

C++ Exam Cheat Sheet

This C++ cheat sheet is adapted from <https://github.com/mortennobel/cpp-cheatsheet>.
All statements assume using namespace std (you may still use std::).
As of C++20 standard.
PROG 19/20, FEUP
Scenic Time

Preprocessor

```
#include <stdio.h>           // Insert standard header file
#include "myfile.h"          // Insert file in current directory
#define X some text          // Replace X with some text
#define F(a,b) a+b           // Replace F(1,2) with 1+2
#undef X                      // Remove definition

#if defined(X)                // Conditional compilation (#ifdef X)
#else                          // Optional (#ifndef X or #if !defined(X))
#endif                        // Required after #if, #ifdef
```

Literals

```
255, 0377, 0xff             // Integers (decimal, octal, hex)
2147483647L, 0x7fffffffL     // Long (32-bit) integers
123.0, 1.23e2                // double (real) numbers
'a', '\141', '\x61'          // Character (literal, octal, hex)
'\n', '\\', '\'', '\"', '\0' // Newline, backslash, single quote, double quote, null terminator
"string"                     // Same as {'h','e','l','l','o','\0'};
true, false                   // bool constants 1 and 0
nullptr                       // Pointer type with the address of 0
```

Types; casts; declarations

Know the standard C types:

```
int8_t, uint8_t, int16_t,    // char, unsigned char, short,
uint16_t, int32_t, uint32_t, // unsigned short, int, unsigned int
int64_t, uint64_t           // long, unsigned long
```

The possible conversion mechanisms:

```
dynamic_cast<T>(x)           // Converts x to a T, checked at run time
                              // T must be a pointer or reference
                              // May convert between classes
                              // Fail in conversion returns nullptr

static_cast<T>(x)            // Converts x to a T, for simple data types
                              // Alerts when possible truncation issues (which C-style casts do not do)
                              // Does not work with classe types

reinterpret_cast<T>(x)       // Interpret bits of x as a T
const_cast<T>(x)             // Casts away const
```

Then start storing your data:

```
int x;                       // Declare x to be an integer (value undefined)
int x=255;                    // Declare and initialize x to 255
```

```

int a, b, c;           // Multiple declarations

int a[10];             // Array of 10 ints (a[0] through a[9])

int a[10] = {2};       // Only first element is 2
                       // Other elements contents depend on the implementation

int a[]={0,1,2};       // Initialized array (or a[3]={0,1,2}; )
                       // You may deduce size of this by doing sizeof(a)/sizeof(int)
                       // However sometimes you lose this ability due to pointer decay
                       // Always pass size as parameter

int a[][2]={{1,2},{4,5}}; // Array of array of ints (only first dimension can be deduced!)
char s[]="hello";      // C String (6 elements including '\0'); same as char* s = "hello"

int *p,*a;             // p and a are pointers to ints
                       // if you did int* p,a; a would not be a pointer!

char* s = "hello";     // s points to first element of array
void* p = nullptr;     // Address of untyped memory (nullptr is 0)
int& r = x;            // r is a reference to (alias of) int x

int* r = x;            // r is the memory location of x

                       // if x is an array of allocated memory (unallocated: undefined behaviour)
                       // do r++ to jump sizeof(int) bytes in the memory (access next index)
                       // you can still do r[index] after this declaration

typedef String char*;  // String s; means char* s;
                       // same as using String = char*;

const int c = 3;       // Constants must be initialized, cannot assign to (read-only)
constexpr int d;      // Same but d must be known at compile time (e.g. d cannot be a parameter)

auto it = m.begin();  // Auto deduces type of variable (in this case an iterator)

Always read right to left:

const int* p=a;        // p is a pointer to a int that is constant (might point elsewhere)
int* const p=a;        // p is a constant pointer to an int (contents might change)
const int* const p=a;  // Both p and its contents are constant
const int& cr=x;       // cr is a reference (alias) of an int that is constant

Also be aware of the objects lifetimes and memory features:

int x;                 // Declare x in the stack. It's automatically popped at end of scope
static int x;          // Global lifetime even if local scope; cannot be used outside with extern
extern int x;          // Compiler is able to access x declared in other translation units

```

Expressions

```

T::X                  // Name X defined in class T
N::X                  // Name X defined in namespace N
::X                   // Global name X

t.x                   // Member x of struct or class t
p->x                  // Member x of struct or class or union that p points to
                       // Dereference to use same syntax as above: (*p).x

```

```

a[i]                // i'th element of array a
f(x,y)              // Call to function f with arguments x and y
T(x,y)              // Object of class T initialized with x and y
typeid(x)            // Returns reference to object of type of x (access name with .name())
decltype(x)          // Get type of x
// Useful for generic programming or to pass objects as their type

sizeof(x)            // Number of bytes used to represent object x
sizeof(T)            // Number of bytes to represent type T
x++                  // Add 1 to x, evaluates to original x (postfix)
x--                  // Subtract 1 from x, evaluates to original x (postfix)
++x                  // Add 1 to x, evaluates to new value (prefix)
--x                  // Subtract 1 from x, evaluates to new value (prefix)
~x                   // Bitwise complement of x
!x                   // true if x is 0, else false (1 or 0 in C)
&x                   // Address of x
*p                   // Contents of address p (*&x equals x)
(T) x                // Convert x to T (obsolete, use .._cast<T>(x))

x * y                // Multiply
x / y                // Divide (return same type of operands - 3/2 is 1)
x % y                // Modulo (result has sign of x, unlike python!)

x + y                // Add, or &x[y]
x - y                // Subtract, or number of elements from *x to *y
x << y                // x shifted y bits to left (x * pow(2, y))
x >> y                // x shifted y bits to right (x / pow(2, y))

x < y                // Less than
x <= y               // Less than or equal to
x > y                // Greater than
x >= y               // Greater than or equal to

x & y                // Bitwise and (3 & 6 is 2)
x ^ y                // Bitwise exclusive or (3 ^ 6 is 5)
x | y                // Bitwise or (3 | 6 is 7)

cond1 && cond2        // cond1 and cond2
// if cond1 is false, cond2 is not evaluated
// to force cond2 execution, do cond1 & cond2

cond1 || cond2        // cond1 or cond2
// if cond1 is true, cond2 is not evaluated
// to force cond2 execution, do cond1 | cond2

x = y                // Assign y to x, returns new value of x
x += y               // x = x + y, also -= *= /= <=> >= &= /= ^=

x ? y : z            // y if x, else z (ternary operator)

```

Statements

```

int x; x=y;          // Declarations and assignments are statements
;                    // Empty statement

{
    int x;            // x is not accessible outside its scope
}

```

```

if (x) a;           // if x is true (not 0), evaluate a
else if (y) b;      // if not x and y (optional, may be ed)
else c;             // if not x and not y (optional)

while (cond) a;     // Repeat while cond is true
                  // ints may be evaluated as conditions (0 false; else true)

for (initial; cond; inc) a; // Equivalent to: initial; while(cond) {cond; inc;}

for (t elem : container) a; // Range-based for loop - do a; for each COPY of element in container

do a; while (x);    // Equivalent to: a; while(x) a;

switch (x) {        // x must be integer known at compile time
    case X1: a;      // if x == X1, do a; b; c; (everything is executed until break)
    case X2: b;      // if x == X2, do b; c; (use break if c; is not desired)
    default: c;      // Same as if (true)
}

break;             // Jump out of while, do, or for loop, or switch
continue;          // Jump to bottom of while, do, or for loop
return x;           // Return x from function to caller

```

Functions

```

int f(int x, int y); // f is a function taking 2 ints BY COPY and returning int BY COPY
Player& f(Player &x); // f is a function taking 1 Player BY REFERENCE and returning it BY REFERENCE
                    // make sure that the return object does not get popped out of stack/scope!

int f(int x);        // overload of f (change parameters, return type alone is not enough)
void f();            // f is a procedure taking no arguments
void f(int a=0);     // Default parameters always come after non-default ones
f();                // Default return type is int (bad practice to hide this info)
inline f() {statement;} // Optimize for speed when defined in this translation unit
f() { statements; }  // Function definition (must be global)
T operator-(T x);    // allows T a; T b = -a;
T operator++(int);   // postfix ++ or -- (parameter ignored)
extern "C" {void f();} // f() was compiled in C

```

Lambda functions - quick, disposable actions

[] is the list of accessible variables from the outer scope. Pass & to allow access to all.

```

auto isMove = [](const string& str){ // must be auto; return type is deduced
    return str.size() == 2 && isupper(str.at(0)) && islower(str.at(1));
};

string candidate1("Ab"), candidate2("3A");

// Is one of them a valid move?
cout << isMove(candidate1) || isMove(candidate2); // print 1

```

Main function

The main functions return the error code, 0 meaning all ok and up something went wrong. At any time, use `exit(intError)` to stop the program and return `intError` in main.

```
int main() { statements... }    // main is the starting point of any program

// One can make main recognize command line arguments:
int main(int argc, char* argv[]) { statements... }
    //argc -> number of arguments when running the program (default 1 - program name)
    //argv -> command strings (char**)
```

Unions

Memory location of all members is the same, its size being determined by the largest of the data members. Only one may be used at given time.

```
union Numbers
{
    int x;
    double d;
};

union Numbers n; // if you do union Numbers* n, access by n->x
n.x = 2; // n.d also gets value 2
```

Enums

Enums are a bit magical, being like declaring multiple integers that are related. Very useful to make switch readable.

```
enum weekend {SAT,SUN,MON};    // weekend is a type wrapping global integer values: SAT=0, SUN=1, MON=2
enum weekend {SAT=6,SUN=7};    // Explicit representation as int
enum weekend day = SAT;        // day is a variable of type weekend
                                // must be assigned to its name, not its value

int anotherDay = 6;
switch (anotherDay){
    case (SAT):
        cout << "Today is Saturday\n"; // this gets executed
        break;
    case (SUN):
        cout << "Sunday it is\n";
        break;
    default:
        cout << "Time to work...\n";
}
```

Classes; operator overloading

Define the class in a header file:

```
#pragma once    // Header files use this directive to avoid conflicting symbols

class T {
private:        // A user defined type
                // Section accessible only to T's member functions
protected:    // Also accessible to classes derived from T
public:         // Accessible to all
```

```

int x;                // Member data
void f();             // Member function

T& g() {return *this;} // Inline member function
                        // Return *this to allow chains of setters/getters

void h() const;       // Does not modify any data members

int operator+(int y); // t+y means t.operator+(y)
int operator-();      // -t means t.operator-()

T& operator++();      // ++t means t.operator++()
T operator++(int);    // t++ means t.operator++(int)
                        // int is a dummy parameter meaning postfix operator

// You cannot overload << and >> for streams inside the class definition
// Check `iostream` for how to do this via a function

T(): x(1) {}          // Constructor with member initialization list
                        // Class attributes are initialized before the body of the constructor
                        // So if instead of using lists you did T() {x=1;}
                        // x would be initialized with nothing and then assigned - what a waste!

                        // A default constructor (no parameters) is generated automatically
                        // Unless you define other constructors yourself
                        // If in that case you still want the compiler generated constructor
                        // Do explicitly: T() = default;

                        // The compiler also generates T(T otherWithSameType) automatically
                        // This may raise issues such as unwanted aliasing
                        // eg. there are pointers as attributes of the class

T(const T& t): x(t.x) {} // Copy constructor (still a constructor... initialize T attributes)
                        // Again, the compiler might take care of this with side effects

T& operator=(const T& t)
{x=t.x; return *this;} // Assignment operator
                        // Again, the compiler might take care of this with side effects

~T();                // Destructor (automatic cleanup routine)
                        // Put manual memory deallocations here if needed

explicit T(int a);    // Allow T t=T(3) but not T t=3
T(float x): T((int)x) {} // Delegate constructor to T(int)

int operator int() const
{return x;}           // Allows int(t)

int operator()(int a) const
{return x+a;}         // One can now do T obj; int sumObj = obj(a);
                        // Functors are useful to pass to STL algorithms since they hold state

friend void i();      // i() has private access (friendship is given by T, not claimed by i())
friend class U;       // Members of class U have private access
static int y;         // Data shared by all T objects
static void l();      // Shared code. May access y but not x
};

```

Then define member functions and use the class in implementation files:

```
#include "T.h"                // Use this directive to access the class definitions

void T::f() {                  // Code for member function f of class T
    this->x = x;}              // this is address of self (means x=x;)

int T::y = 2;                  // Initialization of static member (required)
T::l();                        // Call to static member

T t = 3;                       // Create object t implicit call constructor
                                // Same as T t = T(3) -> implicit conversion
                                // To cancel this conversion use explicit before the constructor

t.f();                         // Call method f on object t
```

Note that operators might also be overloaded outside the class, via a function:

```
bool operator==(const Date& d1, const Date& d2){
    return d1.getYear() == d2.getYear()
    && d1.getMonth() == d2.getMonth()
    && d1.getDay() == d2.getDay();
}
// Same as bool Date::operator==(const Date& other) const {conditions};;
```

Class inheritance and polymorphism

Mind the access between base and child class members:

| Inheritance form | Public in base | Protected in base | Private in base |
|------------------|--------------------|--------------------|-----------------|
| public | public in child | protected in child | - |
| protected | protected in child | protected in child | - |
| private | private in child | private in child | - |

Create a child class according to your needs:

```
struct T {                    // Equivalent to: class T { public:
    T();                      // Class constructor

    virtual void i();          // Virtual -> may be overridden at run time by derived class
                                // Form of polymorphism at run time

    virtual void g(int x)=0;    // Must be overridden (pure virtual)
                                // Doing this T becomes an abstract class that cannot be instantiated!
};

class U: public T {           // public is the inheritance form
public:
    U(): T();                 // Base class constructors are not inherited; use delegation like this
    void g(int x) override;    // Explicitly override method g (do not use override in the definition)
    void g(int x);             // Same as above but compiler does not check if g is virtual in T
    int y;                    // Specific of U, will get sliced away if U is interpreted as a T
};
```

To solve data slicing problems use virtual functions and dynamic_casts:

```
class FeupPerson {
public:
    FeupPerson(string name): _name(name){};
```

```

    string getName() const {return _name;}
    virtual int getId() const{return 0;}; // may be overridden by Student, making this an abstract Class

                                   // making virtual getId() const = 0 would make this pure virtual
                                   // in that case, instantiation of FeupPerson objects would be denied

protected:
    string _name;
};

class Student : public FeupPerson {
public:
    Student(string name, int id): FeupPerson(name), _id(id) {};
                                   // Cannot instantiate _name here; delegate base constructor
    int getId() const override {return _id;};
private:
    int _id;
};

FeupPerson p("Compact");
Student s("Elegant", 2019);

set<FeupPerson*> mySet; // Polymorphic since FeupPerson might be a Student as well
                       // FeupPersons are tested for equality via memory location
                       // Go to `set` for more info

mySet.insert(&p);
mySet.insert(&s); // Student* implicitly becomes FeupPerson*

for (const auto& p: mySet){
    if (dynamic_cast<Student*>(p) != nullptr){ // if conversion to Student is successful
        cout << "This is a student! \n";
    }
    cout << "id: " << p->getId() << endl;
        // the correct version of the member function (returning 0 or _id) is called
        // this is because of the virtual keyword
}

p = s; // possible but data is sliced away - slicing problem (s=p is illegal)

```

Templates - generic programming

Like overloading, this kind of polymorphism is compile time defined. “Overload” a class/function/method for all types:

```

template <class T> // Same as template <typename T>
T f(T t);

template <class T>
class X {
    X(T t);
};

template <class T>
X<T>::X(T t) {}

template <class T, unsigned long n=0> // Template with default parameters
T f(array<int,n> myArray);

```

Then use them for your specific needs:


```
X<int> x(3); // Declare an object of type "X of int"
```

Namespaces - avoid naming conflicts

```
namespace N {class T {};} // Hide name T
N::T t; // Use name T in namespace N
using namespace N; // Make T visible without N::
```

Exceptions - signal errors

```
#include <stdexcept> // to access STL exception classes

try {
    doSomething(); // this may throw an exception

    throw logic_error("received negative value");
        // you may throw an object yourself (in this case a std::exception)
        // runtime_error is another common std::exception
        // base class std::exception can't be constructed with string or char*
}
catch (exception t) { // you could catch any object type (in this case a std::exception)
    cout << t.what() << endl; // print error message describing exception
    throw; // throw t again if you want the program to crash
}
catch (...) { doSomething(); } // catch any thrown object (if previous catch didn't)
```

string - variable sized character container (vector-like; random iteration)

```
#include <string> // Include string (std namespace)

string s1, s2="hello"; // Create strings
string repeated('c',4): // Same as string("cccc");
s1.size(); // Number of characters ('\n' is not counted)
s1 += " world"; // Concatenation with other string
s1 += '!'; // Concatenation with char (same as s1.push_back('!'))
int n = stoi("127"); // Converts string to integer
s1 == "hello world" // Comparison, also <, >, !=, etc.
s1[0]; // 'h'; use s1.at(0) to be able to handle out of bounds exceptions
s1.substr(m, n); // Substring of size n starting at s1[m]
s1.substr(m); // Substring from s1[m] until end of s1
s1.c_str(); // Convert to const char*, restricted lifetime
s1 = to_string(12.05); // Converts number to string
getline(cin, s); // Read line ending in '\n'
s1.find("hello"); // Pointer to first char of found substring, if not found string::npos
```

stringstream (most methods are inherited from ios; allows input and output)

```
#include <sstream> // Include stringstream (std namespace)

stringstream ss("Hello World"); // same as stringstream m; m << "Hello" << " World";
ss.str(); // Return "Hello World"
ss << 127; // operator << is overloaded for number types
while (ss >> a) temp+=a; // extract all ss words; all spaces are stripped (>> operator)
```

```
while (getline(ss,a,'\n')) temp+=a; // read lines; spaces are kept; '\n' is consumed
ss >> hour >> sep >> minute; // If "12 : 27" is on ss, int hour becomes 12 and int minute 27
// sep must be a char or rest of the string would be consumed
```

Reaching the end of ss extraction (») causes eof. To reuse:

```
ss.str(""); // Different from ss.str(); this clears current contents
ss.clear(); // Clear error flags
ss << "Now I say hi" // Reusable again
```

However for best practice use `istringstream` and `ostringstream` for your needs:

```
string temp, finalNoSpaces;
ostringstream oss;
oss << " hi " << " dude";
istringstream iss(oss.str());

while(ss >> temp) finalNoSpaces += temp;
cout << finalNoSpaces; // print "hidude"
```

`iostream.h`, `iostream` (replaces `stdio.h`; inherits from `ios`)

```
#include <iostream> // Include iostream (std namespace)

cin >> x >> y; // Read words x and y from stdin (set fail flags if types mismatch)
// With strings, extract operator stops at whitespaces (consuming them)
// final '\n' (enter) is not consumed, use cin.ignore() later

if (cin) // Good state (same as !cin.fail() && !cin.eof())
if (!cin) cin.clear(); // Set error flags to 0

while(cin>>var) {a;} // store input in var (until whitespace) and do a; in loop
// if input and var types mismatch, fail flag is set and loop breaks
// eof flag (ctrl-z on windows and ctrl-d on linux) will also break

cin.ignore(nChars,Delim); // Ignore nChars characters or until delimiter found
// If a fail occurred because of type mismatch, there are chars in the buffer
// In that case you must ignore after clearing to allow new input

cout << "x=" << 3 << endl; // Write line to stdout (endl is same as cout << '\n' << flush)
cerr << x << y << flush; // Write to stderr and flush
c = cin.get(); // c = getchar();
cin.get(c); // Read char, store in c, consume it
cin.peek(c); // Read char, store in c, do not consume it (still asks if buffer is empty)
cin.getline(s, n, '\n'); // Read line into char s[n] to '\n' (default)
```

Any function that works on streams must use references (so that chains like `stream « var1 « var2` work). To overload operators for streams:

```
istream& operator>>(istream& i, T& x) {i >> ...; x=...; return i;}
ostream& operator<<(ostream& o, const T& x) {return o << ...;} // << operator should not modify variable
```

`iomanip` - output manipulation

Suppose you have an int hour between 0 and 24. To always output in the format HH you can do:

```
cout << setfill('0') << setw(2) << right << hour << endl;
// or left (center does not exist)
// instead of cout, you may also use stringstream
```

You may also manipulate number output:

```
float pi = 3.1415;
cout << setprecision(2) << pi;    // print 3.1 (two digits in total)
cout << setprecision(2) << fixed << pi;    // print 3.14 (two digits after floating point)
```

fstream.h, fstream - file input/output (works mostly like cin and cout)

If you pass filename to the ifstream and ofstream constructors, the opening and closing are taken care for you:

```
#include <fstream>    // Include filestream (std namespace)

ifstream f1("filename");    // Open text file for reading
if (f1)                    // Test if open and input available
f1 >> x;                    // Read object from file
f1.get(c);                  // Read char or line into c

while (getline(inputStream, str)) outputStream << str;    // Read file line by line

ofstream f2("filename");    // Open file for writing
if (f2) f2 << x;            // Write to file
```

You may use fstream and open with flags (do not forget closing):

```
fstream file;
file.open(filename,flag1|flag2...);
    // Some flags:
    // ios::in - open for input operations.
    // ios::out - open for output operations.
    // ios::binary - open in binary mode.
    // ios::app - output operations append to the end of the file
    // ios::ate - same as ios::app but you can move the file cursor
    // ios::trunc - replace current file contents if file exists (used by default)

if (file.is_open()) std::cout << "open file"; // check manually if file exists

if (file.eof()) file.clear(); // if you read until the end, clear the eof flags to reuse

file.close() // mandatory
```

You can write or read from the current cursor position a bunch of data:

```
// To correctly use these operations, file must be open in binary mode.
// You won't be able to open them with a text editor.

file.write((char *) data, nBytes) << flush; // write first nBytes of data variable
    // eg. data is an array or a struct
    // pass sizeof(dataType) to write all data

f.read((char *) data, nBytes); // read nBytes and assign them to data
    // file contents must be ordered correctly
```

You can have some fun with the cursor position and random access the files:

```
file.seekg(offset,flag);    // Put cursor on flag and move offset (reading purposes)
    // Some flags:
    // ios::beg - beginning of the file
    // ios::cur - current cursor position
    // ios::end - end of the file (use non-positive offsets)

// If you were reading struct Person instances you would do
```

```
// file.seekg(recordNumber * sizeof(Person), ios::beg);
// before reading (check above)
```

```
file.seekp(offset,flag); // Same but for writing purposes
```

```
file.tellg(); // Return reading cursor position
file.tellp(); // Return writing cursor position
```

vector - dynamic array (rapid insertions/deletions on back; random iteration)

```
#include <vector> // Include vector (std namespace)

vector<vector<T>> nested; // Nested vector (2D in this case)
vector<int> a(10); // a[0]..a[9] are int (default size is 0)
vector<int> b{1,2,3}; // Create vector with values 1,2,3
a.size(); // Number of elements (10)
a.push_back(3); // Increase size to 11, a[10]=3
e.emplace_back(3) // Push back an object of type T constructed with parameter 3
a.back()=4; // a[10]=4;
a.pop_back(); // Decrease size by 1
a.front(); // a[0];
a[20]=1; // Segmentation fault
a.at(20)=1; // Like a[20] but throws out_of_range()
a.resize(15); // Make vector size 15
// If new size is less than current, diff elements are demolished
// If new size is larger, memory is reserved, but nothing's on contents yet
// eg. do a.at(14) = value; before trying to access that index contents

a.erase(a.begin()+3); // Remove a[3], shifts elements towards back

a.erase(remove_if(a.begin(), a.end(), isOdd), a.end());
// Erase-remove idiom (faster than erasing one-by-one in a for loop)
// Remove_if points to the element after all non-removed elements
// isOdd is the comp function, should return bool and receive two objects

a.insert(a.begin()+2,12) // Make a[2] 12; shifts remaining to the right (linear complexity)

for (int& p : a) p=0; // In C++11 you do not need to use iterators for a quick iteration
for (vector<int>::iterator p=a.begin(); p!=a.end(); ++p) *p=0; // C++03 had no range-based for loop

vector<int> b(a.begin(), a.end()); // same as b = a;

vector<T> c(n, x); // c[0]..c[n-1] init to x
// you may use this syntax to initialize nested vectors
// eg. vector<vector<T>> c(nLines,vector<T>(nCols,valueToRepeat))
```

deque - stack queue (rapid insertions/deletions on front and back; random iteration)

deque<T> is like vector<T>, but also supports:

```
#include <deque> // Include deque (std namespace)

deque a<int>;
a.push_front(x); // Puts x at a[0], shifts elements toward back
a.pop_front(); // Removes a[0], shifts toward front
```

list - doubly linked list (rapid insertion/deletion everywhere, bidirectional iteration)

You cannot access specified index without accessing all on the left/right. Therefore you can't do `l.at(3)` and neither `l.begin()+3`; only `it++` and `it--`. The forward iteration version of this container is `forward_list`.

```
#include <list>    // Include list (std namespace)

list<int> l = {1,2,8,9,12,2};
auto it = find(l.begin(),l.end(),9);
l.insert(it,23); //insert 23 at position where 9 is; shift towards right

l.remove(8);      // remove all elements == 8; reduce container size
l.remove_if(f);   // same as above but use f as comp

l.sort();         // only for lists, use std::sort for random iteration containers
                  // use l.sort(comp) for your own condition

l.unique();       // removes all but the first from every consecutive group of equal elements
                  // l must be sorted for best results
                  // do l.unique(comp) to test for your own condition instead of equality
```

array - statically sized array (lightweight wrapper around C array; random iteration)

```
#include <array>    // Include array (std namespace)

array<int,3> houses = {1,2,4};
houses.at(2)        // Return 4
for (const auto& s: houses) {} // Range-based for loop is supported
houses.size()       // Return 3
```

utility (to use pair)

```
#include <utility>    // Include utility (std namespace)

pair<string, int> a("hello", 3); // A 2-element struct
a.first;                        // "hello"
a.second;                       // 3
```

tuple - fixed-size collection of heterogeneous values (generalization of pair)

```
#include <tuple>    // Include tuple (std namespace)

tuple<string,int,int> student;
student = {"Ana",12,13}; // in earlier C++ versions, student = make_tuple("Ana",12,13)
string studentName = get<0>(student); // cannot do student.at(0)
std::cout << tuple_size<decltype(student)>::value; // Print 3

string name; int grade1,grade2;
tie(name,grade1,grade2) = student; // unpack tuple into variables
```

map - ordered associative container (bidirectional iteration)

The operator `<` must be defined between two key objects. If order is not important, use `unordered_map` instead.

```

#include <map>           // Include map (std namespace)

map<string, int> a;      // Map from string to int

a["hello"] = 3;         // Add or replace element a["hello"]
                        // Same as: a.insert(make_pair("hello",3))

a.erase("hello");       // Erase by key
a.clear();              // Erase all map elements, leaving size at 0

for (const auto& p:a) cout << p.first << ": " << p.second; // Prints "hello: 3"

a.size();               // 1
a.empty();              // Same as !a.size()

```

multimap - store non-unique key-value pairs (bidirectional iteration)

While map can only store unique key-value pairs, multimap stores all pairs, including repeated ones. If you want non-repeated keys with multiple values, consider map<keyType, set<valueType>> instead.

```

#include <map>           // Include map (std namespace); multimap does not exist

multimap<string,int> retiros;
retiros.insert(make_pair("Osvaldo",0));
retiros.insert(make_pair("Osvaldo",0));
retiros.insert(make_pair("Osvaldo",2));

// Print all pairs with "Osvaldo" key
auto range = retiros.equal_range("Osvaldo");
for (auto it = range.first; it != range.second; ++it)
    std::cout << it->first << ": " << it->second << endl;

// Iterate all pairs for more flexibility
for (const auto& it: retiros)
    // compare it.first and it.second with desired conditions

```

set - store unique elements ordered (bidirectional iteration)

For insertion to work, the operator < must be defined between two objects of used type. Elements are considered duplicates (therefore not added) when !(a < b) && !(b < a). If order is not important, use unordered_set instead.

```

#include <set>           // Include set (std namespace)

set<Player> s;          // Empty set of Player's
                        // For later insertion, you must define the operator < for Player

set<int> s(v.begin(), v.end()); // Construct with iterators

set<Player*, decltype(comp)*> players(comp); // You cannot define operator < (neither ==) for two pointers
                                              // To have your own implementation, pass comp as help function

s.insert(123);          // Add element to set

if (s.find(123) != s.end()) // find is set specific (use find for other containers)

s.erase(123);          // no need to use iterators here (for vectors you did)

```

```
cout << s.size();           // Number of elements in set
```

algorithm - collection of algorithms on sequences with iterators

```
#include <algorithm>           // Include algorithm (std namespace)

min(x, y); max(x, y);          // Smaller/larger of x, y (any type defining <)
swap(x, y);                    // Exchange values of variables x and y
sort(a, a+n);                  // Sort array a[0]..a[n-1] by <
sort(a.begin(), a.end());      // Sort containers that support random iteration
sort(a.begin(), a.end(), f);    // Sort array or deque using f as comp (change order if f)
                                // f should be like bool f(T a, T b){return a<b;}

reverse(a.begin(), a.end());    // Reverse vector or deque

find(a.begin(), a.end(), value); // Return pointer to first value if found, else a.end()
binary_search(a.begin(), a.end(), value); // Same as above but container must be sorted

count(a.begin(), a.end(), value); // Return number of occurrences of value in container a
search(a.begin(), a.end(), sequence.begin(), sequence.end()); // Iterator to first occurrence of sequence

remove(a.begin(), a.end(), value); // Place non-removed elements at the beginning
                                    // Capacity isn't changed
                                    // Returns pointer to after last non-removed element

set_intersection(v1.begin(), v1.end(), v2.begin(), v2.end(),
                 inserter(intersectionVector, intersectionVector.begin()));
                                    // insert into intersectionVector the v1 and v2 common values
```

chrono - time related library

```
#include <chrono>
using namespace chrono;

auto from = high_resolution_clock::now();

// ... do some work

auto to = high_resolution_clock::now();

using ms = duration<float, milliseconds::period>; // typedef duration<float, milliseconds::period> ms;

cout << duration_cast<ms>(to - from).count() << "ms";
```

C style random integers

```
#include <chrono>
using namespace chrono;

auto seed = // Get time since 1 Jan 1970
            system_clock::now().time_since_epoch().count();

srand(seed); // Initialize random generator (only once in entire program)
rand() % b + a; // Return integer in range [a, b+a[
```

Dynamic memory allocation (manual allocations on the heap)

C Style:

```
// allocate 1D array
int* intArray = (int*) malloc(nElems * sizeof(int));

// reallocate more memory to intArray if needed
intArray = (int*) realloc(intArray, newNElems * sizeof(int));

// deallocate 1D array
free(intArray); //free takes a void*, but implicit conversion is made
```

C++ Style:

```
// allocate 2D array
int** intMatrix = new int*[nLines];
for (int i=0; i < nLines;++i) intMatrix[i] = new int[nCols];

// deallocate 2D array
for (int i=0; i < nLines;++i) if (intMatrix[i] != nullptr) delete[] intMatrix[i];
if (intMatrix != nullptr) delete[] intMatrix;
```

ctype.h - some C Standard Library predicates (included by default in C++)

Some predicates:

```
isalpha(c); // Used to check if the character is an alphabet or not.
isdigit(c); // Used to check if the character is a digit or not.
isalnum(c); // Used to check if the character is alphanumeric or not.
isupper(c); // Used to check if the character is in uppercase or not
islower(c); // Used to check if the character is in lowercase or not.
iscntrl(c); // Used to check if the character is a control character or not.
isgraph(c); // Used to check if the character is a graphic character or not.
isprint(c); // Used to check if the character is a printable character or not.
ispunct(c); // Used to check if the character is a punctuation mark or not.
isspace(c); // Used to check if the character is a white-space character or not.
isxdigit(c); // Used to check if the character is hexadecimal or not.
```

And to manipulate characters:

```
toupper(c); // Used to convert the character into uppercase.
tolower(c); // Used to convert the character into lowercase.
```

math.h, cmath - floating point math

```
#include <cmath> // Include cmath (std namespace)

sin(x); cos(x); tan(x); // Trig functions, x (double) is in radians
asin(x); acos(x); atan(x); // Inverses
atan2(y, x); // atan(y/x)
sinh(x); cosh(x); tanh(x); // Hyperbolic sin, cos, tan functions
exp(x); log(x); log10(x); // e to the x, log base e, log base 10
pow(x, y); sqrt(x); // x to the y, square root
ceil(x); floor(x); // Round up or down (as a double)
fabs(x); fmod(x, y); // Absolute value, x mod y
```

assert.h, cassert - debugging aid

The definition of the macro `assert` depends on another macro, `NDEBUG`, which is not defined by the standard library.

```
#include <cassert>
#define NDEBUG           // quickly disable all assertions by commenting this line

assert(e);              // if e is false, print message and abort
```

Special Keywords

Reserved keywords (may not be used in other contexts):

```
alignas
alignof
and
and_eq
asm
atomic_cancel
atomic_commit
atomic_noexcept
auto
bitand
bitor
bool
break
case
catch
char
char8_t
char16_t
char32_t
class
compl
concept
const
constexpr
constinit
const_cast
continue
co_await
co_return
co_yield
decltype
default
delete
do
double
dynamic_cast
else
enum
explicit
export
extern
false
float
for
friend
```

goto
if
inline
int
long
mutable
namespace
new
noexcept
not
not_eq
nullptr
operator
or
or_eq
private
protected
public
constexpr
register
reinterpret_cast
requires
return
short
signed
sizeof
static
static_assert
static_cast
struct
switch
synchronized
template
this
thread_local
throw
true
try
typedef
typeid
typename
union
unsigned
using
virtual
void
volatile
wchar_t
while
xor
xor_eq

May be used in function names or objects when not in their special context:

override
final
import
module
transaction_safe
transaction_safe_dynamic