Based on Matt Mahoney's "C++ Quick Reference" sheet.

C++ Quick Reference

PREPROCESSOR

```
#include <iostream>      // STL Input/Output streams
#include <#NAME>         // Include #NAME library from standard
#include "myfile.h"      // Include local defined header file.
#define X 500            // define the element X as 500 (no type)
#define F(a,b) a+b       // Replace F(1,2) with 1+2
#undef X                 // undefined X
#ifndef _NAME_H          // if not defined, define as _NAME_H
#else
#endif
__Example__:
#ifndef _NAME_H
#define _NAME_H
....
#endif
```

this is a guard, to make sure your header file is only included once.

LITERALS

```
255, 0388, 0xff          // Integers (decimal, octal, hex)
2147483647L, 0x7fffffffl // Long (32-bit) integers
123.0, -23.12, 3.14      // double (real) numbers
'a', '\n', 'A'           // Characters  (\ is a special character,
```

we call this an escape-character, \n == new line, \t == tab).

```
"hello world"            // C++ string literal
true, false              // bool constants of 1 and 0
```

DECLARATIONS

```
int x;                   // Declare x to be of type integer
int x = 255;             // Declare x to be of type integer and
```

assigned 255

```
char s = 'a';            // Declare a char to 'a'
unsigned char u = 255;   // Use unsigned notation, recall that
```

unsigned is only positive numbers or zero and signed can be both positive and negative.

```
bool b = true;           // Set bool true
int a,b,c;               // Declare more on one line. a, b and c is
```

of type int

```
int a[10];               // Declare an integer array of size 10.
int a[] ={0,1,2}         // Declare and initialize array of size 3.
int a[2][3] = {{1,2,3},{4,5,6}}; // 2-D array initialized.
char s[] = "Hello";      // C-like array structure initialized to
```

"Hello"

```
int* p;                  // p is a pointer (an address of an integer
```

p)

```
char* s = "hello"        // char-array, pointer to the first char.
void* p = NULL           // address of untyped memory (NULL is 0).
int& r = x;              // r is a reference to (alias of) int x
enum weekend {SAT,SUN}; //weekend is a type with values SAT and SUN
enum weekend day;        // day is a variable of type weekend
enum weekend {SAT=6,SUN}; // Explicit conversion to int. (SUN will
```

automatically be 7, when SAT (processor) is 6.

```
typedef String char*     // String s; means char* s
const int c = 3;         //Constant must be initialized, assigned 3.
const int* p = a;        // Contents of p (element of a) are
```

constant

```
int* const p = a;        // p (but not contents) are constant
const int* const p = a; // Both p and its contents are constant
cosnt int& cr = x;       // cr cannot be assigned to change x
```

STORAGE CLASSES

```
int x;                   // Only exits within scope
static int x;            // exits with global lifetime, remember
```

that a static defined declaration or function isn't shared by exits singular.

```
extern int x;            // Information only, is declared within
```

another file.

STATEMENTS

```
x = y;                   // statement of expression of assignment
int x;                   // declaration of integer
;                        // empty statement
__EXAMPLE__:
for(int i=0; i < 100; ++i) {;} //does nothing for 100 ticks.
```

```
{
....
}                       // a scope in C++. Any local data will be
de-allocated at the end of a scope.

if (x) a;               // if x is true do a.
if (x) { a; }           // same as before, but you can insert
multiple lines inside the if-expression.
else if (y) b;          // another choice in the expression.
else c;                 // if none of the inquiries fire, c fires
as default.

while (x) a;            // for as long as x holds, run a

for(x; y; z) a;         // Equivalent to: x; while(y) (a; z;)
__EXAMPLE__:
for(int i = 0; i < 200; ++i) {...} //here (int i = 0) is x and (i <
200) is y, while ++I is z. x tells us where we start, y is our
destination and z determines the step size of each iteration in our
loop.

do a; while (x)         // same as while, but guarantee x fires
once.
switch(x) {             // test on x
            case X1: a; // if x == X1, then a fires
            case X2: b;
            default: c; // if we can't match anything, c fires.
}
break;                  // break out of expression
continue;               // skip an iteration of the expression
try { a; }              // automatically fires a, if a throws an
exception. Go to catch
catch (T t) { b; }      // catch exception of type T and then fire
b
catch (...) { c; }      // catch all exception and fire c
```

FUNCTIONS

```
int f(int x, int);      // function f with signature int x and int
and returns int type.
void f();               // function f with no arguments and no
return.
void f(int a=0);        // f is default equivalent to f(0), if we
define f(x) a = x; (default parameters)
f();                    // default return is int.
inline f();             // developer optimizes instead of compiler.
f() { statement; }      // Function definition (must be global)
T operator+(T x, T y);  // a + b (if type T) calls operator+(a, b)
T operator-(T x);       // -a calls function operator-(a)
T operator++(int);      // postfix ++ or – (parameter ignored).
extern "C" {void f();}  // f() was compiled in C
```

Functions parameters and return types can be of any type. A function must be declared or defined before it can be used. So a function must be prototyped. A function must be declared globally, but can be defined (prototyped) in other files.

Main have a return type of int, if the program is successful it'll return 0, otherwise it'll return a failure code (e.g. 1).

```
int main() { statements...} or
int main(int argc, char* argv[]) { statements...}
```

argc is the count of arguments when calling the program and argv is the data of the arguments.

```
If we call a program demo –PARAM1 –CODE 3, then;
            - argc = 4
            - argv[] = {
                        argv[0] = "demo",
                        argv[1] = "-PARAM1",
                        argv[2] = "-CODE",
                        argv[3] = "3"}
```

If we have more function with the same name, like;

```
            void cat()
            void cat(string n)
            void cat(string n, int m)
```

then we say we overload our functions with different signatures. If we call cat(); cat("abc"); cat("abc", 32) the program will associate the call with the proper signature.

EXPRESSIONS

Operators are grouped by precedence, highest first. Unary (one) operators and assignment evaluate right to left (int k = f(sum(3 + 4))), here we'll evaluate 3 + 4 first, then provide sum with 7 and in the end provide f with the retuned type. All other operators act

```
from left to right. Precedence does not affect order of evaluation,
which is undefined.

T::X                    // Name X is defined in class T
N::X                    // Name X is defined in namespace N
::X                     // Global name X
t.x                     // Member x of struct or public field of t
p->x                    // Member x of pointer to struct or class
t.
a[i]                    // i'th element of array a
f(x,y)                  // call function with parameters x and y.
T(x,y)                  // construct class T with parameters x and
y.
x++                     // add 1 to x, evaluates after (postfix)
++x                     // add immediately 1 to x, evaluates before
(prefix).
~x                      // bitwise complement of x.
!x                      // true if x is 0, true if x is false.
-x                      // unary minus
+x                      // unary plus
&x                      // address of x
*p                      // contents of address p (*&x equals x)
new T                   // allocate object T on the heap - can go
out of scope.
new T(x,y)
new T[x]
delete p                // destroy an object from the heap and de-
allocate it.
delete[] p              // destroy and free array of objects at p
(T) x                   // convert x to T (use ...cast<T>(x) if
possible).
__casting in C++__
static_cast<type>(expr) // Convert expr to type using defined
conversions
dynamic_cast<type>(expr)// Convert base pointer or reference to
derived if possible
const_cast<type>(expr)  // Convert expr to non-const type
reinterpret_cast<type>(expr) // Pretend expr is of type

__EXAMPLES__
double d; d=static_cast<double>(3); // Explicit 3 to 3.0
d=3;                    // Implicit conversion
d=sqrt(3);              // Implicit sqrt(3.0)
vector<int> v(5)        // Constructors are explicit
v=5; v=static_cast<vector<int>>(5); // Works

int x=3;
const int& r=x; r=4     // Error, r is const
const_cast<int&>(r)=4;  // works, x=4
*const_cast<int*>(p)=5  // works, x=5

*reinterpret_cast<double*>(p)=5 // Crash, writing 8 bytes into 4

x * y                   // Multiply
x / y                   // divide (integer round toward 0)
x % y                   // Modulo (result has sign of x)

x + y                   // add, or &x[y]
x - y                   // subtract.

x << y                  // shift x bitwise to the left by y
x >> y                  // shift x bitwise to the right by 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                  // Equals
x != y                  // not equals

x & y                   // bitwise ( x AND y )
x ^ y                   // bitwise exclusive or (x XOR y)
x | y                   // bitwise OR
x && y                  // x and then y (evaluates y only if x)
x || y                  // x or else y (evaluates y only if is
false)

x = y                   // assign y to x
x += y                  // x = x + y

x ? y : z               // y if x is true, else z

throw x                 // throw exception x

x , y                   // evaluates x and y, returns y
```

CLASSES

```
class T {               // A new object type T
private:                // Section only accessible by T's member
functions.
protected:              // also accessible by classes derived from
T
public:                 // accessible to all
   int x;               // Member data
   void f();            // Member function
   void g() {return;}   // Inline member function
   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() : x(1) {}        // Constructor with initialization list
   T(const T& t): x(t.x) {} // Copy constructor
   T& operator=(const T& t) {x = t.x; return *this; }       //
assignment operator
   ~T()                 // Deconstructor
   explicit T(int a);   // Allow t=T(3) but not t=3 – explicit
define conversions.
   operator int() const {return x;} // Allow int(t)
   friend void i();     // Global function i() has private access
   friend class U;      // Member of class U has private access
   static int y;        // Data shared between all object of type T
   static void l();     // Shared code. May access y but not x
   class Z {};          // nested class
   typedef int V;       // T::V means int
};
void T::f() {           // Code for member function f of class T
            this->x = x;} // this is a pointer that points to the
current object
int T::y = 2            // initialize static member data
T::l();                 // call to static member

struct T {              // Equivalent to: class T { public: …
meaning a struct is default public, while a class is default
private.
virtual void f();       // may be overridden at run time by derived
class
virtual void g()=0; };  // Must be overridden (pure virtual)
```

A pure virtual MUST be overridden, while a virtual may be overridden.

```
class U : public T { }; // Derived class U inherits all members of
base T
class V : private T { };// Inherited members of T become private
class W : public T, public U { }; // Multiple inheritance
class X : public virtual T { }; // Classes derived from X have base
T directly.
```

All classes have a default copy constructor, assignment operator, and destructor, which performs the corresponding operations on each data member and each base as shown above. There is also a default no-argument constructor (required to create arrays) if the class has no constructors. Constructors, assignment, and destructors do not inherit.
Also you cannot overload destructors, but they can be called from other member functions.

TEMPLATES

```
template <class T> T f(T t); // Overload f for all types
template <class T> class X { // Class with type parameter T
   X(T t); };           // A constructor
template <class T> X<T>::X(T t) {} // Definition of constructor
X<int> x(3);            // An object of type "X of int"
template <class T, class U=T, int n=0> // Template with default
parameters
```

class was the original defined term for template type overloading, but was redefined to two types; typename and class. The two is almost complete alike.
One difference (warning: and only to my knowledge – but I might be wrong) is that we can use typename as a dependent type in declarations, like;

```
            typedef typename T::iterator itr_type;
```

NAMESPACES

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

C/C++ STANDARD LIBRARY

Only the most commonly used functions are listed. Header files without .h are in namespace std. File names are actually lower case.

STDIO.H, CSTDIO (input/output)

```
FILE* f=fopen("filename", "r");      // open for reading, NULL (0)
if error // Mode may be "w" (write) "a" append, "a+" update, "rb"
binary.
fclose(f);                 // close file f
fprintf(f, "x=%d", 3)      // print "x=3"
           "%5d %u %-8ld"          // int width 5, unsigned int,
long left just.
           "%o %x %X %lx"          // octal, hex, HEX, long hex
           "%f %5.1f"              // float or double
           "%e %g"                 // 1.23e2, use either for g
           "%c %s"                 // char, char*
           "%%"                    // %
sprint(s, "x=%d", 3);      // Print to array of char s
printf("x=%d", 3)          // print to stdout
fprint(stderr, ...         // print to standard error (not redirected)
getc(f);                   // Read one char (as an int) or EOF from f
ungetc(c, f);              // Put back one c to f
getchar();                 // getc(stdin)
putc(c,f)                  // fprint(f, "%c", c);
putchar(c);                // putc(c, stdout);
fgets(s, n, f)             // read line into char s[n]
gets(s)                    // fgets(s, INT_MAX, f); no bounds check
fread(s, n, l, f);         // read n bytes from f to s
fwrite(s, n, l, f);        // write n bytes of s to f
fflush(f);                 // Flush
fseek(f, n, SEEK_SET)      // Position binary file f at n
ftell(f)                   // fseek(f, 0L, SEEK_SET); clearer(f);
feof(f);                   // Is f at end of file?
ferror(f);                 // Error in f?
perrror(s);                // print char*s and error message
clearer(f);                // Clear error code for f
remove("filename");        // Delete file, return 0 if OK
rename("old", "new");      // Rename file, return 0 if OK
f = tmpfile();             // Crate temporary file in mode "wb+"
tmpnam(s);                 // Put a unique file name in char
s(L_tmpnam)
```

STDLIB.H, CSTDLIB (Misc. functions)

```
atof(s); atol(s); atoi(s); // Convert char* s to float, long, int
srand(), srand(seed)       // Random int 0 to RAND_MAX, reset rand()
void* p = malloc(n);       // Allocate n bytes - use new instead (C++)
free(p);                   // Free memory. Use delete instead (C++)
exit(n);                   // Kill program
system(s);                 // Execute OS command s
getenv("PATH");            // Environment variable or 0
abs(n); labs(ln);          // Absolute value as int, long
```

STRING.H, CSTRING (Character array handling functions)

Strings are type char[] with a '\0' in the last element used. Which is why you cannot directly compare a C-array char* with a C++ string "string".

```
strcpy(dst, src);          // Copy string
strcat(dst, src);          // Concatenat to dst.
strcmp(s1, s2);            // Compare
strncpy(dst, src, n)       // Copy up to n chars
strlen(s);                 // Length of s not counting \0
strchr(s,c); strrchr(s,c)// address of first/last char c in s
strstr(s, sub);            // address of first substring in s
memmove(dst, src, n);      // Copy n bytes from src to dst
memcmp(s1, s2, n);         // Compare n bytes as in strcmp
memchr(s, c, n);           // Find first bytes c in s
memset(s, c, n);           // Set n bytes of s to c
```

CTYPE.H, CCTYPE (Character types)

```
isalnum(c);                // Is c a letter or digit?
isalpha(c); isdigit(c); // Is ca latter? digit?
islower(c); isupper(c); // Is c lower case? Upper case?
tolower(c); toupper(c); // Convert c to lower/upper case
```

MATH.H, CMATH (Floating point math)

```
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
exp(x); log(x); log10(x); // e to be 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
```

TIME.H, CTIME (Clock)

```
Clock()/CLOCKS_PER_SEC  // Time in seconds since program started
time_t t=time(0)           // Absolute time in seconds or -1 if unkown
tm* p=gmtime(&t)           // 0 if UCT unavalibale else pm->tm_X where
X is:
```

```
  sec, min, hour, mday, mon (0-11), year (-1900), wday, yday, isdst
asctime(p);             // "Day Mon dd hh:mm:ss yyyy\n"
asctime(localtime(&t))  // Same format, local time
```

ASSERT.H, CASSERT (Debugging aid)

```
assert(e);              // if e is false, print message and abort
#define NDEBUG          // (before #include <assert.h>), turn off
assert
```

NEW.H, NEW (Out of memory handler)

```
set_new_handler(handler); // change behavior when out of memory
void handler(void) {throw bad_alloc();} // default
```

IOSTREAM.H, IOSTREAM (Replace stdio.h)

```
cin >> x >> y;          // Read words x and y (any type) from stdin
cout << "x=" << 3 << endl; // Write line to stdout
cerr << x << y << flush; // Write to stderr and flush
c = cin.get();          // Read char
cin.get(c);             // Read char
cin.getline(s, n, \n)   // Read line into char s[n] to '\n'
(default)
                        //To read/write any type T:
istream& operator>>(istream& I, T& x) {i >> ...; x=...; return i;}
ostream& operator<<(ostream& o, const T& x) {return o << ...;}
```

You can overload them in your class, doing so make a default association for the class for printing to string, like:

```
            MyClass my; cout << my << endl; (Will call operator<<)
```

FSTREAM.H, FSTREAM (File I/O works like cin, cout as above)

```
ifstream f1("filename");// Open text file for reading
if (f1)                 // Test if open and input available
  f1 >> x;              // Read object from file
//If defined, a class can deploy the same method for operator>>.
            MyClass my; f1 >> my;
f1.get(s);              // Read char or line
f1.getline(s, n);       // Read line into string s[n]
ostream f2("filename"); // Open file for writing
if (f2) f2 << x;        // Write to file
```

IOMANIP.H, IOMANIP (Output formatting)

```
cout << setw(6) << setprecision(2) << setfill('0') << 3.1; //print
"003.10"
```

STRING (Variable sized character array)

```
string s1, s2 = "hello";// Create strings
s1.size(), s2.size()    // Number of characters: 0, 5
s1 += s2 + ' ' + "world" // Concatenation
s1 == "hello world"     // Comparison, also <,>, !=, etc.
// Recall that string (C++) have a \0 end character, and therefore
cannot be directly compared with a C-string of char*.
s1[0];                  // h - element 0 of a string
s1.substr(m, n);        // subdivide string from s1[m] to +n
s1.c_str();             // Convert to const char*
getline(cin, s);        // Read line ending in '\n'
```

VECTOR (Variable sized array/stack with built in memory allocation)

```
vector<int> a(10);      // a[0]..a[9] are int (default size is 0)
a.size();               // Number of elements : 10
a.push_back(3);         // Increase size to 11, a[10] == 3;
a.back() = 4;           // a[10] = 4
a.pop_back();           // Decrease size by 1
a.front();              // a[0]
a[20] = 1               // error: out of bounds
a.at(20) = 1;           // Like a[20] but throws out_of_range()
__ITERATOR__

for(vector<int>::iterator p = a.begin(); p != a.end(); p++)
            *p = 0      // Set all elements of a to 0
vector<int> b(a.begin(), a.end()); // b is copy of a
vector<T> c(n, x);      // c[0]..c[n-1] init to x
T d[10]; vector<T> e(d, d+10); //e is initialized from d
```

DEQUE (array/stack/queue)

```
Deque<T> is like vector<T>, but also supports:
a.push_front(x);        // Puts x at a[0], shifts elements toward
back
a.pop_front();          // Removes a[0], shifts toward front
```

UTLITY (Pair)

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

MAP (associate array)

```
map<string, int> a;     // Map from string to int
a["hello"] = 3;         // Add or replace element a["hello"]
```

```
for (map<string, int>::iterator p=a.begin(); p != a.end(); ++p)
  cout << (*p).first << (*p).second; // prints hello, 3
a.size();               // 1
```

ALGORITHM (A collection of algorithms)

```
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 <
//You can overload operator< in a class and then use;
            List<MyClass> lc; lc.sort();
//and it'll automatic associate operator< with the operation for
sorting.
sort(a.begin(), a.end()); // Sort container with iterator.
sort(a.begin(), a.end(), function); // Sort the container with a
specific comparison function. Use a non-member function or use a
functor.
```

BUILT-IN TYPES

INTEGER TYPES	BITS	RANGE
bool	1	false (0) or true(1)
signed char	8	(-128 to 127)
unsigned char	8	(0 to 255)
char	8	Usually signed
short	16	-32768 to 32767
unsigned short	16	0 to 65535
int	32	-2147483648 to 214783647
unsigned int	32	0 to 4294967295
long	32-64	at least int
unsigned long	32-64	at least unsigned int

FLOATING POINT TYPES	BITS	RANGE
float	32	-1.7e38f to 1.7E38F
double	64	-1.8e308 to 1.8E308
long double	64-80	at least double