**Const**

**Why?**

Constant is something that doesn't change.

**When?**

In C language and C++ we use the keyword const to make program elements constant. const keyword can be used in many contexts in a C++ program. It can be used with:

1. Variables
2. Pointers
3. Function arguments and return types
4. Class Data members
5. Class Member functions
6. Objects

**How?**

Const variable

int main

{

const int i = 10;

const int j = i + 10; // works fine

i++; // this leads to Compile time error

}

Pointers

Pointers can be declared using const keyword. Wecan do it two ways like applying const to a what pointer is pointing to or we can assign const to pointer.

Ex:

Pointer pointing to const variable:

const int \* a;

Pointer pointing to a const variable:

int const\* b;

const pointer

To make a pointer const we need to add const keyword to the right of \*

int x = 1;

int \*const y = &x;

Here we cannot change the pointer, it means it always points to x but the value it points to can be modified.

So if we need a const pointer to a const variable,

const int \*const y = &x;

const function arguments

void f(const int i)

{

i++; // error

}

return type

const int g()

{

return 1;

}

--------------------------------------------------------------------------------

**Mutable**

mutable keyword is used with member variables of class, which we want to change even if the object is of const type. Hence, mutable data members of a const objects can be modified.

class Zee

{

int i;

mutable int j;

public:

Zee()

{

i = 0;

j = 0;

}

void fool() const

{

i++; // will give error

j++; // works, because j is mutable

}

};

int main()

{

const Zee obj;

obj.fool();

}

--------------------------------------------------------------------------------

**References**

Its like giving a alternate name to a variable.

Reference must be initialized when it is created.

Once initialized, we cannot reinitialize a reference.

Null reference is never possible.

Reference is automatically dereferenced.

References are like constant pointers that are automatically dereferenced

**Pgm1**

int main()

{

int y=10;

int &r = y; // r is a reference to int y

cout << r;

}

**Pgm2**

int\* first (int\* x)

{

(\*x++);

return x; // SAFE, x is outside this scope

}

int& second (int& x)

{

x++;

return x; // SAFE, x is outside this scope

}

int& third ()

{

int q;

return q; // ERROR, scope of q ends here

}

int& fourth ()

{

static int x;

return x; // SAFE, x is static, hence lives till the end.

}

int main()

{

int a=0;

first(&a); // UGLY and explicit

second(a); // CLEAN and hidden

}

------------------------------------------------------------------------------------------------------------------------

**Exceptions**

Errors can be broadly categorized into two types. We will discuss them one by one.

1. Compile Time Errors
2. Run Time Errors

**Compile Time Errors** – Errors caught during compiled time is called Compile time errors. Compile time errors include library reference, syntax error or incorrect class import.

**Run Time Errors** - They are also known as exceptions. An exception caught during run time creates serious issues.

Errors hinder normal execution of program. Exception handling is the process of handling errors and exceptions in such a way that they do not hinder normal execution of the system. For example, User divides a number by zero, this will compile successfully but an exception or run time error will occur due to which our applications will be crashed. In order to avoid this we'll introduce exception handling technics in our code.

In C++, Error handling is done using three keywords:

* try
* catch
* throw

**Syntax:**

try

{

//code

throw parameter;

}

catch(exceptionname ex)

{

//code to handle exception

}

**Try**

he code which can throw any exception is kept inside(or enclosed in) atry block. Then, when the code will lead to any error, that error/exception will get caught inside the catch block.

**Catch**

catch block is intended to catch the error and handle the exception condition. We can have multiple catch blocks to handle different types of exception and perform different actions when the exceptions occur. For example, we can display descriptive messages to explain why any particular excpetion occured.

**Throw**

It is used to throw exceptions to exception handler i.e. it is used to communicate information about error. A throw expression accepts one parameter and that parameter is passed to handler.

throw statement is used when we explicitly want an exception to occur, then we can use throw statement to throw or generate that exception.

**Pgm**

#include <iostream>

using namespace std;

class exception1

{

int errorno;

public:

void display()

{

cout << "Error no. " << errorno << endl;

}

exception1(int no):errorno(no)

{}

};

int main()

{

int x = -1;

// Some code

cout << "Before try \n";

try {

cout << "Inside try \n";

if (x < 0)

{

exception1 ob(5);

throw ob;

cout << "After throw (Never executed) \n";

}

}

catch (exception1& ob1 ) {

ob1.display();

cout << "Exception Caught \n";

}

cout << "After catch (Will be executed) \n";

return 0;

}