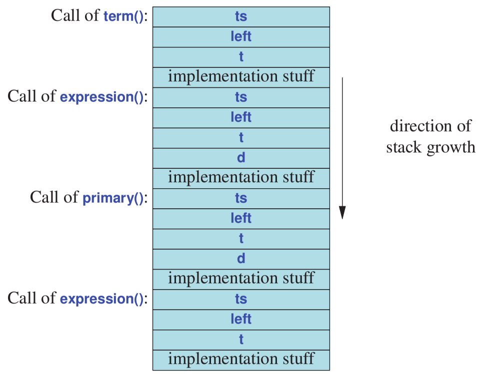
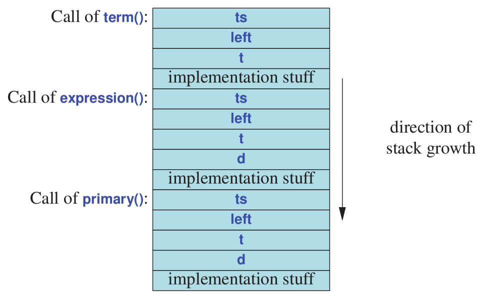
This is starting to get a bit repetitive, but now primary() calls expression():




CC

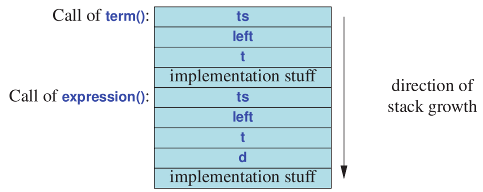
So this call of expression() gets its own activation record, different from the first call of expression(). That's good or else we'd be in a terrible mess, since left and t will be different in the two calls. A function that directly or (as here) indirectly calls itself is called *recursive*.

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:

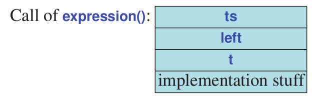
```
auto expression(Token_stream& ts) –> double;     // call expression(ts) and get a double
```

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>;
```

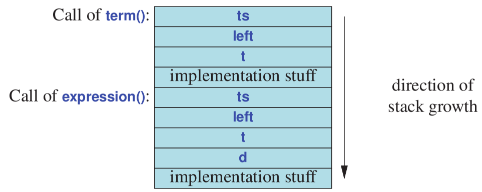
as opposed to



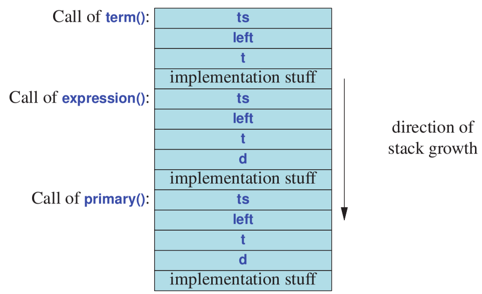



The "implementation stuff" varies from implementation to implementation, but that's basically the information that the function needs to return to its caller and to return a value to its caller. Each function has its own detailed layout of its activation record. Note that from the implementation's point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



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.

CC 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:



```
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*.



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

```
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)
}
```



### 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.

CC

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):

```
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



```
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:

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:



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.

CC

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

- *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
```



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 strings. 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 strings, 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:

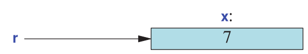
```
void init(vector<double>& v)          // pass-by-reference
{
        for (int i = 0; i<v.size(); ++i)
                v[i] = i;
}

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:

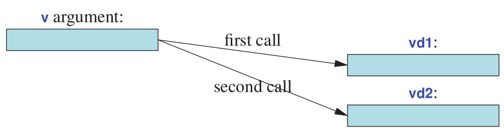


```
void print(const vector<double>& v)          // pass-by-const-reference
{
     cout << "{ ";
     for (int i = 0; i<v.size(); ++i) {
          cout << v[i];
          if (i!=v.size()−1)
               cout << ", ";
     }
     cout << " }\n";
}
```

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)
     // ...
}
```



```
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:

Tokenstream generated interface:

```
// declarations:
export class Token { /* ... */ };
export class Token_stream { /* ... */ };
// ... definitions are implicitly made available ...
```

Tokenstream module definition:

```
module Tokenstream;
// implementation support:
import std;
using namespace std;
// definitions:
export class Token { /* ... */ };
export class Token_stream { /* ... */ };
void Token_stream::putback(Token t) { /*  ... */ }
void Token_stream::get() { /*  ... */
```

calculator.cpp:

```
import Tokenstream;
// uses:
// ...
Token_stream ts;
// ...
Token t = ts.get();
// ...
ts.putback(t);
// ...
```

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.



### 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 **vectors**, 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_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;**.

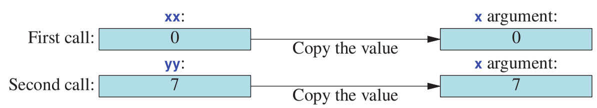
### 7.4.3 Pass-by-value

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
```

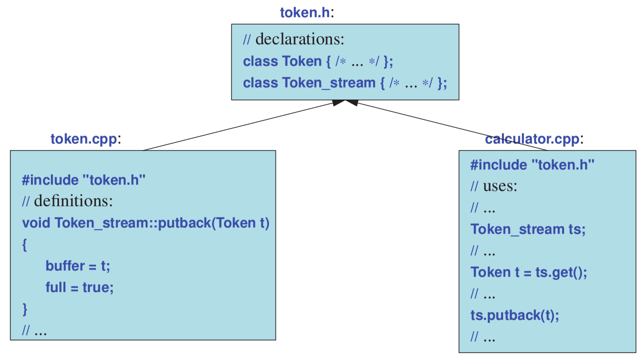
Since a copy is passed, the **x=x+1** in **f()** does not change the values **xx** and **yy** passed in the two calls. We can illustrate a pass-by-value argument passing like this:




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



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

AA

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:

```
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 (`#include`d) 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.

AA

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).

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 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: