Lab8
Pointers and Dynamic Arrays

DEE 1319
Department of Electronics Engineering
National Chiao Tung University

Outline

- Pointers
- Classes, pointers, arrays, and dynamic memory
- Lab8 exercise

Pointer Variables

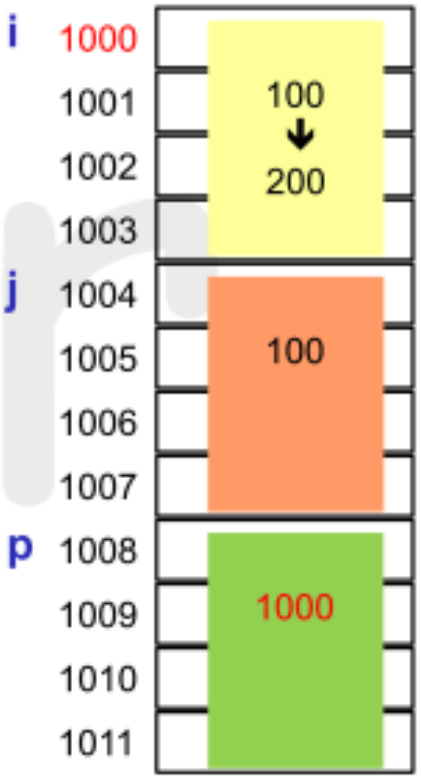
- Pointer variables have types
  - Indicate which type it points to
- Example:

```
int i = 3;      // define variable of type int, its value is an integer
int *ip = &i; // define variable ip of type int*
                // its value is a memory address where an int resides at
                // or, we say ip points to int, or ip is a pointer to int
double d = 3.0;
double *dp = &d;
dp = &i;      // error!! dp is a pointer to double, NOT to int

int **ipp = &ip; // ok, ipp is a pointer to a pointer to int
```

Unary Operators & and *

- Unary address-of (or reference) operator &
  - Get the address of an object
  - int i = 100; //address of i is 1000
    //value stoed in i is 100
  - int *p = &i; //get the address of i , then store in p
    //address of p is 1008
    //value stored in p is 1000 (i's address)



- Unary dereference operator *
  - Refer to the object a pointer points to
  - int j = *p; //p points to i ➔ *p refer to i
    // ➔ equivalently, int j = i;
    *p = 200 //similarly, ➔ i = 200

Pointer Assignment

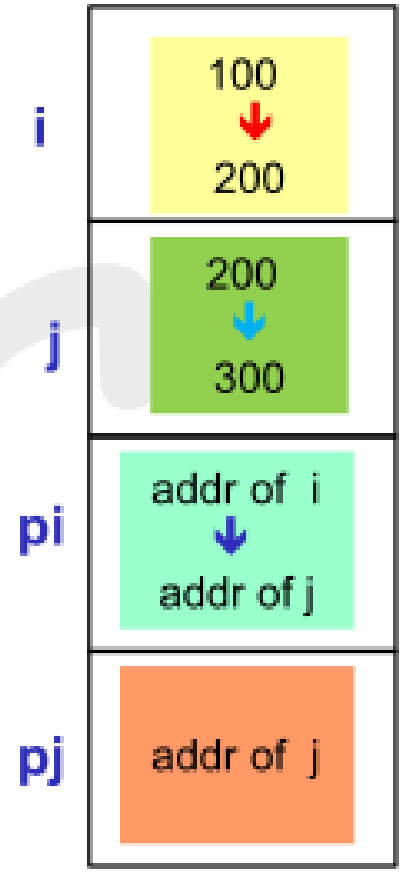
- Example:

```
int i = 100, j = 200, *pi = &i, *pj = &j;
int m = 10, n = 20;
m = n;        //replace the value in m with
              //the value in n ➔ m = 20

*pi = *pj     // ➔ i = j ➔ replace the value in i
              //with the value in j ➔ i = 200

pi = pj;      //replace the value in pi with the value in pj
              // ➔ pi points to what pj points to

*pi = 300; // ➔ j = 300
```



Back to Classes: Operator ->

- Member access operator, ->
  - Member selection from a pointer

```
class X{

public:
    int d1, d2;
    void mbr_func(int);

};

void f(){
    X x, *px = &x;
    a.d1 = 10;              //member selection using .
    a.mbr_func(3);

    px->d2 = 20;        //equivalent to ➔ (*px).d2 = 20;

    px->mbr_func(5);  //equivalent to ➔ (*px). mbr_func(5);

}
```

Dynamic Memory

- Sometimes, we can't know what size of array we need in advance…

```
void f(){
    int score[100];          // the size is FIXED before program execution
    int num;
    cout << " How many students in this class? : ";
    cin >> num;              // what if num > 100??
    for(int i = 0; i < num; ++i)
        cin >> score[i];     // out-of-range runtime error!
    //…
}
```

Dynamic Memory

```
char * ptr;
ptr = (char *) malloc(sizeof(char) * 20);
```

- In C
  - Allocate dynamic memory ➔void *malloc(size_t size); //size in byte
  - Deallocate dynamic memory ➔void free(void *ptr);
- In C++

```
void f(){
    int *score; // score is just a pointer
    int num;
    cout << " How many stufents in this class? : ";
    cin >> num;
    score = new int[num];
    // allocate a memory block from system to store num integers
    //i.e., determine the array size at runtime
    for(int i = 0; i < num; ++i)
          cin >> score[i];   // score cts like an aray!
    //…
    delete[ ] score;  //deallocate (return) the memory block to system
}
```

new Operators

- Allocate dynamic memory from heap
  - when extra memory is required at run time
  - get memory from heap
  - if requesting an object of type X
    ➔ return value of new operator is of type X*

```
int *p = new int(5);   //*p = 5 initially…
                       //do something
delete p;
```

  - p points to allocated memory

delete Operators

- Deallocate dynamic memory
  - When allocated memory is no longer needed
  - Return previously allocate memory to heap
- Example:

```
int *p = new int(5);   //*p = 5 initially
…//do something …
delete p;                    // NO return value for delete operator
```

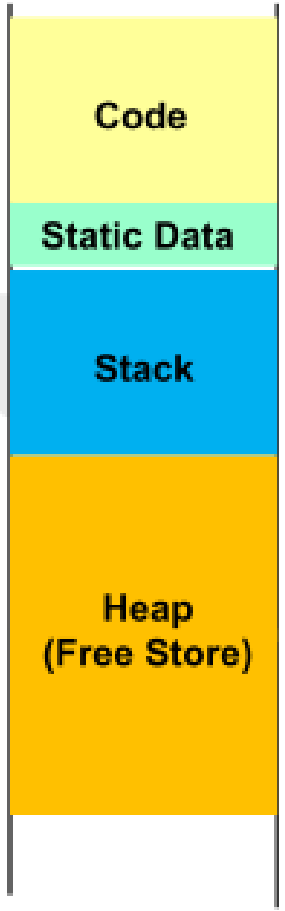
  - deallocate allocated memory pointed by p

Coding Practices

```
void f(){
    int *p1, *p2, *p3;
    p1 = new int;              //allocate ine int from heap
    p2 = p3 = new int;         //allocate one int from heap
    *p2 = 100;
    cout << *p3 << endl;  //100
    p2 = p1;
    *p2 = 200;
    cout << *p1 <<endl;  //200
    delete p1;  //dealloate blue int, p1 now is dangling pointer
    *p1 = 300; //disaster! Do NOT use blue int, it has been deallocated
    p3 = p2;    //there is NO WAY to deallocate green int! BAD!
                //so called memory leak (original p3)
    delete p3; //disaster! Blue int has been deallocated already(delete p1)
}
```

Memory Layout

- Code segment
  - Program execution code
- Static date segment
  - Global objects and static local objects
- Stack
  - Non-static local objects (i.e., automatic objects)
- Heap or free store
  - Free memory, not used by program at the beginning
  - Its size is finite and managed by operating system
- Dynamic memory management
  - Ask for extra memory blocks from heap by **new** operators **dynamically** (i.e., at runtime)
  - Return them back to heap by **delete** operators **dynamiclly**



Dynamic Arrays

- Dynamic array
  - e.g., `int *pi=new int[size];`
  - size can be determined at runtime ➔ flexible
  - However , dynamic memory allocation is relatively time-consuming

- Avoid using dynamic array if size is known at compile time

Creating Dynamic Arrays

- Use new[] operator ➔ X *pX = new X[sz];
  - sz can be a variable, determined at runtime
  - Dynamically allocate enough memory for sz elements of X
  - Return the start address with type X*
  - If X is a user-defined type, **default ctor of X** is invoked for every element in this dynamic array in order
  - What if there is no default ctor? ➔ new[] is not allowed!

```
int sz = 10;
double *pd = new double[sz]; //ok, double is a built-in data type
//assume class X has default ctor and class Y doesn't
X* pX = new X[sz];  //ok, call default ctor for pX[0], …, pX[sz-1]
Y* pY = new Y[sz];  //error ! No default ctor is available
```

Destroying Dynamic Arrays

- Use delete[] operator ➔ delete[] pX;
  - no need to put sz in []
  - need [] to tell compiler that pX points to an array instead of a single object
  - dynamically deallocated previously-allocated memory
  - no return value
  - if X is a user-defined type, **dtor of X** is invoked for every element in this dynamic array in reverse order

```
delete[] pd;
delete[] pX; //call dtor for pX[sz-1], pX[sz-2], …,pX[0]
```

Destructors (1/3)

```
class intArr{                        //int array with runtime range checking
   int size, *arr;
public:
   intArr(int sz) :size(sz) { arr = new int[sz]; }   //ctor
   int& operator[](int idx);     //access idx-th element with range checking
};

void f(int val) {
    intArr ia (100);
    ia[10] = 15;
    ia[20] = ia[10];
    ia[val] = 30;                    //runtime error if val < 0 or val >= 100
}
```

There is a memory leakage issue here ,why?

Destructors (2/3)

- ia is a local (auto) variable
  - Memory for ia (used by ia.size & ia.arr) is automatically allocated/deallocated when ia is created/destroyed
- However, memory block dynamically allocated in ctor never gets deallocated ➔ memory leak!
- But ia is destroyed automatically as soon as f completes…
- How to deallocate that memory?
  ➔ destructor (dtor) !

Destructors (3/3)

```
class intArr{
    int size,*arr;
public:
    intArr(int sz) :size(sz) {arr = new int[sz]; }  //ctor
    ~intArr();                                      //dtor
    int& operator[](int idx);  //access idx-th element with range checking
};
intArr::~intArr( ){ //NO return type and NO parameters are allowed
    delete[] arr; // dellocation here, no memory leak now
}
```

- dtor is automatically invoked right before an object is destroyed
  - Mainly for clean-up operations

Lab8 Exercise (1/3)

- Create a class named BubbleSortArray that has two data member:
  - int * array
    - A pointer to dynamically allocated memory that holds integers
  - int size
    - An integer variable that holds the size of the array

Lab8 Exercise (2/3)

- Write appropriate ctor, dtor, member functions for the class along with the following
  - Constructor (BubbleSortArray(int n))
    - initializes an instance of the class with a specified size n (new())
    - print the message when call the constructor: " ** constructor executed ** “
  - Destructor (~BubbleSortArray())
    - release memory when an instance of the class is destroyed (delete [])
    - print the message when call the destructor: " ** destructor executed ** “

Lab8 Exercise (3/3)

- Write appropriate ctor, dtor, member functions for the class along with the following
  - bubbleSort()
    - the Bubble Sort algorithm to sort the array of integers in ascending order (function has been implemented)
  - display()
    - prints the elements of the array in ascending order
  - findMax()
    - finds the maximum value in the array
    - returns a pointer to the maximum value found in the array

Problem

- This problem involves implementing a class called BubbleSortArray, which manages an array of integers
- The goal is to implement these member functions effectively to provide functionality for initializing, sorting, displaying, and find the maximum value in array
- Limitation in main function
  - only use "bubble_ptr" and "max_ptr“
  - can not initialize other parameter

Input / Output

- Input format
  - Enter the size of the array : <num of element>
  - Enter <num of element> integers : <first number>, <second number >, ….

- Output format
  - Original array
  - Sorted array: in ascending order
  - Max value in the array : <max number in array>

```
Enter the size of the array: 10
 ** constructor executed **
Enter 10 integers: 4 8 3 15 90 -5 -90 81 34 0
Original array: 4 8 3 15 90 -5 -90 81 34 0
Sorted array: -90 -5 0 3 4 8 15 34 81 90
Max value in the array: 90
 ** Destructor executed **
```

Example

- Example 1

```
Enter the size of the array: 5
 ** constructor executed **
Enter 5 integers: 67 9 -15 123 3
Original array: 67 9 -15 123 3
Sorted array: -15 3 9 67 123
Max value in the array: 123
 ** Destructor executed **
```

- Example 2

```
Enter the size of the array: 10
 ** constructor executed **
Enter 10 integers: 4 8 3 15 90 -5 -90 81 34 0
Original array: 4 8 3 15 90 -5 -90 81 34 0
Sorted array: -90 -5 0 3 4 8 15 34 81 90
Max value in the array: 90
 ** Destructor executed **
```