notes of “tour of c++”

Hobo Chen

March 9, 2016

Abstract

The license of this document is CC-BY-NC-ND.

If you think this PDF helps a lot, **please** donate some money to the kids live in west and middle china. They can not afford to go to school, letting alone to learn C++.

If you want to contact me for any questions or suggestions, please send a email to hobochen96@gmail.com .

Contents

Chap 1 Basics

Zen

Basic mechanisms for organizing code into a program are most often seen in C, sometimes called *procedural programming*.

C++ is `compiled`, `statically typed`.

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

- Core language features
- Standart lib components

an instance of C++ program

```
#include <bits/stdc++.h>     // include or "import" all the c++ lib,
                             // supported by clang and gcc

using namespace std;         // make names from std visible without std::

int main() {                 // essential

    return 0;                // unix and linux based os often uses that
                             // return value, but windows do not
}
```

FUNCTION OVERLOAD

If two functions can be called, but neither is better than other, the call is deemed ambiguous and the compiler gives an `error`. For example:

```
void print(double, int)
void print(int, double)
```

SIZEOF

```
int a;
char b;
cout << sizeof(a) << " " << sizeof(b) << endl;
    // sizeof returns Byte, not bit; so it is 4 1
```

INI

Using `{}` to initialize is suggested, `=` is the traditional way of C.

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 sqr t(y) returns
```

AUTO

We use auto where we do *not* have a specific reason to mention the type explicitly.

Specific reasons:

- 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 is to 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.

SCOPE

A declaration introduces its name into a scope:

- 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 namespaces.

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 without fear of it being modified. The compiler enforces the promise made by const.
- 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.

Also `constexpr` could be a kind of return value of a function, and the paras could be mutable.

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

RANGE `FOR` STATEMENT

```
int x[] = {1,2,3,4,5};
    // no need of size of array when ini it with a list
for (auto a : v} {
    cout << a << endl;
}
```

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 hav e to specify an array bound when we initialize it with a list. The range-for-statement can be used for any sequence of elements. If do not want copying, use `auto &a : v` instead.

REFERENCE

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. That means, it can not be computed, such as normal use of pointer `*(pos++)`.

NULLPTR

Trying to ensure that a pointer always points to an element so that dereferencing it is valid, so there is the `nullptr`. There is only one `nullptr` shared by all pointer types.

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

example.

```
int charcount(char* p, char x) {
    if (p == nullptr) return 0;
    int cnt = 0;
    while (p != nullptr) {
        if ( *p == x ) cnt++;
    }
    return cnt;
}
```

Chap 2 User-Defined Types

We call the types that can be built from the fundamental types, the `const` modifier, and the declarator operators built-in types.

CLASSES AND STRUCTS

default of members of classes is private, which is public in struct.

UNION

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.

```
enum Type { double, int };
struct hobo {
    string name;
    Type t;
    double d1;
    int t2;
};
    // d1, t2 can never be used at the same time
    // so may use a union,
union Value {
    double d1;
    int t2;
};
    // c++ doesn't keep track of which kind of value is held
struct hobo {
    string name;
    Type t;
    Value v; // depends on t == double, or t == int
};
```

Maintaining the correspondence between a type field (here, `t`) and the type held in a `union` is errorprone.

At the application level, abstractions relying on such *tagged unions* sare common and useful, but use of **naked** unions is best minimized.

EUNM

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

```
enum class Traffic_light {red, yellow, red};
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;
```

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.(begins at 0)

Chap 3 Modularity

EXCEPTION

```
try {
    // try sth.
}
catch (_exception) {
    // handling
}
```

RETHROW EXCEPTION

```
void test()
{
    try {
        Vector v(-27);
    }
    catch (std::length_error) {
        cout << "test failed: length error\n";
        throw; // rethrow
    }
    catch (std::bad_alloc) {
        // Ouch! test() is not designed to handle memory exhaustion
        std::terminate(); // terminate the program
    }
}
```

STATIC_ASSERT

Exceptions report errors found at run time. If an error can be found at compile time, it is usually preferable to do so. In general, `static_assert(A,S)` prints `S` as a compiler error message if `A` is not true.

The most important uses of `static_assert` come when we make assertions about types used as parameters in generic programming. For runtime-checked assertions, use exceptions.

Chap 4 Classes

STD::INITIALIZER_LIST

The std::initializer_list used to define the initializer-list constructor is a standard-library type known to the compiler.

PURE `VIRTUAL` FUNCTIONS

```
class Container {
public:
    virtual double& operator[] (int) = 0;
    virtual int size() const = 0;
    virtual Container() {}
};
```

The curious `=0` syntax says the function is *pure virtual*; that is, some class derived from `Container` *must* define the function.

ABSTRACT TYPE

A class with a pure virtual function is called an abstract class.

VIRTUAL FUNCTIONS

When `h()` calls `use()` , List_container's `operator[]()` must be called. When `g()` calls `use()`, Vector_container 's `operator[]()` must be called.

To achieve this resolution, a Container object must contain information to allow it to select the right function to call at run time. The usual implementation technique is for the compiler to convert the name of a virtual function into an index into a table of pointers to functions.

That table is usually called the **`virtual function table`** or simply the `vtbl`.

EXPLICIT OVERIDING

```
void draw() override;
```

If has not overridden something, you will get a compile error.

BENEFITS OF INHERITANCE

- interface inheritance
- implementation inheritance

unique_ptr

See in `Utilities`.

COPY AND MOVE

When a class is a resource handle – that is, when the class is responsible for an object accessed through a pointer – the default memberwise copy is typically a *disaster*.

Copying of an object of a class is defined by two members: a copy constructor and a copy assignment.

```
class Vector {
    private:
        double* elem;
            // elem points to an array of sz doubles
        int sz;
    public:
        Vector(int s);
            // constructor: establish invariant, acquire resources
        Vector() { delete[] elem; }
            // destructor: release resources
        Vector(const Vector& a);
            // copy constructor
        Vector& operator=  (const Vector& a);
            // copy assignment
        double& operator [](int i);
        const double& operator [](int i) const;
        int size() const;
};

Vector::Vector(const Vector& a) : elem{new double[a.sz]}, sz{a.sz} {
    // allocate space for elements
    for (int i = 0; i != sz; ++i) // copy elements
        elem[i] = a.elem[i];
} // copy constructor

Vector& Vector::operator = (const Vector& a) {
    // copy assignment
    double* p = new double[a.sz];
    for (int i = 0; i != a.sz; ++i)
        p[i] = a.elem[i];
    delete[] elem; // delete old elements
    elem = p;
    sz = a.sz;
    return *this;
}
```

The name this is predefined in a member function and points to the object for which the member function is called.

MOVE

```
Vector operator + (const Vector& a, const Vector& b) {
    if (a.size() != b.size())
        throw Vector_size_mismatch{};
    Vector res(a.size());
    for (int i=0; i != a.size(); ++i)
        res[i] = a[i] + b[i];
    return res;
}

void f(const Vector& x, const Vector& y, const Vector& z) {
    Vector r;
    r = x + y + z;
        // That would be copying a Vector at least twice
}

class Vector {
    Vector(const Vector& a);                    // copy constructor
    Vector& operator = (const Vector& a);       // copy assignment
    Vector(Vector&& a);                         // move constructor
    Vector& operator = (Vector&& a);           // move assignment
};
Vector::Vector(Vector&& a) : elem{a.elem}, sz{a.sz} {
    // "get" the element from a
    a.elem = nullptr;
        // now has no element
    a.sz = 0;
}
```

The `&&` means **rvalue reference** and is a reference to which we can bind an rvalue. An rvalue is – to a first approximation – a value that can't be assigned to, such as an integer returned by a function call.

A move operation is applied when an rvalue reference is used as an initializer or as the right-hand side of an assignment.

After a move, a moved-from object should be in a state that allows a destructor to be run. Typically, we should also allow assignment to a moved-from object.

The standard-library function `move()`returns doesn't actually move anything. Instead, it returns a reference to its argument from which we may move – an *rvalue reference.*

essential operations, There are five situations in which an object is copied or moved:

- As the source of an assignment
- As an object initializer
- As a function argument
- As a function return value
- As an exception

```
class Y {
    Public:
        Y(Sometype);
        Y(const Y&) = default;
            // do allow auto-generating the contructor
        Y(Y&&) = default;
            // do allow auto-generating the contructor as well
};
```

`EXPLICIT`

```
struct vec {
private:
    int sz;
    double* elem;
public:
    vec() { }
    vec(int s) : sz{s}, elem{new double[s]} {
        for (int i = 0; i != s; i++)
            elem[i] = 0;
    }
};

int main() {
    vec hobo = 7;    // hobo has 7 elements
    return 0;
}
```

This is typically considered **unfortunate**, and the standard-library vector does not allow this int-to-vector *conversion.*

`explicit` should be used.

```
    explicit vec(int s) : sz{s}, elem{new double[s]} {
        for (int i = 0; i != s; i++)
```

```
            elem[i] = 0;
    }

    vec hobox(7);    // hobox has 7 elements
    vec hobo = 7;    // ce
```

Using `explicit` everytime if the constructor has only one parameter, unless there is some other good reason.

SUPPRESSING OPERATIONS

```
class Shape {
    Shape(const Shape&) = delete;                    // no copy operations
    Shape& operator = (const Shape&) = delete;
    Shape(Shape&&) = delete;                         // no move operations
    Shape& operator = (Shape&&) = delete;
};
```

Now an attempt to copy a Shape will be caught by the compiler. If you need to copy an object in a class hierarchy, write a `virtual` clone function.

Zen - GC and RAII

In very much the same way as new and delete disappear from application code, we can make pointers disappear into resource handles. In both cases, the result is simpler and more maintainable code, without added overhead. In particular, we can achieve strong resource safety; that is, we can eliminate resource leaks for a general notion of a resource. Examples are vector s holding memory, thread s holding system threads, and fstream s holding file handles.

In many languages, resource management is primarily delegated to a garbage collector. C++ also offers a garbage collection interface so that you can plug in a garbage collector. However, garbage collection is considered as the last alternative after cleaner, more general, and better localized alternatives to resource management have been exhausted.

Garbage collection is fundamentally a global memory management scheme. Clever implementations can compensate, but as systems are getting more distributed (think multicores, caches, and clusters), locality is more important than ever.

Also, memory is not the only resource. A resource is anything that has to be acquired and (explicitly or implicitly) released after use. Examples are memory, locks, sockets, file handles, and thread handles. A good resource management system handles all kinds of resources. Leaks must be avoided in any long-running systems, but excessive resource retention can be almost as bad as a leak. For example, if a system holds on to memory, locks, files, etc., for twice as long, the system needs to be provisioned with potentially twice as many resources.

Before resorting to garbage collection, systematically use resource handles: Let each resource have an owner in some scope and by default be released at the end of its owners scope. In C++, this is known as RAII (Resource Acquisition Is Initialization) and is integrated with error handling in the form of exceptions. Resources can be moved from scope to scope using move semantics or "smart pointers", and shared ownership can be represented by "shared_ptr".

Chap 5 Template

`BEGIN` AND `END`

INSTANTIATION

Templates are a compile-time mechanism, so their use incurs no run-time overhead compared to hand-crafted code. In fact, the code generated for Vector is **identical** to the code generated for the version of Vector from

Chapter 4. Furthermore, the code generated for the standard-library vector is likely to be better (because more effort has gone into its implementation).

VALUE ARGUMENT

```
template<typename T, int N>
struct Buffer {
    using value_type = T;
    constexpr int size() { return N; }
    T Q[N];
};

Buffer<char,1024> glob;
    // global buffer of characters (statically allocated)
void fct() {
    Buffer<int,10> buf;
        // local buffer of integers (on the stack)
}
```

FUNCTION TEMPLATE

```
template<typename Container, typename Value>
Value sum(const Container& c, Value v) {
    for (auto x : c)
        v += x;
    return v;
}

void user(Vector<int>& vi,
          std::list<double>& ld,
          std::vector<complex<double>>& vc) {
    int x = sum(vi,0);
        // the sum of a vector of ints (add ints)
    double d = sum(vi,0.0);
        // the sum of a vector of ints (add doubles)
    double dd = sum(ld,0.0);
        // the sum of a list of doubles
    auto z = sum(vc,complex<double>{});
        // the sum of a vector of complex<double>
        // the initial value is {0.0,0.0}
}
```

CONCEPTS AND GENERIC PROGRAMMING

Templates offer:

- The ability to pass types (as well as values and templates) as arguments without loss of information. This implies excellent opportunities for inlining, of which implementations take great advantage.
- Delayed type checking (done at instantiation time). This implies opportunities to weave together information from different contexts.
- The ability to pass constant values as arguments. This implies the ability to do compile-time computation.

CONCEPT – C++ 17

So, sum() requires that its first template argument is some kind of container and its second template argument is some kind of number. We call such requirements concepts. Unfortunately, concepts cannot be expressed directly in C++11. All we can say is that the template argument for sum() must be types. There are techniques for checking concepts and proposals for direct language support for concepts, but both are beyond the scope of this note.

REGULAR TYPE

- can be default constructed.
- can be copied (with the usual semantics of copy yielding two objects that are independent and compare equal) using a constructor or an assignment.
- can be compared using == and != .
- doesn't suffer technical problems from overly clever programming tricks.

FUNCTION OBJECT

```
template<typename T>
class Less_than {
    const T val;
    // value to compare against
public:
    Less_than(const T& v) :val(v) { }
    bool operator()(const T& x) const { return x < val; }
        // call operator
};

template<typename C, typename P>
int count(const C& c, P pred) {
    int cnt = 0;
    for (const auto& x : c)
        if (pred(x))
            ++cnt;
    return cnt;
}

void f(const Vector<int>& vec,
       const list<string>& lst,
       int x, const string& s) {
cout << "number of values less than " << x
     << ": " << count(vec, Less_than<int>{x})
     << '\n';
cout << "number of values less than " << s
     << ": " << count(lst, Less_than<string>{s})
     << '\n';
}
```

The beauty of these function objects is that they carry the value to be compared against with them. We don't have to write a separate function for each value (and each type), and we don't have to introduce nasty global variables to hold values.

Also, for a simple function object like `Less_than` inlining is simple. So that a call of `Less_than` is far more efficient than an indirect function call.

LAMBDA EXPR

```
void f(const Vector<int>& vec,
       const list<string>& lst,
       int x, const string& s) {
cout << "number of values less than " << x
     << ": " << count(vec, [&](int a) {return a < x;})
     << '\n';
cout << "number of values less than " << s
     << ": " << count(lst, [&](const string& a) {return a < s;})
     << '\n';
}
```

The notation `[&](int a){ return a < x; }` is called a lambda expression. It generates a function object exactly like Less_than{x} . The [&] is a capture list specifying that local names used (such as x) will be accessed through references. Had we wanted to "capture" only x , we could have said so: [&x] . Had we wanted to give the generated object a copy of x , we could have said so: [=x].

Capture nothing is [ ] , capture all local names used by reference is [&] , and capture all local names used by value is [=] .

VARIADIC TEMPLATE

```
void f() { } // do nothing

template<typename T>
void f(T t) {
    cout << t << " ";
}

template<typename T>
void g(T t) {
    cout << t << " ";
}

template<typename T, typename ... Tail>
void f(T head, Tail... tail) {
    g(head);    // do something to head
    f(tail...); // try again with tail
}

int main () {
    cout << "first: ";
    f(1, 2.2, "hello");
    cout << "\nsecond: ";
    f(0.2, 'c', "yuck!", 0, 1, 2);
    cout << "\n";
    return 0;
}
```

ALIASES

For example, the standard header `<cstddef>` contains a definition of the alias size_t , maybe:

```
using size_t = unsigned int;
```

Chap 6 Lib Overview

`string`, `ostream`, `vector`, `map`, `unique_ptr`, `thread`, `regex`, `complex`

`<algorithm> copy() , find() , sort()`

`<array> array`

`<chrono> duration , time_point`

`<cmath> sqrt() , pow()`

`<complex>  complex , sqrt() , pow()`

`<forward_list> forward_list`

`<fstream> fstream , ifstream , ofstream`

`<future> future , promise`

`<ios> hex , dec , scientific , fixed , defaultfloat`

`<iostream> istream , ostream , cin , cout`

`<map> map , multimap`

`<memory> unique_ptr , shared_ptr , allocator`

`<random> default_random_engine , normal_distribution`

`<regex> regex , smatch`

`<string> string , basic_string`

`<set> set , multiset`

`<sstream> istrstream , ostrstream`

`<stdexcept> length_error , out_of_range , runtime_error`

`<thread> thread`

`<unordered_map> unordered_map , unordered_multimap`

`<utility> move() , swap() , pair`

`<vector> vector`

Chap 7 String and Regex

MUTABLE

```
string name = "Niels Stroustrup";
void m3() {
string s = name.substr(6, 10);
    // s = "Stroustrup"
name.replace(0, 5, "nicholas");
    // name becomes "nicholas Stroustrup"
name[0] = toupper(name[0]);
    // name becomes "Nicholas Stroustrup"
}
```

SHORT-STRING OPTIMIZATION

Short string values are kept in the string object itself, and only longer strings are placed on free store.

When lots of strings of differing lengths are used, **memory fragmentation** can result. That why short-string optimization has implemented.

BASIC__STRING

`string` is just an alias of `basic_string<char>`.

REGEX

`regex_match`, `regex_search`, `regex_replace`, `regex_iterator`, `regex_token_iterator`

The regular expression syntax and semantics are designed so that regular expressions can be compiled into **state machines** for efficient execution. The `regex` type performs this compilation at run time.

SMATCH

An smatch is a `vector` of `sub-matches` of type `string`. The resule of a `regex_search()` is a collection of matches, typically represented as an match.

```
void search() {
    regex pat {"\\w{2}\\s*\\d{5}(-\\d{4})?"};
    int lineno = 0;
    for (string line; getline(cin, line);) {
        lineno++;
        smatch matches;
        if (regex_search(line, matches, pat))
            cout << lineno << ": " << matches[0] << endl;
    }
}
```

REGEX NOTATION

`regex` lib can recognize several variants of the notation for regular expression. It uses ECMA-Regex Standard.

SPECIAL CHAR

- `.` Any single character (a "wildcard")
- `[` Begin character class
- `]` End character class
- `{` Begin count
- `}` End count
- `(` Begin grouping
- `)` End grouping
- `\` Next character has a special meaning
- `*` Zero or more (suffix operation)
- `+` One or more (suffix operation)
- `?` Optional (zero or one) (suffix operation)
- `|` Alternative (or)
- `^` Start of line; negation
- `$` End of line

REPEATED

A pattern can be optional or repeated (the default is exactly once) by adding a suffix.

- `{n}` exactly `n` times

- `{n, }` `n` or more times
- `{n, m}` at least `n` and at most `m` times And `* + ?`.

CHARACTER CLASSES

- `alnum` Any alphanumeric character
- `alpha` Any alphabetic character
- `blank` Any whitespace character that is not a line separator
- `cntrl` Any control character
- `d` Any decimal digit
- `digit` Any decimal digit
- `graph` Any graphical character
- `lower` Any lowercase character
- `print` Any printable character
- `punct` Any punctuation character
- `s` Any whitespace character
- `space` Any whitespace character
- `upper` Any uppercase character
- `w` Any word character (alphanumeric characters plus the underscore)
- `xdigit` Any hexadecimal digit character
- `\d` A decimal digit [[:digit:]]
- `\s` A space (space, tab, etc.) [[:space:]]
- `\w` A letter (a-z) or digit (0-9) or underscore (_) [_[:alnum:]]
- `\D` Not\d [^[:digit:]]
- `\S` Not\s [^[:space:]]
- `\W` Not\w [^_[:alnum:]]

C++ VAR NAMES

```
[:alpha:][:alnum:]*;
    // wrong: characters from the set ":alph" followed by ...
[[:alpha:]][[:alnum:]]*;
    // wrong: doesn't accept underscore ('_' is not alpha)
([[:alpha:]]|_)[[:alnum:]]*;
    // wrong: underscore is not part of alnum either
([[:alpha:]]|_)([[:alnum:]]|_)*;
    // OK, but clumsy
[[:alpha:]_][[:alnum:]_]*;
    // OK: include the underscore in the character classes
[_[:alpha:]][_[:alnum:]]*;
    // also OK
[_[:alpha:]]\w*;
    // \w is equivalent to [_[:alnum:]]
```

GREEDY AND NON-GREEDY

A suffix `?` after any of the repetition notations(above) make the pattern matcher **non-greedy**. And the default is **greedy**.

SUBPATTERN

A `group` (a subpattern) potentially to be represented by a `sub_match` is delimited by parentheses. If you need parentheses that should not define a subpattern, use `(?` instead of `(`.

REGEX_ITERATOR

```
for (void test() {
    string input = "aa as; asd ++e^asdf asdfg";
    regex pat {R"(\s+(\w+))"};
    for (sregex_iterator p(input. begin(), input.end(), pat);
         p!=sregex_iterator{}; ++p
    )
        cout << (*p)[1] << '\n';
}
```

Chap 8 I/O

IO STATE

```
while (cin) {
    for (int i; cin >> i; ) {
        // ... use the integer ...
    }
    if (cin.eof()) {
        // .. all is well we reached the end-of-file ...
    }
    else if (cin.fail()) { // a potentially recoverable error
        cin.clear(); // reset the state to good()
        char ch;
    if (cin>>ch) { // look for nesting represented by { ... }
            switch (ch) {
                case '{':
                    // ... start nested structure ...
                    break;
                case '}':
                // ... end nested structure ...
                break;
                default:
                cin.setstate(ios_base::failbit); // add fail() to cin's state
            }
        }
    }
    // ...
}
```

USER DEFINED TYPES

```
ostream& operator << (ostream& os, const Entry& e) {
    return os << "{\"" << e.name << "\", " << e.number << "}";
}
```

FILESTREAM

In `fstream`, the standard library provides streams to and from a file.

- `ifstreams` for reading from a file
- `ofstreams` for writing to a file
- `fstreams` for reading from and writing to a file

STRING STREAM

In `<sstream>`, the standard library provides streams to and from a string:

- `istringstreams` for reading from a string
- `ostringstreams` for writing to a string
- `stringstreams` for reading from and writing to a string.

Chap 9 Containers

A class with the main purpose of holding objects is commonly called a *container*.

VECTOR.RESERVE()

It can not improve the performance, as allocation method is heuristic.

STORE POINTER | VALUE?

If there is a class hierachy that relies on `virtual` functions to get polymorphic behavior, do not store objects directly. Just store a pointer or smart pointer.

VECTOR RANGE CHECKING

`vector` does not guarantee range checking in subscript, but in `.at()`.

Other containers

During the OI/ACM time, I am quite familiar with `set`, `map`, or `unordered_`. And I strongly recommends you to get some info of `pb_ds`, which is supported in `libc++`, algorithms implement there are more effcient than in STL. For instance, `unordered_set` in STL uses link-list to deal with hash crashes, which is slow.

Chap 11 Utilities

UNIQUE_PTR

It is to represent unique ownership. The most basic use of these “smart pointers” is to prevent memory leaks caused by careless programming, such as early returns and etc.

Ironically, we could have solved the problem simply by not using a `pointer` and not using `new`.

SHARED_PTR

The `shared_ptr` is similar to `unique_ptr` except that `shared_ptr`s are copied rather than moved. The `shared_ptr`s for an object share ownership of an object and that object is destroyed when the last of its `shared_ptr`s is destroyed.

MAKE_SHARED()

Creating an object on the free store and then passing a pointer to it to a smart pointer is logically bit odd and can be verbose. To compensate, the standard library (in `<memory>`) provides a function `make_shared()`.

```
shared_ptr<S> p1 {new S {1, "Ankh Morpork", 4.65}};
auto p2 = make_shared<S>(2, "Oz", 7.62);

template<typename T, typename ... Args>
unique_ptr<T> make_unique(Args&&... args) {
    return std::unique_ptr<T> {new T{std::forward<Args>(args)...}};
}
```

Given unique_ptr and shared_ptr, we can implement a complete "no naked new" policy.

<ARRAY>

An array, defined in `<array>`, is a fixed-size sequence of elements of a given type where the number of elements is specified at compile time.

Thus, an array can be allocated with its elements on the **stack**, in an object, or in `static` storage. The elements are allocated in the scope where the array is defined. An array is best understood as a built-in `array` with its size firmly attached, without `implicit`, potentially surprising conversions to pointer types, and with a few convenience functions provided.

When necessary, an array can be explicitly passed to a C-style function that expects a pointer.

BITSET

Just use `vector<bool>` is ok.

BIND()

A function adaptor takes a function as argument and returns a function object that can be used to invoke the original function. The standard library provides `bind()` and `mem_fn()` adaptors to do argument binding, also called `Currying` or `partial evaluation`.

```
double cube(double t) {
    return t * t * t;
}
auto cube2 = bind(cube, 2);
int main() {
    cout << cube2();
    return 0;
}
```

```
using namespace placeholders;
void f(int,const string&);
auto g = bind(f, 2, _1);
    // bind f()'s first argument to 2
f(2, "hello");
g("hello");
    // also calls f(2, "hello");
```

The curious `_1` argument to the binder is a placeholder telling `bind()` where arguments to the resulting function object should go. In this case, `g()`'s (first) argument is used as `f()`'s second argument.

FUNCTION

If we want to assign the result of bind() to a variable with a specific type, we can use the standard-library type function.

```
int f1(double);
function<int(double)> fct {f1};
    // initialize to f1
int f2(int);
void user() {
    fct = [](double d) { return round(d); };
        // assign lambda to fct
    fct = f1;
        // assign function to fct
    fct = f2;
        // error : incorrect argument type
}
```

TYPE FUNCTION

A *type function* is a function that is evaluated at compile-time given a type as its argument or returning a type. The standard library provides a variety of type functions to help library implementers and programmers in general to write code that take advantage of aspects of the language, the standard library, and code in general.

Oh it is too dificult.

Chap 12 Numberics

`<CMATH>`

`abs(x)`, `ceil(x)`, `floor(x)`, `sqrt(x)`, `cos(x)`, `sin(x)`, `tan(x)`, `acos(x)`

`asin(x)`, `atan(x)`, `sinh(x)`, `cosh(x)`, `tanh(x)`, `exp(x)`, `log(x)`, `log10(x)`

`<NUMERIC>`

- `x = accumulate(b, e, i)`, x is the sum of i and elements of [b, e)
- 'x = accumulate(b, e, i, f)', accumulate using f instead of +
- `x = inner_product(b, e, b2, i, f, f2)`

`RANDOM`

A random number generator consists of two parts: 1. an engine that produces a sequence of random or pseudo-random values. 1. a distribution that maps those values into a mathematical distribution in a range.

Chap 13 Cocurrency

The standard-library support is primarily aimed at supporting systems-level concurrency rather than directly providing sophisticated higher-level concurrency models.

Main standard-library concurrency support facilities: `threads`, `mutexes`, `lock()` operations, `packaged_tasks`, and `futures`. These features are built directly upon what operating systems offer and do not incur performance penalties compared with those.

Do not consider concurrency a panacea.

Also, all the codes in this chapter when copiled, `-pthread` must be added, or it will CE.

THREAD

```
void f() {
    cout << "Hello";
}
struct F {
    public:
        void operator()(){ cout << "Parallel World! "; }
};
int main() {
    thread t1{f};
    thread t2{F()};
    t1.join();
    t2.join();
    return 0;
}
```

CONST REFERENCE

```
void f(const vector<double>& v, double* res) {
    cout << "Hello";
}

struct F {
    public:
        F(const vector<double>& vv, double* p):v{vv}, res{p} { }
        void operator()(){ cout << "Parallel World! "; }
    private:
        const vector<double>& v;
        double* res;
};

int main() {
    vector<double> some_vec;
    vector<double> vec2;

    double res1, res2;

    thread t1{f, cref(some_vec), &res1};
    thread t2{F{vec2, &res2}};

    t1.join();
    t2.join();

    cout << res1 << " " << res2 << endl;
    return 0;
}
```

MUTEX

The access has to be synchromnized so that at most one task at a time has access. The fundamental element of the solution is a `mutex`, a "mutual exclusion object". A `thread` acquires a mutex using a `lock()` operation.

Do not add `-O2` flags when compiling the code below, as it may cause no deadlock.

```
mutex m1, m2;

int de() {
        // designed to delay some time
```

```
    int res;
    for (int i = 0; i < 100000; i++)
        for (int j = 0; j < 100000; j++)
            res = i + j % 1007;
    return res;
}

void f() {
    cout << de();
        // first the a.out will uses 200% of cpu
    unique_lock<mutex> lck1{m1};
        // then as the g aquired m2, aquiring m1
        //              f aquired m1, aquiring m2
        // deadlock
    cout << de();
    unique_lock<mutex> lck2{m2};
}

void g() {
    cout << de();
    unique_lock<mutex> lck1{m2};
    cout << de();
    unique_lock<mutex> lck2{m1};
}

int main() {
    thread t1{f};
    thread t2{g};
    t1.join();
        // must add .join otherwise RE
    t2.join();
    return 0;
}
```

MUTEX OR PASS-BY-VALUE ? - ZEN

Communicating through shared data is pretty low level. In particular, the programmer has to devise ways of knowing what work has and has not been done by various tasks. In that regard, use of shared data is inferior to the notion of call and return. On the other hand, some people are convinced that sharing must be more efficient than copying arguments and returns. That can indeed be so when large amounts of data are involved.

But locking and unlocking are relatively expensive operations. On the other hand, modern machines are very good at copying data, especially compact data, such as vector elements. So don't choose shared data for communication because of **efficiency** without thought and preferably not without measurement.

<CHRONO>

```
using namespace std::chrono;
    // must add, as is it a sub-namespace of std

auto t0 = high_resolution_clock::now();
this_thread::sleep_for(milliseconds{100});
auto t1 = high_resolution_clock::now();

cout << duration_cast<nanoseconds>(t1-t0).count()
     << " nanoseconds passed\n";
```

There is no need to launch a thread; by default, `this_thread` refers to the one and only thread.

The basic support for communicating using external events is proviede by `condition_variables` found in `<condition_variable>`. A `condition_variable` is a mechanism allowing one `thread` to wait for another. In particular, it allows a thread to wait for some condition (often called a *event*) to occur as the result of work done by other `threads`.

MUTEX

```
void consumer(){
    while(true) {
        unique_lock<mutex> lck{mmutex};
            // acquire mmutex
        while (mcond.wait(lck))
            // release lck and wait;
            /* do nothing */;
            // re-acquire lck upon wakeup
        auto m = mqueue.front();
            // get the message
        mqueue.pop();
        lck.unlock();
            // release lck
        // ... process m ...
    }
}

void producer() {
    while(true) {
        Message m;
        // ... fill the message ...
        unique_lock<mutex> lck {mmutex};
            // protect operations
        mqueue .push(m);
        mcond.notify_one();
            // notify
    }
        // release lock (at end of scope)
}
```

FUTURE AND PROMISE

The important point about future and promise is that they enable a transfer of a value between two tasks without explicit use of a lock; “the system” implements the transfer efficiently.

The basic idea is simple: When a task wants to pass a value to another, it puts the value into a promise. Somehow, the implementation makes that value appear in the corresponding future, from which it can be read (typically by the launcher of the task).

The main purpose of a promise is to provide simple **put** operations (called `set_value()` and `set_exception()`) to match future’s `get()`.

```
void f(promise<X>& px) {
    // a task: place the result in px
    try {
        X res;
            // ... compute a value for res ...
        px.set_value(res);
    }
    catch (...) {
            // oops: couldn't compute res
        px.set_exception(current_exception());
            // pass the exception to the future's thread
            // current_exception() refers to the caught exception
    }
}

void g(future<X>& fx) {
    // a task: get the result from fx
    try {
        X v = fx.get();
        // if necessary, wait for the value to get computed
    }
    catch (...) {
        // oops: someone couldn't compute v
    }
}
```

PACKAGED_TASK

The packaged_task type is provided to simplify setting up tasks connected with futures and promises to be run on threads.

```
inline double accum(double* beg, double* end, double init) {
    // compute the sum of [beg:end) starting with the initial value init
    return accumulate(beg, end, init);
}

double comp2(vector<double>& v) {
    using Task_type = double(double*, double*, double);
        // type of task
    packaged_task<Task_type> pt0 {accum};
    packaged_task<Task_type> pt1 {accum};
        // package the task (accum)
    future<double> f0 {pt0.get_future()};
        // get hold of pt0's future
    future<double> f1 {pt1.get_future()};
        // get hold of pt1's future
    double* first = &v[0];
    thread t1 {move(pt0), first, first+v.size()/2, 0};
    thread t2 {move(pt1), first+v.size()/2, first+v.size(), 0};
        // start a thread for pt0
        // start a thread for pt1
    return f0.get()+f1.g et();
}
```

The reason that a packaged_task cannot be copied is that it is a resource handle: it owns its promise and is (indirectly) responsible for whatever resoures its task may own.

ASYNC()

It is the simplest yet still among the most powerful. Treat a task as a function that may happen to run concurrently with other tasks. It is far from the only model supported by the C++ standard library, but it serves well for a wide range of needs.

```
double comp4(vector<double>& v) {
    // spawn many tasks if v is large enough
    if (v.siz e()<10000)
        // is it wor th using concurrency?
        return accum(v.begin(), v.end(), 0.0);
    auto v0 = &v[0];
    auto sz = v.siz e();
    auto f0 = async(accum, v0, v0 + sz / 4, 0.0);
    auto f1 = async(accum, v0  + sz / 4, v0+sz / 2, 0.0);
    auto f2 = async(accum, v0 + sz / 2, v0+sz * 3 / 4, 0.0);
    auto f3 = async(accum, v0 + sz * 3 / 4, v0 + sz, 0.0);
    return f0.get()+f1.g et()+f2.g et()+f3.g et();
        // collect and combine the results
}
```

Using `async()` , you don't have to think about threads and locks. Instead, you think just in terms of tasks that potentially compute their results asynchronously. There is an obvious limitation: Don't even think of using `async()` for tasks that share resources needing locking – with async() you don't even know how many thread s will be used because that's up to `async()` to decide based on what it knows about the system resources available at the time of a call.

THREADS AT MOST

```
cout << thread::hardware_concurrency() << endl;
```

Chap 14 History and Compatibility

HISTORY

C++ was designed to provide Simula's facilites for program organization. Simula is the initial source of C++'s abstraction mechanisms. The class concept (with derived classes and virtual functions) was borrowed from it. However, templates and exceptions came to C++ later with different sources of inspiration.

TIMELINE

The work that led to C++ started in the fall of 1979 under the name "C with Classes." Here is a timeline.

- 1979 Work on "C with Classes" started. The initial feature set included classes and derived classes, public/private access control, constructors and destructors, and function declarations with argument checking. The first library supported non-preemptive concurrent tasks and random number generators.
- 1984 "C with Classes" was renamed to C++. By then, C++ had acquired virtual functions, function and operator overloading, references, and the I/O stream and complex number libraries.
- 1985 First commercial release of C++ (October 14). The library included I/O streams, complex numbers, and tasks (non-preemptive scheduling).
- 1985 The C++ Programming Language was published.
- 1989 The Annotated C++ Reference Manual was published.
- 1991 The C++ Programming Language, Second Edition was published, presenting generic programming using templates and error handling based on exceptions (including the "Resource Acquisition Is Initialization"(**RAII**) general resource management idiom).
- 1997 The C++ Programming Language, Third Edition was published, introduced ISO C++, including namespaces, dynamic_cast , and many refinements of templates. The standard library added the STL framework of generic containers and algorithms.
- 1998 ISO C++ standard [C++,1998]. `c++98`
- 2002 Work on a revised standard, colloquially named C++0x, started.
- 2003 A "bug fix" revision of the ISO C++ standard was issued. A C++ Technical Report introduced new standard-library components, such as regular expressions, unordered containers (hash tables), and resource management pointers, which later became part of C++0x.
- 2006 An ISO C++ Technical Report on Performance was issued to answer questions of cost, predictability, and techniques, mostly related to embedded systems programming [C++,2004].
- 2009 C++0x was feature complete. It provided uniform initialization, move semantics, variadic template arguments, lambda expressions, type aliases, a memory model suitable for concurrency, and much more. The standard library added several components, including threads, locks, and most of the components from the 2003 Technical Report.

- 2011 ISO C++11 standard was formally approved [C++,2011].`-std=c++11`
- 2012 Work on future ISO C++ standards (referred to as C++14 and C++17) started.
- 2013 The first complete C++11 implementations emerged.
- 2013 The C++ Programming Language, Fourth Edition introduced C++11.

EARLY YEARS

Some event-driven simulations is needed, for which Simula would have been ideal, except for performance considerations. Dealing directly with hardware and provide high-performance concurrent programming mechanisms for which C would have been ideal, except for its weak support for modularity and type checking.

The result of adding Simula-style classes to C, `C with Classes`, was used for major projects in which its facilities for writing programs that use minimal time and space were severely tested. It lacked operator overloading, references, virtual functions, templates, exceptions, and many, many details. The first use of C++ outside a research organization started in July 1983.

In the early years, there was no C++ paper design; design, documentation, and implementation went on simultaneously.

BS's view was (and is) that we need every bit of help we can get from languages and tools: the inherent complexity of the systems we are trying to build is always at the edge of what we can express.

C++ grew up in an environment with a multitude of established and experimental programming languages. That was a deliberate policy to have the development of C++ `problem driven` rather than imitative.

ISO C++

C++ grew by about 30% and the standard lib by about 100% between c++98 and c++11. Much of the increase was due to more detailed specification, rather than new functionality.

C++11 added massively to the standard lib and pushed to complete the feature set needed for a programming style that is a synthesis of the "paradigms" and idioms that have proven successful with c++98.

The overall aims for the c++11 efforts were: - make c++ a better language for systems programming and lib building - make c++ easier to teach and learn (*are you kidding me?*)

A major effort was made to make concurrent systems programming type-safe and portable. This is involved a memory model and a set of facilities for **lock-free** programming.

C++11 - LANGUAGE FEATURE

1. Uniform and general initialization using `{}` -lists
2. Type deduction from initializer: `auto`
3. Prevention of narrowing
4. Generalized and guaranteed constant expressions: `constexpr`
5. Range- `for` -statement
6. Null pointer keyword: `nullptr`
7. Scoped and strongly typed `enums` : `enum class`
8. Compile-time assertions: `static_assert`
9. Language mapping of `{}` -list to `std::initializer_list`
10. Rvalue references (enabling move semantics)
11. Nested template arguments ending with `>>` (no space between the `>` s)
12. Lambdas
13. Variadic templates
14. Type and template aliases
15. Unicode characters
16. `long long` integer type
17. Alignment controls: `alignas` and `alignof`
18. The ability to use the type of an expression as a type in a declaration: `decltype`
19. Raw string literals

20. Generalized POD ("Plain Old Data")
21. Generalized `unions`
22. Local classes as template arguments
23. Suffix return type syntax
24. A syntax for attributes and two standard attributes: `[[carries_dependency]]` and `[[noreturn]]`
25. Preventing exception propagation: the `noexcept` specifier
26. Testing for the possibility of a `throw` in an expression: the `noexcept` operator.
27. C99 features: extended integral types (i.e., rules for optional longer integer types); con-
28. catenation of narrow/wide strings; `__STDC_HOSTED__` ; `_Pragma(X)` ; vararg macros and empty macro arguments
29. `__func__` as the name of a string holding the name of the current function
30. `inline` namespaces
31. Delegating constructors
32. In-class member initializers
33. Control of defaults: `default` and `delete`
34. Explicit conversion operators
35. User-defined literals
36. More explicit control of `template` instantiation: `extern template` s
37. Default template arguments for function templates
38. Inheriting constructors
39. Override controls: `override` and `final`
40. Simpler and more general SFINAE rule
41. Memory model
42. Thread-local storage: `thread_local`

c++11 - STL component

1. `initializer_list` constructors for containers
2. Move semantics for containers
3. A singly-linked list: `forward_list`
4. Hash containers: `unordered_map` , `unordered_multimap` , `unordered_set` , and `unordered_multiset`
5. Resource management pointers: `unique_ptr` , `shared_ptr` , and `weak_ptr`
6. Concurrency support: `thread`, `mutexes`, `locks`, and condition variables
7. Higher-level concurrency support: packaged_thread , future , promise , and async()
8. `tuple` s
9. Regular expressions: `regex`
10. Random numbers: `uniform_int_distribution` , `normal_distribution`, `random_engine` , etc.
11. Integer type names, such as `int16_t` , `uint32_t` , and `int_fast64_t`
12. A fixed-sized contiguous sequence container: `array`
13. Copying and rethrowing exceptions
14. Error reporting using error codes: system_error
15. `emplace()` operations for containers
16. Wide use of `constexpr` functions
17. Systematic use of `noexcept` functions
18. Improved function adaptors: `function` and `bind()`
19. `string` to numeric value conversions
20. Scoped allocators
21. Type traits, such as `is_integral` and `is_base_of`
22. time utilities: `duration` and `time_point`
23. Compile-time rational arithmetic: `ratio`
24. Abandoning a process: `quick_exit`
25. More algorithms, such as `move()` , `copy_if()` , and `is_sorted()`
26. Garbage collection ABI
27. Low-level concurrency support: `atomic` s

C++11 - DEPRECATED FEATURE

By deprecating a feature, the standards committee expresses the **wish** that the feature will go away.

- Generation of the copy constructor and the copy assignment is deprecated for a class with a destructor.
- It is no longer allowed to assign a string literal to a `char*` . Instead of `char*` as a target for assignment and initializations with string literals, use `const char*` or `auto`
- C++98 exception specifications are deprecated:
  - `void f() throw(X,Y); // C++98; now deprecated`
  - The support facilities for exception specifications, `unexcepted_handler` , `set_unexpected()` , `get_unexpected()` , and `unexpected()` , are similarly deprecated. Instead, use `noexcept`.
- Some C++ standard-library function objects and associated functions are deprecated. Most relate to argument binding. Instead use `lambdas`, `bind` , and `function`.
- The `auto_ptr` is deprecated. Instead, use `unique_ptr`.
- The use of the storage specifier `register` is deprecated.
- The use of `++` on a bool is deprecated.

C++11 - CAST

C-style casts should have been deprecated in favor of named casts. The named casts are:

- `static_cast` : for reasonably well-behaved conversions, such as from a pointer to a base to its derived class.
- reinterpret_cast : For really nasty, non-portable conversions, such as conversion of an int to a pointer type. (Q: When to use?)
- `const_cast` : For casting away const .

```
Widg et* pw = static_cast<Widget*>(pv);
    // pv is a void* supposed to point to a Widget
auto dd = reintrepret_cast<Device_driver*>(0xFF00);
    // 0xFF is supposed to point to a device driver
char* pc = const_cast<char*>("Casts are inherently dang erous");
```

Expricit type conversion can be completely avoided in most high-level code, so considering every cast (however expressed) a blemish on design is important.

C/C++ COMPATIBILITY

With minor exceptions, C++ is a superset of C11. Most differences stem from C++'s greater emphasis on type checking. Before 2016, C11 is still very new and most C code is `Classic C` or `C99`.

SIDLINGS

Classic C has two main descendants: ISO C and ISO C++. Over the years, these languages have evolved at different paces and in different directions. One result of this is that each language provides support for traditional C-style programming in slightly different ways. The resulting incompatibilities can make life miserable for people who use both C and C++, for people who write in one language using libraries implemented in the other, and for implementers of libraries and tools for C and C++.

C -> C++

There are many minor incompatibilities between C and C++. All can cause problems for a programmer, but all can be coped with in the context of C++. If nothing else, C code fragments can be compiled as C and linked to using the extern “C” mechanism. The major problems for converting a C program to C++ are likely to be:

- Suboptimal design and programming style.
- A `void*` implicitly converted to a `T*` (that is, converted without a cast).
- C++ keywords used as identifiers in C code.

- Incomparible linkage between code fragments compiled as C and code fragments compiled as C++.

STYLE PROB

1. Don't think of C++ as C with a few features added. C++ can be used that way, but only suboptimally. To get really major advantages from C++ as compared to C, you need to apply different **design** and **implementation styles**.
2. Use the C++ standard library as a teacher of new techniques and programming styles. Note the difference from the C standard library (e.g., = rather than `strcpy()` for copying and `==` rather than `strcmp()` for comparing).
3. Macro substitution is almost never necessary in C++. Use `const`, `constexpr`, `enum` or `enum class` to define manifest constants, `inline` to avoid functioncalling overhead, `templates` to specify families of functions and types, and `namespaces` to avoid name clashes.
4. Don't declare a variable before you need it, and initialize it immediately. A declaration can occur anywhere a statement can, in for-statement initializers , and in conditions.
5. Don't use `malloc()`. The `new` operator does the same job better, and instead of `realloc()`, try a vector. Don't just replace `malloc()` and `free()` with "naked" `new` and `delete`.
6. Avoid `void*`, `unions`, and `casts`, except deep within the implementation of some function or class. Their use limits the support you can get from the type system and can harm performance. In most cases, a `cast` is an indication of a design error.
7. If you must use an explicit type conversion, use an appropriate named cast (e.g., `static_cast`) for a more precise statement of what you are trying to do.
8. Minimize the use of arrays and C-style strings. C++ standard-library strings, `arrays` , and `vectors` can often be used to write simpler and more maintainable code compared to the traditional C style. In general, try not to build yourself what has already been provided by the standard library.
9. Avoid pointer arithmetic except in very specialized code (such as a memory manager) and for simple array traversal (e.g., ++p).
10. Do not assume that something laboriously written in C style (avoiding C++ features such as classes, templates, and exceptions) is more efficient than a shorter alternative (e.g., using standard-library facilities). Often (but of course not always), the opposite is true.

VOID*

In C, a `void*` may be used as the right-hand operand of an assignment to or initialization of a variable of any pointer type; in C++ it may not.

```
void f(int n) {
int* p = malloc(n*siz eof(int));
    /* not C++; in C++, allocate using ‘‘new’’ */
}
```

This is probably the single most difficult incompatibility to deal with. Note that the implicit conversion of a `void*` to a different pointer type is not in general harmless:

```
char ch;
void* pv = &ch;
int* pi = pv;
    // not C++
*pi = 666;
    // overwr ite ch and other bytes near ch
```

If you use both languages, cast the result of `malloc()` to the right type. If you use only C++, avoid `malloc()`.