Reference and Constant Parameters:

If you want to change a parameter, you have have its address, which means you have to pass it by reference.

void F (int x) // this passes X by value; you can't change it
void F (int& x) // this passes x by reference; you can change it

In general, value parameters are used for input, and reference parameters are used for input/output (maybe just output).

For pure functions, you would normally want to pass all the parameters by values, because pure functions don't have side effects, and so they don't modify their parameters.

There is one problem. For "big" parameters (e.g., vectors), it may be much more efficient to pass them by reference than by value, because to pass them by value you have to make a whole fresh copy of the actual parameter. This can take more space, but especially more time than passing by reference, which just copies the address of the actual parameter (a few bytes).

As a result C++ programmers pass large parameters by reference, even if they have no intention of modifying them.

```
void printVector (vector<int> vec) { // vec is passed by value
  for (int i = 0; i<vec.size(); i++) {
    cout << vec[i] << " ";
  }
}
```

What most programmers would do:

void printVector (vector<int>& vec) { // vec is passed by reference

This is efficient, but it's misleading, since it suggests that vec might be modified by printVector. The interface is not clear.

Problem: Reference parameters are being used for two different purposes: output from a function and efficient passing of large arguments. This makes programs harder to understand.

C++ solution: constant parameters. When you put "const" in front of a parameter, it means that it cannot be modified in the function. (In other words, you'll get a compile-time error if you try to assign to it.)

void printVector (const vector<int>& vec) /* passes vec as a "constant reference" */

It has the efficiency of a reference parameter and the security of a value parameter.

Constant "variables" in C++:

```
const int upperBound = vec.size() - 1; // a constant variable
// i.e., it is a variable that cannot be changed.
upperBound++; // illegal
upperBound = N+1; // illegal
```

These are useful for giving meaningful names to things, such as intermediate

results.

In principle, you should leave "const" off of variable declarations only if the variable does in fact vary (i.e., it gets assigned to more than once).

The more you can make your assumptions explicit, the better.
* For the human reader.
* In cases like const, the compiler can check to make sure you are obeying your own assumptions.

You know from the book that you usually have a choice whether you want a function to be a member function or a non-member (standalone) function. E.g., consider add() for two Times.

Nonmember function: add (t1, t2)
Member function: t1.add(t2)

In each case the function returns a Time. It doesn't modify the input parameters, so they should be passed as constants.

Nonmember function:

Time add (const Time& t1, const Time& t2); // nonmember

For a member function the first Time is passed implicitly, so how do we say it should be constant?

Time add (const Time& t2) const; /* trailing "const" means implicit first parameter is passed as a constant, and so t1.add(t2) cannot modify t1. */

Function Declaration vs. Definition:

A function declaration gives the name of a function and the types of its inputs and outputs. It's an interface specification. It doesn't say how to to it.

```
void printVector (const vector<int>& vec); // declaration
```

A function definition gives the implementation of the function.

```
void printVector (const vector<int>& vec) { // definition
  for (int i = 0; i<vec.size(); i++) {
    cout << vec[i] << " ";
  }
}
```

When we declare a structure (or class), we can either
(1) define a member function,
(2) declare a member function, and define it elsewhere.

Note that Time is a new data type that we have in effect added to the C++ language. C++ is called an extensible language. You can define your own data types, classes, and other kinds of objects. You can overload the operators to work on your new data types.