

Exception Handling

Faculty: SREEDIVYA I

Branch: S6 ECE

Exceptions in C++

- When executing a C++ code, different errors can occur:
 - coding errors made by the programmer,
 - errors due to wrong input, or
 - others
- When an error occurs, compiler will normally stop and generate an error message. The technical term for this is: C++ will throw an **exception** (throw an error).
- An exception is an unexpected event that occurs during program execution.
- For example,
$$\text{float divide} = 7 / 0;$$
- The above code causes an exception as it is not possible to divide a number by 0.

Exception Handling

- The process of handling these types of errors is known as exception handling.
- Exception Handling in C++ is a process to handle runtime errors.
- Using the exception handling mechanism, the control from one part of the program where the exception occurred can be transferred to another part of the code.
- We perform exception handling so the normal flow of the application can be maintained even after runtime errors.

C++ Standard Exceptions

- C++ has provided us with a number of standard exceptions that we can use in our exception handling. Some of them are shown in the table below.

Exception	Description
<code>std::exception</code>	The parent class of all C++ exceptions.
<code>std::bad_alloc</code>	Thrown when a dynamic memory allocation fails.
<code>std::bad_cast</code>	Thrown by C++ when an attempt is made to perform a <code>dynamic_cast</code> to an invalid type.
<code>std::bad_exception</code>	Typically thrown when an exception is thrown and it cannot be rethrown.

- In C++, exception is an event or object which is thrown at runtime.
- All exceptions are derived from `std::exception` class. It is a runtime error which can be handled.
- If we don't handle the exception, it prints exception message and terminates the program.
- In C++, we handle exceptions with the help of the try and catch blocks along with the throw keyword.
- **try** - code that may raise an exception.
- **throw** - throws an exception when an error is detected.
- **catch** - code that handles the exception thrown by the throw keyword.

Syntax:

```
try {  
    // Code that might throw an exception  
    throw SomeExceptionType("Error message");  
}  
catch( ExceptionName e1 ) {  
    // catch block catches the exception  
    // that is thrown from try block  
}
```

1. try

The try keyword represents a block of code that may throw an exception placed inside the try block. It is followed by one or more catch blocks. If an exception occurs, try block throws that exception.

2. catch

The catch statement represents a block of code that is executed when a particular exception is thrown from the try block. The code to handle the exception is written inside the catch block.

3. throw

An exception in C++ can be thrown using the throw keyword. When a program encounters a throw statement, then it immediately terminates the current function and starts finding a matching catch block to handle the thrown exception.

Program to divide two numbers which throws an exception when the divisor is 0

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

```
int main()
{
    double numerator, denominator, divide;
    cout << "Enter numerator: ";
    cin >> numerator;
    cout << "Enter denominator: ";
    cin >> denominator;
```



```
try {  
    if (denominator == 0)                // throw an exception if denominator is 0  
        throw 0;  
    divide = numerator / denominator;    // not executed if denominator is 0  
    cout << numerator << " / " << denominator << " = " << divide << endl;  
}  
catch (int num_exception) {  
    cout << "Error: Cannot divide by " << num_exception << endl;  
}  
return 0;  
}
```

OUTPUT 1

Enter numerator: 72

Enter denominator: 0

Error: Cannot divide by 0

OUTPUT 2

Enter numerator: 72

Enter denominator: 3

$72 / 3 = 24$

- To handle the exception, we have put the code

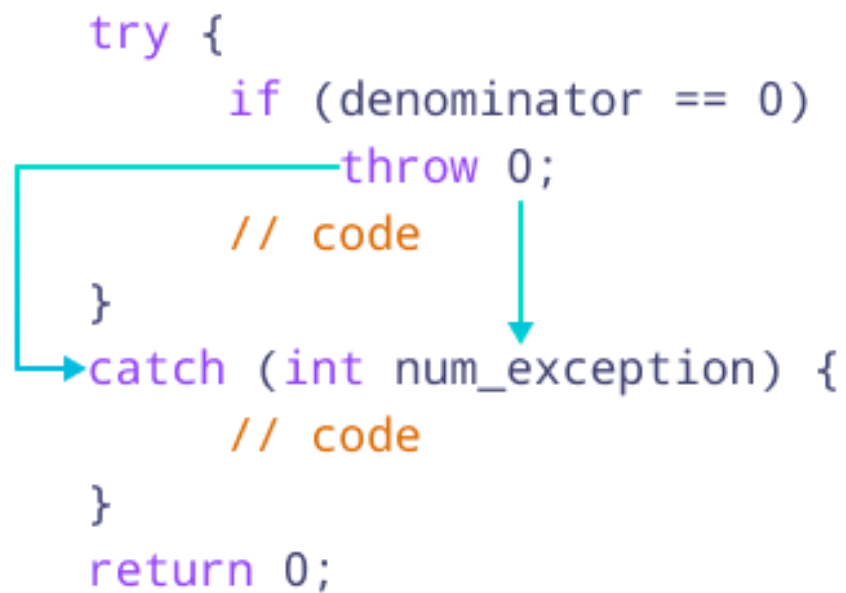
`divide = numerator / denominator;`

inside the try block. Now, when an exception occurs, the rest of the code inside the try block is skipped.

- The catch block catches the thrown exception and executes the statements inside it.
- If none of the statements in the try block generates an exception, the catch block is skipped.

denominator == 0

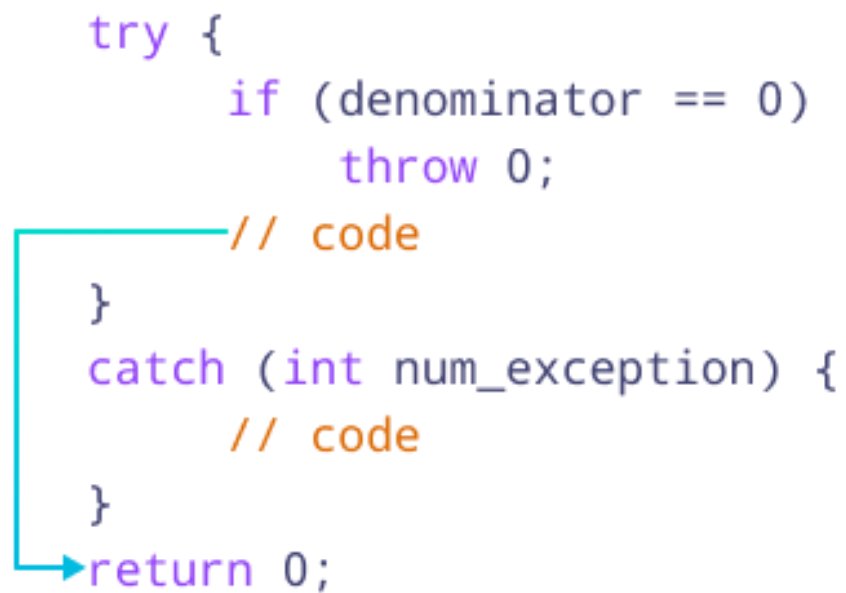
```
try {  
    if (denominator == 0)  
        throw 0;  
    // code  
}  
catch (int num_exception) {  
    // code  
}  
return 0;
```



A flow diagram illustrating the execution of the code for the condition `denominator == 0`. A teal arrow originates from the `throw 0;` statement within the `try` block and points down to the `catch (int num_exception) {` block. Another teal arrow originates from the closing brace of the `try` block and points down to the `catch` block. A final teal arrow originates from the `return 0;` statement and points down to the right, crossing the vertical dashed line.

denominator != 0

```
try {  
    if (denominator == 0)  
        throw 0;  
    // code  
}  
catch (int num_exception) {  
    // code  
}  
return 0;
```



A flow diagram illustrating the execution of the code for the condition `denominator != 0`. A teal arrow originates from the `// code` line within the `try` block and points down to the `catch (int num_exception) {` block. Another teal arrow originates from the closing brace of the `try` block and points down to the `catch` block. A final teal arrow originates from the `return 0;` statement and points down to the right, crossing the vertical dashed line.

- If we do not know the types of exceptions that can occur in our try block, then we can use the ellipsis symbol ... as our catch parameter.

```
try {  
    // code  
}  
catch (...) {  
    // code  
}
```

C++ Multiple catch Statements

In C++, we can use multiple catch statements for different kinds of exceptions that can result from a single block of code.

Syntax

```
try {  
    // code  
}  
catch (exception1) {  
    // code  
}  
catch (exception2) {  
    // code  
}  
catch (...) {  
    // code  
}
```

- Here, our program catches exception1 if that exception occurs. If not, it will catch exception2 if it occurs.
- If there is an error that is neither exception1 nor exception2, then the code inside of catch (...) { } is executed.

Notes:

- `catch (...) { }` should always be the final block in our `try...catch` statement. This is because this block catches all possible exceptions and acts as the default catch block.
- It is not compulsory to include the default catch block in our code.

Example 2: C++ Multiple catch Statements

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

```
int main() {
```

```
    double numerator, denominator, arr[4] = {0.0, 0.0, 0.0, 0.0};
    int index;
```

```
    cout << "Enter array index: ";
    cin >> index;
```

```
    try {
        if (index >= 4)                // throw exception if array out of bounds
            throw "Error: Array out of bounds!";
```

```
cout << "Enter numerator: ";    // not executed if array is out of bounds
cin >> numerator;
```

```
cout << "Enter denominator: ";
cin >> denominator;
```

```
if (denominator == 0)           // throw exception if denominator is 0
    throw 0;
```

```
arr[index] = numerator / denominator; // not executed if denominator is 0
cout << arr[index] << endl;
```

```
}
```

```
catch (const char* msg) {           // catch "Array out of bounds" exception
    cout << msg << endl;
}
```

```
catch (int num) {                   // catch "Divide by 0" exception
    cout << "Error: Cannot divide by " << num << endl;
}
```

```
catch (...) {                       // catch any other exception
    cout << "Unexpected exception!" << endl;
}
```

```
return 0;
}
```

OUTPUT 1

Enter array index: 5

Error: Array out of bounds!

Explanation

Here, the array `arr[]` only has 4 elements. So, index cannot be greater than 3.

In this case, index is 5. So we throw a string literal "Error: Array out of bounds!". This exception is caught by the first catch block.

Notice the catch parameter `const char* msg`. This indicates that the catch statement takes a string literal as an argument.

OUTPUT 2

Enter array index: 2

Enter numerator: 5

Enter denominator: 0

Error: Cannot divide by 0

Explanation

Here, the denominator is 0. So we throw the int literal 0. This exception is caught by the second catch block.

If any other exception occurs, it is caught by the default catch block.

OUTPUT 3

Enter array index: 2

Enter numerator: 5

Enter denominator: 2

2.5

Explanation

Here, the program runs without any problem as no exception occurs.

Exception Handling

Faculty : SREEDIVYA

Branch : S6ECE

Nested try statements

- In C++, a nested try block refers to a try-block nested inside another try or catch block.
- It is used to handle exceptions in cases where different exceptions occur in different parts of the code.
- When try blocks are nested and a throw occurs in a function called by an inner try block, control is transferred outward through the nested try blocks until the first catch block is found whose argument matches the argument of the throw expression.

Syntax of nested try

```
try
{
    // Code..... throw e2
    try
    {
        // code..... throw e1
    }
    catch (Exception e1)
    {
        // handling exception
    }
}
catch (Exception e2)
{
    // handling exception
}
```

Here,
e1: Exception thrown in inner block.
e2: Exception thrown in outer block.

Example of Nested Try Blocks

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

void func (int n)
{
    if (n < 10) {
        throw 22;
    }
    else {
        throw 'c';
    }
}
```

```
int main()
{
    try {
        try {
            cout << "Throwing exception from inner try "
                "block\n";
            func(2);
        }
        catch (int n) {
            cout << "Inner Catch Block caught the exception"
                << endl;
        }
    }
    catch (char c) {
        cout << "Outer catch block caught the exception"
            << endl;
    }

    cout << "Out of the block";

    return 0;
}
```

OUTPUT

Throwing exception from inner try block

Inner Catch Block caught the exception

Out of the block

Explanation

Here, we used `func()` function to throw two exceptions of `int` and `char` type. We used an inner try block to catch integer exceptions. Now, whenever the try blocks throw an exception, the control moves outwards from the nested block till the matching catch block is found. In this case, it was the inner catch block that caught the exception.

What happens if we throw a character exception that the outer catch block is programmed to handle?

```
#include <iostream>
using namespace std;
void func(int n)
{
    if (n < 10) {
        throw 22;
    }
    else {
        throw 'c';
    }
}
```

```
int main()
{
    try {
        try {
            cout << "Throwing exception from inner try "
                "block\n";
            func(12);
        }
        catch (int n) {
            cout << "Inner Catch Block caught the exception"
                << endl;
        }
    }
    catch (char c) {
        cout << "Outer catch block caught the exception"
            << endl;
    }

    cout << "Out of the block";

    return 0;
}
```

OUTPUT

Throwing exception from inner try block

Outer catch block caught the exception

Out of the block

Rethrowing the exception

- If a catch block cannot handle the particular exception it has caught, you can rethrow the exception.
- The rethrow expression (throw without assignment_expression) causes the originally thrown object to be rethrown.
- Rethrowing an exception in C++ involves catching an exception within a try block and instead of dealing with it locally throwing it again to be caught by an outer catch block.
- By doing this, we preserve the type and details of the exception ensuring that it can be handled at the appropriate level within our program.
- Because the exception has already been caught at the scope in which the rethrow expression occurs, it is rethrown out to the next dynamically enclosing try block. Therefore, it cannot be handled by catch blocks at the scope in which the rethrow expression occurred. Any catch blocks for the dynamically enclosing try block have an opportunity to catch the exception.

Example for rethrowing exceptions

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

int divide(int numerator, int denominator)
{
    try {
        if (denominator == 0) {
            throw runtime_error("Division by zero!");
        }
        return numerator / denominator;
    }
    catch (const exception& e) {
        cout << "Caught exception in divide(): " << e.what()
            << endl;
        throw; // Rethrow the caught exception to handle it at a higher level
    }
}
```

```
int calculateSum(int a, int b)
{
    try {
        if (a < 0 || b < 0) {
            throw invalid_argument("Negative numbers not allowed!"); // Throw an invalid_argument exception for negative numbers
        }
        return a + b;
    }
    catch (const exception& e) {
        cout << "Caught exception in calculateSum(): " << e.what() << endl;
        throw; // Rethrow the caught exception to handle it at a higher level
    }
}
```

```
int main()
{
    try {
        int result = calculateSum(10, divide(20, 2));
        cout << "Result: " << result << endl;

        int invalidResult = calculateSum(5, divide(10, 0));
        cout << "Invalid Result: " << invalidResult << endl;
    }
    catch (const exception& e) {
        cout << "Caught exception in main: " << e.what() << endl;
    }

    return 0;
}
```

OUTPUT

Result: 20

Caught exception in divide(): Division by zero!

Caught exception in main: Division by zero!

Explanation:

- The program first calculates the sum of 10 and the result of dividing 20 by 2, which is 20. This result is printed and there are no exceptions raised in this part.
- Next, the program attempts to divide by zero when calculating the sum of 5 and the result of dividing 10 by 0. This triggers a “Division by zero!” exception which is caught within the divide() function and rethrown. The rethrown exception is then caught in the main() function and is printed as “Division by zero!” along with the appropriate exception handling messages.

STREAMS in C++

C++ Streams

- C++ uses the concept of *stream* and *stream classes* to implement its I/O operations with the console and disk files.
- The I/O system in C++ is designed to work with a wide variety of devices including terminals, disks and tape drives.
- The I/O system supplies an interface to the programmer that is independent of the actual device being accessed.
- This interface is known as *stream*.
- A stream is a sequence of bytes.
- It acts either as a source from which the input data can be obtained or as a destination to which the output data can be sent.

- **Input stream** - The source stream that provides data to the program. Data can come from input devices or any storage device.
- **Output stream** - The destination stream that receives output from the program. Data can go to output devices or any storage device.
- C++ contains several predefined streams that are automatically opened when a program starts its execution. For example, **cin** and **cout**.
 - cin – input stream- connected to i/p device (keyboard)
 - cout – output stream - connected to o/p device (screen)

C++ Stream Classes

- C++ I/O system contains a hierarchy of classes that are used to define various streams to deal with both the console and disk files. These classes are called *stream classes*.
- The header file used for i/o operations is *iostream*. This file should be included with all the programs that communicate with the console unit.

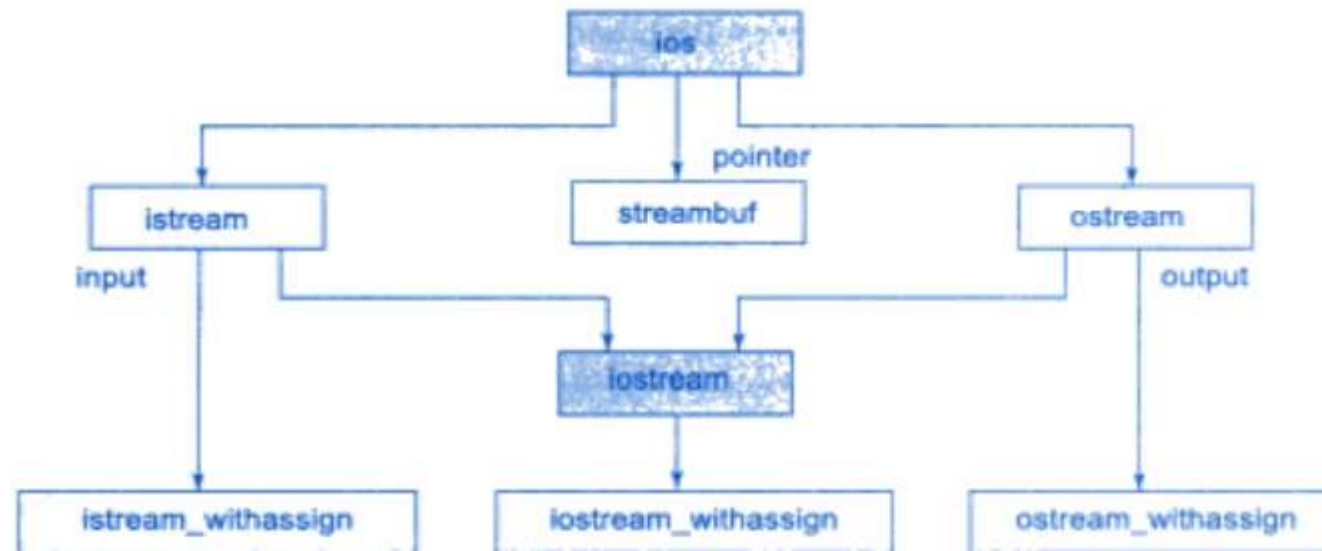


Fig. 10.2 ⇔ Stream classes for console I/O operations

Table 10.1 Stream classes for console operations

<i>Class name</i>	<i>Contents</i>
ios (General input/output stream class)	<ul style="list-style-type: none">• Contains basic facilities that are used by all other input and output classes• Also contains a pointer to a buffer object (streambuf object)• Declares constants and functions that are necessary for handling formatted input and output operations
istream (input stream)	<ul style="list-style-type: none">• Inherits the properties of ios• Declares input functions such as get(), getline() and read()• Contains overloaded extraction operator >>
ostream (output stream)	<ul style="list-style-type: none">• Inherits the properties of ios• Declares output functions put() and write()• Contains overloaded insertion operator <<
iostream (input/output stream)	<ul style="list-style-type: none">• Inherits the properties of ios, istream and ostream through multiple inheritance and thus contains all the input and output functions
streambuf	<ul style="list-style-type: none">• Provides an interface to physical devices through buffers• Acts as a base for filebuf class used in ios files

- **ios** is the base class for **istream** (i/p stream) and **ostream** (o/p stream) which are, in turn, base classes for **iostream** (i/o stream)
- **istream** : provides the facilities for formatted and unformatted input.
- **ostream** : provides the facilities for formatted output.
- **iostream** : provides the facilities for handling both input and output streams.
- **streambuf** : provides an interface to physical devices through buffers.

Unformatted I/O operations

1. Overloaded Operators >> and <<

- The >> operator is overloaded in the **istream** class and << is overloaded in the **ostream** class.
- The following is the general format for reading data from the keyboard :
 cin >> variable1 >> variable2 >> >> variableN.
 where variable1, variable2 are valid C++ variable names.
- This statement will cause the computer to stop execution and look for input data from the keyboard.

- The input statement for this statement would be:

data1 data2 dataN

- >> operator reads the data character by character and assigns to the indicated location.
- The general form for displaying data on the screen is :
cout << item1<<item2 <<.....<<itemN
- The items item1 to itemN may be variables or constants of any basic type.

put() and get() Functions

- To handle the single character input/output operations.
- There are two types of get() functions :
- get(char*) : assigns the input character to its argument.
- get(void) : returns the input character.
- put() : To output a line of text, character by character.

cout.put(ch);

displays the value of variable ch.

getline() and write() Functions

- `getline()` : reads a whole line of text that ends with a newline character. This function can be invoked by using the object `cin` as follows :

`cin.getline (line, size);`

- `write()` : The `write()` function displays an entire line and has the following form :

`cout.write (line, size);`

FORMATTED CONSOLE I/O OPERATIONS

C++ supports a number of features that could be used for formatting the output. These features include :

- ios class functions and flags
- Manipulators
- User-defined output functions

ios format functions

ios class contains a large number of member functions that would help us to format the output in a number of ways. Some of them are:

1. width() : To specify the required field size for displaying an output value.

cout.width(w);

2. precision() : To specify the number of digits to be displayed after the decimal point of a float value.

cout.precision(d);

3. fill() : To specify a character that is used to fill the unused portions of a field.

cout.fill(ch);

4. setf() : To specify format flags that can control the form of output display (such as left-justification and right-justification)

cout.setf(arg1,arg2)

5. unsetf() : To clear the flags specified.

MANAGING OUTPUT WITH MANIPULATORS

- Manipulators are special functions that can be included in the I/O statements to alter the format parameters of a stream.
 - To access these manipulators, the file **iomanip** should be included in the program.
 - They provide the same features as that of the ios member functions and flags.
1. **setw(int w)** : To set the field width to w.
 2. **setprecision(int d)** : To set the floating point precision to d.
 3. **setfill(int c)** : To set the fill character to c.
 4. **setiosflags(long f)** : To set the format flag f.
 5. **resetiosflags(long f)** : To clear the format flag specified by f.
 6. **endl** : Insert new line and flush stream

User- Defined Manipulators

- It is possible to design our own manipulators for certain special purpose.
- Syntax:

```
ostream & manipulator(ostream & output)
{
    .....
    .....
    return output;
}
```

- Example

```
ostream & unit (ostream & output)
{
    output << "inches" ;
    return output;
}
```

The statement,

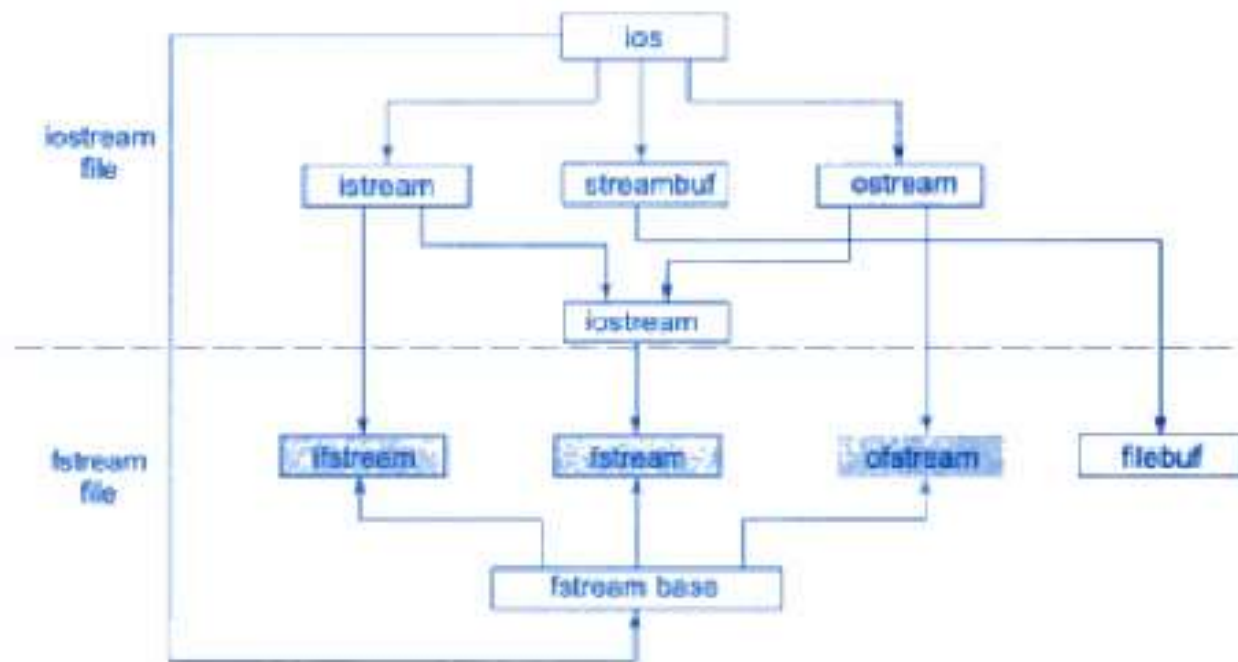
```
cout<<36<<unit;
```

will produce the output as 36 inches

Files in C++

- A file is a collection of related data stored in a particular area on the disk.
- Files in C++ are interpreted as a sequence of bytes stored on some storage device.
- The data of a file is stored in either readable form or in binary code called as text file or binary file.
- The flow of data from any source to a destination is called as a stream
- Computer programs are associated to work with files as it helps in storing data & information permanently.

- File streams acts as an interface between the programs and the files.
- The I/O system of C++ contains a set of classes that define the file handling methods.
- Classes for file stream operations are ifstream, ofstream and fstream
- These classes are derived from **fstreambase** and from the corresponding *istream* class.
- These classes designed to manage the disk files, are declared in *fstream* and therefore we must include this file in any program that uses files.



Stream classes for file operations

Details of file stream classes

<i>Class</i>	<i>Contents</i>
filebuf	Its purpose is to set the file buffers to read and write. Contains Openprot constant used in the open() of file stream classes. Also contain close() and open() as members.
fstreambase	Provides operations common to the file streams. Serves as a base for fstream , ifstream and ofstream class. Contains open() and close() functions.
ifstream	Provides input operations. Contains open() with default input mode. Inherits the functions get() , getline() , read() , seekg() and tellg() functions from istream .
ofstream	Provides output operations. Contains open() with default output mode. Inherits put() , seekp() , tellp() , and write() , functions from ostream .
fstream	Provides support for simultaneous input and output operations. Contains open() with default input mode. Inherits all the functions from istream and ostream classes through iostream .

General file I/O steps

- Declare a file name variable
- Associate the file name variable with the disk file name
- Open the file
- Use the file
- Close the file

File opening and closing

- The filename is a string of characters that make up a valid filename for the operating system. It may contain two parts, a primary name and an optional period with extension.

example: Input.data, Test.doc

- For opening a file, first create a file stream and then link it to the filename. A file stream can be defined using the classes ifstream, ofstream and fstream.
- A file can be opened in two ways:
 1. Using the construction function of the class. (if only one file is used)
 2. Using the member function open() of the class.(for multiple files)

Examples:

```
ofstream outfile("results");
```

```
outfile.open("results");
```

```
outfile.open("results", w)
```

opens a file named "results" for writing output.

```
ifstream infile("data");
```

```
infile.open("data");
```

```
infile.open("data", r);
```

opens a file named "data" for reading the input.

- For closing an opened file, we use `close()` function.
e.g., `outfile.close();`
- Detecting an end-of-file condition is necessary for preventing any further attempt to read data from the file.
- `eof()` returns a non-zero value if the end- of- file condition is encountered and a zero otherwise. `eof()` is a member function of `ios` class

File pointers and their manipulations

- Each file has two associated pointers known as the *file pointers*.
- One of them is called the *input pointer* or *get pointer*.
- The input pointer is used for reading the contents of a given file location.
- Other is called the *output pointer* or *put pointer*.
- The output pointer is used for writing to a given file location.
- We can use these pointers to move through the files while reading or writing.
- Each time an input or output operation takes place, the appropriate pointer is automatically advanced.

Default Actions

- When we open a file in **read-only mode**, the input pointer is automatically set at the beginning so that we can read the file from the start.
- Similarly, when we open a file in **write-only mode**, the existing contents are deleted and the output pointer is set at the beginning.
- This enables us to write to the file from the start.
- In case, we want to open an existing file to add more data, the file is opened in **‘append’** mode. This moves the output pointer to the end of the file.

Functions for manipulation of file pointers

<code>seekg()</code>	- Moves get pointer (input) to a specified location.
<code>seekp()</code>	- Moves put pointer (output) to a specified location.
<code>tellg()</code>	- Gives the current position of the get pointer.
<code>tellp()</code>	- Gives the current position of the put pointer.

For example,

```
infile.seekg(10);
```

Moves the file pointer to the byte number 10.

The bytes in a file are numbered beginning from zero. Thus, the pointer will be pointing to the 11th byte in the file.

Specifying the offset :

- seekg() and seekp() can also be used with two arguments as follows:

```
seekg(offset, reposition);  
seekp(offset, reposition);
```

- The parameter offset represents the number of bytes the file pointer to be moved from the location specified by the parameter reposition.
- The reposition takes one of the following these constant defined in the ios class.

ios::beg	start of the file
ios::cur	current position of the pointer
ios::end	end of the file.

Templates

- Templates is one of the C++ features which enable us to define generic classes and functions and thus provides support for generic programming.
- Generic programming is an approach where generic types are used as parameters in algorithms so that they work for a variety of suitable data types and data structures.
- A template can be used to create a family of classes or functions.
- For example, a class template for an array class would enable us to create arrays of various data types such as int array and float array.
- Similarly, we can define a template for a function, say mul(), that would help us create various versions of mul() for multiplying int, float and double type values.
- Since a template is defined with a parameter that would be replaced by a specified data type at the time of actual use of the class function, the templates are sometimes called parameterized classes or functions.

Working of Templates

```
template <typename T>
T myMax(T x, T y)
{
    return (x > y)? x: y;
}
```

```
int main()
{
    cout << myMax<int>(3, 7) << endl;
    cout << myMax<char>('g', 'e') << endl;
    return 0;
}
```

Compiler internally generates
and adds below code

```
int myMax(int x, int y)
{
    return (x > y)? x: y;
}
```

Compiler internally generates
and adds below code.

```
char myMax(char x, char y)
{
    return (x > y)? x: y;
}
```

Templates can be represented in two ways:

1. Function template : used to create a family of functions with different argument types.
2. Class template : a class defines something that is independent of the data type.

Function Templates

General format

```
template<class T>  
returntype functionname (arguments of type T)  
{  
    //....  
    // Body of function  
    //with type T  
    // wherever appropriate  
    //.....  
}
```

Example

```
#include<iostream>
using namespace std;
template<class T>
void swap(T &x, T &y)
{
    T temp = x;
    x = y;
    y = temp;
}
void func(int m, int n, float a, float b)
{
    cout<<"m and n before swap:" <<m << " " <<n<<"\n";
    swap(m,n);
    cout<<"m and n after swap:" <<m << " " <<n<<"\n";
    cout<<"a and b before swap:" <<a << " " <<b<<"\n";
    swap(a,b);
    cout<<"a and b after swap:" <<a << " " <<b<<"\n";
}
```

```
int main()
{
    func(100,200,11.22,33.44);
    return 0;
}
```

Output

m and n before swap : 100 200

m and n after swap : 200 100

a and b before swap : 11.22 33.44

a and b after swap : 33.44 11.22

Class Templates

General format

```
template<class T>
class classname
{
    //....
    // class member specification
    //with type T
    // wherever appropriate
    //.....
};
```



```
#include<iostream>
using namespace std;
template<class T>
class Number
{
    T num;
public:
    Number(T n): num(n) { }
    T getNum(){
        return num;
    }
};
```

```
int main()
{
    Number<int> numberInt(7);
    Number<double> numberDouble(7.7);
    cout<<" int Number = " << numberInt.getNum() <<endl;
    cout<<" double Number = " << numberDouble.getNum() <<endl;
    return 0;
}
```

Output

int Number = 7

double Number = 7.7