Unit-2/ Lecture-1

Identifying objects and classes [RGPV/Feb2012 (5)]

In object-oriented software design (OOD)

In object-oriented software design (OOD), classes are templates for defining the characteristics and operations of an object. Often, classes and objects are used interchangeably, one synonymous with the other. In actuality, a class is a specification that an object implements.

Identifying classes can be challenging. Poorly chosen classes can complicate the application's logical structure, reduce reusability, and hinder maintenance. This article provides a brief overview of object-oriented classes and offers tips and suggestions to identify cohesive classes.

Note: The following class diagrams were modelled using Enterprise Architect. Many other modelling tools exist. Use the one that is best suited for your purpose and project.

Classes

Object-oriented classes support the object-oriented principles of abstraction, encapsulation, polymorphism and reusability. They do so by providing a template, or blueprint, that defines the variables and the methods common to all objects that are based on it. Classes specify knowledge (attributes) - they know things - and behaviour (methods) - they do things.

Classes are specifications for objects.

Derived from the Use Cases, classes provide an abstraction of the requirements and provide the internal view of the application.

Attributes:-

Attributes define the characteristics of the class that, collectively, capture all the information about the class. Attributes should be protected by their enclosing class. Unless changed by the class' behavior, attributes maintain their values.

The type of data that an attribute can contain is determined by its data type. There are two basic data types: Primitive and Derived.

Primitive data types are fundamental types. Examples are integer, string, float.

Derived data types are defined in terms of the Primitive data types, that is, they form new data types by extending the primitive data types. A Student class, for example, is a derived data type formed by a collection of primitive data types.

When defined in the context of a problem domain, derived data types are called Domain Specific Data types. These are the types that define and constrain attributes to be consistent with the semantics of the data. For example, Address student Address versus string student Address.

Object composition

In computer science, object composition (not to be confused with function composition) is a way to combine simple objects or data types into more complex ones. Compositions are a critical building block of many basic data structures, including the tagged union, the linked list, and the binary tree, as well as the object used in object-oriented

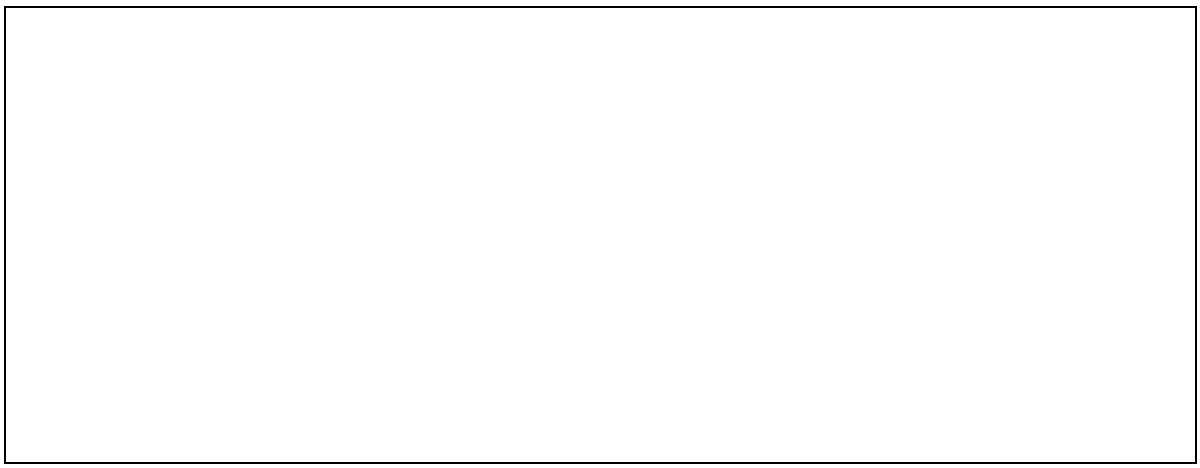
programming.

A real-world example of composition may be seen in the relation of an automobile to its parts, specifically: the automobile' has or is composed from' objects including steering wheel, seat, gearbox and engine.

When, in a language, objects are typed, types can often be divided into composite and non composite types, and composition can be regarded as a relationship between types: an object of a composite type (e.g. car) "has an" object of a simpler type (e.g. wheel). Composition must be distinguished from sub typing, which is the process of adding detail to a general data type to create a more specific data type. For instance, cars may be a specific type of vehicle: car is a vehicle. Sub typing doesn't describe a relationship between different objects, but instead, says that objects of a type are simultaneously objects of another type.

In programming languages, composite objects are usually expressed by means of references from one object to another; depending on the language, such references may be known as fields, members, properties or attributes, and the resulting composition as a structure, storage record, tuple, user-defined type (UDT), or composite type. Fields are given a unique name so that each one can be distinguished from the others. However, having such references doesn't necessarily mean that an object is a composite. It is only called composite if the objects it refers to are really its parts, i.e. have no independent existence.

S.NO	RGPV QUESTIONS	Year	Marks
Q.1	Compare the class diagram & object diagram.	Feb-2010	5
Q.2	Compare the class attributes & method.	Feb-2010	5



Unit-2/Lecture-3

Association, Aggregation & composition [RGPV/June2011(10),Feb2010(10),June2009(10)]

The whole point of OOP is that your code replicates real world objects, thus making your code readable and maintainable. When we say real world, the real world has relationships. Let's consider the simple requirement listed below:

Manager is an employee of XYZ limited corporation.
Manager uses a swipe card to enter XYZ premises.
Manager has workers who work under him.
Manager has the responsibility of ensuring that the project is successful.
Manager's salary will be judged based on project success.

If you flesh out the above five point requirement, we can easily visualize four relationships:-

Inheritance
Aggregation
Association
Composition

Let's understand them one by one.

Requirement 1: The IS A relationship

If you look at the first requirement (Manager is an employee of XYZ limited corporation), it's a parent child relationship or inheritance relationship. The sentence above specifies that Manager is a type of employee, in other words we will have two classes: parent class Employee, and a child class Manager which will inherit from the Employee class.

Note: The scope of this article is only limited to aggregation, association, and composition. We will not discuss inheritance in this article as it is pretty straightforward and I am sure you can get 1000s of articles on the net which will help you in understanding it.

Requirement 2: The Using relationship: Association

Requirement 2 is an interesting requirement (Manager uses a swipe card to enter XYZ premises). In this requirement, the manager object and the swipe card object use each other but they have their own object life time. In other words, they can exist without each other. The most important point in this relationship is that there is no single owner.

The above diagram shows how the SwipeCard class uses the Manager class and the Manager class uses the SwipeCard class. You can also see how we can create objects of the Manager class and SwipeCard class independently and they can have their own object life time.

This relationship is called the "Association" relationship.

Requirement 3: The Using relationship with Parent: Aggregation

The third requirement from our list (Manager has workers who work under him) denotes the same type of relationship like association but with a difference that one of them is an owner. So as per the requirement, the Manager object will own Worker objects.

The child Worker objects can not belong to any other object. For instance, a Worker

object cannot belong to a SwipeCard object.
But… the Worker object can have its own life time which is completely disconnected from the Manager object. Looking from a different perspective, it means that if the Manager object is deleted, the Worker object does not die.
This relationship is termed as an "Aggregation" relationship.

Requirements 4 and 5: the Death relationship: Composition
The last two requirements are actually logically one. If you read closely, the requirements are as follows:

Manager has the responsibility of ensuring that the project is successful.

Manager's salary will be judged based on project success.

Below is the conclusion from analyzing the above requirements:

Manager and the project objects are dependent on each other.

The lifetimes of both the objects are the same. In other words, the project will not be successful if the manager is not good, and the manager will not get good increments if the project has issues.

Below is how the class formation will look like. You can also see that when I go to create the project object, it needs the manager object.
This relationship is termed as the composition relationship. In this relationship, both objects are heavily dependent on each other. In other words, if one goes for garbage collection the other also has to be garbage collected, or putting from a different perspective, the lifetime of the objects are the same. That's why I have put in the heading "Death" relationship.

Putting things together
Below is a visual representation of how the relationships have emerged from the requirements.
Summarizing
To avoid confusion henceforth for these three terms, I have put forward a table below which will help us compare them from three angles: owner, lifetime, and child object.

	Association	**Aggregation**	**Composition**
Owner	No owner	Single owner	Single owner
Life time	Have their own lifetime	Have their own lifetime	Owner's life time
Child object	Child objects all are independent	Child objects belong to a single parent	Child objects belong to a single parent

Dynamic memory[RGPV/June2010 (10)]

In the programs seen in previous chapters, all memory needs were determined before program execution by defining the variables needed. But there may be cases where the memory needs of a program can only be determined during runtime. For example, when the memory needed depends on user input. On these cases, programs need to dynamically allocate memory, for which the C++ language integrates the operators new anddelete.

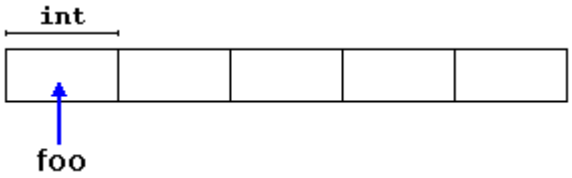
Operators new and new
Dynamic memory is allocated using operator new. new is followed by a data type specifier and, if a sequence of more than one element is required, the number of these within brackets []. It returns a pointer to the beginning of the new block of memory allocated. Its syntax is:

pointer = new type
pointer = new type [number_of_elements]

The first expression is used to allocate memory to contain one single element of type type. The second one is used to allocate a block (an array) of elements of type type, where number_of_elements is an integer value representing the amount of these. For example:

```
1 int * foo;
2 foo = new int [5];
```

In this case, the system dynamically allocates space for five elements of type int and returns a pointer to the first element of the sequence, which is assigned to foo (a pointer). Therefore, foo now points to a valid block of memory with space for five elements of type int.



Here, foo is a pointer, and thus, the first element pointed to by foo can be accessed either with the expressionfoo[0] or the expression *foo (both are equivalent). The second element can be accessed either with foo[1] or *(foo+1), and so on...

There is a substantial difference between declaring a normal array and allocating dynamic memory for a block of memory using new. The most important difference is that the size of a regular array needs to be a constant expression, and thus its size has to be determined at the moment of designing the program, before it is run, whereas the dynamic memory allocation performed by new allows to assign memory during runtime using any variable value as size.

The dynamic memory requested by our program is allocated by the system from the memory heap. However, computer memory is a limited resource, and it can be exhausted. Therefore, there are no guarantees that all requests to allocate memory using operator new are going to be granted by the system.

C++ provides two standard mechanisms to check if the allocation was successful:

One is by handling exceptions. Using this method, an exception of type bad_alloc is thrown when the allocation fails. Exceptions are a powerful C++ feature explained later in these tutorials. But for now, you should know that if this exception is thrown and it is not handled by a specific handler, the program execution is terminated.

This exception method is the method used by default by new, and is the one used in a declaration like:

```
foo = new int [5];  // if allocation fails, an exception is thrown
```

The other method is known as nothrow, and what happens when it is used is that when a memory allocation fails, instead of throwing a bad_alloc exception or terminating the program, the pointer returned by new is a null pointer, and the program continues its execution normally.

This method can be specified by using a special object called nothrow, declared in header <new>, as argument fornew:

```
foo = new (nothrow) int [5];
```

In this case, if the allocation of this block of memory fails, the failure can be detected by checking if foo is a null pointer:

```
int * foo;
foo = new (nothrow) int [5];
if (foo == nullptr) {
// error assigning memory. Take measures.
}
```

This no throw method is likely to produce less efficient code than exceptions, since it implies explicitly checking the pointer value returned after each and every allocation. Therefore, the exception mechanism is generally preferred, at least for critical allocations. Still, most of the coming examples will use the no throw mechanism due to its simplicity.

S.NO	RGPV QUESTIONS	Year	Marks
Q.1	Compare association, aggregation & composition.	RGPV June 2010	10
Q.2	How objects are assigned memory dynamically in C++? Explain by giving proper examples.	RGPV June 2010	10

Unit-2/Lecture-4

Scope Resolution Operator [RGPV/June2011 (10)]

There are two uses of the scope resolution operator in C++.
The first use being that a scope resolution operator is used to unhide the global variable that might have got hidden by the local variables. Hence in order to access the hidden global variable one needs to prefix the variable name with the scope resolution operator (::).
e.g.

```
int i = 10;
int main ()
{
int i = 20;
cout << i; // this prints the value 20
cout << ::i; // in order to use the global i one needs to prefix it with the scope resolution operator.
}
```

The second use of the operator is used to access the members declared in class scope. Whenever a scope resolution operator is used the name of the member that follows the operator is looked up in the scope of the class with the name that appears before the operator.

The scope resolution operator (::) in C++ is used to define the already declared member functions (in the header file with the .hpp or the .h extension) of a particular class. In the .cpp file one can define the usual global functions or the member functions of the class. To differentiate between the normal functions and the member functions of the class, one needs to use the scope resolution operator (::) in between the class name and the member function name i.e. ship::foo() where ship is a class and foo() is a member function of the class ship. The other uses of the resolution operator is to resolve the scope of a variable when the same identifier is used to represent a global variable, a local variable, and members of one or more class(es). If the resolution operator is placed between the class name and the data member belonging to the class then the data name belonging to the particular class is referenced. If the resolution operator is placed in front of the variable name then the global variable is referenced. When no resolution operator is placed then the local variable is referenced.

```
#include <iostream>
using namespace std;
int n = 12; // A global variable
int main() {
int n = 13; // A local variable
cout << ::n << endl; // Print the global variable: 12
cout << n << endl; // Print the local variable: 13
}
```

Type conversions [RGPV/June2011 (10)]

Implicit conversion

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

```
short a=2000;
int b;
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:

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

class A {};

class B {
public:
  // conversion from A (constructor):
  B (const A& x) {}
  // conversion from A (assignment):
  B& operator= (const A& x) {return *this;}
  // conversion to A (type-cast operator)
  operator A() {return A();}
};

int main ()
{
  A foo;
  B bar = foo;    // calls constructor
  bar = foo;      // calls assignment
  foo = bar;      // calls type-cast operator
  return 0;
}
```

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:

```
// explicit:
#include <iostream>
using namespace std;
class A {};
class B {
public:
```

```
  explicit B (const A& x) {}
  B& operator= (const A& x) {return *this;}
  operator A() {return A();}
};
void fn (B x) {}
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 [RGPV/June2011(10)]

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:

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

class Base { virtual void dummy() {} };
class Derived: public Base { int a; };

int main () {
  try {
    Base * pba = new Derived;
    Base * pbb = new Base;
    Derived * pd;

    pd = dynamic_cast<Derived*>(pba);
    if (pd==0) cout << "Null pointer on first type-cast.\n";

    pd = dynamic_cast<Derived*>(pbb);
    if (pd==0) cout << "Null pointer on second type-
cast.\n";

  } catch (exception& e) {cout << "Exception: " <<
e.what();}
  return 0;
}
```

Null pointer on second type-cast.

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:

```
Base * pba = new Derived;
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.

```
class Base {};
class Derived: public Base {};
Base * a = new Base;
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.

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

int main () {
  int * a,b;
  a=0; b=0;
  if (typeid(a) != typeid(b))
  {
    cout << "a and b are of different types:\n";
    cout << "a is: " << typeid(a).name() << '\n';
```

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

```
    cout << "b is: " << typeid(b).name() << '\n';
  }
  return 0;
}
```

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:

```
// typeid, polymorphic class
#include <iostream>
#include <typeinfo>
#include <exception>
using namespace std;

class Base { virtual void f(){} };
class Derived : public Base {};

int main () {
  try {
    Base* a = new Base;
    Base* b = new Derived;
    cout << "a is: " << typeid(a).name() << '\n';
    cout << "b is: " << typeid(b).name() << '\n';
    cout << "*a is: " << typeid(*a).name() << '\n';
    cout << "*b is: " << typeid(*b).name() << '\n';
  } catch (exception& e) { cout << "Exception: " << e.what() << '\n'; }
  return 0;
}
```

```
a is: class Base
b is: class Base
*a is: class Ba
*b is: class De
```

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.

S.NO	RGPV QUESTIONS	Year	Marks
Q.1	Explain scope resolution operators:: & its application in C++.	RGPV June 2011	10
Q.2	What is cast operator? It is possible to have multiple cast operators in a class? Explain with an example.	RGPV June 2011	10

Unit-2/Lecture-5

Object Oriented Modelling [RGPV/Dec2010(12),June2009 (10)]

The prevalence of programming languages such as Java, C++, Object Pascal, C#, and Visual Basic make it incredibly clear that object-oriented technology has become the approach of choice for new development projects. Although procedural languages such as COBOL and PL/1 will likely be with us for decades it is clear that most organizations now consider these environments as legacy technologies that must be maintained and ideally retired at some point. Progress marches on.

My experience is that agile software developers, be they application developers or Agile DBAs, must minimally have an understanding of object orientation if they are to be effective. This includes understanding basic concepts such as inheritance, polymorphism, and object persistence. Furthermore, all developers should have a basic understanding of the industry-standard Unified Modeling Language (UML). A good starting point is to understand what I consider to be the core UML diagrams – use case diagrams, sequence diagrams, and class diagrams – although as I argued in An Introduction to Agile Modeling and Agile Documentation you must be willing to learn more models over time. One of the advantages of working closely with other IT professionals is that you learn new skills from them, and the most effective object developers will learn and adapt fundamental concepts from other disciplines. An example is class normalization, the object-oriented version of data normalization, a collection of simple rules for reducing coupling and increasing cohesion within your object designs.

This article overviews the fundamental concepts and techniques that application developers use on a daily basis when working with object technology. This article is aimed at Agile DBAs that want to gain a basic understanding of the object paradigm, allowing them to understand where application developers are coming from. The primary goal of this article is to provide Agile DBAs with enough of an understanding of objects so that they have a basis from which to communicate with application developers. Similarly, other articles overview fundamental data concepts, such as relational database technology and data modeling that application developers need to learn so that they understand where Agile DBAs are coming from.

Object-Oriented Concepts

Agile software developers, including Agile DBAs, need to be familiar with the basic concepts of object-orientation. The object-oriented (OO) paradigm is a development strategy based on the concept that systems should be built from a collection of reusable components called objects. Instead of separating data and functionality as is done in the structured paradigm, objects encompass both. While the object-oriented paradigm sounds similar to the structured paradigm, as you will see at this site it is actually quite different. A common mistake that many experienced developers make is to assume that they have been “doing objects” all along just because they have been applying similar software-engineering principles. To succeed you must recognize that the OO approach is different than the structured

To understand OO you need to understand common object terminology. The critical terms to understand are summarized in Table 1. I present a much more detailed

explanation of these terms in The Object Primer 3/e. Some of these concepts you will have seen before, and some of them you haven't. Many OO concepts, such as encapsulation, coupling, and cohesion come from software engineering. These concepts are important because they underpin good OO design. The main point to be made here is that you do not want to deceive yourself – just because you have seen some of these concepts before, it don't mean you were doing OO, it just means you were doing good design. While good design is a big part of object-orientation, there is still a lot more to it than that.

Table 1. A summary of common object-oriented terms.

Term	Description
Abstract class	A class that does not have objects instantiated from it
Abstraction	The identification of the essential characteristics of an item
Aggregation	Represents "is part of" or "contains" relationships between two classes or components
Aggregation hierarchy	A set of classes that are related through aggregation
Association	Objects are related (associated) to other objects
Attribute	Something that a class knows (data/information)
Class	A software abstraction of similar objects, a template from which objects are created
Cohesion	The degree of relatedness of an encapsulated unit (such as a component or a class)
Collaboration	Classes work together (collaborate) to fulfill their responsibilities
Composition	A strong form of aggregation in which the "whole" is completely responsible for its parts and each "part" object is only associated to the one "whole" object
Concrete class	A class that has objects instantiated from it
Coupling	The degree of dependence between two items
Encapsulation	The grouping of related concepts into one item, such as a class or component
Information hiding	The restriction of external access to attributes
Inheritance	Represents "is a", "is like", and "is kind of" relationships. When class "B" inherits from class "A" it automatically has all of the attributes and operations that "A" implements (or inherits from other classes)
Inheritance hierarchy	A set of classes that are related through inheritance
Instance	An object is an instance of a class
Instantiate	We instantiate (create) objects from classes
Interface	The definition of a collection of one or more operation signatures that defines a cohesive set of behaviors
Message	A message is either a request for information or a request to perform an action
Messaging	In order to collaborate, classes send messages to each other
Multiple	When a class directly inherits from more than one class

inheritance	
Multiplicity	A UML concept combining the data modeling concepts of cardinality (how many) and optionality.
Object	A person, place, thing, event, concept, screen, or report
Object space	Main memory + all available storage space on the network, including persistent storage such as a relational database
Operation	Something a class does (similar to a function in structured programming)
Override	Sometimes you need to override (redefine) attributes and/or methods in subclasses
Pattern	A reusable solution to a common problem taking relevant forces into account
Persistence	The issue of how objects are permanently stored
Persistent object	An object that is saved to permanent storage
Polymorphism	Different objects can respond to the same message in different ways, enable objects to interact with one another without knowing their exact type
Single inheritance	When a class directly inherits from only one class
Stereotype	Denotes a common usage of a modeling element
Subclass	If class “B” inherits from class “A,” we say that “B” is a subclass of “A”
Superclass	If class “B” inherits from class “A,” we say that “A” is a superclass of “B”
Transient object	An object that is not saved to permanent storage

It is important for Agile DBAs to understand the terms presented above because the application developers that you work with will use these terms, and many others, on a regular basis. To communicate effectively with application developers you must understand their vocabulary, and they must understand yours. Another important aspect of learning the basics of object orientation is to understand each of the diagrams of the Unified Modeling Language (UML) – you don’t need to become a UML expert, but you do need to learn the basics.

2. An Overview of the Unified Modeling Language

The goal of this section is to provide you with a basic overview of the UML, it is not to teach you the details of each individual technique. Much of the descriptiv material in this section is modified from The Elements of UML Style, a pocket-sized book that describes proven guidelines for developing high-quality and readable UML diagrams, and the examples from The Object Primer 3/e. A good starting point for learning the UML is UML Distilled as it is well written and concise. If you want a more thorough look at the UML, as well as other important models that the UML does not include, then you’ll find The Object Primer 3/e to be a better option.

It is also important to understand that you don’t need to learn all of the UML notation available to you, and believe me there’s a lot, but only the notation that you’ll use in practice. The examples presented in this section, there is one for each UML diagram, use the core UML. As you learn each diagram focus on learning the core notation first, you can learn the rest of the notation over time as you need to.

2.1 Core UML Diagrams

Let's begin with what I consider to be the three core UML diagrams for developing business software: UML use case diagrams, UML sequence diagrams, and UML class diagrams. These are the diagrams that you will see used the most in practice – use case diagrams to overview usage requirements, sequence diagrams to analyze the use cases and map to your classes, and class diagrams to explore the structure of your object-oriented software (what I like to refer to as your object schema). These three diagrams will cover 80% of your object modeling needs when building a business application using object technology.

2.1.1 UML Use Case Diagrams

According to the UML specification a use case diagram is "a diagram that shows the relationships among actors and use cases within a system." Use case diagrams are often used to:

Provide an overview of all or part of the usage requirements for a system or organization in the form of an essential (Constantine and Lockwood 1999) model or a business model (Rational Corporation 2001)

Communicate the scope of a development project

Model the analysis of your usage requirements in the form of a system use case model (Cockburn 2001a)

Figure 1 depicts a simple use case diagram. This diagram depicts several use cases, actors, their associations, and optional system boundary boxes. A use case describes a sequence of actions that provide a measurable value to an actor and is drawn as a horizontal ellipse. An actor is a person, organization, or external system that plays a role in one or more interactions with your system. Actors are drawn as stick figures. Associations between actors and classes are indicated in use-case diagrams, a relationship exists whenever an actor is involved with an interaction described by a use case. Associations between actors and use cases are modeled as lines connecting them to one another, with

an optional arrowhead on one end of the line indicating the direction of the initial invocation of the relationship.

S.NO	RGPV QUESTIONS	Year	Marks
Q.1	Explain Rambaugh's OMT in terms of OO modelling.	Dec2010, June 2009	10,12

UNIT- 2/LECTURE- 6

C++ Dynamic Memory [RGPV/June2013(12),Feb2010(10),June2009 (10)]

A good understanding of how dynamic memory really works in C++ is essential to becoming a good C++ programmer. Memory in your C++ program is divided into two parts:

The stack: All variables declared inside the function will take up memory from the stack.

The heap: This is unused memory of the program and can be used to allocate the memory dynamically when program runs.

Many times, you are not aware in advance how much memory you will need to store particular information in a defined variable and the size of required memory can be determined at run time.

You can allocate memory at run time within the heap for the variable of a given type using a special operator in C++ which returns the address of the space allocated. This operator is called **new** operator.

If you are not in need of dynamically allocated memory anymore, you can use **delete** operator, which de-allocates memory previously allocated by new operator.

The new and delete operators:

There is following generic syntax to use **new** operator to allocate memory dynamically for any data-type.

New data-type;

Here, **data-type** could be any built-in data type including an array or any user defined data types include class or structure. Let us start with built-in data types. For example we can define a pointer to type double and then request that the memory be allocated at execution time. We can do this using the **new**operator with the following statements:

```
double* pvalue  = NULL; // Pointer initialized with null
pvalue  = new double;   // Request memory for the variable
```

The memory may not have been allocated successfully, if the free store had been used up. So it is good practice to check if new operator is returning NULL pointer and take appropriate action as below:

```
double* pvalue  = NULL;
if( !(pvalue  = new double ))
{
  cout << “Error: out of memory.” <<endl;
  exit(1);
```

}The **malloc()** function from C, still exists in C++, but it is recommended to avoid using malloc() function. The main advantage of new over malloc() is that new doesn’t just allocate memory, it constructs objects which is prime purpose of C++.

At any point, when you feel a variable that has been dynamically allocated is not anymore required, you can free up the memory that it occupies in the free store with the delete operator as follows:

```
delete pvalue;        // Release memory pointed to by pvalue
```

Let us put above concepts and form the following example to show how new and delete work:

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

```
int main ()
{
  double* pvalue  = NULL; // Pointer initialized with null
  pvalue  = new double;   // Request memory for the variable

  *pvalue = 29494.99;     // Store value at allocated address
  cout << “Value of pvalue : “ << *pvalue << endl;

  delete pvalue;        // free up the memory.

  Return 0;
}
```

If we compile and run above code, this would produce the following result:
Value of pvalue : 29495
Dynamic Memory Allocation for Arrays:
Consider you want to allocate memory for an array of characters, i.e., string of 20 characters. Using the same syntax what we have used above we can allocate memory dynamically as shown below.

```
Char* pvalue  = NULL;   // Pointer initialized with null
pvalue  = new char[20]; // Request memory for the variable
```

To remove the array that we have just created the statement would look like this:

```
delete [] pvalue;       // Delete array pointed to by pvalue
```

Following the similar generic syntax of new operator, you can llocate for a multi-dimensional array as follows:

```
double** pvalue  = NULL;     // Pointer initialized with null
pvalue  = new double [3][4]; // Allocate memory for a 3x4 array
```

However, the syntax to release the memory for multi-dimensional array will still remain same as above:

```
delete [] pvalue;       // Delete array pointed to by pvalue
```

Dynamic Memory Allocation for Objects:
Objects are no different from simple data types. For example, consider the following code where we are going to use an array of objects to clarify the concept:

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

class Box
{
  public:
    Box() {
      cout << “Constructor called!” <<endl;
    }
    ~Box() {
      cout << “Destructor called!” <<endl;
    }
};

int main( )
{
  Box* myBoxArray = new Box[4];
```

```
  delete [] myBoxArray; // Delete array

  return 0;
}
```

If you were to allocate an array of four Box objects, the Simple constructor would be called four times and similarly while deleting these objects, destructor will also be called same number of times.

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

```
Constructor called!
Constructor called!
Constructor called!
Constructor called!
Destructor called!
Destructor called!
Destructor called!
Destructor called!
```

S.NO	RGPV QUESTION	YEAR	MARKS
Q.1	Explain Dynamic memory allocation in C++.	Feb2010	10
Q.2	Describe memory allocation in C++ with the help of code	June2013	12

UNIT- 2/LECTURE- 7

Template classes[RGPV/Dec,June2010(10)]

In the previous two lessons, you learn how function templates and function template instances could be used to generalize functions to work with many different data types. While this is a great start down the road to generalized programming, it doesn't solve all of our problems. Let's take a look at an example of one such problem, and see what templates can do for us further.

Templates and container classes

In the lesson on container classes, you learned how to use composition to implement classes that contained multiple instances of other classes. As one example of such a container, we took a look at the IntArray class. Here is a simplified example of that class:

```
#ifndef INTARRAY_H
#define INTARRAY_H

#include <assert.h> // for assert()

class IntArray
{
private:
   int m_nLength;
   int *m_pnData;

public:
   IntArray()
   {
      m_nLength = 0;
      m_pnData = 0;
   }

   IntArray(int nLength)
   {
      m_pnData = new int[nLength];
      m_nLength = nLength;
   }

   ~IntArray()
   {
      delete[] m_pnData;
   }

   void Erase()
   {
      delete[] m_pnData;
```

```
    // We need to make sure we set m_pnData to 0 here, otherwise it will
    // be left pointing at deallocated memory!
    m_pnData = 0;
    m_nLength = 0;
  }

  int& operator[](int nIndex)
  {
    assert(nIndex >= 0 && nIndex < m_nLength);
    return m_pnData[nIndex];
  }

  int GetLength() { return m_nLength; }
};

#endif
```

While this class provides an easy way to create arrays of integers, what if we want to create an array of doubles? Using traditional programming methods, we'd have to create an entirely new class! Here's an example of DoubleArray, an array class used to hold doubles.

```
#ifndef DOUBLEARRAY_H
#define DOUBLEARRAY_H

#include <assert.h> // for assert()

class DoubleArray
{
private:
  int m_nLength;
  double *m_pdData;

public:
  DoubleArray()
  {
    m_nLength = 0;
    m_pdData= 0;
  }

  DoubleArray(int nLength)
  {
    m_pdData= new double[nLength];
    m_nLength = nLength;
  }

  ~DoubleArray()
  {
    delete[] m_pdData;
  }
```

```
  void Erase()
  {
    delete[] m_pdData;
    // We need to make sure we set m_pnData to 0 here, otherwise it will
    // be left pointing at deallocated memory!
    m_pdData= 0;
    m_nLength = 0;
  }

  double& operator[](int nIndex)
  {
    assert(nIndex >= 0 && nIndex < m_nLength);
    return m_pdData[nIndex];
  }

  // The length of the array is always an integer
  // It does not depend on the data type of the array
  int GetLength() { return m_nLength; }
};

#endif
```

Although the code listings are lengthy, you'll note the two classes are almost identical! In fact, the only substantive difference is the contained data type. As you likely have guessed, this is another area where templates can be put to good use to free us from having to create classes that are bound to one specific data type.

Creating template classes is works pretty much identically to creating template functions, so we'll proceed by example. Here's the IntArray classes, templatated version:

```
#ifndef ARRAY_H
#define ARRAY_H

#include <assert.h> // for assert()

template <typename T>
class Array
{
private:
  int m_nLength;
  T *m_ptData;

public:
  Array()
  {
    m_nLength = 0;
    m_ptData = 0;
  }
```

```
    Array(int nLength)
    {
        m_ptData= new T[nLength];
        m_nLength = nLength;
    }

    ~Array()
    {
        delete[] m_ptData;
    }

    void Erase()
    {
        delete[] m_ptData;
        // We need to make sure we set m_pnData to 0 here, otherwise it will
        // be left pointing at deallocated memory!
        m_ptData= 0;
        m_nLength = 0;
    }

    T& operator[](int nIndex)
    {
        assert(nIndex >= 0 && nIndex < m_nLength);
        return m_ptData[nIndex];
    }

    // The length of the array is always an integer
    // It does not depend on the data type of the array
    int GetLength(); // templated GetLength() function defined below
};

template <typename T>
int Array<T>::GetLength() { return m_nLength; }

#endif
```

As you can see, this version is almost identical to the IntArray version, except we've added the template declaration, and changed the contained data type from int to T.

Note that we've also defined the GetLength() function outside of the class declaration. This isn't necessary, but new programmers typically stumble when trying to do this for the first time due to the syntax, so an example is instructive. Each templated member function declared outside the class declaration needs its own template declaration. Also, note that the name of the templated array class is Array<T>, not Array – Array would refer to a non-templated version of a class named Array.

Here's a short example using the above templated array class:

```
int main()
```

```
{
   Array<int> anArray(12);
   Array<double> adArray(12);

   for (int nCount = 0; nCount < 12; nCount++)
   {
      anArray[nCount] = nCount;
      adArray[nCount] = nCount + 0.5;
   }

   for (int nCount = 11; nCount >= 0; nCount----;)
      std::cout << anArray[nCount] << "\t" << adArray[nCount] << std::endl;

   return 0;
}
```

This example prints the following:

```
11    11.5
10    10.5
9     9.5
8     8.5
7     7.5
6     6.5
5     5.5
4     4.5
3     3.5
2     2.5
1     1.5
0     0.5
```

Templated classes are instanced in the same way templated functions are – the compile stencils a copy upon demand with the template parameter replaced by the actual data type the user needs and then compiles the copy. If you don't ever use a template class, the compile won't even compile it.

Template classes are ideal for implementing container classes, because it is highly desirable to have containers work across a wide variety of data types, and templates allow you to do so without duplicating code. Although the syntax is ugly, and the error messages can be cryptic, template classes are truly one of C++'s best and most useful features.

A note for users using older compilers

Some older compilers (eg. Visual Studio 6) have a bug where the definition of template class functions must be put in the same file as the template class is defined in. Thus, if the template class were defined in X.h, the function definitions would have to also go in X.h (not X.cpp). This issue should be fixed in most/all modern compilers.

S.NO	RGPV QUESTION	YEAR	MARKS

Q.1	Explain Template classesin C++.	June2010,June 2012	10

UNIT -2/LECTURE -8

C++ Objects and Classes [RGPV/Dec2010 (7)]

C++ Tutorials

In object-oriented programming languages like C++, the data and functions (procedures to manipulate the data) are bundled together as a self-contained unit called an object. A class is an extended concept similar to that of structure in C programming language; this class describes the data properties alone. In C++ programming language, class describes both the properties (data) and behaviors (functions) of objects. Classes are not objects, but they are used to instantiate objects.

Features of Class:

Classes contain data known as members and member functions. As a unit, the collection of members and member functions is an object. Therefore, this unit of objects makes up a class.

How to write a Class:

In Structure in C programming language, a structure is specified with a name. The C++ programming language extends this concept. A class is specified with a name after the keyword class.

The starting flower brace symbol '{'is placed at the beginning of the code. Following the flower brace symbol, the body of the class is defined with the member functions data. Then the class is closed with a flower brace symbol '}' and concluded with a colon ';'.

```
class exforsys
{
  data;
  member_functions;
  & #46;&#46;&#46;&#46;&#46;&#46;&#46;&#46;&#46;&#46;&#46;&#46;&#46;&#46;&#46;
};
```

There are different access specifiers for defining the data and functions present inside a class.

Access specifiers:

Access specifiers are used to identify access rights for the data and member functions of the class. There are three main types of access specifiers in C++ programming language:

private
public
protected

A private member within a class denotes that only members of the same class have accessibility. The private member is inaccessible from outside the class.

Public members are accessible from outside the class.

A protected access specifier is a stage between private and public access. If member functions defined in a class are protected, they cannot be accessed from outside the class but can be accessed from the derived class.

When defining access specifiers, the programmer must use the keywords: private, public or protected when needed, followed by a semicolon and then define the data and member functions under it.

```
class exforsys
{
  private:
  int x,y;
  public:
  void sum()
```

```
    {
      & #46;&#46;&#46;&#46;
      & #46;&#46;&#46;&#46;
    }
};
```

In the code above, the member x and y are defined as private access. The member function sum is defined as a public access.

General Template of a class:

General structure for defining a class is:

```
class classname
{
  access_specifier:
  data_member;
  member_functions;

  access_specifier:
  data_member;
  member_functions;
};
```

Generally, in class, all members (data) would be declared as private and the member functions would be declared as public. Private is the default access level. If no access specifiers are identified for members of a class, the members are defaulted to private access.

```
class exforsys
{
  int x,y;
  public:
  void sum()
  {
    & #46;&#46;&#46;&#46;
    & #46;&#46;&#46;&#46;
  }
};
```

In this example, for members x and y of the class exforsys there are no access specifiers identified. exforsys would have the default access specifier as private.

Creation of Objects:

Once the class is created, one or more objects can be created from the class as objects are instance of the class.

Just as we declare a variable of data type int as:

int x;

Objects are also declared as:

class_name followed_by object_name;

Example:

exforsys e1;

This declares e1 to be an object of class exforsys.

For example a complete class and object declaration is given below:

```
class exforsys
```

```
{
  private:
  int x,y;
  public:
  void sum()
  {
    & #46;&#46;&#46;&#46;
    & #46;&#46;&#46;&#46;
  }
};

void main()
{
  exforsys e1;
  & #46;&#46;&#46;&#46;
  & #46;&#46;&#46;&#46;
}
<="" p="">
```

For example:

```
class exforsys
{
  private:
  int x,y;
  public:
  void sum()
  {
    & #46;&#46;&#46;&#46;
    & #46;&#46;&#46;&#46;
  }
}e1;
```

S.NO	RGPV QUESTION	YEAR	MARKS
Q.1	Define object & classess in C++.	RGPV Dec 2010	7

Unit-2/ Lecture- 9

Object Oriented Design (OOD) [RGPV/June2010 (10)]

Object Oriented Design is the concept that forces programmers to plan out their code in order to h better flowing program. The origins of object oriented design are debated, but the first languages supported it included Simula and SmallTalk. The term did not become popular until Grady Booch v the first paper titled Object-Oriented Design, in 1982.
Object Oriented Design is defined as a programming language that has 5 conceptual tools to ai programmer. These programs are often more readable than non-object oriented programs, debugging becomes easier with locality.

Language Concepts
The 5 Basic Concepts of Object Oriented Design are the implementation level features that are buil the programming language. These features are often referred to by these common names:

Encapsulation-A tight coupling or association of data structures with the methods or functions tha on the data. This is called a class, or object (an object is often the implementation of a class).

Data Protection -The ability to protect some components of the object from external entities. T realized by language keywords to enable a variable to be declared as private or protected to the ov class.

Inheritance -The ability for a class to extend or override functionality of another class. The so called class has a whole section that is the parent class and then it has it's own set of functions and data.

Interface -A definition of functions or methods, and their signatures that are available for u manipulate a given instance of an object.

Polymorphism -The ability to define different functions or classes as having the same name but t different data types.

Programming Concepts
There are several concepts that were derived from the new languages once they became popular new standards that came around pushed on three major things:

Re-usability-The ability to reuse code for multiple applications. If a programmer has already writ power function, then it should be written that any program can make a call to that function and it sh work exactly the same.

Privacy -This is important for large programs and preventing loss of data.

Documentation -The commenting of a program in mark up that will not be converted to machine This mark up can be as long as the programmer wants, and allows for comprehensive information passed on to new programmers. This is important for both the re-usability and the maintainabil programs.

S.NO	RGPV QUESTION	YEAR	MARKS
Q.1	Explain Object oriented design in C++.	June2010	10