Type Conversion in C++

A type cast is basically a conversion from one type to another. There are two types of type conversion:

Implicit Type Conversion

Implicit Type Conversion is also known as automatic type conversion done by the compiler on its own without any external statement from the user.

Implicit Type Conversion generally takes place when more than one data type is present in an expression (like int + float * double). 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

It is possible for implicit conversions to lose information, signs can be lost (when signed is implicitly converted to unsigned), and overflow can occur (when long is implicitly converted to float).

Example of Type Implicit Conversion:

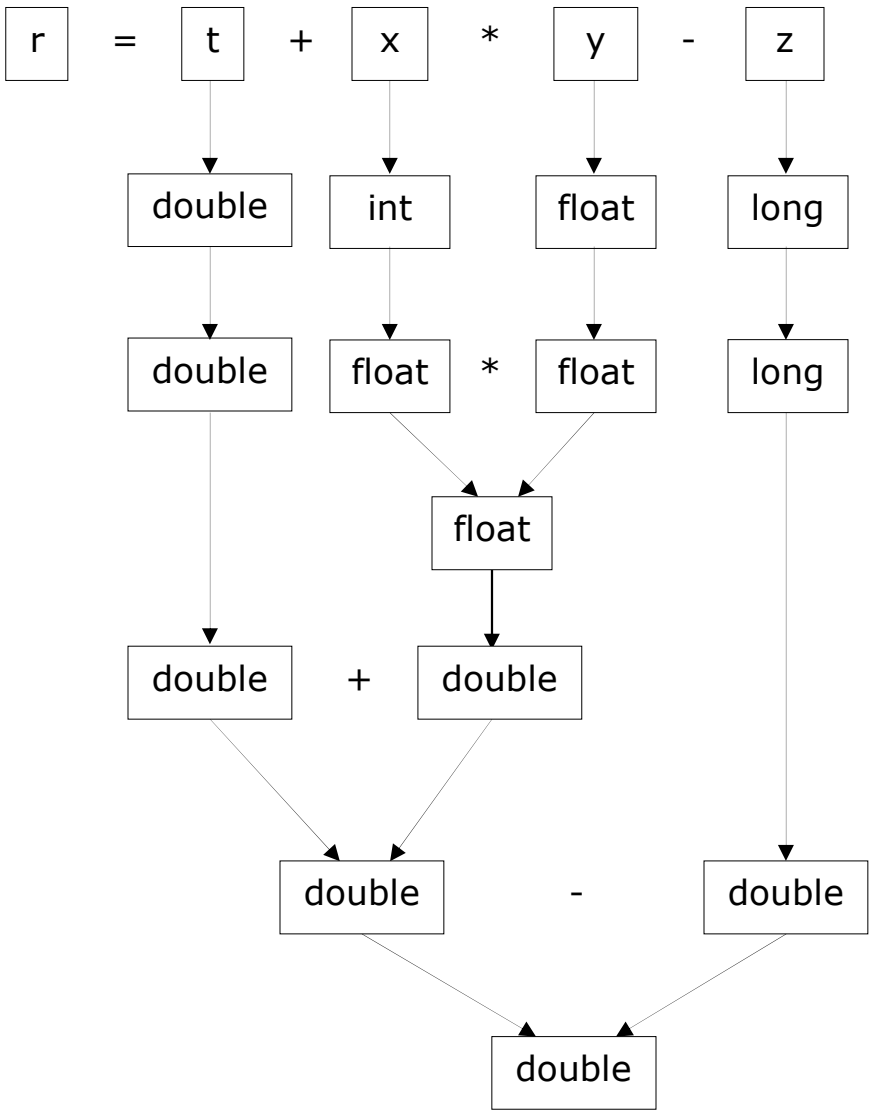
```
int main()
{
        int x = 10;   // integer x
        char y = 'a'; // character c

        // y implicitly converted to int. ASCII value of 'a' is 97
        x = x + y;
        // x is implicitly converted to float
        float z = x + 1.0;
        cout   << "x = " << x << endl << "y = " << y << endl
               << "z = " << z << endl;
        return 0;
}
```

Output: x = 107 y = a z = 108

Consider following code segment

```
int main ()
{
        int x;
        float y;
        long z;
        double t, r;
        r = t + x * y - z;
        return 0;
}
```



Explicit Type Conversion: This process is also called type casting and it is user-defined. Here the user can typecast the result to make it of a particular data type. In C++, it can be done by two ways:

1. Converting by assignment: This is done by explicitly defining the required type in front of the expression in parenthesis. This can be also considered as forceful casting.

   Syntax: (type) expression

   where type indicates the data type to which the final result is converted.

   Example:

```
int main()
{
   double x = 1.2;

   // Explicit conversion from double to int
   int sum = (int)x + 1;

   cout << "Sum = " << sum;

   return 0;
}
```

   Output: Sum = 2

2. Conversion using Cast operator: A Cast operator is a unary operator which forces one data type to be converted into another data type.
   C++ supports four types of casting:
   1. Static Cast
   2. Dynamic Cast
   3. Const Cast
   4. Reinterpret Cast

   ## Static Cast [static_cast Operator]

   The static_cast operator performs a non-polymorphic cast (Polymorphic cast means cast between two classes of which at least one of them have virtual function). It can be used to cast a base class pointer into a derived class pointer. and to cast a larger primitive type to smaller primitive type.

   ### Example:

```
int main() {
    float f = 3.5;
    int b = static_cast<int>(f);
    cout << b;
}
```

   Output: 3

Dynamic Cast [dynamic_cast Operator]

The static_cast operator performs a polymorphic cast (Polymorphic cast means cast between two classes of which at least one of them have virtual function). The dynamic_cast performs a runtime cast that verifies the validity of the cast. If the cast cannot be made, the cast fails and the expression evaluates to null. A dynamic_cast performs casts on polymorphic types and can cast a A* pointer into a B* pointer only if the object pointed by B* is a B object.

```
class Base
{
	protected:
		int bNum;
	public:
		Base(int num) {
			bNum = num;
		}
		virtual void showNum() {cout << bNum}
};

class Derived : public Base
{
	private:
		int dNum;
	public:
		Derived(int b, int d)
		{
			bNum = b;
			dNum = d;
		}
		void printData()
		{
			cout << "Inside Print Data Method";
		}
 };

int main()
{
	Base *b1 = new Derived(10, 20);
	Derived *d1 = dynamic_cast<Derived*>(b1);

	// Dynamic Cast Fails in Following Code
	Base *b2 = new Base(10);
	Derived *d2 = dynamic_cast<Derived*>(b2);
```

```
        // Static Cast Work Fine in Following Code
        Derived *d2 = static_cast<Derived*>(b2);

        delete b1;
        delete b2;
        delete d1;
        delete d2;

        return 0;
}
```

1. The above example conversion from b1 to d1 works because b1 is actually pointing to a Derived object, so converting b1 into a Derived pointer d1 is successful.
2. Conversion from b2 to d2 fails because b2 is actually pointing to a Base object, so converting Base pointer b2 into a Derived pointer d2 is not successful.
3. When dynamic_cast fails, the result of the conversion will be a null pointer.

Constant Cast [const_cast Operator]

The const_cast operator explicitly overrides const or volatile in a cast. The target type must be the same as the source type.

The const_cast operator can be used to remove restriction of modification on this pointer inside a const member function. Consider the following code segment in which inside const member function funConstant(), (this) is treated by the compiler as (const student* const this), i.e. (this) is a constant pointer to a constant object, thus compiler doesn't allow to change the data members through (this) pointer. The const_cast changes the type of (this) pointer to (student* const this).

```
class Student
{
        private:
                int rollNo;
        public:
                Student(int r) { rollNo = r}

                void funConstant () const {
                        ( const_cast <Student*> (this) )-> rollNo = 1005;
                }

                int getRollNum()  { return roll; }
};
int main(void)
```

```
{
   student stu(1003);
   cout << "Old roll number: " << stu.getRollNum() << endl;
   stu.fun();
   cout << "New roll number: " << stu.getRollNum() << endl;
   return 0;
}
```

Output:
Old roll number: 3
New roll number: 5

Reinterpret Cast [reinterpret_cast Operator]

It is used to convert one pointer of another pointer of any type, no matter either the class is related to each other or not. It does not check if the pointer type and data pointed by the pointer is same or not. The reinterpret_cast operator is a very special and dangerous type of casting operator. And is suggested to use it using proper data type i.e., (pointer data type should be same as original data type).

Syntax:

```
data_type *ptr_var1 = reinterpret_cast <data_type *>(ptr_var2);
```

Return Type: It doesn't have any return type. It simply converts the pointer type.

Parameters: It takes only one parameter i.e., the source pointer variable (p in above example).

```
int main()
{
   int* p = new int(65);
   char* ch = reinterpret_cast<char*>(p);
   cout << *p << endl << *ch << endl;
   return 0;
}
```

Output: 65 A

Purpose for using [reinterpret_cast]

1. It can typecast any pointer to any other data type.
2. It is used when we want to work with bits.
3. If we use this type of cast then it becomes a non-portable product. So, it is suggested not to use this concept unless required.
4. It is only used to typecast any pointer to its original type.
5. Boolean value will be converted into integer value i.e., 0 for false and 1 for true.

```
class First {
public:
   void fun_a()
   {
      cout << " In class A\n";
   }
};

class Second {
      public:
            void funSecond()
            {
                  cout << " In class B\n";
            }
};

int main()
{
   Second* ptrs = new Second();

   // converting the pointer to object class B to class A
   First* ptrf = reinterpret_cast<First*>(ptrs);

   ptrf -> funSecond ();     // accessing the function of class First
   return 0;
}
```

Output: In class A

Advantages of Type Conversion:

This is done to take advantage of certain features of type hierarchies or type representations.
It helps to compute expressions containing variables of different data types.

What is inline Function?

The inline functions are a C++ enhancement feature to increase the execution time of a program. Functions can be instructed to compiler to make them inline so that compiler can replace those function definition wherever those are being called. Compiler replaces the definition of inline functions at compile time instead of referring function definition at runtime.
NOTE- This is just a suggestion to compiler to make the function inline, if function is big (in term of executable instruction etc) then, compiler can ignore the "inline" request and treat the function as normal function.

How to make function inline

To make any function as inline, start its definitions with the keyword inline.

```
class AB
{
	public:
		inline int add(int a, int b)
		{
			return (a + b);
		}
};
```

```
class AB
{
	public:
		int add(int a, int b)
};

inline int A:: add(int a, int b)
{
	return (a + b);
}
```

Need of Inline Function

When a normal function call instruction is encountered, the program stores the memory address of the instructions immediately following the function call statement, loads the function being called into the memory, copies argument

values, jumps to the memory location of the called function, executes the function codes, stores the return value of the function, and then jumps back to the address of the instruction that was saved just before executing the called function. Too much run time overhead.

The C++ inline function provides an alternative. With inline keyword, the compiler replaces the function call statement with the function code itself (process called expansion just like macro expansion in C Programming Language) and then compiles the entire code. Thus, with inline functions, the compiler does not have to jump to another location to execute the function, and then jump back as the code of the called function is already available to the calling program.

Advantages

1. It speeds up programs execution speed by avoiding function calling overhead.
2. No overhead of variables push/pop on the stack, when function calling happens.
3. No overhead of return call from a function.
4. It increases locality of reference by utilizing instruction cache.
5. By marking it as inline, function definition can be placed in a header file.

Disadvantages

1. It increases the executable size due to code expansion which may cause thrashing in memory. More number of page fault bringing down performance of program.
2. C++ inline function issue is resolved at compile time any change in inline function need to recompile all the code.
3. When used in a header, it makes your header file larger.

When inline Function is Preferred

1. Prefer to use inline function when performance is needed.
2. Prefer to use inline function over macros.

Some Key Points about inline Function

1. Demand for inline function is just a suggestion not compulsion. Compiler may or may not inline the functions you marked as inline. It may also decide to inline functions not marked as inline at compilation or linking time.
2. Inline works like a copy and paste under the control of the compiler, which is quite different from a pre-processor macro where code is forcibly replaced, that is not easy to debug.
3. All the member function declared and defined within class are Inline by default. So no need to define explicitly.

4. Virtual methods are not supposed to be inline. But when the compiler ensures the type of the object (the object was declared and constructed inside the same function body), even a virtual function will be inline because the compiler knows exactly the type of the object.
5. Most of the compiler would do in-lining for recursive functions but some compiler provides #pragmas.

Difference Between C and C++

	C Language	C++ Language
Approach of Programming	C supports procedural programming paradigm for code development.	C++ supports both procedural and object oriented programming paradigms.
Input and Output mechanism	C uses functions for input/output. For example scanf and printf.	C++ uses objects for input output. For example cin and cout.
Variable Support	C does not support reference variables.	C++ supports reference variables.
Special Functions	C has no support for virtual and friend functions.	C++ supports virtual and friend functions.
Dynamic Memory Allocation	C provides malloc() and calloc() functions for dynamic memory allocation, and free() for memory de-allocation. int *x = (int*) malloc(sizeof(int))	C++ provides new and delete operator for memory allocation and delete operator for memory de-allocation. int x = new int;
For Loop	Loop index is defined outside and before the loop. int i; for (i = 0; i < 10; i ++)	Loop index is defined outside and before the loop. for (int i = 0; i < 10; i ++)

Structure	In C structure can only have data members.
	
	struct test
	{
	int x, y;
	float z;
	};
		In C++ structure can have data members as well as member functions.
		
		struct test {
		int x, y;
		void print() {
		cout <<" Test";
		};

Call by Reference	In C pointers are used to support call by reference.	In C++ pointers as well as reference variables are used to support call by

	Fun(int *a, int *b)	reference. Fun(int &a, int &b)
Type Casting	In C there is only one way to cast between types int anInt = (int)aFloat;	C++ provides a variety of ways to cast between types: • static_cast • reinterpret_cast • const_cast • dynamic_cast
Code Replacement	Macro expansion is only mechanism is used to achieve code replacement. #define squre(x) (x*x)	Macro as well as Inline function is used to achieve code replacement mechanism. inline squre(int x) { return (x*x)}
Key words	C contains 32 keywords.	C++ contains 52 keywords.
Programming Style	Instead of focusing on data, C focuses on method or process.	C++ focuses on data instead of focusing on method or procedure.
Exception Handling	Direct support for exception handling is not supported by C.	Exception handling is supported by C++.

Static Data Members

Static Variables : Variables in a function, Variables in a class
Static Members of Class : Class objects and Functions in a class

Static variables in a Function

When a local variable is declared as static then space is allocated to that variable for the lifetime of the program. Space for the static variable is allocated only once even if the function is called multiple times and the value of static variable after previous call is saved until next function call.

```
void funStaticLocal()
{
   static int count = 0;
   cout << count << " ";
   count++;
}

int main()
{
   for (int i=0; i<5; i++)
      funStaticLocal ();
   return 0;
```

```
}
```

Output: 0 1 2 3 4

In above program variable count is declared as static. So, its value is carried through the function calls. The variable count is not getting initialized for every time the function is called.

Static variables in a class

A static member of a class is shared by all objects of that class. A Static data member is initialized only once when first object is created. If no explicit initialization is there then it is implicitly initialized by zero. The Explicit initialization is put outside the class definition using scope resolution operator.

As the variables declared as static means they are allocated memory space in separate static storage. There cannot be multiple copies of same static variables for different objects so static variables cannot be initialized using constructors.

```
class StaticData
{
        public: static int num;
};

int main() {
        StaticData obj1;
        StaticData obj2;
        obj1.num = 2;
        obj2.num = 3;
        cout << "Value of Num for obj1="<<obj1.num
        cout << "Value of Num for obj2="<<obj2.num
}
```

Output:
Value of Num for obj1 = 0
Value of Num for obj2 = 0

In the above program we have tried to create multiple copies of the static variable num for multiple objects. But this didn't happen. So, a static variable inside a class should be initialized explicitly by the user using the class name and scope resolution operator outside the class as shown below:

```
class StaticData
{
```

```
        public: static int num;
};

int StaticData :: num = 100;

int main() {
   StaticData obj;
   cout << "Value of Num for obj = " << obj.num;
}
```

Output: Value of Num for obj = 100

Class objects as static

Just like variables, objects also when declared as static have a scope till the lifetime of program.

```
class StaticObject
{
  public:
        StaticObject ()
        {
                cout << "Constructor Call\n";
        }

        ~ StaticObject ()
        {
                cout << "Destructor Call\n";
        }
};

int main()
{
        int x = 0;
        if (x==0)
                StaticObject staticRef;
        cout << "End of main\n";
}
```

Output: Constructor Call
Destructor Call
End of main

In the above program the object is declared inside the if block as non-static. So, the scope of variable is inside the if block only. when the object is created the constructor is invoked and as soon as the controls of if block gets over the object staticRef lost its scope and immediately the destructor is invoked.

```
int main()
{
   int x = 0;
   if (x==0) {
      static StaticObject obj;
   }
   cout << "End of main\n";
}
```

Output : Constructor Call
End of main
Destructor Call

Now the destructor is invoked after the end of main. This happened because the scope of static object is throughout the life time of program.

Static member function of class

Static member function is in dependent of any particular object of its class. Static member function never required an object to be invoked although it can be invoked using object and the dot operator (.) but it is recommended to invoke the static members using the class name and the scope resolution operator.

Static member functions are allowed to access only the static data members or other static member function, they cannot access the non-static data members or member functions of the class.

1- Static member function can't be declared const as it doesn't have this pointer access.
2- Static member function can't be a virtual function. So static member function can't be overridden.
3- Instance member function and static member function of same prototype are not allowed together.

```
class StaticFun
{
  public:
       static void printMessage() {
```

```
            cout<<"Welcome to Static Function";
        }
};
int main()
{
   StaticFun:: printMessage ();
}
```

Output : Welcome to Static Function

We can define class member static using static keyword. 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.

A static member is shared by all objects of the class. All static data is initialized to zero when the first object is created, if no other initialization is present. 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.

Static Function Members

By declaring a function member as static, you make it independent of any particular object of the class. 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 ::.

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. You could use a static member function to determine whether some objects of the class have been created or not.

Constant Member Function of Class

The Constant member functions are member functions which are declared as constant in the program. A const member function restricts this pointer from modification this means that inside constant member function definition one can't make changes in an invoking object.

1- A const member function can be called by any type of object (Both const and non-const) but a non-const member function can only be called by non-const objects only.

2- An object declared const can only invoke const member function but non-constant object can call both const and non-const member functions.

```
class A
{
	int x;
	void func() const
	{
		x = 10;	// Compilation Error: Modification Not Allowed
	}
};
```

3- We cannot make constructors const because const objects initialize the values of their data members through constructors and if we make the constructor const, then the constructor would not change the values of the data members and the object would remain uninitialized.

Constant Data Members of Class

Constant data members are those data members which are declared with const keyword and assigned values only once. Const data members are assigned values in the constructor only. They are not assigned values during its declaration. The idea of const data member is to protect accidental changes made by programmer i.e. value of data member can't be changed unintentionally.

```
class ConstData
{
	const int x;
	public:
		A(int y)
		{
			x = y;
		}
};

int main()
{
	ConstData a(5);
	return 0;
}
```

Constant Objects of Classes

We can also make an object of class const. We make an object const by writing the const keyword at the beginning of the object declaration as shown below.

```
const ConstData obj;
```

A const class object is only initialized through the constructor. Modification of properties of constant object is not allowed. When a member function is invoked through const object then inside invoked member function this pointer is restricted for modification just like a const member function.

```
class ConstData
{
	public:
		int num;
		ConstData ()
		{
			num = 0;	//Initialize const data member
		}
		void changeNum() {
			num = num + 2;
		}

};

int main()
{
	const A obj;
	obj.x = 10;		//	Compilation error, Modification is not allowed

	obj.changeNum();	//	Compilation error, Cant invoke Non-Const member
				//	function using const object
	return 0;
}
```

The above program will give compilation error. Since we declared (obj) as a const object of class A, we cannot change its data members. When we created (obj), its constructor got called assigning a value 0 to its data member (num). Since (num) is a data member of a const object, we cannot change its value further in the program. So when we tried to change its value, we got compilation error.

We are trying to invoke non-const member function using const object this cause a compilation error.