

Type casting

Presented by,
Prof. Rupali Shende

Pitfalls of Op Overloading and Conversion

- Code can be innovative and readable but it may also be hard to understand.
- Similar meaning and syntax should be given to the overloaded operators else it makes difficult to understand the functionality.
- Ambiguity should be avoided.
- Remember that all operators cannot be overloaded.

Type Conversion and Its Types

- Type conversion is the method of converting one data type to another. There are two types of Type Conversions in C++:
- Implicit type conversion, and
- Explicit type conversion

Implicit Type Conversion

- The type conversion that is done automatically done by the compiler is known as implicit type conversion. This type of conversion is also known as automatic conversion.
- It does not require any effort from the programmer. The C++ compiler has a set of predefined rules. Based on these rules, the compiler automatically converts one data type to another.

Data Loss During Conversion

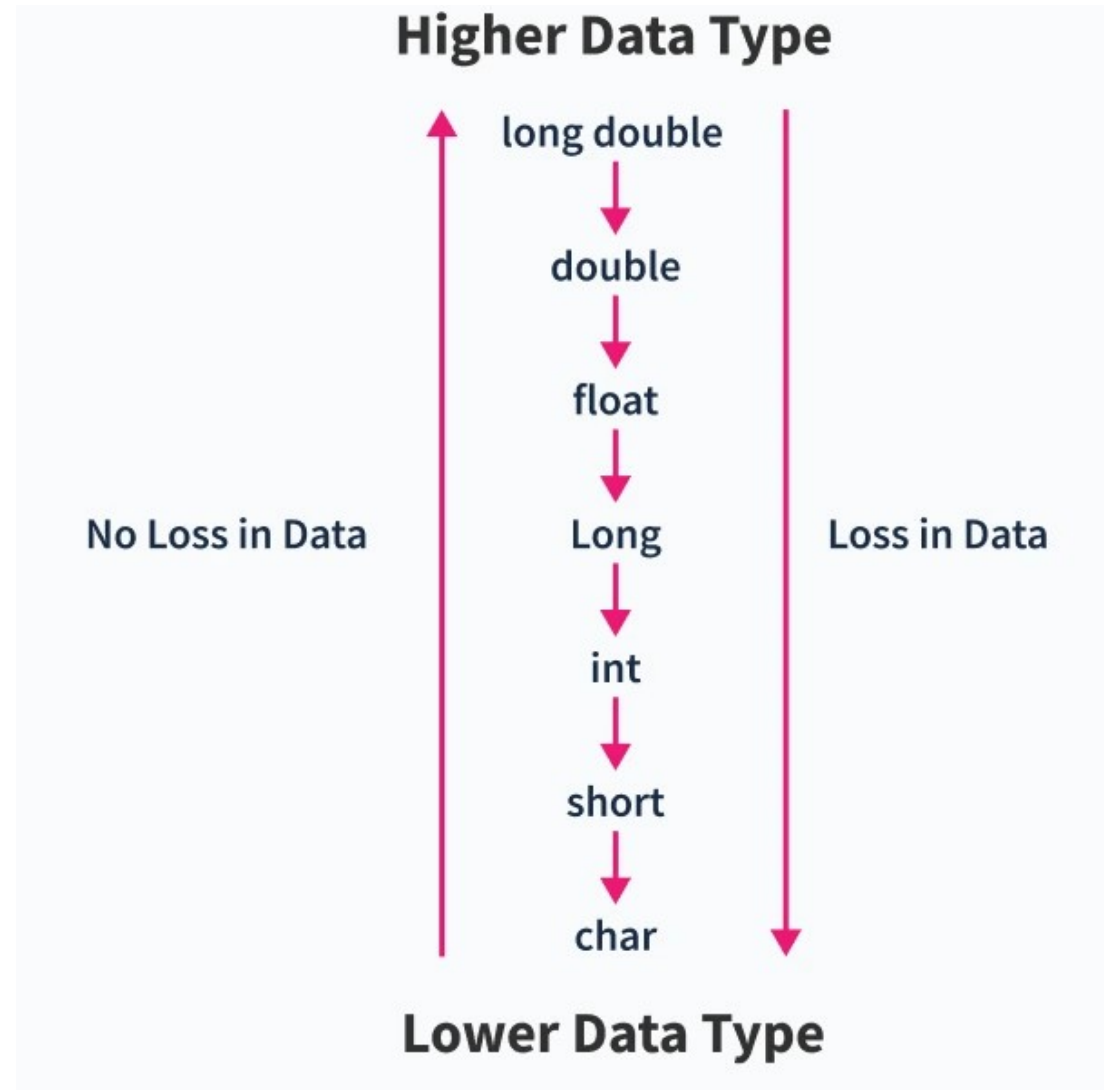
- When there is more than one data type present in an expression, there is a possibility of data loss because different data types are not compatible with each other.
- Data loss occurs if a variable converts from a higher data type to a lower data type. In order to avoid data loss, the compiler automatically converts all the data types to the highest data type present in the expression.
- This is called promotion.

Data Loss During Conversion

Order of the typecast in implicit conversion

The following is the correct order of data types from lower rank to higher rank:

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



Example

```
#include <iostream>
using namespace std;
int main()
{
int num; // declare int type variable
double num2 = 15.25; // declare and assign the double variable

// use implicit type conversion to assign a double value to int variable
num = num2;
cout << " The value of the int variable is: " << num << endl;
cout << " The value of the double variable is: " << num2 << endl;
;
return 0;
}
```

Output

The value of the int variable is: 15
The value of the double variable is: 15.25

Explicit Type Conversion

- Explicit Type Conversions are those conversions that are done by the programmer manually. In other words, explicit conversion allows the programmer to typecast (change) the data type of a variable to another type. Hence, it is also called typecasting. Generally, we use the explicit type conversion if we do not want to follow the implicit type conversion rules.
- Explicit type conversion in C++ can be done in two ways:
 1. Conversion using the Assignment Operator

Conversion Using the Assignment Operator

- This conversion is done by explicitly declaring the required data type in front of the expression. It can be done in two ways:
- **C-style type casting:**
- This type casting is usually used in the C programming language. It is also known as cast notation. The syntax for this casting is:
- `(datatype)expression;`
- **Function style casting**
- As the name suggests, we can perform explicit typecasting using function style notations. It is also known as old C++ style type casting. The syntax for this casting is:
- `datatype(expression);`

C-style type casting:

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

int main() {
    char char_var = 'a';
    int int_var;

    // Explicitly converting a character variable to integer variable
    int_var = (int) char_var; // Using cast notation

    cout << "The value of char_var is: " << char_var << endl;
    cout << "The value of int_var is: " << int_var << endl;

    return 0;
}
```

Output:

```
The value of char_var is: a
The value of int_var is: 97
```

In this example, we explicitly converted a char variable into an int. The result being, the character 'a' was converted to 97.

Function style casting

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

int main() {
    int int_var = 17;

    float float_var;

    float_var = float(int_var) / 2;
    // explicitly converting an int to a float

    cout << "The value of float_var is: " << float_var << endl;

    return 0;
}
```

Output:

```
The value of float_var is: 8.5
```

In this example, we used the function style casting to convert an int variable to float. This is why after dividing the variable by 2, we got 8.5 as the output. If we had not done that, the output would have been 8.

Mutable Keyword

- One such keyword is mutable, its working is the exact opposite of const keyword. In const keyword the value of a variable cannot be changed during the entire program, but what if someone feels a need to change it.
- Mutable keyword comes in handy when in a const declared object, you want to update a few constant data members without updating other data members. For example when you buy a property or land and sell it after some time. The dimension of the property remains the same, its location remains the same just the details of the owner needs to be changed. In this case, you can use the mutable keyword for changing the Owner's Name.

Example

```
#include <iostream.h>
#include <string.h>
using namespace std;
class property{
private:
    mutable char OwnerName[20];
    float area;
    char location[50];
public:
    property(char *o,float a,char *l)
    {
        strcpy(OwnerName ,o);
        area=a;
        strcpy(location,l);
    }

    void changeOwner(char *o)
const{
    strcpy(OwnerName,o);
}
```

```
void changelocation(char *l) {
    strcpy(location,l);
}
void display()const{
    cout <<endl<<"Owner Name="
"<<OwnerName<<endl<<"Area= "<<area<<endl<<"Location="
"<<location;
}
};
int main()
{
    const property p("Biharilal",89.6,"Faridabad");
    p.display();

    p.changeOwner("Ramchandra");
    p.display();

    // p.changelocation("Gurgaon");
    // p.display();
    return 0;
}
```

Output:

- Owner Name= Biharilal
- Area= 89.6
- Location= Faridabad
- Owner Name= Ramchandra
- Area= 89.6
- Location= Faridabad

Explicit keyword

```
2 using namespace std;
3
4 class Blah
5 {
6     public:
7
8         // Overloaded constructor
9         Blah(int blah)
10        {
11            m_blah = blah;
12        }
13
14        int GetBlah()
15        {
16            return m_blah;
17        }
18
19        private:
20            int m_blah;
21 };
22
23 void Ext_Blah (Blah blah)
24 {
25     int x = blah.GetBlah ();
26 }
27
28 int main() {
29     // your code goes here
30     Ext_Blah (3);
31 }
```

Upon executing the code, it can be seen that the code compiles without any error; but, take another look at the definition of Ext_Blah(). Its input parameter is a Blah object, but we are passing an int to this function from main.

There exists a constructor for Blah that takes an int; so, this constructor can be used to convert the parameter to the correct type. The compiler is allowed to do this for each parameter once.

Prefixing the explicit keyword to the constructor prevents the compiler from using that constructor for implicit conversions. It will now create a compiler error at the Ext_Blah(3) function call.

Explicit keyword

```
1 #include <iostream>
2 using namespace std;
3
4 class Blah
5 {
6     public:
7
8         // Overloaded constructor
9         explicit Blah(int blah)
10        {
11            m_blah = blah;
12        }
13
14        int GetBlah()
15        {
16            return m_blah;
17        }
18
19     private:
20         int m_blah;
21 };
22
23 void Ext_Blah (Blah blah)
24 {
25     int x = blah.GetBlah ();
26 }
27
28 int main() {
29     // your code goes here
30     Ext_Blah (3);
```

Output

0.95s

```
main.cpp: In function 'int main()':
main.cpp:30:14: error: could not convert '3' from 'int' to
'Blah'
    Ext_Blah (3);
               ^
```

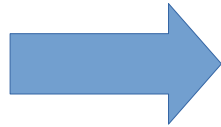
It is now necessary to call for a conversion explicitly with `Ext_Blah(Blah (3))`, as shown below:

```
2  using namespace std;
3
4  class Blah
5  {
6  public:
7
8      // Overloaded constructor
9      explicit Blah(int blah)
10     {
11         m_blah = blah;
12     }
13
14     int GetBlah()
15     {
16         return m_blah;
17     }
18
19 private:
20     int m_blah;
21 };
22
23 void Ext_Blah (Blah blah)
24 {
25     int x = blah.GetBlah ();
26 }
27
28 int main() {
29     // your code goes here
30     Ext_Blah (Blah(3));
31 }
```

Pointers to Base class

A base class pointer can point to a derived class object in C++, but we can only access base class members using the base class pointer.

```
class Base {  
    public:  
        void fun1();  
        void fun2();  
        void fun3();  
};
```



```
class Derived:public Base{  
    public:  
        void fun4();  
        void fun5();  
};
```

Pointers to Base class

Inside the main function, we have created an object b of class Base. Now on this b object, what are the functions that we can call? fun1(), fun2() and fun3() are the functions we can call. This is because all these functions are present inside the Base class.

```
int main(){  
    Base b;  
    b.fun1();  
    b.fun2();  
    b.fun3();  
}
```



```
int main(){  
    Derived d;  
    d.fun1();  
    d.fun2();  
    d.fun3();  
    d.fun4();  
    d.fun5();  
}
```

Now instead of the Base class, let us create an object Derived class as follows.

So total 5 functions we can call using the derived class object d.

Pointers to Base class

This is possible because the Derived class is inherited from the Base class in C++. Now, we are going to make changes here as follows. Here we have written the main function again with some changes.

Here we have taken a Base class pointer p. We can take a pointer of any type. A pointer is a variable that can store the address. Next, we assigned p to the object of the Derived class.

```
int main(){  
    Base *p;  
    p = new Derived ();  
    p->fun1();  
    p->fun2();  
    p->fun3();  
}
```

Pointers to Base class

- Can we call fun4 and fun5?

No, we cannot call these functions.

The point that we learn here is that you can have a base class pointer and a derived class object attached to it and you can call only those functions which are present in the base class.

You cannot call the functions which are defined in the derived class. But the object is a derived class object. So, you can call only those functions which are present in the base class because the pointer reference or pointer is the base class.

```

#include <iostream>
using namespace std;
class Base
{
    public:
        void fun1()
        {
            cout << "fun1 of Base Class" << endl;
        }
        void fun2()
        {
            cout << "fun2 of Base Class" << endl;
        }
        void fun3()
        {
            cout << "fun3 of Base Class" << endl;
        }
};
class Derived:public Base
{
    public:
        void fun4()
        {
            cout << "fun4 of Derived Class" << endl;
        }
        void fun5()
        {
            cout << "fun5 of Derived Class" << endl;
        }
};

```

```

int main()
{
    Base *p;
    p = new Derived ();
    p->fun1 ();
    p->fun2 ();
    p->fun3 ();
    //The following statements will throw error
    //p->fun4 (); //error: class Base has no member
    named fun4;
    //p->fun5 (); //error: class Base has no member
    named fun5;
}

```

Output:

```

fun1 of Base Class
fun2 of Base Class
fun3 of Base Class

```

Virtual function

- A virtual function is a member function that is declared within a base class and redefined by a derived class. To create virtual function, precede the function declaration in the base class with the keyword virtual.
- When a class containing virtual function is inherited, the derived class redefines the virtual function to suit its own needs.

Virtual function

- Base class pointer can point to derived class object. In this case, using base class pointer if we call some function which is in both classes, then base class function is invoked.
- But if we want to invoke derived class function using base class pointer, it can be achieved by defining the function as virtual in base class, this is how virtual functions support runtime polymorphism.

Virtual Function: Significance

- Virtual function makes C++ compiler to determine which function to be used at execution time based on the type of object pointed and not the type of pointer.
- Making the base pointer to point to different derived objects, different versions of the virtual function can be executed.

Example

```
#include
using namespace std;
class b
{
public:
    virtual void show()
    {
        cout<<"\n  Showing base class....";
    }
    void display()
    {
        cout<<"\n  Displaying base
class...." ;
    }
};
class d:public b
{
public:
    void display()
    {
        cout<<"\n  Displaying derived
class....";
    }
    void show()
    {
        cout<<"\n  Showing derived class....";
    }
};
```

```
int main()
{
    b B;
    b *ptr;
    cout<<"\n\t P points to base:\n" ;
        ptr=&B;
        ptr->display();
    ptr->show();

    cout<<"\n\n\t P points to drive:\n";
        d D;
        ptr=&D;
        ptr->display();
    ptr->show();
}
```

Output:

```
P points to base:
Displaying base class....
Showing base class....
```

```
    P points to drive:
```

```
Displaying base class....
Showing derived class....
```

Pure Virtual Function

- Pure virtual Functions are virtual functions with no definition.
- They start with virtual keyword and ends with = 0
- Syntax virtual show() = 0;
- Pure virtual function is also known as abstract function.

Characteristics of a pure virtual function

- A pure virtual function is a "do nothing" function. Here "do nothing" means that it just provides the template, and derived class implements the function.
- It can be considered as an empty function means that the pure virtual function does not have any definition.
- Programmers need to redefine the pure virtual function in the derived class as it has no definition in the base class.
- A class having pure virtual function cannot be used to create direct objects of its own. It means that the class is containing any pure virtual function then we cannot create the object of that class. This type of class is known as an abstract class.

Pure Virtual example

- `#include<iostream>`
- `using namespace std;`
- `class B {`
- `public:`
- `virtual void s() = 0; // Pure Virtual Function`
- `};`
-
- `class D:public B {`
- `public:`
- `void s() {`
- `cout << "Virtual Function in Derived class\n";`
- `}`
- `};`
-
- `int main() {`
- `B *b;`
- `D dobj;`
- `b = &dobj;`
- `b->s();`
- `}`

Output

Virtual Function in Derived
class

Virtual Table

- For every class that contains virtual functions, the compiler constructs a
- virtual table, vtable.
- The vtable contains an entry for each virtual function accessible by the class and stores a pointer to its definition.
- The virtual table is actually quite simple. First, every class that uses virtual functions (or is derived from a class that uses virtual functions) is given its own virtual table. This table is simply a static array that the compiler sets up at compile time. A virtual table contains one entry for each virtual function that can be called by objects of the class. Each entry in this table is simply a function pointer that points to the most-derived function accessible by that class.
- Second, the compiler also adds a hidden pointer that is a member of the base class, which we will call `*__vptr`. `*__vptr` is set (automatically) when a class object is created so that it points to the virtual table for that class.

Virtual Table

```
class Base
{
public:
    VirtualTable* __vptr;
    virtual void function1() {};
    virtual void function2() {};
};
```

```
class D1: public Base
{
public:
    void function1() override
};
```

```
class D2: public Base
{
public:
    void function2() override
};
```

Because there are 3 classes here, the compiler will set up 3 virtual tables: one for Base, one for D1, and one for D2. When a class object is created, *__vptr is set to point to the virtual table for that class. For example, when an object of type Base is created, *__vptr is set to point to the virtual table for Base. When objects of type D1 or D2 are constructed, *__vptr is set to point to the virtual table for D1 or D2 respectively.

Virtual Destructors

```
// CPP program without virtual destructor
// causing undefined behavior
#include <iostream>

using namespace std;

class base {
public:
    base()
    { cout << "Constructing base\n"; }
    ~base()
    { cout<< "Destructing base\n"; }
};

class derived: public base {
public:
    derived()
    { cout << "Constructing derived\n"; }
    ~derived()
    { cout << "Destructing derived\n"; }
};
```

```
int main()
{
    derived *d = new derived();
    base *b = d;
    delete b;
    getchar();
    return 0;
}
```

Output :

```
Constructing base
Constructing derived
Destructing base
```

Deleting a derived class object using a pointer of base class type that has a non-virtual destructor results in undefined behavior. To correct this situation, the base class should be defined with a virtual destructor.

Virtual Destructors

Making base class destructor virtual guarantees that the object of derived class is destructed properly, i.e., both base class and derived class destructors are called. For example,

```
#include <iostream>
```

```
using namespace std;
```

```
class base {  
public:  
    base()  
    { cout << "Constructing base\n"; }  
    virtual ~base()  
    { cout << "Destructing base\n"; }  
};
```

```
class derived : public base {  
public:  
    derived()  
    { cout << "Constructing derived\n"; }  
    virtual ~derived()  
    { cout << "Destructing derived\n"; }  
};
```

```
int main()  
{  
    derived *d = new  
    derived();  
    base *b = d;  
    delete b;  
    getchar();  
    return 0;  
}
```

Output :

```
Constructing base  
Constructing derived  
Destructing derived  
Destructing base
```

Abstract Class

- A class which contains at least one Pure Virtual function in it.
- Used to provide an Interface for its sub classes.
- Its derived Class must provide definition to all the pure virtual functions, otherwise it becomes abstract class.
- A class with at least one pure virtual function is called abstract class.
- Abstract class cannot be instantiated, but pointers and references of Abstract class type can be created.
- Abstract class can have normal functions and variables along with a pure virtual function.
- Abstract classes are mainly used for Up casting, so that its derived classes can use its interface.

Thank you!