

Domains: Move, Copy & Co

Michael Hanke

School of Engineering Sciences

Program construction in C++ for Scientific Computing



- 1 Introduction
- 2 The Domain Class
- 3 The Move Constructor (C++11)
- 4 Accessing Points in a Grid
- 5 Summary

What Do We Already Have

Introduction

The Domain
ClassThe Move
Constructor
(C++11)Accessing
Points in a
Grid

Summary

- A class hierarchy for constructing curves in 2D
- Tools for constructing classes for new curves

```
class Curvebase {  
    public:  
        Curvebase(double a = 0.0, double b = 1.0) : a_(a),  
            double x(double s); // Coordinates in arc length  
            double y(double s);  
            virtual ~Curvebase();  
  
    ...  
};
```

What Do We Want

A class for describing (discrete) domains — grids:

```
class Domain {  
    public:  
        Domain(Curvebase&, Curvebase&, Curvebase&, Curvebase&);  
        void generate_grid (...);  
        // more members  
    private:  
        Curvebase *sides[4];  
        // more members  
};
```

Wishlist: Domain

The Domain class should be able:

- To handle all four-sided domains (being topologically equivalent to a square);
- To generate a structured grid for any given number of discretization points in ξ, η -directions;
- Allowing to access any grid point;
- Allow for a convenient implementation of differential operators;
- Routines for export and import of grids.

Domain Class Skeleton

```
class Domain {  
    public:  
        Domain(Curvebase&, Curvebase&,  
               Curvebase&, Curvebase&);  
        Domain(const Domain&);  
        Domain& operator=(Domain&);  
        ~Domain();  
        void generate_grid (int m, int n);  
        // more members  
    private:  
        Curvebase *sides[4];  
        double *x_, *y_;  
        int m_, n_;  
        bool check_consistency();  
        // more members  
};
```

Note: The default constructor is not defined!

The Constructor

```
Domain::Domain(Curvebase& s1, Curvebase& s2,  
               Curvebase& s3, Curvebase& s4) {  
    sides[0] = &s1;  
    sides[1] = &s2;  
    sides[2] = &s3;  
    sides[3] = &s4;  
    if (~check_consistency())  
        sides[0] = sides[1] = sides[2] = sides[3] = nullptr;  
    m_ = n_ = 0;  
    x_ = y_ = nullptr;  
}
```

Note: The object `nullptr` has been introduced in C++ 11. Earlier (and in C) it is common to use the macro `NULL` instead.

Grid Generation

```
Domain::grid_generation(int m, int n) {  
    if (m <= 0 || n <= 0) ; // Do something meaningful  
    else {  
        if (m_ > 0) { // There exists already a grid!  
            delete [] x_;  
            delete [] y_;  
        }  
        m_ = m; n_ = n;  
        x_ = new double[m_*n_];  
        y_ = new double[m_*n_];  
        // Fill x_[] and y_[] with values!  
    }  
}
```

Note: I use the dimensions $m_*n_$ instead of $(m_+1)*(n_+1)$.


```
Domain::~~Domain() {  
    if (m_ > 0) {  
        delete [] x_;  
        delete [] y_;  
    }  
}
```

Copy Constructor

- The default copy constructor invokes recursively the default constructors of its members.
- Here, it would do something like:

```
Domain::Domain(const Domain& d)
    m_ = d.m_;
    n_ = d.n_;
    x_ = d.x_;
    y_ = d.y_;
}
```

- What is wrong with this constructor?

Default Copy Constructor

- Consider the function

```
Domain something() {  
    Domain d(...);  
    // Do something  
    return d;  
}
```

- The following sequence is forbidden (and will most likely crash your program)

```
Domain d = something();  
d.grid_generation();
```

- Why?* Shallow copy!

```
Domain::Domain(const Domain& d) :  
    m_(d.m_), n_(d.n_), x_(nullptr), y_(nullptr) {  
    if (m_ > 0) {  
        x_ = new double[m_*n_];  
        y_ = new double[m_*n_];  
        for (int i = 0; i < m_*n_; i++) {  
            x_[i] = d.x_[i];  
            y_[i] = d.y_[i];  
        }  
    }  
}
```

- It may be much more efficient to use `memcpy` instead of the `for` loop!
- It would even be much more efficient to use specialized libraries. (Maybe, the compiler does it for you)

The Copy-Assignment Operator

```

Domain& Domain::operator=(const Domain& d) {
    if (this != &d) {        // Do not copy to itself
        if (m_ == d.m_ && n_ == d.n_)
            for (int i = 0; i < m_*n_; i++) {
                x_[i] = d.x_[i];
                y_[i] = d.y_[i];
            }
        else {
            if (m_ > 0) {
                delete [] x_;
                delete [] y_;
                x_ = y_ = nullptr;
            }
            m_ = d.m_;
            n_ = d.n_;
            if (m_ > 0) {
                x_ = new double[m_*n_];
                y_ = new double[m_*n_];
                for (int i = 0; i < m_*n_; i++) {
                    x_[i] = d.x_[i];
                    y_[i] = d.y_[i];
                }
            }
        }
    }
}

```

Introduction

The Domain
ClassThe Move
Constructor
(C++11)Accessing
Points in a
Grid

Summary

The Problem And Its Solution

- Consider the function

```
Domain something() {  
    Domain tmp(...);  
    // Do something  
    return tmp;  
}
```

- What happens when calling `Domain d = something();`?
 - A temporary object `tmp` will be created by a constructor.
 - The return statement will execute the copy constructor since `tmp` leaves scope.
 - Memory for the arrays `x_` and `y_` will be allocated and the arrays will be copied.
- The latter copy is unnecessary!

Way out: **Move constructor**.

References to rvalues

- The references considered so far can be bound to lvalues.
- For using the move idea, [references to temporary objects](#) are needed. This is realized via references to rvalues.
- An rvalue reference is defined by using “&&” instead of “&”:

```
int i = 42;  
int &r = i;    // lvalue reference  
int &&r2 = i*42; // An expression is an rvalue  
const int &r3 = i*42 // OK  
int &r2 = i*42  // Error!
```

- **Rule:** References to rvalues cannot be bound to lvalues and vice versa.

The Move Contructor

- The signature of a move constructor is

```
class(class&& v) noexcept
```
- A move constructor uses only available resources. So usually, it does not throw any exception. The keyword `noexcept` indicates this. It allows the compiler to generate more efficient code.
- The move constructor does not destroy `v`. So it must leave `v` in a consistent state such that the destructor can succeed cleanly!

Move Constructor: Domain

```
Domain::Domain(Domain&& d) noexcept  
: m_(d.m_), n_(d.n_), x_(d.x_), y_(d.y_) {  
    d.m_ = 0;  
    d.n_ = 0;  
    d.x_ = nullptr;  
    d.y_ = nullptr;  
}
```

A move-assignment operator can be defined analogously:

```
Domain& Domain::operator =(Domain&&) noexcept;
```

Note: When move operations are defined, the corresponding copy operations must be defined explicitly!

Access of Data Points in a 2D Array

- The C++ syntax allows to access array components:

```
array[i][j]
```

- `[]` is a usual C++ operator! So it allows for overloading!

```
Point Domain::operator[](int ind) const  
{ return P(x_[ind],y_[ind]); }
```

- The result is an rvalue! So we can write `P = d[ind]`, but not `d[ind] = P`!
- We would need something like

```
Point& Domain::operator[](int ind)
```

- Design error! Really?

Index Checking

- Up to now, we do not have index checking. If `ind` in `d[ind]` is out of bounds, the program will most probably crash.
- Moreover, it would be convenient to allow for double indexing.
- Solution: Overload the function call operator `()`:

```
Point Domain::operator()(int i, int j) const {  
    if (i < 0 || i >= m_ || j < 0 || j >= n_) {  
        exit(-1);  
    }  
    int ind = i+j*m_;  
    return P(x_[ind],y_[ind]);  
}
```

- Now you have (controlled) access to the rvalue `d(i,j)`.

Summary

- Deep copy, move constructor
- Overloading the index operator
- What comes next:
 - Templates
 - STL