Week 8

Muhao Chen
muhaochen@ucla.edu
Outline

- Review
- Pointers and references
- Dynamic memory allocation
- Struct
Pointers
**Pointers**

- **Pointer:**
  - Address of a variable in the memory.

- **Declare a pointer (use asterisk):**
  
  `<data_type> * <pointer_name> [= <initialization>]`; 
  
  e.g.: `int * ptr;`  
  
  double *p, *q;  
  
  double *p, *q, r;  
  
  - `<data_type>`: what type of value is pointed by the pointer.
Pointers

- How to point a pointer to a regular variable?
  - `&<variable_name>`, e.g. `int a; int *ptr = &a;`

- How to get the value at the address indicated by the pointer?
  - `*<pointer_name>`, e.g. `int b = *ptr;`

- * and & are two memory operations
* Operator (dereference)

- * before an already-initialized pointer: dereference
  - i.e. to get the value stored behind the address.
    - int a=5, *p; p=&a;

    
    | p: 001EF800 | 001EF804 | 001EF808 | 001EF80C |
    |---|---|---|---|
    | a: 5 |   |   |   |

- cout << p; //will print the address 001EF800 (hexadecimal)
- cout << *p; // will print out 5
Dereference of a pointer

```c
#include <iostream>

int main()
{
    double x, y; // normal double variables
    double *p;   // a pointer to a double variable
    x = 5.5;
    y = -10.0;
    p = &x;     // assign x’s memory address to p
    std::cout << "p: " << p << std::endl;
    std::cout << "*p: " << *p << std::endl;
    p = &y;
    std::cout << "p: " << p << std::endl;
    std::cout << "*p: " << *p << std::endl;
    return 0;
}
```

Output:
```
p: 001EF8B8
*p: 5.5
p: 001EF8A8
*p: -10
```
& operator (reference)

Used before a variable
- Reference: get the address of a variable
  - int a=5;

<table>
<thead>
<tr>
<th>p: 001EF800</th>
<th>001EF804</th>
<th>001EF808</th>
<th>001EF80C</th>
</tr>
</thead>
<tbody>
<tr>
<td>a: 5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- cout << a;  //5
- cout << &a; //001EF800

Inverted operator of *:
- *&a  *&&*a  a  we’ll get the same value
- &&a  X  not allowed. “The Address of an address” is not a correct semantics.
Does a pointer have an address?

- Yes. It’s also a kind of variable, and stored in the memory.

<table>
<thead>
<tr>
<th>p: 001EF800</th>
<th>001EF804</th>
<th>001EF808</th>
<th>001EF80C</th>
</tr>
</thead>
<tbody>
<tr>
<td>a: 5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10FE3F30</td>
<td>10FE3F34</td>
<td>10FE3F38</td>
<td>10FE3F3C</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>p: 001EF800</td>
</tr>
</tbody>
</table>

- cout<< &p; //10FE3F38
Can we create pointers of pointers?

- Pointer is also a type of variable
  - A pointer also has its own pointer, e.g.
    ```
    int a = 10;
    int* ptr = &a;
    int** ptr2ptr = &ptr;
    ```
What is the size of a pointer

- **4Bytes or 8Bytes**
  - Depends on whether your environment is 32-bit or 64-bit

- Regardless of what type of variable it points to
  - int *p=&a;    double *p2=&b;

```
<table>
<thead>
<tr>
<th>p: 001EF800</th>
<th>001EF804</th>
<th>p2: 001EF808</th>
<th>001EF80C</th>
</tr>
</thead>
<tbody>
<tr>
<td>int a: 5</td>
<td></td>
<td>double b: 3.14159265359</td>
<td></td>
</tr>
<tr>
<td>10FE3F30</td>
<td>10FE3F34</td>
<td>10FE3F38</td>
<td>10FE3F3C</td>
</tr>
<tr>
<td>p2: 01EF808</td>
<td></td>
<td>p: 001EF800</td>
<td></td>
</tr>
</tbody>
</table>
```

Both pointers use 4-byte spaces to store a 4-byte address
Can we perform arithmetic operations on a pointer?

- Yes. It will “move” the pointer. (i.e. changes the pointer it points to).
  - int a[5] = {1,2,3,4,5};
  - int *p = a; //or p = &a[0];
  - cout << *p; //1
  - cout << *(p+3); //4
  - p++; cout << *p; //2

```
 1 2 3 4 5
```
```
p
p+3
```
Arithmetic on pointers

- int *p = &a;  // suppose its address is 0x08000000
- What’s the address of *(p+1) ? 0x08000001?
- Actually it’s 0x08000004 (or 0x08000008)
  - Increase a pointer by 1 always adds the size of its dereference type to it
- double *q;
- q++  adds 8 to the address stored in q
  - Let q point the next “double type block” in the memory
Arithmetic on pointers

- Note: priority of * is lower than that of regular arithmetic operations
  - *(p + 1) means access the next block pointed by p
  - *p + 1 means increase 1 to the element pointed by p

```c
int a[2] = {0, 100}
int *p = &a[0];
cout << *(p + 1); //this will get us 100
cout << *p + 1;    //this will get us 1
```
Arithmetic on pointers

Question:
- int a = 5, *q; q=&a;
- Which one increases a to 6?
  - A. (*q)++  
  - B. *q++  
  - C. A and B
- A
- B will only get the dereference of the next block of q. (i.e. q++, then *q)

Priority of ++ is higher than * (+ << * << ++)
Can we perform comparison operations between pointers?

- int a[5];
- int *p=&a[0], *q=&a[1];
- q > p is true

Yes. Addresses are comparable.
int main() {
    char s[]="<<<-----[";
    char t[100];
    char *p=&s[strlen(s) - 1]; // point p to the last character of s
    char *q=&t[0];  //point q to the last character of t
    while (p >= &s[0]) {  //while pointer p doesn’t go before &s[0]
        *q = *p;  //get the content pointed by p to that of q
        p--; q++;  //p moves left, q moves right.
    }
    *q = ‘\0’;
    cout << t << endl;
}
Two ways of using actual parameters

Formal parameter:  
void addOne(int a) { 
a++; 
}  

int main() {  
    int x = 1;  
    addOne(x);  
    cout << x << endl;  
    return 0;  
}  

// output: 1

Actual parameters:  
void addOne(int* a) { 
    (*a)++; 
}  

int main() {  
    int x = 1;  
    addOne(&x);  
    cout << x << endl;  
    return 0;  
}  

// output: 2 (x will change)

void addOne(int& a) { 
    a++; 
}  

int main() {  
    int x = 1;  
    addOne(x);  
    cout << x << endl;  
    return 0;  
}  

// output: 2 (x will change)
Null Pointer

- A null pointer is to indicate that the pointer does not point to anything. (point to address 0)
  - int * p;
  - p = 0;
  - p = NULL;
  - p = nullptr;
Array is one kind of **constant** pointer

- int a[] = \{1,2,3,4,5\};
- a is actually a fixed pointer that points to the first element of the array
- a == &a[0]
Use an array as a pointer

- Use an array as a pointer
  - int a[5];
  - *(a+1) is equivalent to a[1]
  - *(a+2) is equivalent to a[2]

- Array address is not modifiable
  - a++; a += 5;  X

- [ ] is bounded, *( ) is not bounded
  - a[5] usually causes compile error
  - *(a + 5) is accessible, but is an undefined behavior
Reference Type
Reference type

- `<type> &<name> = <referee>`
- `int a=5; int &ra = a;`
- Create another name of a variable
  - i.e. any change made to `a` will happen to `ra`, vice versa
- When declaring a reference type, must initialize it
  - `int &ra;`  **X**

```cpp
int a=5;
int &ra = a;
cout << a++ << endl;
cout << ra++ << endl;
cout << a <<endl;
cout << ra <<endl;
```

```
5
6
7
7
```
Dynamic Memory Allocation
Static memory allocation

- If we want to save a document paragraph into a C-string.
  - #define MAXLENGTH 10000
  - char s[MAXLENGTH+1]; cin.getline(s);
- What if the paragraph is extremely long?
  - out-of-bound
- What if the paragraph has only five words?
  - Over-allocated memory
Dynamic allocation

What if we want to fit the paragraph into a C-string with right the sufficient space of mem?

Dynamic allocation of an array

- `<type> *<name> = new <type>[<#elements>];`
- `char *article = new char[length + 1];`

```c++
int length;
cout << how many characters are at most in your article? << endl;
cin >> length;
char *article;
if (length >0)
    article = new char[length + 1];
```
char s[] = “Oh my god, they killed Kenny!”;
char *t = new char[strlen(s) + 1];
strcpy(t, s);
What if we want to dynamically allocate a 2-D array

```cpp
int rows = 5; int cols = 20;
int **array = new int*[rows];
for (int i=0; i<rows; ++i)
    array[i] = new int[cols];

//array is now array[5][20]
```
The dynamically allocated memory will not be released automatically.

A program may consume huge resources of memory if we allocate memory too many times without releasing it.

```cpp
//data processing
fstream fin, fo;
fin.open("huge_data_set.csv");
fo.open("processed_data_set.csv", std::out);
while (!fin.eof()) {
    char *line = new char[MAX_LINE_LENGTH];
    fin.readline(line);
    process_data_formate(line); //process data
    fo << line; //write a line to file
}```
Delete

- delete[] s;
- Delete the entire array pointed by s and release all the memory.

```cpp
char s1[] = "Respect my authoritah!";
char *t = new char[s1.size() + 1];
strcpy(t, s1);
cout << t << endl;
delete[] t;
```

- Rules of memory allocation: where there’s a New, there’s a corresponding delete.
Memory Leak

int *p;
p = new int[200000];
p = new int[100000];

• We allocate 200000 blocks of int and point p to it.
• Then we allocate another 100000 and point p to it. p no longer points to the first 100000 blocks.
• The first 200000 blocks of int becomes a ghost. We can no longer access it and release it.
• This phenomenon is called Memory Leak.
New, delete a single object

- int *p = new int;
- int *p = new int[1];
- int p = *(new int);  //delete &p;
- delete p;
Struct
Create a database

- Write a simple database that will store a list of you (students).
  - name
  - student ID
  - email address
  - letter grade

```c
#define NUM_STUDENT 33
string name[NUM_STUDENT];
int id[NUM_STUDENT];
string email[NUM_STUDENT];
char grade[NUM_STUDENT];
```

- Inconvenient
  - What if I want to swap records of two students? Perform four swaps.
Define a struct

A compound type of multiple contents.

```cpp
struct student {
    string name;
    int id;
    string email;
    char grade;
}; //Note: there a semi-colon here
```
Declare objects of a struct

- student eric;
- student students[NUM_STUDENTS];
Initialize objects of a struct

```c
struct student {
    string name;
    int id;
    string email;
    char grade;
};  //Note: there a semi colon here

student students[33];
students[0].name = “Eric Cartman”;
students[0].id = 123456789;
students[0].email = “”;
students[0].grade = ‘C’;
```

Accessing attributes of a uninitialized struct object results in undefined behaviors.
Access attributes in a struct object

- `<object name>..<attribute>`

```cpp
student students[33];
students[0].name = "Eric Cartman";
students[0].id = 123456789;
students[0].email = "";
students[0].grade = 'C';

cout << students[0].name << endl;
```

- Manipulating an attribute is same as manipulating a variable.
Pointers of a struct

- Define and initialize
  - student *s1;
  - s1 = &students[0];

- Dynamic allocation of a struct object
  - student *s2 = new student;
  - Since new allocates memory and return a pointer.
Access attributes of a struct pointer

- student *s1=new student;

**We can use . with dereference**
- (*s1).name;

**But for most of time we use ->**
- s1->name;

**Differences between . and ->**
- . left-hand is a struct object
- -> left-hand is a pointer to a struct object
Example of -> and .

```cpp
student students[33];
students[0].name = “Eric Cartman”;
students[0].id = 123456789;
students[0].email = “”;
students[0].grade = ‘F’;
student *p = students;

cout << students[0].name << endl;
cout << p-> grade – 5 << end;
```

Eric Cartman
A
Bugs in your software are actually special features :)