

# Object Oriented Software Design

## Templates and Exceptions

Giuseppe Lipari  
<http://retis.sssup.it/~lipari>

Scuola Superiore Sant'Anna – Pisa

December 15, 2010

## Outline

- 1 Containers and reuse
- 2 Template syntax
- 3 Typename
- 4 Member templates
- 5 Function templates

- One of the most important keywords in OO programming is **code reuse**
- If you have a piece of code that works correctly, you want to reuse it as much as you can:
  - because it has been tested and used, so there is more probability that it contains less bugs
  - because you do not need to redo the same thing again (so you save time and production cost)
- We have already seen that reusing software is far from being trivial (see LSP)
- However, it is the *lapis philosophorum* (Philosopher's Stone) of all programmers

## Reuse though inheritance

- In the previous slides, we have seen one particular OO technique for reusing code: inheritance
  - it achieves reuse through abstraction
  - A concept is abstracted, and then a hierarchy of concepts are linked together through inheritance
  - however, inheritance may not be the best approach to reuse!

# Containers

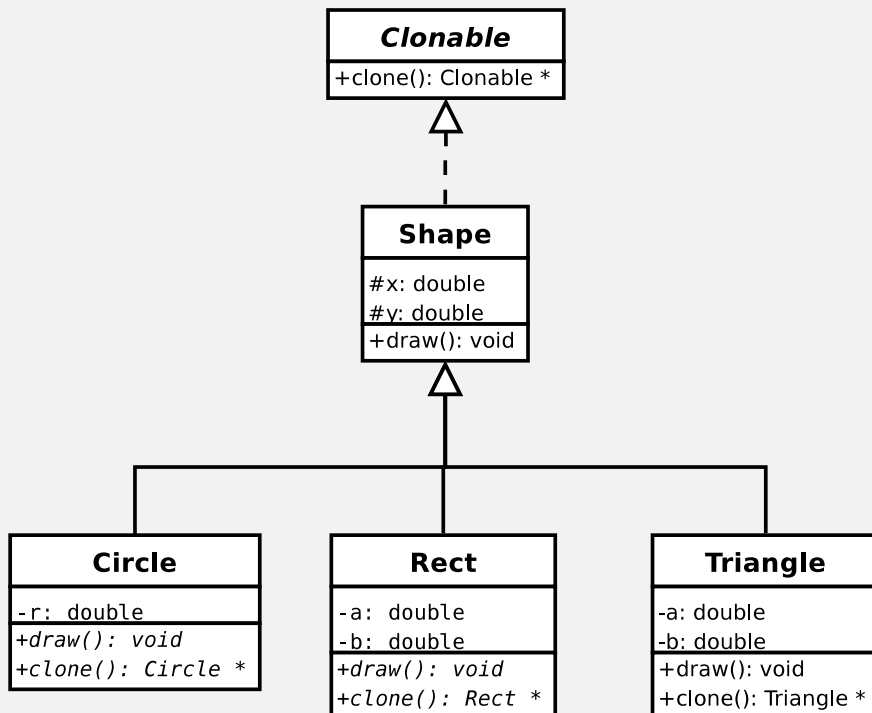
- Consider the problem of providing a generic container of objects
- Example:
  - We designed and developed a Stack class container
  - it is an object that *contains* other objects, and provides operations for inserting, extracting, finding object, and visiting them in a certain order
  - Our stack class contains integers
  - However, the code is generic enough and depends only in minimal part from the fact that it contains integers
- Problem: how to extend it to contains other types of objects?
  - for example, Shapes
- In the early days of programming the solution would have been:
  - Copy and paste the code
  - modify it to use Shape instead of int
- Can you enumerate the problems with this approach?

# Use inheritance

- OO languages that do not have templates, use inheritance for implementing such containers
  - For example, in Smalltalk (and in Java), all classes derive from a common ancestor: Object
  - The containers will contain pointers to Object
  - however, the type is lost when you insert an object in a container
  - The user has to perform an appropriate downcast to get back to the original type
- We can do something similar in C++, by using multiple interface inheritance

# Using interfaces

- The following interface specifies that an object can be cloned



## Using interfaces with Stack

- Stack now contains pointers to Clonable objects (i.e. objects that possess the Clonable interface)

```
class Clonable {
public:
    virtual Clonable * clone() = 0;
    virtual ~Clonable();
};

class Stack {
    class Elem {...};
public:
    class Iterator {...};

    Stack();
    ~Stack();

    void push(Clonable *d);
    Clonable * pop();
    int size();
    ...
};
```

## Exercise

- 1 Extend the previous `Stack` class by deriving it also from the `Clone` interface. It will provide a `clone()` virtual function that creates a new stack that contains copies of all elements in the original stack
  - To show that they are actually different, first implement a static id counter in the `Shape` class, so that every object has its own id
- 2 Now, write a `OrderedList` class that contains objects with interface `Comparable`
  - The objects in the list must be inserted according to an order decided by a function `lessThan()` that returns true if the object is “less” than the object passed as argument:

```
class Comparable {  
public:  
    virtual bool lessThan(Comparable *obj) = 0;  
    virtual ~Comparable() {}  
};
```

- Extend class hierarchy `shapes` to also derive from `Comparable`, and shapes must be ordered by their x position

## Problems with this approach

The problems with this approach are the following:

- It is necessary to modify the code for the objects (they must derive from the appropriate interfaces)
  - A possible solution is to write a *wrapper* object that derives from the contained object and from the interface
- It is necessary to downcast at least once
  - Type safety is lost, the compiler cannot check anything meaningful during compilation time
  - For example, it is not possible to avoid that different types of objects are inserted in the same container by mistake

# Templates

- Templates are used for generic programming
- The general idea is: what we want to reuse is not only the abstract concept, but **the code itself**
- with templates we reuse algorithms by making them general
- As an example, consider the code needed to swap two objects of the same type (i.e. two pointers)

```
void swap(int &a, int &b)
{
    int tmp;
    tmp = a;
    a = b;
    b = tmp;
}
...
int x=5, y=8;
swap(x, y);
```

- Can we make it generic?

## Solution

- By using templates, we can write

```
template<class T>
void swap(T &a, T &b)
{
    T tmp;
    tmp = a;
    a = b;
    b = tmp;
}
...
int x=5, y=8;
swap<int>(x, y);
```

- Apart from the first line, we have just substituted the type `int` with a generic type `T`

# How does it work?

- The template mechanism resembles the macro mechanism in C
  - We can do the same in C by using pre-processing macros:

```
#define swap(type, a, b) { type tmp; tmp=a; a=b; b=tmp; }  
...  
int x = 5; int y = 8;  
  
swap(int, x, y);
```

- in this case, the C preprocessor substitutes the code
  - it works only if the programmer knows what he is doing
- The template mechanism does something similar
  - but the compiler performs all necessary type checking

## Code duplicates

- The compiler will instantiate a version of `swap()` with integer as a internal type
- if you call `swap()` with a different type, the compiler will generate a new version
  - Only when a template is instantiated, the code is generated
    - If we do not use `swap()`, the code is never generated, even if we include it!
    - if there is some error in `swap()`, the compiler will never find it until it tries to generate the code
- Looking from a different point of view:
  - the template mechanism is like cut&paste done by the compiler at compiling time

## Swap for other types

- What happens if we call swap for a different type:

```
class A { ... };  
A x;  
A y;  
...  
  
swap<A>(x, y);
```

- A new version of swap is automatically generated
  - Of course, the class A must support the assignment operator, otherwise the generation fails to compile
  - see `./examples/16.cpp` - `examples/swap.cpp`

## Type Deduction

- Parameters can be automatically implied by the compiler

```
int a = 5, b = 8;  
  
swap(a, b);    // equivalent to swap<int>(a, b);
```

- Sometimes, this is not so straightforward ...

# Generalizing Stack

- Now, let's go back to our Stack class, and generalize it to contain any type of object

```
template<class T>
class Stack {
    class Elem {
    public:
        T data_;
        ...
    };
public:
    class Iterator {
        friend class Stack<T>;
        ...
    public:
        inline T operator*() const { ... }
        ...
    };

    Stack() : head_(0), size_(0) {}
    ~Stack() {...}

    void push(const T &a) {...}
    T pop() {...}
    ...
}
```

## Exercises

- 1 Write a program that inserts pointers to shapes into the stack
  - You will only need to modify the main!
- 2 Write a program that inserts only rectangles into a stack
- 3 Write a `OrderedList` container that makes use of `operator<( )` to compare elements. Insert shapes into it, and order by increasing x coordinate

# Advantages of this solution

- We do not need to modify the original code (i.e. the Shape hierarchy should not be modified)
- The code is polymorphic to the right level (no extra downcast is necessary)
- Can be applied not only to containers but also to any function

## Inlines

- It is possible to define the members of a template class later on
  - Must be preceded by keyword `template`
  - Remember: it must go in the `hpp` file!

```
template<class T>
class Array {
    enum { size = 100 };
    T A[size];
public:
    T& operator[](int index);
};

template<class T>
T& Array<T>::operator[](int index) {
    require(index >= 0 && index < size,
        "Index out of range");
    return A[index];
}

int main() {
    Array<float> fa;
    fa[0] = 1.414;
}
```

# Parameters

- A template can have any number of parameters
- A parameter can be:
  - a class, or any predefined type
  - a function
  - a constant value (a number, a pointer, etc.)

```
template<T, int sz>
class Buffer {
    T v[sz];
    int size_;
public:
    Buffer() : size_(i) {}
};
...
Buffer<char, 127> cbuf;
Buffer<Record, 8> rbuf;
int x = 16;
Buffer<char, x> ebuf; // error!
```

# Default values

- Some parameter can have default value

```
template<class T, class Allocator = allocator<T> >
class vector;
```

# Templates of templates

- The third type of parameter a template can accept is another class template

```
template<class T>
class Array {
    ...
};

template<class T, template<class> class Seq>
class Container {
    Seq<T> seq;
public:
    void append(const T& t) { seq.push_back(t); }
    T* begin() { return seq.begin(); }
    T* end() { return seq.end(); }
};

int main() {
    Container<int, Array> container;
    container.append(1);
    container.append(2);
    int* p = container.begin();
    while(p != container.end())
        cout << *p++ << endl;
}
```

## Using standard containers

- If the container class is well-written, it is possible to use any container inside

```
template<class T, template<class U, class = allocator<U> >
        class Seq>
class Container {
    Seq<T> seq; // Default of allocator<T> applied implicitly
public:
    void push_back(const T& t) { seq.push_back(t); }
    typename Seq<T>::iterator begin() { return seq.begin(); }
    typename Seq<T>::iterator end() { return seq.end(); }
};

int main() {
    // Use a vector
    Container<int, vector> vContainer;
    vContainer.push_back(1);
    vContainer.push_back(2);
    for(vector<int>::iterator p = vContainer.begin();
        p != vContainer.end(); ++p) {
        cout << *p << endl;
    }
    // Use a list
    Container<int, list> lContainer;
    lContainer.push_back(3);
    lContainer.push_back(4);
    for(list<int>::iterator p2 = lContainer.begin();
        p2 != lContainer.end(); ++p2) {
        cout << *p2 << endl;
    }
}
```

# The typename keyword

- The typename keyword is needed when we want to specify that an identifier is a type

```
template<class T> class X {  
    typename T::id i;    // Without typename, it is an error:  
public:  
    void f() { i.g(); }  
};  
  
class Y {  
public:  
    class id {  
    public:  
        void g() {}  
    };  
};  
  
int main() {  
    X<Y> xy;  
    xy.f();  
}
```

## General rule

- if a type referred to inside template code is qualified by a template type parameter, you must use the typename keyword as a prefix,
- unless it appears in a base class specification or initializer list in the same scope (in which case you must not).

# Usage

- The typical example of usage is for iterators

```
template<class T, template<class U, class = allocator<U> >
        class Seq>
void printSeq(Seq<T>& seq) {
    for(typename Seq<T>::iterator b = seq.begin();
        b != seq.end();)
        cout << *b++ << endl;
}

int main() {
    // Process a vector
    vector<int> v;
    v.push_back(1);
    v.push_back(2);
    printSeq(v);
    // Process a list
    list<int> lst;
    lst.push_back(3);
    lst.push_back(4);
    printSeq(lst);
}
```

## Making a member template

- An example for the complex class

```
template<typename T> class complex {
public:
    template<class X> complex(const complex<X>&);
    ...
};

complex<float> z(1, 2);
complex<double> w(z);
```

- In the declaration of w, the complex template parameter T is double and X is float. Member templates make this kind of flexible conversion easy.

## Another example

```
int data[5] = { 1, 2, 3, 4, 5 };
vector<int> v1(data, data+5);
vector<double> v2(v1.begin(), v1.end());
```

- As long as the elements in v1 are assignment-compatible with the elements in v2 (as double and int are here), all is well.
- The vector class template has the following member template constructor:

```
template<class InputIterator>
vector(InputIterator first, InputIterator last,
       const Allocator& = Allocator());
```

- InputIterator is interpreted as vector<int>::iterator

## Another example

```
template<class T> class Outer {
public:
    template<class R> class Inner {
    public:
        void f();
    };
};

template<class T> template<class R>
void Outer<T>::Inner<R>::f() {
    cout << "Outer == " << typeid(T).name() << endl;
    cout << "Inner == " << typeid(R).name() << endl;
    cout << "Full Inner == " << typeid(*this).name() << endl;
}

int main() {
    Outer<int>::Inner<bool> inner;
    inner.f();
}
```

- Member template functions cannot be declared virtual.
  - Current compiler technology expects to be able to determine the size of a class's virtual function table when the class is parsed.
  - Allowing virtual member template functions would require knowing all calls to such member functions everywhere in the program ahead of time.
  - This is not feasible, especially for multi-file projects.

## Function templates

- The standard template library defines many function templates in algorithm
  - sort, find, accumulate, fill, binary\_search, copy, etc.
- An example:

```
#include <algorithm>
...
int i, j;
...
int z = min<int>(i, j);
```

- Type can be deduced by the compiler
- But the compiler is smart up to a certain limit ...

```
int z = min(x, j); // x is a double, error, not the same types
int z = min<double>(x, j); // this one works fine
```

```
template<typename T, typename U>
const T& min(const T& a, const U& b) {
    return (a < b) ? a : b;
}
```

- The problem is: which return value is the most correct? T or U?
- If the return type of a function template is an independent template parameter, you must always specify its type explicitly when you call it, since there is no argument from which to deduce it.

## Example

```
template<typename T> T fromString(const std::string& s) {
    std::istringstream is(s);
    T t;
    is >> t;
    return t;
}
template<typename T> std::string toString(const T& t) {
    std::ostringstream s;
    s << t;
    return s.str();
}
int main() {
    int i = 1234;
    cout << "i == \"\" << toString(i) << \"\" << endl;
    float x = 567.89;
    cout << "x == \"\" << toString(x) << \"\" << endl;
    complex<float> c(1.0, 2.0);
    cout << "c == \"\" << toString(c) << \"\" << endl;
    cout << endl;

    i = fromString<int>(string("1234"));
    cout << "i == \"\" << i << endl;
    x = fromString<float>(string("567.89"));
    cout << "x == \"\" << x << endl;
    c = fromString<complex<float>>(string("(1.0,2.0)"));
    cout << "c == \"\" << c << endl;
}
```