Object Oriented Programming

Pointers

Mr. Usman Wajid

usman.wajid@nu.edu.pk

March 22, 2023

National University
of Computer & Emerging Sciences

Pointer to Objects

- So far we treated objects as ordinary variables that are created when declared and vanished when goes out of scope

Pointer to Objects

- So far we treated objects as ordinary variables that are created when declared and vanished when goes out of scope
- Objects may also be created **dynamically** and **destroyed** when required

Pointer to Objects

- So far we treated objects as ordinary variables that are created when declared and vanished when goes out of scope
- Objects may also be created **dynamically** and **destroyed** when required
- In other words, **objects may appear on demand**

Pointer to Objects example

```
class Student {
        public:
        Student(){
                cout<<"Object constructed"<<endl;
        }
        ~Student(){
                cout<<"Object destructed"<<endl;
        }
};
int main() {

        Student * ptr1 = new Student;
        Student * ptr2 = new Student();
        delete ptr1;
        delete ptr2;
}
```

- an object created using the `new` keyword

Pointer to Objects example

```
class Student {
        public:
        Student(){
                cout<<"Object constructed"<<endl;
        }
        ~Student(){
                cout<<"Object destructed"<<endl;
        }
};
int main() {

        Student * ptr1 = new Student;
        Student * ptr2 = new Student();
        delete ptr1;
        delete ptr2;
}
```

- an object created using the `new` keyword
- We can omit empty parenthesis after the class name. In either case, default-constructor will be called

Pointer to Objects example

```
class Student {
        public:
        Student(){
                cout<<"Object constructed"<<endl;
        }
        ~Student(){
                cout<<"Object destructed"<<endl;
        }
};
int main() {

        Student * ptr1 = new Student;
        Student * ptr2 = new Student();
        delete ptr1;
        delete ptr2;
}
```

- an object created using the `new` keyword
- We can omit empty parenthesis after the class name. In either case, default-constructor will be called
- An object is destroyed using the `delete` keyword

Pointer to Objects example

```
class Student {
        public:
        Student(){
                cout<<"Object constructed"<<endl;
        }
        ~Student(){
                cout<<"Object destructed"<<endl;
        }
};
int main() {

        Student * ptr1 = new Student;
        Student * ptr2 = new Student();
        delete ptr1;
        delete ptr2;
}
```

- an object created using the `new` keyword
- We can omit empty parenthesis after the class name. In either case, default-constructor will be called
- An object is destroyed using the `delete` keyword

```
Object constructed
Object constructed
Object destructed
Object destructed
```

Pointers in fields

- All variables, including objects, are stored in separate area of memory called the stack

Pointers in fields

- All variables, including objects, are stored in separate area of memory called the stack
- The stacks grows when new variables are created and shrinks when the variables goes out of scope

Pointers in fields

- All variables, including objects, are stored in separate area of memory called the stack
- The stacks grows when new variables are created and shrinks when the variables goes out of scope
- Note that this process is beyond your control. You can not affect the way in which the stack changes

Pointers in fields

- All variables, including objects, are stored in separate area of memory called the stack
- The stacks grows when new variables are created and shrinks when the variables goes out of scope
- Note that this process is beyond your control. You can not affect the way in which the stack changes
- The entities created ”on demand” by the `new` keyword are created in specific memory region usually called a heap.

Pointers in fields

- All variables, including objects, are stored in separate area of memory called the stack
- The stacks grows when new variables are created and shrinks when the variables goes out of scope
- Note that this process is beyond your control. You can not affect the way in which the stack changes
- The entities created ”on demand” by the `new` keyword are created in specific memory region usually called a heap.
- Unlike stack, the heap can be controlled manually

Pointers in fields

- All variables, including objects, are stored in separate area of memory called the stack
- The stacks grows when new variables are created and shrinks when the variables goes out of scope
- Note that this process is beyond your control. You can not affect the way in which the stack changes
- The entities created ”on demand” by the `new` keyword are created in specific memory region usually called a heap.
- Unlike stack, the heap can be controlled manually
- an ordinary dot ”.” operation can not be directly performed for entities stored in heap, unless the pointer is deferenced

Pointers in fields

- All variables, including objects, are stored in separate area of memory called the stack
- The stacks grows when new variables are created and shrinks when the variables goes out of scope
- Note that this process is beyond your control. You can not affect the way in which the stack changes
- The entities created ”on demand” by the `new` keyword are created in specific memory region usually called a heap.
- Unlike stack, the heap can be controlled manually
- an ordinary dot ”.” operation can not be directly performed for entities stored in heap, unless the pointer is deferenced
- or one can use the ”arrow” (->) operator instead

Pointer to Fields example

```
class Student {
        private:
                int rollNo;
        public:
                Student(int rollNo=0){
                        setRollNo(rollNo);
                }
                void setRollNo(int rollNo){
                        this->rollNo = rollNo;
                }
        int getRollNo(){
                return rollNo;
        }
};
int main() {

        Student * ptr1 = new Student(1);
        cout<<"rollNo: "<<ptr1->getRollNo()<<endl;
        cout<<"rollNo: "<<(*ptr1).getRollNo()<<endl;
        delete ptr1;
}
```

Pointer to Fields example

```
class Student {
        private:
                int rollNo;
        public:
                Student(int rollNo=0){
                        setRollNo(rollNo);
                }
                void setRollNo(int rollNo){
                        this->rollNo = rollNo;
                }
        int getRollNo(){
                return rollNo;
        }
};
int main() {

        Student * ptr1 = new Student(1);
        cout<<"rollNo: "<<ptr1->getRollNo()<<endl;
        cout<<"rollNo: "<<(*ptr1).getRollNo()<<endl;
        delete ptr1;
}
```

- (*p).getRollNo(); // p is explicitly dereferenced

Pointer to Fields example

```
class Student {
        private:
                int rollNo;
        public:
                Student(int rollNo=0){
                        setRollNo(rollNo);
                }
                void setRollNo(int rollNo){
                        this->rollNo = rollNo;
                }
        int getRollNo(){
                return rollNo;
        }
};
int main() {

        Student * ptr1 = new Student(1);
        cout<<"rollNo: "<<ptr1->getRollNo()<<endl;
        cout<<"rollNo: "<<(*ptr1).getRollNo()<<endl;
        delete ptr1;
}
```

- (*p).getRollNo(); // p is explicitly dereferenced
- p-> getRollNo(); // p is implicitly dereferenced

Pointer to Fields example

```
class Student {
        private:
                int rollNo;
        public:
                Student(int rollNo=0){
                        setRollNo(rollNo);
                }
                void setRollNo(int rollNo){
                        this->rollNo = rollNo;
                }
        int getRollNo(){
                return rollNo;
        }
};
int main() {

        Student * ptr1 = new Student(1);
        cout<<"rollNo: "<<ptr1->getRollNo()<<endl;
        cout<<"rollNo: "<<(*ptr1).getRollNo()<<endl;
        delete ptr1;
}
```

- (*p).getRollNo(); // p is explicitly dereferenced
- p-> getRollNo(); // p is implicitly dereferenced
- **Output:**

```
rollNo: 1
rollNo: 1
```

Memory Leaks

- Entities such a variable and objects are allocated memory to perform their operations

Memory Leaks

- Entities such a variable and objects are allocated memory to perform their operations
- This memory should be released when these operations are done. In most cases it is done automatically

Memory Leaks

- Entities such a variable and objects are allocated memory to perform their operations
- This memory should be released when these operations are done. In most cases it is done automatically
- Failure to clean memory activates a phenomena known as memory leaking, i.e, the un-accessed data residing in memory affects system performance

Memory Leaks Example

```
class Section {
        public:
                int * totalStudents;

                Section(int num){
                        totalStudents = new int[num];
                }

                ~Section(){
                        cout<<"Object destructed"<<endl;
                }
};

void makeALeak(){
        Section secA(50);
}

int main() {
        makeALeak();
}
```

Memory Leaks Example

```
class Section {
        public:
                int * totalStudents;

                Section(int num){
                        totalStudents = new int[num];
                }

                ~Section(){
                        cout<<"Object destructed"<<endl;
                }
};

void makeALeak(){
        Section secA(50);
}

int main() {
        makeALeak();
}
```

```
Object destructed
```

Memory Leaks continued …

- The constructor allocates another part of memory (heap) to pointer **totalStudents**

Memory Leaks continued …

- The constructor allocates another part of memory (heap) to pointer **totalStudents**
- The object ”secA” is an example of automatic variable, i.e., it is removed from memory when it goes out of scope

Memory Leaks continued …

- The constructor allocates another part of memory (heap) to pointer **totalStudents**
- The object ”secA” is an example of automatic variable, i.e., it is removed from memory when it goes out of scope
- On a return from the makeALeak() function, the memory allocated **(stack)** to secA will be retrieved automatically

Memory Leaks continued …

- The constructor allocates another part of memory (heap) to pointer **totalStudents**
- The object ”secA” is an example of automatic variable, i.e., it is removed from memory when it goes out of scope
- On a return from the makeALeak() function, the memory allocated **(stack)** to secA will be retrieved automatically
- Unfortunately, the memory allocated to the dynamic pointer ”totalStudents” stored in another location (**heap**) still resides in memory

Memory Leaks continued …

- The constructor allocates another part of memory (heap) to pointer **totalStudents**
- The object ”secA” is an example of automatic variable, i.e., it is removed from memory when it goes out of scope
- On a return from the makeALeak() function, the memory allocated **(stack)** to secA will be retrieved automatically
- Unfortunately, the memory allocated to the dynamic pointer ”totalStudents” stored in another location (**heap**) still resides in memory
- Hence, a fairly large portion of memory is leaked (still resides in memory but not accessible any more)

Memory Leaks Example

```
class Section {
        public:
        int * totalStudents;

        Section(int num){
                totalStudents = new int[num];
        }

        ~Section(){
                delete [] totalStudents;
                cout<<"Object destructed"<<endl;
        }
};

void makeALeak(){
        Section secA(50);
}

int main() {
        makeALeak();
}
```

- **Solution:**
  Put the necessary code in the destructor to ensure that when object `secA` goes out of scope, the memory allocated to the dynamic constructor `totalStudents` is retrieved or freed

Memory Leaks Example

```
class Section {
        public:
        int * totalStudents;

        Section(int num){
                totalStudents = new int[num];
        }

        ~Section(){
                delete [] totalStudents;
                cout<<"Object destructed"<<endl;
        }
};

void makeALeak(){
        Section secA(50);
}

int main() {
        makeALeak();
}
```

- **Solution:**
  Put the necessary code in the destructor to ensure that when object `secA` goes out of scope, the memory allocated to the dynamic constructor `totalStudents` is retrieved or freed
- **Output:**

```
Object destructed
```