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';
}
```


Terms

code layout	maintenance	scaffolding	commenting
recovery	symbolic constant	error handling	revision history
testing	feature creep	magic constant	

Exercises

[1] Allow underscores in the calculator's variable names.

[2] Provide an assignment operator, **=**, so that you can change the value of a variable after you have introduced it using **let**. Discuss why that can be useful and how it can be a source of problems.

[3] Provide named constants that you really can't change the value of. Hint: You have to add a member to **Variable** that distinguishes between constants and variables and check for it in **set_value()**. If you want to let the user define constants (rather than just having **pi** and **e** defined as constants), you'll have to add a notation to let the user express that, for example, **const pi = 3.14;**.

[4] The **get_value()**, **set_value()**, **is_declared()**, and **define_name()** functions all operate on the variable **var_table**. Define a class called **Symbol_table** with a member **var_table** of type **vector<Variable>** and member functions **get()**, **set()**, **is_declared()**, and **declare()**. Rewrite the calculator to use a variable of type **Symbol_table**.

[5] Modify **Token_stream::get()** to return **Token(print)** when it sees a newline. This implies looking for whitespace characters and treating newline ('**\n**') specially. You might find the standard-library function **isspace(ch)**, which returns **true** if **ch** is a whitespace character, useful.

[6] Part of what every program should do is to provide some way of helping its user. Have the calculator print out some instructions for how to use the calculator if the user presses the H key (both upper- and lowercase).

[7] Change the **q** and **h** commands to be **quit** and **help**, respectively.

[8] The grammar in §6.6.4 is incomplete (we did warn you against overreliance on comments); it does not define sequences of statements, such as **4+4; 5–6;**, and it does not incorporate the grammar changes outlined in §6.8. Fix that grammar. Also add whatever you feel is needed for that comment as the first comment of the calculator program and its overall comment.

[9] Suggest three improvements (not mentioned in this chapter) to the calculator. Implement one of them.

[10] Modify the calculator to operate on **int**s (only); give errors for overflow and underflow. Hint: Use **narrow** (§6.5).

[11] Revisit two programs you wrote for the exercises in §3 or Chapter 4. Clean up that code according to the rules outlined in this chapter. See if you find any bugs in the process.

[12] Modify the calculator to accept input from any **istream**.

"silly" inputs.

[5] Do the testing and fix any bugs that you missed when you commented.

[6] Add a predefined name `k` meaning `1000`.

[7] Give the user a square root function `sqrt()`, for example, `sqrt(2+6.7)`. Naturally, the value of `sqrt(x)` is the square root of `x`; for example, `sqrt(9)` is `3`. Use the standard-library `sqrt()` function to implement that calculator `sqrt()`. Remember to update the comments, including the grammar.

[8] Catch attempts to take the square root of a negative number and print an appropriate error message.

[9] Allow the user to use `pow(x,i)` to mean "Multiply `x` with itself `i` times"; for example, `pow(2.5,3)` is `2.5*2.5*2.5`. Require `i` to be an integer using the technique we used for `%` (§6.5).

[10] Change the "declaration keyword" from `let` to `#`.

[11] Change the "quit keyword" from `quit` to `exit`. That will involve defining a string for `quit` just as we did for `let` in §6.8.2.

Review

[1] What is the purpose of working on the program after the first version works? Give a list of reasons.

[2] Why does `1+2; q` typed into the calculator not quit after it receives an error?

[3] Why did we choose to make a constant character called `number`?

[4] We split `main()` into two separate functions. What does the new function do and why did we split `main()`?

[5] Why do we split code into multiple functions? State principles.

[6] What is the purpose of commenting?

[7] What is the use of symbolic constants?

[8] Why do we care about code layout?

[9] How do we handle `%` (remainder) of floating-point numbers?

[10] What does `is_declared()` do and how does it work?

[11] The input representation for `let` is more than one character. How is it accepted as a single token in the modified code?

[12] What are the rules for what names can and cannot be in a calculator program?

[13] Why is it a good idea to build a program incrementally?

[14] When do you start to test?

[15] When do you retest?

[16] How do you decide what should be a separate function?

[17] How do you choose names for variables and functions? List possible reasons.

[18] What should be in comments and what should not?

[19] When do we consider a program finished?

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
```

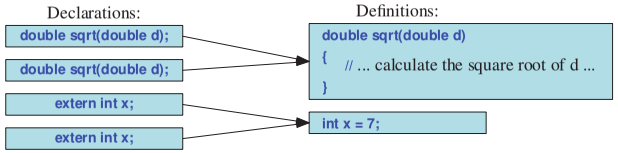
In contrast, a declaration that isn't also a definition simply tells how you can use a name; it is just an interface and doesn't allocate memory or specify a function body. Consequently, you can declare something as often as you like as long as you do so consistently:

```
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:



CC

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.

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();
    // ...
}
```



```
int main()
try {
    // predefine names:
    define_name("pi",3.1415926535);
    define_name("e",2.7182818284);

    calculate();
    return 0;
}
catch (exception& e) {
    cerr << e.what() << '\n';
    return 1;
}
catch (...) {
    cerr << "exception \n";
    return 2;
}
```

6.8.4 Are we there yet?

Not really. We have made so many changes that we need to test everything again, clean up the code, and review the comments. Also, we could define more useful operations. For example, we "forgot" to provide an assignment operator (see exercise 2), and if we have an assignment we might want to distinguish between variables and constants (exercise 3).

Initially, we backed off from having named variables in our calculator. Looking back over the code that implements them, we may have two possible reactions:

[1] Implementing variables wasn't all that bad; it took only about three dozen lines of code.

[2] Implementing variables was a major extension. It touched just about every function and added a completely new concept to the calculator. It increased the size of the calculator by 45% and we haven't even implemented assignment!

AA

In the context of a first program of significant complexity, the second reaction is the correct one. More generally, it's the right reaction to any suggestion that adds something like 50% to a program in terms of both size and complexity. When that has to be done, it is more like writing a new program based on a previous one than anything else, and it should be treated that way. In particular, if you can build a program in stages as we did with the calculator, and test it at each stage, you are far better off doing so than trying to do the whole program all at once.

Drill

[1] Starting from the file `calculator08buggy.cpp`, get the calculator to compile.

[2] Go through the entire program and add appropriate comments.

[3] As you commented, you found errors (deviously inserted especially for you to find). Fix them; they are not in the text of the book.

[4] Testing: prepare a set of inputs and use them to test the calculator. Is your list pretty complete? What should you look for? Include negative values, 0, very small, very large, and

What must we do instead? First we must specify precisely what we want a name to be, and then we must modify `get()` to do that. Here is a workable specification of a name: a sequence of letters and digits starting with a letter. Given this definition,

```
a
ab
a1
Z12
asdsddsfdfdasfdsa434RTHTD12345dfdsa8fsd888fadsf
```

are names and

```
1a
as_s
#
as*
a car
```

are not. Except for leaving out the underscore, this is C++'s rule. We can implement that in the default case of `get()`:

```
default:
    if (isalpha(ch)) {
        string s;
        s += ch;
        while (cin.get(ch) && (isalpha(ch) || isdigit(ch)))
            s+=ch;
        cin.putback(ch);
        if (s == declkey)
            return Token{let};    // declaration keyword
        return Token{name,s};
    }
    error("Bad token");
```

Instead of reading directly into the `string s`, we read characters and put those into `s` as long as they are letters or digits. The `s+=ch` statement adds (appends) the character `ch` to the end of the string `s`. The curious statement

```
while (cin.get(ch) && (isalpha(ch) || isdigit(ch)))
    s+=ch;
```

reads a character into `ch` (using `cin`'s member function `get()`) and checks if it is a letter or a digit. If so, it adds `ch` to `s` and reads again. The `get()` member function works just like `>>` except that it doesn't by default skip whitespace.

6.8.3 Predefined names

Now that we have names, we can easily predefine a few common ones. For example, if we imagine that our calculator will be used for scientific calculations, we'd want `pi` and `e`. Where in the code would we define those? In `main()` before the call of `calculate()` or in `calculate()` before the loop. We'll put them in `main()` because those definitions really aren't part of any calculation:


```
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 strings and vectors. For example:

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


Note first of all the call isalpha(ch). This call answers the question "Is ch a letter?"; isalpha() is part of the standard library. For more character classification functions, see §9.10.4. The logic for recognizing names is the same as that for recognizing numbers: find a first character of the right kind (here, a letter), then put it back using putback() and read in the whole name using >>.

Unfortunately, this doesn't compile; we have no Token that can hold a string, so the compiler rejects Token{name,s}. To handle that, we must modify the definition of Token to hold either a string or a double, and handle three forms of initializers, such as

- Just a kind; for example, Token{'*'}
- A kind and a number; for example, Token{number,4.321}
- A kind and a name; for example, Token{name,"pi"}

We handle that by introducing three initialization functions, known as constructors because they construct objects:

```
class Token {
public:
    char kind;
    double value;
    string name;
    Token() :kind{0} {}                                  // default constructor
    Token(char ch) :kind{ch} { }                         // initialize kind with ch
    Token(char ch, double val) :kind{ch}, value{val} { } // initialize kind and value
    Token(char ch, string n) :kind{ch}, name{n} { }      // initialize kind and name
};
```

Constructors add an important degree of control and flexibility to initialization. We will examine constructors in detail in §8.4.2 and §8.7.

We chose 'L' as the representation of the let token and the string let as our keyword. Obviously, it would be trivial to change that keyword to double, var, #, or whatever by changing the string declkey that we compare s to.

Now we try the program again. If you type this, you'll see that it all works:

```
let x = 3.4;
let y = 2;
x + y * 2;
```

However, this doesn't work:

```
let x = 3.4;
let y = 2;
x+y*2;
```

What's the difference between those two examples? Have a look to see what happens.

The problem is that we were sloppy with our definition of Name. We even "forgot" to define our Name production in the grammar (§6.8.1). What characters can be part of a name? Letters? Certainly. Digits? Certainly, as long as they are not the starting character. Underscores? Eh? The + character? Well? Eh? Look at the code again. After the initial letter we read into a string using >>. That accepts every character until it sees whitespace. So, for example, x+y*2; is a single name – even the trailing semicolon is read as part of the name. That's unintended and unacceptable.

```
        double d = expression();
        define_name(t.name,d);
        return d;
    }
```

Note that we returned the value stored in the new variable. That's useful when the initializing expression is nontrivial. For example:

```
let v = d/(t2–t1);
```

This declaration will define `v` and also print its value. Additionally, printing the value of a declared variable simplifies the code in `calculate()` because every `statement()` returns a value. General rules tend to keep code simple, whereas special cases tend to lead to complications.

This mechanism for keeping track of `Variable`s is what is often called a *symbol table* and could be radically simplified by the use of a standard-library `map`; see PPP2.§21.6.1.

6.8.2 Introducing names

This is all very good, but unfortunately, it doesn't quite work. By now, that shouldn't come as a surprise. Our first cut never – well, hardly ever – works. Here, we haven't even finished the program – it doesn't yet compile. We have no '`=`' token, but that's easily handled by adding a case to `Token_stream::get()` (§6.6.3). But how do we represent `let` and `name` as tokens? Obviously, we need to modify `get()` to recognize these tokens. How? Here is one way:

```
const char name = 'a';                  // name token
const char let = 'L';                   // declaration token
const string declkey = "let";           // declaration keyword

Token Token_stream::get()
{
    if (full) {
        full = false;
        return buffer;
    }
    char ch;
    cin >> ch;
    switch (ch) {
        // ... as before ...
    default:
        if (isalpha(ch)) {
            cin.putback(ch);
            string s;
            cin >> s;
            if (s == declkey)
                return Token{let};    // declaration keyword
            return Token{name,s};
        }
        error("Bad token");
    }
}
```


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()`.

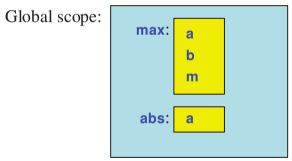
There are several kinds of scopes that we use to control where our names can be used:

- 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.

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
```

AA


```
let v1 = 7;
let v2 = 8;
```

There are logically two parts to defining a Variable with the name var with the value val:

[1] Check whether there already is a Variable called var in var_table.

[2] Add (var,val) to var_table.

We have no use for uninitialized variables. We defined the functions is_declared() and define_name() to represent those two logically separate operations:

```
bool is_declared(string var)
        // is var already in var_table?
{
        for (const Variable& v : var_table)
                if (v.name == var)
                        return true;
        return false;
}

double define_name(string var, double val)
        // add {var,val} to var_table
{
        if (is_declared(var))
                error(var," declared twice");
        var_table.push_back(Variable{var,val});
        return val;
}
```

Adding a new Variable to a vector<Variable> is easy; that's what vector's push_back() member function does:

```
var_table.push_back(Variable{var,val});
```

The Variable{var,val} makes the appropriate Variable and push_back(), then adds that Variable to the end of var_table. Given that, and assuming that we can handle let and name tokens, declaration() is straightforward to write:

```
double declaration()
        // assume we have seen "let"
        // handle: name = expression
        // declare a variable called "name" with the initial value "expression"
{
        Token t = ts.get();
        if (t.kind != name)
                error ("name expected in declaration");

        Token t2 = ts.get();
        if (t2.kind != '=')
                error("= missing in declaration of ", t.name);
```

```
Declaration:
        "let" Name "=" Expression
```

Calculation is the new top production (rule) of the grammar. It expresses the loop (in calculate()) that allows us to do several calculations in a run of the calculator program. It relies on the Statement production to handle expressions and declarations. We can handle a statement like this:

```
double statement()
{
        Token t = ts.get();
        switch (t.kind) {
        case let:
                return declaration();
        default:
                ts.putback(t);
                return expression();
        }
}
```

We can now use statement() instead of expression() in calculate():

```
void calculate()
{
        while (cin)
                try {
                        cout << prompt;
                        Token t = ts.get();
                        while (t.kind == print)          // first discard all "prints"
                                t=ts.get();
                        if (t.kind == quit)              // quit
                                return;
                        ts.putback(t);
                        cout << result << statement() << '\n';
                }
                catch (exception& e) {
                        cerr << e.what() << '\n';        // write error message
                        clean_up_mess();
                }
}
```

We now have to define declaration(). What should it do? It should make sure that what comes after a let is a Name followed by a = followed by an Expression. That's what our grammar says. What should it do with the name? We should add a Variable with that name string and the value of the expression to our vector<Variable> called var_table. Once that's done we can retrieve the value using get_value() and change it using set_value(). However, before writing this, we have to decide what should happen if we define a variable twice. For example:

```
let v1 = 7;
let v1 = 8;
```

We chose to consider such a redefinition an error. Typically, it is simply a spelling mistake. Instead of what we wrote, we probably meant


```
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 {
                // ...
                {
                        // ...
                }
                // ...
        }
}
```



```
void set_value(string s, double d)
        // set the Variable named s to d
{
        for (Variable& v : var_table)
                if (v.name == s) {
                        v.value = d;
                        return;
                }
        error("trying to write undefined variable ", s);
}
```

Did you notice the `&` in the last two functions? It means that `v` is a reference to a `Variable` in the `var_table`, rather than a copy of one. For `set_value`, that's essential because giving a new value to a copy, to something not in the table, would be useless. References are essential for many important programming techniques. They will be presented in detail in §7.4 and§16.2.

We can now read and write "variables" represented as `Variable`s in `var_table`. How do we get a new `Variable` into `var_table`? What does a user of our calculator have to write to define a new variable and later to get its value? We could consider C++'s notation

```
double var = 7.2;
```

That would work, but all variables in this calculator hold `double` values, so saying "double" would be redundant. Could we make do without an explicit "declaration indicator"? For example:

```
var = 7.2;
```

Possibly, but then we would be unable to tell the difference between the declaration of a new variable and a spelling mistake:

```
var1 = 7.2;          // define a new variable called var1
var1 = 3.2;          // define a new variable called var2
```

Oops! Clearly, we meant `var2 = 3.2;` but we didn't say so (except in the comment). We could live with this, but we'll follow the tradition in languages, such as C++, that distinguishes declarations (with initializations) from assignments. We could use `double`, but for a calculator we'd like something short, so – drawing on another old tradition – we choose the keyword `let`:

```
let var = 7.2;
```

The grammar would be

```
Calculation:
        Statement
        Print
        Quit
        Calculation Statement

Statement:
        Declaration
        Expression
```

6.8 Variables

Having worked on style and error handling, we can return to looking for improvements in the calculator functionality. We now have a program that works quite well; how can we improve it? The first wish list for the calculator included variables. Having variables gives us better ways of expressing longer calculations. Similarly, for scientific calculations, we'd like built-in named values, such as `pi` and `e`, just as we have on scientific calculators.

Adding variables and constants is a major extension to the calculator. It will touch most parts of the code. This is the kind of extension that we should not embark on without good reason and sufficient time. Here, we add variables and constants because it gives us a chance to look over the code again and try out some more programming techniques.

6.8.1 Variables and definitions

Obviously, the key to both variables and built-in constants is for the calculator program to keep (*name,value*) pairs so that we can access the value given the name. We can define a `Variable` like this:

```
class Variable {
public:
    string name;
    double value;
};
```

We will use the `name` member to identify a `Variable` and the `value` member to store the value corresponding to that `name`.

How can we store `Variable`s so that we can search for a `Variable` with a given `name` string to find its value or to give it a new value? Looking back over the programming tools we have encountered so far, we find only one good answer: a `vector` of `Variable`s:

```
vector<Variable> var_table;
```

We can put as many `Variable`s as we like into the vector `var_table` and search for a given name by looking at the vector elements one after another. We can write a `get_value()` function that looks for a given `name` string and returns its corresponding `value`:

```
double get_value(string s)
    // return the value of the Variable named s
{
    for (const Variable& v : var_table)
        if (v.name == s)
            return v.value;
    error("trying to read undefined variable ", s);
}
```

The code really is quite simple: go through every `Variable` in `var_table` (starting with the first element and continuing until the last) and see if its `name` matches the argument string `s`. If that is the case, return its `value`.

Similarly, we can define a `set_value()` function to give a `Variable` a new `value`:

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:

discovered the hard way – throw an exception. So we need a new operation. The obvious place to put that is in `Token_stream`:

```
class Token_stream {
public:
    Token get();                  // get a Token
    void putback(Token t);        // put a Token back
    void ignore(char c);          // discard characters up to and including a c
private:
    bool full = false;            // is there a Token in the buffer?
    Token buffer = 0;             // putback() saves its token here
};
```

This `ignore()` function needs to be a member of `Token_stream` because it needs to look at `Token_stream`'s buffer. We chose to make "the thing to look for" an argument to `ignore()` – after all, the `Token_stream` doesn't have to know what the calculator considers a good character to use for error recovery. We decided that argument should be a character because we don't want to risk composing `Token`s – we saw what happened when we tried that. So we get

```
void Token_stream::ignore(char c)
    // c represents the kind of Token
{
    if (full && c==buffer.kind) {          // first look in buffer
        full = false;
        return;
    }
    full = false;

    // now search input:
    char ch = 0;
    while (cin>>ch)
        if (ch==c)
            return;
}
```

This code first looks at the buffer. If there is a `c` there, we are finished after discarding that `c`; otherwise, we need to read characters from `cin` until we find a `c`.

We can now write `clean_up_mess()` rather simply:

```
void clean_up_mess()
{
    ts.ignore(print);
}
```

Dealing with errors is always tricky. It requires much experimentation and testing because it is extremely hard to imagine all the errors that can occur. Trying to make a program foolproof is always a very technical activity; amateurs typically don't care. Quality error handling is one mark of a professional.