OOP/Computer Programming

By: Dr. Danish Shehzad

Can Constructor Be Private? Example:

```
// class A
class A{
private:
            A(){
            cout << "constructor of A\n";
            }
         friend class B;
};
// class B, friend of class A
class B{
public:
            B(){
                        A a1;
                        cout << "constructor of B\n";
            }
};
// Driver program
int main(){
            B b1;
            return 0;
}
```

```
test1.cpp: In constructor ‘B::B()’: test1.cpp:9:5: error:
 ‘A::A()’ is private A(){ ^ test1.cpp:19:11: error:
 within this context A a1;
```

Can Destructor be Private?
Example 1:

```
// CPP program to illustrate
// Private Destructor
#include <iostream>
using namespace std;

class Test {
private:
        ~Test() {}
};
int main()
{
}
```

Example 2:

```
// CPP program to illustrate
// Private Destructor
#include <iostream>
using namespace std;

class Test {
private:
        ~Test() {}
};
int main()
{
        Test t;
}
```

Example 3

```
// CPP program to illustrate
// Private Destructor
#include <iostream>
using namespace std;

class Test {
private:
        ~Test() {}
};
int main()
{
        Test *t;
}
```

EXAMPLE 4

```
// CPP program to illustrate
// Private Destructor

#include <iostream>
using namespace std;

class Test {
private:
        ~Test() {}
};
int main()
{
        Test* t = new Test;
}
```

Example 5

```
// CPP program to illustrate
// Private Destructor
#include <iostream>
using namespace std;

class Test {
private:
        ~Test() {}
};
int main()
{
        Test* t = new Test;
        delete t;
}
```

EXAMPLE 6:

Following is a way to **create classes with private destructors and have a function as friend of the class.** The function can only delete the objects

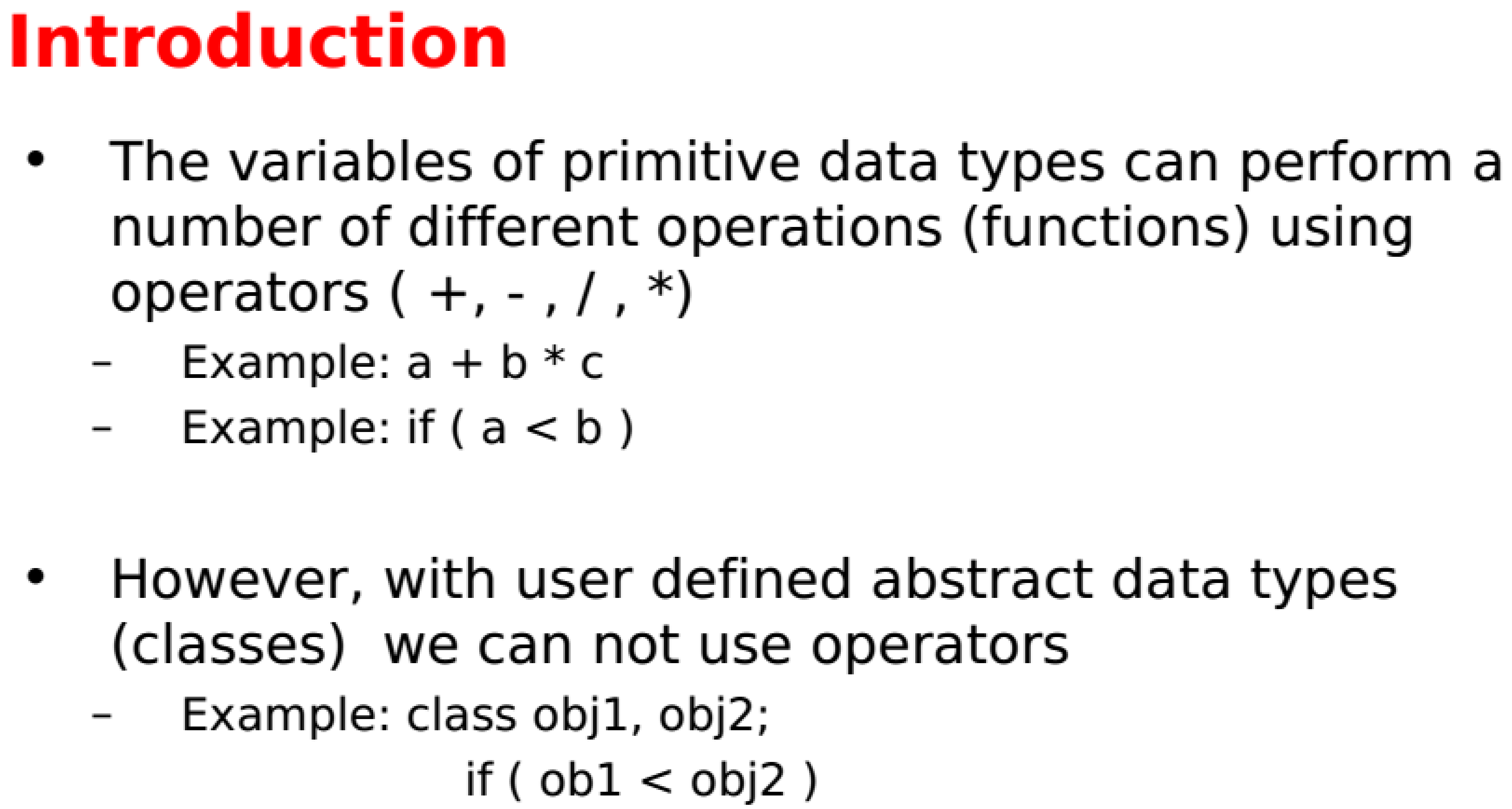
```
// A class with private destuctor
class Test {
private:
		~Test() {}
		friend void destructTest(Test*);
};

// Only this function can destruct objects of Test
void destructTest(Test* ptr)
{
		delete ptr;
}
int main()
{
		// create an object
		Test* ptr = new Test;

		// destruct the object
		destructTest(ptr);

		return 0;
}
```

Operator Overloading



Introduction

- The variables of primitive data types can perform a number of different operations (functions) using operators (+, - , / , *)
  - Example: a + b * c
  - Example: if (a < b)

- However, with user defined abstract data types (classes) we can not use operators
  - Example: class obj1, obj2;

    if (ob1 < obj2)

Introduction

- We want abstract data types to behave like primitive (native) data types

```
class vector { /* … */ };
vector a(2,1), b(3,0);
…
cout << a + b << a*b << -b;
cout << (a != b);
```

Introduction

- With the help of operator overloading we can add operator functionality in the class’s objects

- However, before using any kind of operator we need to implement its functionality in the class

Operator Overloading

- In order to add operator functionality in the class

- First create a function for the class
- Set the name of the function with the operator name
  - **operator+** for the addition operator ‘+’
  - **operator>** for the comparison operator ‘>’

Operator Overloading

- If the mathematical expression is big:
  - Converting it to C++ code will involve complicated mixture of function calls
  - Less readable
  - Chances of human mistakes are very high
  - Code produced is very hard to maintain

OPERATOR OVERLOADING

- C++ provides a very elegant solution:

“***Operator overloading***”

- C++ allows you to overload common operators like **+**, **–** or ***** etc…
- Mathematical statements don’t have to be explicitly converted into function calls

Operator Overloading

- C++ automatically overloads operators for pre-defined types
- Example of predefined types:
  - **`int`**
  - **`float`**
  - **`double`**
  - **`char`**
  - **`long`**

Function Overloading

- An overloaded function is one which has the same name but several different forms.

- For example, we overloaded the constructor for the Date class
  - default Date d;
  - initializing Date d(9,22,99);
  - copy Date d1(d);
  - other Date d(“Sept”,22,1999);

Operator Overloading

- Just as there may be many versions of a function due to overloading, there may be many versions of operators.

- Example: operator/
  - a = 14.0 / 2;   (one float and one int argument)
  - a = 14.0 / 2.0; (two float arguments)
  - a = 14 / 2;   (two ints, INTEGER DIVISION)

- Many operators are already overloaded for us to handle a variety of argument types.

Implementing Overloaded Operators

- The compiler uses the types of arguments to choose the appropriate overloading.

```
int v1, v2; v1 + v2; // int +
float s1, s2; s1 + s2; // flot+
```

Operator overloading

The compiler probably calls the correct overloaded low level function for addition i.e:

```
// for integer addition:

Add(int a, int b)


// for float addition:

Add(float a, float b)
```

Operator overloading

- Operator functions are not usually called directly
- They are automatically invoked to evaluate the operations they implement

Operator Overloading

- Operator Overloading does not allow us to alter the meaning of operators when applied to built-in types
  - one of the operands must be an object of a class
- Operator Overloading does not allow us to define new operator symbols
  - we overload those provided for in the language to have meaning for a new type of data…and there are very specific rules!

Operator Overloading

- It is similar to overloading functions
  - except the function name is replaced by the keyword operator followed by the operator's symbol
  - the arguments represent the 1 or 2 operands expected by the operator

OPERATOR OVERLOADING

- We cannot change the….
  - number of operands an operator expects
  - precedence and associativity of operators
  - or use default arguments with operators

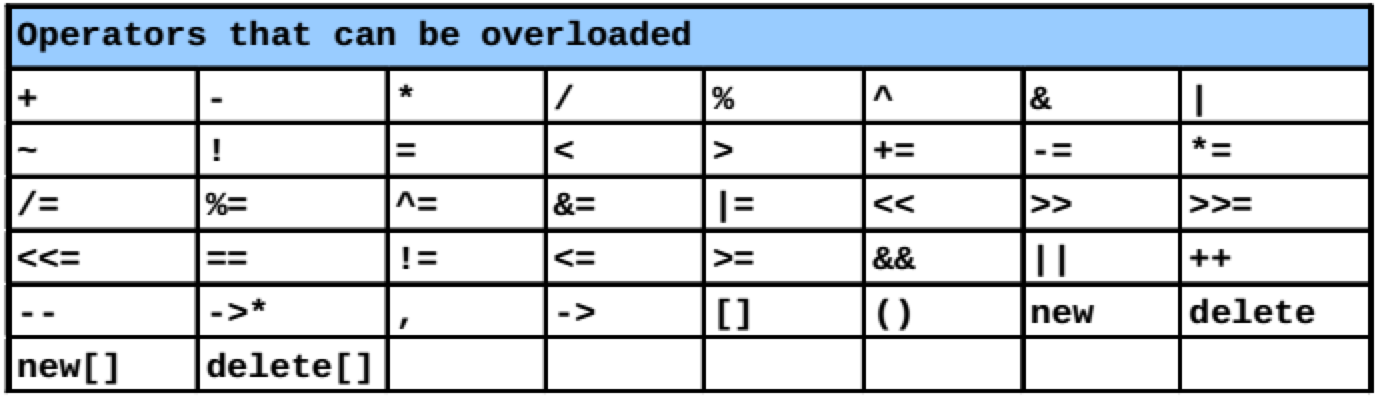
Introduction

- Operator overloading
  - Enabling C++'s operators to work with class objects
  - Using traditional operators with user-defined objects
  - Requires great care; when overloading is misused, program difficult to understand
  - Examples of already overloaded operators
    - Operator **<<** is both the stream-insertion operator
    - **+** and **-**, perform arithmetic on multiple types
  - Compiler generates the appropriate code based on the manner in which the operator is used

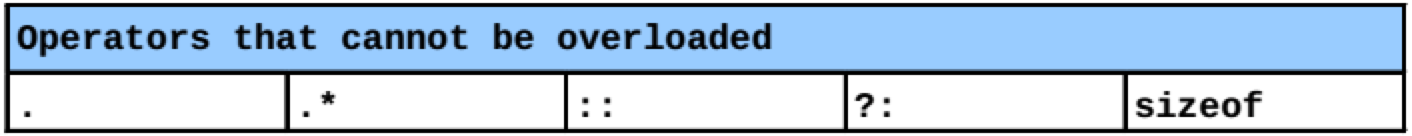
INTRODUCTION

- Overloading an operator
  - Write function definition as normal
  - Function name is keyword **`operator`** followed by the symbol for the operator being overloaded
  - **`operator+`** used to overload the addition operator (**+**)

Restriction on Operator Overloading



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



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

Operator overloading

- The precedence of an operator is **NOT** affected due to overloading
- Example:
  - **`c1*c2+c3`**
  - **`c3+c2*c1`**
  - both yield the same answer

Operator overloading

- Associativity is **NOT** changed due to overloading
- Following arithmetic expression always is evaluated from left to right:
  - **`c1 + c2 + c3 + c4`**

Operator overloading

- Unary operators and assignment operator are right associative, e.g:
  - **`a=b=c`** is same as **`a=(b=c)`**
- All other operators are left associative:
  - **`c1+c2+c3`** is same as
  - **`(c1+c2)+c3`**

Operator overloading

- Always write code representing the operator
- Example:
  - Adding subtraction code inside the + operator will create chaos

Operator overloading

- Creating a new operator is a syntax error (whether unary, binary or ternary)
- You cannot create $

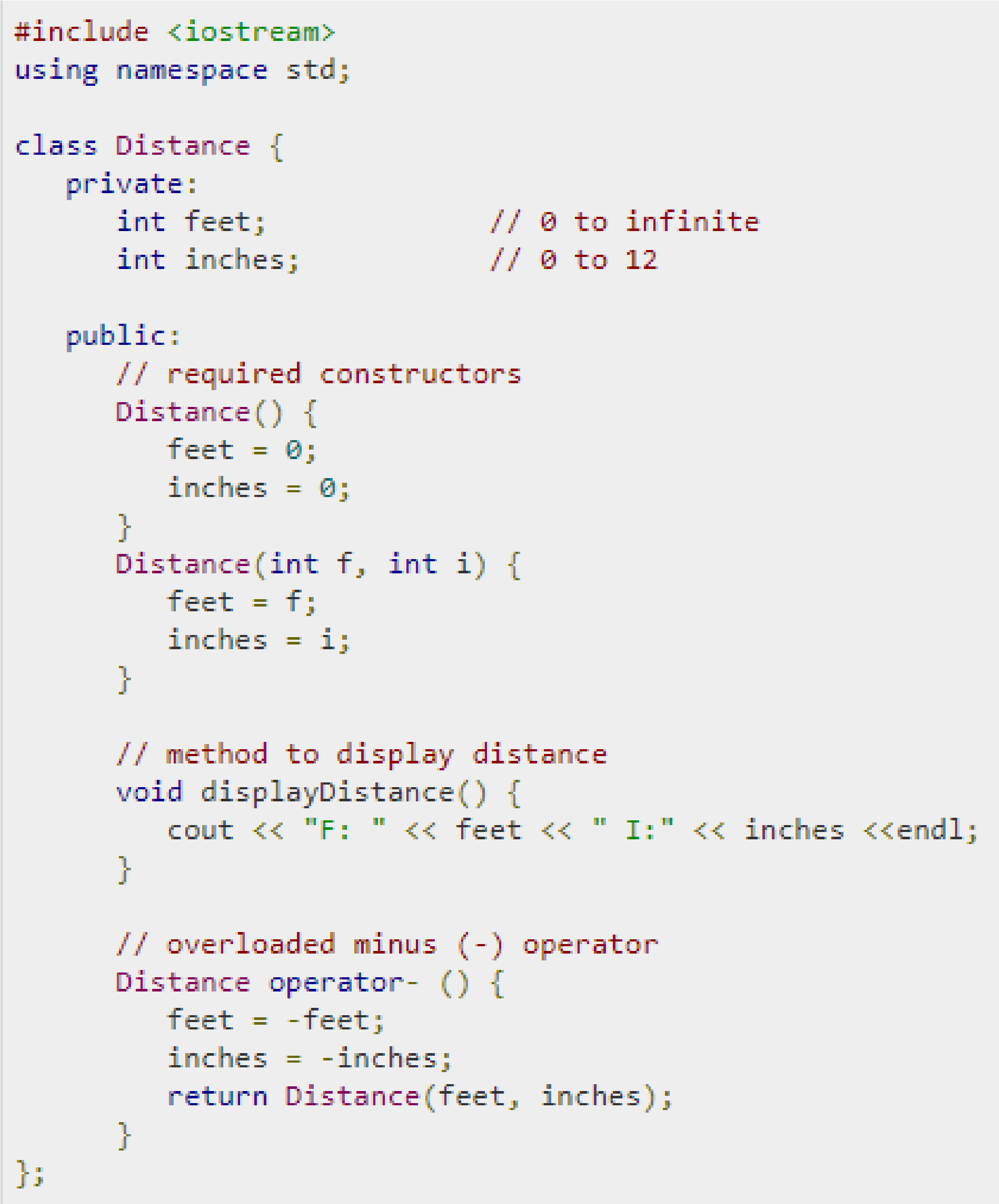
Unary Operators

- ++ (Increment Operator)
- — (Decrement operator)
- – (Unary Minus operator)
- ! (NOT Operator)

etc.

Syntax

- Return_Type classname :: operator op(Argument list)
- {
- Function Body
- }



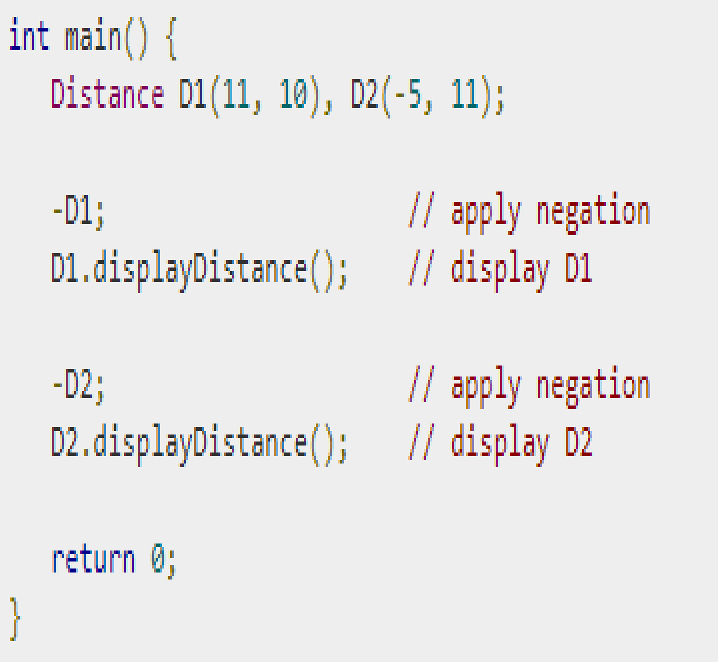
```
#include <iostream>
using namespace std;

class Distance {
   private:
      int feet;             // 0 to infinite
      int inches;           // 0 to 12

   public:
      // required constructors
      Distance() {
         feet = 0;
         inches = 0;
      }
      Distance(int f, int i) {
         feet = f;
         inches = i;
      }

      // method to display distance
      void displayDistance() {
         cout << "F: " << feet << " I:" << inches <<endl;
      }

      // overloaded minus (-) operator
      Distance operator- () {
         feet = -feet;
         inches = -inches;
         return Distance(feet, inches);
      }
};
```

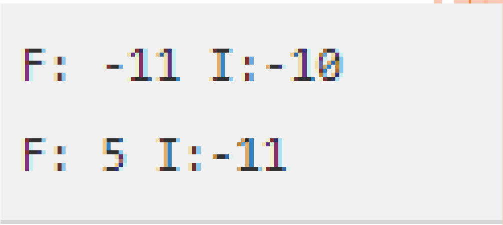


```
int main() {
   Distance D1(11, 10), D2(-5, 11);

   -D1;                     // apply negation
   D1.displayDistance();    // display D1

   -D2;                     // apply negation
   D2.displayDistance();    // display D2

   return 0;
}
```



```
F: -11 I:-10
F: 5 I:-11
```

ACTIITY

- Write a C++ program that overload as operator + as member function of class.to add real and imaginary parts of complex number. The real and imaginary data should defined private.