Modern C++ for Computer Vision and Image Processing

Igor Bogoslavskyi
Outline

Using pointers
Pointers are polymorphic
Pointer “this”
Using const with pointers
Stack and Heap
Memory leaks and dangling pointers
  Memory leak
  Dangling pointer
  RAII
Using pointers in real world
Using pointers for classes

- **Pointers can point to objects of custom classes:**

```cpp
1 std::vector<int> vector_int;
2 std::vector<int>* vec_ptr = &vector_int;
3 MyClass obj;
4 MyClass* obj_ptr = &obj;
```

- **Call object functions from pointer with ->**

```cpp
1 MyClass obj;
2 obj.MyFunc();
3 MyClass* obj_ptr = &obj;
4 obj_ptr->MyFunc();
```

- **obj->Func() ↔ (*obj).Func()**
Pointers are polymorphic

- Pointers are just like references, but have additional useful properties:
  - Can be reassigned
  - Can point to "nothing" (`nullptr`)
  - Can be stored in a vector or an array

**Use pointers for polymorphism**

```
1 Derived derived;
2 Base* ptr = &derived;
```

**Example:** for implementing strategy store a pointer to the strategy interface and initialize it with `nullptr` and check if it is set before calling its methods
```cpp
#include <iostream>
#include <vector>
using std::cout;
struct AbstractShape {
    virtual void Print() const = 0;
};
struct Square : public AbstractShape {
    void Print() const override { cout << "Square\n"; }
};
struct Triangle : public AbstractShape {
    void Print() const override { cout << "Triangle\n"; }
};
int main() {
    std::vector<AbstractShape*> shapes;
    Square square;
    Triangle triangle;
    shapes.push_back(&square);
    shapes.push_back(&triangle);
    for (const auto* shape : shapes) { shape->Print(); }
    return 0;
}
```
Every object of a class or a struct holds a pointer to itself
This pointer is called this
Allows the objects to:
- Return a reference to themselves: `return *this;`
- Create copies of themselves within a function
- Explicitly show that a member belongs to the current object: `this->x();`
Using const with pointers

- Pointers can **point to** a **const** variable:
  
  ```
  // Cannot change value, can reassign pointer.
  const MyType* const_var_ptr = &var;
  const_var_ptr = &var_other;
  ```

- Pointers can be **const**:
  
  ```
  // Cannot reassign pointer, can change value.
  MyType* const var_const_ptr = &var;
  var_const_ptr->a = 10;
  ```

- Pointers can do both at the same time:
  
  ```
  // Cannot change in any way, read-only.
  const MyType* const const_var_const_ptr = &var;
  ```

- Read from right to left to see which const refers to what
Stack and heap
Memory management structures

Working memory is divided into two parts:

Stack and Heap

stack
http://www.freestockphotos.biz

heap
https://pixabay.com
Stack memory

- **Static** memory
- Available for **short term** storage (scope)
- **Small / limited** (8 MB Linux typisch)
- Memory allocation is **fast**
- **LIFO** (**L**ast **i**n **F**irst **o**ut) structure
- Items added to top of the stack with **push**
- Items removed from the top with **pop**
Stack memory

```
#include <stdio.h>
int main(int argc, char const* argv[]) {
    int size = 2;
    int* ptr = nullptr;
    {
        int ar[size];
        ar[0] = 42;
        ar[1] = 13;
        ptr = ar;
    }
    for (int i = 0; i < size; ++i) {
        printf("%d\n", ptr[i]);
    }
    return 0;
}
```
Heap memory

- **Dynamic** memory
- Available for **long** time (program runtime)
- Raw modifications possible with **new** and **delete** (usually encapsulated within a class)
- Allocation is slower than stack allocations
Operators **new** and **new[]**

- User controls memory allocation (unsafe)
- Use **new** to allocate data:

```cpp
1 // pointer variable stored on stack
2 int* int_ptr = nullptr;
3 // 'new' returns a pointer to memory in heap
4 int_ptr = new int;
5
6 // also works for arrays
7 float* float_ptr = nullptr;
8 // 'new' returns a pointer to an array on heap
9 float_ptr = new float[number];
```

- **new** returns an address of the variable on the heap
- **Prefer using smart pointers!**
Operators delete and delete[]

- Memory is not freed automatically!
- User must remember to free the memory
- Use delete or delete[] to free memory:

```cpp
1 int* int_ptr = nullptr;
2 int_ptr = new int;
3 // delete frees memory to which the pointer points
4 delete int_ptr;
5
6 // also works for arrays
7 float* float_ptr = nullptr;
8 float_ptr = new float[number];
9 // make sure to use 'delete[]' for arrays
10 delete[] float_ptr;
```

- Prefer using smart pointers!
Example: heap memory

```cpp
#include <iostream>
using std::cout; using std::endl;

int main() {
    int size = 2; int* ptr = nullptr;
    {
        ptr = new int[size];
        ptr[0] = 42; ptr[1] = 13;
    } // End of scope does not free heap memory!
    // Correct access, variables still in memory.
    for (int i = 0; i < size; ++i) {
        cout << ptr[i] << endl;
    }
    delete[] ptr; // Free memory.
    for (int i = 0; i < size; ++i) {
        // Accessing freed memory. UNDEFINED!
        cout << ptr[i] << endl;
    }
    return 0;
}
```
Possible issues with memory
Memory leak

- Can happen when working with Heap memory if we are not careful
- **Memory leak**: memory allocated on Heap access to which has been lost
#include <iostream>
using std::cout; using std::endl;

int main() {
    double *ptr_1 = NULL;
    double *ptr_2 = NULL;
    int size = 10;
    // Allocate memory for two arrays on the heap.
    ptr_1 = new double[size];
    ptr_2 = new double[size];
    cout << "1: " << ptr_1 << " 2: " << ptr_2 << endl;
    ptr_2 = ptr_1;
    // ptr_2 overwritten, no chance to access the memory.
    cout << "1: " << ptr_1 << " 2: " << ptr_2 << endl;
    delete[] ptr_1;
    delete[] ptr_2;
    return 0;
}
Error: double free or corruption

- The memory under address 0x10a3070 is never freed
- Instead we try to free memory under 0x10a3010 twice
- Freeing memory twice is an error
#include <iostream>
#include <cmath>
#include <algorithm>

using std::cout; using std::endl;

int main() {
    double *data = nullptr;
    size_t size = pow(1024, 3) / 8; // Produce 1GB
    for (int i = 0; i < 5; ++i) {
        // Allocate memory for the data.
        data = new double[size];
        std::fill(data, data + size, 1.23);
        // Do some important work with the data here.
        cout << "Iteration: " << i << " done!" << endl;
    }
    // This will only free the last allocation!
    delete[] data;
    int unused; std::cin >> unused; // Wait for user.
    return 0;
}
Memory leak example

- If we run out of memory an `std::bad_alloc` error is thrown
- Be careful running this example, everything might become slow

```
1 Iteration: 0 done!
2 Iteration: 1 done!
3 Iteration: 2 done!
4 Iteration: 3 done!
5 terminate called after throwing an instance of 'std::bad_alloc'
6 what(): std::bad_alloc
```
Dangling pointer

```c
1 int* ptr_1 = some_heap_address;
2 int* ptr_2 = some_heap_address;
3 delete ptr_1;
4 ptr_1 = nullptr;
5 // Cannot use ptr_2 anymore! Behavior undefined!
```
Dangling pointer

- **Dangling Pointer**: pointer to a freed memory
- Think of it as the opposite of a memory leak
- Dereferencing a dangling pointer causes **undefined behavior**
Dangling pointer example

```cpp
#include <iostream>
using std::cout; using std::endl;

int main() {
    int size = 5;
    int *ptr_1 = new int[size];
    int *ptr_2 = ptr_1;  // Point to same data!
    ptr_1[0] = 100;    // Set some data.
    cout << "1: " << ptr_1 << " 2: " << ptr_2 << endl;
    cout << "ptr_2[0]: " << ptr_2[0] << endl;
    delete[] ptr_1;  // Free memory.
    ptr_1 = nullptr;
    cout << "1: " << ptr_1 << " 2: " << ptr_2 << endl;
    // Data under ptr_2 does not exist anymore!
    cout << "ptr_2[0]: " << ptr_2[0] << endl;
    return 0;
}
```
Even worse when used in functions

```c
#include <stdio.h>
// data processing
int* GenerateData(int size);
void UseDataForGood(const int* const data, int size);
void UseDataForBad(const int* const data, int size);
int main() {
    int size = 10;
    int* data = GenerateData(size);
    UseDataForGood(data, size);
    UseDataForBad(data, size);
    // Is data pointer valid here? Should we free it?
    // Should we use 'delete[]' or 'delete'?
    delete[] data;  // ????????????????
    return 0;
}
```
void UseDataForGood(const int* const data, int size) {
    // Process data, do not free. Leave it to caller.
}

void UseDataForBad(const int* const data, int size) {
    delete[] data;  // Free memory!
    data = nullptr; // Another problem - this does nothing!
}

- **Memory leak** if nobody has freed the memory
- **Dangling Pointer** if somebody has freed the memory in a function
RAII

- **Resource Allocation Is Initialization.**
- New object $\rightarrow$ allocate memory
- Remove object $\rightarrow$ free memory
- Objects **own** their data!

```cpp
class MyClass {
public:
  MyClass() { data_ = new SomeOtherClass; }
  ~MyClass() {
    delete data_;  
    data_ = nullptr;
  }
private:
  SomeOtherClass* data_; 
};
```

- Still cannot copy an object of **MyClass**!!!
struct SomeOtherClass {}

class MyClass {
  public:
    MyClass() { data_ = new SomeOtherClass; }
    ~MyClass() {
      delete data_; 
      data_ = nullptr;
    }
  private:
    SomeOtherClass* data_;
};

int main() {
  MyClass a;
  MyClass b(a);
  return 0;
}

*** Error in `raii_example':
double free or corruption: 0x0000000000877c20 ***
Shallow vs deep copy

- **Shallow copy**: just copy pointers, not data
- **Deep copy**: copy data, create new pointers
- Default copy constructor and assignment operator implement shallow copying
- RAII + shallow copy $\rightarrow$ **dangling pointer**
- RAII + Rule of All Or Nothing $\rightarrow$ **correct**
- **Use smart pointers instead!**