RTTI (Run-Time Type Information) in C++

In C++, **RTTI (Run-time type information)** is a mechanism that exposes information about an object's data type at runtime and is available only for the classes which have at least one virtual function. It allows the type of an object to be determined during program execution.

Runtime Casts

The runtime cast, which checks that the cast is valid, is the simplest approach to ascertain the runtime type of an object using a pointer or reference. This is especially beneficial when we need to cast a pointer from a base class to a derived type. When dealing with the inheritance hierarchy of classes, the casting of an object is usually required. There are two types of casting:

- **Upcasting:** When a pointer or a reference of a derived class object is treated as a base class pointer.
- **Downcasting:** When a base class pointer or reference is converted to a derived class pointer.

Using '*dynamic_cast*': In an inheritance hierarchy, it is used for downcasting a base class pointer to a child class. On successful casting, it returns a pointer of the converted type and, however, it fails if we try to cast an invalid type such as an object pointer that is not of the type of the desired subclass.

For example, dynamic_cast uses RTTI and the following program fails with the error "***cannot dynamic_cast `b' (of type `class B*') to type `class D*' (source type is not polymorphic)***" because there is no virtual function in the base class B.

```
// C++ program to demonstrate
// Run Time Type Identification(RTTI)
// but without virtual function

#include <iostream>
using namespace std;

// initialization of base class
class B {};

// initialization of derived class
class D : public B {};

// Driver Code
int main()
{
    B* b = new D; // Base class pointer
    D* d = dynamic_cast<D*>(b); // Derived class pointer
    if (d != NULL)
        cout << "works";
    else
        cout << "cannot cast B* to D*";
    return 0;
}
```

Adding a **virtual function** to the base class B makes it work.

```
// C++ program to demonstrate
// Run Time Type Identification successfully
// With virtual function

#include <iostream>
using namespace std;

// Initialization of base class
class B {
    virtual void fun() {}
};

// Initialization of Derived class
class D : public B {
};

// Driver Code
int main()
{
    B* b = new D; // Base class pointer
    D* d = dynamic_cast<D*>(b); // Derived class pointer
    if (d != NULL)
        cout << "works";
    else
        cout << "cannot cast B* to D*";
    getchar();
    return 0;
}
```

Output

```
works
```

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

class BaseClass
{
   virtual void f(){}

};
class Derived1 : public BaseClass
{

};
class Derived2 : public BaseClass
{

};

void w(BaseClass &obj)
{

      cout << "obj is pointing to an object of type "<< typeid(obj).name() << endl;
}

int main()
{
      BaseClass *p,baseObj;
      Derived1 ob1;
      Derived2 ob2,ob3;
      long double a;
      cout << "a is a type of " << typeid(a).name() << endl;
      p=&baseObj;
      cout << "p is pointing to an object of type "<< typeid(*p).name() << endl;
      p=&ob1;
      cout << "p is pointing to an object of type "<< typeid(*p).name() << endl;
      p=&ob2;
      cout << "p is pointing to an object of type "<< typeid(*p).name() << endl;

      w(baseObj);
      w(ob1);
      w(ob2);

      if(typeid(ob3) == typeid(ob2))
         cout << "same";
      else
         cout << "Different";
}
```

Namespaces in C++

Consider a situation, when we have two persons with the same name, Zara, in the same class. Whenever we need to differentiate them definitely we would have to use some additional information along with their name, like either the area, if they live in different area or their mother's or father's name, etc.

Same situation can arise in your C++ applications. For example, you might be writing some code that has a function called xyz() and there is another library available which is also having same function xyz(). Now the compiler has no way of knowing which version of xyz() function you are referring to within your code.

A **namespace** is designed to overcome this difficulty and is used as additional information to differentiate similar functions, classes, variables etc. with the same name available in different libraries. Using namespace, you can define the context in which names are defined. In essence, a namespace defines a scope.

Defining a Namespace

A namespace definition begins with the keyword **namespace** followed by the namespace name as follows −

```
namespace namespace_name {
// code declarations
}
```

To call the namespace-enabled version of either function or variable, prepend (::) the namespace name as follows −

```
name::code; // code could be variable or function.
```

Let us see how namespace scope the entities including variable and functions −

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

// first name space
namespace first_space {
  void func() {
    cout << "Inside first_space" << endl;
  }
}

// second name space
namespace second_space {
  void func() {
    cout << "Inside second_space" << endl;
  }
}

int main () {
  // Calls function from first name space.
  first_space::func();

  // Calls function from second name space.
  second_space::func();

  return 0;
}
```

If we compile and run above code, this would produce the following result –

```
Inside first_space
Inside second_space
```

The using directive

You can also avoid prepending of namespaces with the **using namespace** directive. This directive tells the compiler that the subsequent code is making use of names in the specified namespace. The namespace is thus implied for the following code −

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

// first name space
namespace first_space {
  void func() {
    cout << "Inside first_space" << endl;
  }
}

// second name space
namespace second_space {
  void func() {
    cout << "Inside second_space" << endl;
  }
}

using namespace first_space;
int main () {
  // This calls function from first name space.
  func();

  return 0;
}
```

If we compile and run above code, this would produce the following result −

```
Inside first_space
```

The 'using' directive can also be used to refer to a particular item within a namespace. For example, if the only part of the std namespace that you intend to use is cout, you can refer to it as follows −

```
using std::cout;
```

Exception Handling in C++

One of the advantages of C++ over C is Exception Handling. Exceptions are runtime anomalies or abnormal conditions that a program encounters during its execution. There are two types of exceptions: a)Synchronous, b)Asynchronous (i.e., exceptions which are beyond the program's control, such as disc failure, keyboard interrupts etc.). C++ provides the following specialized keywords for this purpose:
try: Represents a block of code that can throw an exception.
catch: Represents a block of code that is executed when a particular exception is thrown.
throw: Used to throw an exception. Also used to list the exceptions that a function throws but doesn't handle itself.

Why Exception Handling?

The following are the main advantages of exception handling over traditional error handling:
1) *Separation of Error Handling code from Normal Code:* In traditional error handling codes, there are always if-else conditions to handle errors. These conditions and the code to handle errors get mixed up with the normal flow. This makes the code less readable and maintainable. With try/catch blocks, the code for error handling becomes separate from the normal flow.
2) *Functions/Methods can handle only the exceptions they choose:* A function can throw many exceptions, but may choose to handle some of them. The other exceptions, which are thrown but not caught, can be handled by the caller. If the caller chooses not to catch them, then the exceptions are handled by the caller of the caller.
In C++, a function can specify the exceptions that it throws using the throw keyword. The caller of this function must handle the exception in some way (either by specifying it again or catching it).
3) *Grouping of Error Types:* In C++, both basic types and objects can be thrown as exceptions. We can create a hierarchy of exception objects, group exceptions in namespaces or classes and categorize them according to their types.

C++ Exceptions:

When executing C++ code, different errors can occur: coding errors made by the programmer, errors due to wrong input, or other unforeseeable things.

When an error occurs, C++ will normally stop and generate an error message. The technical term for this is: C++ will throw an exception (error).

C++ try and catch:

Exception handling in C++ consists of three keywords: try, throw and catch:

The try statement allows you to define a block of code to be tested for errors while it is being executed.

The throw keyword throws an exception when a problem is detected, which lets us create a custom error.

The catch statement allows you to define a block of code to be executed if an error occurs in the try block.

The try and catch keywords come in pairs:

We use the try block to test some code: If the value of a variable "age" is less than 18, we will throw an exception, and handle it in our catch block.

In the catch block, we catch the error if it occurs and do something about it. The catch statement takes a single parameter. So, if the value of age is 15 and that's why we are throwing an exception of type int in the try block (age), we can pass "int myNum" as the parameter to the catch statement, where the variable "myNum" is used to output the value of age.

If no error occurs (e.g. if age is 20 instead of 15, meaning it will be greater than 18), the catch block is skipped.

Exception Handling in C++
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 int is thrown as an exception, but there is no catch block for int, so the catch(...) block will be executed.

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

int main()
{
    try  {
       throw 10;
    }
    catch (char *excp)  {
        cout << "Caught " << 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) If an exception is thrown and not caught anywhere, the program terminates abnormally. For example, in the following program, a char is thrown, but there is no catch block to catch the char.

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

int main()
{
    try  {
       throw 'a';
    }
    catch (int x)  {
        cout << "Caught ";
    }
    return 0;
}
```

Output:

```
terminate called after throwing an instance of 'char'
```

```
This application has requested the Runtime to terminate it in an
unusual way. Please contact the application's support team for
more information.
```

We can change this abnormal termination behavior by writing our own unexpected function.

5) A derived class exception should be caught before a base class exception. See this for more details.

6) Like Java, the C++ library has a standard exception class which is the base class for all standard exceptions. All objects thrown by the components of the standard library are derived from this class. Therefore, all standard exceptions can be caught by catching this type

7) Unlike Java, in C++, all exceptions are unchecked, i.e., the compiler doesn't check whether an exception is caught or not (See this for details). So, it is not necessary to specify all uncaught exceptions in a function declaration. Although it's a recommended practice to do so. For example, the following program compiles fine, but ideally the signature of fun() should list the unchecked exceptions.

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

// This function signature is fine by the compiler, but not recommended.
// Ideally, the function should specify all uncaught exceptions and function
// signature should be "void fun(int *ptr, int x) throw (int *, int)"
void fun(int *ptr, int x)
{
    if (ptr == NULL)
        throw ptr;
    if (x == 0)
        throw x;
    /* Some functionality */
}

int main()
{
    try {
       fun(NULL, 0);
    }
    catch(...) {
        cout << "Caught exception from fun()";
    }
    return 0;
}
```

Output:

```
Caught exception from fun()
```

A better way to write the above code:

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

// Here we specify the exceptions that this function
// throws.
void fun(int *ptr, int x) throw (int *, int) // Dynamic Exception specification
{
    if (ptr == NULL)
        throw ptr;
    if (x == 0)
        throw x;
    /* Some functionality */
}

int main()
{
```

```
    try {
       fun(NULL, 0);
    }
    catch(...) {
        cout << "Caught exception from fun()";
    }
    return 0;
}
```

Note : The use of Dynamic Exception Specification has been deprecated since C++11. One of the reasons for it may be that it can randomly abort your program. This can happen when you throw an exception of another type which is not mentioned in the dynamic exception specification. Your program will abort itself because in that scenario, it calls (indirectly) terminate(), which by default calls abort().

Output:

```
Caught exception from fun()
```

8) 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.

9) 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
```

Ref:

https://www.geeksforgeeks.org/rtti-run-time-type-information-in-cpp/

https://www.tutorialspoint.com/cplusplus/cpp_namespaces.htm#

https://www.geeksforgeeks.org/exception-handling-c/