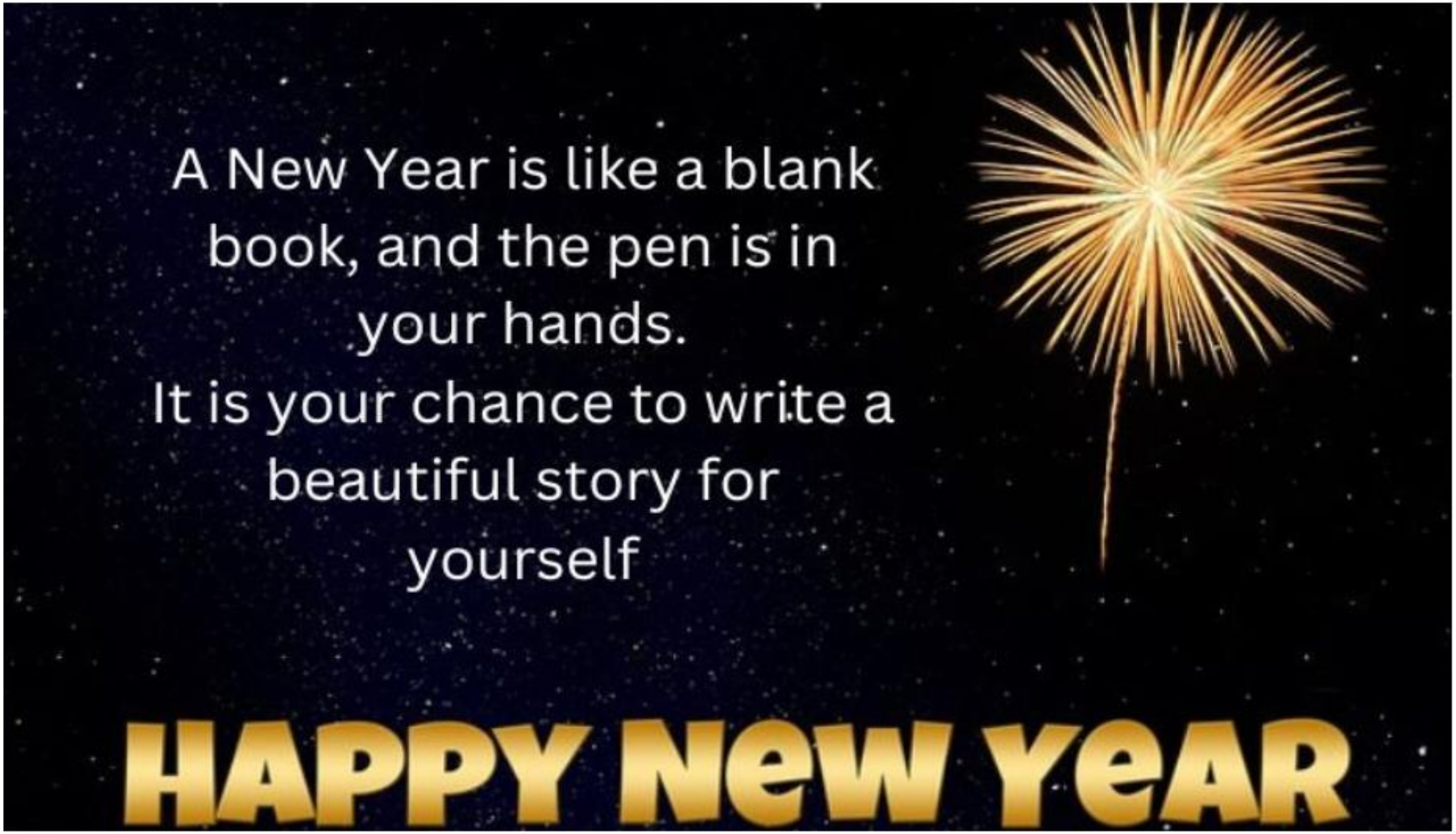
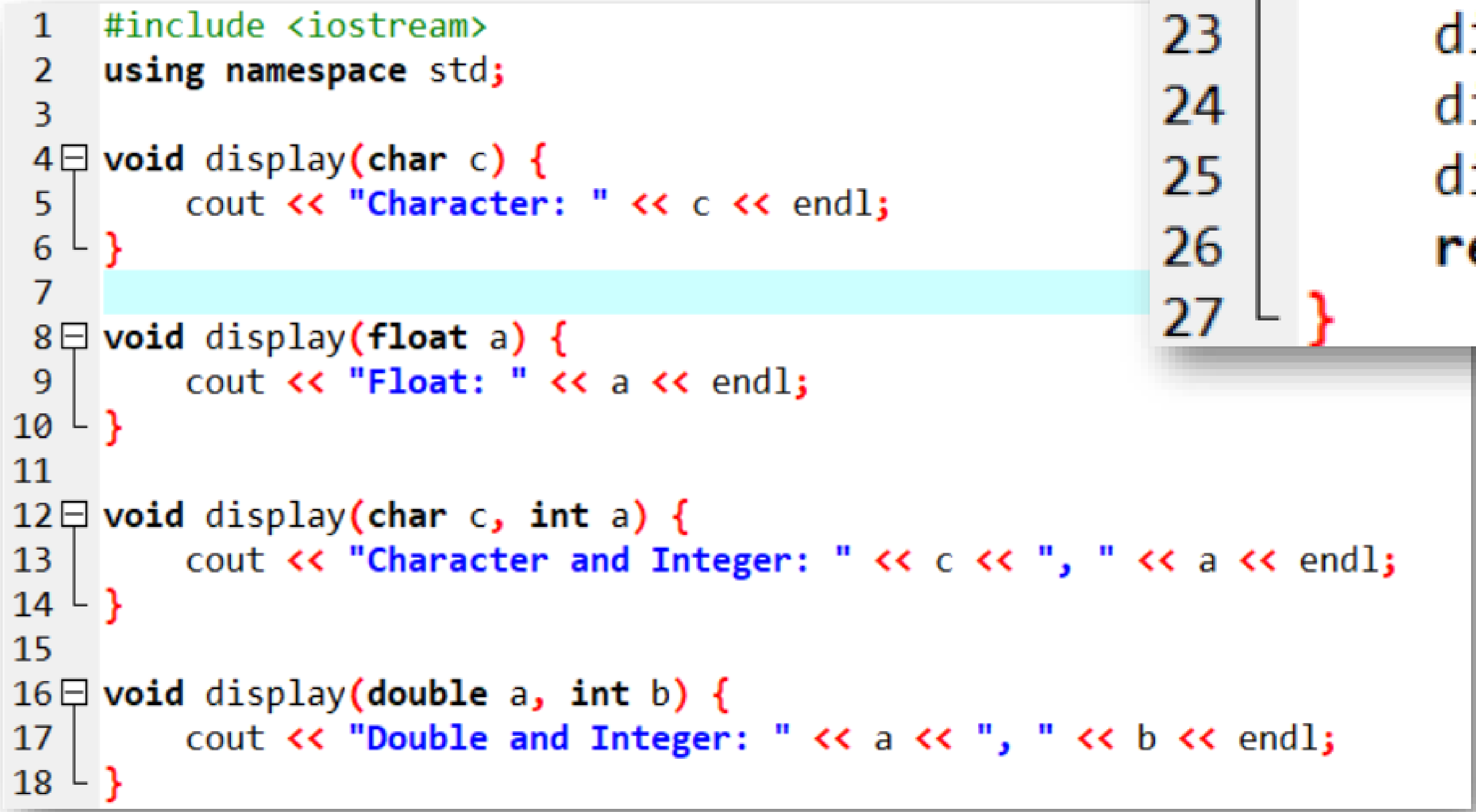
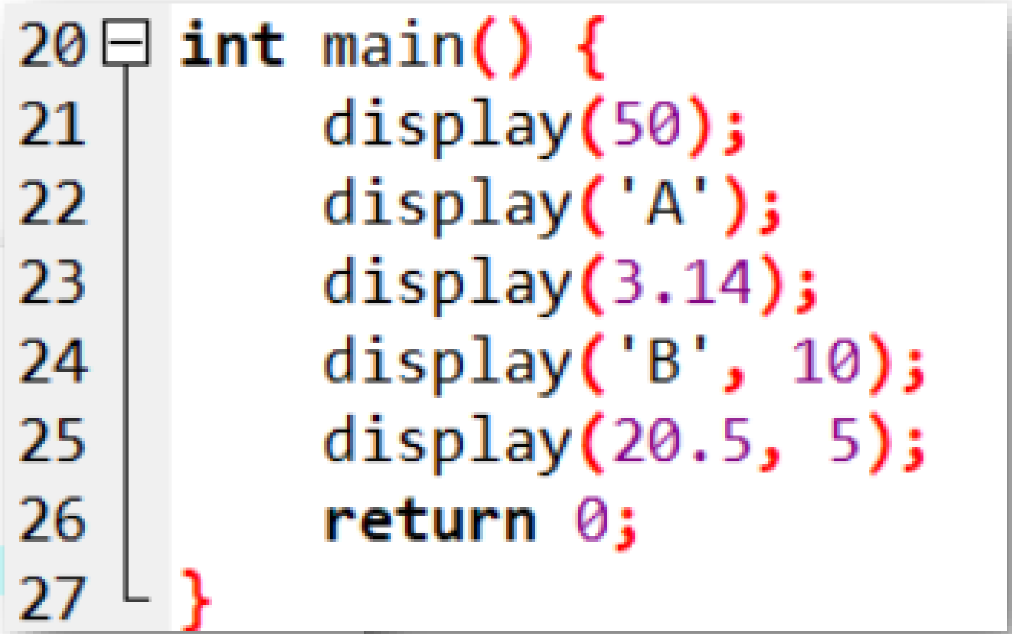

A New Year is like a blank book, and the pen is in your hands.
It is your chance to write a beautiful story for yourself
HAPPY New YeAR

EXERCISE 1

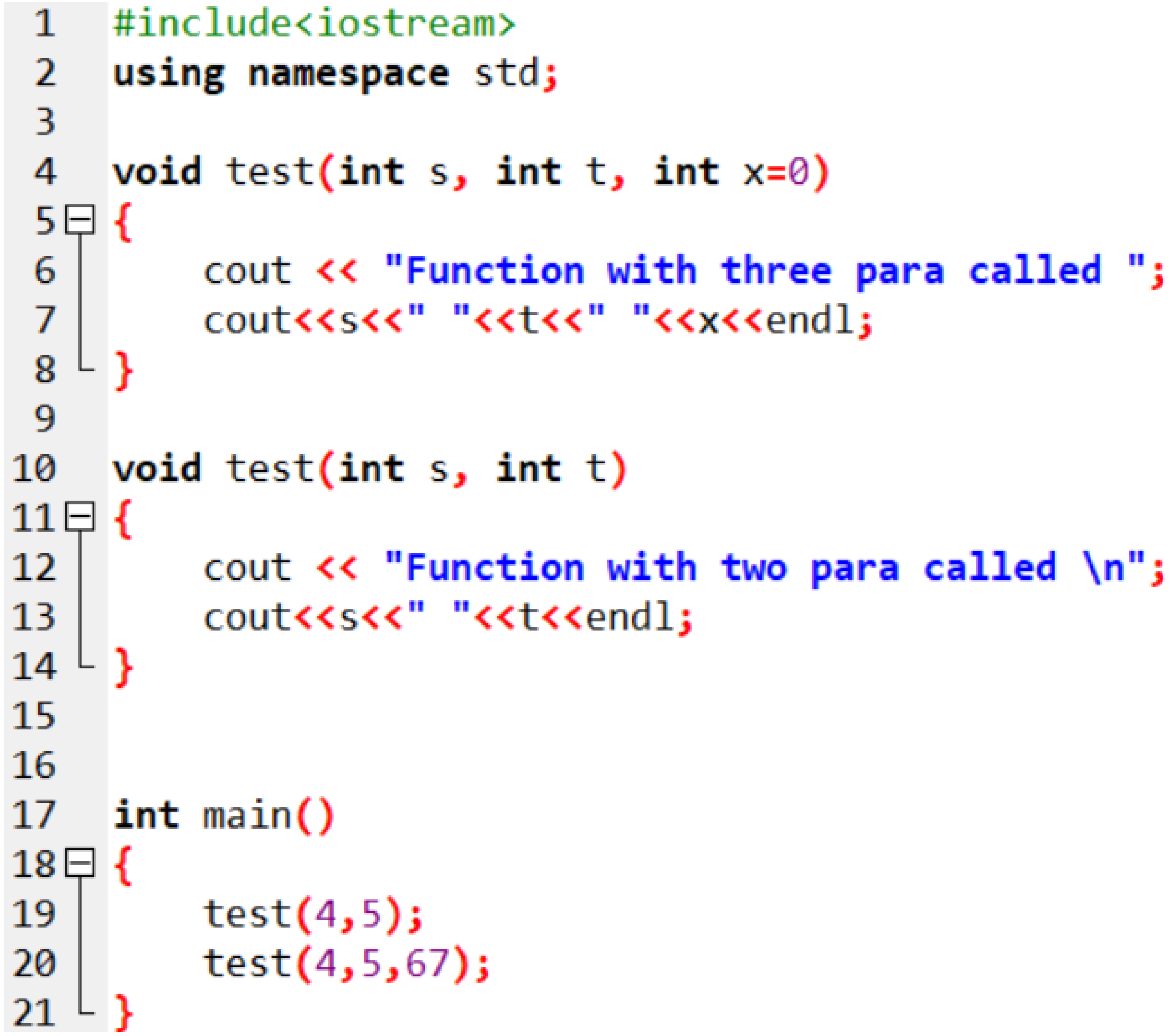


```cpp
#include <iostream>
using namespace std;

void display(char c) {
    cout << "Character: " << c << endl;
}

void display(float a) {
    cout << "Float: " << a << endl;
}

void display(char c, int a) {
    cout << "Character and Integer: " << c << ", " << a << endl;
}

void display(double a, int b) {
    cout << "Double and Integer: " << a << ", " << b << endl;
}
```



```cpp
int main() {
    display(50);
    display('A');
    display(3.14);
    display('B', 10);
    display(20.5, 5);
    return 0;
}
```

EXERCISE 2



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

void test(int s, int t, int x=0)
{
    cout << "Function with three para called ";
    cout<<s<<" "<<t<<" "<<x<<endl;
}

void test(int s, int t)
{
    cout << "Function with two para called \n";
    cout<<s<<" "<<t<<endl;
}


int main()
{
    test(4,5);
    test(4,5,67);
}
```

HOME WORK

- Write a program that calculates the area of different geometric shapes (circle, rectangle, and triangle) using function overloading.

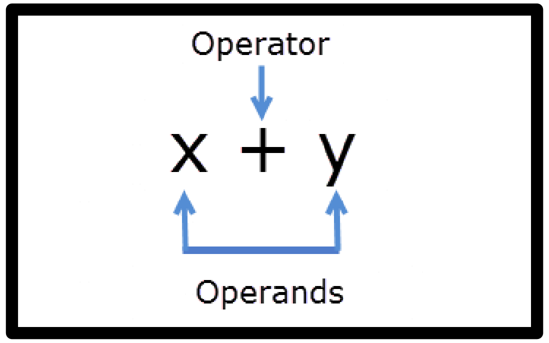
  Hint: Define a class *Geometry* with overloaded functions named *calculateArea*

```
Select a shape to calculate the area:
1. Circle
2. Rectangle
3. Triangle
Enter your choice: 1
Enter the radius of the circle: 5
Area of the circle: 78.54

Select a shape to calculate the area:
1. Circle
2. Rectangle
3. Triangle
Enter your choice: 2
Enter the length and breadth of the
rectangle: 4 6
Area of the rectangle: 24.00

Select a shape to calculate the area:
1. Circle
2. Rectangle
3. Triangle
Enter your choice: 3
Enter the base and height of the triangle: 5 7
Area of the triangle: 17.50
```

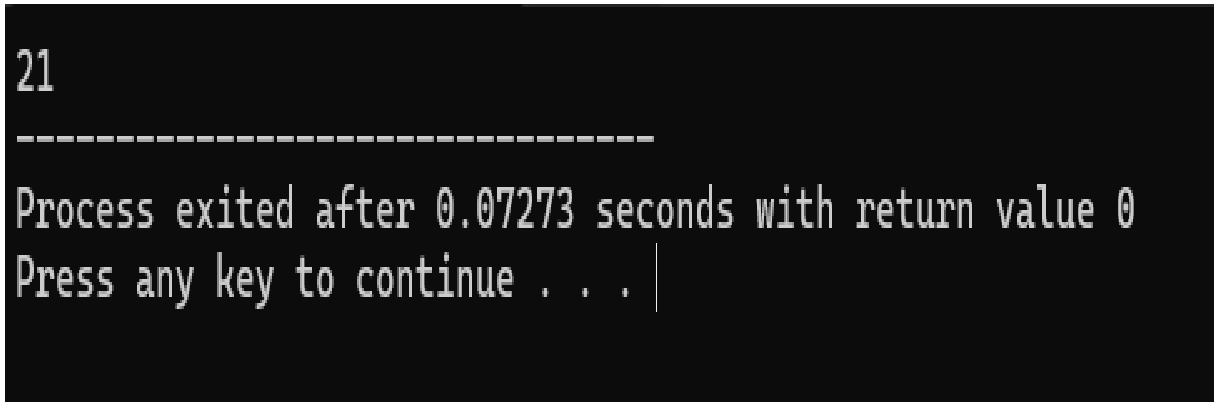
OPERATOR OVERLOADING

What is meant by Operator Overloading?

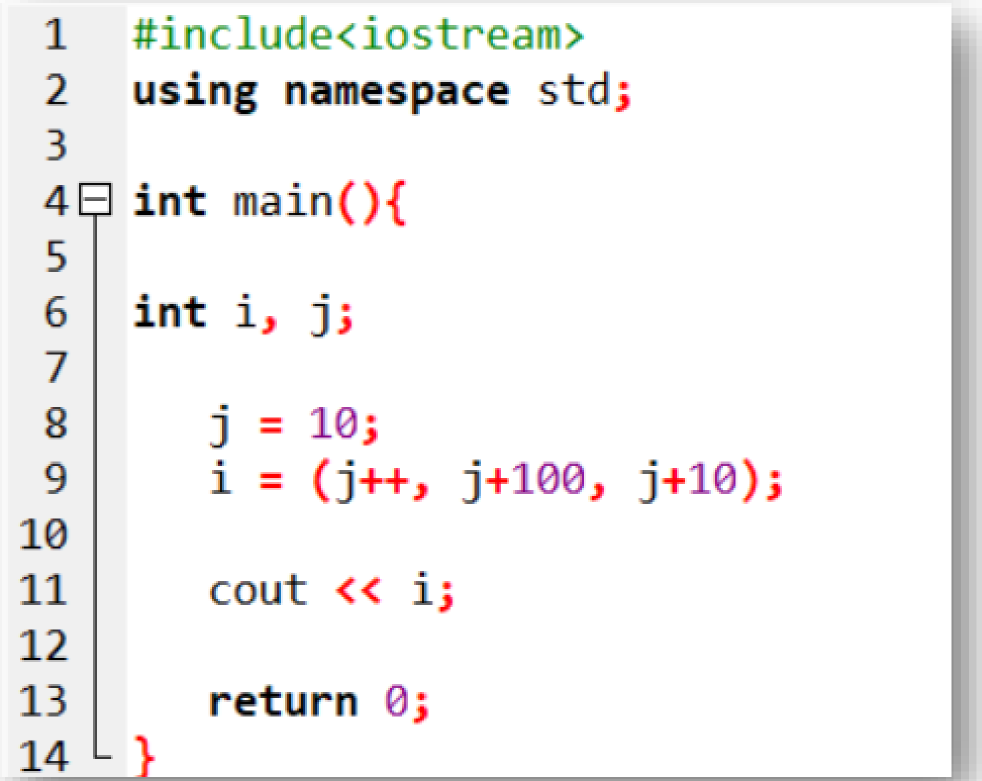
WHY–OPERATOR OVERLOADING



- Operators work with operands.
- Increment operator can be used with int values or int type variables.
  Example: count ++.

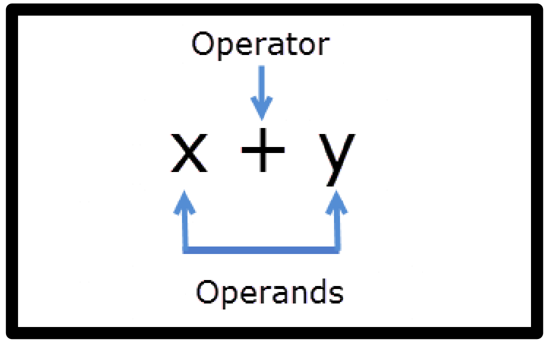


```
21
--------------------------------
Process exited after 0.07273 seconds with return value 0
Press any key to continue . . .
```



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

int main(){

int i, j;

    j = 10;
    i = (j++, j+100, j+10);

    cout << i;

    return 0;
}
```

WHY–OPERATOR OVERLOADING



- Operators work with operands.
- Increment operator can be used with int values or int type variables.
  Example: count ++.

Question: Can we use the above operator with objects directly?

Example: Create an object (N1) from Numbers class and increment all the variables inside the class?

We can't use operators on objects as they are NOT applied for classes.

However, these operators can be overloaded to get special functions done.

```
class Numbers{
        int x, y;

        public:
        Numbers():x(0),y(0){
        }
};
```

```
int main(){
        Numbers N1;
        ++N1;
}
```

WHAT IS OPERATOR OVERLOADING

- In C++, we can make operators to work for user defined classes.
- Allows to change the use of operators such as +, -, ++, & etc.
- Applied for user defined classes to create the own code to give them new functionality.

Example 1: Use '+' operator to concatenate two strings together

str1 = str2 + str3, without calling append() function.

Example 2: Override '++' operator not to increment value by 1, but, by 100.

HOW TO DEFINE OPERATOR OVERLOADING

- Operators are overloaded by defining an operator function.
  - An operator function defines the specific operations that the overloaded operator will perform relative to the class it is designed to work on.
- We must use keyword "operator" before the operator we want to overload
- Works as a function therefore the return type must be defined.

Syntax: operator function

```
returnType operator operatorSymbol(parameter)
 {
   // ...
 }
```

- BEST PRACTISE: Although you are free to perform any operation you want inside an operator function it is usually best to stay within the context of the operator's normal use.

DIFFERENCE BETWEEN OPERATOR FUNCTIONS AND NORMAL FUNCTIONS

SIMILARITY:

- Operator functions are same as normal functions.

DIFFERENCE:

- Name of an operator function is the operator keyword followed by symbol of operator
- Operator functions are called when the corresponding operator is used.

Operator Function: N1++;

Normal Function: N1.numIncrement();

CAN WE OVERLOAD ALL OPERATORS?

NO.

Operators that can NOT be Overloaded:

- The dot operator (.),
- The scope resolution operator (::),
- The dereferencing operator (*),
- The ternary conditional operator (? :) and
- The sizeof operator

TYPES OF OPERATOR OVERLOADING

There are **three approaches**

- Overloading unary operator.
- Overloading binary operator.
- Overloading operators using a friend function.

What is a unary operator?

- Operators act on only one operand.
- Examples: increment (++), decrement (--), minus (-) : -33.

What is a binary operator?

- Operators act on two operands.
- Examples: +, -, <=, >, ==,

OVERLOADING UNARY OPERATOR

- No arguments should be passed.
- Works only with one class objects.
- Overloading an operator operating on a single operand.

OVERLOADING UNARY OPERATOR: EXAMPLE 1

```
class Number{
	int num;

	public:
	Number(int a):num(a){
	}

	void display(){
		cout<<"Number variable: "<<num<<endl;
	}

	void operator -(){
		this->num=-num;
	}
};
```

```
int main()
{
	Number myNum1(6);
	-myNum1;

	myNum1.display();
}
```

OVERLOADING UNARY OPERATOR: EXAMPLE 2

```
class Counter{
        unsigned int count;

        public:
                Counter():count(0) //constructor
        { }

        void operator++(){
                ++count;
                }

        int getCounter()
        {
                return count;
                }
};
```

Prefix Increment

```
int main()
{
        Counter c1;
        ++c1;              //Prefix

        cout<<"Count= "<<c1.getCounter()<<endl;

}
```

OVERLOADING UNARY OPERATOR: EXAMPLE 3

Postfix Increment

The variable is incremented after its value is used in the expression

Not an Argument

```
Counter operator++(int) {     //postfix increment
        return Counter(count++);
        }
```

THE INCREMENT OPERATOR OVERLOADING

The general forms of the prefix and postfix **operator++()** functions

```
// Prefix increment operator   ++ObjectName
returnType ClassName::operator++()
{
  // …
 }
```

```
// Postfix increment operator   ObjectName++
returnType ClassName::operator++(int)
{
   // …
}
```

THE DECREMENT OPERATOR OVERLOADING

The general forms of the prefix and postfix **operator--()** functions

```
// Prefix decrement operator   --ObjectName
returnType ClassName::operator--()
{
  // ...
 }
```

```
// Postfix decrement operator   ObjectName--
returnType ClassName::operator--(int)
{
   // ...
}
```

OVERLOADING BINARY OPERATOR

- Overloading of an operator operating on two operands.
- One argument should be passed to the operator function.

PRACTICE EXERCISE

Complex Number Addition & Subtraction

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

class Complex{
        int real;
        int imagi;

        public:
        Complex():real(0),imagi(0){

        cout<<"Inside the constructor"<<endl;}

        Complex(int x, int y):real(x),imagi(y){}

        Complex(Complex& c){

        cout<<"User Defined copy constructor"<<endl;
        real=c.real;
        imagi=c.imagi;
        }

        Complex operator +(Complex& c){
                Complex tempc;
                tempc.real=real+c.real;
                tempc.imagi=imagi+c.imagi;
                        return (tempc);
        }
```

```
        Complex operator -(Complex& c){
                Complex tempc;
                tempc.real=real-c.real;
                tempc.imagi=imagi-c.imagi;
                        cout<<c.real<<" "<<c.imagi<<endl;
                return (tempc);
        }


        void display(){
        if(imagi < 0)
            cout << "Complex No: "<< real << imagi << "i"<<endl;
         else
            cout << "Complex No: " << real << "+" << imagi << "i"<<endl;   }

        ~Complex()
        {
                cout<<"Deallocate Memory"<<endl;
        }

};

int main(){
    Complex C1(3,4);
        C1.display();
    Complex C2(2,3);
        C2.display();
                C1=C2-C1;
                C1.display();
                Complex C3=C1;
                C3.display();
}
```

ASSIGNMENT OPERATOR

- Compiler by default automatically defines an operator overloading for the assignment operator.
- Used to copy values from one object to another *already existing object*.

  Example: obj1 and obj2

  obj1 = obj2 (copy all the data values of obj2 into obj1)
- It works the same as copy constructor
- only shallow copy (makes a bitwise copy of the class instance) is done hence may cause problem while working with pointers.

OVERLOADING THE ASSIGNMENT OPERATOR

Assignment vs Copy constructor

- The purpose is almost equivalent
  - both copy one object to another.
  - The copy constructor initializes new objects, whereas the assignment operator replaces the contents of existing objects.
- A new object needs to be created before the copying - Copy constructor is called.
- No need to create a new object before the copying can occur - Assignment operator is called.
- The copy constructor creates a separate memory block for the new object.
- The assignment operator does not make new memory space (uses the reference variable to point to the previous memory block).

OVERLOADED ASSIGNMENT OPERATOR

```
ClassName& ClassName::operator=(const ClassName& op)
{
   //…………………..
    return *this;
 }
```

OPERATOR FUNCTIONS: MEMBER/NON-MEMBER

- Operator functions can be either members or non-members of the class they operate on.
- Non-member operator functions:
  - Should be defined as friend functions
  - Need access to the class's private data members.
- The way an operator function is written differs depending on whether it is a member function or a friend function.

FRIEND FUNCTIONS

- Data hiding - A fundamental concept of object-oriented programming.
  - Restricts the access of private members from outside of the class.
  - protected members can only be accessed by derived classes and are inaccessible from outside.

FRIEND FUNCTIONS

- What is a friend function?
  - A friend function is a function that is not a member of a class but is granted access to its private and protected members.
- Why use a friend function?
  - Friend functions are useful when you need a function to operate on private data of a class but want to keep it non-member for modularity or external functionality.
- How do you define and declare a friend function?
  - Declare the function inside the class with the friend keyword.
  - Define it outside the class, similar to a regular function.

FRIEND OPERATOR FUNCTIONS

Binary operator overloaded using a friend function:

- the **left operand is passed in as the first parameter** and the **right operand is passed in as the second parameter**.

```
friend Complex& operator+(const Complex& ob1, const Complex& ob2);
```

Unary operator overloaded using a friend function:

- the operand is passed to the function as a parameter.

```
friend Counter& operator++(Counter& ob);
```

It is common for an overloaded operator function's return type to be the same as the class it operates on, so that several operations may be chained together.

An operator function that changes the value of its invoking operand normally returns a reference to that operand.

MEMBER OPERATOR FUNCTIONS

Binary operator overloaded using a member function:

- the left operand generates the function call, and the right operand is passed to the function as a parameter.

Unary operator overloaded using a member function:

- the operand generates the function call, parameter is not required.

THE ASSIGNMENT OPERATOR

- The assignment operator must be overloaded as a member function, not a friend function.
  - Justify (H/W)
- The left operand invokes the **operator=()** function and accepts a constant reference to the right operand as an argument.

THE OUTPUT OPERATOR

- The output operator must be overloaded as a friend function because its left operand is an output stream such as **cout.**
- The general form of the **operator<<()** function is

```
ostream& operator<<(ostream& out, const ClassName& op)
{
  // ...
}
```

- The **operator<<()** function returns a reference to an **ostream** object so that output operations may be chained together.

```
// Output operator
ostream& operator<<(ostream& out, const Money& amount)
{
   out << '$' << amount.dollars << '.';
   if (amount.cents < 10)
      out << '0';
   out << amount.cents;
   return out;
}
```

THE INPUT OPERATOR

- The input operator must be overloaded as a friend function because its left operand is an input stream such as **cin.**
- The general form of the **operator>>()** function is

```
istream& operator>>(istream& in, ClassName& op)
{
  // ...
 }
```

- The **operator>>()** function returns a reference to an **istream** object so that input operations may be chained together.

```
// Input operator
istream& operator>>(istream& in, Money& amount)
{
  unsigned int d;     // Number of dollars to be read
  unsigned int c;     // Number of cents to be read
  char dollarSign;    // Dollar sign character preceding d
  char decimalPoint; // Decimal point character between
                      // d and c
  in >> dollarSign >> d >> decimalPoint >> c;
  amount = Money(d, c);
  return in;
}
```

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

class Fraction {

private:
        int numerator;
        int denominator;

public:
        // constructor
        Fraction(int x = 0, int y = 1) {
                numerator = x;
                denominator = y;
        }

        friend istream& operator>>(istream& cin, Fraction& c){
                cin >> c.numerator >> c.denominator;
                return cin;
        }

        friend ostream& operator<<(ostream&, Fraction& c)     {
                cout << c.numerator << "/" << c.denominator;
                return cout;
        }
};
```

```
int main()
{
        Fraction x;
        cout << "Enter a numerator and denominator of "
                        "Fraction: ";
        cin >> x;
        cout << "Fraction is: ";
        cout << x;
        return 0;
}
```

DATA CONVERSION

DATA CONVERSION

- Assignments between the same data types are handled by the compiler.
  - Basic types / user-defined types

but….

Example:

```
a=b;
myObj1=myObj2;
```

- What happens when the variables on different sides of the = are of different types?
- Basic types – Compiler handles automatically
- Compiler does not handle automatically
  - Conversions between basic types and user-defined types, and
  - Conversions between different user-defined types.

CONVERSIONS BETWEEN BASIC TYPES

- Basic Data Types
  - Float to int
  - Float to double
  - Char to float
  - Int to Char etc….
- Convert the value of floatvar (expressed in floating-point format) to an integer format and assigned to intvar.

```
intvar = floatvar;
```

DATA TYPE CONVERSION

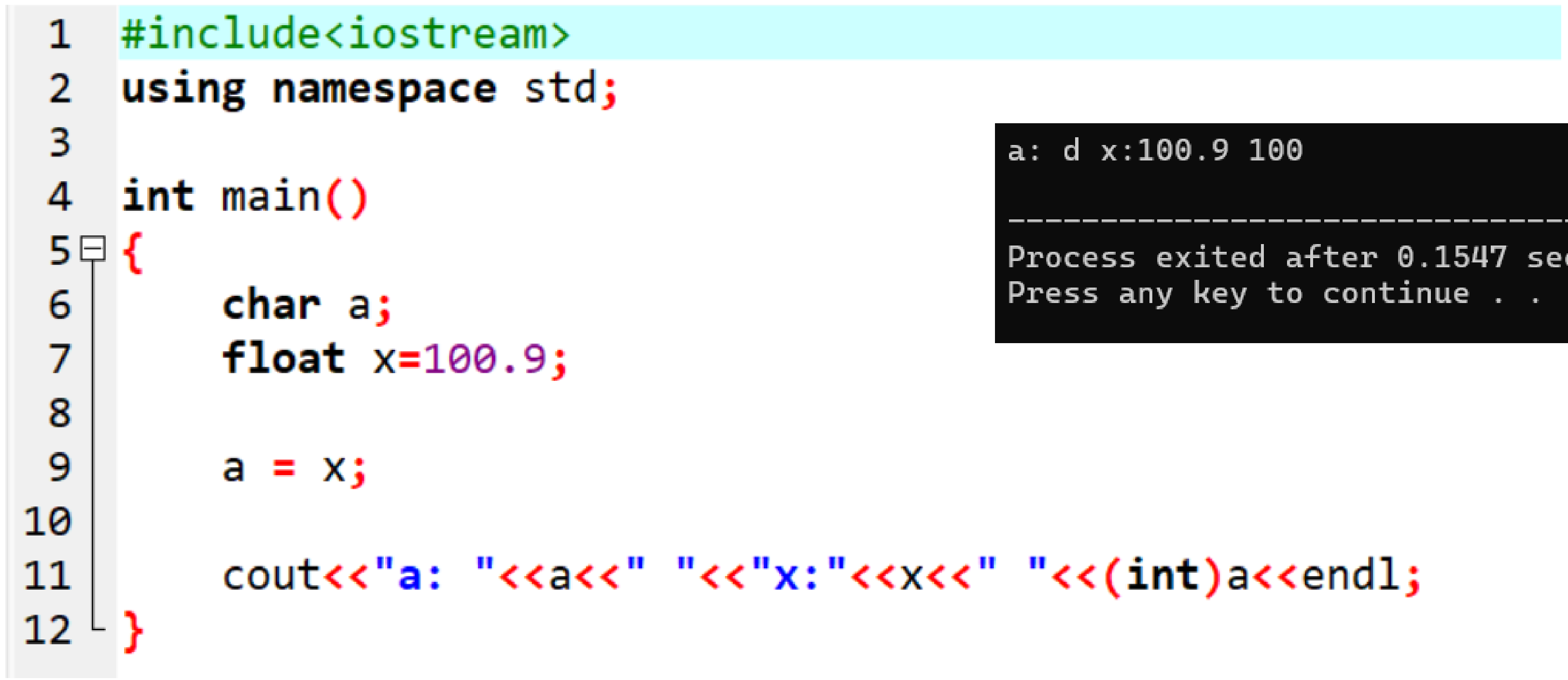
- A type conversion occurs when the compiler changes the type of an object.
- An explicit type conversion occurs when the programmer uses a cast in the code to change the type of an object.

```
double x = double(3) / 2;     // The value of x is 1.5
```

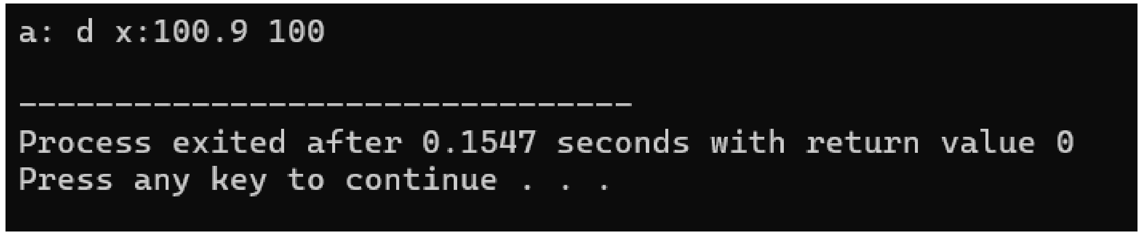
- An implicit type conversion occurs when the compiler automatically changes the type of an object.

```
int  i = 32.4;          // The value of i is 32
```

EXERCISE



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

int main()
{
	char a;
	float x=100.9;

	a = x;

	cout<<"a: "<<a<<" "<<"x:"<<x<<" "<<(int)a<<endl;
}
```



```
a: d x:100.9 100

--------------------------------
Process exited after 0.1547 seconds with return value 0
Press any key to continue . . .
```

TYPE CASTING

- Converting an expression of a given type into another type.
- **Implicit conversion**
  - Do not require any operator.
  - Automatically performed when a value is copied to a compatible type.
- **Explicit conversion**
  - There are many conversions
  - Imply a different interpretation of the value, require an explicit conversion.
- Casting operators:
  - dynamic_cast,
  - reinterpret_cast,
  - static_cast and
  - const_cast.

DATA CONVERSION

Conversion of primitive data type to user-defined

```
// Time Class
class Time {
    int hour;
    int mins;

public:
    // Default Constructor
    Time() : hour(0),mins(0){  }

    // Parameterized Constructor
    Time(int t)
    {
        hour = t / 60;
        mins = t % 60;
    }

    void Display()
    {
        cout << "Time = " << hour
             << " hrs and "
             << mins << " mins\n";
    }
};
```

```
int main()
{
    // Object of Time class
    Time T1;
    int dur = 95;

    // Conversion of int type to class type
    T1 = dur;
    T1.Display();
}
```

DATA CONVERSION

Conversion of class object to primitive data type

CONVERSION OF CLASS OBJECT TO PRIMITIVE DATA TYPE

- Convert from class object and a primitive data type.
- Need to use an overloaded casting operator function (conversion function)

The syntax:

```
operator typename()
{
  // Code
}
```

Example:

The **operator double()** converts a class object to type double,

The **operator int()** converts a class type object to type int etc..

```
class Time {
    int hrs, mins;

public:
    // Constructor
    Time(int, int);

    // Casting operator
    operator int();

};

Time::Time(int a, int b)
{
    hrs = a;
    mins = b;
}
```

```
// Used for Data conversion of class to primitive
Time::operator int()
{
    cout << "Conversion of Class"
        << " Type to Primitive Type"
        << endl;

    return (hrs * 60 + mins);
}

int main()
{
    int hour, mins;
    hour = 3;
    mins = 20;

    Time t(hour,mins);

    int duration = t;

    cout << "Total Minutes are "
        << duration << endl;
}
```

DATA CONVERSION

Conversion of one class type to another class type

```
// Time Class
class Time {
        int hr, mins;
public:
        // Constructors
        Time(int h, int m) : hr(h),mins(m){ }
        Time()
        {
                cout << "\nTime's Object Created";
        }

        int Minute()
        {
                int totalmin = (hr * 60) + mins;
                return totalmin;
        }

        void show()
        {
                cout << "Hour: " << hr << endl;
                cout << "Minute : " << mins << endl;
        }
};
```

```
// Minutes Class
class Minute {
        int mins;

public:
        // Constructors
        Minute()
        {
                mins = 0;
        }
        void operator=(Time T)
        {
                mins = T.Minute();
        }

        void show()
        {
                cout << "\nTotal Minute : "
<< mins << endl;
        }
};
```

```
int main()
{
        int hour, mins;
        hour = 3, mins = 40;

        Time T1(hour,mins);
        T1.show();

        Minute M1;
                M1.show();
        M1=T1;
                M1.show();
}
```