Lightning Talks

ACCU 2012

Bernhard Merkle – Find(ing) Bugs Is Easy

Charles Bailey – The Rant Of Three

Simon Sebright – SharePoint for Thinking Developers

Phil Nash – Why I Do What I Do

Matt Turner - Fluency

Chris Oldwood – Not Only But Also

Frank Birbacher – Style C++for Version Control

Frances Buontempo - TDD

Astrid Byro – A Cry For Help

Didier Verna – Why?

The Rant of Three

Charles Bailey

26th April 2012

The rule of three

The "rule of three" is a principle in writing that suggests that things that come in threes are inherently funnier, more satisfying, or more effective than other numbers of things.

The rule of three

In aviation the rule of three or "3:1 rule of descent" is that 3 miles of travel should be allowed for every 1000 feet descent.

The rule of three

The Rule of Three (also Three-fold Law or Law of Return) is a religious tenet held by some Wiccans. It states that whatever energy a person puts out into the world, be it positive or negative, will be returned to that person three times.

The rule of three

- copy constructor
- copy assignment operator
- destructor

The rule of three

- copy constructor
- move constructor
- copy assignment operator
- move assignment operator
- destructor

The rule of none

You should not provide an user-declared copy constructor, copy assignment operator or destructor (unless you really have to).

The rule of none

You should not provide an user-declared copy constructor, copy assignment operator or destructor (unless you really have to).

```
class None {
    T1 x;
    T2 y;
};
```

The rule of one

If you are writing an interface class you should provide a user-declared destructor.

The rule of one

If you are writing an interface class you should provide a user-declared destructor.

```
class One {
public:
    virtual ~One() {}
    virtual void contract() = 0;
};
```

The other rule of one

If you are writing an immovable entity you may provide a user-declared destructor.

The other rule of one

If you are writing an immovable entity you may provide a user-declared destructor.

```
class One11 {
public:
    ~One11();
    One(const One&) = delete;
    One& operator=(const One&) = delete;
};
```

The rule of two (C++03)

If a class contains a cloneable entity you may provide a copy constructor, copy assignment operator and no desctructor.

The rule of two (C++03)

If a class contains a cloneable entity you may provide a copy constructor, copy assignment operator and no desctructor.

```
class Two {
Two(const Two&);
Two& operator=(const Two&);
private:
std::auto_ptr<Cloneable> member;
};
```

The rule of three

If you are providing a copy constructor, a copy assignment operator and a destructor you may be doing something wrong.

The rule of three

If you are providing a copy constructor, a copy assignment operator and a destructor you may be doing something wrong.

```
class Three {
    Three(const Three&);
    Three& operator=(const Three&);
    ~Three();
private:
RawResource* member;
};
```

The rule of two and a half

If you want to make your class movable but not copyable and it isn't inherently movable and not copyable for any other reason then you may provide a user-declared move constructor, move assignment operator and optionally a destructor.

The rule of two and a half

If you want to make your class movable but not copyable and it isn't inherently movable and not copyable for any other reason then you may provide a user-declared move constructor, move assignment operator and optionally a destructor.

```
class Three11 {
    Three11(Three11&&);
    Three11& operator=(Three11&&);
    ~Three11();
};
```

There is no rule of five.

There is no substitute for considering on a case by case basis what special member functions you should provide for your class.

Hello world in C++11

C++11, have we completely lost it?

Hello world in C++11

```
%:include<iostream>
int main()<%
    char hw<:??)??<"Hello, world!\n"%>;
    std::cout<<hw;%>
```

C++11, have we completely lost it?

```
auto li = char(std::tolower(c));
```

C++11, have we completely lost it?

```
auto li = static_cast<char>(std::tolower(c));
```

C++11, have we completely lost it?

```
auto li = static_cast<char>(std::tolower(
              static_cast<unsigned char>(c)));
```

C++11, have we completely lost it?

```
auto li {static_cast<char>(std::tolower(c))};
```

C++11, have we completely lost it?

```
char li {std::tolower(c)};
```

C++11, have we completely lost it?

```
char li = std::tolower(c);
```

The End

```
while((c = getchar()) != EOF)
{ /* ... */ }

while(std::istream::traits_type::eq_int_type(
                          (c = cin.get()),
                          std::istream::traits_type::eof()))
{ /* ... */ }
```