Subverting the C++ compiler

Gabriel Laskar <gabriel@lse.epita.fr>

What this talk is about ?

- Take control back from the compiler
- Force the compiler to do what you want
- You are not compiler’s slave !

Sometimes we have to make sacrifices

Portability

Best practices

innocence... C++ can be hard and ugly !

Everytime you use something like that…

a kitten die

lambda functions

```
extern "C" void function(void (*f)());
```

```
int main() {
    int a = 12;

    function([]() {
        printf("%u\n", 14); // OK
    });

    function([&]() {
        printf("%u\n", a); // KO
    });

}
```

error:

cannot convert ‘**main()::<lambda()>**’ to ‘**void (*)()**’ for argument ‘1’ to ‘void function(void (*)())’

Lambda functions are objects ! :(

What can we do about it ?

```
std::function<void()> saved_func;

void callback() {
    saved_func();
}

callback capture(std::function<void()> f) {
    saved_func = f;
    return callback;
}
```

```
int main() {
    int a = 12;
    function(capture([&]() {
        printf("%u\n", a);
    }));
}
```

... But we can have that for only one lambda :(

Ideal Solution : Partial function application

```
let f x y = x + y
let g = f 2
let _ = print_int (g 3)
```

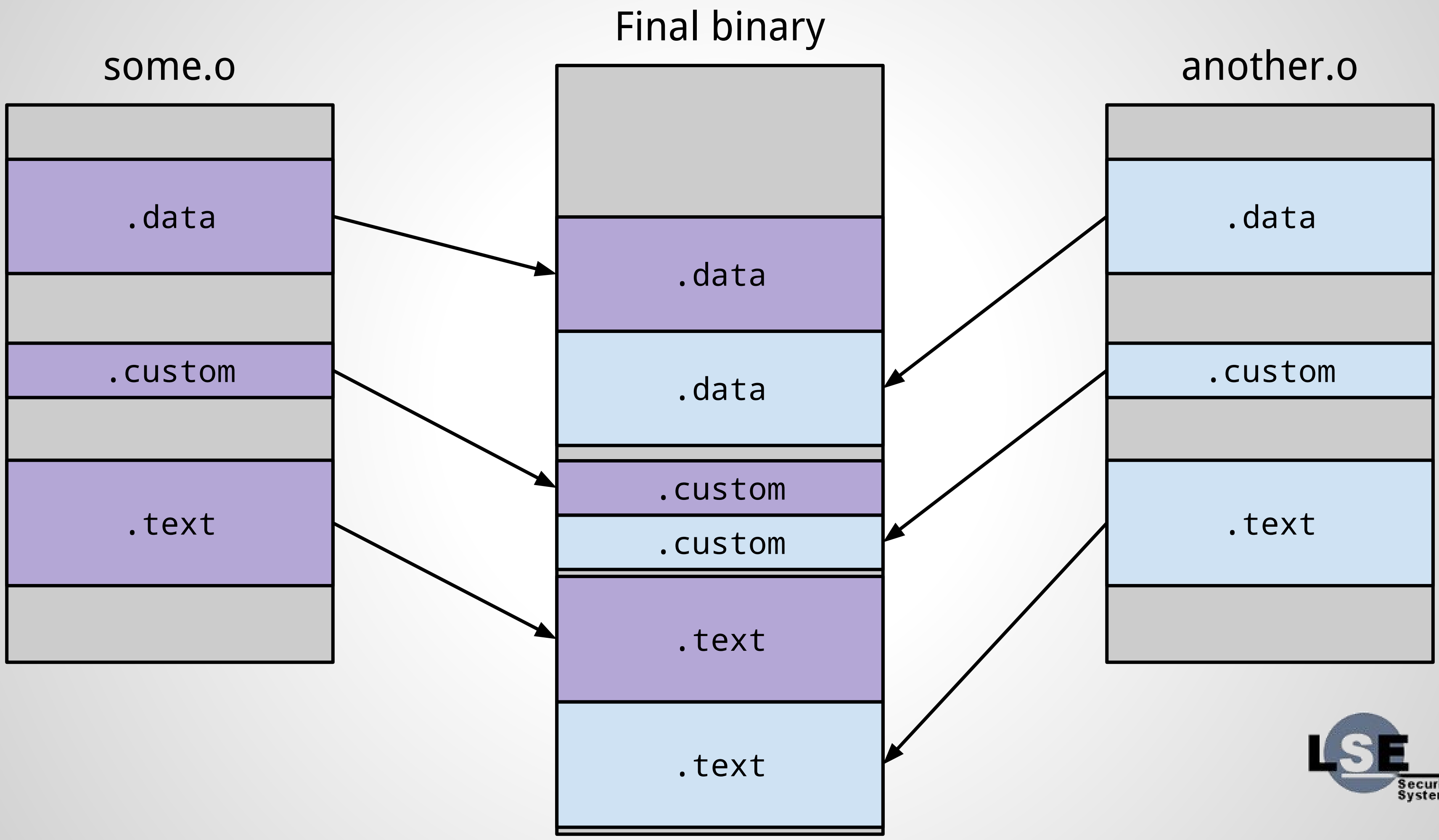
Ok, now in C ?

```
template <typename T, typename T2>
callback jit_this_call(T *f, T2 *arg) {
     unsigned char buf[] = {
          0x48, 0xbf, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, // param
          0x49, 0xbb, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, // address
          0x41, 0xff, 0xe3
     };

     char *addr = (char*)mmap(nullptr, sizeof(buf),
          PROT_WRITE | PROT_EXEC, MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);

     memcpy(addr, buf, sizeof(buf));

     *(void**)(addr + OFFSET_PARAM) = arg;
     *(void**)(addr + OFFSET_ADDR) = (void*)f;

     mprotect(addr, sizeof(buf), PROT_EXEC);

     return reinterpret_cast<callback>(addr);
}
```

```cpp
template <typename T>
callback capture(std::function<T> func) {
    return jit_this_call(caller<T>, &func);
}

int main() {
    int a = 12;
    function(capture<void()>([&]() {
        printf("%u\n", a);
    }));
}
```

Sections in ELF

Sections are a way for the compiler to organise binary data for the link


Final binary
some.o
another.o
.data
.custom
.text
.data
.data
.custom
.custom
.text
.text
.data
.custom
.text

Let’s register objects automatically!

```c
typedef int (*Constructor)();

#define constructor(Name) \
    static Constructor ctor_## Name \
    __section("constructors") __used = Name

extern Constructor __start_constructors[];
extern Constructor __stop_constructors[];

void init() {
    Constructor *ctor = __start_constructors;
    for (; ctor < __stop_constructors; ctor++)
        (*ctor)();
}
```

```c
int foo() {
    puts(__PRETTY_FUNCTION__);
    return 0;
}

int bar() {
    puts(__PRETTY_FUNCTION__);
    return 0;
}

constructor(foo);
constructor(bar);

int main() {
    init();
}
```

Runtime Code Selection

Static code selection… at runtime !

- An issue with opengl calls for example
- What is the standard solution ?
- How can we do better ?

```
extern "C" void glTextureParameteriARB(GLuint, GLenum,
                                        GLenum, GLint);

extern "C" void glTextureParameteri_fallback(GLuint
texture,
                                             GLenum target,
                                             GLenum pname,
                                             GLint param)
{
    GLint cur_tex;
    glGetIntegerv(GL_TEXTURE_BINDING_2D, &cur_tex);
    glBindTexture(target, texture);
    glTexParameteri(target, pname, param);
    glBindTexture(target, cur_tex);
}
```

Standard solution

```
extern "C" void (*glTextureParameteri)(GLuint, GLenum,
                                      GLenum, GLint);

int main()
{
    if (BindlessSupport) {
        glTextureParameteri = glTextureParameteriARB;
    } else {
        glTextureParameteri = glTextureParameteri_fallback;
    }
}
```

And if we don’t want the function pointer call ?

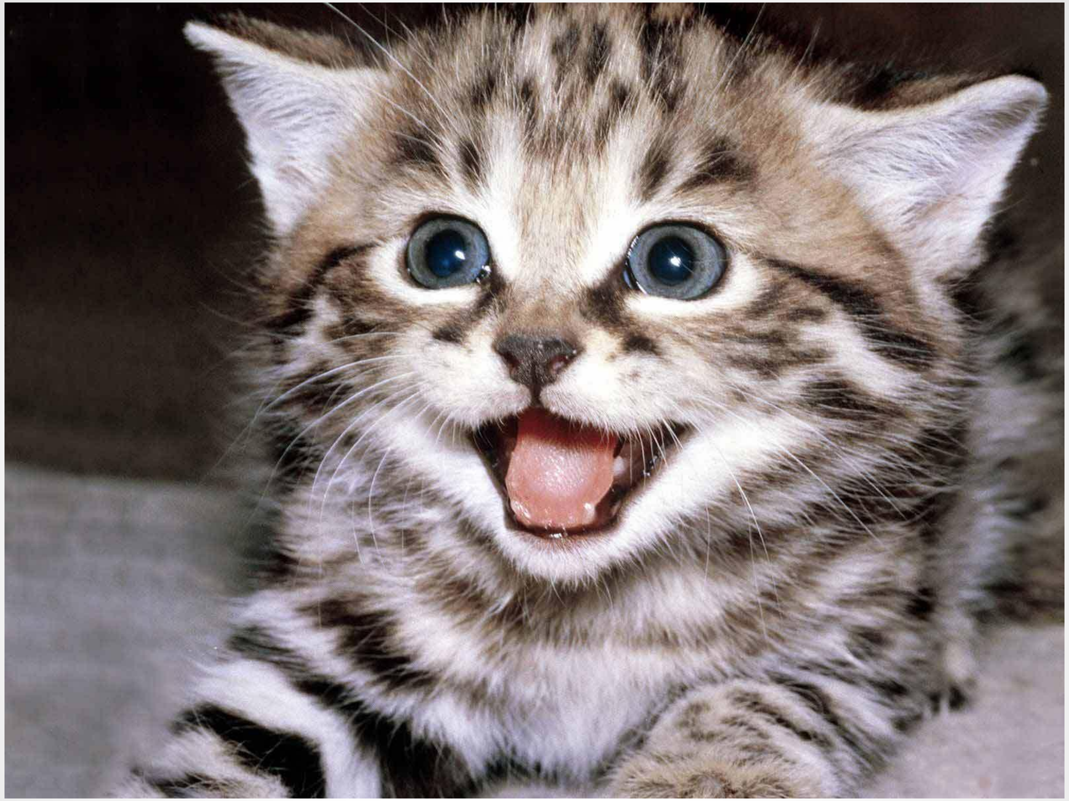
```cpp
enum Feature {
    FeatureA,
    FeatureB
};

template <Feature feature>
void function();

template <>
void function<FeatureA>() {
    puts(__PRETTY_FUNCTION__);
}

template <>
void function<FeatureB>() {
    puts(__PRETTY_FUNCTION__);
}
```

```c
struct alt_call {
    void* call_offset;
    void* call_replacement;
    int predicate;
    unsigned size;
};
```

```
extern "C" struct alt_call __start_altcalls[];
extern "C" struct alt_call __stop_altcalls[];

void patch_alternatives(int predicate)
{
    for (struct alt_call *e = __start_altcalls; e != __stop_altcalls; ++e)
{
        if (predicate == e->predicate) {
            MapWritable mapping(e->call_offset);

            memcpy(e->call_offset, e->call_replacement, e->size);
        }
    }
}
```

```
int main(int argc, char **argv)
{
    if (argc < 2) {
        return 1;
    }

    patch_alternatives(*argv[1] == 'b' ? FeatureB : FeatureA);

    alternative<&function<FeatureA>, &function<FeatureB>, FeatureB>();
    alternative<&function<FeatureA>, &function<FeatureB>, FeatureB>();

    return 0;
}
```

```
template <void T1(), void T2(), int predicate>
inline void alternative()
{
    asm volatile ("1: movq %0, %%r15\n"
        "2:call *%%r15\n"
        ".pushsection .altcalls_instr, \"ax\"\n"
        "    3:movq %1, %%r15\n"
        ".popsection\n"
        ".pushsection altcalls, \"a\"\n"
        "    .quad 1b\n"
        "    .quad 3b\n"
        "    .long %c2\n"
        "    .long 2b - 1b\n"
        ".popsection\n"
        : : "i"(T1), "i"(T2), "i"(predicate) : "r15", "memory");
}
```



\o/

```cpp
struct MapWritable {
    void *base_addr;
    unsigned size;
    unsigned old_flags;

    static inline void* align_page(void *addr)
    {
        return (void *)((unsigned long)addr & ~((1 << 12) - 1));
    }

    MapWritable(void *base_addr, unsigned size = 4096,
                unsigned old_flags = PROT_READ | PROT_EXEC)
        : base_addr(base_addr), size(size), old_flags(old_flags)
    {
        mprotect(align_page(base_addr), size, old_flags | PROT_WRITE);
    }

    ~MapWritable()
    {
        mprotect(align_page(base_addr), size, old_flags | PROT_WRITE);
    }
};
```

Bonus : Template All the things !

I need a string as a template parameter...

```
template <const char *s>
void foo() {
    puts(s);
}

int main() {
    foo<"pouet">();
}
```

error:

‘"pouet"’ is not a valid template argument

for type **‘const char*’**

because string literals can never be used in this context

No String Literals But...

```
template <char... String>
void foo() {
    char str[] = { String... };
    puts(str);
}

int main() {
    foo<"pouet"[0], "pouet"[1], "pouet"[2],
        "pouet"[3], "pouet"[4], "pouet"[5]>();
}
```

‘$’ is valid in an identifier ?

```
int main() {
    const char *foo$bar = "hey";
    puts(foo$bar);
}
```

```
template <char... String>
void foo() {
    char str[] = { String... };
    puts(str);
}

#define $(s) \
    getChr(s,0), getChr(s,1), getChr(s,2), getChr(s,3), \
    getChr(s,4), getChr(s,5), getChr(s,6), getChr(s,7), \
    getChr(s,8), getChr(s,9), getChr(s,10)

#define MAX_CONST_CHAR 10

#define MIN(a, b) ((a) < (b)) ? (a) : (b)

#define getChr(name, idx) \
    ((MIN(idx, MAX_CONST_CHAR)) < sizeof(name) ? name[idx] :
0)

int main() {
    foo<$("pouet")>();
}
```