← Back to index

How to avoid OOP

Shazan Shums #8919
October 11, 2016

I am a beginner in programming and when I look at code online like library code and github code.They are full of OOP and C++ stuff like serialization, inheritance, polymorphism,abstraction, templating and all those stuff which I don't understand. How to avoid this since I heard that OOP is bad for performance.
I would like to know how to convert OOP code to performance oriented code.I am a beginner so can you provide more detailed explanation.
Thanks.

hugo #8920
October 11, 2016

You could show us a project you have done that you want to improve the design.

This playlist is also very on the topic you're worried:
https://www.youtube.com/playlist?...Ltg36xKlu60wk3buYHvAdtSbqPsvNgUDV
This video has interesting criticism of OOP abuse:
https://www.youtube.com/watch?v=Q...Ltg36xKlu60wk3buYHvAdtSbqPsvNgUDV

But in the end "you can write bad code in any style." Try watching the whole Handmade Hero series...

Cheers!

Edited by hugo on October 11, 2016, 4:00pm

Jesse #8922
October 11, 2016

Be aware that transforming OOP code into something more machine centric is not a task for a beginner. It requires knowing both how OOP and how code that utilizes your machine well works!

I addition to hugo's videos, I encourage you watch Casey's intro to C stream and the accompanying Q&A. Then, start programming out problems you enjoy solving with the hardware in mind.

Good luck!

graeme #8925
October 11, 2016

> msmshazan said:
>
> I am a beginner in programming and when I look at code online like library code and github code.They are full of OOP and C++ stuff like serialization, inheritance, polymorphism,abstraction, templating and all those stuff which I don't understand.
> How to avoid this since I heard that OOP is bad for performance.
> I would like to know how to convert OOP code to performance oriented code.I am a beginner so can you provide more detailed explanation.
> Thanks.

serialisation is just the process by which data in a program is saved externally in a format to be used later. not oop, and c doesn't make life easy in this regard, but neither does c++. in OOP you see this implemented as a function call where the function is determined at run time rather than compile time (dynamic dispatch)

inheritance is a big pillar of oop but now oop evangelicals also say "composition over inheritance" so you can do oop without it if you wanted to. in OOP you see this implemented as a function call where the function is determined at run time rather than compile time (dynamic dispatch)

polymorphism in terms of code rather than objects just means code, including compiled machine code, can any operate on any type that conforms to a certain interface. in OOP you see this implemented as a function call where the function is determined at run time rather than compile time (dynamic dispatch). this actually cool but assuredly not for performance reasons--hence COM and D-Bus and cocoa

abstraction is a necessary part of really all programming, although the form it takes differs. the simplest abstraction is a function. In OOP.. dynamic dispatch, yet again

templates are c++'s mechanism for doing all of the above without dynamic dispatch

If the optimizer isn't able to figure out what you are trying to do, dynamic dispatch requires 1) a fetch of the vtable containing the function pointer to call (possible

cache miss) and 2) that the optimizer assume that the called function could have done anything, making it much harder for it to do its job. Reasoning about this stuff *properly* requires not only a working knowledge of how a computer operates but also a knowledge of what the compiler will produce when using a language's features--nowadays you can spend a decade programming and not learn much about either

Saticmotion #8992
October 17, 2016

An important point is that the largest problem, in my opinion, is the **Oriented** part of OOP. i.e. starting your project out, assuming everything needs to be object oriented, and designing your entire architecture around that assumption.

It is well possible that some parts of your codebase work really well if they're object oriented. And it's fine to have these parts. But wait until after you've written your code in the simplest way possible to come to this conclusion.

Thomas Kingsvik #8993
October 17, 2016

Avoiding new/delete
Consider the following:

```
1  class Thing {
2  public:
3      int i;
4      Thing(int In) {
5          this->i = In;
6          printf("Hey\n");
7      }
8  };
9
10 int main(void) {
11     Thing* t = new Thing(1);
12     return 0;
13 }
```

This is an example of what I consider unfortunate OOP: by thinking of Thing as its own conceptual entity, we have introduced allocating the memory on the heap rather than the stack. The downside of this is that new/delete requires the operating system to do work in dealing out virtual memory to the object, which can be bad for performance.

Much the same damage can be done in C without OOP:

```
struct Thing {
    int i;
};

int main(void) {
    Thing* t = malloc(sizeof(Thing));
    t->i = 1;
    return 0;
}
```

The way to avoid virtual memory is to not declare 't' as a pointer:

```
Thing t = {1};
```

This way it will be allocated on the stack, which is usually better and definitely better for small data bundles. It also has the advantage of the 'constructor' being voluntary: you can declare it as

```
Thing t;
```

which will not require the compiler to generate code assigning values to fields upfront, so that you can mete it out only when necessary.

Inheritance

Inheritance in C++ looks something like this:

```
class Parent {
public:
    int i;
    char a;
};

class Child : public Parent {
public:
    double d[3];
};
```

The traditional counterpart to inheritance is composition, which goes something like this in C:

```
struct Parent {
    int i;
    char a;
};
```

```
5
6  struct Child {
7      Parent parent;
8      double d[3];
9  };
```

The problem with this is that it can become verbose when we want to assign values to a Child:

```
1  Child c;
2  c.parent.i = 1;
3  c.parent.a = 'a';
4  c.d[0] = -1.0;
5  c.d[1] = 3.14;
6  c.d[2] = 0.0;
```

As you can see, when we want to access 'int i' and 'char a' we have to first access the Child's 'parent' field. If we keep composing structs like this, the problem compounds and it becomes tedious to write code.

My solution is to mostly avoid struct composition beyond one level, and to keep the names short. One way to add something that kinda looks like inheritance using macros is:

```
1  #define Parent \
2      int i;\
3      char a
4
5  struct Child {
6      Parent;
7      double d[3];
8  };
```

Now we have the option of code reuse as well as immediate access to the fields.

Ginger Bill #8994
October 17, 2016

Instead of subclassing/inheritance, the method I highly prefer is discriminated unions/tagged unions.

```
1  enum EntityKind {
2      Entity_Invalid,
3      Entity_Apple,
```

```
    Entity_Frog,
    Entity_Tree,
};

struct Entity {
    EntityKind kind;
    union { // anonymous union
        struct {
            float  weight;
            Colour colour;
        } Apple;
        struct {
            float volume;
            float jump_height;
        } Frog;
        struct {
            int     branch_count;
            Colour  leaf_colour;
            Entity *children;
            int     children_count;
        } Tree;
    };
};
```

As you'll note, the size of each `Entity` is the same. This means you can easily pre-allocate as much as you need without needing to know the precise type.
i.e.

```
Entity entities[100];
Entity *entities = alloc_array(Entity, 100);
```

Another great thing about tagged unions is that you can just

```
memcpy
```

each thing and even change it if necessary. This is very useful in many applications (such as nodes in a tree).
The main disadvantage of this method over subtype polymorphism, is that it requires more memory. Personally, I think that's very rarely a problem especially compared to the advantages of tagged unions.

If you want some form of a "vtable", I would suggest just using a switch statement and switch depending on the tag/kind/discriminator. This is also more flexible, and faster than C++'s inheritance implementation (as there is no double pointer indirection for functions).

> brothir said:
>
> Avoiding new/delete
> Consider the following:

```
class Thing {
public:
    int i;
    Thing(int In) {
        this->i = In;
        printf("Hey\n");
    }
};

int main(void) {
    Thing* t = new Thing(1);
    return 0;
}
```

> This is an example of what I consider unfortunate OOP: by thinking of Thing as its own conceptual entity, we have introduced allocating the memory on the heap rather than the stack. The downside of this is that new/delete requires the operating system to do work in dealing out virtual memory to the object, which can be bad for performance.

imo this is just poor use of C++ that comes from familiarity with more heap-focused languages and big oop-with-c++ proponents would agree. The very similar problem which is also problematic if used indiscriminately but otherwise considered good practice is, you know, throwing a unique_ptr in there and having the constructor allocate its members. The problem is not that Thing is a conceptual entity per se but that Thing is a conceptual entity which manages, Encapsulates, all of its members. This can be fixed perfectly well while still using C++ features and in a OOP way, but the more complex your use case the more you end up moving away from 'good oop' and the more esoteric your templates and exploitation of std becomes. Incidentally, maintaining the concept of encapsulation while still offering flexibility in how various members are constructed is the *entire reason* for dependency injection to exist, it's full banana cakes

Edited by graeme on October 18, 2016, 1:28am

+ Reply to Thread

← Back to index