UNIT IV - TEMPLATES AND EXCEPTION HANDLING

- **Topics to be discussed,**
  - Function Template and Class Template
  - Namespaces
  - **Casting**
  - Exception Handling

Casting

- Casting is a conversion process wherein data can be changed from one type to another.
- C++ has two types of conversions:
  - **Implicit conversion:** Conversions are performed automatically by the compiler without the programmer's intervention.
  - **Example:**

```
int iVariable = 10;
float fVariable = iVariable; //Assigning an int to a float will trigger a conversion.
```

  - **Explicit conversion:** Conversions are performed only when explicitly specified by the programmer.
  - **Example:**

```
int iVariable = 20;
float fVariable = (float) iVariable / 10;
```

Casting – Cont'd

- The functionality of these explicit conversion operators is enough for most needs with fundamental data types.
- However, these operators can be applied indiscriminately on classes and pointers to classes, which can lead to code that while being syntactically correct can cause runtime errors.

```
// class type-casting
#include <iostream>
using namespace std;
class floatClass
{
   float i,j;
public:
   floatClass(float x,float y)
   {
      i=x;
      j=y;
   }
};
```

```
class intClass
{
   int x,y;
   public:
   intClass (int a, int b)
    {
      x=a;
      y=b;
    }
   int result()
   {
      return x+y;
   }
};
int main ()
{
  floatClass f(1.2,4.5);
  intClass  * pi;
  pi = (intClass*) &f;
  cout << pi->result();
  return 0;
}
```

Output:

```
-2144757350
```

- Traditional explicit type-casting allows to convert any pointer into any other pointer type, independently of the types they point to.
- The subsequent call to member result will produce either a run-time error or a unexpected result.

Casting – Cont'd

- **In order to control these types of conversions between classes, we have four specific casting operators:**
  - dynamic_cast
  - reinterpret_cast
  - static_cast
  - const_cast.
- Their format is to follow the new type enclosed between angle-brackets (<>) and immediately after, the expression to be converted between parentheses.
  - dynamic_cast <new_type> (expression)
  - reinterpret_cast <new_type> (expression)
  - static_cast <new_type> (expression)
  - const_cast <new_type> (expression)
- The traditional type-casting equivalents to these expressions would be:
  - (new_type) expression
  - new_type (expression)

Casting – Cont'd

- **Syntax of the traditional explicit type casting:**

  (type) expression;

- For example, we have a floating pointing number 4.534, and to convert an integer value, then we write 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.

Casting – Cont'd

static_cast

- The static_cast operator is the most commonly used casting operator in C++.
- It performs compile-time type conversion and is mainly used for explicit conversions that are considered safe by the compiler.
- **Syntax :**

  static_cast <dest_type> (source);

  – The return value of static_cast will be of dest_type.
- The static_cast can be used to convert between related types, such as numeric types or pointers in the same inheritance hierarchy.

static_cast Example 1

```
#include <iostream>
using namespace std;
int main()
{
    float f = 3.5;
    cout << "\nThe Value of f: " << f;
     int i1 = f; // Implicit type cast
    cout << "\nThe Value of i1: " << i1;
     // using static_cast for float to int
    int i2 = static_cast<int>(f);
    cout << "\nThe Value of i2: " << i2;
}
```

Output:

```
The Value of f: 3.5
The Value of i1: 3
The Value of i2: 3
```

static_cast Example 2

```
#include <iostream>
#include <typeinfo>
using namespace std;
int main()
{
   int num = 10;
   // converting int to double
   double numDouble = static_cast<double>(num);

   // printing data type
   cout << typeid(num).name() << endl;

   // typecasting
   cout << typeid(static_cast<double>(num)).name() << endl;

   // printing double type t
   cout << typeid(numDouble).name() << endl;

   return 0;
}
```

Output:

```
i
d
d
```

typeid operator in C++

- It is used where the dynamic type or runtime type information of an object is needed.
- It is included in the <typeinfo> library.
- The typeid expression is an lvalue expression.
- **Syntax:**

  typeid(type);

  OR

  typeid(expression);
- **Parameters:** typeid operator accepts a parameter, based on the syntax used in the program:
  - ***type:*** This parameter is passed when the runtime type information of a variable or an object is needed. In this, there is no evaluation that needs to be done inside type and simply the type information is to be known.
  - ***expression:*** This parameter is passed when the runtime type information of an expression is needed. In this, the expression is first evaluated. Then the type information of the final result is then provided.

typeid operator - Example

```
#include <iostream>
#include <typeinfo>
using namespace std;
int main()
{
   int i1, i2;
   char c;
    // Get the type info using typeid operator
   const type_info& ti1 = typeid(i1);
   const type_info& ti2 = typeid(i2);
   const type_info& ti3 = typeid(c);
   if (ti1 == ti2)  // Check if both types are same
      cout << "i1 and i2 are of"
         << " similar type" << endl;
   else
      cout << "i1 and i2 are of"
         << " different type" << endl;
   if (ti2 == ti3)   // Check if both types are same
      cout << "i2 and c are of"
         << " similar type" << endl;
   else
      cout << "i2 and c are of"
         << " different type" << endl;
    return 0;
}
```

Output:

```
i1 and i2 are of similar type
i2 and c are of different type
```

static_cast Example 3

```
int main()
{
  int i = 25;
  char ch = 'a';
  int* iptr = (int*)&ch;
 cout<<*iptr;
   return 0;
}
```

Output:

```
1644040033
```

(Unexpected result)

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5    int i = 25;
6    char ch = 'a';
7    int* iptr = static_cast<int*>(&ch);
8    cout<<*iptr;
9    return 0;
10 }
```

d messages | CppCheck/Vera++ | CppCheck/Vera++ messages | Cscope

	Line	Message
		=== Build file: "no target" in "no project" (compiler: unkn
\CS320...		In function 'int main()':
\CS320...	7	error: invalid static_cast from type 'char*' to type 'int*'
		=== Build failed: 1 error(s), 0 warning(s) (0 minute(s), 0

static_cast for Inheritance in C++

```
#include <iostream>
using namespace std;
class Base
{
   public:
    void show()
   {
      cout<<"\nBase class show method";
   }
};
class Derived :public Base
{
   public:
   void show()
   {
      cout<<"\nDerived class show method";
   }
};
int main()
{
 Derived dobj;
 // Implicit cast allowed
 Base* bptr = (Base *)&dobj; //*bptr=&dobj;
 bptr->show();
 return 0;
}
```

Output:

```
Base class show method
```

```
#include <iostream>
using namespace std;
class Base
{
   public:
   void show()
   {
      cout<<"\nBase class show method";
   }
};
class Derived :public Base
{
   public:
   void show()
   {
      cout<<"\nDerived class show method";
   }
};
int main()
{
 Derived dobj;
 // upcasting using static_cast
 Base* bptr = static_cast<Base*>(&dobj);
 bptr->show();
 return 0;
}
```

Output:

```
Base class show method
```

In the previous example, we inherited the base class as public. What happens when we inherit it as private?

```
#include <iostream>
using namespace std;
class Base
{
   public:
    void show()
   {
      cout<<"\nBase class show method";
   }
};
class Derived :private Base
{
   public:
   void show()
   {
      cout<<"\nDerived class show method";
   }
};
int main()
{
 Derived dobj;
 // Implicit cast allowed
 Base* bptr = (Base *)&dobj; //*bptr=&dobj;
 bptr->show();
 return 0;
}
```

Output:

```
Base class show method
```

```
1  #include <iostream>
2  using namespace std;
3  class Base
4  {
5     public:
6     void show()
7     {
8        cout<<"\nBase class show method";
9     }
10 };
11 class Derived :private Base
12 {
13    public:
14    void show()
15    {
16       cout<<"\nDerived class show method";
17    }
18 };
19 int main()
20 {
21    Derived dobj;
22    // upcasting using static_cast
23    Base* bptr = static_cast<Base*>(&dobj);
24    bptr->show();
25    return 0;
26 }
```

```
                    === Build file: "no target" in "no project" (compiler:
F:\2024\CS320...    In function 'int main()':
F:\2024\CS320...  23  error: 'Base' is an inaccessible base of 'Derived'
                    === Build failed: 1 error(s), 0 warning(s) (0 minute(s),
```

Casting – Cont'd
dynamic_cast

- The dynamic_cast operator is mainly used to perform downcasting (converting a pointer / reference of a base class to a derived class).
- It ensures type safety by performing a runtime check to verify the validity of the conversion.
- **Syntax :**

  dynamic_cast <new_type> (expression);

  – If the conversion is not possible, dynamic_cast returns a null pointer (for pointer conversions) or throws a bad_cast exception (for reference conversions).

dynamic_cast – Example

```
#include <iostream>
using namespace std;
class Base
{
public:
  virtual void show()
  {
    cout << "Base class Show method" << endl;
  }
};
class Derived1 : public Base
{
public:
  virtual void show()
  {
    cout << "Derived1 class Show method" << endl;
  }
};
class Derived2 : public Base
{
public:
  virtual void show()
  {
    cout << "Derived2 class Show method" << endl;
  }
};
int main()
{
  Base* bptr = new Derived1();// base class pointer to derived class object
  // downcasting
  Derived1* d1ptr = dynamic_cast<Derived1*>(bptr);
  if (d1ptr) // checking if the typecasting is successful
  {
    d1ptr->show();
  }
  else
  {
    cout << "Failed to cast to Derived1" << endl;
  }
  // typecasting to other derived class
  Derived2* d2ptr = dynamic_cast<Derived2*>(bptr);
  if (d2ptr) // checking if the typecasting is successful
  {
    d2ptr->show();
  }
  else
  {
    cout << "Failed to cast to Derived2" << endl;
  }
  delete bptr;
  return 0;
}
```

Output:

```
Derived1 class Show method
Failed to cast to Derived2
```

- In this example, the first line of output is printed because the 'bptr' of the 'Base' type is successfully cast to the 'Derived1' type and show() function of the Derived1 class is invoked but the casting of the 'Base' type to 'Derived2' type is failed because 'bptr' points to a 'Derived1' object thus, the dynamic cast fails because the typecasting is not safe.

Casting – Cont'd

const_cast

- The const_cast operator is **used to modify the const or volatile qualifier of a variable**.
- It allows programmers to temporarily remove the constancy of an object and make modifications.
- Caution must be exercised when using const_cast, as modifying a const object can lead to undefined behavior.
- **Syntax :**

  const_cast <new_type> (expression);

const_cast - Example

```
#include <iostream>
using namespace std;
int main()
{
   const int num = 50;
   const int* cptr = &num;
   cout<<"\nold value is "<<*cptr<<"\n";
   int* ptr = const_cast<int*>(cptr);
   *ptr = 100;
   cout<<"new value is "<<*cptr;
   return 0;
}
```

Output:

```
old value is 50
new value is 100
```

Casting – Cont'd
reinterpret_cast

- The reinterpret_cast operator is used to **convert the pointer to any other type of pointer.**
- It does not perform any check whether the pointer converted is of the same type or not.
- **Syntax:**

  reinterpret_cast <new_type> (expression);
- Even if they are unrelated or incompatible, it enables us to convert a pointer of one type to a pointer of a different type.
- Because it might result in undeclared behaviour and system crashes if used carelessly, the reinterpret_cast operator is sometimes regarded as the most hazardous of the C++ type-casting operators.

reinterpret_cast - Example

```
#include <iostream>
using namespace std;
int main()
{
   int intData = 25;
   int* intPtr = &intData;
   // Attempting to reinterpret_cast int pointer to a float pointer
   float* floatPtr=reinterpret_cast<float*>(intPtr);
   // Accessing the original integer data through the float pointer
   int retrievedData = *reinterpret_cast<int*>(floatPtr);
   cout<< "Original Integer Data: " << intData<<endl;
   cout<< "Float Pointer Value: " <<*floatPtr << endl;
   cout<< "Retrieved Integer Data: " <<retrievedData << endl;
   cout << "Integer Address: " << intPtr << endl;
   cout << "Float Address: " << reinterpret_cast<void*>(floatPtr) << endl;
}
```

Output:

```
Original Integer Data: 25
Float Pointer Value: 3.50325e-44
Retrieved Integer Data: 25
Integer Address: 0x61fe08
Float Address: 0x61fe08
```

Casting – Cont'd

- ***Note:*** *const_cast and reinterpret_cast are generally not recommended as they vulnerable to different kinds of errors.*