CS 150 Quick-Reference

```
int main() { return 0; }  // basic C++ structure
```

Standard Library Headers

```
#include <iostream>   // input-output streams
#include <iomanip>    // stream manipulators
#include <fstream>    // file streams
#include <sstream>    // string streams
#include <stdexcept>  // standard exceptions
#include <cmath>      // all math functions
#include <string>     // c++ style strings
#include <vector>     // c++ vector class
#include <cctype>     // character classification
#include <cstdlib>    // exit codes, abs for ints

using namespace std;    // using directive (all names)
using std::cout;        // using declaration (specific name)

std::string s;          // full qualification
```

Comments

```
// single-line comments
/* multi-line or inline */
/** documentation (Doxygen) comments */
```

Input/Output

```
cout << anything << endl;        // only need iostream
cout << fixed << setprecision(2)
    << setw(12) << value;        // need iomanip
cin >> anyVar;                   // skips whitespace, reads one token (word)
cin >> noskipws >> ch;           // reads a character including whitespace
getline(cin, stringVar);         // needs <string> include, reads line

ofstream out("output.txt");      // creates an output file stream out
ostringstream sout;              // creates an output string stream sout
ifstream in("input.txt");        // a file input stream from "input.txt"
istringstream sin("Some text");  // creates an input string stream
string result = sout.str();      // convert ostringstream output to a string
```

// Escape sequences

\t = tab, \n = newline, \" = quote, \\ = backslash \' = single-quote

// Selection—simple if (independent decisions)

if (condition) *// boolean or numeric (non-zero) value*

```
   statement;          // multi-line? enclose in { } block
else
   statement;
```

// Selection—nested if (leveled decisions)

```
if (conditionA)
   if (conditionB)
      statement-if A and B
   else
      statement-if A and not B
else
   if (conditionB)
      statement-if not A and B
   else
      statement-if not A and not B
```

// Selection—sequential if (dependent decisions)

```
if (conditionA)
   statement-if A
else if (conditionB)
   statement-if B
else if (conditionC)
   statement-if C
else
   statement-if not A or B or C
```

// Selection—numbered decisions (single test against a constant)

```
switch (integer-expression-test)
{                              // braces required
   case 1:                     // case block for integer-expression == 1
      statement;
      statement;
      break;                   // needed to end block; fall-through otherwise
   case 5:                     // case block for integer-expression == 5
      statement;
      statement;
      break;                   // needed to end block; fall-through otherwise
   default:                    // optional block (else for switch)
      statement;               // if expression != 1 and != 5
}
```

// while Loops

```
int num, sum = 0;   // Sentinel summing loop, primed variety
cin >> num;
while (cin && num >= 0) // test both cin and num; exit on negative number
{
   sum += num;
   cin >> num;    // read number again to go to next iteration
}

int num, sum = 0;                    // Sentinel summing loop with inline-test
while ((cin >> num) && num >= 0) // test both cin and num; exit on negative number
{
   sum += num;
}

string str;                 // Assume these three statements in front of rest of loops
getline(cin, str);
size_t len = str.size();     // or str.length()

int vowels = 0;  // Counted summing loop (process string or array)
size_t i = 0;
while (i < len)             // always make sure you go < len
{
   char c = str.at(i);   // less safe: str[i]; as string: str.substr(i, 1)
   if (c == 'a' || c == 'e' || str == 'i' || str == 'o' || str == 'u')
      vowels++;
   i++;   // don't forget to update the counter
}
```

// for Loops, strings and cctype functions

```
string str;                 // Assume these three statements in front of rest of loops
getline(cin, str);
size_t len = str.size();     // or str.length()

int digits = 0;   // using a for loop instead
for (size_t i = 0; i < len; i++)
   if (isdigit(str.at(i))) digits++;
```

//Other cctype functions: ispunct(), isspace(), isupper(), islower(), isalpha(), toupper(), tolower()

// strings and substrings (building up new output pattern)

```
string result = "";                    // remove all "dog"s from str
for (size_t i = 3; i < len; i++)       // note condition is < len – (3 – 1)
{
   string subs = str.substr(i - 3, 3);   // note difference from Java; second is length
   if (subs == "dog")
      i += 2;  // skip over dog in output
   else if (i == len – 3)   // last three characters not dog
   {
      result += subs;
      i += 2;
   }
   else
      result += subs.at(0);   // put next character in output
}
```

// string functions

```
string s = "How now brown cow";
s.length() → 17    (type is size_t; may save in an int)

s.find("How") → 0
s.find("how") → string::npos

s.at(0) → 'H'        range-checked if out of bounds
s[0] → 'H'           not range-checked if out of bounds

s.substr(0, 1) → "H"
s.substr(4, 3) → "now"
s.substr(14, 3) → "cow"
s.substr(14, 1000) → "cow"
s.substr(18, 2) → runtime exception
```

// Structures

Declaration; normally appears in a header file. Must be seen before type is used.

```
struct Employee                 // name of the new "type"
{
   std::string name;            // remember std:: if in header
   int age;                     // no initial value when declared
   double salary;
};                              // don't forget semicolon
```

Initialize a structure variable

```
Employee bob = {"Robert", 27, 72953.50};
```

Assign to members

```
bob.age = 28;
```

Composite or group assignment

```
Employee bob2;
bob2 = bob;              // All members copied
```

No composite comparison allowed (without some extra work)

```
if (bob2 == bob)
```
// syntax error here

// vectors

```
vector<int> v;                  // empty vector, no elements
vector<int> v1(5);              // vector with 5 elements, set to 0
vector<int> v2{1, 2, 3};        // vector with 3 elements, initialized to 1,2,3.2011 ONLY

v2.size() → 3;                  //(type is technically vector::size_type; may save in an int)

v2[0] → 1
v2.at(0) → 1

v2.push_back(4) → {1, 2, 3, 4}
v2.pop_back() → {1, 2, 3}
v2.front() → 1
v2.back() → 3
```

// Defining Classes

Definition normally appears in a header file. Library types must be qualified.
Must be seen before new type is used.

```
class Employee                    // name of the new "type"
{
public:                                  // the interface section
   Employee();                           // the default constructor
   explicit Employee(const std::string& name);          // a conversion constructor
   Employee(const std::string& name, double salary);    // working constructor

   std::string getName() const;      // accessor (const)
   double getSalary() const;         // accessor (const)

   void setSalary(double salary);            // mutator
   void setName(const std::string& name);    // mutator

   Employee& operator+=(double raise); // member overloaded operator

private:    // the private implementation
   std::string name;              // remember std:: if in header
   double salary;
};                                // don't forget semicolon
```

Non-member output operator

```
std::ostream& operartor<<(std::ostream& out, const Employee& e);
```

// Using Classes (creating and messaging objects)

Initialize some Employee objects

```
Employee bob("Robert", 27, 72953.50);
Employee bill("William");
Employee unsub();
```

Access data members

```
cout << bob.getName() << ", $" << bob.getSalary() << endl;
bill.setSalary(bob.getSalary() * 1.2);
```

Overloaded Operators

```
cout << "Before raise: " << bob << endl;
bob += 10000.0;
cout << "After raise: " << bob << endl;
```

// Implementing Classes

Normally appears in an implementation (.cpp) file. Library types do not need to be qualified, but you do need the correct using declaration. You must qualify each method->returntype ClassName::method() {..

```
#include "employee.h"                    // implementation must "see" the definition

Employee::Employee() : salary(0) { }     // using the initializer list; must initialize primitives
Employee::Employee() { salary = 0 }      // alternative implementation without initializer list

// Three alternative implementations of conversion constructor. (Leave off explicit in implementation)
Employee::Employee(const string& name)
{
   this->name = name;
   salary = 0;
};
Employee::Employee(const string& name)
{
   Employee::name = name;
   salary = 0;
};
Employee::Employee(const string& name) : name(name), salary(0) { }

// Accessor methods
string Employee::getName() const { return name; }
double Employee::getSalary() const { return salary; }

// Mutator methods
void Employee::setSalary(double salary) { Employee::salary = salary; }

// Overloaded operator. Note that += modifies the object so that it returns a reference to this
// You will need to think about what each operator returns to know how to implement it.
// This is a binary operator. As a member function it takes only one argument.
Employee& Employee::operator+=(double raise)
{
   setSalary(salary + raise);
   return *this;  // reference to current object
}

// Non-member overloaded output.
ostream& operator<<(ostream& out, const Employee& e)
{
   out << fixed << setprecision(2);
   out << e.getName() << ", $" << e.getSalary();
   return out;
}
```