Classes & Objects

1. Default controls
2. Constructor delegating
3. In class initializers
4. Uniform initializers
5. Initializer list
6. Explicit conversion operators
7. Read only objects
8. Explicit Type Conversions
9. Type Traits

Inheritance
final, override keywords
explicit inheritance of base class members

1.Default Controls

a. = default
b. = delete

a. =default:

-> c++11 standard allows us to append '=default;' specifier at end of a function declaration to declare that function as an explicitly defaulted function.

-> This makes compiler generate default implementations for explicitly defaulted functions which are more efficient than manually programmed function implementations.

-> For example, whenever we declare a parameterized constructor, the compiler won't create a default constructor. In such a case, we can use the default specifier in order to create a default one.

Example:

```
// C++ code to demonstrate the
// use of defaulted functions
#include <iostream>
using namespace std;

class A {
public:
```

```
        // A user-defined
        // parameterized constructor
        A(int x)
        {
                cout << "This is a parameterized constructor";
        }

        // Using the default specifier to instruct
        // the compiler to create the default
        // implementation of the constructor.
        A() = default;
};

int main()
{
        // executes using defaulted constructor
        A a;

        // uses parametrized constructor
        A x(1);
        return 0;
}
```

Output: This is a parametrized constructor.

-> From the above example we can observe that in A() we have not declared the body of constructor, instead we have assigned the constructor to default(=default) so that compiler create default implementation of this function.

Constraints with making function defaulted:

-> Non special member functions cant be defaulted.
-> Defaulted function needs to be special member function (default constructor, copy constructor, destructor etc) or has no default arguments

Example:

```
// C++ code to demonstrate that
// non-special member functions
// can't be defaulted
class B {
public:

        // Error, func is not a special member function.
        int func() = default;
```

```
        // Error, constructor B(int, int) is not
        // a special member function.
        B(int, int) = default;

        // Error, constructor B(int=0)
        // has a default argument.
        B(int = 0) = default;
};

// driver program
int main()
{
        return 0;
}
```

a. i Advantages of using '=default' when we can leave body of function using '{}'?

-> User defined constructor makes type not an aggregate and also not trivial. To make class aggregate or trivial type use '=default'.

-> Using '= default' can also be used with copy constructor and destructors. An empty copy constructor, for example, will not do the same as a defaulted copy constructor (which will perform member-wise copy of its members). Using the '= default' syntax uniformly for each of these special member functions makes code easier to read.

b. Deleted function

-> In C++, delete had only one purpose which was to deallocate a memory that has been allocated dynamically.
-> In C++ 11, delete function can be used to disable the usage of member function. This is done by appending the =delete; specifier to the end of the function declaration.

->Any member function whose usage has been disabled by using the '=delete' specifier is known as an expicitly deleted function

-> Some examples are: Disabling copy constructors and then disabling undesirable argument conversion.

i. Disabling copy constructors:

```
        // C++ program to disable the usage of
// copy-constructor using delete operator
#include <iostream>
using namespace std;

class A {
public:
        A(int x): m(x)
        {
        }

        // Delete the copy constructor
        A(const A&) = delete;

        // Delete the copy assignment operator
        A& operator=(const A&) = delete;
        int m;
};

int main()
{
        A a1(1), a2(2), a3(3);

        // Error, the usage of the copy
        // assignment operator is disabled
        a1 = a2;

        // Error, the usage of the
        // copy constructor is disabled
        a3 = A(a2);
        return 0;
}
```

ii. Disabling undesirable argument conversion

```
   // C++ program to disable undesirable argument
// type conversion using delete operator
#include <iostream>
using namespace std;

class A {
```

```
public:
        A(int) {}


        // Declare the conversion constructor as a
        // deleted function. Without this step,
        // even though A(double) isn't defined,
        // the A(int) would accept any double value
        // for it's argumentand convert it to an int
        A(double) = delete;
};

int main()
{
        A A1(1);

        // Error, conversion from
        // double to class A is disabled.
        A A2(100.1);
        return 0;
}
```

What are the advantages of explicitly deleting functions?

i. Deleting of special member functions provides a cleaner way of preventing the compiler from generating special member functions that we don't want. (As demonstrated in 'Disabling copy constructors' example).

ii. Deleting of normal member function or non-member functions prevents problematic type promotions from causing an unintended function to be called (As demonstrated in 'Disabling undesirable argument conversion' example).

2. Constructor delegation

-> It is a feature where one constructor in same class calls other constructor in the same class.

-> Normally set of code in one constructor is repeated in other constructor of same class which leads to code redundancy.

Example 1:

```
  A()
{
x = 0;
y = 0;
z = 0;
}
A(int z)
{
//Below 2 lines are redundant
x=0;
y=0;
this->z=z
}
```

-> Above problem can be solved using **init()**. We can write body of init function and call that function in each constructor.

Example 2:

```
void init()
{
x=0;
y=0;
}

A()
{
init();
z = 0;
}
A(int z)
{
init();
this->z=z
}
```

-> But there is ***drawback*** by using init. It is not readable and since init is not a constructor, it is called during normal program flow and member variables are set and memory is allocated dynamically which leads to addiional complexity like handling new initialization and reinitialization cases properly.

-> To solve above problem (redundant code), we can use ***constructor delegation.***

-> In constructor delegation, it allows us to call a constructor by placing it in the initializer list of other constructors.

Example 3:

```
A()
{
x = 0;
y = 0;
z = 0;
}

// Constructor delegation
A(int z) : A()
{
this->z = z; // Only update z
}
```

3. In class initializers

-> It is basically defined as initializing the field and name in place of declaration.

-> Before c++11, to initialize class member with default value, it had to be done through initialization list in a constructor.

Example 1: Prior to c++ 11

```
class SimpleType {
   int field;
   std::string name;
```

```
    SimpleType() : field(0), name("Hello World") { }
}
```

-> From the above we can see to initialize the class member with default value, it had to be done through initialization list in constructor

Example 2: After c++11

```
class SimpleType {
    int field =0;
    std::string name {“Hello World”};

    SimpleType() :  { }
}
```

-> After c++11, class members can be initialized directly with default value without setting values inside a constructor.

-> This feature is called non static data member initialization or NSDMI.

How NSDMI works?

Let us learn the working with some examples

Example 1:

```
// Implementation of initA() and initB()

int initA() {
    std::cout << "initA() called\n";
    return 1;
}

std::string initB() {
    std::cout << "initB() called\n";
    return "Hello";
}



class SimpleType
{
```

```
    int a { initA() };
    std::string b { initB() };

    SimpleType() { }
    SimpleType(int x) : a(x) { }
};
```

-> Now we can use:

```
std::cout << "SimpleType t0\n";
SimpleType t0;
std::cout << "SimpleType t1(10)\n";
SimpleType t1(10);
```

-> The output is:

```
    SimpleType t0:
initA() called
initB() called
SimpleType t1(10):
initB() called
```

-> From the above output we can observe that t0 is default initialized and other one comes from constructor parameter. For t1, one value is default initialized and other one comes from the constructor parameter.

Example 1.1:

-> Compiler expands the following code:

```
       int a { initA() };
    std::string b { initB() };

    SimpleType() { }
    SimpleType(int x) : a(x) { }
```

into the following:

```
int a { initA() };
    std::string b { initB() };

    SimpleType(): a(initA()), b(initB()) { }
    SimpleType(int x) : a(x), b(initB()) { }
```

4. Uniform initializers

-> Allows usage of consistent syntax to initialize variables and objects ranging from primitive type to aggregates. It introduces brace-initialization that uses braces({}) to enclose initializer values.

-> Syntax: type var_name{arg1, arg2, ... argn}

-> Some examples of brace initialization are:

i. int i{} //unintialized built-in type.
ii. int j{10}; // initialized built in type
iii. int a[]{1,2,3,4}// aggregate initialization
iv. X x1{}; //default constructor
v. X x2{}; //parameterized constructor;
vi. X x4{x3}; //Copy constructor.

-> Some applications of uniform initialization are:

i. Initialization of dynamic allocated arrays:

```
int* pi = new int[5]{ 1, 2, 3, 4, 5 };
```

ii. Initialization of an array data members of a class:

```
A(int x, int y, int z) : arr{ x, y, z } {};
```

5. Initializer list

-> Used to initialize data members of a class. List members to be initalized is indicated with constructor as comma seperated list followed by a colon.

-> Example:

```
Point(int i = 0, int j = 0):x(i), y(j) {}
```

-> The above example, x and y is initalized inside the constructor.

-> Some of applications of Initializer list are:

i. For initialization of non-static const data members:

```
class Test {
 const int t;
 public:
 Test(int t):t(t) {} //Initializer list must be used
 int getT() { return t; }
     };
```

const data members must be initialized in initializer list because no memory is allocated seperately for const data member, it is folded in the symbol table due to which we need to initialize it in the initializer list.

ii. For initialization of reference members:

```
    class Test {
     int &t;
public:
    Test(int &t):t(t) {} //Initializer list must be used
     int getT() { return t; }
};
```

In the above example, t is a reference member of Test class and is initialized using Initializer List.

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

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

class A {
  int i;
public:
  A(int );
};

A::A(int arg) {
   i = arg;
   cout << "A's Constructor called: Value of i: " << i << endl;
}
```

```
// Class B contains object of A
class B {
  A a;
public:
  B(int );
};

B::B(int x):a(x) { //Initializer list must be used
  cout << "B's Constructor called";
}

int main() {
  B obj(10);
  return 0;
}
/* OUTPUT:
A's Constructor called: Value of i: 10
B's Constructor called
*/
```

From the above example, we can observe that object "a" of class A is data member of class "B"(class B contains object of A) and A does not have default constructor. Here initializer list must be used to initialize "a".
If class A had both default and parameterized constructors, then initializer list is not must if we want to initialize object "a" using default constructor.

6. Explicit conversion operators

-> It is basically used where we do not want compiler to do implict conversions. By mentioning the function, operator or constructor as explicit compiler does not do implict conversion.

-> Example :

```
class T {
};

class X {
public:
    explicit operator T() const ;
};

void m() {
    X x;

    // with cast
    T tc = (T)x; // ok
```

```
        or
     T tc = T(x); // ok
         or
      T tc = tc.operator T(); // ok


    // without cast
    T t = x; // error: E2034 Cannot convert 'X' to 'T'
}
```

7. Read only objects

class Sample {
int x;
int y;
public:
constexpr Sample(int p,int q):x(p),y(q) { }
constexpr int getx() const { return x; }
constexpr int gety() const { return y; }
constexpr int getsum() const { return x+y;
}

int getdiff() const { return x – y; }
};

int main() {
constexpr Sample s1(10,20);
constexpr cx = s1.getx();
constexpr cs = s1.getsum();
constexpr int d = ***s1.getdiff(); //error***
Sample s2(11,12); //ok
int x = s2.getx();
constexpr int cy = ***s2.gety(); //error***
}

8. Explicit inheritance of constructors

class A {
int x;
public:A(int p):x(p) { }
};
class B : public A

{
int y;
public:
using A::A;
B(int p,int q):A(p),y(q) { }
};
class C : public B {
public:
using B::B;
};
B b1(10,20); //actual ctor
B b2(15); //inherited ctor
C c1(11,12); //inherited ctor
C c2(18); //inherited ctor

-> Here in above example, we can observe that in class B, we inherit all the constructors of class A (***using A::A;***) and in class C, we inherit all constructors of class B(**using B::B).**

9. Type Traits

-> It gives ability to inspects and transform the properties of types.(used in templates).

-> For example: given a generic type T, it could be int,bool, std::vector or anything. With type trait it is possible to ask compiler some questions like: is it an integer? Is it a function? Is it a pointer or class? Does it have destructor? Will it throw exceptions.

-> Extremely useful in conditional compilation where we instruct compiler to pick the right path according to the type of input.

-> Example: This example demonstrates function that byte swaps values.

```
template <typename T>
T byte_swap( T value ) {
unsigned char *bytes = reinterpret_cast< unsigned char
*>( &value );
 for (size_t i = 0; i < sizeof( T ); i += 2) {
```

```
// Take the value on the left and switch it
// with the value on the right
unsigned char v = bytes[ i ];
bytes[ i ] = bytes[ i + 1 ];
bytes[ i + 1 ] = v;
}
return value;
}
```

-> Here we have one problem: There is no control over what type of T gets passed in. We can pass double, char and it scramble with byte of random memory you dont own and possibly causing crash.

9. i What is type trait?

-> It is a simple template struct that contains a member constant, which in turn holds the answer to the question the type trait asks or the transformation it performs.

-> Example 1, let's take a look at std::is_floating_point, one of the many type traits defined by the C++ Standard Library in the `<type_traits>` header:

```
 template<typename T>
struct is_floating_point;
```

-> This type trait tells whether type T is floating point or not. Member constant called – value for type traits that ask question- will be either set to true or false according to the type passed in as template argument.

-> Example 2: std::remove_reference is a type trait that alters the type T it takes in input:

```
 template<typename T>
struct remove_reference;
```

-> This type trait basically turns T& into T. The member constant — called `type` for those type traits that modify a type — contains the result of the transformation.

9. ii How to use type trait?

```
#include <iostream>

#include <type_traits>
```

```
class Class {};

int main()
{
    std::cout << std::is_floating_point<Class>::value << '\n';
    std::cout << std::is_floating_point<float>::value << '\n';
    std::cout << std::is_floating_point<int>::value << '\n';
}
```

Output:

0

1

0

-> By seeing the output we can observe that value of Class is 0(false) and float is 1(true) and 0(false) for integer.(This is for *is_floating_point* type trait).

9.iii How does type trait work?

-> In the above example, we were passing 3 different types to template struct *std::is_floating_point:* a custom *Class* type, float and a int.

-> Compiler generates 3 different structs under the hood.

```
struct is_floating_point_Class {
    static const bool value = false;
};

struct is_floating_point_float {
    static const bool value = true;
};

struct is_floating_point_int {
    static const bool value = false;
};
```

-> Being static you need to access the member constant with the :: syntax.