Module 09: Programming in C++

Operator Overloading

Intructors: Abir Das and Sourangshu Bhattacharya

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

{*abir, sourangshu*}*@cse.iitkgp.ac.in*

Slides taken from NPTEL course on Programming in C++

by **Prof. Partha Pratim Das**

Module Objectives

- Understand the Operator Overloading

Module Outline

- Basic Differences between Operators & Functions
- Operator Overloading
- Examples of Operator Overloading
  - `operator+` for String & Enum
- Operator Overloading Rules

Operator & Function

- What is the difference between an operator & a function?

```
unsigned int Multiply(unsigned x, unsigned y) {
    int prod = 0;
    while (y-- > 0) prod += x;
    return prod;
}

int main() {
    unsigned int a = 2, b = 3;

    // Computed by '*' operator
    unsigned int c = a * b;          // c is 6

    // Computed by Multiply function
    unsigned int d = Multiply(a, b); // d is 6

    return 0;
}
```

- Same computation by an operator and a function

Difference between Operator & Functions

Operator	Function

• Usually written in **infix** notation
• Examples:
`Infix: a + b; a ? b : c;`
`Prefix: ++a;`
`Postfix: a++;`
	• Always written in **prefix** notation
	• Examples:
	`Prefix: max(a, b);`
	`qsort(int[], int, int, void (*)(void*, void*));`

• Operates on one or more operands, typically up to 3 (Unary, Binary or Ternary)	• Operates on zero or more arguments
• Produces one result	• Produces up to one result
• Order of operations is decided by precedence and associativity	• Order of application is decided by depth of nesting
• Operators are pre-defined	• Functions can be defined as needed

Operator Functions in C++

- Introduces a new keyword: `operator`
- Every operator is associated with an operator function that defines its behavior

Operator Expression	Operator Function
`a + b`	`operator+(a, b)`
`a = b`	`operator=(a, b)`
`c = a + b`	`operator=(c, operator+(a, b))`

- Operator functions are implicit for predefined operators of built-in types and cannot be redefined
- An operator function may have a signature as:

```
MyType a, b; // An enum or struct

MyType operator+(MyType, MyType); // Operator function

a + b // Calls operator+(a, b)
```

- C++ allows users to define an operator function and overload it

Program 09.01: String Concatenation

Concatenation by string functions

```
#include <iostream>
#include <cstring>
using namespace std;
typedef struct _String { char *str;
} String;
int main(){
    String fName, lName, name;
    fName.str = strdup("Partha ");
    lName.str = strdup("Das" );
    name.str = (char *) malloc(
               strlen(fName.str) +
               strlen(lName.str) + 1);
    strcpy(name.str, fName.str);
    strcat(name.str, lName.str);

    cout << "First Name: " <<
            fName.str << endl;
    cout << "Last Name: " <<
            lName.str << endl;
    cout << "Full Name: " <<
            name.str << endl;
    return 0;
}
----------
First Name: Partha
Last Name: Das
Full Name: Partha Das
```

Concatenation operator

```
#include <iostream>
#include <cstring>
using namespace std;
typedef struct _String { char *str; } String;
String operator+(const String& s1, const String& s2)
    String s;
    s.str = (char *) malloc(strlen(s1.str) +
                            strlen(s2.str) + 1);
    strcpy(s.str, s1.str);
    strcat(s.str, s2.str);
    return s;
}
int main() {
    String fName, lName, name;
    fName.str = strdup("Partha ");
    lName.str = strdup("Das");

    name = fName + lName; // Overload operator +

    cout << "First Name: " << fName.str << endl;
    cout << "Last Name: " << lName.str << endl;
    cout << "Full Name: " << name.str << endl;
    return 0;
}
----------
First Name: Partha
Last Name: Das
Full Name: Partha Das
```

Program 09.02: A new semantics for operator+

w/o Overloading +

```
#include <iostream>
using namespace std;
enum E {C0 = 0, C1 = 1, C2 = 2};



int main() {
    E a = C1, b = C2;
    int x = -1;

    x = a + b;
    cout << x << endl;

    return 0;
}
----------
3
```

- Implicitly converts enum E values to int
- Adds by operator+ of int
- Result is outside enum E range

Overloading operator +

```
#include <iostream>
using namespace std;
enum E {C0 = 0, C1 = 1, C2 = 2};

E operator+(const E& a, const E& b) {
    unsigned int uia = a, uib = b;
    unsigned int t = (uia + uib) % 3;
    return (E) t;
}
int main() {
    E a = C1, b = C2;
    int x = -1;

    x = a + b;
    cout << x << endl;

    return 0;
}
----------
0
```

- operator + is overloaded for enum E
- Result is a valid enum E value

Operator Overloading – Summary of Rules

- No new operator such as **, <>, or &| can be defined for overloading
- Intrinsic properties of the overloaded operator cannot be change
  - Preserves arity
  - Preserves precedence
  - Preserves associativity
- These operators can be overloaded:

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

- For unary prefix operators, use: `MyType& operator++(MyType& s1)`
- For unary postfix operators, use: `MyType operator++(MyType& s1, int)`
- The operators :: (scope resolution), . (member access), .* (member access through pointer to member), `sizeof`, and ?: (ternary conditional) cannot be overloaded
- The overloads of operators &&, ||, and , (comma) lose their special properties: short-circuit evaluation and sequencing
- The overload of operator-> must either return a raw pointer or return an object (by reference or by value), for which operator-> is in turn overloaded

Overloading disallowed for

`Operator`	**Reason**
• dot (.)	• The second argument is a name (of the field or member function), rather than a value
• Scope Resolution (::)	• It performs a (compile time) scope resolution rather than an expression evaluation.
• Ternary (? :)	• overloading expr1 ? expr2 : expr3 would not be able to guarantee that only one of expr2 and expr3 was executed
• sizeof	• Sizeof cannot be overloaded because built-in operations, such as incrementing a pointer into an array implicitly depends on it

Do not overload these operators

`Operator`	**Reason**
• && and ||	• In evaluation, the second operand is not evaluated if the result can be deduced solely by evaluating the first operand. However, this evaluation is not possible for overloaded versions of these operators
• Comma (,)	• This operator guarantees that the first operand is evaluated before the second operand. However, if the comma operator is overloaded, its operand evaluation depends on C++'s function parameter mechanism, which does not guarantee the order of evaluation
• Ampersand (&)	• The address of an object of incomplete type can be taken, but if the complete type of that object is a class type that declares operator &() as a member function, then the behavior is undefined

Module Summary

- Introduced operator overloading
- Explained the rules of operator overloading