Advanced C++ programming

Static and constant objects

Giuseppe Lipari - Luigi Palopoli

CRIStAL - Université de Lille
Embedded Intelligence and Robotic Systems - Università di Trento

Outline

Outline

Use of inheritance

- Inheritance should be used when we have a isA relation between objects
  - you can say that a circle is a shape
  - you can say that a rect is a shape
- What if the derived class contains some special function that is useful only for that class?
  - Suppose that we need to compute the diagonal of a rectangle
  - see `examples/rect.hpp`

isA vs. isLikeA

- If we put function `diagonal()` only in `Rect`, we cannot call it with a pointer to shape
  - In fact, `diagonal()` is not part of the interface of `Shape`
- If we put function `diagonal()` in `Shape`, it is inherited by `Triangle` and `Circle`
  - `diagonal()` does not make sense for a `Circle`
  - we should raise an error when `diagonal()` is called on a `Circle`
- One solution is to put the function in the `Shape` interface
  - it will return an error for the other classes, like `Triangle` and `Circle`
- another solution is to put it only in `Rect` and then make a downcasting when necessary
  - see `examples/shapes_main.cpp` for the two solutions

Casting

- Traditional explicit type-casting allows to convert any pointer into any other pointer type, independently of the types they point to.
- The subsequent call to member will produce either a run-time error or a unexpected result.
- There are safer ways to perform casting

```
dynamic_cast <new_type> (expression)
reinterpret_cast <new_type> (expression)
static_cast <new_type> (expression)
const_cast <new_type> (expression)
```

Dynamic cast

- It can be used only with pointers and references to objects.
  - The cast is solved at run-time, by looking inside the structure of the object
  - This feature is called `run-time type identification` (RTTI)
- Its purpose is to ensure that the result of the type conversion is a valid complete object of the requested class.
  - The result is the pointer itself if the conversion is possible;
  - The result is `nullptr` if the conversion is not possible

Example

```
class CBase { virtual void dummy() {} };
class CDerived: public CBase { int a; };

int main () {
    CBase * pba = new CDerived;
    CBase * pbb = new CBase;
    CDerived * pd;
    pd = dynamic_cast<CDerived*>(pba);
    if (pd==nullptr) cout << "Null pointer on first
   type-cast" << endl;
    pd = dynamic_cast<CDerived*>(pbb);
    if (pd==nullptr) cout << "Null pointer on second
   type-cast" << endl;
    return 0;
}
```

static_cast

- `static_cast` can perform conversions between pointers to related classes
- however, no safety check is performed during runtime to check if the object being converted is in fact a full object of the destination type.
- Therefore, it is up to the programmer to ensure that the conversion is safe.

```
class CBase {};
class CDerived: public CBase {};
CBase * a = new CBase;
CDerived * b = static_cast<CDerived*>(a);
```

- `b` would point to an incomplete object of the class and could lead to runtime errors if dereferenced.

reinterpret_cast

- Converts any pointer type to any other pointer type, even of unrelated classes.
- The result of the operation is a simple binary copy of the value from one pointer to the other.
- All pointer conversions are allowed: neither the content pointed nor the pointer type itself is checked.
- It can also cast pointers to or from integer types.
- This can be useful in low-level non portable code (i.e. interaction with interrupt handlers, device drivers, etc.)

const_cast

- This type of casting manipulates the constness of the type, either to be set or to be removed.
- For example, in order to pass a const argument to a function that expects a non-constant parameter

```
   // const_cast
#include <iostream>
using namespace std;

void print (char * str)
{
  cout << str << endl;
}

int main () {
  const char * c = "sample text";
  print ( const_cast<char *> (c) );
  return 0;
}
```

Outline

Meaning of static

- In C/C++ static has several meanings
  - for global variables, it means that the variable is not exported in the global symbol table to the linker, and cannot be used in other compilation units
  - local variables, it means that the variable is not allocated on the stack: therefore, its value is maintained through different function instances
  - for class data members, it means that there is only one instance of the member across all objects (as in Java)
  - a static function member can only act on static data members of the class (as in Java)

Static members

- We would like to implement a counter that keeps track of the number of objects that are around
- We can use a static variable

```
class ManyObj {
    static int count;
    int index;
public:
    ManyObj();
    ~ManyObj();

    int getIndex();
    static int howMany();
};
```

```
int ManyObj::count = 0;

ManyObj::ManyObj() {
    index = count++;
}
ManyObj::~ManyObj() {
    count--;
}
int ManyObj::getIndex() {
    return index;
}
int ManyObj::howMany() {
    return count;
}
```

Static members

```
int main()
{
  ManyObj a, b, c, d;
  ManyObj *p = new ManyObj;
  ManyObj *p2 = 0;
  cout << "Index of p: " << p->getIndex() << "\n";
  {
        ManyObj a, b, c, d;
        p2 = new ManyObj;
        cout << "Number of objs: " << ManyObj::howMany() <<
    "\n";
  }
  cout << "Number of objs: " << ManyObj::howMany() << "\n";
  delete p2; delete p;
  cout << "Number of objs: " << ManyObj::howMany() << "\n";
}
```

Output:

```
Index of p: 4
Number of objs: 10
Number of objs: 6
```

Static members

- There is only one copy of the static variable for all the objects
- All the objects refer to this variable
- How to initialise a static member?
  - cannot be initialised in the class declaration
  - the compiler does not allocate space for the static member until it is initiliazed
  - So, the programmer of the class must define and initialise the static variable

Static data members

- Static data members need to be initialised when the program starts, before the main is invoked
  - they can be seen as global initialised variables (and this is how they are implemented)

```
// include file A.hpp
class A {
  static int i;
public:
  A();
  int get();
};
```

```
// src file A.cpp
#include "A.hpp"

int A::i = 0;

A::A() {...}
int A::get() {...}
```

Initialisation

It is usually done in the .cpp file where the class is implemented

```
int ManyObj::count = 0;

ManyObj::ManyObj() { index = count++;}
ManyObj::~ManyObj() {count--;}
int ManyObj::getIndex() {return index;}
int ManyObj::howMany() {return count;}
```

- There is a famous problem with static members, known as the *static initialisation order failure*

The static initialisation fiasco

- When static members are complex objects, that depend on each other, we have to be careful with the order of initialisation
  - initialisation is performed just after the loading, and before the main starts.
  - Within a specific translation unit, the order of initialisation of static objects is guaranteed to be the order in which the object definitions appear in that translation unit. The order of destruction is guaranteed to be the reverse of the order of initialisation.
  - However, there is no guarantee concerning the order of initialisation of static objects across translation units, and the language provides no way to specify this order. (undefined in C++ standard)
  - If a static object of class `A` depends on a static object of class `B`, we have to make sure that the second object is initialised before the first one

Solutions

- The **Nifty counter** (or Schwartz counter) technique
  - Used in the standard library, quite complex as it requires an extra class that takes care of the initialisation
- The Construction on first use technique
  - Much simpler, use the initialisation inside function

Construction on first use

- It takes advantage of the following C/C++ property
  - Static objects inside functions are only initialised on the first call
- Therefore, the idea is to declare the static objects inside global functions that return references to the objects themselves
- Access to the static objects happens only through those global functions (see Singleton)

Copy constructors and static members

What happens if the copy constructor is called?

```
void func(ManyObj a)
{
    ...
}

void main()
{
    ManyObj a;
    func(a);
    cout << "How many: " << ManyObj::howMany() << "\n";
}
```

- What is the output?
- Solution in `examples/manyobj.cpp`

Example 1: singleton

A singleton is a class for which it is possible to define a single object.

```
struct singleton_t
{

  static
  singleton_t &
  instance()
  {
    static singleton_t s;
    return s;
  }

  singleton_t(const singleton_t &) =
      delete;
  singleton_t & operator = (const
      singleton_t &) = delete;

private:

  singleton_t() {}
  ~singleton_t() {}

};
```

Initialisation takes place when the control flow hits this function for the first time. The lifetime spans from the first time the control flow reaches this point to the end of the prograam.

The copy constructor and the assignment operator are disabled.

The conststructor is private. So it can only be seen from the instance function

Example 2: Factory

Suppose you have the same class Point, which can be based on cartesian or on polar coordinates.

```
struct Point {
  Point(float x, float y){ /*...*/ } // Cartesian co-
  ordinates
  Point(float a, float b){ /*...*/ } // Polar co-ordinates
};
```

Not ok! I cannot overload the same type of arguments!

Example 2: Factory (1)

A second possibility is the following

```
enum class PointType{ cartesian, polar };

class Point {
    Point(float a, float b, PointTypetype = PointType::cartesian) {
        if (type == PointType::cartesian) {
            x = a; b = y;
        }
        else {
            x = a * cos(b);
            y = a * sin(b);
        }
    }
};
```

- It works. But is it elegant?

Example 2: Factory (1)

A second possibility is the following

```
enum class PointType{ cartesian, polar };

class Point {
    Point(float a, float b, PointTypetype = PointType::cartesian) {
        if (type == PointType::cartesian) {
            x = a; b = y;
        }
        else {
            x = a * cos(b);
            y = a * sin(b);
        }
    }
};
```

- It works. But is it elegant?
- We should delegate different instantions to different methods

Example 2: Factory (2)

A third possibility is the factory = private constructor + static functions

```
enum class PointType { cartesian, polar };
class Point {
    float          m_x;
    float          m_y;
    PointType      m_type;
    // Private constructor, so that object can't be created directly
    Point(const float x, const float y, PointType t) : m_x{x}, m_y{y}, m_type{t}
      {}
  public:
    friend ostream &operator<<(ostream &os, const Point &obj) {
        return os << "x: " << obj.m_x << " y: " << obj.m_y;
    }
    static Point NewCartesian(float x, float y) {
        return {x, y, PointType::cartesian};
    }
    static Point NewPolar(float a, float b) {
        return {a * cos(b), a * sin(b), PointType::polar};
    }
};
int main() {
    // Point p{ 1,2 };  // will not work
    auto p = Point::NewPolar(5, M_PI_4);
    cout << p << endl;  // x: 3.53553 y: 3.53553
    return EXIT_SUCCESS;
}
```

Example 2: Factory (3)

If we have dedicated code for construction, we can also have a dedicated calss.

```
class Point {
    float   m_x;
    float   m_y;

    Point(float x, float y) : m_x(x), m_y(y) {}
public:
//The class is defined interally to Point to establish a clear relation between
     points and their factory
struct Factory {
        static Point NewCartesian(float x, float y) { return { x,y }; }
        static Point NewPolar(float r, float theta) { return{ r*cos(theta), r*
     sin(theta) }; }
    };
};

int main() {
    auto p = Point::Factory::NewCartesian(2, 3);
    return EXIT_SUCCESS;
}
```

Outline

Const

- In C++, when something is const it means that it cannot change. Period.
- Then, the particular meanings of const are a lot:
- Don't to get lost! Keep in mind: const = cannot change
- Another thing to remember: constants must have an initial (and final) value!

Constants - I

- As a first use, const can substitute the use of `#define` in C
  - whenever you need a constant global value, use const instead of a define, because it is clean and it is type-safe

```
#define PI 3.14           // C style
const double pi = 3.14; // C++ style
```

- In this case, the compiler does not allocate storage for pi
- In any case, the const object has an `internal linkage`

Constants - II

- You can use const for variables that never change after initialisation. However, their initial value is decided at run-time

```
const int i = 100;
const int j = i + 10;

int main()
{
    cout << "Type a character\n"
    ;
    const char c = cin.get();
    const char c2 = c + 'a';
    cout << c2;

    c2++;
}
```

Compile-time constants

run-time constants

ERROR! c2 is const!

Const and pointers

- There are two possibilities
  1. the pointer itself is constant
  2. the pointed object is constant

```
int a
int * const u = &a;

const int *v;
```

the pointer is constant

the pointed object is constant (the pointer can change and point to another const int!)

- Remember: a const object needs an initial value!

Const function arguments

- An argument can be declared constant. It means the function can't change it
- this is particularly useful with references

```
class A {
public:
    int i;
};

void f(const A &a) {
  a.i++;       // error! cannot modify a;
}
```

- You can do the same thing with a pointer to a constant, but the syntax is messy.

Passing by const reference

- Remember:
  - we can pass argument by value, by pointer or by reference
  - in the last two cases we can declare the pointer or the reference to refer to a constant object: it means the function cannot change it
  - Passing by constant reference is equivalent, from the user point of view, to passing by value
  - From an implementation point of view, passing by const reference is much faster!!

Constant member functions

- A member function can be declared constant
  - It means that it will not modify the object
  - The compiler can call only const member functions on a const object

```
class A {
    int i;
public:
    int f() const;
    void g();
};
void A::f() const
{
    i++;        // ERROR!
    return i;
}
```

```
void myfun(const A &a)
{
    a.f();      // Ok
    a.g();      // ERROR!!
}
```

Constexpr (c++11)

- Since C++11, we can declare a variable or a function as `constexpr`
  - `constexpr` means that the corresponding value can (under certain circumstances) be computed at compile time
  - so, it is useful for optimisation, substitutes and expands inline
  - for objects, constexpr is equivalent to const
  - for functions, it means that the function can be calculated at compile time
  - See constexpr

Example of constexpr

```
// C++11: constexpr funcs use recursion
// C++14: constexpr funcs use local variables and loops
constexpr int factorial(int n)
{
  return n <= 1 ? 1 : (n * factorial(n - 1));
}

int main()
{
  int k = factorial(5); // computed at compile time
  volatile j = 4;
  int h = factorial(j); // computed at run-time
}
```