Introducing C++

David Chisnall

March 15, 2011

Why Learn C++?

- Lots of people used it to write huge, unmaintainable code...
- ...which someone then gets paid a lot to maintain.

C With Classes

- Predecessor of C++
- Added Simula-like classes to C
- Not a terrible language

Then It All Started To Go Wrong...

- Strong typing (kind of)
- References as well as pointers
- Reusing keywords
- Templates
- Operator overloading

These are just some of the exciting features that you can use to make C++ code completely unreadable!

Object Orientation

Or, how to completely miss the point

- C has structures
- C++ has classes
- C++ classes are C structures
- Only with some confusing things added

Member Functions

A bit like methods, in an object-oriented language

```
// C++
struct Foo
{
    int a;
    void get_a(void);
};
// C
struct Foo
{
    int a;
};
void get_a(struct Foo *this);
```

Subclassing?

```
// C++
struct Foo
{
    int a;
    void get_a(void);
};
struct Bar : Foo
{
    void get_a(void);
};
void get_a(struct Foo *this);
```

- Which is called if I do `x.get_a()`?
- Depends on the static (declared) type of `x`;

Virtual?

```
// C++
struct Foo
{
    int a;
    virtual void get_a(void);
};
struct Bar : Foo
{
    virtual void get_a(void);
};
void get_a(struct Foo *this);
```

- Now which is called if I do `x.get_a()`?
- Depends on the run-time (real) type of `x`;

How Does That Work?

```
// C version:
static void foo_get_a(struct Foo*);
static void bar_get_a(struct Bar*);
struct vtable
{
    void (*get_a)(struct Foo*);
};
struct vtable foo_vtable = {foo_get_a};
struct vtable bar_vtable = {bar_get_a};
```

- Each class has a *vtable*.
- Table of function pointers, used for resolving calls

How VTables are Used

```
// C version
struct Foo
{
    // Not visible to C++ code
    struct vtable vtable;
    int a;
};
struct Foo f;
// Compiler generates this
f->vtable = foo_vtable;
// C++ f.get_a() is (roughly) equivalent to:
f->vtable->get_a(&f);
```

Performance of `virtual`

- Every `virtual` call involves an indirect call
- In position-independent code, so do non-virtual calls
- You often get better cache performance from the `virtual` lookup
- It's much harder to inline `virtual` calls
- So generally they're slower

Function Overloading

```
// This is not allowed in C:
float max(float a, float b)
{
    return a>b ? a : b;
}
int max(int a, int b)
{
    return a>b ? a : b;
}
```

- The function that is called will be determined by the arguments

How Does This Work?

```
$ cat simple.c
int max(int a, int b) { return a>b ? a : b; }
$ cc -c simple.c
$ nm simple.o
0000000000000000 T max
$ c++ -c  overload.cc
$ nm overload.o
0000000000000000 T _Z3maxff
000000000000003e T _Z3maxii
$ c++filt _Z3maxff
max(float, float)
```

- C++ version does *name mangling*
- Parameter types encoded in function names

Multiple Inheritance

```
struct A { int a; };
struct B { float a; };
// C inherits from A and B
struct C : A , B { int c; };

int main(void)
{
    C c;
    printf("A: %p\n", (A*)&c);
    printf("B: %p\n", (B*)&c);
    printf("C: %p\n", (C*)&c);
}
```

Pointer Casts Do Arithmetic in C++!

```
$ c++ inherit.cc
$ ./a.out
A: 0x7fff5fbff860
B: 0x7fff5fbff864
C: 0x7fff5fbff860
```

- Multiple inheritance means you must be able to turn a pointer to an object into a pointer to any superclass
- This means that pointer casting must involve arithmetic
- This leads to lots of subtle problems

How Multiple Inheritance Works

```
// Same example, this time in C
struct A { int a; };
struct B { float a; };
struct C
{
    struct A A;
    int c;
    struct B B;
};
```

Memory layout is not guaranteed by C++. Different compilers may do things differently.

No More Type Escaping

- C lets you cast any pointer type to and from `void*`
- C++ can't let you do this, because pointer casts can do arithmetic depending on the types

Data Hiding

Something C++ pretends to have

```
class A
{
    private:
    int a; // Accessible only from this class
    protected:
    int b; // Accessible from any subclass
    public:
    int c; // Accessible from anywhere
}
```

- C++ has somewhat pointless access specifiers
- Only enforced by the compiler, not checked at runtime
- Can be bypassed by pointer arithmetic
- All fields must be specified in the header, so no actual hiding is possible

Real Data Hiding

```
// In the header:
typedef struct foo* foo_t;
int someMethod(foo_t a);
// In a single source file:
struct foo
{
int a;
};
```

- Works in C too!

A Slightly More C++ Way

```
// In a header:
class Public
{
    // Factory method
    static Public* Create();
    // Pure virtual method
    virtual int someMethod() = 0;
};
// In a source file:
class Private : public Public
{
    int a;
    virtual int someMethod() { return a; }
};
Public* Public::Create() {return new Private();}
```

Questions?