6 OPERATOR OVERLOADING

- Most operators can be overloaded for user defined classes
  - can use standard operators to write expressions
  - cannot change precedence nor associativity
  - cannot change meaning for intristic types
  - the leftmost argument in an expression is implicit in the overloaded operator
    - must declare only the remaining arguments, if any
    - expressions with different leftmost arguments must be handled with
      - non-member methods, `friend` if needed to access private data
  - side effects are up to programmer
    - use common sense to avoid confusion
- reversing arguments if applicable
- The following can be overloaded

```
+       -        *      /      %      ^      &      |
~       !        =      <      >      +=     -=     *=
/=      %=       ^=     &=     |=     <<     >>     >>=
<<=     ==       !=     <=     >=     &&     ||     ++
--      ,        ->*    ->     ()     []     new    delete
new[]   delete[]
```

  - cannot overload `::` `?:` `.` and `*.`

6.1 Unary operator

type operator*O*(void); // *O* represents the operator

- only the implicit argument

Example 6.1 Overload unary - for Stock so that `-stock` would mean sell half shares.

```
void Stock::operator-(void){
  this->sell(this->shares/2);
  this->set_tot();
}
// later in a function
     Stock ibm;
     -ibm;
```

6.2 Binary operator

type operator*O*(*argument*); // *O* represents an operator

- left argument of an expression is the implicit (`*this`)
- right argument of an expression corresponds to the interface argument

Example 6.2 Overload + for `Stock` to add number of shares creating a new `Stock`

```
Stock Stock::operator+(const Stock &second) const
// return *this + second
{  int x=this->shares+second.shares;
   Stock s(“Combined”,x);
   return s;
}

// later
   Stock ibm, att;
   Stock ss=ibm+att;
   ibm+att; // does it make sense? what about 3+5?
```

Example 6.3 Same as Example 6.2 but with potential memory leaks and misuse - why?.

```
Stock & Stock::operator+(const Stock &second) const{
   int x=this->shares+second.shares;
   Stock *s=new Stock(“Combined”,x);
   return *s;
}
// later
   Stock ibm, att, kmart, walmart;
   ibm+att=kmart+walmart;  // what is this, and how do we read it?
                          // what if Stock s(“Combined”,x) is used?
```

Example 6.4 Same as Example 6.2 but with potential memory leaks - why?. It can be used most efficiently on the other hand - why?

```
Stock *Stock::operator+(const Stock &second) const{
   int x=this->shares+second.shares;
   Stock *s=new Stock(“Combined”,x);
   return s;
}
// later
   Stock ibm, att, walmart, kmart, *sp;
   sp=ibm+att;
   sp=walmart+kmart;    // memory leak
```

Example 6.5 Overload + for `Stock` so that `ibm+10` would mean by 10 more shares.

```
void Stock::operator+(int toBuy){
   this->shares+=toBuy;
   this->set_tot();
}     // could implement with this->buy(toBuy,this->share_val);

// later
   Stock ibm;
   ibm+10;
```

Example 6.6 `adam+10` means add 10 years.

```
   void Person::operator+(int inc) {
      if (inc>0)
         age+=inc;
   }

   // later in an application
      Person adam(“Adam”,”Vice”,20);
      adam+10;             // adam is 30 now
      10+adam;              // ???bump
```

- Postfix `++`/`--` differentiated by having dummy (`int`) argument in postfix

Exercise 6.1 Stock with overloaded +.

Exercise 6.2 Extend Exercise 6.1 changing ‘+’ so that names are combined, shares added, price averaged. Overload ‘-’ with an integer to mean ‘sell up to that many’ (as many available). For example, ‘ibm-100’ would be sell 100 from the ibm shares object. Then, overload that overloaded ‘-’ to work with `double` argument, meaning change price to that value. Note that both ‘-’ operators will change `*this`.

6.3 Overloading with Non-member Methods

- ❍ Needed when
  - ❍ in binary operators, the left argument is not of the class of interest

```
adam+10;    // done by overloading + for Person
10+adam;    // would have to overload + for int ???
```

  - ❍ if desired to perform automatic argument conversions
- ❍ Implementation
  - ❍ must implement *non-member* operator of two arguments

```
void operator+(int x, Person &p);
```

    - ● cannot access private stuff
    - ● prototyped outside class in the header file
    - ● implemented along with class methods
    - ● as a global method, calls with different argument will have arguments converted
  - ❍ if private access needed, it can be accomplised by `friend`

```
friend void operator+(int, Person &);// inside class declaration
void operator+(int inc, Person &p){// Note no Person::
         if (inc>0)
            p.age+=inc;
}
```

  - ❍ private access can also sometime be implemented by reversing the arguments
    - ● works only if the reversed operator exists and does the same

```
void operator+(int, Person &);
   void operator+(int inc, Person &p) {
         if (inc>0)
            p+inc;
}
```

  - ❍ global function for class C should be declared and implemented with the class
  - ❍ you may not implement the following operators except as methods:
    subscript [], function call (), assignment =, indirection ->

Exercise 6.3 Design a Vector class, for a 2D space. Each vector is represented by cartesian or polar coordinates. Use operator overloading for operations.

6.4 More on friend

- Global methods, such as operators, can be friends
- Any method or any class (and thus all its methods) can be friends

Example 6.7 friends.

```
class A {
   // ...
   int f();
   // ...
};

class B {
   // ...
   friend int A::f();   // makes f method a friend to class B
   friend A;            // makes all methods of A friends of B
   // ...
};
```

- Avoid making too many friends...

6.5 Overloading IO operators

- What about writing

  ```
  cout << adam;
  cin >> baby;
  ```

    - must overload with non-member friend function
  - can be done for a single application
  - can be done for chaining
  - do not handle by reversing arguments
  - declare friend if needed to access private elements

Example 6.8 << overloaded for a single application on Person.

```
   friend void operator<<(ostream &, const Person &); // in class decl.
   void operator<<(ostream &os, const Person &p) {
      os << “My name is “ << p.name << endl;
   }
```

```
// in a function
   Person adam(“adam”), susan(“susan”);
   cout << adam;                             // ok
   cout << adam << “ and “ << susan << endl; // bump
```

Example 6.9 `<<` overloaded for `Person` - with chaining.

```
friend ostream & operator<<(ostream &,const Person &); //in class decl.
ostream &operator<<(ostream &os, const Person &p) {
   os << “My name is “ << p.name << endl;
   return os;
}

// in a function
   Person adam(“adam”), susan(“susan”);
   cout << adam;// ok
   cout << adam << “ and “ << susan << endl;// ok
```

- ❍ general form
  - ● in the same files as member methods
  - ● inside class declaration if `friend`, outside otherwise

```
friend ostream & operator<<(ostream &,const Person &);
```

Exercise 6.4 Redo Exercise 6.3 replacing `show()` method with overloaded `<<`.

6.6 Overloading Assignment

- ❍ Must overload if overloading copy constructor
- ❍ It is **not inherited** (the only exception)

```
ClassName& ClassName::operator=(const ClassName &sourceObject){
   // assign sourceObject to *this and return *this
}
```

- ❍ Needed on
  - ❍ explicit object assignments
  - ❍ potentially on objects created and initialized with =

- ❍ Default assignment
  - ❍ copy bytes
  - ❍ should be the same as copy except that
    - not a constructor so no need to allocate storage but may need to deallocate and allocate
    - prevent not to assign to itself

Example 6.10 The first two are potentially handled by copy constructor only.

```
   Person adam, susan.
   Person john=susan;           // assignment or copy constructor
   Person john=Person(susan);   // assignment or copy constructor
   adam=susan;                  // assignment
```

Example 6.11 Assume class `String` with dynamic allocation as in Example 3.21 Then, we may implement assignment by allocating space (*deep copy*), copying:

```
String& String::operator=(const String& st) {
   if (this==&st)
      return *this;        // no copying to itself
   int x=strlen(st.str);
   if (len>x)
      strcpy(str,st.str);  // enough space here, avoiding delete/new
   else {
      delete [] str;       // return storage as might need more or less
      len=x+1;
      str=new char[len];
      strcpy(str,st.str);
   }
   return *this;
}
```

Exercise 6.5 Strings again, dynamic memory, with overridden copy and assignment.

6.7 Type Conversion from Class

- Conversions from a class to intristic types can also be defined
  - not for converting to another class
  - use *conversion* operators (not constructors)
    - must be methods
    - no return type
    - no arguments

```
operator typeToConvertTo(void);
```

Example 6.12 Suppose `Person` has a member method

```
   operator int(void);        // maybe evaluates to the Person's age

// in a function
   Person adam(23);           // create adam with age=23
   int x;
   x=(int)adam;               // old syntax
   x=int(adam);               // alternative syntax
```

Exercise 6.6 Observe automatic conversions and casts.
Explain what happens with `bigger=325` (there is default assignment so 325 must be converted to `StoneBag`, and this will work after 325 is promoted to `double` 325.0).

- Be careful not to overuse conversions and casting, ambiguity may easily result

6.8 Memory Management Operators

- Memory management (`new`, `new[]`, `delete`, `delete[]`) can be overloaded
  - to control memory manegemnt for all or some classes
- If overloading `new` (`delete`), should also overload the `[]` versions and `delete`(`new`)
- They can be overloaded as either/both
  - top-level
    - will apply to all memory calls except when overloaded as methods
    - prototype is different from other to-level operators

- methods
    - will apply to all objects of the class
- Prototypes are the same for top-level and members
    - both can take other optional parameters
    - new (method will be implemented with resolution operator and declared inside class)

      ```
      void* operator new(size_t);
      void* operator new[](size_t);
      ```

        - `new Person;` will initialize 1st argument to `sizeof(Person)`
        - `new Person[2];` will initialize 1st argument to `sizeof(Person)*2`
    - delete (method will be implemented with resolution operator and declared inside class)

      ```
      void operator delete(void*);
      void operator delete[](void*);
      ```
- You will have to implement class MemoryManager which will alocate a chunk and give it away piece by piece

6.9 Subscript

- `[]` must be overloaded asa method
    - will apply only to this class
    - useful for creating user-defined array-like containers
    - second parameter may be integer, as in index, but can be anything

Example 6.13 If class `A` has `[]` overloaded with an integer paramater, then this will refer to the overloaded operator:

```
A a;
a[i]=x;
```

- `[]` generally requires two forms
    - to handle `const` objects, `const` version must be provided

      ```
      const retType& operator[](parameter) const;
      ```
    - to handle using `[]` in modifying expressions, non-`const` version is needed

      ```
      retType& operator[](parameter);
      ```

Exercise 6.7 Class IntArray handles 1-d arrays, does boundary checking. It uses the same `[]` access operator by overloadidng.

- Templates will allow a class such as IntArray to be created for all types not just integer and to work as multi-dimensional array

6.10 Function Call

- `()` must be overloaded as a member
  - will apply only to this class
- It is used to handle expressions like this

  `object(parameters)`
  - the object will be the implict argument
  - the parameters must be declared in the operator
- Same as with [], we usually need `const` and non-`const` versions

Example 6.14 Suppose we need `Int2DArray`. Double indexing can be handled via `()` operator so that some `Int2DArray` called `a2` can be accessed as

```
   a2(2,3)=5;
```

to write 5 into its 2nd row 3rd column element.

```
int& Int2DArray::operator()(int x, int y) {
   if (x<0 || y<0 || x>=size1 || y>=size2)
      throw “Bad indexes”;
   return p[x*size2+j];    // assuming internal array is 1D of
}                          // size size1*size2
```