Object-Oriented Programming (OOP)
Lecture No. 39

Templates & Static Members

- Each instantiation of a class template has its own copy of static members
- These are usually initialized at file scope

...Templates & Static Members

```
template< class T >
class A {
public:
  static int data;
  static void doSomething( T & );
};
```

...Templates & Static Members

```
int main() {
  A< int > ia;
  A< char > ca;
  ia.data = 5;
  ca.data = 7;
  cout << "ia.data = " << ia.data
       << endl
       << "ca.data = " << ca.data;
  return 0;
}
```

…Templates & Static Members

- Output

```
ia.data = 5
ca.data = 7
```

Templates – Conclusion

- Templates provide
  - Reusability
  - Writability
- But can consume memory if used without care

…Templates – Conclusion

- Templates affect reliability of a program

```
template< typename T >
bool isEqual( T x, T y ) {
   return ( x == y );
}
```

…Templates – Conclusion

- One may use it erroneously

```
int main() {
   char* str1 = "Hello ";
   char* str2 = "World!";
   isEqual( str1, str2 );
        // Compiler accepts!
}
```

Generic Algorithms Revisited

```
template< typename P, typename T >
P find( P start, P beyond,
                        const T& x ) {
  while ( start != beyond &&
                         *start != x )
     ++start;

  return start;
}
```

...Generic Algorithms Revisited

```
int main() {
  int iArray[5];
  iArray[0] = 15;
  iArray[1] = 7;
  iArray[2] = 987;
  …
  int* found;
  found = find(iArray, iArray + 5, 7);
  return 0;
}
```

…Generic Algorithms Revisited

- We claimed that this algorithm is generic
- Because it works for any aggregate object (container) that defines following three operations
  - Increment operator (++)
  - Dereferencing operator (*)
  - Inequality operator (!=)

…Generic Algorithms Revisited

- Let us implement these operations in `vector` to examine the generality of the algorithm
- Besides these operations we need a kind of pointer to track the traversal

Example – Vector

```
template< class T >
class Vector {
private:
  T* ptr;
  int size;
  int index; // initialized with zero
public:
  …
  Vector( int = 10 );
```

…Example – Vector

```
  Vector( const Vector< T >& );
  T& operator [](int);

  int getIndex() const;
  void setIndex( int i );
  T& operator *();
  bool operator !=(
              const Vector< T >& v );
  Vector< T >& operator ++();
};
```

...Example – Vector

```
template< class T >
int Vector< T >::getIndex() const {
  return index;
}
template< class T >
void Vector< T >::setIndex( int i ) {
  if ( index >= 0 && index < size )
     index = i;
}
```

...Example – Vector

```
template< class T >
Vector<T>& Vector<T>::operator ++() {
  if ( index < size )
     ++index;
  return *this;
}
template< class T >
T& Vector< T >::operator *() {
  return ptr[index];
}
```

…Example – Vector

```
template< class T >
bool Vector<T>::operator !=(
                        Vector<T>& v ) {
  if ( size != v.size ||
                    index != v.index )
     return true;
```

…Example – Vector

```
  for ( int i = 0; i < size; i++ )
     if ( ptr[i] != v.ptr[i] )
          return true;

  return false;
}
```

…Example – Vector

```
int main() {
  Vector<int> iv( 3 );
  iv[0] = 10;
  iv[1] = 20;
  iv[2] = 30;
  Vector<int> beyond( iv ),found( 3 );
  beyond.setIndex( iv.getSize() );
  found = find( iv, beyond, 20 );
  cout<<"Index: "<<found.getIndex();
  return 0;
}
```

Generic Algorithm

```
template< typename P, typename T >
P find( P start, P beyond,
                        const T& x ) {
  while ( start != beyond &&
                         *start != x )
     ++start;

  return start;
}
```

Problems

- Our generic algorithm now works fine with container `Vector`
- However there are some problems with the iteration approach provided by class `Vector`

...Problems

- No support for multiple traversals
- Inconsistent behavior
  - We use pointers to mark a position in a data structure of some primitive type
  - Here we use the whole container as marker e.g. `found` in the `main` program
- Supports only a single traversal strategy