Exception Handling:

Exception refers to unexpected condition in a program. The unusual conditions could be faults, causing an error which in turn causes the program to fail. The error handling mechanism of c++ is generally referred to as exception handling.

Generally , exceptions are classified into synchronous and asynchronous exceptions.. The exceptions which occur during the program execution, due to some fault in the input data or technique that is not suitable to handle the current class of data. with in a program is known as synchronous exception.

Example:

errors such as out of range,overflow,underflow and so on.

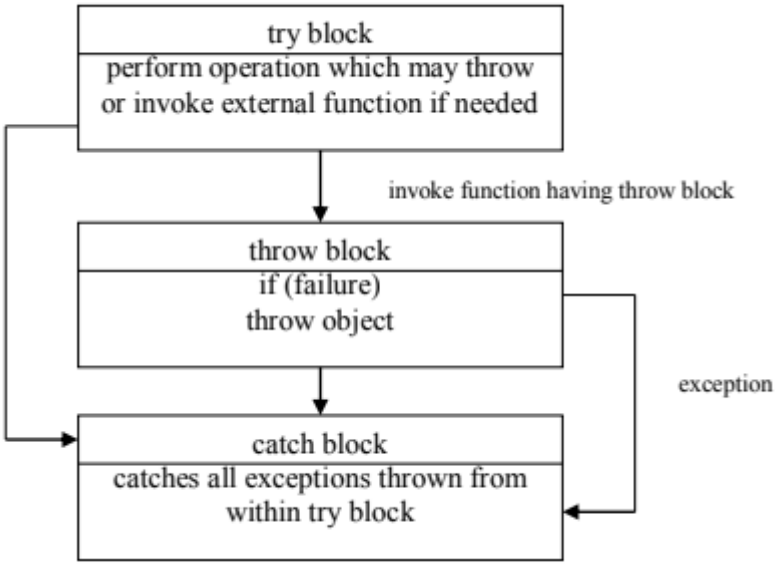
The exceptions caused by events or faults unrelated to the program and beyond the control of program are asynchronous exceptions.

For example, errors such as keyboard interrupts, hardware malfunctions, disk failure and so on.

Exception handling model:

When a program encounters an abnormal situation for which it in not designed, the user may transfer control to some other part of the program that is designed to deal with the problem. This is done by throwing an exception. The exception handling mechanism uses three blocks: try, throw and catch.

The try block must be followed immediately by a handler, which is a catch block. If an exception is thrown in the try block the program control is transferred to the appropriate exception handler. The program should attempt to catch any exception that is thrown by any function. The relationship of these three exceptions handling constructs called the exception handling model is shown in figure:



throw construct:

The keyword throw is used to raise an exception when an error is generated in the comutation. the throw expression initialize a temporary object of the typeT used in thorw (T arg).

syntax:

throw T;

catch construct:

The exception handler is indicated by the catch keyword. It must be used immediately after the

statements marked by the try keyword. The catch handler can also occur immediately after another

catch Each handler will only evaluate an exception that matches.

syntax:

catch(T)

{

// error meassges

}

try construct:

The try keyboard defines a boundary within which an exception can occur. A block of code in which an exception can occur must be prefixed by the keyword try. Following the try keyword is a block of code enclosed by braces. This indicates that the prepared to test for the existence of exceptions. If an exception occurs, the program flow is interrupted.

```
try
{
...
if (failure)
throw T;
}
catch(T)
{
...
}
```

Example:

```
#include<iostream.h>
void main()
{
int a,b;
cout<<”enter two numbers:”;
cin>>a>>b;
try
{
if (b= =0)
throw b;
else
cout<<a/b;
}
catch(int x)
{
cout<<”2nd operand can’t be 0”;
}
}
```

Array reference out of bound:

```
#define max 5
class array
{
private:
int a[max];
public:
int &operator[](int i)
{
if (i<0 || i>=max)
throw i;
else
return a[i];
}
};
void main()
{
array x;
try
{
cout<<"trying to refer a[1]..."
x[1]=3;
cout<<"trying to refer a[13]..."
x[13]=5;
}
catch(int i)
{
cout<<"out of range in array references...";
}}
```

Multiple catches in a program

```
void test(int x)
{
try{
if (x==1)
throw x;
else if (x==-1)
throw 3.4;
else if (x==0)
throw ‘s’;
}
catch (int i)
{
cout<<”caught an integer...”;
}
catch (float s)
{
cout<<”caught a float...”;

}
catch (char c)
{
cout<<”caught a character...”;
}}
void main()
{
test(1);
test(-1);
test(0);
}
```

catch all

```
void test(int x)
{
try{
if (x==1)
throw x;
else if (x==-1)
throw 3.4;
else if (x==0)
throw ‘s’;
}
catch (...)
{
cout<<”caught an error...”;
}
```

Other Directives
Apart from the above directives, there are two more directives that are not commonly used. These are:
#undef Directive
#pragma Directive

1. #undef Directive

The #undef directive is used to undefine an existing macro. This directive works as:

#undef LIMIT
Using this statement will undefine the existing macro LIMIT. After this statement, every "#ifdef LIMIT" statement will evaluate as false.

2. #pragma Directive
This directive is a special purpose directive and is used to turn on or off some features. These types of directives are compiler-specific, i.e., they vary from compiler to compiler. Some of the #pragma directives are discussed below:

#pragma startup: These directives help us to specify the functions that are needed to run before program startup (before the control passes to main()).
#pragma exit: These directives help us to specify the functions that are needed to run just before the program exit (just before the control returns from main()).

// C program to illustrate the #pragma exit and pragma startup

```
#include <bits/stdc++.h>
using namespace std;

void func1();
void func2();

// specifying funct1 to execute at start
#pragma startup func1
// specifying funct2 to execute before end
#pragma exit func2

void func1() { cout << "Inside func1()\n"; }
void func2() { cout << "Inside func2()\n"; }
int main()
{
        void func1();
        void func2();
        cout << "Inside main()\n";

        return 0;
}
```

```
// Preprocessor directive
#include<iostream>
#define gfg 7
#if gfg > 200 //(7>200)
    #undef gfg
    #define gfg 200
#elif gfg < 50 //(7<50)
    #undef gfg
    #define gfg 50 //execute this part
#else
    #undef gfg
    #define gfg 100
#endif
int main()
{
      std::cout << gfg;    // gfg = 50
}
```

//pragma

```
#include<stdio.h>
  int display();
  #pragma startup display
  #pragma exit display
  int main() {
        printf("\nI am in main function");
        return 0;
  }
  int display() {
        printf("\nI am in display function");
        return 0;
  }
```

```
#include<stdio.h>
  #pragma warn -rvl /* return value */
  #pragma warn -par /* parameter never used */
  #pragma warn -rch /*unreachable code */
  int show(int x)
  {
                printf("Tutorials And Examples");
                // function does not have a return statement
  }
  int main()
  {
                show(10);
```

```
        return 0;
}
```

Trigraph Sequences

To write C programs using character sets that do not contain all of C's punctuation characters, ANSI C allows the use of nine trigraph sequences in the source file. These three- character sequences are replaced by a single character in the first phase of compilation. Below table lists the valid trigraph sequences and their character equivalents.

Table :Trigraph Sequences

Trigraph Sequence	Character Equivalent
??=	#
??(	[
?? /	\
??)	]
??'	^
??<	{
??!	|
??>	}
??-	~

No other trigraph sequences are recognized. A question mark (?) that does not begin a trigraph sequence remains unchanged during compilation. For example, consider the following source line:

printf ("Any questions???/n");

After the ??/ sequence is replaced, this line is translated as follows:

printf ("Any questions?\n");

Digraph Sequences

Digraph processing is supported when compiling in ISO C 94 mode (/STANDARD=ISOC94 on OpenVMS systems).

Digraphs are pairs of characters that translate into a single character, much like trigraphs, except that trigraphs get replaced inside string literals, but digraphs do not. Table 2 lists the valid digraph sequences and their character equivalents.

Table 2: Digraph Sequences

Digraph Sequence	Character Represented
<:	[
:>	]
<%	{
%>	}
%:	#
%:%:	##

Example

The symbols `[ ] { } ^ \ | ~ #` are frequently used in C programs, but in the late 1980s, there were code sets in use (ISO 646 variants, for example, in Scandinavian countries) where the ASCII character positions for these were used for national language variant characters (e.g. `£` for `#` in the UK; `Æ Å æ å ø Ø` for `{ } { } | \` in Denmark; there was no `~` in EBCDIC). This meant that it was hard to write C code on machines that used these sets.

To solve this problem, the C standard suggested the use of combinations of three characters to produce a single character called a trigraph. A trigraph is a sequence of three characters, the first two of which are question marks.

The following is a simple example that uses trigraph sequences instead of `#`, `{` and `}`:

```
??=include <stdio.h>

int main()
??<
    printf("Hello World!\n");
??>
```

This will be changed by the C preprocessor by replacing the trigraphs with their single-character equivalents as if the code had been written:

```
#include <stdio.h>

int main()
{
    printf("Hello World!\n");
```

```
}
```