A Tour of C++

ksolomon

June 2, 2024

Abstract

Provide an overview on how c++ language is structured and managed in backend. Introduce core language features and concepts on a high-level.
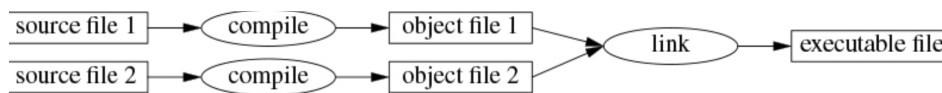
1 The Basics

1.1 Introduction

Figure 1: Build Pipeline

1.2 Programs

ISO standard has 2 types of entities:

- Core Language Features (`char, int, for, while,` etc.)

- STL Components (`vector, map, cout, getline,` etc.)

The STL (Standard Template Library) is (mostly) implemented in C++, providing things like containers, algorithms, etc. which make C++ far more capable as a modern programming language than its predecessor C. Every C++ program must have exactly one global function: `main() {}`. The return value is treated as the error code. Returning void (nothing) or 0 are treated as a pass. See an example 'Hello World' program below:

```cpp
import std; // new to C++20 (instead of #include <iostream>)
int main()
{
    std::cout << "Hello World!\n";
    return 0;
}
```

Alternatively, if you use the namespace you don't have to explicitly define which namespaces are in scope. However, this basically imports that entire namespace into this translation unit, so 'using' namespaces can cause bloat to binary size.

```cpp
import std; // new to C++20 (instead of #include <iostream>)
using namespace std;
int main()
{
    cout << "Hello World!\n";
    return 0;
}
```

1.3 Functions

Declaration: `returnType functionName(argType);`
Defintion: `returnType functionName(argType argName){<implementation>;}`
While declarations may supply argName, that will be ignored by the compiler, still requiring the function definition to supply the argName. Functions can be standalone, or member functions as part of a larger class. The primary goal of the 'function' is to break code into logical blocks. Appropriate usage of functions make code far more readable and maintainable.
In C++ the compiler can handle something called 'function overloading'. This is where you have multiple permutations/implementations of the same function which take different inputs. With this the user can simply call the function with whatever input types are implemented and the compiler will determine which is the appropriate implementation to use.

```cpp
void print(int);
void print(double);
void print(string);
//...
// user code each calls appropriate print() function
print(1);
print(2.0);
print('3');
```

1.4 Types, Variables, Arithmetic

Declaration a statement that introduces an entity into the program and specifies its type.

Type defines a set of possible values and a set of possible operations (for given object).

Object memory that holds a value of some defined type.

Value a collection of Bits interpreted according to its type.

Variable a named object.

When performing operations between two different types the result is stored in the type with highest precision (most bits). e.g.:

```cpp
double d = 1.0;
int i = 2;
auto res = d + i; // res is a double type (3.0)
```

When there are multiple operations on one line the order of operations is as follows (left is hi priority):
`x.y, x->y, x(y), x[y], x<<y, x>>y, x&&y, x||y`
  There are multiple ways to initialize variables. All of said methods involve either = or . See below for examples:

```cpp
double d1 = 2.3;
double d2 {2.3};
double d3 = {2.3};
complex<double> z1 = 1; // complex number with double precision scalar input
complex<double> z2 {d1,d2}; // complex number with variable input
complex<double> z3 = {d1,d2}; // = is optional, but maintained for c compatibility
vector<int> v {1,2,3,4,5};
```

One main reason to avoid equal (=) assignment is that it can implicitly perform narrowing conversions (which lose information). Examples include int-char and double-int. However, = is preferred when using the `auto` type. `auto` allows the compiler to deduce the type given initial value form at compile time e.g.:

```cpp
auto b = true; // bool
auto ch = x; // char
auto i = 123; // int
auto d = 1.2; // double
auto z = sqrt(y); // whatever sqrt returns (could be complex<double>)
auto bb {true}; // bool
```

`auto` is generally preferred unless:

  • the variable has a large scope

- the type of initializer is non-obvious

- we want to explicitly define the precision (e.g. `double` vs `float`)

This allows the compiler to perform optimization more effectively and can make code more readable by limiting usage of cumbersome typenames.

1.5 Scope

Scope lifetime of the variable/entity, defined by the declaration

Local declared in function or lambda. Scope is until the end of the block (defined by)

Class a.k.a. "member" scope. Var is declared within a class, but outside any enum/function/lamba. Scope is within the class open/close

Namespace Defined in a namespace, but outside any class/function/enum/lambda. Scope is anywhere that namespace is used.

Global Defined outside any of above. Available in any translation unit.

An object must be initialized (constructed) before use and it will be destroyed (destructed) at the end of its scope. Attempts to use destructed objects could result in undefined behavior such as use-after-free, stack overflow, etc. Objects can also be created/destroyed via `malloc()` on the heap by use of `new/delete`.

1.6 Constants

const value cannot be changed once set. Can be evaluated at either compile time or runtime.

constexpr function or variable evaluated at compile time. This allows placement into read-only memory (ROM). Exception is that constexpr functions can be called at runtime, but do not generate constexpr output in that case.

consteval a function that can only be used at compile-time. Always generates constexpr output.

```
double sum(const vector<double>&); // can read but not modify input argument
const double s1 = sum(v); // legal. s1 can be calculated at runtime, but not changed
    thereafter.
constexpr double s2 = sum(v); // ILLEGAL. sum(v) is not a constant expression/cannot
    be read-only at compile time.
```

For a function to be used in a constant expression (i.e. evaluated at compile-time) it must be defined as `constexpr/consteval`. These functions CANNOT have side-effects, meaning they may only modify local variables.

1.7 Pointers, Arrays, Reference

Array (v) collection of data in a contiguously allocated sequence of elements of the same type (0-indexed)

Pointer (p) object that points to another object

nullptr: value of pointers when there is no object available. Attempts to dereference will result in nullptr exception or use after free errors. This is recommended over using the C-style `NULL`, which just aliases 0, and can be confused with an integer.
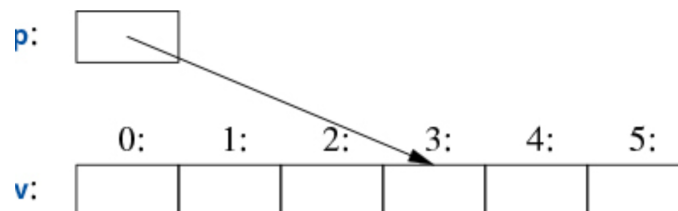

Figure 2: char* p = &v[3]

```
char v[6]; // array of 6 characters
char* p; // ptr to character
char* p = &v[3]; // p points to v's 4th element
char x = *p; // x is the character that p points to (aka same as v[3])
```

For loops are a typical way to iterate through arrays:

```
int v[] = {0,1,2,3,4,5,6,7,8,9};
for(auto i = 0; i < 10; i ++){;}
for(auto x : v){;}
for(auto x : {'a','b','c'}){;}
for(auto& x : v){++x;} // since this give a reference to x you can modify in-place
```

References are especially useful for function arguments since it allows the function to mutate the argument without copying the whole of the argument(s) onto the stack. If we dont want to modify the contents then use const modifier

```
double sum(const vector<double>&){}
```

1.8 Tests (Conditionals)

These check to see if values are some form of equivalent or less than, greater than one another. Common structures include `if()` statements and `switch/case` statements.

```
if(a < b){}
else if(a > b){} // where there is an elseif statement it is good practice to always
    have else case
else {}

switch(ch) {
    case 'a':
        // if ch == 'a' it will trigger code here
    case 'b':
        // 'a' case falls-down and also runs code here since there was no 'break'
            after case 'a'
        // if ch == 'b' then only this section is run, above code in case 'a' section
            is not run.
        break;
    case 'c':
        // if ch == 'c' it will trigger code here
        break;
    default:
        // if ch == anything but 'a'-'c' it will trigger code in the default case
}
```

1.9 Advice for Getting Started

1. Dont panic, you don't need to know all the details to write good code.

2. Don't use built-in features exclusively, STL is powerful.

3. Import libraries when possible.

2 The Basics

Built-In Types: types which can be built entirely from the fundamental types, the `const` modifier, and declarator operators.

2.1 Struct

struct organized/grouped set of variable types that collectively make some larger concept.

```
    // Defintion
    struct Vector {
        double* elem;
        int sz;
    }

    // Usage
    Vector v;
    int size = 10;
    v.elem = new double[size];
    if(v.elem != nullptr) // this would happen if there is not room in memory to allocate
        10 elements
        v.sz = size;
```

2.2 Classes

Class: similar to a struct in that it organizes data and represents some greater entity/concept than just a variable, however a struct has members public by default. A class can hide private implementation. Constructors/Destructors are also a concept unique to classes.

Vector: a handle containing a ptr to elements and the number of elements (sz)

```
class Vector {
  public:
    Vector(int s) :elem{new double[s]}, sz{s} {} // Constructor is same name as class
        with no return type
    double& operator[](int i) { return elem[i];} // Operator overloading for
        dereferencing *elem;
    int size() {return sz;} // access function to read (but not have write-access to)
        private member variable
  private:
    double* elem;
    int sz;
}
```
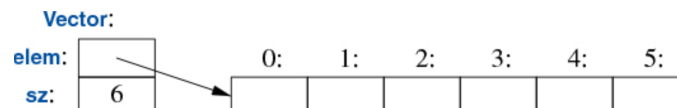

Figure 3: Simple representation of a Vector

In our example vector class we use an initializer list constructor :elem{new double[s]}, sz{s}. This defines elem with a pointer to s elements of type double. Then we initialize sz to s.

2.3 Enum

Enum: User-defined types that have a fixed set of possible values (integer only).

By default enums have assignment, initialization, and comparison operators.

```
// OK
enum class Color {RED, GREEN, BLUE};
enum class TrafficLight {RED, YELLOW, GREEN};
enum names {BETHANY, TAYLOR, KARL}; // C-Style enum is just an integer wrapper
Color col = Color::RED;
TrafficLight tl = TrafficLight::RED;

int i = Color::RED; // ERROR: RED is not an int.
int i = int(Color::RED); // Legal type cast
```

5

```
int i = names::BETHANY; // Legal C-style
Color c = 2; // ERROR: 2 is not a Color
names n = 1; // Legal: C-style maps to names::TAYLOR
Color c {2}; // Legal
```

2.4 Union

Union: All members are allocated at the same memory address.

Variant: stores a value of one of a set of alternative types.

Unions only occupy as much memory as the largest member. Unions are typically used to implement a single variable that can contain multiple different types of data. This way, depending on the context, the variable can be viewed/manipulated differently without dealing with messy type casting. `Variant` types were introduced in C++17. These can effectively replace unions in many cases, and are the preferred choice in C++17 and later.

```
struct Entry {
    string name;
    variant<Node*, int> v; // can represent the variable as EITHER Node* OR int.
}
void f(Entry* pe) {
    if(holds_alternative<int>(pe->v)) {
        cout << get<int>(pe->v) << endl;
    }
}
```

3 Modularity

3.1 Introduction

Crutial for modular code to separate interface from implementation.

Declaration: interface - everything that is needed to use a function.

Definition: implementation - how the function achieves promised interface.

There can be multiple declarations, but only one definition per entity.

3.2 Separate Compilation

In order to achieve modular code it is important to separate code into logical units. Each .c/.cpp file that contains implementaion is compiled separately into its own 'translation unit'. These translation units are linked together with the linker after all of the units are compiled. In order for source code to reference code in a different translation unit the source must either include the header file which defines the interface, or it must import the module which defines a library containing the target code. While the header file route is most popular currently, with the addition of import functionality in C++20 import is now preferred.
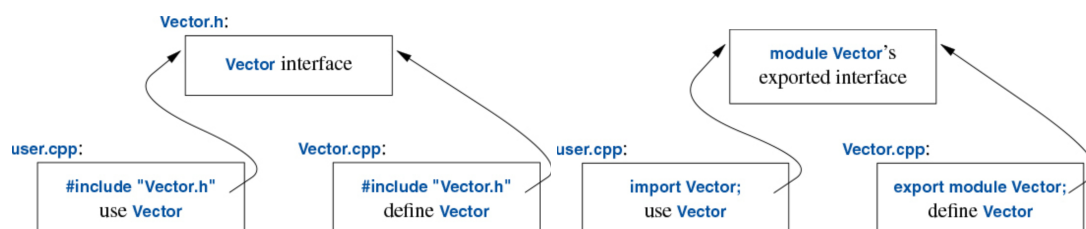

Figure 4: Graphical header vs module reference flow.

Headers The best mental-model is to consider a program as a set of modules, each of which is a translation unit responsible for its own concept. One major disadvantage of the C-style modules using include directives is that each include statement

completely injects/copies the contents of said header file into the translation unit (source/.c/.cpp) file. This means that if N files import x.h then x.h's text is copied into N files. This can create serious bloat and increase compile time. Another disadvantage is that the order of includes matters in the source file. Consider the following legitimate example:

```
#include 'a.h'
#include 'b.h'
doStuff(); // Compiler Error - Object not found

// vs

#include 'b.h'
#include 'a.h'
doStuff(); // Pass
```

Modules Consider the below usage of import. Here we show that we can both import and #include in the same file. The main advantage of using import over include is that we can avoid excessive code copying from include. Only the units which are 'exported' will be included in the translation unit. Additionally, the compiler will check for the existence of the module and prevent redundant imports. This should considerably improve maintainability and compile times. This can prevent unnecessary complication of source code into .hpp and .cpp files since we can just selectively export specific entities in the .cpp.

```
export module Vector;
export class Vector{
    public:
    Vector(int s);
    double& operator[](int i);
    int size();
    private:

}
```

3.3 Namespaces

Namespaces allow you to use class/function/variable names that are present in different namespaces differently. E.g. you can create your own implementation of C++ STL libraries (e.g. vector, printf, etc.). The simplest way to access an entity in a different namespace is to use the :: operator (e.g. std::vector). To gain access to all names in any namespace use using namespace <namespace>;. This is has consequences if used carelessly since it can use names that you may not be aware of and intend to build yourself. These are primarily used to organize libraries.

3.4 Arguments

Key concerns are:

- pass by value or reference

- const or mutable

- side-effects (e.g. if an object is moved then previous reference is invalid).

Default behavior is pass-by-value. However, compiler optimization can implicitly modify to pass-by-reference. It is generally faster to pass-by-value when the sizeof(object) < 4*sizeof(void*). Use const modifier if we want to pass by reference, but don't need to actually modify the value

```
// Doesn't actually do anything. v is passed by value so any modification will
// be lost when returned to caller context
void sumTwoVectors(vector<int> v, const vector<int>& crv) {
    auto i_v v.begin();
    auto i_crv = crv.begin();
    while (i_v != v.end() && i_crv != crv.end()) {
        (*i_v) += (*i_crv);
        i_v++;
        i_crv++;
```

```
    }
}

// adds contents of crv vector to rv vector. rv is modified in-place so we
// pass-by-reference. crv is not modified, but could very well be large so we
// pass by const reference.
void sumTwoVectors(vector<int>& rv, const vector<int>& crv) {
    auto i_v v.begin();
    auto i_rv = rv.begin();
    auto i_crv = crv.begin();
    while (i_v != v.end() && i_crv != crv.end()) {
        (*i_v) += (*i_crv);
        i_v++;
        i_crv++;
    }
    i_crv = crv.begin();
    while (i_rv != v.end() && i_crv != crv.end()) {
        (*i_rv) += (*i_crv);
        i_rv++;
        i_crv++;
    }
}
```

It is common for arguments to have a default value. An example of such usage can be seen here:
```
void print(int value, int base = 10); // 10 is default for base
```

Return by reference only when returning something that is not local to the caller. Anything local to the caller will be destroyed when the function returns. Most compilers should throw this error at compile-time. If a large object is created inside a function and needs to be returned to the caller it can be achieved by generating a move constructor for the object. It is *possible* that even without explicitly defining a move constructor that the compiler can optimize away the copy and implicitly perform a move. This is called *copy elision*. See below for an example in modern copy elision in practice:

```
// Example with Copy Elision (implicit compiler-managed move constructor)
Matrix operator+(const Matrix &x, const Matrix &y) {
    Matrix res;
    // for all i, j: res[i][j] = x[i][j] + y[i][j];
    return res;
}
// user-code
Matrix m1, m2;
Matrix m3 = m1 + m2;

// pre-C++11 style (BAD, DON'T DO)
Matrix *add(const Matrix &x, const Matrix &y) {
    Matrix *ptr = new Matrix;
    // for all i, j: res[i][j] = x[i][j] + y[i][j];
    return ptr;
}
// user-code
// Matrix m1, m2;
Matrix *m3 = add(m1, m2);
//... do stuff
delete m3;  // VERY easy to forget. Often leads to memory leaks.
```

Deduction of return type is possible by returning auto. This is especially valuable for generics and lambdas. It is possible to display the return type after the function handle and arguments like so:

```
auto sqrt(double); // deduction
auto sqrt(double) -> double; // suffix-return type
```

4   Errors

5 Classes

5.1 Concrete Types

Behave just like built-in types e.g. complex numbers and infinite-precision integers. Representation is part of its definition. Typically these are used for types which don't change often and depend on private members. Arithmetic types are a classic example of concrete types, for example complex numbers. Upside:

- Easy to place on stack/heap and in other objects

- Can refer to objects directly, not just via reference

- Initialize objects immediately and completely in constructor

- Copy/move easily

Downside:

- Any change causes recompilation.

Container: object holding a collection of elements.

Destructor: the complement of the constructor. Responsible for destroying the object and freeing (usually with `delete[]`) memory to prevent memory leaks. Represented with a tilde prepending the class name.

RAII: Resource Acquisition Is Initialization. A concept that is used to ensure that objects are properly initialized and destroyed, while preventing user from having to concern with memory management (and thus eliminate "naked" `new` operations.

Initializer List: construct a container with a list of starting values in a provided list. Added in C++11, this abstracts the very common case where the user would simply iteratively call `push_back()` to achieve the same thing.

Type Casts

- `static_cast`:

- `reinterpret_cast`:

- `bit_cast`:

- `const_cast`:

5.2 Abstract Types

5.3 Virtual Functions

5.4 Class Hierarchies

References

[1] . *A Tour of C++: 3rd Edition*. Bjarne Stroustrup, 2022.