The ++ of C++

Jan van Dijk

Outline

The rationale of using C++: a case study

- We create code for handling two particles
- The C code contains lots of errors. Why?
- We gradually improve the C implementation
- We present the improved C++ version
- The expressiveness of C++

Bad-old C: myfirstspecies.c

```
int main()
{
        double M1, M2;
        double Q1, Q2;
        M1=1.7e-27;
        Q1=0;
        M2=3.4e-27;
        Q1=1.6e-19;
        if (Q1 =0) { ... }
}
```

This is sub-optimal code:

- Loose variables
- No expression of concept
- More particles? Hmmm...
- Note the two common bugs
- How do we fix this in C?

Use *structures*

Good-old C: a *structured* code

```
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);
        // ...
        if (s1.m_Q=0) { ... }
}
```

Better readable, but data and functions unrelated (and the bugs remain)

A structure in C++

```
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);
      // ...
      if (s1.m_Q=0) { ... }
      // ...
}
```

In C++ data and related functions are part of a structure.

Constructors: forcing proper initialisation

```
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;
        // ...
        if (s1.m_Q=0) { ... }
        //
        return 0;
}
```

This will not compile. Good! (But the other error remains.)

Limiting data access

```
struct Species
{
  Species(double M, double Q)
  {
    m_M=M;
    m_Q=Q;
  }
  double M() { return m_M; }
  double Q() { 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();
    if (s1.m_Q=0) { ... }
    if (s1.Q()=0) { ... }
}
```

This will not compile. Good!

A class: changing the default access rights

```
class Species
{
public:
  Species(double M, double Q)
  {
      m_M=M;
      m_Q=Q;
  }
  double M() { return m_M; }
  double Q() { 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 mass_1   = s1.M();
      double charge_2 = s2.Q();
}
```

Explicitly declare data/members as being accessible.

Built-in types and user-defined types

Both types can be used in functions:

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

... but this will never work in C:

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

User-defined types are second-class citizens.

Operator overloading

The following snippet is valid C++

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

What is the meaning of addition? We provide a 'function' **operator**+

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

Finishing touches

We may also define multiplication with integers.

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

What does **operator*** look like?

Conclusions

We have presented C++ as a 'better C':

- Forced initialisation
- Limiting data access
- User-defined operators

But there's much more in C++

- *Derivation:* Building a hierarchy, inherit properties
- *Templates:* Write data-type independent code
- *Polymorphism:* Write algorithms in terms of basic properties