Modern C++ for Computer Vision and Image Processing

Igor Bogoslavskyi
Outline

Static variables and methods
Representation of numbers in memory
Raw C arrays
Non-owning pointers in C++
Classes in memory
Static variables and methods

Static member variables of a class
- Exist exactly **once** per class, **not** per object
- The value is equal across all instances
- Must be defined in *.cpp* files

Static member functions of a class
- Do not have an object of a class
- Can access private members but need an object
- **Syntax** for calling:
  ```cpp
  ClassName::MethodName(<params>)
  ```
#include <iostream>
using std::cout; using std::endl;
struct Counted {
    Counted() { Counted::count++; }
    ~Counted() { Counted::count--; }
    static int count; // Static counter member.
};
int Counted::count = 0; // Definition.
int main() {
    Counted a, b;
    cout << "Count: " << Counted::count << endl;
    Counted c;
    cout << "Count: " << Counted::count << endl;
    return 0;
}
Static member functions

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

class Point {
public:
    Point(int x, int y) : x_(x), y_(y) {}
    static float dist(const Point& a, const Point& b) {
        int diff_x = a.x_ - b.x_;  
        int diff_y = a.y_ - b.y_;  
        return sqrt(diff_x * diff_x + diff_y * diff_y);
    }
private:
    int x_ = 0; int y_ = 0;
};

int main() {
    Point a(2, 2), b(1, 1);
    cout << "Dist is " << Point::dist(a, b) << endl;
    return 0;
}
```
Recalling variable declaration

- `int x = 1;`
- `float y = 1.1313f;`

How is the number represented in the memory?
Number representations

Alphanumerical
- char / uint8_t
  - [1 byte = 8 bits]

Numerical
- Floating point
  - Single precision
    - float
      - [4 bytes = 32 bits]
  - Double precision
    - double
      - [8 bytes = 64 bits]
- Integers
  - int / int32_t
    - [4 bytes = 32 bits]
How much memory does a type need?

Get number of bytes for a type:

\[
\text{sizeof(<type>)}
\]

- 1 Bit = \{0, 1\}
- 1 Byte = 8 Bit
- 1024 Byte = 1 KB
- 1024 KB = 1 MB
- 1024 MB = 1 GB
- 1024 GB = 1 TB
// machine specific type sizes

// floating point types

// integral data types

Example sizeof()
# Representing integer types

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

int main() {
    unsigned short int k = 37;
    cout << "sizeof(" << k << ") is " << sizeof(k)
        << " bytes or " << sizeof(k) * 8 << " bits."
    }
```

sizeof(37) is 2 bytes or 16 bit

**Representation in memory:**

<table>
<thead>
<tr>
<th></th>
<th>1 byte</th>
<th>1 byte</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 0 0 0 0 0 0 0</td>
<td>0 0 1 0 0 1 0 1</td>
</tr>
<tr>
<td>32768</td>
<td>⋮</td>
<td>32 16 8 4 2 1</td>
</tr>
<tr>
<td>2^{15}</td>
<td>⋮</td>
<td>2^5 2^4 2^3 2^2 2^1 2^0</td>
</tr>
</tbody>
</table>

37 = 0 ⋅ 2^{15} + ⋮ + 1 ⋅ 2^5 + 0 ⋅ 2^4 + 0 ⋅ 2^3 + 1 ⋅ 2^2 + 0 ⋅ 2^1 + 1 ⋅ 2^0
Representable intervals

- **2 Byte**
  - short int
  - unsigned short int
    - $[-2^{15}, +2^{15})$
    - $[0, +2^{16})$

- **4 Byte**
  - int
  - unsigned int
    - $[-2^{31}, +2^{31})$
    - $[0, +2^{32})$

- **8 Byte**
  - long int
  - unsigned long int
    - $[-2^{63}, +2^{63})$
    - $[0, +2^{64})$
# Floating point numbers

```cpp
#include <iostream>

using std::cout;

int main(int argc, char *argv[]) {
    float k = 3.14159;
    cout << "sizeof(" << k << ") is " << sizeof(k) << " bytes or " << sizeof(k) * 8 << " bits."
    << " bytes or " << sizeof(k) * 8 << " bits.";
}
```

**Output:**

```text
sizeof(3.141590) is 4 bytes or 32 bit
```

**Representation in memory:**

```
bit#  31  23  22  0
      0100000000 10010010000001111111111011011

     sign  exponent  mantissa

     8 bits  23 bits
```
Floating point numbers

### In memory:
- **Sign** $s = 0$
- **Exponent** $e = 1 \cdot 2^7 + 0 \cdot 2^6 + \ldots + 0 \cdot 2^0 - 127 = 1$
- **Mantissa** $m = 1\frac{1}{2^0} + 1\frac{1}{2^1} + 0\frac{1}{2^2} + \ldots 1\frac{1}{2^{22}} = 1.5707964$
- **Number**: $k = -1^s \cdot 2^e \cdot m$

### Representable interval:
- **binary**: $\pm[1.7 \cdot 2^{-126}, 2.2 \cdot 2^{127}]$
- **decimal**: $\pm[1.2 \cdot 10^{-38}, 3.4 \cdot 10^{38}]$
**float vs. double**

- Same representation as `float`
- `double` takes 8 bytes instead of 4 for `float`
- Longer Exponent und Mantissa.
  - Exponent = **11** Bits instead of 8 for `float`
  - Mantissa = **53** Bits instead of 23 for `float`
### What can we represent?

<table>
<thead>
<tr>
<th>Datatype</th>
<th>Memory</th>
<th>Interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>4 Byte</td>
<td>([0, 4.3 \cdot 10^9))</td>
</tr>
<tr>
<td>float</td>
<td>4 Byte</td>
<td>([1.18 \cdot 10^{-38}, 3.4 \cdot 10^{38}])</td>
</tr>
</tbody>
</table>

- **int**: Every number \(|x| \in [0, 2^{32})\) with an increment of 1 can be represented.
- **float**: Increment depends on the magnitude of the Exponent!
  - **Exponent**: Defines the size of representable interval, \(8\) Bit \(\rightarrow [2^{-126}, 2^{127}] = [1.2 \cdot 10^{-38}, 1.7 \cdot 10^{38}]\)
  - **Mantissa**: Generates a constant with 8 significant digits, 23 Bits long
Limited number of significant digits

Addition of 10 000 000 to $\pi$

mantissa with 8 digits

+ 1 0 0 0 0 0 0 0 0

mantissa with 8 digits

= 1 0 0 0 0 0 0 0 0 3 1 4 1 5 9 2 7

mantissa with 8 digits

7 digits lost
# Digits extinction

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

int main() {
    float pi = M_PI;
    float big_number = 1e7;
    cout << "Pi before: " << pi << endl;
    pi += big_number;
    pi -= big_number;
    cout << "Pi after: " << pi << endl;
    cout << "Difference: " << M_PI - pi << endl;
    return 0;
}
```

**Result:**

- Pi before: 3.14159
- Pi after: 3
- Difference: 0.141593
**C style arrays**

- Base for `std::array`, `std::vector`, `std::string`
- The length of the array is **fixed**
- **Indexing begins with 0!**
- Elements of an array lie in continuous memory.

**Declaration:**

```cpp
Type array_name[length];
Type array_name[length] = {n0, n1, n2, ..., nX};
Type array_name[] = { n1, n2, n3};
```
Arrays are simple data containers

```
int main() {
    int shorts[5] = {5, 4, 3, 2, 1};
    double doubles[10];
    char chars[] = {'h', 'a', 'l', 'l', 'o'};
    shorts[3] = 4;
    chars[1] = 'e';
    chars[4] = chars[2];
    doubles[1] = 3.2;
}
```

- Have no methods
- Do not explicitly store their size
Arrays and sizeof()

`sizeof()` of an array is

\[ \text{sizeof}(<\text{type}>) \times <\text{array\_length}> \]

2. double longA[4] = {1.0, 1.1, 1.2, 1.3};
3. `sizeof(shortA)` = 20
4. `sizeof(shortA) / sizeof(shortA[0])` = 5
5. `sizeof(longA)` = 32
6. `sizeof(longA) / sizeof(longA[0])` = 4
Working memory or RAM

- Working memory has **linear addressing**
- Every byte has an **address** usually presented in hexadecimal form, e.g. \(0x7fff\ ffb7335fdc\)
- Any address can be accessed at random
- **Pointer** is a type to store memory addresses


**Pointer**

- `<TYPE>*` defines a pointer to type `<TYPE>`
- The pointers **have a type**
- Pointer `<TYPE>*` can point **only** to a variable of type `<TYPE>`
- Uninitialized pointers point to a random address
- Always initialize pointers to an address or a `nullptr`

**Example:**

```c
1 int* a = nullptr;
2 double* b = nullptr;
3 YourType* c = nullptr;
```
Non-owning pointers

- Memory pointed to by a raw pointer is not removed when pointer goes out of scope
- Pointers can either own memory or not
- Owning memory means being responsible for its cleanup
- **Raw pointers should never own memory**
- We will talk about **smart pointers** that own memory later
Address operator for pointers

- Operator & returns the address of the variable in memory
- Return value type is “pointer to value type”

Example:

```c
int a = 42;
int* a_ptr = &a;
```
Pointer to pointer

Example:

1 int a = 42;
2 int* a_ptr = &a;
3 int** a_ptr_ptr = &a_ptr;

```
int a; int* a_ptr = &a; int** a_ptr2 = &a_ptr;
```
Pointer dereferencing

- Operator * returns the value of the variable to which the pointer points
- Dereferencing of nullptr: **Segmentation Fault**
- Dereferencing of uninitialized pointer: **Undefined Behavior**
# Pointer dereferencing

```cpp
#include <iostream>

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

int main() {
    int a = 42;
    int* a_ptr = &a;
    int b = *a_ptr;
    cout << "a = " "a = " << a << " b = " b << endl;
    *a_ptr = 13;
    cout << "a = " "a = " << a << " b = " b << endl;
    return 0;
}
```

**Output:**

1. `a = 42, b = 42`
2. `a = 13, b = 42`
#include <iostream>
using std::cout;
using std::endl;

int main() {
    int* i_ptr; // BAD! Never leave uninitialized!
    cout << "ptr address: " << i_ptr << endl;
    cout << "value under ptr: " << *i_ptr << endl;
    i_ptr = nullptr;
    cout << "new ptr address: " << i_ptr << endl;
    cout << "ptr size: " << sizeof(i_ptr) << " bytes";
    cout << " (" << sizeof(i_ptr) * 8 << " bit) " << endl;
    return 0;
}

ptr address: 0x400830
value under ptr: -1991643855
new ptr address: 0
ptr size: 8 bytes (64bit)
Important

- Always initialize with a value or a `nullptr`
- Dereferencing a `nullptr` causes a **Segmentation Fault**
- Use `if` to avoid Segmentation Faults

```cpp
if(some_ptr) {
    // only enters if some_ptr != nullptr
}
if(!some_ptr) {
    // only enters if some_ptr == nullptr
}
```
Arrays in memory and pointers

- Array elements are **continuous in memory**
- Name of an array is an alias to a pointer:

```c
1 double ar[3];
2 double* ar_ptr = ar;
3 double* ar_ptr = &ar[0];
```

- Get array elements with operator `[ ]`
Careful! Overflow!

```cpp
#include <iostream>

int main() {
    int ar[] = {1, 2, 3};
    // WARNING! Iterating too far!
    for (int i = 0; i < 6; i++){
        std::cout << i << " : value: " << ar[i] << " \t addr:" << &ar[i] << std::endl;
    }
    return 0;
}
```

0: value: 1 addr:0x7ffd17deb4e0
1: value: 2 addr:0x7ffd17deb4e4
2: value: 3 addr:0x7ffd17deb4e8
3: value: 0 addr:0x7ffd17deb4ec
4: value: 4196992 addr:0x7ffd17deb4f0
5: value: 32764 addr:0x7ffd17deb4f4
Custom objects in memory

- How the parts of an object are stored in memory is not strongly defined
- Usually sequentially
- The compiler can optimize memory

```cpp
class MemoryTester {
public:
  int i;
  double d;
  void SetData(float data) { data_ = data; }
  float* GetDataAddress() { return &data_; }

private:
  float data_; // position of types is important
};
```
```cpp
#include "class_memory.h"
#include <iostream>
using std::cout; using std::endl;

int main() {
    MemoryTester tester;
tester.i = 1; tester.d = 2; tester.SetData(3);
    cout << "Sizeof tester: " << sizeof(tester) << endl;
    cout << "Address of i: " << &tester.i << endl;
    cout << "Address of d: " << &tester.d << endl;
    cout << "Address of _data: "
          << tester.GetDataAddress() << endl;
    return 0;
}

// memory:     |i|i|i|i|_|_|_|_|d|d|d|d|d|d|d|d|d|...
// who is who:  | int i |padding| double d |...