Pointer & Dynamic Allocation

Modern cpp Programming lecture 2

While solving the problems in prerequisites…

- Defining & accessing variable were quite easy
  - define
    - `int number = 2;`
    - `bool trueOrFalse[5] = [true, false, false, true, false];`
  - access
    - `cout << number << endl  // 2`
    - `cout << trueOrFalse[2] << “, “ << trueOrFalse[3] << endl  // 0, 1`
  - parameter
    - `int a = 2; int b = 2;`
    - `sum(a, b) // 4`
- However, this ***simple method*** has ***lots of limitations!!***

Limitation of *simple method*

- Locality of variables
  - Declaration of variables => valid inside the function (or block)
  - How the program not only *read* the value,
  - but also ***refer*** the variable in ***other functions***??
- Inefficient for large data parameter
  - You'll understand later…
    - after you study "Class" and "Objects"!!

Limitation of simple methods

- Declaration is static
  - cannot determine the size of data in runtime
  - sometimes wastes resources!!
  - `cin >> n;  int array[n];` => impossible
- Inefficient Memory usage
  - Most important
  - You'll see the detail…

Pointer & Dynamic allocation

- Solves every problem we noticed
- One of the most powerful tool in programming
- Manage & Control Memory space freely
- Ables efficient cpp programming!!
- However…
  - hard to understand
  - vulnerable to error (if you’re novice programmer)
  - Lot’s of modern Programming Languages hide pointer
- But that’s why you should learn them!!

Pointer

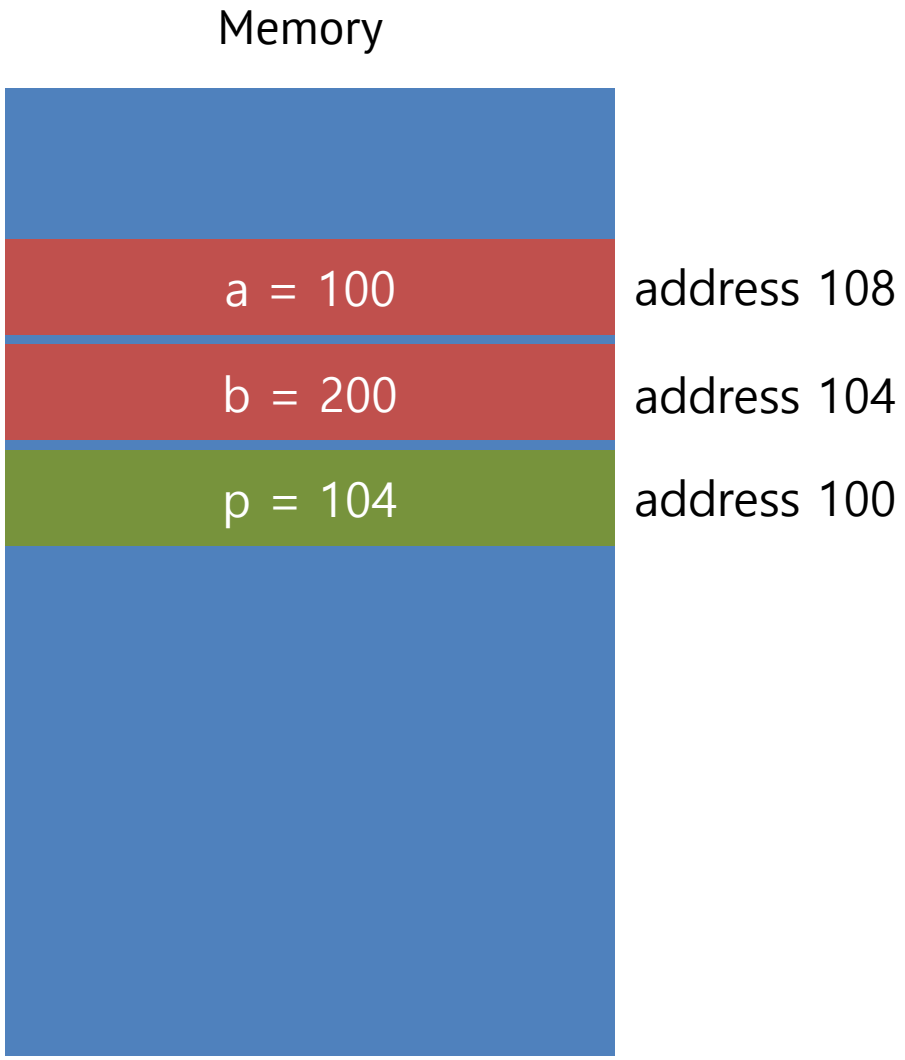
- Pointer is *data type*
  - like int, char, string, bool…
  - which stores the **address** of data
  - address?
    - if you use data, it means that the data is stored in somewhere
    - *Pointer **points*** the location!!
    - ex)
      - `int*` : points the address of int-type data
      - `char*` : points the address of char-type data
      - `void*` : points the address of any data

Pointer

- Address control in cpp
  - key character: * and &
  - * usage
    - declaring pointer : `int* pointer;`
    - dereference operator
      - access the value stored in the address where the pointer is pointing
  - & usage
    - access the address of data
      - `int num = 100; int* pointer = &num;`
      - now `pointer` stores the address of `num`

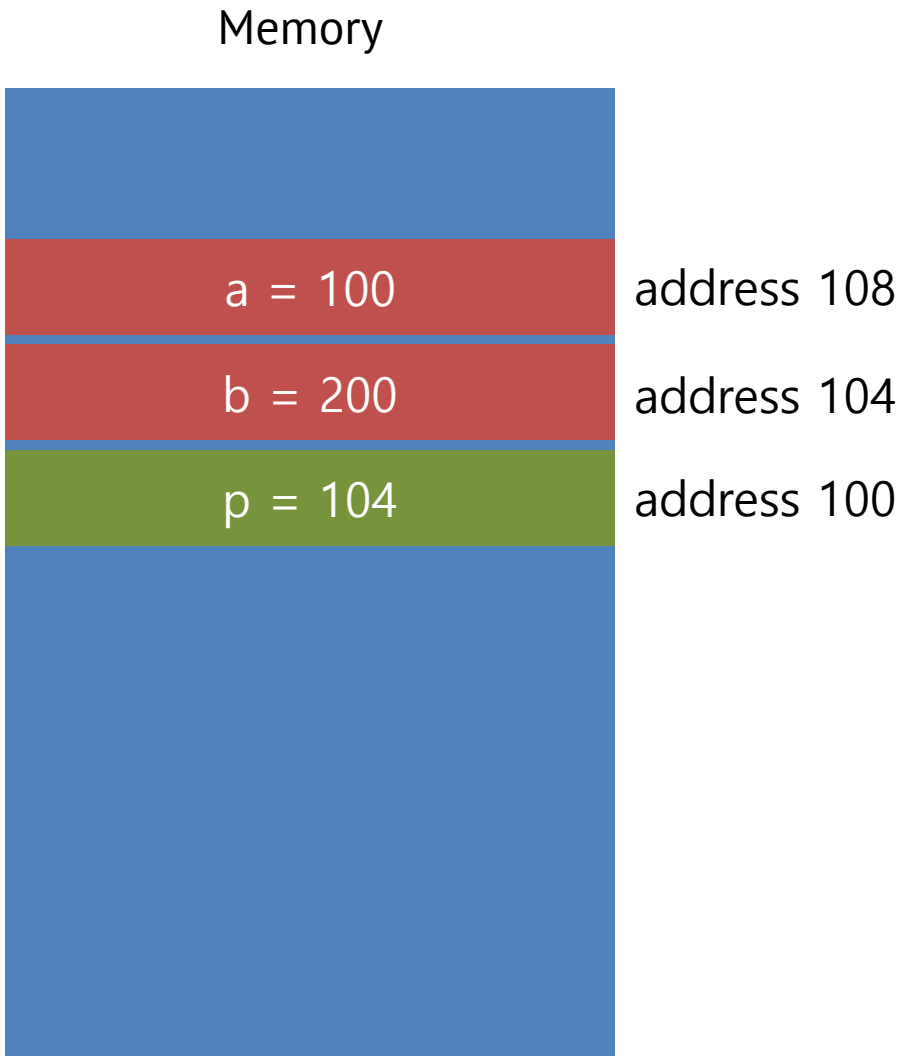
Pointer

```
int a = 100;
int b = 200;
int* p = &b;

cout << a << endl; // 100
cout << b << endl; // 200
cout << p << endl; // 104
```



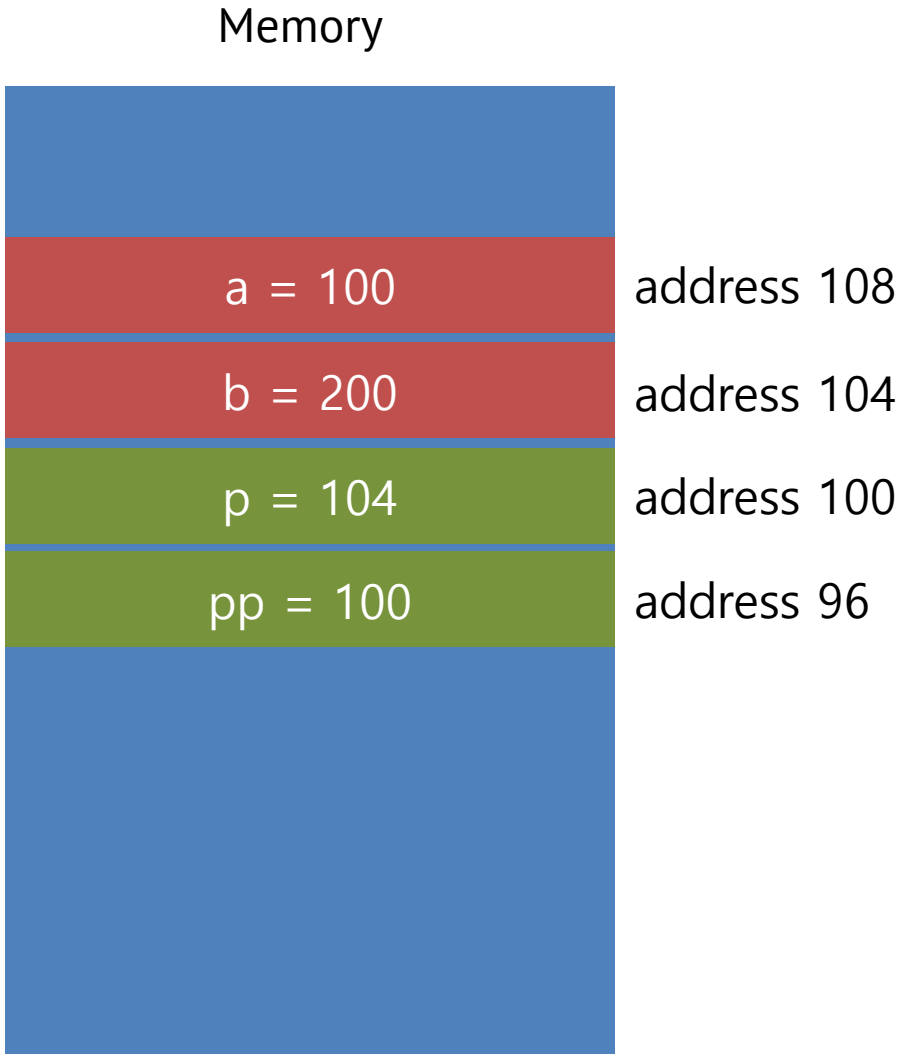
Pointer

```
int a = 100;
int b = 200;
int* p = &b;

cout << a << endl; // 100
cout << b << endl; // 200
cout << p << endl; // 104

cout << &a << endl; // 100
cout << &b << endl; // 104
cout << *p << endl; // 200
```



Pointer

```
int a = 100;
int b = 200;
int* p = &b;
int** pp = &p;
(pointer which points int*)

cout << pp << endl; // 100
cout << *pp << endl; // 104
cout << **pp << endl; // 200
```



Pointer

- Call by value vs. Call by address (pointer)

Call by value

```
void changeVal(int val) {
    val = 20;
    cout << val << endl;
    return;
}

int num = 10;
changeVal(num);
cout << num << endl;
```

```
output:
20
10
```

Call by address

```
void changeVal(int* val) {
    *val = 20;
    cout << *val << endl;
    return;
}

int num = 10;
changeVal(&num);
cout << num << endl;
```

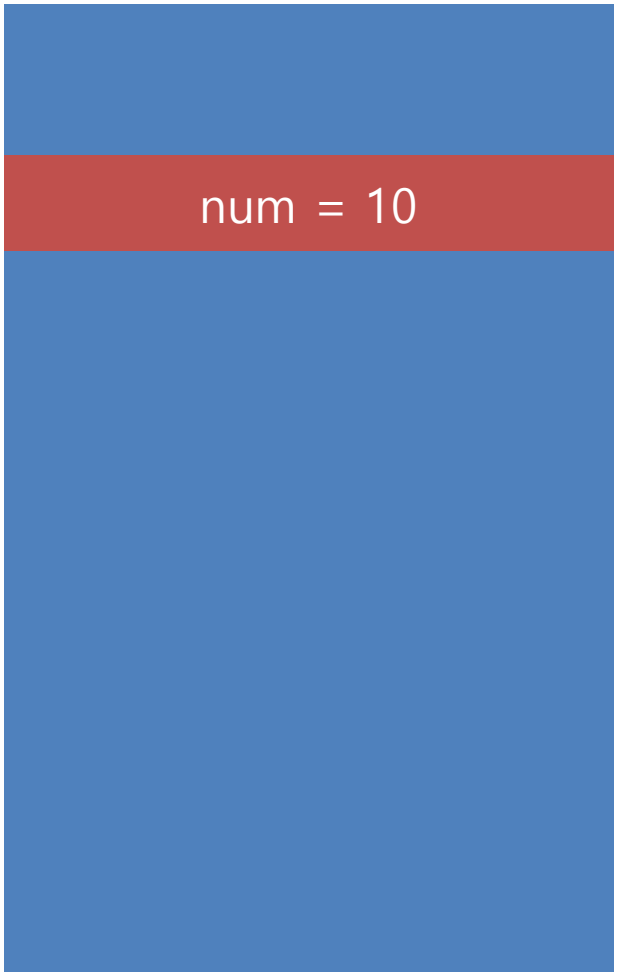
```
output:
20
20
```

- Any difference??

Call by Value

```
void changeVal(int val) {
    val = 20;
    cout << val << endl;
    return;
}

int num = 10;
changeVal(num);
cout << num << endl;
```

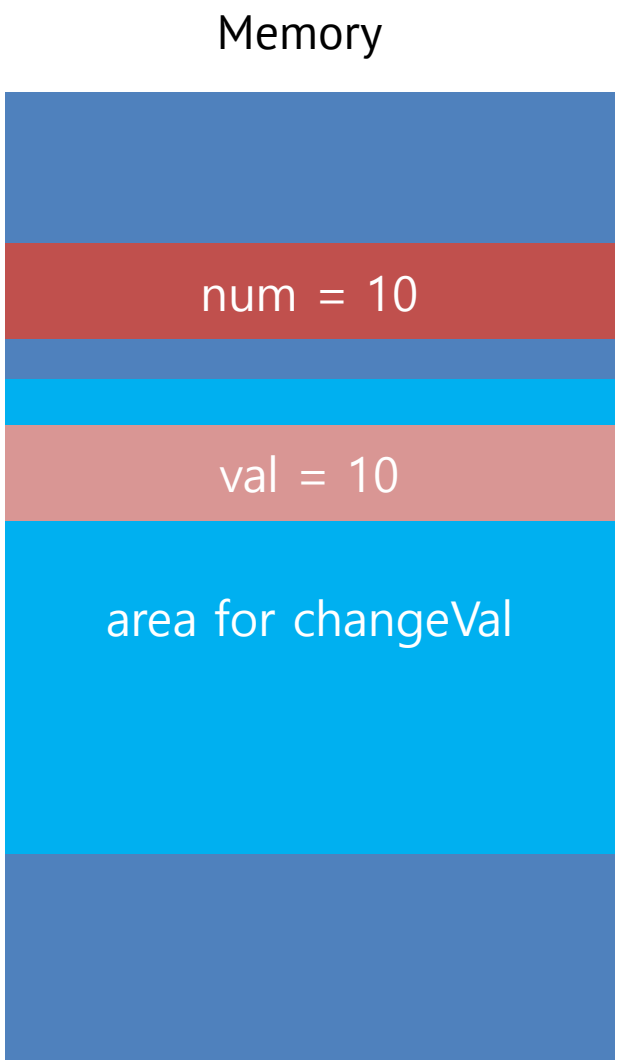
output:



Call by Value

```
void changeVal(int val) {
    val = 20;
    cout << val << endl;
    return;
}

int num = 10;
changeVal(num);
cout << num << endl;
```

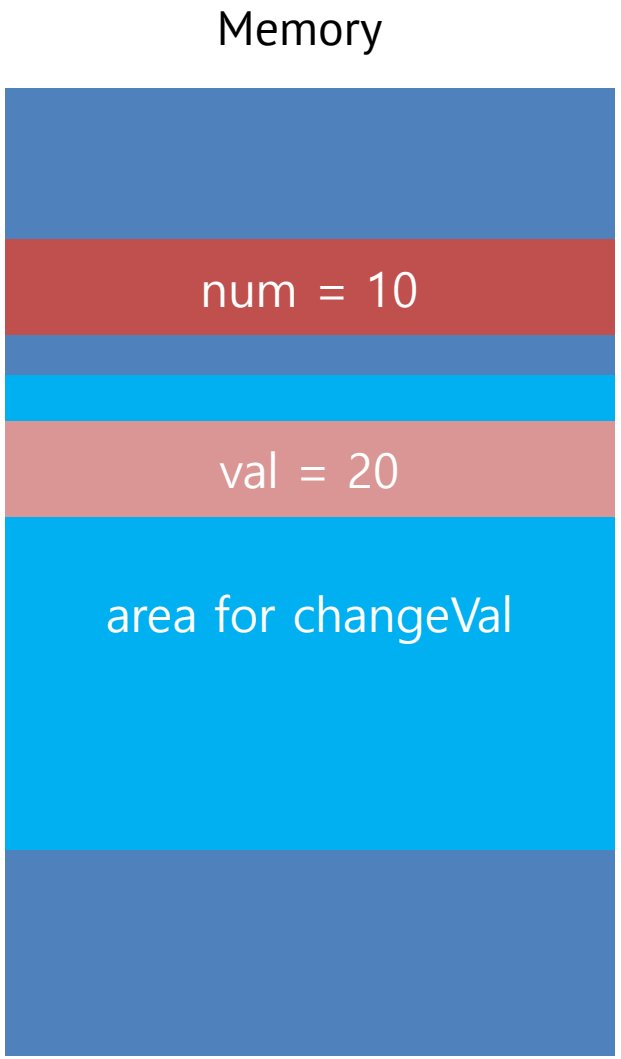
output:



Call by Value

```
void changeVal(int val) {
    val = 20;
    cout << val << endl;
    return;
}

int num = 10;
changeVal(num);
cout << num << endl;
```

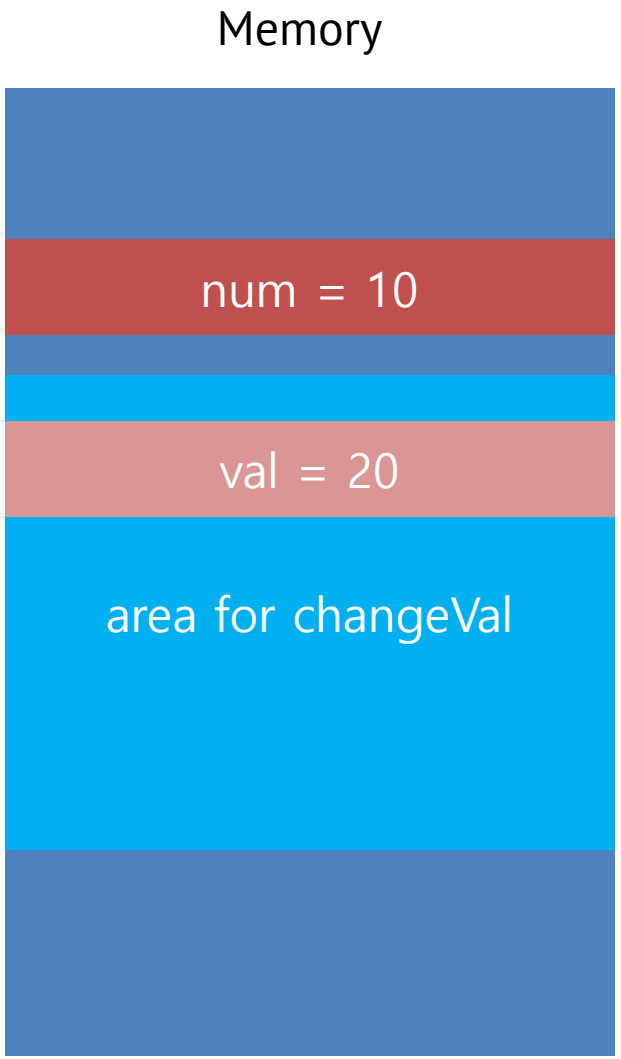
output:



Call by Value

```
void changeVal(int val) {
    val = 20;
    cout << val << endl;
    return;
}

int num = 10;
changeVal(num);
cout << num << endl;
```

```
output:
20
```



Call by Value

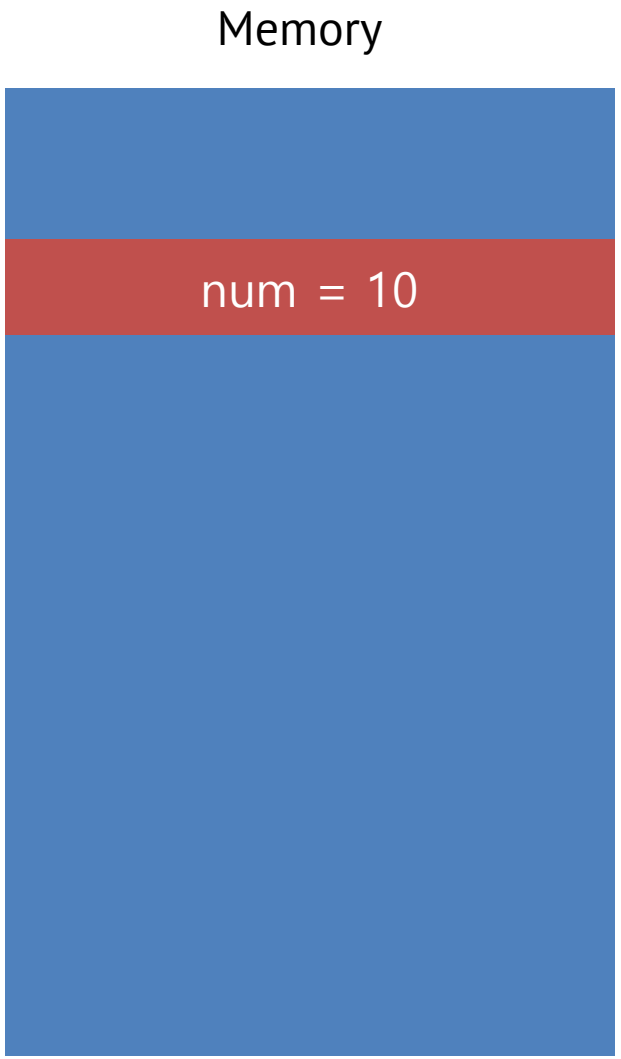
```
void changeVal(int val) {
    val = 20;
    cout << val << endl;
    return;
}

int num = 10;
changeVal(num);
cout << num << endl;
```

```
output:
20
```

Call by Value

```
void changeVal(int val) {
    val = 20;
    cout << val << endl;
    return;
}

int num = 10;
changeVal(num);
cout << num << endl;
```

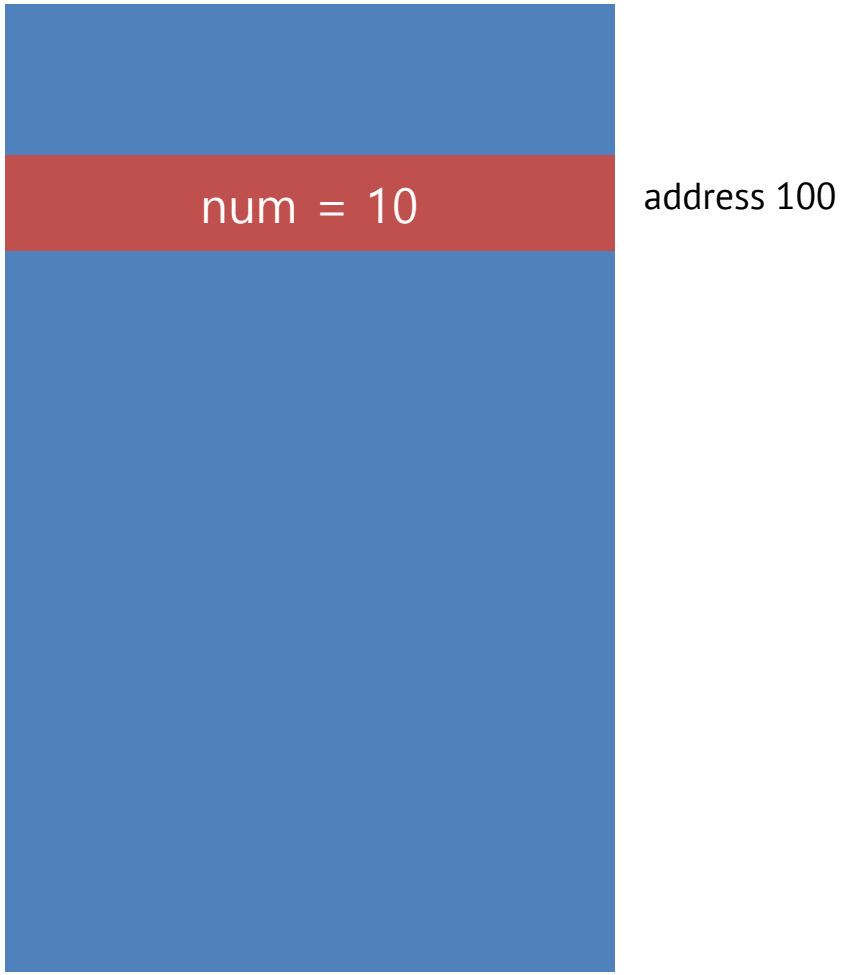
```
output:
20
10
```



Call by Address

```
void changeVal(int* val) {
    *val = 20;
    cout << *val << endl;
    return;
}

int num = 10;
changeVal(&num);
cout << num << endl;
```

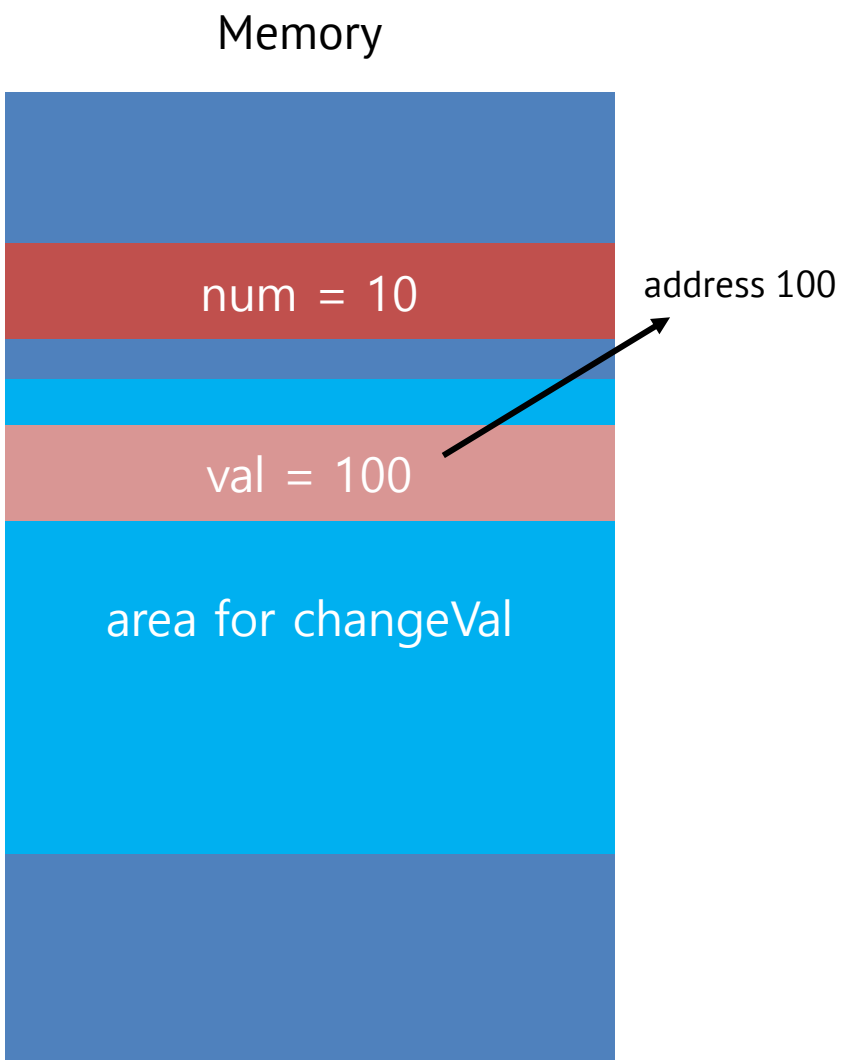
output:



Call by Address

```
void changeVal(int* val) {
    *val = 20;
    cout << *val << endl;
    return;
}

int num = 10;
changeVal(&num);
cout << num << endl;
```

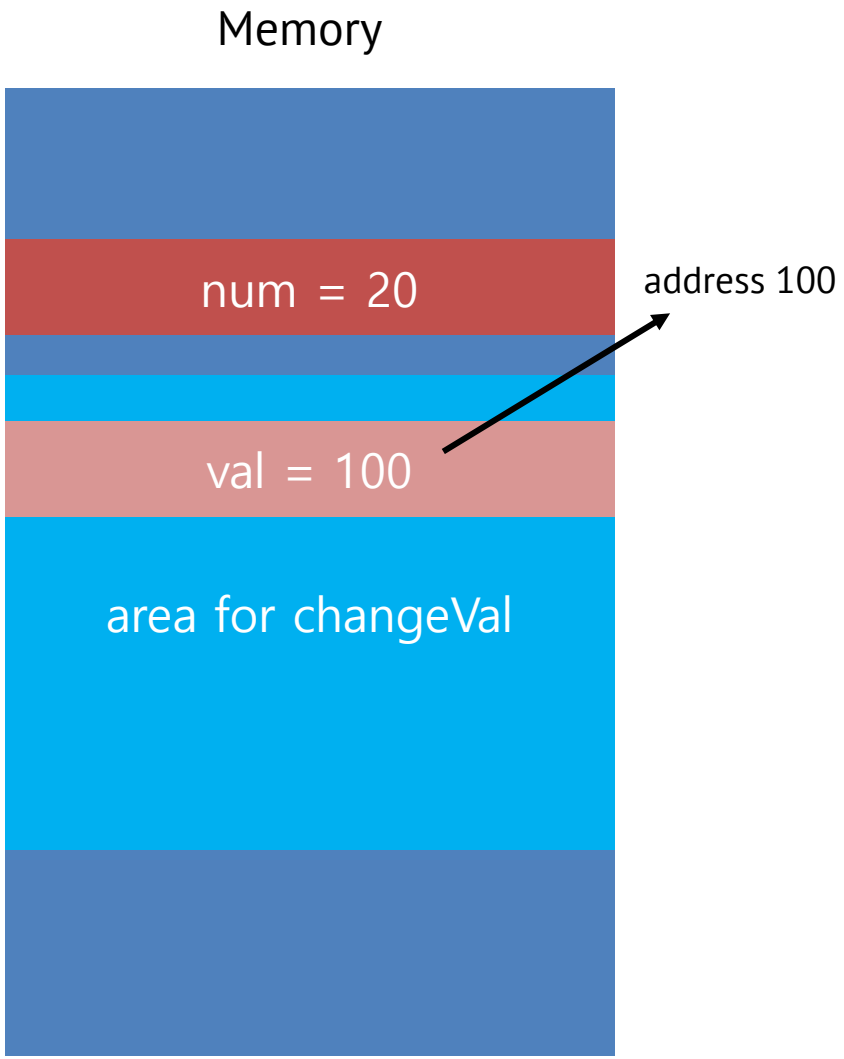
output:



Call by Address

```
void changeVal(int* val) {
    *val = 20;
    cout << *val << endl;
    return;
}

int num = 10;
changeVal(&num);
cout << num << endl;
```

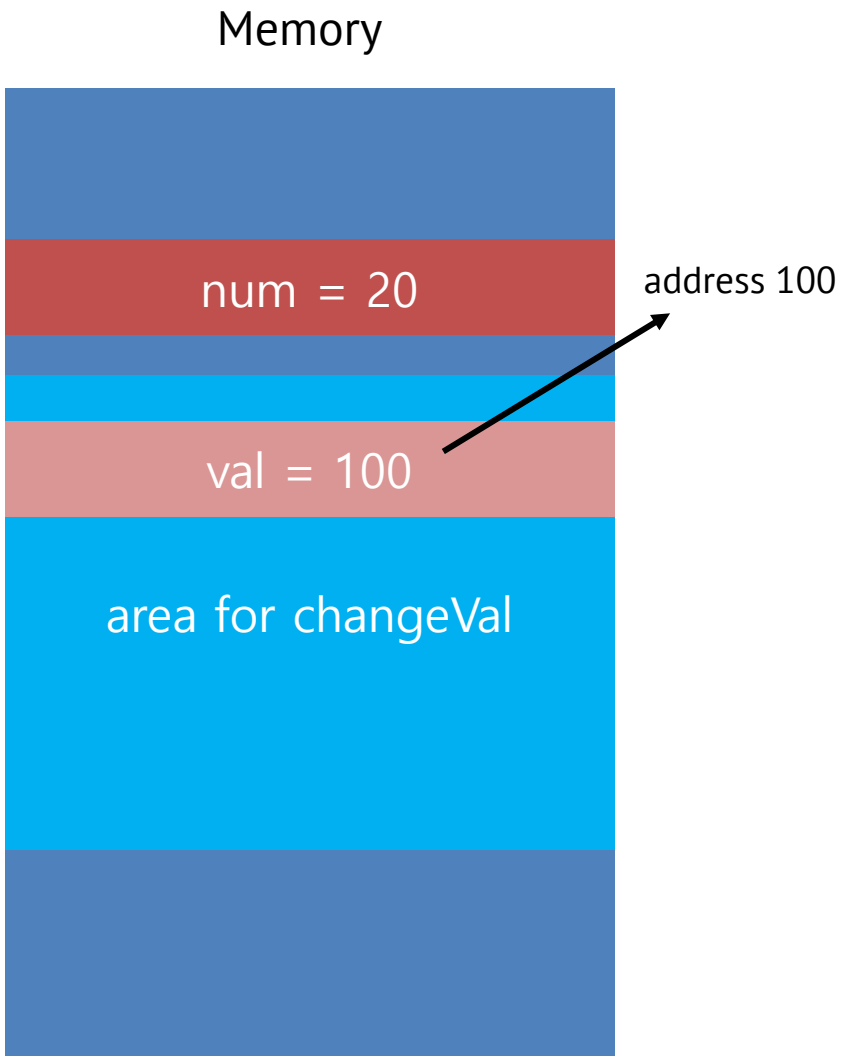
output:



Call by Address

```
void changeVal(int* val) {
    *val = 20;
    cout << *val << endl;
    return;
}

int num = 10;
changeVal(&num);
cout << num << endl;
```

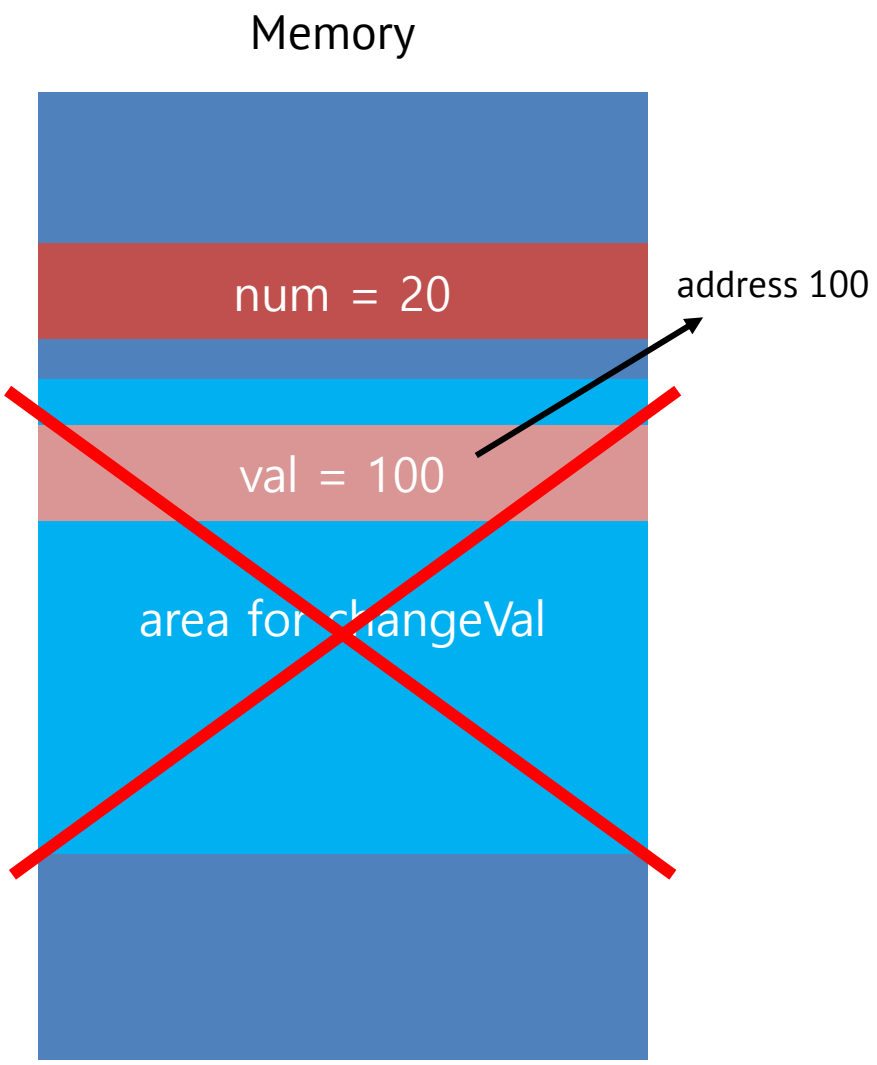
```
output:
20
```



Call by Address

```
void changeVal(int* val) {
    *val = 20;
    cout << *val << endl;
    return;
}

int num = 10;
changeVal(&num);
cout << num << endl;
```

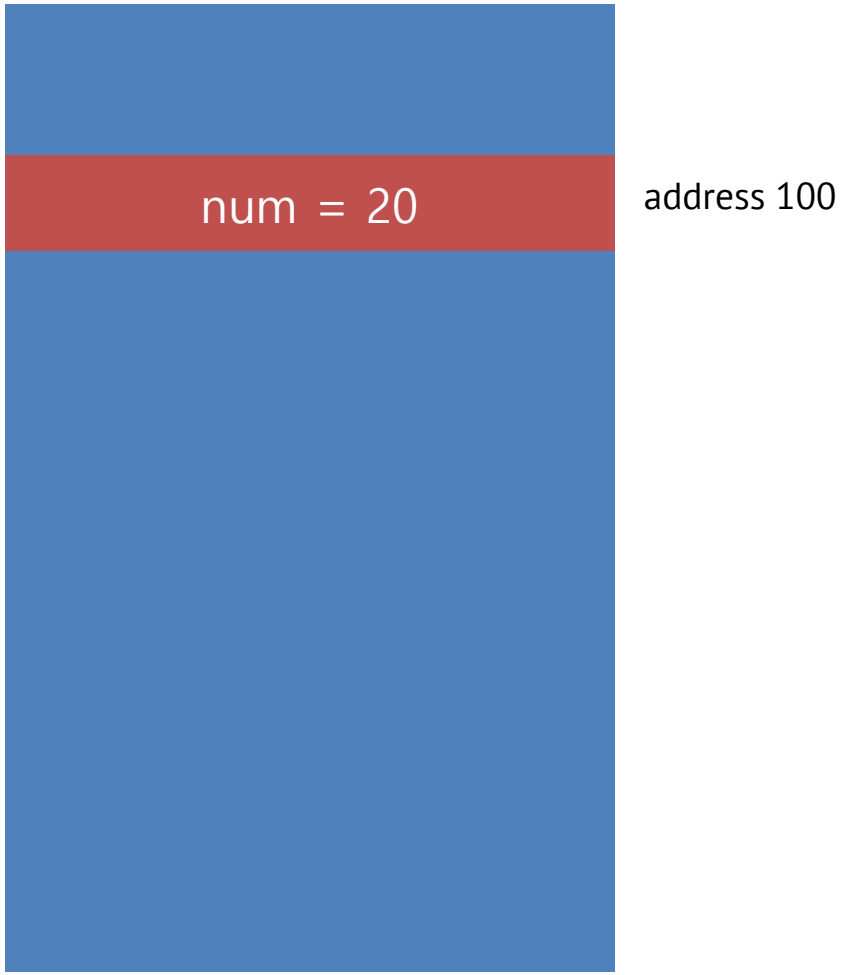
```
output:
20
```



Call by Address

```
void changeVal(int* val) {
    *val = 20;
    cout << *val << endl;
    return;
}

int num = 10;
changeVal(&num);
cout << num << endl;
```

```
output:
20
20
```



Call by address - Application

- Swap

Call by value	Call by address

`void swap(int a, int b) {`
`  int temp = a;`
`  a = b;`
`  b = temp;`
`  cout << a << ", " << b << endl;`
`  return;`
`}`
 	
`int a = 10; int b = 20;`
`cout << a << ", " << b << endl;`
`swap(a, b);`
`cout << a << ", " << b << endl;`
	`void swap(int* a, int* b) {`
	`  int temp = *a;`
	`  *a = *b;`
	`  *b = temp;`
	`  cout << *a << ", " << *b << endl;`
	`  return;`
	`}`
	
	`int a = 10; int b = 20;`
	`cout << a << ", " << b << endl;`
	`swap(&a, &b);`
	`cout << a << ", " << b << endl;`

output: 10, 20 20, 10 10, 20	output: 10, 20 20, 10 20, 10

Tip

- Cout printed Very very very strange number!!!
- Mostly due to uninitialized pointer

```
int* a;
cout << a << endl;      // 0x10fcbc036
cout << *a << endl;     // -125990072
```

  - Wrong.. but no explicit error occurs
  - If your program is complex, impossible to find this bug!!
  - Solution: initialize pointer with NULL (0)

```
int* a = NULL;
cout << a << endl;
cout << *a << endl;

Output:
0x0
[1]    10619 segmentation fault
```

  - Now you can find where the error occurred!!

Array and pointer

- Recall array!!

```
int array[100];
for(int i = 0; i < 100; i++) {
   array[i] = 99 - i;
}
cout << array[77] << endl;   // 22
```

  - name of array : pointer to the first element!!
  - {name of array} + i : pointer to the i-th element!!

```
cout << *array << endl;          // 99
cout << *(array + 65) << endl;   // 34
```

Array and Pointer

- Sending arrays as parameters

```
int sumElement(int i, int j, int* array) {
   int result = array[i] + array[j];
   return result;
}

int array[100];
for(int i = 0; i < 100; i++) {
   array[i] = 99 - i;
}
cout << sumElement(30, 40, array) << endl;   // 128
```

- Nice and convenient!!

Array and Pointer

- 2D array??

```
int array[100][100];
for(int i = 0; i < 100; i++) {
   for(int j = 0; j < 100; j++) {
      array[i][j] = i * j;
   }
}
cout << array[30][40] << endl;     // 1200
```

- Similar to 1D array!!
  - **array points (*array)[0], which points array[0][0]

```
cout << *(*(array + 60) + 70) << endl; // 4200
cout << (*(array + 60))[70] << endl;   // 4200
cout << array[60][70] << endl;         // 4200

cout << *(*(array + 40) + 50) << endl; // 2000
```

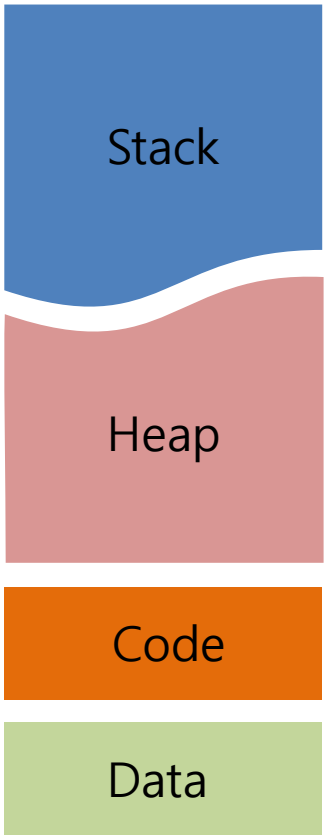
Pointer

- Finished!!
  - Now you know everything about pointer
  - master pointer!!
  - Then there will be no obstacles to learn other programming lang uages!!
- However, pointer itself isn't powerful that much
  - Now it's time to learn Dynamic allocation!!

Dynamic Allocation

Cpp Memory structure

- If you run your program, OS allocates memory for the program



Stores local variables / parameters at function call
memory size is determined at compile time

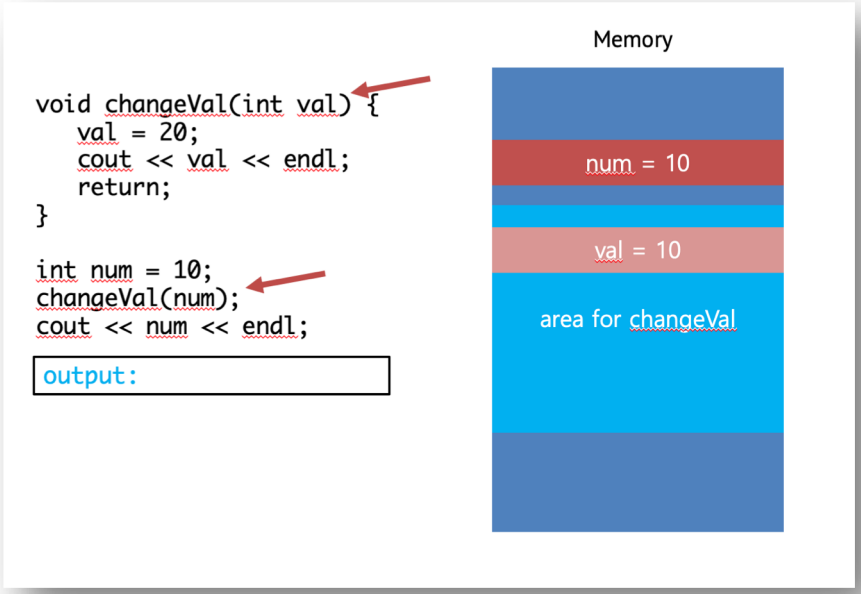
Programmers can allocate memory dynamically,
whenever they need (Dynamic allocation)
memory size determined at run time

Stores program machine code

Stores static / global variable

Cpp Memory structure

- Recall the slide:



- Everything happening in this example, actually happens in:
- ***STACK*** area

Cpp Memory Structure

- So…isn't stack area sufficient for programming?
  - Theoretically, yes!!
  - Practically, no!! we need Heap & dynamic allocation
  - Why??
  - Why we need Dynamic allocation??

Dynamic allocation

- Why?
  1. Live up to it's name
     - Remember!! stack size is determined at compile time
     - Therefore, cpp cannot perform:

```
int n;
cin >> n;
int array[n] = {0};
```

     – as compiler cannot determine the size of *array*!!
     – Solution: dynamic allocation

```
int n;
cin >> n;
int *array = new int[n];
```

       » As heap stores dynamic allocated memory, the size of the stack can be determined at compile time
       » stack stores integer n, pointer *array*

Dynamic allocation

- Why?

  2. Saves memory

```
int array[5000000000];
```

  - Segmentation fault!! => exceed limited stack size

```
int* array = new int[5000000000];
```

  - no error
  - Huge heap area
    - But you should strictly manage dynamic allocation!!

Dynamic allocation

- Cautions
  - Dynamic allocation is not a magic!!
  - Should be used only when needed
    - `int* num = new int[1]; *num = 1;`
    - `int num = 1;`
    - second one is much better!!
  - Deallocate memory if you no longer need it
    - Otherwise…
    - Memory leak!!
      - Derives serious performance problem!!

Dynamic allocation

- Syntax
  - new / delete
    - allocate / deallocate variable
  - new[] / delete[]
    - allocate / deallocate arrays

Dynamic allocation

- Syntax
  - `new int[100]`
    - Allocate memory which can store 100 ints in heap area and return the address
  - `int* arr = new int[100]`
    - Allocate memory which can store 100 ints in heap area and assign it's adress to pointer variable `arr`
  - `delete[] array`
    - delete the allocated array memory in the location where `array` is pointing at.

Dynamic allocation

- Syntax
  - `new int`
    - Allocate memory which can store int in heap area and return the address
  - `int* pointer = new int`
    - Allocate memory which can store int in heap area and assign it's address to pointer variable `pointer`
  - `delete pointer`
    - delete the allocated memory in the location where `pointer` is pointing at.

Allocating 2D array

- Simple example!!

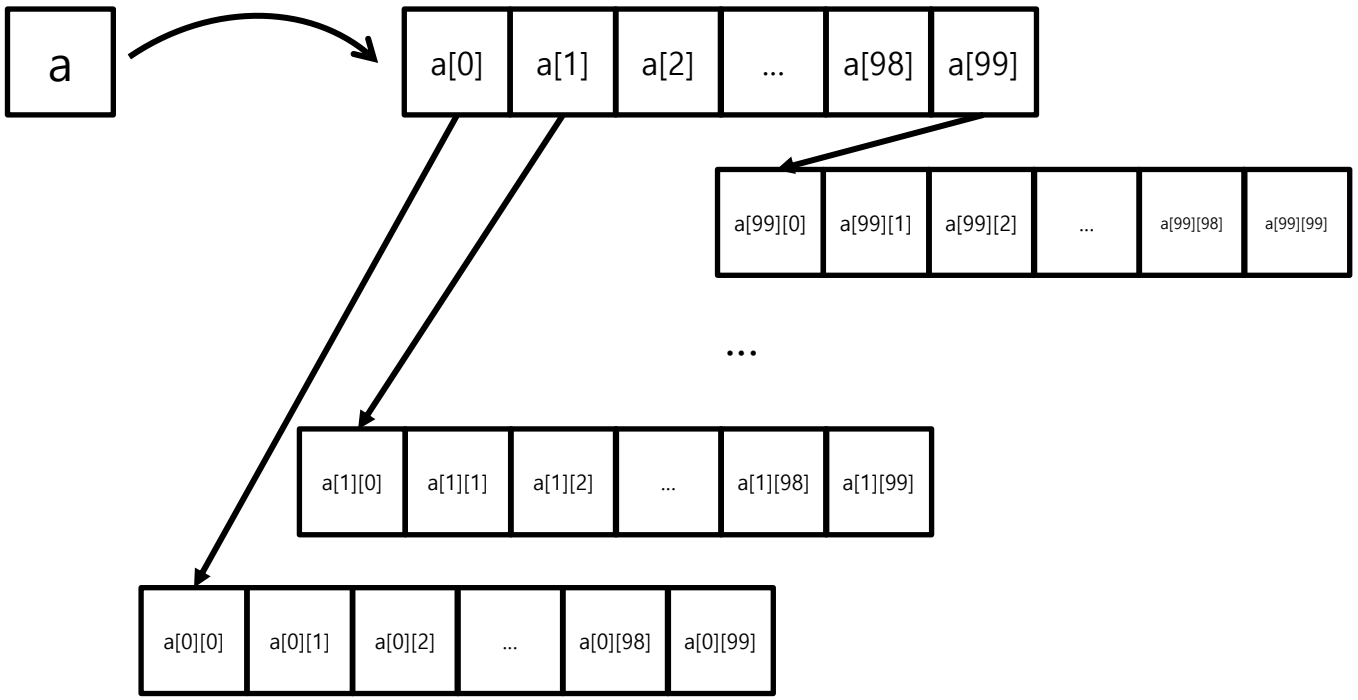
```
int** a;
a = new int*[100];
for(int i = 0; i < 100; i ++) {
   a[i] = new int[100];
}
```

- Result: 100 * 100 array which can be easily accessed by:
  - int** a
  - which is pointing the array consists with 100 `int*`
  - where each `int*` variable is pointing `int[100]`

Allocating 2D array

- Simple example!!

```
int** a;
a = new int*[100];
for(int i = 0; i < 100; i ++) {
   a[i] = new int[100];
}
```



Summary

- Pointer & Dynamic Allocation
  - powerful programming technique
  - which provides freedom to programming
  - however, vulnerable to errors…
- You will recognize the importance
  - after studying the concepts of OOP (Object-Oriented Programming)

Thank you!!

contact: jeonhyun97@postech.ac.kr