Using those, the actual expressions and results are only a minor part of what appears on the screen, and since expressions and results are what matter, nothing should distract from them. On the other hand, unless we somehow separate what the user types from what the computer outputs, the result can be confusing. During initial debugging, we added = as a result indicator. We would also like a short "prompt" to indicate that the program wants input. The > character is often used as a prompt:

```
> 2+3;
= 5
> 5*7;
= 35
>
```

This looks much better, and we can get it by a minor change to the main loop of **main()**:

```
double val = 0;
while (cin) {
        cout << "> ";            // print prompt
        Token t = ts.get();
        if (t.kind == 'q') break;
        if (t.kind == ';')
                cout << "= " << val << '\n';            // print result
        else
                ts.putback(t);
        val = expression();
}
```

Unfortunately, the result of putting several expressions on a line is still messy:

```
> 2+3; 5*7; 2+9;
= 5
> = 35
> = 11
>
```

The basic problem is that we didn't think of multiple expressions on a line when we started out (at least we pretended not to). What we want is

```
> 2+3; 5*7; 2+9;
= 5
= 35
= 11
>
```

This looks right, but unfortunately there is no really obvious way of achieving it. We first looked at **main()**. Is there a way to write out **>** only if it is not immediately followed by a **=**? We cannot know! We need to write **>** before the **get()**, but we do not know if **get()** actually reads new characters or simply gives us a **Token** from characters that it had already read from the keyboard. In other words, we would have to mess with **Token_stream** to make this final improvement.

For now, we decide that what we have is good enough. If we find that we have to modify **Token_stream**, we'll revisit this decision. However, it is unwise to make major structural changes to gain a minor advantage, and we haven't yet thoroughly tested the calculator.

6.3 Error handling

CC The first thing to do once we have a program that "basically works" is to try to break it; that is, we try to feed it input in the hope of getting it to misbehave. We say "hope" because the challenge here is to find as many errors as possible, so that we can fix them before anybody else finds them. If you go into this exercise with the attitude that "my program works and I don't make errors!" you won't find many bugs and you'll feel bad when you do find one. You'd be playing head games with yourself! The right attitude when testing is "I'll break it! I'm smarter than any program – even my own!" So, we feed the calculator a mix of correct and incorrect expressions. For example:

```
1+2+3+4+5+6+7+8
1–2–3–4
!+2
;;;
(1+3;
(1+);
1*2/3%4+5–6;
();
1+;
+1
1++;
1/0
1/0;
1++2;
–2;
–2;;;;
1234567890123456;
'a';
q
1+q
1+2; q
```

TRY THIS

Feed a few such "problematic" expressions to the calculator and try to figure out in how many ways you can get it to misbehave. Can you get it to crash, that is, to get it past our error handling and give a machine error? We don't think you can. Can you get it to exit without a useful error message? You can.

Technically, this is known as *testing* (§4.7.4). There are people who do this – break programs – for a living. Testing is a very important part of software development and can actually be fun. One big question is: "Can we test the program systematically, so that we find all of the errors?" There is no general answer to this question; that is, there is no answer that holds for all programs. However, you can do rather well for many programs when you approach testing seriously. You try to create test cases systematically, and just in case your strategy for selecting tests isn't complete, you do some "unreasonable" tests, such as

```
Mary had a little lamb
srtvrqtiewcbet7rewaewre–wqcntrretewru754389652743nvcqnwq;
!@#%ˆ&∗()˜:;
```

AA Once, when testing compilers, I got into the habit of feeding email that reported compiler errors straight to the compiler – mail headers, user's explanation, and all. That wasn't "sensible" because "nobody would do that." However, a program ideally catches all errors, not just the sensible ones, and soon that compiler was very resilient against "strange input."

The calculator takes input from the keyboard. That makes testing tedious: each time we make an improvement, we have to type in a lot of test cases (yet again!) to make sure we haven't broken anything. It would be much better if we could store our test cases somewhere and run them with a single command. Some operating systems (notably Unix) make it trivial to get **cin** to read from a file without modifying the program, and similarly to divert the output from **cout** to a file. If that's not convenient, we must modify the program to use a file (see §9).

Now consider:

```
1+2; q
```

and

```
1+2 q
```

We would like both to print the result (**3**) and then exit the program. Curiously enough,

```
1+2 q
```

does that, but the apparently cleaner

```
1+2; q
```

elicits a **Primary expected** error. Where would we look for this error? In **main()** where **;** and **q** are handled, of course. We added those "print" and "quit" commands rather quickly to get the calculator to work (§5.7). Now we are paying for that haste. Consider again:

```
double val = 0;
while (cin) {
      cout << "> ";
      Token t = ts.get();
      if (t.kind == 'q')
            break;
      if (t.kind == ';')
            cout << "= " << val << '\n';
      else
            ts.putback(t);
      val = expression();
}
```

If we find a semicolon, we straightaway proceed to call **expression()** without checking for **q**. The first thing that **expression()** does is to call **term()**, which first calls **primary()**, which finds **q**. The letter **q** isn't a **Primary** so we get our error message. So, we should test for **q** after testing for a semicolon. While we were at it, we felt the need to simplify the logic a bit, so the complete **main()** reads

6.1 Introduction

AA When your program first starts running "reasonably," you're probably about halfway finished. For a large program or a program that could do harm if it misbehaved, you will be nowhere near halfway finished. Once the program "basically works," the real fun begins! That's when we have enough working code to experiment with ideas.

In this chapter, we will guide you through the considerations a professional programmer might have trying to improve the calculator from Chapter 5. Note that the questions asked about the program and the issues considered here are far more interesting than the calculator itself. What we do is to give an example of how real programs evolve under the pressure of requirements and constraints and of how a programmer can gradually improve code.

6.2 Input and output

If you look back to the beginning of Chapter 5, you'll find that we decided to prompt the user with

```
Expression:
```

and to report back answers with

```
Result:
```

In the heat of getting the program to run, we forgot all about that. That's pretty typical. We can't think of everything all the time, so when we stop to reflect, we find that we have forgotten something or other.

For some programming tasks, the initial requirements cannot be changed. That's usually too rigid a policy and leads to programs that are unnecessarily poor solutions to the problems that they are written to solve. So, let's consider what we would do, assuming that we can change the specification of what exactly the program should do. Do we really want the program to write **Expression:** and **Result:**? How would we know? Just "thinking" rarely helps. We have to try and see what works best.

```
2+3; 5∗7; 2+9;
```

currently gives

```
= 5
= 35
= 11
```

If we used **Expression:** and **Result:**, we'd get

```
Expression: 2+3; 5∗7; 2+9;
Result : 5
Expression: Result: 35
Expression: Result: 11
Expression:
```

We are sure that some people will like one style and others will like the other. In such cases, we can consider giving people a choice, but for this simple calculator that would be overkill, so we must decide. We think that writing **Expression:** and **Result:** is a bit too "heavy" and distracting.

6 Completing a Program

Keep it simple:
as simple as possible,
but no simpler.
– Albert Einstein

Writing a program involves gradually refining our ideas of what we want to do and how we want to express it. In Chapter 5, we produced the initial working version of a calculator program. Here, we'll refine it. Completing the program – that is, making it fit for users and maintainers – involves improving the user interface, doing some serious work on error handling, adding a few useful features, and restructuring the code for ease of understanding and modification.

```
int main()
try
{
	while (cin) {
		cout << "> ";
		Token t = ts.get();
		while (t.kind == ';')
			t=ts.get();		// eat ';'
		if (t.kind == 'q')
			return 0;
		ts.putback(t);
		cout << "= " << expression() << '\n';
	}
	return 0;
}
catch (exception& e) {
	cerr << e.what() << '\n';
	return 1;
}
catch (...) {
	cerr << "exception \n";
	return 2;
}
```

This makes for reasonably robust error handling. So we can start considering what else we can do to improve the calculator.

6.4 Negative numbers

If you tested the calculator, you found that it couldn't handle negative numbers elegantly. For example, this is an error:

```
−1/2
```

We have to write

```
(0−1)/2
```

That's not acceptable.

AA Finding such problems during late debugging and testing is common. Only now do we have the opportunity to see what our design really does and get the feedback that allows us to refine our ideas. When planning a project, it is wise to try to preserve time and flexibility to benefit from the lessons we learn here. All too often, "release 1.0" is shipped without needed refinements because a tight schedule or a rigid project management strategy prevents "late" changes to the specification; "late" addition of "features" is especially dreaded. In reality, when a program is good enough for simple use by its designers but not yet ready to ship, it isn't "late" in the development sequence; it's the earliest time when we can benefit from solid experience with the program. A realistic schedule takes that into account.

In this case, we basically need to modify the grammar to allow unary minus. The simplest change seems to be in **Primary**. We have

```
Primary:
    Number
    "(" Expression ")"
```

and we need something like

```
Primary:
    Number
    "(" Expression ")"
    "–" Primary
    "+" Primary
```

We added unary plus because that's what C++ does. When we have unary minus, someone always tries unary plus and it's easier just to implement that than to explain why it is useless. The code that implements **Primary** becomes

```
double primary()
{
    Token t = ts.get();
    switch (t.kind) {
    case '(':                    // handle '(' expression ')'
    {
        double d = expression();
        t = ts.get();
        if (t.kind != ')')
            error("')' expected");
        return d;
    }
    case '8':                    // we use '8' to represent a number
        return t.value;          // return the number's value
    case '–':
        return – primary();
    case '+':
        return primary();
    default:
        error("primary expected");
    }
}
```

That's so simple that it actually worked the first time.

6.5 Remainder: %

When we first analyzed the ideals for a calculator, we wanted the remainder (modulo) operator: **%**. However, **%** is not defined for floating-point numbers, so we backed off. Now we can consider it again. It should be simple:

[7] Write a program that reads digits and composes them into integers. For example, **123** is read as the characters 1, 2, and 3. The program should output **123 is 1 hundred and 2 tens and 3 ones**. The number should be output as an **int** value. Handle numbers with one, two, three, or four digits. Hint: To get the integer value **5** from the character **'5'** subtract **'0'**, that is, **'5'–'0'==5**.

[8] A permutation is an ordered subset of a set. For example, say you wanted to pick a combination to a vault. There are 60 possible numbers, and you need three different numbers for the combination. There are $P(60,3)$ permutations for the combination, where P is defined by the formula $P(a,b) = (a!)/((a-b)!)$ where ! is used as a suffix factorial operator. For example, 4! is 4∗3∗2∗1.

Combinations are similar to permutations, except that the order of the objects doesn't matter. For example, if you were making a ''banana split'' sundae and wished to use three different flavors of ice cream out of five that you had, you probably wouldn't care if you put a scoop of vanilla at the beginning or the end or the serving dish. The formula for combinations is $C(a,b) = (P(a,b))/(b!)$

Design a program that asks users for two numbers, asks them whether they want to calculate permutations or combinations, and prints out the result. This will have several parts. Do an analysis of the above requirements. Write exactly what the program will have to do. Then, go into the design phase. Write pseudo code for the program and break it into subcomponents. This program should have error checking. Make sure that all erroneous inputs will generate good error messages.

Postscript

Making sense of input is one of the fundamental programming activities. Every program somehow faces that problem. Making sense of something directly produced by a human is among the hardest problems. For example, many aspects of voice recognition are still research problems. Simple variations of this problem, such as our calculator, cope by using a grammar to define the input.

[17] What is “look-ahead”?
[18] What does putback() do and why is it useful?
[19] Why is the remainder (modulus) operation, %, difficult to implement in the term()?
[20] What do we use the two data members of the Token class for?
[21] Why do we (sometimes) split a class’s members into private and public members?
[22] What happens in the Token_stream class when there is a token in the buffer and the get() function is called?
[23] Why were the ';' and 'q' characters added to the switch-statement in the get() function of the Token_stream class?
[24] When should we start testing our program?
[25] What is a “user-defined type”? Why would we want one?
[26] What is the interface to a C++ “user-defined type”?
[27] Why do we want to rely on libraries of code?

Terms

analysis	grammar	prototype	class
implementation	pseudo code	class member	interface
public	data member	member function	syntax analyzer
design	parser	token	divide by zero
private	use case	token stream	look-ahead

Exercises

[1] If you haven’t already, do the TRY THIS exercises from this chapter.
[2] Add the ability to use {} as well as () in the program, so that {(4+5)*6} / (3+4) will be a valid expression.
[3] Add a factorial operator: use a suffix ! operator to represent “factorial.” For example, the expression 7! means 7 * 6 * 5 * 4 * 3 * 2 * 1. Make ! bind tighter than * and /; that is, 7*8! means 7*(8!) rather than (7*8)!. Begin by modifying the grammar to account for a higher-level operator. To agree with the standard mathematical definition of factorial, let 0! evaluate to 1. Hint: The calculator functions deal with doubles, but factorial is defined only for ints, so just for x!, assign the x to an int and calculate the factorial of that int.
[4] Define a class Name_value that holds a string and a value. Rework exercise 20 in Chapter 3 to use a vector<Name_value> instead of two vectors.
[5] Write a grammar for bitwise logical expressions. A bitwise logical expression is much like an arithmetic expression except that the operators are ! (not), ˜ (complement), & (and), | (or), and ˆ (exclusive or). Each operator does its operation to each bit of its integer operands (see PPP2.§25.5). ! and ˜ are prefix unary operators. A ˆ binds tighter than a | (just as * binds tighter than +) so that x|yˆz means x|(yˆz) rather than (x|y)ˆz. The & operator binds tighter than ˆ so that xˆy&z means xˆ(y&z).
[6] Redo the “Bulls and Cows” game from exercise 12 in Chapter 4 to use four letters rather than four digits.

[1] We add % as a Token.
[2] We define a meaning for %.

We know the meaning of % for integer operands. For example:

```
> 2%3;
= 2
> 3%2;
= 1
> 5%3;
= 2
```

But how should we handle operands that are not integers? Consider:

```
> 6.7%3.3;
```

What should be the resulting value? There is no perfect technical answer. However, modulo is often defined for floating-point operands. In particular, x%y can be defined as x%y==x−y*int(x/y), so that 6.7%3.3==6.7−3.3*int(6.7/3.3), that is, 0.1. This is easily done using the standard-library function fmod() (floating-point modulo) (PPP2.§24.8). We modify term() to include

```
case '%':
{     double d = primary();
      if (d == 0)
            error("%:divide by zero");
      left = fmod(left,d);
      t = ts.get();
      break;
}
```

Alternatively, we can prohibit the use of % on a floating-point argument. We check if the floating-point operands have fractional parts and give an error message if they do. The problem of ensuring int operands for % is a variant of the narrowing problem (§2.9), so we could solve it using narrow (§7.4.7):

```
case '%':
{     int i1 = narrow<int>(left);
      int i2 = narrow<int>(primary());
      if (i2 == 0)
            error("%: divide by zero");
      left = i1%i2;
      t = ts.get();
      break;
}
```

For a simple calculator, either solution will do.

6.6 Cleaning up the code

AA We have made several changes to the code. They are, we think, all improvements, but the code is beginning to look a bit messy. Now is a good time to review the code to see if we can make it clearer and shorter, add and improve comments, etc. In other words, we are not finished with the program until we have it in a state suitable for someone else to take over maintenance. Except for the almost total absence of comments, the calculator code really isn't that bad, but let's do a bit of cleanup.

6.6.1 Symbolic constants

Looking back, we find the use of '8' to indicate a Token containing a numeric value odd. It doesn't really matter what value is used to indicate a number Token as long as the value is distinct from all other values indicating different kinds of Tokens. However, the code looks a bit odd and we had to keep reminding ourselves in comments:

```
case '8':                  // we use '8' to represent a number
        return t.value;    // return the number's value
case '–':
        return – primary();
```

XX

To be honest, we also made a few mistakes, typing '0' rather than '8', because we forgot which value we had chosen to use. In other words, using '8' directly in the code manipulating Tokens was sloppy, hard to remember, and error-prone; '8' is one of those "magic constants" we warned against in §3.3.1. What we should have done was to introduce a symbolic name for the constant we used to represent a number:

```
constexpr char number = '8';   // t.kind==number means that t is a number Token
```

The constexpr modifier (§3.3.1) simply tells the compiler that we are defining an object that is not supposed to change: for example, an assignment number='0' would cause the compiler to give an error message. Given that definition of number, we don't have to use '8' explicitly anymore. The code fragment from primary above now becomes

```
case number:
        return t.value;    // return the number's value
case '–':
        return – primary();
```

AA

This requires no comment. We should not say in comments what can be clearly and directly said in code. Repeated comments explaining something are often an indication that the code should be improved.

Similarly, the code in Token_stream::get() that recognizes numbers becomes

```
case '.':
case '0': case '1': case '2': case '3': case '4':
case '5': case '6': case '7': case '8': case '9':
{       cin.putback(ch);        // put digit back into the input stream
        double val;
        cin >> val;             // read a floating-point number
                return Token{number,val};
}
```

We could consider symbolic names for all tokens, but that seems overkill. After all, '(' and '+' are about as obvious a notation for (and + as anyone could come up with. Looking through the tokens, only ';' for "print" (or "terminate expression") and 'q' for "quit" seem arbitrary. Why not 'p' and

TRY THIS

Get the calculator as presented above to run, see what it does, and try to figure out why it works as it does.

Drill

This drill involves a series of modifications of a buggy program to turn it from something useless into something reasonably useful.

[1] Take the calculator from the file calculator02buggy.cpp. Get it to compile. You need to find and fix a few bugs. Those bugs are not in the text in the book. Find the three logic errors deviously inserted in calculator02buggy.cpp and remove them so that the calculator produces correct results.
[2] Change the character used as the exit command from q to x.
[3] Change the character used as the print command from ; to =.
[4] Add a greeting line in main():

```
"Welcome to our simple calculator.
Please enter expressions using floating–point numbers."
```

[5] Improve that greeting by mentioning which operators are available and how to print and exit.

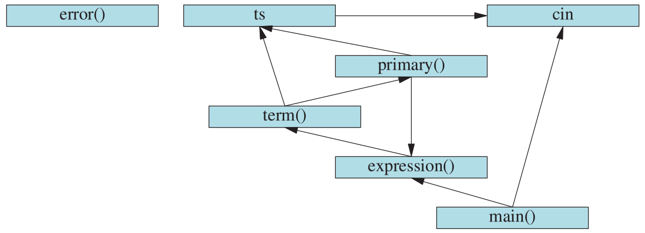
Review

[1] What do we mean by "Programming is understanding"?
[2] The chapter details the creation of a calculator program. Write a short analysis of what the calculator should be able to do.
[3] How do you break a problem up into smaller manageable parts?
[4] Why is creating a small, limited version of a program a good idea?
[5] Why is feature creep a bad idea?
[6] What are the three main phases of software development?
[7] What is a "use case"?
[8] What is the purpose of testing?
[9] According to the outline in the chapter, describe the difference between a Term, an Expression, a Number, and a Primary.
[10] In the chapter, an input was broken down into its components: Terms, Expressions, Primarys, and Numbers. Do this for (17+4)/(5–1).
[11] Why does the program not have a function called number()?
[12] What is a token?
[13] What is a grammar? A grammar rule?
[14] What is a class? What do we use classes for?
[15] How can we provide a default value for a member of a class?
[16] In the expression function, why is the default for the switch-statement to "put back" the token?

```
#include "PPP.h"

class Token { /* ... */ };
class Token_stream { /* ... */ };

void Token_stream::putback(Token t) { /* ... */ }
Token Token_stream::get() { /* ... */ }

Token_stream ts;                    // provides get() and putback()
double expression();                // declaration so that primary() can call expression()

double primary() { /* ... */ }      // deal with numbers and parentheses
double term() { /* ... */ }         // deal with * and /
double expression() { /* ... */ }   // deal with + and -

int main() { /* ... */ }            // main loop and deal with errors
```

The order of the declarations is important. You cannot use a name before it has been declared, so **ts** must be declared before **ts.get()** uses it, and **error()** from **PPP_support** must be declared before the parser functions because they all use it. **CC**

We can represent that graphically (leaving out calls to **error()** – everyone calls **error()**):



There is an interesting loop in the call graph: **expression()** calls **term()** which calls **primary()** which calls **expression()**. This means that we can't just define those three functions: there is no order that allows us to define every function before it is used. We need at least one declaration that isn't also a definition. We chose to declare (''forward declare'') **expression()**.

But does this work? It does, for some definition of "work." It compiles, runs, correctly evaluates expressions, and gives decent error messages. But does it work in a way that we like? The unsurprising answer is "Not really." We tried the first version in §5.6 and removed a serious bug. This second version (§5.7) still has problems. But that's fine (and expected). It is good enough for its main purpose, which is to be something that we can use to verify our basic ideas and get feedback from. As such, it is a success, but try it: it'll (still) drive you nuts!

'e'? In a larger program, it is only a matter of time before such obscure and arbitrary notation becomes a cause of a problem, so we introduce

```
constexpr char quit = 'q';      // t.kind==quit means that t is a quit Token
constexpr char print = ';';     // t.kind==print means that t is a print Token
```

Now we can write **main()**'s loop like this:

```
while (cin) {
        cout << "> ";
        Token t = ts.get();
        while (t.kind == print)
                t=ts.get();
        if (t.kind == quit)
                return 0;
        ts.putback(t);
        cout << "= " << expression() << '\n';
}
```

Introducing symbolic names for ''print'' and ''quit'' makes the code easier to read. In addition, it doesn't encourage someone reading **main()** to make assumptions about how ''print'' and ''quit'' are represented. For example, it should come as no surprise if we decide to change the representation of ''quit'' to **'e'** (for ''exit''). That would now require no change in **main()**.

Now the strings **"> "** and **"= "** stand out. Why do we have these ''magical'' literals in the code? How would a new programmer reading **main()** guess their purpose? Maybe we should add a comment? Adding a comment might be a good idea, but introducing a symbolic name is more effective:

```
constexpr string prompt = "> ";
constexpr string result = "= ";      // used to indicate that what follows is a result
```

Should we want to change the prompt or the result indicator, we can just modify those **constexpr**s. The loop now reads

```
while (cin) {
        cout << prompt;
        Token t = ts.get();
        while (t.kind ==print)
                t=ts.get();
        if (t.kind == quit)
                return 0;
        ts.putback(t);
        cout << result << expression() << '\n';
}
```

6.6.2 Use of functions

The functions we use should reflect the structure of our program, and the names of the functions should identify the logically separate parts of our code. Basically, our program so far is rather good in this respect: **expression()**, **term()**, and **primary()** directly reflect our understanding of the expression grammar, and **get()** handles the input and token recognition. Looking at **main()**, though, we notice that it does two logically separate things:

[1] main() provides general "scaffolding": start the program, end the program, and handle "fatal" errors.

[2] main() handles the calculation loop.

AA

Ideally, a function performs a single logical action (§3.5.1). Having main() perform both of these actions obscures the structure of the program. The obvious solution is to make the calculation loop into a separate function calculate():

```
void calculate()
        // expression evaluation loop
{
        while (cin) {
                cout << prompt;
                Token t = ts.get();
                while (t.kind == print)                 // first discard all "prints"
                        t=ts.get();
                if (t.kind == quit)
                        return;
                ts.putback(t);
                cout << result << expression() << '\n';
        }
}

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

This reflects the structure much more directly and is therefore easier to understand.

6.6.3 Code layout

Looking through the code for ugly code, we find

```
switch (ch) {
case 'q': case ';': case '%': case '(': case ')': case '+': case '–': case '*': case '/':
        return Token{ch};            // let each character represent itself
```

This wasn't too bad before we added 'q', ';', and '%', but now it's beginning to become obscure. Code that is hard to read is where bugs can more easily hide. Using one line per case and adding a couple of comments help. So, Token_stream's get() becomes

```
        case '(': case ')': case '+': case '–': case '*': case '/':
                return Token{ch};                 // let each character represent itself
```

To be honest, we had forgotten ';' for "print" and 'q' for "quit" in our first version. We didn't add them until we needed them for our second solution.

5.8.3 Reading numbers

Now we just have to deal with numbers. That's actually not that easy. How do we really find the value of 123? Well, that's 100+20+3, but how about 12.34, and should we accept scientific notation, such as 12.34e5? We could spend hours or days to get this right, but fortunately, we don't have to. Input streams know what a C++ floating-point literal looks like and how to turn it into a value of type double. All we have to do is to figure out how to tell cin to do that for us inside get():

```
case '.':
case '0': case '1': case '2': case '3': case '4':
case '5': case '6': case '7': case '8': case '9':
{       cin.putback(ch);                  // put digit back into the input stream
        double val = 0;
        cin >> val;                        // read a floating-point number
        return Token{'8',val};             // let '8' represent "a number"
}
```

We – somewhat arbitrarily – chose '8' to represent "a number" in a Token.

How do we know that a number is coming? Well, if we guess from experience or look in a C++ reference (§0.4.1), we find that a numeric literal must start with a digit or . (the decimal point). So, we test for that. Next, we want to let cin read the number, but we have already read the first character (a digit or dot), so just letting cin loose on the rest will give a wrong result. We could try to combine the value of the first character with the value of "the rest" as read by cin; for example, if someone typed 123, we would get 1 and cin would read 23 and we'd have to add 100 to 23. Yuck! And that's a trivial case. Fortunately (and not by accident), cin works much like Token_stream in that you can put a character back into it. So instead of doing any messy arithmetic, we just put the initial character back into cin and then let cin read the whole number.

AA

Please note how we again and again avoid doing complicated work and instead find simpler solutions – often relying on library facilities. That's the essence of good programming: the continuing search for simplicity. Sometimes that's – somewhat facetiously – expressed as "Good programmers are lazy." In that sense (and only in that sense), we should be "lazy"; why write a lot of code if we can find a way of writing far less?

5.9 Program structure

Sometimes, the proverb says, it's hard to see the forest for the trees. Similarly, it is easy to lose sight of a program when looking at all its functions, classes, etc. So, let's have a look at the program with its details omitted:

```
Token Token_stream::get()
{
        if (full) {                         // do we already have a Token ready?
                full = false;               // remove Token from buffer
                return buffer;
        }
        char ch = 0;
        if !(cin >> ch)                     // note that >> skips whitespace (space, newline, tab, etc.)
                error("no input");

        switch (ch) {
        case ';':                                   // for "print"
        case 'q':                                   // for "quit"
        case '(': case ')': case '+': case '–': case '*': case '/':
                return Token{ch};                   // let each character represent itself
        case '.':
        case '0': case '1': case '2': case '3': case '4':
        case '5': case '6': case '7': case '8': case '9':
        {       cin.putback(ch);                    // put digit back into the input stream
                double val = 0;
                cin >> val;                         // read a floating-point number
                return Token{'8',val};              // let '8' represent "a number"
        }
        default:
                error("Bad token");
        }
}
```

Let's examine **get()** in detail. First we check if we already have a **Token** in the buffer. If so, we can just return that:

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

Only if **full** is **false** (that is, there is no token in the buffer) do we need to mess with characters. In that case, we read a character and deal with it appropriately. We look for parentheses, operators, and numbers. Any other character gets us the call of **error()** that terminates the program:

```
default:
        error("Bad token");
```

The **error()** function is described in §4.6.3 and we make it available in **PPP_support**.

We had to decide how to represent the different kinds of **Token**s; that is, we had to choose values for the member **kind**. For simplicity and ease of debugging, we decided to let the **kind** of a **Token** be the parentheses and operators themselves. This leads to extremely simple processing of parentheses and operators:

```
Token Token_stream::get()
        // read characters from cin and compose a Token
{
        if (full) {                 // check if we already have a Token ready
                full = false;
                return buffer;
        }
        char ch;
        cin >> ch;                  // note that >> skips whitespace (space, newline, tab, etc.)

        switch (ch) {
        case quit:
        case print:
        case '(':
        case ')':
        case '+':
        case '–':
        case '*':
        case '/':
        case '%':
                return Token{ch};                           // let each character represent itself
        case '.':                                           // a floating-point-literal can start with a dot
        case '0': case '1': case '2': case '3': case '4':
        case '5': case '6': case '7': case '8': case '9':   // numeric literal
        {       cin.putback(ch);                            // put digit back into the input stream
                double val;
                cin >> val;                                 // read a floating-point number
                return Token{number,val};
        }
        default:
                error("Bad token");
        }
}
```

We could of course have put each digit case on a separate line also, but that didn't seem to buy us any clarity. Also, doing so would prevent **get()** from being viewed in its entirety on a screen at once. Our ideal is for each function to fit on the screen; one obvious place for a bug to hide is in the code that we can't see it because it's off the screen. Code layout matters.

Note also that we changed the plain **'q'** to the symbolic name **quit**. This improves readability.

AA When we clean up code, we might accidentally introduce errors. Always retest the program after cleanup. Better still, do a bit of testing after each set of minor improvements so that if something went wrong you can still remember exactly what you did. Remember: Test early and often.

6.6.4 Commenting

AA We added a few comments as we went along. Good comments are an important part of writing code. We tend to forget about comments in the heat of programming. When you go back to the code to clean it up is an excellent time to look at each part of the program to see if the comments you originally wrote are

[1] Still valid (you might have changed the code since you wrote the comment)
[2] Adequate for a reader (they usually are not)
[3] Not so verbose that they distract from the code

To emphasize that last concern: what is best said in code should be said in code. Avoid comments that explain something that's perfectly clear to someone who knows the programming language. For example:

XX

```
x = b+c;    // add b and c and assign the result to x
```

You'll find such comments in this book, but only when we are trying to explain the use of a language feature that might not yet be familiar to you.

Comments are for things that code expresses poorly. An example is intent: code says what it does, not what it was intended to do (§4.7.2). Look at the calculator code. There is something missing: the functions show how we process expressions and tokens, but there is no indication (except the code) of what we meant expressions and tokens to be. The grammar is a good candidate for something to put in comments or into some documentation of the calculator.

```
/*
    Simple calculator

    Revision history:

    Revised by Bjarne Stroustrup (bjarne@stroustrup.com) November 2023
    Revised by Bjarne Stroustrup November 2013
    Revised by Bjarne Stroustrup May 2007
    Revised by Bjarne Stroustrup August 2006
    Revised by Bjarne Stroustrup August 2004
    Originally written by Bjarne Stroustrup (bs@cs.tamu.edu) Spring 2004.

    This program implements a basic expression calculator.
    Input from cin; output to cout.
    The grammar for input is:

    Statement:
        Expression
        Print
        Quit
    Print:
        ";"
    Quit:
        "q"
    Expression:
        Term
        Expression "+" Term
        Expression "-" Term
```

Note the way we can initialize a data member inside the class itself. That's called *default member initialization* or *in-class initialization* (§8.4.2).

Now, we can define the two member functions. The putback() is easy, so we will define it first. The putback() member function puts its argument back into the Token_stream's buffer:

```
void Token_stream::putback(Token t)
{
    buffer = t;        // copy t to buffer
    full = true;       // buffer is now full
}
```

The keyword void (meaning "nothing") is used to indicate that putback() doesn't return a value.

When we define a member of a class outside the class definition itself, we have to mention which class we mean the member to be a member of. We use the notation

class_name :: *member_name*

for that. In this case, we define Token_stream's member putback.

Why would we define a member outside its class? The main answer is clarity: the class definition (primarily) states what the class can do. Member function definitions are implementations that specify how things are done. We prefer to put them "elsewhere" where they don't distract. Our ideal is to have every logical entity in a program fit on a screen. Class definitions typically do that if the member function definitions are placed elsewhere, but not if they are placed within the class definition ("in-class").

If we wanted to make sure that we didn't try to use putback() twice without reading what we put back in between (using get()), we could add a test:

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

The test of full checks the precondition (§4.7.3.1) "Is the buffer already full?"

Obviously, a Token_stream should start out empty. That is, full should be false until after the first call of get(). We achieve that by initializing the member full right in the definition of Token_stream.

5.8.2 Reading tokens

All the real work is done by get(). If there isn't already a Token in Token_stream::buffer, get() must read characters from cin and compose them into Tokens:

contains what is necessary to implement those public functions, typically data and functions dealing with messy details that the users need not know about and shouldn't directly use.

Let's elaborate the **Token_stream** type a bit. What does a user want from it? Obviously, we want **get()** and **putback()** functions – that's why we invented the notion of a token stream. The **Token_stream** is to make **Token**s out of characters that it reads from input, so we need to define a **Token_stream** that reads from **cin**. Thus, the simplest **Token_stream** looks like this:

```
class Token_stream {
public:
        Token get();                    // get a Token
        void putback(Token t);          // put a Token back
private:
        // ... implementation details ...
};
```

That's all a user needs to use a **Token_stream**. Experienced programmers will wonder why **cin** is the only possible source of characters, but we decided to take our input from the keyboard. We'll revisit that decision in a Chapter 6 exercise.

Why do we use the "verbose" name **putback()** rather than the logically sufficient **put()**? We wanted to emphasize the asymmetry between **get()** and **putback()**; this is an input stream, not something that you can also use for general output. Also, **istream** has a **putback()** function: consistency in naming is a useful property of a system. It helps people remember and helps people avoid errors.

We can now make a **Token_stream** and use it:

```
Token_stream ts;        // a Token_stream called ts
Token t = ts.get();     // get next Token from ts
// ...
ts.putback(t);          // put the Token t back into ts
```

That's all we need to write the rest of the calculator.

5.8.1 Implementing Token_stream

Now, we need to implement those two **Token_stream** functions. How do we represent a **Token_stream**? That is, what data do we need to store in a **Token_stream** for it to do its job? We need space for any token we put back into the **Token_stream**. To simplify, let's say we can put back at most one token at a time. That happens to be sufficient for our program (and for many, many similar programs). That way, we just need space for one **Token** and an indicator of whether that space is full or empty:

```
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
};
```

```
        Term:
            Primary
            Term "*" Primary
            Term "/" Primary
            Term "%" Primary
        Primary:
            Number
            "(" Expression ")"
            "-" Primary
            "+" Primary
        Number:
            floating-point-literal

        Input comes from cin through the Token_stream called ts.
    */
```

Here we used the block comment, which starts with a **/*** and continues until a ***/**. In a real program, the revision history would contain indications of what corrections and improvements were made.

Note that the comments are not the code. In fact, this grammar simplifies a bit: compare the rule for **Statement** with what really happens (e.g., have a peek at the code in the following section). The comment fails to explain the loop in **calculate()** that allows us to do several calculations in a single run of the program. We'll return to that problem in §6.8.1.

6.7 Recovering from errors

Why do we exit when we find an error? That seemed simple and obvious at the time, but why? Couldn't we just write an error message and carry on? After all, we often make little typing errors and such an error doesn't mean that we have decided not to do a calculation. So let's try to recover from an error. That basically means that we have to catch exceptions and continue after we have cleaned up any messes that were left behind.

Until now, all errors have been represented as exceptions and handled by **main()**. If we want to recover from errors, **calculate()** must catch exceptions and try to clean up the mess before trying to evaluate the next expression:

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

We simply made the **while**-loop's block into a **try**-block that writes an error message and cleans up the mess. Once that's done, we carry on as always.

What would "clean up the mess" entail? Basically, getting ready to compute again after an error has been handled means making sure that all our data is in a good and predictable state. In the calculator, the only data we keep outside an individual function is the **Token_stream**. So what we need to do is to ensure that we don't have tokens related to the aborted calculation sitting around to confuse the next calculation. For example,

```
1++2*3; 4+5;
```

will cause an error, and **2*3; 4+5** will be left in the **Token_stream**'s and **cin**'s buffers after the second **+** has triggered an exception. We have two choices:

[1] Purge all tokens from the **Token_stream**.
[2] Purge all tokens from the current calculation from the **Token_stream**.

The first choice discards all (including **4+5;**), whereas the second choice just discards **2*3;**, leaving **4+5** to be evaluated. Either could be a reasonable choice, and either could surprise a user. As it happens, both are about equally simple to implement. We chose the second alternative because it simplifies testing.

So we need to read input until we find a semicolon. This seems simple. We have **get()** to do our reading for us so we can write a **clean_up_mess()** like this:

```
void clean_up_mess()                    // naive
{
        while (true) {                  // skip until we find a print
                Token t = ts.get();
                if (t.kind == print)
                        return;
        }
}
```

Unfortunately, that doesn't work all that well. Why not? Consider this input:

```
1@z; 1+3;
```

The **@** gets us into the **catch**-clause for the **while**-loop. Then, we call **clean_up_mess()** to find the next semicolon. Then, **clean_up_mess()** calls **get()** and reads the **z**. That gives another error (because **z** is not a token) and we find ourselves in **main()**'s **catch(...)** handler, and the program exits. Oops! We don't get a chance to evaluate **1+3**. Back to the drawing board!

We could try more elaborate **try**s and **catch**es, but basically we are heading into an even bigger mess. Many **try**-blocks is a sign of poor design; we have better techniques (§18.4.1, §18.4.2). Errors are hard to handle, and errors during error handling are even worse than other errors. So, let's try to devise some way to flush characters out of a **Token_stream** that couldn't possibly throw an exception. The only way of getting input into our calculator is **get()**, and that can – as we just

```
2;
=2
2+3;
=5
3+4*5;
=23
q
```

At this point we have a good initial version of the calculator. It's not quite what we said we wanted, but we have a program that we can use as the base for making a more acceptable version. Importantly, we can now correct problems and add features one by one while maintaining a working program as we go along.

5.8 Token streams

Before further improving our calculator, let us show the implementation of **Token_stream**. After all, nothing – nothing at all – works until we get correct input. We implemented **Token_stream** first of all but didn't want too much of a digression from the problems of calculation before we had shown a minimal solution.

Input for our calculator is a sequence of tokens, just as we showed for **(1.5+4)*11** above (§5.3.3). What we need is something that reads characters from the standard input, **cin**, and presents the program with the next token when it asks for it. In addition, we saw that we – that is, our calculator program – often read a token too many, so that we must be able to put it back for later use. This is typical and fundamental; when you see **1.5+4** reading strictly left to right, how could you know that the number **1.5** had been completely read without reading the **+**? Until we see the **+** we might be on our way to reading **1.55555**. So, we need a "stream" that produces a token when we ask for one using **get()** and where we can put a token back into the stream using **putback()**. Everything we use in C++ has a type, so we have to start by defining the type **Token_stream**.

You probably noticed the **public:** in the definition of **Token** in §5.3.3. There, it had no apparent purpose. For **Token_stream**, we need it and must explain its function. A C++ user-defined type often consists of two parts: the public interface (labeled **public:**) and the implementation details (labeled **private:**). The idea is to separate what a user of a type needs for convenient use from the details that we need in order to implement the type, but that we'd rather not have users mess with:

```
class Token_stream {
public:
        // user interface
private:
        // implementation details
        // (not directly accessible to users of Token_stream)
};
```

CC Obviously, users and implementers are often just us "playing different roles," but making the distinction between the (public) interface meant for users and the (private) implementation details used only by the implementer is a powerful tool for structuring code. The public interface should contain (only) what a user needs, which is typically a set of functions. The private implementation