Appendix

Introduction to C++ for Java Programmers

Every C++ program must consist of at least the `main` function. The `cin` object handles input from the standard input device (i.e., the keyboard); the `cout` object handles output to the standard output device (i.e., the display monitor). Both `cin` and `cout` require the inclusion of the iostream include file. `endl` represents “`\n`”. The insertion operator `<<` inserts an item into the output stream (i.e., it outputs). The extraction operator `>>` extracts an item from the input stream (i.e., it inputs). The using statement is similar in function to the `import` statement in Java. Without it, we would have to use the fully qualified names of `cout`, `cin`, and `endl` (which are `std::cout`, `std::cin`, and `std::endl`, respectively).

```
// Sample Program 1
#include <iostream>             // required by cin, cout
using namespace std;            /* now do not have to
                                   qualify names of cin,
                                   cout, and endl  */
int main()
{
   int x;

   cout << "Enter integer\n";   // displays "Enter integer"
   cin >> x;                    // reads integer into x
   cout <<  x << " squared = "  // displays "3 squared = 9"
        << x*x << endl;
}
```

Sample session:

```
Enter integer
3
3 squared = 9
=====================================================================
```

A pointer is an address. A pointer variable holds an address. The statement

```
int x, y;
```

declares the int variables `x` and `y`. The statement

```
int *p, *q;
```

declares the int pointer variables `p` and `q`. The statement

```
y = x;
```

assigns the value of x to y. The statement

```
p = &x;
```

assigns the address of `x` to `p` (`&` in this context means "address of"). `*p` is the location to which `p` points. Thus, the statement

```
*p = 5;
```

assigns 5 to the location to which p points. It does not assign 5 to `p`. To dereference a pointer means to follow it to the location to which it points. For example, we are dereferencing `p` in both of these statements:

```
*p = 5;
y = *p;
```

In the first, we are placing 5 into the location to which `p` points; in the second, we are placing the value in the location to which `p` points into `y`. Arithmetic operations can be performed on pointers. For example, suppose an int pointer `p` points to the first element of an `int` array. Then the statement

```
p = p + 1;
```

changes `p` so that it points to the next element of the array. Thus, adding 1 to `p` in a C++ statement actually increases the address in `p` by the size of one array element. If the size of an `int` is four bytes, this statement actually adds 4 to the contents of `p`.

The name of an array without square brackets is interpreted as a pointer to (i.e., the address of) the first slot of that array. For example, suppose a is an int array, and `p` is an `int` pointer. Then

```
p = a;
```

assigns the address of `a[0]` to `p`. Alternatively, we can use the equivalent (but longer) statement

```
p = &a[0];
```

A pointer to an array can be used like the name of that array. For example, if p points to an array a, we can use `p` as the array name. For example, in place of

```
p[2] = 3;         // use p as name of array
```

we can use

```
a[2] = 3;         // use actual array name
```

We can also use `p` as a pointer:

```
*(p = 2) = 3;     // or use p as a pointer
```

The three statements above all have the same effect.

Arrays can be created with the new operator. For example,

```
p = new int[3];
```

allocates an int array consisting of three slots, and assigns its address to `p`. We can then use `p` as the name of this array. For example, to assign 5 to the second slot of this dynamically allocated array, we can use

```
p[1] = 5;
```

Arrays can also be created with global or local declarations. For example, the declaration

```
int a[] = { 1, 2, 3 };
```

within a function declares a local array a containing 1, 2, and 3 in its three slots.

```
// Sample Program 2
#include <iostream>
using namespace std;

int main()
{
   int a[] = { 1, 2, 3 };       // local array
   int x;
   int *p;                      // p is an int pointer

   p = &x;                      // p now points to x
   *p = 5;                      // assign 5 x
   cout << "x = " << x << endl; // display "x = 5"
   p = a;                       // p now points to a[0]
   cout << *p << endl;          // display a[0]
   p = p + 2;                   // p now points to a[2]
   cout << *p << endl;          // display a[2]
   p = new int[3];              // p assigned address of array
   p[1] = 5;                    // assign 5 to 2nd slot
   *(p + 1) = 6;                // assign 6 to 2nd slot
   cout << p[1] << endl;        // display 2nd slot
}
```

Output:

```
x =  5
1
3
6
```

```
=====================================================================
```

A global variable is declared outside a function. It is created and initialized at assembly time. A global variable whose declaration does not specify an initial value is guaranteed to have the initial value 0. A dynamic local variable is a variable declared within a function without the keyword static. It is created on entry to the function and is destroyed on exit. Unless explicitly initialized, the value of a dynamic local variable is undefined. A function call must be preceded by either the function's definition or prototype. A function prototype is like the first line of a function definition, terminated with a semicolon.

```
// Sample Program 3
#include <iostream>
using namespace std;

void f(int z);                     // prototype for f
int gv1;                           // global, initial value is 0
int gv2 = 5;                       // global, initial value is 5

int main()
{
   f(2); f(3);
}
void f(int z)                      // z is a parameter
{
   int x;                          // dyn local created on each call
                                   // value of x undefined
   x = z;                          // now value of x defined
```

```
    cout << "x = " << x << endl;
    cout << "gv1 = " << gv1 << endl;
    cout << "gv2 = " << gv2 << endl;
    gv1++; gv2++                    // increment gv1, gv2
}
```

Output:

```
x = 2
gv1 = 0
gv2 = 5
x = 3
gv1 = 1
gv2 = 6
====================================================================
```

Function calls pass the value of their arguments unless reference parameters are used (see Sample Program 5 below). If `&y` is an argument in a function call, the address of y is passed.

```
// Sample Program 4
#include <iostream>
using namespace std;

void f(int x, int *p)         // x gets 5; p points to y
{
    *p = x;                   // assign x to y
}
int main()
{
   int y;

   f(5, &y);                  // pass 5 and address of y
   cout << "y = " << y << endl;
}
```

Output:

```
y = 5
====================================================================
```

A reference parameter receives the address of its corresponding argument. This address is automatically dereferenced wherever the parameter is used. A & preceding the name of a parameter in the parameter list of a function definition identifies the parameter as a reference parameter.

```
// Sample Program 5
#include <iostream>
using namespace std;

void f(int x, int &r)         // r is ref parm, r points to y
{
    r = x;                    // r dereferenced
}
int main()
{
   int y;
```

```
    f(5, y);                     // pass 5 and address of y
    cout << "y = " << y << endl;
}
```

Output:

```
y = 5
```

==

A struct and a class are almost identical. The only difference is that in a struct, members default to public, but in a class, members default to private.

An instance of a struct or a class can be created with the `new` operator or with an ordinary declaration. For example, suppose `K` is a class and `q` is declared with

```
K *q;
```

Then the statement

```
q = new K;
```

assigns the address of a newly created `K` object to `q`. The object can then be accessed through `q`. For example, if the object has an int field `x`, we can assign 5 to this field with

```
(*q).x = 5;
```

`(*q)` is the object `q` points to. Thus, `(*q).x` is the `x` field of this object. We can also write this statement equivalently with

```
q -> x = 5;
```

Objects can also be created with declarations. For example, the declaration

```
K k;
```

creates an object `k` of type `K`. `k` denotes the object itself—not its address. This object can be accessed directly through the name `k` using the dot operator. For example, if k has a int field `x`, we can assign it 5 with

```
k.x = 5;
```

A class declaration usually contains only the prototypes its member functions. The definitions of member functions appear outside the class. These definitions start with

<return type> <class name>::<function name>(parameter list)

A semicolon should follow the closing brace of a struct or class definition.

```
// Sample Program 6
#include <iostream>
using namespace std;

class K {
public:
   int x;
   void f(int a);                // prototype for f
};                               // remember the semicolon
void K::f(int a)                 // definition of f
```

```
{
   x = a;
}

int main()
{
   K k, *p, *q;                  /* k is a K object
                                    p, q are K-object pointers */
   p = &k;                       // assign p the address of k
   k.f(1);                       // invoke f directly thru k
   (*p).f(1);                    // another way to invoke f
   p -> f(1);                    // another way to invoke f
   cout << k.x << endl;          // display x directly thru k
   cout << (*p).x << endl;       // display x via ptr p
   cout << p -> x << endl;       // display x via ptr p
   q = new K;                    // assign address of new obj to q
   (*q).f(2);                    // invoke f via ptr q
   q -> f(2);                    // invoke f via ptr q
   cout << (*q).x << endl;       // display x via ptr q
   cout << q -> x << endl;       // display x via ptr q
}
```

Output:

```
1
1
1
2
2
```