About the content

This C++ material taken from the following websites:

- https://www.tutorialspoint.com/cplusplus
- https://www.w3schools.com/CPP
- https://www.geeksforgeeks.org/cpp-tutorial
- https://www.javatpoint.com/cpp-tutorial
- https://www.programiz.com/cpp-programming
- https://www.cplusplus.com

UNIT-IV

Module-1

Type Casting in C++

This section will discuss the type casting of the variables in the C++ programming language. Type casting refers to the conversion of one data type to another in a program. Typecasting can be done in two ways: automatically by the compiler and manually by the programmer or user. Type Casting is also known as Type Conversion.

Type Casting is divided into two types: Implicit conversion or Implicit Type Casting and Explicit Type Conversion or Explicit Type Casting.

Implicit Type Casting or Implicit Type Conversion

- It is known as the automatic type casting.
- It automatically converted from one data type to another without any external intervention such as programmer or user. It means the compiler automatically converts one data type to another.
- All data type is automatically upgraded to the largest type without losing any information.
- It can only apply in a program if both variables are compatible with each other.

char - sort **int** -> **int** -> unsigned **int** -> **long int** -> **float** -> **double** -> **long double**, etc.

> *Note: Implicit Type Casting should be done from low to higher data types. Otherwise, it affects the fundamental data type, which may lose precision or data, and the compiler might flash a warning to this effect.*

Program to use the implicit type casting in C++

Let's create an example to demonstrate the casting of one variable to another using the implicit type casting in C++.

```
#include <iostream>
using namespace std;
int main ()
{
```

```
    short x = 200;
    int y;
    y = x;
    cout << " Implicit Type Casting " << endl;
    cout << " The value of x: " << x << endl;
    cout << " The value of y: " << y << endl;

    int num = 20;
    char ch = 'a';
    int res = 20 + 'a';
    cout << " Type casting char to int data type ('a' to 20): " << res << endl;

    float val = num + 'A';
    cout << " Type casting from int data to float type: " << val << endl;
    return 0;
}
```

Output:

```
Implicit Type Casting
The value of x: 200
The value of y: 200
Type casting char to int data type ('a' to 20): 117
Type casting from int data to float type: 85
```

Explicit Type Casting or Explicit Type Conversion

- It is also known as the manual type casting in a program.
- It is manually cast by the programmer or user to change from one data type to another type in a program. It means a user can easily cast one data to another according to the requirement in a program.
- It does not require checking the compatibility of the variables.
- In this casting, we can upgrade or downgrade the data type of one variable to another in a program.
- It uses the cast () operator to change the type of a variable.

Syntax of the explicit type casting

```
(type) expression;
```

type: It represents the user-defined data that converts the given expression.
expression: It represents the constant value, variable, or an expression whose data type is converted.
For example, we have a floating pointing number is 4.534, and to convert an integer value, the statement as:

```
int num;
num = (int) 4.534; // cast into int data type
cout << num;
```

When the above statements are executed, the floating-point value will be cast into an integer data type using the cast () operator. And the float value is assigned to an integer num that truncates the decimal portion and displays only 4 as the integer value.

Program to demonstrate the use of the explicit type casting in C++
Let's create a simple program to cast one type variable into another type using the explicit type casting in the C++ programming language.

```
#include <iostream>
using namespace std;
int main ()
{
   // declaration of the variables
   int a, b;
   float res;
   a = 21;
   b = 5;
   cout << " Implicit Type Casting: " << endl;
   cout << " Result: " << a / b << endl; // it loses some information

   cout << " \n Explicit Type Casting: " << endl;
   // use cast () operator to convert int data to float
   res = (float) 21 / 5;
   cout << " The value of float variable (res): " << res << endl;

   return 0;
}
```

Output:

```
Implicit Type Casting:
Result: 4

Explicit Type Casting:
The value of float variable (res): 4.2
```

In the above program, we take two integer variables, a and b, whose values are 21 and 2. And then, divide a by b (21/2) that returns a 4 int type value.
In the second expression, we declare a float type variable res that stores the results of a and b without losing any data using the cast operator in the explicit type cast method.

Program to cast double data into int and float type using the cast operator

Let's consider an example to get the area of the rectangle by casting double data into float and int type in C++ programming.

```
#include <iostream>
using namespace std;
int main ()
{
   // declaration of the variables
   double l, b;
   int area;

   // convert double data type to int type
   cout << " The length of the rectangle is: " << endl;
   cin >> l;
   cout << " The breadth of the rectangle is: " << endl;
   cin >> b;
   area = (int) l * b; // cast into int type
   cout << " The area of the rectangle is: " << area << endl;

   float res;
   // convert double data type to float type
   cout << " \n \n The length of the rectangle is: " << l << endl;
   cout << " The breadth of the rectangle is: " << b << endl;
```

```
    res = (float) l * b; // cast into float type
    cout << " The area of the rectangle is: " << res;
    return 0;
}
```

Output:

```
The length of the rectangle is:
57.3456
The breadth of the rectangle is:
12.9874
The area of the rectangle is: 740

The length of the rectangle is: 57.3456
The breadth of the rectangle is: 12.9874
The area of the rectangle is: 744.77
```

There are other casting operators supported by C++

In type cast, there is a cast operator that forces one data type to be converted into another data type according to the program's needs. C++ has four different types of the cast operator:

1. Static_cast
2. dynamic_cast
3. const_cast
4. reinterpret_cast

Static Cast:

The static_cast is a simple compile-time cast that converts or cast one data type to another. It means it does not check the data type at runtime whether the cast performed is valid or not. Thus the programmer or user has the responsibility to ensure that the conversion was safe and valid.
The static_cast is capable enough that can perform all the conversions carried out by the implicit cast. And it also performs the conversions between pointers of classes related to each other (upcast - > from derived to base class or downcast - > from base to derived class).

Syntax of the Static Cast

```
static_cast < new_data_type> (expression);
```

Program to demonstrate the use of the Static Cast

Let's create a simple example to use the static cast of the type casting in C++ programming.

```
 #include <iostream>
using namespace std;
int main ()
{
    // declare a variable
    double l;
    l = 2.5 * 3.5 * 4.5;
    int tot;

    cout << " Before using the static cast:" << endl;
    cout << " The value of l = " << l << endl;

    // use the static_cast to convert the data type
    tot = static_cast < int > (l);
    cout << " After using the static cast: " << endl;
```

```
    cout << " The value of tot = " << tot << endl;

    return 0;
}
```

Output:

```
Before using the static cast:
The value of l = 39.375
After using the static cast:
The value of tot = 39
```

Example2:-

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

class CBase { };
class CDerived: public CBase { };

int main ()
{

  CBase b;
  CBase* pb;
  CDerived d;
  CDerived* pd;

  pb = static_cast<CBase*>(&d);  // ok: derived to base class
  pd = static_cast<CDerived*>(&b);  // ok: base to derived class

    return 0;
}
```

Dynamic Cast

The dynamic_cast is a runtime cast operator used to perform conversion of one type variable to another only on class pointers and references. It means it checks the valid casting of the variables at the run time, and if the casting fails, it returns a NULL value. Dynamic casting is based on RTTI (Runtime Type Identification) mechanism.

dynamic_cast is always successful when we cast a class to one of its base classes:

Example:

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

class CBase { };

class CDerived: public CBase { };
int main ()
{

  CBase b;
  CBase* pb;
  CDerived d;
  CDerived* pd;

  pb = dynamic_cast<CBase*>(&d);     // ok: derived to base class
  pd = dynamic_cast<CDerived*>(&b);  // wrong: base to derived class
    return 0;
}
```

The second conversion in this piece of code would produce a compilation error since base-to-derived conversions are not allowed with dynamic_cast unless the base class is polymorphic.

When a class is polymorphic, dynamic_cast performs a special checking during runtime to ensure that the expression yields a valid complete object of the requested class:

Program to demonstrate the use of the Dynamic Cast in C++
Let's create a simple program to perform the dynamic_cast in the C++ programming language.

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

   class parent
   {
      public: virtual void print()
      {

      }
   };
   class derived: public parent
   {

   };

   int main ()
   {
        // create an object ptr
   parent *ptr = new derived;
   // use the dynamic cast to convert class data
   derived* d = dynamic_cast <derived*> (ptr);

   // check whether the dynamic cast is performed or not
   if ( d != NULL)
   {
      cout << " Dynamic casting is done successfully";
   }
   else
   {
      cout << " Dynamic casting is not done successfully";
   }
   }
```

Output:

```
Dynamic casting is done successfully.
```

Reinterpret Cast Type

The reinterpret_cast type casting is used to cast a pointer to any other type of pointer whether the given pointer belongs to each other or not. It means it does not check whether the pointer or the data pointed to by the pointer is the same or not. And it also cast a pointer to an integer type or vice versa.

Syntax of the reinterpret_cast type

```
reinterpret_cast <type> expression;
```

Example1:-

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

class CBase { };

class CDerived{ };
int main ()
{

  CBase b;
  CBase* pb;
  CDerived d;
  CDerived* pd;

  pb = reinterpret_cast <CBase*>(&d);     // ok: derived to base class
  pd = reinterpret_cast <CDerived*>(&b);  // wrong: base to derived class
    return 0;
}
```

Example2:-

Let's write a program to demonstrate the conversion of a pointer using the reinterpret in C++ language.

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

int main ()
{
   // declaration of the pointer variables
   int *pt = new int (65);

   // use reinterpre_cast operator to type cast the pointer variables
   char *ch = reinterpret_cast <char *> (pt);

   cout << " The value of pt: " << pt << endl;
   cout << " The value of ch: " << ch << endl;

   // get value of the defined variable using pointer
   cout << " The value of *ptr: " << *pt << endl;
   cout << " The value of *ch: " << *ch << endl;
   return 0;

}
```

Output:

```
The value of pt: 0x5cfed0
The value of ch: A
The value of *ptr: 65
The value of *ch: A
```

Const Cast

The const_cast is used to change or manipulate the const behavior of the source pointer. It means we can perform the const in two ways: setting a const pointer to a non-const pointer or deleting or removing the const from a const pointer.

Syntax of the Const Cast type

```
const_cast <type> exp;
```

Program to use the Const Cast in C++

Let's write a program to cast a source pointer to a non-cast pointer using the const_cast in C++.

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

// define a function
int disp(int *pt)
{
   return (*pt * 10);
}

int main ()
{
   // declare a const variable
   const int num = 50;
   const int *pt = &num;  // get the address of num

   // use const_cast to chnage the constness of the source pointer
   int *ptr = const_cast <int *> (pt);
   cout << " The value of ptr cast: " << disp(ptr);
   return 0;
}
```

Output:

```
The value of ptr cast: 500
```

.

Module-2

Exception Handling

An exception is a problem that arises during the execution of a program. A C++ exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.

Exceptions provide a way to transfer control from one part of a program to another. C++ exception handling is built upon three keywords: **try, catch,** and **throw**.

- **throw** − A program throws an exception when a problem shows up. This is done using a **throw** keyword.
- **catch** − A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The **catch** keyword indicates the catching of an exception.
- **try** − A **try** block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.

Assuming a block will raise an exception, a method catches an exception using a combination of the **try** and **catch** keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch as follows −

```
try {
// protected code
```

```
} catch( ExceptionName e1 ) {
// catch block
} catch( ExceptionName e2 ) {
// catch block
} catch( ExceptionName eN ) {
// catch block
}
```

You can list down multiple **catch** statements to catch different type of exceptions in case your **try** block raises more than one exception in different situations.

Throwing Exceptions

Exceptions can be thrown anywhere within a code block using **throw** statement. The operand of the throw statement determines a type for the exception and can be any expression and the type of the result of the expression determines the type of exception thrown.

Following is an example of throwing an exception when dividing by zero condition occurs −

```
double division(int a, int b) {
if( b = = 0 ) {
throw "Division by zero condition!";
}
return (a/b);
}
```

Catching Exceptions

The **catch** block following the **try** block catches any exception. You can specify what type of exception you want to catch and this is determined by the exception declaration that appears in parentheses following the keyword catch.

```
try {
// protected code
} catch( ExceptionName e ) {
// code to handle ExceptionName exception
}
```

Above code will catch an exception of **ExceptionName** type. If you want to specify that a catch block should handle any type of exception that is thrown in a try block, you must put an ellipsis, ..., between the parentheses enclosing the exception declaration as follows −

```
try {
// protected code
} catch(...) {
// code to handle any exception
}
```

1) The following is a simple example to show exception handling in C++. The output of the program explains the flow of execution of try/catch blocks.

```
#include <iostream>
using namespace std;
int main()
{
  int x = -1;

  // Some code
  cout << "Before try \n";
  try {
    cout << "Inside try \n";
    if (x < 0)
```

```
    {
      throw x;
      cout << "After throw (Never executed) \n";
    }
  }
  catch (int x ) {
    cout << "Exception Caught \n";
  }

  cout << "After catch (Will be executed) \n";
  return 0;
}
```

Output:

Before try
Inside try
Exception Caught
After catch (Will be executed)

2) There is a special catch block called the 'catch all' block, written as catch(…), that can be used to catch all types of exceptions. For example, in the following program, an,float is thrown as an exception, but there is no catch block for float, so the catch(…) block will be executed.

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

int main()
{
   try  {
     throw 10.5;
   }
   catch (char * excp)  {
      cout << "Caught character" << excp;
   }
  catch (int  x)  {
      cout << "Caught integer " << excp;
   }

   catch (...)  {
      cout << "Default Exception\n";
   }
   return 0;
}
```

Output:

Default Exception

3) Implicit type conversion doesn't happen for primitive types. For example, in the following program, 'a' is not implicitly converted to int.

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

int main()
{
   try  {
```

```
    throw 'a';
  }
  catch (int x)  {
     cout << "Caught " << x;
  }
  catch (...)  {
     cout << "Default Exception\n";
  }
  return 0;
}
```

Output:

Default Exception

4) In C++, try/catch blocks can be nested. Also, an exception can be re-thrown using "throw; ".

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

int main()
{
   try {
      try {
         throw 20;
      }
      catch (int n) {
         cout << "Handle Partially ";
         throw; // Re-throwing an exception
      }
   }
   catch (int n) {
      cout << "Handle remaining ";
   }
   return 0;
}
```

Output:

Handle Partially Handle remaining

A function can also re-throw a function using the same "throw; " syntax. A function can handle a part and ask the caller to handle the remaining.

5) When an exception is thrown, all objects created inside the enclosing try block are destroyed before the control is transferred to the catch block.

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

class Test {
public:
   Test() { cout << "Constructor of Test " << endl; }
   ~Test() { cout << "Destructor of Test " << endl; }
};

int main()
```

```
{
   try {
      Test t1;
      throw 10;
   }
   catch (int i) {
      cout << "Caught " << i << endl;
   }
}
```

Output:

Constructor of Test
Destructor of Test
Caught 10

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

int main() {
  try {
    int age = 15;
    if (age >= 18) {
      cout << "Access granted - you are old enough.";
    } else {
      throw 505;
    }
  }
  catch (int myNum) {
    cout << "Access denied - You must be at least 18 years old.\n";
    cout << "Error number: " << myNum;
  }
  return 0;
}
```

```
Access denied - You must be at least 18 years old.
Error number: 505
```

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

int main() {
  try {
    int age = 15;
    if (age >= 18) {
      cout << "Access granted - you are old enough.";
    } else {
      throw 505;
    }
  }
  catch (...) {
    cout << "Access denied - You must be at least 18 years old.\n";
  }
  return 0;
}
```

```
Access denied - You must be at least 18 years old.
```

The following is an example, which throws a division by zero exception and we catch it in catch block.

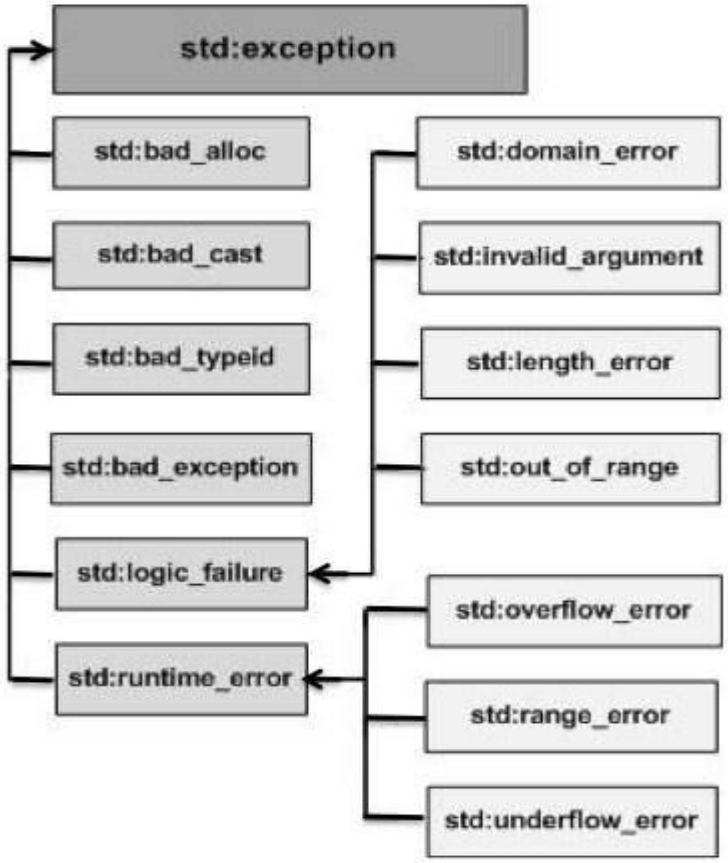
```
#include <iostream>
using namespace std;

int division(int a, int b) {
  if( b == 0 ) {
     throw "Division by zero condition!";
  }
  return (a/b);
}

int main () {
  int x = 50;
  int y = 0;
  double z = 0;

  try {
     z = division(x, y);
     cout << z << endl;
       }
catch (const char* msg) {
    cerr << msg << endl;
     }

  return 0;
}
```

Because we are raising an exception of type **const char***, so while catching this exception, we have to use const char* in catch block. If we compile and run above code, this would produce the following result −

Division by zero condition!

C++ Standard Exceptions

C++ provides a list of standard exceptions defined in **<exception>** which we can use in our programs. These are arranged in a parent-child class hierarchy shown below −

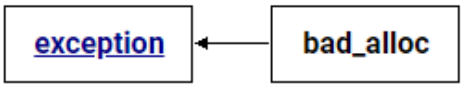


Here is the small description of each exception mentioned in the above hierarchy −

Sr.No	Exception & Description
1	**std::exception** An exception and parent class of all the standard C++ exceptions.
2	**std::bad_alloc** This can be thrown by **new**.
3	**std::bad_cast** This can be thrown by **dynamic_cast**.
4	**std::bad_exception** This is useful device to handle unexpected exceptions in a C++ program.
5	**std::bad_typeid** This can be thrown by **typeid**.
6	**std::logic_error** An exception that theoretically can be detected by reading the code.
7	**std::domain_error** This is an exception thrown when a mathematically invalid domain is used.
8	**std::invalid_argument** This is thrown due to invalid arguments.

9	**std::length_error** This is thrown when a too big std::string is created.
10	**std::out_of_range** This can be thrown by the 'at' method, for example a std::vector and std::bitset<>::operator[]().
11	**std::runtime_error** An exception that theoretically cannot be detected by reading the code.
12	**std::overflow_error** This is thrown if a mathematical overflow occurs.
13	**std::range_error** This is occurred when you try to store a value which is out of range.
14	**std::underflow_error** This is thrown if a mathematical underflow occurs.

bad_alloc exception



Standard C++ contains several built-in exception classes. The most commonly used is bad_alloc, which is thrown if an error occurs when attempting to allocate memory with new.

This class is derived from std::exception class.

Its member what returns a *null-terminated character sequence* identifying the exception. Here's a short example, that shows how it's used :

```
// CPP code for bad_alloc
#include <iostream>
#include<exception>
#include <new>

// Driver code
int main () {
  try
  {
    int* arr = new int[10000000000]; //increase number of zeros in not bad_alloc caught
    cout "allocation done successfully " <<endl
  }
  catch (bad_alloc & e)  //instead of bad_alloc we can use exception type also
  {
    cerr << "exception caught: " << e.what();
  }
  cout "remaining program……" <<endl

  return 0;
}
```

Output:
exception caught: std::bad_alloc

remaining program……

Out_of_range exception

This class defines the type of objects thrown as exceptions to report an out-of-range error.

It is a standard exception that can be thrown by programs. Some components of the standard library, such as vector, deque, string and bitset also throw exceptions of this type to signal arguments out of range.

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

int main() {
	vector<int> vec;
	vec.push_back(10);
	vec.push_back(20);
	// access the third element, that doesn't exist
	try
	{
		cout<<vec.at(5);
	}
	catch (out_of_range& ex) //instead of out_of_range we can use exception type also
	{
		cout << "Exception occurred!" <<ex.what();
	}
	return 0;
}
```

output

Exception occurred!vector::_M_range_check: __n (which is 5) >= this->size() (which is 2)

Define New Exceptions

You can define your own exceptions by inheriting and overriding **exception** class functionality. Following is the example, which shows how you can use std::exception class to implement your own exception in standard way −

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

class MyException : public exception {
  public:
  const char * what () const throw () {
    return "C++ Exception";
  }
};

int main() {
  try {
    MyException m;
    throw m;
  } catch(MyException& e) {
```

```
      cout << "MyException caught" << endl;
      cout << e.what() << endl;
   } catch(exception&   e) {
      //Other errors
   }
   return 0;
}
```

This would produce the following result −

```
MyException caught
C++ Exception
```

Here, **what()** is a public method provided by exception class and it has been overridden by all the child exception classes. This returns the cause of an exception.

Let's see the simple example of user-defined exception in which exception class is used to define the exception.

```
#include <iostream>
#include <exception>
using namespace std;
class MyException : public exception{
   public:
      const char * what() const throw()
      {
         return "Attempted to divide by zero!\n";
      }
};
int main()
{
   try
   {
      int x, y;
      cout << "Enter the two numbers : \n";
      cin >> x >> y;
      if (y == 0)
      {
         MyException z;
         throw z;
      }
      else
      {
         cout << "x / y = " << x/y << endl;
      }
   }
   catch(exception& e)
   {
      cout << e.what();
   }
}
```

Output:

```
Enter the two numbers :
10
2
x / y = 5
```

Output:

```
Enter the two numbers :
10
0
Attempted to divide by zero!
```

Note: In above example what() is a public method provided by the exception class. It is used to return the cause of an exception.

Exception specification-throw()

When declaring a function we can limit the exception type it might directly or indirectly throw by appending a throw suffix to the function declaration:

float myfunction (char param) throw (int);

This declares a function called myfunction which takes one argument of type char and returns an element of type float. The only exception that this function might throw is an exception of type int. If it throws an exception with a different type, either directly or indirectly, it cannot be caught by a regular int-type handler.

If this throw specifier is left empty with no type, this means the function is not allowed to throw exceptions. Functions with no throw specifier (regular functions) are allowed to throw exceptions with any type:

1. int myfunction (int param) throw(); // no exceptions allowed
2. int myfunction (int param);   // all exceptions allowed

This C++ material taken from the following websites:

- https://www.tutorialspoint.com/cplusplus
- https://www.w3schools.com/CPP
- https://www.geeksforgeeks.org/cpp-tutorial
- https://www.javatpoint.com/cpp-tutorial
- https://www.programiz.com/cpp-programming
- https://www.cplusplus.com