Module 08: Programming C++

Default Parameters & Function Overloading

Instructors: Abir Das and Jibesh Patra

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

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

Slides taken from NPTEL course on Programming in Modern C++

by **Prof. Partha Pratim Das**

Module Objectives

- Understand default parameters
- Understand function overloading and Resolution

Module Outline

- Default parameter
  - Motivation
  - Call function with default parameter
  - Highlighted Points
  - Restrictions
- Function overloading
  - Meaning & Motivation
  - Necessity of function overloading in Contrast with C
- Static Polymorphism
  - Meaning
  - Overloading function
- Overload Resolution
- Default parameters and Function Overloading

Motivation: Example `CreateWindow` in MSDN

Declaration of CreateWindow

```
HWND WINAPI CreateWindow(
    _In_opt_ LPCTSTR   lpClassName,
    _In_opt_ LPCTSTR   lpWindowName,
    _In_     DWORD     dwStyle,
    _In_     int       x,
    _In_     int       y,
    _In_     int       nWidth,
    _In_     int       nHeight,
    _In_opt_ HWND      hWndParent,
    _In_opt_ HMENU     hMenu,
    _In_opt_ HINSTANCE hInstance,
    _In_opt_ LPVOID    lpParam
);
```

Calling CreateWindow

```
hWnd = CreateWindow(
    ClsName,
    WndName,
    WS_OVERLAPPEDWINDOW,
    CW_USEDEFAULT,
    CW_USEDEFAULT,
    CW_USEDEFAULT,
    CW_USEDEFAULT,
    NULL,
    NULL,
    hInstance,
    NULL
);
```

- There are 11 parameters in `CreateWindow()`
- Of these 11, 8 parameters (4 are `CWUSEDEFAULT`, 3 are `NULL`, and 1 is `hInstance`) usually get same values in most calls
- Instead of using these 8 fixed valued Parameters at call, we may assign the *values in formal parameter*
- C++ allows us to do so through the mechanism called **Default parameters**

Program 08.01: Function with a default parameter


```
#include <iostream>
using namespace std;

int IdentityFunction(int a = 10) { // Default value for parameter a
    return (a);
}

int main() {
    int x = 5, y;

    y = IdentityFunction(x);  // Usual function call. Actual parameter taken as x = 5
    cout << "y = " << y << endl;

    y = IdentityFunction();   // Uses default parameter. Actual parameter taken as 10
    cout << "y = " << y << endl;
}
----------
y = 5
y = 10
```

Program 08.02: Function with 2 default parameters


```
#include<iostream>
using namespace std;

int Add(int a = 10, int b = 20) {
    return (a + b);
}
int main() { int x = 5, y = 6, z;

    z = Add(x, y); // Usual function call -- a = x = 5 & b = y = 6
    cout << "Sum = " << z << endl;

    z = Add(x);    // One parameter defaulted -- a = x = 5 & b = 20
    cout << "Sum = " << z << endl;

    z = Add();     // Both parameter defaulted -- a = 10 & b = 20
    cout << "Sum = " << z << endl;
}
----------
Sum = 11
Sum = 25
Sum = 30
```

Default Parameter: Highlighted Points

- C++ allows programmer to assign default values to the function parameters
- Default values are specified while prototyping the function
- Default parameters are required while calling functions with fewer arguments or without any argument
- Better to use default value for less used parameters

Restrictions on default parameters

- *All parameters to the right of a parameter with default argument* **must have** *default arguments* (function `f` violates)
- *Default arguments* **cannot be** *re-defined* (second signature of function `g` violates)
- *All non-defaulted parameters* **needed** in a *call* (first call of `g()` violates)

```
#include <iostream>

void f(int, double = 0.0, char *);
// Error C2548: f: missing default parameter for parameter 3

void g(int, double = 0, char * = NULL); // OK
void g(int, double = 1, char * = NULL);
// Error C2572: g: redefinition of default parameter : parameter 3
// Error C2572: g: redefinition of default parameter : parameter 2

int main() {
    int i = 5; double d = 1.2; char c = 'b';

    g(); // Error C2660: g: function does not take 0 arguments
    g(i);
    g(i, d);
    g(i, d, &c);
}
```

Restrictions on default parameters

- Default parameters to be supplied *only in a header file* and *not in the definition* of a function

```
// Header file: myFunc.h
void g(int, double, char = 'a'); // Defaults ch
void g(int i, double f = 0.0, char ch); // A new overload. Defaults f & ch
void g(int i = 0, double f, char ch);   // A new overload. Defaults i, f & ch
// void g(int i = 0, double f = 0.0, char ch = 'a'); // Alternate signature. Defaults all in one go
-----------------------------------------------------
// Source File
#include <iostream>
using namespace std;
#include "myFunc.h" // Defaults taken from header
void g(int i, double d, char c) { cout << i << ' ' << d << ' ' << c << endl; } // No defaults here
-----------------------------------------------------
// Application File
#include <iostream>
#include "myFunc.h"
int main() { int i = 5; double d = 1.2; char c = 'b';
    g();          // Prints: 0 0 a
    g(i);         // Prints: 5 0 a
    g(i, d);      // Prints: 5 1.2 a
    g(i, d, c);   // Prints: 5 1.2 b
}
```

Function overloads: Matrix Multiplication in C

- *Similar functions* with *different* **data types** and **algorithms**

```
typedef struct { int data[10][10]; } Mat;    // 2D Matrix
typedef struct { int data[1][10]; }  VecRow; // Row Vector
typedef struct { int data[10][1]; }  VecCol; // Column Vector

void Multiply_M_M  (Mat a,    Mat b,    Mat* c);    // c = a * b
void Multiply_M_VC (Mat a,    VecCol b, VecCol* c); // c = a * b
void Multiply_VR_M (VecRow a, Mat b,    VecRow* c); // c = a * b
void Multiply_VC_VR(VecCol a, VecRow b, Mat* c);    // c = a * b
void Multiply_VR_VC(VecRow a, VecCol b, int* c);    // c = a * b

int main() {
    Mat m1, m2, rm; VecRow rv, rrv; VecCol cv, rcv; int r;
    Multiply_M_M  (m1, m2, &rm);  // rm  <-- m1 * m2
    Multiply_M_VC (m1, cv, &rcv); // rcv <-- m1 * cv
    Multiply_VR_M (rv, m2, &rrv); // rrv <-- rv * m2
    Multiply_VC_VR(cv, rv, &rm);  // rm  <-- cv * rv
    Multiply_VR_VC(rv, cv, &r);   // r   <-- rv * cv
    return 0;
}
```

- 5 multiplication functions share *similar functionality* but *different argument types*
- C treats them by 5 different function names. Makes it difficult for the user to remember and use
- **C++ has an elegant solution**

Function overloads: Matrix Multiplication in C++

- Functions *having the same* **name**, *similar functionality* but *different* **algorithms**, and identified by *different interfaces* **data types**

```
typedef struct { int data[10][10]; } Mat;    // 2D Matrix
typedef struct { int data[1][10]; }  VecRow; // Row Vector
typedef struct { int data[10][1]; }  VecCol; // Column Vector

void Multiply(const Mat& a,    const Mat& b,    Mat& c);    // c = a * b
void Multiply(const Mat& a,    const VecCol& b, VecCol& c); // c = a * b
void Multiply(const VecRow& a, const Mat& b,    VecRow& c); // c = a * b
void Multiply(const VecCol& a, const VecRow& b, Mat& c);    // c = a * b
void Multiply(const VecRow& a, const VecCol& b, int& c);    // c = a * b

int main() {
    Mat m1, m2, rm; VecRow rv, rrv; VecCol cv, rcv; int r;
    Multiply(m1, m2, rm);  // rm  <-- m1 * m2
    Multiply(m1, cv, rcv); // rcv <-- m1 * cv
    Multiply(rv, m2, rrv); // rrv <-- rv * m2
    Multiply(cv, rv, rm);  // rm  <-- cv * rv
    Multiply(rv, cv, r);   // r   <-- rv * cv
    return 0;
}
```

- These **5 functions** having *different argument types* are represented as *one function name* (`Multiply`) in C++
- This is called **Function Overloading** or **Static Polymorphism**

Program 08.03/04: Function Overloading

- Define *multiple functions* having the *same* **name**
- *Binding* happens at **compile time**

Same # of Parameters

```
#include <iostream>
using namespace std;
int Add(int a, int b) { return (a + b); }
double Add(double c, double d) { return (c + d); }
int main() {
    int x = 5, y = 6, z;
    z = Add(x, y); // int Add(int, int)
    cout << "int sum = " << z;

    double s = 3.5, t = 4.25, u;
    u = Add(s, t); // double Add(double, double)
    cout << "double sum = " << u << endl;
}
```

int sum = 11 double sum = 7.75

- Same `Add` function to add two `int`s or two `double`s
- Same # of parameters but *different types*

Different # of Parameters

```
#include <iostream>
using namespace std;
int Area(int a, int b)  return (a * b);
int Area(int c) { return (c * c); }
int main() {
    int x = 10, y = 12, z = 5, t;
    t = Area(x, y); // int Area(int, int)
    cout << "Area of Rectangle = " << t;

    int z = 5, u;
    u = Area(z); // int Area(int)
    cout << " Area of Square = " << u << endl;
}
```

Area of Rectangle = 12 Area of Square = 25

- Same `Area` function for *rectangle*s and for squares
- *Different number of parameters*

Program 08.05: Restrictions in Function Overloading

- Two functions having the *same signature* but *different return types* cannot be overloaded

```
#include <iostream>
using namespace std;

int    Area(int a, int b) { return (a * b); }
double Area(int a, int b) { return (a * b); }
// Error C2556: double Area(int,int): overloaded function differs only by return type
//              from int Area(int,int)
// Error C2371: Area: redefinition; different basic types

int main() {
    int x = 10, y = 12, z = 5, t;
    double f;

    t = Area(x, y);
    // Error C2568: =: unable to resolve function overload
    // Error C3861: Area: identifier not found

    cout << "Multiplication = " << t << endl;

    f = Area(y, z); // Errors C2568 and C3861 as above
    cout << "Multiplication = " << f << endl;
}
```

Function Overloading – Summary of Rules

- The *same function name* may be used in *several definitions*
- Functions with the *same name* must have *different number* of formal parameters and/or *different types* of formal parameters
- Function selection (*Overload Resolution*) is performed by the compiler
- Two functions having the same signature but *differing only in the return types* will result in a compilation error. The main reason is caller does not have to use the return value, the compiler does not know which return type is the best match
- Two functions having same parameter list but differing only in their default arguments will not compile. Changing the value of a default parameter does not change the *type* of the parameter
- Overloading allows **Static Polymorphism**
- Overload resolution is considered to be one of the areas of the language that is both complex and important. Two good resources:
  - (Intermediate) Overload Resolution Video by CopperSpice
  - (Elaborate) MSDN Article on Function Overloading

Overload Resolution

- To resolve overloaded functions with one parameter
  - Identify the set of *Candidate Functions*
  - From the set of candidate functions identify the set of *Viable Functions*
  - Select the *Best viable function* through (*Order is important*)
    - *Exact Match*
    - *Promotion*
    - *Standard type conversion*
    - *User defined type conversion*

Overload Resolution: Exact Match

- *lvalue-to-rvalue conversion*: Read the value from an object
  - Most common
  - Read more about lvalue and rvalue – internalpointers.com Article
- *Array-to-pointer conversion*

  Definitions:
  ```
  int ar[10];
  void f(int *a);
  ```
  Call:
  ```
  f(ar)
  ```
- *Function-to-pointer conversion*

  Definitions:
  ```
  typedef int (*fp) (int);
  void f(int, fp);
  int g(int);
  ```
  Call:
  ```
  f(5, g)
  ```
- *Qualification conversion*
  - Converting pointer (only) to `const` pointer
  - Converting pointer (only) to `volatile` pointer
  - Converting reference (only) to `const` reference
  - Converting reference (only) to `volatile` reference

Overload Resolution: Promotion & Conversion

- **Promotion**
  - Objects of an integral type can be converted to another wider integral type, that is, a type that can represent a larger set of values. This widening type of conversion is called *integral promotion*
  - C++ promotions are *value-preserving*, as the value after the promotion is guaranteed to be the same as the value before the promotion
  - Examples
    - `char` to `int`; `float` to `double`
    - `enum` to `int` / `short` / `unsigned int` / ...
    - `bool` to `int`

Overload Resolution: Promotion & Conversion

- **Standard Conversions**
  - *Integral conversions* between *integral types* – `char`, `short`, `int`, and `long` with or without qualifiers `signed` or `unsigned`
  - *Floating point Conversions* from *less precise floating type* to a *more precise floating type* like `float` to `double` or `double` to `long double`. Conversion can happen to a *less precise* type, if it is in a range representable by that type
  - *Conversions between integral and floating point types*: Certain expressions can cause objects of floating type to be converted to integral types, or vice versa. **May be dangerous!**
  - *Pointer Conversions*: Pointers can be converted during assignment, initialization, comparison, and other expressions
  - *Bool Conversion*: `int` to `bool` or vice versa based on the context

Example: Overload Resolution with one parameter

- In the context of a list of function prototypes:

```
int g(double);                  // F1
void f();                       // F2
void f(int);                    // F3
double h(void);                 // F4
int g(char, int);               // F5
void f(double, double = 3.4);   // F6
void h(int, double);            // F7
void f(char, char *);           // F8
```

  The call site to resolve is:

```
f(5.6);
```

- Resolution:
  - *Candidate functions* (**by name**): F2, F3, F6, F8
  - *Viable functions* (**by # of parameters**): F3, F6
  - *Best viable function* (**by type** `double` – **Exact Match**): F6

Example: Overload Resolution fails

- Consider the overloaded function signatures:

```
int fun(float a) {...}              // Function 1
int fun(float a, int b) {...}       // Function 2
int fun(float x, int y = 5) {...} // Function 3

int main() {
    float p = 4.5, t = 10.5;
    int s = 30;

    fun(p, s); // CALL - 1
    fun(t);    // CALL - 2
    return 0;
}
```

- CALL - 1: Matches Function 2 & Function 3
- CALL - 2: Matches Function 1 & Function 3
- Results in ambiguity for both calls

Overload Resolution with Multiple Arguments

- For overload resolution between functions F1 and F2:

  F1 is better than F2 if, for some argument i, F1 has a better conversion than F2, and for other arguments F1 has a conversion which is not worse than F2.

Example:

```
int fun(int,int,int);           // F1
int fun(double,double,double);  // F2
int main() {fun(5,5,2.0);}      // Ambiguous
```

The above is ambiguous because neither F1 nor F2 has a better conversion than the other.

```
int fun(int,int,double);          // F1
int fun(int,double,double);       // F2
int main() {fun(5,5,5);}          // F1 wins
```

F1 is better than F2 in the second argument and not worse in the other two arguments.

Program 08.06/07: Default Parameter & Function Overload

- Compilers deal with *default parameters* as a special case of *function overloading*
- These need to be mixed carefully

Default Parameters

```
#include <iostream>
using namespace std;
int f(int a = 1, int b = 2);


int main() {
    int x = 5, y = 6;

    f();     // a = 1, b = 2
    f(x);    // a = x = 5, b = 2
    f(x, y); // a = x = 5, b = y = 6
}
```

- `f` can have 3 possible forms of call

Function Overload

```
#include <iostream>
using namespace std;
int f();
int f(int);
int f(int, int);

int main() {
    int x = 5, y = 6;

    f();     // int f();
    f(x);    // int f(int);
    f(x, y); // int f(int, int);
}
```

- `f` can have 3 possible forms of call
- *No overload* here use *default parameters*.

Program 08.08: Default Parameter & Function Overload

- *Function overloading* can use *default parameter*
- However, *with default parameters*, the overloaded functions should *still be resolvable*

```
#include <iostream>
using namespace std;
// Overloaded Area functions
int Area(int a, int b = 10) { return (a * b); }
double Area(double c, double d) { return (c * d); }
int main() { int x = 10, y = 12, t; double z = 20.5, u = 5.0, f;
    t = Area(x);    // Binds int Area(int, int = 10)
    cout << "Area = " << t << endl; // Area = 100

    t = Area(x, y);    // Binds int Area(int, int = 10)
    cout << "Area = " << t << endl; // Area = 120

    f = Area(z, u); // Binds double Area(double, double)
    cout << "Area = " << f << endl;  // Area = 102.5

    f = Area(z); // Binds int Area(int, int = 10)
    cout << "Area = " << f << endl; // Area = 200

    // Un-resolvable between int Area(int a, int b = 10) and  double Area(double c, double d)
    f = Area(z, y); // Error: call of overloaded Area(double&, int&) is ambiguous
}
```

Program 08.09: Default Parameter & Function Overload

- Function overloading with default parameters may fail

```
#include <iostream>
using namespace std;
int f();
int f(int = 0);
int f(int, int);

int main() {
    int x = 5, y = 6;

    f();      // Error C2668: f: ambiguous call to overloaded function
              // More than one instance of overloaded function f
              // matches the argument list:
              //     function f()
              //     function f(int = 0)

    f(x);     // int f(int);
    f(x, y); // int f(int, int);

    return 0;
}
```

Module Summary

- Introduced the notion of Default parameters and discussed several examples
- Identified the necessity of function overloading
- Introduced static Polymorphism and discussed examples and restrictions
- Discussed an outline for Overload resolution
- Discussed the mix of default Parameters and function overloading

Module 09: Programming in C++

Operator Overloading

Instructors: Abir Das and Jibesh Patra

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

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

Slides taken from NPTEL course on Programming in Modern 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

- Usually written in **infix** notation - at times in **prefix** or **postfix**
- Examples:

```
// Operator in-between operands
Infix: a + b; a ? b : c;
// Operator before operands
Prefix: ++a;
// Operator after operands
Postfix: a++;
```

- Operates on one or more operands, typically up to 3 (Unary, Binary or Ternary)
- Produces *one result*
- Order of operations is decided by *precedence* and *associativity*
- Operators are pre-defined

Function

- Always written in **prefix** notation
- Examples:

```
// Operator before operands
Prefix: max(a, b);
        qsort(int[], int, int,
            void (*)(void*, void*));
```

- Operates on zero or more arguments
- Produces *up to one result*
- Order of application is decided by *depth of nesting*
- Functions can be defined as needed

Operator Functions in C++

- 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

Operator Overloading

- **Operator Overloading** (also called *ad hoc polymorphism*), is a specific case of *polymorphism*, where different operators have different implementations depending on their arguments
- Operator overloading is generally defined by a programming language, For example, in C (and in C++), for `operator/`, we have:

Integer Division	Floating Point Division
`int i = 5, j = 2;` `int k = i / j; // k = 2`	`double i = 5, j = 2;` `double k = i / j; // k = 2.5`

- C does not allow programmers to overload its operators
- C++ allows programmers to overload its operators by using operator functions

Operator Overloading: Advantages and Disadvantages

- **Advantages**:
  - Operator overloading is *syntactic sugar*, and is used because it allows programming using notation nearer to the target domain
  - It also allows user-defined types a similar level of syntactic support as types built into a language
  - It is common in scientific computing, where it allows computing representations of mathematical objects to be manipulated with the same syntax as on paper
  - For example, if we build a `Complex` type in C and `a`, `b` and `c` are variables of `Complex` type, we need to code an expression

    ```
    a + b * c
    ```

    using functions to add and multiply Complex value as

    ```
    Add(a, Multiply(b, c))
    ```

    which is clumsy and non-intuitive
  - Using operator overloading we can write the expression with operators without having to use the functions

Operator Overloading: Advantages and Disadvantages

- **Disadvantages**
  - Operator overloading allows programmers to *reassign the semantics of operators* depending on the types of their operands. For example, for `int a, b`, an expression `a << b` shifts the bits in the variable `a` left by `b`, whereas `cout << a << b` outputs values of `a` and `b` to standard output (`cout`)
  - As operator overloading allows the programmer to change the usual semantics of an operator, it is a good practice to use operator overloading with care to maintain the *Semantic Congruity*
  - With operator overloading certain rules from mathematics can be *wrongly expected* or *unintentionally assumed*. For example, the commutativity of `operator+` (that is, `a + b == b + a`) is not preserved when we overload it to mean *string concatenation* as

    `"run" + "time" = "runtime"` $\neq$ `"timerun" = "time" + "run"`

    Of course, mathematics too has such deviations as multiplication is commutative for real and complex numbers but not commutative in matrix multiplication

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( // Allocation
               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;
}
----------
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); // Allocation
    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; // Overloaded operator +
    cout << "First Name: " << fName.str << endl;
    cout << "Last Name: " << lName.str << endl;
    cout << "Full Name: " << name.str << endl;
}
----------
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; // operator + for int
    cout << x << endl;
}
----------
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) { // Overloaded operator +
    unsigned int uia = a, uib = b;
    unsigned int t = (uia + uib) % 3; // Redefined addition
    return (E) t;
}
int main() { E a = C1, b = C2;
    int x = -1;

    x = a + b; // Overloaded operator + for enum E
    cout << x << endl;
}
----------
0
```

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

Operator Overloading – Summary of Rules

- *No new operator* such as `operators**` or `operators<>` can be defined for overloading
- **Intrinsic properties** of the overloaded operator *cannot be changed*
  - 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*), `operator.` (*member access*), `operator.*` (*member access through pointer to member*), `operator sizeof`, and `operator?:` (*ternary conditional*) *cannot be overloaded*
- The overloads of `operators&&`, `operator||`, and `operator,` (*comma*) *lose their special properties*: *short-circuit evaluation* and *sequencing*

Overloading disallowed for

`operator`	**Reason**
dot (`.`)	It will raise question whether it is for *object reference* or *overloading*
Scope Resolution (`::`)	It performs a (compile time) *scope resolution* rather than an *expression evaluation*
Ternary (`?:`)	Overloading `expr1? expr2: expr3` would not guarantee that *only one of* `expr2` and `expr3` was executed
`sizeof`	Operator `sizeof` cannot be overloaded because *built-in operations*, such as incrementing a pointer into an array *implicitly depends on it*
`&&` 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