Moving from Java to C++

This appendix explains how to transfer your Java programming skills to a substantial subset of C++. This is necessary for students who take their first programming course in Java and the second course in C++. Naturally, it would be easiest if the second course were also offered in Java, but learning to move from one language to another is a fact of life for today's software professionals. Fortunately, C++ has many features in common with Java, and it is easy for a Java programmer to gain a working knowledge of C++. Nevertheless, C++ is a much more complex language than Java. This appendix does not attempt to cover all features of C++. But if you master all of the constructs described in this appendix, you will be able to use C++ effectively.

We only cover the differences between Java and C++. In particular, control flow (`if`, `while`, `for`) is essentially identical in C++ and Java, and it is not covered here.

This appendix describes ANSI C++. Some older C++ compilers lack essential features described here. To use those compilers, you will need to learn more about the parts of C++ that were inherited from C. That is beyond the scope of this discussion.

A3.1. Data Types and Variables

The data types in C++ are similar to those in Java. Like Java, C++ has `int` and `double` types. However, the range of the numeric types such as `int` is machine-dependent. On 16-bit systems such as PCs running DOS or Windows 3.x, `int` are 2-byte quantities with a much more limited range than the 4-byte Java `int` type. On those machines, you need to switch to `long` whenever the `int` range is not sufficient.

C++ has `short` and `unsigned` types that can store numbers more efficiently. It is best to avoid these types unless the added efficiency is crucial.

The Boolean type is called `bool` in C++.

The C++ string type is called `string`. It is quite similar to the Java `String` type. However, pay attention to these differences:

1. C++ strings store ASCII characters, not Unicode characters

2. C++ strings can be modified, whereas Java strings are immutable.

3. The substring operation in C++ is called `substr`. The command `s.substr(i, n)` extracts a substring of length `n` starting at position `i`.

4. You can only concatenate strings with other strings, not with arbitrary objects.

5. To compare strings, use the relational operators `== != < <= > >=`. The last four operators perform lexicographic comparison. This is actually more convenient than the use of `equals` and `compareTo` in Java.

A3.2. Variables and Constants

In C++, local variables are defined just as in Java.

```
int n = 5;
```

There is, however, a major difference between C++ and Java. The C++ compiler does not check whether all local variables are initalized before they are read. It is quite easy to forget initializing a variable in C++. The value of the variable is then the random bit pattern that happened to be in the memory location that the local variable occupies. This is clearly a fertile source of programming errors.

As in Java, classes can have data fields and static variables. Furthermore, variables can be declared outside functions and classes. These so-called *global variables* can be accessed from any function in a program. That makes them difficult to manage. C++ programs should avoid global variables.

In C++, constants can be declared anywhere. (Recall that in Java, they had to be static data of a class.) C++ uses the `const` keyword instead of `final`.

```
const int DAYS_PER_YEAR = 365;
```

A3.3. Classes

The definition of classes in C++ is somewhat different than in Java. Here is an example: a C++ version of the `Point` class:

```
class Point /* C++ */

{

public:

   Point();

   Point(double xval, double yval);

   void move(double dx, double dy);

   double getX() const;

   double getY() const;

private:

   double x;

   double y;

};
```

There are several essential differences.

1. In C++, there are public and private *sections*, started by the keywords `public` and `private`. In Java, each individual item must be tagged with `public` or `private`.

2. The class definition only contains the declarations of the methods. The actual implementations are listed separately.

3. Accessor methods are tagged with the keyword `const`

4. There is a semicolon at the end of the class

The implementation of methods follows the class definition. Because the methods are defined outside the classes, each method name is prefixed by the class name. The `::` operator separates class and method name. Accessor methods that do not modify the implicit parameter are tagged as `const`.

```
Point::Point() { x = 0; y = 0; }


void Point::move(double dx, double dy)
{  x = x + dx;
   y = y + dy;
}


double Point::getX() const
{  return x;
}
```

A3.4. Objects

The major difference between Java and C++ is the behavior of object variables. In C++, object variables hold *values*, not object references. Note that the `new` operator is never used when constructing objects in C++. You simply supply the construction parameters after the variable name.

```
Point p(1, 2); /* construct p */
```

If you do not supply construction parameters, then the object is constructed with the default constructor.

```
Time now; /* construct now with Time::Time() */
```

This is very different from Java. In Java, this command would merely create an uninitialized reference. In C++, it constructs an actual object.

When one object is assigned to another, a copy of the actual values is made. In Java, copying an object variable merely establishes a second reference to the object. Copying a C++ object is just like calling `clone` in Java. Modifying the copy does not change the original.

```
Point q = p; /* copies p into q */
q.move(1, 1); /* moves q but not p */
```

In most cases, the fact that objects behave like values is very convenient. There are, however, a number of situations where this behavior is undesirable.

1. When modifying an object in a function, you must remember to use call by reference (see below)

2. Two object variables cannot jointly access one object. If you need this effect in C++, then you need to use pointers (see below)

3. An object variable can only hold values of a particular type. If you want a variable to hold objects from different subclasses, you need to use pointers

4. If you want a variable point to either null or to an actual object, then you need to use pointers in C++

A3.5. Functions

In Java, every function must be an instance method or a static function of a class. C++ supports instance methods and static functions of classes, but it also permits functions that are not a part of any class. Such functions are called *global functions*.

In particular, every C++ function starts with the global function `main`.

```
int main()

{ . . .

}
```

There is a second version of `main` that you can use to capture command-line arguments, but it requires knowledge of C-style arrays and strings, and we will not cover it here.

By convention, the return value of `main` is zero if the program completed succesfully, a non-zero integer otherwise.

As in Java, function arguments are passed by value. In Java, functions were nevertheless able to modify objects. However, since C++ object values are not references to actual objects, a function receives a copy of the actual argument and hence can never modify the original.

Therefore, C++ has two parameter passing mechanisms, *call by value* (as in Java) and *call by reference*. When a parameter is passed by reference, the function can modify the original. Call by reference is indcated by an `&` behind the parameter type.

```
void raiseSalary(Employee& e, double by)

{ . . .

}
```

Here is a typical function that takes advantage of call by reference. Note that it would be impossible to write such a function in Java.

```
void swap(int& a, int& b)

{  int temp = a;

   a = b;

   b = temp;

}
```

If this function is called as `swap(x, y)`, then the reference parameters `a` and `b` refer to the locations of the arguments `x` and `y`, not the values of these arguments. Hence the function can actually swap the contents of these variables.

In C++, you always use call by reference when a function needs to modify a parameter.

A3.6. Vectors

The C++ vector construct combines the best features of arrays and vectors in Java. A C++ vector has convenient element access, and it can grow dynamically. If `T` is any type, then `vector<T>` is a dynamic array of elements of type `T`. The instruction

```
vector<int> a;
```

makes an initially empty vector. The command

```
vector<int> a(100);
```

makes a vector that has initially 100 elements. You can add more elements with the `push_back` method:

```
a.push_back(n);
```

The call `a.pop_back()` removes the last element from `a`. Use the `size` method to find the current number of elements in `a`.

You access the elements with the familiar `[]` operator.

```
for (i = 0; i < a.size(); i++)
   sum = sum + a[i];
```

As in Java, array indexes must be between 0 and `a.size() - 1`. However, unlike Java, there is no runtime check for legal array indexes. Accessing an illegal index can cause very serious errors.

Just like all other C++ objects, vectors are values. If you assign one vector to another, all elements are copied.

```
vector<int> b = a; /* all elements are copied */
```

Contrast that with the situation in Java. In Java, an array variable is a reference to the array. Making a copy of the variable just yields a second reference to the same array.

For that reason, C++ functions that modify vectors must use reference parameters.

```
void sort(vector<int>& a)
{ . . .
}
```

A3.7. Input and Output

In C++, the standard input and output stream are represented by the `cin` and `cout` objects. You use the <<

operator to write output.

```
cout << "Hello, World!";
```

You can print multiple items as well.

```
cout << "The answer is " << x << "\n";
```

To read a number or a word from input, use the >> operator.

```
double x;

cout << "Please enter x: ";

cin >> x;


string fname;

cout << "Please enter your first name: ";

cin >> fname;
```

The `getline` method reads an entire line of input.

```
string inputLine;

getline(cin, inputLine);
```

If the end of input has been reached, or if a number could not be read correctly, the stream is set to a failed state. You can test for that with the `fail` method.

```
int n;

cin >> n;

if (cin.fail()) cout << "Bad input";
```

Once the stream state has failed, you cannot easily reset it. If your program needs to handle bad input, you should use `getline` and then manually process the input.

A3.8. Pointers

In C++, object variables hold object values. This is different from Java, where an object variable only is a reference to an object value that is stored elsewhere. There are circumstances where the same arrangement is required in C++. In C++, a variable that can refer to an object is called a *pointer*. If `T` is any type, then `T*` is a pointer to an object of type `T`.

Just as in Java, a pointer variable can be initialized with `NULL`, with another pointer variable, or with a call to `new`.

```
Employee* p = NULL;

Employee* q = new Employee("Hacker, Harry", 35000);
```

```
Employee* r = q;
```

Actually, there is a fourth possibility. Pointers can be initialized with the address of another object, by using the & operator.

```
Employee boss("Morris, Melinda", 83000);

Employee* s = &boss;
```

This is usually not a good idea. As a rule of thumb, C++ pointers should only refer to objects allocated wth `new`.

So far, C++ pointers look very much like Java object variables. There is, however, an essential syntactical difference. You must apply the `*` operator to access the object to which a pointer points. If `p` is a pointer to an `Employee` object, then `*p` refers to that object.

```
Employee* p = . . .;

Employee boss = *p;
```

You also need to refer to `*p` when you want to execute a method or access a data field.

```
(*p).setSalary(91000);
```

The parentheses are necessary because the `.` operator has a higher precedence than the `*` operator. The designers of C found this sufficiently ugly that they provided an alternate `->` operator to combine the `*` and `.` operators. The expression

```
p->setSalary(91000);
```

invokes the `setSalary` method on the object `*p`. You can simply remember to use the `.` operator for objects, the `->` operator for pointers.

If you do not initialize a pointer, or if the pointer is NULL or refers to an object that no longer exists, then it is an error to apply the `*` or `->` operator. Unfortunately, the C++ runtime system does not check against these errors. If you make such a mistake, your program can die a horrible death or act flaky.

In Java, these errors are not possible. You cannot have an uninitialized reference. All objects are kept alive as long as there is a reference to it. Hence you cannot have a reference to a deleted object. The runtime system checks for null references and throws a null pointer exception if a null pointer is accessed.

There is another significant difference between C++ and Java. Java has a *garbage collector* that automatically reclaims all objects that are no longer needed. In C++, it is the responsibility of the programmer to manage memory.

Object variables are automatically reclaimed when they go out of scope. However, objects created with `new` must be reclaimed manually with the `delete` operator.

```
Employee* p = new Employee("Hacker, Harry", 38000);

. . .

delete p; /* no longer need this object */
```

If you forget to delete an object, then you can eventually exhaust all memory. This is called a *memory leak*. More importantly, if you delete an object and then continue to use it, you can overwrite data that no longer belongs to you. If you overwrite any of the data fields that are used to manage the recycled storage, the allocation mechanism can malfunction and cause subtle errors that are very difficult to diagnose and fix. For this reason, it is best if you minimize the use of pointers in C++.

A3.9. Inheritance

The basic syntax for inheritance is similar in C++ and Java. In C++, you use `: public` instead of `extends` to denote inheritance. (C++ also supports a concept called private inheritance, but it is not very useful.)

By default, functions are not dynamically bound in C++. If you want which dynamic binding for a particular function, you must declare it as `virtual`.

```
class Manager : public Employee

{

public:

   Manager(string name, double salary, string dept);

   virtualvoid print() const;

private:

   string department;

};
```

As in Java, there is special syntax for a constructor to invoke the constructor of the superclass. Java uses the keyword `super`. In C++, you must call the superclass constructor outside the body of the subclass constructor. Here is an example.

```
Manager::Manager(string name, double salary, string dept)

: Employee(name, salary) /* call superclass constructor */

{  department = dept;

}
```

Java also uses the `super` keyword when a subclass method calls the superclass method. In C++, you use the name of the superclass and the `::`operator instead.

```
void Manager::print() const

{  Employee::print(); /* call superclass method */

   cout << department << "\n";

}
```

A C++ object variable holds objects of a specific type. To exploit polymorphism in C++, you need pointers. A `T*` pointer can point to objects of type `T` or any subclass.of `T`.

```
Employee* e = new Manager("Morris, Melinda", 83000, "Finance");
```

You can collect multiple objects of a mixture of super- and subclasses in a vector of pointers, and then apply a dynamically bound function.

```
vector<Employee*> staff;

. . .

for (i = 0; i < staff.size(); i++)

   staff[i]->print();
```