Migrating from Java to C++

Mckenna Cisler - AP Computer Science

This document provides information on the similarities and (mainly) the differences between Java and C++.

(Java syntax is red and generally on the left, while C++ is blue and generally on the right, unless they're not being compared)

Sectioned preceded by **(ADV)** are not essential information and can safely be ignored if you're looking for the basics.

This is a good source for any style/best practice questions you have:
   http://www.stroustrup.com/bs_faq2.html
This is a good API source (like the JavaDocs):
   http://en.cppreference.com/w/

Setting up a C++ Environment

The easiest way to get a C++ environment is to simply install an IDE like Eclipse for Java. There are a couple options for this (in decreasing order of popularity): See here also: http://www.learncpp.com/cpp-tutorial/05-installing-an-integrated-development-environment-ide/

Cross-Platform:

Code::Blocks - http://codeblocks.org/downloads/26

Eclipse with C++ - http://www.eclipse.org/downloads/packages/eclipse-ide-cc-developers/mars2

Visual Studio Code (with C++ extension) - https://code.visualstudio.com/ with https://marketplace.visualstudio.com/items?itemName=ms-vscode.cpptools

Windows:

Microsoft's Visual C++ (Visual Studio) - https://www.visualstudio.com/en-us/products/visual-studio-express-vs.aspx

OSX:

XCode with C++ extension - https://itunes.apple.com/us/app/xcode/id497799835; follow https://www.cs.drexel.edu/~mcs171/Wi07/extras/xCode_Instructions/index.html

Otherwise, you can try to install a compiler:
If you're on Linux, there is likely already one there - probably gcc or g++
Otherwise, for Windows see this guide (you'll end up using something called MinGW):
https://latedev.wordpress.com/2011/06/20/how-to-install-a-c-compiler-on-windows/

Basic Language Features

```
// these are not quite the same, but see "Making Importable Files" for details
import X                         #include <X> OR
                                 #include "X" // if X is a local .h file
```

// TYPES/VARIABLES

```
// C++ for the most part does variables the same as in Java, but it has
// recently added a keyword that allows automatic typing
// (similar to Javascript or Python)
// BUT, note that Java has done this for good reasons... use it sparingly
// INSTEAD OF:                                CAN DO:
int foo = 5;                                  auto foo = 5; // C++11
double fooD = 5.0                             auto fooD = 5.0;
int bar() { return foo }                      auto bar() { return foo } // C++14

// type casting in C++ is generally the same, but it is sometimes better to use
// another method, because traditional casting can have issues
// (see http://stackoverflow.com/a/1255015   OR
//      https://msdn.microsoft.com/en-us/library/c36yw7x9.aspx)
(char) someInt                                (char) someInt; OR
                                              static_cast<char> someInt; OR
                                              (ADV) dynamic_cast<char> someInt;

// Constant variables work as follows:
final X = val                                 const X = val // NOT #define X val

// NOTE:
// C++ programs use a keyword called "typedef" a lot to basically create aliases
// for given types to make code more readable.
// (you'll likely see things like size_t for int OR speed_t for double where
// the _t indicates (stylistically, not syntactically), a typedef)
// INITIAL TYPEDEF:                           FURTHER USAGE:
typedef int size_t;              ->           size_t size = 5;

// booleans are actually called bools in C++ (and are printed as integers):
boolean a = false;                            bool a = false;
System.out.print(a) // prints ‘false’         cout << a; // prints ‘0’ (vs. ‘1’)

// C++ doesn't typecast primitives predictably to strings, so don’t use '+' to
// concatenate the string version of an int and a normal string:
// use “<<” instead (ONLY) when you’re outputting to cout or another stream.
5 + " is an int"                              5 << " is an int"
```

// INPUT/OUTPUT

```
// for these you’ll have to use #include <iostream>
// (many people also get rid of std:: by doing "using namespace std" above
// somewhere (DON’T do it globally though)... this applies for many other
// std functions too)
System.out.print("X")                         std::cout << "X"
System.out.println("X")                       std::cout << "X" << '\n'; OR
                                              std::cout << "X" << std::endl;
```

```
                                                      // (ADV) the main difference here
                                                      // is that std::endl ENSURES
                                                      // that the output is actually
                                                      // output (i.e. written to a file
                                                      // if cin was actually a file
                                                      // stream) SO only use it if you //
                                                      specifically want this.
                                                      // (i.e. progress bar)

// For INPUT, the following should be good for general use (see note below):
// (using Scanner input = new Scanner(System.in) AND #include <iostream>):
entry = input.nextLine()                              getline(cin, entry)
                                                      // note lowercase 'l'
entry = input.next()                                        cin >> entry
int var = input.nextInt()                             int var; cin >> var;
double var = input.nextDouble()                       double var; cin >> var;

// NOTE on input in C++:
// It is more low-level than Java, and there is room for attacks like
// buffer overflow attacks, so be careful and use LOTS of validation
// for important projects.
// See here for a good overview: http://www.learncpp.com/cpp-tutorial/
//     5-10-stdcin-extraction-and-dealing-with-invalid-text-input/
// An IMPORTANT ISSUE to be aware of is that cin ALWAYS only takes one
// character in... so doing
cin >> chr1; cin >> chr2;
// will set chr1='a' and chr2='b' if the user enters "ab" initially,
// so cin >> chr2 WILL NOT ask the user for input, just use up what is left
// on cin. (This could be avoided with "cin.ignore(32767, '\n')" - see below)

// Validating input: C++ has several means to validate input you have
// because it is generally so raw (the closest in Java is exception catching)
// When trying to parse into an integer or similar (AFTER some operation),
// you can (and should) use these: (they ALSO apply to file streams - see below)
cin.good()          // non-zero (true) if stream is all okay
cin.bad()           // non-zero (true) if stream has had a fatal error
                    //     (reading past end of file, etc.)
cin.fail()          // non-zero (true) if stream has had a NON-fatal error
                    //     (not correctly reading in an int to an int var)
cin.clear()         // clears the above flags
cin.ignore(int n, char delim)    // generally used AFTER a bad or fail flag
                                 // was triggered and clear() was called.
                                 // ignores either n characters or up to
                                 // a delim char is found, whichever's first.
                                 // (Usually n is really large, i.e. 2^15)

// There are also some methods for seeing what type of input the given char is
// (the char is passed in here as an int, but passing a char in will work)
// NOTE: These are all checking if the given int ASCII values are digits, etc.
// so, for example, isdigit(1) is false while isdigit('1') is true.
isalnum(int) // true if letter or digit
isalpha(int) // true if letter
isdigit(int) // true if digit
```

```
ispunct(int) // true if NEITHER alphanumeric NOR whitespace
isspace(int) // true if whitespace
isxdigit(int) // true if hexadecimal digit (0-9, a-f, A-F)

// POINTERS (and REFERENCES)

// These are now VERY applicable in C++:
// (see http://www.learncpp.com/cpp-tutorial/67-introduction-to-pointers/ AND
//     http://eli.thegreenplace.net/2011/12/15/
//     understanding-lvalues-and-rvalues-in-c-and-c/)
// where intLocation is a POINTER to an int at some memory location:
int* intLocation = &someIntVal;
//     * is the "dereference" (go to the value) operator
//     & is the "address of" (where is it) operator

// You can increment/decrement pointers to move through memory:
intLocation++
// NOTE: the pointers move by the sizeof() the object, so in bytes this may be
// 2 or 4 for a short or long
// (arrays are actually just pointers to the first element, and you use the []
// to move through the array)

// ALSO, pointers can be null like in Java:
null                                         nullptr OR 0x0 OR NULL

// (ADV) “void” pointers also exist, but are different from null pointers
// (they’re basically pointers with no type.)
// As a result, they can refer to anything, but they need to be typecasted
// to some type before being dereferenced:
// DEFINITION                                USAGE
void* some_data                  ->          int val = *( (int*) some_data)

// REFERENCES are a special kind of pointer use:
// They are basically addresses (like normal pointers)
int& intRef = someIntVal; // someIntVal is NOT copied
// ...BUT, they are used like just normal variables:
cout << intRef;           // prints someIntVal
intRef = 5;               // now someIntVal == 5
intRef++;                 // now someIntVal == 6

// references are useful because:
//     - They CANNOT be nullptr
//     - The variable they reference CANNOT be CHANGED
//     - It HAS to be initialized when created (hold a value)
//     - They are easier to read than pointers
//     - They can be passed as function parameters INSTEAD of pointers for
//       changing a value in the function
//     - They can be returned by a function so that functions can
//       "be assigned values" and change the reference that was returned
//       (see setValues() )

// some parallels between pointers and references:
```

```
int* intPtr = &someIntVal                     int& intRef = someIntVal
cout << *intPtr                               cout << intRef
(*intPtr)++                                   intRef++
*intPtr = 5                                   intRef = 5
intPtr = &otherIntVal                         [IMPOSSIBLE]
intPtr = nullptr                              [IMPOSSIBLE]

void swap(int* x, int* y)                     void swap(int& x, int& y)
{                                             {
       int temp;                                     int temp;
       temp = *x;                                    temp = x; // store val at x
       *x = *y;                                      x = y;    // put y into x
       *y = temp;                                    y = temp; // put x into y
}                                             }

// (ADV)
// (with int[] vals defined)                  // RETURN a reference
double* setValues(int i)                      double& setValues(int i)
{                                             {
       return &vals[i];                              return vals[i];
}                                             }
// usage of setValues() - pointers are clunky here
int* loc = setValues(i);                      setValues(i) = 5;
*loc = 5;

// see http://www.tutorialspoint.com/cplusplus/cpp_references.htm for more

// IMPORTANT NOTE: basically all objects in C++ are PASSED BY VALUE (copied)
// in functions! The = used with NON-POINTERS is ALWAYS the copy operator!
//               The = used with POINTERS is the move/change reference operator!
// Use REFERENCES (see below) if you want to pass by reference
//    OR are passing OBJECTS (try NOT to use pointers here!)
// (note: using references implies you change function arguments, so add a
//  const specifier if you don't plan on changing them)
// (Arrays are a notable exception though... they are very primitive and still
//  use pointers extensively)
// EXAMPLE:
Obj b = a;   // a is (COPIED (even if Obj is a massive object)
             // note: this is a SHALLOW copy, so fields like strings are
             // just copied by pointing to the original strings (which is bad
             // if one tries to erase or change them)
Obj* c = &a // a is MOVED/REDIRECTED

// GENERALLY:
// - Use PASS/RETURN BY VALUE with [primitives AND when in doubt]
void printVal(int val);
int readVal();

// - Use PASS/RETURN BY ADDRESS with [built-in arrays AND passing actual pointers]
void changeArr(int* arr, int length);
int* getArr();

// - Use PASS/RETURN BY REFERENCE with [large objects* AND things you’re changing]
// NOTE: if you want to return an object, etc. defined INSIDE a method, you can
// make it static so it won't be destroyed after the function is done.
//   BE CAREFUL, though:
//   It will then be a kind of global variable within that function, and if
//   you use the function more than once it WON’T be automatically reassigned
//   (that line won’t even be executed!! For more,
//    see http://www.learncpp.com/cpp-tutorial/43-static-duration-variables/)
// *ULTIMATELY, just use RETURN BY VALUE here, because it turns out modern
// compilers will optimize memory so you aren’t actually COPYING everything:
// see http://stackoverflow.com/a/4643721/3155372
// IF YOU REALLY NEED TO, return a pointer and use Smart Pointers (see below)
void modifyObj(Obj& obj);
Obj& getNewObj() { static Obj obj = new Obj(); return obj; }; // getNewObj() will
                                                           // ALWAYS return the
                                                           // obj created the //
                                                           first time.

// IMPORTANT NOTE: As mentioned above, pointers are fairly primitive things,
// and as such C++ programmers tend to not want to use them (this is a general
// sentiment in C++ -- to move away from these more primitive things towards
// wrappers and redefinitions that work better and require less thought/
// maintenance. Using vector or array over basic arrays is an example of this)
// WITH THAT IN MIND, there are things called Smart Pointers which
// automatically delete themselves like Java would with garbage collection.
// USE THEM WHENEVER YOU CAN
// See https://msdn.microsoft.com/en-us/library/hh279674.aspx for more
// DEFINITION                                      USAGE
// if the ptr will only be referenced once         [just like normal pointers]
std::unique_ptr<Obj> ptr(new Obj());               .get() // to get a normal
// if the ptr will be used in many places                 // pointer
// (inside vector, etc.)
std::shared_ptr<Obj> ptr(new Obj());


// FUNCTIONS

// Because dereferencing pointers to change values (instead of assigning)
// happens in C++, you'll sometimes see this difference:
output = method(input)                       method(output, val)

// C++ allows default parameters (you can do find(val) OR find(val, 2) here):
// (the closest thing in Java is function overloading - C++ has that too)
int find(string val, int start) {};          int find(string val, int start=0)
int find(string val) { find(val, 0) };

// (ADV) Similarly to how ArrayLists can have any type, C++ allows you to create
// function templates that use the same syntax but can have any return or
// argument type (Java has this).
// (Here "T" is the convention for [generic class]... it could be anything)
// DEFINITION                                USAGE
template <class T>                     ->    sum(10, 12) OR
T sum(T x, T y) { return x + y; }                    sum<int>(10, 12) OR
                                             sum(10.0, 12.0) OR
```

```
                                              sum<double>(10.0, 12.0)

// (ADV) C++ allows you to define "inline" functions that are basically just
// copied to where they're executed instead of jumping around the code stored
// in memory to run the function code elsewhere.
// (Anyways, this is sometimes done automatically or even ignored by the compiler)
NOTHING                                       inline int add(int x, int y)
                                              { return x + y; }

// (ADV) Because functions are just pointers, they can be used like them!
// A "function pointer" is defined just like a function, but with
// (*[name]) instead of [name]
// This means they can also be used as function parameters just like
// they're defined, and called with arguments.
// (with int subtraction(int, int) defined...)
// DEFINITION                                 USAGE
int (*minus)(int, int) = subtraction    ->    int val = (*minus)(10, 5)

// IMPORTING AND NAMESPACES

// C++ REQUIRES forward definitions of functions, classes, and other
// global references BEFORE you use them:
int main() { a(); }                           int a();
int a() { ... }                               int main() { a(); }
                                              int a() { ... }

// MEMORY

// C++ mostly uses static (stack) memory, and we don't have to worry about it, but
// dynamic (heap) memory (allocating and deallocating memory allocated at runtime)
// is needed in some cases.
//     Although the new keyword SHOULDN'T be used all the time like in Java,
//     it is used here to DYNAMICALLY allocate sufficient memory for ONE or
//     an ARRAY of a given type, and returns a pointer to the first part.
int* foo = new int[5];

// Be careful, though: this may throw the bad_alloc exception (if there’s not
// enough memory, for example) - it can be avoided with the following, and a later //
check to see if foo==nullptr
// (make sure to #include <new> here)
int* foo = new (nothrow) int[5];

// ALSO, because you're dynamically allocating memory, you HAVE TO “delete”
// what you allocated so it can be used elsewhere. (Memory leaks occur otherwise)
// You can use "delete" or "delete[]" on a pointer to delete values or arrays
delete bar;  // WITH int* bar = new int(1);
delete[] foo;// WITH int* foo = new int[10];

// MISC

// (ADV) C++ defines "namespaces" to explicitly create categories of
// methods and variables, that can be referenced using the "::" operator
// (this is what we're doing with "using namespace std" or
// "using [some_namespace]::[some_variable/function]" )
// DEFINITION                                 USAGE
namespace a { int b = 2; }       ->           cout << a::b; OR
                                              using a::b; std::cout << b;
// ALSO, these can be expanded upon:
namespace a { [part 1] } // then later...
namespace a { [part 2] }

// RANDOM number generation’s a bit different - look at #include <random> for more
// rand() is from 0 to RAND_MAX (usually 2^15)
// ( be sure to #include <cstdlib> )
(int) (Math.random() * 100)                   rand() % 100
```

Strings

```
// C++ supports c-style strings. (But just use string - see below)
String str = "Hello"                          char str[] = {'H','e','l','l','o','\0'}
                                       OR char str[] = "Hello"
                                       OR char* str = “Hello”


// String Methods: (need to #include <string> first)
// Have to use "using namespace std" to access these without the std:: prefix
// See http://www.cplusplus.com/reference/string/string/
String X                                      string X
.length()                                     .size() OR .length()
.indexOf(str)                                 .find(str)
.charAt(i)                                    [i]
.substring(a, b)                              .substr(a, b)
.compareTo(str)                               .compare(str)
new = new String(old)                         new = old.assign(old)
Integer.parseInt(numStr)                      stoi(numStr)
Double.parseDouble(numStr)                    stod(numStr) // stof() for floats

// C++ has (some) MORE than Java!
NOTHING                                       .insert(i, str)
NOTHING                                       .erase(numToErase, start)

// starting at start, replace numToReplace chars from str (insert the rest)
NOTHING                                       .replace(start, numToReplace, str)

// NOTE: there is no .toString() in most things in C++
// (not even an address like in Java)
```

Basic Arrays

```
// definition is slightly different:
int[] arr = new int[3];                       int arr[3];
int[] arr = {1, 2, 3};                        int arr[] = {1, 2, 3}

// NOTE: Object arrays call the default constructor for each new element (they
// need one defined, unless you MANUALLY set them to nullptr)
```

```
        // NOTE: there is no .length -- length can only be found via (with int arr[3]):
        len = sizeof(arr) / sizeof(int);
        // BUT, this only works in the code right after arr is defined - if arr is
        // passed into a function, it "decays" into a simple pointer to the first element,
        // so it has no length reference.
        // (AS A CONSEQUENCE, these "decayed" arrays CANNOT be used in for-each loops)
        // NOTE: if you do go over the end of an array in C++, it just goes into the
        // memory past the end (you may get a segfault though)
        .length                                        NOTHING (keep track!)

        // NOTE: arrays can be dynamically allocated in c++11 (size set dynamically),
        // using the int* OR the auto keyword and the same left-hand-side as in Java.
        // (the auto means that the type is automatic because there syntax can be
        // complex for specifying these arrays as a type),
        // HOWEVER, because they're then just pointers, the length is not kept
        // track of (just like a normal array passed by address into a function above
        int[] arr = new int[3];                 int* arr = new int[3]; // dynamic
        int[] arr = {1, 2, 3};                  int* arr = new int[3] {1, 2, 3}; // c++11

        // for-each loop syntax works and is the same
        // BUT, you can use "auto" instead of a type to get the type automatically
        // from the array:
        for (auto element : array) { ... }
        // ALSO, you can use references here to actually MODIFY values inside a
        // for-each loop (use const if you aren't actually modifying it):
        for (auto& element : array) { element = 5; ... }

        // IMPORTANT (USE THIS MORE OFTEN): the "array" class
        // There is also a std class called array that is between a basic array
        // and a vector (see below). It can be useful, and uses function templates.
        // (need to #include <array>)
        int[] arr = {1, 2, 3};                         array<int, 3> arr {1, 2, 3};
        arr.length                                     arr.size()
        arr[i]                                         arr[i]

        // 2D Arrays:
        // Unless you're using these simply, they can get complicated
        // (mainly with dynamic allocation - SO DON'T DO THIS DYNAMICALLY)
        int[][] arr2d = new int[5][5]                  int arr2d[5][5];
        // (with const ySize and const xSize)
        int[][] arr2d = new int[ySize][xSize]          int arr2d[ySize][xSize];
        int[][] arr2d;                                 auto arr2d;
        arr2d = new int[ySize][xSize];                 arr2d = new int[ySize][xSize]; OR
                                                       arr2d = new int[ySize * xSize];
        // the second one here is more efficient, but must be indexed like:
        // (see http://stackoverflow.com/a/28841507/3155372 - he does column-major though)
        arr2d[x + xSize*y]

        // NOTE: You have to use delete[] on these when they are dynamically allocated.
        // (you may need a loop or nested loops with multidimensional arrays)
```

Variable-Sized Arrays (ArrayList vs. vector)

```
        import java.util.ArrayList (or .List)                #include <vector>
        ArrayList<Integer> lst = new ArrayList<Integer>(size)      vector<int> list(size)

        .get(i)                                       [i] OR .at(i)
        .add(val)                                     .push_back(val)

        // instead of just putting 0, need to use .begin() in C++ to reference the start
        // (it's an iterator, which is basically a kind of pointer in the vector)
        .add(i, val)                                  .insert(.begin() + i, val)
        .add(.size() - i, val)                        .insert(.end() - i, val)

        // unique methods in C++ for some processes
        .remove(.size() - 1)                                .pop_back()
        .get(0)                                       .front()
        .get(.size() - 1)                             .back()

        .remove(i)                                    .erase(i)
        .removeRange(start, end)                      .erase(start, end)

        // Note: in C++, this size isn't just the initial memory allocation like in java,
        // but actually is the .size(). (So things like .push_back() reference this size)
        // I.E.: (with size=4 here):
        .add(5) -> [5]                                .push_back(5) -> [0,0,0,0,5]

        // GENERALLY, though, C++ reallocates memory behind the scenes like Java does
        // with ArrayLists, so you have methods such as the following to ensure that
        // the CAPACITY (which is separate from the normally-used SIZE) can be changed
        // to equal the size (because it may differ after some operations)
        .trimToSize()                                 .shrink_to_fit()

        // SAD NOTE: You cannot use references in a C++ vector, basically because
        // they cannot be reassigned once created, a requirement of vector entries.
        // NOT POSSIBLE: vector<obj&>
```

<u>Key-Value Pairings (Hash Tables / Hash Maps)</u>

```
        // Hash Tables are data structures which, instead of using integers to index
        // values like in arrays, can use any type (basically, they have key-value
        // pairs where the keys can be any type, corresponding to values of any type)
        // This means that, though multiple different keys can have the same value,
        // the same key must always reference the same value.
        // They are useful in some situations; A common example would be a dictionary,
        // where the keys (words) and values (definitions) are string types (in fact,
        // this is what they're called in the Python language)
        // (the following assumes in C++ that there is a "using namespace std")
        HashMap<String, int> mp = new HashMap<String, int>();            map<string, int>
mp;

        // The syntax in C++ is the same as in arrays (except the index can be any type)
        .put(key, val)                                mp[key] = val
        .get(key)                                     mp[key]
        .containsKey(key)                              .find(key)
                                                   // NOTE: .find(key) returns
                                                       // an iterator at that element,
                                                       // so use .find(key) == mp.end()
                                                       // to check if the key is
                                                       // not found (past the end)

        // iterating through keys/values is a bit different, but it just uses iterators
        // (which contain keys/values using .first and .second)
        // (see http://stackoverflow.com/a/110255/3155372)
        lst = mp.keySet().toArray()                          map<string, int>::iterator it =
                                                        m.begin(); // good place for auto
        lst[i]                                         it += i; it.first;
        mp.get(lst[i])                                 it += i; mp[it.first] OR
                                                       it.second;
```

Exception Handling

```
        // must include a library like in java (but just once for all standard exceptions)
        import [some exception]                        #include <exception>

        // normal method is similar
        try                                            try
        {                                              {
                ...                                            ...
                throw new IndexOutOfBoundsException();         throw (bad_alloc);
        }                                              }
        catch (IndexOutOfBoundsException e)            catch(exception& e)
        {                                              {
                e.printStackTrace();                                   cout << e.what() << endl;
        }                                              }

        // BUT, in C++ you can throw anything you want
        NOTHING                                        throw (someInt)

        // also, in C++ you can use "catch overloading" to catch exceptions based on
        // what type was thrown (we could have two different throw cases below
        // depending on what type was used in throw() )
        NOTHING                                        catch (int someInt)
        NOTHING                                        catch (char someChar)
```

File I/O

```
        // C++ files won't throw an error like Java, they will just return null,
        // so you need to check for null before using it.
        // (The writer will create a file to write to if it doesn't exist)
        writer = new FileWriter(new File(filename))   ofstream writer(filename)

        // You can append to a file too. in C++, you can also do a lot more
        // (The following are added as a second argument to a stream contructor):
        ios:trunc     // overwrite (default)
        ios:ate       // start at end of file
        ios::binary   // read binary
        ios::in       // allow reading input;   (default for ifstream)
        ios::out      // allow reading output;  (default for ofstream)
        // (the "|" is used to combine: ios::in | ios:out -> both input and output)
        wr = new FileWriter(new File(filename), true)       ofstream wr(filename, ios::app)

        // works just like cout in C++
        writer.write(str)                             writer << str << endl;

        // readers also work differently from writers in C++
        reader = new Scanner(new File(filename))            ifstream reader(filename)
        reader.hasNextLine()                                !reader.eof(); OR .is_open();

        // C++ reads in characters instead of strings, but there is a way...
        // the second method copies the line into "line" and returns a boolean
        // like !.eof()
        chr = reader.nextLine().charAt(0)                   reader.get(chr)
        str = reader.nextLine()                       getline(myFile, line)

        // There are some lower-level file operations in c++ (for either)
        // (it makes sense to use fstream here - the general superclass of ifstream
        // and ofstream)
        .seekg(pos)         // move cursor for .get() to an absolute position
        .seekg(relPos, base)      //          “”          relative to a base
        .seekp(pos)         // move cursor for .put() to an absolute position
        .seekp(relPos, base)      //         ""           relative to a base
        // base is any one of:
               ios::beg // beginning
               ios::cur // current
               ios::end // end

        .read(memblock, size) // read "size" bytes into an array of bytes(=chars)

        // NOTE: .close() is required for all these to avoid MEMORY LEAKS
```

<u>Objects/Types</u>

```
        // in C++ you can define a class and create one in the same line:
        // here "rect" is an instance of Rectangle
        NOTHING                                 class Rectangle { ... } rect;

        // NOTE: You need a SEMICOLON after a C++ class definition!

        // "new" isn't used here in C++, you just combine the operations
        // (except if you're using dynamic memory allocation)
        Thing x = new Thing(arg1, ...);         Thing x(arg1, ...); OR
                                                Thing x {arg1, ...}; OR
                                                Thing x = arg1; //only with ONE arg

        // C++ has a special syntax for calling other constructors in a class
        Animal(string name)                           Animal(string name) :
        { this(name, something_else); ... }      Animal(name, something_else) { ... }

        // public/private/protected variables AND methods in C++ are done
        // syntactically like CATEGORIES:
        // (But class fields default to private just like in Java, and static is the same
        private int x;                          private:
        private int y;                                int x;
                                                int y;

        // NOTE: When you create an object method that DOESN'T modify the object (i.e. a
        // getter), then add a "const" specifier at the end to indicate this.
        // (C++ will give warnings if you don't have this and try to make the
        // object be const when used as a parameter in a function)
        // FROM                                       TO
        int Student::getID() { ... }            ->    int Student::getID() const { ... }

        // field/method accessing uses different syntax depending on whether
        // they fields/methods are static.
        // (the :: operator is technically the scope operator
        //  (used in namespaces, etc.), while the -> operator dereferences the
        //  object pointer AND THEN accesses the field)
        [class name].[STATIC thing]                   [class name]::[STATIC thing]
        [object pointer].[NON-STATIC thing]     [object pointer]->[NON-STATIC thing]

        // the "this" keyword in c++ is a pointer to the current object
        // (so we need to use the DEREFERENCING accessor ->) equivalent to
        // (*this).[something]
        this.[something]                        this->[something]
        // because "this" is a pointer, you can also do this:
        bool isitme(Thing other) { return this == &other; }

        // In C++ methods can be defined just as prototypes and implemented
        // "statically" later:
        // NOTE: This tends to be the convention with larger methods
        // (NOT basic getter/setter types)
        // (Also, you can just put the type name in the prototype declaration)
        // I.E.: In "class Animal", do:         ...then later in the file, do:
        // OR:
        void setHeight(int);                    ->    void Animal::setHeight(int height)
                                                { ... }

        // C++ has shortcuts for going directly from constructor parameters to fields
        // in the class (a shortcut to doing this->field = arg)
        // This method of initialization is actually EXPECTED with const variables.
        NOTHING                                 Animal (int h) : height(h) {}; OR
                                                Animal (int height) : height(height) {};

        // NOTE: C++ does make automatic default constructors (and destructors;
        // see below), but they DISAPPEAR when others constructors are made.
        // (Java does this too).

        // IMPORTANT NOTE ON THE FOLLOWING: You should try to avoid using dynamic memory
        // (new X and delete X) in classes, because if you do you'll need to write
        // destructors, copy constructors, and copy assignment operators. TRY NOT TO!
        // For example, use string as opposed to char*, and let it do allocations, etc.
        // (because the implicitly defined copy functions/destructor will let them
        //  handle it)
        // (see http://stackoverflow.com/a/4172724 for a full explanation)
        // Constructors are similar, but C++ has DESTRUCTORS.
        // Instead of calling this method [class name], you do ~[class name] (tilda).
        // This is used for things like freeing memory because C++ doesn't have Java's
        // garbage collection.
        // They are called basically whenever Java would garbage collect
        // (when variables go out of scope, there are no more references to them, etc.),
        // but also through "delete" or some other explicit call.
        // see https://msdn.microsoft.com/en-us/library/6t4fe76c.aspx for more

        // NOTE on Copy/Move Constructors (see info in Basic Language Features):
        // Copy constructors do shallow copies (see above), and this may be
        // an issue for complex, object-enclosing classes, so you may want to redefine
        // the copy and assignment operators (they should do the same thing, just
        // the operator is not constructing the object and should return "*this")
        // DEFINITION                                         USAGE
        myObj (const myObj& x) strRef(new string(x.strRef)  myObj newMyObj(copiedMyObj)
                                                            {}
        myObj operator= (const myObj& x)              myObj newMyObj = copiedMyObj;
        {
              delete strRef;
              strRef = new string(x.strRef);
        }
        // see http://www.cplusplus.com/doc/tutorial/classes2/ for more

        // C++ also allows you to create (or override) methods for certain
        // generic operators by making methods a function called
        // "operator[operator to override]"
        // There are a bunch of possible ones,
        // see http://www.cplusplus.com/doc/tutorial/templates/#overloading_operators
        // for more.
        // (keep in mind the required number of parameters and return type of them)
        NOTHING                                       Thing operator+(Thing other)
                                                      { return this->val + other.val); }

        // Objects can also have templates (see Basic Language Features above)
        // DEFINITION                                 USAGE
        template <class T>                            mypair <int> intpair(1, 2);
        class mypair { T a, b; mypair(T a, T b); ... }


        // C++ has a unique feature in object-oriented programs called "friends"
        // A function or an class can be friends with a class, meaning the method
        // or class can access all the private or protected members of the class.
        // A friend method is one defined inside the friendly class but is not a
        // class method.
        // Notes: friends are only one-sided (below, Square can't access
        //  Rectangle's private stuff)
        // AND friends are NOT inherited.
        // DEFINITION                       USAGE
        class Rectangle                     Rectangle duplicate(const Rectangle& x)
        {                                   {
              ...                                 // width and height normally private
              int width;                        Rectangle rect(x.width, x.height);
              int height;                       return rect;
              ...                          }
              friend Rectangle duplicate
               (const Rectangle& x);
        }

        // OR (with prototype convert(Square) in Rectangle class)
        class Square                     void Rectangle::convert (Square a)
        {                                {
              friend class Rectangle;          width = a.side;      // side is normally
              int side;                        height = a.side;     // private
              ...                        }
        }
```

// INHERITANCE

```
// (here, the "public" before Animal in C++ denotes the highest visibility level
// members of Animal can have in the derived class - if it were private, all
// public members of Animal would become private to external functions using
// Dog, but would still be accessible to their normal level inside the Dog class.)
public class Dog extends Animal { ... }class Dog : public Animal { ... };

// The syntax for calling other constructors in the same class is the same in
// C++ for superclass constructors
Dog(string name) { super(name) ... };  Dog(string name) : Animal(name) { ... }

// C++ supports multiple inheritance (the closest Java has are interfaces
// which can be "implemented" by as many classes as possible) - see note below
```

// ABSTRACT/VIRTUAL ("virtual" means "abstract" in c++)

```
// Virtual functions are a bit different than abstract methods in Java: when
// a function in a superclass is labelled virtual, all this does--when a function
// uses polymorphism to generalize the derived class using the superclass--is
// defer calls to that method to the overridden method(s) in the derived
// class(es).
// (It can still be called itself, unlike Java where abstract cannot be called/
// have implementations (in C++, this is done with a "=0" (see below))

// BUT, virtual CLASSES are very similar to abstract classes, with the main
// difference being that they're defined by any "pure virtual" (unimplemented)
// functions inside them: (the "=0" is the crucial part indicating that it's
// "pure virtual" and MUST be implemented in derived classes)
abstract class Animal                       class Animal
{                                           {
      ...                                         ...
      abstract void eat();                              virtual void eat() = 0;
}                                           }

// Some DIFFERENCES:
//    virtual classes in C++ (those containing a virtual method
//     with "=0") CAN have other methods with implementations, and these can call
//     the pure virtual methods
// Some SIMILARITIES:
//     virtual classes cannot be instantiated, are only used in polymorphism,
//     and both methods MUST be implemented (overridden) by derived classes.

// So an INTERFACE in C++ is this kind of class (a virtual class with
//     ONLY pure virtual methods, used through multiple inheritance)
// DEFINITIONS:
public interface Walkable                       class Walkable
{                                               {
       abstract void walk();                           virtual void walk() = 0;
}                                               }
public interface Flyable                        class Flyable
{                                               {
       abstract void fly();                                    virtual void fly() = 0;
}                                               }
// USAGES ( both MUST implement walk() and fly() ):
public class Cat implements Walkable, Flyable          class Cat : public Walkable,
                                                 public Flyable
{ ... }                                         { ... }
```

// ENUMS

```
// The most basic type definition (besides typedef aliasing - see above) are enums
// These basically allow you to declare a type with a finite set of possible
// values that you specify with what are essentially macros.
// (NOTE: it is normal to preface these macro names with the enum type name
// for clarity (because they are then in the global namespace)
// (They are useful for "enumerating" a set of possible states or properties
// rather than using numbers to represent them (for readabilities' sake))
// DEFINITION                                   USAGE
enum Color                                      Color color = COLOR_BLUE;
{                                               bool isRed = color == COLOR_RED;
       COLOR_BLUE          // set to 0          color = static_cast<Color> 2;
       COLOR_RED = 2// can set manually
       COLOR_GREEN   // set to 3 (counts up)
};
```

// STRUCTS

```
// structs are basically the same thing as object, but aren't in Java.
// They are defined just like classes in C++ (see below) and can be
// used just like them. (They generally are just used for data without methods,
// though (more primitively that a class). You should use classes otherwise)
// (The only real differences is that the default visibility is public in structs
// but private in classes)
// DEFINITION                                   USAGE
struct movie { string name; int length; }              movie alien; alien.name =
"Alien";

// They also can have pointers to them:
// (there is actually a special operator for accessing fields in a struct
// pointer too, which dereferences the pointer and accesses the field)
// DEFINITION                                   USAGE
movie* alien_loc = &alien;                      cout << (*alien_loc).name; OR
                                              cout << alien_loc->name;

// even more here: http://www.cplusplus.com/doc/tutorial/other_data_types/
```

Making "Importable/#Includable" Files

```
        // (see http://www.learncpp.com/cpp-tutorial/19-header-files/
        //  AND look at compiling)
        // In Java, when we create a class file and "import" it, it just works.
        // But, in C++, we're closer to what really happens, so it's slightly more
        // complicated:

        // The process is:

        // 1. Create a header (.h) file with the DEFINITIONS of your functions.
        //    Also, to ensure this isn't misused, surround it with these
        //    preprocessor commands:
        #ifndef <uppercase_file_name>_H  // if this name hasn't been defined before,
        #define          ""              // define it and tell the preprocessor
        ...                              // to add our header code to the given file
        #endif                           // using #include
        // see http://www.learncpp.com/cpp-tutorial/1-10a-header-guards/ for more)

        // 2. Create an "implementation" file to IMPLEMENT the defined functions,
        //    with the same name.

        // 3. Actually import it using:
        #include "<file_name>.h"
        //     (note quotes here - brackets are for system header files,
        //      while quotes look in the current directory. You can also look
        //      relatively using /subfolder or ../superfolder BUT you should create and
        //      reference an "include directory" in your compiler/IDE for large projects)
        //     (also note .h here - things like #include <iostream> are shortcuts
        //           associated with the standard library (std) )

        // NOTE: the files are literally COPIED in by the "preprocessor," and the source
        // files are later linked in the compiling "linking" phase so that the program
        // knows how to actually execute the defined functions.

        // NOTE: Any #include <XXX> (library files) you do in the .h file will be usable
        // in the .cpp, but if you're importing local files (#include "XXX"), you'll have
        // to do it according to this guide (section 4):
        // http://www.cplusplus.com/forum/articles/10627/
        // BASICALLY, you'll likely have to forward-declare objects that you use
        // as arguments or return types in functions, etc, AND THEN #include those
        // header files in the .cpp when you're actually using the class
        // (at that point, all the details of the forward declared class will be filled in
        // by the .h file defining it). Otherwise, if you're declaring an object of some
        // type or are using it in a function IMPLEMENTATION, you should just #include it,
        // NOT forward-declare it (this applies mainly to non-local files -- if you end up
        // doing this with header files you should probably move you usage into the .cpp
        // and use the above method)
        // Finally, parent classes to the current one should simply use #include "XXX"
        // (there is a very good example class at the above link illustrating all this)

        // ALSO, see "IMPORTING AND NAMESPACES" above
```

Debugging

```
    // C++ is a bit harder to debug than Java because there aren't usually runtime
    // error messages (just the dreaded "Segmentation Fault" which occurs when
    // off-limits memory (a.k.a. memory past the end of an array OR the null (0x0)
    // pointer) is accessed.
    // SO, to debug, you'll likely want to use a common debugger called GDB
    // (the alternative is print statements).
    // Here is a good starter on GDB:
    //     http://www.cs.umd.edu/~srhuang/teaching/cmsc212/gdb-tutorial-handout.pdf
    // And a good general reference:
    //     http://darkdust.net/files/GDB%20Cheat%20Sheet.pdf
```