# C++ Part II

Elliott Wolin
Summer Student Lecture Series
JLab
20-Jul-2007

# Outline

- Introduction
- Safe Programming
- Efficient Programming
- Topics Not Covered
- Conclusions

# Introduction

- C++ is huge, powerful
- Many ways to do things incorrectly
- Error/debugger support poor
- Program safely, efficiently from the start
  - Develop good habits early and stick with them
  - Extra time up front pays off!
  - Program lifecycle dominated by maintenance!

# Programming Strategy

- modelling
- is-a vs has-a
- patterns

# Safe Programming

- Const correctness
- Exceptions
- Safe casting

# Const Correctness

- Vastly improves reliability

```
// const for variables
T myT;                          // T is some type
T *tPtr;                        // pointer to T
const T *tPtr;                  // pointer to const T;
T *const tPtr = &myT;           // const pointer to T;
const T *const tPtr = &myT; // const pointer to const T
```

# Const Correctness

```
// compare the following two prototypes where the function should
//  not change any of its arguments

// not safe and/or efficient
int myFunc(int i, int *iPtr, myObj o, myObj *oPtr);

// safer and more efficient
int myFunc(int i, const int *iptr, const myObj &o, const myObj *oPtr);
```

# Const member functions

- Says that method does NOT change object
- Allows method to be called via const pointer or reference

```
class MyClass {

private:
  int val;

public:
  void myConstMethod(int x) const {
    cout << val << endl;
    …
  }

};
```

# Exceptions

- Distinguish normal from error return
- Promotes much cleaner code

```
// c/Fortran style
int myFunc(int x, int *result);

int err, result1, result2;
err = myFunc(2,&result1);
if(err!=0) {…}
if(result1>7) {…}
err = myFunc(4,&result2);
if(err!=0) {…}
if(result2>8) {…}
…
```

```
// using exceptions
int myFunc(int x) throw(myException);

try {
  if(myFunc(2)>7) {…}
  if(myFunc(4)>8) {…}
  …
} catch (myException &e) {
  cerr << e.what() << endl;
}
```

# How to throw an exception

```
if(…)throw(MyException(28));   // constructor invoked


// alternate syntax, a la Geant4 examples
int err;
…
if(…)throw(MyException(err=28));
```

# Exception class

```
#include <exception>      // c++ default exception base class

class myException : public exception {

 private:
   int errCode;     //  the error code

 public:
   myException(int code) : errCode(code) {}

   // override exception base class method what(void)
   const char* what(void) const throw() {
      return("myException thrown, something bad happened!");
   }
};
```

# Throw specifications

```
int myFunc(int);              // may throw anything

int myFunc(int) throw(x);     // may throw x
                              // runtime error anything else thrown

int myFunc(int) throw();      // runtime error if anything thrown
```

# Safe Casting

- Do not use C-style casting!
- Old way just tricked the compiler

```
// the old way, for those who possess great faith
int myFunc(void *x) {
   MyObj *oPtr = (MyObj*)x;   // what if x is not a MyObj* ?
   …
}
```

# Safe Casting

- The new, safer way uses:
  - static_cast<>() // cast with minimal checking
  - dynamic_cast<>() // down-cast with run-time check
  - const_cast<>() // remove const-ness
  - reinterpret_cast<>() // don't use this

```
// the new way (templates explained later…)
int myFunc(void *x) {
   myObj *oPtr = static_cast<myObj*>(x);
}



// Example of down-casting with dynamic check

// assume Derived inherits from Base
Base *b = new Derived();
  …
Derived *d = dynamic_cast<Derived*>(b);
if(d!=NULL) {…}   // check if it worked
```

# Efficient Programming

- Variable scope and namespaces
- Strings and streams
- Operator overloading
- Templates
- Standard Template Library (STL)

# Variable Scope

- Variables usually exist between { and }
- Variable definition may hide another one

```
int myFunc() {
    int x = 1;
    if(…) {
       int y = 2;  // y only exists in the if clause, between the braces
       …
       int x = 3;  // creates a new variable, hides original!
       …
    }
    y = 10;  // ERROR: y is not defined here!
}
```

# Namespaces

- Avoid name clashes
- All packages should be in their own namespace

```
namespace fred {
  int x;
  float y;
  …
}
fred::x = 2;
```

or:

```
using namespace fred;
x = 2;
```

# Strings

- No more c-style char* needed!
- Use full-featured ANSI string class

```
#include <string>
using namespace std;          // otherwise must type std::string

string s;
s = "hello";
s += " world";                // can add to strings
if(s=="goodbye world") {…}    // case-sensitive string comparison
if(s.length()>10) {…}         // string length

oldFunc(s.c_str());           // can get c-style char* if needed
```

# Streams

- I/O, string streams, other streams

```
#include <iostream>
#include <iomanip>
using namespace std;                    // or else need std::cout, etc.

cout << "Hello World" << endl;          // prints to screen

#include <sstream>
stringstream ss;
int x = 123456;
ss << "x in hex is: 0x" << hex << x << ends;
cout << ss.str() << endl;               // convert to string and print

// use of stream paradigm
myContainer << anObject << anotherObject;
```

# Operator Overloading

- Can redefine the meaning of + - * / etc.

```
MyVector v1, v2, v3;
MyMatrix m1;

v3 = v1 + v2;     // operator + defined for myVector
v2 = m1 * v1;     // operator * defined between matrix and vector
m1++;             // operator ++ defined for matrix
v3 = 10 * v3;     // operator * defined for integer and vector
```

```
MyVector operator+ (const MyVector &vLeft, const MyVector &vRight) {
      // should do some error checking here…

      int len = vLeft.length;
      MyVector vSum(len);
      for(int i = 0; i<len; i++) { vSum[i] = vLeft[i] + vRight[i]; }
      return(vSum);
}
```

- Can redefine almost all C++ operators
  - () [] ^ % ! | & << >> and many others
- Useful, but easy to abuse
  - e.g. do NOT redefine + in non-intuitive way!
  - cannot redefine operators between built-in types

# Templates

- Generic programming, independent of type
- This is a very big subject!
- Not compiled unless used (instantiated)

```
template <typename T> class MyTemplateClass {
  T myT;                          // variable of type T
  static T min(T t1, T t2) {  // method takes 2 T's, returns T
      return((t1<t2)?t1:t2);
  }
};

// usage
MyClass<MyVector> v;     // MyVector flavor of MyTemplateClass
MyClass<int> i;          // int flavor of MyTemplateClass
```

# Standard Template Library

Large library of utilities, including:

- Containers
  - vector, list, map, queue, etc.
- Iterators
  - Iterate over objects in containers
- Algorithms
  - Do something to objects in containers (sort, etc.)
  - Generally very efficient
- Also, function objects, function object adaptors, utilities, etc.

# Containers

```
#include <vector>
using namespace std;

vector<int>   iVector(10);
vector<float> fVector;

for(int i=0; i<10; i++) {
  iVector[i]=i;
  fVector.push_back(i*1.234);
}
```

# Iterators

```
#include <list>
using namespace std;

list<int> l;
for(int i=0; i<10; i++) l.push_back(i);

// iterate over list contents
list<int>::iterator iter;
for(iter=l.begin(); iter!=l.end(); iter++) {
   cout << *iter << endl;   // *iter is current element in the list
}
```

# Algorithms

```
#include <vector>
#include <algorithm>
using namespace std;

// create vector of 100 random integers < 1000
vector<int>   vec;
for(int i=0; i<100; i++) vec.push_back(ran(1000));

// sort using STL algorithm
sort(vec.begin(), vec.end());
```

# Topics Not Covered

- Multi-threading
  - multiple, simultaneous execution threads
  - → concurrency problems, need locking mechanism
- Smart pointers
  - avoid memory leaks, no need to call "delete"
- RTTI
  - run-time type information
- Doxygen documentation facility
  - documentation derived from special comments in your code
- Member function pointers
  - can point to an member function of an object
- many others…

# Conclusions

- C++ is a big, complicated language
- "Use the good features of a language, avoid the bad ones", The Elements of Programming Style, Kernighan and Plauger, 1978
- Be familiar with C++ features, learn and use the good ones as needed
- Practice safe programming
  - const correctness, exceptions, safe casting, etc.
- Practice efficient programming
  - use strings, templates, the STL, etc.