UNIVERSITY OF CALIFORNIA
Department of Electrical Engineering
and Computer Sciences
Computer Science Division

CS61B **P. N. Hilfinger**
Summer 1998

C++ for Java Programmers*

Reference Materials. Any serious would-be C++ programmer should own the third edition of the *The C++ Programming Language* by Bjarne Stroustrup (Addison-Wesley, 1997, ISBN 0-201-88954-4). You may find Stanley Lippman's *C++ Primer* useful as an introduction.

1 Program Structure

C++ allows functions that are not part of any class (*non-member functions*). Thus,

```
bool isPrime(int x)
{
   if (x < 2)
      return false;
   for (int i = 0; i < x/2; i += 1)
      if (x % i == 0)
         return false;
   return true;
}
```

This function may be called, without any qualification—as in `if (isPrime(x))`—from anywhere in the program (including from code written in other files). If we change the first line to

```
static bool isPrime(int x) ...
```

We have a function that is local to the compilation unit that contains it (a *compilation unit* is a single item submitted to a C++ compiler to translate).

Variables may also be defined outside any class or function, as in

```
int numberOfTasks;
int size = 42;
static double Foo = 3;
```

These are all like static fields in Java, except that the variable `Foo` above (like a function declared static) is local to the compilation unit.

C++ has no packages as such, but a new feature, *namespaces*, get a similar effect. I refer you to the reference manual.

The *main program* in a C++ program is a non-member function called `main`, usually declared

```
main (int argc, char* argv[])   // Or char** argv
{ ... }
```

—that is, without a return type. The two parameters together serve the purpose of the single parameter to Java main programs. Two are needed because C++ arrays do not carry their length along with them.

2 Declarations vs. Definitions, Header Files

In Java, a function's parameter types, return type, name, and body are all specified together. In C++, this is called a function *definition.* C++ also has related constructs called *declarations,* which do not include the bodies (or in general, the values) of the entities being introduced. For example, corresponding to `isPrime` above, we have either of the declarations

```
extern bool isPrime (int x);
/* or */
static bool isPrime (int x);
```

(depending on whether the corresponding definition specifies `static`).

Declarations of outer-level variables (outside of any class or function) look like this:

```
extern int numberOfTasks;
extern int size;
static double Foo;
```

(Actually, the `extern`s can be left off, but I consider that bad (actually, old) style. Variable declarations have no initializers (i.e., '`= 42`' and the like).

Good style in C/C++ is to use *both* declarations and definitions of each function and global variable, and to gather them into *header files,* whose names typically end in `.h`. Here's a typical example, a file `IntList.h` that declares a type `IntList`, a function `reverse`, and a global variable, `totalListElements`.

```
/* IntList.h: List of integers */

#ifndef _INTLIST_H
#define _INTLIST_H

extern int totalListElements;

class IntList {
public:
   int head;
   IntList* tail;

   IntList (int head, IntList* tail) {
       this->head = head; this->tail = tail;
       totalListElements += 1;
   }

   int sum ();
};

extern IntList* reverse (IntList* L);

#endif
```

A program that wished to use these definitions would have, near the front,

```
#include "IntList.h"
```

which means "remove this `#include` line and, in its place, substitute the contents of the file `IntList.h`."

The strange statements beginning with '`#`' are known as *preprocessor statements.* These particular ones are a standard C/C++ idiom (or kludge, if you prefer) that prevent the same set of declarations from being processed twice (a given program may need several header files, each of which includes a common header file whose definitions it uses).

In the `IntList` example, the constructor is defined, but the member function *sum* is only declared (no body). One would typically put the body of `sum` in another file, such as `IntList.cc`:

```
/* IntList.cc: Implementations of IntList. */

#include "IntList.h"

IntList* reverse (IntList* L) {
   ...
}

IntList* IntList::sum (IntList* L) {
   ...
}
```

Where F is a class and X is something defined in that class, the notation $F::X$ means "the X declared in F." Thus, the definition of `sum` here means "The `sum` function declared in class `IntList` is defined as...."

3 Memory Model

In Java, one can write

```
int x;                    // A
Foo y = new Foo ();       // B
Foo z = y;                // C
```

and the meaning is that `x` is a box (variable) that can contain integers, `y` is a box that can contain pointers to `Foo`s, and `z` is a different box that is initialized to contain the same pointer that `y` has at that moment. It is *not* possible to have something that points to an `int` directly, and it is *not* possible to have a variable that actually contains a `Foo` (without going through a pointer to get to it).

In C/C++, all these things are possible, requiring more syntax.

```
int x;             // A
x = 3;
int* xp;           // xp contains pointers to ints
xp = new int;      // xp is set to point to a new int box.
xp = &x;           // xp is set to point to x.
Foo* y = new Foo;  // B. y contains pointers to Foo
Foo* z = y;        // C. z and y now point at the same thing
Foo q;             // q contains Foos directly
```

We also need more syntax to get at these things:

```
*xp = 42;          // Set the thing pointed to by xp to 42
                   // (so x is now 42).
(*y).print ();     // Print the Foo pointed to by y
y->print ();       // Shorthand for preceding statement
```

Syntax warning. In C/C++,

```
int* x,y;
```

means "`x` is a pointer to an `int`; `y` is an `int`." You need

```
int *x, *y;
```

to make them both `int`s. The spacing doesn't matter; I prefer the star against the type when possible, but it is confusing when multiple things are defined. *End of warning.*

3.1 Arrays

In C/C++, arrays do not carry bounds information and are closely allied with (though not identical to) pointers. Given the declaration

```
int A[4];
```

`A` is an array of 4 `int`s. Whenever the name `A` appears, it is implicitly converted to a pointer to `A[0]`. Thus,

```
    A[0] = 3;
/* is the same as */
    int* Ap = A;   // or    int* Ap =  & A[0];
    *Ap = 3;
/* and now */
    A[2] = 4;
/* is the same as */
    Ap[2] = 4;
```

That is, a pointer is always taken to be (potentially) a pointer to some element in the middle of an array. If pointer p points to element i of an array, then $p+k$ is also a pointer, for k an integer, and it points to element $i+k$ of the same array. In fact, the syntax `A[i]` is by definition equivalent to `*(A+i)`.

On the one hand, this makes it easy to pass a sub-array to a function. If `sum(A, N)` adds elements `A[0]`,...,`A[N-1]`, then `sum(A+1, N-1)` adds elements `A[1]`,...,`A[N-1]`. On the other hand, since there is no bounds information carried around with arrays, it is easy to make errors, and these errors are not typically caught by C/C++ systems. That is, very few systems will catch

```
int* A = new int[4];
A[4] = 3;     // But there is no A[4]!
```

3.2 References

In C/C++, it is possible to do something that was impossible in Java. The following function in Java does not do what it says:

```
/** Increment x (??) */
void incr (int x) { x += 1; }
```

The call `incr(y)` does *not* increment `y`, only a (useless) copy of it. In C/C++, on the other hand, you may write

```
void incr (int* x) { *x += 1; }
```

and call it with, e.g., `incr(&y)`. That is, a *pointer* to `y` is passed *by value,* which gives the effect of passing `y` *by reference,* as for Pascal's `var` parameters.

C++ provides an additional shorthand in the form of the *reference type.* A reference type is essentially a pointer type whose values are always automatically deferenced (`*`) or references (`&`) as needed. For example,

```
void incr (int& x) { x += 1; }
...
incr (y);
```

essentially passes a pointer to `y` as the parameter `x`, and all uses of `x` in the body of the function are, in effect, "starred."

4 Classes and Inheritance

The Java declaration

```
class A extends B implements C {
   public declaration 1

   protected declaration 2

   public declaration 3

   public declaration 4

   private declaration 5
}
```

becomes

```
class A : public B, public C {
public:
   declaration 1

protected:
   declaration 2

public:
   declaration 3

   declaration 4

private:
    declaration 5
};
```

in C++. There is no distinction between 'extends' and 'implements' and a C++ class can extend any number of base classes.

A C++ member function declared

```
virtual void f () { ... }
```

Is like an ordinary Java member function. Functions in other classes that override this declaration need not say `virtual` (and as a stylistic matter, generally don't). To get the effect of an abstract member function:

```
virtual void f () = 0;
```

(C++ classes are not declared to be abstract; they just are if they contain abstract methods). Non-overriding functions declared without the `virtual` keyword are somewhat like Java's `final` functions, except that `Java` allows its `final` functions to be virtual, whereas C++ does not (that

is, Java final methods can't be overridden, but they may themselves override something in their parent). You can *hide* the declaration of a non-virtual `f` in a child, but you can't *override* it to get the effects of dynamic dispatching of methods based on their run-time types.

5 Templates

The designer of C++ is against the `instanceof` operator and also against "downcasting," which is what happens in Java when you write `(String) (myVector.elementAt(k))`. Although C++ recently introduced these features, that's clearly not where its heart is. Instead, C++ handles problems such as the definition of the `Vector` class with *templates*. The declaration

```
template <class T>
class Vector {
public:
    Vector (int N) { vals = new T[N]; this->N = N; }
    int size () { return N; }
    T elementAt (int k) {
        if (k < 0 || k >= N) abort ();
        return vals[k];
    }
private:
    T* vals;
};
```

allows you to define new `Vectors` containing values of a single type, as in

```
Vector< int > q(10);
Vector< Vector<int>* >* P = new Vector< Vector<int>* >(100);
```

Here, `q` is a `Vector` containing 10 ints, and `P` is a pointer to a `Vector` of pointers to `Vectors` of `ints`, initialized to point to a new `Vector` of 100 integers. (By the way, this example also illustrates the placement of the parameters to the constructor in cases where one does not use `new`.)

One can also templatize non-member functions:

```
template<class T, class U>
U f(U r0, T x) {
   U r = r0;
   for (int i = 0; i < x.size (); i += 1) {
      r += x[i];
   return r;
}
```

As you can see, multiple arguments are possible. This function is called just like an ordinary one—`f(V)`. It is overloaded on all possible types `U` and `T`.

6 Strings

C/C++ represent primitive strings as arrays of `char` (which, therefore, get passed around as values of type `char*`). Since you really need the length of a string and C/C++ arrays don't carry such information, they use the convention of putting a null (`'\000'`) character after the last character of the string. Thus, after

```
char Hello[6] = { 'H', 'e', 'l', 'l', 'o', '\000' };
char* hi = Hello;
```

The array `Hello` and the pointer `hi` both represent the string `"Hello"`. Indeed, the string syntax in C/C++, as in `"foo"`, represents an anonymous array of characters, ending with a null. The standard C library function `strlen` computes the length of a string and there is also `strcmp(S0,S1)` (like the `compareTo` method in Java), `strcpy(S0,S1)` (move the string in `S1` into the array pointed to by `S0`, causing unutterable random catastrophe if there isn't room), and `strcat(S0,S1)` (find the end of `S0` and append `S1` to it, again with the same potential for disaster), among others. These functions come from the standard header file included by

```
#include <string.h>
// < ... > here is for standard library headers
```

The C++ library has a distinct standard type `string`, which you get by putting

```
#include <string>      // No .h
```

at the beginning of your program. The `string` type knows about primitive C strings, generally converting them to type `string` when needed. It is like Java's `StringBuffer`, and supports `+` and `+=` for concatenating strings. Like the primitive strings, `S[i]` for `S` a `string`, is character number `i`.

7 I/O

Standard C has a header file, `stdio.h` (`cstdio` when used in C++) that defines functions `printf` and `fprintf` like the `ucb.io` packages `.format` method. Where in our Java programs, you can write

```
stdout.format ("(%d, %d)\n").put (left).put (right);
outfile.format ("%s %s").put (Name).put (Title);
```

in C, you can write

```
printf ("(%d, %d)\n", left, right);
fprintf (outfile, "%s %s", Name, Title);
```

and where in our programs you can write

```
stdin.scan ("%d %d ");
x = stdin.intItem(0); y = stdin.intItem (1);
inp.scan ("%d %d ");
x = stdin.intItem(0); y = stdin.intItem (1);
```

in C, you can write:

```
scanf ("%d %d ", &x, &y);   // Returns -1 on EOF.
fscanf (inp, "%d %d ", &x, &y);
```

In C++, there is a library, `iostream.h`, that provides for something closer to `System.out` in Java:

```
cout << "(" << left << ", " << right << ")" << endl;
```

is like Java's

```
System.out.println ("(" + left + ", " + right + ")");
```

Input is rather nice; you can write

```
int x, y;
string z;
cin >> x >> y >> z;
```

and input integers into `x` and `y` and a string (delimited by whitespace) into `z` (which automatically expands to receive it).