**C++ Survival Guide**

Version 8.2

**Basic Notes on Syntax**

**of**

**pointers, references, classes,**

**strings, streams, and vectors**

Jim Fawcett

09 January 2015**C++ Pointers and References:**

1. **Create pointers and references:** Note: & in declaration is a reference, & in expression is an address, for example,

& on left of assignment is a reference, & on right of assignment is an address

* 1. int x = 23; // declare and define x   
     int \*pInt = &x; // create pointer to x
  2. int y[4] = { 1, 2, 3, 4 }; // declare and define array of ints  
     int \*pIntArray = y; // point to beginning of array
  3. struct CStructType { int x; double d; char z; } CStruct = { 3, -23.5, ‘z’ };  
      // declare a structure type and define one  
     CStructType \*pStr = &CStruct; // create a pointer to that structure
  4. int& rX = x; // create a reference to an integer
  5. int& fun(const int &x) { … } // create a reference on the stack frame of fun and return a reference to something

1. **Use pointers and references:**
   1. int z = \*pInt; // return the contents of the location pointed to
   2. \*pInt = -23; // change the value of the location pointed to
   3. \*(pIntArray +2) = 5; // same as y[2] = 5;
   4. pStr->d = 3.1415927; // change the value of CStruct.d
   5. int w = rX; // return value of reference, e.g., value of x
   6. rX = 15; // modify value of reference, e.g., value of x
   7. int u = fun(x); // create a reference to x on the stack frame of fun. If fun changes this value then  
       // the caller’s value is also changed. Assign the value of the returned integer to u.
2. **Allocating and deallocating memory:**When new is invoked, memory is allocated and then initialized with a class constructor to create a functioning object.  
    When delete is invoked, the class destructor is called on that object before the heap memory allocation is returned.  
   1. CStructType \*pStr = new CStructType; // allocate a CStructType object on the dynamic heap
   2. delete pStr; // return the dynamic memory allocation to the process
   3. char \*pCs = new char[10]; // allocate an array of 10 chars on the heap
   4. delete [] pCs; // deallocate the entire array

References:

1. The C++ Programming Language, Stroustrup, Addison-Wesley, 1997, Chap 2 & 4
2. [www.ecs.syr.edu/faculty/fawcett/handouts/cse687/code/basic/basic0.cpp](http://www.ecs.syr.edu/faculty/fawcett/handouts/cse687/code/basic/basic0.cpp)

**C++ Classes:**

1. **Declare class:** Note: names of formal parameters, like f and val, have no syntactic value and can be omitted.class cl {  
    public:  
    cl(); // default constructor  
    cl(const cl& f); // copy constructor  
    cl(cl&& f) // move constructor  
    cl& operator=(const cl& f) // copy assignment  
    cl& operator=(const cl&& f) // move assignment  
    cl(int val); // promotion constructor  
    ~cl(); // destructor  
    int& access(); // accessor  
    private:  
    int value; // data member  
   };
2. **Define class members** (more complex implementations elided)**:**  
   cl::cl() : value(0) { } // create cl with value initialized to zero  
   cl::cl(const cl& f) : value(f.value()) { } // create cl object as a copy of f  
   cl::cl(int val) : value(val) { } // create cl object with value = val  
   cl::~cl() { } // destroy cl object – does nothing  
   int& cl::access() { return value; } // provide read/write access to value  
    // move construction and assignment will be discussed in class
3. **Create and use an object of cl class**cl f; // create cl object with f1.value = 0  
   cl f1 = f; // create cl object with f1.value = f.value  
   cl f2(15); // create cl object with value = 15  
   int n = f2.access(); // read cl::value  
   f2.access() = 23; // modify cl::value

References:

1. The C++ Programming Language, Stroustrup, Addison-Wesley, 1997, Chap 10
2. <http://www.ecs.syr.edu/faculty/fawcett/handouts/CSE687/code/str/str.h>
3. [www.ecs.syr.edu/faculty/fawcett/handouts/CSE687/code/str/str.cpp](http://www.ecs.syr.edu/faculty/fawcett/handouts/CSE687/code/str/str.cpp)

**C++ Class Relationships:**

1. **Declare class used for composition**class C { // details omitted };
2. **Declare classes used by base and derived classes**class U1 { // details omitted }; class U2 { // details omitted };
3. **Declare base class:** //member function definitions omittedclass B {  
    public:  
    B() : C() { } // default constructor, one of two overloaded member functions  
    B(const B &b); // copy constructor, the other of two overloaded member functions  
    virtual void m1(U1 u1); // virtual member function may be overridden, uses a U1 object passed by value  
    virtual void m2(const U1 &u1); // virtual member function may be overridden, pass object by const reference  
    int m3(); // non-virtual member function ***should not be overridden***  
    virtual ~B(); // ***virtual***  destructor  
    private:  
    C c; // **composition relationship**  
    U1\* pU1 = new U1; // **aggregation relationship**  
   };
4. **Declare derived class** // member function definitions omittedclass D : public B { // **inheritance relationship**  
    D() : B(), pU2(0) { } // requiring base part constructed with B’s void ctor, initializing pU2 to null pointer  
    D(const D &d) : B(d), pU2(0) { } // requesting compiler to use B’s copy ctor to copy base part, also initializing pU2  
    virtual m1(U1 u1); // overriding (redefining) B::m1(U1), means for D to use U1 object  
    void register(U2 \*ptr) { pU2 = ptr}; // **using relationship** -means for D to use U2 object  
    // other details omitted  
    private:  
    U2 \*pU2; // **using relationship**  
   };
5. **Creating and using objects of these classes**C c; B b; D d; U1 u1; U2 u2; // creating all default objects  
   d.register(&u2); // give d access to u2  
   d.m1(u1); // invoke redefined m1

References:

1. <http://www.ecs.syr.edu/faculty/fawcett/handouts/CSE687/code/relationships>

**Standard C++ Strings:**

C++ strings represent arrays of characters. You do not have to provide any memory management operations – C++ strings   
take care of that for you.

1. **Access string library:**

#include <string>

1. **Create a string:**
   1. std::string s; // empty string
   2. std::string s = “this is C string”; // promote a C-string
   3. std::string s1 = s2; // copy
2. **Append character or string:**
   1. s += ‘a’; // silently allocates more memory if needed
   2. s += “more stuff”; // “ “ “ “ “ “
3. **Assignment:**
   1. s2 = s1;
   2. s2 = “new contents”; // create temp and assign
4. **Access characters:**
   1. char ch = s[1]; // read 2nd character
   2. s[2] = ‘z’; // modify third character
   3. ch = s.at(3); // throw out of range exception
   4. const char \*pStr = s.c\_str(); // returns pointer to char array
5. **Array size:**
   1. unsigned int len = s.size();
   2. s.resize(3); // truncates or expands
   3. s.erase(2,3); // remove 3 chars starting at s[2]
6. **Find char or substring:**
   1. size\_t pos = s.find(‘z’); // find first ‘z’
   2. size\_t pos = s.find(‘z’,5); // find first ‘z’ at or after s[5]
   3. size\_t pos = s.find(“foo”,5);
   4. size\_t pos = s.find(s1,5); // see also find\_last\_of(….)

References:

1. The C++ Standard Library, Josuttis, Addison-Wesley, 1999, Chap 11
2. The C++ Programming Language, Stroustrup, Addison-Wesley, 1997, Chap 20

**Standard C++ iostreams**

C++ streams provide connections between your program And the platform’s input and output devices.

1. **Access iostreams library:** #include <iostream>
2. **Create:**
   1. std::istream in;
   2. std::ostream out;
   3. std::cin, std::cerr, and std::cout are created for you by the iostream library
3. **Read:**
   1. in >> x; // attempts to read value[[1]](#footnote-1) of an object of type x,  
       // throwing away leading whitespace
   2. int i = in.get(); // unformatted read single extended char
   3. in.get(ch); // unformatted read
   4. in.get(buffer,bufferSize,’\n’); // reads a line, if it fits into bufferSize
   5. in.putback(ch); // returns a single char to in – don’t call twice
   6. in.read(buffer,bufferSize); // read up to bufferSize chars
4. **Write:**
   1. out << x; // if type of x is known to ostream, e.g., all the primitive types,   
       // value of x is written to stream1
   2. out.put(ch); // write a char to out stream
   3. out.write(buffer,bufferSize); // write a buffer of chars to out
   4. out.flush(); // forces contents of internal streambuf to be sent to output device
5. **Stream state:**
   1. bool b = in.good(); // is the state good(), bad(), fail()?
   2. in.clear(); // reset stream state to good so you can use it again

References:

1. The C++ Standard Library, Josuttis, Addison-Wesley, 1999, Chap 13
2. The C++ Programming Language, Stroustrup, Addison-Wesley, 1997, Chap 21
3. [www.ecs.syr.edu/faculty/fawcett/handouts/cse687/code/iostreams](http://www.ecs.syr.edu/faculty/fawcett/handouts/cse687/code/iostreams)

**Standard C++ fstreams:**

C++ fstreams represent a connection between your program and files in your platform’s file system.

1. **Access fstreams library:** #include <fstream>
2. **Create:**
   1. std::ifstream in(filename); // create and attach to a file if possible
   2. std::ifstream in; // create an unattached stream  
      in.open(filename); // attempt to attach stream to file  
      in.close(); // release attachment
   3. std::ofstream out(filename); // create and attach to a file if possible
   4. std::ofstream out; // create an unattached stream  
      out.open(filename); // attempt to attach stream to file  
      out.close(); // release attachment
3. **Read:**
   1. in >> x; // attempts to read value1 of An object of type x, throwing away leading whitespace
   2. int i = in.get(); // unformatted read single extended char
   3. in.get(ch); // unformatted read
   4. in.get(buffer,bufferSize,’\n’); // reads a line, if it fits into bufferSize
   5. in.putback(ch); // returns a single char to in – don’t call twice
   6. in.read(buffer,bufferSize); // read up to bufferSize chars
4. **Write*:***
   1. out << x; // if type of x is known to ostream, e.g., all the primitive types, value of x is written to stream1
   2. out.put(ch); // write a char to out stream
   3. out.write(buffer,bufferSize); // write a buffer of chars to out
   4. out.flush(); // forces contents of internal streambuf to be sent to output device
5. **Streamstate*:***
   1. bool b = in.good(); // is the state good(), bad(), fail()?
   2. in.clear(); // reset stream state to good so you can use it again
6. **Changestreamposition*:***
   1. in.seekg(pos); // go to pos bytes from beginning of file, pos must be ios::pos\_type
   2. in.seekg(offset, pos); // go to pos+offset bytes, pos must be ios::beg, ios::cur, or ios::end
   3. ios::pos\_type pos = in.tellg(); // record current file position
   4. out.seekp(pos); // go to pos bytes from beginning of file, pos must be ios::pos\_type
   5. out.seekp(offset, pos); // go to pos+offset bytes, pos must be ios::beg, ios::cur, or ios::end

References:

1. The C++ Standard Library, Josuttis, Addison-Wesley, 1999, Chap 13
2. The C++ Programming Language, Stroustrup, Addison-Wesley, 1997, Chap 21
3. [www.ecs.syr.edu/faculty/fawcett/handouts/cse687/code/iostreams](http://www.ecs.syr.edu/faculty/fawcett/handouts/cse687/code/iostreams)

**Standard C++ stringstreams:**

C++ string streams allow you to interact with in-memory buffers using stream operations. Especially important is the   
 format conversions that streams provide between primitive data types and characters.

1. **Access stringstreams library:** #include <sstream>
2. **Create:**
   1. std::istringstream in(s); // create istringstream in, holding C++ string s in its streambuf
   2. std::ostringstream out; // create empty istringstream object
3. **Read:**
   1. in >> x; // attempts to read value1 of an object of type x,  
       // throwing away leading whitespace
   2. int i = in.get(); // unformatted read single extended char
   3. in.get(ch); // unformatted read
   4. in.get(buffer,bufferSize,’\n’); // reads a line, if it fits into bufferSize
   5. in.putback(ch); // returns a single char to in – don’t call twice
   6. in.read(buffer,bufferSize); // read up to bufferSize chars
4. **Write:**
   1. out << x; // if type of x is known to ostream, e.g., all the primitive types,   
       // value of x is written to stream1
   2. out.put(ch); // write a char to out stream
   3. out.write(buffer,bufferSize); // write a buffer of chars to out
   4. out.flush(); // forces contents of internal streambuf to be sent to output device
5. **Access internal string:**
   1. std::string s = in.str(); // returns internal streambuf string as a standard C++ string
   2. std::string s = out.str(); // returns internal streambuf string as a standard C++ string

References:

1. The C++ Standard Library, Josuttis, Addison-Wesley, 1999, Chap 13
2. The C++ Programming Language, Stroustrup, Addison-Wesley, 1997, Chap 21
3. [www.ecs.syr.edu/faculty/fawcett/handouts/cse687/code/iostreams](http://www.ecs.syr.edu/faculty/fawcett/handouts/cse687/code/iostreams)

**Standard C++ Iterators and Vectors:**

C++ iterators act like pointers on steroids. C++ vectors act like generic extendable arrays that manage their own memory for you.

1. **access library for vector container And its iterators:**#inclu*de <vector>*
2. **create:**
   1. std::vector<int> vint; // create an empty vector of integers
   2. std::vector<double> vdouble(10); // create a vector with space to hold 10 doubles
   3. std::vector<int> v = vint; // copy an existing vector
   4. std::vector<int>::iterator firstit = vint.begin(); // create an iterator pointing to the first element of vint
   5. std::vector<int>::iterator endit = vint.end(); // create an iterator pointing to one past the last element of vint
3. **add and remove elements**:
   1. vint.push\_back(3); // put the integer value 3 at the end of the vector. Reallocate memory   
       // if there is not enough to hold the new element.
   2. Std::vector<double>::iterator it = vdouble.begin(); // create an iterator pointing to the beginning of vdouble  
      vdouble.insert(it, 3.1415927); // insert a double value at the element pointed to by iterator it
   3. double d = vdouble.pop\_back(); // remove the last item from the vector
   4. std::vector<int>::iterator first = ++vint.begin(); // create iterator pointing to beginning of vint, then move forward one  
      std::vector<int>::iterator last = --vint.end(); // create an iterator pointing one past the end of vint, then back up one.  
      vint.erase(first, last); // erase All but the first and last elements.
4. **size:**
   1. size\_t len = vdouble.size(); // returns number of elements in vector
   2. vdouble.resize(10); // expands or truncates vdouble
5. **access to elements:**
   1. vdouble[m] = -2.8e-13; // will throw an exception if vdouble.size() < m+1
   2. double d = vdouble[n]; // will throw an exception if vdouble.size() < n+1
   3. std::vector<double>::iterator it = vdouble.begin() + 3;  
      double d = \*it; // access value of fourth element in vdouble

References:

1. The C++ Standard Library, Josuttis, Addison-Wesley, 1999, Chaps 6 & 7
2. The C++ Programming Language, Stroustrup, Addison-Wesley, 1997, Chap 17 & 19
3. [www.ecs.syr.edu/faculty/fawcett/handouts/cse687/code/STL](http://www.ecs.syr.edu/faculty/fawcett/handouts/cse687/code/STL)

1. Note that this may imply a format conversion from the storage type, e.g., chars in a file, to the in-memory type, e.g., double. If the read fails, the stream state will go bad. [↑](#footnote-ref-1)