A Tour Of C++

Bjarne Stroustrup

March 16, 2022

Contents

1 The Basics

1.1 Introduction

The operator << ("put to") writes its second argument onto its first

A function declaration gives the name of the function, the type of the value returned (if any), and the number and types of the arguments that must be supplied in a call

If two functions are defined with the same name, but with different argument types, the compiler will choose the most appropriate function to invoke for each call.

Defining multiple functions with the same name is known as function **overloading** and is one of the essential parts of generic programming

1.2 Types, Variables and Arithemtic

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

Unfortunately, conversions that lose information, **narrowing conversions**, such as `double` to `int` and `int` to `char`, are allowed and implicitly applied when you use = (but not when you use {})

When defining a variable, you don't 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
auto bb {true};    //bbisabool
```

With `auto`, we tend to use the = because there is no potentially troublesome type conversion involved, but if you prefer to use {} initialization consistently, you can do that instead.

1.3 Scope and Lifetime

- **Local scope**: A name declared in a function or lambda is called a local name. Its scope extends from its point of declaration to the end of the block in which its declaration occurs. A **block** is delimited by a `{ }` pair. Function argument names are considered local names.
- **Class scope**: A name is called a **member name** (or a **class member name**) if it is defined in a class , outside any function , lambda, or enum class. Its scope extends from the opening `{` of its enclosing declaration to the end of that declaration.
- **Namespace scope**: A name is called a **namespace member name** if it is defined in a namespace outside any function, lambda, class, or enum class. Its scope extends from the point of declaration to the end of its namespace.

```
vector<int> vec; // vec is global
struct Record {
    string name; // name is a member of Record
// ...
};
void fct(int arg) { // fct is global (a global function)
                    // arg is local (an integer argument)
    string motto {"Who dares wins"}; // motto is local
    auto p = new Record{"Hume"};
    // p points to an unnamed Record (created by new)
    // ...
}
```

1.4 Constants

C++ supports two notions of immutability:

- `const`: meaning roughly "I promise not to change this value." This is used primarily to specify interfaces so that data can be passed to functions using pointers and references without fear of it being modified. The compiler enforces the promise made by `const`. The value of a `const` can be calculated at run time.
- `constexpr`: meaning roughly "to be evaluated at compile time." 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. The value of a `constexpr` must be calculated by the compiler.

For example

```
constexpr int dmv = 17;             // dmv is a named constant
int var = 17;                       // var is not a constant
const double sqv = sqrt(var);       // sqv is a named constant,
                                    // possibly computed at run time
double sum(const vector<double>&);// sum will not modify
                                    // its argument
vector<double> v {1.2, 3.4, 4.5}; // v is not a constant
const double s1 = sum(v);           // OK: sum(v) is evaluated at
                                    // run time
constexpr double s2 = sum(v);       // error: sum(v) is not a
                                    // 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; }
constexpr double max1 = 1.4*square(17);
// OK 1.4*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
```

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. That way, we don't have to define essentially the same function twice: once for constant expressions and once for variables.

To be `constexpr`, a function must be rather simple and cannot have side effects and can only use information passed to it as arguments. In particular, it cannot modify non-local variables, but it can have loops and use its own local variables. For example:

```
constexpr double nth(double x, int n) // assume 0<=n {
{
```

```
    double res = 1;
    int i = 0;
    while (i<n) {
        res*=x;
        ++i;
    }
    return res;
}
```

1.5 Pointers, Arrays, and References

```
char* p = &v[3];
char x = *p;
```

in an expression, prefix unary * means "contents of" and prefix unary & means "address of"

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) // add 1 to each x in 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.

References are particularly useful for specifying function arguments. For example:

```
void sort(vector<double>& v); // sort v
                              // v is a vector of doubles
```

By using a reference, we ensure that for a call `sort(vec)`, we do not copy `vec` and that it really is `vec` that is sorted and not a copy of it.

When used in declarations, operators (such as `&`, `*`, and `[]`) are called declarator operators:

```
T a[n] // T[n]: a is an array of n Ts
T* p   // T*: p is a pointer to T
T& r   // T&: r is a reference to T
T f(A) // T(A): f is a function taking an argument of type A
       // returning a result of type T
```

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
```

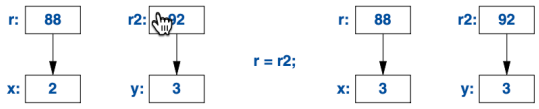
1.6 Tests

1.7 Mapping to Hardware

An assignment of a built-in type is a simple machine copy operation.

A reference and a pointer both refer/point to an object and both are represented in memory as a machine address. However, the language rules for using them differ. Assignment to a reference does not change what the reference refers to but assigns to the referenced object:

```
int x = 2;
int y = 3;
int& r = x; // r refers to x
int& r2 = y; // now r2 refers to y
r = r2; // read through r2, write through r: x becomes 3
```



2 User-Defined Types

2.1 Introduction

Types built out of other types using C++'s abstraction mechanisms are called **user-defined types**. They are referred to as **classes** and **enumerations**.

2.2 Structures

The `new` operator allocates memory from an area called the **free store** (also known as **dynamic memory** and **heap**). Objects allocated on the free store are independent of the scope from which they are created and "live" until they are destroyed using the `delete` operator

2.3 Classes

The language mechanism for that is called a **class**. A class has 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.

```
class Vector {
    public:
        Vector(int s) :elem{new double[s]}, sz{s} { }
        double& operator[](int i) { return elem[i]; }
        int size() { return sz; }
    private:
        double* elem; // pointer to the elements
        int sz; // the number of elements
};
```

`Vector(int)` defines how objects of type `Vector` are constructed. 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&` allowing both reading and writing)

There is no fundamental difference between a `struct` and a `class`; a `struct` is simply a class with members `public` by default.

2.4 Unions

A `union` is a `struct` in which all members are allocated at the same address so that the `union` occupies only as much space as its largest member. Naturally, a `union` can hold a value for only one member at a time.

```
union Value {
    Node* p;
    int i;
};
```

The language doesn't keep track of which kind of value is held by a union, so the programmer must do that:

```
enum Type { ptr, num }; // a Type can hold values ptr and num

struct Entry {
    string name;
    Type t;
    Value v; // use v.p if t==ptr; use v.i if t==num
};

void f(Entry* pe) {
    if (pe->t == num)
        cout << pe->v.i;
    // ...
}
```

Maintaining the correspondence between a **type field** (here, `t`) and the type held in a `union` is error-prone.

The standard library type, `variant`, can be used to eliminate most direct uses of unions. A `variant` stores a value of one of a set of alternative types.

```
struct Entry {
    string name;
    variant<Node*,int> v;
};

void f(Entry* pe) {
if (holds_alternative<int>(pe->v))
    // does *pe hold an int?
    cout << get<int>(pe->v);
```

```
    // get the int
    // ...
}
```

For many uses, a `variant` is simpler and safer to use than a `union`

2.5 Enumerations

```
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.

```
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; // initialization error: 2 is not a Color
```

By default, an `enum class` has only assignment, initialization, and comparisons. 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
```

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`. The enumerators from a "plain" `enum` are entered into the same scope as the name of their enum and implicitly converts to their integer value

```
enum Color { red, green, blue };
int col = green;
```

Here `col` gets the value 1. By default, the integer values of enumerators start with `0` and increase by one for each additional enumerator.

3 Modularity

3.1 Introduction

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();
    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"

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

```
Vector::Vector(int s) // definition of the constructor
    :elem{new double[s]}, sz{s}
     // initialize members
{
}
double& Vector::operator[](int i) {
    // definition of subscripting
    return elem[i];
}
int Vector::size() {
    // definition of size()
    return sz;
}
```

3.2 Separate Compilation

C++ supports a notion of separate compilation where user code sees only declarations of the types and functions used. The definitions of those types and functions are in separate source files and are compiled separately.

This can be used to organize a program into a set of semi-independent code fragments. Such separation can be used to minimize compilation times and to strictly enforce sepa- ration of logically distinct parts of a program (thus minimizing the chance of errors). A library is often a collection of separately compiled code fragments (e.g., functions).

Typically, we place the declarations that specify the interface to a module in a file with a name indicating its intended use. Example:

```
// Vector.h:
class Vector {
    public:
        Vector(int s);
        double& operator[](int i); int size();
```

```
    private:
        double* elem;
        int sz;
};
```

This declaration would be placed in a file `Vector.h`. Users then **include** that file, called a **header file**, to access that interface. For example:

```
// user.cpp:
#include "Vector.h" // get Vector's interface
#include <cmath> // get the standard-library
                 // math function interface including sqrt()
double sqrt_sum(Vector& v)
{
    double sum = 0;
    for (int i=0; i!=v.size(); ++i)
        sum+=std::sqrt(v[i]);
    return sum;
}
```

To help the compiler ensure consistency, the `.cpp` file providing the implementation of `Vector` will also include the .h file providing its interface:

```
// Vector.cpp:
#include "Vector.h" // get Vector's interface

Vector::Vector(int s)
    :elem{new double[s]}, sz{s}
{
}
double& Vector::operator[](int i)
{
    return elem[i];
}
int Vector::size()
{
    return sz;
}
```

The code in `user.cpp` and `Vector.cpp` shares the `Vector` interface information presented in `Vector.h`, but the two files are otherwise independent and can be separately compiled.

A `.cpp` file that is compiled by itself (including the h files it `#includes`) is called a **translation unit**. A program can consist of many thousand translation units.

3.3 Modules (C++20)

The use of `#includes` is a very old, error-prone, and rather expensive way of composing programs out of parts. If you `#include header.h` in 101 translation units, the text of `header.h` will be processed by the compiler 101 times. If you `#include header1.h` before `header2.h` the declarations and macros in `header1.h` might affect the meaning of the code in `header2.h`. If instead you `#include header2.h` before `header1.h`, it is `header2.h` that might affect the code in `header1.h`. Obviously, this is not ideal, and in fact it has been a major source of cost and bugs since 1972 when this mechanism was first introduced into C.

Consider how to express the `Vector` and `sqrt_sum()` example from §3.2 using `modules`:

```
// file Vector.cpp:
module; // this compilation will define a module
// ... here we put stuff that Vector might
// need for its implementation ...
export module Vector; // defining the module called "Vector"

export class Vector {
    public:
        Vector(int s);
        double& operator[](int i); int size();
    private:
        double* elem; // elem points to an array of sz doubles
        int sz;
};

Vector::Vector(int s)
:elem{new double[s]}, sz{s}
{
}

double& Vector::operator[](int i)
{
```

```
return elem[i];
}

int Vector::size()
{
return sz;
}

export int size(const Vector& v) { return v.size(); }
```

This defines a module called `Vector`, which exports the class Vector, all its member functions, and the non-member function `size()`

The way we use this module is to `import` it where we need it. For example:.

```
// file user.cpp:
//
import Vector; // get Vector's interface
#include <cmath>

double sqrt_sum(Vector& v)
{
    double sum = 0;
    for (int i=0; i!=v.size(); ++i)
        sum+=std::sqrt(v[i]);
    return sum;
}
```

The differences between headers and modules are not just syntactic. • A module is compiled once only (rather than in each translation unit in which it is used). • Two modules can be `imported` in either order without changing their meaning. • If you import something into a module, users of your module do not implicitly gain access to (and are not bothered by) what you imported: `import` is not transitive.

3.4 Namespaces

C++ offers **namespaces** as a mechanism for expressing that some declarations belong together and that their names shouldn't clash with other names

```
namespace My_code {
    class complex {
```

```
        // ...
    };
    complex sqrt(complex);
    // ...
    int main();
}

int My_code::main()
{
    complex z {1,2};
    auto z2 = sqrt(z);
    std::cout << '{' << z2.real() << ',' << z2.imag() << "}\n";
    // ...
}

int main()
{
    return My_code::main();
}
```

By putting my code into the namespace `My_code`, I make sure that my names do not conflict with the standard-library names in namespace `std`

If repeatedly qualifying a name becomes tedious or distracting, we can bring the name into a scope with a `using`-declaration:

```
void my_code(vector<int>& x, vector<int>& y)
{
    using std::swap; // ...
    swap(x,y);
    other::swap(x,y); // ...
}
```

To gain access to all names in the standard-library namespace, we can use a `using`-directive:

```
using namespace std;
```

3.5 Error Handling

3.5.1 Exceptions

Consider again the `Vector` example.

Assuming that out-of-range access is a kind of error that we want to recover from, the solution is for the `Vector` implementer to detect the attempted out-of-range access and tell the user about it. The user can then take appropriate action. For example, `Vector::operator[]()` can detect an attempted out-of-range access and throw an `out_of_range` exception:

```
double& Vector::operator[](int i)
{
    if (i<0 || size()<=i)
        throw out_of_range{"Vector::operator[]"};
    return elem[i];
}
```

The `throw` transfers control to a handler for exceptions of type `out_of_range` in some function that directly or indirectly called `Vector::operator[]()`. To do that, the implementation will **unwind** the function call stack as needed to get back the context of that caller. That is, the exception handling mechanism will exit scopes and functions as needed to get back to a caller that has expressed interest in handling that kind of exception, invoking destructors (§4.2.2) along the way as needed. For example:

```
void f(Vector& v) {
// ...
    try { // exceptions here are handled by
          // the handler defined below
        v[v.size()] = 7; // try to access beyond the end of v
    }
    catch (out_of_range& err) {
    // ... handle range error ...
        cerr << err.what() << '\n';
    }
    // ...
}
```

We put code for which we are interested in handling exceptions into a `try`-block. The attempted assignment to `v[v.size()]` will fail. Therefore, the `catch`-clause providing a handler for exceptions of type `out_of_range` will be entered. The `out_of_range` type is defined in the standard library (in `<stdexcept>`) and is in fact used by some standard-library container access functions.

The main technique for making error handling simple and systematic (called **Resource Acquisition Is Initialization**; RAII) is explained in §4.2.2.

The basic idea behind RAII is for a constructor to acquire all resources necessary for a class to operate and have the destructor release all resources, thus making resource release guaranteed and implicit.

A function that should never throw an exception can be declared `noexcept`. For example:

```
void user(int sz) noexcept {
    Vector v(sz);
    iota(&v[0],&v[sz],1); // fill v with 1,2,3,4...
    // ...
}
```