Intro

- Different actions for most of known operators can be defined.
- An ADT can have its "own" meaning of + for example. This "new meaning" is defined in a function that overloads the "regular" + operator.
- You've encountered it implicitly: "int i + float f" or "cout<<"
- Always overload operators with a similar meaning; don't overload "+" with "-" !!!

Operator functions: friends VS members

- Overloading an operator is similar to defining a function, with the use of the operator keyword.
- An operator can be overloaded as a member function, or outside the class it servers – as a friend function.

Overloading the unary [] operator as a member function:

```
#include <iostream.h>

class MyArray {
public:
   MyArray( int size=5 )     { ptr = new int[size]; }  //constructor
   ~MyArray()                { delete ptr; }           //destructor
   int &operator[]( int i )  {
      cout<<"extracting element number "<<i+1<<endl;
      return ptr[i];
   }
   void setElement( int i, int el )  { ptr[i]=el; }
private:
   int *ptr;
};

int main() {
   MyArray array(3);
   array.setElement( 0, 127 );
   cout<<"first element is set to "<<array[0]<<endl;
   return 0;
}
```

output:

```
extracting element number 1
first element is set to 127
```

explenation: The compiler interprets the call `array[0]` as `array.operator[]( 0 )` whose prototype is `int &operator[]( int )`. This is actually a function call!

- the return type of the overloaded [] is a reference to the requested array element.
- The overloading function accesses the private member ptr in order to extract the element.

A complete example in figure 8.4 from the "C++ how to program" book

The technical difference between operator overloading and a regular function call:
A **regular function call** involves using the *full name* of the function and the *parentheses* "()" to transfer parameters, while **operator overloading** uses merely the *operator's symbol* ("+", not the function full name "operator+") and *no parentheses* is required,

Overloading the binary == and != operators as a member function:

```
#include <iostream.h>
#include <string.h>

class MyString {
public:
   MyString( const char* );          //constructor
   ~MyString() { delete str; }       //destructor
   bool operator==( const MyString &right ) const {
      return strcmp( this->str, right.str )==0;
   }
   bool operator!=( const MyString &right ) const { return !(*this==right); }
private:   char* str;
};

MyString::MyString( const char* s ) {
   str = new char[strlen(s)+1];
   strcpy( str, s );
}

int main() {
   MyString s1( "one" );     MyString s2( "one" );       MyString s3( "two" );

   if( s1==s2 )  cout<<"s1 and s2 are equal."<<endl;
   else          cout<<"s1 and s2 are not equal."<<endl;

   if( s1!=s3 )  cout<<"s1 and s3 are not equal."<<endl;
   else          cout<<"s1 and s3 are equal."<<endl;

   return 0;
}
```

output:

```
s1 and s2 are equal.
s1 and s3 are not equal.
```

- Note that the == overloading is a class member, and has a ”this” pointer.
- The == indeed returns a Boolean answer.
- The == operator has 2 operands: the left operand is the ”this” object, and the right operand is the argument of the function.
- The != operator is overloaded using the NEW == definition.
- Operator overloading don’t have to be inlined. They behave like regular functions.

The “this” handle refers to the object that call the function “operator==” (for example), the object that initiate the function call.

Member-function-implementation requires that the ”this” object will be the left operand. Therefore, a stream operator cannot be overloaded as member function:
cout<<s1 has the object as right operand!!!
Of course it can be implemented as s1>>cout, but this is not intuitive!!!

The solution: use friend functions

Overloading the stream insertion & stream extraction operators, as friend function:

```
#include <iostream.h>
#include <string.h>

class MyString {

   friend ostream& operator<<( ostream &, const MyString & );
   friend istream& operator>>( istream &, const MyString & );

public:
   MyString() { str = new char[100]; }  //constructor
   ~MyString() { delete str; }          //destructor

private:
   char* str;
};

ostream& operator<<( ostream &output, const MyString &s ) {
   return output<<s.str;
}

istream& operator>>( istream &input, const MyString &s ) {
   char temp[100];
   input>>temp;
   strcpy( s.str, temp );
   return input;
}

int main() {
   MyString s1;
   cin>>s1;
   cout<<"you entered "<<s1<<endl;
   return 0;
}
```

output:

```
hello
you entered hello
```

- non-member function have to take 2 parameters for 2-operands: left operand and right operand. In this case cin/cout are the left operands, and the MyString object is the right operand.
- cin is an istream object, while cout is an ostream object. Both overloaded operators return these objects in order to allow cascading.
- As friend functions, both overloading functions have access to the private member str.
- A friend function does not, however, have a "this" pointer. Because they do not ***belong*** to the overloaded class object

A complete example in figure 8.5 from the "C++ how to program" book

Special case: overloading ++ (and --)

- The problem: distinguishing between ++obj (pre-increment) and obj++ (pos-tincrement).
- Another problem: simulating the obj++ call: incrementing while returning the pre-incremented object.

- The pre-increment overloaded function has a prototype of **<obj> &operator++()**
- The post-increment overloaded function has a prototype of **<obj> operator++(int)**; the int type is dummy. Also note that the function doesn't return the incremented object itself.

```
#include <iostream.h>

class three_d {
   int x, y, z; // 3-D coordinates
public:
   three_d(int i=0, int j=0, int k=0) {x = i; y = j; z = k; }
   three_d operator=(three_d op2);   // assignment operator
   three_d operator++();             // prefix version of ++
   three_d operator++(int notused);  // postfix version of ++
   void three_d::show( ) { cout << x << ", " << y << ", " << z << endl; }
};

three_d three_d::operator=(three_d op2) {
   x = op2.x; // These are integer assignments
   y = op2.y; // and the = retains its original
   z = op2.z; // meaning relative to them.
   return *this;
}

three_d three_d::operator++() {      // prefix version of ++
   x++;  // increment x, y, and z
   y++;
   z++;
   return *this; // return altered value
}

three_d three_d::operator++(int notused) {    // postfix version of ++
   three_d temp = *this; // save original value
   x++; // increment x, y, and z
   y++;
   z++;
   return temp; // return original value
}

int main() {

   three_d a(1, 2, 3), b;

   cout<<"a is ";
   a.show();
   cout<<"b receives a's value prior to increment:\n";
   b=a++;
   b.show();

   cout<<"\na is ";
   a.show();
   cout<<"b receives a's value after increment:\n";
   b=++a;
   b.show();
   return 0;
}
```

output:

```
a is 1, 2, 3
b receives a's value prior to increment:
1, 2, 3

a is 2, 3, 4
b receives a's value after increment:
3, 4, 5
```

Restrictions:

- Operators that **cannot** be overloaded: **.** **.*** **::** **?:** **sizeof**
- "new", "delete" and the "," operators are especially hard to overload. We won't discuss them.
- The precedence of operators cannot be changed; overloaded "*" will always evaluate before overloaded "+".
- The number of arguments that an operator takes cannot be changed: "+" will always have two operands.
- Each operator have to be overloaded explicitly: Overloading "+" does not mean an automatic overloading of the different operator "+=".