C++ QUICK REFERENCE

C++98

Preprocessor

```
// Comment to end of line
/* Multi-line comment */
#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
#define X \
 some text // Multiline definition
#undef X // Remove definition
#if defined(X) // Conditional compilation
  (#ifdef X)
#else // Optional (#ifndef X or #if !
  defined(X))
#endif // Required after #if, #ifdef
```

Data Types

Data Type	Description
bool	boolean (true or false)
char	character ('a', 'b', etc.)
char[]	character array (C-style string if null terminated)
string	C++ string (from the STL)
int	Integer (1, 2, -1, 1000, etc.)
long int	long integer
float	single precision floating point
double	double precision floating point

These are the most commonly used types: this is not a complete list.

Operators

The most commonly used operators in order of precedence:

1. ++ (post-increment), -- (post-decrement)
2. !(not), ++ (pre-increment), -- (pre-decrement)
3. */, %(modulus)
4. +-
5. <, <=, >, >=
6. == (equal-to), != (not-equal-to)
7. && (and)
8. || (or)
9. = (assignment), *, /, %=, +=, -=

Looping

while Loop

```
while (expression)
  statement;
```

Example:

```
while (x < 100)
  cout << x++ << endl;
```

```
while (expression){
  statement;
  statement;
}
```

Example:

```
while (x < 100){
  cout << x << endl;
  x++;
}
```

do-while Loop

```
do
  statement;
while (expression);
```

Example:

```
do
  cout << x++ << endl;
while (x < 100);
```

```
do{
  statement;
  statement;
} while (expression);
```

Example:

```
do {
  cout << x << endl;
  x++;
} while (x < 100);
```

For Loop

```
for (initialization; test; update)
  statement;
```

```
for (initialization; test; update){
  statement;
  statement;
}
```

Example:

```
for (count = 0; count < 10; count++) {
  cout << "count equals: " << count << endl
    ;

}
```

Decision Statements

if

```
if (expression)
  statement;
```

Example:

```
if (x < y)
  cout << x;
```

if/else

```
if (expression)
  statement;
else
  statement;
```

Example:

```
if (x < y)
  cout << x;
else
  cout << y;
```

switch/case

```
switch(int expression)
{
  case int-constant:
    statement(s);
    break;
  case int-constant:
    statement(s);
    break;
  default:
    statement;
}
```

Example:

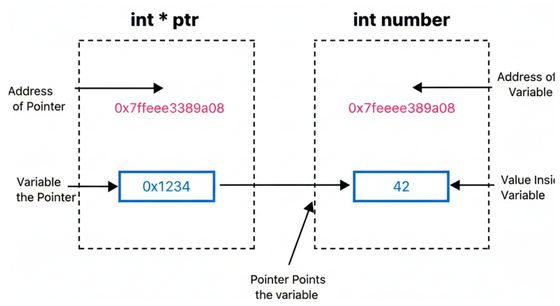
```
switch(choice)
{
  case 0:
    cout << "Zero";
    break;
  case 1:
    cout << "One";
    break;
  default:
    cout << "What?";
}
```

Pointers

A pointer variable (or just pointer) is a variable that stores a memory address. Pointers allow the indirect manipulation of data stored in memory. Pointers are declared using *. To set a pointer's value to the address of another variable, use the & operator. Example:

```
char c = 'a';
char* cPtr;
cPtr = &c;
```

Use the indirection operator (*) to access or change the value that the pointer references.



Example:

```
// continued from example above
*cPtr = 'b';
cout << *cPtr << endl; // prints the char
  b
// prints the char b
```

Array names can be used as constant pointers, and pointers can be used as array names. Example:

```
int numbers[] = {10, 20, 30, 40, 50};
int* numPtr = numbers;
cout << numbers[0] << endl; // prints 10
cout << *numPtr << endl; // prints 10
cout << numbers[1] << endl; // prints 20
cout << *(numPtr + 1) << endl; // prints
  20
cout << numPtr[2] << endl; // prints 30
```

Dynamic Memory

Allocate Memory

Syntax	Examples
ptr = new type;	int* iPtr; iPtr = new int;
ptr = new type [size];	int* intArray; intArray = new int[5];

Deallocate Memory

Syntax	Examples
delete ptr;	delete iPtr;
delete [] ptr;	delete [] intArray;

Once a pointer is used to allocate the memory for an array, array notation can be used to access the array locations. Example:

```
int* intArray;
intArray = new int[5];
intArray[0] = 23;
intArray[1] = 32;
```

I/O

Console Input/Output

- cout « console out, printing to screen
- cin » console in, reading from keyboard
- cerr « console error

Example:

```
cout << "Enter an integer: ";
cin >> i;
cout << "Input: " << i << endl;
```

File Input/Output

Example (input):

```
ifstream inputFile;
inputFile.open("data.txt");
inputFile >> inputVariable;
// you can also use get (char) or
// getline (entire line) in addition to >>
inputFile.close();
```

Example (output):

```
ofstream outFile;
outFile.open("output.txt");
outFile << outputVariable;
outFile.close();
```

Function

Functions

Functions return at most one value. A function that does not return a value has a return type of void. Values needed by a function are called parameters.

```
return_type function(type p1, type p2,...)
{
  statement;
  statement;
}
```

Examples:

```
int timesTwo(int v) {
  int d;
  d = v * 2;
  return d;
}
void printCourseNumber() {
  cout << "CS 3911" << endl;
  return;
}
```

Passing Parameters by Value

```
return_type function(type p1)
```

Variable is passed into the function but changes to p1 are not passed back.

Passing Parameters by Reference

```
return_type function(type &p1)
```

Variable is passed into the function and changes to p1 are passed back.

Default Parameter Values

```
return_type function(type p1=val)
```

val is used as the value of p1 if the function is called without a parameter.

Struct/Classes

```
class classname {
public:
  classname (params); // constructor
  ~classname(); // destructor
  type member1;
  type member2;
protected:
  type member3;
private:
  type member4;
};
```

Example:

```
class Square {
public:
  Square();
  Square (float w);
  void setWidth(float w);
  float getArea();
private:
  float width;
};
```

- public members are accessible from anywhere the class is visible.
- private members are only accessible from the same class or a friend (function or class).
- protected members are accessible from the same class, derived classes, or a friend (function or class).
- constructors may be overloaded just like any other function. You can define two or more constructors as long as each constructor has a different parameter list.

Definition of Member Functions

```
return_type classname::functionName (params
  ) {
  statements;
}
```

Examples:

```
Square::Square() {
  width = 0;
}
void Square::setWidth(float w) {
  if (w >= 0)
    width = w;
  else
    exit(-1);
}
float Square::getArea() {
  return width * width;
}
```

Definition of Instances

```
classname varName;
classname* ptrName;
```

Example:

```
Square s1();
Square s2(3.5);
Square* sPtr;
sPtr = new Square (1.8);
```

Accessing Members

```
varName.member = val;
varName.member();
ptrName->member = val;
ptrName->member();
```

Example:

```
s1.setWidth(1.5);
cout << s1.getArea();
cout << sPtr->getArea();
```

C++ QUICK REFERENCE

OOP, Templates, Exceptions

OOP Principles

Encapsulation: Hide internal implementation details by using access specifiers (`public`, `private`, `protected`) to control access to member variables and functions.
Abstraction: Define interfaces that represent only essential features while hiding complexity. Use abstract classes with pure virtual functions.

Inheritance

Inheritance allows a new class to be based on an existing class. The new class inherits all the member variables and functions (except the constructors and destructor) of the class it is based on.
Example:

```
class Student
{
public:
  Student (string n, string id);
  void print();
protected:
  string name;
  string netID;
};
class GradStudent : public Student
{
public:
  GradStudent(string n, string id, string
    prev);
  void print();
protected:
  string prevDegree;
};
```

Visibility of Members after Inheritance

Inheritance Specification	Access Specifier in Base		
	private	protected	pub
private	private	private	priv
protected	protected	protected	prot
public	protected	protected	pub

Multiple Inheritance

Multiple inheritance allows a class to inherit from more than one base class. Useful for mixing interfaces/behaviors, but watch for ambiguity and the diamond problem.

```
class A { public: void f() {} };
class B { public: void g() {} };
class C : public A, public B { };
// C has both A::f and B::g
```

Ambiguities arise when multiple bases provide the same member name:

```
class A { public: void x(); };
class B { public: void x(); };
class C : public A, public B { };
// call must be qualified: c.A::x(); or c.
  B::x();
```

Diamond problem: When two base classes share a common base, the derived may contain multiple subobjects of the common base. Use virtual inheritance to ensure only one shared base subobject:

```
class Top { public: int v; };
class Left : virtual public Top { };
class Right: virtual public Top { };
class Bottom: public Left, public Right {
  };
// Bottom has one Top subobject when Left/
  Right inherit virtually
```

Notes: - Virtual inheritance adds complexity to constructors: explicitly initialize the virtual base in the most-derived class. - Ambiguous member names must be qualified with the base class name. - Prefer composition or interfaces (pure virtual classes) when multiple inheritance is only used for code reuse.

Polymorphism

Compile-time (Static) Polymorphism: Function overloading and templates.
Runtime (Dynamic) Polymorphism: Use virtual functions to allow derived classes to override base class methods. Write code that works with base class pointers/references to handle derived classes dynamically.
Example:

```
class Shape // Abstract class
{
public:
  virtual void draw() = 0; // pure virtual
  virtual double area() = 0;
  virtual ~Shape() {}
};

class Circle : public Shape
{
public:
  void draw() { /* implementation */ }
  double area() { return 3.14 * r * r; }
private:
  double r;
};

Shape* shapes[10];
shapes[0] = new Circle();
shapes[0]->draw(); // calls Circle::draw()
```

Rule of Three:

If a class defines a destructor, copy constructor, or copy-assignment, it probably needs all three.
Example

```
class MyVector {
public:
    MyVector(int n) : size(n), data(new int
      [n]) {}
    ~MyVector() { delete[] data; }

    // Copy constructor
    MyVector(const MyVector& other) : size(
      other.size), data(new int[other.size
      ]) {
        for (int i = 0; i < size; ++i)
            data[i] = other.data[i];
    }

    // Copy assignment operator
    MyVector& operator=(const MyVector&
      other) {
        if (this != &other) {
            delete[] data;
            size = other.size;
            data = new int[size];
            for (int i = 0; i < size; ++i)
                data[i] = other.data[i];
        }
        return *this;
    }

    int& operator[](int i) { return data[i
      ]; }
    int sizeOf() const { return size; }

private:
    int size;
    int* data;
};
```

Operator Overloading

C++ allows you to define how standard operators (+, -, *, etc.) work with classes that you write. For example, to use the operator + with your class, you would write a function named `operator+` for your class. Example: Prototype for a function that overloads + for the `Square` class:

```
Square operator+ (const Square &);
```

If the object that receives the function call is not an instance of a class that you wrote, write the function as a `friend` of your class. This is standard practice for overloading « and ».

Example: Prototype for a function that overloads « for the `Square` class:

```
friend ostream & operator<<
  (ostream &, const Square &);
```

Make sure the return type of the overloaded function matches what C++ programmers expect. The return type of relational operators ($<, >, ==$, etc.) should be `bool`, the return type of « should be `ostream &`, etc.
Complete Example:

```
  #include <iostream>

class Complex {
public:
    Complex(double r = 0, double i = 0) :
      real(r), imag(i) {}
    ~Complex() {}

    // Copy constructor
    Complex(const Complex& other) : real(
      other.real), imag(other.imag) {}

    // Copy assignment operator
    Complex& operator=(const Complex& other
      ) {
        if (this != &other) {
            real = other.real;
            imag = other.imag;
        }
        return *this;
    }

    // Addition operator
    Complex operator+(const Complex& other)
      const {
        return Complex(real + other.real,
          imag + other.imag);
    }

    // Print function
    void print() const {
        std::cout << real << " + " << imag
          << "i" << std::endl;
    }

private:
    double real, imag;
};

// Usage example:
// Complex a(1.0, 2.0), b(3.0, 4.0);
// Complex c = a + b;
// c.print(); // Output: 4 + 6i
```

Namespaces

Namespace usage prevents name collisions. Avoid 'using namespace std;' in headers.

```
namespace N { class T {}; } //create a
  namespace
N::T t;
using namespace N; // ok in implementation
   units, avoid in headers
```

Exceptions

Example:

```
try
{
  // code here calls functions that might
  // throw exceptions
  quotient = divide (num1, num2);

  // or this code might test and throw
  // exceptions directly
  if (num3 < 0)
    throw -1; // exception to be thrown
      can
            // be a value or an object
}
catch (int)
{
  cout << "num3 can not be negative!";
  exit(-1);
}
catch (char* exceptionString)
{
  cout << exceptionString;
  exit(-2);
}
// add more catch blocks as needed
```

Template

Templates in C++ allow you to write generic and reusable code for functions and classes. They enable the creation of code that works with any data type, reducing duplication and increasing flexibility.

Function Templates

Example:

```
template <class T>
T getMax(T a, T b)
{
  if (a > b)
    return a;
  else
    return b;
}
```

Example calls:

```
int a = 9, b = 2, c;
c = getMax(a, b);
float f = 5.3, g = 9.7, h;
h = getMax(f, g);
```

Class Templates

Class templates define parameterized types. Define member functions inside the class or provide out-of-class definitions with the 'template<>' prefix.

```
template <class T>
class Point {
public:
  Point(T x, T y) : x(x), y(y) {}
  T getX() const { return x; }
  T getY() const { return y; }
private:
  T x, y;
};

Point<int> p(3,4);
std::cout << p.getX() << "," << p.getY() <<
   std::endl;
```

Specialization example (full specialization):

```
template <>
class Point<char> {
  // specialized implementation for char
};
```

Notes:

- When defining member functions outside the class body, repeat the 'template <...>' header.
- Class templates can be partially specialized (for classes), but not for functions in C++98.
- Prefer simple interfaces; templates increase compile-time and complexity.

Another Example

```
template <class T>
class Point
{
public:
  Point(T x, T y);
  void print();
  double distance (Point<T> p);
private:
  T x;
  T y;
};
//usage
Point<int> p1(3, 2);
Point<float> p2(3.5, 2.5);
p1.print();
p2.print();
```

Suggested Websites

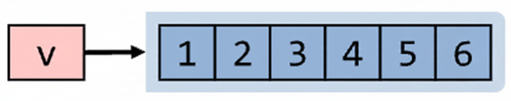
- C++ Reference: `http://www.cppreference.com/`
- C++ Tutorial: `http://www.cplusplus.com/doc/tutorial/`

C++ QUICK REFERENCE

- Containers: store collections (`vector`, `list`, `map`, `set`, `deque`, ...)
- Iterators: traverse containers (input, forward, bidirectional, random-access)
- Algorithms: operate on iterator ranges (`sort`, `find`, `for_each`, ...)
- Functors: objects like functions, adapters (`stack`, `queue`)

std::vector

Dynamic contiguous array.




```
#include <vector>
#include <iostream>

int main() {
    std::vector<int> v;
    for (int i = 0; i < 5; ++i) v.push_back(i
      *10);

    for (std::vector<int>::size_type i = 0; i
       < v.size(); ++i)
        std::cout << v[i] << " ";
    std::cout << '\n';

    for (std::vector<int>::iterator it = v.
      begin(); it != v.end(); ++it)
        std::cout << *it << " ";
    std::cout << '\n';
}
```

std::list

Doubly-linked list.

```
#include <list>
#include <iostream>

int main() {
    std::list<int> L;
    L.push_back(1); L.push_back(2); L.
      push_back(3);
    for (std::list<int>::const_iterator it =
      L.begin(); it != L.end(); ++it)
        std::cout << *it << " ";
}
```

Associative containers

```
#include <map>
#include <string>
#include <iostream>

int main() {
    std::map<std::string, int> ages;
    ages["Alice"] = 30;
    ages["Bob"] = 25;

    for (std::map<std::string,int>::
      const_iterator it = ages.begin(); it !=
       ages.end(); ++it)
        std::cout << it->first << ": " << it
          ->second << '\n';
}
```

Container adapters

```
#include <queue>
#include <iostream>

int main() {
  std::queue<int> q;
  q.push(1);
  q.push(2);
  q.push(3);

  while (!q.empty()) {
    std::cout << q.front() << " ";
    q.pop();
  }
  std::cout << '\n';
}
```

Unordered maps and pairs

```
#include <map>
#include <string>
#include <utility>
#include <iostream>

int main() {
  std::map<std::string,int> ages;
  ages.insert(std::make_pair(std::string("
    Alice"), 30));
  ages.insert(std::make_pair(std::string("Bob
    "), 25));

  for (std::map<std::string,int>::
    const_iterator it = ages.begin(); it !=
    ages.end(); ++it)
    std::cout << it->first << ": " << it->
      second << '\n';

  // If your implementation provides
    unordered_map:
  /* #include <unordered_map>
     std::unordered_map<std::string,int> um;
     um["Alice"] = 30;
  */
}
```

Iterators

```
#include <vector>
#include <algorithm>
#include <iostream>

int main() {
    std::vector<int> v;
    v.push_back(3); v.push_back(1); v.
      push_back(2);
    std::sort(v.begin(), v.end());
    for (std::vector<int>::const_iterator it
      = v.begin(); it != v.end(); ++it)
        std::cout << *it << ' ';
}
```

Algorithms

Remove-Erase Idiom

```
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> v;
    v.push_back(1); v.push_back(2); v.
      push_back(3); v.push_back(2);
    v.erase(std::remove(v.begin(), v.end(),
      2), v.end());
}
```

Transform and Accumulate

```
#include <vector>
#include <algorithm>
#include <numeric>
#include <iostream>

struct Square { int operator()(int x) const
  { return x*x; } };

int main() {
    std::vector<int> v;
    v.push_back(1); v.push_back(2); v.
      push_back(3);
    std::vector<int> squares(v.size());
    std::transform(v.begin(), v.end(),
      squares.begin(), Square());
    int sum = std::accumulate(squares.begin()
      , squares.end(), 0);
    std::cout << "sum = " << sum << '\n';
}
```

Functors and Comparators

```
#include <string>
#include <vector>
#include <algorithm>

struct ByLength {
    bool operator()(const std::string& a,
      const std::string& b) const {
        return a.size() < b.size();
    }
};

int main() {
    std::vector<std::string> v;
    v.push_back("apple"); v.push_back("kiwi")
      ; v.push_back("banana");
    std::sort(v.begin(), v.end(), ByLength())
      ;
}
```

Useful Idioms

- Use `reserve()` on `vector` when size is known.
- Pass large containers by `const&`.
- Use `lower_bound` / `upper_bound`.

```
#include <vector>
#include <algorithm>
#include <iostream>
#include <numeric>
#include <map>
#include <string>

struct SquareFunctor { int operator()(int x)
   const { return x * x; } };

int main() {
    std::vector<int> v;
    for (int i = 1; i <= 5; ++i)
          v.push_back(i);

    std::vector<int> sq(v.size());
    std::transform(v.begin(), v.end(),
                   sq.begin(),
                     SquareFunctor());

    int sum = std::accumulate(sq.begin(), sq.
      end(), 0);
    std::cout << "sum of squares = " << sum
      << '\n';

    std::vector<std::string> words;
      words.push_back("apple");
      words.push_back("kiwi");
      words.push_back("banana");
    std::map<int,int> freq;
    for (std::vector<std::string>::
      const_iterator it =
                       words.begin(); it !=
                         words.end(); ++it)
        ++freq[it->size()];

    for (std::map<int,int>::const_iterator it
       = freq.begin();
                      it != freq.end(); ++it
                        )
        std::cout << "length " << it->first
               << ": " << it->second << "
                 word(s)\n";
}
```

C++11

Type inference: 'auto', 'decltype'

```
#include <vector>
#include <iostream>
int main() {
  std::vector<int> v = {1,2,3,4};
  for (auto it = v.begin(); it != v.end();
    ++it)
    std::cout << *it << ' ';
  std::cout << "\n";

  auto sum = 0; // int
  decltype(sum) total = 0; // same type as
    sum
}
```

Range-based for

```
#include <vector>
#include <iostream>

int main() {
  std::vector<int> v = {10,20,30};
  for (auto x : v) std::cout << x << ' ';
  std::cout << '\n';
  for (auto &x : v) x += 1; // modify in-
    place
}
```

Lambda expressions

```
#include <algorithm>
#include <vector>
#include <iostream>

int main() {
  std::vector<int> v = {3,1,2};
  std::sort(v.begin(), v.end(), [](int a,
    int b){ return a<b; });
  std::for_each(v.begin(), v.end(), [](int
    x){ std::cout << x << ' '; });
  std::cout << '\n';
}
```

Move Semantics in C++

Beginning with C++11, the C++ language introduced *move semantics*, a mechanism enabling efficient transfer of resources from one object to another. Move semantics reduce unnecessary copying of objects that manage heavy resources such as memory buffers, strings, or file handles.
We provide a detailed and rigorous explanation of move semantics, beginning with value categories and progressing to move constructors, move assignment, and usage recommendations.

Value Categories and Reference Types

Understanding move semantics requires understanding how C++ distinguishes persistent objects from temporary ones.

Lvalues

An **lvalue** refers to an object with a persistent memory location.

```
int x = 10; // x is an lvalue
```

Lvalues bind to lvalue references:

```
T&
```

Rvalues

An **rvalue** is a temporary expression without a stable address.

```
10 // rvalue literal
x + 5 // rvalue
std::string("hi") // temporary
```

Rvalues bind to rvalue references:

```
T&&
```

Temporaries may be safely moved from, enabling efficient resource transfers.

Move Constructors and Move Assignment

Classes that manage resources must implement:

- **Move constructor**
- **Move assignment operator**

Example Class

```
class MyString {
private:
  char* data;

public:
  // Constructor
  MyString(const char* s) {
    size_t n = std::strlen(s) + 1;
    data = new char[n];
    std::memcpy(data, s, n);
  }

  // Destructor
  ~MyString() {
    delete[] data;
  }

  // Copy constructor
  MyString(const MyString& other) {
    size_t n = std::strlen(other.data) + 1;
    data = new char[n];
    std::memcpy(data, other.data, n);
  }
```

Move Constructor

```
  MyString(MyString&& other) noexcept
    : data(other.data)
  {
    other.data = nullptr;
  }
```

Move Assignment Operator

```
  MyString& operator=(MyString&& other)
    noexcept {
    if (this != &other) {
      delete[] data;
      data = other.data;
      other.data = nullptr;
    }
    return *this;
  }
};
```