Operator Overloading

Operator Overloading

```cpp
#include <iostream>
using namespace std;
class Point{
    int x,y;
    public:
    Point(){}
    Point(int x, int y): x(x),y(y){}
    void print()
    {
            cout<<"x = "<<x<<" y  = "<<y<<endl;
    }

};

int main()
{
    Point p1(2,3), p2(4,5);
    Point p3=p2+p1;
    /* The above statemenet,
    generates compiler error*/
    p3.print();
    return 0;
}
```

```cpp
#include <iostream>
using namespace std;
class Point{
    int x,y;
    public:
    Point(){}
    Point(int x, int y): x(x),y(y){}
    Point operator +(const Point &r)
    {
            Point p;
            p.x = x+r.x;
            p.y = y+r.y;
            return p;
    }
    void print()
    {
            cout<<"x = "<<x<<" y  = "<<y<<endl;
    }

};

int main()
{
    Point p1(2,3), p2(4,5);
    Point p3=p2+p1;
    /* The above statemenet,
    indirectly calls p2.operator+(p1)*/
    p3.print();
    return 0;
}
```

Why use Operator Overloading?

- Operators like **+, -, *,** etc., are already defined for built-in types such as int, float, double, etc.
- But when working with user-defined types (like class Complex, class Fraction, or class BigInteger), these operators don't work.
- Example: Built-in types

```
int a=10;
float b=20.5, sum;
sum = a + b;
```

- Here, + works because it's predefined for int and float.

The problem with User-Defined Types

- Let’s say we have a class A, and we try to add two of its objects using +:

```
class A {
   // class definition
};

A a1, a2, a3;
a3 = a1 + a2;  //  Error! '+' not defined for class A
```

- C++ doesn't know how to add two objects of class A.
- To solve this, we overload the + operator.

Why use Operator Overloading?

- We can define the behavior of the + operator to work with objects as well.
- This concept of defining operators to work with objects and structure variables is known as operator overloading.
- The goal of operator overloading is to redefine the behavior of an operator so that it works with objects of a user-defined type, while still retaining its meaning for built-in types.
- Note: Overloading doesn't replace existing functionality — it extends it for user-defined types.
- Syntax for operator overloading
- The syntax for overloading an operator is similar to that of a function, with the addition of the operator keyword followed by the operator symbol.

```
returnType operator symbol (arguments) {
... .. ...        // function body
}
```

- Here,
  - **returnType** - the return type of the function
  - **operator** - a special keyword
  - **symbol** - the operator we want to overload (+, <, -, ++, etc.)
  - **arguments** - the arguments passed to the function

Difference between Operator Functions and Normal Functions

Feature	Operator Function	Normal Function
Syntax	Uses operator keyword	Standard function name
Invocation	Triggered by using an operator	Called explicitly by name
Purpose	Redefines behavior of operators	Performs defined actions
Example	operator+()	add()

Operator Overloading

- A process of enabling C++'s operators to work with class objects is known as operator overloading.
- One example of an overloaded operator built into C++ is <<, which is used both as the stream insertion operator and as the bitwise left-shift operator.
- Similarly, >> also is overloaded; it's used both as
  - the stream extraction operator—defined via operator overloading in the C++ Standard Library—and
  - The bitwise right-shift operator—defined as part of the C++ language.

Operator Overloading

Operators that can be overloaded

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

Operators that cannot be overloaded

.	.*	::	?:

Using the Overloaded Operators of Standard Library Class `string`

- The following program demonstrates many of class `string`'s overloaded operators and several other useful member functions, including `empty`, `substr` and `at`.
- Function `empty` determines whether a `string` is empty, function `substr` (for "substring") returns a `string` that's a portion of an existing `string` and, function `at` returns the character at a specific index in a `string` (after checking that the index is in range).

```
1  // Fig. 10.1: fig10_01.cpp
2  // Standard Library string class test program.
3  #include <iostream>
4  #include <string>
5  using namespace std;
6
7  int main() {
8     string s1{"happy"};
9     string s2{" birthday"};
10    string s3; // creates an empty string
11
12    // test overloaded equality and relational operators
13    cout << "s1 is \"" << s1 << "\"; s2 is \"" << s2
14       << "\"; s3 is \"" << s3 << '\"'
15       << "\n\nThe results of comparing s2 and s1:" << boolalpha
16       << "\ns2 == s1 yields " << (s2 == s1)
17       << "\ns2 != s1 yields " << (s2 != s1)
18       << "\ns2 >  s1 yields " << (s2 > s1)
19       << "\ns2 <  s1 yields " << (s2 < s1)
20       << "\ns2 >= s1 yields " << (s2 >= s1)
21       << "\ns2 <= s1 yields " << (s2 <= s1);
22
```

Fig. 10.1 | Standard Library `string` class test program. (Part 1 of 6.)

```
23      // test string member function empty
24      cout << "\n\nTesting s3.empty():\n";
25
26      if (s3.empty()) {
27         cout << "s3 is empty; assigning s1 to s3;\n";
28         s3 = s1; // assign s1 to s3
29         cout << "s3 is \"" << s3 << "\"";
30      }
31
32      // test overloaded string concatenation assignment operator
33      cout << "\n\ns1 += s2 yields s1 = ";
34      s1 += s2; // test overloaded concatenation
35      cout << s1;
36
37      // test string concatenation with a C string
38      cout << "\n\ns1 += \" to you\" yields\n";
39      s1 += " to you";
40      cout << "s1 = " << s1;
41
42      // test string concatenation with a C++14 string-object literal
43      cout << "\n\ns1 += \", have a great day!\" yields\n";
44      s1 += ", have a great day!"s; // s after " for string-object literal
45      cout << "s1 = " << s1 << "\n\n";
46
```

Fig. 10.1 | Standard Library `string` class test program. (Part 2 of 6.)

```
47      // test string member function substr
48      cout << "The substring of s1 starting at location 0 for\n"
49         << "14 characters, s1.substr(0, 14), is:\n"
50         << s1.substr(0, 14) << "\n\n";
51
52      // test substr "to-end-of-string" option
53      cout << "The substring of s1 starting at\n"
54         << "location 15, s1.substr(15), is:\n" << s1.substr(15) << "\n";
55
56      // test copy constructor
57      string s4{s1};
58      cout << "\ns4 = " << s4 << "\n\n";
59
60      // test overloaded copy assignment (=) operator with self-assignment
61      cout << "assigning s4 to s4\n";
62      s4 = s4;
63      cout << "s4 = " << s4;
64
65      // test using overloaded subscript operator to create lvalue
66      s1[0] = 'H';
67      s1[6] = 'B';
68      cout << "\n\ns1 after s1[0] = 'H' and s1[6] = 'B' is:\n"
69         << s1 << "\n\n";
70
```

Fig. 10.1 | Standard Library `string` class test program. (Part 3 of 6.)

```
71      // test subscript out of range with string member function "at"
72      try {
73         cout << "Attempt to assign 'd' to s1.at(100) yields:\n";
74         s1.at(100) = 'd'; // ERROR: subscript out of range
75      }
76      catch (out_of_range& ex) {
77         cout << "An exception occurred: " << ex.what() << endl;
78      }
79   }
```

```
s1 is "happy"; s2 is " birthday"; s3 is ""

The results of comparing s2 and s1:
s2 == s1 yields false
s2 != s1 yields true
s2 >  s1 yields false
s2 <  s1 yields true
s2 >= s1 yields false
s2 <= s1 yields true
```

Fig. 10.1 | Standard Library `string` class test program. (Part 4 of 6.)

```
Testing s3.empty():
s3 is empty; assigning s1 to s3;
s3 is "happy"

s1 += s2 yields s1 = happy birthday

s1 += " to you" yields
s1 = happy birthday to you

s1 += ", have a great day!" yields
s1 = happy birthday to you, have a great day!

The substring of s1 starting at location 0 for
14 characters, s1.substr(0, 14), is:
happy birthday
```

Fig. 10.1 | Standard Library `string` class test program. (Part 5 of 6.)

```
The substring of s1 starting at
location 15, s1.substr(15), is:
to you, have a great day!

s4 = happy birthday to you, have a great day!

assigning s4 to s4
s4 = happy birthday to you, have a great day!

s1 after s1[0] = 'H' and s1[6] = 'B' is:
Happy Birthday to you, have a great day!

Attempt to assign 'd' to s1.at(100) yields:
An exception occurred: invalid string position
```

Fig. 10.1 | Standard Library `string` class test program. (Part 6 of 6.)

Using the Overloaded Operators of Standard Library Class `string` (cont.)

- Class `string`'s overloaded equality and relational operators perform lexicographical comparisons (i.e., like a dictionary ordering) using the numerical values of the characters (see Appendix B, ASCII Character Set) in each `string`.
- Class `string` provides member function `empty` to determine whether a `string` is empty
  - Returns `true` if the `string` is empty; otherwise, it returns `false`.
- Class `string`'s overloaded += operator performs string concatenation.
  - A string literal can be appended to a `string` object by using operator +=

10.2 Using the Overloaded Operators of Standard Library Class `string` (cont.)

- Class `string` provides member function `substr` to return a *portion* of a string as a `string` object.
  - The call to `substr` in line 50 obtains a 14-character substring (specified by the second argument) of `s1` starting at position 0 (specified by the first argument).
  - The call to `substr` in line 54 obtains a substring starting from position 15 of `s1`.
  - When the second argument is not specified, `substr` returns the *remainder* of the `string` on which it's called.
- Class `string`'s overloaded `[]` operator can create *lvalues* that enable new characters to replace existing characters in `s1`.
  - *Class `string`'s overloaded `[]` operator does not perform any bounds checking.*

10.2 Using the Overloaded Operators of Standard Library Class `string` (cont.)

- Class `string` *does* provide bounds checking in its member function `at`, which throws an exception if its argument is an invalid subscript.
  - If the subscript is valid, function `at` returns the character at the specified location as a modifiable *lvalue* or a nonmodifiable *lvalue* (e.g., a `const` reference), depending on the context in which the call appears.

10.3 Fundamentals of Operator Overloading

- Overloaded operators provide a concise notation for manipulating string objects.
- You can use operators with your own user-defined types as well.
- Although C++ does not allow new operators to be created, it does allow most existing operators to be overloaded so that, when they're used with objects, they have meaning appropriate to those objects.

10.3.1 Operator Overloading Is Not Automatic

- Operator overloading is not automatic—you must write operator-overloading functions to perform the desired operations.
- An operator is overloaded by writing a non-static member function definition or a non-member function definition as you normally would, except that the function name starts with the keyword operator followed by the symbol for the operator being overloaded.
  - For example, the function name operator+ would be used to overload the addition operator (+) for use with objects of a particular class.
- When operators are overloaded as member functions, they must be non-static, because *they must be called on an object of the class* and operate on that object.

10.3.2 Operators That You Do Not Have to Overload

- To use an operator on class objects, you must define overloaded operator functions for that class—with three exceptions.
  - The *assignment operator (=)* may be used with *most* classes to perform *memberwise assignment* of the data members—each data member is assigned from the assignment's "source" object (on the right) to the "target" object (on the left).
  - The *address operator (&)* returns a pointer to the object; this operator also can be overloaded.
  - The *comma operator* evaluates the expression to its left then the expression to its right, and returns the value of the latter expression.

10.3.3 Operators That Cannot Be Overloaded

- Most of C++'s operators can be overloaded.
- Figure 10.2 shows the operators that cannot be overloaded.

Operators that cannot be overloaded			
.	.* (pointer to member)	::	?:

Fig. 10.2 | Operators that cannot be overloaded.

10.3.4 Rules and Restrictions on Operator Overloading

- An operator's precedence cannot be changed by overloading.
  - However, parentheses can be used to force the order of evaluation of overloaded operators in an expression.
  - Eg: (a+b)*c
- An operator's associativity cannot be changed by overloading
  - if an operator normally associates from left to right, then so do all of its overloaded versions.
- An operator's "arity" (that is, the number of operands an operator takes) cannot be changed
  - overloaded unary operators remain unary operators; overloaded binary operators remain binary operators. Operators &, *, + and - all have both unary and binary versions; these unary and binary versions can be separately overloaded.

10.3.4 Rules and Restrictions on Operator Overloading

- Only existing operators can be overloaded.
- You should not overload operators how an operator works for fundamental type values
  - For example, you should not make the + operator subtract two `int`s.
  - Operator overloading works only with objects of user-defined types or with a mixture of an object of a user-defined type and an object of a fundamental type.

10.3.4 Rules and Restrictions on Operator Overloading

- Related operators, like + and +=, must be overloaded separately.
- When overloading (), [], -> or any of the assignment operators, the operator overloading function must be declared as a class member.
  - For all other overloadable operators, the operator overloading functions can be member functions or non-member functions.

Overloading Unary operator- Using Member Function

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

class Space
{
    int x, y, z;
    public:
        void getdata(int a, int b, int c);
        void display(void);
        void operator- ();      // Overload minus operator
};

void Space :: getdata(int a, int b, int c)
{
    x = a; y = b; z = c;
}

void Space :: display(void)
{
    cout<<"x = "<<x<<" ";
    cout<<"y = "<<y<<" ";
    cout<<"z = "<<z<<" ";
}
```

```cpp
void Space :: operator- ()
{
    x = -x; y = -y; z = -z;
}

int main()
{
    Space S;
    S.getdata(10, -20, 30);
    cout<<"S : ";
    S.display();

    -S;          // activates operator- () function
    cout<<"\n-S : ";
    S.display();
    return 0;
}
```

```
S : x = 10 y = -20 z = 30
-S : x = -10 y = 20 z = -30
```

Overloading Unary operator- Using Global Function

```cpp
#include <iostream>
using namespace std;
class Number {
    int value;

public:
    Number(int v = 0) : value(v) {}

    void display() const {
        cout << "Value = " << value << endl;
    }
    // Pre-increment: ++obj
    friend Number operator++(Number &n);
    // Post-increment: obj++
    friend Number operator++(Number &n, int);
};
// Pre-increment (++obj): No dummy parameter
Number operator++(Number &n) {
    ++n.value;              // increment first
    return n;               // return updated object
}

// Post-increment (obj++): Has a dummy int parameter
Number operator++(Number &n, int) {
    Number temp = n;        // copy original
    n.value++;              // increment later
    return temp;            // return old value
}
```

```cpp
int main() {
    Number n1(5);

    cout << "Initial: ";
    n1.display();

    cout << "After Pre-increment (++n1): ";
    (++n1).display();

    cout << "After Post-increment (n1++): ";
    (n1++).display();

    cout << "Final value of n1: ";
    n1.display();

    return 0;
}
```

```
Initial: Value = 5
After Pre-increment (++n1): Value = 6
After Post-increment (n1++): Value = 6
Final value of n1: Value = 7
```

Overloading Binary operator- Using Member Function

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

class Complex
{
    float x;                    // real part
    float y;                    // imaginary part
    public:
        Complex(){ }            // constructor 1
        Complex(float real, float imag) // constructor 2
        { x = real; y = imag; }
        Complex operator+(Complex);
        void display(void);
};

Complex Complex :: operator+(Complex c)
{
    Complex temp;               // temporary
    temp.x = x + c.x;           // these are
    temp.y = y + c.y;           // float additions
    return(temp);
}
```

```
void Complex :: display(void)
{
    cout<<x<<" + j"<<y<<"\n";
}

int main()
{
    Complex c1, c2, c3;
    c1 = Complex(2.5, 3.5);
    c2 = Complex(1.6, 2.7);
    c3 = c1 + c2;

    cout<<"c1 = "; c1.display();
    cout<<"c2 = "; c2.display();
    cout<<"c3 = "; c3.display();

    return 0;
}
```

```
c1 = 2.5 + j3.5
c2 = 1.6 + j2.7
c3 = 4.1 + j6.2
```

Overloading Binary operator-Using Global Function

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

class Number {
    int value;

public:
    // Constructor
    Number(int v = 0) : value(v) {}

    // Getter
    int getValue() const {
        return value;
    }

    /* Friend declaration allows
    global operator- to access private members*/
    friend Number operator-(Number &n1, Number &n2);
};

```

```cpp
// Global operator- function
Number operator-(Number &n1, Number &n2) {
    Number n;
    n.value = n1.value-n2.value;
    return n;
}

int main() {
    Number a(20), b(8);
    Number result = a - b;   // Calls global operator-

    cout << "Result of subtraction: " << result.getValue() << endl;
    return 0;
}
```

```
Result of subtraction: 12
```

Overloading the Stream Insertion and Extraction Operators

```
1  // PhoneNumber.h
2  // PhoneNumber class definition
3  #ifndef PHONENUMBER_H
4  #define PHONENUMBER_H
5
6  #include <iostream>
7  #include <string>
8
9  class PhoneNumber {
10    friend std::ostream& operator<<(std::ostream&, const PhoneNumber&);
11    friend std::istream& operator>>(std::istream&, PhoneNumber&);
12 private:
13    std::string areaCode; // 3-digit area code
14    std::string exchange; // 3-digit exchange
15    std::string line; // 4-digit line
16 };
17
18 #endif
```

Overloading the Stream Insertion and Extraction Operators (cont)

```cpp
// PhoneNumber.cpp
// Overloaded stream insertion and stream extraction operators
// for class PhoneNumber.
#include <iomanip>
#include "PhoneNumber.h"
using namespace std;

// 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& number) {
   output << "Area code: " << number.areaCode
          << "\nExchange: "<< number.exchange
          << "\nLine: " << number.line << "\n"
          << "(" << number.areaCode << ") " << number.exchange << "-"<< number.line << "\n";
   return output; // enables cout << a << b << c;
}

// overl                    n operator; cannot be a member function
// if we                    it with cin >> somePhoneNumber;
istream& operator>>(istream& input, PhoneNumber& number) {
   input.ignore(); // skip (
   input >> setw(3) >> number.areaCode;
   input.ignore(2); // skip ) and space
   input >> setw(3) >> number.exchange; // input exchange
   input.ignore(); // skip dash (-)
   input >> setw(4) >> number.line; // input line
   return input; // enables cin >> a >> b >> c;
}
```

(901) 756-1234

skip '('

read next three "901", and store it in areaCode

Overloading the Stream Insertion and Extraction Operators (cont)

```cpp
// main.cpp
// Demonstrating class PhoneNumber's overloaded stream insertion
// and stream extraction operators.
#include <iostream>
#include "PhoneNumber.h"
using namespace std;

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

   cout << "Enter phone number in the form (555) 555-5555:" << endl;

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

   cout << "\nThe phone number entered was:\n";

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

```
Enter phone number in the form (555) 555-5555:
(901) 756-1234

The phone number entered was:
Area code: 901
Exchange: 756
Line: 1234
(901) 756-1234
```

Example: Case Study on Date class

Date: 12, 27, 2010
d1 = December 27, 2010

d1 += 7 is January 3, 2011

Date: 2, 28, 2008
d2 = February 28, 2008
++d2 = February 29, 2008 (leap year allows 29th)

d3 = July 13, 2010
++d3 = July 14, 2010
d3 = July 14, 2010
d3++ = July 14, 2010
d3 = July 15, 2010

Example: Case Study on Date class (cont..)

```
#ifndef DATE_H
#define DATE_H

#include <array>
#include <iostream>

class Date {

   friend std::ostream& operator<<(std::ostream&, const Date&);

public:
   Date(int m = 1, int d = 1, int y = 1900); // default constructor
   void setDate(int, int, int);              // set month, day, year
   Date& operator++();                       // prefix increment operator
   Date operator++(int);                     // postfix increment operator
   Date& operator+=(unsigned int);           // add days, modify object
   static bool leapYear(int);                // is year a leap year?
   bool endOfMonth(int) const;               // is day at the end of month?

private:
   unsigned int month;
   unsigned int day;
   unsigned int year;
   static const std::array<unsigned int, 13> days; // days per month
   void helpIncrement();      // utility function for incrementing date
};

#endif
```

Example: Case Study on Date class (cont..)

```cpp
//Date.cpp
#include <iostream>
#include <string>
#include "Date.h"
using namespace std;

// set month, day and year
void Date::setDate(int mm, int dd, int yy) {
    if (mm >= 1 && mm <= 12) {
        month = mm;
    }
    else {
        cout << "Invalid month! Month must be 1-12.\n";
        return; // exit the function or handle as needed
    }
    if (yy >= 1900 && yy <= 2100) {
        year = yy;
    }
    else {
        cout << "Invalid year! Year must be >= 1900 and <= 2100.\n";
        return; // exit the function or handle as needed
    }
    // test for a leap year
    if ((month == 2 && leapYear(year) && dd >= 1 && dd <= 29) ||
        (dd >= 1 && dd <= days[month])) {
        day = dd;
    }
    else {
        cout << "Invalid day! Day is out of range for current month and year.\n";
        return; // exit the function or handle as needed
    }
}
// initialize static member; one classwide copy
const array<unsigned int, 13> Date::days{
   0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

```

Example: Case Study on Date class (cont..)

```cpp
// Date constructor
Date::Date(int month, int day, int year) {
   setDate(month, day, year);
}

// overloaded prefix increment operator
Date& Date::operator++() {
   helpIncrement(); // increment date
   return *this; // reference return to create an lvalue
}

// overloaded postfix increment operator
Date Date::operator++(int) {
   Date temp{*this}; // hold current state of object
   helpIncrement();
   // return unincremented, saved, temporary object
   return temp; // value return; not a reference return
}

// add specified number of days to date
Date& Date::operator+=(unsigned int additionalDays) {
   for (unsigned int i = 0; i < additionalDays; ++i) {
      helpIncrement();
   }

   return *this; // enables cascading
}

// if the year is a leap year, return true; otherwise, return false
bool Date::leapYear(int testYear) {
   return (testYear % 400 == 0 ||
      (testYear % 100 != 0 && testYear % 4 == 0));
}
```

```cpp
// determine whether the day is the last day of the month
bool Date::endOfMonth(int testDay) const {
   if (month == 2 && leapYear(year)) {
      return testDay == 29; // last day of Feb. in leap year
   }
   else {
      return testDay == days[month];
   }
}

// function to help increment the date
void Date::helpIncrement() {
   // day is not end of month
   if (!endOfMonth(day)) {
      ++day; // increment day
   }
   else {
      if (month < 12) { // day is end of month and month < 12
         ++month; // increment month
         day = 1; // first day of new month
      }
      else { // last day of year
         ++year; // increment year
         month = 1; // first month of new year
         day = 1; // first day of new month
      }
   }
}

// overloaded output operator
ostream& operator<<(ostream& output, const Date& d) {
   static string monthName[13]{"", "January", "February",
      "March", "April", "May", "June", "July", "August",
      "September", "October", "November", "December"};
   output << monthName[d.month] << ' ' << d.day << ", " << d.year;
   return output; // enables cascading
}
```

Example: Case Study

```cpp
//main.cpp
#include <iostream>
#include "Date.h" // Date class definition
using namespace std;

int main() {
   Date d1{12, 27, 2010};
   Date d2; // defaults t

   cout << "d1 is " << d1
   cout << "\n\nd1 += 7 i

   d2.setDate(2, 28, 2008
   cout << "\n\n  d2 is "
   cout << "\n++d2 is " <

   Date d3{7, 13, 2010};

   cout << "\n\nTesting t
      << "  d3 is " << d3
   cout << "++d3 is " <<
   cout << "  d3 is " <<

   cout << "\n\nTesting the postfix increment operator:\n"
      << "  d3 is " << d3 << endl;
   cout << "d3++ is " << d3++ << endl;
   cout << "  d3 is " << d3 << endl;
}
```

```cpp
// Date constructor
Date::Date(int month, int day, int year) {
   setDate(month, day, year);
}
```

```cpp
// set month, day and year
void Date::setDate(int mm, int dd, int yy) {
   if (mm >= 1 && mm <= 12) {
      month = mm;
   }
   else {
      cout << "Invalid month! Month must be 1-12.\n";
      return; // exit the function or handle as needed
   }
   if (yy >= 1900 && yy <= 2100) {
      year = yy;
   }
   else {
      cout << "Invalid year! Year must be >= 1900 and <= 2100.\n";
      return; // exit the function or handle as needed
   }
   // test for a leap year
   if ((month == 2 && leapYear(year) && dd >= 1 && dd <= 29) ||
      (dd >= 1 && dd <= days[month])) {
      day = dd;
   }
   else {
      cout << "Invalid day! Day is out of range for current month and year.\n";
      return; // exit the function or handle as needed
   }
}
```

```
1 is December 27, 2010

(leap year allows 29th)

nt operator:

ent operator:
```

Overloading the Function Call “( )” Operator

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

class Multiply {
    int factor;
public:
    Multiply(int f) : factor(f) {}

    // Overload function call operator
    int operator()(int x) {
        return factor * x;
    }
};

int main() {
    Multiply times3(3);   // factor = 3
    Multiply times5(5);   // factor = 5

    cout << "3 * 4 = " << times3(4) << endl; // object used like a function
    cout << "5 * 7 = " << times5(7) << endl;

    return 0;
}
```

```
3 * 4 = 12
5 * 7 = 35
```

Overloading the Subscript Operator “[ ]” Operator

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

class Array {
    int arr[5];
public:
    Array() {
        for (int i = 0; i < 5; i++) arr[i] = i * 10;
    }

    // Overload subscript operator
    int& operator[](int index) {
        if (index < 0 || index >= 5) {
            throw out_of_range("Index out of range");
        }
        return arr[index]; // return reference so it can be modified
    }
};

int main() {
    Array a;

    cout << "a[2] = " << a[2] << endl;  // get value
    a[2] = 100;                         // modify value
    cout << "a[2] after update = " << a[2] << endl;

    return 0;
}
```

```
a[2] = 20
a[2] after update = 100
```

Operator as Member Function Vs Non Member Function

Thumb Rules (Not Mandatory):

- Overload operators such as [], and () as member functions because they work on the object itself and need direct access to this.
- Overload unary operators as member functions because they naturally apply to a single object (the left operand), so making that object *this is the cleanest design.
- Overload binary operators that do not modify the left operand (like +) as non-member (or friend) functions because this ensures symmetry, allows conversions on both operands, and avoids forcing the left operand to be a class type.
  i) object1+object2,
  ii) object+5
- Overload binary operators that modify the left operand but whose left operand is not your class (like ostream << obj) as non-member (or friend) functions because you cannot add members to built-in/standard classes like ostream.
  cout<<object
- Overload binary operators that modify the left operand and whose left operand is your class (like +=) as member functions because the modification clearly applies to the calling object (*this), and that makes the intent unambiguous.
  object1 += object2 (equivalent to object1 = object1+object2)

Converting Between Types

- The compiler cannot know in advance how to convert among user-defined types, and between user-defined and fundamental types.
- Such conversions can be performed using:
    i) Conversion Constructors
    ii) Conversion Operators

Conversion Constructor

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

class Complex {
    double real, imag;
public:
    // Conversion constructor (double → Complex)
    Complex(double r) : real(r), imag(0) {}
    void show() { cout << real << " + " << imag << "i" << endl; }
};

int main() {
    Complex c1 = 5.0; // double → Complex
    c1.show();
    return 0;
}

```

```
5 + 0i
```

Conversion(Cast) Operator

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

class Complex {
    double real, imag;
public:
    Complex(double r, double i) : real(r), imag(i) {}
    // Conversion operator (Complex → double)
    operator double() { return real; }
};

int main() {
    Complex c1(3.5, 2.1);
    double d = c1; // Complex → double
    cout << "Converted value = " << d << endl;
    return 0;
}
```

```
Converted value = 3.5
```

Operator Overloading Debugging Exercises-1

```cpp
#include<iostream>
using namespace std;
class A
{
    int i;
public:
    A(int ii = 0) : i(ii) {}
    void show() {  cout << i << endl;  }
};

class B
{
    int x;
public:
    B(int xx) : x(xx) {}
    operator A() {  return A(x); }
};

void g(A a)
{
    a.show();
}

int main()
{
    B b(10);
    g(b);
    g(20);
    return 0;
}
```

```
10
20
```

Operator Overloading Debugging Exercises-2

```cpp
#include<stdlib.h>
#include<stdio.h>
#include<iostream>

using namespace std;

class Test {
    int x;
public:
    void* operator new(size_t size);
    void operator delete(void*);
    Test(int i) {
        x = i;
        cout << "Constructor called \n";
    }
    ~Test() { cout << "Destructor called \n"; }
};


void* Test::operator new(size_t size)
{
    void *storage = malloc(size);
    cout << "new called \n";
    return storage;
}

```

```cpp
void Test::operator delete(void *p )
{
    cout<<"delete called \n";
    free(p);
}

int main()
{
    Test *m = new Test(5);
    m->~Test();
    delete m;
    return 0;
}
```

```
new called
Constructor called
Destructor called
Destructor called
delete called
```

Operator Overloading Debugging Exercises-3

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

class Point {
private:
  int x, y;
public:
  Point(int a, int b) : x(a), y(b) { }

  Point& operator()(int dx, int dy);

  void show() {cout << "x = " << x << ", y = " << y<<endl; }
};

Point& Point::operator()(int dx, int dy)
{
    x = dx;
    y = dy;
    return *this;
}

int main()
{
  Point pt(4,5);
  pt(5,6);
  pt.show();
  pt.show();
  return 0;
}
```

```
x = 5, y = 6
x = 5, y = 6
```