Can Organizing Data & Specializing Functions be Easier?

The organization of data and function specialization that you have just seen
◦ helps when developing large systems in C,
◦ but puts the burdens of learning and following best practices on the programmer.

In about the 1980s,
◦ people started thinking about automating these tasks
◦ to enforce best practices and
◦ to reduce opportunities for human error.*

*I saw it, but it was too ironic to fix.

C++ Uses Data and Function Inheritance to Automate

In particular, why not have the programmer
◦ specify only the type hierarchy and
◦ which functions should be changed for a subtype?
The compiler can then lay out the data, create the virtual function tables, and so forth.

C++ performs such automation,
◦ leveraging data and function inheritance
◦ to produce structures and functions that
◦ usually look exactly the same as one would produce in C.

Parent is the Base, Child is the Derived Type/Class

What are data and function inheritance?

Think back to our type hierarchy.
In any given parent-child relationship,
◦ the parent is called the base type or class
  (or, historically, the super-type or class), and
◦ the child is the derived type or class
  (or, historically, the sub-type or class).
Derived Types Inherit Data and Can Be Used in Functions

**Data inheritance** means that
- if a base type has a field,
- so do all types derived from that base type
- (a child inherits all fields of its parent).

**Function inheritance** means that
- if a function operates on a base type,
- the function can also operate on any type derived from that base type
- (a child inherits all functions on its parent).

Data and Function Inheritance Simply Programming

Data and function inheritance make programming big systems easier.

**How?**
- avoid replicating common code
- simplify usage of existing code
- simplify extensibility

Let’s discuss each point in more detail.

Specify Common Data and Behavior Once

**avoid replicating common code**
- when one structure is just a special kind of another structure,
- most of the data and behavior (functions) are identical.
- With inheritance, shared data and behavior are specified once.

Behavior Defaults to that Defined for Parent Class

**simplify usage of existing code**
- in our bibliography example, we
- reuse functions defined for ancestor types
- by explicitly adding them to the function table for a derived type
- Function inheritance
- makes such behavior the default:
- everything that is not overridden explicitly is the same.
Only Differences in Behavior Must be Added or Modified

**simplify extensibility**
- To create a new derived type
- one need **specify only the differences** from the base type:
  - any **new data** needed,
  - any **new behavior** needed, and
  - any **existing behaviors that must be modified**.
- Everything else is automatically inherited.

Modules and Object-Oriented Concepts 50+ Years Old

A little more background...

**1960s:**
- **software systems** become large enough
- that they are **divided into modules**
- to make reasoning about them easier.

**Late 1960s, 1970s:**
- **object-oriented languages**
- such as Simula and Smalltalk **emerge**

Information Hiding Introduced in 1972

**1972:**
- David Parnas defines **information hiding**:
  - a module defines an **interface** (functions),
  - but **how**
  - **functions are implemented** and
  - what **information** (fields, data structures) are **used**
  - **should be hidden** from other modules

C++ Evolved in the 1980s

**1979: C with Classes**
(Bjarne Stroustrup’s early name for C++)

**1980s: C++ takes off**
- blends benefits of object-oriented design
- with clarity and performance of C.

**C++ design philosophy:**
- **pay only for what you use**
- features that aren’t used incur no overhead
  (no space, no time)
- ... works well if you understand the language ...
Let's review the contents of a module.

- You've seen at least one: the memory allocator
- Then we will discuss how each piece maps into C++.

Typically,
- a module is organized
- around one or more data structures.

A **data structure** is
- a **struct** with fields (sometimes > 1)
- related **static data**
- **interface functions** that operate on one or more instances
- **internal / implementation functions**
- **initialization / teardown routines** for an instance

A module also includes initialization / teardown routines for the module.

In C++, the **smallest module is a class**.

**Classes** are
- usually defined in header files,
- as the definition includes
- specification of all interface functions.

In fact, declarations for everything related to a class must appear in the class definition.
Class Fields Look Exactly Like struct Fields

Fields are exactly the same as in a struct.
Add lines that look like variable declarations

```plaintext
int32_t   x;
double    y;
player_t* p;
```
and each instance has fields named x, y, and p.
Order in memory is order of declaration.
Fields come after fields from parent class
(as with best practice for subtypes in C).

Class Variables Prefixed with static

To declare the existence of a static variable
◦ called a class variable—one for the class, not one per instance,
◦ prefix the declaration in the class definition
  with static:

```plaintext
static int32_t   classInt;
static double    classDouble;
static player_t* classPlayer;
```
These variables reside in the global data area.

Declaration in Class Definition is Not Sufficient

```plaintext
static int32_t classInt;
```
The line above
◦ declares the existence of classInt,
◦ but does not create storage,
◦ so classInt cannot be initialized in the class definition.
Static variables must also be declared outside of the class definition.

Declare Class Variables in a Source File as Well

```plaintext
static int32_t classInt;
```
Outside of MyClass' definition, the variable above is called MyClass::classInt.
It must be declared exactly once
◦ outside of the class definition
◦ (so not in a header file)
◦ and can be initialized there.
◦ :: is a new operator.
Common to Pass a Pointer to a Struct to Functions

Think back to some of the functions that we defined for use with structures...
For stacks, we defined `stack_init`, `stack_empty`, `stack_full`, `stack_push`, and `stack_pop`.
For players, we defined `player_init`, `player_start_game`, `player_finish_game`, and `player_delete`.
All of them took a pointer to the associated struct!

Member Functions Have an Implicit Class* Argument

C++ redefines syntax to make declaration of functions associated with a class easier.
A line such as
```c
int32_t memFunc (char x, double* y);
```
declares a member function or method.
Member functions
* have an implicit first argument,
  * a pointer to an instance of the class.
With the implicit first argument, member function implementations obey the usual calling convention.

Member Functions’ Implicit Argument is Called this

In other words, if the function
```c
int32_t memFunc (char x, double* y);
```
appears in the definition `MyClass`, the actual function signature, in C syntax, is
```c
int32_t memFunc (MyClass* this, char x, double* y);
```
Notice that
* MyClass* argument is always first, and
  * the name, this, is also implicit.

Member Functions Used as If They Were Fields

Member functions are used as if they were fields. Given
```c
MyClass a;
MyClass* ptr;
```
we can call memFunc by writing
```c
a.memFunc ('z', NULL);  // this is &a
ptr->memFunc ('z', NULL);  // this is ptr
```
Class Functions Declared with static Prefix

To avoid the implicit argument, prefix the function declaration with static. Such functions are called class functions. Their function signatures
- are exactly the same as in C
- (except that in C, the static qualifier restricts access to the file;
- class function access is handled differently in C++).

Use Class:: Prefix to Invoke a Class Function

How does one invoke a class function?
As with class (static) variables,
- everything in a class
- is prefixed with the name of the class.
- For example, MyClass:::
These prefixes reduce naming conflicts and avoid the need for having programmers do it by hand (as with C).

Compiler Infers Which Class to Use for Member Functions

For member functions, a compiler infers the class from the instance type. For example,
```
    MyClass m;
    m.someFunc (...);
```
The compiler knows that
- m has type MyClass, so
- it looks first at MyClass::someFunc.

Use Class:: Prefix to Invoke a Class Function

The compiler does not guess which class holds a class function. If MyClass contains ...
```
    static void doSomething (int32_t q);
```
use the namespace operator (::) to invoke the class function:
```
    MyClass::doSomething (42);
```
C++ supports hierarchical namespaces with ::, but for now you need merely understand the use with classes.
Definitions Outside Class Definition Also Require Prefix

Both member and class function definitions are usually specified outside of the class definition. Again, the compiler must be given the context, so...

\[
\text{int32_t MyClass::memFunc (char x, double* y)} \{
\}
\]

*Short/simple functions can be written into the class definition, in which case the code is often inlined into calling code.

Within Functions, Class:: and this-> are Often Inferred

Within a member or class function definition, symbols from the class can be used without the namespace prefix. Fields and member functions used in a member or class function implicitly add the prefix “this->”

\[
\text{field++; // means this->field++;}
\text{memFunc ();}
\text{// means this->memFunc ();}
\]

Class Function Definitions Do Not Include “static”

For class function definitions, do not write “static”. If a class definition includes static void doSomething (int32_t q);
outside of the class definition, the class function is defined as follows:

\[
\text{void MyClass::doSomething (int32_t q) \{}
\]

Class Functions Have No this Argument

Class functions have no this argument, so...

\[
\text{void MyClass::doSomething (int32_t q) \{}
\text{field++; // invalid}
\text{method (); // not allowed}
\text{classFunc (q - 1); // ok -- MyClass:: is inferred}
\]

Data and Function Inheritance Implications for C++

Now back to inheritance. Given

```cpp
class MyClass : public ParentClass {
    // class definition
}
```
data inheritance implies that a `MyClass` has all fields of a `ParentClass`, and
function inheritance implies that we can invoke any member function for `ParentClass` on a `MyClass`.

Compilers Implicitly Convert Derived* to Base*

What if both `MyClass` and `ParentClass` define member function `aFunc`?

Let’s say that we have three variables

```cpp
MyClass m;
ParentClass p;
ParentClass* ptr = &m;  // Huh?
```
Compilers can implicitly convert pointers to derived classes to pointers to base classes.
Remember that in many cases, no instructions are needed for such conversion!

Compilers Use the Type of the Object to Choose a Class

Now back to the question.

```cpp
MyClass m;
ParentClass p;
ParentClass* ptr = &m;
```

```cpp
m.aFunc (); // calls MyClass::aFunc
p.aFunc (); // calls ParentClass::aFunc
ptr->aFunc (); // ParentClass::aFunc
```

But `*ptr` is actually a `MyClass` instance!

Use `virtual` to Create Subtype-Specific Functions

If you want the behavior that we implemented in C using tables of function pointers, mark `aFunc` as `virtual` in `ParentClass`.

```cpp
virtual void aFunc (void);
```

C++ compilers automatically

* define virtual function pointer tables,
* fill them for each class, and
* place a pointer to the table into each class instance (usually at the start of the structures).

(`virtual` is the default in Java, by the way.)
Pitfall: Marking Functions Virtual Only in Child Classes

The `virtual` qualifier is **inherited** implicitly,
- so adding it to `aFunc` in `ParentClass` implies its use for all child classes, such as `MyClass`.
- Add it to `MyClass` anyway for clarity.

The `virtual` qualifier is **not back-propagated** to parent classes:
- if you only add it to `MyClass`,
- the call on the previous slide will STILL go to `ParentClass::aFunc`.

© 2018 Steven S. Lumetta. All rights reserved.