Operator Overloading

Jiun-Long Huang
National Chiao Tung University

Introduction

- Consider a user-defined class
- How to add two vectors?
  - With a function
    - TwoDVector v3=v1.add(v2);
  - With operator overloading
    - TwoDVector v3=v1+v2;
- Operator overloading enables programmers to use existing operators to manipulate objects of their own classes.

```
class TwoDVector
{
  private:
    float x, y;
  public:
  ...
};
TwoDVector v1, v2;
```

Fundamentals of Operator Overloading

- Operator overloading is a special type of function overloading.
  - Programmers are able to instruct the C++ compiler to apply existing operators in an effort to manipulate objects of their own data types.
- An operator is overloaded if it can be used to perform multiple operations.

Fundamentals of Operator Overloading (contd.)

- The * operator
  - can be used to multiply two values of built in types such as int, float, or double,
  - can be to declare or dereference a pointer.
  - can be further overloaded to multiply two objects of user defined types.

Example: TwoDVector

```
int x=3, y=5, z;
z=x*y;
//* is used to perform integer multiplication
float f=3.4;
float *fpt=&f;    //* is used to declare a pointer
*fpt=5.6;   //* is used to dereference the pointer
TwoDVector v1(1,45), v2(2,0), v3;
v3=v1*v2;
//* is used to perform a user defined operation
//on class TwoDVector
```

Operator Functions

- An operator function defines an operation that is performed on objects of specific types.

```
return_type operator@( parameter list )
{
    //operation to be performed
}
```

Operator Function

- Operator functions can be designed as
  - Non-static member functions
  - Non-member functions
- Member operator functions must be non-static in order to access non-static data members of the class.

- The only way for non-friend, non-member functions to access the class's private and protected data is through public member interface functions.
  - This process involves unnecessary function calls and may decrease the speed of a program.
- Use friend functions when overloading operators through non-member functions rather than non-friend functions

Example: TwoDVector

- Addition of 2 vectors
  - v1: (x1, y1)
  - v2: (x2, y2)
  - v1+v2: (x1+x2, y1+y2)
- Multiplication of 2 vectors
  - $v1*v2=|v1|*|v2|*\cos\theta=x1*x2+y1*y2$
  - v1*2=(2*x1, 2*y1)
  - 2*v1=(2*x1, 2*y1)

Example: TwoDVector (contd.)

```
TwoDVector v1(1,1), v2(2,2) v3;
v3=v1+v2;
// Case 1: v3=operator+(v1, v2);
// Prepare: TwoDVector operator+(TwoDVector, TwoDVector)
// Case 2: v3=v1.operator+(v2);
// Prepare: operator+(TwoDVector) in class TwoDVector
```

```
#include <iostream>
using namespace std;
class TwoDVector
{
private:
  float x, y;
public:
  TwoDVector(float x=0, float y=0)
  {
    this->x=x;
    this->y=y;
  }
  void print(void);
  friend TwoDVector operator+(const TwoDVector &,
                              const TwoDVector &);
  friend float operator*(const TwoDVector &,
                         const TwoDVector &);
  friend TwoDVector operator*(const TwoDVector &, float);
  friend TwoDVector operator*(float, const TwoDVector &);
};
```

Use friend functions
Use copy by reference for performance

```
void TwoDVector::print(void)
{
  cout<<"("<<x<<","<<y<<")"<<endl;
}
TwoDVector operator+(const TwoDVector & v1,
                     const TwoDVector & v2)
{
  TwoDVector v;
  v.x=v1.x+v2.x;
  v.y=v1.y+v2.y;
  return v;
}
TwoDVector operator*(const TwoDVector & v1, float a)
{
  TwoDVector v;
  v.x=v1.x*a;
  v.y=v1.y*a;
  return v;
}
```

```
TwoDVector operator*(float a, const TwoDVector & v1)
{
  TwoDVector v;
  v.x=v1.x*a;
  v.y=v1.y*a;
  return v;
}
float operator*(const TwoDVector & v1,
                const TwoDVector & v2)
{
  return v1.x*v2.x+v1.y*v2.y;
}
int main()
{
  TwoDVector v1(1,2), v2(3,4), v3;
  v3=v1+v2;
  v3.print();
  cout<<v1*v2<<endl;
  v3=2*v1;
  v3.print();
  return 0;
}
```

```
(4,6)
11
(2,4)
```

```
#include <iostream>
using namespace std;
class TwoDVector
{
private:
  float x, y;
public:
  TwoDVector(float x=0, float y=0) {
    this->x=x;
    this->y=y;
  }
  void print(void);
  TwoDVector operator+(const TwoDVector &);
  friend float operator*(const TwoDVector &,
                         const TwoDVector &);
  friend TwoDVector operator*(const TwoDVector &, float);
  friend TwoDVector operator*(float, const TwoDVector &);
};
```

Use member function for +

Cannot use member function for *
Cannot implement float.operator+(TwoDvector)

```
TwoDVector TwoDVector::operator+(const TwoDVector & v2)
{
 // The first operand is passed by this pointer
   TwoDVector v;
   v.x=this->x+v2.x;
   v.y=this->y+v2.y;
   return v;
}
// ...
// other non-member functions are omitted
// ...
int main()
{
   TwoDVector v1(1,2), v2(3,4), v3;
   v3=v1+v2;
   v3.print();
   cout<<v1*v2<<endl;
   v3=2*v1;
   v3.print();
   return 0;
}
```

Operators

- There are four operators that can be overloaded in both unary and binary forms:
  - + - * &
- The following five operators cannot be overloaded
  - . .* :: ?: sizeof

Operators

- Operators that can be overloaded

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

Note

- When overloading operators, the following cannot be changed:
  - Operator precedence
  - Grouping (which symbols can be grouped together to create a new operator)
  - Number of operands
  - Original meaning (an operation that is performed on objects of built in types)
- Otherwise → Syntax error

Member Function and Non-Member Function

- Most of the overloadable operators (except the operators = () [] ->) can be overloaded by either non-static member functions or friend functions.
- The operators = () [] -> must be overloaded using member operator functions.

- Member operator functions cannot be used in the following case
  - When an object on the left side of a binary operator is not instantiated from the operator's function class.

```
TwoDVector v1;
TwoDVector v2=v1*2;
//operator*(v1,2)
//v1.operator*(2)
```

```
TwoDVector v3=2*v1;
//operator*(2,v1)
//2.operator*(v1) ERROR
```

Operator = and &

- The = (assignment) operator and the & (address of) operator can be used with objects of every user defined type without explicitly overloading these two operators relative to any specific class.
  - The = operator, by default, creates a bit-by-bit copy of an object,
  - The & operator returns a memory address of the object.

= Assignment

- It is particularly important to overload the = operator relative to a class containing pointers as members, to avoid having the same pointer in two or more different objects.
- Destroying one object in this case would damage another objects.

- The = operator can only be overloaded by a member operator function.
- C++ automatically creates the default assignment operator function (bit-copy) for every class for which a user defined = operator function is not supplied.

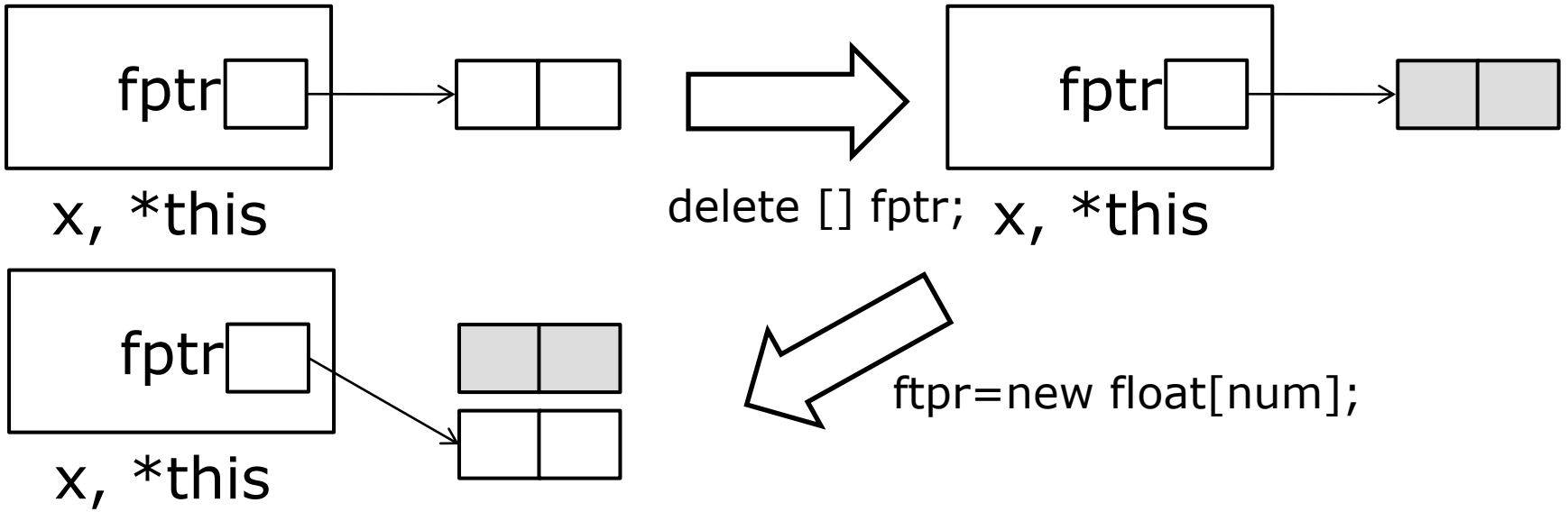
```
class Numbers
{
private:
  float *fptr; //points to a float array
  int num;  //size of the array
public:
  Numbers(int=6); //regular constructor
  Numbers(const Numbers &);   //copy constructor
  ~Numbers() { delete [] fptr; }    //destructor
  const Numbers & operator=(const Numbers &);
};
Numbers::Numbers(int x)
{
  num=x;
  fptr=new float[num];  //Allocates memory dynamically
  if(!fptr) {
    cout<<"Memory allocation error!";
    exit(1);
  }
  for(int i=0; i<num; i++)    //Initializes array
    fptr[i]=0;
}
```

```
Numbers::Numbers(const Numbers & f) {
  num= f.num;
  fptr=new float[num];  //Allocates memory dynamically
  if(!fptr) {
    cout<<"Memory allocation error!";
    exit(1);
  }
  for(int i=0; i<num; i++)
    fptr[i]=f.fptr[i];  //Copies arrays
}
```

```
const Numbers & Numbers::operator=(const Numbers & f)
{
  if(&f!=this)    //Prevents self assignment
  {
    //Frees memory allocated by constructor
    delete [] fptr;
    num=f.num;
    fptr=new float[num];
    if(!fptr)
    {
      cout<<" Memory allocation error.";
      exit(1);
    }
    for(int i=0; i<num; i++)
      fptr[i]=f.fptr[i];
  }
  return *this;
}
```

- What will happen when we do not prevent self assignment?

```
Numbers x;
x=x;  //x is assigned to itself
```



- If the operator=() function return type is changed to void, the multiple assignments shown above cannot be done.

```
Numbers x, y, z;
x=y=z;
```

Overloading Unary Operators

```
#include <iostream>
using namespace std;
class TwoDVector
{
private:
  float x, y;
public:
  TwoDVector(float x=0, float y=0)
  {
    this->x=x;
    this->y=y;
  }
  void print(void);
  friend TwoDVector operator!(TwoDVector &);
};
```

```
void TwoDVector::print(void)
{
  cout<<"("<<x<<","<<y<<")"<<endl;
}
TwoDVector operator!(TwoDVector & v1)
{
  TwoDVector v;
  v.x=-1*v1.x;
  v.y=-1*v1.y;
  return v;
}
int main()
{
  TwoDVector v1(1,2), v2;
  v2=!v1;
  v2.print();
  return 0;
}
```

++ and --

- Both the ++ operator and the -- operator have two forms: prefix and postfix.
  - ++a, a++
  - We should overload them in both forms.
- Each form requires a separate operator function.
  - Both operator functions in this case have the same name operator++( ) or operator--() and the same parameter lists.
  - How to distinguish them?

- Overloaded functions must have different parameter lists to prevent ambiguity (compiler's confusion).
  - C++ adds a dummy argument to the parameter list of the postfix form in order to distinguish between the two forms when overloading ++ and -- .
- This useless argument is an unnamed int and its only purpose is to eliminate ambiguity.

```
#include <iostream>
using namespace std;
class TwoDVector
{
private:
  float x, y;
public:
  TwoDVector(float x=0, float y=0) {
    this->x=x;
    this->y=y;
  }
  void print(void);
  friend TwoDVector operator++(TwoDVector &);
  friend TwoDVector operator++(TwoDVector &, int);
};
void TwoDVector::print(void) {
  cout<<"("<<x<<","<<y<<")"<<endl;
}
```

Use copy by reference for functionality

```
TwoDVector operator++(TwoDVector & v1) {
  v1.x++;
  v1.y++;
  cout<<"Prefix form"<<endl;
  return v1;
}
TwoDVector operator++(TwoDVector & v1, int) {
  TwoDVector v(v1);
  v1.x++;
  v1.y++;
  cout<<"Postfix form"<<endl;
  return v;
}
int main() {
  TwoDVector v1(1,2), v2;
  v2=++v1;
  v1.print();
  v2.print();
  v2=v1++;
  v1.print();
  v2.print();
  return 0;
}
```

```
Prefix form
(2,3)
(2,3)
Postfix form
(3,4)
(2,3)
```

Overloading the Stream Operators

- Overload the stream insertion (<<) and stream extraction (>>) operators in order to perform user defined I/O operations.
- The two classes called istream and ostream are used when overloading the << and >> operators.
  - cin is an instance of the istream class and is connected to the standard input device, most commonly a keyboard.

- cout is an instance of the ostream class and is connected to the standard output device, most commonly a display screen.

- To overload the stream insertion operator, a friend operator function should be used.

```
friend ostream & operator<<(ostream &,
                            const class_name &);
// in class class_name
ostream & operator<<(ostream & s, const class_name & o)
{
  //body of the insertion operator
  return s;
}
```

Caution!

- The body of operator>>() defines stream input operations to be performed on an object of the user defined class, for which the operator function is defined.

```
friend istream & operator>>(istream &, class_name &);
// in class class_name

istream & operator>>(istream & str, class_name & obj)
{
  //body of the extraction operator
  return str;
}
```

```
#include <iostream>
using namespace std;
class TwoDVector
{
private:
  float x, y;
public:
  TwoDVector(float x=0, float y=0)
  {
    this->x=x;
    this->y=y;
  }
  friend ostream & operator<<(ostream &,
                              const TwoDVector &);
  friend istream & operator>>(istream &, TwoDVector &);
};
```

```
ostream & operator<<(ostream & os,
                     const TwoDVector & v)
{
  os<<"("<<v.x<<","<<v.y<<")";
  return os;
}
istream & operator>>(istream & is, TwoDVector & v)
{
  is>>v.x;
  is>>v.y;
  return is;
}
int main()
{
  TwoDVector v1;
  cin>>v1;
  cout<<v1;
  return 0;
}
```

```
2 3
(2,3)
```

Class string

- Class string
  - Header <string>, namespace std
  - Can initialize string s1("hi");
  - Overloaded << (as in cout << s1)
  - Overloaded relational operators
    - ==, !=, >=, >, <=, <
  - Assignment operator =
  - Concatenation (overloaded +=)

- Class string (Cont.)
  - Substring member function substr
  - s1.substr(0, 14): Starts at location 0, gets 14 characters
  - s1.substr(15): Substring beginning at location 15, to the end
  - Overloaded []
    - Access one character
  - No range checking (if subscript invalid)

- Member function at
  - Accesses one character
  - Example: s1.at(10);
  - Has bounds checking, throws an exception if subscript is invalid
  - Will end program (learn more in Chapter 16)
- Member function length
  - Return the length of the string
- Member function empty
  - Test whether the string is empty

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
  string a("string1"), b("string2");
  if (a.length()>=6)
    cout << a[6] << endl;
  a+=b;
  cout<<a;
}
```

```
1
string1string2
```