60-objects-classes

June 2, 2024

1 About objects and classes

Modern C++ does not revolutionize the object-oriented features but rather brings many small improvements for most of the identified lacks.

1.1 Improvement of object constructions

Delegating constructors

[1]:
```
#include <iostream>
```

[2]:
```
struct ClassA {
    ClassA() : ClassA(0) { m_default = true ; }
    ClassA( int a_x ) : m_x {a_x} { m_default = false ; }
    int m_x ;
    bool m_default ;
} ;
```

[3]:
```
void display( ClassA const & a_obj )
{
  if (a_obj.m_default) { std::cout<<"(default)"<<a_obj.m_x<<std::endl ; }
  else { std::cout<<"(explicit)"<<a_obj.m_x<<std::endl ; }
}
```

[4]:
```
ClassA a1 {}, a2 {0} ;
display(a1) ;
display(a2) ;
```

```
(default)0
(explicit)0
```

Initialization of data members

[5]:
```
struct ClassB {
  ClassB() {} ;
  ClassB( int a_x ) : m_x {a_x} {}
  int m_x = 0 ;
} ;
```

[6]:
```
ClassB b0 {} ;
ClassB b1 {1} ;
std::cout << b0.m_x << std::endl ;
std::cout << b1.m_x << std::endl ;
```

```
0
1
```

Inheritance and constructors

[7]:
```
#include <iostream>
```

[8]:
```
struct ClassA {
    ClassA( int ) { std::cout << "ClassA(int)" << std::endl ; }
    ClassA( int, int ) { std::cout << "ClassA(int,int)" << std::endl ; }
} ;
```

[9]:
```
struct ClassB : public ClassA {
  using ClassA::ClassA ; // either A::A( int) or A::A( int, int ) can be used
                         // as if they were declared as B::B( int) and B::B(␣
 ↪int, int )
} ;
```

[10]:
```
ClassB b1 {1} ;
ClassB b2 {1,2} ;
```

```
ClassA(int)
ClassA(int,int)
```

1.2 Improvement of declaration of member functions

Forbid a function

[11]:
```
// class cannot be copied in C++03
class no_copies {
  public:
    no_copies() {}
  private:
    no_copies( const no_copies & ) ;
    no_copies & operator=( const no_copies & ) ;
} ;
```

[12]:
```
// class cannot be copied in C++11
class no_copies_v2 {
  public:
    no_copies_v2() {}
    no_copies_v2( no_copies_v2 const & ) = delete ;
    no_copies_v2 & operator=( no_copies_v2 const & ) = delete ;
} ;
```

We can now prevent certain implicit conversions

[13]:
```
struct FooStruct {
  void foo_method(short) {}
  void foo_method(int) = delete ;
} ;
```

[14]:
```
FooStruct s ;
s.foo_method(42) ; // Error, int overload declared deleted
s.foo_method(static_cast<short>(42)) ; // OK
```

```
input_line_20:3:3: error: call to deleted member
function 'foo_method'
s.foo_method(42) ; // Error, int overload declared deleted
~~^~~~~~~~~~

input_line_19:3:8: note: candidate function has been
explicitly deleted
  void foo_method(int) = delete ;
       ^

input_line_19:2:8: note: candidate function
  void foo_method(short) {}
       ^
```

```
Interpreter Error:
```

Modify the signature but keep the default implementation

[15]:
```
class Y {
  public:
    Y & operator=( Y const & ) = default ; // Make it explicit
    virtual ~Y() = default ; // Add virtual
  protected:
    Y() = default ; // Change access
} ;
```

1.3 Hiding is still an issue

When one use both inheritance and overloading, i.e. multiple functions with the same name but different signatures (the number or type of arguments), how does it work?

[23]:
```
struct Base
 {
  void display( int ) { std::cout<<"Base::display( int )"<<std::endl ; }
  void display( float ) { std::cout<<"Base::display( float )"<<std::endl ; }
 } ;
```

[24]:
```
struct Derived : public Base
 {
  void display( float ) { std::cout<<"Derived::display( float )"<<std::endl ; }
  void display( double ) { std::cout<<"Derived::display( double )"<<std::endl ;␣
 ↪}
 } ;
```

[25]:
```
void check( Derived obj )
 {
  obj.display(42) ;
  obj.display(3.14f) ;
  obj.display(3.14) ;
 }
```

```
input_line_31:3:7: error: call to member function

'display' is ambiguous
  obj.display(42) ;
  ~~~~^~~~~~~

input_line_30:3:8: note: candidate function
  void display( float ) { std::cout<<"Derived::display( float )"<<std::endl ; }
       ^

input_line_30:4:8: note: candidate function
  void display( double ) { std::cout<<"Derived::display( double )"<<std::endl ;
}
       ^
```

```
Interpreter Error:
```

When compiling `obj.mf(...)`, the compiler follow those steps: 1. search some member function named `mf` in the class of `obj`; 2. while not found at least one `mf`, move to its base classes, one after the other; 3. in the selected class, within all overloaded `mf` functions, select the one which fit better the call arguments `(...)`.

1.3.1 Good old-fashioned practice

When you redefine an inherited member function, **redefine all the base class functions which share the same name**, or you will hide some of them.

1.4 Improvement of virtual member functions

Control the redefinition of virtual methods

[16]:
```
class ClassA
 {
  public :
```

```
    virtual void fct1() =0 ;
    virtual void fct2( int ) =0 ;
    virtual void fct3( bool ) =0 ;
 } ;
```

[17]:
```
class ClassB : public ClassA
 {
  public :
    void fct1() override ;       // OK
    void ft2( int ) override ;   // error: A::ft2 does not exist
    void fct2( bool ) override ; // error: not the good types
 } ;
```

```
input_line_23:5:21: error: only virtual member
functions can be marked 'override'
    void ft2( int ) override ;   // error: A::ft2 does not exist
                    ^~~~~~~~

input_line_23:6:23: error: non-virtual member
function marked 'override' hides virtual member function
    void fct2( bool ) override ; // error: not the good types
                      ^

input_line_22:5:18: note: hidden overloaded virtual
function '__cling_N515::ClassA::fct2' declared here: type mismatch at 1st
parameter ('int' vs 'bool')
    virtual void fct2( int ) =0 ;
                 ^
```

```
Interpreter Error:
```

Forbid redefining virtual methods and abstract classes

[18]:
```
struct ClassA {
    virtual void fct1() =0 ;
    virtual void fct2( int ) =0 ;
} ;
```

[20]:
```
struct ClassB : public ClassA {
    virtual void fct3( bool ) final ;
} ;
```

[21]:
```
struct ClassC final : public ClassB {
    void fct1() override ;       // OK
    void fct3( bool ) override ; // error: B::fct3 is final
} ;
```

```
input_line_27:4:10: error: declaration of 'fct3'
overrides a 'final' function
    void fct3( bool ) override ; // error: B::fct3 is final
         ^

input_line_26:2:18: note: overridden virtual function
is here
    virtual void fct3( bool ) final ;
                 ^

input_line_27:2:9: warning: abstract class is
marked 'final' [-Wabstract-final-class]
 struct ClassC final : public ClassB {
        ^

input_line_24:3:18: note: unimplemented pure virtual
method 'fct2' in 'ClassC'
    virtual void fct2( int ) =0 ;
                 ^
```

```
Interpreter Error:
```

[22]:
```
struct ClassD : public ClassC {} ;                // error: C is final
```

```
input_line_28:1:24: error: expected class name
struct ClassD : public ClassC {} ;                // error: C is final
                       ^
```

```
Interpreter Error:
```

Reminder: make the destructor virtual Indeed, you may create a derived object with `new` and store its address in a base class pointer. When calling `delete` on this pointer, one will bypass the derived destructor, unless it is virtual.

[31]:
```
class ClassA
 {
  public :
    virtual void fct1() =0 ;
    virtual void fct2( int ) =0 ;
    virtual void fct3( bool ) =0 ;
    virtual ~ClassA() {}
 } ;
```

Reminder: avoid virtual functions at low level Because of the cost, virtual functions are rather used for large and not numerous objects: the upper software layers of your application. For what concerns the numerous and small objects, it is preferable to use templates (which slow down the compilation but have no effect during execution).

1.5 Reminder about function-objects

If a class supply an `operator()`, its objects can behave similarly to a function. The member variables of the class can be seen as extraneous parameters for this function.

[1]:
```
#include <iostream>
```

[2]:
```
class LinearFunction
 {
  public :
    LinearFunction( double constant ) : m_constant(constant) {}
    double operator()( double value ) { return m_constant*value ; }
  private :
    double m_constant ;
 } ;
```

[3]:
```
LinearFunction times2(2) ;
const int SIZE = 10 ;
double values[SIZE] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 } ;
for ( int i=0 ; i<SIZE ; ++i )
 { std::cout << times2(values[i]) << " " ; }
```

```
0 2 4 6 8 10 12 14 16 18
```

1.5.1 The recommended use of function-objects

This type of classes is especially used with algorithms of the standard library.

Below, we review the previous function-object, in order to integrate the call to `std::cout`, and then we combine it with `std::foreach`.

[4]:
```
#include <iostream>
```

[5]:
```
class LinearFunctionPrint {
  public :
    LinearFunctionPrint( double constant ) : m_constant(constant) {}
    void operator()( double value )
     { std::cout << (m_constant*value) << " " ; }
  private :
    double m_constant ;
 } ;
```

[6]:
```
const int SIZE = 10 ;
double values[SIZE] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 } ;
```

[7]:
```
#include <algorithm>
```

[8]:
```
LinearFunctionPrint times2cout(2) ;
std::for_each(values,values+SIZE,times2cout) ;
```

```
0 2 4 6 8 10 12 14 16 18
```

[9]:
```
std::for_each(values,values+SIZE,LinearFunctionPrint(2)) ;
```

```
0 2 4 6 8 10 12 14 16 18
```

Objet-functions are classes. They can also use inheritance:

[10]:
```
class LinearFunctionPrint : public LinearFunction
 {
  public :
    LinearFunctionPrint( double constant ) : LinearFunction(constant) {}
    void operator()( double value )
     { std::cout << LinearFunction::operator()(value) << " " ; }
 } ;
```

[11]:
```
std::for_each(values,values+SIZE,LinearFunctionPrint(2)) ;
```

```
0 2 4 6 8 10 12 14 16 18
```

2 Take away

- The new `delete` keyword forbids more explicitly some functions.
- The new `override` keyword helps to detect typos when redefining inherited member functions.
- Still, be aware of the **hiding pitfall** if you overload with different signatures.
- Avoid the costly virtual functions for low level objects.
- Function-objects can incorporate parameters and be passed on to algorithms.

3 Questions ?

4 Exercise

In the code below: - insert one `= delete`, one `= default` and one `override` ; - just in case, give default values to the members variables ; - prevent the compiler from implicitly transforming a double into a particle. - what is still lacking?

[24]:
```
%%file tmp.objets.cpp

#include <cstdlib> // pour std::rand()
#include <iostream>
#include <string>

class Particle
 {
```

```
  public  :
    Particle( double a_mass ) : m_mass {a_mass} {}
    double mass() { return m_mass ; }
    virtual std::string name() { return "Particle" ; }
    ~Particle() {}
  private  :
    Particle( Particle const & ) ; // non copiable
    double m_mass ;
 } ;

class ChargedParticle : public Particle
 {
  public  :
    ChargedParticle( double a_mass, double a_charge )
     : Particle(a_mass), m_charge {a_charge} {}
    double charge() { return m_charge ; }
    virtual std::string name() { return "ChargedParticle" ; }
  private  :
    double m_charge ;
 } ;

void print( Particle & a_p  )
 {
  std::cout << a_p.name() << std::endl ;
  std::cout << "  mass = " << p.mass() << std::endl ;
 }

int main()
 {
  for ( int i = 0 ; i < 5 ; ++i )
   {
    if ( std::rand() < (.5*RAND_MAX) )
     {
      Particle p {2} ;
      print(p) ;
     }
    else
     {
      ChargedParticle p {1,1} ;
      print(p) ;
      std::cout << "  charge = " << p.charge() << std::endl ;
     }
   }
 }
```

```
Overwriting tmp.objets.cpp
```

[25]:
```
!rm -f tmp.objets.exe && g++ -std=c++17 tmp.objets.cpp -o tmp.objets.exe
```

[26]:
```
!./tmp.objets.exe
```

```
ChargedParticle
  mass = 1
  charge = 1
Particle
  mass = 2
ChargedParticle
  mass = 1
  charge = 1
ChargedParticle
  mass = 1
  charge = 1
ChargedParticle
  mass = 1
  charge = 1
```

5 Resources

- CTAD