Virtual Method Dispatch in C++

What are virtual methods?

- Virtual methods are methods that can be overridden by subclasses
- When a virtual method is invoked, the version of the method that is executed is determined by which the subclass of the object the method is invoked on
- This is in contrast to non-virtual methods which cannot be overridden, but instead “shadowed”
- Unlike C++, in Java methods are virtual by default

What are virtual methods?

- Virtual method calls incur a small performance penalty because they are calculated jumps compared to unconditional call instructions

Non-virtual method example (aka shadowing)

```
#include <cstdio>

class Base {
public:
    virtual ~Base() { }

    virtual void virtual_dispatch_method() {
        puts("Base::virtual_dispatch_method");
    }

    void static_dispatch_method() {
        puts("Base::static_dispatch_method");
    }
};

class StaticDerived : public Base {
public:
    void static_dispatch_method() {
        puts("StaticDerived::static_dispatch_method");
    }
};
```

```
void call_static_dispatch(Base *obj) {
    obj->static_dispatch_method();
}


void call_static_dispatch_derived(StaticDerived *obj) {
    obj->static_dispatch_method();
}

Base base;
StaticDerived derived;
call_static_dispatch(&base);
call_static_dispatch(&derived);
call_static_dispatch_derived(&derived);
```

Output:

```
Base::static_dispatch_method
Base::static_dispatch_method
StaticDerived::static_dispatch_method
```

Virtual method example

```
#include <cstdio>

class Base {
public:
    virtual ~Base() { }

    virtual void virtual_dispatch_method() {
        puts("Base::virtual_dispatch_method");
    }

    void static_dispatch_method() {
        puts("Base::static_dispatch_method");
    }
};

class VirtualDerived : public Base {
public:
    virtual void virtual_dispatch_method() {
      puts("VirtualDerived::virtual_dispatch_method");
    }
};
```

```
void call_virtual_dispatch(Base *obj) {
    obj->virtual_dispatch_method();
}


Base base;
VirtualDerived derived;
call_virtual_dispatch(&base);
call_virtual_dispatch(&derived);
```

Output:

```
Base::virtual_dispatch_method
VirtualDerived::virtual_dispatch_method
```

Vtables

- A virtual table, or vtable, is a global structure created automatically for each class that contains virtual methods
- Vtables are used in the implementation of virtual method calls
- All instances of classes with virtual methods contain a pointer to that class's vtable as a hidden field (and in the case of multiple-inheritance, may contain multiple vtable pointers)

VTable Example

```
#include <cstdio>

class HasNoVirtualMethods {
    void* field;
};

class HasVirtualMethods {
    void* field;
    virtual ~HasVirtualMethods();
};

printf("void*: %d\n", sizeof(void*));
printf("HasNoVirtualMethods: %d\n", sizeof(HasNoVirtualMethods));
printf("HasVirtualMethods: %d\n", sizeof(HasVirtualMethods));
```

Output:

```
8
8
16
```

C++ name mangling

- In order to allow overloading functions, C++ uses to a technique called “name mangling”

- Information about the signature of the function is incorporated into the mangled name to create a unique linker-symbol for each overload

- For example, GCC mangles `MyClass::some_method(int,char)` into `__ZN7MyClass11some_methodEic`

C++ name mangling

- Not all compilers use the same name mangling scheme
  - Clang, GCC, and ICC all use the same scheme, which is part of the itanium abi
  - MSVC uses its own scheme

- Functions declared `extern "C"` do not have their linker-symbols mangled
  - This allows them to be called from C code
  - However they cannot be overloaded

c++filt

- c++filt is a command-line utility for de-mangling mangled GCC-produced C++ symbols

- Examples:

```
$ c++filt __ZN24AppleBusControllerCS84098_spiReadEhhPh
AppleBusControllerCS8409::_spiRead(unsigned char, unsigned
char, unsigned char*)
% c++filt __ZTV23com_CalDigit_UserClient
vtable for com_CalDigit_UserClient
```