Objective:

- To look into the technical and theoretical aspects of the Polymorphism specifically virtual inheritance and definition of a pure virtual function.
- Detail discussion on typecast operators in C++.

Some more points to Ponder ☺:

1. Pure Virtual functions can also have definition but out of line.

2. A destructor can also be pure virtual and we have to give its definition out of line otherwise all the derive classes will be abstract.
   Important point over here is that if the base class defines a pure virtual dtor then all the derive classes have to define their dtors.

```
class base
{
public:
        virtual void f()=0;
        virtual ~base()=0;
};
void base::f()                  // always out of line
{
}
void base::~base()              // always out of line
{
}
class derive : public base
{
public:
        virtual void f()
        {
        }
        virtual ~derive()
        {
        }
};
void main()
{
        derive d;
        d.f();
        d.base::~base();   //Now there will be
        d.base::f();         //no linker error
}
```

3. Object Slicing

There is a distinct difference between passing the addresses of objects and passing objects by value when using polymorphism. All the examples you've seen, and virtually all the examples you should see, pass addresses and not values. This is because addresses all have the same size, so passing the address of an object of a derived type (which is usually a bigger object) is the same as passing the address of an object of the base type (which is usually a smaller object). As explained before, this is the goal when using polymorphism – code that manipulates a base type can transparently manipulate derived-type objects as well.

If you upcast to an object instead of a pointer or reference, something will happen that may surprise you: the object is "sliced" until all that remains is the subobject that corresponds to the destination type of your cast. In the following example you can see what happens when an object is sliced:

```
class base
{
public:
        int a;
        float b;
        base()
        {
                a=1;
                b=2.2;
        }
        base(base &ref)
        {
                a=ref.a;
                b=ref.b;
        }
        void operator = (base &ref)
        {
                a=ref.a;
                b=ref.b;
        }
        void show()
        {
                cout<<a<<endl<<b;
        }
};
class derive : public base
{
public:
        int e
        derive()
        {
                e=5;
                }
        derive(derive &ref):base(ref)
        {
                e=ref.e;
        }
        void operator = (derive &ref)
        {
                base::operator=(ref);
                e=ref.e;
        }
        void show()
        {
                cout<<e;
        }
}

/*
```

The function copy takes derive object by reference, since base does not know anything about derive so while copying (copy constructor and assignment operator) only base part of object d will be copied to base object b. This is commonly referred to as object slicing and can be a source of surprises and errors.

One reason to pass pointers and references to objects of classes in a hierarchy is to avoid slicing.
Other reasons are to preserve polymorphic behavior and to gain efficiency.
*/

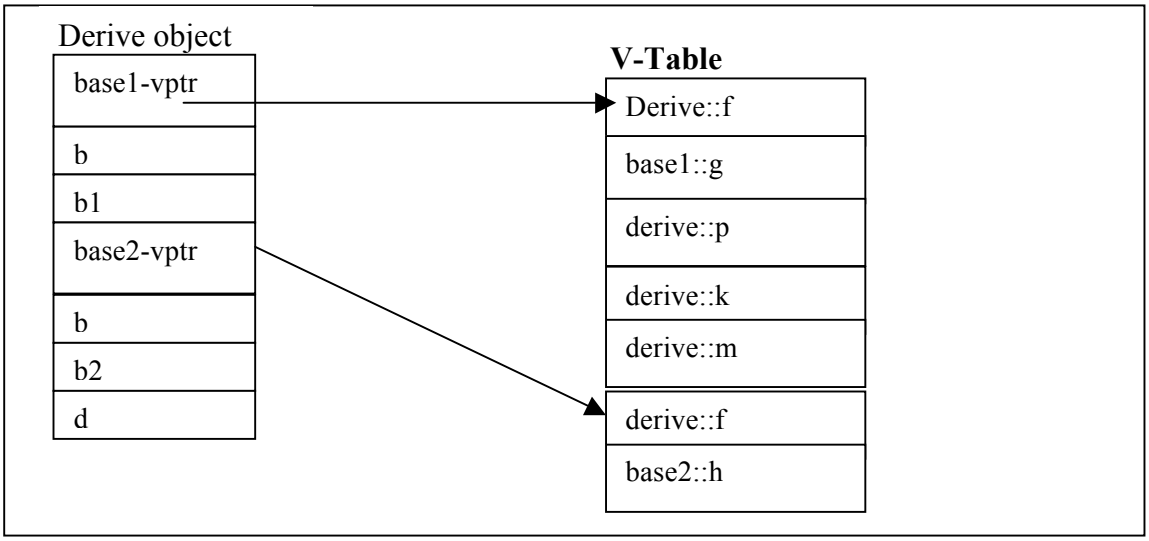
```
void copy(derive &d) //no object slicing
{
    base b=d;  //object slicing
    b=d;       //object slicing
}
void main()
{
clrscr();
    base b;
    derive d;
    fun(d);

getch();
}
```

4. DIAMOND INHERITANCE PROBLEM

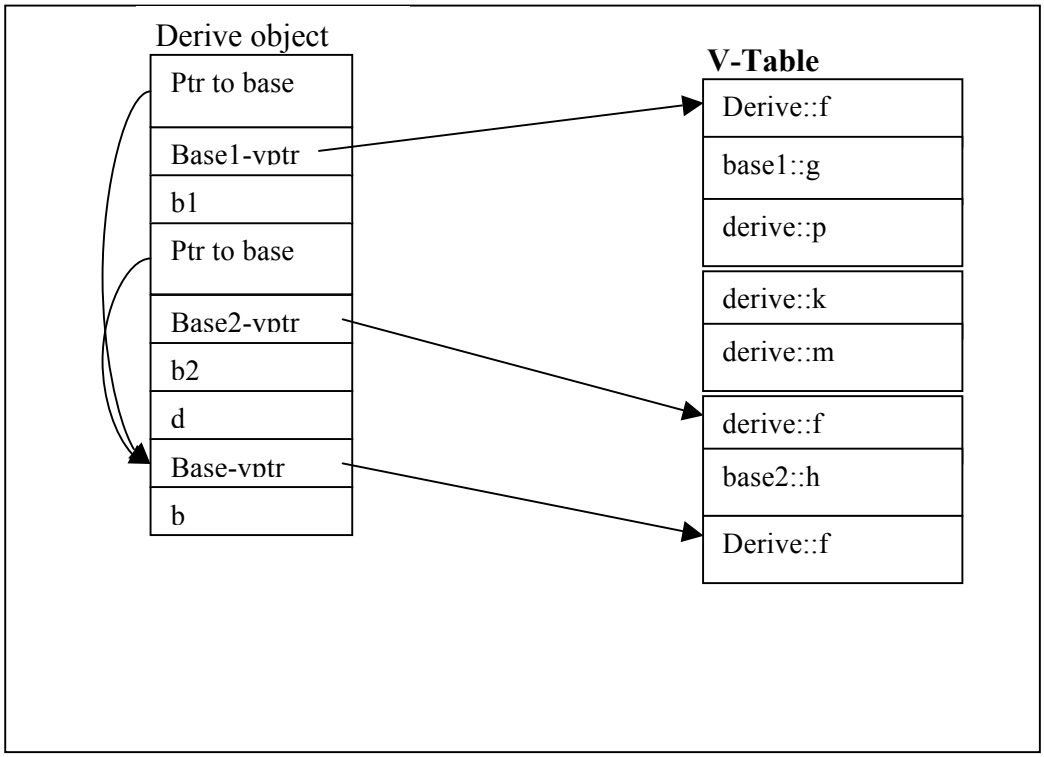
```
class base
{
public:
        int b;
        virtual void f() {}
};
class base1 : public base
{
public:
        int b1;
        virtual void f() {}
        virtual void g() {}
};
class base2 : public base
{
public:
        int b2;
        virtual void f() {}
        virtual void h() {}
};

 class derive:public base1, public base2
 {
 public:
        int d;
        virtual void f() {}
        virtual void p() {}
        virtual void k() {}
        virtual void m() {}
};
```



SOLUTION TO DIAMOND INHERITANCE PROBLEM

The class "base" object is duplicated in class "derive" object So to remove the duplication we inherit class "base" virtually result in following shape of memory.



Constructor Calling in Virtual Inheritance:

```
class base
{};
class base2
{};
class level1 : public base2, virtual public base
{};
class level2 : public base2, virtual public base
{};
class toplevel : public level1, virtual public level2
```

```
{};

void main()
{
        toplevel view;
}
```

The construction order of view would be as follows:

```
base();      // virtual base class highest in hierarchy

           // base is constructed only once
base2();     // nonvirtual base of virtual base level2
           // must be called to construct level2
level2();    // virtual base class
base2();     // nonvirtual base of level1
level1();    // other nonvirtual base

toplevel();
```

→ If a class hierarchy contains multiple instances of a virtual base class, that base class is constructed only once**. If, however, there exist both virtual and nonvirtual instances of the base class, the class constructor is invoked a single time for all virtual instances and then once for each nonvirtual occurrence of the base class.**

Type conversions [Taken From: http://www.cplusplus.com/doc/tutorial/typecasting/]

Implicit conversion

Implicit conversions are automatically performed when a value is copied to a compatible type. For example:

```
1  short a=2000;
2  int b;
3  b=a;
```

Here, the value of `a` is promoted from `short` to `int` without the need of any explicit operator. This is known as a *standard conversion*. Standard conversions affect fundamental data types, and allow the conversions between numerical types (`short` to `int`, `int` to `float`, `double` to `int`...), to or from `bool`, and some pointer conversions.

Converting to `int` from some smaller integer type, or to `double` from `float` is known as *promotion*, and is guaranteed to produce the exact same value in the destination type. Other conversions between arithmetic types may not always be able to represent the same value exactly:

- If a negative integer value is converted to an unsigned type, the resulting value corresponds to its 2's complement bitwise representation (i.e., `-1` becomes the largest value representable by the type, `-2` the second largest, ...).
- The conversions from/to `bool` consider `false` equivalent to *zero* (for numeric types) and to *null pointer* (for pointer types); `true` is equivalent to all other values and is converted to the equivalent of `1`.
- If the conversion is from a floating-point type to an integer type, the value is truncated (the decimal part is removed). If the result lies outside the range of representable values by the type, the conversion causes *undefined behavior*.
- Otherwise, if the conversion is between numeric types of the same kind (integer-to-integer or floating-to-floating), the conversion is valid, but the value is *implementation-specific* (and may not be portable).

Some of these conversions may imply a loss of precision, which the compiler can signal with a warning. This warning can be avoided with an explicit conversion.

For non-fundamental types, arrays and functions implicitly convert to pointers, and pointers in general allow the following conversions:

- *Null pointers* can be converted to pointers of any type
- Pointers to any type can be converted to `void` pointers.
- Pointer *upcast*: pointers to a derived class can be converted to a pointer of an *accessible* and *unambiguous* base class, without modifying its `const` or `volatile` qualification.

Implicit conversions with classes

In the world of classes, implicit conversions can be controlled by means of three member functions:

- **Single-argument constructors:** allow implicit conversion from a particular type to initialize an object.
- **Assignment operator:** allow implicit conversion from a particular type on assignments.
- **Type-cast operator:** allow implicit conversion to a particular type.

For example:

```
1  // implicit conversion of classes:
2  #include <iostream>
3  using namespace std;
```

```
4
5   class A {};
6
7   class B {
8   public:
9     // conversion from A (constructor):
10    B (const A& x) {}
11    // conversion from A (assignment):
12    B& operator= (const A& x) {return *this;}
13    // conversion to A (type-cast operator)
14    operator A() {return A();}
15  };
16
17  int main ()
18  {
19    A foo;
20    B bar = foo;     // calls constructor
21    bar = foo;       // calls assignment
22    foo = bar;       // calls type-cast operator
23    return 0;
24  }
```

The type-cast operator uses a particular syntax: it uses the `operator` keyword followed by the destination type and an empty set of parentheses. Notice that the return type is the destination type and thus is not specified before the `operator` keyword.

Keyword explicit

On a function call, C++ allows one implicit conversion to happen for each argument. This may be somewhat problematic for classes, because it is not always what is intended. For example, if we add the following function to the last example:

```
void fn (B arg) {}
```

This function takes an argument of type `B`, but it could as well be called with an object of type `A` as argument:

```
fn (foo);
```

This may or may not be what was intended. But, in any case, it can be prevented by marking the affected constructor with the `explicit` keyword:

```
1   // explicit:
2   #include <iostream>
3   using namespace std;
4
5   class A {};
6
7   class B {
8   public:
9     explicit B (const A& x) {}
10    B& operator= (const A& x) {return *this;}
11    operator A() {return A();}
12  };
13
14  void fn (B x) {}
15
16  int main ()
```

```
{
  A foo;
  B bar (foo);
  bar = foo;
  foo = bar;

//  fn (foo);  // not allowed for explicit ctor.
  fn (bar);

  return 0;
}
```

Additionally, constructors marked with `explicit` cannot be called with the assignment-like syntax; In the above example, `bar` could not have been constructed with:

```
B bar = foo;
```

Type-cast member functions (those described in the previous section) can also be specified as `explicit`. This prevents implicit conversions in the same way as `explicit`-specified constructors do for the destination type.

→ Type casting

C++ is a strong-typed language. Many conversions, specially those that imply a different interpretation of the value, require an explicit conversion, known in C++ as *type-casting*. There exist two main syntaxes for generic type-casting: *functional* and *c-like*:

```
double x = 10.3;
int y;
y = int (x);    // functional notation
y = (int) x;    // c-like cast notation
```

The functionality of these generic forms of type-casting 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. For example, the following code compiles without errors:

```
// class type-casting
#include <iostream>
using namespace std;

class Dummy {
    double i,j;
};

class Addition {
    int x,y;
  public:
    Addition (int a, int b) { x=a; y=b; }
    int result() { return x+y;}
};

int main () {
  Dummy d;
  Addition * padd;
  padd = (Addition*) &d;
  cout << padd->result();
  return 0;
}
```

The program declares a pointer to `Addition`, but then it assigns to it a reference to an object of another unrelated type using explicit type-casting:

```
padd = (Addition*) &d;
```

Unrestricted 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 some other unexpected results.

In order to control these types of conversions between classes, we have four specific casting operators: `dynamic_cast`, `reinterpret_cast`, `static_cast` and `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)
```

but each one with its own special characteristics:

→ dynamic_cast

`dynamic_cast` can only be used with pointers and references to classes (or with `void*`). Its purpose is to ensure that the result of the type conversion points to a valid complete object of the destination pointer type.

This naturally includes *pointer upcast* (converting from pointer-to-derived to pointer-to-base), in the same way as allowed as an *implicit conversion*.

But `dynamic_cast` can also *downcast* (convert from pointer-to-base to pointer-to-derived) polymorphic classes (those with virtual members) if -and only if- the pointed object is a valid complete object of the target type. For example:

```
1  // dynamic_cast
2  #include <iostream>
3  #include <exception>
4  using namespace std;
5
6  class Base { virtual void dummy() {} };
7  class Derived: public Base { int a; };
8
9  int main () {
10     Base * pba = new Derived;
11     Base * pbb = new Base;
12     Derived * pd;
13
14     pd = dynamic_cast<Derived*>(pba);
15     if (pd==0) cout << "Null pointer on first type-cast.\n";
16
17     pd = dynamic_cast<Derived*>(pbb);
```

```
Null pointer on
second type-
cast.
```

```
18      if (pd==0) cout << "Null pointer on second type-cast.\n";
19  return 0;
20  }
```

Compatibility note: This type of `dynamic_cast` requires *Run-Time Type Information (RTTI)* to keep track of dynamic types. Some compilers support this feature as an option which is disabled by default. This needs to be enabled for runtime type checking using `dynamic_cast` to work properly with these types.

The code above tries to perform two dynamic casts from pointer objects of type `Base*` (`pba` and `pbb`) to a pointer object of type `Derived*`, but only the first one is successful. Notice their respective initializations:

```
1  Base * pba = new Derived;
2  Base * pbb = new Base;
```

Even though both are pointers of type `Base*`, `pba` actually points to an object of type `Derived`, while `pbb` points to an object of type `Base`. Therefore, when their respective type-casts are performed using `dynamic_cast`, `pba` is pointing to a full object of class `Derived`, whereas `pbb` is pointing to an object of class `Base`, which is an incomplete object of class `Derived`.

When `dynamic_cast` cannot cast a pointer because it is not a complete object of the required class -as in the second conversion in the previous example- it returns a *null pointer* to indicate the failure. If `dynamic_cast` is used to convert to a reference type and the conversion is not possible, an exception of type `bad_cast` is thrown instead.

`dynamic_cast` can also perform the other implicit casts allowed on pointers: casting null pointers between pointers types (even between unrelated classes), and casting any pointer of any type to a `void*` pointer.

→ static_cast

`static_cast` can perform conversions between pointers to related classes, not only *upcasts* (from pointer-to-derived to pointer-to-base), but also *downcasts* (from pointer-to-base to pointer-to-derived). No checks are performed during runtime to guarantee that the object being converted is in fact a full object of the destination type. Therefore, it is up to the programmer to ensure that the conversion is safe. On the other side, it does not incur the overhead of the type-safety checks of `dynamic_cast`.

```
1  class Base {};
2  class Derived: public Base {};
3  Base * a = new Base;
4  Derived * b = static_cast<Derived*>(a);
```

This would be valid code, although `b` would point to an incomplete object of the class and could lead to runtime errors if dereferenced.

Therefore, `static_cast` is able to perform with pointers to classes not only the conversions allowed implicitly, but also their opposite conversions.

`static_cast` is also able to perform all conversions allowed implicitly (not only those with pointers to

classes), and is also able to perform the opposite of these. It can:
. Convert from `void*` to any pointer type. In this case, it guarantees that if the `void*` value was obtained by converting from that same pointer type, the resulting pointer value is the same.
. Convert integers, floating-point values and enum types to enum types.

Additionally, `static_cast` can also perform the following:
. Explicitly call a single-argument constructor or a conversion operator.
. Convert to *rvalue references*.
. Convert `enum class` values into integers or floating-point values.
. Convert any type to `void`, evaluating and discarding the value.

→ reinterpret_cast

`reinterpret_cast` converts any pointer type to any other pointer type, even of unrelated classes. The operation result is a simple binary copy of the value from one pointer to the other. All pointer conversions are allowed: neither the content pointed nor the pointer type itself is checked.

It can also cast pointers to or from integer types. The format in which this integer value represents a pointer is platform-specific. The only guarantee is that a pointer cast to an integer type large enough to fully contain it (such as `intptr_t`), is guaranteed to be able to be cast back to a valid pointer.

The conversions that can be performed by `reinterpret_cast` but not by `static_cast` are low-level operations based on reinterpreting the binary representations of the types, which on most cases results in code which is system-specific, and thus non-portable. For example:

```
class A { /* ... */ };
class B { /* ... */ };
A * a = new A;
B * b = reinterpret_cast<B*>(a);
```

This code compiles, although it does not make much sense, since now `b` points to an object of a totally unrelated and likely incompatible class. Dereferencing `b` is unsafe.

→ const_cast

This type of casting manipulates the constness of the object pointed by a pointer, either to be set or to be removed. For example, in order to pass a const pointer to a function that expects a non-const argument:

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

void print (char * str)
{
  cout << str << '\n';
}

int main () {
  const char * c = "sample text";
  print ( const_cast<char *> (c) );
  return 0;
}
```

```
sample text
```

The example above is guaranteed to work because function `print` does not write to the pointed object. Note though, that removing the constness of a pointed object to actually write to it causes *undefined behavior*.

→ typeid

`typeid` allows to check the type of an expression:

```
typeid (expression)
```

This operator returns a reference to a constant object of type `type_info` that is defined in the standard header `<typeinfo>`. A value returned by `typeid` can be compared with another value returned by `typeid` using operators == and `!=` or can serve to obtain a null-terminated character sequence representing the data type or class name by using its `name()` member.

```
1  // typeid
2  #include <iostream>
3  #include <typeinfo>
4  using namespace std;
5
6  int main () {
7    int * a,b;
8    a=0; b=0;
9    if (typeid(a) != typeid(b))
10   {
11     cout << "a and b are of different types:\n";
12     cout << "a is: " << typeid(a).name() << '\n';
13     cout << "b is: " << typeid(b).name() << '\n';
14   }
15   return 0;
16 }
```

```
a and b are of different
types:
a is: int *
b is: int
```

When `typeid` is applied to classes, `typeid` uses the RTTI to keep track of the type of dynamic objects. When `typeid` is applied to an expression whose type is a polymorphic class, the result is the type of the most derived complete object:

```
1  // typeid, polymorphic class
2  #include <iostream>
3  #include <typeinfo>
4  #include <exception>
5  using namespace std;
6
7  class Base { virtual void f(){} };
8  class Derived : public Base {};
9
10 int main () {
11     Base* a = new Base;
12     Base* b = new Derived;
13     cout << "a is: " << typeid(a).name() << '\n';
14     cout << "b is: " << typeid(b).name() << '\n';
15     cout << "*a is: " << typeid(*a).name() << '\n';
16     cout << "*b is: " << typeid(*b).name() << '\n';
17 return 0;
18 }
```

```
a is: class Base *
b is: class Base *
*a is: class Base
*b is: class Derived
```

Note: The string returned by member `name` of `type_info` depends on the specific implementation of your compiler and library. It is not necessarily a simple string with its typical type name, like in the compiler used to produce this output.

Notice how the type that `typeid` considers for pointers is the pointer type itself (both `a` and `b` are of type `class Base *`). However, when `typeid` is applied to objects (like `*a` and `*b`) `typeid` yields their dynamic type (i.e. the type of their most derived complete object).

If the type `typeid` evaluates is a pointer preceded by the dereference operator (`*`), and this pointer has a null value, `typeid` throws a `bad_typeid` exception.