

Memory pointers + types of pointers + the call stack

0x1000	0x4
0x1004	0x1000
0x1008	
0x100C	
0x1010	
0x1014	
0x1018	

`int x = 4;` → integer `x` is set to value 4

`int *pX = &x;` → "integer pointer `pX` is set to the address of `x`"
`x` by reference instead of value

`int y = *pX;`

integer named `y` is set to the thing pointed to by `pX`

Address Value

Static vs dynamic memory allocation pointer to memory out of scope
 ↳ fixed size at compile time vs changing

3 different problems exist with pointers

1. wild pointer: pointers not initialized properly
2. dangling pointer: pointers which store addresses of memory no longer allocated
3. memory leak: losing address of memory allocated to your program

Wild pointers

related to uninitialized local variables no way to know what the value is

array-t `*p-vec;` default value on windows is to nullptr however on linux it random address

↳ results in a seg fault

Consider this code:

```

#include <iostream>
int main() {
    void f();
    int main() {
        In f(): 0x7f51abb16b9e
        Segmentation fault (core dumped)
    }
}
    
```

`std::cout << "In main()..." << std::endl;` ↳ uses memory not allocated by program resulting in a seg fault
 ↳ this line creates a wild pointer
`int *p_value;`
`std::cout << "In f():" << p_value << std::endl;`
`*p_value = 42;` ↳ right before init it prints 42 then reassigns the address pointer to hold int 42
 *go to address stored in `p-value` and write 42

Avoiding wild pointers

↳ Always initialize with nullptr

When memory is being deallocated set pointer to nullptr even if its going out of scope

scope

```

void init() {
    int *p_value(new int(42));
    std::cout << "In init():" << p_value << std::endl;
    // Use 'p_value'...
    delete p_value; // sets heap int to nullptr
    // Why bother setting 'p_value' to nullptr?
}

void no_init() {
    int *p_local;
    std::cout << "In no_init():" << p_local << std::endl;
    *p_local = 91;
}
    
```

→ Asks OS for a fresh int on heap
 ↳ store 42 at address where 42 is
 ↳ then store the address where 42 is
 ↳ heap int 42 is @ 0x1ca3010
 ↳ still holds address
 ↳ In `no_init()`: 0x1ca3010
 ↳ Your results may differ...
 ↳ heap memory Always alive until you delete
 ↳ writes to heap memory
 ↳ overwrites smth possibly

`int *p_value` reserves space on the stack for the pointer

↳ bits which were left earlier become that value

↳ program tries to write at location with garbage address
 OS sees memory isn't allocated for program seg fault

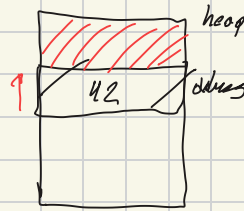
Lucky outcome is Segfault!!

↳ if the garbage address was in scope of memory assigned to program can corrupt said program with successful write

Dangling Pointers

pointers have values `nullptr` is a known invalid address

dangling pointer: a pointer that stores an address that is no longer allocated



All the `delete` keyword does is pass the value of operand to OS to be freed and allocated to something else. The pointer itself is still pointing to that address. If memory is deallocated, the system is welcome to reuse for future requests.

- The single most greatest difficulty for students studying algorithms and data structures is this belief that `delete` does something magical
 - The statement `delete p_data;` does **nothing** more than send the address to the operating system.
- The operating system simply flags the memory at that location as being available for future allocations.

`delete` doesn't do anything but tell OS it's free for future allocation.

```
#include <iostream>
int main() {
    long *p_value_1( new long(42) );
    delete p_value_1;

    double *p_value_2( new double(91.8) );
    std::cout << "Before: " << *p_value_2 << std::endl;

    *p_value_2 = 158;
    std::cout << "After: " << *p_value_2 << std::endl;
    std::cout << "Before: " << *p_value_2 << std::endl;

    return 0;
}
```

`delete` only frees memory for program. It does not touch the pointer.

Memory Leaks

- dynamic memory allocation. OS returns address of memory.
- Think of a friend and they give you a phone # as long as you have it you can contact them. Losing it you can't.
- ynamically allocated memory same issue.

If you dynamically allocate loads of memory losing the address means you can no longer deallocate it. For short programs not too bad All dynamic memory reclaimed at exit. However in an embedded system the OS may run out of available memory. It's lucky reboot otherwise unstable.

Memory leaks occur when pointers go out of scope

```
int f( parameters... ) {
    double *a_data( new double[10]{} );
    // Do something with the allocated memory

    return return-value;
}
```

```
int f( parameters... ) {
    double *a_data(new double[10]{});
    while ( condition ) {
        // Do something with the allocated memory

        if ( another-condition ) {
            return return-value; // memory not deallocated
        }
    }

    delete[] a_data;
    return return-value;
}
```

if memory req. in function call `delete`

Memory leaks also occur if pointers are overwritten without memory first being deallocated

→ should deallocate before assigning again

```
int f( parameters... ) {
    int *a_data[new int{10}];

    // Do something with the allocated memory
    if ( some-condition ) {
        // We need a larger array
        a_data = new int{20};
    }

    delete[] a_data;
    return return-value;
}
```

```
int f( parameters... ) { ~ Nothing happens...
    int *a_data_array[new int{10}];

    // Do something with the allocated memory
    if ( some-condition ) {
        // Deallocate first, then assign new memory
        delete[] a_data_array;
        a_data_array = new int{20};
    }

    delete[] a_data_array;
    return return-value;
}
```

Tips to avoid memory leaks

→ only allocate memory during initialization

→ Check pointers before they leave scope

↳ Should be a return value or deallocated set to null

→ pointers only to be defined

smallest scope

Premature allocation.

```
• Suppose an algorithm needs an array
double calculate_distance( int n, int m, std::size_t cap ) {
    double distance{0.0};
    double *a_distance(new double{cap});

    if ( n == m ) {
        distance = 0.0;
    } else {
        // calculate 'distance' using 'a_distance' as a
        // temporary array
        delete[] a_distance;
        a_distance = nullptr;
    }

    // Use 'distance' in some future calculation...
    return distance;
}
```

↳ only when needed it should reach for it

or else it's gonna just grab memory of every call from block

```
Instead, declare and initialize only when the array is needed
double calculate_distance( int n, int m, std::size_t cap ) {
    double distance{0.0};

    if ( n == m ) {
        distance = 0.0;
    } else {
        int *a_distance(new int{cap});
        // calculate 'distance' using 'a_distance_array' as a
        // temporary array
        delete[] a_distance;
        a_distance = nullptr;
    }

    // Use 'distance' in some future calculation...
    return distance;
}
```

- Try to restrict allocation to initialization
 - If possible, only allocate when a variable is being initialized

```
double *function_name( parameters... ) {
    // do not declare a_int here

    while ( condition ) {
        int *p_int(new int{some_value});
        // Do something with 'p_int'
        delete p_int;
        p_int = nullptr;

        // Do something else...
        assert( p_int == nullptr );
    }
}
```

- Avoiding memory leaks requires rigorous control on the assignment:
 - After deleting a pointer, set the value to nullptr
 - At the end of a function, either all pointers should be nullptr or returned

```
double *function_name( parameters... ) {
    int *p_int(new int{42});
    int *a_int[ p_int ];
    int *a_return_value(new double{10});

    // Do something...
    // - at some point you must delete 'p_int' and
    //   assign it the value 'nullptr'
    assert( p_int == nullptr );
    assert( p_int == nullptr );
    return a_return_value;
}
```

- Use the smallest scope necessary
 - If you only need a pointer in a block, don't declare it outside that block

```
double *function_name( parameters... ) {
    // Do not declare 'a_vec' here...

    while ( condition ) {
        // Do something...

        if ( special-condition ) {
            double *a_vec(new double{1});

            // Use 'a_vec' and at some point, delete[] it
            // and set 'a_vec' to nullptr...
            assert( a_vec == nullptr );
        }
    }
}
```