Lec_15.md

Lecture 15 - Pointers, Scopes, Arrays

Oct. 14/2020

Note-takers Note: Good luck on the 212 quiz! If you're taking that at 6pm EST, I have no idea.

Arrays

What if we want a **data-type** that carries multiple values of a **primitive** type?

- e.g. List of student ID's
- e.g. Consecutive prime numbers or something

Quick review of **arrays** from APS105:

main.cpp

```
int main(){
  int a[4];
  a[0] = 4;
  cout << a[3];
}
```

Notes:

1. `int a[4];`
   - In order to define the array this way, the size must be *known at compile-time*
     - e.g. cannot be **dynamically allocated**
2. `a[0]=4;`
   - Assign "4" to the first element of the array
3. `cout << a[3];`
   - Print the 3rd element of the array to the screen

Arrays and Pointers

Arrays and pointers have a special relationship:

- Name of the array ***is also*** a **pointer** to the first element of the array
- The following rows are equivalent:

Array Indexing	Pointer Dereferencing
a[0]	*a
a[1]	*(a+1)
a[i]	*(a+i)
i[a]	*(i+a)

Every row in the table above contains equivalent code.

Pointer Arithmetic

Notice when we index `a[1]` using the pointer `a`, we write `*(a+1)`

- What does 1 represent?
    - Is this a single integer value?

The 1 represents **one address unit** in **memory**

- Adding 1 to a pointer results in the *next* memory block
    - 0x0000AB14 + 1 = 0x0000AB18
    - Depends on the ***size*** of the variable represented by the address

Pointer Arithmetic is not great coding practice, as it is not always clear what the size of the value at `a` is.

- Thus not clear how many bytes between `*(a+1)` and `*(a)`

Dynamic Allocation of Arrays

Because arrays and pointers have this special relationship, we can **dynamically allocate** arrays

main.cpp

```
#include <iostream>

using namespace std;

int main(){
  int* myarray;
  int size;

  cin >> size;
  myarray = new int[size];
}
```

Notes:

1. `myarray = new int[size];`
    - Create a **integer array pointer**
        - Allocates "size" number of "integer memory blocks" for the array
            - In this case, an "integer memory block" is usually 4 bytes large
            - So `new int[size]` will allocate "4*size" bytes
    - This does ***not*** define each individual array element

main.cpp

```
#include <iostream>

using namespace std;

int main(){
  int* myarray;
  int size;

  cin >> size;
  myarray = new int[size];

  myarray[0]=0;
  for(int i = 1;i < size;++i){
    myarray[i] = 1;
```

```
  }
}
```

Notes:

1. `for(int i = 1;i < size;++i){ myarray[i] = 1; }[i`
    - Loop through elements from i=1 to i=size-1
        - Set elements in `myarray` to '1'

main.cpp

```
#include <iostream>

using namespace std;

int main(){
  int* myarray;
  int size;

  cin >> size;
  myarray = new int[size];

  myarray[0]=0;
  for(int i = 1;i < size;++i){
    myarray[i] = 1;
  }

  delete[] myarray;
  myarray = null;
}
```

Notes:

1. `delete[] myarray;`
    - The `delete` keyword has *slightly different* behaviour when used with arrays
        - De-allocates ***all*** memory that was originally allocated for `myarray` with `new` keyword
        - Do not need to de-allocate all individual elements of array
2. `myarray = null;`
    - Good practice
    - Avoid dangling pointers

Arrays of Structs

Allocating **arrays** of structs

```
struct node{
  int ID;
  struct node* next;
}
struct node a[4];
```

Dynamically allocating **arrays** of structs

main.cpp

```
struct node{
  int ID;
  struct node* next;
```

```
}

int main(){
  cin >> size;
  struct node* a;
  a = new struct node[size];
}
```

Notes:

1. `struct node* a;`
    - Defines a pointer to a **struct node**
2. `a = new struct node[size];`
    - Creates a **struct node array pointer**
        - Allocates "size" amount of "struct node memory blocks" for the array
            - In this case, a "struct node memory block" is
                - The size of an integer and
                - The size of a **struct node pointer**

main.cpp

```
struct node{
  int ID;
  struct node* next;
}

int main(){
  cin >> size;
  struct node* a;
  a = new struct node[size];

  delete [] a;
  a = nullptr;
}
```

Notes:

1. `delete [] a;`
    - De-allocates ***all*** memory that was originally allocated for `a` with `new` keyword
2. `a = null;`
    - Good practice
    - Avoid dangling pointers

main.cpp

```
struct node{
  int ID;
  struct node* next;
}

int main(){
  cin >> size;
  struct node* a;
  a = new struct node[size];

  int i = 0;
  a = new struct node;

  a[i].ID = 2;
  *(a+i).ID = 2;
```

```
    (a+i)->ID = 2;

    delete [] a;
    a = nullptr;
  }
```

Notes:

1. `a = new struct node;`
    - Create a node variable at `a[0]`
        - Remember that `a` is a pointer to `a[0]`
            - `a[0]` is the first element in the array
2. `a[i].ID = 2;`
    - Set the ID of the `i` th struct node element to 2
3. `*(a+i).ID = 2;`
    - Set the ID of the `i` th struct node element to 2
4. `(a+i)->ID = 2;`
    - Set the ID of the `i` th struct node element to 2
5. (2.,3.,4.) do the exact same thing.
    - Difference is in readability :)

Arrays of Pointers

How to allocate an array `a` of 100 **integer pointers**

`int* a[100];`

Or, **dynamically**:

main.cpp

```
int main(){

  cin >> size;
  int** a;
  a = new int*[size];
}
```

Notes:

1. `int** a;`
    - Define a **double pointer** of type `int` called a
2. `a = new int*[size];`
    - Dynamically create an array of **integer pointers** of size "size"

main.cpp

```
int main(){

  cin >> size;
  int** a;
  a = new int*[size];
  for(int i = 0;i < size;++i){
    a[i] = new int;
  }
}
```

Notes:

1. `for(int i = 0;i < size;++i){ a[i] = new int; }`
   - Loop through elements from i=1 to i=size-1
     - Define each element of a as an **integer pointer**

We've figured out how to make each pointer in the array point to an int

- In other words, we've figured out how to **initialize** each of the pointer elements in the array `a`
- How do we delete this **dynamically allocated** data?

main.cpp

```
int main(){

  cin >> size;
  int** a;
  a = new int*[size];
  for(int i = 0;i < size;++i){
    a[i] = new int;
  }

  for(int i = 0;i < size;++i){
    delete a[i];
    a[i] = null;
  }
  delete [] a;
  a = nullptr;
}
```

Notes:

1. `for(int i = 0;i < size;++i){ }`
   - Loop through elements from i=1 to i=size-1
2. In For Loop: `delete a[i];`
   - De-allocate the pointer element at `a[i]`
3. In For Loop: `a[i] = null;`
   - Set the recently deleted pointers to `null`
4. What if we don't delete all the individual array elements (**pointers**)
   - We end up with **dangling pointers**
     - The pointers are still allocated in memory
5. `delete [] a;`
   - De=allocates ***all*** memory that was originally allocated for `a` with `new` keyword
6. `a = null;`
   - Set array pointer `a` to `null;`

How do we allocate an array called `a` of 100 **pointers** (to struct nodes)

```
struct node* a[100];
```

main.cpp

```
int main(){

  cin >> size;
  struct node** a;
  a = new struct node*[size];

  for(int i = 0;i < size;++i){
```

```
      a[i] = new struct node;
    }

    for(int i = 0;i < size;++i){
      delete a[i];
      a[i] = null;
    }

    delete [] a;
    a = nullptr;
  }
```