Lecture 6

Pointers and Dynamic Memory Allocation

Topics covered by last two lectures

- Objects and classes
- Abstraction, encapsulation and data hiding
- Class declaration, definition and application
- Constructor
- Destructor
- class composition

```cpp
#include <iostream>
using namespace std;
const int SIZE = 10;

void fillArray(int a[],int size){
  for (int i = 0; i < size; i++)
  a[i] = rand() % 100;
}

void printArray(int a[],int size){
  for (int i = 0; i < size; i++)
  cout << a[i] << " ";
}

int main() {
  int arr[SIZE];
  srand(time(0));

  fillArray(arr, SIZE);
  printArray(arr, SIZE);

  return 0;
}
```

Procedural style

```cpp
#include <iostream>
using namespace std;
const int SIZE = 10;

class ooArray {
public:
  void fillArray(int a[], int size){
  for (int i = 0; i < size; i++)
    a[i] = rand() % 100;
  }

  void printArray(int a[], int size){
    for (int i = 0; i < size; i++)
      cout << a[i] << " ";
  }
};

int main() {
  int arr[SIZE];
  srand(time(0));

  ooArray oo;
  oo.fillArray(arr, SIZE);
  oo.printArray(arr, SIZE);

return 0;
}
```

Fake OO style

```cpp
#include <iostream>
using namespace std;
const int SIZE = 10;

class ooArray {
public:
  void fillArray(int a[], int size){
  for (int i = 0; i < size; i++)
    a[i] = rand() % 100;
  }

  void printArray(int a[], int size){
    for (int i = 0; i < size; i++)
      cout << a[i] << " ";
  }
};

int main() {
  int arr[SIZE];
  srand(time(0));

  ooArray oo;
  oo.fillArray(arr, SIZE);
  oo.printArray(arr, SIZE);

return 0;
}
```

Fake OO

```cpp
#include <iostream>
using namespace std;
const int SIZE = 10;

class ooArray {
  int a[SIZE];
  int size;
public:
  ooArray() {size = SIZE;}
  void fillArray() {
    for (int i = 0; i < size; i++)
    a[i] = rand() % 100;
  }

  void printArray() {
    for (int i = 0; i < size; i++)
    cout << a[i] << " ";
  }
};

int main() {
  srand(time(0));

  ooArray oo;
  oo.fillArray();
  oo.printArray();

  return 0;
}
```

Real OO

Object Oriented Programming Style

```
#include <iostream>
using namespace std;
const int SIZE = 10;

class ooArray {
  int a[SIZE];
  int size;
public:
  ooArray() {size = SIZE;}
  void fillArray();
  void printArray();
};

void ooArray::fillArray() {
  for (int i = 0; i < size; i++)
  a[i] = rand() % 100;
}

void ooArray::printArray() {
  for (int i = 0; i < size; i++)
  cout << a[i] << " ";
}
```

Myoo.h

```
int main() {
  srand(time(0));

  ooArray oo;
  oo.fillArray();
  oo.printArray();

  return 0;
}
```

MyooApp.cpp

Well organised OO style

Class communication in composition

```
class Board {
   char grid[BOARDSIZE][BOARDSIZE];
public:
   bool addMove(int x1,int y1, int x2,int y2);
   bool checkWin();
   bool validInput(int x, int y);
   void printBoard();
};
```

```
class Player {
protected:
   int playerType;
public:
   void getMove(Board b,int& x,int& y);
   int getType();
};
```

```
class Game {
   Board board;
   Player player[2];
public:
   void play();
};
```

A game consists of a game board and two players.

```
void Game::play() {
   while(!board.checkWin()) {
      int x1, y1, x2, y2;
      player[0].getMove(board, x1, y1);
      player[1].getMove(board, x2, y2);
      board.addMove(x1,y1,x2,y2);
   }
}
```

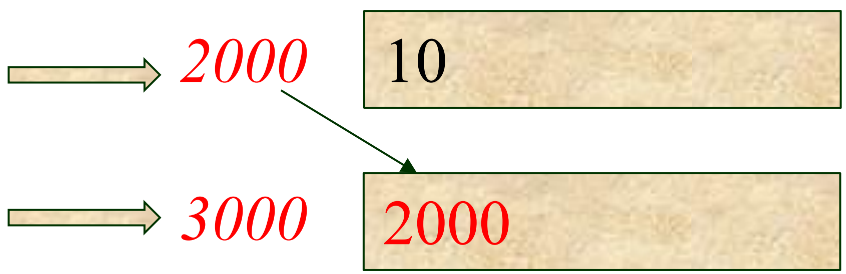
Topics of this lecture

- Pointers
- Pointers and arrays
- Pointers and functions
- new operator
- Dynamic memory allocation
- Copy constructor

Pointers

- Pointer is a variable that stores the address of a memory cell in certain data type.

```
int a =10;
int *pa;
pa = &a;
```



Declaration of pointers

- Declare pointers of built-in data types:

```
int *ptr1;
double *ptr2;
string *ptr3;
```

```
int* ptr1;//either way
```

- Declare pointers of user-defined data types:

```
Office *o;
LectureTheater *lt;
Kitchen *k;
Board *board;
```

01001100	10111001	01101010

Note that a pointer only points to the first memory cell of the data. Therefore we need to know the type of data.

Address operator &

Ooh, now I understand call-by-reference

- The "*address of*" operator, &, returns the address of a variable

```
int value = 10;
int *ptr;
ptr = &value;
```

```
TicTacToe board;
TicTacToe *boartPtr;
boardPtr = &board;
```

- Assign right address to right pointer: *match types*.

```
int value = 10;
int *ptr1;
double *ptr2;
ptr1 = &value; //legal
ptr2 = &value; //illegal
```

Dereference operator *

pointer.cpp

- The dereference operator * obtains the data item (can also be an *l-value*) the pointer points to.
  - `*p` means "the variable that `p` points to"

Use the pointer to get the value

Assign directly

```
int value;
int *ptr;
ptr = &value;
value = 100;
cout << *ptr << endl;
*ptr = 200; /
 cout << value << endl;
```

ptr | 2000 | → value *2000* | 100

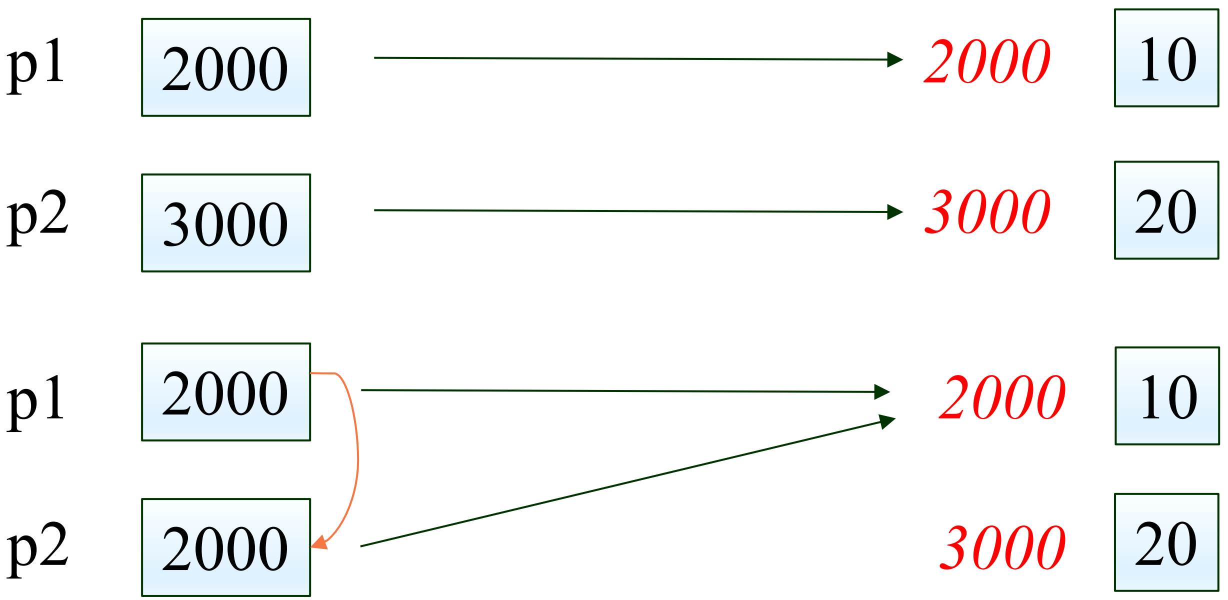
Assign to the variable ptr point to

ptr | 2000 | → value *2000* | 200

& and * are dual operators

Pointer Assignments

- Pointer variables can be "assigned":
  ```
  int *p1, *p2;
  p2 = p1;
  ```
  assign address
  - Assigns one pointer to another
  - "Make p2 point to where p1 points"



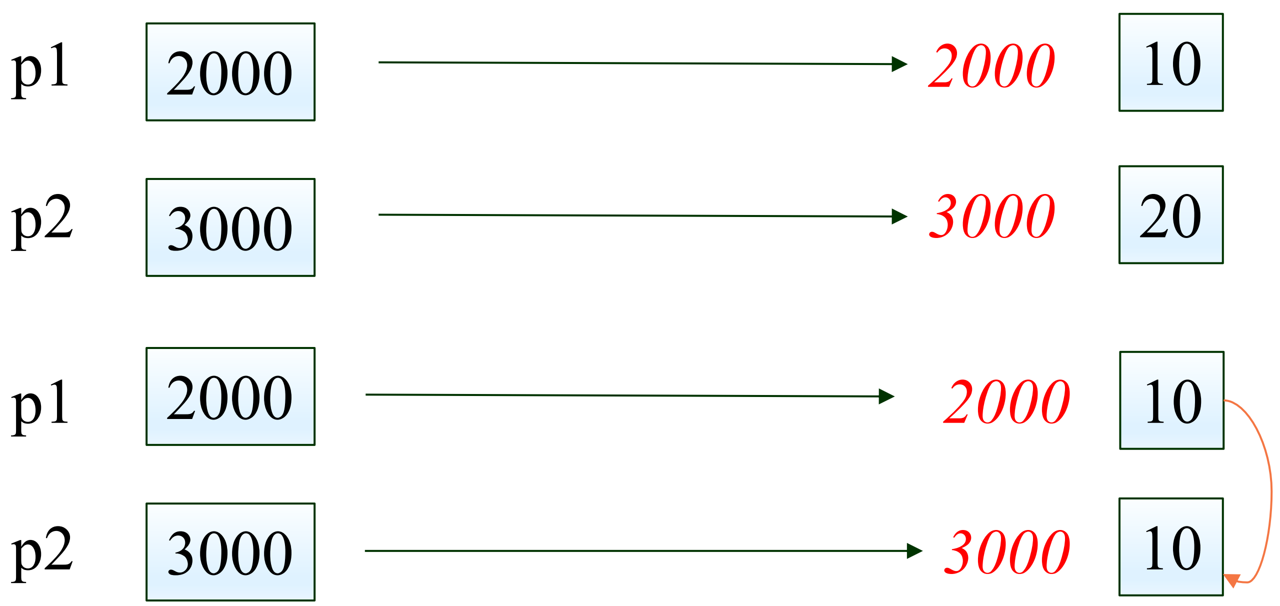
Pointer Assignments

- Pointer variables can be "assigned":

```
int *p1, *p2;
*p2 = *p1;
```

assign value

  - Get the value of the variable p1 point to
  - Assign its value to the variable p2 point to



Initialise pointers and pointer offset

- A pointer can be initialized by NULL, which indicate that the pointer points to nothing.

```
int *ptr = NULL;
```

- Pointer can be offset to another address:

```
cout << *(ptr+1) <<endl;
ptr = ptr +1;
ptr++;
```

Pointers and Functions

- An address can be passed to a function via a pointer
- A function can also return an address via a pointer
- Example:

return an address via a pointer

pass an address via a pointer

```
int* functionPointer(int* ptr);
```

  – A pointer as a formal argument. Note that the actual argument must be either a pointer variable of the same type or an address of the type.
  – If a function returns a pointer, you need to create a pointer of the same type in the calling module to catch the value of the function.

functionPointer.cpp

Array and Pointers

- An array name is a pointer, pointing to the first memory cell of the array:

```
int a[] = {1,2,3,4,5};
cout << *a << endl;
```

arrayPointer.cpp

- Can perform arithmetic on pointers
  - "Address" arithmetic

```
double d[10];
```

  - d contains the address of d[0]: *d is equivalent to d[0].
  - d+1 contains the address of d[1]: *(d+1) is equivalent to d[1].
  - d+2 contains the address of d[2]: *(d+2) is equivalent to d[2].
    - » *Address oriented* vs *value oriented*

Return an array by returning a pointer

- Array type is NOT allowed as return-type of a function.

Example:

```
int[] someFunction();    // ILLEGAL!
```

- Instead return a pointer of the array:

```
int* someFunction();   // LEGAL!
```

arrayFunction.cpp

The new Operator

newOperator.cpp

- What's wrong with the following program?

```
int main() {
      int *ptr;
      *ptr = 20;
      cout << *ptr << endl;
}
```

You have memory to store an address of an integer

You do not have memory to store an integer value

- Operator *new* creates memories (dynamic memory allocation)

```
int *ptr;
ptr = new int;
*ptr = 20;
```

ptr is the only indicator for the memory cell

  - The new operator creates a new memory cell of integer and returns the address of the memory cell
  - You can access the memory cell using a pointer

Memory Management

- Dynamically-allocated memories are from the "*freestore*", also called "heap"
- You use new operator to request a memory cell from the freestore.
- If your request is succeeded, you will receive an address with the allocated memory.
- The heap is limited. Once it is full, future new operations will fail. In this case, new will return a NULL.
- You release memory using the "delete" operator.
- The technology is called *dynamic memory allocation*.

Dynamic Object Creation

- Create an object with *new* operator
  - *MyType *fp = new MyType;*
  - *MyType *fp = new MyType(1,'a');*
- Free an object with *delete* operator
  - *delete fp;*
- Allocate an array of objects with *new[]* operator
  - *MyType* fparr = new MyType[100];*
- Free a list of objects with delete[] operator
  - *delete[ ] fparr;*

Dynamic Object Creation

- Two ways to create variables/objects

```
int value; //get memory to store an integer
int *ptr; //get memory to store an integer pointer
ptr = new int; //get memory to store an integer
delete ptr; // return memory

Board board;   //get memory to store a Board object

Board *boardPtr;   //get memory to store a Board pointer
boardPtr = new Board;   //get memory to store a Board object

delete boardPtr; // return memory
```

The -> operator and pointer of pointers

Call a data member or member function from a pointer:

```
Date *p;
p = new Date;
(*P).showdate();
```

Equivalently, we write:

```
p->showdate();
```

```
Board board;
board.play(); //Call a method from an object

Board *boardPtr;
boardPtr = new Board;
boardPtr->play(); //Call a method from an object pointer
```

Dynamic array

You may try to use vector, which is a dynamic array template.

```
class DynamicArray {
private:
    int* darray;
    int size;
public:
    DynamicArray() {
        cout << ”Input the size of the array:" << endl;
        cin >> size;
        darray = new int[size];
    }

   void input() {
       for (int i = 0; i < size; i++)  cin >> darray[i];
   }

   void output() {
       for (int i = 0; i < size; i++) cout << darray[i] <<
endl;
   }

   ~DynamicArray() {  delete[ ] darray; }
};
```

You do not have to know the size of the array at the time you program.

dynamicArray.cpp
arrayFunction.cpp
10-09.cpp

two-dimensional dynamic array

int* grid[10]; // array that stores 10 integer pointers
int** grid; // grid is an array of arrays

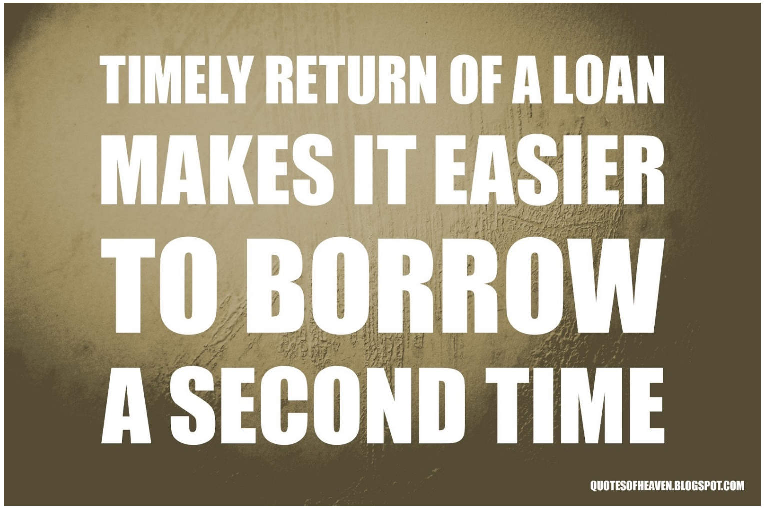
Return an array

creatArray.cpp

```
int* createArray(int size) {
    int* temp = new int[size];
    for (int i =0; i < size; i++)
        temp[i] = 0;
    return temp;
}

int main( )
{
    int* a = createArray(1000000);
    for (int i = 0; i < 1000000; i++)
        cout << a[i] << " ";
    delete[] a;
    return 0;
}
```

Borrow memory



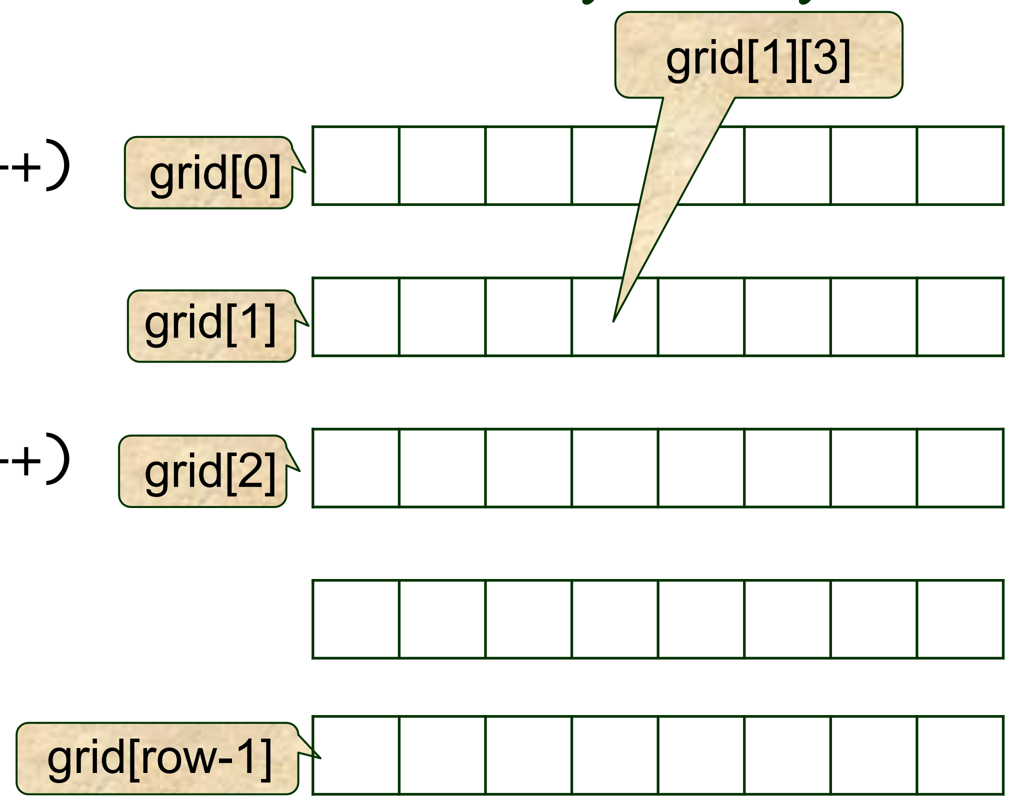
Return memory

Two-dimensional array

- A two-dimensional array is an array of arrays thus the name of a two-dimensional array is a pointer of pointers
- To create a 2D array, you must create an array of pointers to store the addresses of rows (a set of one-dimensional arrays).
- To delete a 2D array, you also need to delete an array of arrays

```
int **grid = new int*[row];
for (int i = 0; i < row; i++)
   grid[i] = new int[col];

for (int i = 0; i < row; i++)
    delete[] grid[i];
delete[] grid;
```



When do we need a pointer?

- Dynamic memory allocation requires a pointer to access the money
- Sometime pass the address of an object is more efficient and safer than pass the object itself.
- You can return an array from a function if the array is created using dynamic memory allocation.
- You can even pass a function to another function because the function name is a pointer points to the code of the function.
- Sometimes, inheritance requires to use pointers to avoid object slicing (Lecture 10).
- Other chances to use pointers. The more you use pointers, the more you love them.

Homework

- Read textbook Chapter 10.
- Complete practical 4 if you haven't. The practical contains more training for assignment 1.
- Work on practical 5