Unit 4 Operator overloading

Introduction

- C++ provides a rich collection of operators(e.g., +, -, /, *….etc). Such operators are used with the fundamental types(int, float, char,…..etc).
  - E.g., int a, b, c;   a= b+c is perfectly valid
- But if we try to use these operators directly with user defined types (e.g., objects) compiler will produce error.
  - E.g., if a, b, c are objects of user defined class then the statement a = b+c is not valid
- So to use operators with objects we need to overload operators (i.e., Customizes the C++ operators for operands of user-defined types.).
  - E,g, a= b+c can be made valid by using overloading.

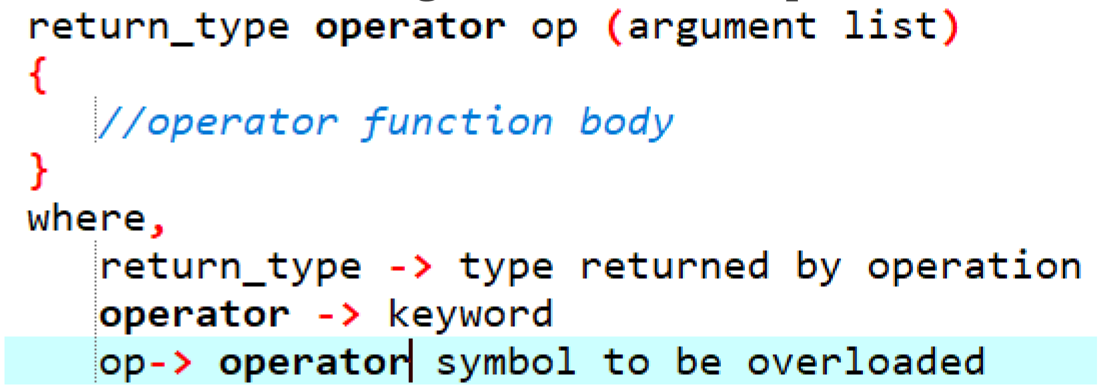
Contd..

- **What is operator overloading?**
  - The mechanism of giving special meanings to an operator is called operator overloading.
  - i.e., redefine the way operator behaves so that we can use them with user-defined types.
  - Although the semantics of an operator can be extended, we can not change its syntax and semantics that govern its use such as the number of operand, precedence and associativity.
    - E.g., For example, the operator + should be overloaded for addition, not subtraction.

Contd..

- Syntax:

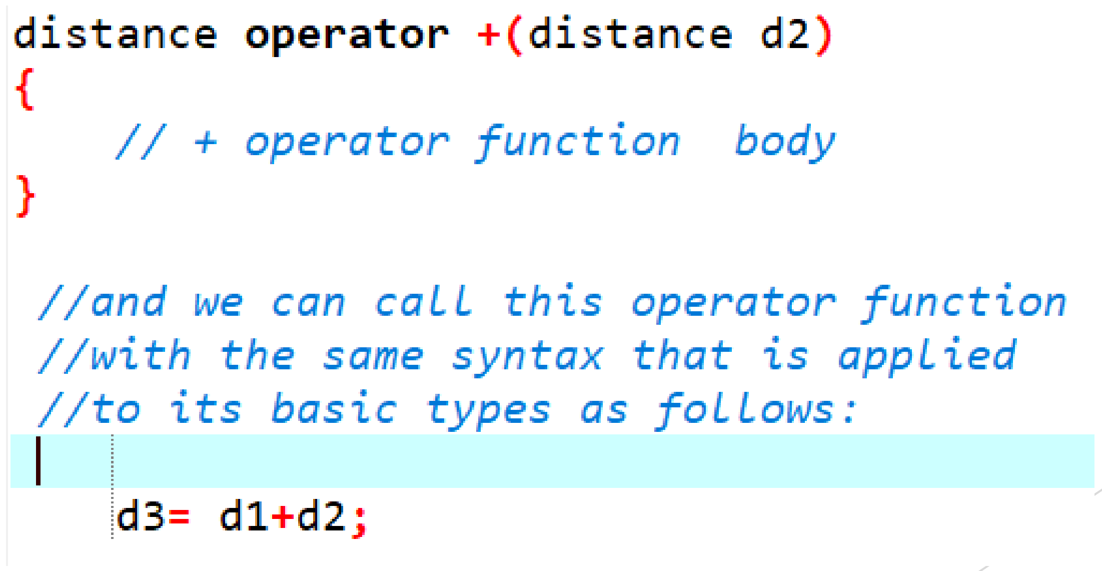
- operator overloading is done with the help of special function, called the operator function. The general form of operator function is as follows:



```
return_type operator op (argument list)
{
    //operator function body
}
where,
    return_type -> type returned by operation
    operator -> keyword
    op-> operator symbol to be overloaded
```

Contd..

- Example:
  - To add two objects of type distance each having a data members feet of type int and inches of type float, we can over load + operator as follows:



```
distance operator +(distance d2)
{
    // + operator function  body
}

 //and we can call this operator function
 //with the same syntax that is applied
 //to its basic types as follows:

    d3= d1+d2;
```

General rules(Restrictions) for operator overloading

- We can not invent new operators that do not exist in the language.
- i.e., only existing operator in the language can be overloaded.
  - E.g., @ is not a valid c++ operator and can not be overloaded

Contd..

- Overloaded operator must follow the same syntax:
  - E.g., if x and y are objects of a user defined class then
    - X++12; // not allowed: ++ is not a binary operator
    - X++; is allowed
    - %x ; is not allowed: % is not an unary operator
    - Y= x%3; is allowed
- One can not change the number of operand used by the operator.
- Try to overload operator where they make sense.

Contd..

- We can not change the way an operator works between native c++ types.
  - E.g ., we can not change how works between a double and an integer variable.
  - But we can change the working of operator for user defined types.

Contd..

- We can not change the operator precedence rules.
- i.e., the order of operator evaluation remains unchanged.

Contd..

- We can not change the operator associativity rules.
  - E.g., the unary minus(-) operator associates from right to left.

    x= -y;// unary minus associates from right to left

x = y-; // we can not alter unary minus to go from left to right.

Contd..

- Overloading some operators does not implicitly define other operators for us.
  - E.g., overloading the + and = operators for a class does not mean that the += operator is overloaded automatically for us. We will need to write overload code for the += operator as well.

Contd..

- The overloaded operator must have at least one operand that is of user-defined type.

- When overloading an operator using a member function:
    - The leftmost operand of the overloaded operator must be an object of the class type.
    - The leftmost operand becomes the implicit parameter. All other operands become function parameter.

Contd..

- Almost any operator can be overloaded in C++. However there are few operator which can not be overloaded. **Operator that are not overloaded** are follows
  - scope operator (::)
  - sizeof
  - member selector –(.)
  - member pointer selector (.*)
  - ternary operator (?:)

Overloading Unary operator

- The operators which acts upon only one operand are called unary operators.

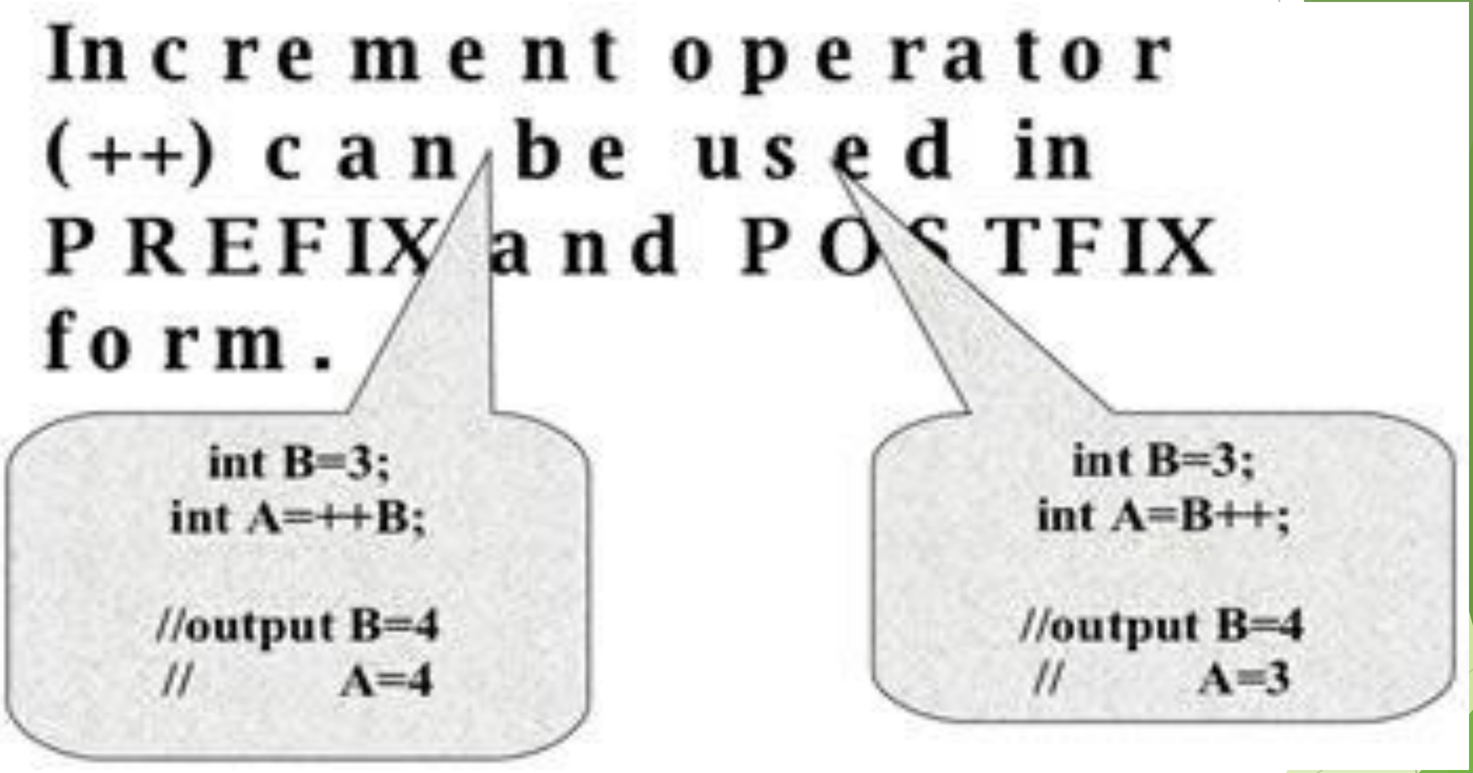
- The following is a list of unary operator that can be overloaded in c++:

  - Unary minus(-)
  - Unary plus(+)
  - Ones complement(~)
  - Pointer dereference(*)
  - Not (!)
  - Increment operator(++)
  - Decrement operator(--)

Contd..

- **Prefix(++, --) and postfix (++. --)unary operator overloading:**
  - Increment operators increase the value of operand by 1, while decrement operators(--) decrease value of operand by 1.
  - They can be written either before the operand or after it. Depending on its location, they can be classified as either prefix operators or postfix operators.
  - If the increment and decrement operators are written before the operand, then they are termed as prefix operators. However, if they are written after the operand, then they are termed as postfix operators.

Contd..



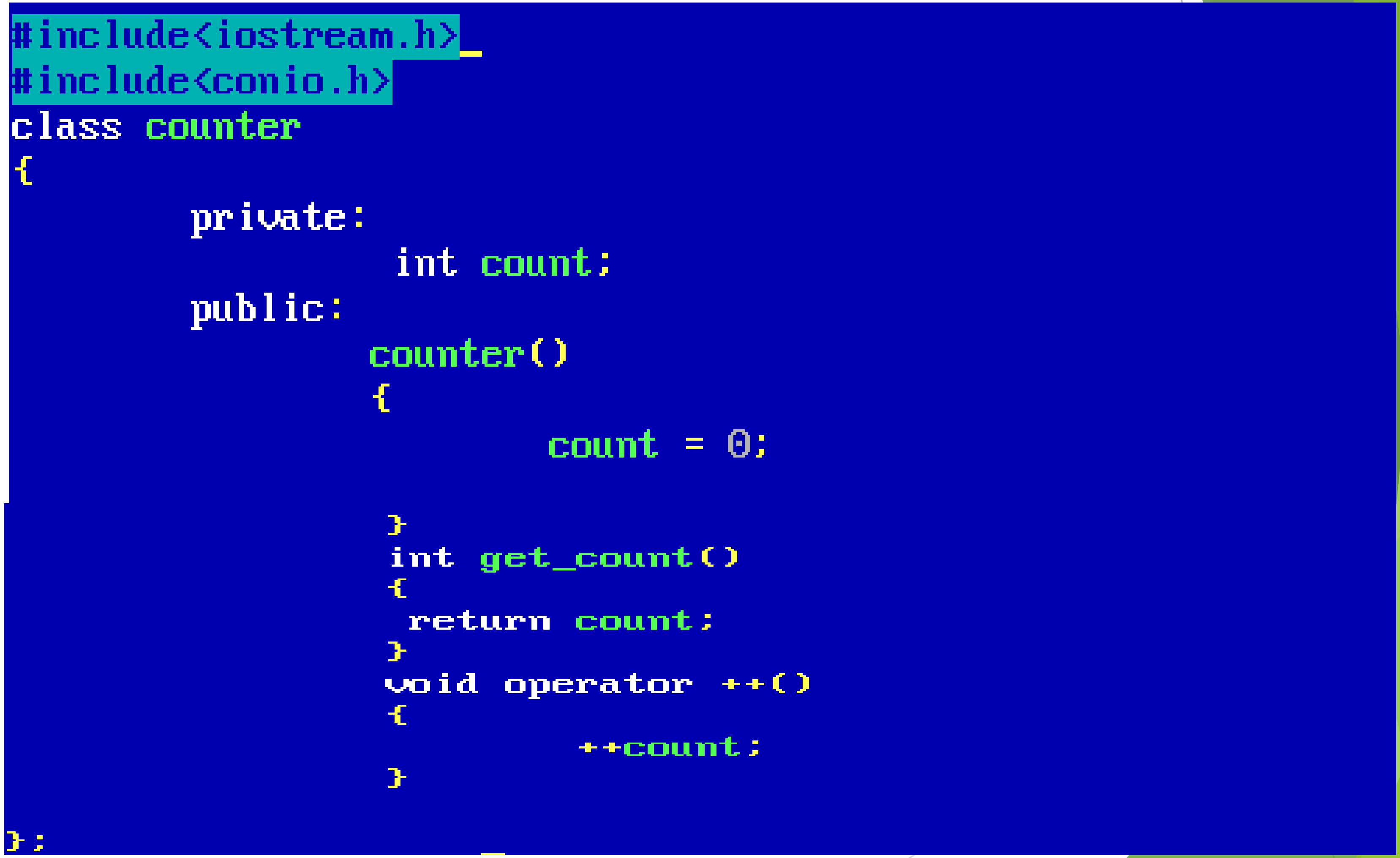
Increment operator (++) can be used in PREFIX and POSTFIX form.

```
int B=3;
int A=++B;

//output B=4
//       A=4
```

```
int B=3;
int A=B++;

//output B=4
//       A=3
```

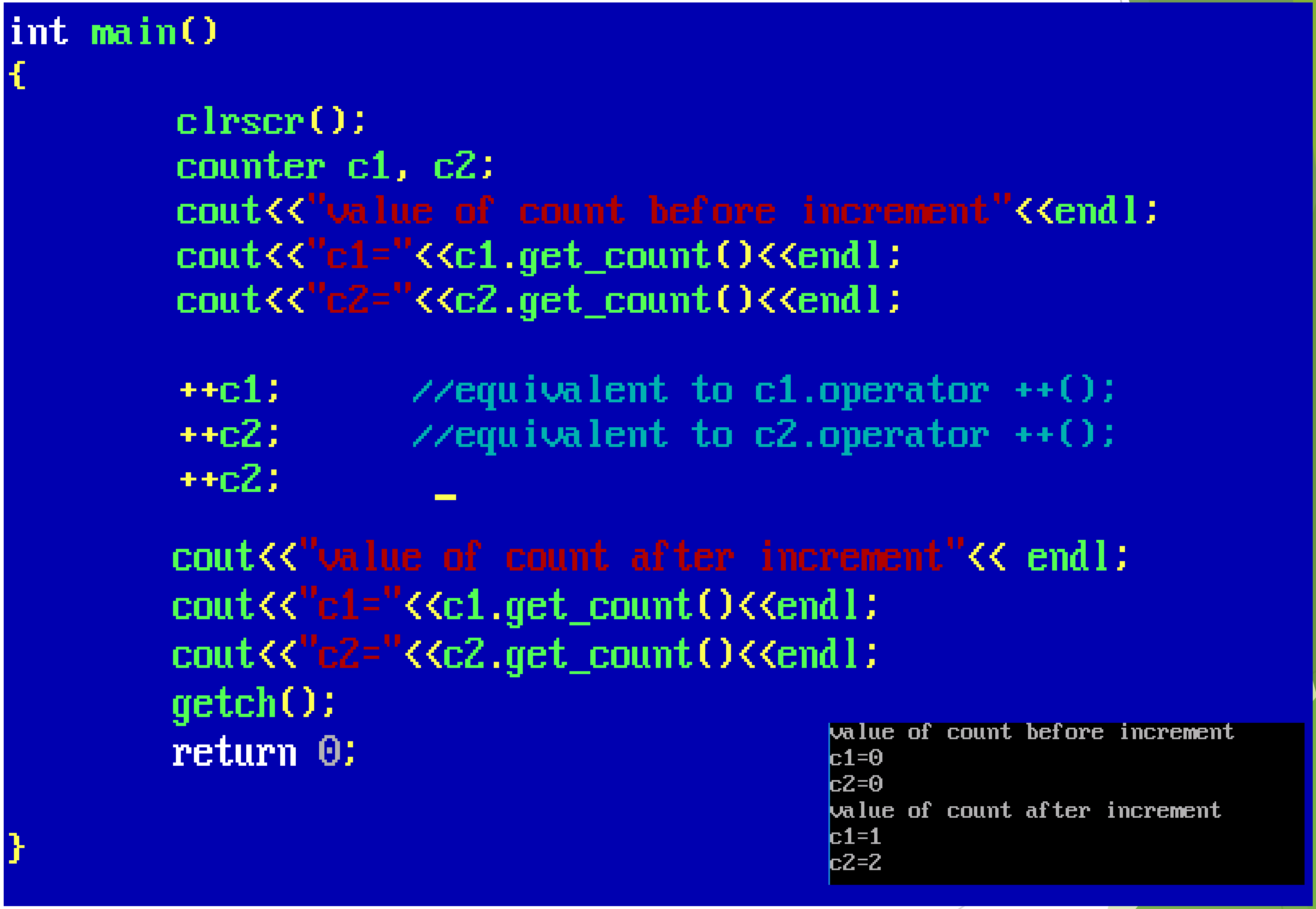
Contd..

- Overloading prefix increment operator:



```
#include<iostream.h>
#include<conio.h>
class counter
{
        private:
                 int count;
        public:
                counter()
                {
                        count = 0;

                 }
                 int get_count()
                 {
                  return count;
                 }
                 void operator ++()
                 {
                          ++count;
                 }

};
```

Contd..

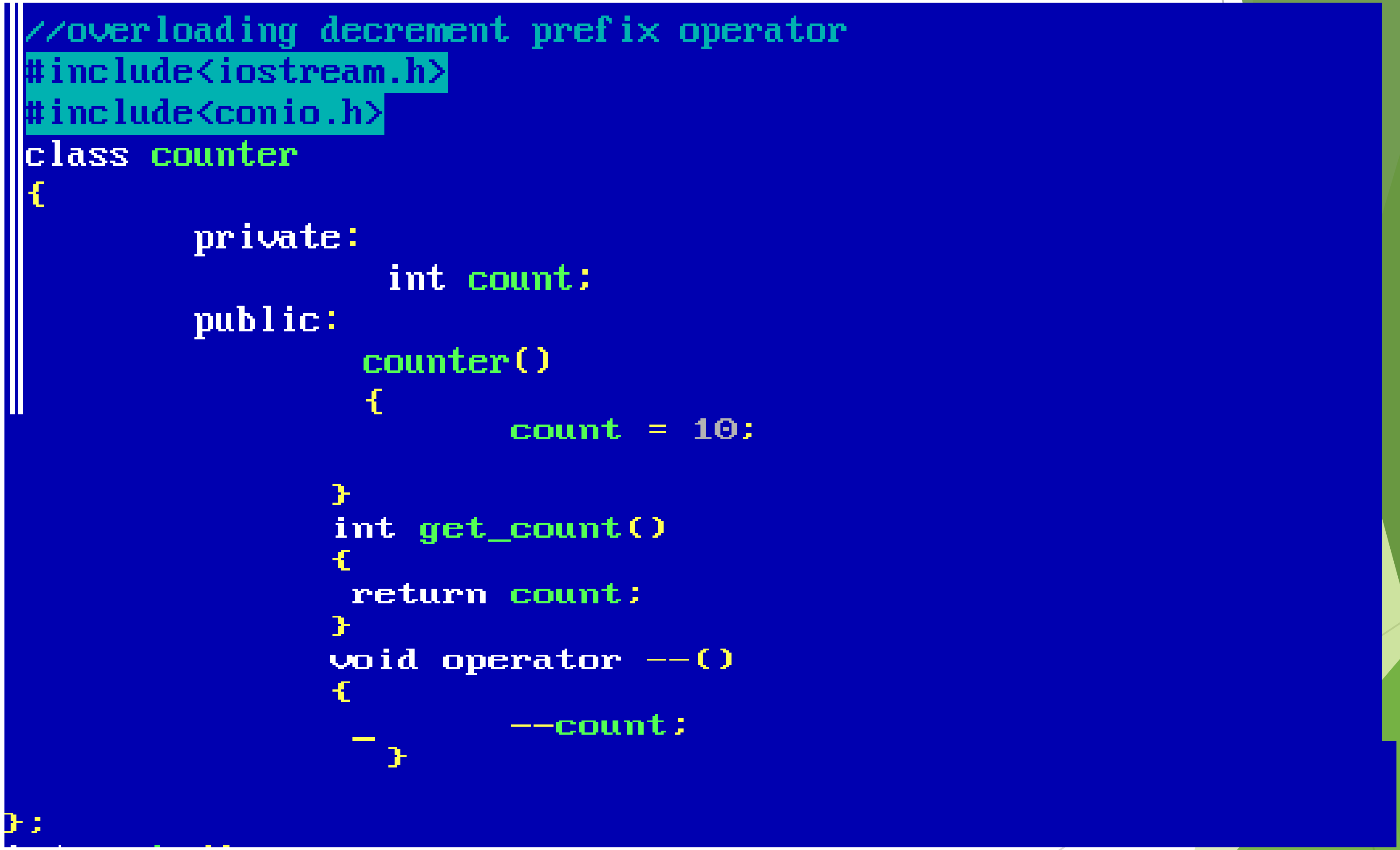


```
int main()
{
        clrscr();
        counter c1, c2;
        cout<<"value of count before increment"<<endl;
        cout<<"c1="<<c1.get_count()<<endl;
        cout<<"c2="<<c2.get_count()<<endl;

        ++c1;       //equivalent to c1.operator ++();
        ++c2;       //equivalent to c2.operator ++();
        ++c2;

        cout<<"value of count after increment"<< endl;
        cout<<"c1="<<c1.get_count()<<endl;
        cout<<"c2="<<c2.get_count()<<endl;
        getch();
        return 0;

}
```

```
value of count before increment
c1=0
c2=0
value of count after increment
c1=1
c2=2
```

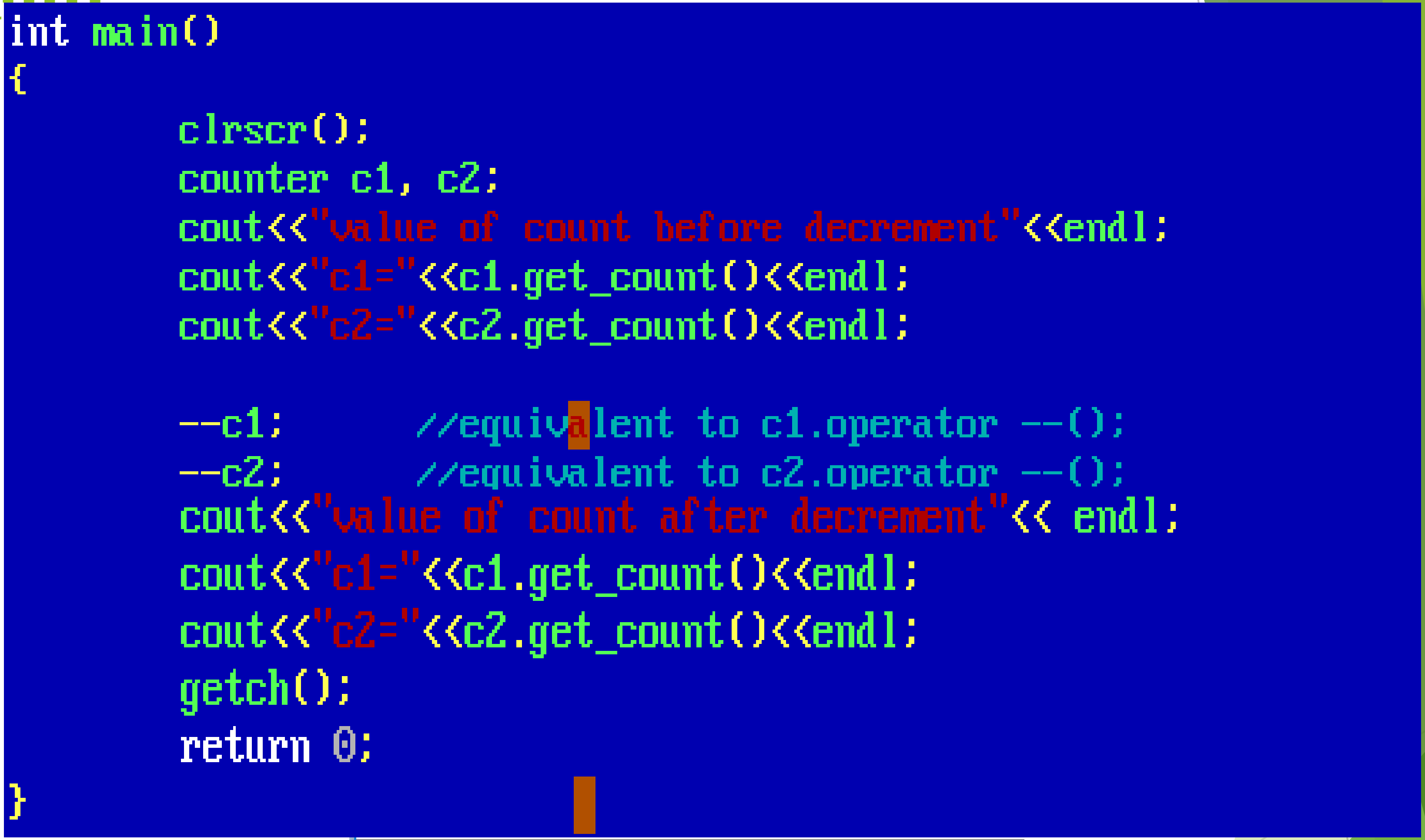
Contd..

- Overloading prefix decrement operator:

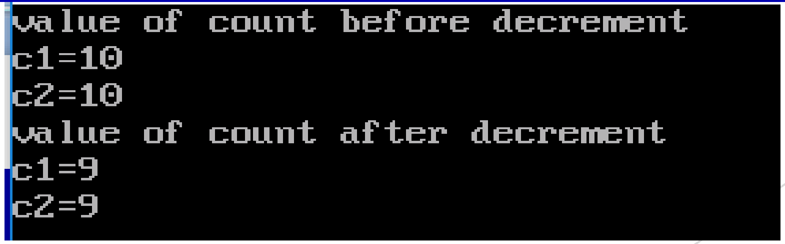


```
//overloading decrement prefix operator
#include<iostream.h>
#include<conio.h>
class counter
{
        private:
                 int count;
        public:
                counter()
                {
                       count = 10;

               }
               int get_count()
               {
                return count;
               }
               void operator --()
               {
                _       --count;
                  }

};
```

Contd



```
int main()
{
        clrscr();
        counter c1, c2;
        cout<<"value of count before decrement"<<endl;
        cout<<"c1="<<c1.get_count()<<endl;
        cout<<"c2="<<c2.get_count()<<endl;

        --c1;       //equivalent to c1.operator --();
        --c2;       //equivalent to c2.operator --();
        cout<<"value of count after decrement"<< endl;
        cout<<"c1="<<c1.get_count()<<endl;
        cout<<"c2="<<c2.get_count()<<endl;
        getch();
        return 0;
}
```



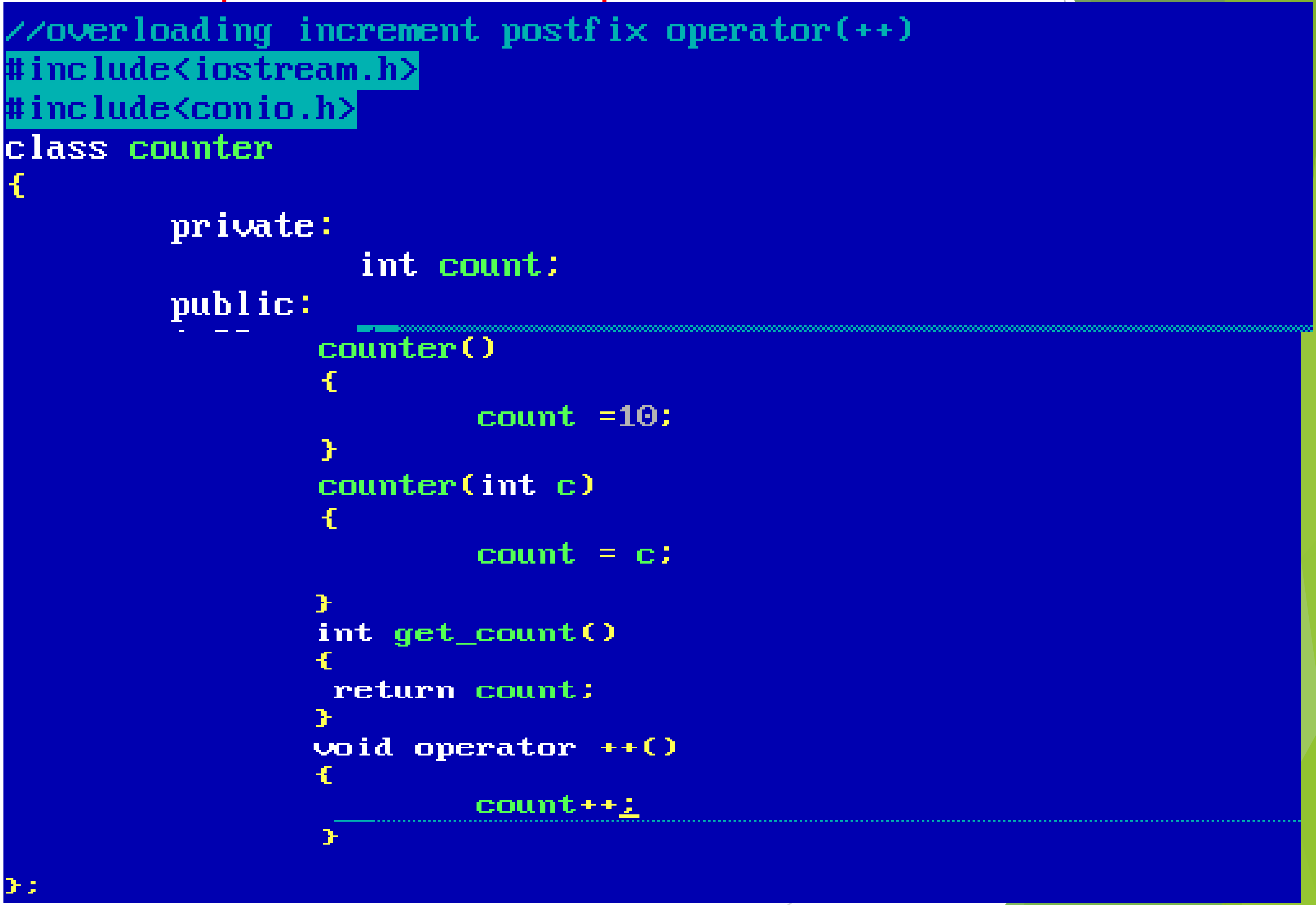
```
value of count before decrement
c1=10
c2=10
value of count after decrement
c1=9
c2=9
```

Contd..

- Class work: WAP to overload prefix operator as shown above use parameterized constructor to initialize the value of data member count.

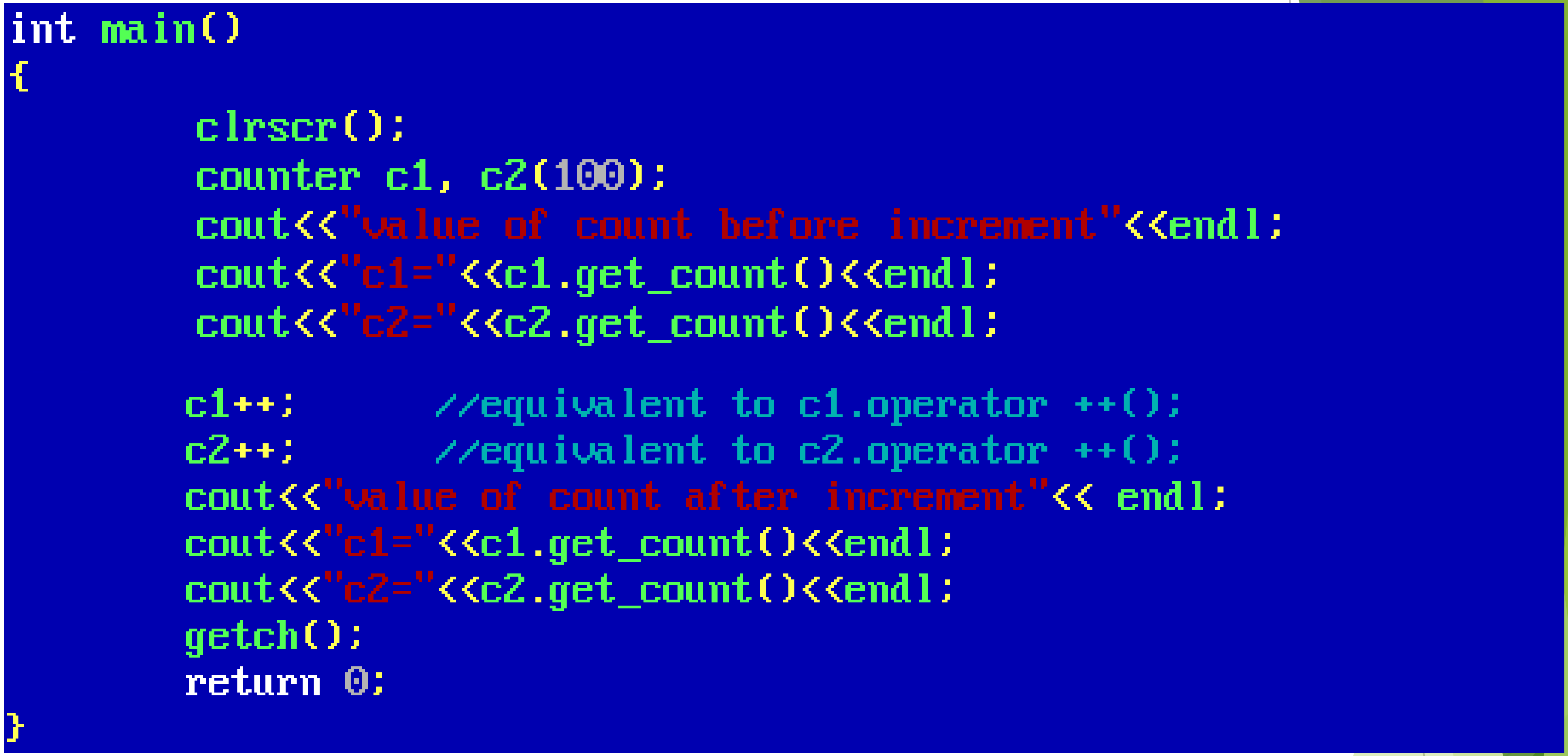
Contd..

- Overload postfix increment operator:

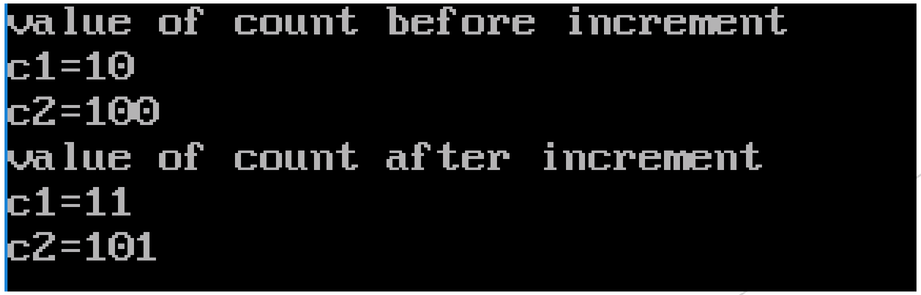


```
//overloading increment postfix operator(++)
#include<iostream.h>
#include<conio.h>
class counter
{
        private:
                 int count;
        public:
               counter()
               {
                       count =10;
               }
               counter(int c)
               {
                       count = c;

               }
               int get_count()
               {
                return count;
               }
               void operator ++()
               {
                       count++;
               }

};
```

Contd..



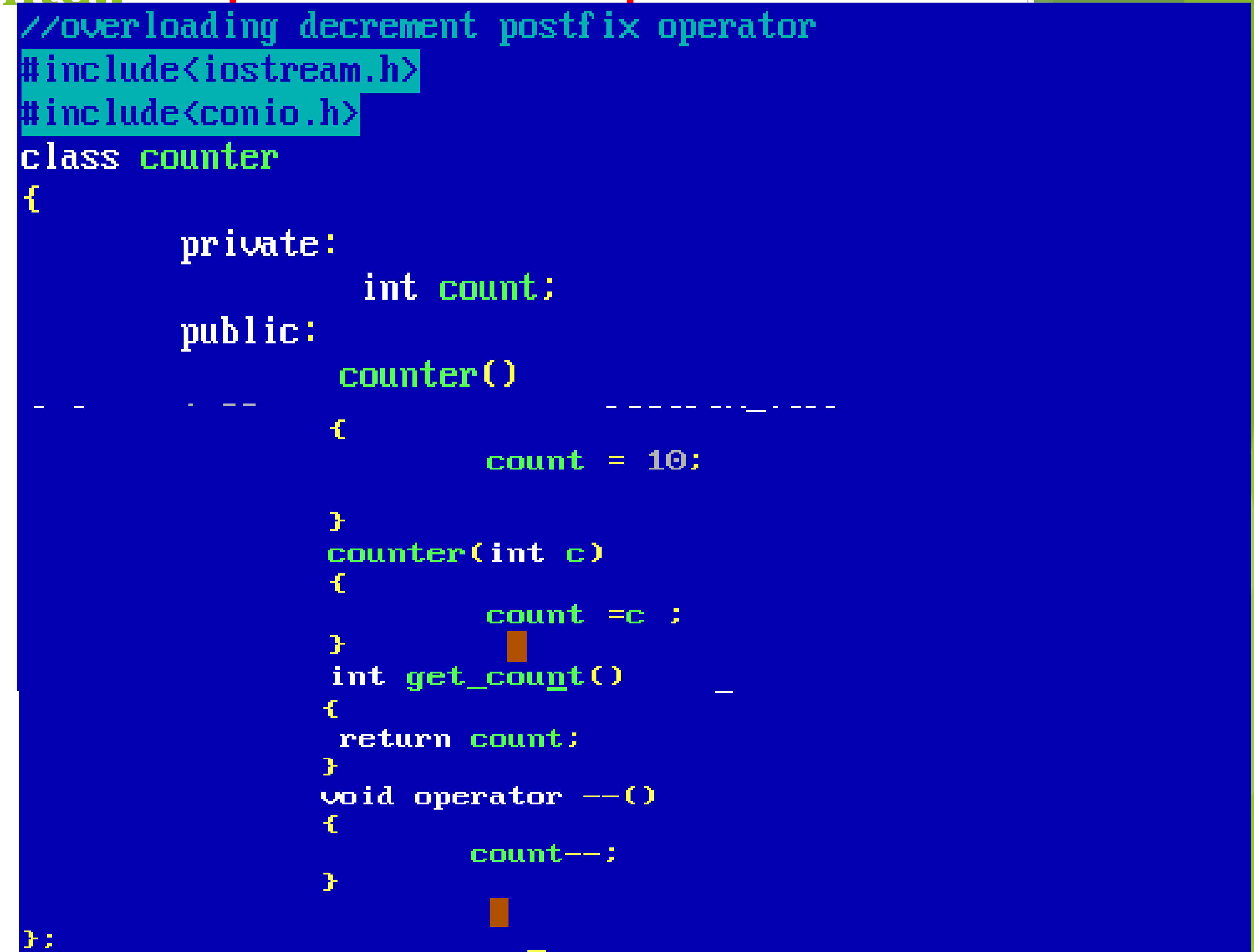
```
int main()
{
        clrscr();
        counter c1, c2(100);
        cout<<"value of count before increment"<<endl;
        cout<<"c1="<<c1.get_count()<<endl;
        cout<<"c2="<<c2.get_count()<<endl;

       c1++;       //equivalent to c1.operator ++();
       c2++;       //equivalent to c2.operator ++();
       cout<<"value of count after increment"<< endl;
       cout<<"c1="<<c1.get_count()<<endl;
       cout<<"c2="<<c2.get_count()<<endl;
       getch();
       return 0;
}
```



```
value of count before increment
c1=10
c2=100
value of count after increment
c1=11
c2=101
```

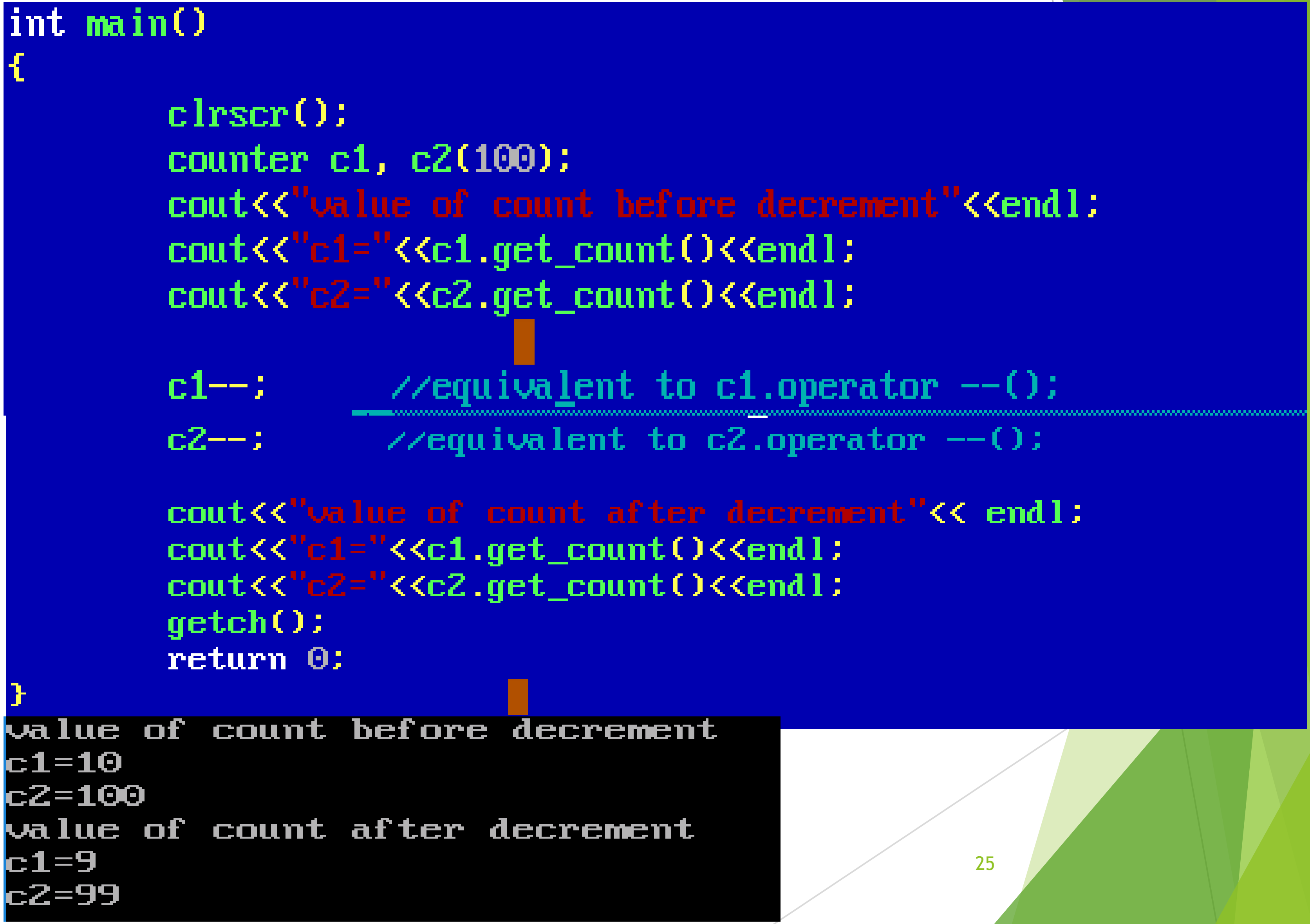
Contd..

Overload postfix decrement operator:



```
//overloading decrement postfix operator
#include<iostream.h>
#include<conio.h>
class counter
{
        private:
                 int count;
        public:
                counter()
               {
                       count = 10;

               }
               counter(int c)
               {
                       count =c ;
               }
               int get_count()
              {
               return count;
              }
              void operator --()
              {
                      count--;
              }

};
```

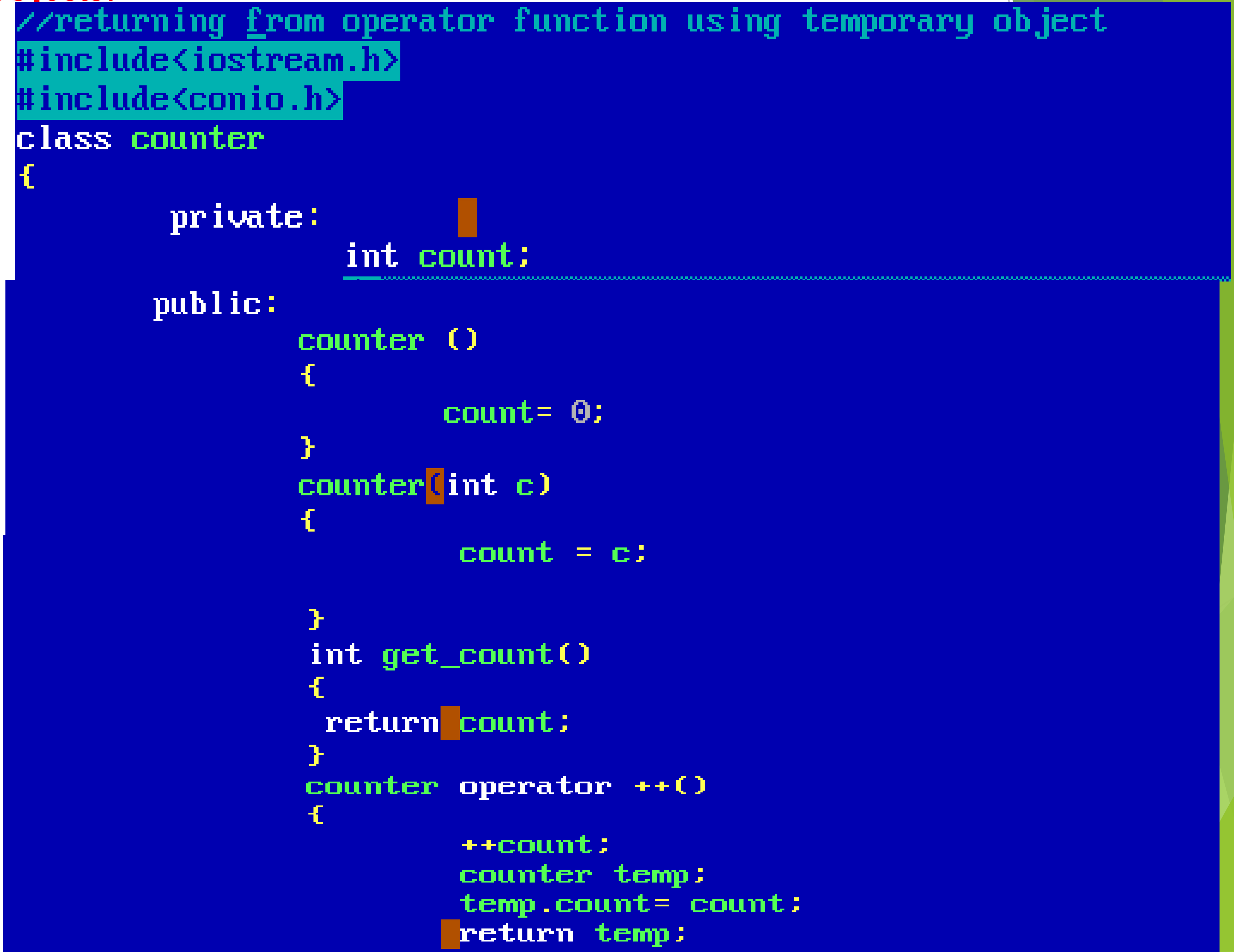
Contd..



```
int main()
{
        clrscr();
        counter c1, c2(100);
        cout<<"value of count before decrement"<<endl;
        cout<<"c1="<<c1.get_count()<<endl;
        cout<<"c2="<<c2.get_count()<<endl;

        c1--;       //equivalent to c1.operator --();
        c2--;       //equivalent to c2.operator --();

        cout<<"value of count after decrement"<< endl;
        cout<<"c1="<<c1.get_count()<<endl;
        cout<<"c2="<<c2.get_count()<<endl;
        getch();
        return 0;
}
```

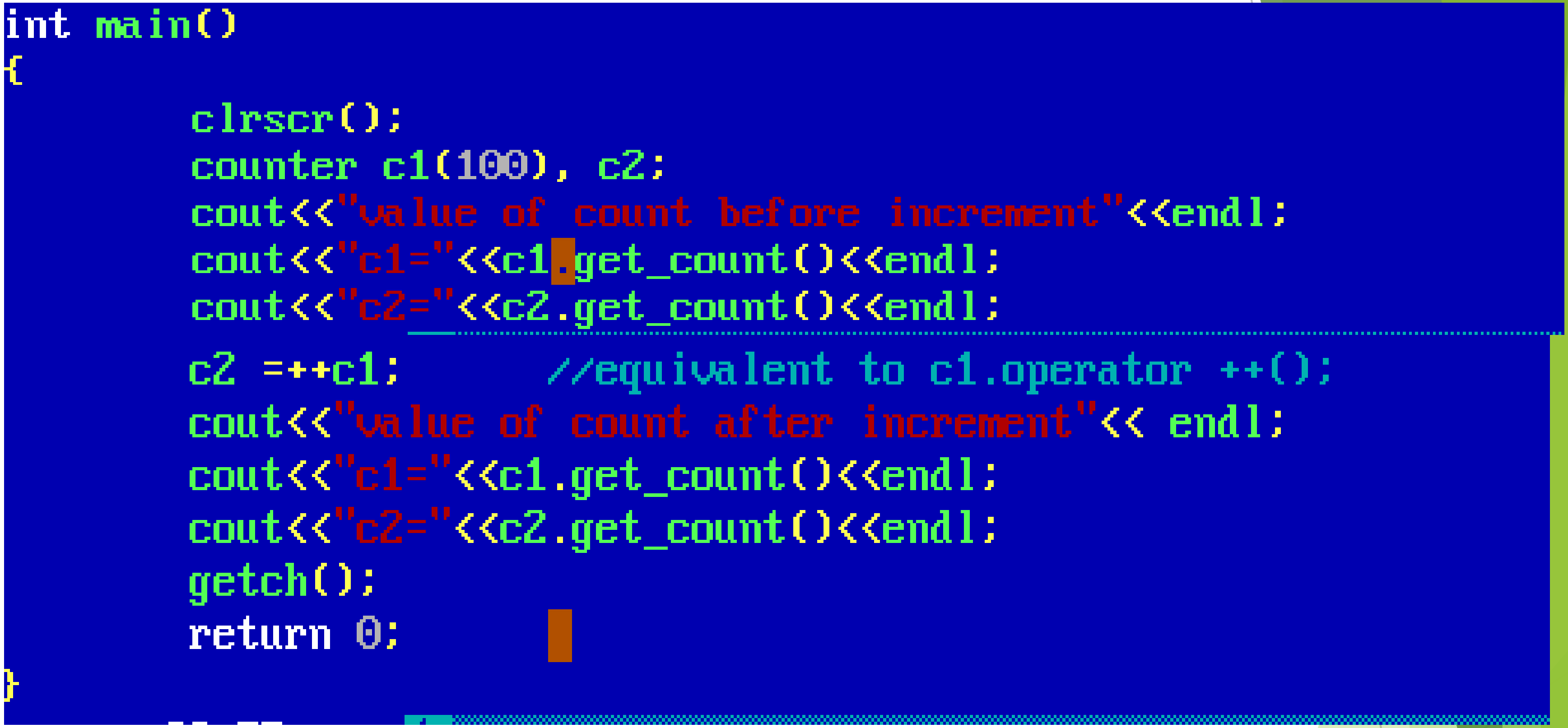
```
value of count before decrement
c1=10
c2=100
value of count after decrement
c1=9
c2=99
```

Returning values from operator function: Returning values using temporary objects:

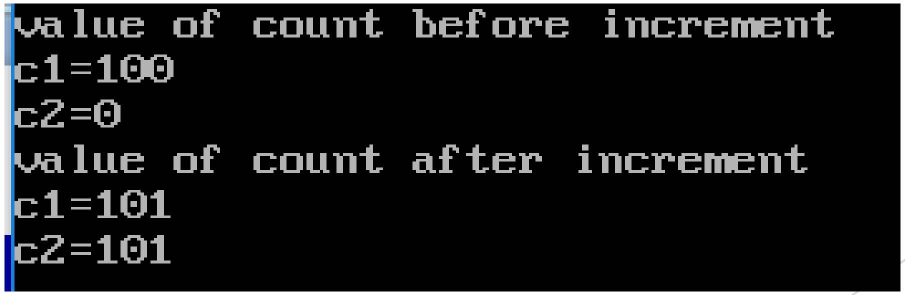


```
//returning from operator function using temporary object
#include<iostream.h>
#include<conio.h>
class counter
{
        private:
                 int count;
       public:
              counter ()
              {
                     count= 0;
              }
              counter(int c)
              {
                      count = c;

               }
               int get_count()
               {
                return count;
               }
               counter operator ++()
               {
                      ++count;
                      counter temp;
                      temp.count= count;
                     return temp;
```

Contd..



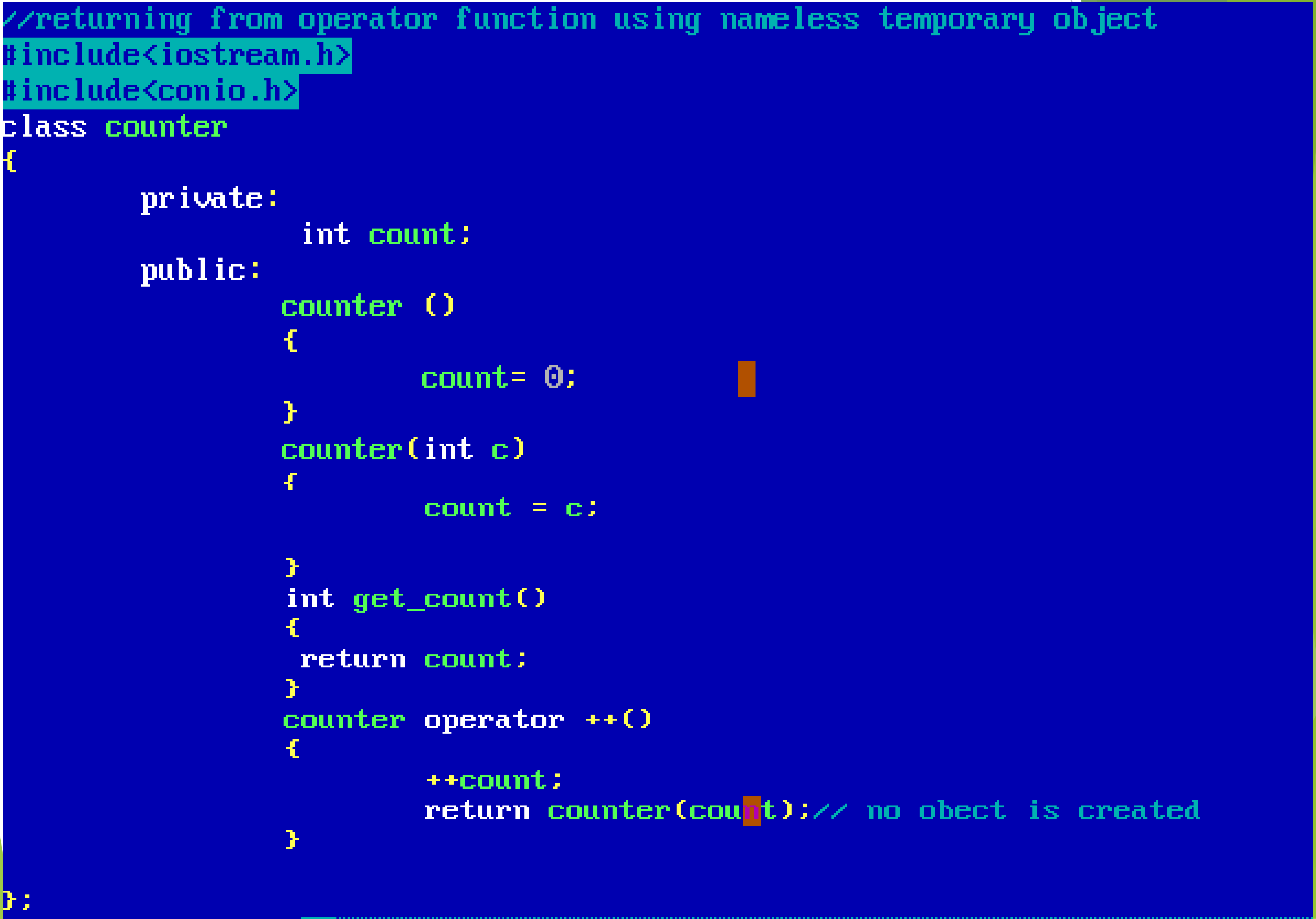
```
int main()
{
        clrscr();
        counter c1(100), c2;
        cout<<"value of count before increment"<<endl;
        cout<<"c1="<<c1.get_count()<<endl;
        cout<<"c2="<<c2.get_count()<<endl;
        c2 =++c1;       //equivalent to c1.operator ++();
        cout<<"value of count after increment"<< endl;
        cout<<"c1="<<c1.get_count()<<endl;
        cout<<"c2="<<c2.get_count()<<endl;
        getch();
        return 0;
}
```



```
value of count before increment
c1=100
c2=0
value of count after increment
c1=101
c2=101
```

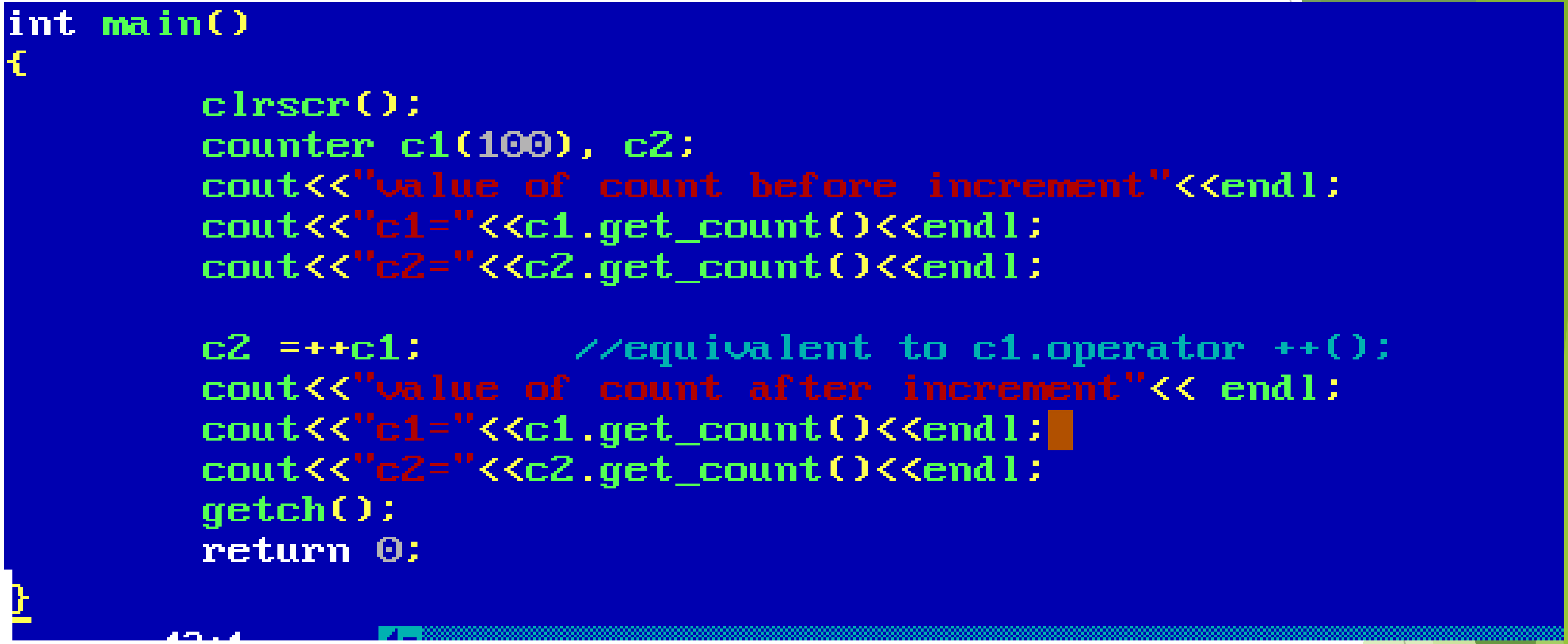
Contd..

Returning values using nameless temporary objects:

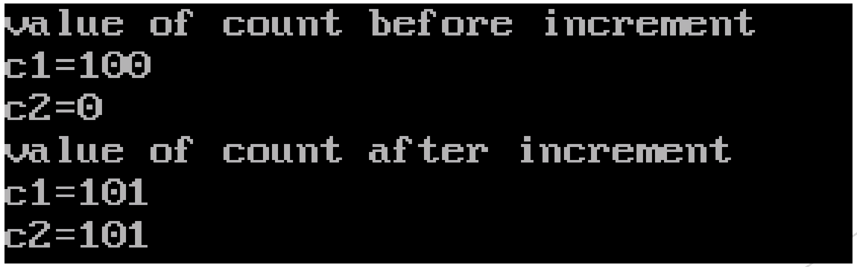


```
//returning from operator function using nameless temporary object
#include<iostream.h>
#include<conio.h>
class counter
{
        private:
                 int count;
        public:
                counter ()
                {
                        count= 0;
                }
                counter(int c)
                {
                        count = c;

                }
                int get_count()
                {
                 return count;
                }
                counter operator ++()
                {
                        ++count;
                        return counter(count);// no obect is created
                }

};
```

Contd..



```
int main()
{
        clrscr();
        counter c1(100), c2;
        cout<<"value of count before increment"<<endl;
        cout<<"c1="<<c1.get_count()<<endl;
        cout<<"c2="<<c2.get_count()<<endl;

        c2 =++c1;      //equivalent to c1.operator ++();
        cout<<"value of count after increment"<< endl;
        cout<<"c1="<<c1.get_count()<<endl;
        cout<<"c2="<<c2.get_count()<<endl;
        getch();
        return 0;
}
```



```
value of count before increment
c1=100
c2=0
value of count after increment
c1=101
c2=101
```

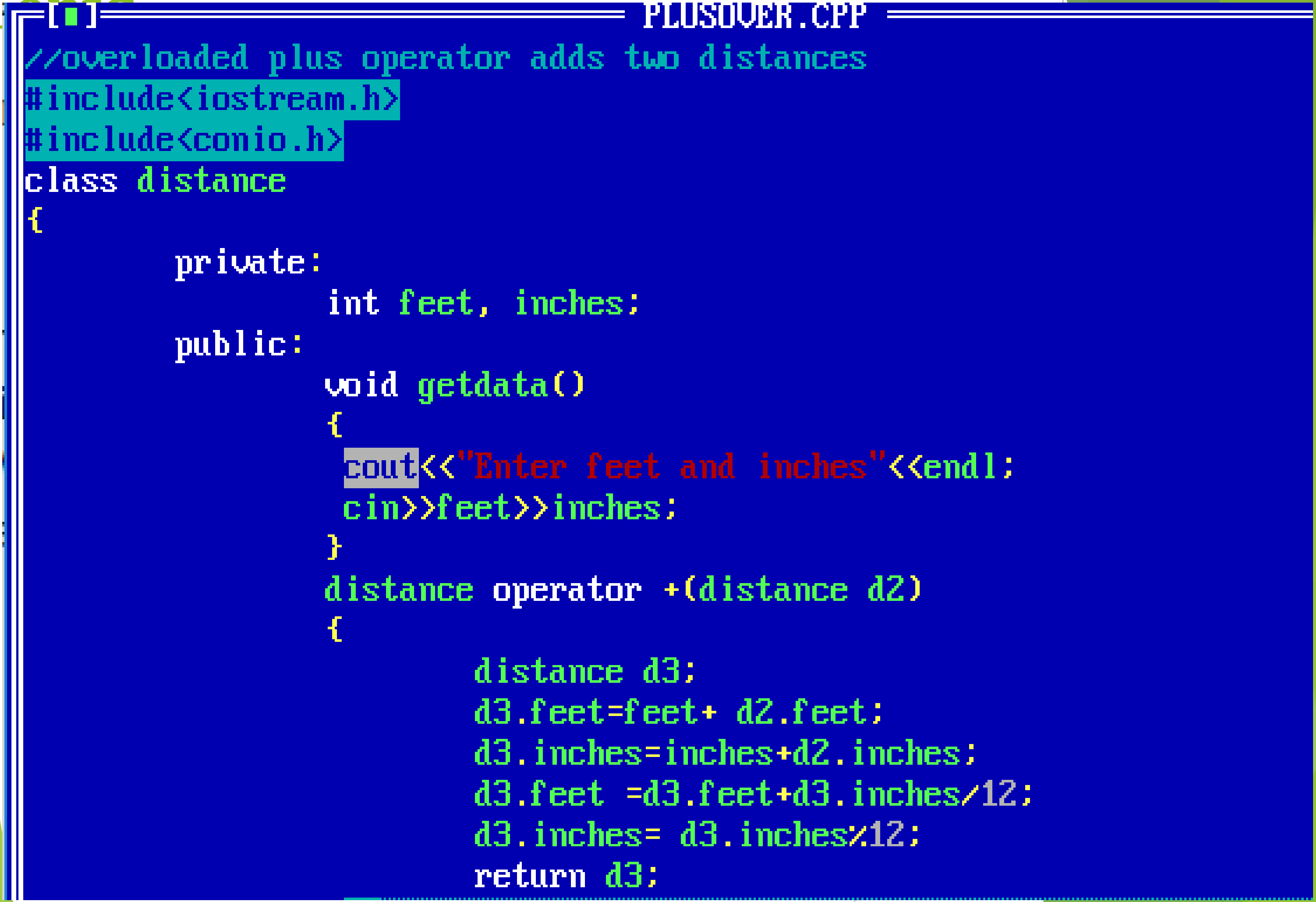
Overloading Binary Operators

- Binary operator are operators, which require two operands to perform the operation.
  - Arithmetic operators(+, -, *, /, %)
  - Assignment operator(= , +=, -=, /=, *=, %=)
  - Comparison operator(>, <, <=, >=, ==, !=)
- Binary operators can be overloaded as unary operator however binary operator requires an additional parameter.
- In overloading binary operators the object to the left of the operator is used to invoke the operator function while the operand to the right of the operator is always passed as an argument to the function.

Arithmetic operators overloading

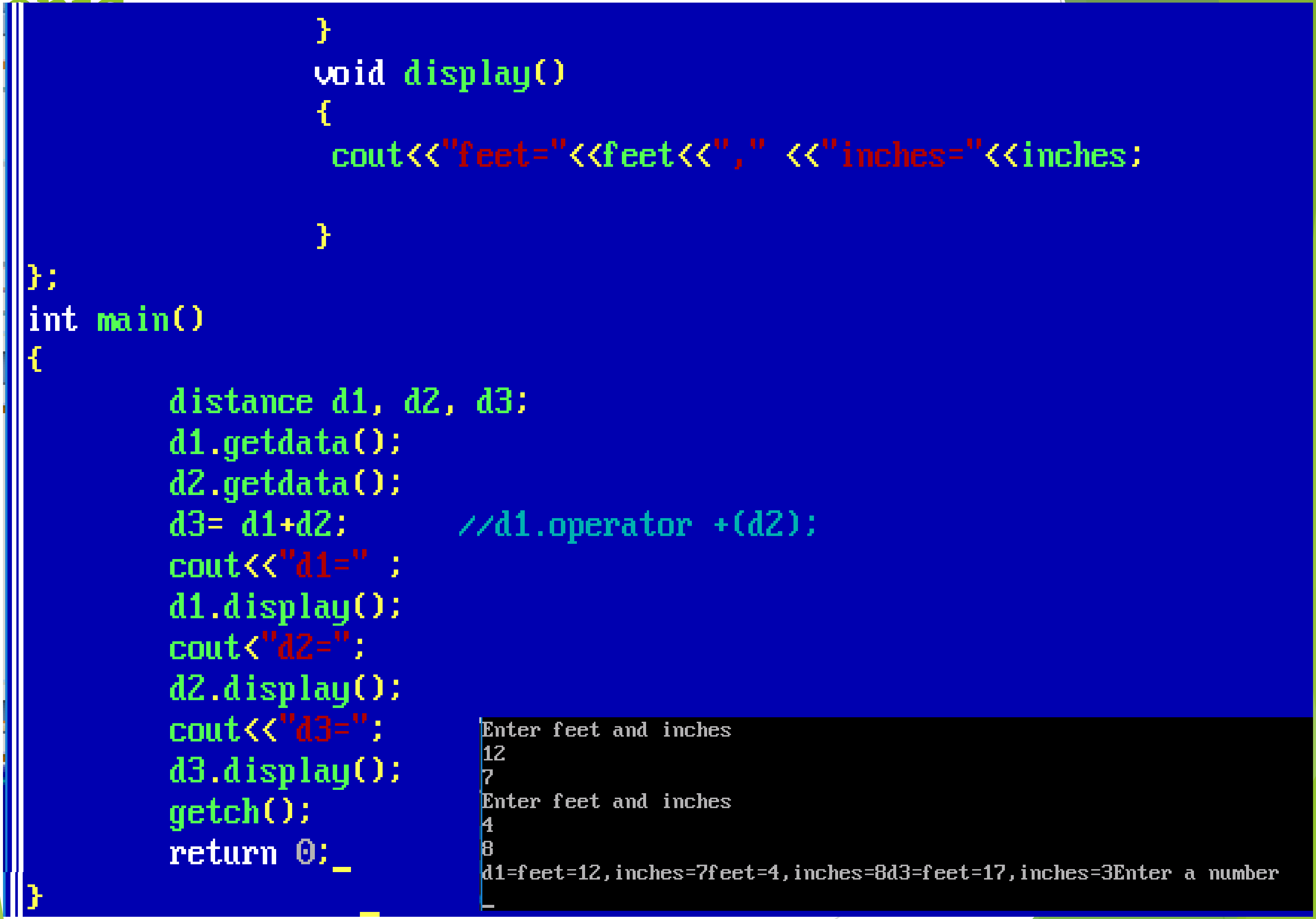
- Overloading plus operator:
    - This operator adds the values of two operands when applied to a basic data item.
    - This operator can be overloaded to add the values of corresponding data members when applied to two objects.

Contd



```
PLUSOVER.CPP
//overloaded plus operator adds two distances
#include<iostream.h>
#include<conio.h>
class distance
{
        private:
                int feet, inches;
        public:
                void getdata()
                {
                 cout<<"Enter feet and inches"<<endl;
                 cin>>feet>>inches;
                }
                distance operator +(distance d2)
                {
                        distance d3;
                        d3.feet=feet+ d2.feet;
                        d3.inches=inches+d2.inches;
                        d3.feet =d3.feet+d3.inches/12;
                        d3.inches= d3.inches%12;
                        return d3;
```

Contd



```
                    }
                    void display()
                    {
                     cout<<"feet="<<feet<<"," <<"inches="<<inches;

                    }
};
int main()
{
         distance d1, d2, d3;
         d1.getdata();
         d2.getdata();
         d3= d1+d2;       //d1.operator +(d2);
         cout<<"d1=" ;
         d1.display();
         cout<"d2=";
         d2.display();
         cout<<"d3=";
         d3.display();
         getch();
         return 0;_
}
```

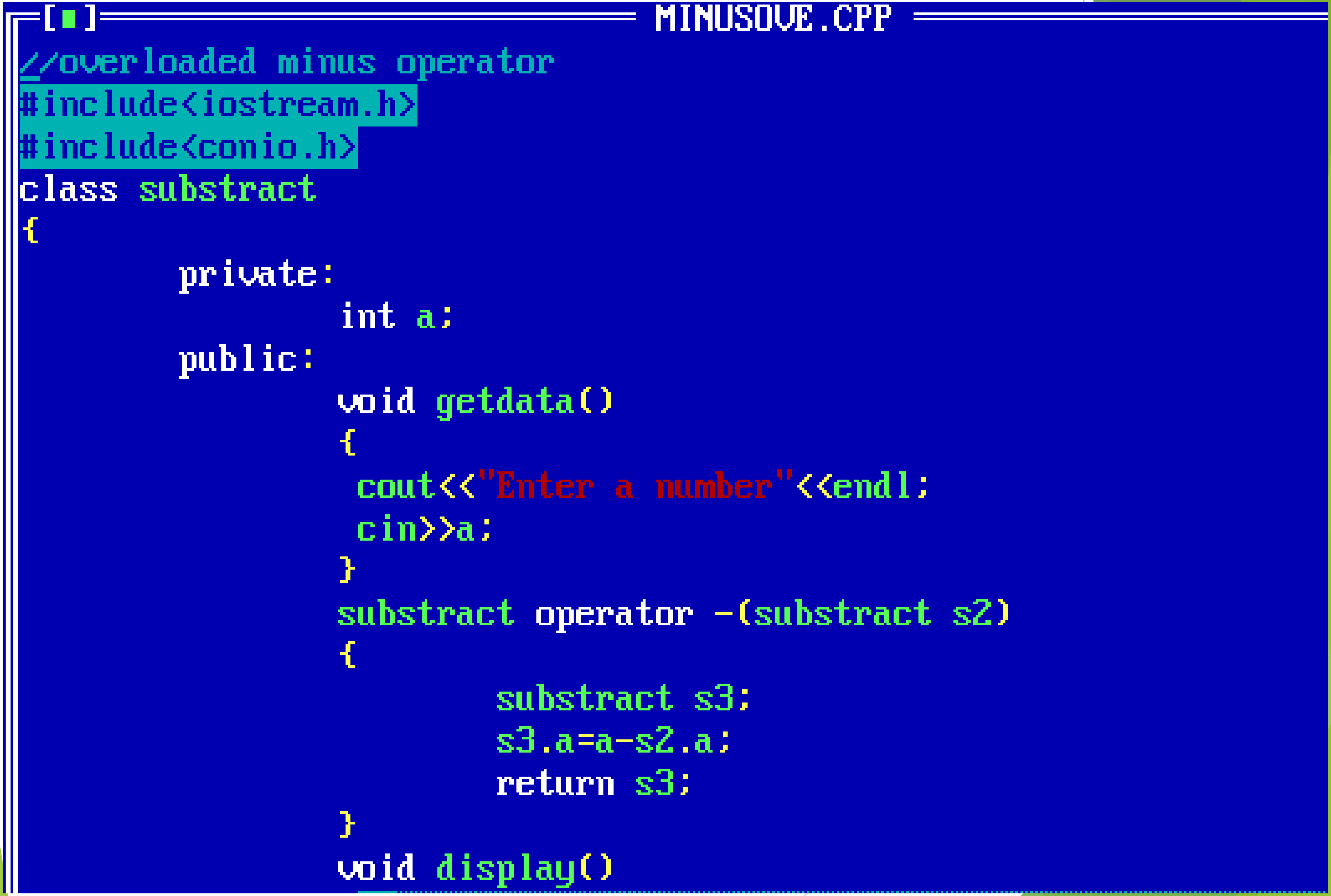
```
Enter feet and inches
12
7
Enter feet and inches
4
8
d1=feet=12,inches=7feet=4,inches=8d3=feet=17,inches=3Enter a number
_
```

Contd..

- Note: in the same way we can overload other arithmetic operators(-, *, /, %) to subtract, multiply, and divide objects.

Contd..

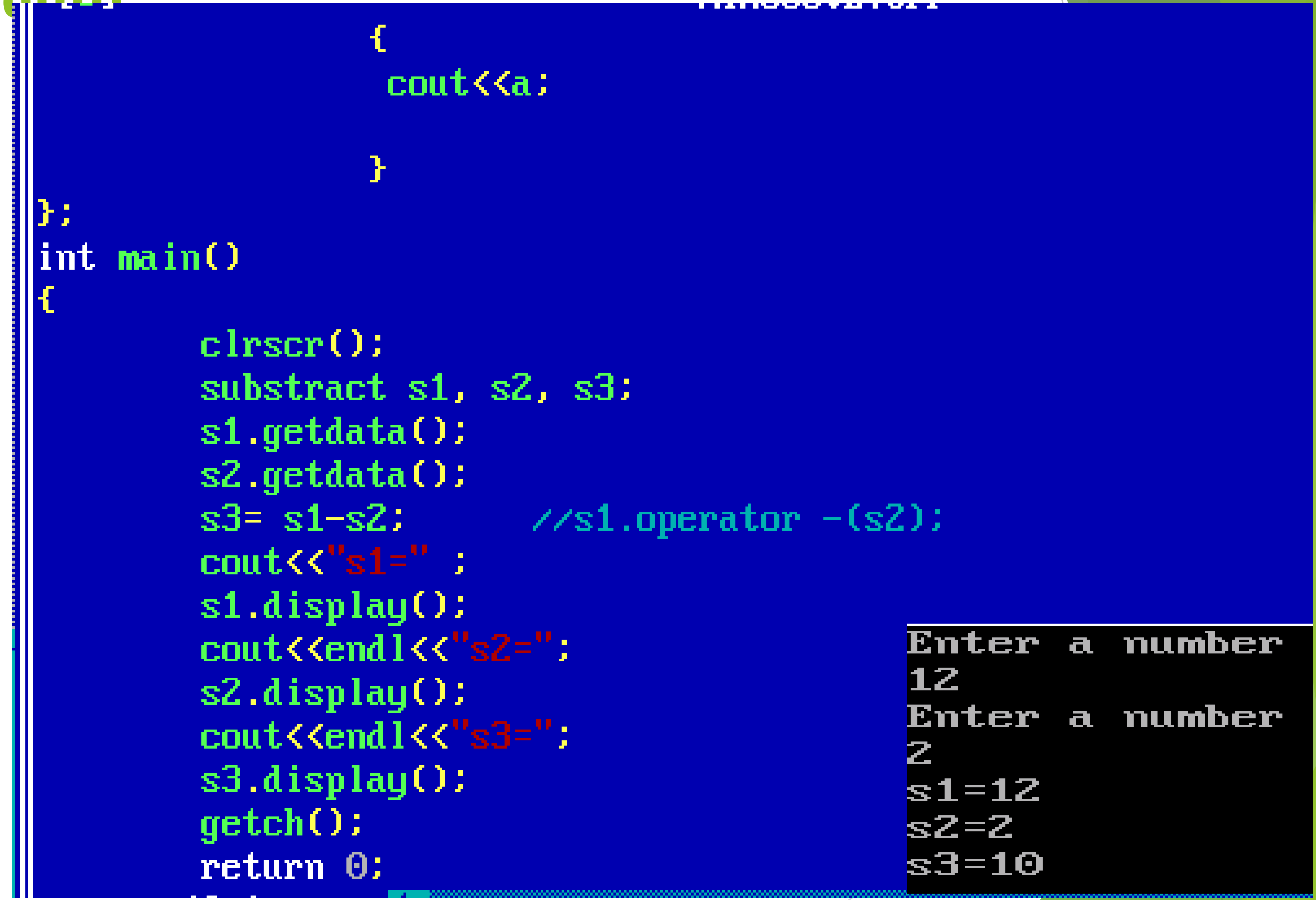
Example: WAP to subtract one object from another object.



MINUSOVE.CPP

```
//overloaded minus operator
#include<iostream.h>
#include<conio.h>
class substract
{
        private:
                int a;
        public:
                void getdata()
                {
                 cout<<"Enter a number"<<endl;
                 cin>>a;
                }
                substract operator -(substract s2)
                {
                        substract s3;
                        s3.a=a-s2.a;
                        return s3;
                }
                void display()
```

Contd



```
                {
                 cout<<a;

                }
};
int main()
{
        clrscr();
        substract s1, s2, s3;
        s1.getdata();
        s2.getdata();
        s3= s1-s2;      //s1.operator -(s2);
        cout<<"s1=" ;
        s1.display();
        cout<<endl<<"s2=";
        s2.display();
        cout<<endl<<"s3=";
        s3.display();
        getch();
        return 0;
```

```
Enter a number
12
Enter a number
2
s1=12
s2=2
s3=10
```

Contd..

- Homework:
  - Write a program to compute subtraction of two complex numbers using operator overloading.
  - Write a program to compute addition of two complex numbers using operator overloading.
  - WAP to multiply two objects.
  - WAP to illustrate modulo and normal division of objects using operator overloading.

Overloading comparison operator

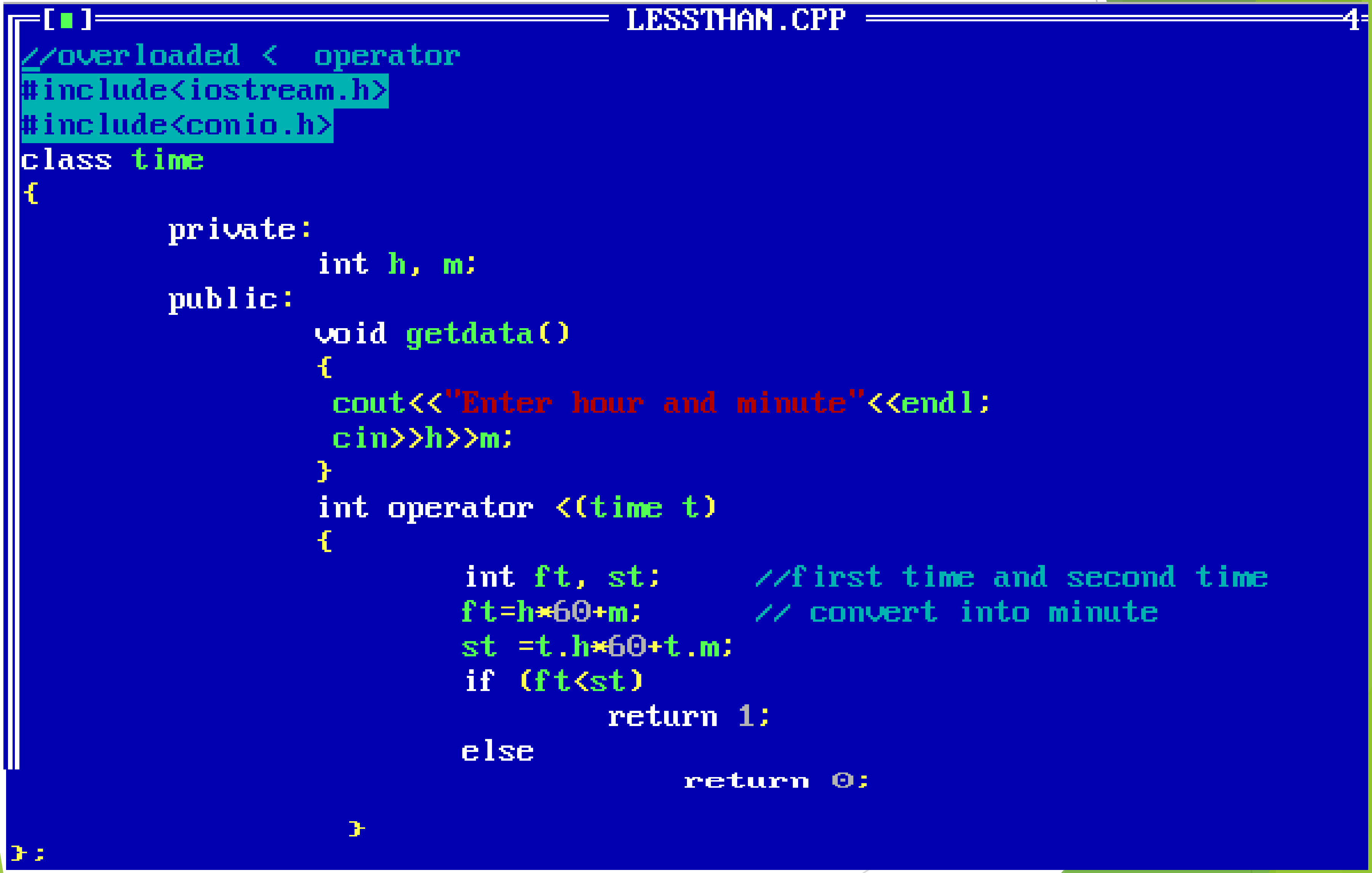
- Comparison operator(>, <, <=, >=, ==, !=)

- Overloading comparison operator is almost similar to overloading arithmetic operator except that it must return value of an integer type.

- This is because result of comparison is always true or false.

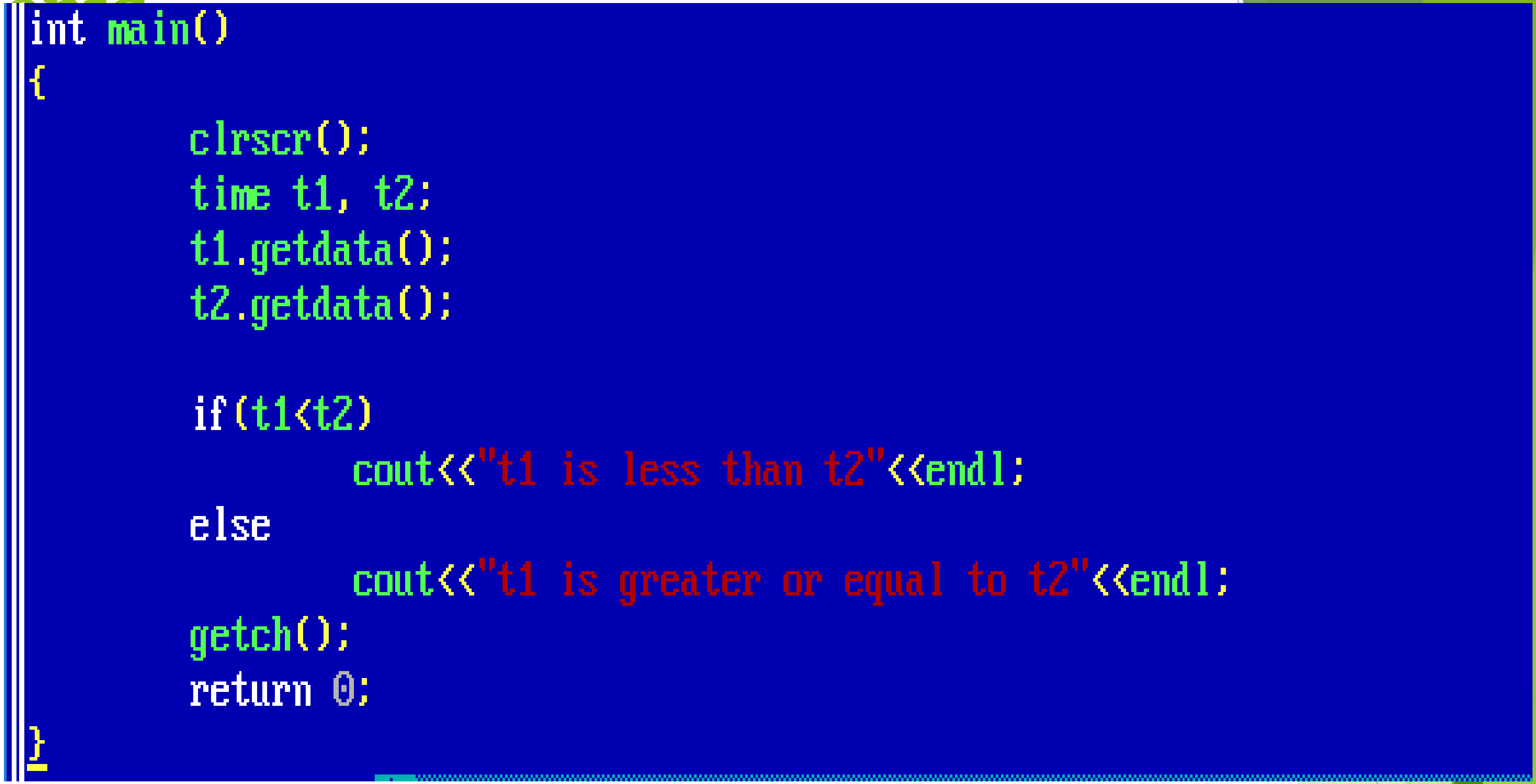
Contd..

Overloading less than (<) operator:



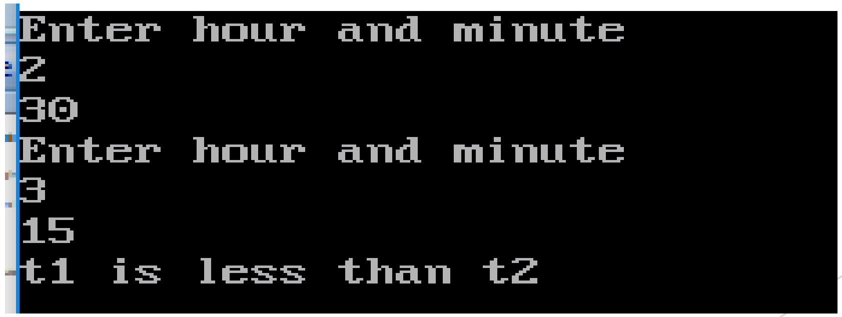
```
[■]══════════════ LESSTHAN.CPP ══════════════4═
//overloaded <  operator
#include<iostream.h>
#include<conio.h>
class time
{
        private:
                int h, m;
        public:
                void getdata()
                {
                 cout<<"Enter hour and minute"<<endl;
                 cin>>h>>m;
                }
                int operator <(time t)
                {
                        int ft, st;     //first time and second time
                        ft=h*60+m;      // convert into minute
                        st =t.h*60+t.m;
                        if (ft<st)
                                return 1;
                        else
                                return 0;

                }
};
```

Contd



```
int main()
{
        clrscr();
        time t1, t2;
        t1.getdata();
        t2.getdata();

        if(t1<t2)
                cout<<"t1 is less than t2"<<endl;
        else
                cout<<"t1 is greater or equal to t2"<<endl;
        getch();
        return 0;
}
```

- **Output:**



```
Enter hour and minute
2
30
Enter hour and minute
3
15
t1 is less than t2
```

Note: in the same way other relational operators can be overloaded.

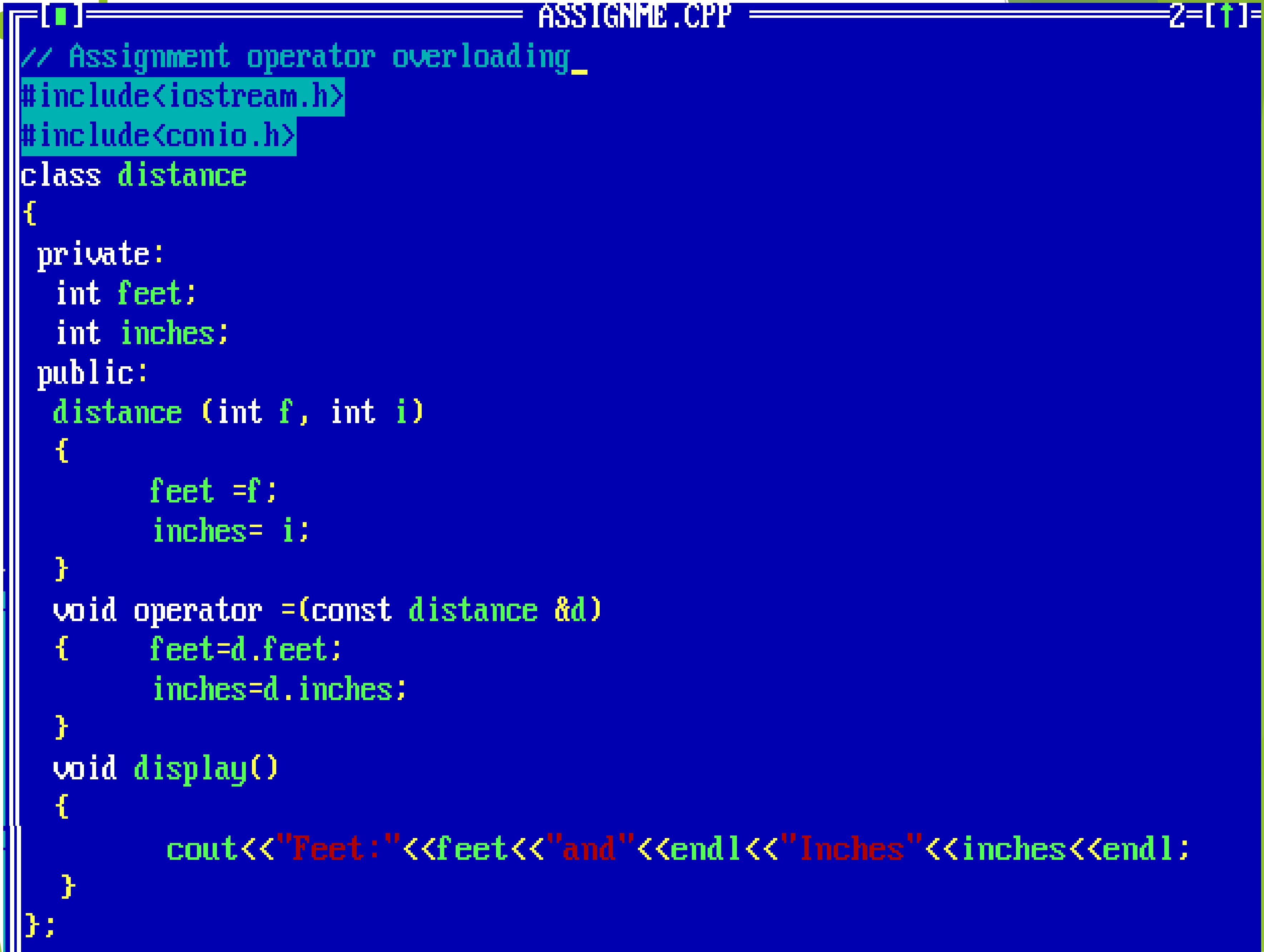
Contd..

- Homework:
  - Write a program that takes two amount (can be in RS and PS) as input and decide which one is less.
  - Write a program to compare the two distances taken as input in the program and decide which one is greater than other.

Overloading assignment operator

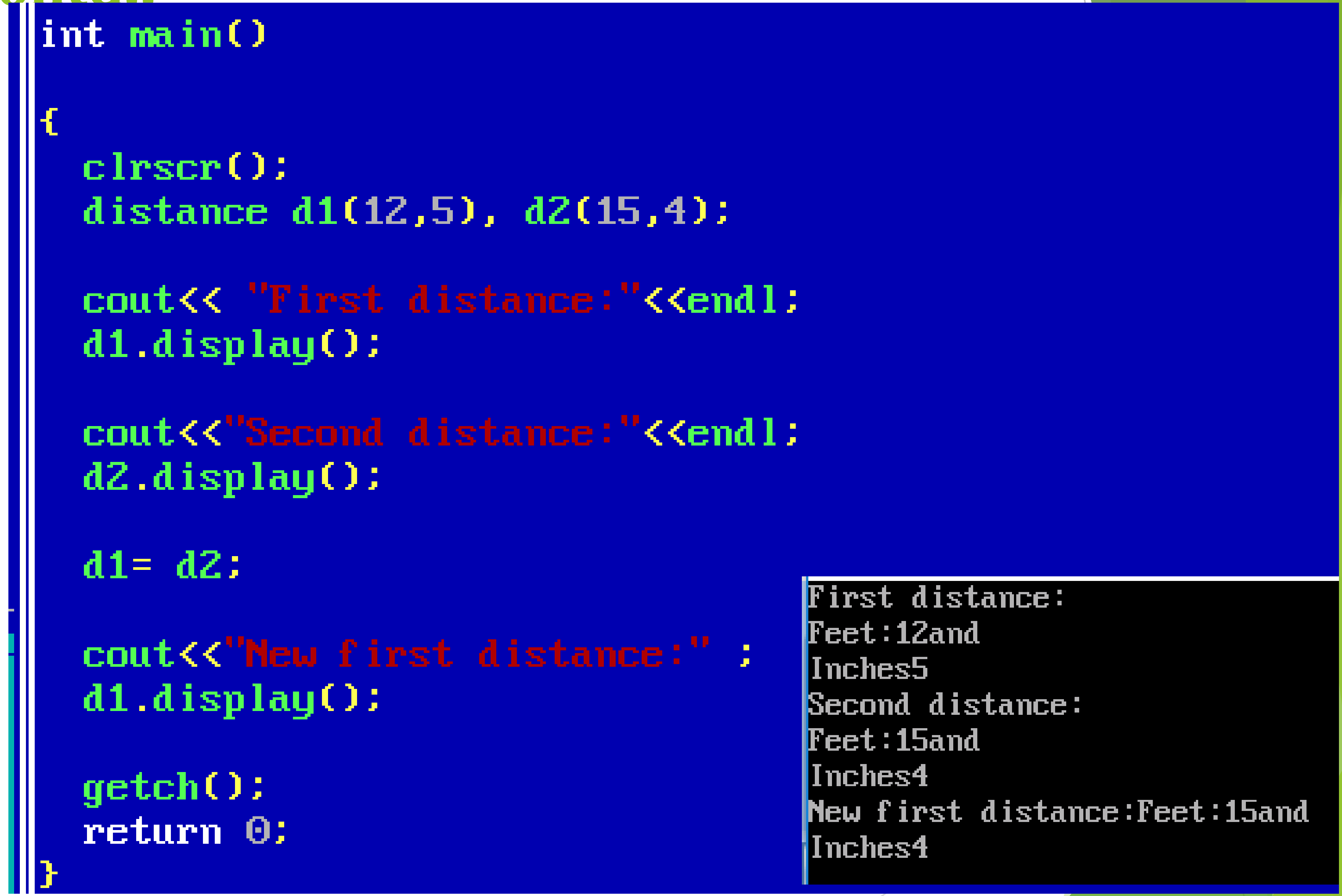
- In c++ overloading of assignment(=) operator is possible.
- By overloading assignment operator, all values of one object can be copied to another object by using assignment operator.
- Assignment operator must be overloaded by non-static member function only.
- If the overloading function for the assignment operator is not written in the class, the compiler generates the function to overload the assignment operator.

Co



```
// Assignment operator overloading
#include<iostream.h>
#include<conio.h>
class distance
{
 private:
  int feet;
  int inches;
 public:
  distance (int f, int i)
  {
        feet =f;
        inches= i;
  }
  void operator =(const distance &d)
  {     feet=d.feet;
        inches=d.inches;
  }
  void display()
  {
         cout<<"Feet:"<<feet<<"and"<<endl<<"Inches"<<inches<<endl;
   }
};
```

Contd..



```
int main()

{
  clrscr();
  distance d1(12,5), d2(15,4);

  cout<< "First distance:"<<endl;
  d1.display();

  cout<<"Second distance:"<<endl;
  d2.display();

  d1= d2;

  cout<<"New first distance:" ;
  d1.display();

  getch();
  return 0;
}
```

```
First distance:
Feet:12and
 Inches5
Second distance:
Feet:15and
 Inches4
New first distance:Feet:15and
 Inches4
```

Contd..

- **Similar to copy constructor overriding:**
  - if the members of the objects are pointers then during assignment only the address are copied. That is, both of the objects point to the same memory area and they don not have distinct data.
- **The difference is:**
  - copy constructor initializes during object creation where as the assignment operator assigns(copies) to the already created(well constructed) object.

Data Conversion(Type conversion)

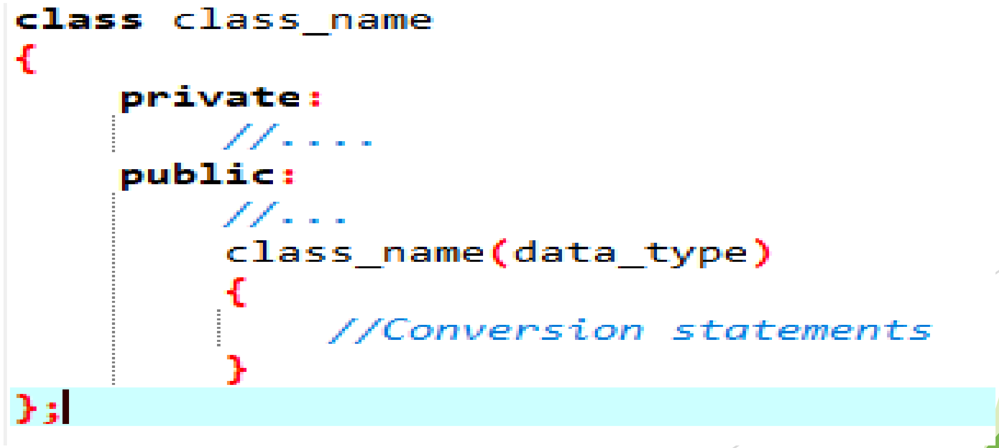
- It is the process of converting one type into another. In other words converting an expression of a given type into another is called type conversion.
- A type conversion may either be explicit or implicit, depending on whether it is ordered by the programmer or by the compiler.
- Explicit type conversions are used when a programmer want to get around the compiler's typing system.
- If the data types are user defined, the compiler does not support automatic type conversion and therefore, we must design the conversion routine by ourselves.

Contd..

- Four types of situations might arise in the data conversion:
  - Basic to basic
  - Basic to user-defined
  - User-defined to basic
  - User-defined to user-defined

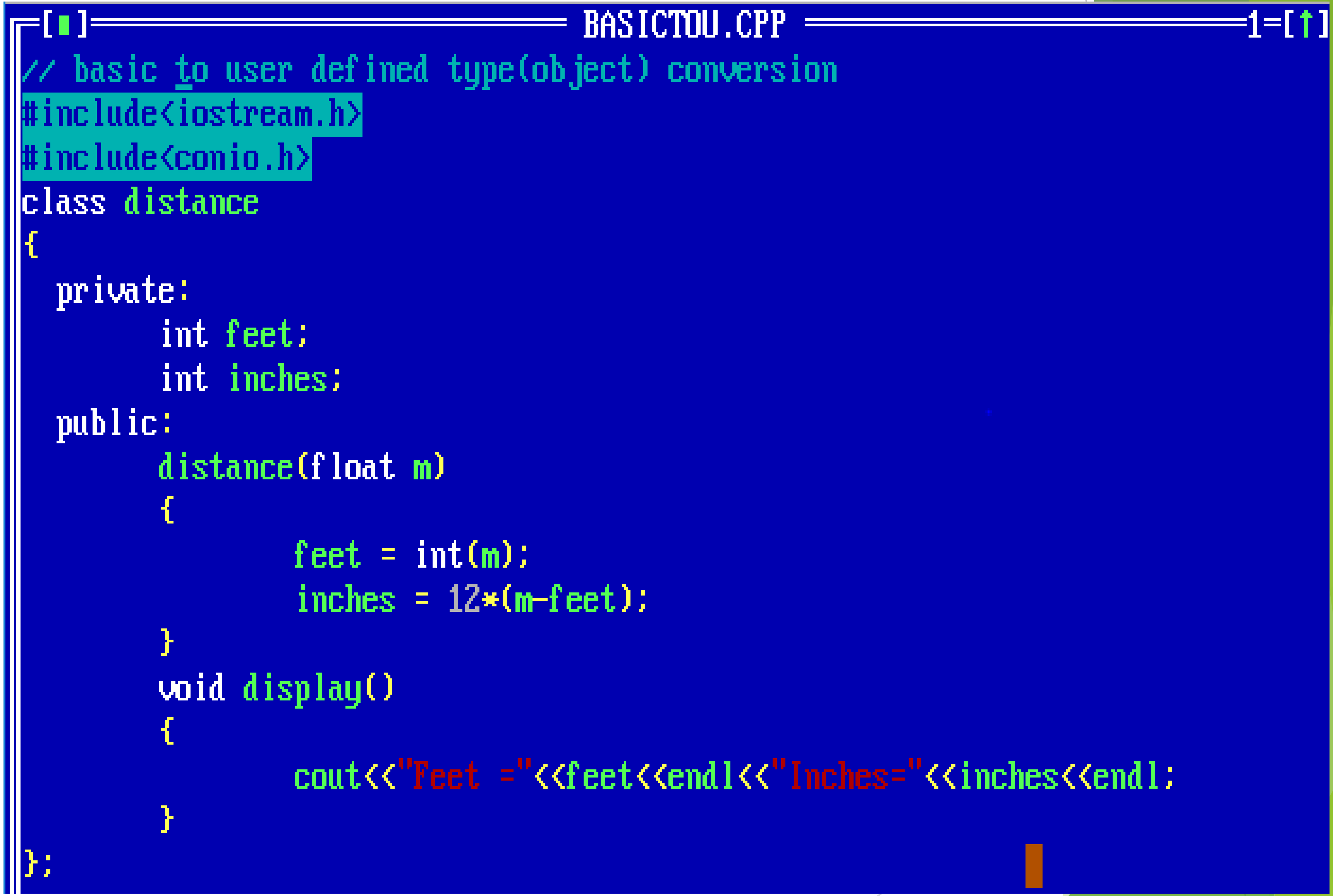
Contd..

- Conversion from basic type to user defined type:
  - To convert basic types to user defined type(object) it is necessary to use the constructor.
  - The constructor in this case takes single argument whose type is to be converted.
  - Syntax:



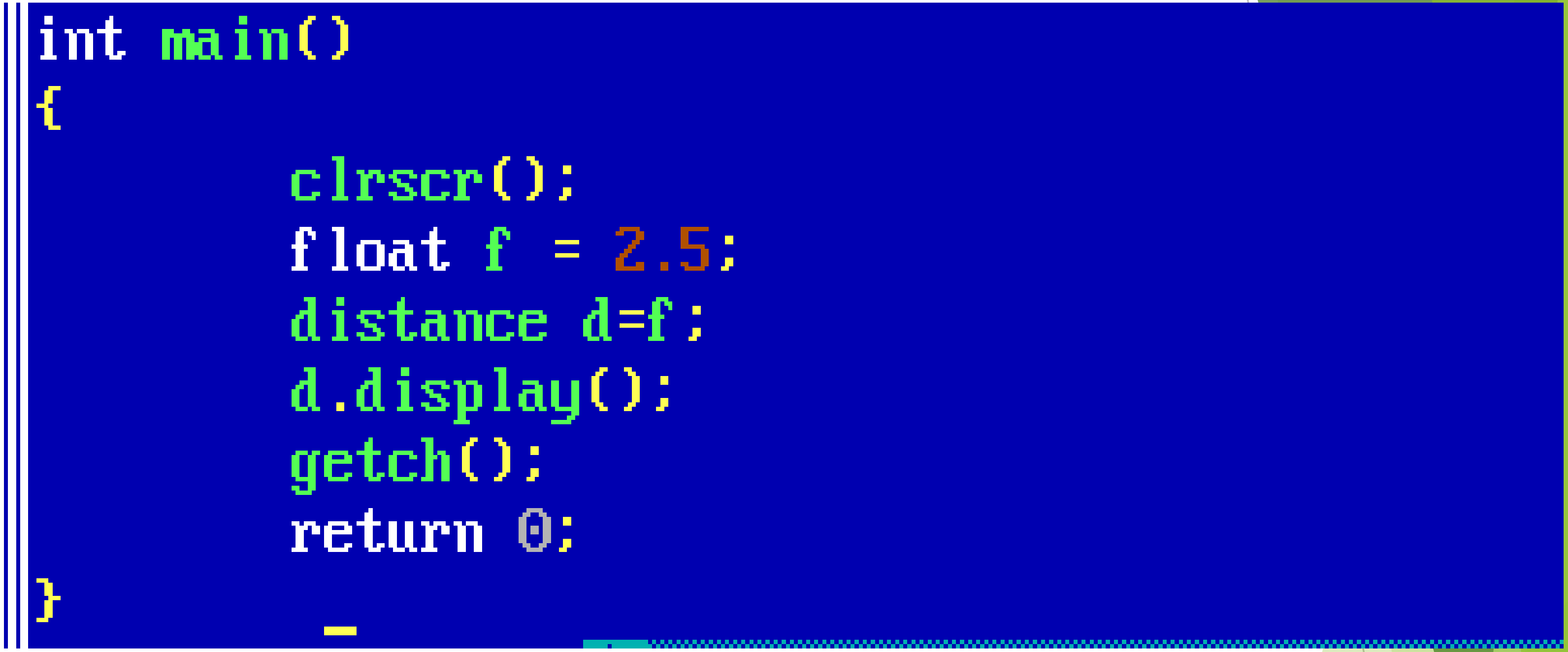
```
class class_name
{
          private:
                    //....
          public:
                    //...
                    class_name(data_type)
                    {
                              //Conversion statements
                    }
};
```

Contd..



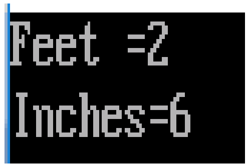
```
BASICTOU.CPP
// basic to user defined type(object) conversion
#include<iostream.h>
#include<conio.h>
class distance
{
  private:
        int feet;
        int inches;
  public:
        distance(float m)
        {
                feet = int(m);
                inches = 12*(m-feet);
        }
        void display()
        {
                cout<<"Feet ="<<feet<<endl<<"Inches="<<inches<<endl;
        }
};
```

Contd..



```
int main()
{
        clrscr();
        float f = 2.5;
        distance d=f;
        d.display();
        getch();
        return 0;
}
```

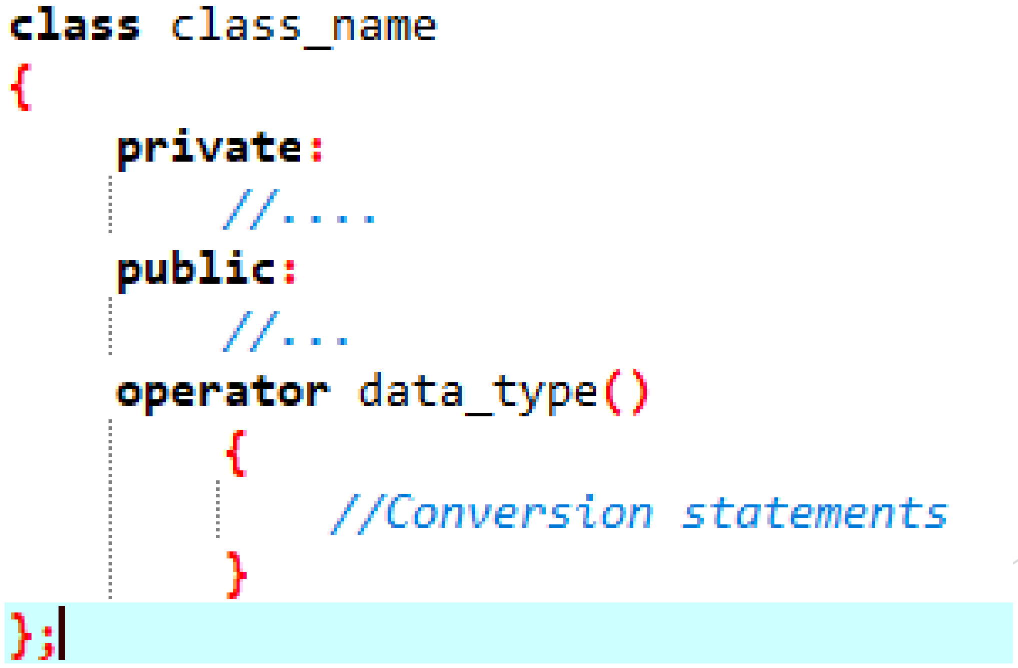
Output:



```
Feet =2
Inches=6
```

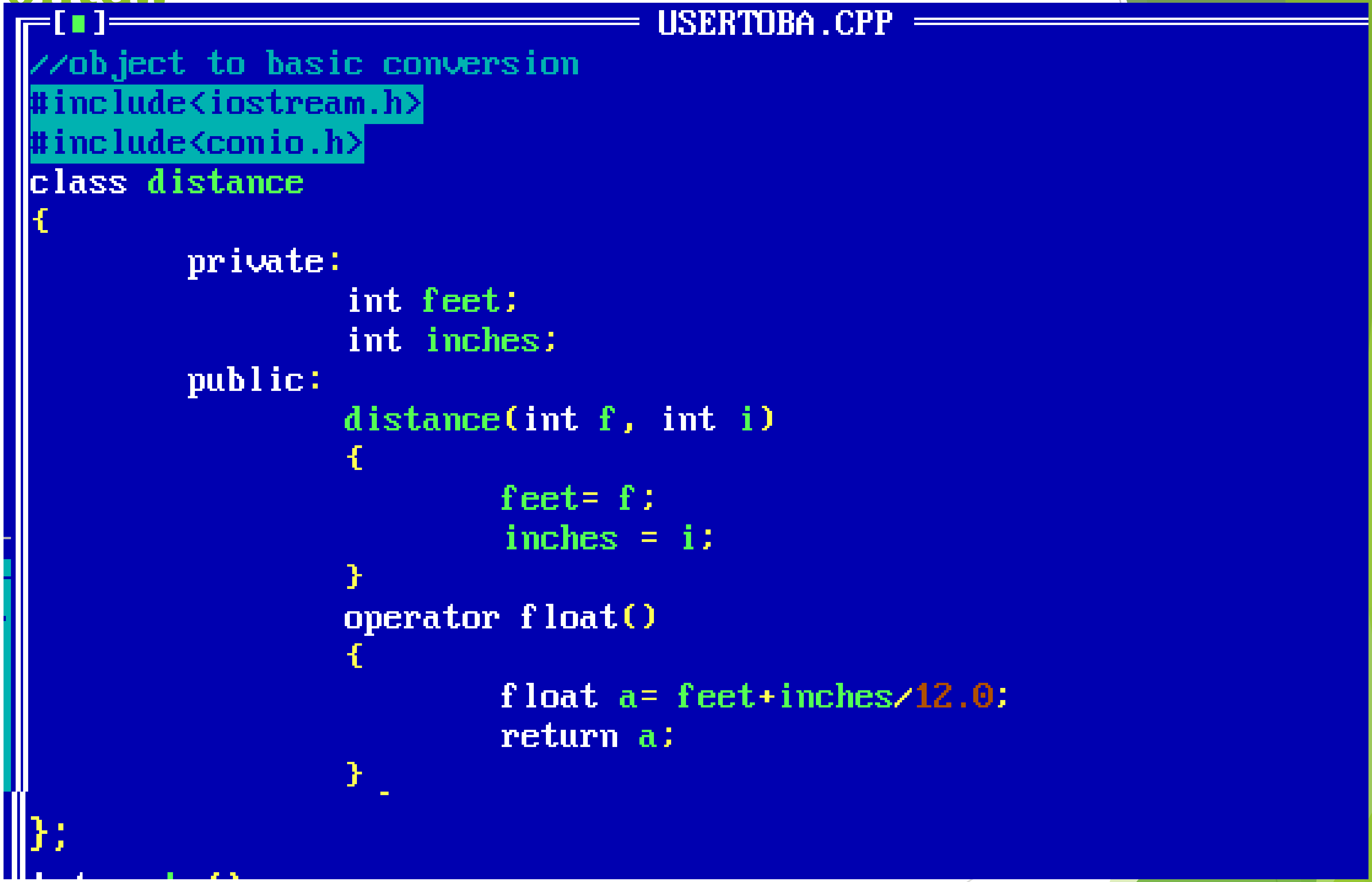
Contd..

- **Conversion from user defined type to basic type:**
  - To convert user defined types(objects) to basic type it is necessary to overload cast operator.
  - The overloaded cast operator does not have return type. Its implicit return type is the type to which object need to be converted.
  - To convert object to basic type, we use conversion function as below:
  - **Syntax:**



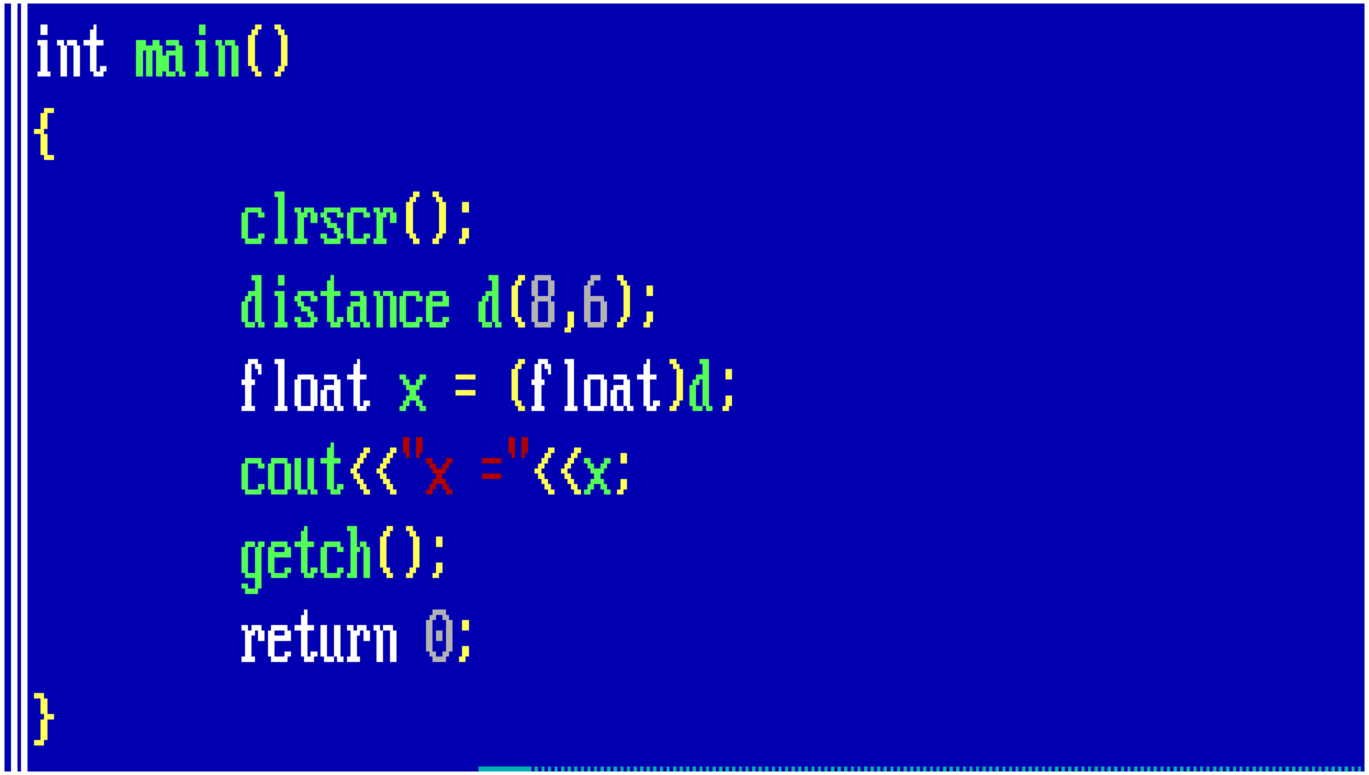
```
class class_name
{
    private:
        //....
    public:
        //...
    operator data_type()
        {
            //Conversion statements
        }
};
```

Contd..

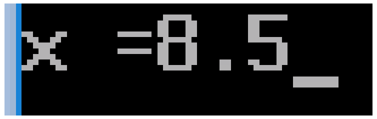


```
USERTOBA.CPP
//object to basic conversion
#include<iostream.h>
#include<conio.h>
class distance
{
        private:
                int feet;
                int inches;
        public:
                distance(int f, int i)
                {
                        feet= f;
                        inches = i;
                }
                operator float()
                {
                        float a= feet+inches/12.0;
                        return a;
                }
};
```

Contd..



```
int main()
{
        clrscr();
        distance d(8,6);
        float x = (float)d;
        cout<<"x ="<<x;
        getch();
        return 0;
}
```

- Output: 

```
x =8.5_
```

Contd..

- Conversion from user defined type to another user defined type:
  - C++ compiler does not support data conversion between objects of user-defined classes.
  - This type of conversion can be carried out either by:
    - A One argument constructor or
    - an operator function.
  - The choice between these two methods for data conversion depends on whether the conversion function be defined in the source class or destination class.

Contd..

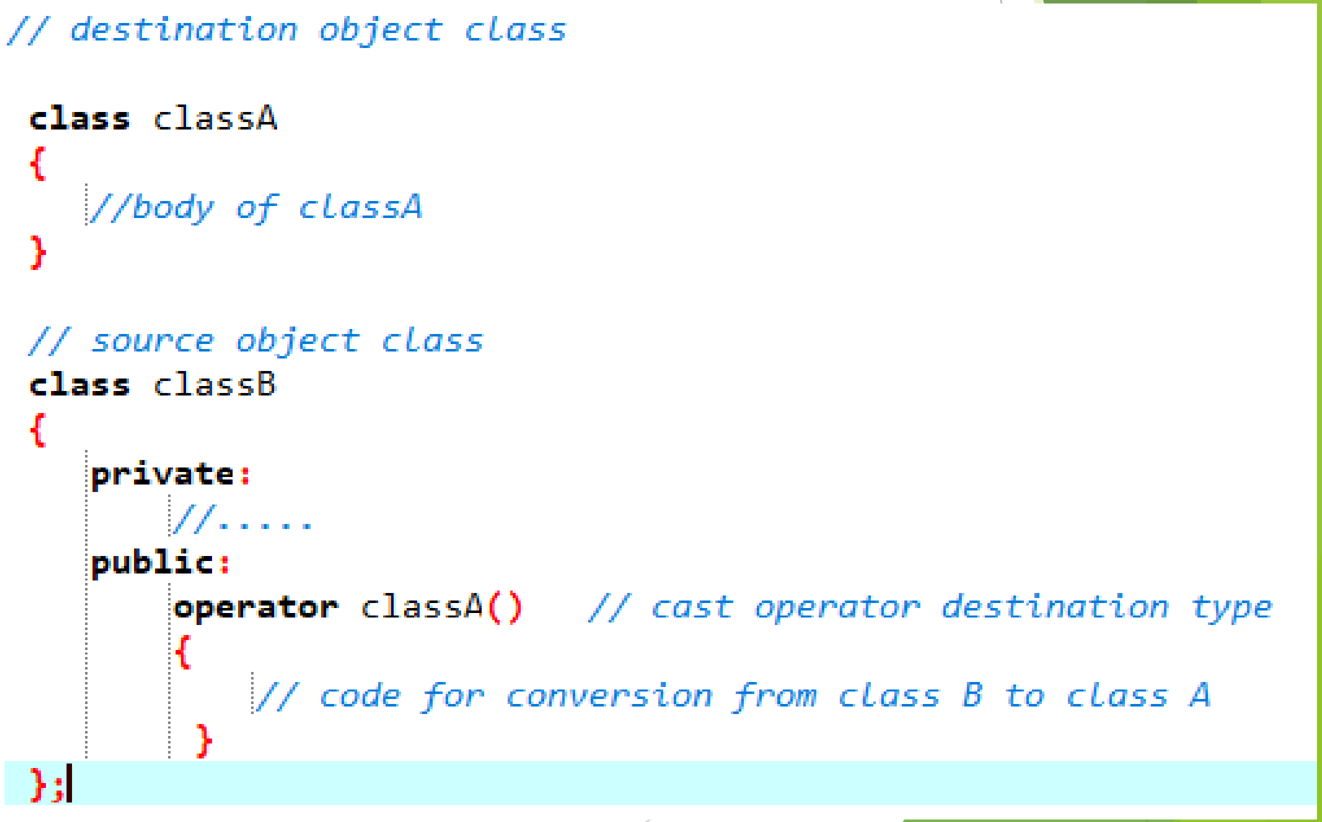
- Consider the example:

```
Class A objA;
ClassB objB;
//…………
objA= ObjB;
```

- Conversion function can be defined in classA(destination) or classB(source).
- If the conversion function is to be defined in classA, one argument constructor is used or
- if the conversion function is to be defined in source the conversion operator function should be used.

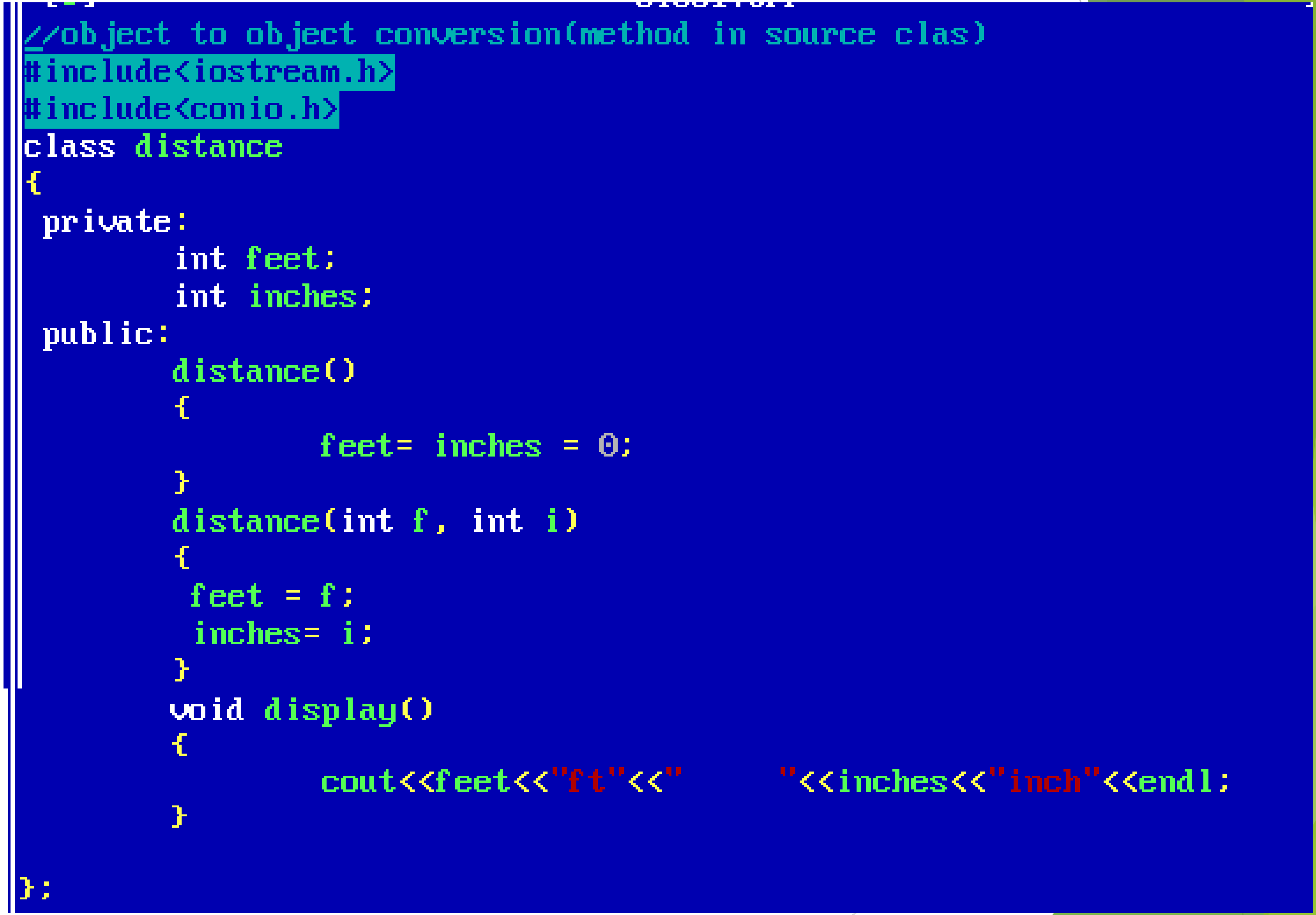
Contd..

- Routine in source class: The syntax of conversion from user-defined to user –defined when conversion is specified in the source class:



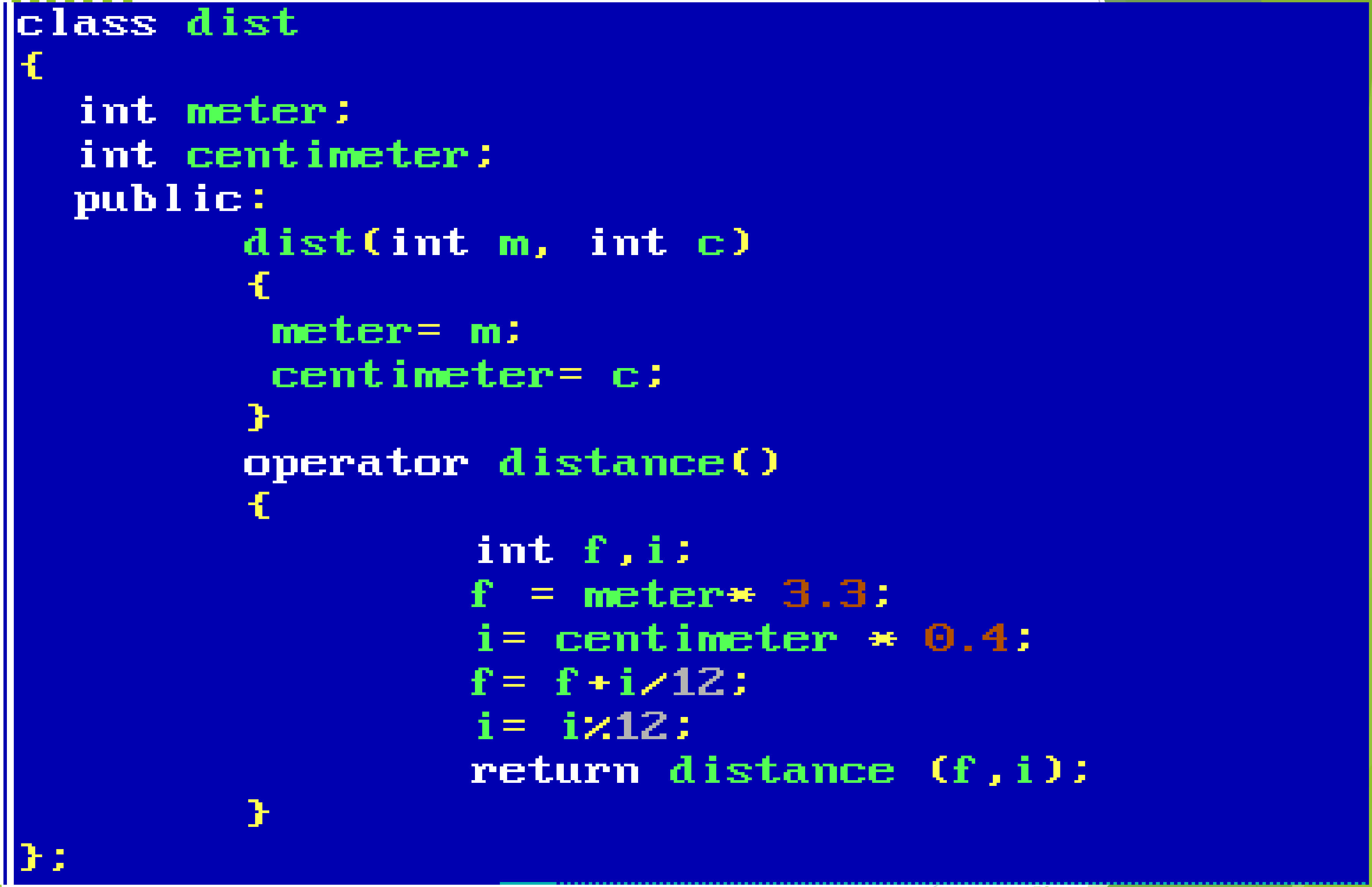
```
// destination object class

 class classA
 {
    //body of classA
 }

 // source object class
 class classB
 {
    private:
        //.....
    public:
        operator classA()   // cast operator destination type
        {
            // code for conversion from class B to class A
         }
 };
```

Contd..



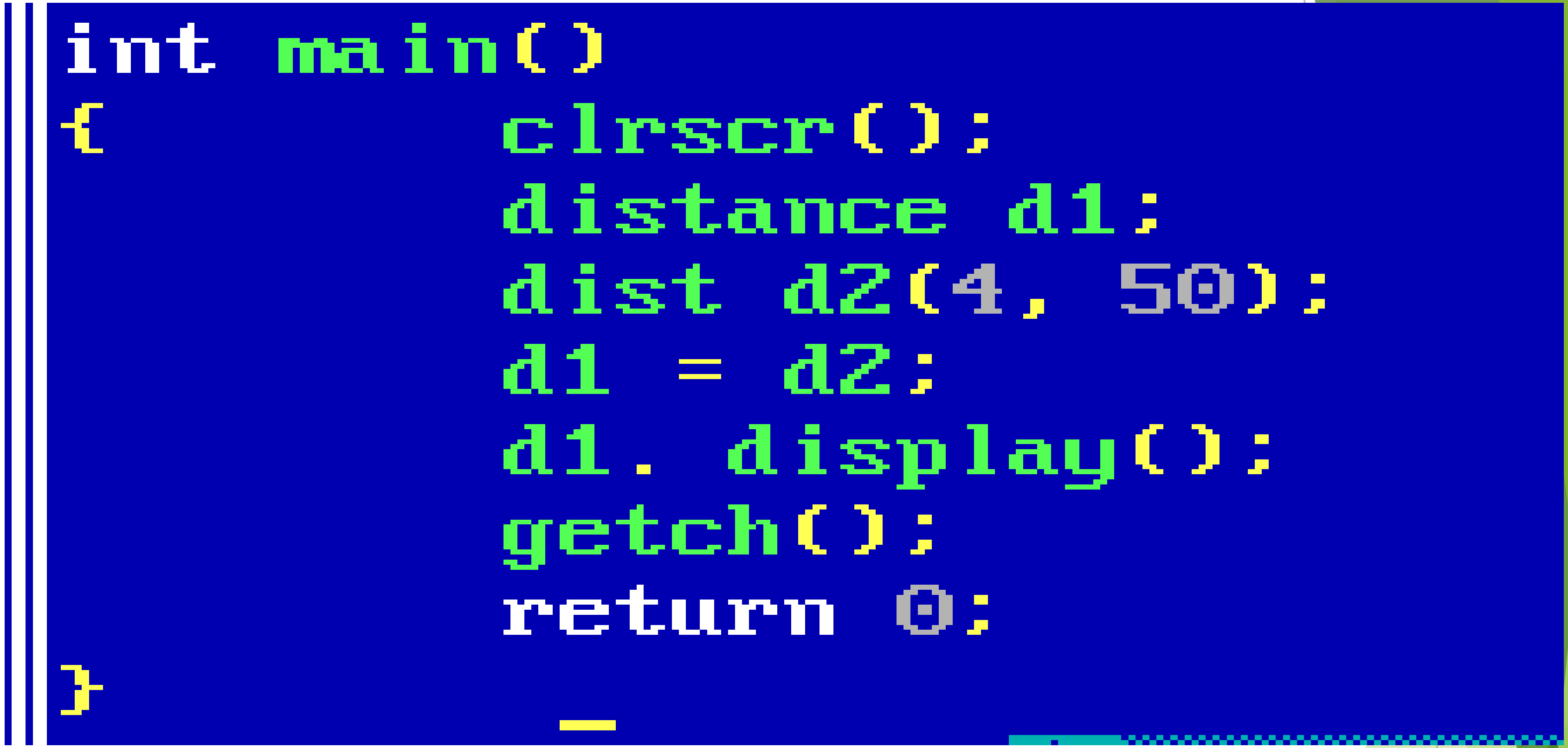
```
//object to object conversion(method in source clas)
#include<iostream.h>
#include<conio.h>
class distance
{
 private:
        int feet;
        int inches;
 public:
        distance()
        {
                feet= inches = 0;
        }
        distance(int f, int i)
        {
         feet = f;
         inches= i;
        }
        void display()
        {
                cout<<feet<<"ft"<<"     "<<inches<<"inch"<<endl;
        }

};
```

Contd



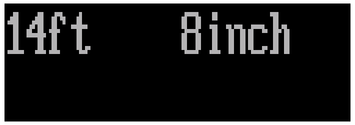
```
class dist
{
   int meter;
   int centimeter;
   public:
         dist(int m, int c)
         {
          meter= m;
          centimeter= c;
         }
         operator distance()
         {
                 int f,i;
                 f = meter* 3.3;
                 i= centimeter * 0.4;
                 f= f+i/12;
                 i= i%12;
                 return distance (f,i);
         }
};
```

Contd..



```
int main()
{          clrscr();
           distance d1;
           dist d2(4, 50);
           d1 = d2;
           d1. display();
           getch();
           return 0;
}
```

- **Output:**



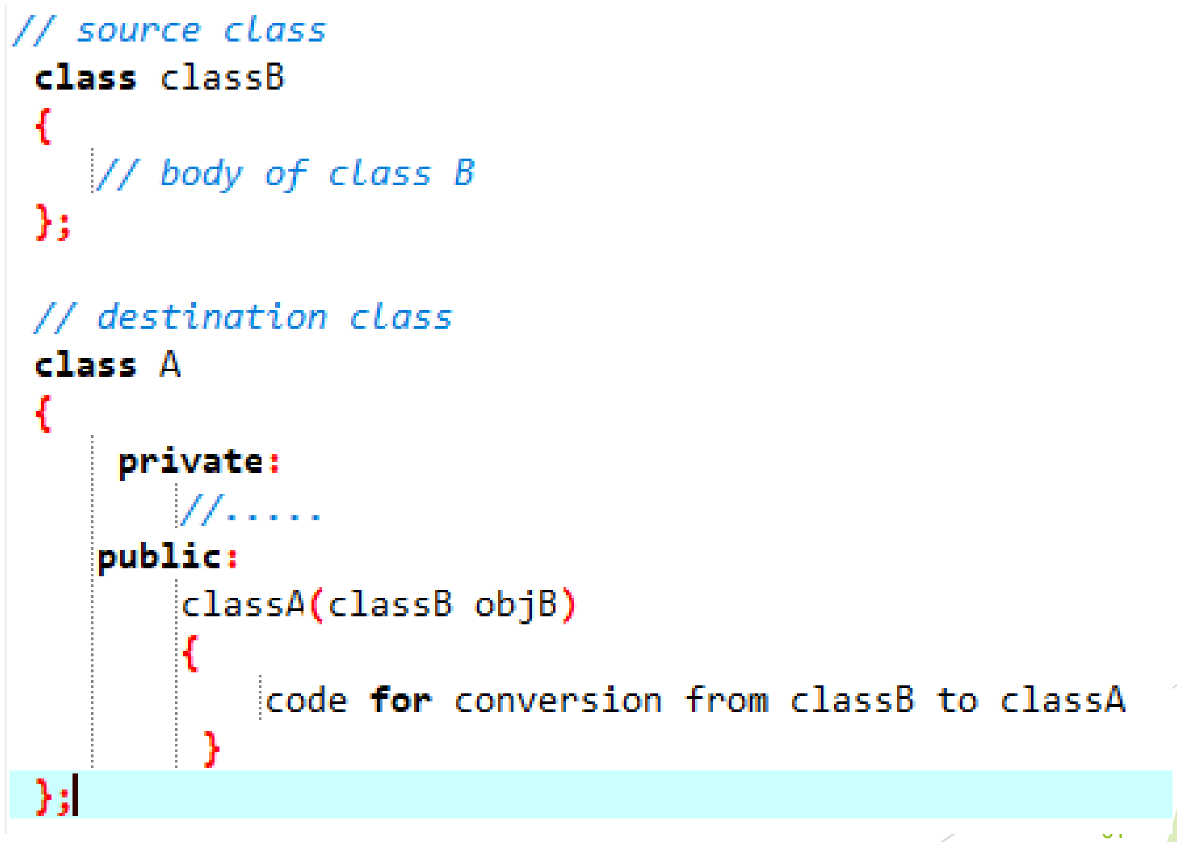
```
14ft    8inch
```

Contd..

- Routine in destination class:
  - In this case, it is necessary that the constructor be placed in the destination class.
  - This constructor is a single argument constructor and serves as instruction for converting the argument's type to the class type of which it is a member.

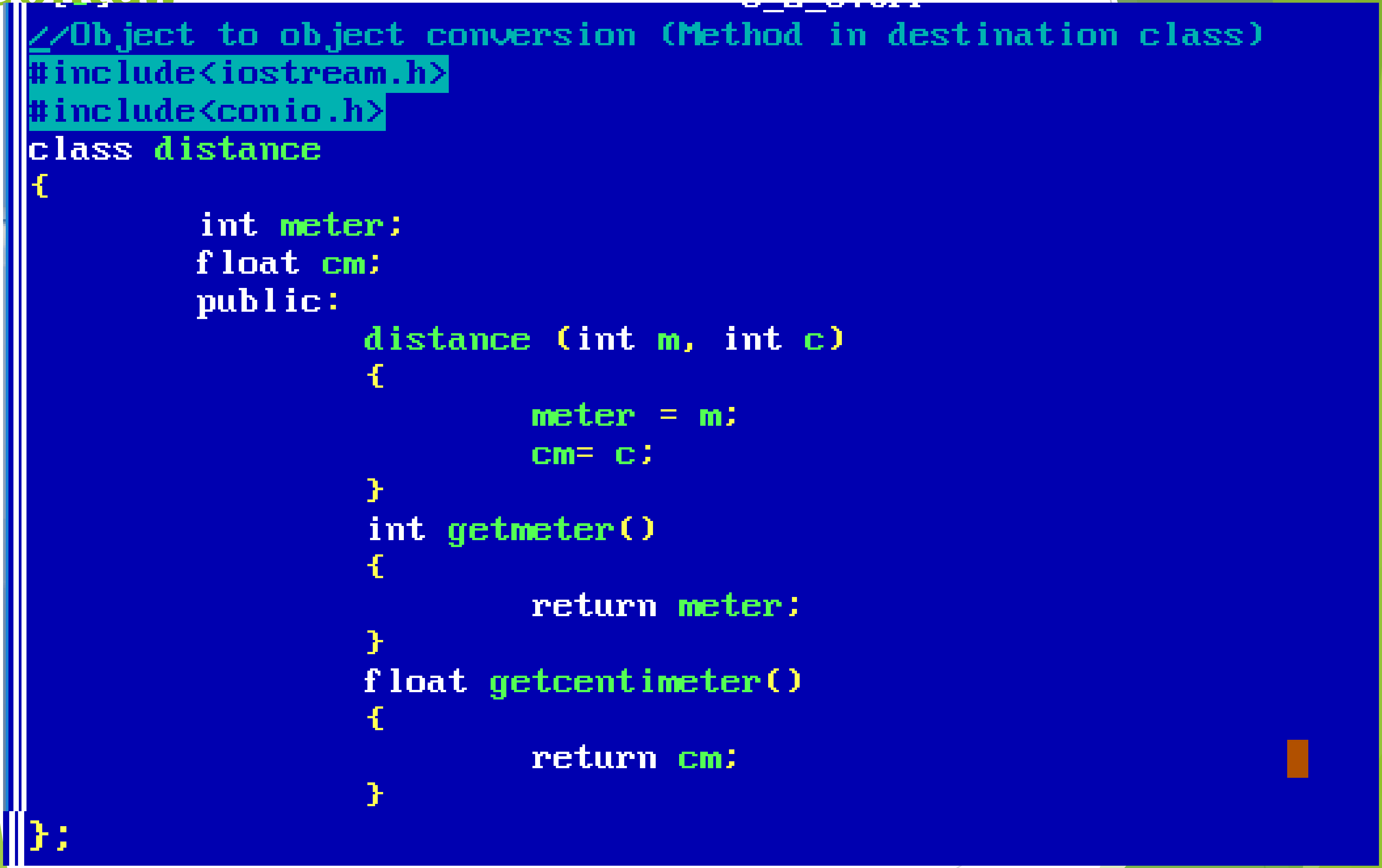
Contd..

- Following is the syntax of conversion when conversion function is in destination class:



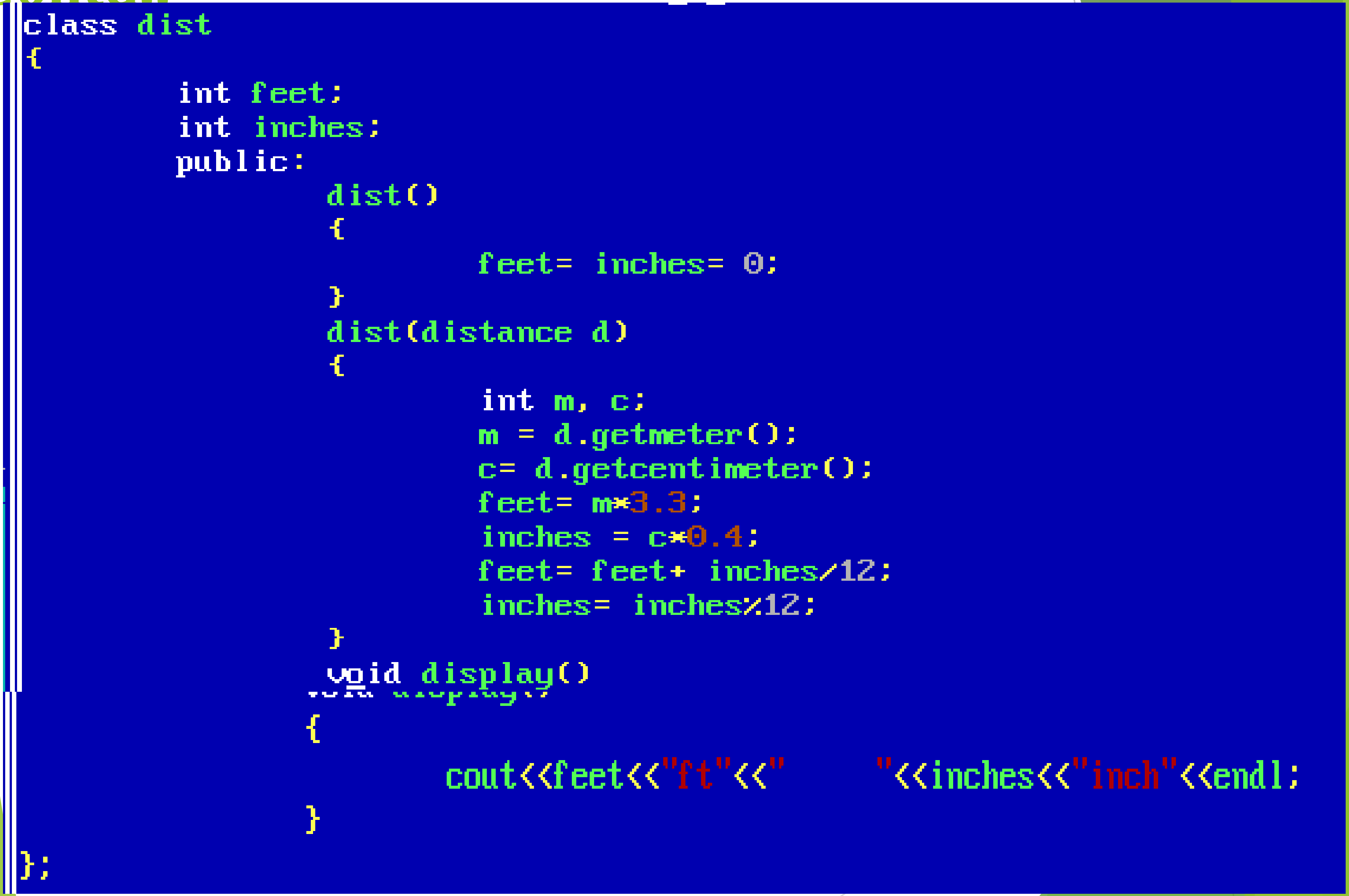
```
// source class
 class classB
 {
    // body of class B
 };

 // destination class
 class A
 {
     private:
         //.....
    public:
        classA(classB objB)
        {
            code for conversion from classB to classA
         }
 };
```

Contd..



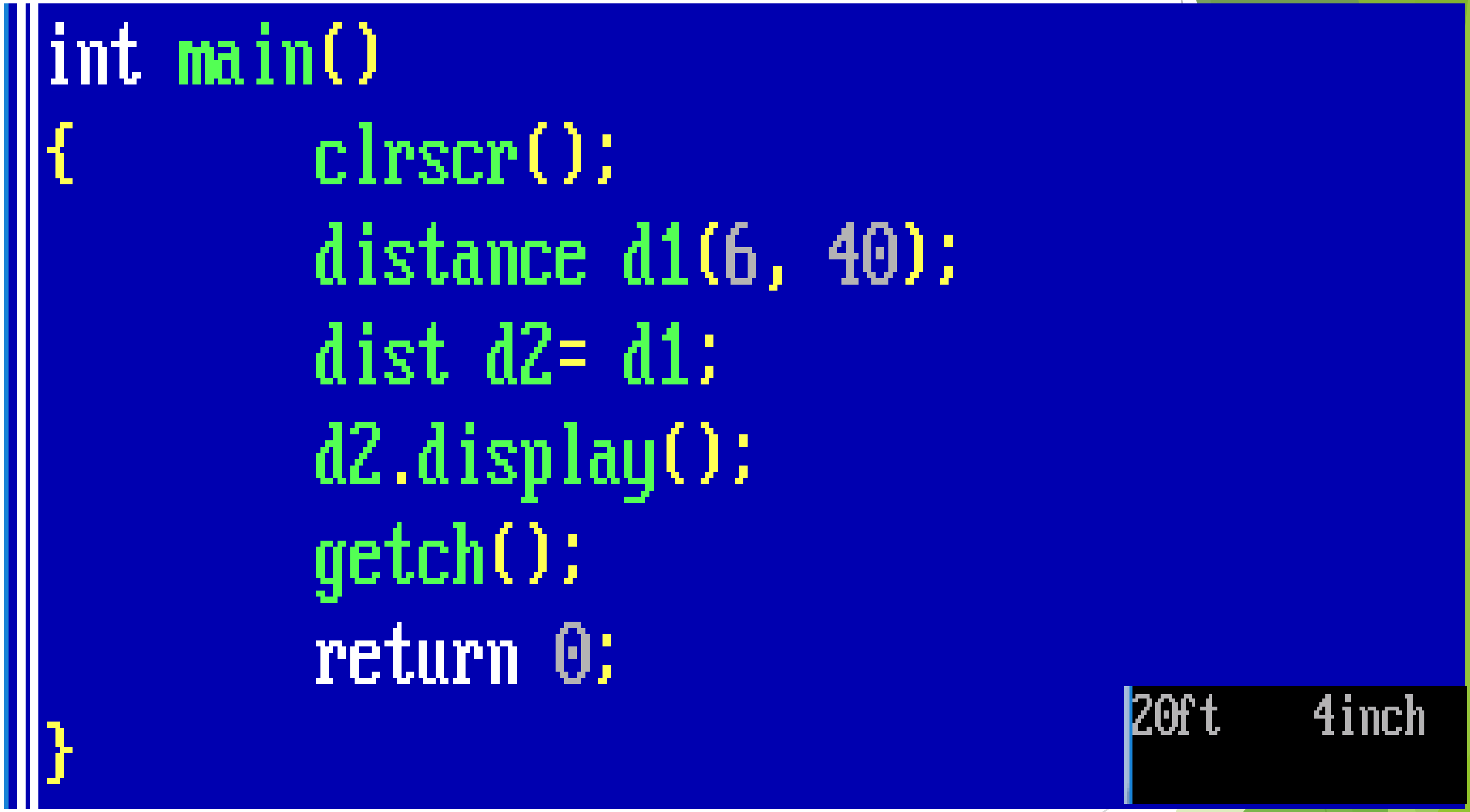
```
//Object to object conversion (Method in destination class)
#include<iostream.h>
#include<conio.h>
class distance
{
        int meter;
        float cm;
        public:
                distance (int m, int c)
                {
                        meter = m;
                        cm= c;
                }
                int getmeter()
                {
                        return meter;
                }
                float getcentimeter()
                {
                        return cm;
                }
};
```

Contd..



```
class dist
{
        int feet;
        int inches;
        public:
                dist()
                {
                        feet= inches= 0;
                }
                dist(distance d)
                {
                        int m, c;
                        m = d.getmeter();
                        c= d.getcentimeter();
                        feet= m*3.3;
                        inches = c*0.4;
                        feet= feet+ inches/12;
                        inches= inches%12;
                }
                void display()
               {
                     cout<<feet<<"ft"<<"     "<<inches<<"inch"<<endl;
               }
};
```

Contd..


```
int main()
{       clrscr();
        distance d1(6, 40);
        dist d2= d1;
        d2.display();
        getch();
        return 0;
}
```

```
20ft    4inch
```

Homework

- What is operator overloading? What are the benefits of operator overloading in C++? How is operator overloading different from function overloading.
- Which operators are not allowed to be overloaded? Why?
- What are the differences between overloading a unary operator and that of a binary operator? Illustrate with suitable examples.
- What is difference between overloading a unary operator and that of a binary operator? Illustrate with suitable examples.
- Write a program to convert polar co-ordinate into rectangular coordinate using conversion/cast function.

Contd..

- Design a class Matrix of dimension 3x3. overload + operator to find sum of two materics.
- Write a program that decreases an integer value by 1 by overloading – - operator.
- Write a program that increases an integer value by 1 by overloading ++ operator.
- Write a program to find next element of Fibonacci series by overloading ++ operator.
- Why data conversion is needed? Write a program to convert kilogram into gram using user defined to user defined data conversion.(1kg = 100gm).
- Write a program to convert an object of dollar class to object of rupees class. Assume that dollar class has data members dol and cent an rupees class have data members rs and paisa.
- Write a program to convert object from class that represents temperature in Celsius scale to object of a class that represents it in Fahrenheit scale.
- Write a program to convert object from a class that represents weight of gold in Nepal tola, to object of a class that represents international gold measurement of weight in gram scale(1 tola = 11.664 gram)

Thank You !