① Initializer List in C++ :- (Geeks for Geeks) ①

- Initializer list is used to initialize data members of a class.
- The list of members to be initialized is indicated with a constructor as a comma seperated list followed by a colon.

Ex:-
```
class Point {
private:
    int x;
    int y;
public:
    Point(int i=0, int j=0): x(i), y(j) { }
};
```

① For initialisation of non-static const data members:

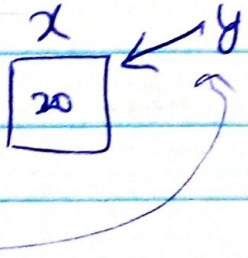
- Must be initialised using a initializer list

Ex:
```
class Test {
    const int t;
public:
    Test(int t) : t(t) { } };
```

② For initialization of reference members:

- must be initialised using initializer list

```
class Test {
    int &t;
public:
    Test(int &t) : t(t) { }
    int getT() {
        return t; }
};
int main() {
    int x = 20;
    Test t1(x);
    cout << t1.getT() << endl;   // 20
    x = 30
    cout << t1.getT() << endl;   // 30
    return 0;
}
```



- int x = 20
- int &y = x

- As 'y' is pointing to x, if 'x' changes 'y' also changes.

(2)

③ For initialization of member objects which do not have default constructor.

- In the following example, an object "a" of class "A" is data member of class "B", and "A" doesn't have default constructor.
- Initializer must be used to initialize "a".

Ex:

```
class A {
        int i;
public:
        A(int);
};
A::A(int arg) {
        i=arg;
        cout<<"A's const called; value of i:"<<i<<"\n";
}

class B {
        A a
public:
        B(int);
};
B::B(int x):a(x){
        cout<<" B's constructor called";
}

int main() {
        B obj(10);
        return 0;
}
```

output:

A's const called,
value of i = 10
B' const called

④ For initialisation of base class members.

Like point 3, parameterised constructor of base class can only be ~~initialised~~ called using initialiser list.

Ex:-

```
class A {
        int i;
public:
        A(int i);
};
A::A(int arg) {
        i = arg; cout << "A's const. value, i"; << i << "\n";
}
class B {
public:
        B(int);
};
B::B(int x): A(x) {
        cout << "B's const. called";
}
int main() {
        B obj(10);
        return 0;
}
```

5) When constructor's parameter name is same as data member:

. If constructor's parameter name is same as data member name then we can initialise either using "this" pointer or initialiser list.

Ex:-
```
class A {
    int i;
public:
    A(int);
    int getI() const {return i;}
};
A::A(int i) : i(i) { }
/*
    A::A(int i) {
        this.i = i;
    }
*/
```

(6) For Performance reasons:-

. It is better to initialise all class variables in Initializer list instead of assigning values inside body.

because, of this example,

YOUTUBE: InfuYMXjZsA

```
class Example {
public:
    Example () {
std:: cout << "created Entity !" << "\n";
    }
    Example (int x) {
    std:: cout << "created Entity with" << x << "! \n";
    }
};
class Entity {
private :
    std:: string m_name; Example ex;
public:
    Entity ()
    {
    m_Name = "abc";
    ex = Example(8);
    }
```

→ If you do this, you'll get

created Entity !
created Entity with 8!

In this, initialisation, default constructor is called & then constructor with parameter

```
Entity (): m_name("abc"), ex(8)
{
}
```

→ If you do this, you'll get

created Entity with 8!

So, only the parameterised constructor is called.

CONST (const) KEYWORD:

- It'll be used before the variables, which don't change.

Ex:-

```
const int a = 10;
a = 11; ✗ error
```

- In Pointers:-

const int * a ≡ int const * a

int b = 10; int c = 11;

* if we do `const int * a = &b;`

*a = c ✗ error

If we do this, we'll not be able to change the content or value of the pointer, but

a = &c ✓ correct

*

If we do, `int b = 10; int c = 11;`

`int * const a = &b;`

*a = c ✓

a = &c ✗* error

So, if,

* const p; ⇒ pointer is constant ⇒ p

const *p; value is constant ⇒ *p

→ In functions:-

• If we make class function as const, then that function cannot modify the object, on which they are called.

Ex:-

```
class Test {
        int value;
    public:
        Test(int v=0) { value = v; }
        // we get compiler error if we add a line like "value = 100;" in this function
        int getValue() const {
            return value; }
};
int main() {
    Test t(20);
    t.getValue();
}
```

MUTABLE KEYWORD:-

• If we use "mutable" keyword for class variable, then even in "const" function it'll be able to modify the variable.

```
class Test
{
    mutable int value;
    public:
    int getTest() const { value = 10;
        return value;
```

'explicit' Keyword:-

- "Implicit" conversion is done in 'C++', when we do, [complex c1 = 5] (or) (Metre m = 10.0) (or) m to cms.

```
class Text {
    string a;
    Text (string b)
    {
        a = b
    }
};
int main { Text c("abc")
```

string ← char

- To stop above 'implicit' conversions we use the keyword "explicit" before the constructor.

Resource Acquisition Is Initialization (RAII):-

- Resource acquisition is initialization (or) RAII, is a C++ programming technique which binds the life cycle of a resource that must be acquired before use (allocated heap memory, thread of execution, open socket

open file, locked mutex, disk space, database connection - anything that exists in limited supply) to the lifetime of an object.

- It uses C++ features like "object lifetime", "scope exit", "order of initialization" and "stack unwinding" to achieve this.

RAII can be summarized as below:-

* Encapsulate each resource into a class, where,
  - the constructor acquires the resource and establishes all class invariants or throws an exception if that cannot be done.
  - the destructor releases the resources and never throws exception.
* Always use the resource via an instance of a RAII-class that either.
  - has automatic storage duration (or) temporary lifetime itself; or
  - has lifetime that is bounded by the lifetime of an automatic (or) temporary object.

* RAII uses the concept of "Stack based variables" and "Heap based variables".

For example,

X

```
int* arrayReturn()
{
    int array[50];
    return arr;
}
```

Though, it is returning ptr it'll not work, because above array declaration creates memory on stack as, soon as it exists function memory gets deallocated.

```
int * arrayReturn()
{
    int arr[50] = new int[50];
    return &arr;
}
```

It works because, the memory gets created on heap space, & can be accessed by the other variables, even after the function returns.

Similar, concept is used in RAII to allocate & dellocate memory and other management using "class" & "Object".

Example:

```
class Entity
{
public:
    Entity()
    {
        cout << "created Entity!";
    }
    ~Entity()
    {
        cout << "Destroyed Entity!";
    }
};
int main()
{
    {Entity *e = new Entity();}
}
```

→ If we do this, the memory allocated on heap, will not freed even after the scope.

So, we use a RAII class to create and destroy objects.

RAII class:

```
class ScopedPtr {
private:
    Entity * m_ptr;
public:
    ScopedPtr(Entity *ptr) : m_Ptr(ptr){}
    ~ScopedPtr(){
        delete m_Ptr;
    }
};
```

We do create the resource in the below manner,

```
int main() {

    {
      ScopedPtr e = new Entity();
    }

}
```

So, with this, memory will be automatically deallocated in the destructor.

- Similar concept can be used for any other resources like mutex, threads, socket, file closing etc.

Static Linking:

- In static linking binaries will be actually be part of executable code.
- like including ".h" files and binaries
- In dynamic linking, binaries will be loaded into memory during runtime. (.dll files) and will be linked & used.

(1) → Google test

→ Google C++ Testing, GTest, GMock

Framework :-

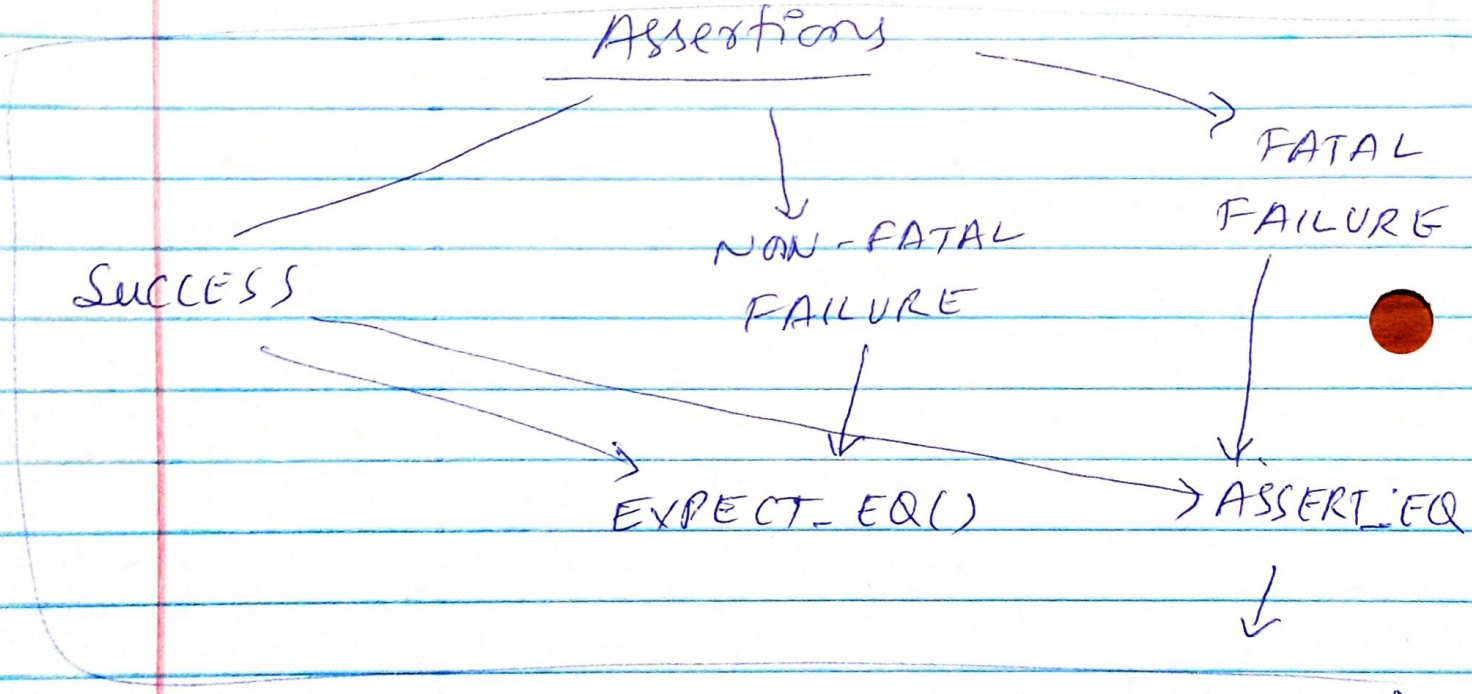
```
#include <gtest/gtest.h>

TEST(TestName, Subtest_1) {
    ASSERT_FALSE(1==2);
}

TEST(TestName2, Subtest_1) {
    ASSERT_TRUE(1==2);
}

int main(int argc, char **argv) {
    testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

Test code 1

Test Case 2

- There can be fatal and non-fatal failures in Test cases.

- If "EXPECT_EQ()" → fails than it is a non-fatal failure
- but if "ASSERT_EQ" → fails than it is a FATAL FAILURE



- After "ASSERT" → no statement will get executed

- After "EXPECT_EQ" → ~~statement~~ it does not stop execution even after failure of Expect

(2) → Google Test.

Deadlock:

Equal → EXPECT_EQ() - ASSERT_EQ()

not equal → EXPECT_NQ() - ASSERT_NQ()

Less Than Equal → EXPECT_LE() - ASSERT_LQ()

⋮ ⋮ ⋮

↓ ↓

Non-Fatal Fatal.

* We should not write multiple "ASSERTS" or "EXPECT", because, ⊘

① A single failure of "EXPECT" makes the entire test case as failed

② A single "ASSERT" fails, the following test cases will not be executed.

---

i) ~~A Test case~~ Any "Unit Test" must consist of three things.

TEST()

↓

Arrange → Act → Assert

Arrange ↓ Act ↓ Assert ↓

Arrange everything required to run Test

Run the Test

Verify the output.

Example:-

```
TEST ( TestName, Subtest_1) {
    // Arrange
    int value = 100;
    int increment = 5;

    // Act
    value = value + increment;

    // ASSERT
    ASSERT_EQ(value, 105);
}
```

UNIT_TEST Properties :

1) Should run extremely fast (within milliseconds)
2) Must be able to run independently.
3) Doesn't depend upon any external input

→ For STRINGS:

Equal → EXPECT_STREQ() - ASSERT_STREQ()

Case Equal → EXPECT_STRCASEEQ() → ASSERT_STRCASEEQ()

TEST Fixtures :-

- As part of writing many Test cases we might repeat the same code in different test cases.
- To avoid that, we have a structure called Test-fixtures, in which we have SetUp(); & TearDown(); functions, where we can initialise & clean up the common code & objects.

Example:

```
class MyClass {
    int baseValue;
public:
    MyClass(int _bv) : baseValue(_bv) { }
    void Increment(int byValue) {
        baseValue += byValue;
    }
    int getValue() { return baseValue; }
};
```

Test fixture

```
struct MyClassTest : public testing::Test {
    MyClass *mc;
    void setUp() { cout << "Alive";
                   mc = new MyClass(100);
    }
    void TearDown() { cout << "Dead";
                      delete mc; }
};
```

inheritance

we use "underscore F" to tell we are using fixture code

```
TEST_F (MyClassTest, Increment_by_5){
    mc->Increment(5);
    ASSERT_EQ(mc->getValue(), 105);
}
```

we pass class name

Google Mock - The C++ Mocking Framework

- Mocks are used for testing the behaviour of API(s) interfaces which will be used in component under test.

→ Mock this dependency to test "My component"

→

```
#include <gmock/gmock.h>
using ::testing::AtLeast;
using ::testing::Return;
using ::testing::_;
```

. In Gmock,

. we use, EXPECT_CALL, to say that for sure function call will be do done else test will fail

. we use, ON_CALL, to say that function call may (or) may not be done

. Both are used to create mocks in the TestCase function code.

→ SAMPLE CODE:-

```
#include<iostream>
#include<vector>
#include<gtest/gtest.h>
#include<gmock/gmock.h>
using namespace std;
using ::testing::AtLeast;
using ::testing::Return;
using ::testing::_;
```

```
class DatabaseConnect {
public:
    virtual bool login(string username,
                       string password) {
        return true; }
    virtual bool logout(string username) {
        return true; }
    virtual int fetchRecord() {
        return -1; }
};
class MockDB : public DataBaseConnect {
public:
    MOCK_METHOD0(fetchRecord, int());
    MOCK_METHOD1(logout, bool(string
                              username));
    MOCK_METHOD2(login, bool(string
                             username, string password));
};
class MyDatabase {
    DataBaseConnect &dbc;
public:
    MyDatabase(DatabaseConnect &_dbc):
        dbc(_dbc) { }
```

# of argument for this fun.

⑤ Google unit test

~~class MyDatabase {~~

```
int Init(string username, string passwd)
{
    if(dbc.login(username, password)!=true){
        cout << "DB FAILURE";
        return -1;
    } else {
        cout << "DB SUCCESS";
        return 1;
    }
};
TEST(MyDBTest, LoginTest) {
    //Arrange
    MockDB mdb;
    MyDatabase db(mdb);
    EXPECT_CALL(mdb, login("Terminator",
                "I'm back"));
    .Times(1)
    .WillOnce(Return(True));
    int retvalue = db.Init("Terminator",
                "I'm back");
    EXPECT_EQ(retValue, 1);
}
```

Actual Mocks