Templates and Polymorphism

Generic functions and classes
Polymorphic Functions

- Generic function that can act upon objects of different types
  - The action taken depends upon the types of the objects
- Overloading is a primitive form of polymorphism
  - Define functions or operators with the same name
    - Rational addition operator +
    - Function Min() for the various numeric types
- Templates
  - Generate a function or class at compile time
- True polymorphism
  - Choice of which function to execute is made during run time
    - C++ uses *virtual* functions
Function Templates

- **Current scenario**
  - We rewrite functions Min(), Max(), and InsertionSort() for many different types
  - There has to be a better way

- **Function template**
  - Describes a function format that when instantiated with particulars generates a function definition
    - Write once, use multiple times
Function Templates

```cpp
template <class T>
    T Min(const T &a, const T &b) {
        if (a < b)
            return a;
        else
            return b;
    }
```
Min Template

- Code segment
  ```cpp
  int Input1;
  int Input2;
  cin >> Input1 >> Input2;
  cout << Min(Input1, Input2) << endl;
  ```
- Causes the following function to be generated from the template
  ```cpp
  int Min(const int &a, const int &b) {
    if (a < b) {
      return a;
    } else {
      return b;
    }
  }
  ```
Min Template

- Code segment
  ```c
  float x = 19.4;
  float y = 12.7;
  ```
- Causes the following function to be generated from the template
  ```c
  float min(const float &a, const float &b) {
    if (a < b) {
      return a;
    } else {
      return b;
    }
  }
  ```
Function templates

- Location in program files
  - In current compiler template definitions are part of header files
- Possible template instantiation failure scenario
  ```
  cout << min(7, 3.14); // different parameter types
  ```
Generic Sorting

```cpp
template <class T>
void InsertionSort(T A[], int n) {
    for (int i = 1; i < n; ++i) {
        if (A[i] < A[i-1]) {
            T val = A[i];
            int j = i;
            do {
                A[j] = A[j-1];
                --j;
            } while ((j > 0) && (val < A[j-1]));
            A[j] = val;
        }
    }
}
```
Template Functions And STL

- STL provides template definitions for many programming tasks
  - Use them! Do not reinvent the wheel!
- Searching and sorting
  - find(), find_if(), count(), count_if(), min(), max(), binary_search(), lower_bound(), upper_bound(), sort()
- Comparing
  - equal()
- Rearranging and copying
  - unique(), replace(), copy(), remove(), reverse(), random_shuffle(), merge()
- Iterating
  - for_each()
Class Templates

- Rules
  - Type template parameters
  - Value template parameters
    - Place holder for a value
    - Described using a known type and an identifier name
  - Template parameters must be used in class definition described by template
  - Implementation of member functions in header file
    - Compilers require it for now
A Generic Array Representation

- We will develop a class Array
  - Template version of IntList
  - Provides additional insight into container classes of STL
Homegrown Generic Arrays

Array<int> A(5, 0);       // A is five 0's
const Array<int> B(6, 1); // B is six 1's
Array<Rational> C;        // C is ten 0/1's
A = B;
A[B[1]] = 2;

cout << "A = " << A << endl;   // [ 1 2 1 1 1 3 ]
cout << "B = " << B << endl;   // [ 1 1 1 1 1 1 ]
cout << "C = " << D << endl;
   // [ 0/1 0/1 0/1 0/1 0/1 0/1 0/1 0/1 0/1 0/1 ]
template <class T>
class Array {

    public:

    Array(int n = 10, const T &val = T());
    Array(const T A[], int n);
    Array(const Array<T> &A);
    ~Array();
    int size() const {
        return NumberValues;
    }
    Array<T> & operator=(const Array<T> &A);
    const T & operator[](int i) const;
    T & operator[](int i);

    private:

    int NumberValues;
    T *Values;

};
Auxiliary Operators

```cpp
template <class T>
ostream& operator<<(ostream &sout, const Array<T> &A);

template <class T>
istream& operator>>(istream &sin, Array<T> &A);
```
Default Constructor

template <class T>
Array<T>::Array(int n, const T &val) {
    assert(n > 0);
    NumberValues = n;
    Values = new T [n];
    assert(Values);
    for (int i = 0; i < n’ ++ i) {
        Values[i] = A[i];
    }
}
template <class T>
Array<T>::Array(const Array<T> &A) {
    NumberValues = A.size();
    Values = new T [A.size()];
    assert(Values);
    for (int i = 0; i < A.size(); ++i) {
        Values[i] = A[i];
    }
}
Destructor

template <class T>
Array<T>::~Array() {
    delete [] Values;
}

template <class T>
Array<T> & Array<T>::operator=(const Array<T> & A) {
    if (this != &A) {
        if (size() != A.size()) {
            delete [] Values;
            NumberValues = A.size();
            Values = new T [A.size()];
            assert(Values);
        }
        for (int i = 0; i < A.size(); ++i) {
            Values[i] = A[i];
        }
    }
    return *this;
}
Inspector for Constant Arrays

template <class T>
const T& Array<T>::operator[](int i) const {
    assert((i >= 0) && (i < size()));
    return Values[i];
}
Nonconstant Inspector/Mutator

template <class T>
    T& Array<T>::operator[](int i) {
        assert((i >= 0) && (i < size()));
        return Values[i];
    }
Generic Array Insertion Operator

template <class T>
    ostream& operator<<(ostream &sout,
                     const Array<T> &A) {
    sout << "[ ";
    for (int i = 0; i < A.size(); ++i) {
        sout << A[i] << " ";
    }
    sout << "]";
    return sout;
}

● Can be instantiated for whatever type of Array we need
Specific Array Insertion Operator

- Suppose we want a different Array insertion operator for Array<char> objects

```cpp
ostream& operator<<(ostream &sout, 
    const Array<char> &A){
    for (int i = 0; i < A.size(); ++i) {
        sout << A[i] << " ";
    }
    return sout;
}
```
Scenario

- Suppose you want to manipulate a list of heterogeneous objects with a common base class
  - Example: a list of EzWindows graphical shapes to be drawn
    ```
    // what we would like
    for (int i = 0; i < n; ++i) {
        A[i].Draw();
    }
    ```
  - Need
    - Draw() to be a virtual function
      - Placeholder in the Shape class with specialized definitions in the derived class
    - In C++ we can come close
Virtual Functions

TriangleShape T(W, P, Red, 1);
RectangleShape R(W, P, Yellow, 3, 2);
CircleShape C(W, P, Yellow, 4);


for (int i = 0; i < 3; ++i) {
    A[i]->Draw();
}

- For virtual functions
  - It is the type of object to which the pointer refers that determines which function is invoked
Shape Class with Virtual Draw

class Shape : public WindowObject {
    public:
        Shape(SimpleWindow &w, const Position &p,
            const color c = Red);
        color GetColor() const;
        void SetColor(const color c);
        virtual void Draw();  // virtual function!
    private:
        color Color;
};
Virtual Functions

- If a virtual function is invoked via either a dereferenced pointer or a reference object
  - Actual function to be run is determined from the type of object that is stored at the memory location being accessed rather than the type of the pointer or reference object
  - The definition of the derived function overrides the definition of the base class version
- Determination of which virtual function to use cannot be made at compile time and must instead be made during run time
  - More overhead is associated with the invocation of a virtual function than with a nonvirtual function
Pure Virtual Function

- A virtual member function is a pure virtual function if it has no implementation.
- A pure virtual function is defined by assigning that function the null address within its class definition.
- A class with a pure virtual function is an abstract base class.
  - Convenient for defining interfaces.
  - Base class cannot be directly instantiated.
class Shape : public WindowObject {
    public:
        Shape(SimpleWindow &w, const Position &p, const color &c = Red);
        color GetColor() const;
        void SetColor(const color &c);
        virtual void Draw() = 0;
        // pure virtual function!

    private:
        color Color;
};