Modern C++ - Life

8. Life

Life and scope rules

In imperative programming languages variables have two important properties:

1. **Life** - the TIME under run-time when the *memory area* is valid and usable.
2. **Scope** - the AREA in the program where a *name* is binded to a memory area.

Life and scope is defined by the *declaration* of the variables. More precisely, the *place* of the declaration and the specified *storage class* is important. Declaration also specifies the *type* of the language objects.

Life categories in C++

In C objects are mapped into the memory based on their storage types. Different storage types means different life rules.

String literals

String literals are values known at compile-time. The type of a string literal is *character array of N*. Write attempt of a string literal is undefined. These character arrays are *not* allocated in the writeable area of the program. Those should be considered as read-only memory. (In many implementations they really are allocated in read-only memory.)

```
char *hello1 = "Hello world";
// ...
// BAD!
hello1[1] = 'a'; // likely run-time error!
```

To avoid this situation, declare pointers to string literals as const char *

```
const char *hello1 = "Hello world";
// ...
hello1[1] = 'a'; // syntax error!
```

```
$ g++  -Wall  s.cpp
s.coo: In function ‘main’:
```

```
s.cpp:11:13: error: assignment of read-only location ‘*(hello2 + 1u)’
   hello2[1] = 'a'; // likely run-time error!
             ^
```

Moreover, **hello1** and **hello2** could be stored only one time, therefore the pointer value of hello1 and hello2 could be equal:

```
hello1 == hello2
```

This is different from using arrays to store strings. Those are allocated in the program area, and they can be read and write.

```
char t1[] = {'H','e','l','l','o','\0'};
char t2[] = "Hello";
char t3[] = "Hello";

char t1[1] = 'a'; // ok
```

and the address of t1, t2 and t3 are different.

Automatic life

Objects local to a block (and not declared **static**) has *automatic life*. Such objects are created in the stack. The stack is safe in a multithreaded environment. Objects created when the declaration is encountered and destroyed when control leaves the declaration block.

```
void f()
{
    int i = 2;  // life starts here with initialization
    ....
}               // life finished here
```

There should be no reference to a variable after its life has ended.

```
//
// This is BAD!!!
//
int *f()
{
    int i = 2;  // life starts here with initialization
    ....
    return &i;  // likely a run-time error
}               // life finished here
```

Using the return value is invalid, since the life of **i** is already finished, and the memory area of **i** might be reused for other purposes.

Dynamic Life

Objects with dynamic life is created in the *free store* or *heap*. The lifetime starts with the call of the **new** expression. The life ends with the call of **delete** expression. This may be called in a different place of the program.

There are separate operators for allocating/freeing arrays: **new[]** and **delete []** . If a storage has been allocated by the array version of new then it should be deallocated by the array version of delete.

```
char   *buffer = new char[1024];   // life starts here, alloc 1024 chars
double *dbls   = new double(3.14); // single double intilized to 3.14
//...
delete [] buffer;                  // life finished here
delete dbls;                       // life finished here
```

The **new** expression allocates a memory area, calls the appropriate constructor for the allocated object and returns a pointer to the object.

If memory allocation was unsuccesfull, then **new** throws **bad_alloc** exception. It is also possible that the constructor called on the initialization throws exception. Separate **new** exist for non-throwing, and returning NULL pointer on failure.

```
double *dbls;
try
{
  dbls = new double[3.14];
  // ok, succesfull allocation
}
catch( std::bad_alloc )
{
  // unsuccesfull allocation attempt
}
```

There are no *realloc* expression in C++.

Static life

Global variables, and static global variables have static life. Static life starts at the beginning of the program, and ends at the end of the program.

```
char buffer[80]; // static life is initialized automatically to '\0's
static int j;    // static life is initialized automatically to 0

int main()
{
// ...
}   // life finished here
```

The order of creation is well-defined inside a compilation unit, but not defined order between source-files. This can lead to the static initialization problems.

Local Static Variables

Local statics are declared inside a function as local variables, but with the static keyword. The life starts (and the initialization happens) when the declaration first time encountered and ends when the program is finishing.

```
void f()
{
  static int cnt = 0;  // life starts here on the first occurance
  ++cnt;
}
// ...
int main()
{
  while (... )
  {
    f();
  }
}   // life finished here
```

Array and class elements

Array elemenst are created with the array itself in order of indeces. Array elements destroyed when the array itself is deleted.

Built-in arrays with static life should have size known by the compiler, aka constant expression. Since C++11 arrays with automatic life may have variable size.

```
int main()
{
  int n;
  // read n

  int t[n]; // variable size array, ok since C++11
}
```

Non-static data members are created when their holder object is created. If they have constructor, then their constructor will be woven into the container object constructor. The subobjects will be initialized by their constructor. However built-in types have no constructor, so they must be explicitly initialized.

A member of a union has two constraints:

- Member must not have constructor or destructor
- The union must not have static field.

Temporaries

temporary storage is created under the evaluation of an expression and destroyed when the full expression has been evaluated.

Lets consider the following example:

```
1 void f( string s1, string s2)
2 {
3     const char *cs = (s1+s2).c_str();
4     cout << cs;     // Bad!!
5
6     if ( strlen(cs = (s1+s2).c_str()) < 8 && cs[0] == 'a' ) // Ok
7         cout << cs;     // Bad!!
8 }
```

The problem is, that in line 4 and in line 7 when we refer to the temporary objects they may already be destroyed.

The correct way would be trust on the good, optimized implementation of the string class.

```
void f( string &s1, string &s2)
{
    cout << s1 + s2;
    string s = s1 + s2;

    if ( s.length() < 8 && s[0] == 'a' )
        cout << s;
}
```

When we assign a name to a temporary, the scope of the name will define the life of the temporary:

```
void f( string &s1, string &s2, string &s3)
{
    cout << s1 + s2;
    const string &s = s2 + s3;

    if ( s.length() < 8 && s[0] == 'a' )
        cout << s;  // Ok
} // s1+s2 destroyes here: when the const ref goes out of scope
```