Week 7

Ling Ding

Email: lingding@cs.ucla.edu

Slides modified from Muhao Chen, with permission

Outline

- Pointers and references
- Dynamic Memory Allocation
- Struct
- Class

Outline

- ***Pointers and references***
- Dynamic Memory Allocation
- Struct
- Class

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

```
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
        cout << "p: " << p << endl;
        cout << "*p: " << *p << endl;
        p = &y;
        cout << "p: " << p << endl;
        cout << "*p: " << *p << endl;
        return 0;
}
```

Output:

p: 001EF8B8
***p: 5.5**
p: 001EF8A8
***p: -10**

& operator (reference)

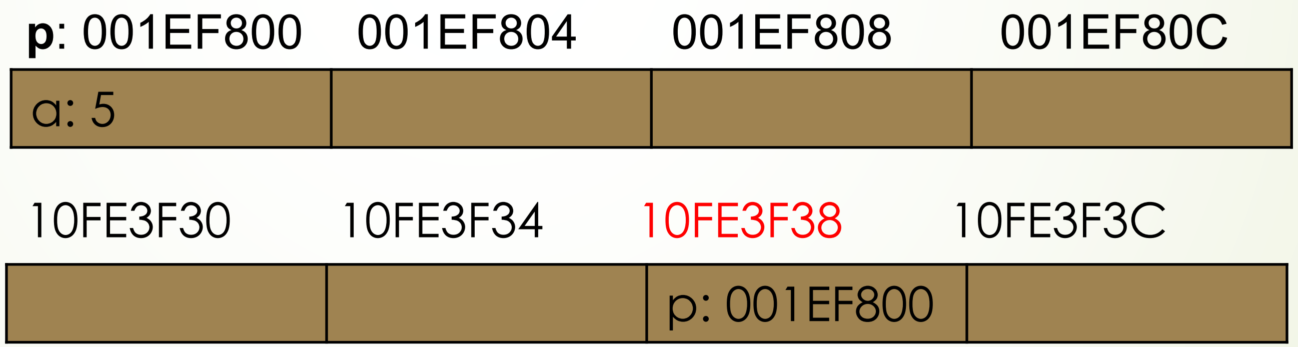
- Used before a variable
  - Reference: get the address of a variable
    - int a=5;

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

    - 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 semantically incorrect.

Does a pointer have an address?

- Does a pointer have an address?
  - Yes. It's also a kind of variable, and stored in the memory.

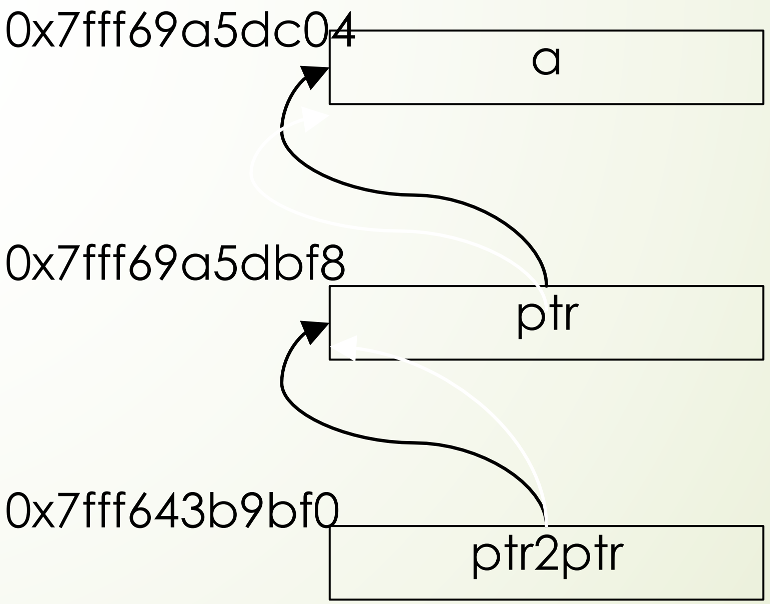


  - cout<< &p; //10FE3F38

Can we create pointers of pointers?

- Pointer is also an 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;

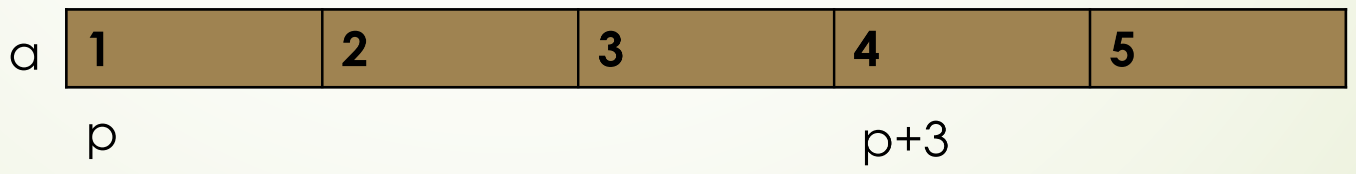
p: 001EF800	001EF804	**p2**: 001EF808	001EF80C
int a: 5		double b: 3.14159265359	

10FE3F30	10FE3F34	10FE3F38	10FE3F3C
p2: 01EF808		p: 001EF800	

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



Pointer Arithmetic

- 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 points to the next "double type block" in the memory

Pointer Arithmetic

- Note: priority of * is higher 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

```
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
```

Pointer Arithmetic

- Question:
  - int a = 5, *q; q=&a;
  - Which one increase *a* into 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 lower 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.

Copy an inverted C-string

```
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;
}
```

```
[-----<<<
```

s										
	<	<	<	-	-	-	-	-	[	\0

p

t										
	[	-	-							\0

q

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)
```

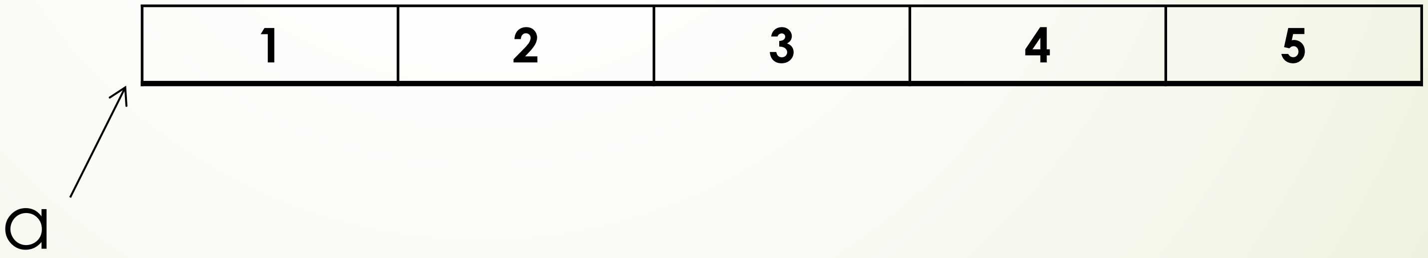
```
void addOne(int& a){
    a++;
}

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;

Pointer VS Array

- 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] causes compile error
  - *(a + 5) is accessible, but is an undefined behavior

Use an array as a pointer

```
int main()
{
        int a[] = {1,2,3,4,5};
        cout << a << endl;
        cout << *a << endl;
        cout << *(a+1) << endl;
        cout << &a[0] << endl;
        cout << &a[1] << endl;
        cout << &a[2] << endl;
        return 0;
}
```

Output:

C4D51D70
1
2
C4D51D70
C4D51D74
C4D51D78

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

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

```
5
6
7
7
```

Outline

- *Pointers and references*
- ***Dynamic Memory Allocation***
- Struct
- Class

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 very long?
  - out-of-bound
- What if the paragraph has only five letters?
  - 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];

Int variable

```
int length;
cout << "How many characters are in a paragraph at the most?" << endl;
cin >> length;
char *article;
if (length >0)
     article = new char[length + 1];
```

Yet another safe copy of a C-string

```
char s[] = “Oh my god, they killed Kenny!”;
char *t = new char[strlen(s) + 1];
strcpy(t, s);
```

new

- new will dynamically allocate a space for the desired type and size.
- new will always return a pointer to the start of the allocated space.
- int array=new int[size]; ✘
- int *array = new int[size]; ✔

What if we want to dynamically allocate a 2-D array

```
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]
```

Delete

- The dynamically allocated memory will not be released automatically.
- A program may consume huge resources (and may even crush) if we allocate memory too many times without releasing it.

```
//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.

```
char s1[] = "They took our jobs!", s2[] = "Respect my authoritah!";
char *t = new char[s1.size() + 1];
strcpy(t, s1);
cout << t << endl;
delete[] t;

t = new char[s2.size() + 1];
strcpy(t, s2);
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;   // delete p;
- int *p = new int[1]; // delete p;
- int p = *(new int); //delete &p;

Outline

- *Pointers and references*
- *Dynamic Memory Allocation*
- ***Struct***
- Class

Create a database

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

```
#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 members.

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

Declare objects of a struct

- student eric;
- student students[NUM_STUDENTS];

Initialize objects of a struct

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

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

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

Access attributes in a struct object

- <object name>.<attribute>

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

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

- Manipulating an attribute is the same as manipulating a variable.

Pointers of a struct

- Define and initialize
  - student *s1;
  - s1 = &students[0];
- If we want to create an object as a pointer
  - 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 .

```
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
```

Outline

- *Pointers and references*
- *Dynamic Memory Allocation*
- *Struct*
- ***Class***

Class

```
class vending_machine {
  public:
      int get_num() const; //accessor
      double get_price() const; //accessor
      void set_num(const int& num);//modifier
  private:
      int num;
      double price;
};
```

```
class human {
  public:
      bool buy_one(const vending_machine &vm);
  private:
      int num_items;
      double cash;
};
```

Implement simple member functions

Accessor:

```
int vending_machine::get_num() const {
    return num;
}
```

Or we can say: (this is a pointer that points to the object itself).

```
int vending_machine::get_num() const {
    return this -> num;
}
```

Modifier:

```
void vending_machine::set_num(const int& num) {
    this -> num = num;
}
```

Constructors: functions to specify the behavior of object initiation

Default constructor (no parameter):

```
vending_machine::vending_machine() {
    num = 10;
    price = 1.75;
}
...
vending_machine vm; //vm is a vending machine that sells 10 items at $1.75 each
```

Function name is the same as the class name. No return type specification.

Constructor with parameters:

```
vending_machine::vending_machine(const int& num, const double & price) {
    this->num = 10;
    this->price = 1.75;
}
...
vending_machine vm(30, 2.0); //vm sells 20 items at $2 each
```

Constructors: functions to specify the behavior of object initiation

If we do not specify any constructors for a class, an **empty constructor** will be provided by default. But it will not if we have specified a constructor.

```
class human {
   public:
       bool buy_one(const vending_machine &vm);
   private:
       int num_items;
       double cash;
};

human::human(const int& num, const double & cash) {
    this->num_items=num;
    this->cash=cash;
}
```

Can we do this?

```
human hm;
```

No. We have to do:

```
      human hm(0, 80.0);
```

Because default constructor is not available.

Constructors: functions to specify the behavior of object initiation

Multiple constructors with different combinations of parameter types.

```
class human {
    public:
        bool buy_one(const vending_machine &vm);
    private:
        int num_items;
        double cash;
};
human::human(const int& num, const double & cash) {
    this->num_items=num;      this->cash=cash;
}
human::human(const double & cash) {
    this->num_items=0; this->cash=cash;
}
human::human() {
    this->num_items=0; this->cash=60.0;
}
```

Caution: private member variables/functions

A private member variable/function can only be seen by the code of this class.

```
class vending_machine {
    public:
        int get_num() const; //accessor
        double get_price() const; //accessor
        void set_num(const int& num);//modifier
    private:
        int num;
        double price;
};
```

Cannot be seen by a human.
Cannot be seen by other functions.

```
class human {
    public:
        bool buy_one(const vending_machine &vm);
    private:
        int num_items;
        double cash;
};
```

Cannot be seen by a vending_machine.
Cannot be seen by other functions (incl. main).

Caution: private member variables/functions

To implement buy_one, can we do:

```
bool human::buy_one(const &vending_machine vm) {
    if (vm.num <= 0 || this->cash <= vm.price) return false;
    vm.num -= 1;
    this->cash -= vm.price;
    return true;
}
```

```
bool human::buy_one(vending_machine &vm) {
    if (vm.get_num() <= 0 || this->cash <= vm.get_price()) return false;
    vm.set_num(vm.get_num() - 1);
    this->cash -= vm.get_price();
    return true;
}
```

Destructors: things to do when an object is destructed

```
vending_machine::~vending_machine() {
    cout<< “A vending machine is out of order.”
}

int main() {
    vending_machine vm;
    return 0;
}
// we’ll see “A vending machine is out of order.”
```

Destructors

A destructor is necessary when we have dynamically allocated member variables:

```
class vending_machines {
    public:
        vending_machines(int num) {
            vms = new vending_machine*[num];
            for (int i=0; i<num; ++i)
                vms[i] = new vending_machine;
            this->num = num;
        }
   private:
        vending_machine **vms;
        int num;
};
```

```
vending_machines::~vending_machines() {
    for (int i=0; i<num; ++i) delete vms[i];
    delete[] vms;
}
```

Class: Things to be careful during the final (esp. coding parts)

- Do not access the private members of another class (page 52)
- Use the correct version of constructor (page 49)
- Release dynamic allocated items in the destructor (page 54)

More Exercises

Thank you!