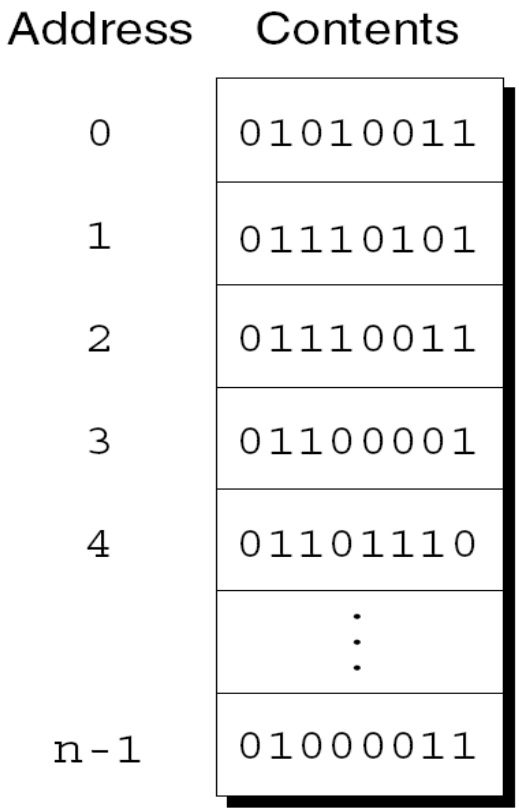
LAB 3

POINTER & DYNAMIC ARRAY

Department of Electronics Engineering
National Chiao Tung University

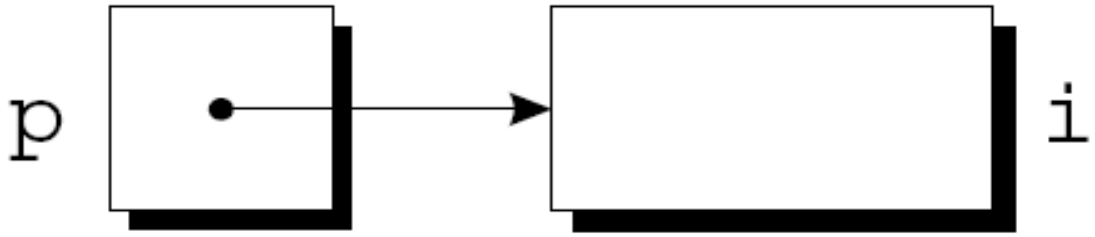
Addresses

- Essentially, the computer's memory is made up of bytes.
- Each byte has a number, an address, associated with it.
- Each byte has a unique ***address***.



Pointer Variables

- Addresses can be stored in ***pointer variables***
- When we store the address of a variable i in the pointer variable p, we say that p "points to" i



Declaring Pointer Variables

- When a pointer variable is declared, its name must be preceded by an **asterisk ***

```
int *p; /*points only to integers  */
```

- `p` is a pointer variable capable of pointing to objects of type `int`

- Pointer variables can appear in declarations along with other variables

```
int i, j, a[10], b[20], *p, *q;
```

Address and Indirection Operators

- **&** (address) operator : Find the address of a variable

- ***** (indirection) operator : Gain access to the object that a pointer points to

Example:

```
int i = 5;
cout << "value of i = " << i << endl;
cout << "address of i = " << &i << endl;
```

Output:

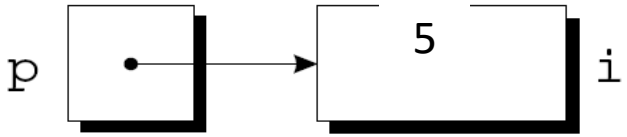
```
value of i = 5
address of i = 0x7ffd93615b9c
```

The Indirection Operator

- Once a pointer variable points to an object, we can use the * (indirection) operator to access what's stored in the object

Example:

```
int i = 5;
int *p = &i;
cout << "i = " << *p << endl;
```



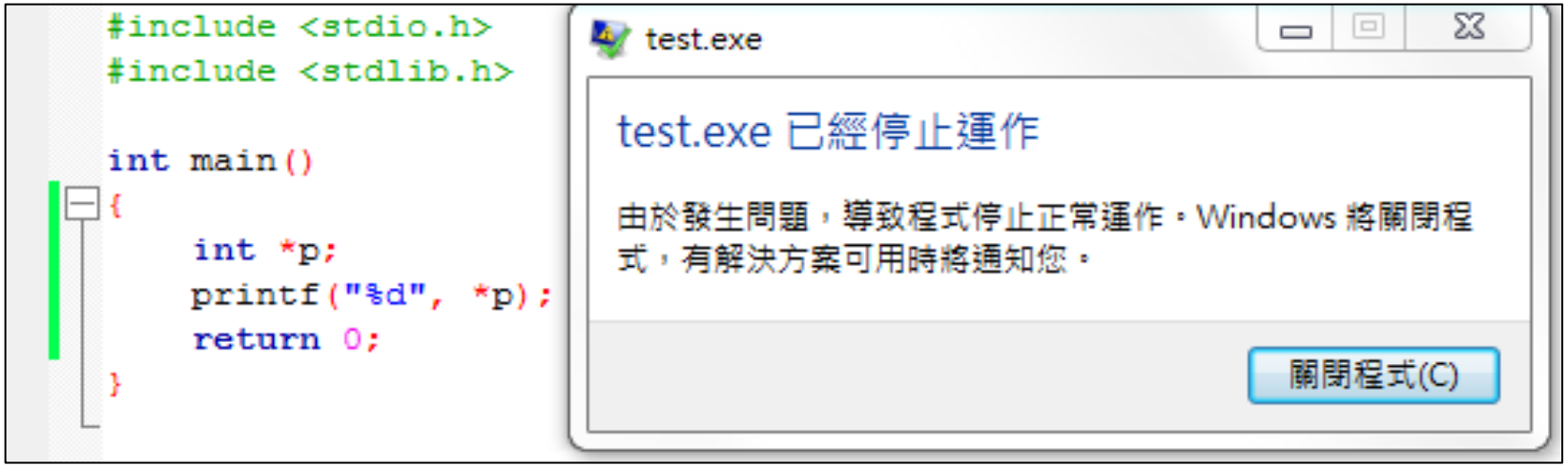
Output:

```
i = 5
```

- `*p` has the same value as `i`
- Changing the value of `*p` changes the value of `i`

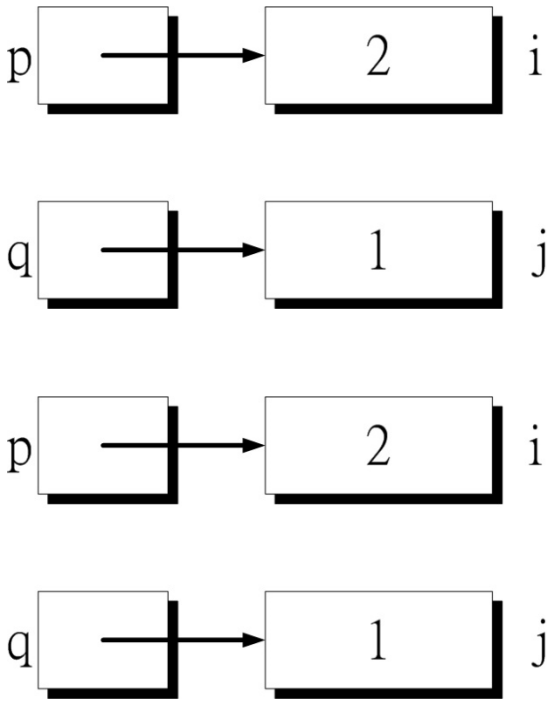
The Indirection Operator

- Applying the indirection operator to an uninitialized pointer variable causes undefined behavior
- int *p;
- cout<<*p; /*** WRONG ***/

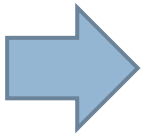


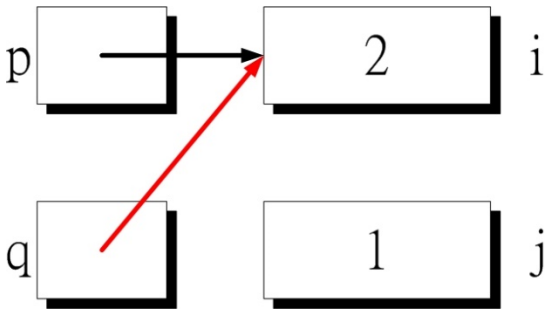
Pointer Assignment

- Difference between q = p; and *q = *p;
- The first statement is a pointer assignment, but the second is not

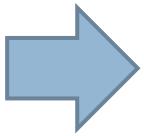


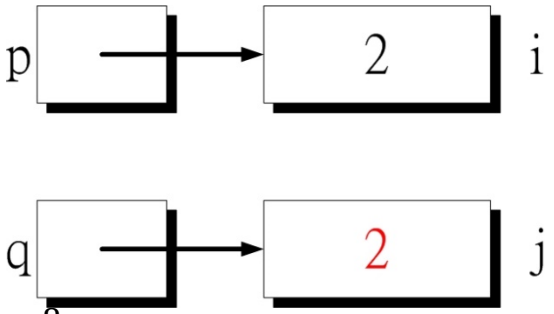
q = p





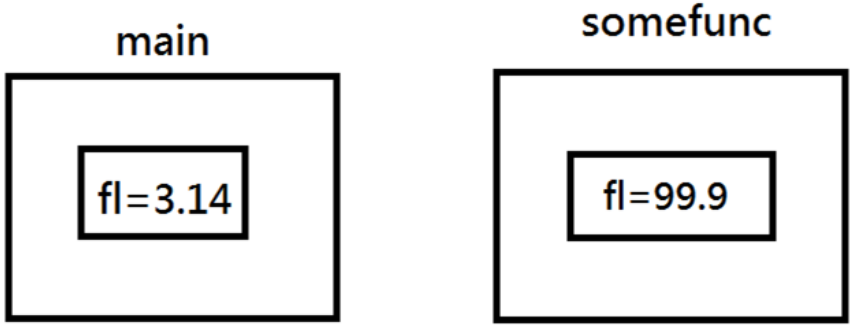
*q = *p





Pass by value

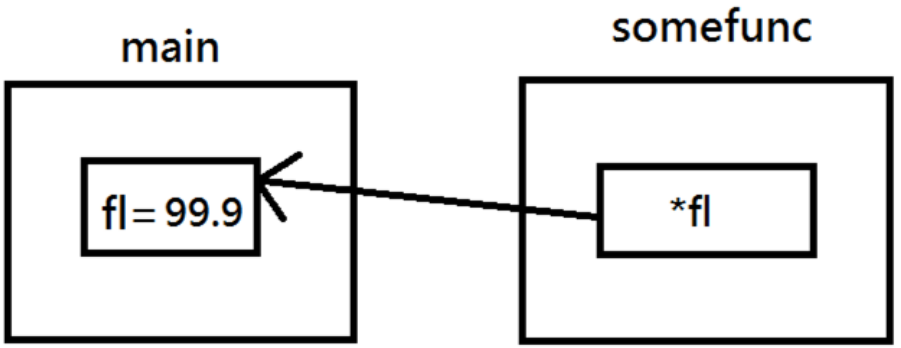
```
1: #include <stdio.h>
2: void somefunc(float fl)
3: {
4:       fl=99.9;
5: }
6: int main()
7: {
8:       float fl=3.14;
9:       somefunc(fl);
10:      cout<<fl;
11:      return 0;
12: }
```



- Result ?
  - fl = 3.14

Pass by pointer

```
 1  #include <stdio.h>
 2
 3  void somefunc(float* fl)
 4    {
 5      *fl = 99.9;
 6    }
 7
 8  int main()
 9   {
10      float fl=3.14;
11      somefunc(&fl);
12      cout<<fl;
13      return 0;
14
15   }
```

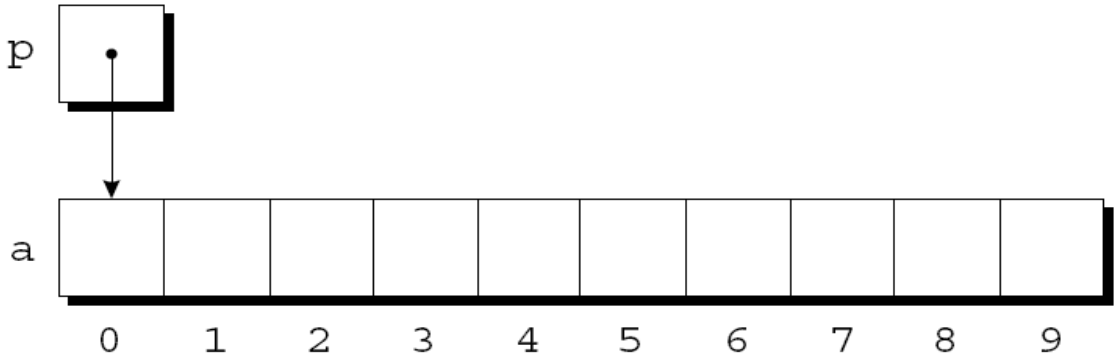


- Result ?
  - fl = 99.9

Pointers and Arrays (1/2)

- Pointers can point to array elements

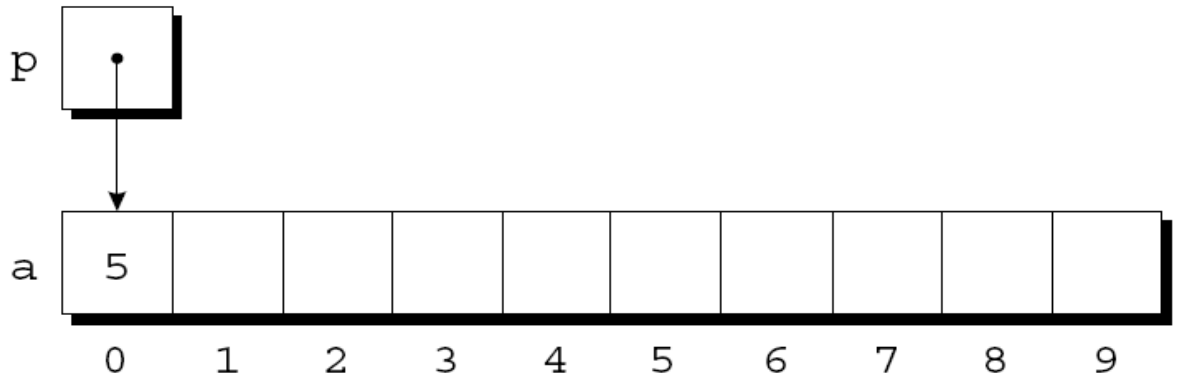
```
int a[10], *p;
p = &a[0];
```



Pointers and Arrays (2/2)

- We can now access a[0] through p; for example, we can store the value 5 in a[0] by writing

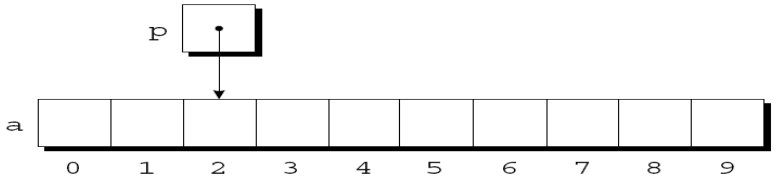
```
int *p = &a[0];
*p = 5;
```



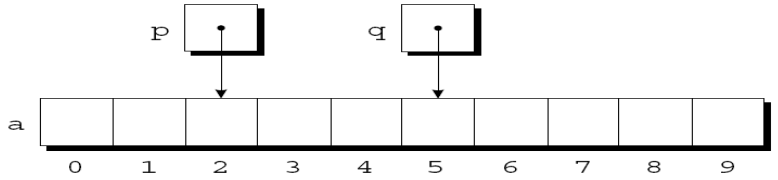
Adding an Integer to a Pointer

- Example of pointer addition : `int *p,*q;`

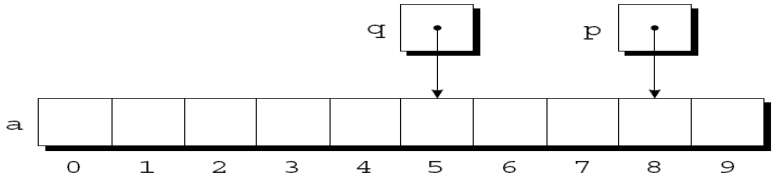
```
p = &a[2];
```



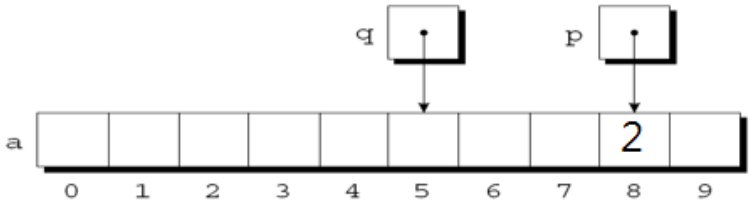

```
q = p + 3;
```




```
p += 6;
```




```
*p=2;
```



Combining the * and ++ Operators

- Because the prefix version of ++ takes precedence over *, the compiler sees this as

  `*++p = j` is equal to `*(++p) = j`

- Compare *p++ & *++p

  *p++ increment p later

  *++p increment p first

Example:

```
int *p = &a[0];
```

`*p++ = 5` or `*p++ = 5` means set a[0] = 5, then p points to a[1]

`*++p = 5` or `*(++p) = 5` means p points to a[1], then set a[1]=5

Using an Array Name as a Pointer

Example:

```
int main(){
    int a[4] = {0,0,0,0};
    *a = 7;          //stores 7 in a[0]
    *(a+2) = 12;     //stores 12 in a[2]
    cout<<a[0]<<" "<<a[1]<<" ";
    cout<<a[2]<<" "<<a[3]<<endl;

    return 0;
}
```

Output:

```
7 0 12 0
```

- In general, `a+i` is the same as `&a[i]`
  - Both represent a pointer to element `i` of `a`
- Also, `*(a+i)` is equivalent to `a[i]`
  - Both represent element `i` itself

Array Arguments (1/2)

- The fact that an array argument is treated as a pointer has some important consequence

- For example, the following function modifies an array by storing zero into each of its elements

```
void store_zeros(int a[], int n)
{
  int i;

  for(i = 0; i < n; ++i)
      a[i] = 0;
}
```

Array Arguments (2/2)

- An array parameter can be declared as a pointer if desired
- `store_zeros` could be defined as follows

```
void store_zeros(int *a, int n)
{
    int i;
    for(i = 0; i < n; ++i)
        *(a+i)= 0;
}
int main()
{
    int a[10];
    store_zeros(&a[0],10)
       …
}
```

- The compiler treats `a[i]` as `*(a+i)`

Standard vs. Dynamic Arrays(1/2)

- Standard array
- fixed dimensions for array
- size for each dimension needs to be a constant
  - must specified size first (estimate maximum, waste memory)

Example:

```
const int MAX_SIXE = 1000000;
int Array[Max_SIZE];
```

but what if we only need 100 integer?

Standard vs. Dynamic Arrays(2/2)

- Dynamic Array
  - size not specified at programming time (can grow and shrink as needed)
  - determined while program running

Creating Dynamic Arrays

- Use **new** operator
  - dynamically allocate with pointer variable
  - treat like standard array

Example:

```
int size = 0;
cin >> size;
double  *ptr;
ptr = new double[size]; // contain size elements
                        // of type double
```

Deleting Dynamic Arrays

- Allocate dynamically at run-time
  - so should be destroyed at run-time
- Continue the previous example

```
……
ptr = new double[size];
…… //some processing
delete [] ptr;
```

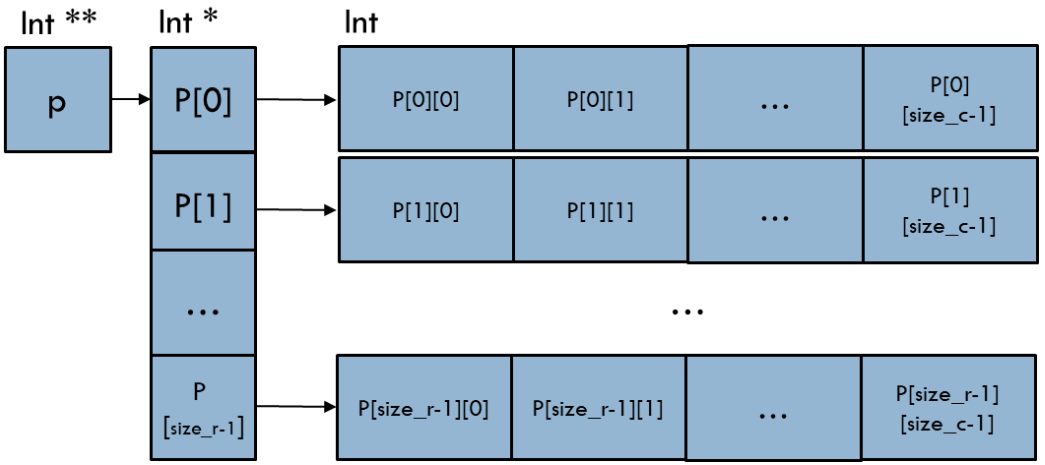
  - de-allocate all memory for dynamic array
  - brackets [] indicate array is there
  - note that ptr still points there => dangling!
    - should add “ptr = NULL;” immediately

Dynamic Multi-dimensional Arrays

- Multi-dimensional arrays are arrays of arrays
  - Create a size_r * size_c dynamic array

Example:

```
int **Array2D = new int*[size_r]; // every row has size_c coloumn
for(int i = 0; i < size_r; i++)
    *(Array2D+i) = new int[size_c];
    //or Array2D[i] = new int[size_c];
```



Delete Dynamic Arrays

- Clean reversely from last allocated memory

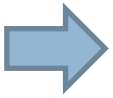
Example:

```
for(int i = 0; i < size_r;  i++)
    delete [] Array2D[i];
delete [] Array2D;
Array2D = NULL;
```

valgrind

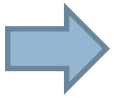
- valgrind --leak-check=full -s --show-leak-kinds=all --track-origins=yes ./binary

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

int main(){
    int *a = new int[20];
    return  0;
}
```



```
[coherent17@NVL4 ~/tttt]$ valgrind --leak-check=full -s --show-leak-kinds=all --track-origins=yes ./a
==3803819== Memcheck, a memory error detector
==3803819== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==3803819== Using Valgrind-3.19.0 and LibVEX; rerun with -h for copyright info
==3803819== Command: ./a
==3803819==
==3803819==
==3803819== HEAP SUMMARY:
==3803819==     in use at exit: 80 bytes in 1 blocks
==3803819==   total heap usage: 2 allocs, 1 frees, 72,784 bytes allocated
==3803819==
==3803819== 80 bytes in 1 blocks are definitely lost in loss record 1 of 1
==3803819==    at 0x4C38B6F: operator new[](unsigned long) (vg_replace_malloc.c:640)
==3803819==    by 0x4006F7: main (in /home/vdalab/coherent17/tttt/a)
==3803819==
==3803819== LEAK SUMMARY:
==3803819==    definitely lost: 80 bytes in 1 blocks
==3803819==    indirectly lost: 0 bytes in 0 blocks
==3803819==      possibly lost: 0 bytes in 0 blocks
==3803819==    still reachable: 0 bytes in 0 blocks
==3803819==         suppressed: 0 bytes in 0 blocks
==3803819==
==3803819== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

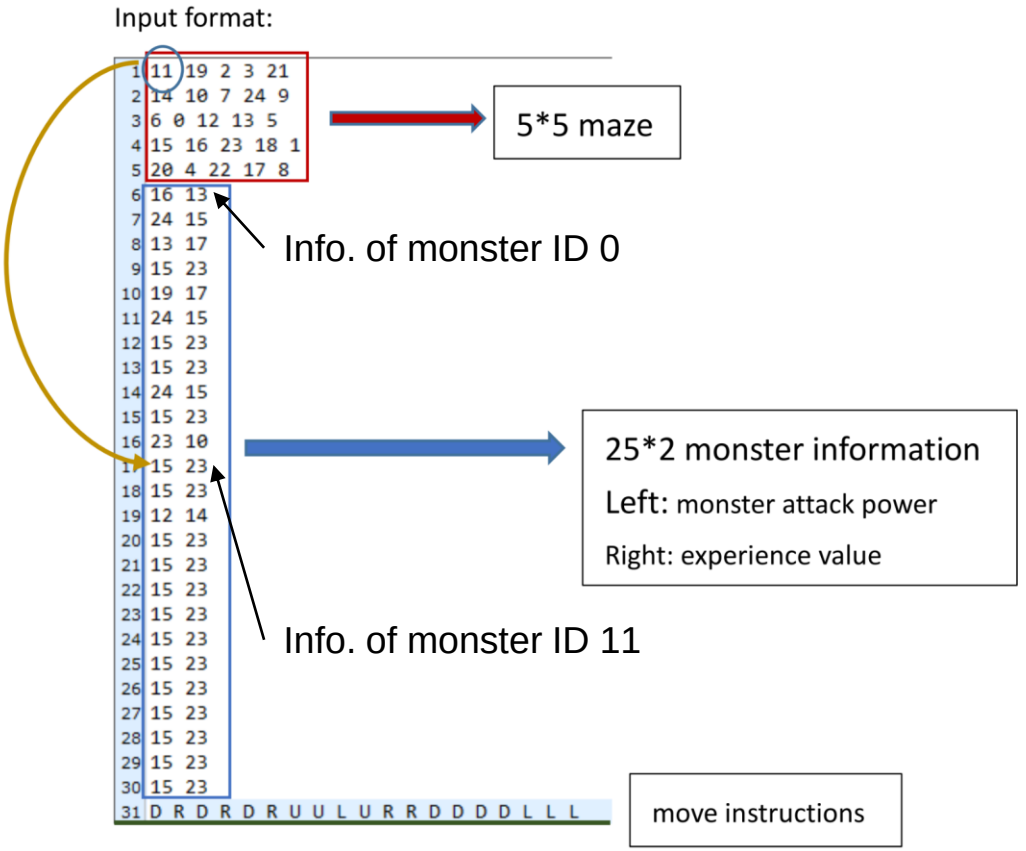
```cpp
#include <iostream>
using namespace std;

int main(){
    int *a = new int[20];
    delete []a;
    a = nullptr;
    return  0;
}
```



```
coherent17@NVL4 ~/tttt]$ g++ a.cpp -o a
coherent17@NVL4 ~/tttt]$ valgrind --leak-check=full -s --show-leak-kinds=all --track-origins=yes ./a
=3804053== Memcheck, a memory error detector
=3804053== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
=3804053== Using Valgrind-3.19.0 and LibVEX; rerun with -h for copyright info
=3804053== Command: ./a
=3804053==
=3804053==
=3804053== HEAP SUMMARY:
=3804053==     in use at exit: 0 bytes in 0 blocks
=3804053==   total heap usage: 2 allocs, 2 frees, 72,784 bytes allocated
=3804053==
=3804053== All heap blocks were freed -- no leaks are possible
=3804053==
=3804053== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Exercise

- Read 5*5 maze, 25*2 monster information and some move instructions from a file by argv (./Lab-03.o 1.in)
  - The number in the 5*5 maze is monster's ID.
  - The first element in monster information is **attack power** and the second is **experience value**
  - The ID of the monster corresponds to this 25*2 matrix (start from 0)
  - **U** stands for up, **D** stands for down, **L** stands for left, **R** stands for right



- You should create the maze and the monster information by dynamic array

Exercise

- You will play a player to explore in this maze
  - Initial HP is 100
  - Initial experience value is 0
  - Initial level is 1
  - Initial position is at (0, 0)
  - HP will be deducted from the monster's attack power
  - EXP will be added to the experience value
  - EXP reaches 100, the player will be upgraded
  - When the player is upgraded, HP will return to 100 and the EXP will return to 0
  - If the monster's ATK is greater than the player's HP, output "DEAD" and end the program
  - There is no need to calculate the position (0,0) at the beginning
  - The movement command will not exceed the maze, so there is no need to judge that it is beyond the boundary
- Output the status in each instruction by **cout**
- The output format is as follows.

output format:

```
step 1:
level:0 hp:85 exp:23

step 2:
level:0 hp:62 exp:33

step 3:
level:0 hp:46 exp:46

step 4:
level:0 hp:31 exp:69

step 5:
level:0 hp:16 exp:92

step 6:
level:1 hp:100 exp:0

step 7:
level:1 hp:88 exp:14

step 8:
level:1 hp:73 exp:37

step 9:
level:1 hp:58 exp:60

step 10:
level:1 hp:45 exp:77

step 11:
level:1 hp:30 exp:100

step 12:
level:2 hp:100 exp:0

step 13:
level:2 hp:85 exp:23

step 14:
level:2 hp:61 exp:38

step 15:
level:2 hp:37 exp:53

step 16:
level:2 hp:13 exp:68

DEAD
```

Exercise

- Create a directory “OOP112” (mkdir OOP112)
- Change your working directory to “OOP112” (cd OOP112)
- Create a cpp file “Lab-03.cpp” (touch Lab-03.cpp)
- Write your code in Lab-03.cpp
- The objection of this exercise is to **practice dynamic array**
  - You should use **new** operator and **delete**
  - Check that you are successfully freeing the memory by **Valgrind**
  - TA will check your code
- Use the following command to demo:

  /home/share/demo_OOP112 Lab 03

Submission

- Ask TAs for demo
- Try your best to debug your code by yourself
- Upload all your **.cpp** to new E3
- Naming rule : studentID_lab3.cpp