Type Conversion in C++

By – Mohammad Imran

Type Conversion

The conversion of a variable from one data type to another is called Type Conversion. Type conversion in C++ is most commonly used to perform mathematical and logical operations on two variables with different data types.

By – Mohammad Imran

Type Conversion

Types

Type conversion is the method of converting one data type to another.

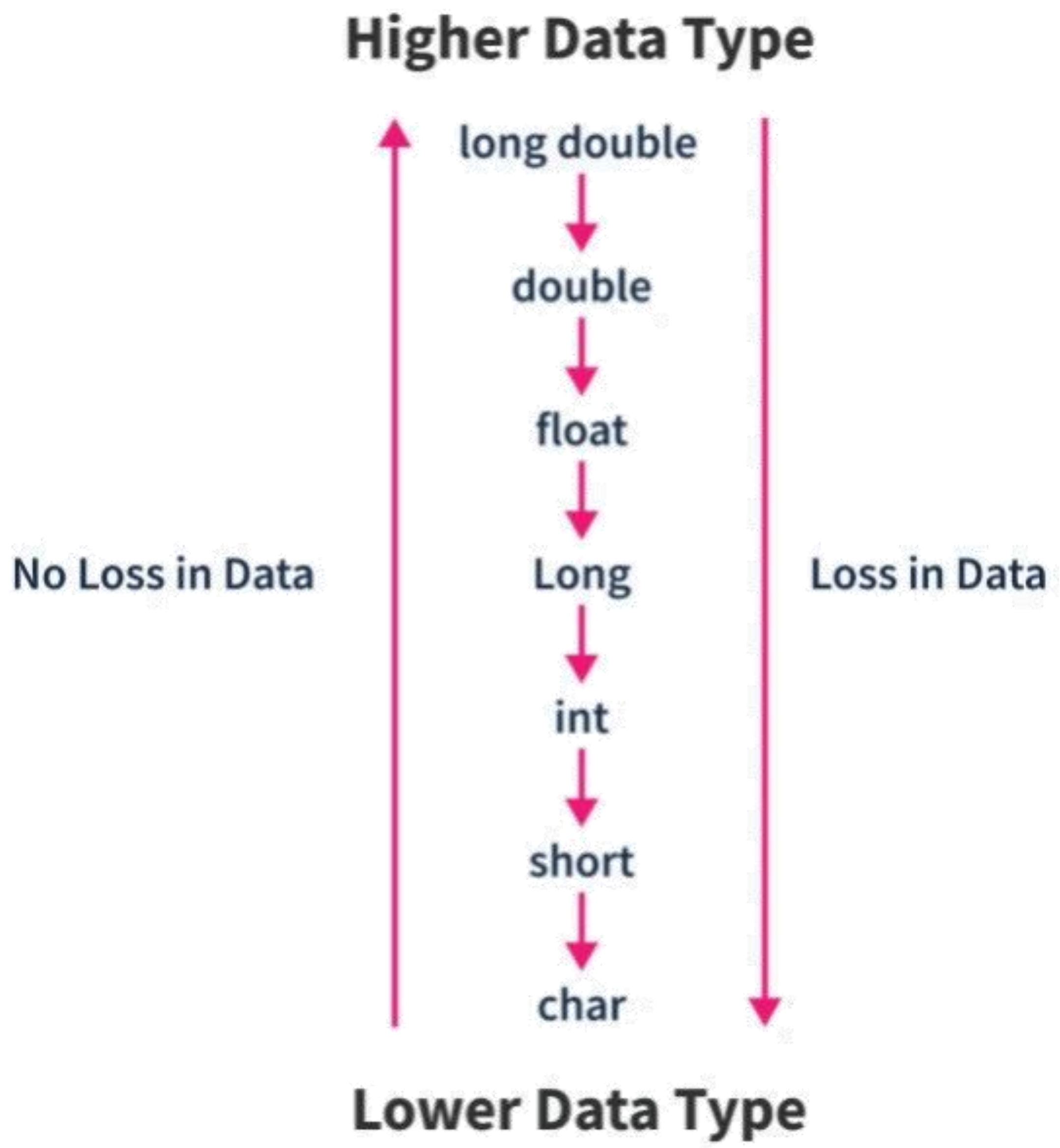
There are two types of Type Conversions in C++:

- ✓ Implicit Type Conversion
- ✓ Explicit Type Conversion

Implicit Type Conversion

The Implicit Type Conversion is where the type conversion is done automatically by the compiler. It does not require any effort from the programmer. The C++ compiler has a set of predefined rules. The compiler automatically converts one data type to another based on these rules. Therefore, implicit type conversion is also known as automatic type conversion.

Implicit Type Conversion



By – Mohammad Imran

Implicit Type Conversion

Example - 1

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

int main()
{
     int iVar;
     float fVar = 20.5;
     iVar = fVar;
     cout << "iVar = " << iVar << endl;
     cout << "fVar = " << fVar << endl;
     return 0;
}
```

OUTPUT

```
iVar = 20
fVar = 20.5
```

By – Mohammad Imran

Implicit Type Conversion

Example - 2

```
#include <iostream>
using namespace std;
class Conversion
{
    int integer;
    char character;
   public:
    Conversion()
    {
        integer = 50;
        character = 'a';
    }
```

```
void intChar()
{
    integer = integer + character;
    cout << "50 + 'a' = " << integer
    << endl;
}
void intFloat()
{
    float f = integer + 1.5;
    cout << "f + 1.5 = " << f << endl;
}
```

By – Mohammad Imran

Implicit Type Conversion

Example - 2

```
int main()
{
   Conversion cn;
   cn.intChar();
   cn.intFloat();
   return 0;
}
```

OUTPUT

```
50 + 'a' = 147
f + 1.5 = 148.5
```

By – Mohammad Imran

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.

Explicit type conversion in C++ can be done in two ways:

- ✓ Conversion using the Assignment Operator
- ✓ Conversion using the Cast Operator

Explicit Type Conversion

Conversion by Assignment Operator

Explicit typecasting in C++ is done using the assignment operator is also referred to as a forced casting.

It can be done in two ways:

- ✓ C-Style Type Casting
- ✓ Function Style Type Casting

Explicit Type Conversion

C-Style Type Casting

Example - 3

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

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

   int int_var;

   int_var = (int) char_var;

   cout << "char_var: " << char_var << endl;

   cout << "int_var: " << int_var << endl;

   return 0;

}
```

OUTPUT

```
char_var = a

Int_var = 97
```

By – Mohammad Imran

Explicit Type Conversion

Function Style Casting

Example - 4

```
#include <iostream>
using namespace std;
int main()
{
     int int_var = 17;
     float float_var;
     float_var = float(int_var) / 2;
     cout << "float_var: " << float_var << endl;
     return 0;
}
```

OUTPUT

```
float_var: 8.5
```

By – Mohammad Imran

Basic To Class Type Conversion

To perform this conversion, the idea is to use the constructor to perform type conversion during the object creation. Below is the example to convert **int** to **user-defined** data type:

By – Mohammad Imran

Basic To Class Type Conversion

Example - 5

```
#include <bits/stdc++.h>
using namespace std;

class Time
{
   int hour;
   int mins;
   public:
   Time()
   {
      hour = 0;
      mins = 0;
   }
   Time(int t)
   {
      hour = t / 60;
      mins = t % 60;
   }
   void Display()
   {
      cout << hour << " Hrs" << endl;
      cout << mins << " Mins" << endl;
   }
};
```

By – Mohammad Imran

Basic To Class Type Conversion

Example - 5

```
int main()
{
     Time T1;
     int dur = 95;
     T1 = dur;
     T1.Display();
     return 0;
}
```

OUTPUT

```
 1 hrs
35 mins
```

By – Mohammad Imran

Basic To Class Type Conversion

Example - 5

Explanation of main()

Object Creation:

```
Time T1
```

Creates an object **`T1`** of type **`Time`** using the default constructor.

At this point, **`T1`** has **`hour = 0`** and **`mins = 0`**

Integer Assignment:

```
int dur = 95; T1 = dur;
```

`dur` is an integer with a value of **`95`**. This line attempts to assign an integer value to **`T1`**.

By – Mohammad Imran

Basic To Class Type Conversion

Example - 5

Key Points

The assignment **`T1 = dur;`** is not directly supported by the class as it doesn't have an assignment operator that takes an **`int`**. This causes the code to fail to compile unless the compiler can implicitly convert int to **`Time`**.

If the compiler provides implicit conversion (which is non-standard and compiler-specific), it would use the constructor **`Time(int t)`** to convert **`dur`** into a **`Time`** object and then assign it to **`T1`**.

By – Mohammad Imran

Class To Basic Type Conversion

In this conversion, the **from** type is a class object and the **to** type is primitive data type. The normal form of an overloaded casting operator function, also known as a conversion function. Below is the syntax for the same:

Class To Basic Type Conversion

Example - 6

```
#include <bits/stdc++.h>
using namespace std;
class Time
{
   int hrs, mins;
   public:
   Time(int, int);
   operator int();
   ~Time()
   {
      cout << "Destructor" << endl;
   }
};
```

By – Mohammad Imran

Class To Basic Type Conversion

Example - 6

```
Time::Time(int a, int b)
{
   hrs = a;
   mins = b;
}
Time::operator int()
{
   cout << "Class To Basic" << endl;
   return (hrs * 60 + mins);
}
```

By – Mohammad Imran

Class To Basic Type Conversion

Example - 6

```
void TypeConversion(int hour, int mins)
{
   int duration;
   Time t(hour, mins);
   duration = t;
   cout << "Total Minutes: " << duration << endl;
   cout << "2nd method operator overloading " << endl;
   duration = t.operator int();
   cout << "Total Minutes are " << duration << endl;
   return;
}
```

By – Mohammad Imran

Class To Basic Type Conversion

Example - 6

```
int main()
{
   int hour, mins;
   hour = 2;
   mins = 20;
   TypeConversion(hour, mins);
   return 0;
}
```

OUTPUT

```
Class To Basic
Total Minutes: 140
2nd method operator overloading
Class To Basic
Total Minutes are 140
Destructor
```

By – Mohammad Imran

Type Conversion

Using Cast Operator

A Cast operator is an **unary operator** which forces one data type to be converted into another data type. C++ supports 4 types of casting:

- ✓ Static Cast
- ✓ Dynamic Cast
- ✓ Const Cast
- ✓ Reinterpret Cast

By – Mohammad Imran

Cast Operator

Static Cast

The Static Cast is the simplest among all four types of Casting in C++. The static cast can perform all the conversions that are done implicitly such as int to float or pointer to void.

It can perform upcast (conversion from a derived class to a base class) operations and downcast (conversion from a base class to a derived class) operations.

Syntax:

```
static_cast <datatype> (expression)
```

By – Mohammad Imran

Cast Operator

Static Cast **Example - 7**

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

int main()
{
   double num = 3.7 * 5.5;
   cout << "Before static_cast " << endl;
   cout << "num = " << num << endl;
   int cast_var;
   cast_var = static_cast <int> (num);
   cout << "After static_cast" << endl;
   cout << "cast_var = " << cast_var;
   return 0;
}
```

OUTPUT

```
Before static_cast
num = 20.35
After static_cast
cast_var = 20
```

By – Mohammad Imran

Static Cast | Behavior in Different Scenario | Example - 8

1. static_cast for primitive data type pointers:

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

int main()
{
   int a = 10;
   char c = 'a';
   int* q = (int*)&c;
   int* p = static_cast<int*>(&c); //Error
   return 0;
}
```

OUTPUT

```
Error
Undefined Behavior
```

By – Mohammad Imran

Static Cast

Behavior in Different Scenario

Explanation of the Code:

Casting the Address of ‘c’ to ‘int*’: `int *q = (int*)&c`

- ✓ (int*)&c performs a C-style cast, converting the address of the char variable c to an int*.
- ✓ This line essentially tells the compiler to treat the address of the char variable as if it points to an int.
- ✓ This cast can be unsafe because char and int have different sizes and alignment requirements. char is usually 1 byte, while int is typically 4 bytes. This can lead to alignment issues or incorrect data access.

Static Cast **Behavior in Different Scenario**

Explanation of the Code:

Static Cast of Address of ‘c’ to ‘int*’: `int *p = static_cast<int*>&c`

- ✓ static_cast<int*>(&c) attempts to use static_cast to convert the address of c to int*.
- ✓ Unlike a C-style cast, static_cast in C++ is designed to perform more type-safe conversions, but it still cannot change the fundamental nature of the data being pointed to.
- ✓ In this case, static_cast<int*>(&c) is effectively equivalent to the C-style cast and will result in the same issues.

2. Converting an Object using a User-Defined Conversion Operator

static_cast is able to call the conversion operator of the class if it is defined. Let’s take another example of converting an object to and from a class.

Static Cast | Behavior in Different Scenario | Example - 9

2. Converting an Object using a User-Defined Conversion Operator

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

class Integer
{
   int x;
  public:
   Integer(int x_in = 0): x{ x_in }
   {
      cout << "Constructor Called" << endl;
   }
```

By – Mohammad Imran

Static Cast | Behavior in Different Scenario | Example - 9

2. Converting an Object using a User-Defined Conversion Operator

```
    // user defined conversion operator to string type
    operator string()
    {
        cout << "Conversion Operator Called" << endl;
        return to_string(x);
    }
};
```

By – Mohammad Imran

Static Cast | **Behavior in Different Scenario** | **Example - 9**

2. Converting an Object using a User-Defined Conversion Operator

```
// Driver code
int main()
{
   Integer obj(3);
   string str = obj;
   obj = 20;

   // using static_cast for typecasting
   string str2 = static_cast<string>(obj);
   obj = static_cast<integer>(30);
   return 0;
}
```

OUTPUT

```
Constructor Called
Conversion Operator Called
Constructor Called
Conversion Operator Called
Constructor Called
```

By – Mohammad Imran

Static Cast | **Behavior in Different Scenario** | **Example - 9**

Explanation:

User-defined Conversion Operator:

- ✓ **operator string()** defines a conversion operator that allows an integer object to be implicitly converted to a string.
- ✓ This method prints "Conversion Operator Called" and converts the **integer x** to a string using **to_string(x)**.

Creating an integer Object:

integer obj(3); –

- ✓ Constructs an integer object with x initialized to 3.
- ✓ Outputs: "Constructor Called".

By – Mohammad Imran

Static Cast | Behavior in Different Scenario | Example - 9

Explanation:

Implicit Conversion to 'string':

string str = obj; –

- ✓ The **obj** is implicitly converted to a string using the conversion operator.
- ✓ **Outputs:** "Conversion Operator Called".
- ✓ The **str** variable is assigned the value "**3**".

Assignment with 'int':

obj = 20;

- ✓ This line is valid but doesn't involve a user-defined conversion operator. It assigns **20** to **x**.

By – Mohammad Imran

Static Cast | **Behavior in Different Scenario** | **Example - 9**

Explanation:

Explicit Conversion to ‘static_cast’:

string str2 = static_cast<string>(obj); –

- ✓ This explicitly uses **static_cast** to convert obj to string by invoking the user-defined conversion operator.
- ✓ **Outputs:** "Conversion Operator Called".
- ✓ **str2** is assigned the value "**20**" (since obj was modified to 20).

By – Mohammad Imran

Explanation:

Explicit Conversion to integer using ‘static_cast’:

obj = static_cast<integer>(30); –

- ✓ This creates a temporary **integer object** with **x** initialized to **30** using the constructor and assigns it to **obj**.
- ✓ **Outputs:** "Constructor Called".

Behavior in Different Scenario

3. Static Cast for Inheritance

static_cast can provide both upcasting and down casting in case of inheritance. The following example demonstrates the use of static_cast in the case of upcasting.

Static Cast | Behavior in Different Scenario | Example - 10

```cpp
#include <iostream>
using namespace std;
class Base
{
  public:
   Base()
   {
      cout << "Base Constructor Called" << endl;
   }
   virtual ~Base() // Virtual destructor for proper cleanup
   {
      cout << "Base Destructor Called" << endl;
   }
   void baseFunction()
   {
      cout << "Function in Base Class" << endl;
   }
};
```

Static Cast | Behavior in Different Scenario | Example - 10

```
class Derived : public Base
{
  public:
   Derived()
   {
      cout << "Derived Constructor Called" << endl;
   }
   ~Derived()
   {
      cout << "Derived Destructor Called" << endl;
   }
   void derivedFunction()
   {
      cout << "Function in Derived Class" << endl;
   }
};
```

By – Mohammad Imran

Static Cast | Behavior in Different Scenario | Example - 10

```
// Driver code
int main()
{
   Derived d1;

   // Implicit cast allowed
   Base* b1 = (Base*)(&d1);
   b1->baseFunction();

   // upcasting using static_cast
   Base* b2 = static_cast<Base*>(&d1);
   return 0;
}
```

OUTPUT

```
Base Constructor Called
Derived Constructor Called
Function in Base Class
Derived Destructor Called
Base Destructor Called
```

By – Mohammad Imran

Explanation: Difference between

```
Base* b1 = (Base*)(&d1);

Base* b2 = static_cast<Base*>(&d1);
```

The two statements **Base* b1 = (Base*)(&d1);** and **Base* b2 = static_cast<Base*>(&d1);** both perform casting from a Derived pointer to a Base pointer, but they differ in several important aspects regarding type safety, compiler checks, and general usage.

Static Cast | Behavior in Different Scenario | Explanation

1. C-Style Cast:

```
Base* b1 = (Base*)(&d1);
```

- **Description:** This is a C-style cast. It directly converts the address of d1 to a Base*.
- **Behavior**
  - ✓ **Type Safety:** The C-style cast is a more general cast that can perform various types of conversions, including static, reinterpret, and const casts. It does not provide compile-time checks to ensure the safety of the conversion.
  - ✓ **Error Handling:** If the cast is not safe or results in undefined behavior, the compiler will not catch it. The responsibility for ensuring the cast is correct lies with the programmer.
  - ✓ **Usage:** C-style casts are less preferred in C++ due to their lack of specificity and safety. They can lead to ambiguous or unsafe conversions.

By – Mohammad Imran

Static Cast | Behavior in Different Scenario | Explanation

1. Static Cast:

```
Base* b2 = static_cast<Base*>(&d1);
```

- **Description:** This is a static cast, which is a more explicit and type-safe cast introduced in C++.
- **Behavior**
  - ✓ **Type Safety:** The static performs compile-time checks to ensure that the cast is safe. For upcasting (casting from a derived class to a base class), it is generally safe and allowed, but it still performs checks to ensure that the conversion is logically correct.
  - ✓ **Error Handling:** If static cast encounters a type mismatch or an incorrect conversion, it will generate a compile-time error. This helps in catching potential issues earlier in the development process.

Static Cast

Behavior in Different Scenario

```
#include <iostream>
using namespace std;
class Base
{

};
class Derived: private Base //privately access
{

};
int main()
{
   Derived d1;
   Base* b1 = (Base*)(&d1);
   Base* b2 = static_cast<Base*>(&d1); // static_cast not allowed
   return 0;
}
```

By – Mohammad Imran

Static Cast

Behavior in Different Scenario

4. Cast ‘to and from’ Void Pointer

static_cast operator allows casting from any pointer type to void pointer and vice versa.

Static Cast | Behavior in Different Scenario | Example - 12

```
#include <iostream>
using namespace std;
int main()
{
   int i = 10;
   void* v = static_cast<void*>(&i);
   int* ip = static_cast<int*>(v);
   cout << *ip;
   return 0;
}
```

OUTPUT

```
10
```

By – Mohammad Imran

Cast Operator

Dynamic Cast

A cast is an operator that converts data from one type to another type. In C++, dynamic casting is mainly used for safe downcasting at run time. To work on **dynamic_cast** there must be one virtual function in the base class. A **dynamic_cast** works only polymorphic base class because it uses this information to decide safe downcasting.

Syntax:

```
dynamic_cast <datatype> (expression)
```

By – Mohammad Imran

Cast Operator

Dynamic Cast

Downcasting: Casting a base class pointer (or reference) to a derived class pointer (or reference) is known as downcasting. In figure 1 casting from the Base class pointer/reference to the "derived class 1" pointer/reference showing downcasting (Base ->Derived class).

Upcasting: Casting a derived class pointer (or reference) to a base class pointer (or reference) is known as upcasting. In figure 1 Casting from Derived class 2 pointer/reference to the "Base class" pointer/reference showing Upcasting (Derived class 2 -> Base Class).

As we mention above for dynamic casting there must be one Virtual function.

Cast Operator | Dynamic Cast | Example - 13

```
class Base
{
   public:
    virtual void print()
    {
        cout << "Base print" << endl;
    }
};
class Derived: public Base
{
   public:
    void print()
    {
        cout << "Derived print" << endl;
    }
};
```

By – Mohammad Imran

Cast Operator | Dynamic Cast | Example - 13

```
int main()
{
   Base* b = new Derived;
   Derived* d = dynamic_cast <Derived*>(b);
   if (d != NULL)
   {
      cout << "dynamic_cast done" << endl;
   }
   else
   {
      cout << "dynamic_cast not done" << endl;
   }
   return 0;
}
```

By – Mohammad Imran

Cast Operator | Dynamic Cast | Example - 13

OUTPUT

```
 [Error] cannot dynamic_cast 'bp' (of type 'class Base*') to
type 'class Derived2*' (source type is not polymorphic)
```

Understanding the Error

This error arises because dynamic_cast requires the source type to be polymorphic. In C++, a class is considered polymorphic if it has at least one virtual function.

By – Mohammad Imran

Cast Operator

Const Cast

The Const Cast is used to change an object's constant value or to remove the constant nature of any object. Const cast is generally used in programs with one or more objects with some constant value(s) that need to be changed at some point in the program.

For a const cast operation to be successful, the pointer and the source being cast should be of the same data type.

Syntax: `const_cast <datatype> (expression)`

Cast Operator | Const Cast | Example - 14

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

int main()
{
   const int var1 = 10;
   const int* ptr1 = &var1;
   cout << "The old value of ptr1 is: " << *ptr1 << endl;
   int* ptr2 = <int*> (ptr1);
   *ptr2 = 3;
   cout << "The new value of ptr1 is: " << *ptr1 << endl;
   return 0;
}
```

OUTPUT

```
The old value of ptr1 is: 10
The new value of ptr1 is: 3
```

By – Mohammad Imran

Cast Operator

Reinterpret Cast

The Reinterpret Cast is used to convert one pointer type to another, regardless of whether the classes are related. It does not check whether the pointer type and data pointed out by the pointer are the same. That is why reinterpreting cast should not be used unless required.

Reinterpret cast is mainly used to work with bits. It does not return any value. It directly changes the pointer type. If reinterpret cast is used on boolean values, the boolean values are converted to integers - 0 for false and 1 for true.

Syntax: `reinterpret_cast <datatype> (expression)`

By – Mohammad Imran

Cast Operator | Const Cast | Example - 15

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

int main()
{
   int* ptr = new int(98);
   char* ch = reinterpret_cast <char*> (ptr);
   cout << "The value of *ptr is: " << *ptr << endl;
   cout << "The value of ptr is: " << ptr << endl;
   cout << "The value of *ch is: " << *ch << endl;
   cout << "The value of ch is: " << ch << endl;
   return 0;
}
```

OUTPUT

```
The old value of ptr1 is: 10
The new value of ptr1 is: 3
```

By – Mohammad Imran

Coding Questions

By – Mohammad Imran

Question - 1

Write a C++ program to overload unary pre-increment operator (++) to increment by default 2 when used with object.

Sample Output

```
X = 10
++obj
X = 12
```

Click here to see code

By – Mohammad Imran

Question - 1

Write a C++ program to overload binary minus (-) operator using reference and friend function.

Click here to see code

By – Mohammad Imran

QUIZ

By – Mohammad Imran

Quiz - 1

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

int main()

{

   double pi = 3.14159;

   int intPi = static_cast<int>(pi);

   cout << "Value of pi is: " << pi << endl;

   cout << "Value of intPi is: " << intPi << endl;

   return 0;

}
```

ANSWER

```
Value of pi is: 3.14159
Value of intPi is: 3
```

By – Mohammad Imran

Quiz - 2

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

int main()

{

   int num = 65;

   char* ptr = reinterpret_cast<char*>(&num);

   cout << “Integer value is: " << num << endl;

   cout << “Character is: " << *ptr << endl;

   return 0;

}
```

ANSWER

```
Integer value is: 65
Character is: A
```

By – Mohammad Imran

Quiz - 3

```
#include <iostream>
using namespace std;
void printValue(const int* ptr)
{
   int* modifiablePtr = const_cast<int*>(ptr);
   *modifiablePtr = 100;
   cout << "Value of x: " << *ptr << endl;
}
int main()
{
   int x = 10;
   printValue(&x);
   cout << "Modified x: " << x << endl;
   return 0;
}
```

ANSWER

```
Value of x: 100
Modified x: 100
```

By – Mohammad Imran