



Operator Overloading

Examples

- There are two main types of polymorphism in C++:

1. **Compile-Time Polymorphism (Static Binding)**

- The function or operator to be executed is determined at compile time.

- **Achieved Through:**

- **Function Overloading:** Using the same function name with different parameter types or numbers to perform different tasks.

- **Operator Overloading:** Redefining standard operators (like `+`, `-`, `*`, etc.) for user-defined data types to provide custom behavior.

- **Key Features:**

- Increases code readability and reusability.
 - Decided at compile time, making it faster.

2. Run-Time Polymorphism (Dynamic Binding)

- The function to be executed is determined at runtime based on the type of the object.

•Achieved Through:

- Virtual Functions:** Functions declared in the base class with the `virtual` keyword and overridden in derived classes.
- Abstract Classes:** Classes with at least one pure virtual function, which act as blueprints for derived classes.

•Key Features:

- Enables dynamic method dispatch.
- Allows derived class behavior to replace base class behavior when accessed via a base class reference or pointer.

❑ **Operator Overloading as Polymorphism:** A form of compile-time polymorphism where operators are redefined for user-defined types.

•Purpose:

- To perform custom operations using standard operators on complex data types like classes and structs.

•Key Benefits:

- Enhances code intuitiveness by using operators with familiar syntax for custom tasks.
- Reduces the need for verbose function calls.




Overloading the Extraction and Insertion Operators

Overloading the Extraction and Insertion Operators

- Overloading the extraction and insertion operators is a powerful feature of C++.
- It lets you treat I/O for user-defined data types in the same way as basic types like int and double.
- For example, if you have an object of class `crwdad` called `cd1`, you can display it with the statement `cout << "\ncd1=" << cd1;` just as if it were a basic data type

Overloading the Extraction and Insertion Operators

- We can overload the extraction and insertion operators so they work with the display and keyboard (cout and cin) alone.
- With a little more care, we can also overload them so they work with disk files.
- We'll look at examples of both of these situations.



Overloading for cout and cin

ENGLIO

Instructor: Ms. Aasma Sajjad

12/18/2024

7

English Distance class

- `#include <iostream>`
- `using namespace std;`
- `class Distance //English Distance class`
- `{`
- `private:`
- `int feet;`
- `float inches;`
- `public:`
- `Distance() : feet(0), inches(0.0) //constructor (no args)`
- `{}`
- `//constructor (two args)`
- `Distance(int ft, float in) : feet(ft), inches(in)`
- `{}`
- `friend istream& operator >> (istream& s, Distance& d);`
- `friend ostream& operator << (ostream& s, Distance& d);`
- `};`

operator >>

- `istream& operator >> (istream& s, Distance& d)` `//get Distance`
- `{` `//from user`
- `cout << "\nEnter feet: "; s >> d.feet;` `//using`
- `cout << "Enter inches: "; s >> d.inches;` `//overloaded`
- `return s;` `//>> operator`
- `}`

operator <<

- ostream& operator << (ostream& s, Distance& d) *//display*
- { *//Distance*
- s << d.feet << "\"'-" << d.inches << "\""; *//using*
- return s; *//overloaded*
- }

Main

- `int main()`
- `{`
- `Distance dist1, dist2; //define Distances`
- `Distance dist3(11, 6.25); //define, initialize dist3`
- `cout << "\nEnter two Distance values:";`
- `cin >> dist1 >> dist2; //get values from user`
- `//display distances`
- `cout << "\ndist1 = " << dist1 << "\ndist2 = " << dist2;`
- `cout << "\ndist3 = " << dist3 << endl;`
- `return 0;`
- `}`

Output

- Enter feet: 10
- Enter inches: 3.5
- Enter feet: 12
- Enter inches: 6
- $\text{dist1} = 10' - 3.5''$
- $\text{dist2} = 12' - 6''$
- $\text{dist3} = 11' - 6.25''$

Convenience in using overloaded <<, >>

- Notice how convenient and natural it is to treat Distance objects like any other data type, using statements like
- `cin >> dist1 >> dist2;`
- and
- `cout << "\ndist1=" << dist1 << "\ndist2=" << dist2;`

Operator return Value

- The << and >> operators are overloaded in similar ways.
- They return, by reference, an object of istream (for >>) or ostream (for <<).
- These return values permit chaining.

Operator Arguments

- The operators take two arguments, both passed by reference.
- The first argument for >> is an object of istream (such as cin). For << it's an object of ostream (such as cout).
- The second argument is an object of the class to be displayed, Distance in this example.
- The >> operator takes input from the stream specified in the first argument and puts it in the member data of the object specified by the second argument.
- The << operator removes the data from the object specified by the second argument and sends it into the stream specified by the first argument.

Friend Operator<>()

- The operator<>() functions must be friends of the Distance class, since the istream and ostream objects appear on the left side of the operator.
- You can overload the insertion and extraction operators for other classes by following these same steps.



Overloading the Assignment Operator

ASSIGN

Alpha Class

- `#include <iostream>`
- `using namespace std;`
- `class alpha{`
- `private:`
- `int data;`
- `public:`
- `alpha() {} //no-arg constructor`
- `alpha(int d) { data = d; } //one-arg constructor`

- `void display() //display data`
- `{ cout << data; }`
- `alpha operator = (alpha& a) //overloaded = operator`
- `{`
- `data = a.data; //not done automatically`
- `cout << "\nAssignment operator invoked";`
- `return alpha(data); //return copy of this alpha`
- `}`
- `};`

Main

- `int main()`
- `{`
- `alpha a1(37);`
- `alpha a2;`
- `a2 = a1; //invoke overloaded =`
- `cout << "\na2="; a2.display(); //display a2`

- `alpha a3 = a2; //does NOT invoke =`
- `cout << "\na3="; a3.display(); //display a3`
- `cout << endl;`
- `return 0;`
- `}`

Output

- Assignment operator invoked
- a2=37
- a3=37

Initialization Is Not Assignment

- In the last two lines of ASSIGN, we initialize the object `a3` to the value `a2` and display it.
- Don't be confused by the syntax here.
- The equal sign in
- `alpha a3 = a2; // copy initialization, not an assignment`
- is not an assignment but an initialization, with the same effect as
- `alpha a3(a2); // alternative form of copy initialization`
- This is why the assignment operator is executed only once, as shown by the single invocation of the line Assignment operator invoked in the output of ASSIGN

Taking Responsibility

- When you overload the = operator you assume responsibility for doing whatever the default assignment operator did.
- Often this involves copying data members from one object to another.
- The alpha class in ASSIGN has only one data item, data, so the operator=() function copies its value with the statement
- `data = a.data;`
- The function also prints the Assignment operator invoked message so that we can tell when it executes.

Passing by Reference

- Notice that the argument to `operator=()` is passed by reference.
- It is not absolutely necessary to do this, but it's usually a good idea. Why?
- As you know, an argument passed by value generates a copy of itself in the function to which it is passed.
- The argument passed to the `operator=()` function is no exception.
- If such objects are large, the copies can waste a lot of memory.
- Values passed by reference don't generate copies, and thus help to conserve memory.

Passing by Reference

- Also, there are certain situations in which you want to keep track of the number of objects (as in the STATFUNC example, where we assigned numbers to the objects).
- If the compiler is generating extra objects every time you use the assignment operator, you may wind up with more objects than you expected.
- Passing by reference helps avoid such spurious object creation.

Returning a Value

- As we've seen, a function can return information to the calling program by value or by reference.
- When an object is returned by value, a new object is created and returned to the calling program.
- In the calling program, the value of this object can be assigned to a new object or it can be used in other ways.
- When an object is returned by reference, no new object is created.
- A reference to the original object in the function is all that's returned to the calling program.

Returning a Value

- The operator=() function in ASSIGN returns a value by creating a temporary alpha object and initializing it using the one-argument constructor in the statement
- `return alpha(data);`
- The value returned is a copy of, but not the same object as, the object of which the overloaded = operator is a member.
- Returning a value makes it possible to chain = operators:
- `a3 = a2 = a1;`

Returning a Value

- However, returning by value has the same disadvantages as passing an argument by value:
- It creates an extra copy that wastes memory and can cause confusion.
- Can we return this value with a reference, using the declarator shown here for the overloaded = operator?
- `alpha& operator = (alpha& a) // bad idea in this case`
- Unfortunately, we can't use reference returns on variables that are local to a function.

Returning a Value


- Remember that local (automatic) variables—that is, those created within a function (and not designated static)—are destroyed when the function returns.
- A return by reference returns only the address of the data being returned, and, for local data, this address points to data within the function.
- When the function is terminated and this data is destroyed, the pointer is left with a meaningless value.
- Your compiler may flag this usage with a warning.

Not Inherited

- The assignment operator is unique among operators in that it is not inherited.
- If you overload the assignment operator in a base class, you can't use this same function in any derived classes.

Beware of Self-Assignment

- A corollary of Murphy's Law states that whatever is possible, someone will eventually do.
- This is certainly true in programming, so you can expect that if you have overloaded the = operator, someone will use it to set an object equal to itself:
- `alpha = alpha;`
- Your overloaded assignment operator should be prepared to handle such self-assignment.



Pitfalls of Operator Overloading and Conversion

Pitfalls of Operator Overloading and Conversion

- Operator overloading and type conversions give you the opportunity to create what amounts to an entirely new language.
- When a , b , and c are objects from user-defined classes, and $+$ is overloaded, the statement $a = b + c$; can mean something quite different than it does when a , b , and c are variables of basic data types.
- The ability to redefine the building blocks of the language can be a blessing in that it can make your listing more intuitive and readable.
- It can also have the opposite effect, making your listing more obscure and hard to understand.

Use Similar Meanings

- Use overloaded operators to perform operations that are as similar as possible to those performed on basic data types.
- You could overload the + sign to perform subtraction, for example, but that would hardly make your listings more comprehensible.

Use Similar Meanings

- Overloading an operator assumes that it makes sense to perform a particular operation on objects of a certain class.
- If we're going to overload the + operator in class X, the result of adding two objects of class X should have a meaning at least somewhat similar to addition.
- For example, in this chapter we showed how to overload the + operator for the English Distance class. Adding two distances is clearly meaningful.

Use Similar Meanings

- We also overloaded + for the String class.
- Here we interpret the addition of two strings to mean placing one string after another to form a third. *This also has an intuitively satisfying interpretation.*
- But for many classes it may not be reasonable to talk about “adding” their objects.
- You probably wouldn’t want to add two objects of a class called employee that held personal data, for example

Use Similar Syntax

- Use overloaded operators in the same way you use basic types.
- For example, if alpha and beta are basic types, the assignment operator in the statement
- `alpha += beta;`
- sets alpha to the sum of alpha and beta.

Use Similar Syntax

- Any overloaded version of this operator should do something analogous.
- It should probably do the same thing as
- $\alpha = \alpha + \beta$;
- where the $+$ is overloaded.

Use Similar Syntax

- If you overload one arithmetic operator, you may for consistency want to overload all of them.
- This will prevent confusion.
- Some syntactical characteristics of operators can't be changed.
- As you may have discovered, you can't overload a binary operator to be a unary operator, or vice versa.

Show Restraint

- Remember that if you have overloaded the + operator, anyone unfamiliar with your listing will need to do considerable research to find out what a statement like
- $a = b + c;$
- really means.
- If the number of overloaded operators grows too large, and if the operators are used in nonintuitive ways, the whole point of using them is lost, and reading the listing becomes harder instead of easier

Show Restraint

- Use overloaded operators sparingly, and only when the usage is obvious.
- When in doubt, use a function instead of an overloaded operator, since a function name can state its own purpose.
- If you write a function to find the left side of a string, for example, you're better off calling it `getleft()` than trying to overload some operator such as `&&` to do the same thing.

Avoid Ambiguity

- Suppose you use both a one-argument constructor and a conversion operator to perform the same conversion (time24 to time12, for example).
- How will the compiler know which conversion to use? It won't.
- The compiler does not like to be placed in a situation where it doesn't know what to do, and it will signal an error.
- So avoid doing the same conversion in more than one way.

Not All Operators Can Be Overloaded

- The following operators cannot be overloaded: the member access or dot operator (.), the scope resolution operator (::), the dereferencing operator (*), sizeof operator, and the conditional operator (?:).
- Also, the pointer-to-member operator (->), which we have not yet encountered, cannot be overloaded.
- In case you wondered, no, you can't create new operators (like *&) and try to overload them; only existing operators can be overloaded.

Not All Operators Can Be Overloaded

- Precedence of an operator can't be changed by overloading but parenthesis can be used to force line order of evaluation of overload.
- The associativity (order of evaluation of operator) can't be overloaded.
 - $X+y+z$ is left to right associative i.e.
 - $(x+y)+z$
 - $X=y=0$ is right to left
- We can't change the arity of operators i.e. the number of operands an operator requires.
 - Unary remain unary
 - Binary remains binary

Member vs Non Member

- Operator functions are functions of operators.
- Operator functions can be member or non-member functions
- Non=member functions are made friend functions
- In this case both class arguments must be explicitly listed in a non member function call.
- When overloading (),[],-> or any of the assignment operators(+=,-=,*=,/=,%=) the operator overload function must be a class member.
- Implementing as a member or non-member function does not has any effect on calling statement
- So which is best?

Left-most operand as class object

- When an operator function is implemented as a member function, the left most or only operand must be a class object of the operators class.
- If the left most operand must be an object of a different class as built-in-type, this operator function must be implemented as a non-member function
- If the non-member function needs to access the private or protected data members of that class directly, then it must be declared as friend

Left-most operand as class object

- Example: overload << operator must have a left operand of type ostream & such as cout in expression
- `cout<<classobj`
- so it must be a non-member function.
- As they require access to private data members of the class object being input or output, hence implemented as friend functions.



Overloading the Assignment Operator

ASSIGN

Alpha Class

- `#include <iostream>`
- `using namespace std;`
- `class alpha{`
- `private:`
- `int data;`
- `public:`
- `alpha() {} //no-arg constructor`
- `alpha(int d) { data = d; } //one-arg constructor`

- `void display() //display data`
- `{ cout << data; }`
- `alpha operator = (alpha& a) //overloaded = operator`
- `{`
- `data = a.data; //not done automatically`
- `cout << "\nAssignment operator invoked";`
- `return alpha(data); //return copy of this alpha`
- `}`
- `};`

Main

- `int main()`
- `{`
- `alpha a1(37);`
- `alpha a2;`
- `a2 = a1; //invoke overloaded =`
- `cout << "\na2="; a2.display(); //display a2`

- `alpha a3 = a2; //does NOT invoke =`
- `cout << "\na3="; a3.display(); //display a3`
- `cout << endl;`
- `return 0;`
- `}`

Output

- Assignment operator invoked
- a2=37
- a3=37

Initialization Is Not Assignment

- In the last two lines of ASSIGN, we initialize the object `a3` to the value `a2` and display it.
- Don't be confused by the syntax here.
- The equal sign in
- `alpha a3 = a2; // copy initialization, not an assignment`
- is not an assignment but an initialization, with the same effect as
- `alpha a3(a2); // alternative form of copy initialization`
- This is why the assignment operator is executed only once, as shown by the single invocation of the line Assignment operator invoked in the output of ASSIGN

Taking Responsibility

- When you overload the = operator you assume responsibility for doing whatever the default assignment operator did.
- Often this involves copying data members from one object to another.
- The alpha class in ASSIGN has only one data item, data, so the operator=() function copies its value with the statement
- `data = a.data;`
- The function also prints the Assignment operator invoked message so that we can tell when it executes.

Passing by Reference

- Notice that the argument to `operator=()` is passed by reference.
- It is not absolutely necessary to do this, but it's usually a good idea. Why?
- As you know, an argument passed by value generates a copy of itself in the function to which it is passed.
- The argument passed to the `operator=()` function is no exception.
- If such objects are large, the copies can waste a lot of memory.
- Values passed by reference don't generate copies, and thus help to conserve memory.

Passing by Reference

- Also, there are certain situations in which you want to keep track of the number of objects.
- If the compiler is generating extra objects every time you use the assignment operator, you may wind up with more objects than you expected.
- Passing by reference helps avoid such spurious object creation.

Returning a Value

- As we've seen, a function can return information to the calling program by value or by reference.
- When an object is returned by value, a new object is created and returned to the calling program.
- In the calling program, the value of this object can be assigned to a new object or it can be used in other ways.
- When an object is returned by reference, no new object is created.
- A reference to the original object in the function is all that's returned to the calling program.

Returning a Value

- The operator=() function in ASSIGN returns a value by creating a temporary alpha object and initializing it using the one-argument constructor in the statement
- `return alpha(data);`
- The value returned is a copy of, but not the same object as, the object of which the overloaded = operator is a member.
- Returning a value makes it possible to chain = operators:
- `a3 = a2 = a1;`

Returning a Value

- However, returning by value has the same disadvantages as passing an argument by value:
- It creates an extra copy that wastes memory and can cause confusion.
- Can we return this value with a reference, using the declarator shown here for the overloaded = operator?
- `alpha& operator = (alpha& a) // bad idea in this case`
- Unfortunately, we can't use reference returns on variables that are local to a function.

Returning a Value

- Remember that local (automatic) variables—that is, those created within a function (and not designated static)—are destroyed when the function returns.
- A return by reference returns only the address of the data being returned, and, for local data, this address points to data within the function.
- When the function is terminated and this data is destroyed, the pointer is left with a meaningless value.
- Your compiler may flag this usage with a warning.

Not Inherited

- The assignment operator is unique among operators in that it is not inherited.
- If you overload the assignment operator in a base class, you can't use this same function in any derived classes.



Overloading the Subscript Operator

- The subscript operator `[]` is normally used to access array elements. This operator can be overloaded to enhance the existing functionality of C++ arrays.
- The subscript operator is one of the operators that must be overloaded as a member function.
- It's a binary operator. That makes the class object the left-hand-side operand, while the right-hand-side operand is an integer (or `size_t`) subscript i.e. the index number contained within the square brackets.
- An overloaded operator `[]` function will always take one parameter: the subscript that the user places between the hard braces.
- the `[]` operator must return a reference to an array element to support modification of the array
- the `[]` operator always takes an index, so it can check whether the index is in bounds
 - In our example case, the user will pass in an integer index, and we'll return an integer value back as a result.
 - Now, whenever we use the subscript operator (`[]`) on an object of our class, the compiler will return the corresponding element address from the array! This allows us to both get and set values of array directly

```
const int SIZE = 10;
class testSubscript
{
    int myArray[SIZE];
public:
    int& operator[](const int index);
};
```

```
int& testSubscript::operator[](const int index)
{
    if(index > SIZE)
    {
        cout<<"index out of range:";
        return myArray[0];
    }
    return myArray[index];
}
```

```
int main()
{
```

```
    testSubscript test;
    for(int i=0; i< SIZE;i++)
        test[i]=i+i*2;
    for(int i=0; i< SIZE;i++)
        cout<<test[i]<<endl;
    cout<<"\n Value of test[2]:"<<test[2]<<endl;
    cout<<"\n Value of test[2]:"<<test[5]<<endl;
    cout<<"\n Value of test[2]:"<<test[20]<<endl;
    system("pause");
```

```
    return 0;
}
```

```
0
3
6
9
12
15
18
21
24
27
Value of test[2]:6
Value of test[2]:15
index out of range:
Value of test[2]:0
Press any key to continue . . .
```