Inheritance in C++

Rationale for inheritance

We have : circle class, nice features (encapsulation, protection).

We want rectangle class. Expanded features : translation and printing. Then we want to handle *shapes*. Rectangle and circle are shapes, shapes can be translated and printed. A shape is either circle or rectangle.

There is a shape module in our program, sub-modules are particular kind of shapes. We want the module to be expandable.

Abstract vs Concrete

Class:

- Abstract represent an abstraction, cannot be instantiated, has at least one abstract method.
- Concrete can be instantiated and has no asbtract method.

An abstract method is a method whose code cannot be given, is just declared and will be defined in sub-classes.

Thus shape is an abstraction, and is an abstract type representing several concrete types.

However an abstract class can have attributes and can provide methods with their definitions. (attributes $\rightarrow$ a constructor)

Shape as a C++ abstract class :

1. shape has an interface
2. a constructor
3. a destructor
4. a translation method
5. an abstract print method
6. a "protected" accessibility area
7. a couple of hidden attributes.

virtual is a keyword for abstraction.

To make a method abstract in C++, its declaration starts with **virtual** and ends with "= 0".

```
virtual print() const = 0;
```

Calling print on a shape is valid, but shape::print cannot be coded.

An abstract class looks like a concrete one.

Definitions + playing with words

"is-a" relationship is known as sub-classing, then *circle* is a sub-class of *shape*, inherits from *shape*, derives from *shape* and extends *shape*.

A set of classes related by a "is-a" relationship is called a **class hierarchy**, usually a tree, depicted upside-down (superclasses at the top, subclasses at the bottom).

Subclassing

Circle as a C++ subclass:

8. knowing the base class of circle is required
9. the sub-class relationship is translated by **public**
10. **public** start the class interface
11. a constructor
12. a print definition tagged with **override**
13. new attributes

The class circle inherited from shape:

- the translate method
- the couple of attributes (x, y)

except that is implicit.

so:

- a circle can be translated
- circle has three attributes

```
sizeof(circle) == 3 * sizeof(float) + sizeof(void*))
```

In .cc:

```
void circle::print() const // no ”override”, only in .hh
{
  assert(r > 0.f_); // invariant
  std::cout << '(' << x_ << ”,” << y_ << ”,” << r_ << ')' << std::endl;
}
```

Playing with types

Transtyping

The static type of the object is the type of the variable, known at compile-time.

The dynamic type of the object is its type at instantiation, “exact type”, known at run-time.

```
void foo(const shape& s)
{
  s.print();
}
```

static type of s is shape, but its dynamic type is unknown.

```
void foo(const shape& s)
{
  s.print();
}

int main()
{
  foo(circle{0, 0, 1});
}
```

s has circle has a dynamic type and shape has a static type.

Remark that we can “const reference” a temporary object !

We can write

```
circle* c = new circle{0, 0, 1};
shape* s = c;
```

This is a transtypage.

A pointer to a shape is expected (s), you give a pointer to a circle (c), the assignment is valid.

The same goes for references.

We cannot instantiate abstract class but we can manipulate instances of an abstract class.

We can then , with transtypage, promote constness, change the static type from a derived to a base class and

Accessibility

final means last override.

```cpp
class A
{
  virtual void foo() = 0;
}

class B : public A
{
  void foo() override final;
}

class C : public B
{
  // cannot override foo
}
```

In C++, copy on return are optimized, not in call.

C++ allow overloading.

Conclusion

We can only create instances of leaf classes of the hierarchy.

Object-orientaion = Object + Class hierarchies

```cpp
class Base
{
  // ...
}

class Derived : Base
{
  // ...
}
```

```cpp
derived::derived()
  : base(),
    d_(0) // ...
{
```

```
  // ... allocate ressources when needed.
}
```

Please strictly follows idioms of slides

Smart pointers: Part I

(Raw) Pointers

- Pointers in C are a powerful means to play with memory.
- Pointers are an important means to refer to another place.
- Pointers are 0/1 containers.
- Pointers manage dynamically allocated memory

All this points are wrong in C++.

Ownership on pointers : point to (do not delete it) or hold some new'd object (do delete it).

Many OO languages offer only reference semantics, so everything is actually a pointer, Java, C#

Deletion comes back to ownership.

Smart pointers:

- looks like pointers
- behave like pointers
- manage ownership
- make your programs more robust

Shared pointers

```
int main()
{
  using shape_ptr
    = const shape*;
  auto v
    = std::vector<shape_ptr>{};
  v.push_back(new circle{});
  v.emplace_back(new square{});
  for (auto s: v)
    s->print;
}
```

outputs:

```
virtual void circle::print() const
virtual void square::print() const
```

std::vector :

- a dynamic (resizable) array of shape_ptr
- emplace_back and push_back means “append”

```
int main()
{
  using shape_ptr
    = std::shared_ptr<const shape>;
  auto v
    = std::vector<shape_ptr>{};
  v.push_back(new circle{});
  v.emplace_back(new square{});
  for (auto s: v)
    s->print;
}
```

outputs:

```
virtual void circle::print() const
virtual void square::print() const
virtual shape::~shape()
virtual shape::~shape()
```

We have no more memory leaks

When a shared pointer is destroyed, it deletes the memory it points to only if it is the last shared pointer to points at this memory.

For shared pointers, do std::make_shared<Foo>(args)

Introducing **auto** and **decltype**

```
auto p = std::make_shared<test>();
p->noop();

decltype(p) p2 = p;
```

auto is often for not writing types

auto and **decltype** are also great to rely on the compiler