Lec_13.md

Lecture 13 - Pointers, Scopes, and Arrays

Oct.8/2020

Pointers and the New Keyword

Using pointers to change values of addresses and reference variables is cool

- But we can also use pointers to create *new variables* at **run-time**
  - These variables are **dynamically allocated**

In C programming (APS105), we use `malloc` to allocate a certain amount of memory

- In C++, there is a syntactically sweeter way to allocate memory
  - **new** keyword

main.cpp

```
#include <iostream>
using namespace std;
int main(){
  int x = 8;
  int* px = &x;
  cout << *px << endl;
  px = new int;
  *px = 5;
  cout << *px << endl;
  px = new int;
  *px = 6;
  cout << *px << endl;
  return 0;
}
```

address	memory contents	symbol
0x0000AB00	8	x
0x0000AB04	5	
0x0000AB08	6	
0x0000AB0c		
0x0000AB10		
0x0000AB14	0x0000AB08	px
0x0000AB18		

Notes:

1. `int* px = &x;`
   - Create a **pointer** variable of type `int*` and of name `px`
   - Point `px` to the *address* of x
2. `px = new int;`

    - Defines a **dynamically allocated** piece of memory (of size `int` - 4 bytes)
        - The amount of memory allocated by `new` depends on the **type** of the variable
3. Notice above that there are no symbols for memory addresses `0x0000AB04` and `0x0000AB08`
    - So how can we change the value of `0x0000AB04` (which is 5)?
        - We can't, because we do not have a pointer or address of this memory
            - This is called a **memory leak**

Memory leaks are bad for your program

- Cannot reference that memory address

main.cpp

```
#include <iostream>
using namespace std;
int main(){
  int x = 8;
  int* px = &x;
  cout << *px << endl;
  px = new int;
  *px = 5;
  cout << *px << endl;
  px = new int;
  *px = 6;
  cout << *px << endl;
  delete px;
  return 0;
}
```

address	memory contents	symbol
0x0000AB00	8	x
0x0000AB04	5	
0x0000AB08		
0x0000AB0c		
0x0000AB10		
0x0000AB14	0x0000AB08	px
0x0000AB18		

Notes:

1. `delete px;`
    - ***Important***:
        - The `delete` keyword **deletes** the value (frees the memory) at **the value** pointed to by `px`
        - It ***does not*** delete the address stored by `px`
    - `delete` cannot delete **non-dynamic** data
2. Notice that `px` **still** points to an address
    - That address (in this case, `0x0000AB08`) is ***empty***
        - Attempting to use `px` to change the value at the address is a *bug*
    - Make sure to **not use this pointer address**

main.cpp

```
#include <iostream>
using namespace std;
int main(){
  int x = 8;
  int* px = &x;
  cout << *px << endl;
  px = new int;
  *px = 5;
  cout << *px << endl;
  px = new int;
  *px = 6;
  cout << *px << endl;
  delete px;
  px = nullptr;
  px = NULL;
  return 0;
}
```

address	memory contents	symbol
0x0000AB00	8	x
0x0000AB04	5	
0x0000AB08		
0x0000AB0c		
0x0000AB10		
0x0000AB14		px
0x0000AB18		

Notes:

1. `px=nullptr`
   - This will set the pointer variable to `nullptr`
     - A null pointer is an empty pointer - *evaluates to 0*: zero, nilch, nada.
       - Indicates that the pointer is not pointing to anything
     - Good practice to set the pointer to `null` after `delete`
2. `px=NULL`
   - Identical to the command in 1. (`px=nullptr`)

Pointers to Structs

Also throwback to `structs` in C programming (APS105)

- We can also use structs in C++

main.cpp

```
#include <iostream>
using namespace std;

struct node{
  int ID;
  float value;
}

int main(){
```

```
  struct node* ptr;
  ptr = new struct node;

  return 0;
}
```

Notes:

1. `struct node`
   - Same `struct` definition as C
   - Defines an abstract data type with multiple **fields**
2. `ptr = new struct node;`
   - **Dynamically allocate** new `struct node`

main.cpp

```
#include <iostream>
using namespace std;

struct node{
  int ID;
  float value;
}

int main(){
  struct node* ptr;
  ptr = new struct node;

  (*ptr).ID = 2;
  (*ptr).value = 6.2;

  return 0;
}
```

Notes:

1. `(*ptr).ID = 2;`
   - Access the **field** `ID` of the **data type** at address `ptr`
   - Why are there parentheses around `*ptr` ?
     - Shouldn't `*ptr.ID = 2;` work as well?
       - The **field access operator** `.` has higher **precedence** than the **dereference operator** `*`
         - The `.` operator will run first, which will return an error, since `ptr` is not a variable that has fields that the compiler recognizes
   - Access the **field** `ID` of `*ptr` and set it equal to 2
2. `(*ptr).value = 6.2;`
   - Access the **field** `value` of `*ptr` and set it equal to 6.2

main.cpp

```
#include <iostream>
using namespace std;

struct node{
  int ID;
  float value;
}

int main(){
  struct node* ptr;
```

```
  ptr = new struct node;

  ptr->ID = 2;
  ptr->value = 6.2;

  return 0;
}
```

Notes:

1. `ptr->ID = 2;`
   - This is **syntactic sugar** in C++, as in
     - This operator `->` does the same thing as `*(variable).`
   - `ptr->ID` has ***identical*** functionality to `(*ptr).ID`

Pointer in Structs

We've seen pointers used to refer to structs and **dynamically allocate** them

- We can also have pointers *inside* structs

main.cpp

```
#include <iostream>
using namespace std;

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

int main(){
  struct node* head;
  struct node* ptr;

  head = new struct node;
  head->ID = 0;
  head->next = nullptr;

  ptr = new struct node;
  (*ptr).ID = 1;
  (*ptr).next = nullptr;

  (*head).next = ptr;
  ptr = NULL;
  return 0;
}
```

Notes:

1. First, we can recognize that this is the basic implementation for a **linked list**0
2. `head->next = nullptr;`
   - Set the next item/node in the linked list (after the head) to a nullptr
3. `(*ptr).next = nullptr;`
   - Set the next item/node in the linked list (after the ptr node) to a nullptr
   - Remember that `(*ptr).` is *identical* to `ptr->`

What if we have more than two elements?

main.cpp

```
#include <iostream>
using namespace std;

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

int main(){
  struct node* head;
  struct node* ptr;
  struct node* ptr2;

  head = new struct node;
  head->ID = 0;
  head->next = nullptr;

  ptr = new struct node;
  (*ptr).ID = 1;
  (*ptr).next = nullptr;

  ptr2 = new struct node;
  (*ptr2).ID = 2;
  (*ptr2).next = nullptr;

  (*head).next = ptr;
  (*ptr).next = ptr2;
  return 0;
}
```

Notes:

1. How can we access the `ID` field of `ptr2` ?
   - We can easily use `ptr2->ID` , but what if we ***only*** know the address of the head node?
     - `head->next` will return `ptr`
       - `next` field in `head` points to address of `ptr`
     - `head->next->next` will return `ptr2`
       - `next` field in `head` points to address of `ptr` , same as above
       - Then `next` field in `ptr` points to address of `ptr2`
       - So `head->next->next` will give `ptr2`
         - Can then use `head->next->next` to reference **fields** in `ptr2`

Pointers to Pointers

We've already seen pointers pointing to regular variables

- `int* p = &x` , where `x` is a variable of type `int`

But what about pointers to pointers?

- e.g. what if we want a pointer to point to another pointer?

main.cpp

```
#include <iostream>
using namespace std;

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

```
}

int main(){
  int** p2p;
  int* p;
  int* q;

  p = new int;
  *p = 5;

  p2p = &p;
}
```

address	memory contents	symbol
0x0000AB00	5	
0x0000AB04		
0x0000AB08		
0x0000AB0c		
0x0000AB10	0x0000AB00	p
0x0000AB14	0x0000AB10	p2p
0x0000AB18		q

Notes:

1. `int** p2p;`
   - Create a **pointer to pointer** of type `int`
     - Where a **pointer** points to the address of a variable
     - A **pointer to pointer** points to the address of a **pointer**
2. `int* p;`
   - Define a pointer that points to a memory address
3. `*p = 5`
   - Set the value at the address pointed to by `p` to 5
4. `p2p = &p;`
   - Take the **address** of `p` (a pointer), and store that address in `p2p`
     - Notice that `p2p` has the contents `0x0000AB10`, or the **address** of `p`

main.cpp

```
#include <iostream>
using namespace std;

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

int main(){
  int** p2p;
  int* p;
  int* q;

  p = new int;
  *p = 5;

  p2p = &p;
```

```
    q = *p2p;

    *q = 8;

    cout << **p2p;

    return 0;
}
```

address	memory contents	symbol
0x0000AB00	8	
0x0000AB04		
0x0000AB08		
0x0000AB0c		
0x0000AB10	0x0000AB00	p
0x0000AB14	0x0000AB10	p2p
0x0000AB18	0x0000AB00	q

Notes:

1. `q = *p2p;`
    - Assign to q the value at the address pointed to by `p2p`
        - In this case, assign `0x0000AB00` to `q`
            - Why is the assign `0x0000AB00` ?
                - Since `0x0000AB10` is the value in `p2p` .
                    - This is the content/value of `p2p`
                    - This is the address pointed to by `p2p`
                - The value at `0x0000AB10` is `0x0000AB00`
        - `q` now points to `0x0000AB00`
            - In other words, `q` and `p` now point to ***exactly*** the same thing
2. `*q = 8;`
    - Assign the value at the address pointed by `q` to "8"
        - Remember from part 1. that `q` and `p` point to the same thing.
        - Assign the value at `0x0000AB00` to "8"
            - `*p` would also now be "8", since `q` and `p` point to the same thing
3. `cout << **p2p;`
    - Print the value of the (the value store by pointer `p2p`)
    - Lets break this down:
        - Recall that `q` is identical to `*p2p`
            - This means that `*p2p` contains the value `0x0000AB00`
        - So, `**p2p` refers to the value at address `0x0000AB00`
            - Which is "8"
    - So `cout << **p2p` will print "8", or the value of `*p` or `*q`