Lecture 19

- Prefix/postfix operators
- Static members
- Casts
- Template functions

Prefix/Postfix Mysteries

```
class Complex {
public:
   ...
   Complex &operator++();      // prefix ++
   Complex operator++(int);    // postfix ++
};

// postfix: can't return a reference to the variable
// because the returned value is different
Complex Complex::operator++(int) {
   Complex temp = *this;
   re++;
   return temp;  // returns the previous contents
}

// with prefix++ returning a reference is OK
// variable is changed and reference is returned
Complex &Complex::operator++() {
   re++;
   return *this; // prefix operators are more efficient!
}
```


"Complex" Application

```
#include "Complex.H"

int main() {

   Complex a(1.0, 0.0);
   Complex b(0.0, 1.0);
   Complex c;

   c = (a + b) * (a - b);
   c += Complex(4,3);
   c = c + 3.0;
   c++;
   ...
};
```


Static Members

- Sometimes it is useful if all objects of a class have access to the same variable
  - e.g. a "global" class option or a counter that keeps track of how many objects have been created
- Can also save space
  - e.g. shared pointer to error-handling routine
- Advantages:
  - **Information hiding** can be enforced. Static members can be private - global variables cannot
  - Static members are not entered in global namespace, **limiting accidental name conflicts**
- Syntax: **static** qualifier before var/func decl.

Example

```
class X {
public:
   X() { ++count; ... }

   // static function
   static int get_count() { return count; }

private:
   static int count; // #of X objects,
                     // shared by all X vars.
};

int X::count = 0; // must be defined in .C file

int main() {
   X x;
   cout << X::get_count() << endl; // 1
   return 0;
}
```


Casts

- **static_cast**
  - used for standard conversions
  - compile-time operator - no run-time check
  - int i; double d; i = static_cast<int>(d);
  - **Do not use for down-casts – see dynamic_cast**
- **reinterpret_cast**
  - conversion from one pointer type to **any** other pointer type
  - Can also convert pointers to ints and vice versa
  - int a[1000]; char *p = reinterpret_cast<char*>(a);
  - **Dangerous! Can result in unportable code**

const_cast

```
class X {
public:
   int get_index() const;

private:
#ifndef NDEBUG
   int getn; // counter
#endif
};
```

```
int X::get_index() const {
#ifndef NDEBUG
   // increment counter,
   // preserve constness
   const_cast<X*>(this)->getn++;
#endif

   // get_index body
   ...
}
```

- const_cast changes status: **const** <-> **non const**
- used for changing "unessential" private data members in const functions (can also be accomplished by "mutable")

Alternative Implementation

```
class X {
public:
   int get_index() const;

private:
#ifndef NDEBUG
   mutable int getn; // mutable counter
   // "mutable" indicates that this variable
   // can be changed in const functions
#endif
};
```

```
int X::get_index() const {
#ifndef NDEBUG
   // increment counter
   ++getn;
#endif
   ...
}
```

dynamic_cast

- useful for down-casting pointers
  (trying to treat them as derived class pointer)
- Only works for polymorphic types (with VFTP)
- run-time check (slows down program)
- returns 0 if cast is illegal, and parameter otherwise

```
class X { virtual void foo(); };  // VFTP added

class Y : public X { };

void bar(X *p) {
   Y *q = dynamic_cast<Y*>(p);
   assert(q != 0);  // q == 0 iff *p is not
                    // of type Y nor derived from Y
}
```


Generic Programming (new)

- Code is often independent of actual types
  - Sorting routines (qsort)
  - Containers (vectors, lists, sets)
- Generic programming:
  use identical code for arbitrary types
- Benefit: code is easy to maintain!
- C way:
  use void* as generic pointer type and
  pass function pointers
- C++ way:
  template functions and class templates

Template Functions

```
int min(int a, int b) { return a < b ? a : b; }

float min(float a, float b) { return a < b ? a : b; }

...

No need for listing defining long list of identical
functions! The following generic definition covers all:

template <typename T> T min(T a, T b) {
   return a < b ? a : b;
}

Function min is now paramerized by type T
```

- Compiler generates implementations for
  actual type instances when function is used

Example

```
template <typename T> T max(T a, T b){
   return a > b ? a : b;
}

template <typename T> void swap(T &a, T &b){
   T temp = a; a = b; b = temp;
}

int main() {

   int a=10, b=5, c = min(a,b); // min<int,int> called

   float e=2.0, f=1.0, g = min(e,f); // min<float,float>

   swap(a,b); // swap<int,int>
   swap(e,f); // swap<float,float>
}
```