Abstract

Professional C++ code appears at various levels. Marshall Cline, the original owner of C++ FAQs, wrote in the first ed. of the book:

Q: How long does it take to become a good OOP in C++?

A: Between six months and three years, **depending on your definition of good**.

My experience was a bit different:

I got some big compliments on my code after three years of partially using C++, but my big "Ah Ha!" came in the fifth year, and then in the tenth year.

When we are experienced in programming, we follow most of the formal rules and practices for quality software. However, I believe that our inner state may differ, depending on our personality; namely, the better code is not made by only following formal rules. Indeed, I think that SW development is an art.

Therefore, I'll try to express some of my ideas on programming, as openly as possible, allowing different interpretations by different people.

[Full disclosure: All code examples are illustrations of my ideas from many years ago.]

C Code – Inspired by C++

You have 7 weeks to implement a given algorithm for FV

- A too long function – a bit over 40 lines

```
void MyFunction(t0 arg0, t1 arg1, t2 arg2, t3 arg3)
{
  some variables;
  some code;
  for (i = a; ManyTimes; Step) {
    Intensive work!
  }
  some concluding code;
}
```

C Code – The naive way

```
#define SUB_FUNCTION(marg0, marg1, marg2, marg3, marg4)          \
        {  for (marg0 = marg1; ManyTimes(marg2); Step(marg3) ) { \
              Intensive work with marg4!                         \
           }                                                     \
        }

void MyFunction(t0 arg0, t1 arg1, t2, arg2, t3 arg3)
{
  some variables;
  some code;
  SUB_FUNCTION(a, b, c, d, e);
  some concluding code;
}
#undef SUB_FUNCTION
```

C template – Using MACROs

Inspired by
a C-template linked-list
written by a member of VIS project:
A Synthesis & FV tool-set, UC Berkeley

```
#define M_ENTRY_T int // complex
#define T_ADD(a, b) ((a) + (b)))
                    // (c_add((a), (b))
#define MATRIX M_ENTRY_T**
...
#define M_LEN len //  my_mat_len()
                  // with a struct arg
#define M_NEW  Matrix(M_LEN)
```

```
MATRIX  M_ADD(m1, m2)
{ int i, j;
  MATRIX m = M_NEW;
  for (i = 0; i < M_LEN; ++i) {
    for (j = 0; j < M_LEN; ++j) {
      m[i][j] = T_ADD(m1[i][j],
                      m2[i][j]);
    }
  }
  return m;
}
```

A Template Connects two Tools

There were two (FV) tools that needed to be connected

- One written in C and one written in C++

There were good reasons to templatize the connection.

One class had to have a complicated structure, so I have consulted the members of ACCU (Association of C & C++ Users - https://accu.org/ I am a member since 1999) and the suggestions that I've got in a short while, solved my problem. I use this opportunity to recommend ACCU.

I don't have any record about that code that I have developed, except that the files were total of 10K lines – probably net of 8K LOC.

That code was termed as "*a marvelous piece of C++ code*".

Improving Efficiency

I have offered a company to improve a tool's performance:
- I only modify the C++ style (no change to algorithms)
- I don't touch functions of more than 100 LOC
The tool was 70K LOC

I partially worked there, for three months, while teaching at the Technion.
I dealt with about half of the code – didn't look at the other half.
Total improvement was almost 20% - functions were improved 3x – 6x.

At the end, I gave a presentation with examples of what I did.
Here is one example.

Improving Efficiency

A ~60 LOC functuion:

```
A function(params)
{ if (exp1) return exp11;  // They were a bit
  if (exp2) return exp21;  // similar,
  if (exp3) return exp31;  // not too similar

  switch(exp) {
       //  A few simple cases

       //  A single complicated case
  }
}
```

Improving Efficiency

What I did:

```
Template<class T> A t_function(arg){ };// Not easy unifying
/* This template is for that function() */
A function(params)
{ if (exp1) return t_fundtion(exp12);  // Similarity
  if (exp2) return t_fundtion(exp22);  // was
  if (exp3) return t_fundtion(exp32);  // obvious

  switch(exp) {
    //  A few simple cases – the same

    // A single case – simpler, with a function call
  }
}
```

function() became 24 LOC. Performance improved by 4x (CPU cache?)

Reducing Complexity: O(n*m) => O(n+m)

I am not a wizard.

Reducing computational complexity O(n*m) => O(n+m) is either magic, or just correcting a terrible error.

It is about reducing space complexity, and again – related to templates.

There was a template class with, say, 9 parameters:

```
template<class T1,… T9> // Not all types are different
class Simple { // Also relevant for value parameters.
  class A { Using T1 – T4  }
  class B { Using T5 – T9  }
  // Doing the work
};
```

Reducing Complexity: O(n*m) => O(n+m)

The Problem:

If there is a 1-1 correspondence between A's parameters and B's, as they appear in Simple, then for each instance of A, B is created only once, and vice versa.

But if the correspondence is not 1-1, there will be an instance of A that pairs with different Bs in different Simples, and vice versa. For such cases, instances of A and/or B will be duplicated inside different instances of class Simple.

```
template<class [9 int]>
class Simple {
  class A { Using 4 int };
  class B { Using 5 int };
  // Doing the work
};
```

```
template<class [4 int, 5 char]>
class Simple {
  class A { Using 4 int  };
  class B { Using 5 char };
  // Doing the work
};
```

This may also happen with a single inner class

Reducing Complexity: O(n*m) => O(n+m)

The Solution:

Use multiple inheritance!

```
template<class T1,.. T4>
class A {     };
```

```
template<class T5,.. T9>
class B {      };
```

```
template<class T1,.. T9>
class Simple: public A<T1,.. T4>, public B<T5,.. T9> {
   // ...
};      // A side benefit: The Single Responsibility Principle
```

A C-style module in C++ tool

The Problem:

A file a-la C: 2K LOC, arrays, enums, switches

The Solution:

Make it OO using advanced C++:

Some STL, classes, little multiple inheritance

- resulted in 30% CPU improvement
- despite heavy usage of virtual functions.

Q & A

My Worst Error in Programming

Thank You

- Coding with reason
  - In 97 Things Every Programmer should Know (K. Henny, ed.), 2010
- In Search of Bug Free SW
  - August Penguin 2017 – Hamakor
- Teaching C++ as the First PL
  - Core C++ 2019, Teaching C++ Workshop

A Spoiler (Cont.)

- **Linux Kernel accepts `goto` in some cases**
  - **I totally agree with their criteria**
  - **But it's a slippery slope – be careful**
- **Here is an industrial example:**

```
Status Class::set_status(int id, State state)
{    Status status = OK;
     Container::const_iterator itr = cont_.find(id);
     if (itr == cont_.end()) {
         status = NOT_FOUND;
         goto bail;
     }
     status = itr->second->act(state);
  bail:
     return status;
}
```

A Spoiler (Cont.)

- Linux Kernel accepts `goto` in some cases
  - I totally agree with their criteria
  - But it's a slippery slope – be careful
- Here is the code without `goto`:

```
Status Class::set_status(int id, State state)
{   Container::const_iterator itr = cont_.find(id);
    if (itr != cont_.end()) {
        return itr->second->act(state);
    }
    return NOT_FOUND;
}
```


24