C++ Survival Guide

Version 7

Basic Notes on Syntax
of
pointers, references, classes,
strings, streams, and vectors

Jim Fawcett

29 January 2007

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

```
a.  int x = 23;                      // declare and define x
    int *pInt = &x;                  // create pointer to x
b.  int y[4] = { 1, 2, 3, 4 };       // declare and define array of ints
    int *pIntArray = y;              // point to beginning of array
c.  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
d.  int &rX = x;                     // create a reference to an integer
e.  int& fun(const int &x) { … }     // create a reference on the stack frame of fun and return a reference to something
```

2. ***Use pointers and references:***

```
a.  int z = *pInt;                   // return the contents of the location pointed to
b.  *pInt = -23;                     // change the value of the location pointed to
c.  *(pIntArray +2) = 5;             // same as y[2] = 5;
d.  pStr->d = 3.1415927;             // change the value of CStruct.d
e.  int w = rX;                      // return value of reference, e.g., value of x
f.  rX = 15;                         // modify value of reference, e.g., value of x
g.  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.
```

3. ***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.

```
a.  CStructType *pStr = new CStructType;  // allocate a CStructType object on the dynamic heap
b.  delete pStr;                          // return the dynamic memory allocation to the process
c.  char *pCs = new char[10];             // allocate an array of 10 chars on the heap
d.  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

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(int val);          // promotion constructor
    ~cl();                // destructor
    int& access();        // accessor
  private:
    int value;            // data member
};
```

2. ***Define class members:***

```
cl::cl() : value(0) { }                       // create cl with value initiClized 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
```

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:

3. The C++ Programming Language, Stroustrup, Addison-Wesley, 1997, Chap 10
4. http://www.ecs.syr.edu/faculty/fawcett/handouts/CSE687/code/str/str.h
5. 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 omitted

```
class 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
    // other members
  private:
    C c;                                  // composition relationship
    // other member data
};
```

4. ***declare derived class*** // member function definitions omitted

```
class D : public B {
    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};  // means for D to use U2 object
    // other details omitted
  private:
    U2 *pU2;
};
```

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:

6. http://www.ecs.syr.edu/faculty/fawcett/handouts/CSE687/code/relationships

7. *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.

4. ***access string library:***
    #include <string>
5. ***create a string:***
    a. std::string s; // empty string
    b. std::string s = “this is C string”; // promote a C-string
    c. std::string s1 = s2; // copy
6. ***append character or string:***
    a. s += ‘a’; // silently allocates more memory if needed
    b. s += “more stuff”; // " " " " " "
7. ***assignment:***
    a. s2 = s1;
    b. s2 = “new contents”; // create temp and assign
8. ***access characters:***
    a. char ch = s[1]; // read 2nd character
    b. s[2] = ‘z’; // modify third character
    c. ch = s.at(3); // throw out of range exception
    d. const char *pStr = s.c_str(); // returns pointer to char array
9. ***array size:***
    a. unsigned int len = s.size();
    b. s.resize(3); // truncates or expands
    c. s.erase(2,3); // remove 3 chars starting at s[2]
10. ***find char or substring:***
    a. size_t pos = s.find(‘z’); // find first ‘z’
    b. size_t pos = s.find(‘z’,5); // find first ‘z’ at or after s[5]
    c. size_t pos = s.find(“foo”,5);
    d. size_t pos = s.find(s1,5); // see also find_last_of(….)

References:

6. The C++ Standard Library, Josuttis, Addison-Wesley, 1999, Chap 11
7. 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:***
   a. std::istream in;
   b. std::ostream out;
   c. std::cin, std::cerr, and std::cout are created for you by the iostream library
3. ***read:***
   a. in >> x; // attempts to read value[1] of an object of type x,
   // throwing away leading whitespace
   b. int i = in.get(); // unformatted read single extended char
   c. in.get(ch); // unformatted read
   d. in.get(buffer,bufferSize,’\n’); // reads a line, if it fits into bufferSize
   e. in.putback(ch); // returns a single char to in – don’t call twice
   f. in.read(buffer,bufferSize); // read up to bufferSize chars
4. ***write:***
   a. out << x; // if type of x is known to ostream, e.g., all the primitive types,
   // value of x is written to stream[1]
   b. out.put(ch); // write a char to out stream
   c. out.write(buffer,bufferSize); // write a buffer of chars to out
   d. out.flush(); // forces contents of internal streambuf to be sent to output device
5. ***stream state:***
   a. bool b = in.good(); // is the state good(), bad(), fail()?
   b. 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

[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.

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

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

References:

4. The C++ Standard Library, Josuttis, Addison-Wesley, 1999, Chap 13
5. The C++ Programming Language, Stroustrup, Addison-Wesley, 1997, Chap 21
6. 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:***

    #include <vector>

2. ***create:***

    a. std::vector<int> vint; // create an empty vector of integers
    b. std::vector<double> vdouble(10); // create a vector with space to hold 10 doubles
    c. std::vector<int> v = vint; // copy an existing vector
    d. std::vector<int>::iterator firstit = vint.begin(); // create an iterator pointing to the first element of vint
    e. std::vector<int>::iterator endit = vint.end(); // create an iterator pointing to one past the last element of vint

3. ***add and remove elements***:

    a. 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.
    b. 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
    c. double d = vdouble.pop_back(); // remove the last item from the vector
    d. 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:***

    a. size_t len = vdouble.size(); // returns number of elements in vector
    b. vdouble.resize(10); // expands or truncates vdouble

5. ***access to elements:***

    a. vdouble[m] = -2.8e-13; // will throw an exception if vdouble.size() < m+1
    b. double d = vdouble[n]; // will throw an exception if vdouble.size() < n+1
    c. std::vector<double>::iterator it = vdouble.begin() + 3;
    double d = *it; // access value of fourth element in vdouble

References:

7. The C++ Standard Library, Josuttis, Addison-Wesley, 1999, Chaps 6 & 7
8. The C++ Programming Language, Stroustrup, Addison-Wesley, 1997, Chap 17 & 19
9. www.ecs.syr.edu/faculty/fawcett/handouts/cse687/code/STL