The ++ of C++

Jan van Dijk

February 12, 2004

Contents

Abstract

In this text we shall discuss the concept of classes and operator overloading in C++. Starting from a C implementation of code which creates and manages variables which represent the mass and charge of a species, step by step we will show the advantages of an equivalent implementation in C++ which uses *classes*. We will then show how C++ allows the programmer to give a meaning to operators like '+', '-' et cetra for his user-defined types.

1 Introduction

In this text we will try to give you a feeling for the advantages of using C++ classes in a computational plasma physics situation. The concept we will be

```
int main()
{
        double M1, M2;
        double Q1, Q2;
        M1=1.7e-27;
        Q1=0;
        M2=3.4e-27;
        Q1=1.6e-19;             // BUG! Q2 was meant!
        // ...
        if (Q1=0) { ... }       // BUG! Always false!
}
```

Figure 1: species0.c: A pathological C program in which the mass and charge of two particles are stored in variables of type double. Please note the two common mistakes.

dealing with is that of a *particle type*. In our contrived sample code a particle will be assumed to have merely two properties: a mass and a charge.

In the next section we will show how classes can be seen as *better structures*. Classes have much in common with C-style structures, but will be shown to have a number of clear advantages.

After the class concept has been introduced, the discussion will move to the second (and last) subject of this lecture. It will be shown that in C++ the user-defined types can be made to behave just like built-in types (like ***int***, ***double*** *et cetera*) . More specifically, we will show that species can be added and multiplied, just like the numbers 1 and 1.

2 The class concept

In a number of steps we will transform an extremely silly program in C into a beatiful small program in C++ which uses *classes*. In the course of the discussion it will be made clear how C++ concepts can be used to better express the programmer's intent. In fact, a number of typical programming errors can be prevented by cleverly controlling rights to access data.

2.1 species0.c

Let's write some code in which particles are involved. For the moment, particles have two properties: a mass and a charge. For these propoerties we may introduce variables of the C(++) built-in type ***double***.

If we are dealing with two particles we may call these variables $M1$, $M2$, $Q1$ and $Q2$. The creation of these variables, their initialisation and their usage may look as in the (pathological) C program *species0 . c* in figure 1.

The program may look contrived, but clearly demonstrates two (common) mistakes. Firstly, one of the variables is initialised twice, the other not at all. This is a common copy-paste error.

The second error is the assignment to *Q1* in the last line. But is it a bug? Probably it is, and a *comparison* of *Q1* with 0 was meant. On the other hand, this is perfectly valid C: an assignment is an expression, the result is the value which was assigned, 0. As a result the code is equivalent to ***if*** *(0)*{}, which is never going to happen. Worse, because this is valid C(++) the best you can get is a compiler warning.

In the following sections we will slowly transform this small program in a C++ program in which *classes* are used. In the course of the discussion we will see a number of ways in which the language (C++) provides mechanisms to prevent the silliest of errors, like the ones above.

2.2 species1.c

Let us first look at the reasons for the erroneous double assignment. Such errors are more likely to occur in situations where a lot of variables are into play. That reduces the *readability* of the code. C provides some support for structuring the code to improve its readibility, in the form of *structures*.

Let us look at a rewritten version of the previous program, *species1 . c*, in figure 1. Here a structure *Species* has been introduced which represents the properties of a particle type. The initialisation is done with a dedicated *function*, *init_species* . The data members have *m_* prepended, this is merely a convention.

With these definitions we have pushed some of the 'complexity' of the code outside the main program in which the functionality is needed. The code for the structure and the function can be tested and reused, the main function will be simpler and, as a result, less error-prone.

Unfortunately, there are still two errors in this program. Although the likelyhood of these errors is smaller in view of the improved readability, the language cannot prevent such errors from being made.

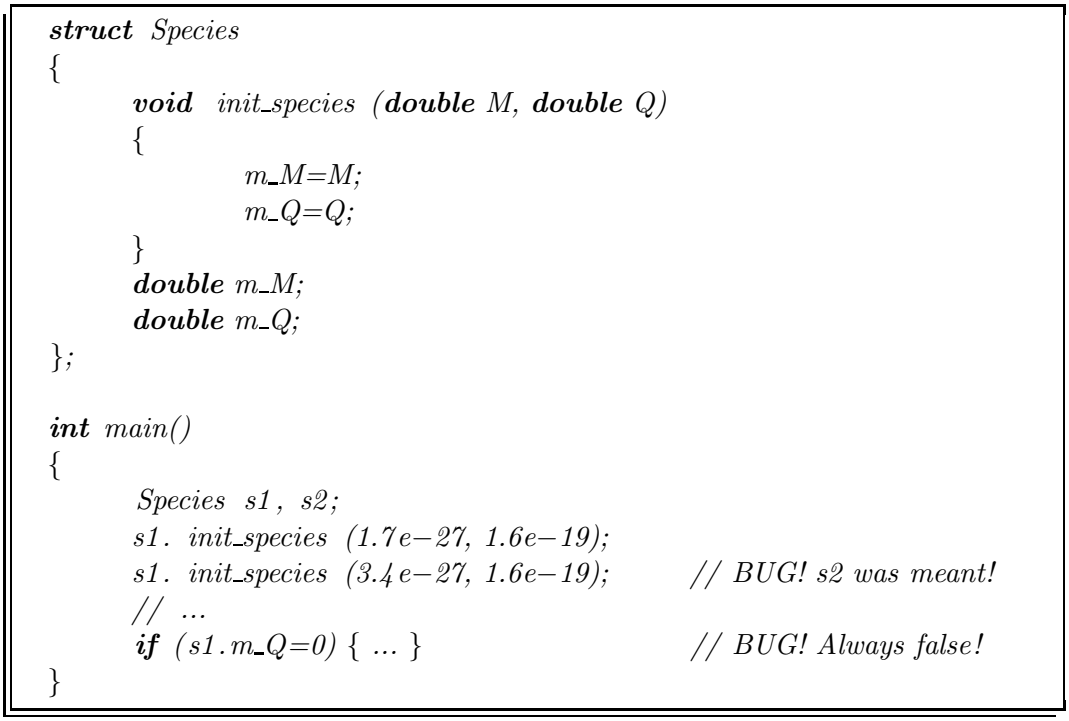
2.3 species2.cpp

We will slowly address the issues we have just mentioned by using some C++ features. First we rewrite the program to use a feature which is available only in C++, *member functions*, see figure 3.

```
struct Species
{
        double m_M;
        double m_Q;
};
void  init_species (Species* s, double M, double Q)
{
        s->m_M=M;
        s->m_Q=Q;
}

int main()
{
        Species s1, s2;
         init_species (&s1, 1.7e-27, 1.6e-19);
         init_species (&s1, 3.4e-27, 1.6e-19);     // BUG! s2 was meant!
        // ...
        if (s1.m_Q=0) { ... }                      // BUG! Always false!
}
```

Figure 2: species1.c: A (still) pathological C program. A structure of type *Species* is introduced which represents the properties of a particle type. The initialisation of the code is done with a dedicated function.



```
struct Species
{
        void  init_species (double M, double Q)
        {
                m_M=M;
                m_Q=Q;
        }
        double m_M;
        double m_Q;
};

int main()
{
        Species s1, s2;
        s1. init_species (1.7e-27, 1.6e-19);
        s1. init_species (3.4e-27, 1.6e-19);        // BUG! s2 was meant!
        // ...
        if (s1.m_Q=0) { ... }                       // BUG! Always false!
}
```

Figure 3: species2.cpp: A (still) pathological C++ program. The structure *Species* now contains not only data, but also the functions which operate on those data (in this case only the initialisation-member). Inside those functions we can use the data members of the structure.

```
struct Species
{
        Species (double M, double Q)
        {
                m_M=M;
                m_Q=Q;
        }
        double m_M;
        double m_Q;
};

int main()
{
        Species s1(1.7e-27, 0);
        Species s2;                          // OK. Does not compile
        // ...
        if (s1.m_Q=0) { ... }                // BUG! Always false!
}
```

Figure 4: species3.cpp: An (even less) pathological C++ program. The member *init_species* in the structure has been renamed to *Species*, the name of the structure in which it is contained. Such members are called *constructors*.

In this listing we have made the function which initializes a structure of type *Species* part if its definition. In the next version the advantages of this approach will be made clear, let us for now just appreciate the conceptual improvement. The bundling of data items and functions which operate on these data is typical for the C++ concept of *structures* and *classes*.

2.4 species3.cpp

In the program *spec3.cpp* in figure 4 the member *init_species* in the structure has been renamed to *Species*, the name of the structure in which it is contained. Such members are called *constructors*. If a structure contains at least one such members and a variable of the structure type is created, such a constructor is called. Creation (construction) of such variables is only possible if the user-provided arguments match those of one of the available constructors.

As a result, one of the errors we have previously made can be prevented. Neither the re-initialisation of *s1*, nor the uninitialised structure *s2* is possible: such code will simply not compile. By providing constructors, the writer of

```
struct Species
{
        Species (double M, double Q)
        {
                m_M=M;
                m_Q=Q;
        }
        double M() const { return m_M; }
        double Q() const { return m_Q; }
  private:
        double m_M;
        double m_Q;
};

int main()
{
        Species  s1(1.7e-27, 1.6e-19);
        Species  s2(3.4e-27, 1.6e-19);

        double charge = s2.Q();          // OK
        if (s1.m_Q=0) { ... }            // OK. Does not compile
        if (s1.Q()=0) { ... }            // OK. Does not compile
}
```

Figure 5: species4.cpp: A quite beatiful C++ program. Those data and functions which are present behind the ***private:*** keyword cannot be accessed from the outside, preventing accidental modification.

a piece of code can ensure that objects are in a well-defined state from the very moment these are created.

But still we have the erroneous assignment to *m_Q*. Let's prevent that, too.

2.5 species4.cpp

In the program species4.cpp in figure 5 we have introduced a new C++ keyword, ***private***. Data and functions which are present behind the ***private:*** keyword cannot be accessed from the outside. Instead, one can obtain the species' mass and charge with the help of two new function members, *M()* and *Q()*. This way accidental modification of the structure members is not possible. The indicated erroneous lines in the program simply cannot be

```
class Species
{
  public:
        Species (double M, double Q)
        {
                m_M=M;
                m_Q=Q;
        }
        double M() const { return m_M; }
        double Q() const { return m_Q; }
  private:
        double m_M;
        double m_Q;
};

int main()
{
        Species  s1(1.7e-27, 1.6e-19);
        Species  s2(3.4e-27, 1.6e-19);

        double charge = s2.Q();          // OK
        if (s1.m_Q=0) { ... }            // OK. Does not compile
        if (s1.Q()=0) { ... }            // OK. Does not compile
}
```

Figure 6: species5.cpp: An absolutely beatiful C++ program. A ***class*** is almost the same as a ***struct***, but members are ***private*** by default and must be made accessible with th ***public*** keyword.

compiled.

2.6 species5.cpp

We finally arrive at the famous concept of *classes*. A class is almost the same as a structure but members are ***private*** by default[1]. Members must explicitly be made accessible with the ***public*** keyword.

[1]The other diferences are of a highly technical nature and go beyond the scope of this small introduction.

3 Beyond classes

3.1 Overview

In the previous sections we have given a very practical introduction to the concept of classes and structures in C++, tailored to those who are familiar with C (and, like me, have undoubtedly made the abovementioned errors numerous times).

At this moment it may seem as if classes can be used only to prevent silly mistakes in code. Fortunately, there is more to the class concept and to C++ in general:

- built-in and user-defined data types are equal;
- Class derivation is supported;
- Polymorphism is supported;
- Templates provide a mechanism to write code which works for multiple data types.

This text will deal with the first topic. It will give you an idea of the expressiveness of the language, which facilitates developing testing and debugging complex projects, like Plasimo. The other topics may be discussed during the practical exercises which accompany the course.

3.2 Equivalence of built-in and user-defined data types

Bundling related data —like the mass and charge of a specific particle type— in a structure leads to better-readable code, also in bare C. C structures can be passed as a single argument to a function, that can then access all of the members of that particular structure. As an example, appropriately defined functions can be called as:

```
int i;
Species s;
// initialisation
func_of_int (i);
func_of_species (s);
```

So far, so good. But in reality, in C structures are second-class citizens. For the built-in types operations like addition, subtraction, multiplication and division have a well-defined meaning. For user-defined types these concepts are not defined.

Admittedly, addition is not meaningful for every user-defined type. But let us consider the *Species* structure. It would be great if we could express the formation of a molecule as the addition of atoms, as in

```
Species H;
Species O;
// initialisation of H and O
Species H2O=H+H+O;
```

In C, this cannot be made to work. There is no way '+', '-' and the like can be given a meaning for such user-defined types. (And, as stated before, creation of a variable and its initialisation are seperate actions.)

In C++, *operator overloading* is available. That is, the common operations can be defined for non-built-in types. With the species definitions as before we can write

```
int main()
{
        Species H( 1*1.6e-27, 0.0);
        Species O(16*1.6e-27, 0.0);
        Species H2O=H+H+O;
}
```

How do we tell the computer what it means to add two particles? The implementation of the binary addition operator is relatively straightforward. It looks much like a function, but with a special name, ***operator+***. It returns a *Species* object[2].

```
Species operator+(Species s1, Species s2)
{
        double Mtot=s1.M()+s2.M();
        double Qtot=s1.Q()+s2.Q();
        return Species(Mtot,Qtot);
}
```

Similarly, multiplication of a *Species* with an integer could be provided, allowing us to write

```
int main()
{
        Species H( 1*1.6e-27, 0.0);
        Species O(16*1.6e-27, 0.0);
        Species H2O=2*H+O;
}
```

The implementation of the required ***operator****(**int** n, Species s)* is left as an exercise for the reader.

[2]Experienced C++ hands will notice that the arguments s1 and s2 should rather be passed *by reference* to make the implementation more efficient. Such optimisation issues are outside the scope of this text

4 Conclusions

In this text some nice features of C++ have been unfolded. The remaining question may be: how does this relate to the course I am following? I am studying numerical plasma physics, right? The answer is twofold:

- It may help you to understand what happens under the hood in the code you are dealing with in the practical work which is part of the course. You may better understand the code which is present in *plasma.h*; everything we have said here about the implementation of a *Species* class applies to the code in that file as well;
- You may appreciate the software-engineering aspects of the job of numerically modelling (plasma) physics. We have seen that a well-designed C++ program can be more expressive and less error-prone than its equivalent in bare C. Such advantages tend to become more important in complex codes.

  You can imagine that a program which is capable of modeling fluid and kinetic, thermal and non-thermal plasmas, inductive, capacitive and waveguide EM fields, transport of mass, momentum, energy and radiation qualifies as such a code.

C++ has helped to develop Plasimo to its present state. But, concluding with a quote of Bjarne Stroustrup, the inventor of C++:

> Good design and the absence of errors cannot be guaranteed merely by the presence or absence of specific language features.