Design Patterns, Metaprogramming and OOP in C and Fortran

Dr. Axel Kohlmeyer

Associate Dean for Scientific Computing, CST
Associate Director, Institute for Comput. Molecular Science
Assistant Vice President for High-Performance Computing

a.kohlmeyer@temple.edu

TEMPLE UNIVERSITY®

TEMPLE UNIVERSITY®

Templates

- Generic functions and classes with properties like types or value deferred until instantiation
- Type safe way to avoid redundant source code
- Particularly useful for frequently recurring tasks -> containers, algorithms, “design patterns”
- Standard Template Library (STL) is part of C++, BOOST is another popular C++ template library
- Practical problem: where to place the compiled implementation of the class “body”?
  => write templates as header-only classes

Template Example 1

- Many C/C++ codes use this macro:

```
#define MAX(a,b) ((a) > (b)) ? (a) : (b)
```

- problematic if applied to an expression with side effect (a or b may be executed twice)
- Alternative with template function:

```
template<typename T> T MAX(T a, T b)
 {return (a > b) ? a : b;}
```

- Better syntax and type checking, consistent behavior regarding side effects
- Downside: “a” and “b” have to be of same type

Template Example 2

```
template<typename T> class Stack
{
    Stack() { top = 0; }
    void push(T val) { data[top++] = val; }
    const T &pop()   { return data[--top]; }
    int size() const { return top; }
protected:
    int top;
    T  data[100];
};
    ...
Stack<int> a;
Stack<double> b;
Stack<Stack<int> > c;
```

Template Example 3

- Scientific program packages often contain very similar subroutines that differ in small details
- Using a template function helps to hoist the (invariant!) if to a higher level for more speed

```
template <int FLAG> void compute () {
    for (int i=0; i < imax; ++i) {
        ...
        if (FLAG) sum += data;
        else sum -= data;
}}
if (flag) compute<1>();
else compute<0>();
```

Value of FLAG is known at compile time, so the compiler can eliminate the if statement.

Design Patterns

- Term was made popular by the book “Design Patterns: Elements of Reusable Object-Oriented Software” (1994)
- Collection of generic solutions of common problems in object oriented software
- Often used in libraries (STL, BOOST)
  -> partially included in C++ standards
- Part of the vocabulary (or jargon) that computer scientists use to better communicate the purpose of a piece of code

Design Patterns: Main categories

Creational Patterns

- Abstract Factory
- Builder
- Factory Method
- Prototype
- Singleton

Structural Patterns

- Adapter
- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Proxy

Behavioral Patterns

- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template
- Visitor

Factory Methods

- Create objects without having to know specific language type
- Circumvent limitations of constructors
  - **No return result**
    only exceptions
  - **Constrained naming**
    e.g. can't have two constructors with same parameter types
  - **Statically bound creation**
    there is no dynamic binding for constructors, you have to know which type you want to instantiate
  - **No virtual constructors**
- Factory methods can range from very simple implementations to complex selection schemes

Template Example 4

- Here is an “Animal Factory”:

```
template <typename T> Animal *creator() {
    return new T;}

typedef AnimalCreator Animal *(*maker)();

std::map<std::string, AnimalCreator> zoo;
zoo[“dog”] = &creator<Dog>;
zoo[“cat”] = &creator<Cat>;

Animal *beast = (*zoo[“cat”])();
beast->say();
```

Template function

Function pointer

Associate function pointer with string

Look up function by string, derefence & execute

Adapter

- Used to make an object of one type compatible to another
- Typical use case:
  - You defined your own types of objects with a certain interface
  - You want to use an external library to manipulate your objects
  - However the interface expected by library is different to the one you used
- Instead of rewriting you code, you can create an Adapter class, which maps one interface to another.

Adapter – Example

```
class ForceComputation {
public:
    virtual void compute_force(Vector3D & force);
};
```

```
class LegacyClass {
public:
    virtual void compute_force(double * force);
};
```

```
class ForceComputationAdapter : public ForceComputation {
    LegacyClass * legacy;
public:
    ForceComputationAdapter(LegacyClass * src) : legacy(src) {
    }

    virtual void compute_force(Vector3D & force) {
       double f[3];
       legacy->compute_force(&f);
       force.x = f[0];
       force.y = f[1];
       force.z = f[2];
    }
};
```

Strategy

- Used to keep parts of a larger implementations replacable
- You define a common interface to do a certain task
- Any class which implements that interface can be used in larger implementation
- Allows you to exchange object of that interface during runtime
- Typical use case:
  - Define a common interface to get data
  - Interface can be implemented by classes which use files, databases, web services, etc.

Strategy - Example

```
class IRandomNumberGenerator {
public:
    double getNextDouble() = 0;
}
```

```
class DiceRoll : public IRandomNumberGenerator {
public:
    double getNextDouble() {
        // guaranteed to be random,
        // determined with a fair dice roll
        return 4;
    }
}
```

```
class MyUncrackableEncryption {
    IRandomNumberGenerator * random;

    void setRandomNumberGenerator(IRandomNumberGenerator * r) {
        random = r;
    }

    void encrypt(char * data, size_t length) {
        double r = random->getNextDouble();
        ...
    }

    void decrypt(char * data, size_t length) {
        ...
    }
}
```

```
MyUncrackableEncryption e;
DiceRoll d;

e.setRandomNumberGenerator(d);
e.encrypt(…)
```

OOP in Other Languages

- Object oriented programming is more of a design concept than a language feature, but a suitable language makes OOP easier
- Key concept is grouping of data and functions
  => need data structures for it
  => conventions have to replace syntax
  => implicit actions must be coded explicitly
  - C: use `struct` to group data,
    function pointers to include methods
  - Fortran (since Fortran 90): modules, derived types

Simple Stack Class in C++

```
class Stack {
  public:
    Stack() {top=-1;}
    void push(int val) {++top; data[top] = val;}
    int pop() {int rv=data[top]; --top; return rv;}
    int size() {return top+1;}
  private:
    int top, data[100];
};

/*************************************************************/
Stack a;
a.push(10); a.push(-2); a.push(30);
while(a.size() > 0) { std::cout << a.pop() << std::endl;}
```

Simple Stack Class in C (1)

```
typedef struct _stack stack_t;
struct _stack {
    int top;
    int data[100];
};

stack_t *create_stack() {
    stack_t *this;
    this = (stack_t *)calloc(1,sizeof(stack_t));
    this->top = 0;
    return this;
}

void delete_stack(stack_t *this) {
    if (this != NULL) free(this); }
```

Simple Stack Class in C (2)

```
void stack_push(stack_t *this, int val)
{
    this->data[this->top] = val;
    ++(this->top);
}

int  stack_pop(stack_t *this) {
    --(this->top);
    return this->data[this->top];
}

int  stack_size(stack_t *this) {
    return this->top;
}
```

Simple Stack Class in C (3)

```
int main(int argc, char **argv) {
    stack_t *a;

    a = create_stack();

    stack_push(a,10);
    stack_push(a,1);

    while (stack_size(a) > 0)
        printf("stack a: %4d\n", stack_pop(a));

    delete_stack(a);
    return 0;
}
```

Improved Stack Class in C (1)

```
typedef struct _stack stack_t;
struct _stack {
    int top;
    int data[100];
    void (*push)(stack_t *, int);
    int  (*pop) (stack_t *);
};

stack_t *create_stack() {
    stack_t *this;
    this = (stack_t *)calloc(1,sizeof(stack_t));
    this->top = 0;
    this->push = &stack_push;
    this->pop = &stack_pop;
    return this;}
```

Can be static functions
=> less namespace clashes

Improved Stack Class in C (2)

```
int main(int argc, char **argv)
{
    stack_t *a;

    a = create_stack();

    a->push(a,10);
    a->push(a,1);

    while (a->size(a) > 0)
        printf("stack a: %4d\n", a->pop(a));

    delete_stack(a);
    return 0;
}
```

Stack Class in Fortran 90

```
MODULE stackmgr
  TYPE stack
     INTEGER  :: top
     INTEGER  :: vals(100)
  END TYPE stack
CONTAINS

  SUBROUTINE init(this)
    TYPE (stack), INTENT(inout) :: this
    this%top = 1
  END SUBROUTINE init

  SUBROUTINE push(this,val)
    TYPE (stack), INTENT(inout) :: this
    INTEGER, intent(in) :: val
    this%vals(this%top) = val
    this%top = this%top + 1
  END SUBROUTINE push

  INTEGER FUNCTION pop(this)
    TYPE (stack), INTENT(inout) :: this
    this%top = this%top - 1
    pop = this%vals(this%top)
  END FUNCTION pop
  INTEGER FUNCTION size(this)
    TYPE (stack), INTENT(in) :: this
    size = this%top - 1
  END FUNCTION size
END MODULE stackmgr

PROGRAM teststack
  USE stackmgr
  TYPE (stack) :: a;
  CALL init(a)
  CALL push(a,10);
  DO WHILE (SIZE(a) > 0)
     PRINT*, "stack a:", pop(a)
  END DO
...
```

Stack Class in Fortran 2008

```
MODULE stackmgr
  TYPE stack
     INTEGER  :: top
     INTEGER  :: vals(100)
  CONTAINS
        PROCEDURE :: push
        PROCEDURE :: pop
        PROCEDURE :: num
        FINAL :: delete_stack
  END TYPE stack
  INTERFACE stack
     MODULE PROCEDURE init
  END INTERFACE

CONTAINS
  TYPE(stack) FUNCTION init()
    init%top = 1
  END FUNCTION init
...
END MODULE stackmgr

PROGRAM teststack
  USE stackmgr

  TYPE (stack) :: a;
  a = stack()

  CALL a%push(10);
  CALL a%push(1);

  DO WHILE (a%num() > 0)
     PRINT*, "stack a:", a%pop()
  END DO

END PROGRAM teststack
```

Destructor subroutine

Constructor,
can be overloaded

TEMPLE UNIVERSITY®