Pointers

A pointer to a variable is the address in the memory of that variable.

```
int ten = 10;         // declare an integer variable
int *p = &ten;        // get a pointer to the variable


cout << p << end;     // print the pointer (memory address)


cout << *p << endl;   // print the value the pointer points to
                      // (dereferencing the pointer p)
```

&x	is the address of the variable x
*p	dereferences the pointer p (returns the value the pointer p points at)

Pointers

What's the difference between the following three assignments?

```
int x = 1;

// 1
    int *p1 = &x;

// 2
    int *p2;
    *p2 = &x;

// 3
    int *p3;
    p3 = &x;
```

Pointers

What's the difference between the following three assignments?

```
int x = 1;

// 1
    int *p1 = &x;

// 2
    int *p2;
    *p2 = &x;  //  error: (*p2) is an integer,
               //          not a pointer
// 3
    int *p3;
    p3 = &x;
```

Allocating in the heap

- if we need to allocate a lot of memory (for example large arrays)
- if we want to create and return a big object from a function, and making additional copies is not an option. (references do mitigate the issue in C++)

```
double *pd = new double; // allocated in the heap
                         // using the operator new

*pd = 1.234;    // assign some value

cout << *pd;    // 1.234

delete pd;      // release the memory
```

Allocating in the heap

One can combine memory allocation in the heap and a constructor call:

```
double *d = new double(1.234);
cout << *d + 1;
delete d;

MyClass obj(127, "ABC");
obj.print();

MyClass *obj2 = new MyClass(128, "DEFH");
(*obj2).print();

delete obj2;
```

The arrow operator (->)

Instead of `(*Obj).member`, you can write `Obj->member`

It is added for convenience, because pointers are very common, and this additional “syntactic sugar” makes the code more readable.

```
MyClass *obj2 = new MyClass(128, "DEFH");

(*obj2).print();

obj2 -> print(); // equivalent

delete obj2;
```

The following program crashes. Why?

```
#include <iostream>

using namespace std;

int f() {
  int data [10000000];
  data[0] = 1;
  return data[0];
}

int main() {
  f();
}
```

Dynamic array allocation

```
int n;
cin >> n; // ask the user to choose the size of the array


double *arr = new double[n]; // allocate the array
                             // in the heap

for(int i = 0; i < n; i++){
  arr[i] = i;
  cout << arr[i] << ' ';
}

delete[] arr; // deallocate it
```

The size of a dynamically array can be bigger than the size of an array you can allocate in the stack

Dynamically allocated 2D array

```
int n; cin >> n;

double **arr = new double* [n];
for(int i = 0; i < n; i++)
  arr[i] = new double[n];

for(int i = 0; i < n; i++) {
  for(int j = 0; j < n; j++) {
    arr[i][j] = i+j;  cout << arr[i][j] << '\t';
  }
  cout << endl;
}

for(int i = 0; i < n; i++)
  delete[] arr[i];

delete[] arr;
```

The following program crashes. Why?

```
#include <iostream>

using namespace std;

int f() {
  return f();
}

int main() {
  f();
}
```

The following program crashes. Why?

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

int f() {
  int *x = new int(123);
  while(*x >= 100) {
    x = new int;
    *x = 100 + rand() % 2;
  }
  return *x;
}

int main() {
  f();
}
```

Classes that themselves allocate in the heap

If a class or structure (for example in its constructor) allocates some memory in the heap, this memory should be deallocated when the object gets “destroyed” (e.g.) when the program execution leaves the scope where the varaible is defined

In C++, the user can create a so called destructor, the function that will be called automatically when the object passes out of scope.

Classes that themselves allocate in the heap

"Rule of 3": If a class defines one of the following it should probably explicitly define all three:

- destructor
- copy constructor
- copy assignment operator

In C++11, things got a bit crazier, and there is “Rule of 5”