C++ OO implementation

- Brief look at C++ implemention of classes/objects
- Basic implementation of fields/methods using structs and function pointers
- struct layout for inherited fields and statically-dispatched methods (the default)
- Handling of dynamic dispatch through virtual function tables

Class relationship to C structs

- Idea: store fields of a class based on C-style structs
- Methods: referred to from within struct using function pointers, actually implemented as (global) functions
- Name of the function representing a method specifies class and method, e.g. SomeClass::Amethod
- Parameter list for the function needs to know which actual struct contains the data it’s supposed to access, done by passing pointer to the struct (“this” pointer)

Hidden inclusion of “this” pointer

- Developer view vs (simplified) underlying implementation

```
// source code view
class C {
public:
     int i;
      int foo(int x) (
};
// sample call
C a;
result = a.foo(3);
```

```
// underlying struct might look like
struct C {
        int i;
        int (*foo)(C*,int);
};

// actual function might look like
int C::foo(C* this, int x)
{
        return (x + C->i);
}

// actual call might look like
result = (*(a.foo))(&a, 3);
```

Simple inheritance

- If one class inherits from another, its underlying struct contains space for all the fields from both

```
class Parent {
int x, y;
};
class Child: Parent {
int y, z;
};
```

```
// underlying child struct
// names are just for clarity,
// compiler would use offsets
{
    int Parent::x, Parent::y;
    int Child::y, Child::z;
}
```

Dynamic dispatch

- By default C++ uses static dispatch of methods, needed a syntax for developer to specify they want dynamic, done by creating an abstract base class, in which the function pointer for the relevant method is set to null,e.g.

```
virtual void foo() = 0; // within the class definition
```

- Means all “real” classes derived from this abstract base must override that method with an actual implementation
- Compiler must insert code so that, whenever foo is called, the correct “real” method can be found and called

Virtual function tables (vtable)

- Compiler creates virtual function tables (vtable) for each class that uses dynamic dispatch
- vtable contains pointers to the locally overrides for each such method, and also pointers to its ancestors vtables
- At point of call, the inserted lookup code searches the vtables to find the correct (most localized) version of the function to be invoked
- Means that actual call to such methods might have to traverse multiple vtables before identifying and calling the correct function