Lec_14.md

Lecture 14 - Pointers, Scopes, Arrays

Oct. 13/2020

Note-takers Note: Happy Thanksgiving! I hope you had a wonderful long weekend and took some time off to spend with family/friends!

Moving onto the lecture:

We've learned how to dynamically allocate variables and structs

- But what about dynamically allocating **objects**?

Dynamic Allocation of Objects

main.cpp

```
class DayOfYear{
private:
  int day;
  int month;
public:
  DayOfYear();
  DayOfYear(int d,int m);
  void setDay(int d);
  void setMonth(int m);
  void print();
}

int main(){
  DayOfYear* day1;
  day1 = new DayOfYear;
}
```

Notes:

1. `DayOfYear* day1;`
    - Create an **pointer** of type `DayOfYear`
2. `day1 = new DayOfYear;`
    - **Dynamically allocate** enough memory for a `DayOfYear` object
        - Store the pointer to this `DayOfYear` object in `day1`
    - This does not **instantiate** the object
        - No **constructor** is called

main.cpp

```
class DayOfYear{
private:
  int day;
  int month;
public:
  DayOfYear();
  DayOfYear(int d,int m);
  void setDay(int d);
  void setMonth(int m);
```

```
  void print();
}

int main(){
  DayOfYear* day1;
  day1 = new DayOfYear;
  DayOfYear* day2;
  day2 = new DayOfYear(1,1);
}
```

Notes:

1. `DayOfYear* day2;`
   - Create another **pointer** of type `DayOfYear`
2. `day2 = new DayOfYear(1,1);`
   - Create a *new* object and store the address in `day2`
   - Remember that this **instantiates** an object
     - **Constructor** is called
       - But which **constructor**?
         - In this case, `DayOfYear(int d,int m)` because of parameters given in object **instantiation** (`DayOfYear(1,1);`)

main.cpp

```
class DayOfYear{
private:
  int day;
  int month;
public:
  DayOfYear();
  DayOfYear(int d,int m);
  void setDay(int d);
  void setMonth(int m);
  void print();
}

int main(){
  DayOfYear* day1;
  day1 = new DayOfYear;
  DayOfYear* day2;
  day2 = new DayOfYear(1,1);

  delete day1;
  day1 = nullptr;
  delete day2;
  day2 = nullptr;
}
```

Notes:

1. `delete day2;`
   - `delete` frees the memory of the **value** at `day1`
     - But remember that when **objects** are deleted, the **destructor** is called
       - Which **destructor**?
         - Trick question, only one **destructor**
2. `day2 = nullptr;`
   - Remember to set the pointer `day2` to a `nullptr` as to avoid memory leaks/dereferencing invalid memory

Variable Scopes

The **scope** of a variable is the part of the program in which the variable can be used

- Variables are usually **scoped** inside the **code blocks** they are in
- Global variables are defined in 'main.cpp' and are available **globally**, e.g. anywhere in the code

Local vs Global Variables

main.cpp

```
int g;

int main(){
  int i;
  float x;

  return 0;
}
void func1(int y){
  float x;
  int z;
}
```

Notes:

1. `int g;` is defined **globally**
   - `g` can be used *anywhere* in the code
2. `int i; float x;` are defined in the **scope** of `int main()`
   - They can only be used/referenced in the **scope** of `main`
     - Within the code blocks `{ }` of `main`
3. `float x; int z;` are defined in the **scope** of `void func1()`
   - They can only be used/referenced in the **scope** of `func1()`
     - Within the code blocks `{ }` of `func1`

In general, variables are scoped to the **code block** they are declared in.

Scope Hiding/Masking/Eclipsing

main.cpp

```
int g;

int main(){
  int i;
  float x;
  if(c){
    int i;
    i = 5;
  }
  return 0;
}
```

Notes:

1. Notice that `int i;` is called twice, one inside `main` and one inside `if(c)`
   - This is called **Scope Hiding** or **Scope Eclipsing**
     - Unclear which `int i` we should be using
     - Bad coding practice
2. `i = 5;`

- Like mentioned above, *which variable i are we trying to modify?*
  - Not exactly clear, so avoid redefining variables like this.

Scope of Dynamic Data

main.cpp

```
void allocate_int(){
  int* q;
  q = new int;
  *q = 5;
}

int main(){
  allocate_int();
  return 0;
}
```

Notes:

1. `int* q; q = new int; *q = 5;`
   - Create a **pointer** q, allocate enough memory for an `int`, and set the value at address `q` to 5
     - But this **dynamic data** is defined in the **scope** of `allocate_int()`
       - So we cannot use the value of `q` outside `allocate_int()`
2. What exactly is the scope of **dynamic data**?

main.cpp

```
int* allocate_int(){
  int* q;
  q = new int;
  *q = 5;
  return (q);
}

int main(){
  int* p = allocate_int();
  *p = 8;
  return 0;
}
```

Notes:

1. `int* q; q = new int; *q = 5;`
   - Same as previous example.
2. `return q;`
   - Return `q` from the function `allocate_int()`
     - But what *exactly* is `allocate_int()` returning?
   - `return q;` returns the value of q (ok, this was obvious)
     - It returns the **address** of q
       - `q` is a pointer type, so it's value is an **address**
   - After return runs, `q` goes out of scope.
     - We cannot use the value of `q` anymore.
3. `int* p = allocate_int()`
   - `p`, an integer pointer, is assigned the value of `allocate_int()`
     - Remember that `return q;` in `allocate_int()` returns an **address**
       - Specifically, the **address** of `q` (before q went out of scope)

- Now, `p` has the same address `q` *had*

4. `*p = 8;`
    - We can assign the value at the address of `p` (or equivalently, the value at the address of `q` before `q` went out of scope)
5. The scope of **dynamic data** is anywhere
    - Between the bounds of creation and deletion of the **dynamic data**
    - As long as you know the address of the **dynamic data**
        - You can access it

main.cpp

```
int* do_something(){
  int x;
  x = 5;
  return &x;
}

int main(){
  int* a = do_something();
  *a = 8;
  return 0;
}
```

1. `int x; x = 5;`
    - Defines a local variable (not **dynamic**)
        - Sets the value to '5'
2. `return &x;`
    - Return the **address** of x
3. `int* a = do_something();`
    - Set the value of **pointer** `a` to the value of x
        - Except:
            - `x` went out of scope in `do_something();`
                - The memory of `x` has ***been freed***
            - `x` is ***not dynamic***
        - So the **address** of `x` does ***not*** refer to `x`
            - Could be some other variable you defined after
            - Could be a random location of memory now
    - This does not have to generate a **Segmentation Fault**
        - Could give you a compiler error (depends on version and compiler)
    - **Pointer** `a` is now set to some random memory address
        - Since `x` is not defined
4. `*a = 8;`
    - Set the value at the address of `a` to '8'
        - What is this value?
            - Still points to **address** of `x`
            - But unsure what the value is, since `x` is no longer in scope

In this example, `a` is called a **Dangling Pointer**. The data that `a` points to is no longer valid.

Variable Types

1. Global
2. Local
3. Function Arguments

4. Dynamic

type	classification	memory location	description
Global	Automatic Variable	stack	Declared outside **all** functions. Visible everywhere in file.
Local	Automatic Variable	stack	Declared inside a **function** or **code block**. Visible only within that **code block**.
Function	Automatic Variable	stack	Declared within **function headers**. Visible only within **function**.
Dynamic	User-Managed	heap	Allocated by **new**. Exist from **allocation** to **deletion**. visible ***anywhere*** so long as there is a pointer to it.

All of the memory used by the program (instructions, code, variables) are defined inside the memory in 4 distinct areas:

Memory	
Code	Instructions
Data	Global/Static Variables
Stack	Automatic Variables
Heap	Dynamic Variables

When the program finishes, all the memory is **reclaimed** by the OS (Operating System).

- **Reclaiming** is not **deletion**
  - The memory is freed for another program to use.
- So why do we bother deleting/checking for memory leaks? - If OS will handle all that on program completion?
  - What if OS has memory leaks?
  - What if your program is a running **process**?
    - e.g. a Web Server
  - Just check for **dangling pointers** and **memory leaks**