Executable Bloat

How it happens and how we can fight it

Andreas Fredriksson <dep@dice.se>

Background

- Why is ELF/EXE size a problem?
  - Executable code takes up memory
  - Consoles have precious little RAM
  - More available memory enables richer gameplay
  - It is a lot of work to fix after the fact

Software Bloat

Software bloat is a term used to describe the tendency of newer computer programs to have a larger installation footprint, or have many unnecessary features that are ***not used by end users***, or just generally use more system resources than necessary, while ***offering little or no benefit to its users***.

- Wikipedia has this definition (emphasis mine)
- We are wasting memory we could be using to make better games

C++ Bloat

- Let's look at a few typical C++ techniques
  - Some are even enforced in coding standards
- Often recommended in literature
- Often accepted without question

Interface Class Hiding

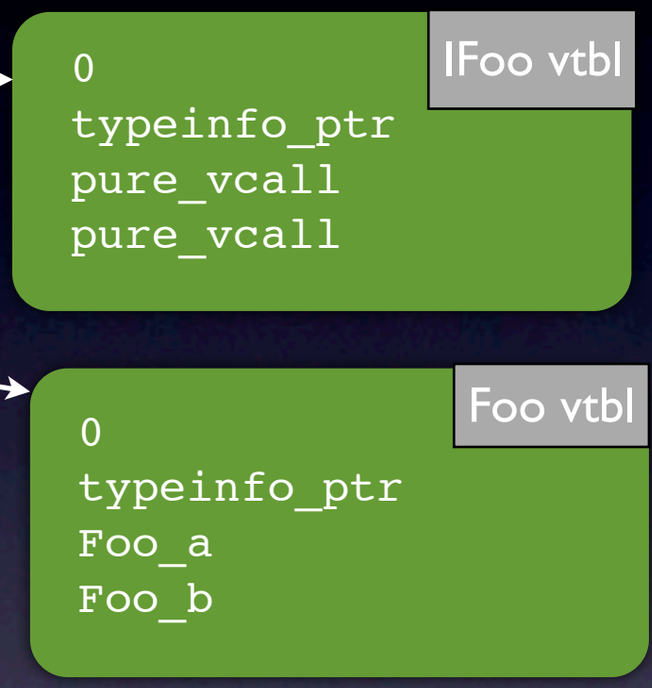
- Hiding single class Foo behind interface IFoo
  - Intent is to avoid including heavy Foo header everywhere
- Sometimes used for PRX/DLL loading

```
struct IFoo {
  virtual void func() = 0;
};

class Foo : public IFoo {
  virtual void func();
};
```

Interface Class Hiding

```
struct IFoo {
  virtual void a() = 0;
  virtual void b() = 0;
};
class Foo : public IFoo {
  virtual void a();
  virtual void b();
};
```



- One hidden cost - virtual tables
  - One for each class
  - In PPU ABI cost is higher as `Foo_a` will be a pointer to a 4+4 byte ABI struct

Interface Class Hiding

- Additional overhead
  - Every call site needs vcall adjustment
  - ctor/dtor needs vptr adjustment
- Total SNC overhead, 8 functions: 528 bytes
  - Likely more, due to callsites
  - These bytes have no value = bloat

Excessive Inlining

- Typically associated with templates
- Templates are almost always inline
- Smart pointers
- String classes

Excessive Inlining

```
bool eastl_string_find(eastl::string& str) {
    return str.find("foo") != eastl::string::npos;
}
bool c_string_find(const char* str) {
    return strstr(str, "foo") != 0;
}
```

- Compare two string searches
  - c_string_find version - 56 bytes (SNC)
  - eastl_string_find - 504 bytes (SNC)
- These bytes add zero value = bloat

Excessive Inlining

- Many other common inlining culprits
  - operator+= string concatenation
  - Call by value w/ inline ctor
- Hard to control via compiler options
  - Sometimes you want inlining
  - Better to avoid pointless cases

Static Initialization

```
static const eastl::string strings[] = { "foo0", "foo1" };
static const Vec3 vecs[] = { { 0.2, 0.3, 0.5 }, { ... } };
```

- Extreme hidden costs for constructors
- Generates initializer function
  - Have seen arrays of this kind generate over 20 kb of initializer code in the wild (SNC)
    - Array of 10 eastl::strings - 1292 bytes
    - Array of 10 vec3 - 368 bytes

Static Initialization

```
static const char* strings[] = { "foo0", "foo1", ... };
static const float vecs[][3] = { { 0.2, 0.3, 0.5 }, ... };
```

- Just don't do it - prefer POD types
  - Make sure data ends up in .rodata segment
  - Adjust code using array accordingly
- Alternatively make data load with level
  - No space overhead when not used

operator<<

- A special case of excessive inlining
- Creeps up in formatted I/O
  - Assert macros
- Prefer snprintf()-style APIs if you must format text at runtime
  - Usually less than half the overhead
  - Ideally avoid text formatting altogether

Sorting

- STL sort is a bloat generator
  - Specialized for each type - faster compares..
    - ..but usually whole merge/quicksort duplicated per parameter type! - often around 1-2kb code
  - We have 140 sort calls in the code base - up to 280 kb overhead..

Sorting

- Prefer qsort() for small/medium datasets
  - Adds callback overhead on PPU..
  - Rule of thumb - qsort < 32-64 elements
- Same applies to many other template algorithms
  - Use only when it really buys something

Part 2:
What you can do

Accept the Domain

- Console coding is a very specific problem space
  - Think and verify before you apply general desktop C++ advice or patterns
    - Bloat is caused by humans, not compilers
- Example: "Virtual functions are essentially free"
  - True on x86 architecture (most of the time)
  - On PS3 PPU often two cache misses - ~1200 cycle penalty + ELF size bloat already covered

Day to day

- Think about full impact of your changes
  - .text size impact
  - .data/.rodata size impact
  - Bring it up & discuss in code reviews
- *Make sure your code is reasonably sized for the problem it solves!*

Avoid “reuse” bloat

- YAGNI - “You ain’t gonna need it”
  - Just write simple code that can be extended if needed
  - We typically never extend systems without altering their interfaces anyway
- Game code is disposable, a means to an end
  - Make sure it works well NOW
  - Avoid “single-feature frameworks”

Avoid repetition

- Can often move repeated code to data
  - Higher information density but same end result

```
RegisterFunc("foo", func_foo);
RegisterFunc("bar", func_bar);
RegisterFunc("baz", func_baz);
// ...
```

```
static const struct {
    const char *name; void (*func)(void);
} opdata[] = {
    { "foo", func_foo },
    { "bar", func_bar },
    { "baz", func_baz },
    // ...
};

for (int i=0; i < sizeof_array(opdata); ++i)
    RegisterFunc(opdata[i].name, opdata[i].func);
```

Compiler Output

- Look at the generated code!
  - That's what you're checking in, not C++
  - Don't assume code is improved by the compiler
  - No magic going on, compilers are stupid
- Develop an intuition for what to expect
  - Verify assumptions as you code

Assembly

- Learn enough assembly to read compiler output
  - Function calls (calling convention)
  - Memory loads and stores
  - FP/Vector instructions
- It's not very difficult - just do it
  - Also improves your debugging skills

Guidelines

- Avoid string classes, concatenation
  - Excessive inlining
- Avoid template containers for simple problems
  - Inlining + instantiation cost
  - Prefer C arrays for most jobs

Guidelines

- Avoid complex types in function signatures and interfaces
  - Requires caller to jump through hoops
  - Often bloats all call sites
  - Prefer raw POD types
    - `(T* ptr, int count)` is better than `(std::vector<T>&)`

Guidelines

- Avoid inheritance, interfaces and virtual functions
- Hidden costs are subtle
- Prefer function pointers for callbacks
- Prefer free functions on predeclared types for header stripping

Guidelines

- Avoid operator<< streaming
  - Prefer printf() style APIs
  - Easy to make your own formatter for often-used types
- Avoid singletons
  - They just add bloat around data that's just as global anyway
  - Prefer free functions around static data

Summary

- Make sure the code/data you're adding is reasonably sized for the problem it solves
  - Use no more than necessary
- Pick up some assembly and look at the compiler output
- Always measure, examine & question!

Questions?

Twitter: @deplinenoise
Email: <dep@dice.se>