Good coding habits and mentality

Coding convention

What are coding conventions

- A guideline for a team detailing how to write code for a codebase.
- Varies from team to team.
- The guide might include:
  - How to name variables
  - How to write comments
  - How to indent code

Coding convention examples

- Google
- Microsoft
- JSF-AV
- GCC
- MIT
- Bungie

Why have coding conventions?

Because many artists working on one canvas results in chaos.

Because we are writing code for others to read.

Because we are writing code for the you of tomorrow to read.

If there's a standardized way to read code, reading code becomes faster and that increases productivity

(if it slows productivity, maybe it’s time to relook at the guidelines)

Remember that it's just a guideline.

It's okay to get upset but don't get *too* upset if someone didn't follow it.

What is programming, really?

Programming is a craft

An artist who draws go through many iterations before ending up with a final product that they are satisfied with.

Programming is similar. We write code, throw away code, learn from mistakes and iterate until we end up with a final product that we are satisfied with.

A pen is just a tool.

The paper is the canvas.

Programming languages is a tool.

Our hardware (CPU, GPU, monitor, speakers, etc.) is the canvas.

And all canvases have constraints

(e.g. a paper has limited space. A computer has limited RAM)

And the more we practice, the better we become ☺

“Avoid premature optimization”

What is ‘premature optimization?’

- ‘Spending time on something that you might not need’.
- But how do you know what you are working on is what you might not need?

“Method A is bad.
Method B is definitely better!”

“Declaring variables is bad.

Putting everything into one statement is definitely better!”

“Rolling your own code is bad.
Using the other’s library is definitely better!”

“Long code blocks is bad.

Splitting them into many functions is definitely better!”

(in case you are really interested in the previous slides, I attached some relevant discussions here:

Jon Carmack on inlined functions and the HackerNews.net comments

When it comes to developing a product, we are pretty much in an ‘engineering’ field.

This means that we focus on practicality and bringing an imaginary product to reality

Spoiler alert!

There are no one-size-fit-all solutions.

There are no ‘wheels’ (yet).

And don’t believe everything you hear on the internet!

(otherwise, we’d be out of jobs soon)

When comparing two solutions, always PROFILE and MEASURE with the context of your project.

(Protip: If you are too lazy to profile now, then it's most probably not that important yet)

In the end, it’s all about balance! ☺

Premature optimization case study

Which is better?

```
struct Mat4f {
    float e[4][4];
};

Mat4f Identity1() {
    Mat4f result;
    result.e[0][0] = 1.f;
    result.e[1][1] = 1.f;
    result.e[2][2] = 1.f;
    result.e[3][3] = 1.f;

    return result;
}
```

```
struct Mat4f {
    float e[4][4];
};

Mat4f Identity2() {
    Mat4f result;
    for (int i = 0; i < 4; ++i) {
        result.e[i][i] = 1.f;
    }

    return result;
}
```

Premature optimization case study

Compiler: x86-64 clang (trunk), -O5 flag

```
Identity1():                            # @Identity1()
        mov     rax, rdi
        mov     dword ptr [rdi], 1065353216
        mov     dword ptr [rdi + 20], 1065353216
        mov     dword ptr [rdi + 40], 1065353216
        mov     dword ptr [rdi + 60], 1065353216
        ret
```

```
Identity2():                            # @Identity2()
        mov     rax, rdi
        mov     dword ptr [rdi], 1065353216
        mov     dword ptr [rdi + 20], 1065353216
        mov     dword ptr [rdi + 40], 1065353216
        mov     dword ptr [rdi + 60], 1065353216
        ret
```

Premature optimization case study

Compiler: x86-64 clang (trunk), -O1 flag

```
Identity1():                            # @Identity1()
        push    rbp
        mov     rbp, rsp
        mov     rax, rdi
        movss   xmm0, dword ptr [rip + .LCPI0_0] # xmm0 = mem[0],zero
        movss   dword ptr [rdi], xmm0
        movss   xmm0, dword ptr [rip + .LCPI0_0] # xmm0 = mem[0],zero
        movss   dword ptr [rdi + 20], xmm0
        movss   xmm0, dword ptr [rip + .LCPI0_0] # xmm0 = mem[0],zero
        movss   dword ptr [rdi + 40], xmm0
        movss   xmm0, dword ptr [rip + .LCPI0_0] # xmm0 = mem[0],zero
        movss   dword ptr [rdi + 60], xmm0
        pop     rbp
        ret
```

```
.LCPI0_0:
        .long   0x3f800000                      # float 1
Identity2():                            # @Identity2()
        push    rbp
        mov     rbp, rsp
        mov     qword ptr [rbp - 24], rdi       # 8-byte Spill
        mov     qword ptr [rbp - 16], rdi       # 8-byte Spill
        mov     dword ptr [rbp - 4], 0
.LBB0_1:                                # =>This Inner Loop Header: Depth=1
        cmp     dword ptr [rbp - 4], 4
        jge     .LBB0_4
        mov     rax, qword ptr [rbp - 24]       # 8-byte Reload
        movsxd  rcx, dword ptr [rbp - 4]
        shl     rcx, 4
        add     rax, rcx
        movsxd  rcx, dword ptr [rbp - 4]
        movss   xmm0, dword ptr [rip + .LCPI0_0] # xmm0 = mem[0],zero,zero,zero
        movss   dword ptr [rax + 4*rcx], xmm0
        mov     eax, dword ptr [rbp - 4]
        add     eax, 1
        mov     dword ptr [rbp - 4], eax
        jmp     .LBB0_1
.LBB0_4:
        mov     rax, qword ptr [rbp - 16]       # 8-byte Reload
        pop     rbp
        ret
```

Premature optimization recap

- So which code is better?
  - Do we avoid loops from now on?
    - (of course not!)
- Consider the pros and cons.
- Ask if it matters to your product now.
- Profile if you really think it matters.
- Don't forget the more pressing problems that you need to solve!

tl;dr don't create problems when you have none!

(the best algorithm is no algorithm!)

‘Compression-Oriented’ Programming

A bottom-up approach to programming

Caveats

- This is just an approach some people use in programming.
- Useful in developing products, games, or R&D, where we iterate a lot and constantly change decisions based on how we progress.
- Useful for learning new technologies as well!
- It has its pros and cons, like all approaches. It might not be applicable to other programming domains.

The mentality to have when doing ‘Compression-Oriented’ Programming

Some of us are quick to pre-define groups of code (structs, functions, etc) before writing code that actually DOES something.

This happens when we think we know the ‘shape’ of our problem.

You usually don’t

(unless you really have tons of experience in that field)


+59.6
+57.6
6,816.1 +44.6
743 +56.5
20
1,037 +2
6/6
0
300
300
TURN 515/1500
1070 AD
5:48 PM
World Tracker
Square Rigging
Turns: 30
Kill a unit with a Musketman.
Mercenaries
1
Turns: 11
Have 8 land combat units in your milit...
Perth
Armagh
York
Cork
Cardiff
London
Brussels
Coast
Bristol Channel
Movement Cost: 1
1 Food
1 Gold
PLEASE WAIT
75 / 89


+5
194 (+5)
6
113/500
19/25 (+1)
Turn: 19 3240 BC | HELP | MENU
Bronze Working (1)
2 WARSAW
3 VIENNE
NEXT TURN

To understand the problem, we must explore the problem concretely.

(But how?)

Instead of planning from the start and having faith that the plan will work, we write code first and have faith that making mistakes will give you a better idea about the problem.

- Write simple procedural code.
- Use simple data structures (arrays!).
- This will ensure that your code is performant by default and straightforward for you (and others) to understand.

- Hard code and use global variables if you must.
- Copy and paste if you need to.
- We will refactor them later.
- KEEP IT AS SIMPLE AS POSSIBLE

Use advanced techniques and algorithms only when identified to be absolutely necessary!

(if you don't understand them, don't do it!)

The most important thing is
to get a task done first
before refactoring

When faced with a problem

Step back

Re-evaluate the issue

THEN continue to code.

Rinse and repeat.

Every problem encountered is a win!

It's progress!

You learnt something!

Don’t be afraid to delete code and rewrite code! Archive it somewhere if you are feeling protective!

(Good programmers delete/archive code and move on with their lives)

The ‘refactoring’ step

Now that your task is done, you should have the full picture of what makes your code works!
There should be no more mysteries!

We refactor to review and cleanup our code to see if we can express what we want better

Extract patterns that you see in your finished task's code and group them into functions or structs to improve scalability and readability.

We let the patterns in our messy but working code tell us what groups of code exists, which we can then decide to compress them into structs, functions, etc

In other words, we let the problem tell us how to group the code, not the other way around!

As we code more, we become better at ‘guessing’ the shape of future, similar problems.

The process then becomes faster because you make certain decisions faster.

It's like a craft really.

Like cooking.

Like drawing.

Like engineering.

Like acting.

Like programming.

Remember that there is no hard rule in what is a ‘task’ and when to ‘refactor’.

This is up to experience and personal preference, which only gets better with more practice.

Case Study

```
void DrawTextboxes() {

    // Simple textbox rendering code, where we render buttons
    // from top to bottom.
    //
    // Messy, hard coded, but assume it is working.
    // There is *some* pattern, so let's try to extract.
    float padding = 10.f;
    float current_x = 10.f;
    float current_y = 10.f;
    {
        float button_w = 100.f;
        float button_h = 100.f;
        CP_DrawRect(current_x, current_y, button_w, button_h);
        CP_Font_DrawText("Play", current_x + 20.f, current_y + 20.f);
        current_y -= button_h - padding;
    }

    {

        float button_w = 50.f;
        float button_h = 50.f;
        CP_DrawRect(current_x, current_y, button_w, button_h);
        CP_Font_DrawText("Credits", current_x + 20.f, current_y + 20.f);
        current_y -= button_h - padding;
    }
    //...etc
    // Would start to look messy if we have like 5 or 10 of these.
}void DrawTextboxes() {

```

Case Study

- Note the uses of code blocks and local variables
- Note that you can see a pattern in the code
- The code works, but it is not scalable.
- It is getting harder to think about all the variables at once.
- We kind of want to group them into ‘smaller units’ of sorts. (You can think about a ‘unit’ in C as something between two curly braces {})

```
36/// 2
37struct VPanel{
38    float padding;
39    float current_x;
40    float current_y;
41};
42
43void VPanel_Begin(VPanel* panel,
44                  float start_x,
45                  float start_y,
46                  float padding)
47{
48    panel->current_x = start_x;
49    panel->current_y = start_y;
50    panel->padding = padding;
51}
52
53void VPanel_PushTextbox(VPanel* panel,
54                        float w, float h,
55                        const char* text)
56{
57    CP_DrawRect(panel->current_x, panel->current_y, w, h);
58
59    // TODO: Maybe expose the text offset as a function argument
60    // as well? Or is that too many things for the user to input?
61    CP_Font_DrawText(text, current_x + 20.f, current_y + 20.f);
62    panel->current_y -= h - panel->padding;
63}
64
```

```
66void RefactoredDrawTextboxes() {
67    VPanel panel = {0};
68    VPanel_Begin(&panel, 10.f, 10.f);
69    VPanel_PushTextbox(&panel, 100.f, 100.f, "Play");
70    VPanel_PushTextbox(&panel, 50.f, 50.f, "Credits");
71    //...etc
72    // 10 of these would be easier to read.
73    // VPanel is reusable.
74    // Easier to think about.
75    //
76}
```

Case Study

- The code is more compact now!
- If another part of your code base a similar system, we can simply reuse the struct and its related functions!
- Maybe we can start adding more items in VPanel other than textboxes?

Final words

Remember that ALL CODE becomes Machine Language/Assembly indstuctions at some point for a CPU to run.

- ALU used for calculation
- FPU used calculate floating point values
- CU to run instructions
- RAM/Caches/Registers for data storage.
- GPU for parallel processing and display

The computer is a canvas, and the language is just a tool.

- No matter what tool an artist use to draw on a piece of paper, they are bound by the constraints of a paper.

There is no magic

Influences

- Mike Acton's "Data-Oriented Design and C++" talk
  - This is before he became VP of Unity DOTS
  - https://www.youtube.com/watch?v=rX0ItVEVjHc
- Casey Muratori's "Semantic Compression" post
  - Worked in RAD Tools, which many of its tools are still used in professional apps today from Age of Empires to Destiny to Unreal Engine.
  - https://caseymuratori.com/blog_0015
- Jonathan Blow's "How to program indie games"
  - Creator of Braid and The Witness.
  - https://www.youtube.com/watch?v=JjDsP5n2kSM&t=1477s