Modern C++ - Common errors regarding scope and life

9. Common errors regarding scope and 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.

There are plenty of problems junior C++ programmers meet when make mistakes in scope or life rules.

How (not to) make scope-life errors?:

The task:

1. write a question to stdout
2. read a string as answer from stdin
3. print the answer to stdout

```
//
//  This is a VERY BAD program
//
#include <iostream>

using namespace std;
char *answer( const char *question);

int main()
{
    cout << answer( "How are you? ") << endl;
    return 0;
}
char *answer( const char *question)
```

```
{
    cout << question;
    char buffer[80];    // local scope, automatic life
                        // char[] converts to char*
    cin >> buffer;      // ERROR1: possible buffer overrun!!
    return buffer;      // ERROR2: return pointer to local: never do this!
}
```

There are two big errors in the code above:

1. The **cin » buffer** call reads charakters into **buffer** until the first separator. The buffer could be (and sooner or later will be) overflow, i.e. we read more charactres then the room we have. This *buffer overflow* problem is perhaps the most critical security errors in C++.
2. The function returns a pointer to an automatic life local variable. When we try to use that pointer, the memory behind it already gone. As the life of the local buffer is over, we may overwrite other values.

Lets try to fix the program with making **buffer** to global, therefore its life to static. Also we avoid buffer overrun problem using **getline**.

```
#include <iostream>

using namespace std;

char *answer( const char *question);
char buffer[80];   // global scope, static life

int main()
{
    cout << answer( "How are you? ") << endl;
    return 0;
}
char *answer( const char *question)
{
    cout << question;
//  char buffer[80];
    cin.getline(buffer,80);  // reads max 79 char + places '\0'
    return buffer;
}
```

This is working (in this example), but **buffer** is visible in too many places. This is a maintenance nightmare. In fact, buffer is not a global concept in this program, it is only an implementation detail of **answer** function.

We should try to narrow the scope of buffer.

```
#include <iostream>

using namespace std;

char *answer( const char *question);
// char buffer[80];   // global scope, static life

int main()
{
    cout << answer( "How are you? ") << endl;
    return 0;
}
char *answer( const char *question)
{
    cout << question;
    static char buffer[80];  // local scope, static life
    cin.getline(buffer,80);
    return buffer;
}
```

This works as we expected, and the scope of **buffer** is minimal. The buffer is not visible but still valid outside of the **answer** function.

However, this solution is also far from perfect:

```
#include <iostream>

using namespace std;

char *answer( const char *question);

int main()
{
    cout << answer("Sure?: ") << ", " << answer( "How are you?: ") << endl;
    return 0;
}
char *answer( const char *question)
{
    cout << question;
    static char buffer[80];
    cin.getline(buffer,80);
    return buffer;
}
```

```
$ g++ -ansi -pedantic -Wall -W howareyou.cpp
$ ./a.out
How are you?: fine
Sure?: yes
yes, yes
```

(Also, consider the reverse evaluation order for the two calls.)

The problem is that we have *only one* buffer for the two answers, and the second answer overwrites the first one. The second answer will be printed twice.

In real world the same situation happens with concurrent programs executing multiply *threads*.

We need a separate buffer for each simultanious calls of **answer**. Lets try this with dynamic memory.

```
#include <iostream>

using namespace std;

char *answer( const char *question);

int main()
{
    cout << answer("Sure?: ") << ", " << answer( "How are you?: ") << endl;
    return 0;
}
char *answer( const char *question)
{
    cout << question;
    char *buffer = new char[80];
    cin.getline(buffer,80);
    return buffer;
}
```

```
$ ./a.out
How are you?: fine
Sure?: yes
yes, fine
```

We finally have got two separate answers (in wrong order). But the real problem is hidden: no one freed the allocated buffers. In long run-time, with many calls of answer() we will run out of the memory!

This fenomenon is called **memory leak** and it is a *fatal error* in C++.

The right solution is to

1. Having an exact **owner** of every memory area. This case the owner is the *caller* function (here the main()).
2. Use **sequence points** to separate sequential events.

```
#include <iostream>

using namespace std;

char *answer( const char *question, char *buffer, int size);

int main()
{
    const int bufsize = 80;
    char buffer1[bufsize],
    char buffer2[bufsize];

    cout << answer("How are you?: ", buffer1, bufsize) << endl;
    cout << answer("Sure?: ", buffer2, bufsize) << endl;
    return 0;
}
char *answer( const char *question, char *buffer, int size)
{
    cout << question;
    cin.getline(buffer,size);
    return buffer;
}
```

This is correct, but also hard to maintain solution. The **main** function, which uses the memory allocates it. The **answer** function receives the parameters, and fills the buffer. The maximum size of the characters to read is passed as an extra parameter.

However, in C++ we can use the standard library **sdt::string** class. There are a lot of advantages of using std::string.

- The size of the answer is flexible, the memory behind the std::string grows dynamically on demand.
- The string class can be defined locally and **answer** can be returned by value (i.e.we copy the local string back). The characters behind the string will be copied by the *copy constructor* of std::string.
- When **answer** returns the local std::string object is destroyed (after its value copied from) by the *destructor* function of the std::string class. Therefore there will be no memory leak.

```
#include <iostream>
#include <string>     // for std::string class

using namespace std;

string answer( string question);

int main()
{
    string a1 = answer("How are you? ");
    string a2 = answer("Sure? ");
    cout << a1 << ", " << a2 << endl;
    return 0;
}
string answer( string question)
{
    cout << question;
    string answ;
    getline( cin, answ);
    return answ;
}
```

This code not only works well (even in multithreaded code) but also looks more natural.

```
$ ./a.out
How are you? fine
Sure? yes
fine, yes
```

Use the most straitforward solutions with the help of the standard library classes!