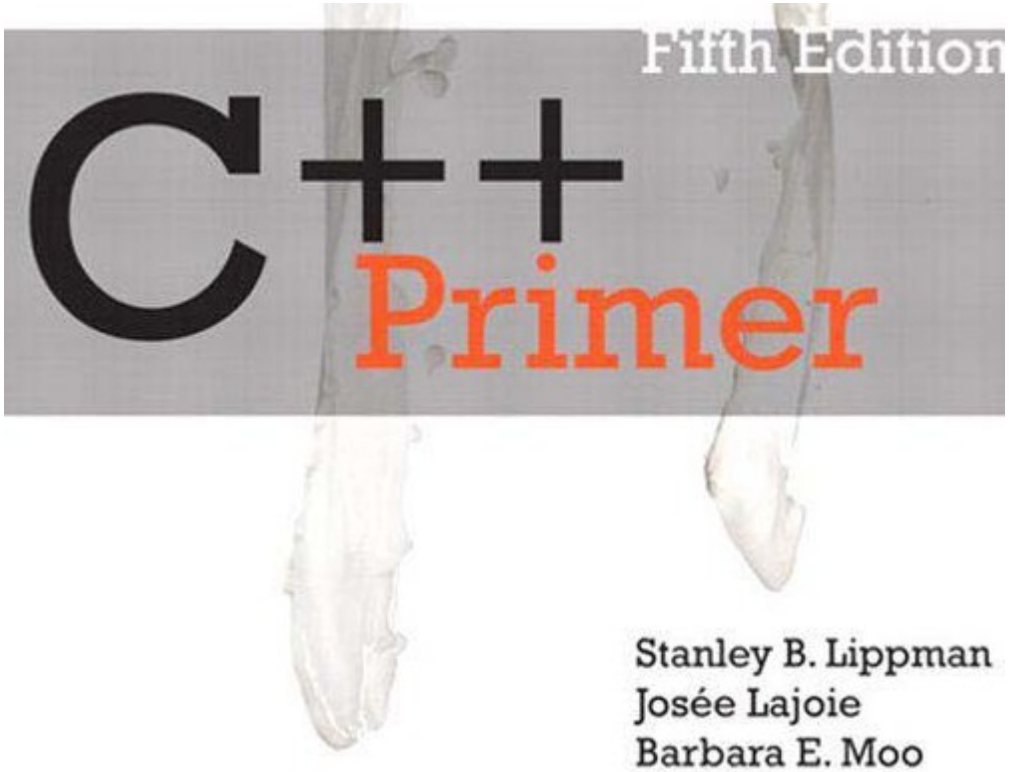
C++ Language Basics

Chapter 1. Getting Started


Fifth Edition
C++
Primer
Stanley B. Lippman
Josée Lajoie
Barbara E. Moo

1.1. Writing a Simple C++ Program

Every C++ program contains one or more functions, one of which must be named **main**. The operating system runs a C++ program by calling `main`.

Here is a simple version of `main` that does nothing but return a value to the operating system:

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

A function definition has four elements:

- return type,
- function name,
- (possibly empty) parameter list enclosed in parentheses,
- function body.

Although `main` is special in some ways, we define `main` the same way we define any other function.

1.2. A First Look at Input/Output

The C++ language does not define any statements to do input or output (IO). Instead, C++ includes an extensive standard library that provides IO (and many other facilities).

a few basic concepts and operations from the IO library can get programming started

Most of the examples in this book use the iostream library. Fundamental to the iostream library are two types named istream and ostream, which represent input and output streams, respectively.

A stream is a sequence of characters read from or written to an IO device. The term stream is intended to suggest that the characters are generated, or consumed, sequentially over time

Standard Input and Output Objects

The library defines four IO objects.

- Input, we use an object of type `istream` named **cin** (pronounced see-in). This object is also referred to as the standard input.

- Output, we use an `ostream` object named **cout** (pronounced see-out). This object is also known as the standard output.

- The library also defines two other `ostream` objects, named **cerr** and **clog** (pronounced see-err and see-log, respectively).

- We typically use `cerr`, referred to as the standard error, for warning and error messages and

- `clog` for general information about the execution of the program

```
#include <iostream>
int main()
{
  std::cout << "Enter two numbers:" <<  std::endl;
  int v1 = 0, v2 = 0;
  std::cin >> v1 >> v2;
  std::cout << "The sum of " << v1 << " and " << v2
  << " is " << v1 + v2 << std::endl;
  return 0;
}
```

Using Names from the Standard Library

note that this program uses `std::cout` and `std::endl` rather than just `cout` and `endl`. The prefix `std::` indicates that the names `cout` and `endl` are defined inside the namespace named **std**.

Namespaces

allow avoiding inadvertent collisions between the names we define and uses of those same names inside a library. All the names defined by the standard library are in the **`std`** namespace.

One side effect of the library's use of a namespace is that when we use a name from the library, we must say explicitly that we want to use the name from the `std` namespace.

Scope Operator

Writing `std::cout` uses the **scope operator** (the **::** operator) to say that we want to use the name `cout` that is defined in the namespace `std`.

§ 3.1 (p.82) shows a simpler way to access names from the library.

3.1. Namespace **using** Declarations

Up to now, our programs have explicitly indicated that each library name we use is in the `std` namespace. For example, to read from the standard input, we write `std::cin`. These names use the **scope** operator (`::`) (§ 1.2, p. 8), which says that the compiler should look in the scope of the left-hand operand for the name of the right-hand operand. Thus, `std::cin` says that we want to use the name `cin` from the namespace `std`.

Referring to library names with this notation can be cumbersome. Fortunately, there are easier ways to use namespace members. The safest way is a **using** declaration. § 18.2.2 (p. 793) covers another way to use names from a namespace.

A `using` declaration lets us use a name from a namespace without qualifying the name with a `namespace_name::` prefix. A `using` declaration has the form `using` *`namespace::name;`*

Once the `using` declaration has been made, we can access name directly:

A Separate **using** Declaration Is Required for Each Name

```
#include <iostream>
// using declarations for names from the standard library
using std::cin;
using std::cout; using std::endl;
int main()
{
cout << "Enter two numbers:" << endl;
int v1, v2;
cin >> v1 >> v2;
cout << "The sum of " << v1 << " and " << v2
<< " is " << v1 + v2 << endl;
return 0;
}
```

A **using** directive, like a `using` declaration, allows us to use the unqualified form of a namespace name. Unlike a `using` declaration, we retain no control over which names are made visible—they all are.

A `using` directive begins with the keyword `using`, followed by the keyword `namespace`, followed by a namespace name. It is an error if the name is not a previously defined namespace name. A `using` directive may appear in global, local, or namespace scope. It may not appear in a class scope.
These directives make all the names from a specific namespace visible without qualification. The short form names can be used from the point of the `using` directive to the end of the scope in which the `using` directive appears.

In the case of a `using` declaration, we
are simply making name directly accessible in the local scope. In contrast, a `using` directive makes the entire contents of a namespace available In general, a namespace might include definitions that cannot appear in a local scope. As a consequence, a `using` directive is treated as if it appeared in the nearest enclosing namespace scope.

When a namespace is injected into an enclosing scope, it is possible for names in the namespace to conflict with other names defined in that (enclosing) scope.

Caution: Avoid **using** Directives

`using` directives, which inject all the names from a namespace, are deceptively simple to use: With only a single statement, all the member names of a namespace are suddenly visible. Although this approach may seem simple, it can introduce its own problems. If an application uses many libraries, and if the names within these libraries are made visible with `using` directives, then we are back to square one, and the global namespace pollution problem reappears. Moreover, it is possible that a working program will fail to compile when a new version of the library is introduced. This problem can arise if a new version introduces a name that conflicts with a name that the application is using.

1.4.3. Reading an Unknown Number of Inputs

```
#include <iostream>
int main()
{
  int sum = 0, value = 0;
  // read until end-of-file, calculating a running total of all values read
  while (std::cin >> value)
  sum += value; // equivalent to sum = sum + value
  std::cout << "Sum is: " << sum << std::endl;
  return 0;
}
```

If we give this program the input
3 4 5 6
then our output will be 18

Numbers are separated by white space. Each value is read until an end-of-line or white space is encountered.

When we use an istream as a condition, **the effect is to test the state of the stream.** If the stream is valid—that is, if the stream hasn't encountered an error—then the test succeeds.

An istream becomes invalid when we hit end-of-file or encounter an invalid input, such as reading a value that is not an integer. An istream that is in an invalid state will cause the condition to yield false.

Header Files

Although we can define a class inside a function, such classes have limited functionality. As a result, classes ordinarily are not defined inside functions.

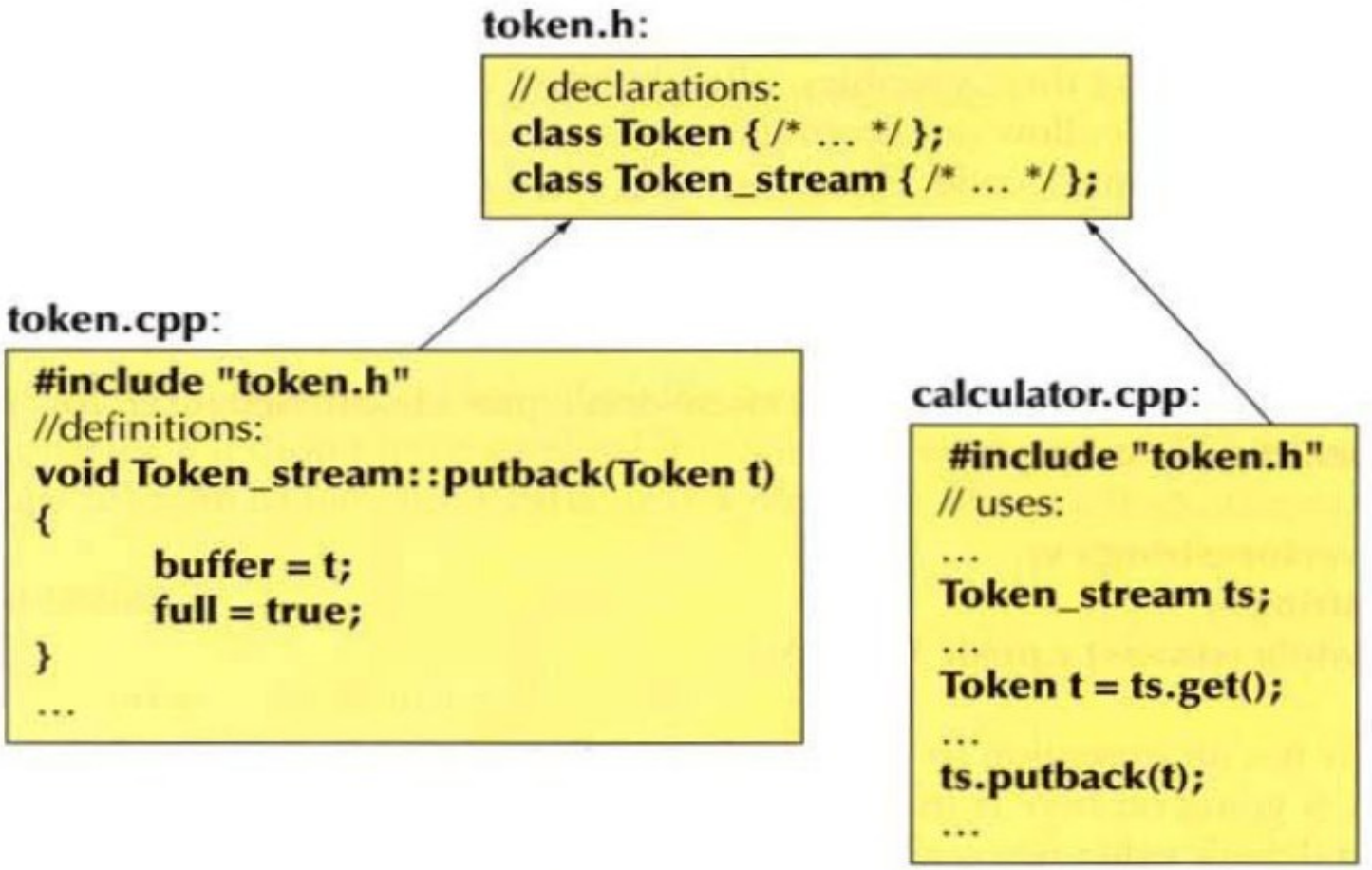
When we define a class outside of a function, there may be only one definition of that class in any given source file. In addition, if we use a class in several different files, the class' definition must be the same in each file.

In order to ensure that the class definition is the same in each file, classes are usually defined in header files.

Typically, classes are stored in headers whose name derives from the name of the class. For example, the string library type is defined in the string header. Similarly, as we've already seen, we will define our Sales_data class in a header file named Sales_data.h.

Header & Source files (abc.h and abc.cpp from Stroustrup p 262)

We could define a header file **token.h** containing declarations needed to use **Token** and **Token_stream**:



The declarations of **Token** and **Token_stream** are in the header **token.h**. Their definitions are in **token.cpp**. The **.h** suffix is the most common for C++ headers, and the **.cpp** suffix is the most common for C++ source files.

Header Guards

C++ programs also use the preprocessor to define header guards. Header guards rely on preprocessor variables.

Preprocessor variables have one of two possible states: defined or not defined. The #define directive takes a name and defines that name as a preprocessor variable.

There are two other directives that test whether a given preprocessor variable has or has not been defined: #ifdef is true if the variable has been defined, and #ifndef is true if the variable has not been defined. If the test is true, then everything following the #ifdef or #ifndef is processed up to thematching #endif.

We can use these facilities to guard against multiple inclusion as follows:

```
#ifndef SALES_DATA_H
#define SALES_DATA_H
#include <string>
struct Sales_data {
   std::string bookNo;
   unsigned units_sold = 0;
   double revenue = 0.0;
};
#endif
```

2.4. **const** Qualifier

We can make a variable unchangeable by defining the variable's type as **const**:

```
const int bufSize = 512; // input buffer size
```

defines `bufSize` as a constant. Any attempt to assign to `bufSize` is an error:

Because we can't change the value of a `const` object after we create it, it must be initialized. As usual, the initializer may be an arbitrarily complicated expression:

When a `const` object is initialized from a compile-time constant

```
const int bufSize = 512; // input buffer size
```

the compiler will usually replace uses of the variable with its corresponding value during compilation. That is, the compiler will generate code using the value `512` in the places that our code uses `bufSize`.

When we split a program into multiple files, every file that uses the `const` must have access to its initializer. In order to see the initializer, the variable must be defined in every file that wants to use the variable's value. To support this usage, yet avoid multiple definitions of the same variable, **`const` variables are defined as local to the file.** When we define a `const` with the same name in multiple files, it is as if we had written definitions for separate variables in each file.

Extern keyword

Sometimes we have a `const` variable that we want to share across multiple files but whose initializer is not a constant expression. In this case, we don't want the compiler to generate a separate variable in each file. Instead, we want the `const` object to behave like other (non`const`) variables. We want to define the `const` in one file, and declare it in the other files that use that object.

To define a single instance of a `const` variable, we use the keyword **`extern`** on both its definition and declaration(s):

```
// file_1.cc defines and initializes a const that is accessible to other files
extern const int bufSize = fcn();
// file_1.h
extern const int bufSize; // same bufSize as defined in file_1.cc
```

In this program, `file_1.cc` defines and initializes `bufSize`. Because this declaration includes an initializer, it is (as usual) a definition. However, because `bufSize` is `const`, we must specify `extern` in order for `bufSize` to be used in other files. The declaration in `file_1.h` is also `extern`. In this case, the `extern` signifies that `bufSize` is not local to this file and that its definition will occur elsewhere.

Exercises Section 2.4.2

C++ is a strongly typed language

3.8 Types and objects

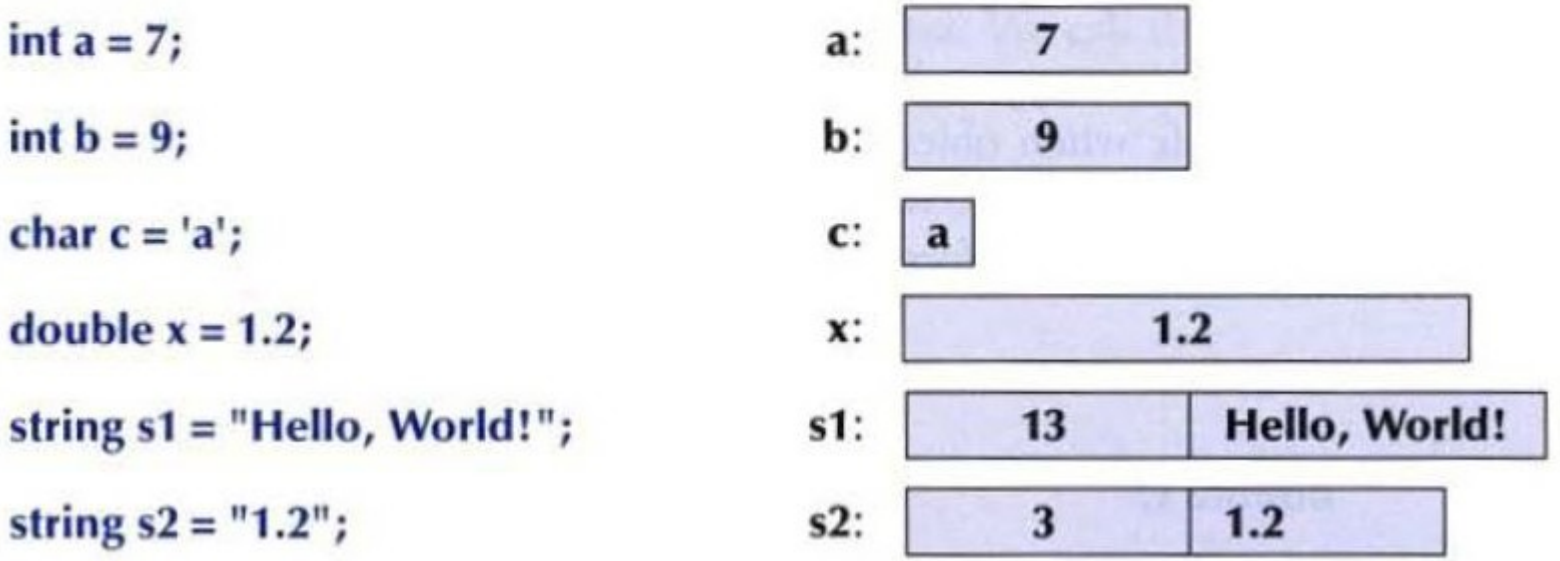
The notion of type is central to C++ and most other programming languages. Let's take a closer and slightly more technical look at types, specifically at the types of the objects in which we store our data during computation. It'll save time in the long run, and it may save you some confusion.

- A *type* defines a set of possible values and a set of operations (for an object).
- An *object* is some memory that holds a value of a given type.
- A *value* is a set of bits in memory interpreted according to a type.
- A *variable* is a named object.
- A *declaration* is a statement that gives a name to an object.
- A *definition* is a declaration that sets aside memory for an object.

Stroustrup

Everything has a type declaration – The compiler strongly checks type agreement

Informally, we think of an object as a box into which we can put values of a given type. An **int** box can hold integers, such as **7**, **42**, and **–399**. A **string** box can hold character string values, such as **"Interoperability"**, **"tokens: !@#$%^&*"**, and **"Old McDonald had a farm"**. Graphically, we can think of it like this:



3.9 Type safety

Every object is given a type when it is defined. A program – or a part of a program – is type-safe when objects are used only according to the rules for their type. Unfortunately, there are ways of doing operations that are not type-safe. For example, using a variable before it has been initialized is not considered type-safe:

Wrong

```
int main()
{
        double x;              // we "forgot" to initialize:
                               // the value of x is undefined

        double y = x;          // the value of y is undefined
        double z = 2.0+x;      // the meaning of + and the value of z are undefined
}
```