Functions Pointers

terça-feira, 6 de agosto de 2024 21:30

Header File (`math_operations.h`)

```cpp
#pragma once

extern int (*add_ptr)(const int, const int);

int add(const int a, const int b);
int addTwoPlusTwo();
```

1. `#pragma once`: This is a header guard to ensure the file is included only once in a single compilation.
2. Function Pointer Declaration:

```cpp
extern int (*add_ptr)(const int, const int);
```

This declares a global function pointer `add_ptr` that can point to any function taking two `int` arguments and returning an `int`. The `extern` keyword makes it accessible across multiple translation units (i.e., source files).

3. Function Prototypes:

```cpp
int add(const int a, const int b);
int addTwoPlusTwo();
```

These are declarations for the `add` and `addTwoPlusTwo` functions.

Source File (`math_operations.cpp`)

```cpp
#include "math_operations.h"

// Initialize the function pointer to point to the actual add function
int (*add_ptr)(const int, const int) = add;

int add(const int a, const int b) {
    return a + b;
}

int addTwoPlusTwo() {
    return add_ptr(2, 2);
}
```

1. Include Header:

```cpp
#include "math_operations.h"
```

This includes the header file to use the declared functions and variables.

```cpp
#include "math_operations.h"
```

This includes the header file to use the declared functions and variables.

2. **Function Pointer Initialization:**

```cpp
int (*add_ptr)(const int, const int) = add;
```

Here, `add_ptr` is initialized to point to the `add` function. This means that `add_ptr` can now be used to call `add`.

3. **Function Definitions:**

```cpp
int add(const int a, const int b) {
    return a + b;
}
```

The `add` function simply adds two integers.

```cpp
int addTwoPlusTwo() {
    return add_ptr(2, 2);
}
```

The `addTwoPlusTwo` function calls `add_ptr` with arguments `2` and `2`. By default, `add_ptr` points to `add`, so this will call `add(2, 2)`. However, `add_ptr` can be changed to point to a different function for testing purposes.

Test File (`test_math_operations.cpp`)

```cpp
#include <gtest/gtest.h>
#include <gmock/gmock.h>
#include "math_operations.h"

using ::testing::Return;

// Mock function
int mock_add(const int a, const int b) {
    return 4;  // Always return 4 for testing
}

TEST(MathOperationsTest, AddTwoPlusTwo) {
    // Save the original function pointer
    int (*original_add_ptr)(const int, const int) = add_ptr;

    // Redirect the function pointer to the mock function
    add_ptr = mock_add;

    int result = addTwoPlusTwo();

    EXPECT_EQ(result, 5);

    // Restore the original function pointer
    add_ptr = original_add_ptr;
}
```

```
    add_ptr = original_add_ptr;
}
```

1. Include GTest and GMock:

```cpp
#include <gtest/gtest.h>
#include <gmock/gmock.h>
```

These includes are necessary for writing and running tests using Google Test and Google Mock.

2. Mock Function:

```cpp
int mock_add(const int a, const int b) {
    return 4;  // Always return 4 for testing
}
```

This mock function always returns `4` regardless of the input. This allows us to control the behavior of `add` during testing.

3. Test Case:

```cpp
TEST(MathOperationsTest, AddTwoPlusTwo) {
    // Save the original function pointer
    int (*original_add_ptr)(const int, const int) = add_ptr;

    // Redirect the function pointer to the mock function
    add_ptr = mock_add;

    int result = addTwoPlusTwo();

    EXPECT_EQ(result, 5);

    // Restore the original function pointer
    add_ptr = original_add_ptr;
}
```

- Save Original Pointer:

```cpp
int (*original_add_ptr)(const int, const int) = add_ptr;
```

Save the original function pointer to restore it later.

- Redirect Pointer:

```cpp
add_ptr = mock_add;
```

Redirect `add_ptr` to point to `mock_add`.

- Run Test:

```cpp
int result = addTwoPlusTwo();
```

```cpp
int result = addTwoPlusTwo();
```

Call `addTwoPlusTwo` which now uses `mock_add` via `add_ptr`.

- **Verify Result**:

```cpp
EXPECT_EQ(result, 5);
```

Check that the result is `5`, as `mock_add(2, 2)` returns `4`, and `addTwoPlusTwo` adds `1`.

- **Restore Original Pointer**:

```cpp
add_ptr = original_add_ptr;
```

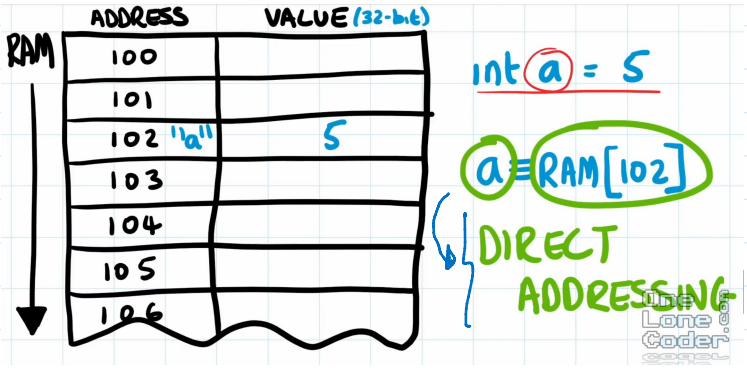
Restore the original function pointer after the test.

Explanation of Concepts

1. **Function Pointers**:
   Function pointers allow us to dynamically change which function is called at runtime. This is useful for testing, as we can redirect function calls to mock functions.
2. **Mocking**:
   Mocking is a technique used in unit testing to simulate the behavior of complex objects or functions. This allows for testing of code in isolation, without dependencies on external systems.
3. **Google Test and Google Mock**:
   - **Google Test**: A framework for writing and running C++ tests.
   - **Google Mock**: A library that provides the ability to create mock objects and functions, which is useful for testing.

By using function pointers, you can effectively mock free functions in C++ and write unit tests that verify the behavior of your code in a controlled environment. This approach provides flexibility and ensures that your tests remain isolated from the actual implementation, making them more reliable and easier to maintain.

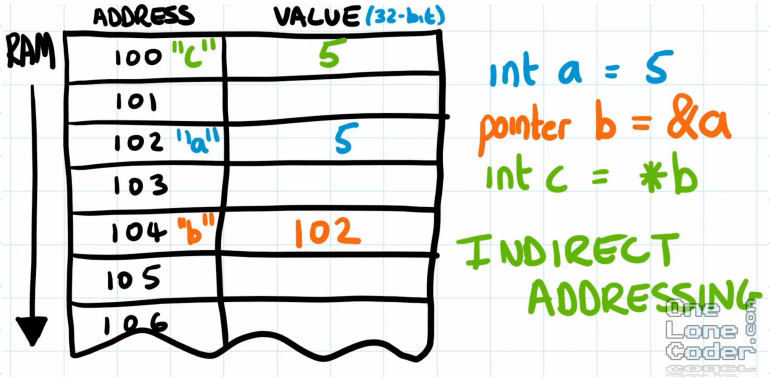
→ Pointer is just a variable that holds a memory address !!! (moves with memory)



→ Here we usually use a simple variable

& → get the adress!

* → get the value of!



(b) → Get the adress of variable a = 102

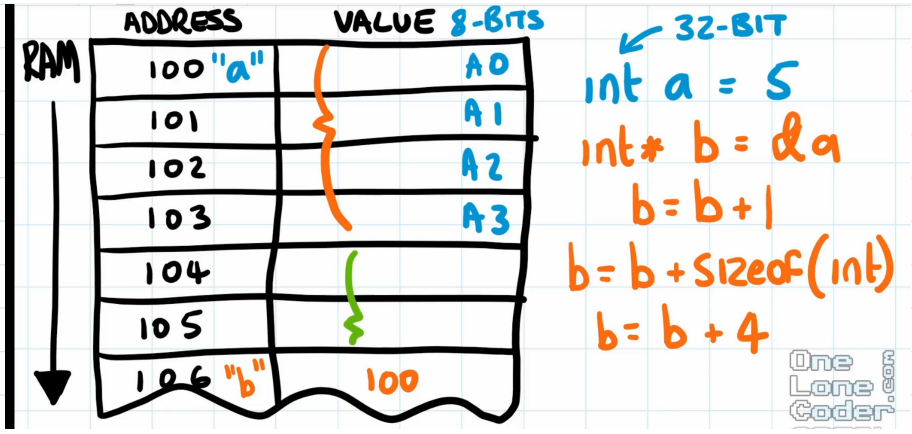
(c) → Get the value at the located pointed by (b) = 5

↳ Indirect annotate the value of (a) to (c)

int* b = &a → Declare me as a pointer

↳ Say that it's a pointer to an adress

↳ Create a variable that holds a memory adress to another variable that is int (Important!)



↳ Ref to the size saved in mem

→ Pointers arg is defined based on the arg of the defined to

↳ int = 4 Byte = 32 bits

```cpp
int SomeArray[10] = { 3, 6, 9, 12, 15, 18, 21, 24, 27, 30 };

int *pLocation0 = &SomeArray[0];

for (int i = 0; i < 10; i++)
{
    cout << SomeArray + i  << " = " << *(SomeArray + i) << en
}
```

```
Select E:\work\code_cpp\PointersVideo\Debug
004FF790 = 3
004FF794 = 6
004FF798 = 9
004FF79C = 12
004FF7A0 = 15
004FF7A4 = 18
004FF7A8 = 21
004FF7AC = 24
004FF7B0 = 27
004FF7B4 = 30
```

4 Bytes

→ Gets the value on that specific location

→ Returns the address of the first element + the number of bytes in the loop (differ by 4 bytes each iteration)

```cpp
// Heap (run time)
sSomeObject *pSomeObject = new sSomeObject[10];

delete[] pSomeObject;
```

→ Returns a pointer to where this object is stored in memory. Defined in the heap in this case!

→ Important to clean it up in the end (to deal with memory leak)

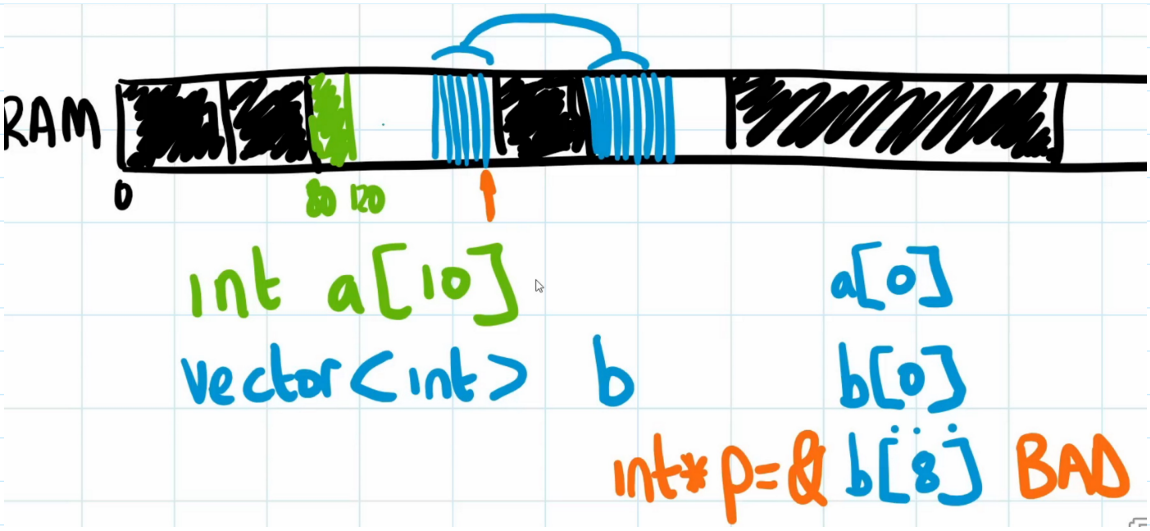
```cpp
// Heap (run time)
sSomeObject **pSomeObject = new sSomeObject*[10];
```

Array of pointers

→ A pointer to the start of an array of pointers

```cpp
for (int i = 0; i < 10; i++)
    delete pSomeObject[i];
```

→ Need to free each element of the vectors



↳ Never create a pointer to an element of a vector!!

→ Smart Pointers

↳ Concept of ownership of the memory allocated

↳ when every thing is done using, it'll delete itself

Shared → Multiple accessors to a pointer

```
shared_ptr<sSomeObject> spSomeObject1 = make_shared<sSomeObject>();
```

↳ sSomeObject * spSomeObject1 = new sSomeObject() } Similar

→ Duration is based on the scope it is using

```
22      shared_ptr<sSomeObject> spSomeObject1 = make_sha
23
24      { // Scope 2
25
26          shared_ptr<sSomeObject> spSomeObject2 = spSo
27
28      } // End Scope 2 ≤ 1ms elapsed
29
30  } // End Scope 1
31
```

Value	Type
shared_ptr {x=0xc3c2c1c0 y=0xd3d2d1d0 } [0x00000002 strong refs] [{_Sto...	std::shared_ptr<`main'...
0x00b2fe14 {x=0xc3c2c1c0 y=0xd3d2d1d0 }	main::__l2::sSomeObje...
{_Storage={_Val=0xc3c2c1c0 _Pad=0x00b2fe14 "ÀÁÂÃĐÑÒÓ... } }	std::_Ref_count_base {...
{_Storage={_Val=0xc3c2c1c0 _Pad=0x00b2fe14 "ÀÁÂÃĐÑÒÓ... } }	std::_Ref_count_obj<`...
0x003debe8 {PointersVideo.exe!void(* std::_Ref_count_obj<`main'::`2'::sSo...	void * *
0x00000002	unsigned long
0x00000001	unsigned long
{...}	std::shared_ptr<`main'...
shared_ptr {x=0xc3c2c1c0 y=0xd3d2d1d0 } [0x00000002 strong refs] [{_Sto...	std::shared_ptr<`main'...

Memory 1
Address: 0x00B2FE14

```
0x00B2FE14  c0 c1 c2 c3 d0 d1 d
0x00B2FE24  3d 4a 00 00 b0 5e b
0x00B2FE34  dd dd dd dd dd dd d
0x00B2FE44  dd dd dd dd 5a b5 5
0x00B2FE54  88 fe b2 00 f0 76 a
0x00B2FE64  08 00 00 00 68 00 0
0x00B2FE74  70 fc b2 00 fd fd f
```

• 2 things pointing to the same loc in memory

↳ Deleted after leaving the scope

Unique → Only one thing can have access to the object

Unique → Only one thing can have access to the object

```
// Smart Pointers - Unique - Only ever ONE accessor to pointer

{ // Scope 1
    unique_ptr<sSomeObject> upSomeObject1 = make_unique<sSomeObject>();

    { // Scope 2

        unique_ptr<sSomeObject> upSomeObject2 = upSomeObject1;

    } // End Scope 2

} // End Scope 1
```

- Not able even to compile it
  - Transfer ownership of the pointer
  - Memory remains the same but the owner changed

```
32
33      // Smart Pointers - Unique - Only ever ONE accessor to pointer
34
35      { // Scope 1
36          unique_ptr<sSomeObject> upSomeObject1 = make_unique<sSomeObject>();
37
38          { // Scope 2
39
40              unique_ptr<sSomeObject> upSomeObject2 = std::move(upSomeObject1);
41
42          } // End Scope 2
43
```

What Are Pointers? (C++)