

# The C++ Programming Language

Fourth Edition

Bjarne Stroustrup

◆ Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco  
New York • Toronto • Montreal • London • Munich • Paris • Madrid  
Capetown • Sydney • Tokyo • Singapore • Mexico City

# Contents

<b>Contents</b>	<b>iii</b>
<b>Preface</b>	<b>v</b>
Preface to the Fourth Edition .....	v
Preface to the Third Edition .....	ix
Preface to the Second Edition .....	xi
Preface to the First Edition .....	xii
<b>Part I: Introductory Material</b>	<b>1</b>
1. Notes to the Reader .....	3
2. A Tour of C++: The Basics .....	37
3. A Tour of C++: Abstraction Mechanisms .....	59
4. A Tour of C++: Containers and Algorithms .....	87
5. A Tour of C++: Concurrency and Utilities .....	111
<b>Part II: Basic Facilities</b>	<b>133</b>
6. Types and Declarations .....	135
7. Pointers, Arrays, and References .....	171
8. Structures, Unions, and Enumerations .....	201
9. Statements .....	225
10. Expressions .....	241

11. Select Operations .....	273
12. Functions .....	305
13. Exception Handling .....	343
14. Namespaces .....	389
15. Source Files and Programs .....	419

**Part III: Abstraction Mechanisms****447**

16. Classes .....	449
17. Construction, Cleanup, Copy, and Move .....	481
18. Operator Overloading .....	527
19. Special Operators .....	549
20. Derived Classes .....	577
21. Class Hierarchies .....	613
22. Run-Time Type Information .....	641
23. Templates .....	665
24. Generic Programming .....	699
25. Specialization .....	721
26. Instantiation .....	741
27. Templates and Hierarchies .....	759
28. Metaprogramming .....	779
29. A Matrix Design .....	827

**Part IV: The Standard Library****857**

30. Standard Library Summary .....	859
31. STL Containers .....	885
32. STL Algorithms .....	927
33. STL Iterators .....	953
34. Memory and Resources .....	973
35. Utilities .....	1009
36. Strings .....	1033
37. Regular Expressions .....	1051
38. I/O Streams .....	1073
39. Locales .....	1109
40. Numerics .....	1159
41. Concurrency .....	1191
42. Threads and Tasks .....	1209
43. The C Standard Library .....	1253
44. Compatibility .....	1267

**Index****1281**

---

---

## A Tour of C++: The Basics

*The first thing we do, let's  
kill all the language lawyers.  
– Henry VI, Part II*

- Introduction
- The Basics
  - Hello, World!; Types, Variables, and Arithmetic; Constants; Tests and Loops; Pointers, Arrays, and Loops
- User-Defined Types
  - Structures; Classes; Enumerations
- Modularity
  - Separate Compilation; Namespaces; Error Handling
- Postscript
- Advice

### 2.1 Introduction

The aim of this chapter and the next three is to give you an idea of what C++ is, without going into a lot of details. This chapter informally presents the notation of C++, C++'s model of memory and computation, and the basic mechanisms for organizing code into a program. These are the language facilities supporting the styles most often seen in C and sometimes called *procedural programming*. Chapter 3 follows up by presenting C++'s abstraction mechanisms. Chapter 4 and Chapter 5 give examples of standard-library facilities.

The assumption is that you have programmed before. If not, please consider reading a textbook, such as *Programming: Principles and Practice Using C++* [Stroustrup,2009], before continuing here. Even if you have programmed before, the language you used or the applications you wrote may be very different from the style of C++ presented here. If you find this “lightning tour” confusing, skip to the more systematic presentation starting in Chapter 6.

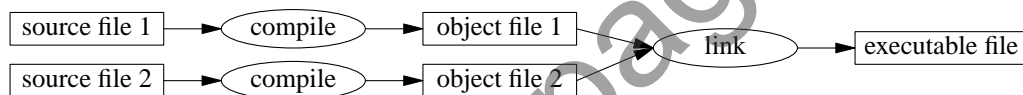
This tour of C++ saves us from a strictly bottom-up presentation of language and library facilities by enabling the use of a rich set of facilities even in early chapters. For example, loops are not discussed in detail until Chapter 10, but they will be used in obvious ways long before that. Similarly, the detailed description of classes, templates, free-store use, and the standard library are spread over many chapters, but standard-library types, such as `vector`, `string`, `complex`, `map`, `unique_ptr`, and `ostream`, are used freely where needed to improve code examples.

As an analogy, think of a short sightseeing tour of a city, such as Copenhagen or New York. In just a few hours, you are given a quick peek at the major attractions, told a few background stories, and usually given some suggestions about what to see next. You do *not* know the city after such a tour. You do *not* understand all you have seen and heard. To really know a city, you have to live in it, often for years. However, with a bit of luck, you will have gained a bit of an overview, a notion of what is special about the city, and ideas of what might be of interest to you. After the tour, the real exploration can begin.

This tour presents C++ as an integrated whole, rather than as a layer cake. Consequently, it does not identify language features as present in C, part of C++98, or new in C++11. Such historical information can be found in §1.4 and Chapter 44.

## 2.2 The Basics

C++ is a compiled language. For a program to run, its source text has to be processed by a compiler, producing object files, which are combined by a linker yielding an executable program. A C++ program typically consists of many source code files (usually simply called *source files*).



An executable program is created for a specific hardware/system combination; it is not portable, say, from a Mac to a Windows PC. When we talk about portability of C++ programs, we usually mean portability of source code; that is, the source code can be successfully compiled and run on a variety of systems.

The ISO C++ standard defines two kinds of entities:

- *Core language features*, such as built-in types (e.g., `char` and `int`) and loops (e.g., `for`-statements and `while`-statements)
- *Standard-library components*, such as containers (e.g., `vector` and `map`) and I/O operations (e.g., `<<` and `getline()`)

The standard-library components are perfectly ordinary C++ code provided by every C++ implementation. That is, the C++ standard library can be implemented in C++ itself (and is with very minor uses of machine code for things such as thread context switching). This implies that C++ is sufficiently expressive and efficient for the most demanding systems programming tasks.

C++ is a statically typed language. That is, the type of every entity (e.g., object, value, name, and expression) must be known to the compiler at its point of use. The type of an object determines the set of operations applicable to it.

## 2.2.1 Hello, World!

The minimal C++ program is

```
int main() {}           // the minimal C++ program
```

This defines a function called `main`, which takes no arguments and does nothing (§15.4).

Curly braces, `{ }`, express grouping in C++. Here, they indicate the start and end of the function body. The double slash, `//`, begins a comment that extends to the end of the line. A comment is for the human reader; the compiler ignores comments.

Every C++ program must have exactly one global function named `main()`. The program starts by executing that function. The `int` value returned by `main()`, if any, is the program’s return value to “the system.” If no value is returned, the system will receive a value indicating successful completion. A nonzero value from `main()` indicates failure. Not every operating system and execution environment make use of that return value: Linux/Unix-based environments often do, but Windows-based environments rarely do.

Typically, a program produces some output. Here is a program that writes `Hello, World!`:

```
#include <iostream>

int main()
{
    std::cout << "Hello, World!\n";
}
```

The line `#include <iostream>` instructs the compiler to *include* the declarations of the standard stream I/O facilities as found in `iostream`. Without these declarations, the expression

```
std::cout << "Hello, World!\n"
```

would make no sense. The operator `<<` (“put to”) writes its second argument onto its first. In this case, the string literal `"Hello, World!\n"` is written onto the standard output stream `std::cout`. A string literal is a sequence of characters surrounded by double quotes. In a string literal, the backslash character `\` followed by another character denotes a single “special character.” In this case, `\n` is the newline character, so that the characters written are `Hello, World!` followed by a newline.

The `std::` specifies that the name `cout` is to be found in the standard-library namespace (§2.4.2, Chapter 14). I usually leave out the `std::` when discussing standard features; §2.4.2 shows how to make names from a namespace visible without explicit qualification.

Essentially all executable code is placed in functions and called directly or indirectly from `main()`. For example:

```
#include <iostream>
using namespace std;           // make names from std visible without std:: (§2.4.2)

double square(double x)       // square a double precision floating-point number
{
    return x*x;
}
```

```

void print_square(double x)
{
    cout << "the square of " << x << " is " << square(x) << "\n";
}

int main()
{
    print_square(1.234);    // print: the square of 1.234 is 1.52276
}

```

A “return type” **void** indicates that a function does not return a value.

## 2.2.2 Types, Variables, and Arithmetic

Every name and every expression has a type that determines the operations that may be performed on it. For example, the declaration

```
int inch;
```

specifies that **inch** is of type **int**; that is, **inch** is an integer variable.

A *declaration* is a statement that introduces a name into the program. It specifies a type for the named entity:

- 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 some type.
- A *value* is a set of bits interpreted according to a type.
- A *variable* is a named object.

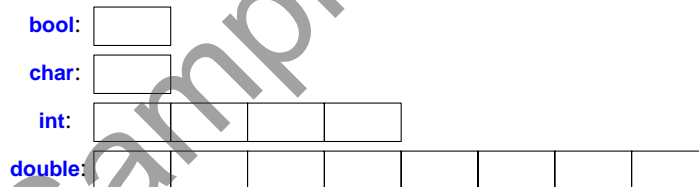
C++ offers a variety of fundamental types. For example:

```

bool    // Boolean, possible values are true and false
char    // character, for example, 'a', 'z', and '9'
int     // integer, for example, -213, 42, and 1066
double  // double-precision floating-point number, for example, 3.14 and 299793.0

```

Each fundamental type corresponds directly to hardware facilities and has a fixed size that determines the range of values that can be stored in it:



A **char** variable is of the natural size to hold a character on a given machine (typically an 8-bit byte), and the sizes of other types are quoted in multiples of the size of a **char**. The size of a type is implementation-defined (i.e., it can vary among different machines) and can be obtained by the **sizeof** operator; for example, **sizeof(char)** equals **1** and **sizeof(int)** is often **4**.

The arithmetic operators can be used for appropriate combinations of these types:

```

x+y      // plus
+x      // unary plus
x-y      // minus
-x      // unary minus
x*y      // multiply
x/y      // divide
x%y      // remainder (modulus) for integers

```

So can the comparison operators:

```

x==y     // equal
x!=y     // not equal
x<y      // less than
x>y      // greater than
x<=y     // less than or equal
x>=y     // greater than or equal

```

In assignments and in arithmetic operations, C++ performs all meaningful conversions (§10.5.3) between the basic types so that they can be mixed freely:

```

void some_function() // function that doesn't return a value
{
    double d = 2.2;    // initialize floating-point number
    int i = 7;         // initialize integer
    d = d+i;         // assign sum to d
    i = d*i;         // assign product to i (truncating the double d*i to an int)
}

```

Note that `=` is the assignment operator and `==` tests equality.

C++ offers a variety of notations for expressing initialization, such as the `=` used above, and a universal form based on curly-brace-delimited initializer lists:

```

double d1 = 2.3;      // initialize d1 with 2.3
double d2 {2.3};      // initialize d2 with 2.3

complex<double> z = 1; // a complex number with double-precision floating-point scalars
complex<double> z2 {d1,d2};
complex<double> z3 = {1,2}; // the = is optional with { ... }

vector<int> v {1,2,3,4,5,6}; // a vector of ints

```

The `=` form is traditional and dates back to C, but if in doubt, use the general `{}`-list form (§6.3.5.2). If nothing else, it saves you from conversions that lose information (narrowing conversions; §10.5):

```

int i1 = 7.2;        // i1 becomes 7 (surprise?)
int i2 {7.2};        // error: floating-point to integer conversion
int i3 = {7.2};     // error: floating-point to integer conversion (the = is redundant)

```

A constant (§2.2.3) cannot be left uninitialized and a variable should only be left uninitialized in extremely rare circumstances. Don't introduce a name until you have a suitable value for it. User-defined types (such as `string`, `vector`, `Matrix`, `Motor_controller`, and `Orc_warrior`) can be defined to be implicitly initialized (§3.2.1.1).



When defining a variable, you don't actually need to state its type explicitly when it can be deduced from the initializer:

```
auto b = true;      // a bool
auto ch = 'x';     // a char
auto i = 123;      // an int
auto d = 1.2;      // a double
auto z = sqrt(y);  // z has the type of whatever sqrt(y) returns
```

With `auto`, we use the `=` syntax because there is no type conversion involved that might cause problems (§6.3.6.2).

We use `auto` where we don't have a specific reason to mention the type explicitly. "Specific reasons" include:

- The definition is in a large scope where we want to make the type clearly visible to readers of our code.
- We want to be explicit about a variable's range or precision (e.g., `double` rather than `float`).

Using `auto`, we avoid redundancy and writing long type names. This is especially important in generic programming where the exact type of an object can be hard for the programmer to know and the type names can be quite long (§4.5.1).

In addition to the conventional arithmetic and logical operators (§10.3), C++ offers more specific operations for modifying a variable:

```
x+=y    // x = x+y
++x     // increment: x = x+1
x-=y    // x = x-y
--x     // decrement: x = x-1
x*=y    // scaling: x = x*y
x/=y    // scaling: x = x/y
x%=y    // x = x%y
```

These operators are concise, convenient, and very frequently used.

### 2.2.3 Constants

C++ supports two notions of immutability (§7.5):

- **const**: meaning roughly "I promise not to change this value" (§7.5). This is used primarily to specify interfaces, so that data can be passed to functions without fear of it being modified. The compiler enforces the promise made by `const`.
- **constexpr**: meaning roughly "to be evaluated at compile time" (§10.4). This is used primarily to specify constants, to allow placement of data in read-only memory (where it is unlikely to be corrupted), and for performance.

For example:

```
const int dmv = 17;           // dmv is a named constant
int var = 17;                // var is not a constant
constexpr double max1 = 1.4*square(dmv); // OK if square(17) is a constant expression
constexpr double max2 = 1.4*square(var); // error: var is not a constant expression
const double max3 = 1.4*square(var);    // OK, may be evaluated at run time
```

```

double sum(const vector<double>&);           // sum will not modify its argument (§2.2.5)
vector<double> v {1.2, 3.4, 4.5};         // v is not a constant
const double s1 = sum(v);                 // OK: evaluated at run time
constexpr double s2 = sum(v);            // error: sum(v) not constant expression

```

For a function to be usable in a *constant expression*, that is, in an expression that will be evaluated by the compiler, it must be defined **constexpr**. For example:

```
constexpr double square(double x) { return x*x; }
```

To be **constexpr**, a function must be rather simple: just a **return**-statement computing a value. A **constexpr** function can be used for non-constant arguments, but when that is done the result is not a constant expression. We allow a **constexpr** function to be called with non-constant-expression arguments in contexts that do not require constant expressions, so that we don't have to define essentially the same function twice: once for constant expressions and once for variables.

In a few places, constant expressions are required by language rules (e.g., array bounds (§2.2.5, §7.3), case labels (§2.2.4, §9.4.2), some template arguments (§25.2), and constants declared using **constexpr**). In other cases, compile-time evaluation is important for performance. Independently of performance issues, the notion of immutability (of an object with an unchangeable state) is an important design concern (§10.4).

## 2.2.4 Tests and Loops

C++ provides a conventional set of statements for expressing selection and looping. For example, here is a simple function that prompts the user and returns a Boolean indicating the response:

```

bool accept()
{
    cout << "Do you want to proceed (y or n)?\n"; // write question

    char answer = 0;
    cin >> answer; // read answer

    if (answer == 'y') return true;
    return false;
}

```

To match the **<<** output operator (“put to”), the **>>** operator (“get from”) is used for input; **cin** is the standard input stream. The right-hand operand of **>>** is the target of the input operation and that operand's type determines what input the **>>** accepts. The **\n** character at the end of the output string represents a newline (§2.2.1).

The example could be improved by taking an **n** (for “no”) answer into account:

```

bool accept2()
{
    cout << "Do you want to proceed (y or n)?\n"; // write question

    char answer = 0;
    cin >> answer; // read answer
}

```

```

switch (answer) {
case 'y':
    return true;
case 'n':
    return false;
default:
    cout << "I'll take that for a no.\n";
    return false;
}
}

```

A **switch**-statement tests a value against a set of constants. The case constants must be distinct, and if the value tested does not match any of them, the **default** is chosen. If no **default** is provided, no action is taken if the value doesn't match any case constant.

Few programs are written without loops. For example, we might like to give the user a few tries to produce acceptable input:

```

bool accept3()
{
    int tries = 1;
    while (tries<4) {
        cout << "Do you want to proceed (y or n)?\n"; // write question
        char answer = 0;
        cin >> answer; // read answer

        switch (answer) {
        case 'y':
            return true;
        case 'n':
            return false;
        default:
            cout << "Sorry, I don't understand that.\n";
            ++tries; // increment
        }
    }
    cout << "I'll take that for a no.\n";
    return false;
}
}

```

The **while**-statement executes until its condition becomes **false**.

### 2.2.5 Pointers, Arrays, and Loops

An array of elements of type **char** can be declared like this:

```
char v[6]; // array of 6 characters
```

Similarly, a pointer can be declared like this:

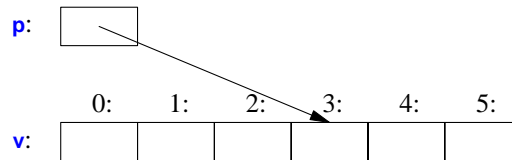
```
char* p; // pointer to character
```

In declarations, **[ ]** means “array of” and **\*** means “pointer to.” All arrays have **0** as their lower

bound, so `v` has six elements, `v[0]` to `v[5]`. The size of an array must be a constant expression (§2.2.3). A pointer variable can hold the address of an object of the appropriate type:

```
char* p = &v[3];      // p points to v's fourth element
char x = *p;         // *p is the object that p points to
```

In an expression, prefix unary `*` means “contents of” and prefix unary `&` means “address of.” We can represent the result of that initialized definition graphically:



Consider copying ten elements from one array to another:

```
void copy_fct()
{
    int v1[10] = {0,1,2,3,4,5,6,7,8,9};
    int v2[10];      // to become a copy of v1

    for (auto i=0; i!=10; ++i) // copy elements
        v2[i]=v1[i];
    // ...
}
```

This `for`-statement can be read as “set `i` to zero; while `i` is not `10`, copy the `i`th element and increment `i`.” When applied to an integer variable, the increment operator, `++`, simply adds `1`. C++ also offers a simpler `for`-statement, called a range-`for`-statement, for loops that traverse a sequence in the simplest way:

```
void print()
{
    int v[] = {0,1,2,3,4,5,6,7,8,9};

    for (auto x : v) // for each x in v
        cout << x << '\n';

    for (auto x : {10,21,32,43,54,65})
        cout << x << '\n';
    // ...
}
```

The first range-`for`-statement can be read as “for every element of `v`, from the first to the last, place a copy in `x` and print it.” Note that we don’t have to specify an array bound when we initialize it with a list. The range-`for`-statement can be used for any sequence of elements (§3.4.1).

If we didn’t want to copy the values from `v` into the variable `x`, but rather just have `x` refer to an element, we could write:

```

void increment()
{
    int v[] = {0,1,2,3,4,5,6,7,8,9};

    for (auto& x : v)
        ++x;
    // ...
}

```

In a declaration, the unary suffix **&** means “reference to.” A reference is similar to a pointer, except that you don’t need to use a prefix **\*** to access the value referred to by the reference. Also, a reference cannot be made to refer to a different object after its initialization. When used in declarations, operators (such as **&**, **\***, and **[]**) are called *declarator operators*:

```

T a[n];    // T[n]: array of n Ts (§7.3)
T* p;     // T*: pointer to T (§7.2)
T& r;     // T&: reference to T (§7.7)
T f(A);   // T(A): function taking an argument of type A returning a result of type T (§2.2.1)

```

We try to ensure that a pointer always points to an object, so that dereferencing it is valid. When we don’t have an object to point to or if we need to represent the notion of “no object available” (e.g., for an end of a list), we give the pointer the value **nullptr** (“the null pointer”). There is only one **nullptr** shared by all pointer types:

```

double* pd = nullptr;
Link<Record>* lst = nullptr; // pointer to a Link to a Record
int x = nullptr;           // error: nullptr is a pointer not an integer

```

It is often wise to check that a pointer argument that is supposed to point to something, actually points to something:

```

int count_x(char* p, char x)
    // count the number of occurrences of x in p[]
    // p is assumed to point to a zero-terminated array of char (or to nothing)
{
    if (p==nullptr) return 0;
    int count = 0;
    for (; *p!=0; ++p)
        if (*p==x)
            ++count;
    return count;
}

```

Note how we can move a pointer to point to the next element of an array using **++** and that we can leave out the initializer in a **for**-statement if we don’t need it.

The definition of **count\_x()** assumes that the **char\*** is a *C-style string*, that is, that the pointer points to a zero-terminated array of **char**.

In older code, **0** or **NULL** is typically used instead of **nullptr** (§7.2.2). However, using **nullptr** eliminates potential confusion between integers (such as **0** or **NULL**) and pointers (such as **nullptr**).

## 2.3 User-Defined Types

We call the types that can be built from the fundamental types (§2.2.2), the `const` modifier (§2.2.3), and the declarator operators (§2.2.5) *built-in types*. C++'s set of built-in types and operations is rich, but deliberately low-level. They directly and efficiently reflect the capabilities of conventional computer hardware. However, they don't provide the programmer with high-level facilities to conveniently write advanced applications. Instead, C++ augments the built-in types and operations with a sophisticated set of *abstraction mechanisms* out of which programmers can build such high-level facilities. The C++ abstraction mechanisms are primarily designed to let programmers design and implement their own types, with suitable representations and operations, and for programmers to simply and elegantly use such types. Types built out of the built-in types using C++'s abstraction mechanisms are called *user-defined types*. They are referred to as classes and enumerations. Most of this book is devoted to the design, implementation, and use of user-defined types. The rest of this chapter presents the simplest and most fundamental facilities for that. Chapter 3 is a more complete description of the abstraction mechanisms and the programming styles they support. Chapter 4 and Chapter 5 present an overview of the standard library, and since the standard library mainly consists of user-defined types, they provide examples of what can be built using the language facilities and programming techniques presented in Chapter 2 and Chapter 3.

### 2.3.1 Structures

The first step in building a new type is often to organize the elements it needs into a data structure, a `struct`:

```
struct Vector {
    int sz;           // number of elements
    double* elem;    // pointer to elements
};
```

This first version of `Vector` consists of an `int` and a `double*`.

A variable of type `Vector` can be defined like this:

```
Vector v;
```

However, by itself that is not of much use because `v`'s `elem` pointer doesn't point to anything. To be useful, we must give `v` some elements to point to. For example, we can construct a `Vector` like this:

```
void vector_init(Vector& v, int s)
{
    v.elem = new double[s]; // allocate an array of s doubles
    v.sz = s;
}
```

That is, `v`'s `elem` member gets a pointer produced by the `new` operator and `v`'s `sz` member gets the number of elements. The `&` in `Vector&` indicates that we pass `v` by non-`const` reference (§2.2.5, §7.7); that way, `vector_init()` can modify the vector passed to it.

The `new` operator allocates memory from an area called *the free store* (also known as *dynamic memory* and *heap*; §11.2).

A simple use of `Vector` looks like this:

```
double read_and_sum(int s)
    // read s integers from cin and return their sum; s is assumed to be positive
{
    Vector v;
    vector_init(v,s);           // allocate s elements for v
    for (int i=0; i!=s; ++i)
        cin>>v.elem[i];       // read into elements

    double sum = 0;
    for (int i=0; i!=s; ++i)
        sum+=v.elem[i];        // take the sum of the elements
    return sum;
}
```

There is a long way to go before our `Vector` is as elegant and flexible as the standard-library `vector`. In particular, a user of `Vector` has to know every detail of `Vector`'s representation. The rest of this chapter and the next gradually improve `Vector` as an example of language features and techniques. Chapter 4 presents the standard-library `vector`, which contains many nice improvements, and Chapter 31 presents the complete `vector` in the context of other standard-library facilities.

I use `vector` and other standard-library components as examples

- to illustrate language features and design techniques, and
- to help you learn and use the standard-library components.

Don't reinvent standard-library components, such as `vector` and `string`; use them.

We use `.` (dot) to access `struct` members through a name (and through a reference) and `->` to access `struct` members through a pointer. For example:

```
void f(Vector v, Vector& rv, Vector* pv)
{
    int i1 = v.sz;           // access through name
    int i2 = rv.sz;         // access through reference
    int i4 = pv->sz;        // access through pointer
}
```

### 2.3.2 Classes

Having the data specified separately from the operations on it has advantages, such as the ability to use the data in arbitrary ways. However, a tighter connection between the representation and the operations is needed for a user-defined type to have all the properties expected of a "real type." In particular, we often want to keep the representation inaccessible to users, so as to ease use, guarantee consistent use of the data, and allow us to later improve the representation. To do that we have to distinguish between the interface to a type (to be used by all) and its implementation (which has access to the otherwise inaccessible data). The language mechanism for that is called a *class*. A class is defined to have a set of *members*, which can be data, function, or type members. The interface is defined by the `public` members of a class, and `private` members are accessible only through that interface. For example:

```

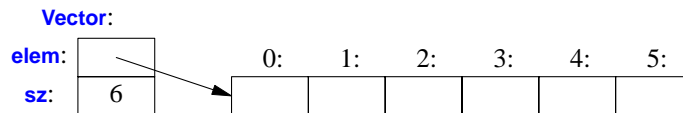
class Vector {
public:
    Vector(int s) : elem(new double[s]), sz{s} { } // construct a Vector
    double& operator[](int i) { return elem[i]; } // element access: subscripting
    int size() const { return sz; } // a "const" suffix means "can be applied to const objects" (§3.2.1.1)
private:
    double* elem; // pointer to the elements
    int sz; // the number of elements
};

```

Given that, we can define a variable of our new type **Vector**:

```
Vector v(6); // a Vector with 6 elements
```

We can illustrate a **Vector** object graphically:



Basically, the **Vector** object is a “handle” containing a pointer to the elements (**elem**) plus the number of elements (**sz**). The number of elements (6 in the example) can vary from **Vector** object to **Vector** object, and a **Vector** object can have a different number of elements at different times (§3.2.1.3). However, the **Vector** object itself is always the same size. This is the basic technique for handling varying amounts of information in C++: a fixed-size handle referring to a variable amount of data “elsewhere” (e.g., on the free store allocated by **new**; §11.2). How to design and use such objects is the main topic of Chapter 3.

Here, the representation of a **Vector** (the members **elem** and **sz**) is accessible only through the interface provided by the **public** members: **Vector()**, **operator[]()**, and **size()**. The **read\_and\_sum()** example from §2.3.1 simplifies to:

```

double read_and_sum(int s)
{
    Vector v(s); // make a vector of s elements
    for (int i=0; i!=v.size(); ++i)
        cin>>v[i]; // read into elements

    double sum = 0;
    for (int i=0; i!=v.size(); ++i)
        sum+=v[i]; // take the sum of the elements
    return sum;
}

```

A “function” with the same name as its class is called a *constructor*, that is, a function used to construct objects of a class. So, the constructor, **Vector()**, replaces **vector\_init()** from §2.3.1. Unlike an ordinary function, a constructor is guaranteed to be used to initialize objects of its class. Thus, defining a constructor eliminates the problem of uninitialized variables for a class.



`Vector(int)` defines how objects of type `Vector` are constructed. In particular, it states that it needs an integer to do that. That integer is used as the number of elements. The constructor initializes the `Vector` members using a member initializer list:

```
:elem{new double[s]}, sz{s}
```

That is, we first initialize `elem` with a pointer to `s` elements of type `double` obtained from the free store. Then, we initialize `sz` to `s`.

Access to elements is provided by a subscript function, called `operator[]`. It returns a reference to the appropriate element (a `double&`).

The `size()` function is supplied to give users the number of elements.

Obviously, error handling is completely missing, but we'll return to that in §2.4.3. Similarly, we did not provide a mechanism to “give back” the array of `double`s acquired by `new`; §3.2.1.2 shows how to use a destructor to elegantly do that.

### 2.3.3 Enumerations

In addition to classes, C++ supports a simple form of user-defined type for which we can enumerate the values:

```
enum class Color { red, blue, green };
enum class Traffic_light { green, yellow, red };
```

```
Color col = Color::red;
Traffic_light light = Traffic_light::red;
```

Note that enumerators (e.g., `red`) are in the scope of their `enum class`, so that they can be used repeatedly in different `enum classes` without confusion. For example, `Color::red` is `Color`'s `red` which is different from `Traffic_light::red`.

Enumerations are used to represent small sets of integer values. They are used to make code more readable and less error-prone than it would have been had the symbolic (and mnemonic) enumerator names not been used.

The `class` after the `enum` specifies that an enumeration is strongly typed and that its enumerators are scoped. Being separate types, `enum classes` help prevent accidental misuses of constants. In particular, we cannot mix `Traffic_light` and `Color` values:

```
Color x = red;           // error: which red?
Color y = Traffic_light::red; // error: that red is not a Color
Color z = Color::red;   // OK
```

Similarly, we cannot implicitly mix `Color` and integer values:

```
int i = Color::red;     // error: Color::red is not an int
Color c = 2;            // error: 2 is not a Color
```

If you don't want to explicitly qualify enumerator names and want enumerator values to be `ints` (without the need for an explicit conversion), you can remove the `class` from `enum class` to get a “plain” `enum` (§8.4.2).

By default, an `enum class` has only assignment, initialization, and comparisons (e.g., `==` and `<`; §2.2.2) defined. However, an enumeration is a user-defined type so we can define operators for it:

```

Traffic_light& operator++(Traffic_light& t)
    // prefix increment: ++
{
    switch (t) {
        case Traffic_light::green:    return t=Traffic_light::yellow;
        case Traffic_light::yellow:   return t=Traffic_light::red;
        case Traffic_light::red:      return t=Traffic_light::green;
    }
}

Traffic_light next = ++light;      // next becomes Traffic_light::green

```

## 2.4 Modularity

A C++ program consists of many separately developed parts, such as functions (§2.2.1, Chapter 12), user-defined types (§2.3, §3.2, Chapter 16), class hierarchies (§3.2.4, Chapter 20), and templates (§3.4, Chapter 23). The key to managing this is to clearly define the interactions among those parts. The first and most important step is to distinguish between the interface to a part and its implementation. At the language level, C++ represents interfaces by declarations. A *declaration* specifies all that's needed to use a function or a type. For example:

```

double sqrt(double);    // the square root function takes a double and returns a double

class Vector {
public:
    Vector(int s);
    double& operator[](int i);
    int size() const;    // a "const" suffix means "can be applied to const objects" (§3.2.1.1)
private:
    double* elem; // elem points to an array of sz doubles
    int sz;
};

```

The key point here is that the function bodies, the function *definitions*, are “elsewhere.” For this example, we might like for the representation of **Vector** to be “elsewhere” also, but we will deal with that later (abstract types; §3.2.2). The definition of **sqrt()** will look like this:

```

double sqrt(double d)    // definition of sqrt()
{
    // ... algorithm as found in math textbook ...
}

```

For **Vector**, we need to define all three member functions:

```

Vector::Vector(int s)    // definition of the constructor
    :elem(new double[s]), sz{s}    // initialize members
{
}

```