# Review: Pointers & Memory Allocation

## 1. Pointers

Let `int y;` then `&y` means the memory direction of y.

```
1 char c;
2 int* pc;
3 pc = &c; // error
```

```
1 char c;
2 char* pc;
3 pc = &c // no error
```

- **Declaration**

```
1 int x;
2 int* px; // declaration
```

- **Initialization**

Initialization of a pointer is when in its declaration, you also assign its value, which is a memory address.

```
1 int* px = &x;
```

- **Assignment**

After a pointer declaration, you can assign a memory address as the pointer value.

```
1 px = &x;
```

- **Indirection**

Indirection is accessing to the value of the variable the pointer is pointing at.

```
1 cout << *px;
```

#### Pointers To Pointers

```
1 float y = 2.5;
2 float *py = &y; // needs to be float because y is float //
     simple pointer
3 float **ppy = &py; // mem address of the pointer py, needs to
     be float because of y
4 float ***pppy = &ppy; // triple pointer, not so common
5 cout << ***pppy; // 2.5
```

### Type of Pointers

- **void pointers**

Also called a generic pointer, they do not have a defined value of the variable they will point at, most of the times because the variable type they will point at is unknown. Void Pointers can point to whatever type of value.

```
char c;
int x;
void* pc;

pc = &c;
pc = &x; // all valid
```

- **null pointers**

A null pointer does not point to any direction, does not point to trash.

```
char c;
char* p = NULL;
```

- **constant pointers (int *const)**

```
int x = 4;
int* const px = &x; // pointer cannot change its memory
   address (but x can change), and must be initialized, else
   error
cout << *px; // 4 (print the value of x through a pointer)
*px = 8; // x or *px can change
cout << *px; // 8
// error px = &y;
```

- **pointers to constants (const int*)**

```
const int x_const = 12;
// error int* p3 = &x_const;
const int* p3 = &x_const; // no error, also this pointer can
   point to different const int variables // error if int * p3
    = &const_var
*p3 = 11; // error bc you're changing x_const or p3
const int y_const = 10;
p3 = &y_const; // p3 can change its target, but its variable
   type is const int
```

### Pointers To Arrays

The name of an array is a pointer to its first element.

```
int arr[3] = {5,7,9};
int *p = arr; // same as p = &arr[0]
cout << *p; // 5
cout << p[0]; // 5
cout << *(p + 1); // 7, bc we are advancing one memory cell
cout << *(p++); // 7
```

### Exercise 1

Code a function with no return value that receives three pointers to vectors (arrays) of integers.The function must add the vectors (v1[1] + v2[1]). For the implementation use:

- 2 arrays initialized with 5 elements each.
- 1 array to store the sum.
- 3 pointers that point to these vectors.

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

void func(int* p1, int* p2, int* pr) {

    for (int i = 0; i < 5; i++) {
        pr[i] = p1[i] + p2[i];
    }

    return;
}

int main()
{
    int a1[5] = { 1,2,3,4,5 };
    int a2[5] = { 6,7,8,9,10 };
    int r[5] = { 0 };

    int* pr = r;
    int* p1 = a1;
    int* p2 = a2;

    func(p1, p2, pr);

    for (int i = 0; i < 5; i++) {
        cout << pr[i] << " "; // 7, 9, 11, 13, 15
    }

    for (int i = 0; i < 5; i++) {
        cout << r[i] << " "; // 7, 9, 11, 13, 15
    }
```

```
32 }
```

### Pointers To Arrays of Pointers

It is posible to create arrays of pointers, and pointers that point to these arrays.

```
1 int v_int = 12;
2 int v_int2 = 3;
3
4 int *pt_array[3]; // array of pointers
5 int  **p_pt_array = pt_array; // pointer to array of pointers
6
7 pt_array[0] = &v_int;
8 pt_array[1] = &v_int2; // same as p_pt_array[1] = &v_int2
```

## 2. Dynamic Memory

The **new** operator in c++ is used to reserve memory in execution time, while the operator **delete** frees this memory.

- Syntaxis

```
1 void* new dataType;
2 void delete void* block;
3 void delete [] void* block;
```

- Examples

```
1 int* p;
2 p = new int;
3 *p = 10;
4 cout << *p; // 10
5 delete p;
```

```
1 int* p;
2 p = new int [10];
3 for (int i = 0; i < 10; i++){
4     p[i] = i;
5     cout << p[i] << "-";
6 }
7 delete [] p;
```

C's function `malloc()` is used to reserve memory in execution time, while the function `free()` frees this memory.

- Syntaxis

```
1 void* malloc(size_t size)
2 void free(void* block)
```

- `malloc()` returns a void* pointer and so we cast it as `(int *)`.

```
1 int *p;
2 p = (int*)malloc(sizeof(int)); // create a space in dynamic
     memory (exec mem) that is referenced by p pointer
3 *p = 45
4 cout<<*p;
5 free(p); // param is the name of the pointer that references
     to mem you want to free
```

```
1 int *p;
2 p = (int*)malloc(sizeof(int) * 10);
3 for(int i = 0; i < 10; i++){
4     p[i] = i;
5     cout << p[i] << "-";
6 }
7 free(p);
```

### Exercise 2

Code a function that has no return value, and that it receives two pointers to integers. The function must swap the values of the parameters that it receives. These values must be from user input and stored in dynamic memory. For the implementation, use:

- Dynamic memory
- Pointers

```
1  #include <iostream>
2  using namespace std;
3
4  void func(int *p1, int *p2) {
5
6      int aux;
7
8      aux = *p1;
9      *p1 = *p2;
10     *p2 = aux;
11
12     return;
13 }
14
15 int main()
16 {
17     int* v1 = (int*)malloc(sizeof(int));
```

```
18      int* v2 = (int*)malloc(sizeof(int));
19
20      cin >> *v1;
21      cin >> *v2;
22
23      func(v1, v2);
24      cout << "v1: " << *v1 << "\nv2: " << *v2; // swap values
            of v1 and v2
25  }
```

## Some Findings

```
1  bool* ptr;
2  *ptr = false;
3  if (ptr) {
4      // will always be true without *
5  }
```

```
1  bool* ptr = &true; // error
```

```
1  int a = 5;
2  int* b = &a;
3  int c = *b // c is a separate location with value 5
```

```
1  int arr[10];
2  int* p6 = &arr[6];
3  int* p0 = &arr[0];
4
5  cout << (int)p6 << " " << (int)p0; // 162328 162304 (24
       difference, 6 times 4(int))
6  cout <<"diff: " << p6 - p0; // 6 (pointers work on a space
       defined by the data type they point to)
```

```
1  int arr[10] = {3,6,9,12,15,18,21,24,27,30};
2  int* p0 = arr;
3
4  for (int i = 0; i < 10; i++) {
5      cout << (arr + i) << endl; // 10 addresses that differ by
           4 (int)
6      cout << *(arr + i) << endl; // all array nums
7      cout << p0 << " = " << *p0 << endl; // 10 addresses = each
           num
8      p0++;
9  }
```

```
1  char word[] = "hello!";
2  char* p = word;
3  char* p0 = &word[0];
```

```
char* p3 = &word[3];

cout << p << endl; // hello! // char pointers are especially treated as strings, thats why
cout << p0 << endl; // hello! // these prints dont show addresses
cout << p3 << endl; // lo!
```

```
// dynamic memory: when you know the size of memory only at run time, not at compile time
int size;
int* ptr;

cout << "Enter size: ";
cin >> size;

ptr = (int*)malloc(sizeof(int) * size);

for (int i = 0; i < size; i++) {
    cout << "Value: ";
    cin >> ptr[i];
}

for (int i = 0; i < size; i++) {
    cout << ptr[i] << " ";
}
```

```
int* ptr1;
int* ptr2;

ptr1 = (int*)malloc(10 * sizeof(int));
*ptr1 = { 0 }; // only sets ptr[0] = 0
for (int i = 0; i < 10; i++) {
    cout << ptr1[i] << " ";
}
// 0 -842150451 -842150451 -842150451 -842150451 -842150451 -842150451 -842150451 -842150451 -842150451
for (int i = 0; i < 10; i++) {
    ptr1[i] = i * 10;
}
cout << *(ptr1 + 5) << endl; // 50
```

## Passing in Funtions

There are 3 ways in which you can pass arguments (values) to functions:

- **Pass by Value**

This method copies the actual **value** of an argument into the parameter of the function,

where this parameter is only **local** to the function. Changes made to the parameter inside the function have no effect on the argument. By default, c++ uses this method of passing in functions. You create a new memory location with the value of the passed argument.

```
void passByValue(int a, int b){
    a++;
    b++;
}
int main(){
    int a = 2, b = 3;
    passByValue(a, b);
    cout << a << " " << b; // a = 2, b = 3
}
```

- **Pass by Reference**

This method copies the **reference** of an argument into the formal parameter. Inside the function, the reference is used to access the **actual argument** used in the call. Therefore, changes made to the parameter affect the passed argument. In this case, x is a new name given to a, and y is the new name for b. X is a new name for the same memory location named a.

```
void passByReference(int &x, int &y){
    x++;
    y++;
}
int main(){
    int a = 2, b = 3;
    passByReference(a, b);
    cout << a << " " << b; // a = 3, b = 4
}
```

- **Pass by Address**

Also named Pass by Pointer, this method copies the **address** of an argument into the formal parameter of a function. Inside the function, the **actual address** is used to access the actual argument used in the call. Therefore, changes made to the parameter affect the passed argument. To do this, argument pointers are passed to the function. You are passing the address of a to the inside of pointer x in the function argument.

```
void passByAddress(int* x, int* y){
    *x += 1;
    *y += 1;
}
int main(){
    int a = 2, b = 3;
    int* pa = &a, * pb = &b;
    passByAddress(&a, &b); // a = 3, b = 4
    passByAddress(pa, pb);
```

```
10      cout << a << " " << b; // a = 4, b = 5
11  }
```