C++ Programming

Uniform initialization

Mostafa S. Ibrahim
Teaching, Training and Coaching since more than a decade!

Artificial Intelligence & Computer Vision Researcher
PhD from Simon Fraser University - Canada
Bachelor / Msc from Cairo University - Egypt
Ex-(Software Engineer / ICPC World Finalist)

The **Nightmare** of Initialization in C++

- There are many initialization types in C++
  - Before C++11, you have to use different types of initialization for different cases
- One may try to initialize an integer in at **least 19 ways** (4 them CE)
- Understanding when an object is uninitialized is **not trivial**.
- With C++ 11: We are rescued with **uniform initialization**!

Initialization Terminology

- **Direct initialization:**
  - Initial value passed directly
- **Copy initialization:**
  - Initialization with =
  - not possible with `explicit` (C++17 relaxes this)
- **Default initialization:**
  - Object is initialized only if corresponding constructor is defined
- **Zero initialization:**
  - Object initialized by (converted) 0
  - Used for global/static/thread-local objects
- **Value initialization:**
  - Object always gets a value (initialized by constructor or zero initialized)
- **List initialization**
  - Object is initialized by braces (both direct-list-initialization or copy-list-initialization)
  - Object always gets a value (initialized by constructor or zero initialized)
- **Aggregate initialization**
  - Special form of list initialization if type is aggregate

```
int i(42);   // direct initialization
int j{42};   // direct list initialization
int k=42;    // copy initialization
int l={42}; // copy list initialization
```

```
int i1;                    // undefined value
int i2 = 42;               // note: inits with 42
int i3(42);                // inits with 42
int i4 = int();            // inits with 0
int i5{42};                // inits with 42
int i7{};                  // inits with 0
int i6 = {42};             // inits with 42
int i8 = {};               // inits with 0
auto i9 = 42;              // inits int with 42
auto i10{42};              // C++11: std::initializer_list<int> , C++14: int
auto i11 = {42};           // inits std::initializer_list<int> with 42
auto i12 = int{42};        // inits int with 42
int i13();                 // declares a function
int i14(7, 9);             // compile-time error
int i15 = (7, 9);          // OK, inits int with 9 (comma operator)
int i16 = int(7, 9);       // compile-time error
auto i17(7, 9);            // compile-time error
auto i18 = (7, 9);         // OK, inits int with 9 (comma operator)
auto i19 = int(7, 9);      // compile-time error
```

Uniform initialization

- Use {...}
- Uniform initialization: everything can be initialized in much the same way
  - The **same** in all initialization **contexts**
  - E.g. in constructor initialization list, for declaring automatic, temporary, global, heap-allocated variables, for class members.
- It makes our life simpler and more consistent
- Sometimes it is more safer
  - Better initialization for missing cases (e.g. with a class without default constructor)
  - Possible warnings/Compiler errors (e.g. narrowing)
- Sometimes the only way to solve a problem
  - Initializing in a template: which initialization type to use
- Some debates also around it (and potential errors)

Primitives

```
22
23      char arr1[] = "hello";      // Copy initialization
24      char arr2[] = {"hello"};    // Copy initialization
25      char arr3[] {"hello"};      // Direct initialization (MORE efficient)
26      char arr4[] ("hello");
27
28      int x0;     // Default initialization for primitives = garbage
29      int x1 = 5; // Copy initialization
30      int x2(5);  // Direct initialization
31      int x3{5};  // Direct initialization
32
33      int y1 {};  // value initialization for primitives = 0
34      int y2();   // FUNCTION!
35
36      double z = 10;
37      int z1 = z;
38      // warning: narrowing conversion of 'z' from 'double' to 'int' [-Wnarrowing]
39      int z2 {z}; // warning or CE
40
41
```

Objects

```
// Aggregate class:
// public/No constructors/no virtual/inheritance/no in-class initializers
struct Employee {
    int id;         // Default initialization = garbage
    string name;    // Default initialization => call constructor = ""
};

int main() {
    Employee e1;    // Default initialization
    Employee e2();  // FUNCTION
    Employee e3{};  // value initialization

    // temporary object with () or {} => Value initialization
    Employee();
    Employee{};

    Employee e4 {10, "Mostafa"};            // Aggregate initialization
    char arr5[] {'h', 'e', 'l', 'l', 'o'};  // Aggregate initialization
```

Simple informal rules

- Forgot to initialize ⇒ Default initialization
  - Primitive ⇒ Garbage
  - Struct object ⇒ Default constructor
- Initialized with something? {blabla} (blabla)
  - Using **=** ⇒ Copy initialization (ok for primitives, slow otherwise)
  - Without = ⇒ Direct initialization
- Initialized without something? {} ()
  - Primitive ⇒ zero-initialization
  - Struct object ⇒ Default constructor
- Future reading

Tips

- Can you stick to uniform initialization? Great. Otherwise:
  - You may use = for primitives (int x = 10;)
  - Defined user types: stick to uniform initialization
- Did you explicitly initialized a scalar object?
  - If no: for safety assume uninitialized
- In structs: use always a default member initializer
  - helps if forgot to initialize in some constructor
  - Int id = 0;
- Later (Defaulted functions lecture): another example administrate why **{}** is more safer choice than default initialization
- Later in templates: Again **{}** comes to rescue us

"Acquire knowledge and impart it to the people."

"Seek knowledge from the Cradle to the Grave."