ACA Summer School 2014
Advanced C++

Pankaj Prateek

ACA, CSE, IIT Kanpur

June 24, 2014

Pointers Revisited

- Pointers are derived data types that contain memory address of some other variable
- Dereferencing a pointer and pointer arithmetic works exactly in the same way as with structs and built-in types
- Accessing (Class Pointers):

```
item x(12,100);
item* ptr = &x;
//
Ways to access
x.getDetails();
ptr->getDetails();
(*ptr).getDetails();
// All three ways are equivalent
```

Pointers Revisited

- Pointers are derived data types that contain memory address of some other variable
- Dereferencing a pointer and pointer arithmetic works exactly in the same way as with structs and built-in types
- Accessing (Class Pointers):

```
item x(12,100);
item* ptr = &x;
//
Ways to access
x.getDetails();
ptr->getDetails();
(*ptr).getDetails();
// All three ways are equivalent
```

Pointers Revisited

- Pointers are derived data types that contain memory address of some other variable
- Dereferencing a pointer and pointer arithmetic works exactly in the same way as with structs and built-in types
- Accessing (Class Pointers):

```
item x(12,100);
item* ptr = &x;
//
Ways to access
x.getDetails();
ptr->getDetails();
(*ptr).getDetails();
// All three ways are equivalent
```

Pointers Revisited

- Void Pointers:
  - Generic pointers which can refer to variables of any type
  - Need to be typecasted to proper type before dereferencing
- Null Pointers:
  - Pointers to a specific data type which does not point to any object of that type
  - "Null" or "0" pointers

Pointers Revisited

- Void Pointers:
  - Generic pointers which can refer to variables of any type
  - Need to be typecasted to proper type before dereferencing
- Null Pointers:
  - Pointers to a specific data type which does not point to any object of that type
  - "Null" or "0" pointers

Pointers Revisited

- Void Pointers:
  - Generic pointers which can refer to variables of any type
  - Need to be typecasted to proper type before dereferencing
- Null Pointers:
  - Pointers to a specific data type which does not point to any object of that type
  - "Null" or "0" pointers

Pointers Revisited

- Void Pointers:
  - Generic pointers which can refer to variables of any type
  - Need to be typecasted to proper type before dereferencing
- Null Pointers:
  - Pointers to a specific data type which does not point to any object of that type
  - "Null" or "0" pointers

Pointers Revisited

- Void Pointers:
  - Generic pointers which can refer to variables of any type
  - Need to be typecasted to proper type before dereferencing
- Null Pointers:
  - Pointers to a specific data type which does not point to any object of that type
  - "Null" or "0" pointers

Pointers Revisited

- Void Pointers:
  - Generic pointers which can refer to variables of any type
  - Need to be typecasted to proper type before dereferencing
- Null Pointers:
  - Pointers to a specific data type which does not point to any object of that type
  - “Null” or “0” pointers

Class: `this` pointer

- `this` pointer is used to represent the object for which the current member function was called
- Automatically passed as an implicit arguement when a member function is called
- `*this` is the reference to the object which called the function

Class: `this` pointer

- `this` pointer is used to represent the object for which the current member function was called
- Automatically passed as an implicit arguement when a member function is called
- `*this` is the reference to the object which called the function

Class: `this` pointer

- `this` pointer is used to represent the object for which the current member function was called
- Automatically passed as an implicit arguement when a member function is called
- `*this` is the reference to the object which called the function

Class: this pointer

```
class Person {
  double height;
public:
  person& taller(person& x) {
    if (x.height > height)
      return x;
    else
      return *this;
  }
};

Person A, B, tallest;
tallest = A.taller(B);
```

Classes without constructors

- We have been using member functions to set the values of member variables

```
class item {
  int number, cost;
public:
  void setValue(int itemNum, int itemCost);
  void getValue(void);
};
```

- Classes should allow to use customized definitons in the same way as we use built-in definitions. Initialization and destruction properties are important for this.

Classes without constructors

- We have been using member functions to set the values of member variables

```
class item {
  int number, cost;
public:
  void setValue(int itemNum, int itemCost);
  void getValue(void);
};
```

- Classes should allow to use customized definitons in the same way as we use built-in definitions. Initialization and destruction properties are important for this.

Class: Constructors

- Constructors are special member functions whose task is to initialize objects of a class.
- Constructors have the same name as the class
- Constructors are invoked when an object of the class is created.
- Constructors do not have a return type

Class: Constructors

- Constructors are special member functions whose task is to initialize objects of a class.
- Constructors have the same name as the class
- Constructors are invoked when an object of the class is created.
- Constructors do not have a return type

Class: Constructors

- Constructors are special member functions whose task is to initialize objects of a class.
- Constructors have the same name as the class
- Constructors are invoked when an object of the class is created.
- Constructors do not have a return type

Class: Constructors

- Constructors are special member functions whose task is to initialize objects of a class.
- Constructors have the same name as the class
- Constructors are invoked when an object of the class is created.
- Constructors do not have a return type

Class: Constructors

```
class item {
  int number;
  int cost;
public:
  item(void) {  // Constructor
    number = cost = 0;
  }
  void getValue(void);
};
int main() {
  item soap, pencil, pen;
  soap.getValue();
  pencil.getValue();
  pen.getValue();
}
```

Class: Constructor Properties

- Constructors should be declared in the public section of the class, except for some really special cases (singleton design patterns)
- Constructors cannot have a return type (not even void)
- Constructors can have multiple parameters and default values (just like normal functions)
- A class can have multiple constructors. All have the same name (the name of the class), but which constructor is called depends on the signature of the constructor. This is called constructor overloading.
- Constructors cannot be virtual (discussed later)
- Constructors can have reference to an object of the same class as an input parameter. Such constructors are copy constructors

Class: Constructor Properties

- Constructors should be declared in the public section of the class, except for some really special cases (singleton design patterns)
- Constructors cannot have a return type (not even void)
- Constructors can have multiple parameters and default values (just like normal functions)
- A class can have multiple constructors. All have the same name (the name of the class), but which constructor is called depends on the signature of the constructor. This is called constructor overloading.
- Constructors cannot be virtual (discussed later)
- Constructors can have reference to an object of the same class as an input parameter. Such constructors are copy constructors

Class: Constructor Properties

- Constructors should be declared in the public section of the class, except for some really special cases (singleton design patterns)
- Constructors cannot have a return type (not even void)
- Constructors can have multiple parameters and default values (just like normal functions)
- A class can have multiple constructors. All have the same name (the name of the class), but which constructor is called depends on the signature of the constructor. This is called constructor overloading.
- Constructors cannot be virtual (discussed later)
- Constructors can have reference to an object of the same class as an input parameter. Such constructors are copy constructors

Class: Constructor Properties

- Constructors should be declared in the public section of the class, except for some really special cases (singleton design patterns)
- Constructors cannot have a return type (not even void)
- Constructors can have multiple parameters and default values (just like normal functions)
- A class can have multiple constructors. All have the same name (the name of the class), but which constructor is called depends on the signature of the constructor. This is called constructor overloading.
- Constructors cannot be virtual (discussed later)
- Constructors can have reference to an object of the same class as an input parameter. Such constructors are copy constructors

Class: Constructor Properties

- Constructors should be declared in the public section of the class, except for some really special cases (singleton design patterns)
- Constructors cannot have a return type (not even void)
- Constructors can have multiple parameters and default values (just like normal functions)
- A class can have multiple constructors. All have the same name (the name of the class), but which constructor is called depends on the signature of the constructor. This is called constructor overloading.
- Constructors cannot be virtual (discussed later)
- Constructors can have reference to an object of the same class as an input parameter. Such constructors are copy constructors

Class: Constructor Properties

- Constructors should be declared in the public section of the class, except for some really special cases (singleton design patterns)
- Constructors cannot have a return type (not even void)
- Constructors can have multiple parameters and default values (just like normal functions)
- A class can have multiple constructors. All have the same name (the name of the class), but which constructor is called depends on the signature of the constructor. This is called constructor overloading.
- Constructors cannot be virtual (discussed later)
- Constructors can have reference to an object of the same class as an input parameter. Such constructors are copy constructors

Class: Default Constructor

- Even if when a constructor has not been defined, the compiler will define a default constructor. This constructor does nothing, it is an empty constructor.
- If some constructors are defined, but they are all non-default, the compiler will not implicitly define a default constructor. This might cause problems. Thus, define a default constructor whenever a non-default constructor is defined.

Class: Default Constructor

- Even if when a constructor has not been defined, the compiler will define a default constructor. This constructor does nothing, it is an empty constructor.
- If some constructors are defined, but they are all non-default, the compiler will not implicitly define a default constructor. This might cause problems. Thus, define a default constructor whenever a non-default constructor is defined.

Class: Default Constructor

```
class item {
  int number, cost;
public:
  item(); //default
  item(int itemNum, int itemCost) {
    number = itemNum; cost = itemCost;
  }
};
item pen1, pen2;
item pencil1(123, 40), pencil2(123, 10);
```

Class: Multiple Constructors

- Which constructor is called depends on the constructor signature.

Class: Copy Constructors

- Constructors which are used to declare and initialize an object from another object, most probably via a reference to that object.

Class: Copy Constructors

```
class item {
  int number, cost;
public:
  item(); //default
  item(int itemNum, int itemCost) {
    number = itemNum; cost = itemCost;
  }
  item(item& temp) {
    // Accessing private members
    number = temp.number;
    cost = temp.cost;
  }
};
item pen1(123, 40);
item pen2(&pen1);
```

Private Members

- Why can one access private members of another object in the definition of the copy constructor?
- Access modifiers (private, public) work only at class level and not at object level. Gence two objects of the same class can access each other's private members without any error!!

Private Members

- Why can one access private members of another object in the definition of the copy constructor?
- Access modifiers (private, public) work only at class level and not at object level. Gence two objects of the same class can access each other's private members without any error!!

Class: Copy Constructors

- The process of Initialization of an object through a copy cnstructor is known as copy initialization.

Class: Destructors

- Special functions like constructors. Destroy objects created by a constructor
- They neither have any input arguement, not a return value.
- Implicitly called by a compiler when the object goes out of scope.

Class: Destructors

- Special functions like constructors. Destroy objects created by a constructor
- They neither have any input arguement, not a return value.
- Implicitly called by a compiler when the object goes out of scope.

Class: Destructors

- Special functions like constructors. Destroy objects created by a constructor
- They neither have any input arguement, not a return value.
- Implicitly called by a compiler when the object goes out of scope.

Code Reusability

- Nobody likes to write code for the same functionality again and again without any significant improvement
- Using already written code is more reliable, saves time, money and frustration for the programmer of debugging
- Example of reuse: for students and teachers, the properties of the class person are common, and hence should be implented only once
- Reuse of class : Inheritance
- Inheritance : Using old classes and their properties to make new classes
- Old class : Base class
- New class : Derived class
- The derived class inherits, some or all, properties of the base class

Code Reusability

- Nobody likes to write code for the same functionality again and again without any significant improvement
- Using already written code is more reliable, saves time, money and frustration for the programmer of debugging
- Example of reuse: for students and teachers, the properties of the class person are common, and hence should be implented only once
- Reuse of class : Inheritance
- Inheritance : Using old classes and their properties to make new classes
- Old class : Base class
- New class : Derived class
- The derived class inherits, some or all, properties of the base class

Code Reusability

- Nobody likes to write code for the same functionality again and again without any significant improvement
- Using already written code is more reliable, saves time, money and frustration for the programmer of debugging
- Example of reuse: for students and teachers, the properties of the class person are common, and hence should be implented only once
- Reuse of class : Inheritance
- Inheritance : Using old classes and their properties to make new classes
- Old class : Base class
- New class : Derived class
- The derived class inherits, some or all, properties of the base class

Code Reusability

- Nobody likes to write code for the same functionality again and again without any significant improvement
- Using already written code is more reliable, saves time, money and frustration for the programmer of debugging
- Example of reuse: for students and teachers, the properties of the class person are common, and hence should be implented only once
- Reuse of class : Inheritance
- Inheritance : Using old classes and their properties to make new classes
- Old class : Base class
- New class : Derived class
- The derived class inherits, some or all, properties of the base class

Code Reusability

- Nobody likes to write code for the same functionality again and again without any significant improvement
- Using already written code is more reliable, saves time, money and frustration for the programmer of debugging
- Example of reuse: for students and teachers, the properties of the class person are common, and hence should be implented only once
- Reuse of class : Inheritance
- Inheritance : Using old classes and their properties to make new classes
- Old class : Base class
- New class : Derived class
- The derived class inherits, some or all, properties of the base class

Code Reusability

- Nobody likes to write code for the same functionality again and again without any significant improvement
- Using already written code is more reliable, saves time, money and frustration for the programmer of debugging
- Example of reuse: for students and teachers, the properties of the class person are common, and hence should be implented only once
- Reuse of class : Inheritance
- Inheritance : Using old classes and their properties to make new classes
- Old class : Base class
- New class : Derived class
- The derived class inherits, some or all, properties of the base class

Code Reusability

- Nobody likes to write code for the same functionality again and again without any significant improvement
- Using already written code is more reliable, saves time, money and frustration for the programmer of debugging
- Example of reuse: for students and teachers, the properties of the class person are common, and hence should be implented only once
- Reuse of class : Inheritance
- Inheritance : Using old classes and their properties to make new classes
- Old class : Base class
- New class : Derived class
- The derived class inherits, some or all, properties of the base class

Code Reusability

- Nobody likes to write code for the same functionality again and again without any significant improvement
- Using already written code is more reliable, saves time, money and frustration for the programmer of debugging
- Example of reuse: for students and teachers, the properties of the class person are common, and hence should be implented only once
- Reuse of class : Inheritance
- Inheritance : Using old classes and their properties to make new classes
- Old class : Base class
- New class : Derived class
- The derived class inherits, some or all, properties of the base class