Friend Functions

- A *friend function* of a class is defined outside that class's scope, yet has the right to access *private* members of the class
- Even though the prototypes for friend functions appear in the class definition, friend functions are still *not* member functions

1

An Example

```
class Count {
   friend void setX( Count &, int );    // friend functions first
public:
   Count( ) { x = 0; }
   void print const ( ) { cout << x << endl; }
private:
   int  x;
};

void setX( Count &c, int val ) {
   c.x = val;
}
```

2

An Example

```
int main( )
{
    Count counter;

    cout << "counter.x after instantiation: ";
    counter.print( );
    cout << "counter.x after call to setX friend function: ";
    setX( counter, 8 );
    counter.print( );
    return 0;
}
```

3

Friend Classes

- An entire class may be declared to be a friend of another class
- *All* member functions of a *friend class* are *friend functions*
- Friendship is *granted*, not taken
- Friendship is *neither symmetric nor transitive*

4

An Example

```
class Oracle;

class Count {
   friend class Oracle;    // friend classess first
public:
   Count( ) { x = 0; }
   void print const ( ) { cout << x << endl; }
private:
   int  x;
};
```

5

An Example

```
class Oracle {
public:
   Oracle( ) { y = 0; }
   void setX( Count &, int );  // frined function of Count
   void inc( Count & );        // frined function of Count
private:
   int y;
};
void Oracle::setX( Count &c, int val ) {
   c.x = val;
}
void Oracle::inc( Count &c ) {
   y += c.x;
}
```

6

Operator Overloading

- *Operators* provide a *concise notation* for expressing manipulations of objects of *built-in types*
- *Operator overloading* allows us to use *existing* operators to specify object manipulations of *user-defined types*
- We *cannot* use operator overloading to create *new* operators

7

An Example

```
class String {
public:
   String( const char * = "" );
   const String &operator=( const String & );
   const String &operator+=( const String & );
   ~String( );
private:
   int length;
   char *sPtr;
   void setString( const char * );
};
```

8

An Example

```
String::String( const char * str )
{
   length = strlen( str );
   setString( str );
}

void String::setString( const char *str )
{
   newStr = new char[ length + 1 ];
   strcpy( newStr, str );
}

String::~String( ) { delete [ ] sPtr; }
```

9

An Example

```
const String &operator=( const String &right )
{
   if ( &right != this ) {
      delete [ ] sPtr;
      length = right.length;
      setString( right.sPtr );
   } else {
      cout << "Attempted self assignment\n";
   }
   return *this;
}
```

10

An Example

```
const String &operator+=( const String &right )
{
   char *tempPtr = sPtr;
   length += right.length;
   sPtr = new char[ length + 1 ];
   strcpy( sPtr, tempPtr );
   strcat( sPtr, right.sPtr );
   delete [ ] tempPtr;
   return *this;
}
```

11

An Example

```
int main( )
{
   String s1( "happy" ), s2( "birthday" ), s3;

   s3 = s1;          // s3 = "happy"
   s3 += s2;         // s3 = "happybirthday"
   return 0;
}
```

12

Operator Functions

- Operator overloading functions can be made *member functions* or *friend functions*
- When an operator function is a *member* function, the leftmost operand must be a class object of the operator's class
- A *non-member* friend operator function enables the operator to be commutative

13

An Example

```
class PhoneNo {
   friend ostream &operator<<(ostream &, const PhoneNo &);
   friend istream &operator>>(istream &, PhoneNo &);
public:
   PhoneNo( );
private:
   char areaCode[ 3 ];
   char lineCode[ 8 ];
};
```

14

An Example

```
ostream &operator<<(ostream &output, const PhoneNo &pn)
{
   output << pn.areaCode << " " << pn.lineCode;
   return output;
}

istream &operator>>(istream &input, PhoneNo &pn)
{
   input >> pn.areaCode >> pn.lineCode;
   return input;
}
```

15

An Example

```
int main( )
{
    PhoneNo pn;

    cout << "Enter phone no in the form xx xxxxxxx:\n";
    cin >> pn;
    cout << "The phone no entered was:\n";
    cout << pn << endl;
    return 0;
}
```

16

Overloading Unary Operators

- A *unary* operator for a class can be overloaded as a member function with no argument or as a non-member friend function with one argument
- It is preferable to make a unary operator function a *member function* since the use of friends violates the *encapsulation* of a class

17

An Example

```
class String {
public:
   bool operator!( ) const;
   …
};

bool operator!( ) const
{
   return length == 0;
}
```

18

An Example

```
int main( )
{
    String s1( "happy" ), s3;

    if ( !s3 ) {   // s3 is empty
       s3 = s1;
    }
    return 0;
}
```

19

Overloading ++ and --

- The preincrement and postincrement must have *distinct prototypes*
- The *preincrement* is overloaded as any other prefix unary operators
- The *postincrement* is overloaded as any other prefix unary operators with one extra "*dummy*" argument

20

An Example

```
class Time {
public:
   Time &operator++( );
   Time operator++( int );
   …
};
```

```
Time &Time::operator++( )
{
   helpIncrement( );
   return *this;
}

Time Time::operator++( int )
{
   Time temp = *this;
   helpIncrement( );
   return temp;
}
```

21

An Example

```
void Time:helpIncrement( )
{
   if ( second == 59 ) {
      second = 0;
      if ( minute == 59 ) {
         minute = 0;
         if ( hour == 23 )  hour = 0;
         else  hour++;
      } else  minute++;
   } else  second++;
}
```

22

Conversion Operators

- A *conversion operator* can be used to convert an object of one class into an object of another class or into an object of a built-in type
- The function prototype
  A::operator B( ) const;
  declares an *overloaded* conversion operator function from type A to type B
- Conversion operators *cannot* be friends

23

An Example

```
String::operator char *( ) const
{
   char *ptr = new char[ length + 1 ];
   strcpy( ptr, sPtr );
   return ptr;
}

char *s = "Happy Birthday";
cout << (char *) s << endl;          // cast
cout << s << endl;                   // coercion
```

24

Restrictions on Operators

- Operators that *can* be overloaded
  + - * / % ^ & | ~ ! = < > += -= *= /= %= ^= &= |= << >> >>= <<= == != <= >= && || ++ -- ->* , -> [ ] ( ) new delete new[ ] delete[ ]
- Operators that *cannot* be overloaded
  . .* :: ?: sizeof

25

Case Study: An Array Class

- Range checking
- Array assignment
- Array objects with size information
- Array input and output
- Array comparison

26

An Example

```
class Array {
   friend ostream &operator<<(ostream &, const Array &);
   friend istream &operator>>(istream &, Array &);
public:
   Array( int = 10 );   Array( const Array & );  ~Array( );
   int getSize( ) const;
   const Array &operator=( const Array & );
   bool operator==( const Array & ) const;
   bool operator!=( const Array & ) const;
   int &operator[ ]( int );
   const int &operator[ ]( int ) const;
private:
   int size, *ptr;
}
```

27

An Example

```
Array::Array( int arraySize ) {
   size = ( arraySize > 0 ? arraySize : 10 );
   ptr = new int[ size ];
   assert( ptr != 0 );
   for ( int i = 0; i < size; i++ ) ptr[ i ] = 0;
}
Array::Array( const Array &init ) : size( init.size ) {
   ptr = new int[ size ];
   assert( ptr != 0 );
   for ( int i = 0; i < size; i++ ) ptr[ i ] = 0;
}
Array::~Array( ) {
   delete [ ] ptr;
}
```

28

An Example

```
const Array &Array::operator=( const Array &right ) {
   if ( &right != this ) {
      if ( size != right.size ) {
         delete [ ] ptr;
         size = right.size;
         ptr = new int[ size ];
      }
      for ( int i = 0; i < size; i++ )
         ptr[ i ] = right.ptr[ i ];
   return *this;
}
```

29

An Example

```
bool Array::operator==( const Array &right ) const {
   if ( size != right.size ) return false;
   for ( int i = 0; i < size; i++ )
      if ( ptr[ i ] != right.ptr[ i ] ) return false;
   return true;
}
int &Array::operator[ ]( int subscript ) {
   assert( 0 <= subscript && subscript < size );
   return ptr[ subscript ];
}
const int &Array::operator[ ]( int subscript ) const {
   assert( 0 <= subscript && subscript < size );
   return ptr[ subscript ];
}
```

30

An Example

```
istream &operator>>(istream &input, Array &a) {
   for ( int i = 0; i < a.size; i++ ) input >> a.ptr[ i ];
   return input;
}
ostream &operator<<(ostream &output, const Array &a) {
   int i;
   for ( int i = 0; i < a.size; i++ ) {
      input >> a.ptr[ i ];
      if ( (i + 1) % 4 == 0 ) output << endl;
   }
   if ( i % 4 != 0 ) output << endl;
   return output;
}
```

31

An Example

```
int main( ) {
   Array intergers1( 7 ), integers2;
   cout << integers1.getSize( ) << integers1 << endl;
   cout << integers2.getSize( ) << integers2 << endl;
   cin >> integers1 >> integers2;
   cout << integers1 << integers2 << endl;
   if (integers1 != integers2) cout << "not equal\n";
   Array integers3( integers1 );
   cout << integers3.getSize( ) << integers3 << endl;
```

32

An Example

```
   integers1 = integers2;
   if (integers1 == integers2) cout << "equal\n";
   cout << integers1[ 5 ] << endl;
   integer1[ 5 ] = 1000;
   cout << integers1 << endl;
   integers1[ 15 ] = 1000;
   return 0;
}
```

33