of the array (not after the type).

Initialization

Uninitialized arrays start with completely random values. You can use a list in { } to initialize them. If you initialize with just { } the array will start with all 0's or all

```
int scores[10];     //uninitialized - filled with random values

//initialize with values
string names[4] = {"Alex", "Maria", "Curtis", "Nancy"};

int ages[10] = {};  //empty initialization list will fill with 0's

ages[0] = 17;       //fine
cout << ages[50];   //unpredictable, may give random value, may crash
```

Arrays must be constant sized

Arrays can live on either the stack ("automatic" memory managed by the compiler) or the heap (memory you have to manage) - this document focuses on stack based arrays. Stack based arrays MUST have a size based on a constant, not a variable.

```
int x = 10;
const int LIST_SIZE = 10;

int list[x];            //illegal - variable size
int list2[10];          //literal constant
int list2[LIST_SIZE];   //OK - constant
```

Arrays and functions

Because arrays do not know their own size, we typically pass it into functions that work on an array as an additional parameter.

```
//take in an array and its size
void foo(int array[], int size) {
   for(int i = 0; i < size; i++)
       cout << i << endl;
}

int main() {
   int list[4] = {1, 2, 3, 4};
   foo(list, 4);       //pass array and size
}
```

Arrays are stored as a memory address where the data begins. If you use just the name of the array, you are asking for the memory address, not the contents of the array. You can see this by printing out an array name:

```
int list[10];
cout << list << endl;   //a memory address like 0x68fe20
```

When you pass an array into a function, you are passing its address. That means it works like a reference - anything you do to the array in the function works with the original data.

```
//array in this function is naming the same memory as list from main
void zeroOut(int array[], int size) {
   for(int i = 0; i < size; i++)
       array[i] = 0;
}

int main() {
   int list[4] = {1, 2, 3, 4};
   zeroOut(list, 4);
```

```
   //list is now {0, 0, 0, 0}
}
```

To prevent modification, you can declare that the data the array is using will be kept **const**.

```
void zeroFirst(const int array[]) {
   array[0] = 0;  //error array is const!
}
```

Vectors

If you are looking for something easier to use than a plain array, like Java's ArrayList, you want the C++ **std::vector** class. We cover them in CS162. But you are expected to be comfortable using plain old arrays before you move on to using vector.

Headers

C++ code too complex to fit into a single .cpp file is usually split into headers (.h) files and code files (.cpp).

The short version is:

- .h files are used to declare functions (and constants, classes and other data types).
- .cpp files are used to provide definitions.
- When we compile a program, each .cpp file gets compiled separately.
    - Each .cpp has to make sense on its own - we need to declare any functions it is going to use.
        - To get the needed declarations, .cpp files include .h files
- Once each .cpp file is compiled, the linker merges all the individual files. At this point, there can only be one definition for each function.
    - This means we can't have two .cpp files that each define the same function (provide a body)
        - It is OK if they both declare it, as long as only one defines it.

This video shows how we divide up a program into files:
https://www.youtube.com/watch?v=i9akyIe0DJU

I also recommend reading these pages for a more in depth description of the building process and how headers are used:

https://www.learncpp.com/cpp-tutorial/introduction-to-the-compiler-linker-and-libraries/
https://www.learncpp.com/cpp-tutorial/header-files/

Command Line

You should be comfortable moving around and compiling/running code from the command line.

This video shows you the basics:
https://www.youtube.com/watch?v=BCITBmTFgk8

This PDF has more detailed instructions including how to set up your PATH:
http://computerscience.chemeketa.edu/CSResources/CommandLineGuide.pdf