ACA Summer School 2014
Advanced C++

Pankaj Prateek

ACA, CSE, IIT Kanpur

June 27, 2014

Pointer: Basics

- Variable

  Name/Identifier $\leftrightarrow$ Contains a value $\leftrightarrow$ Memory address

- Memory cells are numbered in continuation, giving every cell a unique number, which is called its address
- Pointer

  Name/Identifier $\leftrightarrow$ Contains memory address of another variable

- Pointers contain the address of some other variable. They are said to "point to" that variable

Pointer: Basics

- Variable

  Name/Identifier $\leftrightarrow$ Contains a value $\leftrightarrow$ Memory address

- Memory cells are numbered in continuation, giving every cell a unique number, which is called its address
- Pointer

  Name/Identifier $\leftrightarrow$ Contains memory address of another variable

- Pointers contain the address of some other variable. They are said to "point to" that variable

Pointer: Basics

- Variable

  Name/Identifier $\leftrightarrow$ Contains a value $\leftrightarrow$ Memory address

- Memory cells are numbered in continuation, giving every cell a unique number, which is called its address
- Pointer

  Name/Identifier $\leftrightarrow$ Contains memory address of another variable

- Pointers contain the address of some other variable. They are said to “point to” that variable

Pointer: Basics

- Variable

  Name/Identifier $\leftrightarrow$ Contains a value $\leftrightarrow$ Memory address
- Memory cells are numbered in continuation, giving every cell a unique number, which is called its address
- Pointer

  Name/Identifier $\leftrightarrow$ Contains memory address of another variable
- Pointers contain the address of some other variable. They are said to "point to" that variable

Pointer: Basics

```
int a;
int *ptr = &a; //Reference operator
a = 10;
*ptr = 12;    //Dereference operator
```

Pointer: Basics

- Pointers are type-specific (that is there is a different pointer for every different type of variable)
- Pointer to a pointer is also possible (double reference)
- Extremely useful in call by reference mechanism
- Also useful in dynamic memory allocation
- Void pointer : a type-less pointer, but cannot be dereferenced without explicit type cast
- Null pointer : A pointer that points to nothing or 0. Cannot be derefrenced. Doing that raises a run-time error/segmentation fault
- Equivalent array names

Pointer: Basics

- Pointers are type-specific (that is there is a different pointer for every different type of variable)
- Pointer to a pointer is also possible (double reference)
- Extremely useful in call by reference mechanism
- Also useful in dynamic memory allocation
- Void pointer : a type-less pointer, but cannot be dereferenced without explicit type cast
- Null pointer : A pointer that points to nothing or 0. Cannot be derefrenced. Doing that raises a run-time error/segmentation fault
- Equivalent array names

Pointer: Basics

- Pointers are type-specific (that is there is a different pointer for every different type of variable)
- Pointer to a pointer is also possible (double reference)
- Extremely useful in call by reference mechanism
- Also useful in dynamic memory allocation
- Void pointer : a type-less pointer, but cannot be dereferenced without explicit type cast
- Null pointer : A pointer that points to nothing or 0. Cannot be derefrenced. Doing that raises a run-time error/segmentation fault
- Equivalent array names

Pointer: Basics

- Pointers are type-specific (that is there is a different pointer for every different type of variable)
- Pointer to a pointer is also possible (double reference)
- Extremely useful in call by reference mechanism
- Also useful in dynamic memory allocation
- Void pointer : a type-less pointer, but cannot be dereferenced without explicit type cast
- Null pointer : A pointer that points to nothing or 0. Cannot be derefrenced. Doing that raises a run-time error/segmentation fault
- Equivalent array names

Pointer: Basics

- Pointers are type-specific (that is there is a different pointer for every different type of variable)
- Pointer to a pointer is also possible (double reference)
- Extremely useful in call by reference mechanism
- Also useful in dynamic memory allocation
- Void pointer : a type-less pointer, but cannot be dereferenced without explicit type cast
- Null pointer : A pointer that points to nothing or 0. Cannot be derefrenced. Doing that raises a run-time error/segmentation fault
- Equivalent array names

Pointer: Basics

- Pointers are type-specific (that is there is a different pointer for every different type of variable)
- Pointer to a pointer is also possible (double reference)
- Extremely useful in call by reference mechanism
- Also useful in dynamic memory allocation
- Void pointer : a type-less pointer, but cannot be dereferenced without explicit type cast
- Null pointer : A pointer that points to nothing or 0. Cannot be derefrenced. Doing that raises a run-time error/segmentation fault
- Equivalent array names

Pointer: Basics

- Pointers are type-specific (that is there is a different pointer for every different type of variable)
- Pointer to a pointer is also possible (double reference)
- Extremely useful in call by reference mechanism
- Also useful in dynamic memory allocation
- Void pointer : a type-less pointer, but cannot be dereferenced without explicit type cast
- Null pointer : A pointer that points to nothing or 0. Cannot be derefrenced. Doing that raises a run-time error/segmentation fault
- Equivalent array names

Pointers to structures

```
struct temp {
  int a; float b;
};
int main() {
  struct temp obj1;
  struct temp * ptr;
  obj1.a = 10;
  obj1.b = 3.14;
  ptr = &obj1;
  cout << ptr -> a << endl;
  cout << (*ptr).b << endl;
}
```

Pointer Arithmetic

- ++ and −− operators are allowed on pointers
- They correspond to advancing and retreating the pointer by the size of one object in memory
- + and - are also allowed on pointers, which advance or retreat them suitably
- *(a + 10) is equivalent to a[10] while (a + 10) is equivalent to &a[10]
- Should be used with caution. Can possibly be the result of array out of bounds errors, resulting in segmentation faults
- Array names are treated as constant pointers. Operations like A = A+1 are invalid for them

Pointer Arithmetic

- ++ and −− operators are allowed on pointers
- They correspond to advancing and retreating the pointer by the size of one object in memory
- + and - are also allowed on pointers, which advance or retreat them suitably
- *(a + 10) is equivalent to a[10] while (a + 10) is equivalent to &a[10]
- Should be used with caution. Can possibly be the result of array out of bounds errors, resulting in segmentation faults
- Array names are treated as constant pointers. Operations like A = A+1 are invalid for them

Pointer Arithmetic

- ++ and −− operators are allowed on pointers
- They correspond to advancing and retreating the pointer by the size of one object in memory
- + and - are also allowed on pointers, which advance or retreat them suitably
- *(a + 10) is equivalent to a[10] while (a + 10) is equivalent to &a[10]
- Should be used with caution. Can possibly be the result of array out of bounds errors, resulting in segmentation faults
- Array names are treated as constant pointers. Operations like A = A+1 are invalid for them

Pointer Arithmetic

- ++ and −− operators are allowed on pointers
- They correspond to advancing and retreating the pointer by the size of one object in memory
- + and - are also allowed on pointers, which advance or retreat them suitably
- *(a + 10) is equivalent to a[10] while (a + 10) is equivalent to &a[10]
- Should be used with caution. Can possibly be the result of array out of bounds errors, resulting in segmentation faults
- Array names are treated as constant pointers. Operations like A = A+1 are invalid for them

Pointer Arithmetic

- ++ and −− operators are allowed on pointers
- They correspond to advancing and retreating the pointer by the size of one object in memory
- + and - are also allowed on pointers, which advance or retreat them suitably
- *(a + 10) is equivalent to a[10] while (a + 10) is equivalent to &a[10]
- Should be used with caution. Can possibly be the result of array out of bounds errors, resulting in segmentation faults
- Array names are treated as constant pointers. Operations like A = A+1 are invalid for them

Pointer Arithmetic

- ++ and −− operators are allowed on pointers
- They correspond to advancing and retreating the pointer by the size of one object in memory
- + and - are also allowed on pointers, which advance or retreat them suitably
- *(a + 10) is equivalent to a[10] while (a + 10) is equivalent to &a[10]
- Should be used with caution. Can possibly be the result of array out of bounds errors, resulting in segmentation faults
- Array names are treated as constant pointers. Operations like A = A+1 are invalid for them

Pointer: Basics

- References are alias (second name) to variables
- They do not contain memory address like pointers do
- Unlike pointers, once assigned, then cannot be changed later
- There are no Null references. They have to be initialized when they are declared.
- The actual pass by reference is implemented through them, passing pointers is still only pass by value

Pointer: Basics

- References are alias (second name) to variables
- They do not contain memory address like pointers do
- Unlike pointers, once assigned, then cannot be changed later
- There are no Null references. They have to be initialized when they are declared.
- The actual pass by reference is implemented through them, passing pointers is still only pass by value

Pointer: Basics

- References are alias (second name) to variables
- They do not contain memory address like pointers do
- Unlike pointers, once assigned, then cannot be changed later
- There are no Null references. They have to be initialized when they are declared.
- The actual pass by reference is implemented through them, passing pointers is still only pass by value

Pointer: Basics

- References are alias (second name) to variables
- They do not contain memory address like pointers do
- Unlike pointers, once assigned, then cannot be changed later
- There are no Null references. They have to be initialized when they are declared.
- The actual pass by reference is implemented through them, passing pointers is still only pass by value

Pointer: Basics

- References are alias (second name) to variables
- They do not contain memory address like pointers do
- Unlike pointers, once assigned, then cannot be changed later
- There are no Null references. They have to be initialized when they are declared.
- The actual pass by reference is implemented through them, passing pointers is still only pass by value

Pointers to Objects

```
class ABC {
  int var1;
public:
  setVar(int a);
  int getVar();
};
ABC obj1;
ABC* ptr1;
ptr1 = &obj1;
ptr1->setVar(20);
cout<<ptr1->getVar();
```

Pointers to Objects

- Obviously, private data members cannot be accessed through pointers.
- Way to access the public members is exactly like that in structures
- The type checking of pointers is slightly loose in some sense when it comes to classes

Pointers to Objects

- Obviously, private data members cannot be accessed through pointers.
- Way to access the public members is exactly like that in structures
- The type checking of pointers is slightly loose in some sense when it comes to classes

Pointers to Objects

- Obviously, private data members cannot be accessed through pointers.
- Way to access the public members is exactly like that in structures
- The type checking of pointers is slightly loose in some sense when it comes to classes

Pointers to Objects of derived class

```
class ABC {
  int var1;
public:
  setVar(int a);
  int getVar();
};
class XYZ: public ABC {
int var2;
public:
  void setVar2(int b);
  int getVar2();
};
ABC obj1;
ABC* baseptr;
XYZ obj2;
baseptr = &obj1;
baseptr = &obj2;
```

Pointers to Objects of derived class

- Derived class object also contains a part of itself as the Base class
- Hence, a pointer to base class can also point to an object of the derived class
- But such a pointer can point only to the part of the derived class which houses the base class
- To access member of the derived class, we need an explicit pointer to the derived class
- Even when using a base class pointer for a derived class object, only the inherited members are accessible and not the whole class

Pointers to Objects of derived class

- Derived class object also contains a part of itself as the Base class
- Hence, a pointer to base class can also point to an object of the derived class
- But such a pointer can point only to the part of the derived class which houses the base class
- To access member of the derived class, we need an explicit pointer to the derived class
- Even when using a base class pointer for a derived class object, only the inherited members are accessible and not the whole class

Pointers to Objects of derived class

- Derived class object also contains a part of itself as the Base class
- Hence, a pointer to base class can also point to an object of the derived class
- But such a pointer can point only to the part of the derived class which houses the base class
- To access member of the derived class, we need an explicit pointer to the derived class
- Even when using a base class pointer for a derived class object, only the inherited members are accessible and not the whole class

Pointers to Objects of derived class

- Derived class object also contains a part of itself as the Base class
- Hence, a pointer to base class can also point to an object of the derived class
- But such a pointer can point only to the part of the derived class which houses the base class
- To access member of the derived class, we need an explicit pointer to the derived class
- Even when using a base class pointer for a derived class object, only the inherited members are accessible and not the whole class

Pointers to Objects of derived class

- Derived class object also contains a part of itself as the Base class
- Hence, a pointer to base class can also point to an object of the derived class
- But such a pointer can point only to the part of the derived class which houses the base class
- To access member of the derived class, we need an explicit pointer to the derived class
- Even when using a base class pointer for a derived class object, only the inherited members are accessible and not the whole class

The this pointer (revisited)

- Suppose there is a class A, with a member function f()
- When f() is called via some object of the class A, then inside body of f(), the keyword "this" stores the address of that object
- The this pointer is passed as a hidden argument to all nonstatic member function calls
- The this pointer is available as a "local" variable within the body of all nonstatic functions.
- It is used to reference any member of the class if it is hidden due to some other local variable with same name

The this pointer (revisited)

- Suppose there is a class A, with a member function f()
- When f() is called via some object of the class A, then inside body of f(), the keyword "this" stores the address of that object
- The this pointer is passed as a hidden argument to all nonstatic member function calls
- The this pointer is available as a "local" variable within the body of all nonstatic functions.
- It is used to reference any member of the class if it is hidden due to some other local variable with same name

The this pointer (revisited)

- Suppose there is a class A, with a member function f()
- When f() is called via some object of the class A, then inside body of f(), the keyword "this" stores the address of that object
- The this pointer is passed as a hidden argument to all nonstatic member function calls
- The this pointer is available as a "local" variable within the body of all nonstatic functions.
- It is used to reference any member of the class if it is hidden due to some other local variable with same name

The this pointer (revisited)

- Suppose there is a class A, with a member function f()
- When f() is called via some object of the class A, then inside body of f(), the keyword "this" stores the address of that object
- The this pointer is passed as a hidden argument to all nonstatic member function calls
- The this pointer is available as a "local" variable within the body of all nonstatic functions.
- It is used to reference any member of the class if it is hidden due to some other local variable with same name

The this pointer (revisited)

- Suppose there is a class A, with a member function f()
- When f() is called via some object of the class A, then inside body of f(), the keyword "this" stores the address of that object
- The this pointer is passed as a hidden argument to all nonstatic member function calls
- The this pointer is available as a "local" variable within the body of all nonstatic functions.
- It is used to reference any member of the class if it is hidden due to some other local variable with same name

Virtual

- What we know : Its used to avoid ambiguity while inheriting from multiple classes
- New thing : its applicable not only in inheritance, but also to indivisual class members
- A member of a class that can be re-defined in its derived classes is called a virtual member
- In order to declare a member of a class as virtual, we must precede its declaration with the keyword virtual

Virtual

- What we know : Its used to avoid ambiguity while inheriting from multiple classes
- New thing : its applicable not only in inheritance, but also to indivisual class members
- A member of a class that can be re-defined in its derived classes is called a virtual member
- In order to declare a member of a class as virtual, we must precede its declaration with the keyword virtual

Virtual

- What we know : Its used to avoid ambiguity while inheriting from multiple classes
- New thing : its applicable not only in inheritance, but also to indivisual class members
- A member of a class that can be re-defined in its derived classes is called a virtual member
- In order to declare a member of a class as virtual, we must precede its declaration with the keyword virtual

Virtual

- What we know : Its used to avoid ambiguity while inheriting from multiple classes
- New thing : its applicable not only in inheritance, but also to indivisual class members
- A member of a class that can be re-defined in its derived classes is called a virtual member
- In order to declare a member of a class as virtual, we must precede its declaration with the keyword virtual

Virtual

```
class Polygon {
protected:
  int width, height;
public:
  void setVal(int w, int h);
  virtual float area();
};
```

- The area function can be "redefined" in the classes derived from polyon
- Exactly same function signature is needed in that case

Virtual

```
class Polygon {
protected:
  int width, height;
public:
  void setVal(int w, int h);
  virtual float area();
};
```

- The area function can be "redefined" in the classes derived from polyon
- Exactly same function signature is needed in that case

Virtual

```
class Polygon {
protected:
  int width, height;
public:
  void setVal(int w, int h);
  virtual float area();
};
```

- The area function can be “redefined” in the classes derived from polyon
- Exactly same function signature is needed in that case

Virtual

```
class Polygon {
protected:
  int width, height;
public:
  void setVal(int w, int h) {
    width = w; height = h;
  }
  virtual float area() {return 0;}
};
class Rect: public Polygon {
public:
  float area() {return width*height;};
};
class Triangle: public Polygon {
public:
  float area() {return 0.5*width*height;};
};
```

Pure Virtual Function

- A function with the virtual keyword is called a virtual function
- A function declared virtual can be kept undefined
- Such undefined virtual functions are called “Pure virtual functions

```
virtual float area() = 0;
```

The zero at the end tells the compiler that area() is a pure virtual function

Pure Virtual Function

- A function with the virtual keyword is called a virtual function
- A function declared virtual can be kept undefined
- Such undefined virtual functions are called “Pure virtual functions

```
virtual float area() = 0;
```

The zero at the end tells the compiler that area() is a pure virtual function

Pure Virtual Function

- A function with the virtual keyword is called a virtual function
- A function declared virtual can be kept undefined
- Such undefined virtual functions are called “Pure virtual functions

```
virtual float area() = 0;
```

The zero at the end tells the compiler that area() is a pure virtual function

Pure Virtual Function

- A function with the virtual keyword is called a virtual function
- A function declared virtual can be kept undefined
- Such undefined virtual functions are called "Pure virtual functions

```
virtual float area() = 0;
```

The zero at the end tells the compiler that area() is a pure virtual function

Pure Virtual Function

- A function with the virtual keyword is called a virtual function
- A function declared virtual can be kept undefined
- Such undefined virtual functions are called “Pure virtual functions

  ```
  virtual float area() = 0;
  ```

  The zero at the end tells the compiler that area() is a pure virtual function

Abstract Class: Concept

- Its a program design concept
- A class which is designed to be specifically used as a base class
- No concrete objects are created out of it.
- It only represents the common basic properties of the derived classes
- It has at least one pure virtual function. (explained later)
- Any class derived from an abstract class will also be an abstract class if the virtual function is not defined.

Abstract Class: Concept

- Its a program design concept
- A class which is designed to be specifically used as a base class
- No concrete objects are created out of it.
- It only represents the common basic properties of the derived classes
- It has at least one pure virtual function. (explained later)
- Any class derived from an abstract class will also be an abstract class if the virtual function is not defined.

Abstract Class: Concept

- Its a program design concept
- A class which is designed to be specifically used as a base class
- No concrete objects are created out of it.
- It only represents the common basic properties of the derived classes
- It has at least one pure virtual function. (explained later)
- Any class derived from an abstract class will also be an abstract class if the virtual function is not defined.

Abstract Class: Concept

- Its a program design concept
- A class which is designed to be specifically used as a base class
- No concrete objects are created out of it.
- It only represents the common basic properties of the derived classes
- It has at least one pure virtual function. (explained later)
- Any class derived from an abstract class will also be an abstract class if the virtual function is not defined.

Abstract Class: Concept

- Its a program design concept
- A class which is designed to be specifically used as a base class
- No concrete objects are created out of it.
- It only represents the common basic properties of the derived classes
- It has at least one pure virtual function. (explained later)
- Any class derived from an abstract class will also be an abstract class if the virtual function is not defined.

Abstract Class: Concept

- Its a program design concept
- A class which is designed to be specifically used as a base class
- No concrete objects are created out of it.
- It only represents the common basic properties of the derived classes
- It has at least one pure virtual function. (explained later)
- Any class derived from an abstract class will also be an abstract class if the virtual function is not defined.

Abstract Class

- Earlier, abstract class was that class which was inherited virtually
- Now : Abstract class is a class with at least one pure virtual function

```
class Polygon {
protected:
  int width, height;
public:
  void setVal(int w, int h);
  virtual float area() = 0;
};
```

Abstract Class

- Earlier, abstract class was that class which was inherited virtually
- Now : Abstract class is a class with at least one pure virtual function

```
class Polygon {
protected:
  int width, height;
public:
  void setVal(int w, int h);
  virtual float area() = 0;
};
```

Abstract Class

- Earlier, abstract class was that class which was inherited virtually
- Now : Abstract class is a class with at least one pure virtual function

```
class Polygon {
protected:
  int width, height;
public:
  void setVal(int w, int h);
  virtual float area() = 0;
};
```

Abstract Class

- There is at least one function without a definition. Hence no objects possible. Abstract class cannot be instantiated
- However, we can create a pointer for an abstract class !!
- Pointer to abstract class can still be used to point to objects of other classes

Abstract Class

- There is at least one function without a definition. Hence no objects possible. Abstract class cannot be instantiated
- However, we can create a pointer for an abstract class !!
- Pointer to abstract class can still be used to point to objects of other classes

Abstract Class

- There is at least one function without a definition. Hence no objects possible. Abstract class cannot be instantiated
- However, we can create a pointer for an abstract class !!
- Pointer to abstract class can still be used to point to objects of other classes

Polymorphism

- Pointer to a derived class is type-compatible with a pointer to its base class.
- Polymorphism is the art of taking advantage of this simple but powerful and versatile feature
- Use the pointer to abstract base class
- Dynamically manipulate the objects of one of the derived classes

Polymorphism

- Pointer to a derived class is type-compatible with a pointer to its base class.
- Polymorphism is the art of taking advantage of this simple but powerful and versatile feature
- Use the pointer to abstract base class
- Dynamically manipulate the objects of one of the derived classes

Polymorphism

- Pointer to a derived class is type-compatible with a pointer to its base class.
- Polymorphism is the art of taking advantage of this simple but powerful and versatile feature
- Use the pointer to abstract base class
- Dynamically manipulate the objects of one of the derived classes

Polymorphism

- Pointer to a derived class is type-compatible with a pointer to its base class.
- Polymorphism is the art of taking advantage of this simple but powerful and versatile feature
- Use the pointer to abstract base class
- Dynamically manipulate the objects of one of the derived classes

Polymorphism: The CS example

- Every player has some weapon to kill the enemy
- Weapon can be used by left-click or right-click of the mouse
- All weapons do the same function on moving the mouse
- But each weapon has its own unique function on clicking the mouse
- Base class : Weapon
- Derived classes : Knife, Maverick, AK47, Magnum, DEagle ....

Polymorphism: The CS example

- Every player has some weapon to kill the enemy
- Weapon can be used by left-click or right-click of the mouse
- All weapons do the same function on moving the mouse
- But each weapon has its own unique function on clicking the mouse
- Base class : Weapon
- Derived classes : Knife, Maverick, AK47, Magnum, DEagle ....

Polymorphism: The CS example

- Every player has some weapon to kill the enemy
- Weapon can be used by left-click or right-click of the mouse
- All weapons do the same function on moving the mouse
- But each weapon has its own unique function on clicking the mouse
- Base class : Weapon
- Derived classes : Knife, Maverick, AK47, Magnum, DEagle ....

Polymorphism: The CS example

- Every player has some weapon to kill the enemy
- Weapon can be used by left-click or right-click of the mouse
- All weapons do the same function on moving the mouse
- But each weapon has its own unique function on clicking the mouse
- Base class : Weapon
- Derived classes : Knife, Maverick, AK47, Magnum, DEagle ....

Polymorphism: The CS example

- Every player has some weapon to kill the enemy
- Weapon can be used by left-click or right-click of the mouse
- All weapons do the same function on moving the mouse
- But each weapon has its own unique function on clicking the mouse
- Base class : Weapon
- Derived classes : Knife, Maverick, AK47, Magnum, DEagle ....

Polymorphism: The CS example

- Every player has some weapon to kill the enemy
- Weapon can be used by left-click or right-click of the mouse
- All weapons do the same function on moving the mouse
- But each weapon has its own unique function on clicking the mouse
- Base class : Weapon
- Derived classes : Knife, Maverick, AK47, Magnum, DEagle ....

Polymorphism: The CS Example

```
class Weapon {
protected:
  int x,y;
public:
  void mouseMove(int x, int y);
  virtual void leftClick() = 0;
  virtual void rightClick() = 0;
};
```

- Every weapon moves the same way when the mouse is moved. This functionality is common to all weapons, and hence defined in the base class itself.
- Every weapon has its own special functionality on a leftClick or a rightClick, and hence these weapon dependent features are left undefined in base class.
- They will have to be defined properly in the actual derived class of every weapon

Polymorphism: The CS Example

```
class Weapon {
protected:
  int x,y;
public:
  void mouseMove(int x, int y);
  virtual void leftClick() = 0;
  virtual void rightClick() = 0;
};
```

- Every weapon moves the same way when the mouse is moved. This functionality is common to all weapons, and hence defined in the base class itself.
- Every weapon has its own special functionality on a leftClick or a rightClick, and hence these weapon dependent features are left undefined in base class.
- They will have to be defined properly in the actual derived class of every weapon

Polymorphism: The CS Example

```
class Weapon {
protected:
  int x,y;
public:
  void mouseMove(int x, int y);
  virtual void leftClick() = 0;
  virtual void rightClick() = 0;
};
```

- Every weapon moves the same way when the mouse is moved. This functionality is common to all weapons, and hence defined in the base class itself.
- Every weapon has its own special functionality on a leftClick or a rightClick, and hence these weapon dependent features are left undefined in base class.
- They will have to be defined properly in the actual derived class of every weapon

Polymorphism: The CS Example

```
class Weapon {
protected:
  int x,y;
public:
  void mouseMove(int x, int y);
  virtual void leftClick() = 0;
  virtual void rightClick() = 0;
};
```

- Every weapon moves the same way when the mouse is moved. This functionality is common to all weapons, and hence defined in the base class itself.
- Every weapon has its own special functionality on a leftClick or a rightClick, and hence these weapon dependent features are left undefined in base class.
- They will have to be defined properly in the actual derived class of every weapon

Polymorphism: The CS Example

```
class M4A1: public Weapon {
public:
  void mouseMove(int x, int y);
  virtual void leftClick() {
    // code to shoot a bullet
    // reload if no bullets
  }
  virtual void rightClick() {
    // Enable/Disable Silencer
  }
};
```

Polymorphism: The CS Example

```
int main() {
  Player comp = new Player();
  AK47 gun1();
  M4A1 gun2();
  // Player.gun is a pointer to the class weapon
  Player.gun = &gun1;
  // Calls to AK47
  Player.gun.leftClick();
  Player.gun.rightClick();

  Player.gun = &gun2;
  // Calls to M4A1
  Player.gun.leftClick();
  Player.gun.rightClick();
}
```

Polymorphism: The CS Example

- Player.gun is a pointer to class Weapon
- Hence it can also be used to point to objects of AK47 and M4A1
- True magic of polymorphism is that it invokes the correct function depending on the object being pointed to
- In the example, it was never specified which type of weapon is being used.

Polymorphism: The CS Example

- Player.gun is a pointer to class Weapon
- Hence it can also be used to point to objects of AK47 and M4A1
- True magic of polymorphism is that it invokes the correct function depending on the object being pointed to
- In the example, it was never specified which type of weapon is being used.

Polymorphism: The CS Example

- Player.gun is a pointer to class Weapon
- Hence it can also be used to point to objects of AK47 and M4A1
- True magic of polymorphism is that it invokes the correct function depending on the object being pointed to
- In the example, it was never specified which type of weapon is being used.

Polymorphism: The CS Example

- Player.gun is a pointer to class Weapon
- Hence it can also be used to point to objects of AK47 and M4A1
- True magic of polymorphism is that it invokes the correct function depending on the object being pointed to
- In the example, it was never specified which type of weapon is being used.

Virtual Destructors

```
class Base {
public:
  Base() {cout<<''construct base''<<endl;}
  ~Base() {cout<<''Destroy base''<<endl;}
};
class Derive: public Base {
public:
  Dervie() {cout<<''construct derive''<<endl;}
  ~Derive() {cout<<''Destroy derive''<<endl;}
};
int main() {
  Base *baseptr = new Derive();
  delete basePtr;
}
```

Virtual Destructors

- Output
  Construct Base
  Construct Derive
  Destroy Base
- The pointer to base class does not have access to the destructor of the derived class
- Memory taken by members belonging to the derived class is not returned back to system.
- Memory leak

Virtual Destructors

- Output
  Construct Base
  Construct Derive
  Destroy Base
- The pointer to base class does not have access to the destructor of the derived class
- Memory taken by members belonging to the derived class is not returned back to system.
- Memory leak

Virtual Destructors

- Output
  Construct Base
  Construct Derive
  Destroy Base
- The pointer to base class does not have access to the destructor of the derived class
- Memory taken by members belonging to the derived class is not returned back to system.
- Memory leak

Virtual Destructors

- Output
  Construct Base
  Construct Derive
  Destroy Base
- The pointer to base class does not have access to the destructor of the derived class
- Memory taken by members belonging to the derived class is not returned back to system.
- Memory leak

Virtual Destructors

```
class Base {
public:
  Base() {cout<<''construct base''<<endl;}
  virtual ~Base() {cout<<''Destroy base''<<endl;}
};
class Derive: public Base {
public:
  Dervie() {cout<<''construct derive''<<endl;}
  ~Derive() {cout<<''Destroy derive''<<endl;}
};
int main() {
  Base *baseptr = new Derive();
  delete basePtr;
}
```

Virtual Destructors

- Output
  Construct Base
  Construct Derive
  Destroy Derive
  Destroy Base
- Memory leak avoided by making destructor virtual, and hence accessible to base class pointer
- Make destructor protected and non-virtual to deliberately avoid this

Virtual Destructors

- Output
  Construct Base
  Construct Derive
  Destroy Derive
  Destroy Base
- Memory leak avoided by making destructor virtual, and hence accessible to base class pointer
- Make destructor protected and non-virtual to deliberately avoid this

Virtual Destructors

- Output
  Construct Base
  Construct Derive
  Destroy Derive
  Destroy Base
- Memory leak avoided by making destructor virtual, and hence accessible to base class pointer
- Make destructor protected and non-virtual to deliberately avoid this