# 7

# Technicalities: Functions, etc.

*No amount of genius can overcome*
*obsession with detail.*
*– Traditional*

In this chapter and the next, we change our focus from programming to our main tool for programming: the C++ programming language. We present language-technical details to give a slightly broader view of C++'s basic facilities and to provide a more systematic view of those facilities. These chapters also act as a review of many of the programming notions presented so far and provide an opportunity to explore our tool without adding new programming techniques or concepts.

## 7.1 Technicalities

Given a choice, we'd much rather talk about programming than about programming language features; that is, we consider how to express ideas as code far more interesting than the technical details of the programming language that we use to express those ideas. To pick an analogy from natural languages: we'd much rather discuss the ideas in a good novel and the way those ideas are expressed than study the grammar and vocabulary of English. What matters are ideas and how those ideas can be expressed in code, not the individual language features.

CC However, we don't always have a choice. When you start programming, your programming language is a foreign language for which you need to look at "grammar and vocabulary." This is what we will do in this chapter and the next, but please don't forget:

- Our primary study is programming.
- Our output is programs/systems.
- A programming language is (only) a tool.

Keeping this in mind appears to be amazingly difficult. Many programmers come to care passionately about apparently minor details of language syntax and semantics. In particular, too many get the mistaken belief that the way things are done in their first programming language is "the one true way." Please don't fall into that trap. C++ is in many ways a very nice language, but it is not perfect; neither is any other programming language.

CC Most design and programming concepts are universal, and many such concepts are widely supported by popular programming languages. That means that the fundamental ideas and techniques we learn in a good programming course carry over from language to language. They can be applied – with varying degrees of ease – in all languages. The language technicalities, however, are specific to a given language. Fortunately, programming languages do not develop in a vacuum, so much of what you learn here will have reasonably obvious counterparts in other languages. In particular, C++ belongs to a group of languages that also includes C (PPP2.Ch27), Java, and C#, so quite a few technicalities are shared with those languages.

Note that when we are discussing language-technical issues, we deliberately use nondescriptive names, such as `f`, `g`, `X`, and `y`. We do that to emphasize the technical nature of such examples, to keep those examples very short, and to try to avoid confusing you by mixing language technicalities and genuine program logic. When you see nondescriptive names (such as should never be used in real code), please focus on the language-technical aspects of the code. Technical examples typically contain code that simply illustrates language rules. If you compiled and ran them, you'd get many "variable not used" warnings, and few such technical program fragments would do anything sensible.

Please note that what we write here is not a complete description of C++'s syntax and semantics – not even for the facilities we describe. The 2023 ISO C++ standard is about 1600 pages of dense technical language aimed at experienced programmers (about 3/4 defines the standard library). We do not try to compete with the standard in completeness and comprehensiveness; we compete in comprehensibility and value for time spent reading.

## 7.2 Declarations and definitions

A *declaration* is a statement that introduces a name into a scope (§7.3)

- Specifying a type for what is named (e.g., a variable or a function)
- Optionally, specifying an initializer (e.g., an initializer value or a function body)

For example:

```
int a = 7;                      // an int variable
const double cd = 8.7;          // a double-precision floating-point constant
double sqrt(double);            // a function taking a double argument and returning a double result
vector<Token> v;                // a vector-of-Tokens variable
```

Before a name can be used in a C++ program, it must be declared. Consider:

```
int main()
{
    std::cout << f(i) << '\n';
}
```

The compiler will give at least four "undeclared identifier" errors for this: **std**, **cout**, **f**, and **i** are not declared anywhere in this program fragment. We can get **cout** declared by **import**ing the standard-library module **std**, which contains its declaration:

```
import std;          // we find the declaration of cout in here

int main()
{
    std::cout << f(i) << '\n';
}
```

Now, we get only two "undefined" errors. As you write real-world programs, you'll find that most declarations are found in modules or headers (§7.7). That's where we define interfaces to useful facilities defined "elsewhere." Basically, a declaration defines how something can be used; it defines the interface of a function, variable, or class. Please note one obvious but invisible advantage of this use of declarations: we didn't have to look at the details of how **cout** and its **<<** operators were defined; we just **import**ed their declarations. We didn't even have to look at their declarations; from textbooks, manuals, code examples, or other sources, we just know how **cout** is supposed to be used. The compiler reads the declarations in the header that it needs to "understand" our code.

However, we still have to declare **f** and **i**. We could do that like this:

```
import std;              // we find the declaration of cout in here

int f(int);              // declaration of f

int main()
{
    int i = 7;           // declaration of i
    cout << f(i) << '\n';
}
```

This will compile because every name has been declared, but it will not link (§1.4) because we have not defined `f()`; that is, nowhere have we specified what `f()` actually does.

> TRY THIS
>
> Compile the three examples above to see how the compiler complains. Then add a definition of `f()` to get a running version.

A declaration that (also) fully specifies the entity declared is called a *definition*. For example:

```
int a = 7;
vector<double> v;
double sqrt(double d) { /* ... */ }
```

Every definition is (by definition) also a declaration, but only some declarations are also definitions. Here are some examples of declarations that are not definitions; if the entity it refers to is used, each must be matched by a definition elsewhere in the code:

```
double sqrt(double);      // no function body here
extern int a;             // "extern plus no initializer" means "not definition"
```

When we contrast definitions and declarations, we follow convention and use *declarations* to mean ''declarations that are not definitions'' even though that's slightly sloppy terminology.

A definition specifies exactly what a name refers to. In particular, a definition of a variable sets aside memory for that variable. Consequently, you can't define something twice. For example:

```
double sqrt(double d) { /* ... */ }      // definition
double sqrt(double d) { /* ... */ }      // error: double definition

int a;                // definition
int a;                // error: double definition
```

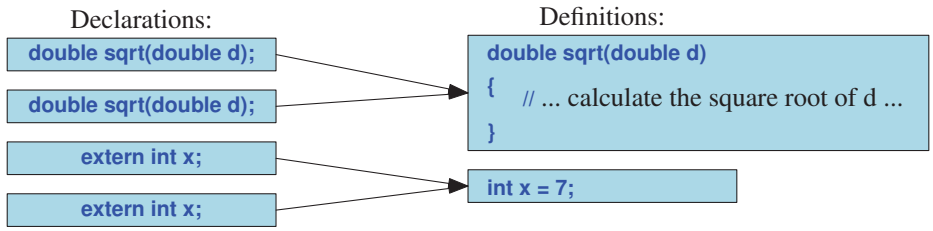
In contrast, a declaration that isn't also a definition simply tells how you can use a name; it is just an interface and doesn't allocate memory or specify a function body. Consequently, you can declare something as often as you like as long as you do so consistently:

```
int x = 7;                // definition
extern int x;             // declaration
extern int x;             // another declaration

double sqrt(double);                      // declaration
double sqrt(double d) { /* ... */ }       // definition
double sqrt(double);                      // another declaration of sqrt
double sqrt(double);                      // yet another declaration of sqrt

int sqrt(double);                         // error: inconsistent declarations of sqrt
```

Why is that last declaration an error? Because there cannot be two functions called `sqrt` taking an argument of type `double` and returning different types (`int` and `double`).

The `extern` keyword used in the second declaration of `x` simply states that this declaration of `x` isn't a definition. It is rarely useful. We recommend that you don't use it, but you'll see it in other people's code, especially code that uses too many global variables (see §7.3 and §7.5.2).

We can represent the difference between definitions and declarations that are not definitions graphically:




Why does C++ offer both declarations and definitions? The declaration/definition distinction reflects the fundamental distinction between what we need to use something (an interface) and what we need for that something to do what it is supposed to (an implementation). For a variable, a declaration supplies the type but only the definition supplies the object (the memory). For a function, a declaration again provides the type (argument types plus return type) but only the definition supplies the function body (the executable statements). Note that function bodies are stored in memory as part of the program, so it is fair to say that function and variable definitions consume memory, whereas declarations don't. CC

The declaration/definition distinction allows us to separate a program into many parts that can be compiled separately. The declarations allow each part of a program to maintain a view of the rest of the program without bothering with the definitions in other parts. As all declarations (including the one definition) must be consistent, the use of names in the whole program will be consistent. We'll discuss that further in §7.7. Here, we'll just remind you of the expression parser from Chapter 5: **expression()** calls **term()** which calls **primary()** which calls **expression()**. Since every name in a C++ program has to be declared before it is used, there is no way we could just define those three functions:

```
double expression();      // just a declaration, not a definition

double primary()
{
    // ...
    expression();
    // ...
}

double term()
{
    // ...
    primary();
    // ...
}
```

```
double expression()
{
    // ...
    term();
    // ...
}
```

We can order those four functions any way we like; there will always be one call to a function defined below it. Somewhere, we need a "forward" declaration. Therefore, we declared **expression()** before the definition of **primary()** and all is well. Such cyclic call chains are very common.

Why does a name have to be declared before it is used? Couldn't we just require the language implementation to read the program (just as we do) and find the definition to see how a function must be called? We could, but that would lead to "interesting" technical problems, especially in large programs, so we decided against that. The C++ definition requires declaration before use (except for class members; see §8.4.4). Also, this is the convention for ordinary (non-program) writing: when you read a textbook, you expect the author to define terminology before using it; otherwise, you have to guess or go to the index all the time. Having to know the declarations only of what we use saves us (and the compiler) from looking through huge amounts of program text.

### 7.2.1 Kinds of declarations

There are many kinds of entities that a programmer can define in C++. The most interesting are

- Variables and constants (§7.2.2)
- Functions (§7.4)
- Namespaces (§7.6)
- Modules (§7.7)
- Types (classes and enumerations; Chapter 8)
- Templates (Chapter 18)
- Concepts (§18.1.3)

### 7.2.2 Variable and constant declarations

The declaration of a variable or a constant specifies a name, a type, and optionally an initializer:

```
int a;                      // no initializer
double d = 7;               // initializer using the = syntax
vector<int> vi(10);         // initializer using the ( ) syntax
vector<int> vi2 {1,2,3,4}; // initializer using the { } syntax
```

Constants have the same declaration syntax as variables. They differ in having **const** or **constexpr** as part of their type and requiring an initializer (§3.3.1):

```
const int x = 7;      // initializer using the = syntax
const int x2 {9};     // initializer using the {} syntax
const int y;          // error: no initializer
```

AA The reason for requiring an initializer for a **const** is obvious: how could a **const** be a constant if it didn't have a value? It is almost always a good idea to initialize variables also; an uninitialized variable is a recipe for obscure bugs. For example:

```
void f(int z)
{
        int x;          // uninitialized
        // ...
        x = 7;          // give x a value
        // ...
}
```

This looks innocent enough, but what if the first `...` included a use of `x`? For example:

```
void f(int z)
{
        int x;          // uninitialized
        // ... no assignment to x here ...
        if (z>x) {
                // ...
        }
        x = 7;          // give x a value
        // ...
}
```

Because `x` is uninitialized, the result of executing `z>x` would be undefined. The comparison `z>x` could give different results on different machines and different results in different runs of the program on the same machine. In principle, `z>x` might cause the program to terminate with a hardware error, but most often that doesn't happen. Instead we get unpredictable results.

Naturally, we wouldn't do something like that deliberately, but if we don't consistently initialize variables it will eventually happen by mistake. Remember, most "silly mistakes" (such as using an uninitialized variable before it has been assigned to) happen when you are busy or tired. Compilers try to warn, but in complicated code – where such errors are most likely to occur – compilers are not smart enough to catch all such errors. There are people who are not in the habit of initializing their variables. Some refrain because they learned to program in languages that didn't allow or encourage consistent initialization. Others are used to languages where every variable is initialized to a default value (which may or may not be appropriate in all cases). Therefore, you'll see examples of uninitialized variables in other people's code. Please just don't add to the problem by forgetting to initialize the variables you define yourself.

We have a preference for the `=` syntax for initializations that just copy a value and for the `{ }` initializer syntax for initializations that do more complex construction.

### 7.2.3 Default initialization

You might have noticed that we often don't provide an initializer for `string`s and `vector`s. For example:

```
vector<string> v;
string s;
while (cin>>s)
        v.push_back(s);
```

This is not an exception to the rule that variables must be initialized before use. What is going on here is that **string** and **vector** are defined so that variables of those types are initialized with a default value whenever we don't supply one explicitly. Thus, **v** is empty (it has no elements) and **s** is the empty string (**""**) before we reach the loop. The mechanism for guaranteeing default initialization is called a *default constructor* (§8.4.2).

Unfortunately, the language doesn't allow us to make such guarantees for built-in types. However, uninitialized variables are a significant bug source and the Core Guidelines ban them [CG: ES.20]. You have been warned!

## 7.3 Scope

**CC** A *scope* is a region of program text. A name is declared in a scope and is valid (is "in scope") from the point of its declaration until the end of the scope in which it was declared. For example:

```
void f()
{
    g();            // error: g() isn't (yet) in scope
}

void g()
{
    f();            // OK: f() is in scope
}

void h()
{
    int x = y;      // error: y isn't (yet) in scope
    int y = x;      // OK: x is in scope
    g();            // OK: g() is in scope
}
```

Names in a scope can be seen from within scopes nested within it. For example, the call of **f()** is within the scope of **g()** which is "nested" in the global scope. The global scope is the scope that's not nested in any other. The rule that a name must be declared before it can be used still holds, so **f()** cannot call **g()**.

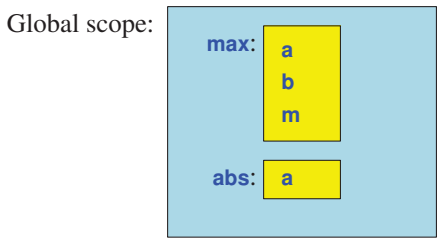
There are several kinds of scopes that we use to control where our names can be used:

- The *global scope:* the area of text outside any other scope
- A *module scope*: the area of text within a module (§7.7.1)
- A *namespace scope*: a named scope nested in the global scope or in another namespace (§7.6)
- A *class scope*: the area of text within a class (§8.2)
- A *local scope*: between **{ ... }** braces of a block or in a function argument list
- A *statement scope*: e.g., in a **for**-statement

The main purpose of a scope is to keep names local, so that they won't interfere with names declared elsewhere. For example:

```
int max(int a, int b)        // max is global; a and b are local
{
        int m;               // m is local
        if (a>=b)
                m = a;
        else
                m = b;
        return m;
}

int abs(int a)               // abs is global; a is local
{
        return (a>=0) ? a : –a;
}
```

Or graphically:




The **a** in **max()** is different from the **a** in **abs()**. They don't "clash" because they are in different scopes. Two incompatible declarations in the same scope is often referred to as a *clash*.

The **?:** construct used in **abs()** is called an *arithmetic if* or a *conditional expression*. The value of **(a>=0)?a:–a** is **a** if **a>=0** and **–a** otherwise. In many cases, a conditional expression saves us from writing long-winded code using **if** like the one in **max()**. It also saves us from the temptation to have an uninitialized variable because "we are just about to assign to it." You find **max()** and **abs()** in the standard library, so you don't have to write them yourself.

So, with the notable exception of the global scope, a scope keeps names local. For most purposes, locality is good, so keep names as local as possible. When I declare my variables, functions, etc. within functions, classes, namespaces, etc., they won't interfere with yours. Remember: Real programs have *many* thousands of named entities. To keep such programs manageable, most names have to be local. AA

Here is a larger technical example illustrating how names go out of scope at the end of statements and blocks (including function bodies):

```
// no r, i, or v here
class My_vector {
        vector<int> v;                    // v is in class scope
```

```
public:
        int largest()
        {
                int r = 0;                              // r is local
                for (int i = 0; i<v.size(); ++i)
                        r = max(r,abs(v[i]));           // i is in the for's statement scope
                // ... no i here ...
                return r;
        }
        // ... no r here ...
};
// no v here

int x = 0;              // a global variable – avoid those where you can
int y = 0;

int f()
{
        int x = 0;      // a local variable, hides the global x
        x = 7;          // the local x
        {
                int x = y;      // local x initialized by global y, hides the previous local x
                ++x;            // the x from the previous line
        }
        ++x;            // the x from the first line of f()
        return x;
}
```

Whenever you can, avoid such complicated nesting and hiding. Remember: "Keep it simple!"

The larger the scope of a name is, the longer and more descriptive its name should be: **x**, **y**, and **f** are horrible as global names. The main reason that you don't want global variables in your program is that it is hard to know which functions modify them. In large programs, it is basically impossible to know which functions modify a global variable. Imagine that you are trying to debug a program and you find that a global variable has an unexpected value. Who gave it that value? Why? What functions write to that value? How would you know? The function that wrote a bad value to that variable may be in a source file you have never seen! A good program will have only very few (say, one or two), if any, global variables. For example, the calculator in Chapter 5 and Chapter 6 has two global variables: the token stream, **ts**, and the symbol table, **names**.

Note that most C++ constructs that define scopes nest:

- Functions within classes: member functions (see §8.4.2)

```
class C {
public:
        void f();
        void g() { /* ... */ }          // a member function can be defined within its class
        // ...
};
```

```
void C::f()                    // a member definition can be outside its class
{
        // ...
}
```

This is the most common and useful case.

- Classes within classes: member classes (also called nested classes)

```
class C {
public:
        class M {
                // ...
        };
        // ...
};
```

This tends to be useful only in complicated classes; remember that the ideal is to keep classes small and simple.

- Classes within functions: local classes

```
void f()
{
        class L {
                // ...
        };
        // ...
}
```

- Functions within functions: local functions (also called nested functions)

```
void f()
{
        void g() { /* ... */ }      // error: nested function
        // ...
}
```

Function nesting is not legal in C++. Instead, use a lambda (§13.3.3, §21.2.3).

- Blocks within functions and other blocks: nested blocks

```
void f(int x, int y)
{
        if (x>y) {
                // ...
        }
        else {
                // ...
                {
                        // ...
                }
                // ...
        }
}
```

XX Nested blocks are unavoidable, but be suspicious of complicated nesting: it can easily hide errors. Similarly, if you feel the need for a local class, your function is probably far too long.

AA We use indentation to indicate nesting. Without consistent indentation, nested constructs become unreadable. Consider:

```
// dangerously ugly code
struct X {
void f(int x) {
struct Y {
int f() { return 1; } int m; };
int m;
m=x; Y m2;
return f(m2.f()); }
int m; void g(int m) {
if (0<m) f(m+2); else {
g(m+2.3); }}
X() { } int m3() {
}

void main() {
X a; a.f(2);}
};
```

Hard-to-read code usually hides bugs. When you use an IDE, it tries to automatically make your code properly indented (according to some definition of "properly"), and there exist "code beautifiers" that will reformat a source code file for you (often offering you a choice of formats). However, the ultimate responsibility for your code being readable rests with you.

> TRY THIS
>
> Type in the example above and indent it properly. What suspicious constructs and bugs can you now find?

## 7.4 Function call and return

CC Functions are the way we represent actions and computations. Whenever we want to do something that is worthy of a name, we write a function. The C++ language gives us operators (such as `+` and `*`) with which we can produce new values from operands in expressions, and statements (such as `for` and `if`) with which we can control the order of execution. To organize code made out of these primitives, we have functions.

To do its job, a function usually needs arguments, and many functions return a result. This section focuses on how arguments are specified and passed.

### 7.4.1 Declaring arguments and return type

Functions are what we use in C++ to name and represent computations and actions. A function declaration consists of a return type followed by the name of the function followed by a list of parameters in parentheses. For example:

```
double fct(int a, double d);          // declaration of fct (no body)

double fct(int a, double d)           // definition of fct
{
    return a*d;
}
```

A definition contains the function body (the statements to be executed by a call), whereas a declaration that isn't a definition just has a semicolon. Parameters are often called *formal arguments*. If you don't want a function to take arguments, just leave out the formal arguments. For example:

```
int current_power();                  // current_power doesn't take an argument
```

If you don't want to return a value from a function, give **void** as its return type. For example:

```
void increase_power_to(int level);  // increase_power_to() doesn't return a value
```

Here, **void** means "doesn't return a value" or "return nothing."

You can name a parameter or not as it suits you in both declarations and definitions. For example:

```
int my_find(vector<string> vs, string s, int hint);   // naming arguments: search starting from hint

int my_find(vector<string>, string, int);             // not naming arguments
```

In declarations, formal argument names are not logically necessary, just very useful for writing good comments. From a compiler's point of view, the second declaration of **my_find()** is just as good as the first: it has all the information necessary to call **my_find()**. **CC**

Usually, we name all the arguments in the definition. For example:

```
int my_find(vector<string> vs, string s, int hint)
    // search for s in vs starting at hint
{
    if (hint<0 || vs.size()<=hint)
        hint = 0;
    for (int i = hint; i<vs.size(); ++i)      // search starting from hint
        if (vs[i]==s)
            return i;
    for (int i = 0; i<hint; ++i)              // if we didn't find s, so search before hint
        if (vs[i]==s)
            return i;
    return –1;
}
```

The **hint** complicates the code quite a bit, but the **hint** was provided under the assumption that users could use it to good effect by knowing roughly where in the **vector** a **string** will be found. However, imagine that we had used **my_find()** for a while and then discovered that callers rarely used **hint** well, so that it actually hurts performance. Now we don't need **hint** anymore, but there is lots of code "out there" that calls **my_find()** with a **hint.** We don't want to rewrite that code (or can't because it is someone else's code), so we don't want to change the declaration(s) of **my_find()**. Instead, we just don't use the last argument. Since we don't use it we can leave it unnamed:

```
int my_find(vector<string> vs, string s, int)    // 3rd argument unused
{
    for (int i = 0; i<vs.size(); ++i)
        if (vs[i]==s) return i;
    return –1;
}
```

### 7.4.2 Returning a value

We return a value from a function using a **return**-statement:

```
T f()    // f() returns a T
{
    V v;
    // ...
    return v;
}
```

Here, the value returned is exactly the value we would have gotten by initializing a variable of type **T** with a value of type **V**:

```
V v;
// ...
T t(v);    // initialize t with v
```

That is, value return is a form of initialization. Unfortunately, it is the potentially narrowing form of initialization (§2.9), but compilers often warn and the Core Guidelines will catch narrowing.

A function declared to return a value must return a value. In particular, it is an error to "fall through the end of the function":

```
double my_abs(int x)    // warning: buggy code
{
    if (x < 0)
        return –x;
    else if (x > 0)
        return x;
}    // error: no value returned if x is 0
```

Actually, the compiler probably won't notice that we "forgot" the case **x==0**. In principle it could, but few compilers are that smart. For complicated functions, it can be impossible for a compiler to know whether or not you return a value, so be careful. Here, "being careful" means to make really sure that you have a **return**-statement or an **error()** for every possible way out of the function.

For historical reasons, **main()** is a special case. Falling through the bottom of **main()** is equivalent to returning the value **0**, meaning "successful completion" of the program.

In a function that does not return a value, we can use **return** without a value to cause a return from the function. For example:

```
void print_until(vector<string> v, string quit)
    // print until the string called "quit" is found
{
    for (string s : v) {
        if (s==quit)
            return;
        cout << s << '\n';
    }
}
```

As you can see, it is acceptable to "drop through the bottom" of a **void** function. This is equivalent to a **return;**.

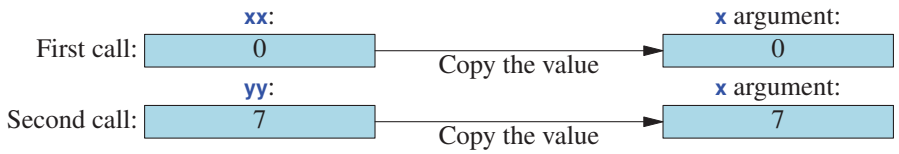
### 7.4.3 Pass-by-value

CC

The simplest way of passing an argument to a function is to give the function a copy of the value you use as the argument. An argument of a function **f()** is a local variable in **f()** that's initialized each time **f()** is called. For example:

```
// pass-by-value (give the function a copy of the value passed)
int f(int x)
{
    x = x+1;        // give the local x a new value
    return x;
}

int xx = 0;
cout << f(xx) << '\n';      // write: 1
cout << xx << '\n';         // write: 0; f() doesn't change xx

int yy = 7;
cout << f(yy) << '\n';      // write: 8
cout << yy << '\n';         // write: 7; f() doesn't change yy
```

Since a copy is passed, the **x=x+1** in **f()** does not change the values **xx** and **yy** passed in the two calls. We can illustrate a pass-by-value argument passing like this:




Pass-by-value is pretty straightforward, and its cost is the cost of copying the value.

### 7.4.4 Pass-by-const-reference

Pass-by-value is simple, straightforward, and efficient when we pass small values, such as an **int**, a **double**, or a **Token** (§5.3.2). But what if a value is large, such as an image (often, several million bits), a large table of values (say, thousands of integers), or a long string (say, hundreds of characters)? Then, copying can be costly. We should not be obsessed by cost, but doing unnecessary work can be embarrassing because it is an indication that we didn't directly express our idea of what we wanted. For example, we could write a function to print out a **vector** of floating-point numbers like this:

```
void print(vector<double> v)        // pass-by-value; appropriate?
{
        cout << "{ ";
        for (int i = 0; i<v.size(); ++i) {
                cout << v[i];
                if (i!=v.size()–1)
                        cout << ", ";
        }
        cout << " }\n";
}
```

We could use this **print()** for **vector**s of all sizes. For example:

```
void f(int x)
{
        vector<double> vd1(10);             // small vector
        vector<double> vd2(1000000);        // large vector
        vector<double> vd3(x);              // vector of some unknown size

        // ... fill vd1, vd2, vd3 with values ...

        print(vd1);
        print(vd2);
        print(vd3);
}
```

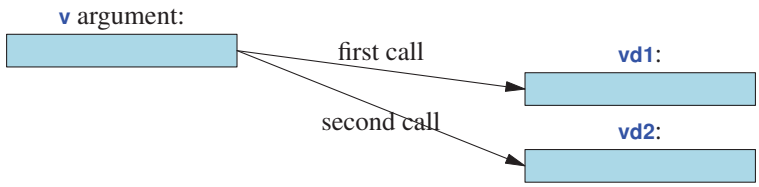
**CC** This code works, but the first call of **print()** has to copy ten **double**s (probably 80 bytes), the second call has to copy a million **double**s (probably 8 megabytes), and we don't know how much the third call has to copy. The question we must ask ourselves here is: "Why are we copying anything at all?" We just wanted to print the **vector**s, not to make copies of their elements. Obviously, there has to be a way for us to pass a variable to a function without copying it. As an analogy, if you were given the task to make a list of books in a library, the librarians wouldn't ship you a copy of the library building and all its contents; they would send you the address of the library, so that you could go and look at the books. So, we need a way of giving our **print()** function "the address" of the **vector** to **print()** rather than the copy of the **vector**. Such an "address" is called a *reference* and is used like this:

```
void print(const vector<double>& v)          // pass-by-const-reference
{
        cout << "{ ";
        for (int i = 0; i<v.size(); ++i) {
                cout << v[i];
                if (i!=v.size()–1)
                        cout << ", ";
        }
        cout << " }\n";
}
```

The **&** means ‘‘reference’’ and the **const** is there to stop **print()** from modifying its argument by accident. Apart from the change to the argument declaration, all is the same as before; the only change is that instead of operating on a copy, **print()** now refers back to the argument through the reference. Note the phrase ‘‘refer back’’; such arguments are called references because they ‘‘refer’’ to objects defined elsewhere. We can call this **print()** exactly as before:

```
void f(int x)
{
        vector<double> vd1(10);          // small vector
        vector<double> vd2(1000000);     // large vector
        vector<double> vd3(x);           // vector of some unknown size

        // ... fill vd1, vd2, vd3 with values ...
        print(vd1);
        print(vd2);
        print(vd3);
}
```

We can illustrate that graphically:




Compare to the pass-by-value example in §7.4.3.

A **const** reference has the useful property that we can’t accidentally modify the object passed. For example, if we made a silly error and tried to assign to an element from within **print()**, the compiler would catch it:

```
void print(const vector<double>& v)          // pass-by-const-reference
{
        // ...
        v[i] = 7;               // error: v is a const (is not mutable)
        // ...
}
```

Pass-by-**const**-reference is a useful and popular mechanism. Consider again the **my_find()** function (§7.4.1) that searches for a **string** in a **vector** of **string**s. Pass-by-value could be unnecessarily costly:

```
int my_find(vector<string> vs, string s);  // pass-by-value: copy
```

If the **vector** contained many thousands of **string**s, you might notice the time spent even on a fast computer. So, we could improve **my_find()** by making it take its arguments by **const** reference:

```
int my_find(const vector<string>& vs, const string& s); // pass-by-const-reference: no copy, read-only
```

### 7.4.5 Pass-by-reference

But what if we did want a function to modify its arguments? Sometimes, that's a perfectly reasonable thing to wish for. For example, we might want an **init()** function that assigns values to **vector** elements:

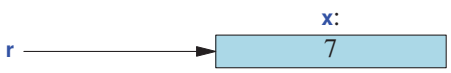
```
void init(vector<double>& v)          // pass-by-reference
{
      for (int i = 0; i<v.size(); ++i)
            v[i] = i;
}

void g(int x)
{
      vector<double> vd1(10);          // small vector
      vector<double> vd2(1000000);     // large vector
      vector<double> vd3(x);           // vector of some unknown size

      init(vd1);
      init(vd2);
      init(vd3);
}
```

Here, we wanted **init()** to modify the argument vector, so we did not copy (did not use pass-by-value) or declare the reference **const** (did not use pass-by-**const**-reference) but simply passed a "plain reference" to the **vector**.

Let us consider references from a more technical point of view. A reference is a construct that allows us to declare a new name for an object. For example, **int&** is a reference to an **int**, so we can write

```
int x = 7;
int& r = x;
```

Or graphically:



That is, any use of r is really a use of x.

References can be useful as shorthand. For example, we might have a

```
vector< vector<double> > v;  // vector of vector of double
```

and we need to refer to some element v[f(x)][g(y)] several times. Clearly, v[f(x)][g(y)] is a complicated expression that we don't want to repeat more often than we have to. If we just need its value, we could write

```
double val = v[f(x)][g(y)];      // val is the value of v[f(x)][g(y)]
```

and use val repeatedly. But what if we need to both read from v[f(x)][g(y)] and write to v[f(x)][g(y)]? Then, a reference comes in handy:

```
double& var = v[f(x)][g(y)];     // var is a reference to v[f(x)][g(y)]
```

Now we can read and write v[f(x)][g(y)] through var. For example:

```
var = var/2+sqrt(var);
```

This key property of references, that a reference can be a convenient shorthand for some object, is what makes them useful as arguments as shown for print() in §7.4.4.

CC

Pass-by-reference is clearly a very powerful mechanism: we can have a function operate directly on any object to which we pass a reference. For example, swapping two values is an immensely important operation in many algorithms, such as sorting. Using references, we can write a function that swaps doubles like this:

```
void swap(double& d1, double& d2)
{
        double temp = d1;        // copy d1's value to temp
        d1 = d2;                 // copy d2's value to d1
        d2 = temp;               // copy d1's old value to d2
}

int main()
{
        double x = 1;
        double y = 2;
        cout << "x == " << x << " y== " << y << '\n';        // write: x==1 y==2
        swap(x,y);
        cout << "x == " << x << " y== " << y << '\n';        // write: x==2 y==1
}
```

The standard library provides a swap() for every type that you can copy, so you don't have to write swap() yourself for each type.

## 7.4.6 Pass-by-value vs. pass-by-reference

When should you use pass-by-value, pass-by-reference, and pass-by-const-reference? Consider first a technical example:

```
void f(int a, int& r, const int& cr)
{
    ++a;    // change the local a
    ++r;    // change the object referred to by r
    ++cr;   // error: cr is const
}
```

If you want to change the value of the object passed, you must use a non-**const** reference: pass-by-value gives you a copy and pass-by-**const**-reference prevents you from changing the value of the object passed. So we can try

```
void g(int a, int& r, const int& cr)
{
    ++a;          // change the local a
    ++r;          // change the object referred to by r
    int x = cr;   // read the object referred to by cr
}

int main()
{
    int x = 0;
    int y = 0;
    int z = 0;

    g(x,y,z);     // x==0; y==1; z==0
    g(1,2,3);     // error: reference argument r needs a variable to refer to
    g(1,y,3);     // OK: since cr is const we can pass a literal
}
```

So, if you want to change the value of an object passed by reference, you have to pass an object. Technically, the integer literal **2** is just a value (an rvalue), rather than an object holding a value. What you need for **g()**'s argument **r** is an lvalue, that is, something that could appear on the left-hand side of an assignment.

Note that a **const** reference doesn't need an lvalue. It can perform conversions exactly as initialization or pass-by-value. Basically, what happens in that last call, **g(1,y,3)**, is that the compiler sets aside an **int** for **g()**'s argument **cr** to refer to:

```
g(1,y,3);   // means: int compiler_generated = 3; g(1,y, compiler_generated)
```

Such a compiler-generated object is called a *temporary object* or just a *temporary*.

**AA**

Our rule of thumb is:

[1] Use pass-by-value to pass very small objects.
[2] Use pass-by-**const**-reference to pass large objects that you don't need to modify.
[3] Return a result rather than modifying an object through a reference argument.
[4] Use pass-by-reference only when you have to.

These rules lead to the simplest, least error-prone, and most efficient code. By "very small" we mean one or two **int**s, one or two **double**s, or something like that.

That third rule reflects that you have a choice when you want to use a function to change the value of a variable. Consider:

```
int incr1(int a) { return a+1; }        // return the new value as the result
void incr2(int& a) { ++a; }             // modify object passed as reference

int x = 7;
x = incr1(x);           // pretty obvious
incr2(x);               // pretty obscure
```

Why do we ever use non-**const**-reference arguments? Occasionally, they are essential **CC**

- For manipulating containers (e.g., **vector**) and other large objects
- For functions that change several objects

For example:

```
void larger(vector<int>& v1, vector<int>& v2)
    // make each element in v1 the larger of the corresponding elements in v1 and v2;
    // similarly, make each element of v2 the smaller
{
    if (v1.size()!=v2.size())
        error("larger(): different sizes");
    for (int i=0; i<v1.size(); ++i)
        if (v1[i]<v2[i])
            swap(v1[i],v2[i]);
}
```

Using pass-by-reference arguments (or logical equivalents; §19.3.2) is the only reasonable choice for a function like **larger()**.

If we use a reference simply to avoid copying, we use a **const** reference. Consequently, when we see a non-**const**-reference argument, we assume that the function changes the value of its argument; that is, when we see a pass-by-non-**const**-reference we assume that not only can that function modify the argument passed, but it will, so that we have to look extra carefully at the call to make sure that it does what we expect it to. **AA**

## 7.4.7 Argument checking and conversion

Passing an argument is the initialization of the function's formal argument with the actual argument specified in the call. Consider:

```
void f(T x);
f(y);
T x = y;                // initialize x with y (see §7.2.2)
```

The call **f(y)** is legal whenever the initialization **T x = y;** is, and when it is legal both **x**s get the same value. For example:

```
void f(double x);

void g(int y)
{
    f(y);
    double x = y;       // initialize x with y (see §7.2.2)
}
```

Note that to initialize x with y, we have to convert an int to a double. The same happens in the call of f(). The double value received by f() is the same as the one stored in x.

XX Conversions are often useful, but occasionally they give surprising results (see §2.9). Consequently, we have to be careful with them and hope for compiler warnings. For example:

```
g(7.8);          // truncate 7.8 to 7; did you really mean to do that?
int x = 7.8;     // truncate 7.8 to 7; did you really mean to do that?
```

If you really mean to truncate a double value to an int, say so explicitly [CG: ES.46]. We can use narrowing operations from the Core Guideline support library, provided as part of PPP_support: narrow<T>(x): checks x and throws narrowing_error if there would be a loss of information after converting x to a T. For example:

```
void conv1(double y)
{
    int x = narrow<int>(y);          // checked conversion
}
```

That way, the next programmer to look at this code can see that you thought about the potential problem and you'll get an error if information is lost. When, on the other hand, we want rounding, we can use round_to() from PPP_support. For example:

```
void conv2(double y)
{
    int x = round_to<int>(y);        // 4/5 rounding
}
```

In scientific calculations, we often have to convert from integers to floating point values and back. You can find examples in our graphics library (§11.7.5 and §13.3). For conversions from int to double, we use the double(i) notation: For example:

```
void conv3(int x, int y)
{
    double z = double(x)/y;          // x/y would have truncated
}
```

> TRY THIS
>
> Try examples like the ones above converting all combinations of an int, a double, and a char. Use values 1001, 7.7, and 'x'. Try with implicit conversion and narrow. Write out the results for the cases where the program compiles. What errors and warnings did you get?

## 7.4.8 Function call implementation

But how does a computer really do a function call? The expression(), term(), and primary() functions from Chapter 5 and Chapter 6 are perfect for illustrating this except for one detail: they don't take any arguments, so we can't use them to explain how arguments are passed. But wait! They *must* take some input; if they didn't, they couldn't do anything useful. They do take an implicit argument: they use a Token_stream called ts to get their input; ts is a global variable. That's a bit

sneaky. We can improve these functions by letting them take a Token_stream& argument. Here they are with a Token_stream& parameter added and everything that doesn't concern function call implementation removed.

First, expression() is completely straightforward; it has one argument (ts) and two local variables (left and t):

```
double expression(Token_stream& ts)
{
    double left = term(ts);
    Token t = ts.get();
    // ...
}
```

Second, term() is much like expression(), except that it has an additional local variable (d) that it uses to hold the result of a divisor for '/':

```
double term(Token_stream& ts)
{
    double left = primary(ts);
    Token t = ts.get();
    // ...
        case '/':
        {
            double d = primary(ts);
            // ...
        }
    // ...
}
```

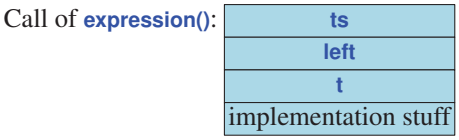
Third, primary() is much like term() except that it doesn't have a local variable left:

```
double primary(Token_stream& ts)
{
    Token t = ts.get();
    switch (t.kind) {
    case '(':
        {   double d = expression(ts);
            // ...
        }
        // ...
    }
}
```

Now they don't use any "sneaky global variables" and are perfect for our illustration: they have an argument, they have local variables, and they call each other. You may want to take the opportunity to refresh your memory of what the complete expression(), term(), and primary() look like, but the salient features as far as function call is concerned are presented here.

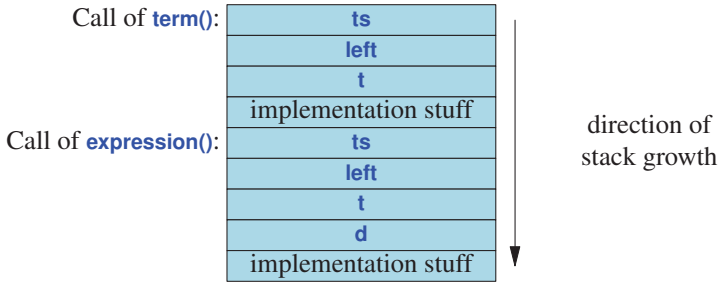
When a function is called, the language implementation sets aside a data structure, called a *function activation record*, containing a copy of all its parameters and local variables. For example, when expression() is first called, the compiler ensures that a structure like this is created:

CC

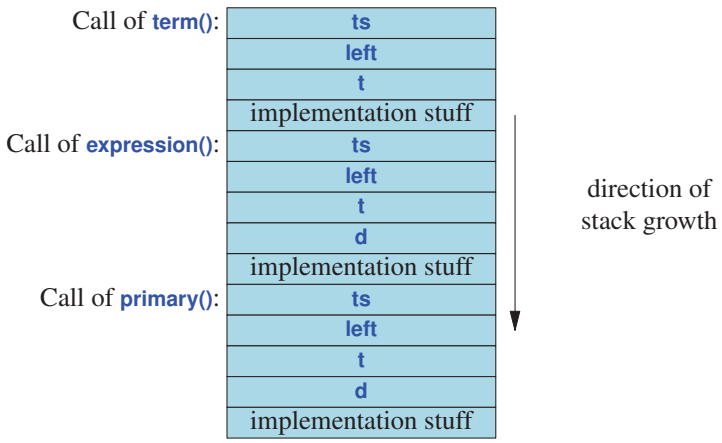



The "implementation stuff" varies from implementation to implementation, but that's basically the information that the function needs to return to its caller and to return a value to its caller. Each function has its own detailed layout of its activation record. Note that from the implementation's point of view, a parameter is just another local variable.

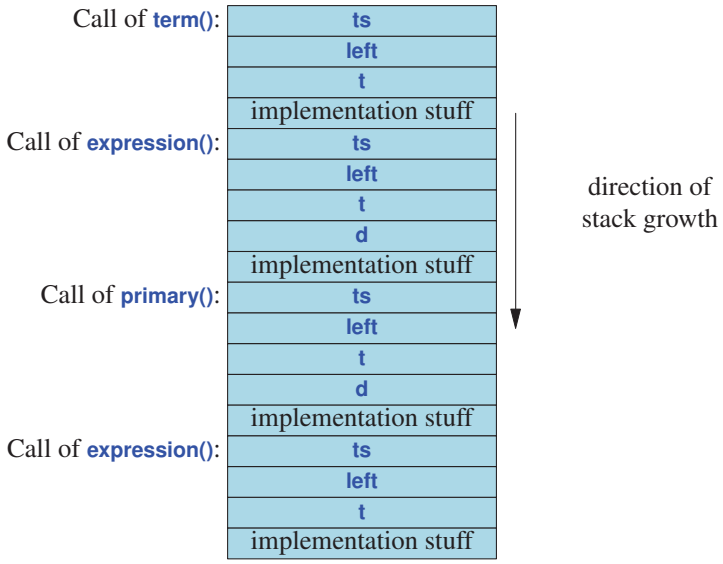
So far, so good, and now **expression()** calls **term()**, so the compiler ensures that an activation record for this call of **term()** is generated:




Note that **term()** has an extra variable **d** that needs to be stored, so we set aside space for that in the call even though the code may never get around to using it. That's OK. For reasonable functions (such as every function we directly or indirectly use in this book), the run-time cost of laying down a function activation record doesn't depend on how big it is. The local variable **d** will be initialized only if we execute its **case '/'**. Next, **term()** calls **primary()** and we get

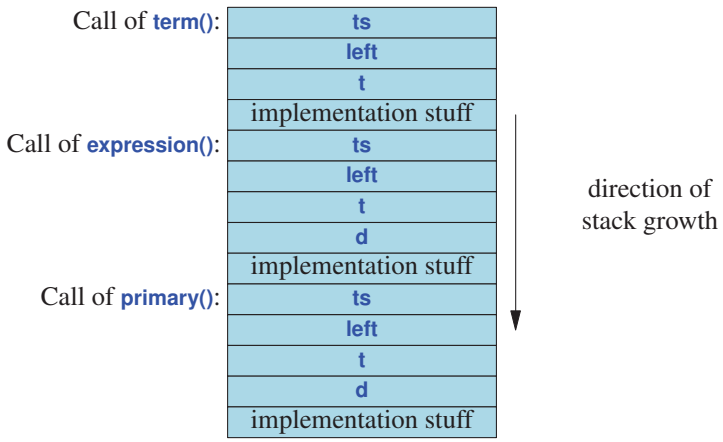


This is starting to get a bit repetitive, but now **primary()** calls **expression()**:

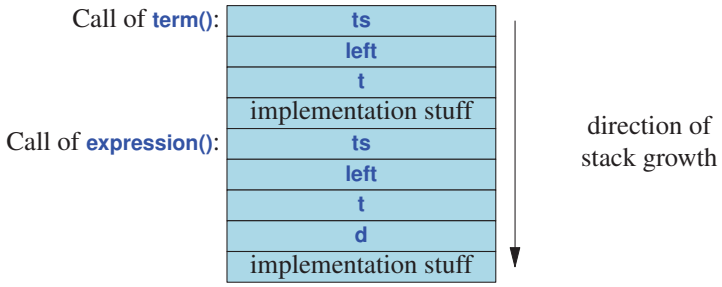



So this call of **expression()** gets its own activation record, different from the first call of **expression()**. That's good or else we'd be in a terrible mess, since **left** and **t** will be different in the two calls. A function that directly or (as here) indirectly calls itself is called *recursive*. **CC**

So, each time we call a function the *stack of activation records*, usually just called the *stack*, grows by one record. Conversely, when the function returns, its record is no longer used. For example, when that last call of **expression()** returns to **primary()**, the stack will revert to this:



And when that call of primary() returns to term(), we get back to




And so on. The stack, also called the *call stack*, is a data structure that grows and shrinks at one end according to the rule "Last in, first out."

Please remember that the details of how a call stack is implemented and used vary from C++ implementation to C++ implementation, but the basics are as outlined here. Do you need to know how function calls are implemented to use them? Of course not! You have done well enough before this implementation subsection, but many programmers like to know and many use phrases like "activation record" and "call stack," so it's better to know what they mean.

## 7.4.9 Compile-time computation

A function represents a calculation, and sometimes we want to do a calculation at compile time. The reason to want a calculation to be evaluated by the compiler is usually to avoid having the same calculation done millions of times at run time. We use functions to make our calculations comprehensible, so naturally we sometimes want to use a function in a constant expression. We convey our intent to have a function evaluated by the compiler by declaring the function constexpr (§3.3.1). A constexpr function can be evaluated by the compiler if it is given constant expressions as arguments. For example:

```
constexpr double xscale = 10;    // scaling factors
constexpr double yscale = 0.8;

constexpr Point scale(Point p) { return {xscale*p.x,yscale*p.y}; };
```

Assume that Point is a simple struct with members x and y representing 2D coordinates. Now, when we give scale() a Point argument, it returns a Point with coordinates scaled according to the factors xscale and yscale. For example:

```
void user(int x, int y)
{
    Point p1 {x,y};                  // at compile time, we don't know the value of p1
    constexpr Point p2 {10,10};

    Point p3 = scale(p1);            // OK: p3 == {100,8}; run-time evaluation is fine
    constexpr Point p4 = scale(p2);  // OK: p4 == {100,8}; scale(p2) is a constant
```

```
        constexpr Point p5 = scale(p1);      // error: scale (p1) is not a constant expression
        // ...
}
```

A **constexpr** function behaves just like an ordinary function until you use it where a constant is needed. Then, it is calculated at compile time and its arguments must be constant expressions (e.g., **p2**) and gives an error if they are not (e.g., **p1**). To enable that, a **constexpr** function must be so simple that the compiler (every standard-conforming compiler) can evaluate it. That includes simple loops and local variables. For example:

```
constexpr int sum(const vector<int>& v)
{
        int s = 0;
        for (int x : v)
                s += x;
        return s;
}
```

When called in a constant expression, a **constexpr** function cannot have side effects; that is, it cannot change the value of objects outside its own body. At compile time, such objects do not exist.

If you want a function that can be evaluated only at compile time, declare it **consteval** rather than **constexpr**. For example:

```
consteval half(double d) { return d/2; }

double x1 = half(7);        // OK: 7 is a constant
double x2 = half(x1);       // error: x1 is a non-const variable
```

## 7.4.10 Suffix return type

The traditional function declaration syntax that we have used so far is:

*type-identifier function-identifier* **(** *parameter-list* **)**

For example:

```
double expression(Token_stream& ts);        // double is the result of calling expression(ts)
```

However, there is another notation that puts the return type to the end where it arguably belongs:

**auto** *function-identifier* **(** *parameter-list* **) –>** *type-identifier*

For example:

```
auto expression(Token_stream& ts) –> double;     // call expression(ts) and get a double
```

This is called the *suffix return type* notation or the *trailing return* notation. It is occasionally essential when the return type is expressed in terms of the argument types and has the nice property that names align. For example:

```
auto Token_stream::get() –> Token;
auto statement() –> double;
auto find_all(string s) –> vector<Variable>;
```

as opposed to

```
Token Token_stream::get();
double statement();
vector<Variable> find_all(string s);
```

The difference becomes more noticeable when the return names become longer and more elaborate.

When using **auto** to introduce a function definition, the return type can be deduced from the **return**-statement:

```
auto expression(Token_stream& ts)       // deduce the return type from the definition
{
        double left = term(ts);
        // ...
        return left;
}
```

Like in **auto** object definitions (§2.10), the return type is deduced from the initializer (here, the function body). This deduction mechanism shouldn't be overused, like **auto** in general shouldn't, because it could lead to harder-to-read code. Worse, the type of a function could be unintentionally changed as the result of a change of its implementation.

## 7.5 Order of evaluation

CC

The evaluation of a program – also called the execution of a program – proceeds through the statements according to the language rules. When this "thread of execution" reaches the definition of a variable, the variable is constructed; that is, memory is set aside for the object and the object is initialized. When the variable goes out of scope, the variable is destroyed; that is, the object it refers to is in principle removed and the compiler can use its memory for something else. For example:

```
string program_name = "silly";
vector<string> v;                       // v is global

void f()
{
        string s;                       // s is local to f
        while (cin>>s && s!="quit") {
                string stripped;        // stripped is local to the loop
                string not_letters;
                for (char x : s)        // x has statement scope
                        if (isalpha(x))
                                stripped += x;
                        else
                                not_letters += x;
                v.push_back(stripped);
        }
        // ... we can still use s here ...
}
```

Global variables, such as **program_name** and **v**, are initialized before the first statement of **main()** is executed. They "live" until the program terminates, and then they are destroyed. They are

constructed in the order in which they are defined (that is, program_name before v) and destroyed in the reverse order (that is, v before program_name).

When someone calls f(), first s is constructed; that is, s is initialized to the empty string. It will live until we return from f().

Each time we enter the block that is the body of the while-statement, stripped and not_letters are constructed. Since stripped is defined before not_letters, stripped is constructed before not_letters. They live until the end of the loop, where they are destroyed in the reverse order of construction (that is, not_letters before stripped) before the condition is reevaluated. So, if ten strings are seen before we encounter the string quit, stripped and not_letters will each be constructed and destroyed ten times.

Each time we reach the for-statement, x is constructed. Each time we exit the for-statement, x is destroyed before we reach the v.push_back(stripped); statement.

Please note that compilers (and linkers) are clever beasts and they are allowed to – and do – optimize code as long as the results are equivalent to what we have described here. In particular, compilers are clever at not allocating and deallocating memory more often than is really necessary. For example, only one stripped is ever used at any time, so the stack frame for f() will contain space for just one stripped which will be reused repeatedly.

### 7.5.1 Expression evaluation

XX

The order of evaluation of sub-expressions is governed by rules designed to please an optimizer rather than to make life simple for the programmer. That's unfortunate, but you should avoid complicated expressions anyway, and there is a simple rule that can keep you out of trouble: if you change the value of a variable in an expression, don't read or write it twice in that same expression. For example:

```
f(++i,++i);                 // don't: undefined order of evaluation
x = ++i + i;                // don't: undefined order of evaluation
z = f(x)+g(y)               // don't if the order of f(x) and g(y) matters
h(f(x),g(y))                // don't if the order of f(x) and g(y) matters
```

Unfortunately, not all compilers warn if you write such bad code; it's bad because you can't rely on the results being the same if you move your code to another computer, use a different compiler or use a different optimizer setting. Compilers really differ for such code; just don't do it.

Fortunately, some order has been imposed. The order of evaluation is left-to-right for x.y, x–>y, x(y), x[y], x<<y, x>>y, x,y, x&&y, and x||y. For assignments (e.g., x=y and x+=y), the order is right-to-left. That actually makes most sensible constructs behave as one would naively expect. For example:

```
if (0<=x && v[x]!=0) ...    // v[x] will never be executed for x<0
v[i] = ++i;                 // i will be incremented before being used as a subscript
cout << ++i << ' ' << ++i;  // will print "2 3" if invoked with i==1
```

For &&, the second (right-hand) operand is not executed unless the first (left-hand) operand is true. Similarly, for ||, the second operand is not executed unless the first operand is false. This is sometimes called *short-circuit evaluation*.

### 7.5.2 Global initialization

XX Using a global variable in anything but the most limited circumstances is usually not a good idea. The programmer has no really effective way of knowing which parts of a large program read and/or write a global variable (§7.3). Unfortunately, global variables are common in older code.

Global variables (and namespace variables; see §7.6) in a single translation unit (§7.7.1) are initialized in the order in which they appear. For example:

```
// file f1.cpp
int x1 = 1;
int y1 = x1+2;          // y1 becomes 3
```

This initialization logically takes place "before the code in **main()** is executed." However, the order of initialization of global variables in different translation units is not defined. For example:

```
// file f2.cpp
extern int y1;
int y2 =y1+2;           // y2 becomes 2 or 5
```

Such code is to be avoided for several reasons: it uses global variables, it gives the global variables short names, and it uses complicated initialization of the global variables. If the globals in file **f1.cpp** are initialized before the globals in **f2.cpp**, **y2** will be initialized to **5** (as a programmer might naively and reasonably expect). However, if the globals in file **f2.cpp** are initialized before the globals in **f1.cpp**, **y2** will be initialized to **2** (because the memory used for global variables is initialized to **0** before complicated initialization is attempted). Avoid such code, and be very suspicious when you see global variables with nontrivial initializers. For global variables, consider any initializer that isn't a constant expression complicated.

But what can we do when we really need a global variable (or constant) with a complicated initializer? A plausible example would a **Date** initialized to today's date at program startup every morning.

```
const Date today = get_date_from_clock();          // suspicious definition
```

How do we know that **today** is never used before it was initialized? Basically, we can't know, so we shouldn't write that definition. The technique that we use most often is to call a function that returns the value. For example:

```
const Date today()
{
    return get_date_from_clock();          // return today's date
}
```

AA This constructs a **Date** every time we call **today()**. If **today()** is called often and if a call to **get_date_from_clock()** is expensive, we'd like to construct that **Date** once only. We can do that by using a **static** local variable:

```
const Date& today()
{
    static const Date today = get_date_from_clock();      // initialize today the first time we get here
    return today;
}
```

This Date is initialized (constructed) the first time its function is called (only). Note that we returned a reference to eliminate unnecessary copying. In particular, we returned a const reference to prevent the calling function from accidentally changing the value. The arguments about how to pass an argument (§7.4.6) also apply to returning values.

## 7.6 Namespaces

We use blocks to organize code within a function (§7.3). We use classes to organize functions, data, and types into a type (Chapter 8). A function and a class both do two things for us:

- They allow us to define a number of "entities" without worrying that their names clash with other names in our program.
- They give us a name to refer to what we have defined.

What we lack so far is something to organize classes, functions, data, and types into an identifiable and named part of a program without defining a type. The language mechanism for such grouping of declarations is a *namespace*. For example, we might like to provide a graphics library with classes called Color, Shape, Line, Function, and Text (see Chapter 11):

CC

```
namespace Graph_lib {
        struct Color { /* ... */ };
        struct Shape { /* ... */ };
        struct Line : Shape { /* ... */ };
        struct Function : Shape { /* ... */ };
        struct Text : Shape { /* ... */ };
        // ...
        int gui_main() { /* ... */ }
}
```

Most likely somebody else in the world has used those names, but now that doesn't matter. You might define something called Text, but our Text doesn't interfere. Graph_lib::Text is one of our classes and your Text is not. We have a problem only if you have a class or a namespace called Graph_lib with Text as its member. Graph_lib is a slightly ugly name; we chose it because the "pretty and obvious" name Graphics had a greater chance of already being used somewhere.

Let's say that your Text was part of a text manipulation library. The same logic that made us put our graphics facilities into namespace Graph_lib should make you put your text manipulation facilities into a namespace called something like TextLib:

```
namespace TextLib {
        class Text { /* ... */ };
        class Glyph { /* ... */ };
        class Line { /* ... */ };
        // ...
}
```

Had we both used the global namespace, we could have been in real trouble. Someone trying to use both of our libraries would have had really bad name clashes for Text and Line. Worse, if we both had users for our libraries, we would not have been able to change our names, such as Line and Text, to avoid clashes. We avoided that problem by using namespaces; that is, our Text is

Graph_lib::Text and yours is TextLib::Text. A name composed of a namespace name (or a class name) and a member name combined by :: is called a *fully qualified name*.

### 7.6.1 Using-declarations and using-directives

Writing fully qualified names can be tedious. For example, the facilities of the C++ standard library are defined in namespace std and can be used like this:

```
import std;          // get the ISO C++ standard library

int main()
{
    std::string name;
    std::cout << "Please enter your first name\n";
    std::cin >> name;
    std::cout << "Hello, " << name << '\n';
}
```

Having seen the standard-library string and cout thousands of times, we don't really want to have to refer to them by their "proper" fully qualified names std::string and std::cout all the time. A solution is to say that "by string, I mean std::string," "by cout, I mean std::cout," etc.:

```
using std::string;          // from here on, string means std::string
using std::cout;            // from here on, cout means std::cout
// ...
```

That construct is called a using declaration; it is the programming equivalent to using plain "Greg" to refer to Greg Hansen, when there are no other Gregs in the room.

Sometimes, we prefer an even stronger "shorthand" for the use of names from a namespace: "If you don't find a declaration for a name in this scope, look in std." The way to say that is to use a using directive:

```
using namespace std;          // make names from std directly accessible
```

So we get this common style:

```
import std;                   // get the ISO C++ standard library
using namespace std;          // make names from std directly accessible

int main()
{
    string name;
    cout << "Please enter your first name\n";
    cin >> name;
    cout << "Hello, " << name << '\n';
}
```

The cin is std::cin, the string is std::string, etc. As long as you use PPP_support, you don't need to worry about standard headers and the std namespace.

XX It is usually a good idea to avoid using directives for any namespace except for a namespace, such as std, that's extremely well known in an application area. The problem with overuse of using

directives is that you lose track of which names come from where, so that you again start to get name clashes. Explicit qualification with namespace names and **using** declarations doesn't suffer from that problem. So, putting a **using** directive in a header file (so that users can't avoid it) is a very bad habit. However, to simplify our initial code we did place a **using** directive for **std** in **PPP_support**. That allows us to write

```
import PPP_import;

int main()
{
        string name;
        cout << "Please enter your first name\n";
        cin >> name;
        cout << "Hello, " << name << '\n';
}
```

## 7.7 Modules and headers

How do we manage our declarations and definitions? After all, they have to be consistent, and in real-world programs there can be tens of thousands of declarations; programs with hundreds of thousands of declarations are not rare. Typically, when we write a program, most of the definitions we use are not written by us. For example, the implementations of **cout** and **sqrt()** were written by someone else many years ago. We just use them.

The keys to managing declarations of facilities defined "elsewhere" in C++ are the module and the header. **CC**

- *The header*: an old and established mechanism for composing programs out of files.
- *The module*: a modern language mechanism for directly expressing modularity.

The module is by far the superior mechanism for ensuring modularity and thereby speeding up compilation. However, header files have been used for more than 50 years and there are billions of lines of code using them, so we must know how to use them well.

### 7.7.1 Modules

Imagine that we wanted to package the **Token_stream** abstraction as a separate facility that people could import as a whole, classes, functions, and all, into their program. We could enable that by defining a **module** called **Tokenstream**:

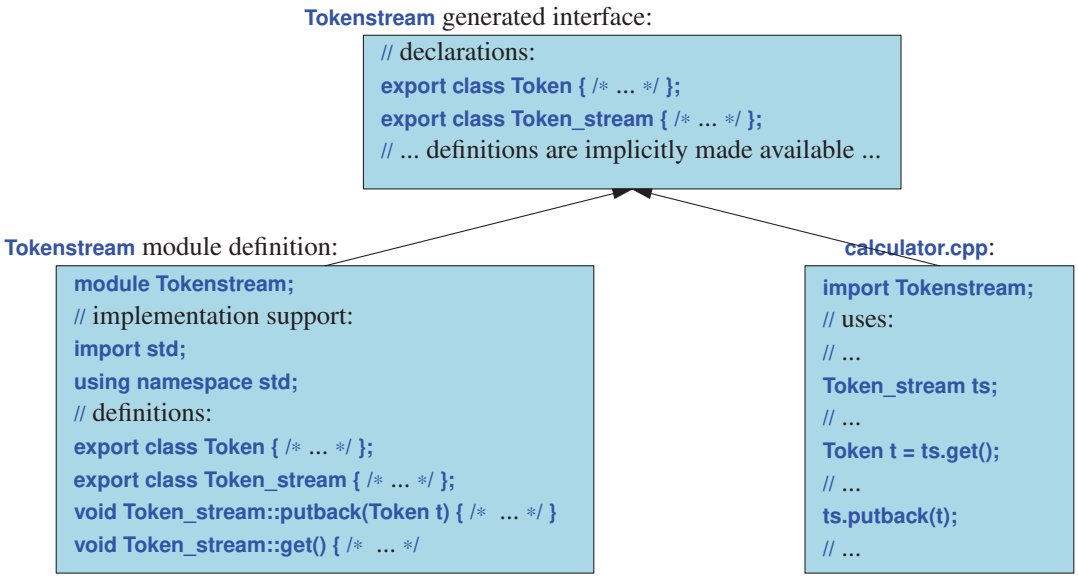
```
module Tokenstream;     // we are defining a module called "Tokenstream"

import std;             // implementation "details"
using namespace std;    // implicitly accessing std - only within Tokenstream
```

```
export class Token {
public:
        char kind;              // what kind of token
        double value;           // for numbers: a value
        Token(char k) :kind{k}, value{0.0} {}           // construct from one value
        Token(char k, double v) :kind{k}, value{v} {}   // construct from two values
};

export class Token_stream {
public:
        Token get();            // get a Token (get() is defined in §5.8.2)
        void putback(Token t);  // put a Token back
private:
        bool full = false;      // is there a Token in the buffer?
        Token buffer;           // putback() saves its token here
};

void Token_stream::putback(Token t)
{
        if (full)
                error("Token_stream::putpack() into a full buffer");
        buffer = t;             // copy t to buffer
        full = true;            // buffer is now full
}

Token Token_stream::get()
{
        if (full) {             // do we already have a Token ready?
                full = false;   // remove Token from buffer
                return buffer;
         }

        // ... use iostream and create a Token ...
}
```

This is of course a very tiny module. Modules like **std** and **PPP_support** offer far more significant benefits, but **Tokenstream** serves as a manageable example.

The definitions marked **export** are made available to users that **import** the module. A module can itself **import** the modules it needs, as is done here for **std**. Importantly, only **exported** declarations are made available to uses, so in this case the user of **Tokenstream** is not implicitly burdened by all of the standard library from the imported module **std**.

We can represent a use of **Tokenstream** graphically:




Unfortunately, there is no standardized suffix for module definition (Microsoft uses **.ixx**, GCC **.cxx**, and Clang **.cppm**). To compile and use modules, you need to know how your specific C++ implementation handles that; **cppreference.com** and **www.stroustrup.com/programming.html** may offer some help (§0.4.1).

The generated module interface is not meant to be seen by programmers, just to be **import**ed.

Modules offer many benefits, including fast compilation – much faster compilation than alternatives – and better isolation of concerns. That is, "implementation details" such as the use of the **iostream** library within **Tokenstream** is not visible to **import**ing code. This has important implications, such as that modules can be **import**ed in any order:

```
import m1;
import m2;
```

means the same as

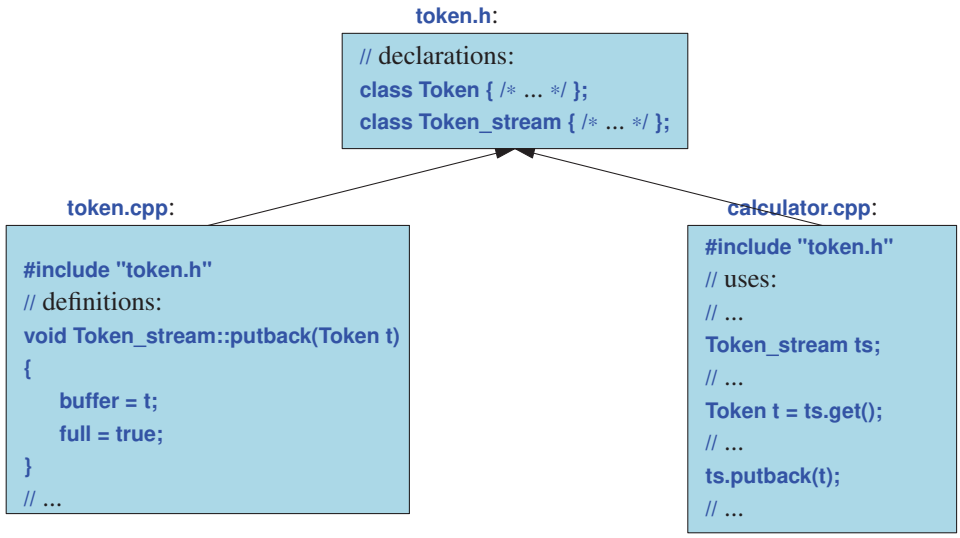
```
import m2;
import m1;
```

That is a great help to both compilers and human readers.

## 7.7.2 Header files

At the time of writing, modules are still rather new in C++. Before that, for 50 years, modularity was "simulated" through file manipulation using the notion of a *header file*. Given that billions of lines of code use header files and millions of programmers are familiar with them, they will be in use for many more years.

Basically, a *header* is a collection of declarations, typically defined in a file, so a header is also called a *header file*. Such headers are then **#include**d in our source files. For example, we might decide to improve the organization of the source code for our calculator (Chapter 5 and Chapter 6) by separating out the token management. We could define a header file **token.h** containing declarations needed to use **Token** and **Token_stream**:




The declarations of **Token** and **Token_stream** are in the header **token.h**. Their definitions are in **token.cpp**. The **.h** suffix is the most common for C++ headers, and the **.cpp** suffix is the most common for C++ source files. Actually, the C++ language doesn't care about file suffixes, but some compilers and most program development environments insist, so please use this convention for your source code.

In principle, **#include "file.h"** simply copies the declarations from **file.h** into your file at the point of the **#include**. For example, we could write a header **f.h**:

```
// f.h
int f(int);
```

and include it in our file **user.cpp**:

```
// user.cpp
#include "f.h"

int g(int i)
{
    return f(i);
}
```

When compiling **user.cpp** the compiler would do the **#include** and compile

```
int f(int);

int g(int i)
{
        return f(i);
}
```

Since #includes logically happen before anything else a compiler does, handling #includes is part of what is called *preprocessing* (PPP2.§27.8).

To ease consistency checking, we #include a header both in source files that use its declarations and in source files that provide definitions for those declarations. That way, the compiler catches errors as soon as possible. For example, imagine that the implementer of Token_stream::putback() made mistakes: AA

```
Token Token_stream::putback(Token t)
{
        buffer.push_back(t);
        return t;
}
```

This looks innocent enough. Fortunately, the compiler catches the mistakes because it sees the (#included) declaration of Token_stream::putback(). Comparing that declaration with our definition, the compiler finds that putback() should not return a Token and that buffer is a Token, rather than a vector<Token>, so we can't use push_back(). Such mistakes occur when we work on our code to improve it, but don't quite get a change consistent throughout a program.

Similarly, consider these mistakes:

```
Token t = ts.gett();    // error: no member gett
// ...
ts.putback();           // error: argument missing
```

The compiler would immediately give errors; the header token.h gives it all the information it needs for checking.

A header will typically be included in many source files. That means that a header should only contain declarations that can be duplicated in several files (such as function declarations, class definitions, and definitions of numeric constants). AA

In §10.8.1, we offer a slightly more realistic example of header use.

## Drill

[1] Write three functions swap_v(int,int), swap_r(int&,int&), and swap_cr(const int&, const int&). Each should have the body

```
{ int temp; temp = a, a=b; b=temp; }
```

where a and b are the names of the arguments.

Try calling each swap like this

```
int x = 7;
int y = 9;
swap_?(x,y);          // replace ? by v, r, or cr
swap_?(7,9);
const int cx = 7;
const int cy = 9;
swap_?(cx,cy);
swap_?(7.7,9.9);
double dx = 7.7;
double dy = 9.9;
swap_?(dx,dy);
swap_?(7.7,9.9);
```

Which functions and calls compiled, and why? After each swap that compiled, print the value of the arguments after the call to see if they were actually swapped. If you are surprised by a result, consult §7.5.

[2] Write a program using a single file containing three namespaces **X**, **Y**, and **Z** so that the following **main()** works correctly:

```
int main()
{
    X::var = 7;
    X::print();             // print X's var
    using namespace Y;
    var = 9;
    print();                // print Y's var
    {
        using Z::var;
        using Z::print;
        var = 11;
        print();            // print Z's var
    }
    print();                // print Y's var
    X::print(); // print X's var
}
```

Each namespace needs to define a variable called **var** and a function called **print()** that outputs the appropriate **var** using **cout**.

[3] Create a module **foo** with the suffix appropriate to your system:

```
int foo = 0;
export void print_foo() { ... };
export void set_foo(int x} { foo = x; }
export int get_foo() { return x; }
```

Add what it takes to get the **...** part to print **foo**. Write file **use.cpp** that **import**s **foo** and tests it. Get the resulting program to compile and run.

[4] Create a header file: **foo.h**:

```
extern int foo;
void print_foo();
void print(int);
```

Write a file **foo.cpp** that implements the functions declared in **foo.h**. Write file **use.cpp** that **#include**s **foo.h** and tests it. Get the resulting program to compile and run.

## Review

[1] What is the difference between a declaration and a definition?
[2] How do we syntactically distinguish between a function declaration and a function definition?
[3] How do we syntactically distinguish between a variable declaration and a variable definition?
[4] Why can't you use the functions in the calculator program from Chapter 5 without declaring one or more of them first?
[5] Is **int a;** a definition or just a declaration?
[6] Why is it a good idea to initialize variables as they are declared?
[7] What can a function declaration consist of?
[8] What is the *suffix return type* notation, and why might you use it?
[9] What good does indentation do?
[10] What is the scope of a declaration?
[11] What kinds of scope are there? Give an example of each.
[12] What is the difference between a class scope and local scope?
[13] Why should a programmer minimize the number of global variables?
[14] What is the difference between pass-by-value and pass-by-reference?
[15] What is the difference between pass-by-reference and pass-by-**const**-reference?
[16] What is a **swap()**?
[17] Would you ever define a function with a **vector<double>** as a by-value parameter?
[18] Give an example of undefined order of evaluation. Why can undefined order of evaluation be a problem?
[19] What do **x&&y** and **x||y**, respectively, mean?
[20] Which of the following is standard-conforming C++: functions within functions, functions within classes, classes within classes, classes within functions?
[21] What goes into an activation record?
[22] What is a call stack and why do we need one?
[23] What is the purpose of a namespace?
[24] How does a namespace differ from a class?
[25] What is a **using** declaration?
[26] Why should you avoid **using** directives in a header?
[27] What is namespace **std**?

## Terms

| | | | |
|---|---|---|---|
| activation record | function | pass-by-reference | argument |
| function definition | pass-by-value | argument passing | global scope |
| recursion | call stack | header file | **return** |
| class scope | initializer | return value | **const** |
| local scope | scope | **constexpr** | **namespace** |
| statement scope | declaration | namespace scope | technicalities |
| definition | nested block | undeclared identifier | **extern** |
| parameter | **using** declaration | forward declaration | pass-by-**const**-reference |
| **using** directive | **auto** | **–>** | suffix return type |

## Exercises

[1] Modify the calculator program from Chapter 6 to make the input stream an explicit parameter (as shown in §7.4.8), rather than simply using **cin**. Also give the **Token_stream** constructor (§6.8.2) an **istream&** parameter so that when we figure out how to make our own **istream**s (e.g., attached to files), we can use the calculator for those. Hint: Don't try to copy an **istream**.

[2] Write a function **print()** that prints a **vector** of **int**s to **cout**. Give it two arguments: a **string** for "labeling" the output and a **vector**.

[3] Create a **vector** of Fibonacci numbers and print them using the function from exercise 2. To create the **vector**, write a function, **fibonacci(x,y,v,n)**, where integers **x** and **y** are **int**s, **v** is an empty **vector<int>**, and **n** is the number of elements to put into **v**; **v[0]** will be **x** and **v[1]** will be **y**. A Fibonacci number is one that is part of a sequence where each element is the sum of the two previous ones. For example, starting with 1 and 2, we get 1, 2, 3, 5, 8, 13, 21, ... . Your **fibonacci()** function should make such a sequence starting with its **x** and **y** arguments.

[4] An **int** can hold integers only up to a maximum number. Find an approximation of that maximum number by using **fibonacci()**.

[5] Write two functions that reverse the order of elements in a **vector<int>**. For example, 1, 3, 5, 7, 9 becomes 9, 7, 5, 3, 1. The first reverse function should produce a new **vector** with the reversed sequence, leaving its original **vector** unchanged. The other reverse function should reverse the elements of its **vector** without using any other **vector**s (hint: **swap**).

[6] Write versions of the functions from exercise 5, but with a **vector<string>**.

[7] Read five names into a **vector<string> name**, then prompt the user for the ages of the people named and store the ages in a **vector<double> age**. Then print out the five (**name[i],age[i]**) pairs. Sort the names (**sort(name.begin(),name.end())**) and print out the (**name[i],age[i]**) pairs. The tricky part here is to get the **age vector** in the correct order to match the sorted **name vector**. Hint: Before sorting **name**, take a copy and use that to make a copy of **age** in the right order after sorting **name**.

[8] Do the previous exercise but allow an arbitrary number of names.

[9] Write a function that given two **vector<double>**s **price** and **weight** computes a value (an "index") that is the sum of all **price[i]*weight[i]**. Make sure to have **weight.size()==price.size()**.

[10] Write a function **maxv()** that returns the largest element of a **vector** argument.

[11] Write a function that finds the smallest and the largest element of a **vector** argument and also computes the mean and the median. Do not use global variables. Either return a **struct** containing the results or pass them back through reference arguments. Which of the two ways of returning several result values do you prefer and why?

[12] Improve **print_until_s()** from §7.4.2. Test it. What makes a good set of test cases? Give reasons. Then, write a **print_until_ss()** that prints until it sees a second occurrence of its **quit** argument.

[13] Write a function that takes a **vector<string>** argument and returns a **vector<int>** containing the number of characters in each **string**. Also find the longest and the shortest **string** and the lexicographically first and last **string**. How many separate functions would you use for these tasks? Why?

[14] Can we declare a non-reference function argument **const** (e.g., **void f(const int);**)? What might that mean? Why might we want to do that? Why don't people do that often? Try it; write a couple of small programs to see what works.

## Postscript

We could have put much of this chapter (and much of the next) into an appendix. However, you'll need most of the facilities described here in the rest of this book. You'll also encounter most of the problems that these facilities were invented to help solve very soon. Most simple programming projects that you might undertake will require you to solve such problems. So, to save time and minimize confusion, a somewhat systematic approach is called for, rather than a series of "random" visits to manuals and appendices.