CS106L-C++

Introduction

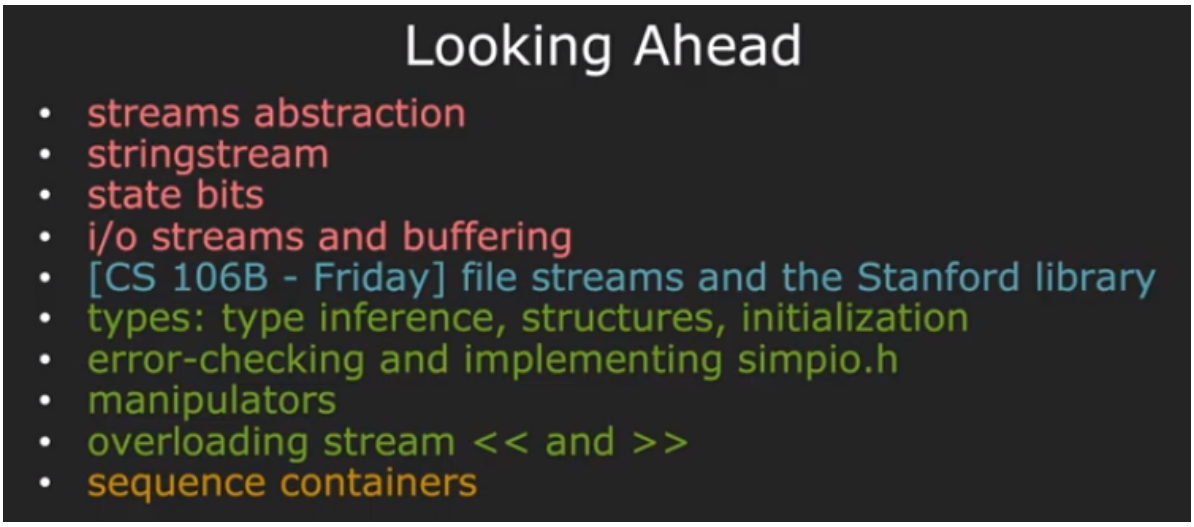
C++ History

Assembly——C——C++

Design Philosophy of C++

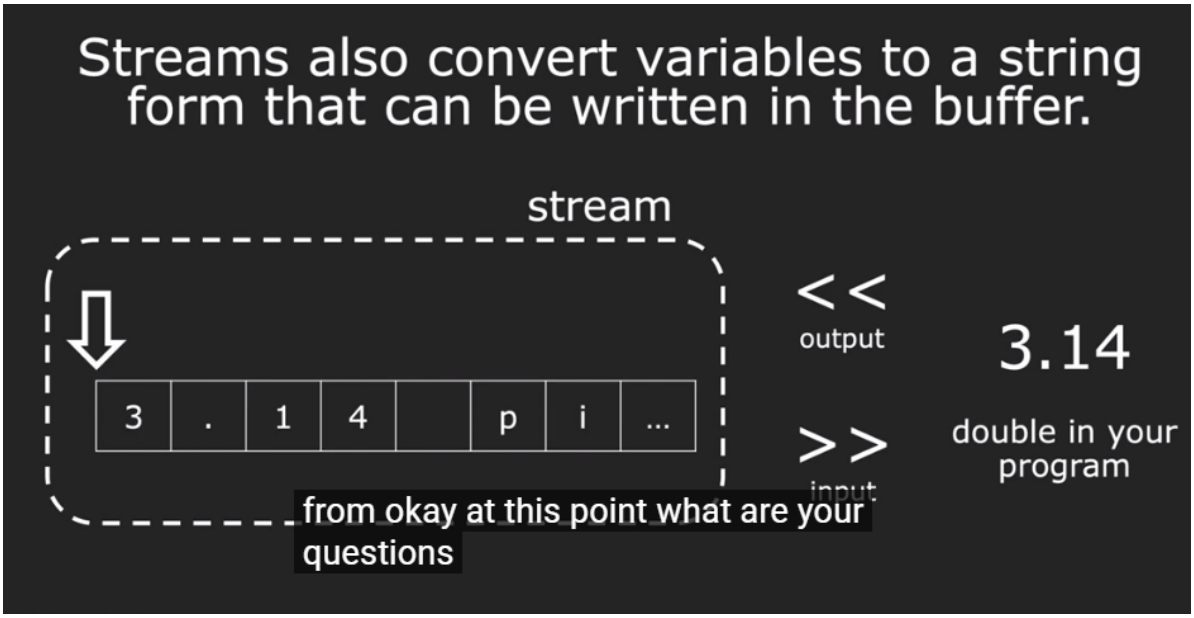
- Allow all kinds of paradigms
- Express the intent directly
- Enforce safety at compile time whenever possible
- Do not waste time and space
- Compartmentalize messy constructs

Looking Ahead



Streams

Stream



iostream

> - cin
> - cout (buffered)
> - cerr (unbuffered)
> - clog (buffered)

cout

cout doesn't print the word on the console immediately but store in a buffer. After a flush, sometimes explicit one, the contents can be printed.

endl is newline plus a flush

cin

> ```
> 1 cin >> MyInterger >> endl; // Wrong! Cannot read into endl
> ```
>
> cin has no safety checking!

How cin work:

- cin finds a empty buffer and wait for your input
- it will read up to the next whitespace character
- if the buffer is not empty, it will not ask for more input

fstream

```
// Method 1
ifstream ifs("file.txt");
ifs >> myInteger;

// Method 2
ifstream ifs;
ifs.open("file.txt");

if(ifs.is_open())
    cerr << "Fail" << endl;
// cerr is similar to cout but sometimes handle differently
```

```
ifstream input(myString.c_str());
// ifstream and ofstream is predate the string type so they only support C-style string
```

Stream Manipulators

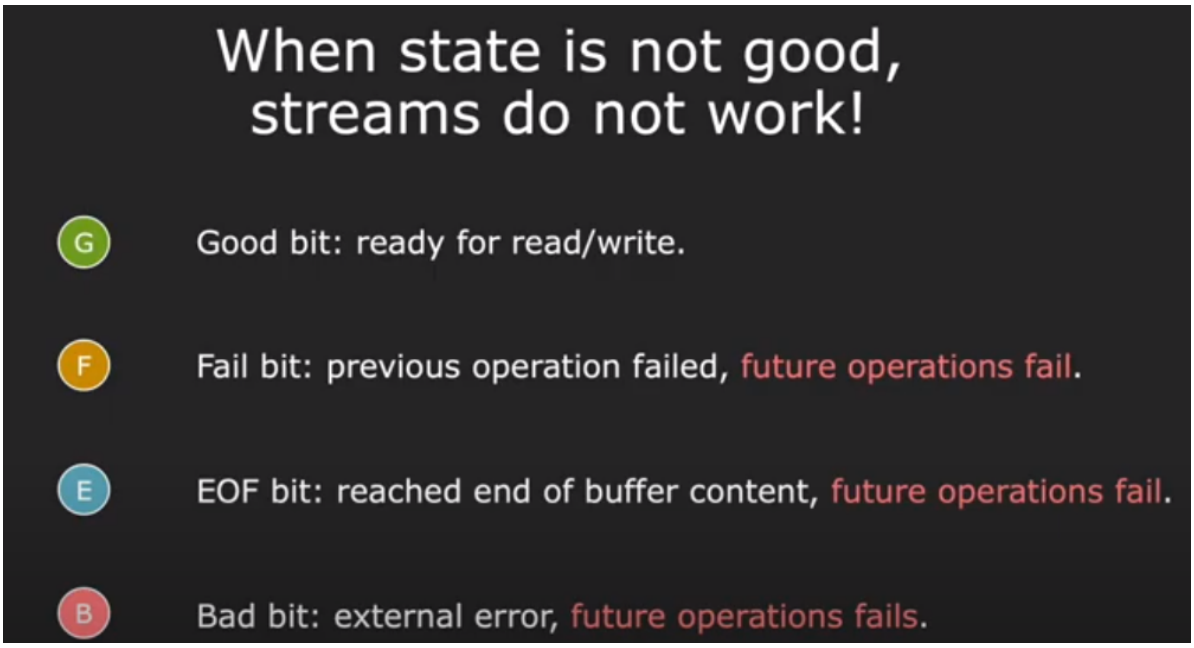
```
#include<iomanip>

cout << setw(10) << 137 << endl;
// setw(set width) force the rest output with n width

cout << '[' << left << setw(10) << "Hello!" << ']' << endl;
// [    Hello!]
// pay attention to 'left'

cout << setfill('0') << setw(8) << 1000 << endl; // 00001000
cout << setw(8) << 1000 << endl; // 00001000 because of last setfill
// setfill is only meaningful with setw

cout << setfill('-') << setw(10) << "" << setfill(' '); // ----------
```

```
// boolalpha
cout << true << endl; // 1
cout << boolalpha << true << endl; // true
cout << noboolalpha << true << endl; // 1

// setw(n)

// hex, dec, oct
cout << dec << 10 << endl;
cout << oct << 10 << endl;
cout << hex << 10 << end;
cin >> hex >> x;

// ws:Skip any whitespace in the stream
```

```
myStream >> ws >> value;
```

Stream Problem

> First, the user could enter something that is not an integer, causing cin to fail. Second, the user could enter too much input, such as 137 246 or Hello 37, in which case the operation succeeds but leaves extra data in cin that can garble future reads.



When Streams Go Bad

```
cin >> myInteger;
if (cin.fail()) {
    // if a stream is in a fail state, it will not change the contents meanwhile not receive the input. You have to use .clear() to get out of error state and extract the wrong input!
}
```

```
// Right Way
while (true) {
    ifs >> myInteger;
    if (ifs.fail()) break;
    /* process data here */
}

// Wrong Way
while (!ifs.fail()) {
    ifs >> myInteger;
    // It will cause the loop to execute once more
}

// Upgrade Way
while (ifs >> myInteger) {
    /* process data here */
}
```

When Streams Do Too Much

Example

```
int x;
double y;
cin >> x;
cin >> y;
// Input:2.7
// Res:No error signal. x=2, y=0.7
```

An alternative: getline

```
string myStr;
getline(cin, myStr);
// getline(stream, string)
// getline read the stream until a newline
// getline(stream, string, '\n'): read until '\n' and don't include the '\n'

// GetLine in <simpio.h> and getline is the same
string GetLine() {
    string result;
    getline(cin, result);
    return result;
}

// however, there is a small problem
cin >> a;
cin >> b; // this cin ignores the last newline and whitespace
// The newline stored in cin after the user enters a value for first is eaten by cin before second is read

// don't mix >> and getline!
cin >> a;
getline(cin, str); // str will be empty
// getline won't skip the newline and white space, so getline gets a newline by last cin, and terminates.

// Method 1
cin >> a;
cin.ignore(); // ignore one character
getline();
```

Suggestion: Avoid using stream extraction operations but use GetLine and GetInteger

```
// avoid loop-and-a-half
while (true) {
    getline(stream, str);
    //
    if (stream.fail())
        break;
}

// a wiser way
while (getline(stream, str)) {
    /* process */
}
```

String Stream

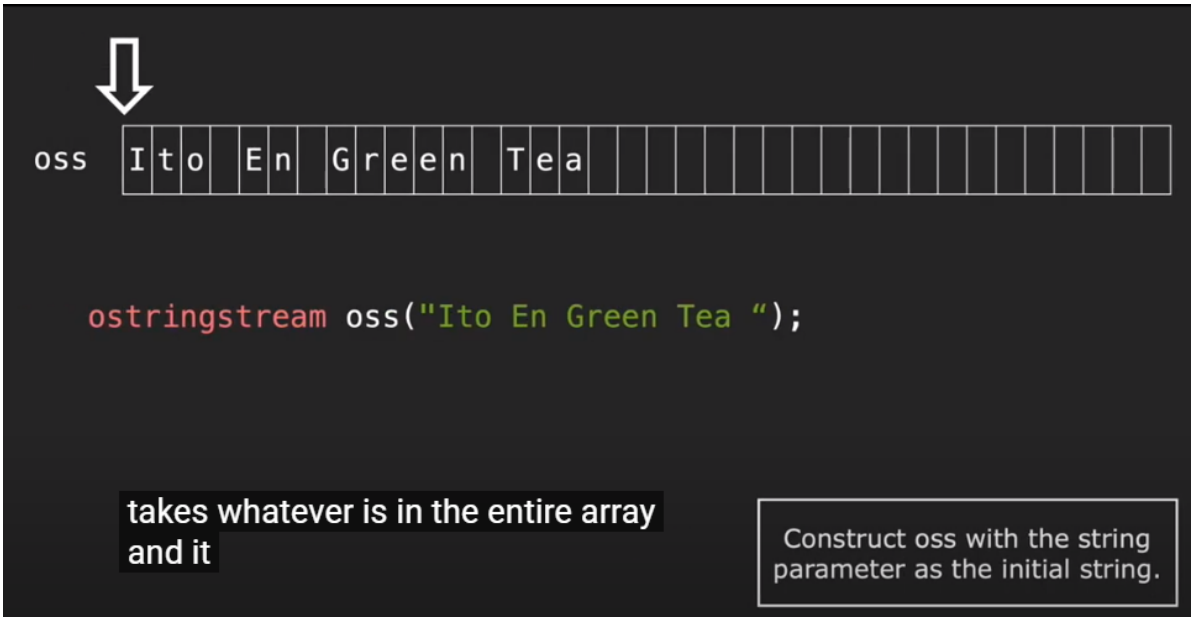
Stringstream is an **iostream** for both input and output

```
stringstream ss;
ss << Integer << "is an integer"; // int + string is not allowed, but ss can transform int to string
ss >> Integer; // ss can transform string to other type
```

ostringstream

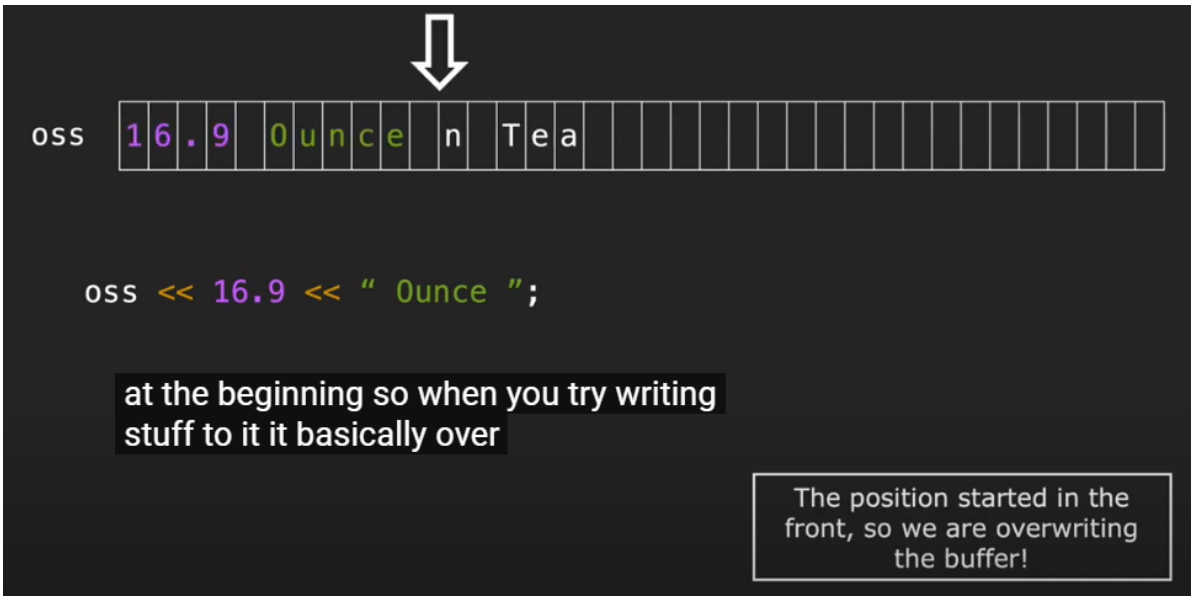
View oss as an array

```
#include<iostream>
#include<sstream>
using namespace std;
int main()
{
    // Code 1
    ostringstream oss("Ito En Green Tea");
    cout << oss.str()<< endl;
    // use .str() to retrieve the string
    // Ito En Green Tea

    oss << 16.9 << " Ounce ";
    cout << oss.str() << endl;
    // 16.9 Ounce n Tea

    // Code 2
    ostringstream oss("Ito En Green Tea", ostringstream::ate);
    // ate : at end
    // open the stream at the end

    return 0;
}
```



OSS can be wiritten or read from

```
g++ -std=c++17 -Wall stringsteam.cpp -o stringstream
```



Write from the beginning(Default) **first time after initialization**, but afterwards add at the end(use position ptr)

Use **ate** to guarantee add at the end after initialization(Actually, it just **change the position ptr**)

Pay attention that every dtype has been **transform to string**(16.9->'16.9')

Cout reprint the **entire** string in the oss

Cout oss cause error

" Ounce " is a C-style string, but not a string

Stringstream is not connected to anything

Error-checking

```
int stringToInteger(const string& str) {
    // pass by reference and const
    istringstream iss(str);

    int result;
    iss >> result; // fail, fail bit is on
    if (iss.fail()) throw domain_error(...);

    char remain;
    iss >> remain; // check the remaining
    if (!iss.fail()) throw domain_error(...);
    return result;
}

// iss >> a returns iss, you can chain it
// Method 1
iss >> ch;
if (iss.fail()) {...};
// Method 2
if (!(iss >> ch)) {...};

// Simplify the first code
// stream using as boolean will be convert to stream.fail()
int stringToInteger(const string& str) {
    istringstream iss(str);
    int result; char remain;
    if (!(iss >> result) || (iss >> ch))
```

```
        throw domain_error(...);
    return result;
}
```

Exercise: getInteger

```
int getInteger(const string& prompt, const string& reprompt) {
    while (true) {
        cout << prompt;
        string line;
        if (!getline(cin, line))
            throw domain_error("...");

        istringstream iss(line);
        int val; char remain;

        if (iss >> val && !(iss >> remain))
            return val;

        cout << reprompt << endl;
        // the below is unnessary, cuz we create new iss every iteration
        // iss.clear(); // just set the good bit
        // iss.ignore(numeric_limits<streamsize>::max(), '\n')
    }
    return 0;
}
```

Types

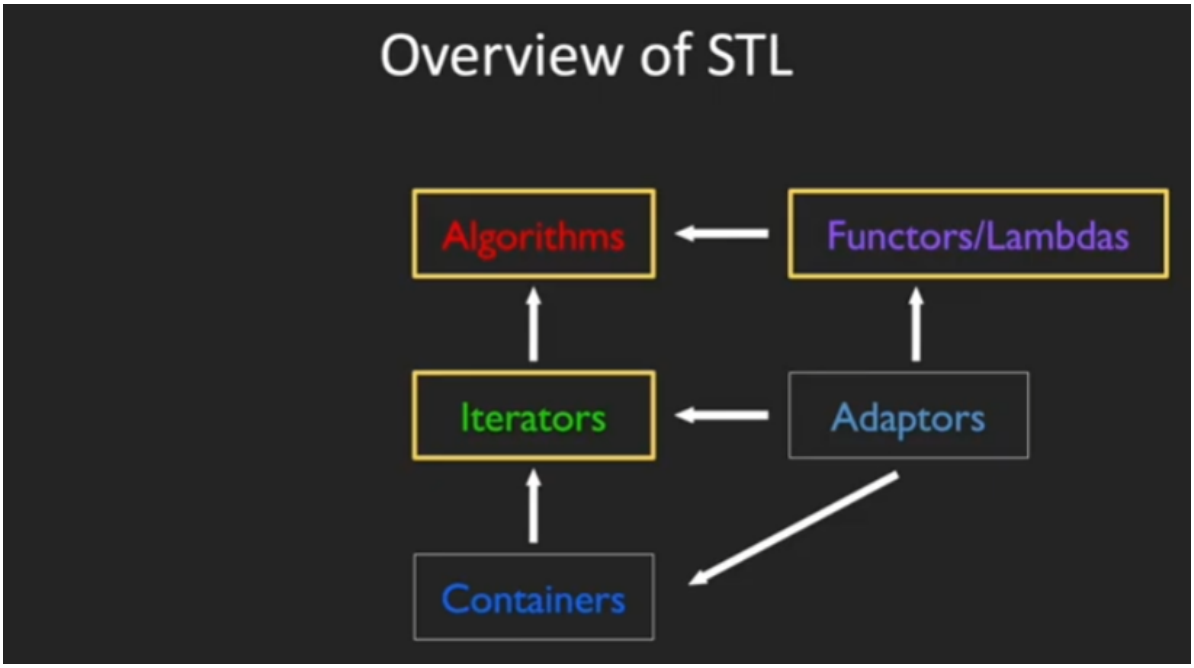
```
using name = ...(dtype);

// Auto
// C++11 use auto type to figure out the dtype
// can't be used as parameter dtype!!!!
auto func = [](auto i) {return i*2};
// It's a lambda function and has no type, so you have to use auto

// Structure
pair<int, int> findPriceRange(int dist); // extend to tuple if necessary
auto [min, max] = findPriceRange(dist); // C++17
// better way is structure
```

Initialization

```cpp
// Uniform Initialization
int main(){
    vector<int> vec{3, 1, 4, 5};
    Course now{"CS106L", {15, 30}, {16, 30}, {"Wang", "Zeng"}};
}

// initializer
vector::vector(initializer_list<T> init);

vector<int> vec1{3}; // vector = {3}
vector<int> vec2(3); // vector = {0, 0, 0}
```

Sequence Containers

Overview of STL



Sequence Containers

Container denotes containers storing other data.

- `std::vector<T>`
- `std::deque<T>`
- `std::list<T>`
- `std::array<T>`
- `std::forward_list<T>`

Vector

```cpp
vector<int> v;
vector<int> v(n);
vector<int> v(n, k);
v.push_back(k);
v.clear();
int k = v.at(i); // int k = v[i];
if (v.empty()) ...;
```

```
// index out of bound(assume vec has 2 elements)
vec.at(100);// throw an out-of-range exception
vec[100]; // undefined behavior: no error or warning for Mac, but error for Windows
```

```
// push_front is slow
// cuz it will shift all the elements back before push_front
```

Deque(double-ended queue)

Deque is faster to **push_front** and **push_back**, but not as fast as vector in **accessing middle elements**.

Container Adaptors

> Stack: Just limit the functionality of a vector/deque to only allow `push_back` and `pop_back`
>
> Queue: Just limit the functionality of a deque to only allow `push_back` and `pop_front`
>
> Plus only allow access to top element
>
> So, stacks and queues are known as **containter adaptors**.

Why we use stacks and queues?

- Express ideas in a direct way.
- Compartmentalize messy constructs.

Associative Containers

Have no idea of a sequence.

Data is accessed using the **key** instead of indexes.

- `std::map<T1, T2>`
- `std::set<T>`
- `std::unordered_map<T1, T2>`
- `std::unordered_set<T>`

> **map** and **set** is based on **ordering property** of keys. keys need to be comparable using `<` operator. faster to **iteratate** through a range of elements.
>
> **unordered_map** and **unordered_set** is based on hash function. You need to define how the key can be **hashed**. faster to **access individual elements** by key.

Map

```
// use key to access element
map[word]; // if not find word, it will create an entry and default initialize it
map.at(word); // if not find word, throw an exception

// check key existence
map.containsKey(key); // return boolean, C++20 or Stanford
map.count(response); // count the key appeared in the set, only return 0 or 1. But in multimap, it can be zero and any number above.
```

Set

Key and Value are equal.

Iterators

Iterators allow iteration over **any** data structure, whether it is ordered or not.

Iterators let us view non-linear collection in a **linear** manner.

```
// .begin() returns a iterator pointing to the first element of the set
set<int>::iterator iter = mySet.begin();

// dereference * operator
int a = *iter;

// ++
iter++;

// compare iter with .end() to check whether we hit the end
if (iter == mySet.end())
    return;
```

Map iterators

```
// pair
std::pair<string, int> p;
p.first = "Phone";
p.second = 123;

std::pair<string, int> p{"Phone", 123};
std::make_pair("Phone", 123);
{"Phone", 123};

// Map iterators
map<int, int> m;
map<int, int>::iterator i = m.begin();
map<int, int>::iterator end = m.end();
while (i != end) {
    cout << (*i).first << (*i).second << endl;
    i++;
}
```

Further Iterator Usages

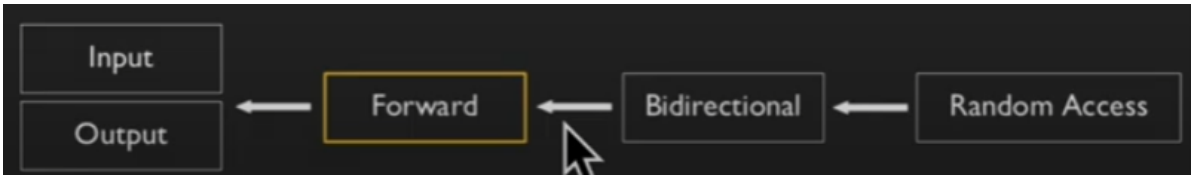
```
// sort
std::sort(vec.begin(), vec.end());
```

```cpp
// find key
// Actually, count function is based on find function
// Case 1
auto it = std::find(vec.begin(), vec.end(), elementToFind);
if (it == vec.end()) cout << "Not Found";
else
    cout << *it << endl;
// pay attention: end() points to the nonexist element after the last element

// Case 2
set<int> mySet{1, 3, 57, 137};
set<int>::iterator iter = mySet.lower_bound(2); // point to the smallest element strictly greater than 2
set<int>::iterator end = mySet.upper_bound(57); // point to the smallest element greater than or equal to 57
```

	[a, b]	[a, b)	(a, b]	(a, b)
begin	lower_bound(a)	lower_bound(a)	upper_bound(a)	upper_bound(a)
end	upper_bound(b)	lower_bound(b)	upper_bound(b)	lower_bound(b)

```cpp
// Range Based for Loop
map<string, int> myMap;
for (auto thing : myMap) {
    doSomething(thing.first, thing.second);
}
```

Iterator Types



- Input
- Output
- Forward
- Bidirectional
- Random Access

> The arrow in the above graph denotes: a kind of iterator is a kind of the iterator on the former's left, that is, a Random iterator is a kind of Bidirectional iterator, cuz the Random includes all the functionality of the Bidirectional.

All iterators share a few **common** traits:

- Can be created from existing iterator
- Can be advanced using `++`
- Can be compared with `==` and `!=`

Input:

- for sequential, **single-pass** input
- read only *i.e.* can be only dereferenced on the right side of expression

- `find` and `count`
- input streams

Output:

- sequential, single-pass output
- write only *i.e.* can be only dereferenced on the left side of expression
- `copy`
- output streams

Forward:

- combine input and output iterators, except can make multiple passes
- can read and write to (if not **const** iterator)
- `replace`
- `std::forward_list(sequence container, think of as singly-linked list)`

Bidirectional:

- same as forward but also support decrement operator --
- `reverse`
- `std::map, std::set`
- `std::list (double-linked list)`

Random Access:

- support increment and decrement arbitrary amounts using `+` and `-`
- `std::vector, std::deque, std::string`
- `Pointers`

Advanced Containers

Multimaps

Allow different values to the same key.

```
multimap<int, int> myMap;
myMap.insert(make_pair(3, 3));
myMap.insert({3, 12});
cout << myMap.count(3) << endl;
```

Template

```
// inform the compiler that T is a type
template <typename T>
pair<T, T> my_minmax(T a, T b) {
    if (a < b) return {a, b};
    else return {b, a};
}
```

Explicit Instantiation

```
// suggested
auto {a, b} = my_minmax<int, int>(1, 2);
```

Example

```
// basic
template <typename Collection, typename DataType>
int CountOccurences(const Collection<DataType>& list, DataType val) {
    int count = 0;
    for (auto iter = list.begin(); iter != list.end(); ++iter)
        if (*iter == val)
            ++count;
    return count;
}

// support iteration in the middle
template <typename InputIterator, typename DataType>
int CountOccurences(InputIterator begin, InputIterator end, DataType val) {
    int count = 0;
    for (auto iter = begin; iter != end; ++iter)
        if (*iter == val)
            ++count;
    return count;
}
```

```
// C++20: named requirements on the template arguments, explicitly interface. It will first check the requirements.
template <typename It, typename Type>
    requires Input_Iterator<It> && Iterator_of<It> && Equality_comparable<Value_type<It>, Type>
int countOccurences(It begin, It end, Type val);
```

Function and Lambdas

Predicate

Definition: Function which takes in some number of arguments and returns a boolean.

```
// Uniary Predicate (one argument)
bool isEqual3(int val) {
    return val == 3;
}

// Binary Predicate (two arguments)
bool isDivisibleBy(int dividend, int divisor) {
    return dividend % divisor == 0;
}

// Example
template <typename InputIterator, typename UniaryPredicate>
int CountOccurences(InputIterator begin, InputIterator end, UniaryPredicate predicate) {
```

```
    int count = 0;
    for (auto iter = begin; iter != end; ++iter)
        if (predicate(*iter))
            ++count;
    return count;
}
```

Lambda Functions

There exist two **problems**:

```
// First, it's kinda annoying that we have to write a separate functions for similar tasks
template <typename DataType>
inline bool lessThanTwo(DataType val){
    return val < 2;
}

template <typename DataType>
inline bool lessThanThree(DataType val){
    return val < 3;
}

// Second, what if the function needs information not available at compile time(eg. user input)
template <typename DataType>
inline bool greaterThan(DataType val) {
    return val >= limit;
}

int main() {
    int limit = getIntger("Minimum for A?");
    vector<int> grades = readStudentGrades();
    cout << countOccurences(grades.begin(), grades.end(), greaterThan);
    // It doesn't work!
}
```

Solutions:

```
// C++11 Method
class GreaterThan {
    public:
        GreaterThan(int limit):limit(limit) {}
        bool operator() (int val) {
            return val >= limit;
        }
    private:
        int limit;
}

int main() {
    int limit = getInteger("Minimum for A?");
    vector<int> grades = readStudentGrades();
    GreaterThan func(limit);
    countOccurences(grades.begin(), grades.end(), func);
}
```

```
18
19 // C++11 Lambda Function
20 int main() {
21     ...;
22     auto func = [limit](auto val) -> bool{
23         return val >= limit;
24     }
25     ...;
26 }
```

Lambda functions

```
1  auto func = [capture-clause](parameter) -> return-value(optional) {
2      //body
3  }
4  // Actually, the lambda function will be transfromed to a class (sucn as
   above example) by compiler
5
6  // reference is available
7  set<string> teas = {"black", "green"};
8  string banned = "boba";
9  auto likedByAvery = [&teas, banned](auto type) {
10     return teas.count(type) && type != banned;
11 }
12
13 // lazy way of capturing variables
14 // capture all by value, except teas is by reference
15 auto func1 = [=, &teas] -> return-value{
16     //body
17 };
18 // capture all by reference, except banned is by value
19 auto func2 = [&, banned] -> return-value {
20     // body
21 };
22 // suggest that don't use lambda as static or global func
```

STL Algorithms

```
1  // std::sort
2  auto compareRating = [](const Course& c1, const Course& c2) {
3      return c1.rating < c2.rating;
4  }
5  std::sort(numbers.begin(), numbers.end(), compareRating);
6
7  // std::nth_element
8
9  // std::stable_partition
10 // rearrange the sequence : 'true' at the begining, 'false' after, and the
   original order is kept
11 auto isDep = [](auto element) {
12     return element.name.size() > 2 && element.name.substr(0, 2) == "CS";
13 }
14 std::stable_partition(courses.begin(), courses.end(), isDep);
15
16 // std::copy_if
17 // copy "CS" courses to csCourses
```

```
std::copy_if(courses.begin(), courses.end(), csCourses.begin(), isDep); // but csCourses' size is limited, this way is not suitable
std::copy_if(courses.begin(), courses.end(), back_inserter(csCourses), isDep); // back_inserter:iterator adapter, take a iterator and enable it to freely insert backward

// std::copy
// for stream
std::copy(courses.begin(), coureses.end(), std::ostream_iterator<Course>(std::cout, "\n"));

// std::remove_if
std::remove_if(courses.begin(), courses.end(), isDep);
// Actually remove_if can't change the size of the container, so it can't indeed remove the elements. It puts the trash element at the end of the container waiting for erase.
// so, we combine erase and remove_if
v.erase(std::remove_if(v.begin(), v.end(), pred), v.end());

// std::find
// pay attention: every container has find function, and map.find() and set.find() is faster than std::find
```

STL Summary

```
#include <iostream>
#include <algorithm>
#include <string>
#include <fstream>
#include <cctype>
#include <numeric> // accumulate, inner_product
using std::cout;    using std::endl; // good practice
using std::vector;  using std::string;
string fileToString(ifstream &file) {
    string ret = "";
    string line;

    while (std::getline(file, line)) {
        /*
        for (char c : line)
            c = tolower(c); // <cctype>
        ret += c;
        */
        std::transform(line.begin(), line.end(), line.begin(), tolower);
        /*
        std::transfrom(line.begin(), line.end(), std::back_inserter(ret), tolower);
        */
        ret += line + " ";
    }
    return ret;
}

int countOccurences(const string& word, const string& text) {
    string toFind = " " + word + " ";
    // std::count can only count characters, but std::search can count strings
```

```
    auto curr = text.begin();
    auto end = text.end();
    int count = 0;
    while (curr != end) {
        auto found = std::search(curr, end, toFind.begin(), toFind.end());
        if (found == end) break;
        ++count;
        curr = find + 1;
    }
    return count;
}

vector<int> CreateFreqVec(const string& text) {
    vector<int> result;
    for (const auto& word : FEATURE_VEC) {
        result.push_back(countOccurences(word, text));
    }
}

int dotProduct(const vector<int>& vec1, const vector<int>& vec2) {
    return std::inner_product(vec1.begin(), vec1.end(), vec2.begin(), 0);
    // the 4th parameter is an extra constant to be added
}

int mag(const vector<int>& vec) {
    return std::sqrt(dotProduct(vec, vec));
}

double getSimilarity(const string &text1, const string& text2) {
    vector<int> freqVect1 = CreateFreqVec(text1);
    vector<int> freqVect2 = CreateFreqVec(text2);

    int dotProd = dotProduct(freqVect1, freqVect2);

    return dotProd / (mag(freqVec1) * mag(freqVec2));
}

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

Const

```
// One common but essential usage
void f(int x, int y) {
    if (x=1) // actually x == 1
        return y;
}

void f(const int x, const int y) {
    if (x=1) // syntax error!
        return y;
}
```

const function

```
class Student {
    public:
    string getName() const;
    private:
    string name;
}

Student::getName() const {
    return name;
}
```

const pointer

```
// const pointer to a non-const int
int * const p; // (*p)++; OK!
               // p++; Not Allowed!

// non-const pointer to a const int
const int *p; // p++; OK!
int const* p; // (*p)++; Not Allowed;

// const pointer to const int
const int * const p;
int const * const p;
```

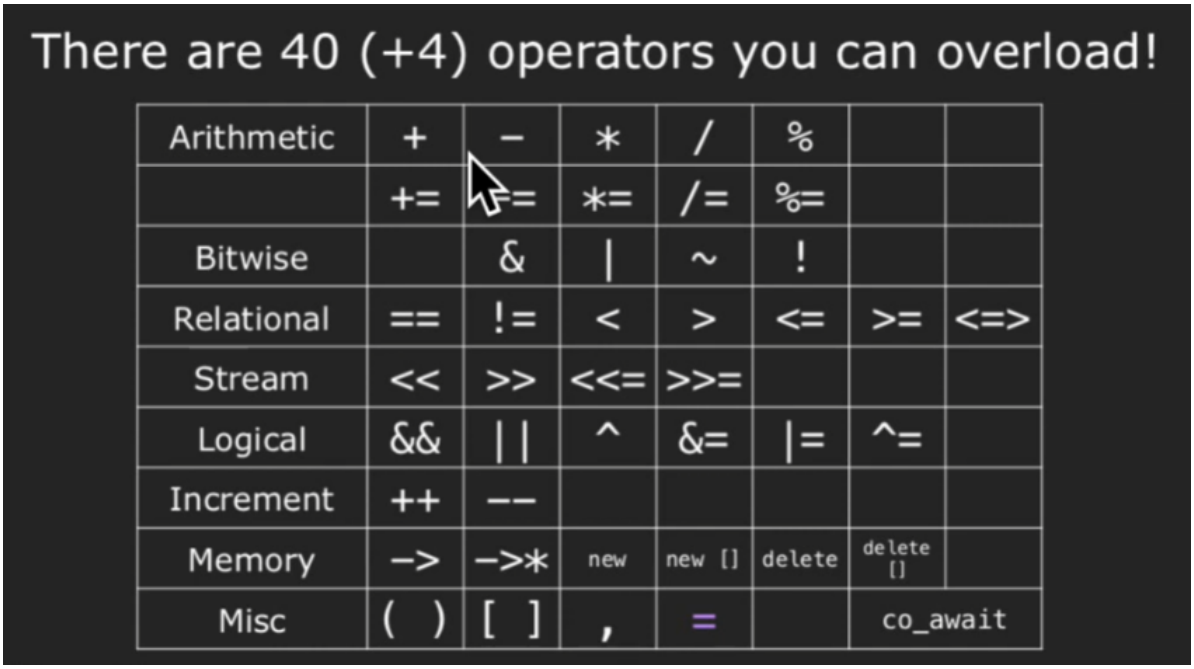
const iterators

```
// const vector<int>::iterator itr, acts like int* const itr
// To make an iterator read only, define a new const_iterator
vector v{1, 1234};
const vector<int>::iterator itr = v.begin();
++itr; // Not Allowed
*itr = 15; // Allowed

vector<int>::const_iterator itr = v.begin();
int value = *itr; // Allowed
++itr; // Allowed
*itr = 15; // Not Allowed
```

Challenge

```
const int* const myClassMethod(const int* const & param) const;
// 1. The function returns a pointer to a const int
// 2. The function returns a const pointer
// 3. The function takes in a pointer to a const int
// 4. The function taks in a const pointer
// 5. This is a const member function, i.e. this function can't modify any
variables of the 'this' instance
```

Operators




```cpp
// Case 1
cout.operator<<(v.operator[](0));
v.operator[](1).operator+=("!");

// Case 2
operator<<(cout, v.operator[](0));
operator+=(operator[](v, 1), "!");
```

```cpp
// Overload +=
vector<string>& vector<string>::opearator+=(const string& element) {
    push_back(element);
    return *this;
}
// const parameter accepts const and non-const, non-const parameter only accepts non-const

vector<string>& vector<string>::operator+=(const vector<string>& other) {
    for (string val : other)
        push_back(val);
    return *this;
}
// The return value is reference, otherwise it return a copy

// The below is equilvalent, so the push_back has its object. push_back is a member function
vect += "Hello";
vect.operator+=("Hello");
```

```cpp
// overload +
// Member function
StringVector StringVector::operator+(const StringVector& other) const{
    StringVector result = *this; // copy constructor
    for (const std::string& s : other)
        result.push_back(s);
    return result;
}
// 'this' may change the object, so we use const function
```

```
10
11  // Non-member function(better)
12  StringVector operator+(const StringVector& first, const StringVector&
    second) {
13      StringVector result = first;
14      for (const std::string&s :second)
15          result += s; // Fail! cuz StringVector has no copy constructor,
    compiler will create a default copy constructor, which will copy the array
    pointer pointing the original places! So the memory will be freed many
    times!
16      return result;
17  }
```

```
1  // Overload <<
2  ostream& operator<<(std::ostream& os, const Fraction& f) {
3      os << f.getNum() << "/" << f.getDenom();
4      // Or use friend to get access to private member
5      // declare in the Fraction class:
6      // friend operator<<(ostream& os, const Fraction& f);
7      // os << f.num << "/" << f.denom
8  }
```

Rules of member and non-member

- Some operators must be implemented as members (eg. [], (), ->, =)
- Some must be implemented as non-members (eg. <<, if you are writing class for rhs, not lhs)
- If unary operator (eg. ++), implement as member
- If binary operator and treats both operands equally (eg. both unchanged) implement as non-member (maybe friend). Example: +, <
- If binary operator and not both equally (changes lhs) implement as member (allow easy access to lhs private members). Examples: +=

POLA(Principle of Least Astonishment)

- Design operators primarily to mimic **conventional** usage.

> a, 3, 4, 5; // What is it? Non-sense.

- Use **nonmember** functions for **symmetric** operators.

> a + 1 == b; // Work.
> 1+ a == b; // Fail!

- Always provide all out of a set of **related operators**.

> a < 1; // Work.
> 1 >= a; // Fail!

- Always think about **const** vs. **non-const** for member functions. (const object can only call const functions)

Special Member Functions

Constructor/Copy

```
// constructor
// If you create a const object, the member elements can't be change, so how to construct?
// Use Initialization List
StringVector::StringVector():
    logicalSize(0), allocatedSize(kInitialSize) {
        elements = new std::string[allocatedSize];
    }

// copy construction: object is created as a copy of an existing object
StringVector::StringVector(const StringVector& other) noexcept:
    logicalSize(other.logicalSize), allocatedSize(other.allocatedSize) {
    elements = new std::string[allocatedSize];
    std::copy(other.begin(), other.end(), begin());
}
// noexcept: don't throw any exception(C++11)

// copy assignment: existing object is replaced as a copy of another exising object
StringVector& StringVector::operator=(const StringVector& other) {
    if (this != &other) {
        // *this == other fails! equal sign should be overloaded

        delete[] elements;
        logicalSize = other.logicalSize;
        allocatedSize = other.allocatedSize;
        elements = new std::string[allocatedSize];
        copy(other.begin(), other.end(), begin());
    }

    return *this;
}
// Can't use initialization list, cuz it's not an initialization

// You can prevent copies!
class LoggedVector {
    public:
    LoggedVector(const LoggedVector& rhs) = delete;
    LoggedVector& operator=(const LoggedVector& rhs) = delete;
}
```

tricky cases

```
StringVector function(StringVector vec0) {
    // vec0: copy construction
    StringVector vec1; // defalut constructor
    StirngVector vec2{"Hello"}; // regular constructor
    StringVector vec3(); // declare function!
    StringVector vec4(vec2); // copy constructor
    StringVector vec5{}; // default constructor
    StringVector vec6{vec3 + vec4}; // copy constructor
    StringVector vec7 = vec4; // copy constructor
    vec7 = vec2; // copy assignment
    return vec7; // copy constructor
}
```

Rule of three

If you explicitly define (or delete) a **copy constructor**, **copy assignment**, or **destructor**, you should define (of delete) all **three**.

Rule of Zero

If the **default** operations work, **don't** define your own custom ones.

Move Semantics

emplace_back(C++17)

```
vector<President> elections;
elections.emplace_back("Nelson Mandela", "South Africa", 1994);
```

problem of copying

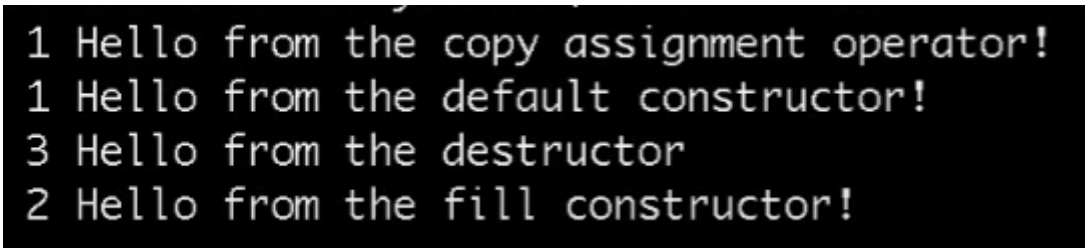
There is too much copy!

```
// Count the times of each function
StrVector readNames(size_t size) {
    StrVector names(size, "Ito");
    return names;
}

int main() {
    StrVector name1 = readNames(54321234);
    // readNames:
    // fill(regular) constructor
    // copy constructor(return)
    // destructor(in readNames function)
    // destructor(out readNames function)
    // name1 = readNames:
    // copy constructor
    StrVector name2;
    // default constructor
    name2 = readNames(54321234);
    // readNames:
    // fill(regular) constructor
    // copy constructor(return)
    // destructor(in readNames function)
```

```
    // destructor(out readNames function)
    // name2 = readNames:
    // copy assignment
    return 0;
    // destructor(name1)
    // destructor(name2)
}
```

Copy elison(C++17)

Compiler will skip some copy function. For example, above `readNames` function will create a StrVector in the `main` function, skipping the copy at the time of `return`



However, we can still optimize the **copy assignment** part, that is, `name2 = readNames(5431234)`. The object created by `readNames(54321234)` is temporary!

lvalues and rvalues

> lvalues: expression with a name(identity). Can find address using address-of operator(&)
>
> rvalues: expression with no name(identity). Temporary values. Cannot find address using address-of operator(&)

```
int val = 2; // val:lvalue; 2: rvalue
int *ptr = 0X12345678; // ptr:lvalue; 0X12345678: rvalue
vector<int> v1{1, 2, 3};// v1:lvalue; {1, 2, 3}:rvalue

auto v4 = v1 + v2; // v1: lvalue; v2: lvalue; v1 + v2: rvalue; v4: rvalue
v1 += v4; // v1: lvalue; v4: lvalue
size_t size = v.size(); // v:lvalue; v.size():rvalue
val = static_cast<int>(size); // size: lvalue; static_cast<int>(size): rvalue
v1[1] = 4 * i; // 4 * i: rvalue
ptr = &val; // &val: rvalue
v1[2] = *ptr; // *ptr: lvalue!
```

```
// rvalue reference and lvalue reference
auto& ptr2 = ptr; // ptr2: lvalue reference
auto&& v4 = v1 + v2;// v4: r-value reference, extend the lifetime of r-value. Everytime you change v4, v1 + v2 is changed
auto& ptr3 = &val;// ERROR: can't bind lvalue reference to rvalue
auto&& val2 = val;// ERROR: can't bing rvalue reference to lvalue
const auto& ptr3 = ptr + 5;// OK: CAN bind const lvalue reference to rvalue(Why?because const guarantees you won't change ptr+5)
```

```
void nocos_Lref(vector<int>& v);
void const_Lref(const vector<int>& v);
void nocos_Rref(vector<int>&& v);

nocos_Lref(v1); // OK
nocos_Rref(v1); // ERROR
nocos_Lref(v2 + v3); // ERROR
const_Rref(v2 + v3); // OK
nocos_Rref(v2 + v3); // OK
```

- An object that is an lvalue is NOT disposable, so you can copy from, but definitely cannot move from.
- An object that is an rvalue is disposable, so you can either copy or move from.

Move constructor/assignment

- Move constructor (create new from existing r-value)
- Move assignment (overwrite existing from existing r-value)

```
StringVector();
StringVector(const StringVector& other) noexcept;
StringVector& operator=(const StringVector& rhs) noexcept;
~StringVector();

StringVector(StringVector&& other) noexcept;
StringVector& operator=(StringVector&& rhs) noexcept;

// move constructor
StrVector::StrVector(StrVector&& other) noexcept:
    elems(other.elems),
    logicalSize(other.logicalSize),
    allocatedSize(other.allocatedSize)  {
    other.elems = nullptr;
}

// move assignment
// Not Perfect Method:
StrVector& StrVector::operator=(StrVector&& rhs) noexcept: {
    if (this != &rhs) {
        delete [] elems;
        logicalSize = rhs.logicalSize;
        elems = rhs.elems;
        rhs.elems = nullptr;
    }
}

// Why not Perfect? Example:
Axess(Axess&& other) : students(other.students) {}
Axess& operator=(Axess&& rhs) {
    student = rhs.students;
}
// When the class incorporate another class, then "student = rhs.students"
call the copy function, because rhs is rvalue but rhs.students is lvalue!
// optimization
Axess(Axess&& other) : students(std::move(other.students)) {}
Axess& operator=(Axess&& rhs) {
```

```
    students = std::move(rhs.students);
}
// move changes everything it takes to rvalue

// Perfect Method
StrVector::StrVector(StrVector&& other) noexcept:
    elems(std::move(other.elems)),
    logicalSize(std::move(other.logicalSize)),
    allocatedSize(std::move(other.allocatedSize))  {
    other.elems = nullptr;
}
StrVector& StrVector::operator=(StrVector&& rhs) noexcept: {
    if (this != &rhs) {
        delete [] elems;
        logicalSize = std::move(rhs.logicalSize);
        elems = std::move(rhs.elems);
        rhs.elems = nullptr;
    }
}
```

Example: Swap

```
template <typename T>
void swap(T& a, T& b) {
    T temp = std::move(a);
    a = std::move(b);
    b = std::move(temp);
}
```

Namespaces

There are `std::` and `StringVector::`

There exists **namespace clash**

```
namespace Lecture {
int count(const vector<int>& v) {
    int ctr = 0;
    for (auto i : v) {
        if (i == 1)
            ctr++;
    }
    return ctr;
}
}

int main() {
    ...;
    cout << "Lecture count:" << Lecture::count(v);
    return 0;
}
```

Inheritance

> Template is implicit interface, while inheritance is explicit one.

Terminology

- **Base** (aka **superclass** or **parent**) class
- **Derived** (aka **subclass** or **child**) class

Virtual Function

```
class Drink {
public:
    virtual void make() = 0;
    // pure virtual function:
    // the class inheriting the class has to define the 'make' function
};

class Tea : public Drink {
public:
    void make() {
        // implement
    }
};
```

```
// Abstract class: has at least one pure virtual function
// Abstract classes can't be instantiated
class Base {
public:
    virtual void foo() = 0; // pure
    virtual void foo2(); // non-pure
    void bar() = {return 42;}; // regular function
}
```

Member functions

```
// constructor
class Derived : public Base {
    Derived() : Base(args), /*others*/ {
        // rest of constructor
    }
};

// destructor
// if you intend to make your class inheritable, make your destructor virtual. Otherwise memory leaks can be caused
virtual ~Base() {};
```

Access Specifiers

- private
  - Can only be accessed by this class
- protected
  - Can only be access by this class or derived classes
- public
  - Can be access by anyone

Example

```cpp
#include <iostream>
using std::cout; using std::enl;

class Drink {
public:
    Drink() = default;
    Drink(std::string flavor) : _flavor(flavor) {}
    virtual void make() = 0;
    virtual ~Drink() = default;
private:
    std::string _flavor;
}

class Tea : public Drink {
public:
    Tea() = default;
    Tea(std::string flavor) : Drink(flavor) {}
    ~Tea() = default;

    void make() {
        cout << "Made tea from the Tea class" << endl;
    }
}
int main() {
    Tea t("red");
    t.make(); // If parent class also defines make function, then it will still call the subclass' make function
    // t.Drink::make(); // this can forcibly call the superclass' make function
}
```

Template vs. Derived Class

template is **static** polymorphism (compile time, create a lot of versions of code, **code bloat**), while derived class is **dynamic** polymorphism (runtime).

Cast

```cpp
int a = (int)b;
int a = int(b);
int a = static_cast<int>(b); // Best
```

Template Classes

```cpp
template <class T, class Container = std::vector<T>>
class Priority_Q {
public:
    Priority_Q() = default;
    ~Priority_Q() = default;

    T pop() const {}
    void pop() {}
```

```
    void push(T val) {}
private:
    Container _heap;
    size_t _count{0};
}

int main() {
    Priority_Q<vector<string>, vector<vector<string>>> q;
    q.push({"Fruit"});
}

// C++20: concept
```

RAII(Resource Acquisition Is Initialization)

Introduction

```
// How many potential code paths here?
string EvaluateSalaryAndReturnName(Employee e) {
    if (e.Title() == "CEO" || e.Salary > 100000)
        cout << e.First << "" << e.Last() << "is overpaid" << endl;
    return e.First << "" << e.Last();
}
// 1. false false return
// 2. false true return
// 3. true return
// 4. Employee e can throw an exception
// 5. Title()
// 6. ==
// 7. ...
// 23. return string can throw exception
```

If the program throws an exception, then it may not release the memory. You can ban exception to fix the problem, but we have a better way.

	Acquire	Release
Heap Memory	new	delete
Files	open	close
Locks	try_lock	unlock
Sockets	socket	close

RAII/SBRM/CADRE

> - RAII: Resource Acquisition Is Initialization
> - SBRM: Scope Based Memory Management
> - CADRE: Constructor Acquires, Destructor Releases (Best Description!)

```
void printFile() {
    ifstream input("hamlet.txt");

    string line;
    while (getline(input, line))
        cout << line << endl;
    // no close call needed!
}// stream destructor, releases access to file
```

Smart Pointer

- `std::unique_ptr`

```
// Unique_ptr: owns its resource and deletes it when the object is destroyed
// Unique_ptr class disallows copying, by deleting the copy constructor and copy assignment
void rawPtrFn () {
    std::unique_ptr<Node> n(new Node);
    // do some stuff with n
    // Freed!
}
```

- `std::shared_ptr`

```
// Resource can be stored by any number of shared_ptrs
// Deleted when none of them point to it
{
    std::shared_ptr<int> p1(new int);
    // Use p1
    {
        std::shared_ptr<int> p2 = p1;
        // Use p1 and p2
    }
    // Use p1
}
// Freed!
```

> unique_ptr and shared_ptr store an int that keeps track of the number currently referencing that data. Frees the resource when reference count hits 0

- `std::weak_ptr`

This ptr doesn't contribute to the reference count with some special implements

```
// Better smart ptr creators!
std::unique_ptr<Node> n = std::make_unique<Node>();
std::shared_ptr<Node> n = std::make_shared<Node>();
```

Multithreading

RAII

```
void cleanDatabase(mutex& dbLock, map<int, int>& database) {
    lock_guard<mutex> lg(databaseLock);
}
```

Classes

- atomic

Use atomic_int rather than int! Those atomic datatype guarantees the validity of **atomic operation**. (reference to the STL library)

- mutex

Added: unique_lock guarantees that you can freely unlock before the automatic unlock of the mutex, and when out of scope, the mutex will be automatically unlocked.

Mutex can be normal, timed, **recursive**(multiple locks).

- condition_variable

Two threads commuicate with each back and forth while they are running.

- future

Asynchronous functions.

Examples

```
// Case 1
void greet(int id) {
    cout << "My id is " << id << endl;
}

int main() {
    cout << "Greeting from my threads ..." << endl;
    std::thread thread1(greet, 1);
    std::thread thread2(greet, 2);
    cout << "All greetings done!" << endl;
    return 0;
}
// Result:
// Greeting from my threads ...
// All greetings done!
// My id is
// My id is 1
// t2rminate ... (unpredictable, because cout isn't atomic)

// Case 2
void greet(int id) {
    std::this_thread::sleep_for(std::chrono::seconds(5));
    cout << "My id is " << id << endl;
}
// Result:
// Greeting from my threads ...
// All greetings done!
// terminate ...
```

```cpp

// Case 3
std::mutex mtx;
void greet(int id) {
    std::lock_guard<std::mutex> lg(mtx);
    cout << "My id is " << id << endl;
}
int main() {
    cout << "Greeting from my threads ..." << endl;
    std::thread thread1(greet, 1);
    std::thread thread2(greet, 2);
    thread1.join();
    thread2.join(); // wait for two threads to finish
    // Pay Attention: any thread can finish earlier!
    cout << "All greetings done!" << endl;
    return 0;
}

// Case 4
const size_t threadNum = 10;
int main() {
    vector<std::thread> threads;
    for (size_t i = 0; i < threadNum; i++)
        threads.push_back(std::thread(greet, i));
    for (std::thread& t : threads) // reference is essential!
        t.join();
}
```

Where to go