Part 3 - Memory

Tyler Swann

Agenda

- Pointers
- Slices
- References
- Dynamic Memory
- The Standard Library

Pointers

Part 3

Pointers

- Pointers are an object that holds the numeric memory address of another object
- To access the value pointed to, the pointer must be dereferenced
- Because a pointer is a numeric value, you can perform any arithmetic operation on the pointer itself
- A pointer can be made polymorphic by being a `void` pointer
- You can a pointer to a constant object or a constant pointer to an object (or both!).
- You can also have pointer-to-pointers
- Pointers are easily misused as they can point to nothing

Pointers in Memory

Address	Value
0x00007fff59ae6ea4	…
0x00007fff59ae6e9d	0x00000004
0x00007fff59ae6e99	0x00009 1f5
0x00007fff59ae6e94	0x00007fff59ae6e9d
0x00007fff59ae6e90	…

Pointers Example 1

```cpp
#include <iostream>
#include <memory>

auto main () -> int
{
    int a {4};
    int b {37365};
    int* pa {&a};
    int* pb {std::addressof(b)};

    std::cout << "a = " << a << std::endl;
    std::cout << "pa = " << pa << std::endl;
    std::cout << "*pa = " << *pa << std::endl;

    std::cout << "b = " << b << std::endl;
    std::cout << "pb = " << pb << std::endl;
    std::cout << "*pb = " << *pb << std::endl;

    return 0;
}
```

Pointers Example 2

```cpp
#include <iostream>
#include <memory>

auto main () -> int
{
    int a {4};
    int b {37365};
    const int* pa {&a};
    int* const pb {std::addressof(b)};

    std::cout << "*pa = " << *pa << std::endl;
    *pa += 3;  ///< Fails, comment out to run
    pa = std::addressof(b);
    std::cout << "*pa = " << *pa << std::endl;

    std::cout << "*pb = " << *pb << std::endl;
    *pb += 3;
    pb = std::addressof(a);  ///< Fails, comment out to run
    std::cout << "*pb = " << *pb << std::endl;

    return 0;
}
```

Pointers Example 3

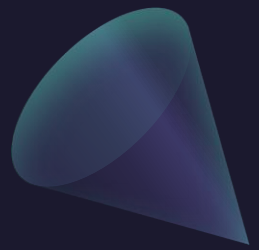
```cpp
#include <iostream>
#include <memory>

auto main () -> int
{
    int a {4};
    void* pa {std::addressof(a)};

    std::cout << "*pa = " << *static_cast<int*>(pa) << std::endl;
    std::cout << "*pa = " << *pa << std::endl;  ///< This will fail, comment out to run

    return 0;
}
```

Pointers Example 4

```cpp
#include <iostream>

auto main () -> int
{
    auto greeting {"Hello!"};
    const char* response {"Hi!!!"};

    for (auto i {0}; i < 7; ++i)
        std::cout << greeting[i];
    std::cout << std::endl;

    for (auto i {0}; i < 6; ++i)
        std::cout << *(response + i);
    std::cout << std::endl;

    /// These will have the same type
    std::cout << "typeid(greeting).name() = " << typeid(greeting).name() << std::endl;
    std::cout << "typeid(response).name() = " << typeid(response).name() << std::endl;

    for (auto i {0}; i < 6; ++i)
        std::cout << *(response++) << std::endl;

    std::cout << "response = " << response << std::endl;  ///< This now points to whatever is stored after `response`.

    return 0;
}
```

Pointers Example 5

```cpp
#include <iostream>

auto main () -> int
{
    int* p {nullptr};

    std::cout << "p = " << p << std::endl;      ///< p = 0

    /// Compiles (on Godbolt) but throws a runtime error (see return of program is not zero)
    std::cout << "*p = " << *p << std::endl;

    return 0;
}
```



Pointers Example 6

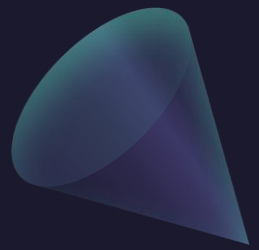
```cpp
#include <iostream>
#include <memory>

auto main () -> int
{
    int a {6};
    int* p {std::addressof(a)};
    int** pp {std::addressof(p)};

    std::cout << "pp = " << pp << std::endl;
    std::cout << "*pp = " << *pp << std::endl;
    std::cout << "**pp = " << **pp << std::endl;

    return 0;
}
```

Slices

Part 3

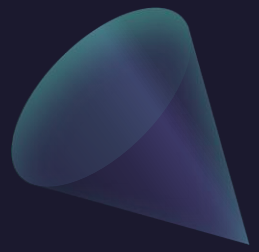
Slices

- Contiguous homogenous sequence of objects
- Size of slice must be known at compile time
- Indexed based access using [ ]
- Indexing starts at `0`
- Slices can decay into a pointer of the same type as the slice pointing to the first element of the slice
- String literals are slices of type *char*



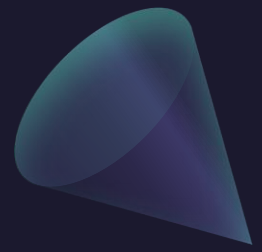
Slices Example 1

```cpp
#include <iostream>

void print(int arr[], std::size_t s)
{
    std::cout << "[ ";
    for (auto i {0}; i < s; ++i)
        std::cout << arr[i] << ", ";
    std::cout << "]" << std::endl;
}

auto main () -> int
{
    int nums[] { 1, 2, 3, 4, 5 };

    print(nums, 5);

    return 0;
}
```



Slices Example 2

```cpp
#include <iostream>

void print(int* arr, std::size_t s)
{
    std::cout << "[ ";
    for (auto i {0}; i < s; ++i)
        std::cout << arr[i] << ", ";
    std::cout << "]" << std::endl;
}

auto main () -> int
{
    int nums[] { 1, 2, 3, 4, 5 };

    print(nums, 5);

    return 0;
}
```



Dynamic Memory

Part 3

Dynamic Memory

- There are two memory sources in C++, the stack and the heap
- Stack resources are ones used by variables and objects in a C++ program
- Heap resources are allocated memory from the free store of the computer
- Heap memory must be explicitly requested and returned to and from the OS
- `new` and `delete` are used to allocate and free memory respectively from the heap in C++
- Memory for slices can also be allocated and freed respectively from the heap using `new[]` and `delete[]`

Dynamic Memory Example 1

```cpp
#include <iostream>

auto main () -> int
{
    int* ip = new int(7);   ///< Creates an `int` initialised with the value `7` on the heap

    std::cout << "ip = " << ip << std::endl;
    std::cout << "*ip = " << *ip << std::endl;

    delete ip;
    ip = nullptr;

    return 0;
}
```

Dynamic Memory Example 2

```cpp
#include <iostream>

void print(int arr[], std::size_t s)
{
    std::cout << "[ ";
    for (auto i {0}; i < s; ++i)
        std::cout << arr[i] << ", ";
    std::cout << "]" << std::endl;
}

auto main () -> int
{
    int* nums = new int[]{ 1, 2, 3, 4, 5 }; ///< Creates a slice of `int` initialised with brace list

    print(nums, 5);

    delete[] nums;
    nums = nullptr;

    return 0;
}
```

References

Part 3

References

- Act as an alias to an existing object
- Any operations on a reference act upon the referred object without the need for dereferencing
- References cannot refer to nothing, ie. Must be bound to an existing object
- References are always constant meaning they cannot be rebound to alias a different object
- A constant reference means the object it aliases is constant

Pointers vs References

Pitfall	Pointers	References	Meaning
Nullable	☑	✗	Pointers can point to nothing, references cannot
Dereferencable	☑	✗	You cannot dereference a reference
Rebindable	☑	✗	A reference cannot be rebound to a new value. Operations done on the reference affect the underlying value, even assignment.
Multiple levels of indirection	☑	✗	You cannot have a reference of a reference.
Pointer arithmetic	☑	✗	You cannot increment (etc.) a reference like a pointer

References Example 1

```cpp
#include <iostream>

auto main () -> int
{
    int i {7};
    int& ir {i};

    std::cout << "i = " << i << std::endl;
    std::cout << "ir = " << ir << std::endl;

    ir += 6;
    std::cout << "i = " << i << std::endl;

    i -= 4;
    std::cout << "ir = " << ir << std::endl;

    return 0;
}
```

References Example 2

```cpp
#include <iostream>

auto main () -> int
{
    int i {7};
    int& ir {i};
    const int& cir {i};

    std::cout << "i = " << i << std::endl;
    std::cout << "ir = " << ir << std::endl;
    std::cout << "cir = " << cir << std::endl;

    ir += 6;
    std::cout << "i = " << i << std::endl;

    i -= 4;
    std::cout << "ir = " << ir << std::endl;

    cir += 7;   ///< Fails, `cir` is read-only
    std::cout << "i = " << i << std::endl;

    return 0;
}
```

The Standard Library

Part 3

The C++ Standard Library

- The C++ Standard Library is home to a very large collection of features available to C++ programmers from containers and algorithms to concurrency and random number generation.
- Any non-language specific feature in C++ can be found in the Standard Library
- Library components are stored in headers and are imported into source files using `#include` directives

Standard Library Types

SEQUENCES

- `std::initializer_list` - A concrete type for the constructor sequences using
- `std::array` - C++'s array type
- `std::span` - A view over any contiguous sequence of homogenous elements
- The type of any sequence must be known or deducible at compile time

STRINGS

- `std::string` - C++'s string type
- `std::string_view` - A view over a character slice or string
- `""s` - Creates a string from a character or string literal
- There are other string types that hold different character types e.g. `std::wstring`

SMART POINTERS

- `std::unique_ptr` - Assumes unique ownership of a dynamic memory resource
- `std::shared_ptr` - Assumes shared ownership of a dynamic memory resource. Only when the last owner is deleted will the resource
- `std::weak_ptr` - Assumes temporary shared ownership of a dynamic memory resource
- All smart pointers automatically delete the dynamic memory resource when the smart pointer goes out of scope

Sequences Example 1

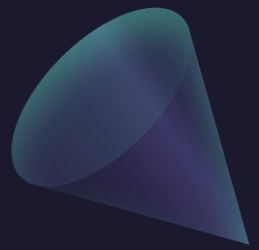
```cpp
#include <iostream>
#include <array>

void print(std::array<int, 6> arr)
{
    std::cout << "[ ";
    for (auto i {0}; i < arr.size(); ++i)
        std::cout << arr[i] << ", ";
    std::cout << "]" << std::endl;
}

auto main () -> int
{
    auto a = std::array<int, 6>{ 1, 2, 3, 4, 5, 6 };
    auto b = std::to_array<int>({ -1, -2, -3, -4, -5, -6});  ///< Size can be deduced

    print(a);
    print(b);

    return 0;
}
```

Sequences Example 2

```cpp
#include <array>
#include <iostream>
#include <span>

void print(std::span<int> span)
{
    std::cout << "[ ";
    for (auto& e : span)
        std::cout << e << ", ";
    std::cout << "]" << std::endl;
}

auto main () -> int
{
    auto array = std::to_array<int>({ 1, 2, 3, 4, 5, 6 });
    int slice[] = {4, 46, 57};

    print(array);
    print(slice);

    return 0;
}
```

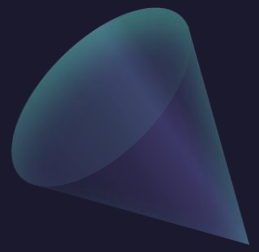
Strings Example 1

```cpp
#include <iostream>
#include <string>

auto main () -> int
{
    auto str1 {"Hello"};
    auto str2("Goodbye");

    std::cout << str1 << std::endl;
    std::cout << str2 << std::endl;

    return 0;
}
```



Strings Example 2

```cpp
#include <iostream>
#include <string_view>

void print(std::string_view s)
{ std::cout << s << std::endl; }

auto main () -> int
{
    print("Hello");

    return 0;
}
```



Strings Example 3

```cpp
#include <iostream>
#include <string>
#include <string_view>

using namespace std::literals;

void print(std::string_view s)
{ std::cout << s << std::endl; }

auto main () -> int
{
    print("Hello"sv);

    std::cout << typeid("Hello").name() << std::endl;
    std::cout << typeid("Hello"s).name() << std::endl;
    std::cout << typeid("Hello"sv).name() << std::endl;

    return 0;
}
```

Smart Pointers Example 1

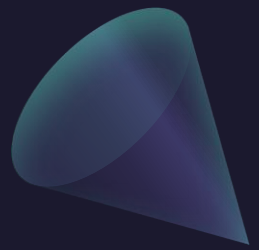
```cpp
#include <iostream>
#include <memory>

void print(std::unique_ptr<int>& ptr)
{
    std::cout << ptr << std::endl;
    std::cout << *ptr << std::endl;
}

void add_magic(std::unique_ptr<int>& ptr)
{ *ptr += 42; }

auto main () -> int
{
    std::unique_ptr<int> p1(new int(6));
    auto p2 = std::make_unique<int>(7);
    auto p3 = std::unique_ptr<int>{nullptr};

    print(p1);
    print(p2);

    add_magic(p1);
    // add_magic(p3);  ///< Would fail

    print(p1);
    // print(p3);  ////< Would fails

    return 0;
}
```

Smart Pointers Example 2

```cpp
#include <iostream>
#include <memory>

void print(std::shared_ptr<int> ptr)
{
    std::cout << "ptr = " << ptr << std::endl;
    std::cout << "*ptr = " << *ptr << std::endl;
    std::cout << "ptr.use_count() = " << ptr.use_count() << std::endl;
}

void add_magic(std::shared_ptr<int>& ptr)
{ *ptr += 42; }

auto main () -> int
{
    auto p = std::make_shared<int>(7);

    std::cout << "p.use_count() = " << p.use_count() << std::endl;

    print(p);
    add_magic(p);

    return 0;
}
```

Smart Pointers Example 3

```cpp
#include <iostream>
#include <memory>

void print(std::weak_ptr<int> ptr)
{
    std::cout << "ptr.use_count() = " << ptr.use_count() << std::endl;

    if (auto sp = ptr.lock())
    {
        std::cout << "sp.use_count() = " << sp.use_count() << std::endl;
        std::cout << "sp = " << sp << std::endl;
        std::cout << "*sp = " << *sp << std::endl;
    }
    else
        std::cout << "ptr is expired" << std::endl;
}

auto main () -> int
{
    auto p = std::make_shared<int>(7);

    std::cout << "p.use_count() = " << p.use_count() << std::endl;

    print(p);

    return 0;
}
```



Discussion

- Any questions?
- Need help?
- Open discussion.
- Concerns?

Next Week

- Advanced Functions
- Namespaces
- Enumerations
- Unions
- Structures

Summary

Throughout this part we learnt about pointers and references and how they can be used to manipulate data owned by someone else. We also learnt about slices, arrays, spans and strings and how they are used to store collections of information. Finally, we learnt about dynamic memory management from manual control to the use of smart pointers.

Thank You

Tyler Swann

https://github.com/MonashDeepNeuron/HPP