CORE FEATURES OF C++

- **How typed C++ is, and why it matters**
- **The Interface Principle in C++**
- **The *real* difference between struct and class**
- **To RAII or Not to RAII?**

Fluent {C++}

How typed C++ is, and why it matters

For the third time I attended the meetup of Software Craftmanship in Paris. And like the first time and the second one, it was superb experience again.

This meetup lets the participants collectively choose the topics to be discussed. There I proposed to confront, as a group, the various languages that the people in the room knew (the crew wasn't especially C++ oriented), in order to point out some of the differences between the languages. The point was to discover new approaches taken by other languages, and by comparison, to better understand our own.

This experience allowed me to understand C++ a bit better, and for this reason I want to share with you what was exchanged during the event.

The biggest part of the debate was oriented on the theme of **typing**. And in C++ we use typing as a powerful tool to make code more expressive, like with strong types.

But while some common languages like C++ and Java are classified as typed, some others – just as common – aren't, like JavaScript or Python. And some languages – like Haskell or Idris – are even more typed than C++. More about this in just a moment.

In general, people coding with strongly typed languages can't imagine how the others can live a decent life, and vice versa. And I think this is a wrong-headed approach: there are lots of people coding with both categories of languages, so there are necessarily at least pros (and probably cons) in both.

And I believe that understanding where your language lies in the spectrum of typing strength provides you with an enriching perspective.

So let's get into it, like if you were there!

We will start by the advantages of typing.

Typing brings protection

The first advantage of typing is profoundly anchored in the mindset of C++ programmers: typing prevents programmers from making a whole class of mistakes. And these mistakes essentially consist in **passing the wrong object to the wrong context**.

A type is embodied in C++ by a `struct`, a `class` or a primitive type (`int`, `double`, `char`, etc.). It represents a concept after which all objects of this type are modeled. For example, `int` models an integral number, `Employee` represents someone from the personnel, and `Date` represents a moment in time.

By stating what type is expected in a certain context – like the arguments of a function – the compiler ensures that the type of the object passed to this context is the one expected. Some languages try to go further in this direction, by using the type system to prevent unexpected states to occur. The video Making impossible states impossible about the language Elm was proposed to illustrate this point.

For this reason, strong typing in a language typically **reduce the amount of necessary tests**. Since some things cannot happen by construction of the language, there is no need to confirm it with tests. In a typed language like C++, you wouldn't write a test that passes an int to a function that expects an `std::string`, to make sure it behaves ok, because such code would be downright rejected by the compiler.

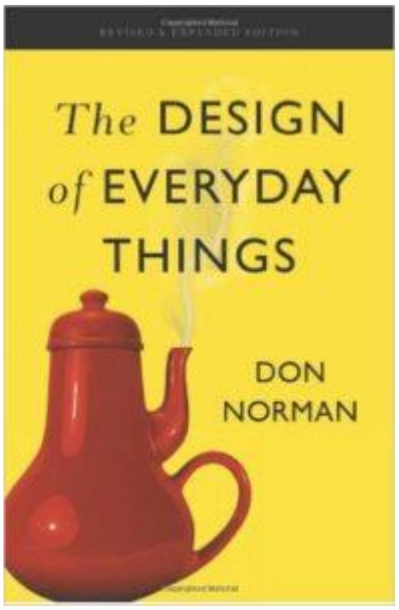
Typing brings expressiveness

Someone rose an interesting question during the debate: **do types serve the compiler**?

And the answer that was given was: **No, typing helps human developers** seeing more clearly in their programs. Sure enough, the compiler is aware of types and make checks based on them, but these are merely to prevent humans from being overwhelmed by their code growing in complexity. When you think about it, once a source file is compiled down to machine code, there are nothing but 0s and 1s remaining. The types are then long gone.

In fact, typing can act as a form of documentation. For instance, if you consider a function taking an object of type `FirstName` and an object of type `LastName`, and returning an `Age`, you immediately get a very strong intuition that this function somehow retrieves the age of a person. And one can understand this without any other information than the types involved in the prototype.

Some languages go further than this, by defining prototypes that can involve **any type**. For example, consider a function that can take an object of any type A as an argument, and return an object of type A. At first glance there must be a great number of functions that could fit this prototype. But it turns out that there is only one: the identity function. This is a surprising way of how types can provide information in a prototype. More about this and how this can be useful in C++ in the excellent talk Using Types Effectively given by Ben Deane at CppCon.



An interesting parallel to typing was brought up with how real life objects look like. Indeed, Their appearance provide information about what they are and how they are meant to be used. Types in code are a bit like that too, because they are inherently part of the objects they model, and they convey meaning and useful information about them.

To go deeper into this way of thinking, the following book was suggested: **The Design Of Everyday Things**. It's not the first that time this book comes up in a developers discussion, and it is definitely in my reading list now. When I've read it, you can count on me to make a dedicated post to share its relevant insights to make code more expressive, if there are any.

Isn't there anything wrong with typing?

So far only positive aspects of typing have been brought up. But aren't there any downside coming with it? Even though I'm a hardened C++ programmer, I refuse to believe that typing is all good. I mean, there are so many people and so much code out there that use weakly typed languages that there must be interesting aspects in doing so. So I asked the question: what do you find annoying with typing? Here is what came out of the discussion.

The first interest of weakly typed language is that they provide a **sensation of freedom**, by not being constrained by the type system. This puts typed languages in perspective as more unpleasant and complex.

Developing in weakly typed languages go **faster**. Quite a lot faster. I didn't quite catch a concrete illustration of this point though.

With typed languages, you're brought to write **casts** at some point or other, even in cases where you precisely know what types you're expecting in the code. Someone brought up an example involving system calls to illustrate this point.

Finally, it can be useful to be able to perform operations on objects that offer certain functionalities, **regardless of their type**. This is called **duck-typing** (because if a something runs like a duck, flies like a duck and makes "quacks" like a duck, we may as well consider it a duck and use it like so). This correspond exactly to C++ **implicit interfaces**, which is the current state for templates.

To Type or Not To Type

So there are upsides and downsides to typing. Is there a way to decide when typing is more appropriate, and when it isn't?

A interesting point of view was suggested. Typing brings protection and clarity, but with some hassle along the way. So this is appropriate for **complex applications**, like large software. For smaller tasks, like scripts for examples, typing isn't worth the hassle and a weakly typed language will do. A compelling illustration of this is that **scripting languages tend to be very weakly typed**.

This idea really resonated with me, but was finally argued against, because it may involve a sampling bias. Indeed if you've worked on big projects involving C++ it's easy to conceive that typing is necessary. But when you think about it, there are also many very big projects that are written in weekly typed languages, and that are still successful! Facebook is written in PHP for example. So I'm not too sure what to think of this.

C++ in the midst of all this

The following has been explained to me by the one of organizers of the Software Craftmanship meetup himself, Xavier Detant, who has a very rich knowledge about software development.

The type system is so important and useful in C++ that we may think that C++ is a really strongly typed language. But it turns out that there are languages that make even more use of their type system. Here are two:

- **Haskell**: Haskell is arguably more strongly typed that C++, because types include richer information. This is well illsutrated by Algebraic Data Types, that consist of describing a type with several different forms. For example, a `Tree` can be either a `Node` or a `Tree` bearing a value and two `Tree`s as children. And functions taking a `Tree` perform pattern matching on the two forms of the type. More about how this can be applied to C++ in the talk of Ben Deane cited above.
- **Idris**: Idris is a very strongly typed language, even more so than Haskell. It share a lot of commonalities with it in its syntax, but the possibilities of the type system are even wider. Idris use **dependent types**, which means that the choice of a type, like the return type of a function, can be determined by a *value*. You can look at the doc here if you're curious about it. I must admit that I haven't completely wrapped my head around this (for the moment its rather this concept that has wrapped itself around my head), but I think there must be some interesting takeaways from this way of programming. When I get to it, you'll certainly have a new post about how to use them to write expressive... C++!

The Interface Principle in C++

The Interface Principle in C++ encompasses a specific combination of features and ways of considering what an interface is, that allows to write expressive C++ code that preserves encapsulation. It has been around for a while, is still currently used, and may be enriched in the future versions of the language. So it's worth being aware of.

Note that the Interface Principle goes beyond the general concept of having interfaces, and is not directly related to polymorphism.

The convention we will use throughout this article is this:

- a **method** designates a routine that is a member of a class,
- a **(free) function** is a routine that is not part of a class.

Non-member (non-friend) functions

In Item 23 of Effective C++, Scott Meyers encourages us to pull methods of a given class *outside* of the class, whenever it is possible to implement them in terms of the public interface of the class (or with other methods that have been taken out of the class).

To illustrate this let's consider the `Circle` class that provides its radius, area and perimeter:

```
class Circle
{
public:
    explicit Circle(double radius) : m_radius(radius) {}

    double getRadius() const {return m_radius;}
    double getPerimeter() const {return 2 * Pi * m_radius;}
    double getArea() const {return Pi * m_radius * m_radius;}

private:
    double m_radius;
};
```

A first improvement would be to use the public interface inside the implementation of the methods:

```
double getPerimeter() const {return 2 * Pi * getRadius();}
    double getArea() const {return Pi * getRadius() * getRadius();}
```

And then these methods can be taken out of the class. Indeed they don't need to be class methods, because they don't use anything a external function couldn't use. Taking them out of the class and making them free functions guarantees that this characteristic of not using anything else than the public interface will be preserved, and therefore contributes to the encapsulation of the insides of the `Circle` class.

```
class Circle
{
public:
    explicit Circle(double radius) : m_radius(radius) {}

    double getRadius() const {return m_radius;}

private:
    double m_radius;
};

double getPerimeter(Circle const& circle) {return 2 * Pi *
circle.getRadius();}
double getArea(Circle const& circle) {return Pi * circle.getRadius()
* circle.getRadius();}
```

Another way to see this is that this decreased the amount of code that could be impacted by a change in the implementation of the class `Circle`, therefore making the code a bit more robust to future change.

If you want a way to reproduce this consistently, here is the methodology we applied:

- check that the implementation of a given methods only depends on the public interface (or make it so if it's not too much hassle),
- create a free function with the **same name** as the method,
- add the type of the class as **first parameter**:
    - pass it by reference if the methods was not const
    - pass it by reference-to-const if the method was const

- paste the implementation, adding the object name before each call to the class public interface.

It is important to note that the new free function should have the **same name** as the old method. Sometimes we are reluctant to call a free function `getPerimeter`. We would be more inclined to call it something like `getCirclePerimeter`. Indeed, since it is not enclosed in the `Circle` class, we may feel it is ambiguous to omit the term “Circle”. But this is wrong: the term “Circle” already appears in the type of the first argument. Therefore it is reasonably expressive both for a human and a compiler to omit the type name in the function name.

Actually, including the argument type in the function name would even lead to code looking slightly weird:

```
getCirclePerimeter(circle); // "Circle" mentioned twice
```

as opposed to:

```
getPerimeter(circle);
```

which reads more naturally. Also, the fact that the argument type is a `Circle` makes it unambiguous for the compiler that this is the function you mean to call, even if there are other overloads sharing the name `getPerimeter`.

The Interface Principle

The new version of the class `Circle` has something that may seem disturbing: it has functionality declared outside of its interface. That was the purpose of making methods non-members in the first place, but normally a class should expose its responsibilities within its “public:” section, right?

True, a class should expose its responsibilities in its *interface*. But an interface can be defined by something more general than just the public section of a class declaration. This is what the **Interface Principle** does. It is explained in great details in Herb Sutter’s Exceptional C++ from Item 31 to 34, but its definition is essentially this:

A free function is part of a class interface if:

- it takes an object of the class type as a parameter,
- it is in the **same namespace** as the class,
- it is shipped with the class, meaning that it is declared in the **same header** as the class.

This is the case for the `getPerimeter` and `getArea` functions (here they are in a global namespace, but the next section adds namespaces to precisely see how this interacts with the Interface Principle). Therefore if you declare a function taking an object of the class type as a parameter, declared in the same namespace and header as a class, then your are expressing that this function is conceptually part of the class interface.

As a result, the only difference between a function and a method of the class interface becomes its invocation syntax:

```
getPerimeter(circle);
```

for the function, versus

```
circle.getPerimeter();
```

for the method. But beyond this difference, the Interface Principle implies that these two syntaxes express the same thing: invoking the `getPerimeter` routine from the `Circle` interface.

This lets us take code away from the class to improve encapsulation, while still preserving the semantics of the method.

The ADL: the Interface Principle playing nice with namespaces

With solely the above definition of the Interface Principle, there would be an issue with namespaces: calling non-member functions would have a burden over calling methods, because it would need to add the namespace to the invocation.

To illustrate, let's put the interface of `Circle` in a namespace, `geometry`:

```
namespace geometry
{

class Circle
{
public:
    explicit Circle(double radius) : m_radius(radius) {}

    double getRadius() const {return m_radius;}

private:
    double m_radius;
};

double getPerimeter(Circle const& circle) {return 2 * Pi *
circle.getRadius();}
double getArea(Circle const& circle) {return Pi * m_radius *
circle.getRadius();}

} // end of namespace geometry
```

Then calling the function provided in the interface could be done the following way:

```
geometry::getArea(circle);
```

Compare this to the call to method:

```
circle.getArea();
```

This discrepancy is a problem, because the Interface Principle wants the method and the free function to be considered as semantically equivalent. Therefore you shouldn't have to provide any additional information when calling the free function form. And the problem gets bigger in the case of nested namespaces.

This is solved by the Argument Dependent Lookup (ADL), also called Koenig lookup.

The ADL is a native C++ feature that brings **all functions declared in the namespaces of the arguments types of the call** to the scope of the functions searched for resolving the call. In the above example, `circle` being an object of the type `Circle` in the namespace `geometry`, all free functions in this namespace are considered for resolving the function call. And this includes `getArea`. So you can write the following code:

```
getArea(circle);
```

which therefore expresses just as much as what a human and a compiler need to understand what you mean.

Generic code

On the top of encapsulation, free functions let you do more flexible things than methods in cases of generic code.

We saw in the first section of this article that it was preferable to avoid adding the argument type in the function name, for code clarity. But having general names also makes it easier to create generic code. Imagine you had a class `Rectangle` over which you can calculate a perimeter too:

```
double getPerimeter(Rectangle const& rectangle);
```

Then the `getPerimeter` function can be used in generic code more easily than if it contained superfluous information about argument types in its name:

```
template <typename Shape>
void operateOnShape(Shape const& shape)
{
    double perimeter = getPerimeter(shape);
    ....
}
```

Consider how much harder this would be to write such code with functions like `getCirclePerimeter` and `getRectanglePerimeter`.

Also, there are types on which you cannot add methods, because they are native C++ types for example, or because it is code that for some reason you don't have the possibility to change. Then you can define free functions that accept these types as argument.

An example can be found in the STL with the *functions* (not methods) std::begin and std::end. These functions call the begin and end methods of their container arguments, and have a specific implementation for arrays (T[]), because arrays don't have begin and end methods. This lets you write generic code that can accept both containers and arrays indifferently.

A uniform function call syntax in C++?

The language already has features that facilitate benefiting from the Interface Principle. The ADL is one of them. And there seems to be a trend with new or future features to go in that direction.

`std::invoke` allows to have exactly the same syntax for calling a function or a method. The following syntax:

```
std::invoke(f, x, x1, ..., xn);
```

- calls `f(x, x1, ..., xn)` if f is not a class method,
- calls `x.f(x1, ..., xn)` if f is a class method.

`std::invoke` becomes available in C++17.

Finally, there have been discussions around the proposal to implement this equivalence natively in the language, so that

```
f(x, x1, ..., xn);
```

calls `x.f(x1, ..., xn)` if f is not a function but a method, and

```
x.f(x1, ..., xn);
```

calls `f(x, x1, ..., xn)` if f is a not method but a free function. This is called the Unified Call Syntax, here is a description of it by Bjarne Stroustrup and Herb Sutter.

I don’t know if this particular proposal will make it to the standard one day, but one thing is sure: the language has been evolving and continues to evolve in that direction. Keeping this in mind when designing code makes it more natural, more robust, and more expressive.

Related articles:

- How to choose good names for your code

The *real* difference between struct and class

“Should I use a `struct` or a `class`?”

Such is the question many C++ programmers ask themselves, or ask around to more experienced co-workers, when designing their code.

There is sometimes a cloud of misconception about what the difference between `struct` and `class` technically is, particularly amongst the youngest developers. And once we get to understand the technical difference, some degree of uncertainty often remains about which one to use in a given context. Sometimes developers even disagree about which is the more appropriate in their code.

Let’s start by clearing up the situation, by stating the technical difference between `struct` and `class`, and then propose rules to choose between the two, by looking at what the C++ Core Guidelines written by the Jedis of C++ have to say about it.

The legal difference

In terms of language, except one little detail, there is no difference between `struct` and `class`. Contrary to what younger developers, or people coming from C believe at first, a `struct` can have constructors, methods (even virtual ones), public, private and protected members, use inheritance, be templated… **just like a `class`**.

The only difference is if you don’t specify the visibility (public, private or protected) of the members, they will be public in the `struct` and private in the `class`. And the visibility by default goes just a little further than members: for inheritance if you don’t specify anything then the `struct` will inherit publicly from its base class:

```
struct T : Base // same thing as "struct T : public Base"
{
    ...
};
```

while the `class` will do private inheritance:

```
class T : Base // same thing as "class T : private Base"
{
    ...
};
```

That's it. No other difference.

Once we get past this language precision, the following question arises: if `struct` and `class` are so similar, when should I use one or the other?

The *real* difference between struct and class: what you express by using them

The difference that really matters between `struct` and `class` boils down to one thing: **convention**. There are some conventions out there that are fairly widespread and that follow a certain logic. Following these conventions gives you a way to express your intentions in code when designing a type, because as we'll see in a moment, implementing it as a `struct` doesn't convey the same message as implementing it as a `class`.

struct

In a word, a `struct` is a **bundle**. A `struct` is several related elements that needed to be tied up together in a certain context. Such a context can be passing a restricted number of arguments to a function:

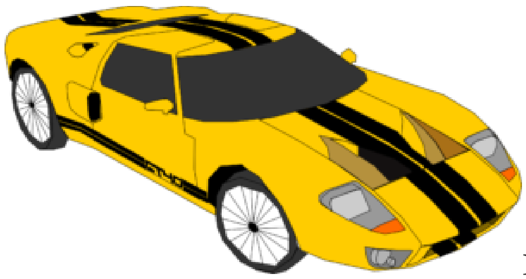
```
struct Point
{
   double x;
   double y;
};

void distance(Point p1, Point p2);
```

Although it's a bundle, `struct` can be used to effectively raise the level of abstraction in order to improve the code: in the above example, the `distance` function expects Points rather than doubles. And on the top of this, the `struct` also has the benefit of logically grouping them together.

Another context is returning several values from a function. Before C++17 and structured bindings, returning a `struct` containing those values is the most explicit solution. Have a look at Making your functions functional for more about making function interfaces clearer.

class


In two words, a class can **do things**. A class has responsibilities. These responsibilities can be quite simple, like retrieving data that the class may even contain itself. For this reason, you want to use the term `class` when you are modelling a concept (that has an existence in the business domain or not), the concept of a **object** that can perform actions.

Contrary to a `struct`, a class is made to offer an **interface**, that has some degree of separation from its implementation. A `class` is not just there to store data. In fact a user of a class is not supposed to know what data the class is storing, or if it contains any data at all for that matter. All he cares about is its responsibilities, expressed via its interface.

A `class` raise the level of abstraction between interface and implementation even more than a `struct` does.

Sometimes a type that was initially implemented as a `struct` ends up turning into a `class`. This happens when you realize the various bits that were bundled turn out to form a higher level concept when they are considered together, or have a stronger relation than what was perceived initially.

This is where **invariants** come into play. An invariant is a relation between the data members of a class that must hold true for the methods to work correctly. For example, a `std::string` can hold a `char*` and a `size` in its implementation (well at least conceptually, since modern string implementations are more complex than that due to optimizations). Then an invariant is that the number of characters in the allocated `char` buffer must match the value in the `size` member. Another invariant is that the `char*` is initialized and points to valid memory.

Invariants are set into place by the constructor of the `class` and the methods make the assumption that all the invariants hold true when they are called, and ensure they remain true when they finish. This can be a tacit agreement, or, as has been discussed for standardization, such pre-conditions and post-conditions in methods could one day be explicitly stated in code, and checked at run-time.

Finally a simple rule of thumb for choosing between `struct` or `class` is to go for `class` whenever there is **at least one private member** in the structure. Indeed, this suggests that there are implementation details that are to be hidden by an interface, which is the purpose of a class.

The C++ Core Guidelines

The above has been inspired by the C++ Core Guideline (which is a great read by the way), in particular the following:

C.1: Organize related data into structures (`struct`s or `class`es)
C.2: Use `class` if the class has an invariant; use `struct` if the data members can vary independently
C.3: Represent the distinction between an interface and an implementation using a class
C.8: Use `class` rather than `struct` if any member is non-public

To RAII or Not to RAII?

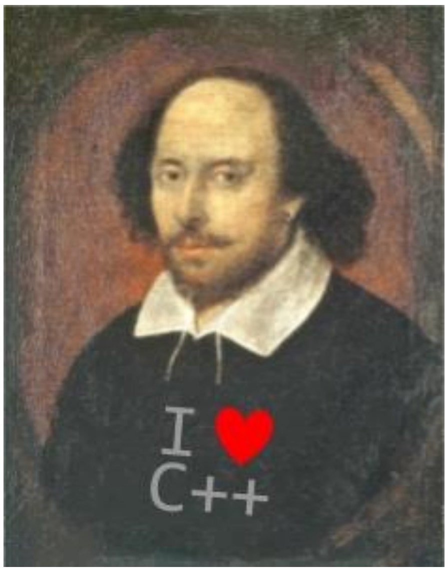
RAII is a central concept in C++, that consists in relying on the compiler to call destructors automatically in certain cases. Putting appropriate code in such destructors then relieves us from calling that code – the compiler does it for us.

RAII is an idiomatic technique of C++, but can we use RAII for everything? Is it a good idea to shift every possible piece of code to the destructor of some class, to leave the work to the compiler and make calling code as light as can be?

Since this question comes down to asking if the proverbial hammer is a tool fit for every single task, the answer to that question is probably the proverbial No.

But then, in which cases would RAII improve the design of a piece of code?

In this article we'll see a case where RAII is adapted, then a case where RAII is NOT adapted. And after that we'll see a case open to discussion. We'll then conclude with how to use levels of abstractions to make the decision to RAII or not to RAII.



"To RAII or not to RAII, that is the question" – *Shakespeare at the London C++ meetup*

A typical case for RAII : smart pointers

Smart pointers are classes that contain a pointer and take care of deleting them when going out of scope. If this sentence doesn't make sense, you can look at this refresher on smart pointers, where we get into more details about the stack, the heap, and the principle of RAII illustrated with smart pointers.

Smart pointers are considered an improvement over raw pointers (the "smart" qualification says something about them). Pointers allow dynamic allocation useful for polymorphism, but are difficult to deal with, particularly with their life cycle. Indeed, if we forget to call `delete` on a pointer it causes a memory leak, and if we call `delete` more than once we get undefined behaviour, typically a crash of the application.

Moreover, some functions can return earlier than the end of their implementation because of an early return or an uncaught exception. In those cases it is tricky to make sure we call `delete` correctly.

Smart pointers relieve us from those troubles (Hey, people from other languages, C++ is getting simpler!), and they do it by using RAII. Indeed, when a smart pointer is instantiated manually on the stack, or returned from a function, or contained in a object, the compiler automatically calls its destructor which in turns calls `delete` on the raw pointer. Even in the case of function with an early return or uncaught exception.

(Well, there are various smart pointers, and some of them like `std::shared_ptr` have a more elaborate way to deal with memory, but that's essentially the idea.)

So in the case of smart pointer, using RAII is considered to be a good thing.

A distortion of RAII

EDIT: this section has gone through some changes since the original version of the article, thanks to Andrew Haining and Daryn's inputs. I'm grateful to them for this.

Just for the sake of illustrating a case where putting a destructor in charge of some logic is not adapted, let's consider the following slightly contrived example.

We have a `Persistor` class in charge of saving some data in a database. It receives this data through its constructor. Now let's suppose we use something that looks like RAII to trigger the saving procedure of the data, so we put everything related to saving in its destructor:

```
class Persistor
{
public:
    explicit Persistor(Data const& data);
    ~Persistor()
    {
      // open database connection
      // save data_ in database
      // close database connection
    }
private:
    Data data_;
};
```

In this case, a calling code could look like this:

```
void computeAndSaveData()
{
    Data data = // code that
                // computes the
                // data to be saved

    Persistor myPersistor(data); // we just create a Persistor

} // myPersistor's destructor is called - the data gets saved
```

This code has the problem that it would trigger a question in the mind of its reader: why isn't this variable used? To this we could answer why else a persistor would be there unless to save data? But still, the code would be clearer if it just mentioned that it did a saving operation.

Let's move the code saving the data from the destructor over to a new `save` method. The calling code is now:

```
void computeAndSaveData()
```

```
{
    Data data = // code that
                // computes the
                // data to be saved

    Persistor myPersistor(data);
    myPersistor.save();
}
```

Which is clearer.

However, it would make sense to leave *some* code to RAII in the destructor: the closing of the database connection for instance. So we'd be using RAII for this, and that would be somewhat similar to smart pointers: we would dispose of a **resource** in the class destructor.

Here is how the code would look like:

```
class Persistor
{
public:
  explicit Persistor(Data const& data)
  {
    connection_ = ...; /* open database connection */
  }
  ~Persistor()
  {
    /* close database connection */
  }
  save(Data data)
  {
    /* save data in database */
  }
private:
  DatabaseConnection connection_;
};
```

At this point it is worthy to note that the 'R' in RAII stands for **Resource** (if you were wondering, the other letters mean "Acquisition Is Inialization". But I don't think it matters qs much).

Is this to say that RAII is only useful for making sure we dispose of a resource correctly, and for nothing else?

Let's see one last example to check that.

A case to discuss: a contextual logger

The case

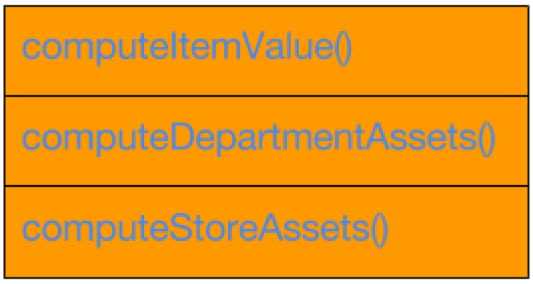
We have a program that perform many calculations, and we want to log some of these computed values to an output file. Every logged information should be made of two parts:

- the value that the program is computing,
- the context within which this value is computed.

This looks like a project I've worked on but I'm stripping off every domain aspect here, so let's consider an imaginary context of retail.

We have a supermarket that contains departments, that themselves contain items. To compute the total value of the assets owned by a store, we sum all the values of the items contained in each department.

The call stack looks like this one:



Call stack of the valuation

Now here is an excerpt of the desired output log:

```
Store = Madison Av > Dpt = Laptops > Item #42 | Item value = 1000
Store = Madison Av > Dpt = Laptops > Item #43 | Item value = 500
Store = Madison Av > Dpt = Laptops | Item value = 1500
Store = Madison Av > Dpt = Webcams > Item #136 | Item value = 12
```

Each message starts with a context that can have various depths corresponding the levels of the call stack, and ends with a message about a local calculation.

The first two lines and the 4th one are emitted from the `computeItemValue` function. They output the value of the current item being considered. The third line is emitted from the `computeDepartmentAssets` function, that adds up the values of the items in a department.

How can we implement such a logger?

A solution using RAII

One way to go about that is to maintain a context stack in the logger, to which we push context information (e.g. `Dpt = Laptops`) when the execution enters a given context, and pop it off when it gets out of that context. We can then pile up deeper inner contexts (`Item #42`) before going out of an outer context.

To model this, let's create a `Logger` class to which we can push or pop additional context. `Logger` also has a `log` method that takes a message (the second part of the line) and sends a line constituted of the current context and the message, to the output file:

```
class Logger
{
public:
    pushContext(std::string const& context);
    popContext();

    sendMessage(std::string const& message);
};
```

To push or pop a context, we can use RAII through a helper class `Context`. This class accepts an incremental context and pushes to the `Logger` in its constructor, and pops it off in its destructor:

```
class LoggingContext
{
public:
    LoggingContext(Logger& logger, std::string const& context)
    : logger_(logger)
    {
        logger_.pushContext(context);
    }
    ~LoggingContext()
    {
        logger_.popContext();
    }
private:
    Logger& logger_;
};
```

We can instantiate this class at the beginning of the function, and allowing to maintain the correct context.

Here is how the call computing the value of an item would perform it logging:

```
double computeItemValue(Item const& item)
{
    LoggingContext loggingContext(logger, "Item #" +
std::to_string(item.getId()));

    // code that computes the value of an item...
    logger.sendMessage("Item value = " + std::to_string(value));
    // return that value
}
```

And at department level:

```
double computeDepartmentAssets(Department const& department)
{
    LoggingContext loggingContext(logger, "Dpt = " +
department.getName());

    // code that calls 'computeItemValue'
    // and adds up the values of each item
    logger.sendMessage("Dpt value = " + std::to_string(sum));
    // return that sum
}
```

And we'd have something similar at store level.

The variable `loggingContext` is instantiated, but not used directly in the function. Its purpose it to push an additional context information to the logger at the beginning of the function, and to pop it when its destructor is called when the function ends.

We use RAII here to pop off the context without having to write code for it, but there is **no resource** handled here. Is this good design?

Let’s see the advantages and drawbacks of this technique:

Advantages:

- The context is popped off the logger no matter how the function ends (normal ending, early return, uncaught exception)
- A declarative approach: the calling code merely states that it is about a given context, and doesn’t have to manipulate the logger.
- This has a side effect to document the code for readers too, to say what a piece of code is about (we’ve used it for a whole function, but this technique can also be used in a block inside a function (delimited by braces `{}`))

Drawbacks:

- An unused variable can be surprising.

What is your opinion on this?

There is one important aspect here: some code (manipulating the logger) has been hidden from the calling context. Is it a good thing or a bad thing? It comes down to…

Levels of abstraction

The code that computes the assets of a department in a store, for example, shouldn’t be too much concerned with logging. Its main job is to perform calculations, and the fact that it sends them to a log is incidental. And how exactly the logger works, with its contexts stacking up and everything, is not part of the abstraction of a function that performs calculations.

Here RAII encapsulates this lower level of abstraction of **how** we do logging and lets the function express **what** it is about, by stating its own context. RAII helped us respecting levels of abstraction here.

Let’s try to see the previous examples, the smart pointer and the database saving, with the perspective of levels of abstraction.

Smart pointers use RAII to hide the manipulation of pointers, which are a lower level than business code, so RAII helps respect levels of abstraction in that case too. This is true for resource management in general, including database connection. We just want resources to be managed correctly, and not to pollute our higher-level business code.

The code that saves the data in the `Persistor` example is at the level of abstraction of the code that instantiates the persistor. Indeed, the role of the calling function was to save, and RAII got in the way by taking this code away to a destructor, so it wasn't adapted to that situation.

Two aspects of RAII

In conclusion, we've seen two aspects to consider to decide whether or not to use RAII in a given situation:

- is there code we want to be called no matter how a function ends, be there normal ending, early return or uncaught exceptions (which is the case for releasing a resource and popping off the logger)?
- does it help respect levels of abstraction?

If one of those two questions answers Yes, RAII is an option you want to give a chance to.