Polymorphism in C++

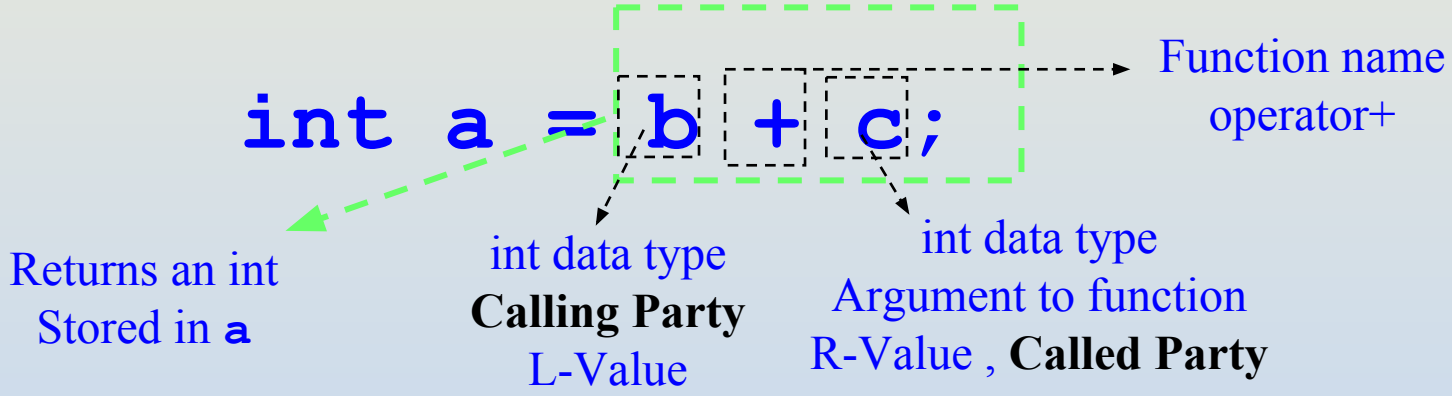
OPERATOR OVERLOADING

Operator Overloading

- **Operator Overloading** allows a programmer to define new types from the built-in types.
  - **Operator Overloading** is useful for redefining built-in operations for user defined types.
  - **Operator Overloading** should be used to perform the same function or similar function on class objects as the built-in behavior.
- Overloading an operator does **not change**:
  - the operator precedence,
  - the associativity of the operator,
  - the arity of the operator, or
  - the meaning of how the operator works on objects of built-in types.

Operator Overloading

- Each individual operator must be overloaded for use with user defined types.
  - Overloading the assignment operator and the subtraction operator does **not** overload the **-=** operator.
- Operator Overloading enables to apply standard operators (such as +,-,*,<, and so on) to objects of the programmer defined type.



- It helps to enhance simplicity in program structure.

Operator Overloading

- **What?**
  - an operator that has multiple meanings
  - varies depending on use
- **Why?** Ease of use is a principle of OO
- **How?** by defining as an operator function
  - functions that can extend meaning of built-in operators (cannot define your own new operators)
  - keyword operator is in definition, followed by the operator to be overloaded
- **Used**
  - method syntax or operator syntax
    - s1.operator>(s2) vs. s1 > s2

Why Operation Overloading

- makes statements more intuitive and readable.

  for example:

```
Date d1(12,3,1989);
Date d2;
d2.add_days(d1,45);
// can be written with the + operator as
d2=d1+45;
```

- Extension of language to include user-defined types
  - I.e., classes
- Make operators sensitive to context
- Generalization of function overloading

Restrictions on Overloading

Operators that can be overloaded							
+	-	*	/	%	^	&	|
~	!	=	<	>	+=	-=	*=
/=	%=	^=	&=	|=	<<	>>	>>=
<<=	==	!=	<=	>=	&&	||	++
--	->*	,	->	[]	()	new	delete
new[]	delete[]						

Operators that cannot be overloaded				
.	.*	::	?:	sizeof

- The order of precedence cannot be changed for overloaded operators.
- Default arguments may not be used with overloaded operators
- New operators cannot be created
- Overloading must be explicit, i.e. overloading + does not imply += is overloaded

Table 8-1 Unary operators that can be overloaded

Operator	Usual use	Associativity
->	member	left to right
->*	indirect pointer to member	left to right
!	not	right to left
&	address of	right to left
*	indirection (dereference)	right to left
+	positive value	right to left
-	negative value	right to left
++	increment	right to left
--	decrement	right to left
~	complement	right to left

Table 8-3 Precedence of operators

Operator	Description
::	scope resolution
. (dot operator) ->[ ] ( ) ++ --	member member pointer array element subscript function call postfix increment postfix decrement
++ -- ! + - * & new delete	prefix increment prefix decrement not positive value negative value dereference address allocate memory deallocate memory
* / %	multiply divide modulus
+ -	addition subtraction
<< >>	insertion extraction

Table 8-2 Binary operators that can be overloaded

Operator	Usual use	Associativity
*	multiplication	left to right
/	division	left to right
%	remainder (modulus)	left to right
+	addition	left to right
-	subtraction	left to right
<<	shift bits to left	left to right
>>	shift bits to right	left to right
>	greater than	left to right
<	less than	left to right
>=	greater than or equal to	left to right
<=	less than or equal to	left to right
==	equal to	left to right
!=	not equal to	left to right
&&	logical AND	left to right
||	logical OR	left to right
&	bitwise AND	left to right
|	bitwise inclusive OR	left to right
^	bitwise exclusive OR	left to right
=	assignment	right to left
+=	add and assign	right to left
-+	subtract and assign	right to left
*=	multiply and assign	right to left
/=	divide and assign	right to left
%=	modulus and assign	right to left
&=	bitwise AND and assign	right to left
|=	bitwise OR and assign	right to left
^=	bitwise OR and assign	right to left
<<=	shift left and assign	right to left
>>=	shift right and assign	right to left
()	function call	left to right
[]	array element subscript	left to right
->	member pointer	left to right
new	allocate memory	right to left
delete	deallocate memory	right to left
,	comma	left to right

Cont..

Table 8-3 Precedence of operators (continued)

Operator	Description
< > <= >=	less than greater than less than or equal greater than or equal
== !=	equal to not equal to
&&	logical AND
||	logical OR
= += -= *= /= %=	assignment add and assign subtract and assign multiply and assign divide and assign modulus and assign

Table 8-4 Operators that cannot be overloaded

Operator	Usual use
. (dot operator)	member
*	pointer to member
::	scope resolution
?:	conditional
sizeof	size of

Operator Functions

- Operator functions may be defined as either member functions or as non-member functions.
  - Non-member functions are usually made **friends** for performance reasons.
  - Member functions usually use the **this** pointer implicitly.

Syntax:

***returnType* operator*(*parameters*);**

↑ ↑ ↑

any type *keyword* *operator symbol*

- *Return type* may be whatever the operator returns
  - Including a reference to the object of the operand
- *Operator symbol* may be any overloadable operator from the list.

Cont..

- The operator overloading functions for overloading (), [], -> or the assignment operators **must** be declared as a class member.
- All other operators may be declared as non-member functions.
- Operator overload function is a function just like any other
- Can be called like any other – e.g.,

a.operator+(b)

- *C++* provides the following short-hand

a+b

- If operator overload function is declared global then

operator+(a, b)

- also reduces to the following short-hand

a+b

Cont..

- To use any operators on a class object, …
  - The operator must be overloaded for that class.

- Three Exceptions: {overloading not required}
  - Assignment operator (=)
    - Memberwise assignment between objects
    - Dangerous for classes with pointer members!!
  - Address operator (&)
    - Returns address of the object in memory.
  - Comma operator (,)
    - Evaluates expression to its left then the expression to its right.
    - Returns the value of the expression to its right.

Operator Functions as Class Members

- Leftmost operand must be of *same class* as operator function.
- Use **this** keyword to implicitly get left operand argument.
- Operators (), [], -> or any assignment operator must be overloaded as a class member function.
- Called when
  - Left operand of binary operator is of this class.
  - Single operand of unary operator is of this class.

Operator Functions as Global Members

- Need parameters for both operands.
- Can have object of different class than operator.
- Can be made a **friend** to access **private** or **protected** data.

- Stream Insertion and Extraction Operators as Global Functions
  - Overload << operator used where
    - Left operand of type `ostream &`
      - » Such as **`cout`** object in **`cout << classObject`**
  - Overload >> has left operand of `istream &`
    - Left operand of type `istream &`
      - » Such as **`cin`** object in **`cout >> classObject`**
  - Reason:–
    - These operators are associated with class of *right* operand

Cont..

- May need + to be commutative
  - So both "**`a + b`**" and "**`b + a`**" work as expected.
- Suppose we have two different classes
  - Overloaded operator can only be member function when its class is on left.
    - **`HugeIntClass + long int`**
    - Can be member function
  - For the other way, you need a global overloaded function.
    - **`long int + HugeIntClass`**

Example

- << and >> operators
  - Already overloaded to process each built-in type (pointers and strings).
  - Can also process a user-defined class.
    - Overload using global, friend functions
- Example program
  - Class **PhoneNumber**
    - Holds a telephone number
  - Prints out formatted number automatically.
    - **(123) 456-7890**

Cont..

```
// Overloading the stream-insertion and  stream-extraction operators.
#include <iostream>

using std::cout;
using std::cin;
using std::endl;
using std::ostream;
using std::istream;

#include <iomanip>

using std::setw;

// PhoneNumber class definition
class PhoneNumber {
  friend ostream &operator<<( ostream&, const PhoneNumber & );
  friend istream &operator>>( istream&, PhoneNumber & );

private:
  char areaCode[ 4 ];  // 3-digit area code and null
  char exchange[ 4 ];  // 3-digit exchange and null
  char line[ 5 ];      // 4-digit line and null
}; // end class PhoneNumber
```

Notice function prototypes for overloaded operators **>>** and **<<**

They must be non-member **`friend`** functions, since the object of class **`Phonenumber`** appears on the right of the operator.

```
cin >> object
cout<< object
```

Cont..

```
// overloaded stream-insertion operator; cannot be
// a member function if we would like to invoke it with
// cout << somePhoneNumber;
ostream &operator<<( ostream &output,const PhoneNumber &num )
{
   output << "(" << num.areaCode <<") "
        << num.exchange <<"-" << num.line;

   return output;    // enables cout << a << b << c;

} // end function operator<<
// overloaded stream-extraction operator; cannot be  a member function if we would like to invoke it with
// cin >> somePhoneNumber;
istream &operator>>( istream &input, PhoneNumber &nu
{
   input.ignore();              // skip (
   input >> setw( 4 ) >> num.areaCode
   input.ignore( 2 );           // skip )
   input >> setw( 4 ) >> num.exchange;// input exchange
   input.ignore();              //skip dash (-)
   input >> setw( 5 ) >> num.line;    // input line
   return input;     // enables cin >> a >> b >> c;
} // end function operator>>
```

The expression:
`cout << phone;`
is interpreted as the function call:
`operator<<(cout, phone);`

`output` is an alias for **`cout.`**

This allows objects to be cascaded.
`hone1 << phone2;`

`<(cout, phone1)`, and

Next, **`cout << phone2`** executes.

`ignore()` skips specified number of characters from input (1 by default).

Stream manipulator **`setw`** restricts number of characters read. **`setw(4)`** allows 3 characters to be read, leaving room for the null character.

```
int main()
   {
     PhoneNumber phone;// create object phone

     cout << "Enter phone number in the form (123) 456-7890:\n";

    // cin >> phone invokes operator>> by implicitly issuing
    // the non-member function call operator>>( cin, phone )
    cin >> phone;

    cout << "The phone number entered was: ";

     // cout << phone invokes operator<< by implicitly issuing
     // the non-member function call operator<<( cout, phone )
     cout << phone << endl;
    return 0;
   } // end main
```

```
Enter phone number in the form (123) 456-7890:
(800) 555-1212
The phone number entered was: (800) 555-1212
```

Unary Operators

- The unary operators operate on the object for which they were called and normally, this operator appears on the left side of the object, as in !obj, -obj, and ++obj but sometime they can be used as postfix as well like obj++ or obj--.

- Can overload as
  - Non-**static** member function with no arguments.
  - As a global function with one argument.
    - Argument must be class object or reference to class object.
- Why non-**static**?
  - **static** functions only access static data
  - Not what is needed for operator functions

Example: **Integer** Class

```
#include <iostream>
using namespace std;
// Non-member functions
class Integer
{
    long i;
    Integer* This()
    {
        return this;
    }
public:
    Integer(long ll = 0) : i(ll) {}
    // No side effects takes const& argument:
    friend const Integer& operator+(const Integer& a);
    friend const Integer operator-(const Integer& a);
    friend const Integer operator~(const Integer& a);
    friend Integer* operator&(Integer& a);
    friend int operator!(const Integer& a);
    // Side effects have non-const& argument:
    friend const Integer& operator++(Integer& a);// Prefix
    friend const Integer operator++(Integer& a, int);// Postfix
    friend const Integer& operator--(Integer& a);// Prefix
    friend const Integer operator--(Integer& a, int);// Postfix
};
```

```
// Global operators:
const Integer& operator+(const
Integer& a)
{
    cout << "+Integer\n";
    return a; // Unary + has no effect
}
const Integer operator-(const
Integer& a)
{
    cout << "-Integer\n";
    return Integer(-a.i);
}
const Integer operator~(const
Integer& a)
{
    cout << "~Integer\n";
    return Integer(~a.i);
}
Integer* operator&(Integer& a)
{
    cout << "&Integer\n";
    return a.This(); // &a is
recursive!
}
int operator!(const Integer& a)
{
    cout << "!Integer\n";
    return !a.i;
}
```

```
// Prefix; return incremented value
const Integer& operator++(Integer& a)
{
    cout << "++Integer\n";
    a.i++;
    return a;
}
// Postfix; return the value before
increment:
const Integer operator++(Integer& a,
int)
{
    cout << "Integer++\n";
    Integer before(a.i);
    a.i++;
    return before;
}
// Prefix; return decremented value
const Integer& operator--(Integer& a)
{
    cout << "--Integer\n";
    a.i--;
    return a;
}
// Postfix; return the value before
decrement:
const Integer operator--(Integer& a,
int)
{
    cout << "Integer--\n";
    Integer before(a.i);
    a.i--;
    return before;
```

```
// Show that the overloaded
operators work:
void f(Integer a)
{
    +a;
    -a;
    ~a;
    Integer* ip = &a;
    !a;
    ++a;
    a++;
    --a;
    a--;
}
// Member functions (implicit
"this"):
class Byte
{
    unsigned char b;
public:
    Byte(unsigned char bb = 0) :
b(bb) {}
// No side effects: const member
function:
    const Byte& operator+() const
    {
        cout << "+Byte\n";
        return *this;
    }
const Byte operator-() const
    {
        cout << "-Byte\n";
        return Byte(-b);
    }
const Byte operator~() const
    {
        cout << "~Byte\n";
        return Byte(~b);
    }
Byte operator!() const
    {
        cout << "!Byte\n";
        return Byte(!b);
    }
Byte* operator&()
    {
        cout << "&Byte\n";
        return this;
    }
```

```
// Side effects: non-const member function:
const Byte& operator++()   // Prefix
    {
        cout << "++Byte\n";
        b++;
        return *this;
    }
const Byte operator++(int)   // Postfix
    {
        cout << "Byte++\n";
        Byte before(b);
        b++;
        return before;
    }
const Byte& operator--()   // Prefix
    {
        cout << "--Byte\n";
        --b;
        return *this;
    }
const Byte operator--(int)   // Postfix
    {
        cout << "Byte--\n";
        Byte before(b);
        --b;
        return before;
    }
```

```
void g(Byte b)
{
    +b;
    -b;
    ~b;
    Byte* bp = &b;
    !b;
    ++b;
    b++;
    --b;
    b--;
}


int main()
{
    Integer a;
    f(a);
    Byte b;
    g(b);
}
```

Overloading Binary Operators

- Non-static member function with one argument.

```
return_type operator symbol(R-Value);
```

or

- Global function with two arguments:
  - One argument must be class object or reference to a class object.

```
return_type   operator symbol(L-Value, R-Value);
```

This is the mechanism by which the compiler prevents you from redefining built-in operations!

Overloading Binary Operators

- If a non-**`static`** member function, it needs one argument.

```
class String {
public:
    String & operator+=( const String &);
    …
};
```

- By shorthand rule

  `y += z` becomes **`y.operator+=( z )`**

Cont..

- If a global function, it needs two arguments

```
class String {
public:
    String & operator+=( String &,
       const String & );
    …
};
```

- By short-hand rule
  - **`y += z`** becomes **`operator+=(y, z)`**