C++ Programming for Scientists

Lecture # 6

Advanced C++:

Templates, Exceptions & Observations

C++ templates

The `template` facility in C++ essentially allows one to write functions and classes with variable *types*.
Remember our function for `max`?

```
inline double max(double a, double b)
{
    return (a > b ? a : b);
}
```

We noticed that was better than the macro

```
#define MAX(a,b)    ( a > b ? a : b)
```

because of type checking and evaluating its arguments only once. But our `inline` function only works for variables of type `double`.
C++ developers realized this problem and came up with a `template` mechanism to solve it.

```
template <class Type>
inline Type max(Type  a, Type b)
{
    return (a > b ? a : b);
}
```

This is just like the previous function, but with `double` replaced by a variable name `Type` and the extra line

`template <class Type>`

in the function declaration.

C++ templates (cont'd.)

Now we can call `max()` with any matching pair of types:

```
i = max( 31, 56 );            // calls int max(int, int)
x = max( 5.6, 9.2);           // calls double max(double, double)
c = max( 'c', 'A');           // call char max(char, char)
```

even user-defined types

```
i = max(BigInt("209209832"), BigInt("283745343"));
```

all that is required of a user-defined class is that `operator>` and `operator=` be defined on it. (You can see this directly from the definition of `max()`).
Notice however, that the types must match the templated function *exactly*, i.e.

```
max(3.1, 4)
```

won't work since the compiler would look for a template description of `max(double, int)`.

A Template Example for PCG

```
template < class Matrix, class Vector, class Preconditioner, class Real >
int
CG(const Matrix &A, Vector &x, const Vector &b,
   const Preconditioner &M, int &max_iter, Real &tol)
{
  Real resid;
  Vector p, z, q;
  Vector alpha(1), beta(1), rho(1), rho_1(1);

  Real normb = norm(b);
  Vector r = b - A*x;

    /* ... */
}
```

- Same source (cg.h) works for *any* matrix, vector or preconditioner consistent with the above interface.
- These types need to be known at compile-time.
- Argument classes (`Matrix`, `Vector`, `Preconditioner`, `Real`) have to satisfy the operators used in function `CG()`.
  - matrices and vector need operators '+', '*', etc
  - preconditioner `M`; `M` requires only two methods, those for finding the solution z to $Mz = r$ or $M^T z = r$.

Vector, Matrix Interface requirements

$scalar \leftarrow \textbf{dot}(\ Vector,\ Vector\ )$

$Vector \leftarrow Matrix\ \textbf{operator*}\ Vector$

$Vector \leftarrow Matrix^T\ \textbf{operator*}\ Vector$

$Vector \leftarrow Scalar\ \textbf{*}\ Vector$

$Vector \leftarrow Vector\ +/-\ Vector$

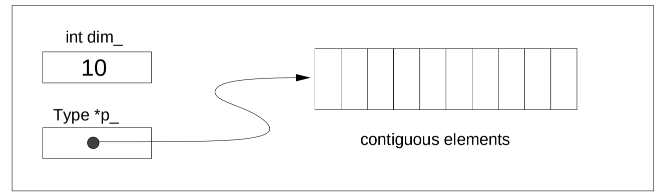
$scalar \leftarrow Vector::\textbf{norm()}$

$Vector \leftarrow Preconditioner::\textbf{solve}(Vector)$

$Vector \leftarrow Preconditioner::\textbf{trans\_solve}(Vector)$

$Vector \leftarrow \textbf{operator=}(Vector)$

Templated C++ Vector Class



class Vector<Type>

```
template <class Type>
class Vector
{
  private:
    int dim_;                           // size of vector
    Type *p_;                           // memory where data
                                        // is kept.
  public:

    Vector();                               // constructors
    Vector(unsigned int, Type t=0.0);
    Vector(unsigned int, const Type*);
    Vector(const Vector &);
    ~Vector();                              // destructor


                                            // access functions
    Type&     operator[](int i){return p_[i];}
    int         size() const { return dim_;}
    int         null() const {return dim_== 0;}

    Vector& operator=(const Vector&);   // assignment
    Vector& operator=(Type);
};
```

Templated C++ Vector Class

Just like `Vector` declaration before, except that rather than holding just `doubles`, it can hold any declared type:

```
Vector<double> A(10);          // a vector of doubles

Vector<int>    B(5);           // a vector of ints

Vector<Book>   L(1000);        // a vector Books

B[3] = 178;                    // looks and acts like any other Vector

L[64].set_title("The Firm");

A[0] = sin(A[1] / 3.14159);
```


Everything you didn't want to know about errors...

Structured programming actually *impedes* the management of errors

Three common ways to handle them:

1. existentialist: assume no errors (i.e. do nothing).
2. bureaucratic: encode an error in return value, let someone else worry about it.

   ```
   f(x, y, z, N, k);
   ```

   becomes

   ```
   errno = f(x, y, z, N, k);
   if (errno ==1) call MyErrorHandler1();
   else
   if (errno >= 2 && errno < 10) call MyErrorHandler2();
   ...
   ```

   - clutters up application code.
   - returned error codes often cryptic.
   - programmers often ignore them. (How many times are `malloc()` and `fopen()` used without checking return value?)
   - do not "scale" well, particularly in multi-level software components. Have to be handled immediately, or become lost.

3. fascist: shut program down (i.e. `exit()`)

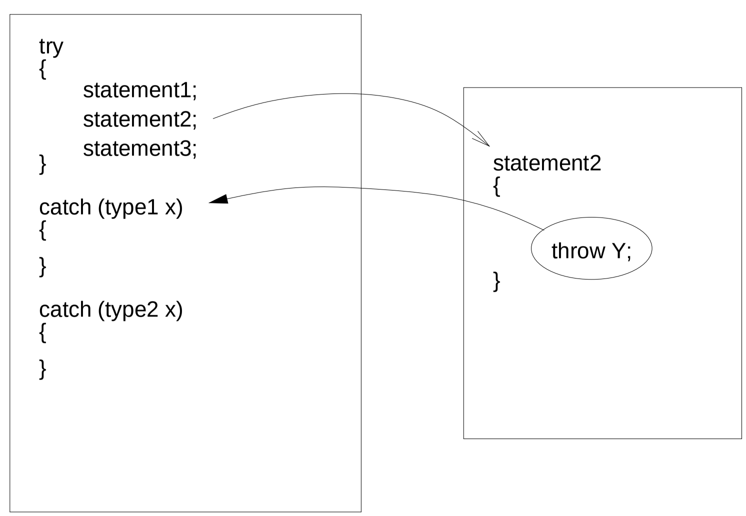
   - OK, *only* in `main()` level of application
   - not great for interactive applications (e.g. X apps), control applications (e.g. robotics), compilers, OS, etc.

C++ Exception Handling

ANSI C++ provides a better mechanism for handling errors. It introduces three new keywords:

`try`, `catch` and `throw`.

- **throw** is used to signal an error.
- **catch** is used to process errors.
- **try** is used to group the executable statements in your code that are treated by a `catch` statment.



Examples of Exception Handling

```
#include <iostream.h>

int main()
{
    try
    {
        throw 17;
        cout << "This statement will never execute.\n";
        cout << "Nor will this one.\n";
    }
    catch (int i)
    {
        cout << "Caught error #" << i << "\n";
    }

    return 0;
}
```

Produces the output:

```
Caught error #17
```

- Exceptions need not be processed immediately. They are caught by the `catch` clause which may occur several levels above.
- C++ exceptions are like `setjump()/longjump()` in C, except that they properly handle class destructors.
- supported by standard functions, e.g. `new` throws a `bad_alloc` exception (ANSI C++) that can be later tested. (Otherwise, returns 0, or `NULL`.)

Wrap-Up

Overview, general comments, and observations after several years of C++ hacking...

Overview of C++: What we've learned

- Typesafe C
- C Enhancements
  - const, inline, references, function overloading
- Memory Management
  - new, delete; constructors/destructors
- Data Abstraction
  - classes, exceptions, operator overloading, templates
- OO Programming
  - inheritance, virtual functions

Issues in C++ Library Design

- what are the basic objects?
- how should they interact?
- how general an interface?
- concrete data types, or abstract base classes?

Two ways to make code generic

- inheritance
  - describe operations in terms of abstract base class(es)
  - foundation of classic OO design
    * describe algorithms in terms of base class
    * create similar (derived) classes
    * apply existing algorithm to new class
  - use virtual function calls
  - some run-time overhead (3x regular function call)
- templates
  - describe skeletal code segments where "types" are arguments
  - no runtime costs
  - support varies among compilers
    * no nested templates
    * different linking semantics
    * some compilers expect separate declaration and implementations
  - static arguments must match exactly to trigger

What has worked in Scientific C++

- concrete data types
- reusing legacy Fortran kernels
- breaking application into separate computational levels
- expressing mathematical transformations at higher levels
- reference semantics to reduce copying of large data structures (e.g. const &)

Common Pitfalls

- over-generalizing
- deep, specialized, non-resuable hierarchies
- inconsistent assignment semantics
  - copy constructor must match operator=
- trying to derive off concrete classes
- development driven by “features” rather than good design

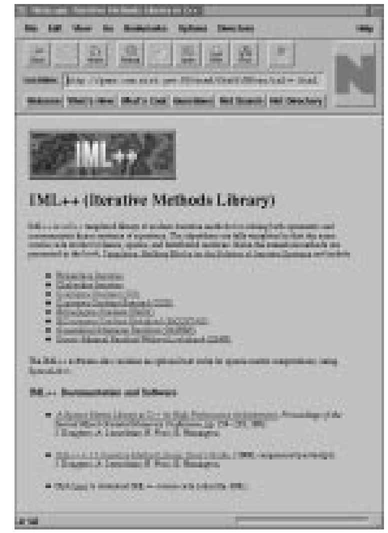
C++ Class Libraries

- General
  - LEDA
  - NIH
  - Gnu G++ Libs
  - Standard Templates Library (STL)
  - Booch
  - Microsoft Foundation Classes (MFC)

- Scientific
  - adaptive grid refinement (A++/P++)
  - linear algebra (LAPACK++)
  - sparse matrices (SparseLib++)
  - iterative methods (IML++)
  - finite element PDEs (Diffpack)
  - general math (Math.h++)
  - arrays, matrices (M++)

Some Available Software

- SparseLib++
- MV++
- IML++
- Lapack++



`http://math.nist.gov/acmd/Staff/RPozo/`

References

- Introductory
  - *The Complete C++ Reference*, 2nd ed., H. Schildt, 1995.
  - *C++ Primer*, S. Lippman, 1992.
  - *Effective C++*, S. Meyers, 1993.
  - *Scientific and Engineering C++*, J. Barton, L. Nackman, 1994
  - *A Book on C++*, I. Pohl, 1994.
- Advanced
  - *Annotated C++ Reference Manual*, M. A. Ellis, B. Stroustrup, 1990.
  - *ANSI C++ Draft Standard*, ANSI/ISO, 1995.
  - *C++ Report*
  - *Journal of Object Oriented Programming*