2.1 A first C++ program

This section presents a first complete C++ program and introduces the syntactical and semantical terms necessary to understand all its parts.

Here is our first C++ program. It asks for a number a as input and outputs its eighth power a^8. If you have never seen a C++ program before, even this short one might look scary, since it contains a lot of strange-looking symbols and words that are not found in natural language. On the other hand, this is good news: as short as it is, this program already contains many important features of the C++ language. Once we have gone through them in this section, this program (and even other, bigger programs) won’t look scary anymore.

```cpp
// Program: power8.C
// Raise a number to the eighth power.

#include <iostream>

int main()
{
    // input
    std::cout << "Compute a^8 for a =? ";
    int a;
    std::cin >> a;

    // computation
    int b = a * a; // b = a^2
    b = b * b; // b = a^4

    // output b * b, i.e., a^8
    std::cout << a << "^8 = " << b * b << "\n";
    return 0;
}
```

Program 1: props/power8.C

If you compile this program on your computer and then run the executable file produced by the compiler, you find the following line on the standard output. Typically, the standard output is attached to some window on your computer screen.

Compute a^8 for a =?

You can now enter an integer, e.g., 2, using the keyboard. After pressing ENTER, the output on your screen reads as follows.

Compute a^8 for a =? 2
2^8 = 256.
Before discussing the program `power8.C` in detail, let us go over it once quickly. The lines starting with two slashes // are comments; they document the program such that it can easily be understood by a (human) reader. Line 4 contains an `include-directive`; in this case, it indicates that the program uses the input/output library `iostream`. The `main` function, which is the heart of every C++ program, spans lines 6-20. This function is called by the operating system when the program is started; it ends with a return statement in line 19. The value 0 is returned to the operating system, which by convention signals that the program terminated successfully.

The main function is divided into three parts. First, in lines 8-11 the input number is read. Line 9 outputs a message to the user that tells her which kind of input the program expects. In line 10 a variable `a` is declared that acts as a placeholder to store the input. The keyword `int` indicates that `a` is an integer. In line 11, finally, the variable `a` receives its value from the input.

Then in lines 13-15 the actual computation takes place. In line 14, a new variable `b` is declared which acts as a placeholder to store the result of the computation. The variable `b` is initialized to the product `a * a`. Line 16 computes the product `b * b`, that is, `a^2` and stores this result again in `b`.

The third part in lines 17-18 provides the program output. Part of it is the product `b * b`, that is, `a^2`.

Syntax and semantics. In order to understand the program `power8.C` in detail, and more importantly, to write programs yourself later, you need to know the rules according to which programs are written. These rules form the syntax of C++. You further need to know how to interpret a program ("what does the program do?"), and this is determined by the semantics of C++. Even a program that is well-formed according to the C++ syntax may be twisted from a semantical point of view. A valid program is one that is syntactically and semantically correct.

It's the same with natural language: grammar tells you what sentences are, but the interpretation of a sentence (in particular, whether it makes sense at all) requires a concept of meaning.

When a program is invalid, the compiler may output an error message, and this will definitely happen when the program contains syntax errors. A program that is semantically invalid may compile without errors, but we are not allowed to make any assumptions about its behavior; the program could run fine, for example if the semantic error in question has no consequences on a particular platform. On other platforms, the program may behave strangely, or crash. Even on the same platform, it might work sometimes, but fail at other times. We say that the program's behavior is undefined. Clearly, one should avoid writing programs that exhibit undefined behavior.

Sometimes, even valid programs behave differently on different platforms. This is called implementation defined behavior. In contrast to undefined behavior, you can rely on a well-defined and usually small set of possible behaviors. Implementation defined behavior should also be avoided, but sometimes, this is just not possible. In this case, the programmer has to make sure that the program works under all legal behaviors envisioned by the semantics.

The syntax of C++ is specified formally in a mathematical language. The description of the semantics is less strict; it rather resembles the text of a law, and as such it suffers from omissions and possible misinterpretations. The official law of C++ covering both syntax and semantics, is the ISO/IEC standard 14882 from 1998.

While such a formal specification is indispensible (otherwise, how should a compiler know whether your program text is actually a C++ program, and what it is supposed to do?), it is not suitable for learning C++. Throughout this book, we explain the relevant syntactical and semantical terms in natural language and by example. For the sake of readability, we will often not strictly distinguish between syntactical and semantical terms: some terms are most naturally introduced as having both syntactical and semantical aspects, and it depends on the context which aspect is relevant.

### 2.1.1 Comments and layout

Every good program contains comments, for example

```cpp
// Program: power8.C
// Raise a number to the eighth power.
```

A comment starts with two slashes // and continues until the end of the line. Comments do not provide any functionality, meaning that the program would do exactly the same without them. Why is a program without comments bad, then? We do not only write programs for the compiler to translate them into executables, but we also write them for other people (including ourselves) to read, modify, correct or extend them.

Without comments, the latter tasks become very tedious when the program is not completely trivial. Trust us: Even you will not be able to understand your own programs after a couple of weeks, without comments. There is no standard way of writing comments, but we will follow some common-sense guidelines. One of them is that every program—even if it is very simple—should start with one or more lines of comments that mention the program’s name and say what it does. In our case, the above two lines fully suffice.

Another key feature of a readable program is its layout: consider the version of `power8.C` shown in Program 2. We have removed comments, and all “unnecessary” layout elements like spaces, line breaks, blank lines, and indentations.

```cpp
1 #include <iostream>
2 int main(){std::cout<<"Compute a^8 for a = ? \n"; 
3 int a; std::cin>>a; int b=a*a; b=b*b; std::cout<< 
4 a<<"^8 = "<<b<<\n"; return 0; }
```

Program 2: `progs/power8.condensed.C`
2.1, A FIRST C++ PROGRAM

The compiler is completely ignorant about these changes, but a person reading the program will find this condensed version quite difficult to understand. The purpose of a good layout is to visualise the program structure. This for example means that logical blocks of the program should be separated by blank lines, or that one line of source code should be responsible for only one thing. Indentation, like power8.c, has its between the pair of curly braces, is another indispensable ingredient of good layout, although you will only later be able to fully appreciate this.

Typically, collaborative software projects have layout guidelines, making sure that everybody in the project can easily read everybody else’s code. At the level of the simple programs discussed in this book, such formal guidelines are not necessary; we simply adhere to standard guidelines that have proven to work well in practice, and that are being used in almost any other book on C++ as well.

2.1.2 Include directives

Every useful program contains one or more include directives, such as

```cpp
#include <iostream>
```

Usually, these appear at the very beginning of the program. #include directives are needed since, in C++, many important features are not part of the core language. Instead, they are implemented in the so-called standard library which is part of every C++ implementation. A library is a logical unit used to group certain functionality and to provide to the user in a succinct form. In fact, the standard library consists of several libraries one of which is the input/output library.

A library presents its functionality to the user in the form of one or several headers. Each such header contains information that is needed by the compiler. In order to use a certain feature from a library, one has to include the corresponding header into the program by means of an #include directive. In our case, we want to use input and output which are (maybe surprisingly) not part of the core language. The corresponding header of the standard library is called iostream.

A well-designed C++ library puts its functionality into a namespace. The namespace of the standard library is called std. Then, in order to access a feature from the library, we have to qualify its name with the namespace, like in std::cin (this is the feature that allows us to read input from the keyboard). This mechanism helps to avoid name clashes in which different features accidentally get the same name. At the same time, explicit qualification increases the readability of a program, as it is immediately apparent which library a given feature comes from. A name that is not qualified is called unqualified and usually corresponds to a feature defined in our own program.

2.1.3 The main function

Every C++ program must have a main function. The shortest program reads as follows,

```cpp
int main() { return 0; }
```

This program does nothing. The main function is called by the operating system when you tell it to run the program; but why is it a function, and what is "return 0" supposed to mean? Just like a mathematical function, the main function can have arguments given to it upon execution of the program, and the computations within the curly braces yield a function value that is given back (or returned) to the operating system. In our case, we have written a main function that does not expect any arguments (this is indicated by the empty brackets () behind main) and whose return value is the integer 0. The fact that the return value must be an integer is indicated by the word int before main. By convention, return 0 tells the operating system that the program has run successfully (or that we don’t care whether it has), while any other value explicitly signals failure.

In a strict mathematical sense, the main function of power8.c is utterly boring. The whole functionality of the program comes from the effect of the function. This effect is to read a number from the standard input and write its eighth power to the standard output. The fact that functions can have effects sets C++ apart from many functional programming languages.

2.1.4 Values and effects

The value and effect of a function are determined by the C++ semantics. Merely knowing the syntactical rules of writing functions does not tell us anything about values and effects. In this sense, value and effect are purely semantical terms.

For example, we have to know that in C++, the character 0 is interpreted as the integer 0 (although this is not difficult to guess). It is also important to understand that value and effect depend on the concrete program state in which the function is called.

2.1.5 Types and functionality

The word int is the name of a C++ type. This type is used since the program power8.c deals with integers. In mathematics, integers are modeled by the ring \( \mathbb{Z}, +, \cdot \). This algebraic structure defines the integers in terms of their value range (the set \( \mathbb{Z} \)), and in terms of their functionality (addition and multiplication). In C++, integers are modeled by the type int. Like a "mathematical type", a C++ type has a name, a value range, and functionality, defining what we can do with it. When we refer to a type, we will do so by its name. Note that the name is a syntactical aspect of the type, while value range and functionality are of semantical nature.

Conveniently, C++ contains a number of fundamental types (sometimes called built-in types) for typical applications. The type int is one of them. The major difference to the "mathematical type" \( \mathbb{Z}, +, \cdot \) is that int has a finite value range only.
2.1.6 Literals

A literal represents a constant value of some type. For example, in line 19 of the program `power8.C`, 0 is a literal of type int, representing the value 0. For each type, it is separately defined how its literals look like, and what their values are. A literal can be seen as the syntactical counterpart of a value; it makes its value "visible" in the program.

2.1.7 Variables

The line

```
int a;
```

is a declaration of a variable. A variable represents a not necessarily constant value of some type. The variable has a name, a type, a value, and an address (typically in the computer's main memory; you can think of the address simply as the position of the variable in the main memory). The purpose of the address is to know where to store and look up the value. The reason for calling such an entity a variable is that its value can be changed by modifying the memory content at the corresponding address. The address itself may change as well. In contrast, the name and type remain fixed.

When we refer to a variable, we will do so by its name. The declaration `int a;` defines a variable with the following characteristics:

<table>
<thead>
<tr>
<th>name</th>
<th>type</th>
<th>value</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>int</td>
<td>undefined</td>
<td>chosen by compiler/OS</td>
</tr>
</tbody>
</table>

You might wonder why this is called a definition of a, even though it does not define the value of a. But recall that this value depends on the program state, and that the definition fully specifies how the value is obtained: look it up at a's address. Saying that a variable has a value is therefore somewhat imprecise, but we'll stick to it, just like mathematicians talk about function value when they actually mean the value obtained by evaluating the function with concrete arguments. We even go one step further with our sloppiness: if a has value 2, for example, we also say that "a is 2". This is the way that programmers usually talk about variables and their values. We will get to know mechanisms for assigning and changing values of variables in Section 2.1.12.

2.1.8 Identifiers and names

The name of any variable must be an identifier, according to the following definition, and it must be different from certain reserved names like int.

Definition 1 An identifier is any sequence of characters composed of the 52 letters a...z and A...Z, the 10 digits 0...9, and the underscore (_). The first character has to be a letter.

2.1.9 Objects

An object represents a not necessarily constant and possibly anonymous value of a type. Like a variable, an object has a type, a value of its type, and an address, but not necessarily a name. An object is a purely semantical entity: you can't "see" objects in C++ code.

The concept of an object is more general than that of a variable, and we will see shortly that there are natural situations where we access and change values not represented by variables.

More is true: if you want to write interesting programs, it is absolutely necessary to work with values that do not have a variable name attached. This can be seen by the following simple thought experiment: suppose that you have written a program that stores a sequence of values to be read from a file. Now you look at your program and count the number of variables that it contains. Say this number is 31. But in these 31 variables, you can store no more than 31 values. If your program is correct, it can certainly process a file with 32 values, but then there must be at least one value that cannot be stored under a variable name.

2.1.10 Expressions

In the program `power8.C`, three character sequences stand out, because they look familiar and are responsible for the functionality of the program: these are the character sequences `a * a` in line 14 and `b * b` in lines 15 and 18.

An expression represents a computation involving other expressions. More precisely, an expression is either a primary expression, for example a literal or a name, or it is a composite expression. A composite expression is obtained by combining expressions through certain operations, or by putting a pair of parentheses () around an expression.

The expression `a * a` is an arithmetic expression, involving numeric variables and the multiplication operator, just like we know it from mathematics. According to our above definition, `a * a` is a composite expression, built from the multiplication operator and the two primary expressions `a` and `a`.

According to the above definition, an expression is a syntactical entity, but it has semantical aspects as well; any expression has a type, a value of this type, and possibly an effect. The type is fixed, but the value and the effect only materialize when the expression gets evaluated, meaning that the computation it represents is carried out. Evaluating an expression is the most frequent activity going on while a C++ program is

---

1The truth is that the expression involves the names of the variables, but for the sake of readability, we suppress this subtly.
executed; the evaluation computes the value of the expression and carries out its effect (if any).

Type and value of a primary expression are determined by its defining literal, or by type and value of the entity behind its defining name. Primary expressions have no effect. Type, value and effect of a composite expression are determined by the involved operation, depending on the values and effects of the involved subexpressions. Putting parentheses () around an expression yields an expression with the same type, value and effect.

The expression a * a, for example, is of type int, and not unexpectedly, its value is the square of the value of a. The expression has no effect. The expression b = b * b, built from the assignment operator and the two expressions b and b * b, has the same type and value as b * b, but it has an additional effect: it assigns the square of b back to b.2

We say that an expression is evaluated rather than executed, because many expressions do not have an effect, so that their functionality is associated with the value only. Even for expressions with effect, some books use the term side effect to emphasize that the important thing is the value. The C++ entities chiefly responsible for effects are the statements to which we get below.

We want to remark that the only way of accessing an expression’s value is to evaluate it, and this also carries out its effect. You cannot get the value without the effect.

Valid and implementation-defined expressions. An expression is called valid in the context of a program if the C++ semantics defines its possible values and effects, depending on the platform, the program and all its inputs. How can an expression be invalid? For example, after defining the variable a with int a; the C++ standard stipulates that its value is undefined; the next occurrence of a in the program is then an invalid expression. It is important to understand that this also holds if your compiler for example decides to initialise a in int a; with value 0. In this case, the value is defined on your platform, but probably not on other platforms. The relevant fact is that the C++ semantics does not define the value.

An invalid expression is not bad per se: if only its value is undefined, but this value is not used, there is no problem. In our program powers.c, the line int a; is followed by the line std::cout << a, and although a is an invalid expression in that line, its undefined value has no influence on the behavior of the program. This is because std::cout << a; sets a to the value entered by the user, completely ignoring the previous value.

Invalid expressions with undefined effects are always bad, since they lead to undefined behavior of the program. Again, on your specific platform, everything might work, since the compiler may choose to implement an effect that is not harmful in your application, but you cannot rely on it on other platforms.

2 This is a shorthand for the correct, but somewhat clumsy formulation that the new value of b is set to the square of the old value of b.

Here is a scenario where undefined values and effects come up. With rare exceptions, it is not specified in which order the subexpressions in a composite expression are evaluated. If the effect of a subexpression influences values or effects of other subexpressions, value and effect of the composite expression may depend on the evaluation order. In this case, the C++ standard defines the composite expression as invalid.

Among the valid expressions, there are first-class citizens, namely the ones whose values and effects do not depend on the platform. Ideally, all valid expressions in a program should be of that type, since the overall goal is to write platform-independent code. In reality, platforms are so different that this is hard to achieve. As many other languages, C++ allows controlled use of platform-specific features.3 A valid expression whose value or effect depends on the platform is said to have implementation-defined semantics. For such an expression, the C++ standard usually prescribes a small set of possible values and effects. It will then depend on your platform which value and effect from this small set the expression has.

There is no logical way of deducing whether an expression is invalid or has implementation-defined semantics. For example, if the value of an expression depends on the evaluation order of just two subexpressions, you could argue that there are only two possible outcomes. In this case, shouldn’t the value be called implementation-defined rather than undefined? The designers of C++ didn’t think so, and they made the law. You can question the law, of course, but in programming C++, you still have to respect it, otherwise there’s trouble ahead. In our evaluation order example, you simply must not rely on the set of two possible outcomes: since the C++ semantics does not prescribe this set, your compiler might do something else (for example, issue an error message, or let the program crash).

2.1.1 Values and rvalues

An lvalue is an expression that has an address. In the program powers.c, the variable b is an lvalue, and its address is the address of the variable b.

The value of an lvalue is defined as the value of the object at its address. An lvalue can therefore be viewed as the syntactical counterpart of an object: it gives the object a (temporary) name and makes it “visible” within a C++ program. We also say that the lvalue refers to the object at its address.

In particular, any variable is an lvalue. But lvalues provide a means for accessing and changing object values, even without having a corresponding variable. As we will see in Section 2.1.12 below, the expression std::cout << “a hidden” in line 18 is such an lvalue.

Any expression that is not an lvalue is a rvalue. For example, literals are rvalues: there is no address associated with the int literal 0, say. Putting a pair of parenthesis around an rvalue yields an lvalue, and similarly for rvalues.

3 Mostly due to historical reasons, C++ also allows uncontrolled use of platform-specific features, unfortunately. We won’t discuss such uses here.
The terms `lvalue` and `rvalue` already indicate that we think about them not so much in terms of expressions, but rather in terms of their values. We will often identify an lvalue with the object it refers to, and an rvalue simply with its value.

### 2.1.12 Operators

Line 14 of `power8.cpp`, for example, features the binary multiplication operator `*`.

Like a function, an operator expects arguments (here also called operands) of specified types, from which it computes a return value of a specified type, according to its functionality. In addition, these computations may have an effect.

This was the semantic view; on the syntactical level, the operands as well as the composite expression (built from the operator and its operands, see Section 2.1.10), are expressions; the operator specifies for each of them whether it is an lvalue or an rvalue. If the composite expression is an lvalue, the operator is said to return the object referred to by the lvalue. If the composite expression is an rvalue, the operator simply returns its value.

The number of operands is called the arity of the operator. Most operators have arity 1 (unary operators) or 2 (binary operators).

Whenever an rvalue is expected as an operand, it is also possible to provide an lvalue. In this case, the lvalue will simply be interpreted as an rvalue, meaning that its address is only used to look up the value, but not to change it. This is known as lvalue-to-rvalue conversion. In stating that an operand must be an rvalue, the operator therefore guarantees that the operand’s value remains unchanged; by expecting an lvalue, the operator explicitly signals its intention to change the value.

#### Evaluation of composite expressions

When a composite expression involving an operator gets evaluated, the operands are evaluated first (recall that this also carries out the effects of the operands, if any). Based on the resulting values, the operator computes the value of the composite expression. The latter computations may have additional effects, and all effects together form the effect of the composite expression.

#### Operator specifics

What is it that sets operators apart from functions? On the one hand, there is only a finite number of possible operator tokens such as `+` or `*`. Many of these tokens directly correspond to well-known mathematical operator symbols indicating the functionality of the operator. On the other hand, and most conveniently, operator calls do not have to obey the usual function call notation, like in `f(x, y)`. After all, we want to write `a * a` in a program, and not `*(a, a)`. In summary, operators let us write more natural and more readable code.

Unfortunately, the token `*` corresponds to mathematical assignment `=`, and not to mathematical equality `=`; a constant source of confusion for beginners.

Four different operators (all of them binary) occur in `power8.cpp`, namely the multiplication operator `*`, the assignment operator `=`, the input operator `>>`, and the output operator `<<`. Let us discuss them in turn.

#### Multiplication operator

The multiplication operator `*` expects two rvalue operands of some type `T`, and it returns the product of its two operands as an rvalue. The multiplication operator has no effect on its own.

#### Assignment operator

The assignment operator `=` expects an lvalue of some type `T` as its first operand, and an rvalue of the same type as its second operand. It assigns the value of the second operand to the first operand and returns the first operand as an lvalue.

In our program `power8.cpp`, the expression `b = b * b` therefore sets the value of `b` to the square of its previous value, and then returns `b`.

In fact, the letter `*` in the term `lvalue` stands for the fact that the expression may appear on the left hand side of an assignment operator. Similarly, the term `rvalue` signals an expression that may appear only on the right hand side of an assignment operator.

#### Input Operator

In `power8.cpp`, the composite expression `std::cin >> a` in line 11 sets the variable `a` to the next value from the standard input, usually the keyboard.

In general, the input operator `>>` expects as its first operand an lvalue referring to an input stream. The second operand is an lvalue of some type `T`. The operator sets the second operand to the next value read from the input stream and returns the stream as an lvalue.

An input stream represents the state of some input device. We think of this device as producing a continuous stream of data that can be tapped to provide input on demand. Under this point of view, the state of the stream corresponds to the sequence of data not yet read. In setting the value of its second operand, the input operator removes one data item from the stream to reflect the fact that this item has now been read. For this, it is important that the stream comes as an lvalue. Conceptually, an input stream is also considered part of the program state.

How much of the data is read as one item, and how exactly it is interpreted as a value of type `T` highly depends on the type `T` of the second operand. For now, it is enough to know that this interpretation is readily defined for the type `int` and for the other fundamental types that we will encounter in the following sections.

In C++, the value `std::cin` refers to the variable `cin` defined in the input/output library, and this variable corresponds to the standard input stream.

It is up to the program’s caller to fill the standard input stream with data. For example, suppose that the program was started from a command shell. Then usually, while the program is running, all input to the command shell is forwarded to the program’s standard input stream. It is also possible to redirect the program’s standard input stream to read data from a file instead.
Output Operator. In power8.C, the composite expression std::cout << a in line 18 writes the value of a to the standard output, usually the computer screen.

In general, the output operator << expects as its first operand an value referring to an output stream. The second operand is an value of some type T. The operator writes the value of the second operand to the output stream and returns the output stream as an value.

An output stream represents the state of some output device. We think of this device as storing the continuous stream of output data that is generated by the program. In writing to the stream, the output operator therefore changes the stream state, and this makes it necessary to provide the stream as an value. Conceptually, an output stream is also considered part of the program state.

It depends on the type T in which format the second operand's value is written to the stream; for the type int and the other fundamental types, this format is readily defined.

C++ defines a standard output stream std::cout and a standard error stream std::cerr in the input/output library.

It is up to the program’s caller to process these output streams. For example, suppose that the program was started from a command shell, then usually, while the program is running, both standard output stream and standard error stream are forwarded to the command shell, but it is also possible to redirect one or both of these streams to write to a file instead. This can be useful to separate regular output (sent to std::cout) from error output (sent to std::cerr).

As indicated above for input streams, it is possible to output several values through one expression, as in

```cpp
std::cout << a << " + " << b + b << ".
```

May be this looks a bit strange, because there is more than one << operator token and more than two operands; but in mathematics, we also write a + b + c as a shortcut for either (a + b) + c or a + (b + c); because addition is associative, we don’t even have to specify which variant we intend.

In C++, such shortcuts are also allowed in order to avoid cluttering up the code with parentheses. But C++ operators are in general not associative, so we have to know the ‘logical parentheses’ in order to understand the meaning of the shortcut.

The operators >> and << are left-associative, meaning that the above expression is logically parenthesized as follows,

```cpp
((std::cout << a) << " + ") << b + b) << ".
```

Recall that the innermost expression std::cout << a is an value referring to the standard output stream. Hence, this expression serves as a legal first operand for the next

outer composite expression (std::cout << a) << " + " and so on. The full expression therefore outputs the values of all expressions occurring after some <<, from left to right. The rightmost of these expressions ends with ‘\n’ which encodes a line break (newline).

2.1.13 Statements
A statement is a basic building block of a C++ program, and it possibly has an effect. The effect depends on the program state and materializes when the statement is executed. As with expressions, we say that a statement does something. A statement usually ends with a semicolon and represents a “step” of the program. Statements are executed in top-to-bottom order. The shortest possible statement is the null statement consisting only of the semicolon; it has no effect. In a typical program, most statements evaluate one or several expressions.

A statement is not restricted to one line of source code; on the contrary, readability often requires to break up statements into several lines of code. The compiler ignores these line breaks, as long as we do not put them at unreasonable places like in the middle of a name.

In power8.C, there are three kinds of statements.

Expression statement. Appending a semicolon to an expression leads to an expression statement. It evaluates the expression but does not make use of its value. This is a frequent form of statements, and in our small program, the statement

```cpp
b = b + b;
```

as well as all statements starting with std::cin or std::cout are expression statements.

Declaration statement. Such a statement introduces a new name into the program. This can be the name of a variable of a given type, like in the declaration statement

```cpp
int a;
```

and

```cpp
int b = a + a;
```

A declaration statement consist of a declaration and a concluding semicolon. In our program power8.C, we deal with variable declarations; they can be of the form

```
T x
```

or

```
T x = expr
```
Return statement. Such a statement is of the form
\[
\text{return expression;}
\]
where expression is a value. It only occurs within a function. The return statement evaluates expression, finishes the function’s computations, and puts expression’s value at some (temporary) address that the caller of the function can access. Abstracting from these technicalities, we simply say that the statement returns expression to the caller.

We have seen only one example so far: the statement return 0; returns the value 0 (formally, the literal 0 of value 0) to the operating system which has called the main function of our program.

Figure 2 summarizes the syntactical and semantical terms we have introduced, along with their relations. The figure emphasizes the central role that expressions play in C++.

### 2.1.14 Details

Commenting. There is a way of writing comments that are not limited to one line of code. Any text enclosed by /* (start of comment) and */ (end of comment) is ignored by the compiler. The initial comment of our program `power8.c` could also have been written as
\[
\text{ /* Program: power8.c }
\text{ Raise a number to the power 8. */}
\]

This mechanism may seem useful for longer comments spanning several lines of code, but the problem is that you do not immediately recognize a line in the middle of such a construction as a comment: you always have to look for the enclosing /* and */ to be sure.

Sometimes, /* and */ are used for very short comments within lines of code, like in `c = a + /* don't subtract! */ b;`.

For readability reasons, we do not advocate this kind of comment, either.

Identifiers starting with an underscore. Occasionally, real-life C++ code contains "identifiers" starting with the underscore character _, although this is not allowed according to Definition 1. The truth is that the programmer is not allowed to use such "identifiers"; they are reserved for internal use by the compiler. Compilers should issue at least a warning, when they discover such a badly formed "identifier", but often they just let it pass.

Define variables where needed! In C++, it is good general practice to define a variable immediately before it is used the first time. The readability of a program improves, if for any variable that appears in some expression, the corresponding definition is nearby and can hence be found quickly. This guideline is in contrast to some other programming languages; for example, in C all variables have to be declared at the beginning of the
2.1. A FIRST C++ PROGRAM

function where they are used.

The main function. The main function is an exceptional function in several ways. One particular specialty is that the return statement can be omitted. A main function without a return statement at the end behaves precisely as if it would end with return 0. This definition has been made for historical reasons mostly; it is an anomaly compared to other functions (which will be discussed later). Therefore, we stick to the explicit return statement and ask you to do the same.

Using directives. It is possible to avoid all std:: prefixes through one additional line of code, a using directive. In case of power8.C, this would look like in Program 3:

```cpp
1 // Program: power8.C
2 // Raise a number to the eighth power.
3 #include <iostream>
4 using namespace std;
5 int main()
6 {
7 // input
8 cout << "Compute x^8 for x = ? ";
9 int a;
10 cin >> a;
11 // computation
12 int b = a * a; // b = a^2
13 b = b * b; // b = a^4
14 // output b * b, i.e., a^8
15 cout << a << "^8 = " << b * b << ".n";
16 return 0;
17 }
```

Program 3: progs/power8_using.C

The using directive is a declaration statement of the form

```cpp
using namespace X;
```

It allows us to use all features from namespace X without qualifying them through the prefix X::. This mechanism seems quite helpful at first sight, but it has severe drawbacks that prevent us from using (let alone advocating) it in this book.

Let's start with the major drawback. Namespaces may have a large number of features (in particular, the namespace std has), with a large number of names. cin and cout are two such names from the namespace std. It is very difficult (and also not desirable) to know all these names. On the other hand, it would be good to know them in order to avoid conflicts with the names we introduce. For example, if we define a variable named cout somewhere in Program 3, we are asking for trouble: when we later use the expression cout, it is not clear whether it refers to the standard output stream, or to our variable.

We can easily avoid the variable name cout, of course, but we may accidentally introduce another name that also appears in the namespace std. The unfortunate consequence is that in some expression of our program, this name might not refer to the feature we introduced, but to a feature of the same name from the standard library. We may end up silently using a feature from the standard library that we don't even know and that we never intended to use. The resulting strange behavior of our program can be very difficult to track down.

In the original program power8.C, introducing a name cout (or any other name also appearing in namespace std) does not cause any harm: without the std:: qualification, it can never "accidentally" refer to something from the standard library.

Here is the second drawback of using directives. A large program contains many names, and in order to keep track of, it is desirable that the name "tell" us where it comes from: is it a name we have introduced, or does it come from a library? If so, from which one? With using directives, we lose that information, meaning that the program becomes less readable and more difficult to maintain.

2.1.15 Goals

Dispositional. At this point, you should . . .

1) understand the basic syntactical and semantical terms of C++, in particular expression, operator, statement, lvalue, rvalue, literal and variable;
2) understand syntax and semantics of the program power8.C.

Operational. In particular, you should be able to . . .

(G1) describe the basic syntactical and semantical terms of C++ (as listed above) in your own words and give examples;
(G2) tell whether a given character sequence is an identifier;
(G3) tell whether a given character sequence is a simple expression, as defined below;
(G4) find out whether a given simple expression is an lvalue or an rvalue;
(G5) evaluate a given simple expression;
(G6) write programs with functionality similar to power8.C.

A simple expression is an expression which only involves int-literals, identifiers, the binary multiplication operator, the assignment operator, and parentheses.
2.1.16 Exercises

Exercise 1 Which of the following character sequences are not C++ identifiers, and why not? (G2)
(a) identifier (b) int (c) x_i (d) 4x_ (e) A99_ (f) _tmp (g) T# (h) x12b

Exercise 2 Which of the following character sequences are not valid C++ expressions, and why not? Here, a and b are variables of type int. (G3)
(a) 1*(2*3) (b) a=(b=5) (c) 1=a (d) (a=1) (e) (a=5)*(b=7) (f) (1 (g) (a-b)*(b=5) (h) (a+3)=(b+5)

Exercise 3 For all of the valid expressions that you have identified in Exercise 2, decide whether these are lvalues or rvalues, and explain your decisions. (G4)

Exercise 4 Determine the values of the valid expressions that you have identified in Exercise 2, and explain how these values are obtained. (G5)

Exercise 5 Write a program multhree.C that reads three integers a, b, c from standard input and outputs their product abc. (G6)

Exercise 6 Write a program power20.C that reads an integer a from standard input and outputs a^20 using at most five multiplications. (G6)