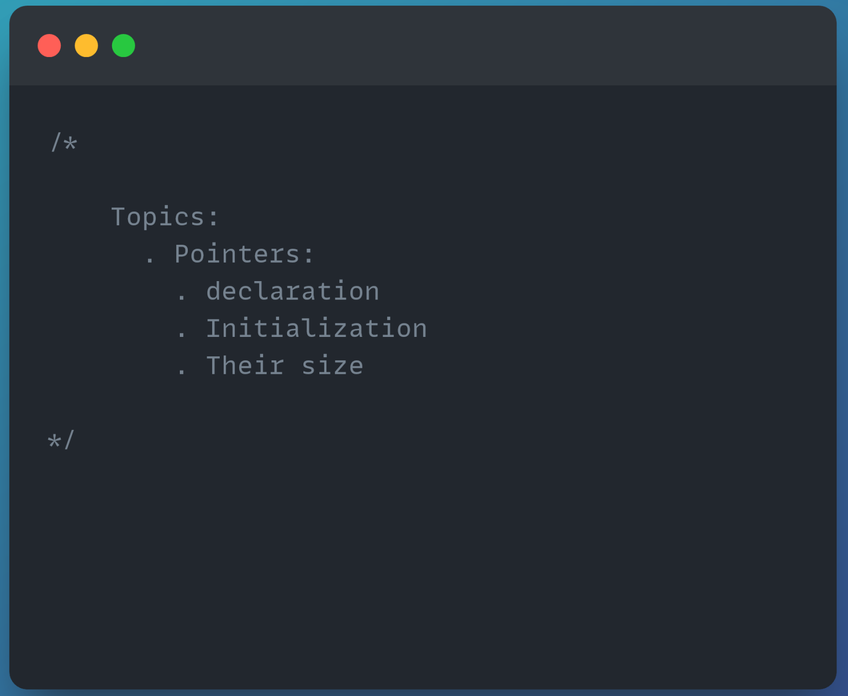
Dynamic memory allocation



```
/*

    Topics:
      . Pointers:
        . declaration
        . Initialization
        . Their size

*/
```

IMPORTANT	*You should rarely have to manually allocate memory like we're doing in this project. Modern C++ highly discourages that. You should mostly rely on RAII (Resource Acquisition Is Initialization) in your modern C++ projects.

How we've used pointers so far

```cpp
//new and delete
//How we've used pointers so far
int number {22}; // Stack
int * p_number = &number;

fmt::println("");
fmt::println( "Declaring pointer and assigning address: ");
fmt::println( "number: {}", number);
fmt::println( "p_number: {}", fmt::ptr(p_number));
fmt::println( "&number: {}", &number);
fmt::println( "*p_number: {}", *p_number);


int * p_number1; // Uninitialized pointer , contains junk address
int number1 {12};
p_number1 = &number1; // Make it point to a valid address
fmt::println( "Uninitialized pointer: " );
fmt::println( "*p_number1: {}", *p_number1 );
```

Uninitialized pointers are BAD!

```
// Writing into uninitialized pointer through dereference
int *p_number2; // Contains junk address: could be anything
fmt::println( "Writting in the 55" );
*p_number2 = 55; // Writing into junk address : BAD!
fmt::println( "p_number2 : {}", fmt::ptr(p_number2)); // Reading from junk address.
fmt::println( "Dereferencing bad memory" );
fmt::println( "*p_number2 : {}", *p_number2 );
```

Initializing pointers to null

```cpp
// Initializing pointer to null
//int *p_number3{nullptr}; // Also works
int * p_number3 {}; // Initialized with pointer equivalent of zero : nullptr
                                          // A pointer pointing nowhere
fmt::println( "Writting into nullptr memory" );
//*p_number3 = 33; // Writting into a pointer pointing nowhere : BAD, CRASH
fmt::println( "p_number3 : {}", fmt::ptr(p_number3));
//fmt::println( "*p_number3 : {}", *p_number3 );// Reading from nullptr
                                                //BAD, CRASH.
```

Dynamic memory

```cpp
// Dynamic heap memory
int *p_number4{nullptr};
p_number4 = new int;        // Dynamically allocate space for a single int on the heap
                            // This memory belongs to our program from now on. The system
                            // can't use it for anything else, untill we return it.
                            //After this line executes, we will have a valid memory location
                            // allocated. The size of the allocated memory will be such that
                            //it can store the type pointed to by the pointer


*p_number4 = 77; // Writting into dynamically allocated memory
fmt::println( "Dynamically allocating memory : " );
fmt::println("*p_number4: {}", *p_number4 );

//Return memory to the OS
delete p_number4;
p_number4 = nullptr;
```

Initialize dynamic memory at declaration

```cpp
// It is also possible to initialize the pointer with a valid
// address up on declaration. Not with a nullptr
int *p_number5{ new int };// Memory location contains junk value
int *p_number6{ new int(22) };// use direct initialization
int *p_number7{ new int{ 23 } };// use uniform initialization

fmt::println("");
fmt::println("Initialize with valid memory address at declaration : ");
fmt::println("p_number5: {}", fmt::ptr(p_number5));
fmt::println("*p_number5: {}", *p_number5);// Junk value

fmt::println("p_number6: {}", fmt::ptr(p_number6));
fmt::println("*p_number6: {}", *p_number6);

fmt::println("p_number7: {}", fmt::ptr(p_number7));
fmt::println("*p_number7: {}", *p_number7);

// Remember to release the memory
delete p_number5;
p_number5 = nullptr;

delete p_number6;
p_number6 = nullptr;

delete p_number7;
p_number7 = nullptr;
```

Initialize dynamic memory at declaration

```
// It is also possible to initialize the pointer with a valid
// address up on declaration. Not with a nullptr
int *p_number5{ new int };// Memory location contains junk value
int *p_number6{ new int(22) };// use direct initialization
int *p_number7{ new int{ 23 } };// use uniform initialization

fmt::println("");
fmt::println("Initialize with valid memory address at declaration : ");
fmt::println("p_number5: {}", fmt::ptr(p_number5));
fmt::println("*p_number5: {}", *p_number5);// Junk value

fmt::println("p_number6: {}", fmt::ptr(p_number6));
fmt::println("*p_number6: {}", *p_number6);

fmt::println("p_number7: {}", fmt::ptr(p_number7));
fmt::println("*p_number7: {}", *p_number7);

// Remember to release the memory
delete p_number5; p_number5 = nullptr;
delete p_number6;p_number6 = nullptr;
delete p_number7;p_number7 = nullptr;

// Can reuse pointers
p_number5 = new int(81);
fmt::println("*p_number: {}", *p_number5);


delete p_number5;p_number5 = nullptr;
```

Calling delete more than once: BAD!

```
// Calling delete twice on a pointe : BAD!
p_number5 = new int(99);
fmt::println("*p_number5: {}", *p_number5);

delete p_number5;
delete p_number5;
```

Dangling pointers

```
/*

            Topics:
                  . Dangling pointers
                        . Uninitialized pointer
                        . deleted pointer
                        . Multiple pointers pointing to one address:
                              . one releases memory
                              . the other is left hanging
                        . Solutions
                              . Initialize your pointers at declaration
                              . Reset the pointer to nullptr after delete
                              . In case of multiple pointers pointing to an address:
                                    . Chose one and make it the owner.
                                    . The other will never release memory

*/
```

Dangling pointers: Uninitialized pointer

```rust
// Dangling pointer case 1: Uninitialized pointer
int * p_number; // Dangling uninitialized pointer

fmt::println("");
fmt::println( "Case 1 : Uninitialized pointer : ." );
fmt::println( "p_number : {}", p_number );
fmt::println( "*p_number : {}" , *p_number ); //CRASH!
```

Dangling pointers: Deleted pointer

```cpp
// Case 2 : deleted pointer
fmt::println("");
fmt::println( "Case 2: Deleted pointer" );
int * p_number1 {new int{67}};

fmt::println( "*p_number1 (before delete): {}", *p_number1 );

delete p_number1;

fmt::println( "*p_number1(after delete): {}", *p_number1 );
```

Dangling pointers: Multiple pointers

```cpp
// Case 3 : Multiple pointers pointing to same address
fmt::println("");
fmt::println( "Case 3 : Multiple pointers pointing to same address : " );

int *p_number3 {new int{83}};
int *p_number4 {p_number3};

fmt::println( "p_number3 - {}:  {}", fmt::ptr(p_number3), *p_number3 );
fmt::println( "p_number4 - {}:  {}", fmt::ptr(p_number4) , *p_number4 );

//Deleting p_number3
delete p_number3;

//p_number4 points to deleted memory. Dereferncing it will lead to
//undefined behaviour : Crash/ garbage or whatever
fmt::println( "p_number4(after deleting p_number3) -{} :  {}", fmt::ptr(p_number4) , *p_number4 );
```

Dangling pointers: Solution #1

```cpp
// Solution1 : Initialize your pointers immediately upon declaration
fmt::println( "Solution 1: " );
int *p_number5{nullptr};
int *p_number6{new int(87)};

//Check for nullptr before use

if(p_number6!= nullptr){
    fmt::println( "p_number6 - {} - {}", fmt::ptr(p_number6), *p_number6 );
}else{
    fmt::println( "Invalid address" );
}

if(p_number5){
    fmt::println( "p_number5 - {} - {}", fmt::ptr(p_number5), *p_number5 );
}else{
    fmt::println( "Invalid address" );
}
```

Dangling pointers: Solution #2

```cpp
// Solution2 :
// Right after you call delete on a pointer, remember to reset
// the pointer to nullptr to make it CLEAR it doesn't point anywere
fmt::println("Solution 2: ");
int *p_number7{ new int{ 82 } };

// Use the pointer however you want
fmt::println("p_number7 - {} : {}", fmt::ptr(p_number7), *p_number7);

delete p_number7;
p_number7 = nullptr;// Reset the pointer

// Check for nullptr before use
if (p_number7 =/= nullptr) {
    fmt::println("*p_number7 : {}", *p_number7);
} else {
    fmt::println("Invalid memory access!");
}
```

Dangling pointers: Solution #3

```cpp
// Solution 3
// For multiple pointers pointing to the same address ,  Make sure there is
// one clear pointer (master pointer) that owns the memory ( responsible for releasing when
// necessary) , other pointers should only be able to dereference when the master pointer is valid

fmt::println("");
fmt::println("Solution 3 : ");
int *p_number8{ new int{ 382 } };// Let's say p_number8 is the master pointer
int *p_number9{ p_number8 };

// Dereference the pointers and use them
fmt::println("p_number8 - {} : {}", fmt::ptr(p_number8), *p_number8);

if (!(p_number8 == nullptr)) {// Only use slave pointers when master pointer is valid
    fmt::println("p_number9 - {} : {}", fmt::ptr(p_number9), *p_number9);
}

delete p_number8;// Master releases the memory
p_number8 = nullptr;

if (!(p_number8 == nullptr)) {// Only use slave pointers when master pointer is valid
    fmt::println("p_number9 - {} : {}", fmt::ptr(p_number9), *p_number9);
} else {
    fmt::println("WARNING : Trying to use an invalid pointer");
}
```

When new fails

```
int * data = new int[10000000000000000000000000000]; //CRASH if memory is not available
for(size_t i{0} ; i < 10000000 ; ++i){
    fmt::println( "Iteration: {}", i );
    int * data = new int[100000000000000];
}
```

When new fails: Exceptions

```
for(size_t i{0} ; i < 100 ; ++i){
    try{
        int * data = new int[100000000000000];
    }catch(std::exception& ex){
        fmt::println( "Something went wrong: {}", ex.what() );
    }
}
```

When new fails: std::nothrow

```cpp
for (size_t i{ 0 }; i < 100; ++i) {

    int *data = new (std::nothrow) int[1000000];

    if (data =/= nullptr) {
    fmt::println("Data allocated");
    } else {
    fmt::println("Data allocation failed");
    }
}
```

Null pointer safety

```
// Verbose nullptr check

// It is OK call delete on a nullptr
// Calling delete on a nullptr is OK
int *p_number1{};

delete p_number1;// This won't cause any problem
                 // if p_number1 contains nullptr

if(p_number1 =/= nullptr){   // No need for this check.
    delete p_number1;
 }
```

Memory leaks

```cpp
//Example #1
//Non deleted memory
int *p_number {new int{67}}; // Points to some address, let's call that 0x1111

//Should delete and reset here

int number{55}; // stack variable at address 0x2222

p_number = &number; // Now p_number points to 0x2222
                    // Memory at 0x1111 leaked. It was allocated to our program but we just lost access to it.
```

Memory leaks

```
//Example #2
// Double allocation
int *p_number1 {new int{55}};

//Use the pointer

//Should delete and reset here.

p_number1 = new int{44}; // memory with int{55} leaked.

delete p_number1;
p_number1 = nullptr;
```

Memory leaks

```
//Example #2
// Nested scopes with dynamically allocated memory
{
    int *p_number2{ new int{ 57 } };

    // Use the dynamic memory
}
// Memory with int{57} leaked.
```

Dynamically allocated arrays

```cpp
constexpr size_t size{ 10 };

        // Different ways you can declare an array
        // dynamically and how they are initialized

        double *p_salaries{ new double[size] };// salaries array will
                                               // contain garbage  values
        int *p_students{ new (std::nothrow) int[size]{} };// All values initialized to 0

        double *p_scores{ new (std::nothrow) double[size]{ 1, 2, 3, 4, 5 } };


        // nullptr check and use the allocated array
        if (p_scores) {
            fmt::println("size of scores (it's a regular pointer) : {}", sizeof(p_scores));
            fmt::println("Successfully allocated memory for scores.");

            // Print out elements. Can use regular array access notation, or pointer arithmetic
            for (size_t i{}; i < size; ++i) { fmt::println("value : {}: {}", p_scores[i], *(p_scores + i)); }
        }

        delete[] p_salaries;
        p_salaries = nullptr;

        delete[] p_students;
        p_students = nullptr;

        delete[] p_scores;
        p_scores = nullptr;
```

Dynamically VS Static arrays

```
int scores[10]{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };// Lives on the stack

fmt::println("scores size: {}", std::size(scores));
for (auto s : scores) {
    fmt::println("value: {}", s);
}

int *p_scores1 = new int[10]{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };// Lives on the heap.
//fmt::println( "p_scores1 size: {}" , std::size(p_scores1) );
/*
for( auto s : p_scores1){
    fmt::print( "value : " , s );
}
*/
```