
Week 5:Typecasting, const, static

— By, Mahrukh Khan —

```
#include<iostream>
using namespace std;
int main()
{
    cout<<9/2;
}
```

C:\Users\Administrator\Desktop

4

Process exited after 0.0

```
1  #include<iostream>
2  using namespace std;
3  int main()
4  {
5      cout<<9/2.0;
6  }
```

C:\Users\Administrator\Desktop\CP-Spring

4.5

Process exited after 0.03652 sec

Press any key to continue . . .

```

#include<iostream>
using namespace std;
int main()
{
    int a=9,b=2;
    double ans;
    ans=a/b;
    cout<<ans;
}

```

C:\Users\Administrator\Desktop\CP-Spring2019\Quizzes\QUIZ 1\Untitled1

```

4
-----
Process exited after 0.01558 seconds with return value 4
Press any key to continue . . .

```

```

1 #include<iostream>
2 using namespace std;
3 int main()
4 {
5     int a=9;
6     double b=2;
7     cout<<a/b;
8 }

```

C:\Users\Administrator\Desktop\CP-Spring2019\Quizzes\QUIZ 1\Untitled1

```

4.5
-----
Process exited after 0.1587 seconds with return value 4.5
Press any key to continue . . .

```

Typecasting

How to convert from type int to type double that you can use with either a constant or a variable?

Typecasting is making a variable of one type, such as an int, act like another type, such as char..

```
static_cast<double>(9)
```



This expression is called Typecast.

- **Implicit cast:** Perform by the compiler automatically.
- **Explicit cast:** Perform by the programmer. Data loss might occur.

Implicit Casting

- 'automatic type conversion' done by the compiler on its own, without any external trigger from the user.
- Generally takes place when in an expression more than one data type is present. In such condition type conversion (type promotion) takes place to avoid lose of data.
- All the data types of the variables are upgraded to the data type of the variable with largest data type.

**bool -> char -> short int -> int -> unsigned int -> long -> unsigned -> long
-> float -> double -> long double**

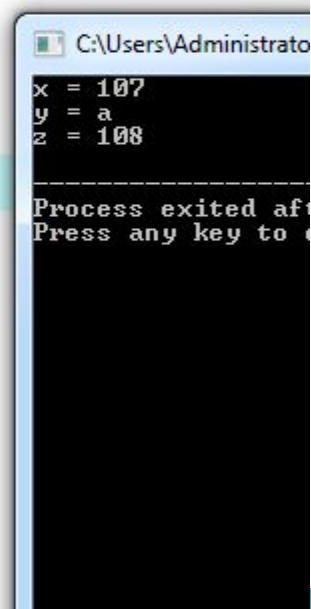
Example

```
using namespace std;
int main()
{
    int x = 10; // integer x
    char y = 'a'; // character c
    x = x + y;

    float z = x + 1.0;

    cout << "x = " << x << endl
         << "y = " << y << endl
         << "z = " << z << endl;

    return 0;
}
```



A screenshot of a Windows command prompt window. The title bar shows the path 'C:\Users\Administrato'. The window contains the following text: 'x = 107', 'y = a', 'z = 108', followed by a dashed line and the message 'Process exited after' and 'Press any key to c'.

```
C:\Users\Administrato
x = 107
y = a
z = 108
-----
Process exited after
Press any key to c
```

Explicit Casting

- User defined typecast for the result to make it to a particular data type.
- In C++, it can be done by two ways:
 - By Assignment
 - Also known as forceful casting where *type* indicates the data type to which the final result is converted.
 - By cast Operator
 - A Cast operator is an **unary operator** which forces one data type to be converted into another data type.

```
(type) expression
```

RTTI

Run-time type information or **run-time type identification (RTTI)** is a feature of C++ that does safe typecast by telling information about an object's datatype at runtime.

1. Static Cast
2. Dynamic Cast
3. Const Cast
4. Reinterpret Cast

Const member function

The idea of const functions is not allow them to modify the object on which they are called. It is recommended practice to make as many functions const as possible so that accidental changes to objects are avoided.

```
// on this function.  
int getValue() const {  
    value=2;  
    return value;  
}
```

	Message
019\Quiz...	In member function 'int Test::getValue() const':
\Quizzes...	[Error] assignment of member 'Test::value' in read-only object

Syntax

For member functions defined outside of the class definition, the `const` keyword must be used on both the function prototype in the class definition and on the function definition:

```
class Test {
    int value;
public:
    Test(int v ):value(v)
    {
    }
    int getV() const;
};

int get_month() const;           // prototype in Date class definition

int Date::get_month() const     // method definition
{
    return month;
}

int Test::getV() const
{
    return value;
}
```

Const member function

Constant member function cannot access non-constant member functions

```
int getValue() const {  
    set(value);  
    return value;  
}  
void set(int value)  
{  
      
}  
}
```

	Message
Quiz...	In member function 'int Test::getValue() const':
izzes...	[Error] passing 'const Test' as 'this' argument of 'void Test::set(int)' discards qualifiers [-fpermissive]

Const member function

Constructors and Destructors cannot be const

Constructor and destructor are used to modify the object to a well defined state

```
class Time{  
public:  
    Time() const {} //error...  
    ~Time() const {} //error...  
};
```

Const data member

- To make a variable constant, simply put the const keyword either before or after the variable type
- Const variables *must* be initialized when you define them, and then that value can not be changed via assignment.
- Declaring a variable as const prevents us from inadvertently changing its value
- const variables can be initialized from non-const values
- Defining a const variable without initializing it will also cause a compile error
- Making a function parameter const does two things.
 - First, it tells the person calling the function that the function will not change the value of myValue.
 - Second, it ensures that the function doesn't change the value of myValue.

const data member

- Data members of a class may be declared as const. Such a data member **must** be initialized by the constructor using an initialization list. Once initialized, a const data member may never be modified, not even in the constructor or destructor.
- Data members that are both static and const have their own rules for initialization.

```
const int value;  
public:  
    Test(int v )  
    {  
        value=v; //ERROR  
    }  
    void setV(int v)  
    {  
        // value=v; //ERROR  
    }
```

```
const int value;  
public:  
    Test(int v ):value(v)  
    {  
    }  
    ... ..
```

Example

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

int main() {
    int age;
    const double gravity= 9.8; // preferred use of const before type
    int const sidesInSquare { 4 }; // okay, but not preferred
    //const int a; //ERROR const data must be defined
    cin>>age;
    const int aa { age };

    return 0;
}
```

const Objects

- Objects can be declared constant with the use of const keyword, just like any other C++ variable. Constant objects cannot change their state
- The const property of an object goes into effect after the constructor finishes executing and ends before the class's destructor executes. So the constructor and destructor can modify the object, but other methods of the class can't.
- Once a const class object has been initialized via constructor, any attempt to modify the member variables of the object is disallowed, as it would violate the const-ness of the object. This includes both changing member variables directly (if they are public), or calling member functions that set the value of member variables.

```
const Date birthday(7, 3, 1969);
```

```
int main() {  
    const Test t(2);  
    // t.setV(3); //ERROR  
    t.value=22; //ERROR  
  
    return 0;  
}
```


Const Object

- const objects can only call const methods. Objects that are not const can call either const or non-const methods.

```
class Test {  
    int value;  
public:  
    Test(int v ):value(v)  
    {  
    }  
    void setV(int v)  
    {  
        //value=v;  
    }  
    int getV() const  
    {  
        return value;  
    }  
};
```

```
int main() {  
    const Test t(2);  
    Test t1(3);  
    //t.setV(3); //ERROR trying to call non-const func  
    cout<<t.getV(); //OK as a const obj can only call const func  
    t1.setV(3); //non const obj can call non-const  
    cout<<t1.getV(); //non const obj can call const  
}
```

const member function

- A const method can be overloaded with a non-const version. The choice of which version to use is made by the compiler based on the context in which the method is called.
- The const version of the function will be called on any const objects, and the non-const version will be called on any non-const objects:

Example

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

class Test {
    int value;
public:
    Test(int v = 0)
    {value = v;}
    int getValue() const
    {
        cout<<value<<"constant"<<endl;
        return value;
    }
    int getValue(int a)
    {
        cout<<value<<"Non constant"<<endl;
        return value+a;
    }
};
```

```
int main() {
    Test t;
    cout << t.getValue();
    cout<<t.getValue(22);
    return 0;
}
```

E:\Mahrukh\constt.exe

```
0constant
00Non constant
22
-----
Process exited after 2.9 seconds w
Press any key to continue . . .
```

static members

- There is an important exception to the rule that each object of a class has its own copy of all the data members of the class. In certain cases, only one copy of a variable should be shared by all objects of a class. A static data member is used for these and other reasons.
- Static members exist as members of the class rather than as an instance in each object of the class.
- If static variables are not explicitly initialized then they are initialized to 0 of appropriate type.
- When we declare a member of a class as static it means no matter how many objects of the class are created, there is only one copy of the static member.

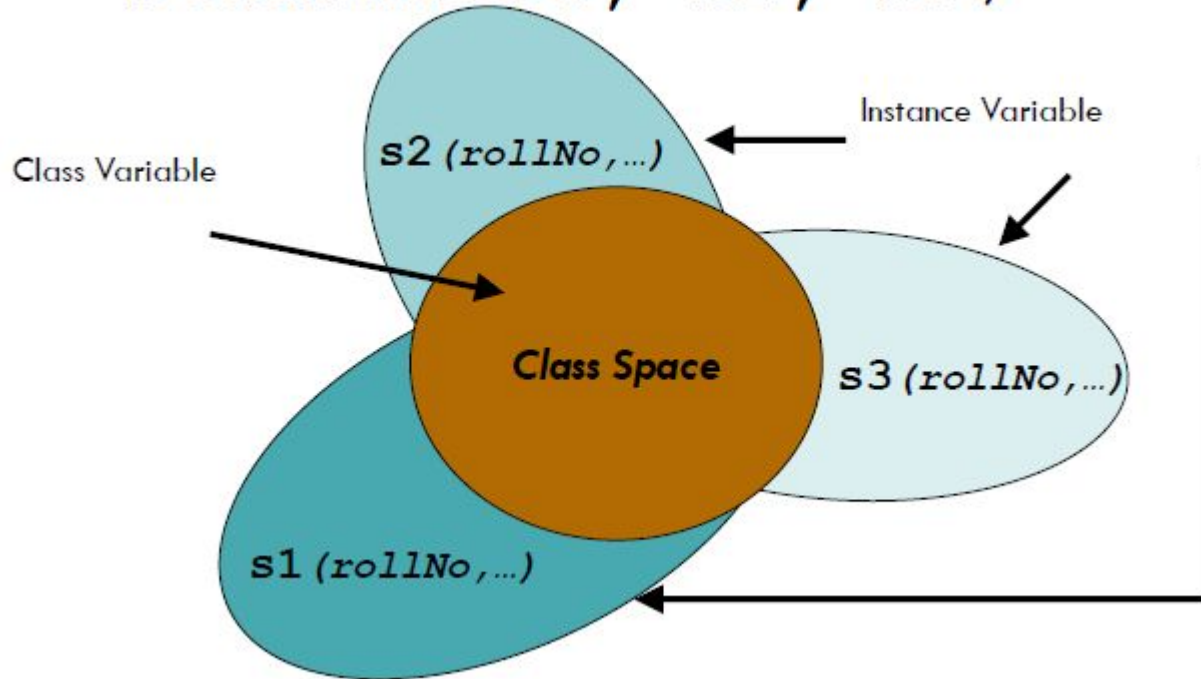
Static variable

- Lifetime of static variable is throughout the program life. They are created even when there is no object of a class. They remain in memory even when all objects of a class are destroyed
- Declared inside, defined outside. We can't put it in the class definition but it can be initialized outside the class as done in the following example by redeclaring the static variable, using the scope resolution operator `::` to identify which class it belongs to.

```
class X
{
public:
    static int i;
};
int X::i = 0; // definition outside class declaration
```

class v/s instance variable

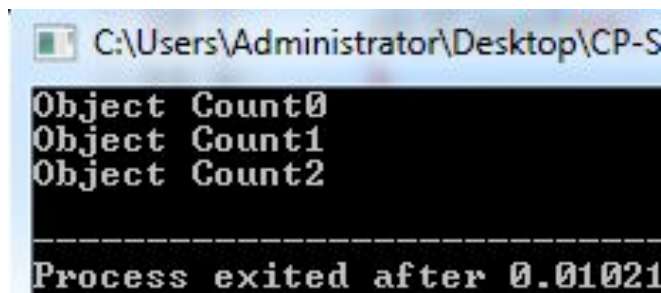
Student s1, s2, s3;



Example

```
class Person
{
    int a;
    static int objCount;
public:
    Person()
    {
        objCount++;
    }
    Person(int a)
    {
        this->a=a;
        objCount++;
    }
    static int getobjCount()
    {
        return objCount;
    }
};
```

```
int Person::objCount=0;
int main()
{
    cout<<"Object Count"<<Person::getobjCount()<<endl;
    Person p1;
    cout<<"Object Count"<<Person::getobjCount()<<endl;
    Person(2);
    cout<<"Object Count"<<Person::getobjCount()<<endl;
}
```



```
C:\Users\Administrator\Desktop\CP-S
Object Count0
Object Count1
Object Count2
-----
Process exited after 0.01021
```

the static member cannot be accessed
outside the class except for initialization*/

Static Member Functions

- The function that needs access to the members of a class, yet does not need to be invoked by a particular object, is called static member function”
- A static member function can be called even if no objects of the class exist and the static functions are accessed using only the class name and the scope resolution operator ::.

Static member functions

- A static member function can only access static data member, other static member functions and any other functions from outside the class.
- Static member functions have a class scope and they do not have access to the this pointer of the class.

Static function can only access static data member

```
1 #include <iostream>
2 using namespace std;
3
4 class Student {
5     public:
6         int rollnumber, age;
7         static int m;
8
9         static modifyAge () //Error
10    {
11        age = 10;
12    }
13};
```

```
1 #include <iostream>
2 using namespace std;
3
4 class Student {
5     public:
6         int rollnumber, age;
7         static int m;
8
9         static modifyAge () //Not an Error
10    {
11        m = 10;
12    }
13};
```

Static function can only access static member function

```
1 #include <iostream>
2 using namespace std;
3
4 class Student {
5     public:
6         int rollnumber, age;
7         static int m;
8         void print()
9     {
10         cout<<"Hello"<<endl;
11     }
12     static modifyAge () |
13     {
14         m = 10;
15         print(); //Error
16     }
17 };
```

```
1 #include <iostream>
2 using namespace std;
3
4 class Student {
5     public:
6         int rollnumber, age;
7         static int m;
8         static void print()
9     {
10         cout<<"Hello"<<endl;
11     }
12     static modifyAge ()
13     {
14         m = 10;
15         print(); //Not an Error
16     }
17 };
```

Static func can be called even when no object is created

```
int main() {  
    //Student S1;  
    Student::print();  
}
```

Const keyword with Pointers

When we use **const** with pointers, we can do it in two ways, either we can apply **const** to what the pointer is pointing to, or we can make the pointer itself a constant

```
int main() {  
    const int* u; //pointer is pointing to a const variable.  
                //Here, u is a pointer that can point to a const int type variable  
  
    int x = 1;   //Here, w is a pointer, which is const, that points to an int.  
    int* const w = &x; //Now we can't change the pointer, which means it will always  
                    //point to the variable x but can change the value that it points to,  
                    //by changing the value of x  
}
```

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.

Example

```
1 class Zee
2 {
3     int i;
4     mutable int j;
5     public:
6     Zee()
7     {
8         i = 0;
9         j = 0;
10    }
11
12    void fool() const
13    {
14        i++;    // will give error
15        j++;    // works, because j is mutable
16    }
17 };
18
19 int main()
20 {
21     const Zee obj;
22     obj.fool();
23 }
```

Const expr

C++ actually has two different kinds of constants.

- Runtime constants are those whose initialization values can only be resolved at runtime (when your program is running).
- Compile-time constants are those whose initialization values can be resolved at compile-time (when your program is compiling).

Example

```
constexpr double gravity (9.8); // ok, the value of 9.8 can be resolved at compile-time
constexpr int sum = 4 + 5; // ok, the value of 4 + 5 can be resolved at compile-time

std::cout << "Enter your age: ";
int age;
std::cin >> age;
constexpr int myAge = age; // not okay, age can not be resolved at compile-time
```

Reading Assignment

- Difference between Static const, const and static data member.

What we have done so far

- Introduction to OOP Basics : Class, Object, C++ Basics → Week 1
- Defining class, Member Functions inside and outside class, accessing class objects and functions, Inline Functions
- Introduction to OOP Pillars: Encapsulation, Inheritance, Abstraction, Polymorphism
- Access specifiers: Public/Private/Protected
- Accessor and Mutator Functions
- Constructor, Destructor, this pointer, Member Initializer
- Function Overloading, Constructor Overloading
- Typecasting
- static/ const data, mutable/ constexpr