

MODULE 22 TYPECASTING

My Training Period: hours

Abilities

- Understand the type casting.
- Understand and use `static_cast`, `const_cast`, `dynamic_cast` and `reinterpret_cast`.
- Understand and use the `explicit` keyword.

22.1 C Typecasting

- Typecasting is used to convert the type of a variable, function, object, expression or return value to another type.
- Throughout this tutorial you have encountered many codes that use simple C-style type cast.
- One of the said advantageous of C++ is the type safe feature. During the compile or run time there are type checking process that not available in C. This can avoid a lot of program bugs and unexpected logical errors.
- In C an expression, `expression`, of type `type`, can be cast to another type by using the following syntax:

```
(type) expression OR  
  
//look like a function :o) isn't it?  
type (expression)
```

- For example:

```
int p;  
double dou;  
  
//same as p = int (dou);  
p = (int) dou;
```

- The previous example used the **explicit type conversion** that is done by programmers. Integral type promotion and demotion (automatic type casting, as explained in Module 2); is the **implicit type conversion**.
- What ever it is, explicit type conversion should be adopted for good programming habits such as for troubleshooting and readability.
- The weaknesses in C type cast are listed below:
 - The syntax is same for every casting operation from simple variables to objects and classes. For complex type casting, we as well as compiler don't know the intended purpose of the casting and this will create ambiguity.
 - When we do the debugging, it is very difficult to locate the related cast problems, although by using the tools provided by the compiler, because there are many codes that use parentheses.
 - It allows us to cast practically any type to any other type. This can create many program bugs. If the program compiled and run successfully, the result still can contain logical errors.
- The four type casting operators in C++ with their main usage is listed in the following table:

Type caster keyword	Description
<code>static_cast</code>	To convert non polymorphic types.
<code>const_cast</code>	To add or remove the <code>const</code> -ness or <code>volatile</code> -ness type.
<code>dynamic_cast</code>	To convert polymorphic types.
<code>reinterpret_cast</code>	For type conversion of unrelated types.

Table 22.1: Type caster

- The syntax is same for the four type cast except the cast name:

```
name_cast<new_type> (expression)
```

- Where:

<code>name_cast</code>	either one of the <code>static</code> , <code>const</code> , <code>dynamic</code> or <code>reinterpret</code>
<code>new_type</code>	The result type of the cast.
<code>expression</code>	Expression to be cast

22.2 static_cast

- It allows casting a pointer of a derived class to its base class and vice versa. This cast type uses information available at compile time to perform the required type conversion.
- The syntax is:

```
name_cast<new_type> (expression)
```

- If `new_type` is a reference type, the result is an lvalue; otherwise, the result is an rvalue
- Explicitly can be used to perform conversion defined in classes as well as performing standard conversion between basic data types, for example:

```
int p;
double dou;

p = static_cast<int> (dou);
```

- Program example:

```
#include <iostream.h>
#include <stdlib.h>

int main()
{
    int sum = 1000;
    int count = 21;

    double averagel = sum/count;
    cout<<"Before conversion = "<<averagel<<endl;

    double average2 = static_cast<double>(sum)/count;

    cout<<"After conversion = "<<average2<<endl;
    system("pause");
    return 0;
}
```

Output :

```
C:\bc5\bin\hohoh.exe
Before conversion = 47
After conversion = 47.619
Press any key to continue . . .
```

- Other usage of the `static_cast` includes the conversion of `int` to `enum`, reference of type `P&` to `Q&`, an object of type `P` to an object of type `Q` and a pointer to member to another pointer to member within the same class hierarchy.
- You also can convert any expression to `void` using `static_cast`, which the value of the expression is discarded.
- `static_cast` cannot be used to convert the `const`-ness and `volatile`-ness (`cv` qualification), use `const_cast` instead and polymorphic types.
- An integral type to enumeration conversion can be done using `static_cast`. The conversion results in an enumeration with the same value as the integral type provided the integral type value is within the range of the enumeration. The value that is not within the range should be undefined.
- Keep in mind that, `static_cast` is not as safe as `dynamic_cast`, because it does not have the run time check, for example, for ambiguous pointer, `static_cast` may return successful but a `dynamic_cast` pointer will fail.
- Program example:

```

#include <iostream.h>
#include <stdlib.h>

//enum data type
enum color {blue, yellow, red, green, magenta};

int main()
{
    int p1 = 3;

    cout<<"integer type, p1 = "<<p1<<endl;
    cout<<"color c1 = static_cast<color> (p1)"<<endl;
    color c1 = static_cast<color> (p1);
    cout<<"enum type, c1 = "<<c1<<endl;

    system("pause");
    return 0;
}

```

Output:

```

C:\bc5\bin\hohoh.exe
integer type, p1 = 3
color c1 = static_cast<color> (p1)
enum type, c1 = 3
Press any key to continue . . .

```

22.3 const_cast

- This cast type is used to add to or remove the const-ness or volatile-ness of the expression.
- The syntax is:

```
const_cast<new_type> (expression)
```

- `new_type` and `expression` must be of the same type except for `const` and `volatile` modifiers. Casting is resolved at compile time and the result is of type `new_type`.
- A pointer to `const` can be converted to a pointer to non-`const` that is in all other respects an identical type. If successful, the resulting pointer refers to the original object.
- A `const` object or a reference to `const` cast results in a non-`const` object or reference that is otherwise an identical type.
- The `const_cast` operator performs similar typecasts on the `volatile` modifier. A pointer to `volatile` object can be cast to a pointer to non-`volatile` object without otherwise changing the type of the object. The result is a pointer to the original object. A `volatile`-type object or a reference to `volatile`-type can be converted into an identical non-`volatile` type.
- Simple integral program example of removing the const-ness:

```

//demonstrates const_cast
#include <iostream.h>
#include <stdlib.h>

int main()
{
    //p = 10 is a constant value, cannot be modified
    const int p = 20;

    cout<<"const p = "<<p<<"\nq = p + 20 = "<<(p + 20)<<endl;
    //The following code should generate error, because
    //we try to modify the constant value...
    //uncomment, recompile and re run, notice the error...
    //p = 15;
    //p++;

    //remove the const...
    int r = const_cast<int&> (p);
    //the value of 10 should be modified now...
    --r;
    cout<<"Removing the const, decrement by 1,\nNew value = "<<r<<endl;
    system("pause");
}

```

Output:

- Another simple program example:

```
//Demonstrate const_cast
#include <iostream.h>
#include <stdlib.h>

struct One
{
    //test function...
    void funct1()
    { cout<<"Testing..."<<endl;}
};

//const argument, cannot be modified...
void funct2(const One& c)
{
    //will generate warning/error...
    c.funct1();
}

int main()
{
    One b;

    funct2(b);
    system("pause");
    return 0;
}
```

- We have to remove the const of the argument. Change `c.funct1();` to the following statements recompile and rerun the program.

```
//remove the const...
One &noconst = const_cast<One&> (c);
cout<<"The reference = "<<&noconst<<endl;
noconst.funct1();
```

Output:

- Another program example.

```
//Demonstrates type casting
#include <iostream.h>
#include <stdlib.h>

double funct1(double& f)
{
    //do some work here...
    f++;
    cout<<"f = "<<f<<endl;
    //return the incremented value...
    return f;
}
```

```

//const argument, can't be modified...
void funct2(const double& d)
{
    cout<<"d = "<<d<<endl;
    //remove const...
    //use the non-const argument, making function call...
    double value = funct1(const_cast<double&> (d));
    //display the returned value...
    cout<<"value = "<<value<<endl;
}

int main()
{
    double c = 4.324;

    //first function call...
    funct2(c);
    system("pause");
    return 0;
}

```

Output:

- volatile and const removal program example:

```

//Demonstrate type casting
#include <iostream.h>
#include <stdlib.h>

class One
{
    public:
    void funct()
    {cout<<"Testing..."<<endl;};
};

//const and volatile...
const volatile int* Test1;
//const...
const int* Test2;

void TestConstVol()
{
    One Test3;

    //remove const...
    const_cast<One&>(Test3).funct();
    //remove const and volatile...
    const_cast<int*> (Test1);
}

int main()
{
    TestConstVol();
    system("pause");
    return 0;
}

```

Output:

- Removing the const this pointer program example

```
//removing the const-ness of the
//this pointer
#include <iostream.h>
#include <stdlib.h>

class Test
{
public:
void GetNumber(int);
//Read only function...
void DisplayNumber() const;

private:
int Number;
};

void Test::GetNumber(int Num)
{Number = Num;}

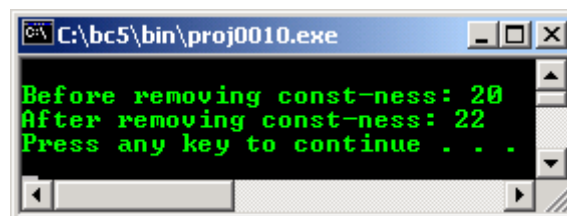
void Test::DisplayNumber() const
{
    cout<<"\nBefore removing const-ness: "<<Number;
    const_cast<Test*>(this)->Number+=2;
    cout<<"\nAfter removing const-ness: "<<Number<<endl;
}

int main()
{
    Test p;

    p.GetNumber(20);
    p.DisplayNumber();

    system("pause");
    return 0;
}
```

Output :



- This function const-ness removal also can be achieved by using the mutable specifier.
- Program example using mutable keyword to modify the const function member variable.

```
//using mutable to remove the
//const-ness of the function...
#include <iostream.h>
#include <stdlib.h>

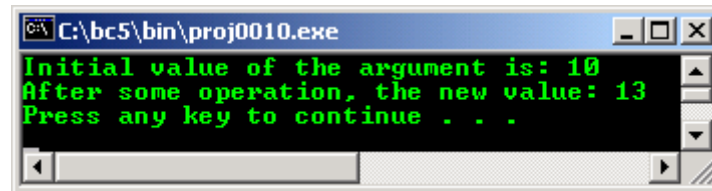
class Test
{
    //using mutable
    mutable int count;
    mutable const int* ptr;
public:
    //Read only function can't
    //change const arguments.
    int funct(int num = 10) const
    {
        //should be valid expression...
        count = num+=3;
        ptr = &num;
        cout<<"After some operation, the new value: "<<*ptr<<endl;
        return count;
    }
};
```

```

int main(void)
{
    Test var;
    cout<<"Initial value of the argument is: 10"<<endl;
    var.funct(10);
    system("pause");
    return 0;
}

```

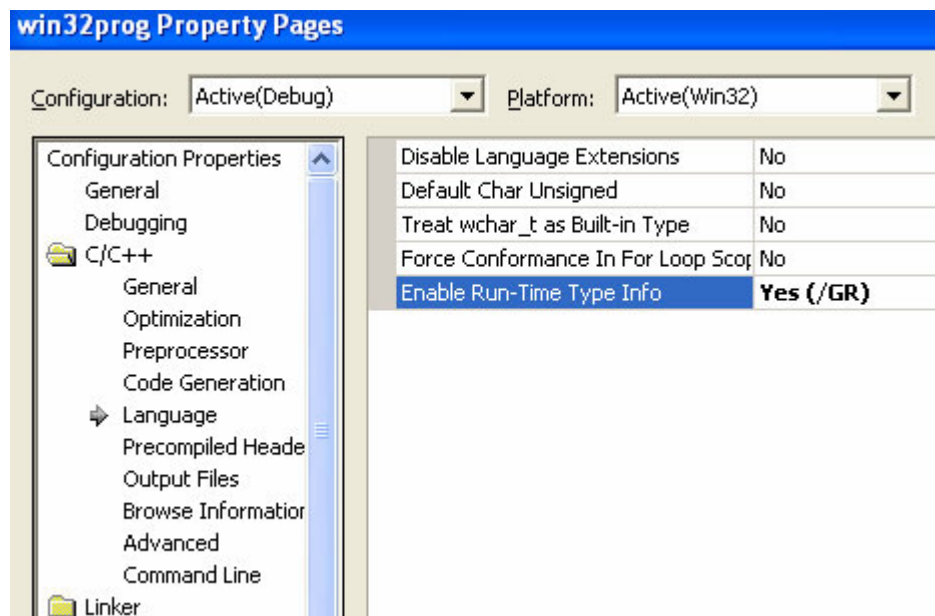
Output:



22.4 dynamic_cast

Note:

For this part, you must enable the Run-Time Type Information (RTTI) setting of your compiler :o). For Visual C++ .Net: **Project** menu → **your_project_name Properties...** → **C / C++** folder → **Language** setting.



- This cast is exclusively used with **pointers** and **references** to objects for class hierarchy navigation.
- The syntax:

```
dynamic_cast<new_type> (expression)
```

- That means converts the operand **expression** to an object of type, **new_type**. The **new_type** must be a pointer or a reference to previously defined class type or a pointer to void. The type of **expression** must be a pointer if **new_type** is a pointer or *lvalue* if **new_type** is a reference.
- It can be used to cast from a derived class pointer to a base class pointer (upcasting), cast a derived class pointer to another derived (sibling) class pointer (crosscast) or cast a base class pointer to a derived class pointer (downcast).
- Differing from other cast, **dynamic_cast** operator is part of the C++ run time type information (**rtti**) tally to the term dynamic instead of static, hence it usage closely related to the polymorphic classes, classes which have at least one virtual function.
- As you have learned, for non-polymorphic class, use the **static_cast**.
- The validity or safety of the type casting is checked during the run time, if the pointer being cast is not a pointer to a valid **complete object** of the requested type, the value returned is a NULL pointer.

- It is safe if the object being pointed to is of type derived class. The actual object is said to be the complete object. The pointer to the base class is said to point to a sub-object of the complete object.
- The following diagram is the simple class hierarchy. There are base and derived classes. Derived class is the class that inherits the base class(s) member variable(s) and function(s) with restrictions implemented using `public`, `private` or `protected` keywords.

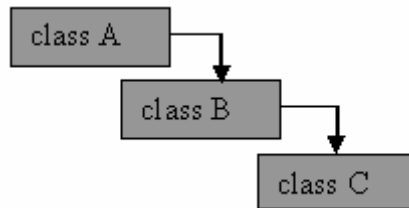


Figure 22.1: Simple class hierarchy

- An object of class C could be depicted as the following diagram. For class C instance, there is a B and A sub-objects. The instance of class C, including the A and B sub-objects, is the complete object.

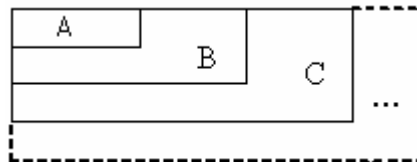


Figure 22.2: Class C with sub-objects B and A

- Type conversion from base class pointer to a derived class pointer is called **downcast**.
- Type conversion from derived class pointer to a base class pointer, is called **upcast**.
- Another one is **crosscast**, a cast from a class to a sibling class in class hierarchy or sibling class. Two classes are siblings if a class is directly or indirectly derived from both of their base classes and one is not derived from the other. It is a multi inheritance class hierarchy.
- Let do some experiment through program examples starting from the upcasting.

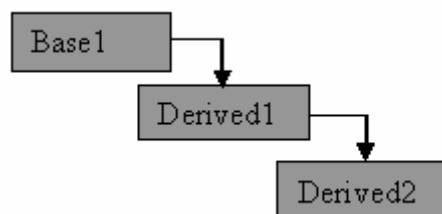


Figure 22.3: Upcasting, from Derived2 to Derived1/Base1

```

//upcast conversion using dynamic_cast
#include <iostream.h>
#include <stdlib.h>

//base class
class Base1 {};

//derived class...
class Derived1:public Base1 {};

//another derived class
class Derived2:public Derived1{};

//dynamic_cast test function...
void funct1()
{
    //instantiate an object...
    Derived2* Test1 = new Derived2;

    //upcasting, from derived class to base class,
    //Derived1 is a direct from Base1
    //making Test2 pointing to Derived1 sub-object of Test1
}
  
```



```

Derived1* Test2 = dynamic_cast<Derived1*>(Test1);
cout<<"Derived1* Test2 = dynamic_cast<Derived1*>(Test1);"<<endl;
if(!Test2)
    cout<<"The conversion is fail..."<<endl;
else
    cout<<"The conversion is successful..."<<endl;

//upcasting, from derived class to base class
//Derived2 is an indirect from Base1
Base1* Test3 = dynamic_cast<Derived1*>(Test1);
cout<<"\nBase1* Test3 = dynamic_cast<Derived1*>(Test1);"<<endl;
if(!Test3)
    cout<<"The conversion is fail..."<<endl;
else
    cout<<"The conversion is successful..."<<endl;
}

int main()
{
    funct1();
    system("pause");
    return 0;
}

```

Output:

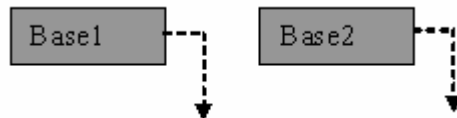


Figure 22.4: void* type, from base to base class

- void* type conversion program example.

```

//If new_name is void*, the result of
//conversion is a pointer to the complete
//object pointed to by the expression

//void* and dynamic_cast
#include <iostream.h>
#include <stdlib.h>

//base class
class Base1
{
    public:
    virtual void funct1(){};
};

//another base class...
class Base2
{
    public:
    virtual void funct2(){};
};

//dynamic_cast test function...
void funct3()
{
    //instantiate objects...
    Base1 * Test1 = new Base1;
    Base2 * Test2 = new Base2;

    //making Test3 pointing to an object of type Base1
    void* Test3 = dynamic_cast<void*>(Test1);
}

```

```

cout<<"void* Test3 = dynamic_cast<void*>(Test1);"<<endl;
if(!Test3)
    cout<<"The conversion is fail..."<<endl;
else
    cout<<"The conversion is successful..."<<endl;

//making Test3 pointing to an object of type Base2
Test3 = dynamic_cast<void*>(Test2);
cout<<"\nTest3 = dynamic_cast<void*>(Test2);"<<endl;
if(!Test3)
    cout<<"The conversion is fail..."<<endl;
else
    cout<<"The conversion is successful..."<<endl;
}

int main()
{
    funct3();
    system("pause");
    return 0;
}

```

Output:

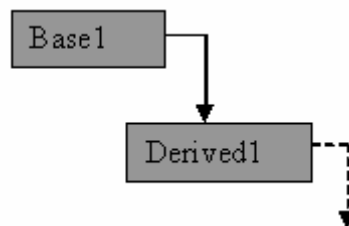


Figure 22.5: Downcast, from Base1 to Derived1 class

```

//downcast conversion using dynamic_cast
#include <iostream.h>
#include <stdlib.h>

//base class
class Base1 {
public:
    virtual void funct1(){};
};

//derived class...
class Derived1:public Base1 {
public:
    virtual void funct2(){};
};

//dynamic_cast test function...
void funct3()
{
    //instantiate objects...
    Base1* Test1 = new Derived1;
    Base1* Test2 = new Base1;

    //making Test1 pointing to Derived1
    Derived1* Test3 = dynamic_cast<Derived1*>(Test1);
    cout<<"Derived1* Test3 = dynamic_cast<Derived1*>(Test1);"<<endl;
    if(!Test3)
        cout<<"The conversion is fail..."<<endl;
    else
        cout<<"The conversion is successful..."<<endl;
}

```

```

//should fails coz Test2 pointing
//to Base1 not Derived1, Test4 == NULL
Derived1* Test4 = dynamic_cast<Derived1*>(Test2);
cout<<"\nDerived1* Test4 = dynamic_cast<Derived1*>(Test2);"<<endl;
if(!Test4)
    cout<<"The conversion is fail..."<<endl;
else
    cout<<"The conversion is successful..."<<endl;
//reconfirm, should be NULL pointer...
cout<<"Should be NULL pointer =  "<<Test4<<endl;
}

int main()
{
    funct3();
    system("pause");
    return 0;
}

```

Output:

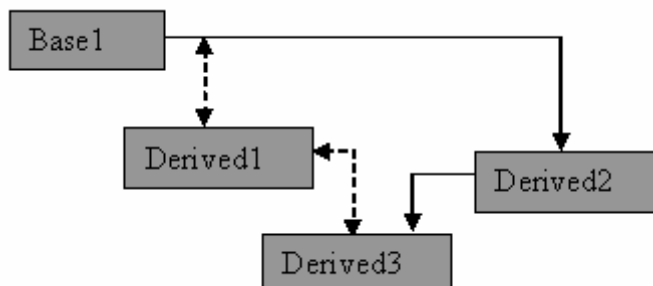


Figure 22.6: Multiple conversion, from Derived3 to Base1

```

//multiple inheritance
//conversion using dynamic_cast
#include <iostream.h>
#include <stdlib.h>

//base class
class Base1 {};

class Derived1:public Base1{};

class Derived2:public Base1{};

//derived class...
class Derived3:public Derived1, public Derived2
{
    public:
    virtual void funct1(){}
};

//dynamic_cast test function...
void funct2()
{
    //instantiate an object...
    Derived3 *Test1 = new Derived3;

    //-----start comment out-----
    //may fail, ambiguous...from Derived3 direct
    //conversion to Base1...
    //if you use good compiler, please comment out this
}

```

```

//part, there should be run time error:-)
Base1* Test2 = dynamic_cast<Base1*>(Test1);
cout<<"Base1* Test2 = dynamic_cast<Base1*>(Test1);"<<endl;
if(!Test2)
    cout<<"The conversion is fail..."<<endl;
else
    cout<<"The conversion is successful..."<<endl;
//reconfirm the pointer
cout<<"The pointer should be NULL ==> " <<Test2<<endl;
//-----end comment out-----

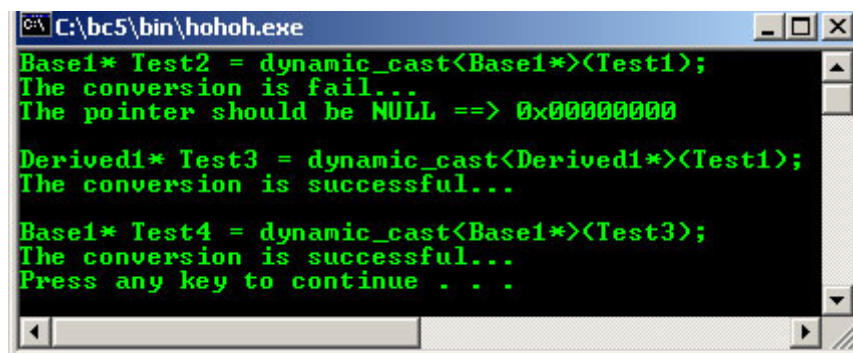
//solution, traverse, recast...
//firstly, cast to Derived1
Derived1* Test3 = dynamic_cast<Derived1*>(Test1);
cout<<"\nDerived1* Test3 = dynamic_cast<Derived1*>(Test1);"<<endl;
if(!Test3)
    cout<<"The conversion is fail..."<<endl;
else
    cout<<"The conversion is successful..."<<endl;

//then cast to base1...
Base1* Test4 = dynamic_cast<Base1*>(Test3);
cout<<"\nBase1* Test4 = dynamic_cast<Base1*>(Test3);"<<endl;
if(!Test4)
    cout<<"The conversion is fail..."<<endl;
else
    cout<<"The conversion is successful..."<<endl;
}

int main()
{
    funct2();
    system("pause");
    return 0;
}

```

Output:



```

C:\bc5\bin\hohoh.exe
Base1* Test2 = dynamic_cast<Base1*>(Test1);
The conversion is fail...
The pointer should be NULL ==> 0x00000000

Derived1* Test3 = dynamic_cast<Derived1*>(Test1);
The conversion is successful...

Base1* Test4 = dynamic_cast<Base1*>(Test3);
The conversion is successful...
Press any key to continue . . .

```

- Let try the crosscast program example.

Note:

The next two program examples will generate warning and runtime error if you use a very 'good' compiler :o). The unreliable type conversions have been protected by the compiler during runtime.

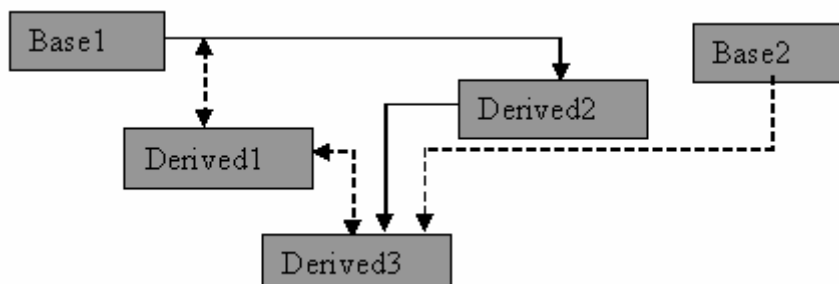


Figure 22.7: Crosscast, from Base2 to Derived1

```

//testing the crosscast: downcast, upcast and crosscast
//conversion using dynamic_cast
#include <iostream.h>

```

```

#include <stdlib.h>

//base class
class Base1
{
    public:
    virtual void funct1(){};
};

class Derived1:public Base1
{
    public:
    virtual void funct2(){};
};

class Derived2:public Base1{
    public:
    virtual void funct3(){};
};

//derived class...
class Base2
{
    public:
    virtual void funct4(){};
};

class Derived3:public Derived1,public Derived2,public Base2
{};

//dynamic_cast test function...
void funct5()
{
    //instantiate an object
    //Test1 of type Base2...
    //or test1 of type Derived2...
    //you can choose either one:-)

    Base2* Test1 = new Base2;
    //Derived2* Test1 = new Derived2;

    //start with downcast, type Base2/Derived2 to Derived3...
    Derived3* Test2 = dynamic_cast<Derived3*>(Test1);
    cout<<"Firstly, Derived3* Test2 = dynamic_cast<Derived3*>(Test1);"<<endl;
    if(!Test2)
    {
        cout<<"The conversion is fail lor!"<<endl;
        cout<<"Checking the pointer = "<<Test2<<endl;
    }
    else
        cout<<"The conversion is successful..."<<endl;

    //Upcast, type derived3 to type derived1...
    Derived1* Test3 = dynamic_cast<Derived1*>(Test2);
    cout<<"\nThen, Derived1* Test3 = dynamic_cast<Derived1*>(Test2);"<<endl;
    if(!Test3)
    {
        cout<<"The conversion is fail lor!"<<endl;
        cout<<"Checking the pointer = "<<Test3<<endl;
    }
    else
        cout<<"The conversion is successful..."<<endl;

    //crosscast, direct, type Base2/Derived2 to Derived1...
    Derived1* Test4 = dynamic_cast<Derived1*>(Test1);
    cout<<"\nThen, Derived1* Test4 = dynamic_cast<Derived1*>(Test1);"<<endl;
    if(!Test4)
    {
        cout<<"The conversion is fail lor!"<<endl;
        cout<<"Checking the pointer = "<<Test3<<endl;
    }
    else
        cout<<"The conversion is successful..."<<endl;
    delete Test1;
}

int main()
{

```

```

    funct5();
    system("pause");
    return 0;
}

```

Output:

```

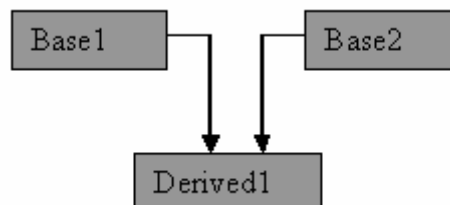
C:\Program Files\Microsoft Visual Studio\MyProjects\temple\Debug\...
Firstly, Derived3* Test2 = dynamic_cast<Derived3*>(Test1);
The conversion is fail lor!
Checking the pointer = 0x00000000

Then, Derived1* Test3 = dynamic_cast<Derived1*>(Test2);
The conversion is fail lor!
Checking the pointer = 0x00000000

Then, Derived1* Test4 = dynamic_cast<Derived1*>(Test1);
The conversion is fail lor!
Checking the pointer = 0x00000000
Press any key to continue . . .

```

- Another tough program example.



```

//dynamic_cast ambiguous conversion experiment :o)
#include <iostream.h>
#include <stdlib.h>

//a class with virtual function...
//polymorphic...
class Base1
{
    public:
    virtual void FuncBase1()
    {};
};

//another class with virtual function...
class Base2
{
    public:
    virtual void FuncBase2()
    {};
};

//derived class from Base1 and Base2 classes
//public virtual and private...
class Derived1:public virtual Base1, private Base2
{};

//dynamic_cast test function...
void DynamicCastSample()
{
    //instantiate an object of type Derived1 class...
    Derived1 DerivedObj;

    //simple assignment, derived to base class, upcasting...
    //cast needed to break private protection...
    Base2* Base2Obj = (Base2*) &DerivedObj;

    //another assignment, derived to base class, upcasting...
    //public inheritance, no need casting..
    Base1* Base1Obj = &DerivedObj;

    //base class to derived class, downcast

```

```

Derived1& Derived1Obj = dynamic_cast<Derived1&>(*Base2Obj);
if(!&Derived1Obj)
    cout<<"Conversion is failed!...."<<endl;
else
    cout<<"Conversion is OK...."<<endl;
cout<<"The address.."<<&Derived1Obj<<endl;

//base class to derived class, downcast
Base1Obj = dynamic_cast<Base1*>(Base2Obj);
if(!Base1Obj)
    cout<<"Conversion is failed!...."<<endl;
else
    cout<<"Conversion is OK...."<<endl;
cout<<"The address.."<<Base1Obj<<endl;

//base class to base class, ????
//no inheritance...
Base2Obj = dynamic_cast<Base2*>(Base1Obj);
if(!Base2Obj)
    cout<<"Conversion is failed!...."<<endl;
else
    cout<<"Conversion is OK...."<<endl;
cout<<"The address.."<<Base2Obj<<endl;

//derived class to base class, upcast
Base1Obj = dynamic_cast<Base1*>(&Derived1Obj);
if(!Base1Obj)
    cout<<"Conversion is failed!...."<<endl;
else
    cout<<"Conversion is OK...."<<endl;
cout<<"The address.."<<Base1Obj<<endl;

//derived class to base class...
//Derived1Obj is derived from non-virtual, private Base2...
Base2Obj = dynamic_cast<Base2*>(&Derived1Obj);
if(!Base2Obj)
    cout<<"Conversion is failed!...."<<endl;
else
    cout<<"Conversion is OK...."<<endl;
cout<<"The address.."<<Base2Obj<<endl;
}

int main()
{
    int *ptr = NULL;
    int var;

    cout<<"Benchmarking..."<<endl;
    cout<<"Address of var = "<<&var<<endl;
    //NULL pointer
    cout<<"NULL *ptr = "<<ptr<<endl;
    cout<<endl;

    //call the function for dynamic_cast testing...
    DynamicCastSample();
    system("pause");
    return 0;
}

```

Output:

```

C:\bc5\bin\proj0010.exe
Benchmarking...
Address of var = 0x11390ffa
NULL *ptr = 0x00000000

Conversion is OK....
The address..0x11390fe4
Conversion is OK....
The address..0x11390fea
Conversion is failed!....
The address..0x00000000
Conversion is OK....
The address..0x11390fea
Conversion is failed!....
The address..0x00000000
Press any key to continue . . .

```

- Well, tired playing with type casting huh?

22.5 rtti

- Run time type information/identification (RTTI) is a mechanism which the type of an object can be determined during the program execution where the type of the object cannot be determined by the static information.
- It can be applied on the pointers and references. RTTI elements consists of:

RTTI element	Brief description
<code>dynamic_cast</code>	Polymorphic types conversion.
<code>typeid()</code> operator	Used to identify the exact type of an object.
<code>type_info</code> class	Used for holding the type information returned by the <code>typeid</code> operator.

Table 22.2: RTTI elements

- The `typeid` operator syntax:

```

typeid( expression )
typeid( type_name )

```

- You can use `typeid` to get run-time identification of `type_name` and `expressions`. A call to `typeid` returns a reference to an object of type `const type_info&`. The returned object represents the type of the `typeid` operand.
- If the `typeid` operand is a dereferenced pointer or a reference to a polymorphic type (class with virtual functions), `typeid` returns the dynamic type of the actual object pointed or referred to in the expression. If the operand is non-polymorphic, `typeid` returns an object that represents the static type. `typeid` operator can be used with fundamental data types as well as user-defined types.
- If the `typeid` operand is a dereferenced NULL pointer, the `bad_typeid` exception handler is thrown.
- Program example, don't forget to include the `typeinfo.h` header file.

```

//using typeid operator, type_info::before()
//and type_info::name() member functions
#include <iostream.h>
#include <stdlib.h>
#include <typeinfo.h>

//T - True, F - False
#define T 1
#define F 0

//a base class
class A { };
//a derived class
class B : A { };

int main()
{
char c;

```



```

float f;

//using typeid operator, == for comparison
if (typeid(c) == typeid(f))
    cout<<"c and f are the same type."<<endl;
else
    cout<<"c and f are different type."<<endl;

//using true and false comparison...
//name() and before() are typeid member functions...
cout<<typeid(int).name();
cout<<" before " <<typeid(double).name()<<": " <<
(typeid(int).before(typeid(double)) ? T:F)<<endl;

cout<<typeid(double).name();
cout<<" before " <<typeid(int).name()<<": " <<
(typeid(double).before(typeid(int)) ? T:F)<<endl;

cout<<typeid(A).name();
cout<<" before " <<typeid(B).name()<<": " <<
(typeid(A).before(typeid(B)) ? T:F)<<endl;
system("pause");
return 0;
}

```

Output:

- Another program example:

```

//getting the run time type information...
#include <iostream.h>
#include <stdlib.h>
#include <typeinfo.h>

//polymorphic base class...
class __rtti Test
{
    //This makes Test a polymorphic class type.
    virtual void func() {};
};

//derived class...
class Derived : public Test {};

int main(void)
{
    //Instantiate Derived type object...
    Derived DerivedObj;
    //Declare a Derived type pointer
    Derived *DerivedPtr;
    //Initialize the pointer
    DerivedPtr = &DerivedObj;

    //do the run time checking...
    if(typeid(*DerivedPtr) == typeid(Derived))
        //check the type of *DerivedPtr
        cout<<"Ptr *DerivedPtr type name is " <<typeid(*DerivedPtr).name();
    if(typeid(*DerivedPtr) != typeid(Test))
        cout<<"\nPointer DerivedPtr is not a Test class type.\n";
    system("pause");
    return 0;
}

```

Output:

```

C:\bc5\bin\proj0010.exe
Ptr *DerivedPtr type name is Derived
Pointer DerivedPtr is not a Test class type.
Press any key to continue . . .

```

- If the expression is dereferencing a NULL pointer, typeid() will throw a bad_typeid exception handler. If the expression is neither a pointer nor a reference to a base class of the object, the result is a type_info reference representing the static type of the expression.
- Another program example.

```

//run time type information...
#include <iostream.h>
#include <stdlib.h>
#include <typeinfo.h>

class Base
{
public:
    virtual void funct(){}
};

class Derived:public Base{};

int main()
{
    Derived* Test1 = new Derived;
    Base* Test2 = Test1;

    cout<<"The type name of Test1 is: ";
    cout<<typeid(Test1).name()<<endl;
    cout<<"The type name of *Test1 is: ";
    cout<<typeid(*Test1).name()<<endl;
    cout<<"The type name of Test2 is: ";
    cout<<typeid(Test2).name()<<endl;
    cout<<"The type name of *Test2 is: ";
    cout<<typeid(*Test2).name()<<endl;

    delete Test1;
    system("pause");
    return 0;
}

```

Output:

```

C:\bc5\bin\proj0010.exe
The type name of Test1 is: Derived *
The type name of *Test1 is: Derived
The type name of Test2 is: Base *
The type name of *Test2 is: Derived
Press any key to continue . . .

```

22.6 reinterpret_cast

- This operator is used to convert any pointer to any other pointer type. It also can be used to convert any integral type to any pointer type and vice versa.
- Because of the unrelated or 'random' type conversion can be done using reinterpret_cast, it can be easily unsafe if used improperly and it is non portable. It should only be used when absolutely necessary.
- It cannot be used for const-ness and volatile-ness conversion.
- Can be used to convert for example, int* to char*, or classA to classB, which both class are unrelated classes, between two unrelated pointers, pointers to members or pointers to functions.
- For null pointer, it converts a null pointer value to the null pointer value of the destination type.
- Program example. If you change the for loop from -10 to 0, the conversion values still same, may need to use 2's complement.

```

//using reinterpret_cast, int to

```

```

//unsigned int pointers conversion
#include <iostream.h>
#include <stdlib.h>

unsigned int* Test(int *q)
{
    //convert int pointer to unsigned int pointer
    unsigned int* code = reinterpret_cast<unsigned int*>(q);
    //return the converted type data, a pointer...
    return code;
}

int main(void)
{
    //array name is a pointer...
    int a[10];

    cout<<"int pointer          unsigned int pointer"<<endl;
    for(int i = 0;i<=10;i++)
        cout<<(a+i)<<" converted to "<<Test(a+i)<<endl;
    system("pause");
    return 0;
}

```

Output:

```

C:\bc5\bin\proj0010.exe
int pointer          unsigned int pointer
0x0012ff64 converted to 0x0012ff64
0x0012ff68 converted to 0x0012ff68
0x0012ff6c converted to 0x0012ff6c
0x0012ff70 converted to 0x0012ff70
0x0012ff74 converted to 0x0012ff74
0x0012ff78 converted to 0x0012ff78
0x0012ff7c converted to 0x0012ff7c
0x0012ff80 converted to 0x0012ff80
0x0012ff84 converted to 0x0012ff84
0x0012ff88 converted to 0x0012ff88
0x0012ff8c converted to 0x0012ff8c
Press any key to continue . . .

```

2.2.7 explicit Keyword

- Keyword **explicit** used to avoid a single argument constructor from defining an automatic type conversion.
- A typical **explicit** usage example is in a collection class in which you can pass the initial size as constructor argument. For example, you could declare a constructor that has an argument for the initial size of a stack as shown below:

```

//simple class
//compiled using visual C++ .Net
#include <iostream>
using namespace std;

class MyStack
{
public:
    //create a stack with initial size
    MyStack(int initsize);
    ~MyStack(void);
};

MyStack::MyStack(int initsize)
{
    static x;
    cout<<"Constructor: Pass #"<<x<<endl;
    x++;
}

MyStack::~MyStack(void)
{
    static y;
    cout<<"Destructor: Pass #"<<y<<endl;
}

```

```

        y++;
    }

//----main program----
int main()
{
    //The initial stack size is 10
    MyStack p(20);

    //but, there will be new stack objects
    //with size of 30!
    p = 30;
    cout<<"Without the explicit keyword!\n";
    return 0;
}

```

Output:

```

e:\projectvc\string\string\D...
Constructor: Pass #0
Constructor: Pass #1
Destructor: Pass #0
Without the explicit keyword!
Destructor: Pass #1
Press any key to continue

```

- Here, without explicit keyword the constructor would define an automatic type conversion from int type to MyStack object type.
- From the program output also, it is clear that the constructor was invoked two times, once for MyStack with size of 20 and another one with size 30. This is not our intention.
- Then we could assign an integer, 30 to MyStack wrongfully, as shown below:

```
p = 30;
```

- The automatic type conversion would convert the integer 30 to MyStack, with 30 elements (size) and then assign it to p.
- By declaring the int constructor as an explicit, the assignment `p = 30;` will result an error at compile time. The following is the program example using explicit keyword.

```

//simple class
//compiled using visual C++ .Net
#include <iostream>
using namespace std;

class MyStack
{
public:
    //create a stack with initial size
    explicit MyStack(int initsize);
    ~MyStack(void);
};

MyStack::MyStack(int initsize)
{
    static x;
    cout<<"Constructor: Pass #"<<x<<endl;
    x++;
}

MyStack::~MyStack(void)
{
    static y;
    cout<<"Destructor: Pass #"<<y<<endl;
    y++;
}

//----main program----
int main()
{
    //The initial stack size is 10
    MyStack p(20);
    //but, there will be new stack objects

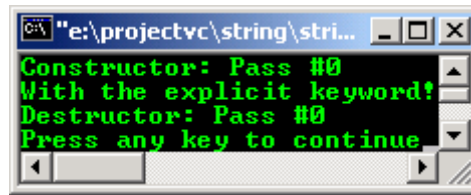
```

```

//with size of 30!
//p = 30;
cout<<"With the explicit keyword!\n";
return 0;
}

```

Output :



- You can try un-commenting the `p = 30` code, then recompile and re run the program. It should generate an error.
- Note that `explicit` also rules out the initialization with type conversion by using the assignment syntax as shown below:

```

MyStack p1(30); //OK
MyStack p2 = 30; //error

```

- The previous program example based on the template of the STL. More information is in [Module 24](#) above.
- Program example compiled using `VC++ / VC++ .Net`.

```

//run time type information...
//compiled using VC++/VC++ .Net
#include <iostream>
#include <typeinfo.h>
using namespace std;

class Base
{
public:
virtual void funct(){}
};

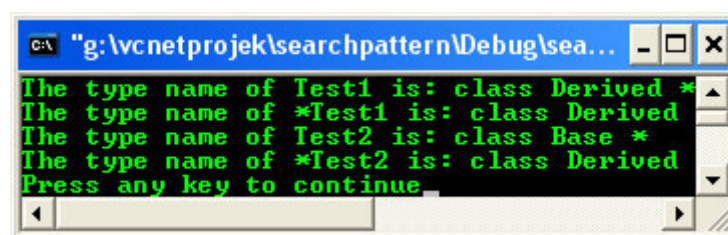
class Derived:public Base{};

int main()
{
Derived* Test1 = new Derived;
Base* Test2 = Test1;

cout<<"The type name of Test1 is: ";
cout<<typeid(Test1).name()<<endl;
cout<<"The type name of *Test1 is: ";
cout<<typeid(*Test1).name()<<endl;
cout<<"The type name of Test2 is: ";
cout<<typeid(Test2).name()<<endl;
cout<<"The type name of *Test2 is: ";
cout<<typeid(*Test2).name()<<endl;
delete Test1;
return 0;
}

```

Output :



- Program example compiled using `g++`.

```

//*****-typecast.cpp-*****
//upcast conversion using dynamic_cast
#include <iostream>
using namespace std;

//base class
class Base1 {};

//derived class...
class Derived1:public Base1 {};

//another derived class
class Derived2:public Derived1{};

//dynamic_cast test function...
void funct1()
{
    //instantiate an object.
    Derived2* Test1 = new Derived2;

    //upcasting, from derived class to base class,
    //Derived1 is a direct from Base1
    //making Test2 pointing to Derived1 sub-object of Test1
    Derived1* Test2 = dynamic_cast<Derived1*>(Test1);
    cout<<"Derived1* Test2 = dynamic_cast<Derived1*>(Test1);"<<endl;
    if(!Test2)
        cout<<"The conversion is fail..."<<endl;
    else
        cout<<"The conversion is successful..."<<endl;

    //upcasting, from derived class to base class
    //Derived2 is an indirect from Base1
    Base1* Test3 = dynamic_cast<Derived1*>(Test1);
    cout<<"\nBase1* Test3 = dynamic_cast<Derived1*>(Test1);"<<endl;
    if(!Test3)
        cout<<"The conversion is fail..."<<endl;
    else
        cout<<"The conversion is successful..."<<endl;
}

int main()
{
    funct1();
    return 0;
}

```

```

[bodo@bakawali ~]$ g++ typecast.cpp -o typecast
[bodo@bakawali ~]$ ./typecast

```

```

Derived1* Test2 = dynamic_cast<Derived1*>(Test1);
The conversion is successful...

```

```

Base1* Test3 = dynamic_cast<Derived1*>(Test1);
The conversion is successful...

```

-----0o0-----

Further reading and digging:

1. Check the best selling C/C++, Object Oriented and pattern analysis books at [Amazon.com](https://www.amazon.com).