CMPS109  - C++

- C++ operator overloading
- C++ generics
- C++ STL containers
- C++ iterators and Algorithms

Problem 3 – use of templates

C++ Simple Class

class point {
public:
   point() : x(0), y(0) { }  // default
   point(double u) : x(u), y(0) { } // double to point
   point(double u, double v) : x(u), y(v) { }

private:
   double x, y;};

C++ v  Java

- Access keyword by section
- Conversion constructors
- Initializations : x(0)
- A type but not automatically a ref type
- Default is not a virtual method

Point plus

- void point::plus(point c) // call by value
- {
   x += c.x;
   y += c.y;
- }
- // point& c  -- call by reference

Member function operator+

- Inline point point::operator+(point c) {
   point temp; // calls default constructor
   temp.x = x + c.x; // this->x + c.x
   temp.y = y + c.y;
   return temp;
- }

Op overloading

- For a member function first argument is this
- point(): x(0), y(0) {} invoked – use stack
- operator+ is method name
- Pass by value is return mechanism
- p1 + p2 // p1 is this argument
- p1.operator+(p2) can be used

Use of a friend

- point operator+(point a, point b) {
   return point(a.x + b.x, a.y + b.y);
- }
- In class point need declaration
- friend point operator+(point a, point b);
- Need access to private
- Better because of symmetry

Op overloading tips

- Use only for a math type
- Be complete +, *, < != …
- Worry about conversions

Templates

- Template is the generic mechanism of choice (not void*)
- Combines type safety with efficiency
- Big win in reuse
- Downside: Syntax and runtime – errors are harder to decipher

Polymorphism Categories

- Ad hoc polymorphism - coercion
- Ad hoc polymorphism - overloading
- A function is called based on its signature
- Pure polymorphism - inclusion
- A type is a subtype of another type
- Pure polymorphism - parametric
- The type is left unspecified and is later instantiated

Developing Code with Templates

- Develop prototype “usual” case as an ordinary class
- Templatize
- Debug

Templates: stack.cpp

template <class TYPE>
class stack {
public:
   stack() : max_len(1000) { s = new TYPE[1000]; top = EMPTY; }
   void push(TYPE c) { s[++top] = c; }
   TYPE pop() { return (s[top——]); }
   TYPE top_of() { return (s[top]); }
   . . .

Declarations of Template stack

- stack<char> st_ch; // 1000 char
- stack<char*> st_str(200); // 200 char*

- This saves rewriting class declarations when the only variation would be type declarations
- It is an alternate scheme to using void* as a universal pointer type
- The code must always use the angle brackets as part of the declaration

Function Templates and Classes

- Functions independent of classes can be parameterized
- template <class Type>
- void copy(Type a[], Type b[], int n)
- template <class T>
- void max_min(vect<T>, T&, T&)

- The keyword class is used, but is unrelated to a particular class

A Template copy Function

template<class TYPE>
void copy(TYPE a[], TYPE b[], int n) {
   for (int i = 0; i < n; ++i) { a[i] = b[i]; }
}

- The invocation of copy() with specific arguments causes the compiler to generate the actual function

Using the Template copy Function

double f1[50], f2[50];
char c1[25], c2[50];
copy(f1, f2, 50); // instantiate with double
copy(c1, c2, 10); // instantiate with char