**Classes & Objects**

Default controls

Constructor delegating

In class initializers

Uniform initializers

Initializer list

Explicit conversion operators

Read only objects

Explicit Type Conversions

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;

}

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).