Object Oriented Programming - Lecture 5

Diana Borza - diana.borza@ubbcluj.ro

March 29, 2023

Content

- OOP features (cont'd)
- Polymorphism
- Virtual methods
- Upcasting and downcasting
- Abstract classses
- Multiple Inheritance
- What's new in C++ (cont'd) - auto keyword

OOP - features

- **Polymorphism**: allows an object to be one of several types, and determining at runtime how to "process" it, based on its type.
- **Abstraction**: separating an object's specification from its implementation.
- **Inheritance**: organize classes to be arranged in a hierarchy that represents "IS A" relationships $\rightarrow$ easy re-use of the code, in addition to potentially mirroring real-world relationships in an intuitive way.
- **Encapsulation**: binds together the data and functions that manipulate the data, and that keeps both safe from outside interference and misuse.

Polymorphism - definitions

- Etimology: From Ancient Greek (polús, "many, much") + (morphe, "form, shape") $\rightarrow$ the ability to take multiple forms;
- Polymorphism is the property of an entity to react differently depending on its type.
- Polymorphism is the property that allows different entities to behave in different ways to the same action.
- Polymorphism allows different objects to respond in different ways to the same message.

Polymorphism

- Usually, polymorphism occurs when classes are related with each other through inheritance.
- A call to a member function will cause the execution of a different code, depending on the type of object that invokes the function.
- The code to be executed is determined dynamically, at run time. The decision is based on the actual object.

Polymorphism

Polymorphism

- C++ allows us to set a Base pointer or reference to a Derived object
- Advantages:
  - containers of pointers to base classes to store objects of different derived classes;
  - functions that operate on any of the derived class.
- "Disadvantage": they can only see members of Base (or any classes that Base inherited).

Virtual functions I

- A **virtual function** is a special type of function that, when called, resolves to the most-derived version of the function that exists between the base and derived class.
- A derived function is considered a match if **it has the same signature** (name, parameter types, and whether it is const) and return type as the base version of the function.
- To mark a function as virtual, you need to specify the keyword virtual before its **declaration**: virtual function_signature;
- Examples:
  - virtual void print();
  - virtual int getAge();
  - virtual std::string getName();

Virtual functions II

- The function in the derived class that is overriding the function in the base class can use the override specifier to ensure that the function is overriding a virtual function from the base class.
- Examples:
  - void print() override;
  - int getAge() override;
- override is an identifier with a special meaning when used after member function declarations and otherwise, it is not a reserved keyword.
- if the function does not override a base class function (or is applied to a non-virtual function), the compiler will rise a compile error.
  - **Exception: covariant return types**: If the return type of a virtual function is a pointer or a reference to a class, override functions can return a pointer or a reference to a derived class

Virtual functions III

- When dealing with inheritance you should mark your destructors as virtual
  - In the case of a derived class object, it is essential that both the destructor of the base class and the destructor of the derived class are called.
  - In the case of a derived class object, it is essential that both the destructor of the base class and the destructor of the derived class are called.
- **Good practices**:
  - If you intend your class to be inherited from, make sure your destructor is virtual.
  - If you do not intend your class to be inherited from, mark your class as final. This will prevent other classes from inheriting from it in the first place, without imposing any other use restrictions on the class itself.

- **Ignoring virtualization**: simply use the scope resolution operator to call the function from the base class.

Virtual functions IV

- **Constructors cannot be virtual.**
- When creating an object, one must know exactly what type of object one is creating.
- Usually, the virtual table pointer is initialized in the constructor.

Early and late binding

- **Early/static binding**
  - the compiler (or linker) is able to directly associate the identifier name (such as a function or variable name) with a machine address.
  - The choice of which function to call is made at **compile time**.

- **Late/dynamic binding**
  - in some cases it is not possible to know which function will be called until runtime (when the program is run):
    - **pointers to functions**;
    - **virtual functions**: only at runtime, the compiler takes the decision of which function body to execute, according to the actual type of the object.
  - Dynamic binding only works with non-value types: i.e. **references and pointers**.

The virtual table I

- To implement virtual functions, C++ uses a special form of late binding known as the **virtual table** ("vtable", "virtual function table", "virtual method table" or "dispatch table").
- The virtual table is a static array set up by the compiler at compile time.
- The virtual table is a lookup table of functions used to resolve function calls in a dynamic/late binding manner.

The virtual table II

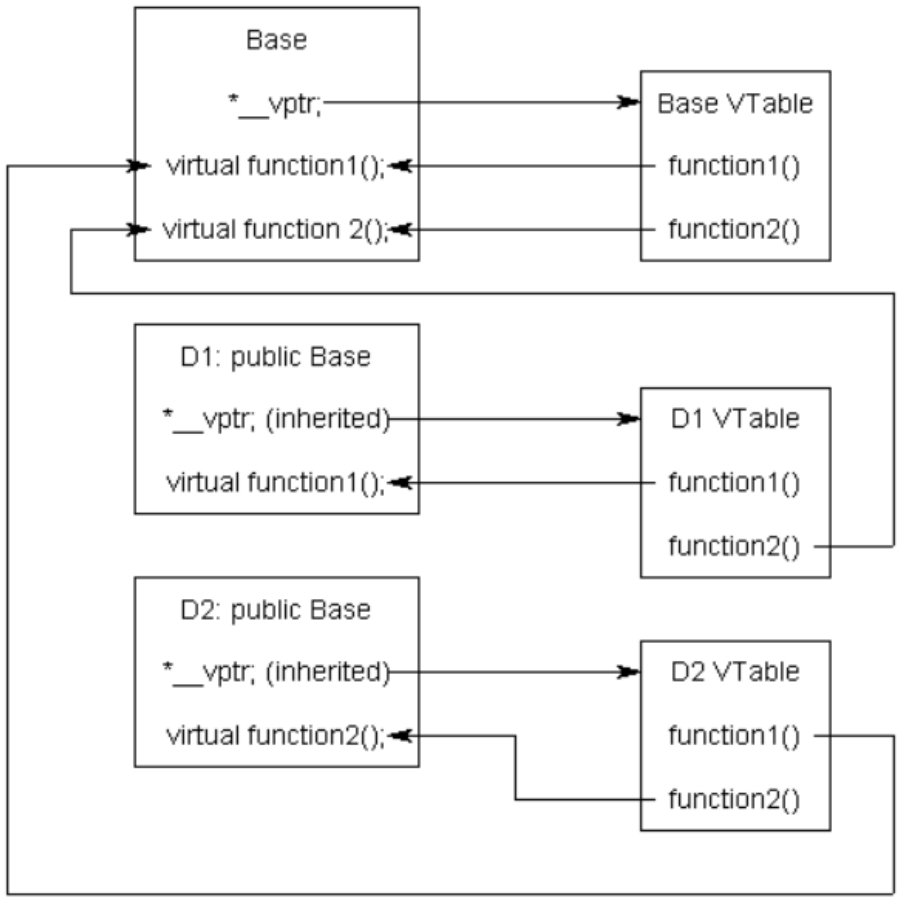
- Each class that uses virtual functions (or is derived from a class that uses virtual functions) is given its own virtual table. Each virtual function corresponds to an entry in the virtual table.
- Each entry in the virtual table is simply a function pointer that points to the most-derived function accessible by that class.
- The compiler also adds a hidden pointer to the base class: vptr
- A pointer to the virtual table (vptr) is added to the base class - and inherited by the derived classes.
- When a class object is created, the pointer to the virtual table is set to point to the virtual table for that class. When a call is made through a pointer or reference, the compiler generates code that dereferences the pointer to the object‘s virtual table and makes an indirect call using the address of a member function stored in the table.

The virtual table III

```
class Base
{
public:
    virtual void function1() {};
    virtual void function2() {};
};

class D1: public Base
{
public:
    virtual void function1() {};
};

class D2: public Base
{
public:
    virtual void function2() {};
};
```

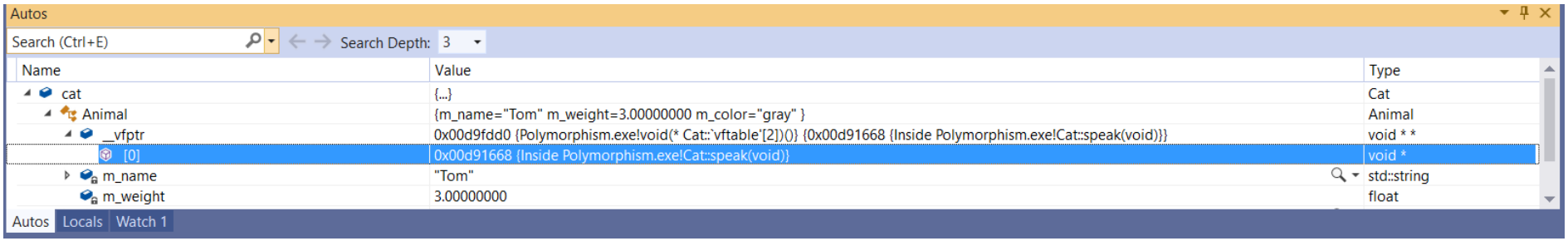


The virtual table IV

- The virtual function mechanism works only with pointers and references, but not with value-types (objects).
- Calling a virtual function is slower than calling a non-virtual function:
  1. Use the vptr to access the correct virtual table.
  2. Find the correct function to call in the virtual table.
  3. Call the function.
- Declare functions as virtual only if necessary.

The virtual table V

```
Cat cat{ "Tom" , 3, "gray"};
std::cout << "cat is named " << cat.getName() << ", and it says " << cat.speak() << '\n';
```



Object slicing

- Polymorphism only works with pointers and references.
- **When we assign a Derived object to a Base object, only the Base portion of the Derived object is copied! (the Derived portion is not!)**
- The assignment of a Derived class object to a Base class object is called **object slicing**.

Dynamic cast

- C++ provides a casting operator called dynamic_cast, which can be used in cases where you have a pointer to a base class, but you want to access some information that exists only in a derived class.
- Syntax: dynamic_cast<Derived_Type>(object);
- **Rule: Always ensure your dynamic casts actually succeeded by checking for a null pointer result.**

Pure virtual functions

- A **pure** virtual function (or abstract function) is a virtual function has no body at all.
- A pure virtual function simply acts as a *placeholder* that is meant to be redefined by derived classes.
- **Example**:
  - virtual int getValue() = **0**; // a pure virtual function

Abstract classes I

- Any class with one or more pure virtual functions becomes an **abstract base class**.
- Abstract classes **cannot** be instantiated!
- Any class derived from an abstract class must define a body for the virtual functions, or that derived class will be considered an abstract base class as well.

Interface classes

- **An interface class** is a class that has no member variables, and where all of the functions are pure virtual.
- Other languages (like Java or C#) take this concepts even further and also include the *interface* type.
- They are useful when we want to define the functionality that derived classes must implement, but leave the details of how the derived class implements that functionality entirely up to the derived class.
- Abstract clases also have virtual tables.
  - The virtual table entry for a pure virtual function contains either a null pointer, or point to a generic function that prints an error (sometimes this function is named __purecall) if no override is provided.

UML

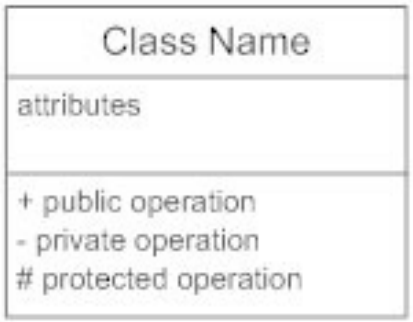
- **Unified Modelling Language (UML)** - general-purpose, developmental, modeling language which provides a standard way to visualize the design of a system;
- developed by Grady Booch, Ivar Jacobson and James Rumbaugh in 1994-1995;
- in 1997, UML was adopted as a standard by the Object Management Group (OMG);
- in 2005 published by the International Organization for Standardization (ISO) as an approved ISO standard;
- the software project is visualized using a collection of **diagrams**.

UML - diagram types

- *Structural diagrams*: represent the structure of the program; what must be present in the system being modeled?
  - Class diagram
  - Package diagram
  - Object diagram
  - Component diagram
  - Composite structure diagram
  - Deployment diagram
- *Behavioral/interaction diagrams*: illustrate the behavior/functionality of a system; what must happen in the system?
  - Activity diagram
  - Sequence diagram
  - Use case diagram
  - State diagram
  - Communication diagram
  - Interaction overview diagram
  - Timing diagram

UML - class diagrams I

- classes are represented with rectangles divided into 3 compartments:
  - first partition: name of the class (centered, bold)
  - second partition: attributes (left aligned)
  - third partition: methods (left aligned)



Visibility

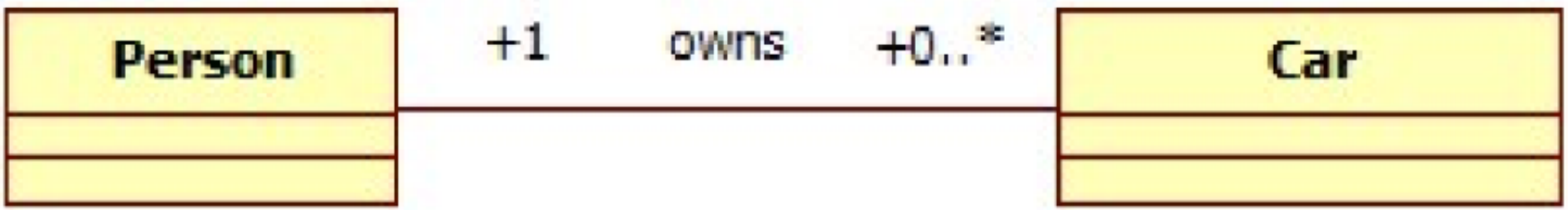
Marker	Visibility
+	public
-	private
#	protected

UML - associations

- **UML associations** describe relationships of structural dependency between classes.
- An association may have:
  - a role name;
  - a multiplicity;
  - navigability (uni/bi-directional).

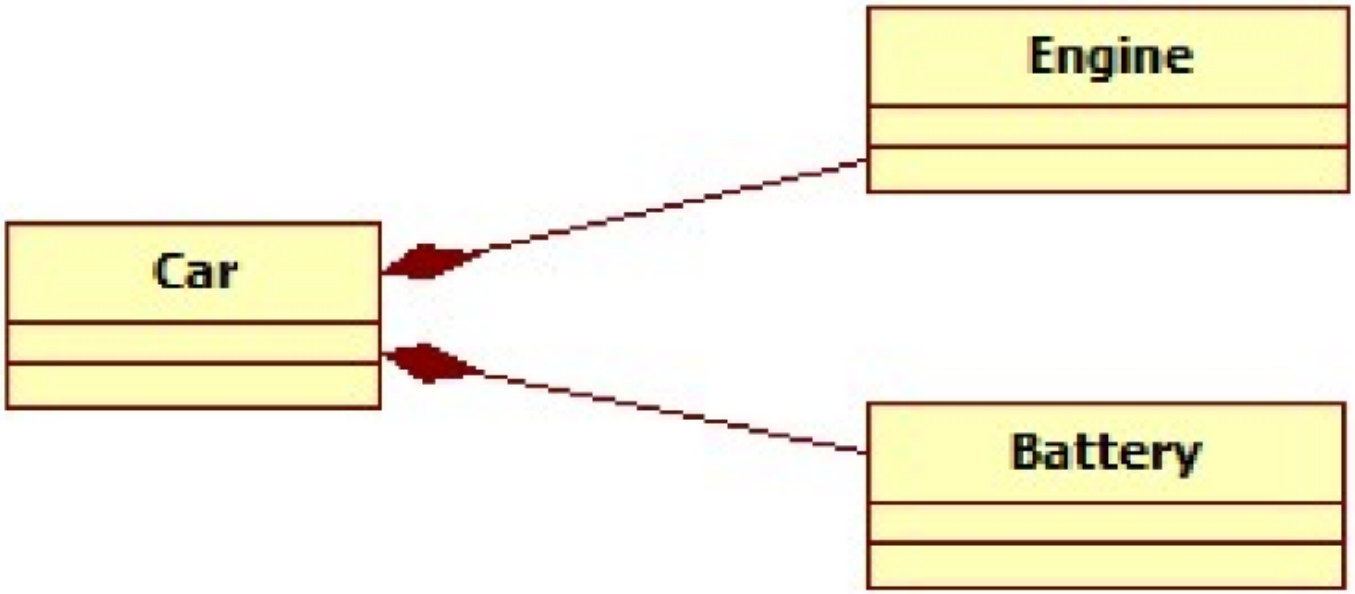
UML - association types I

- **Association** (*knows a*) - is a reference based relationship between two classes. A class *A* holds a class level reference to another class *B*;
- shows that instances of classes could be either linked to each other or combined logically or physically into some aggregation.



UML - association types II

- **Composition/composite aggregation** - (*has a*) - when class B is composed by class A, class A instance owns the creation or controls lifetime of instance of class B. When class A instance is destructed, so is the class B instance;
- represents a part-whole relationship;
- a part could be included in at most one composite (whole) at a time.

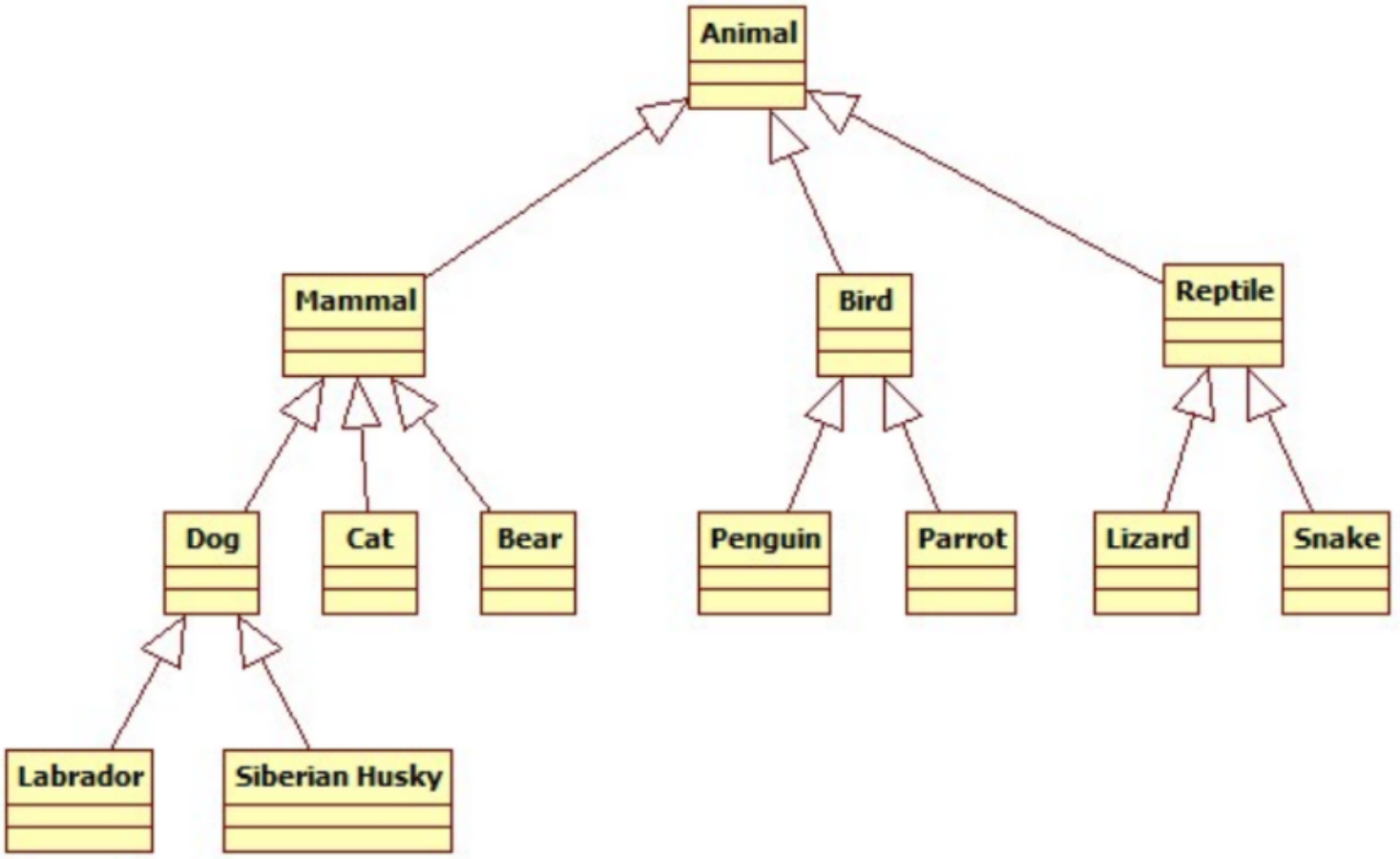


UML - association types III

- **Dependency** (*uses a*) - when class *A* uses a reference to class *B*, as part of a particular method (parameter or local variable). A modification to the class *B*'s interface may influence class *A*;
- it is called also called *supplier - client* relationship, where supplier provides something to the client;
- shows that an element requires, needs or depends on other elements for specification or implementation.

UML - association types IV

- **Specialization/inheritance** (*is a*) - every instance of the derived class is an instance of the base class;
- inheritance allows us to create hierarchies of classes.



What's new in C++ - auto keyword

- When initializing a variable, the auto keyword can be used in place of the type to tell the compiler to infer the variable's type from the initializer's type.
- This is called **type inference/type deduction**.
- Examples:
  - auto d{ 25.0 }; // 25.0 is a double literal, so d will be type double
  - auto i{ 1 + 2 }; // 1 + 2 evaluates to an int, so i will be type int
- Since C++14, the auto keyword was extended to be able to deduce a function's return type from return statements in the function body.
- Trailing return type syntax: the return type is specified after the function signature.

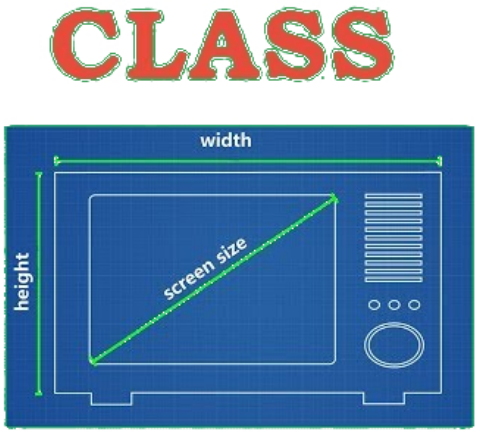
Summary I

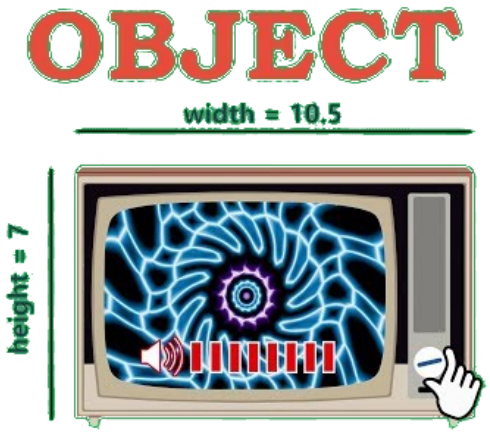
- A virtual function is a special type of function that resolves to the most-derived version of the function (called an override) that exists between the base and derived class.
- A function that is intended to be an override should use the override specifier to ensure that it is actually an override.
- A virtual function can be made pure virtual/abstract by adding "= 0" to the end of the virtual function prototype.
- A class containing a pure virtual function is called an abstract class, and can not be instantiated.
- An interface class is one with no member variables and all functions are pure virtual functions.

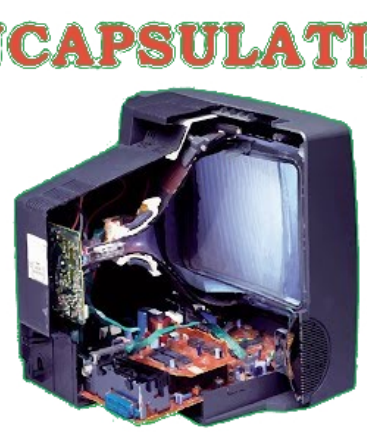
Summary II

- Early binding occurs when the compiler encounters a direct function call. The compiler or linker can resolve these function calls directly.
- Late binding occurs when a function pointer is called. In these cases, which function will be called can not be resolved until runtime.
- Dynamic casting can be used to convert a pointer to a base class object into a pointer to a derived class object. This is called *downcasting*.

Summary III


CLASS
width
height
screen size


OBJECT
width = 10.5
height = 7


ENCAPSULATION


OBJECT ORIENTED PROGRAMMING
INHERITANCE
ABSTRACTION
POLYMORPHISM
InvolveInInnovation

Next time

- c++ standard library
- templates