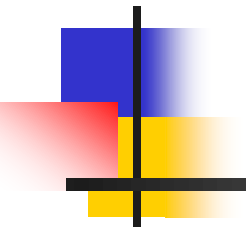
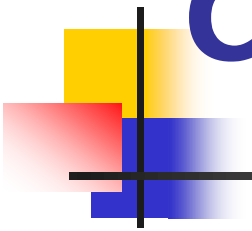


New Feature in C++

C++的新特性

http://www.stroustrup.com/C++11FAQ.html
http://en.cppreference.com/w/cpp/language
C++ Primer Plus, sixth edition, Stephen Prata

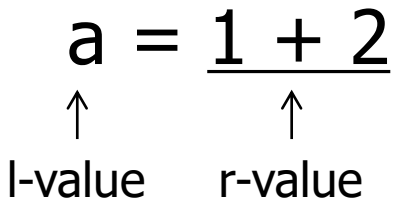
Content



- R-value Reference and Move Constructor
- Extern Templates
- Constant Expressions
- Lambda Function
- Delegating Constructor
- Uniform Initialization
- nullptr

R-value Reference - 1

- L-values: have storage addresses that are programmatically accessible to the running program。




```
class A{};
int main() {
    A a = A();
}
```

r-value

R-value Reference - 2

- In C++, non-const references can bind to l-values and const references can bind to l-values or r-values, but there is nothing that can bind to a non-const r-value.

```
class A{};
A getA(){
    return A();
}
```

```
int main() {
    int a = 1;
    int &ra = a; //OK
    const A &ca = getA();//OK
    A &aa = getA();//ERROR
}
```

R-value Reference - 3

- An r-value reference can bind to an r-value.

```
class A{
    int val;
    void setVal(int v) {
        val = v;
    }
};
A getA(){
    return A();
}
```

```
int main() {
    int a = 1;
    int &ra = a; //OK
    const A &cra = getA();//OK
    A &&aa = getA();//OK
    aa.setVal(2);//OK
    //…
}
```

Move constructor.

```
class MyArray {
    int size;
    int *arr;
public:
    MyArray():size(0),arr(NULL){}
    MyArray(int sz):
        size(sz),arr(new int[sz]) {
        //init array here…
    }
    MyArray(const MyArray &other):
        size(other.size),
        arr(new int[other.size]) {
        for (int i = 0; i < size; i++) {
            arr[i] = other.arr[i];
        }
    }
  MyArray (MyArray &&other):
        size(other.size), arr(other.arr) {
        other.arr = NULL;
  }
~ MyArray() {
        delete[] arr;
    }
}
```

```
MyArray change_aw(const MyArray &other)
{
    MyArray aw(other.get_size());
    //Do some change to aw.
    //….
    return aw;
}

int main() {
    MyArray myArr(5);
    MyArray myArr2 = change_aw(myArr);
}
```

copy constructor

copy constructor

copy constructor

move constructor

copy constructor

Move assignment.

```
class MyArray {
public:
    //...
    MyArray &operator=(const
        MyArray &other) {
        if (this == &other)
            return *this;
        if (arr) {
            delete[] arr;
            arr = NULL;
        }
        size = other.size;
        memcpy(arr, other.arr, size *
            sizeof(int));
        return *this;
    }
    MyArray &operator=(ArrayWrapper
        &&other) {
        size = other.size;
        arr = other.arr;
        other.arr = NULL;
        return *this;
    }
}
```

```
int main() {
    MyArray myArr;
    myArr = MyArr(5);
}
```

Extern Templates

- Avoid of unnecessary instantiation.

```
//myfunc.h
template<typename T>
    void myfunc(T t){}
```

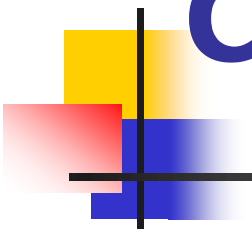
```
//test.cpp
#include "myfunc.h"
int foo(int a){
    myfunc(1);
    return 1;
}
```

```
//main.cpp
#include "myfunc.h"

/*Tell compiler: this instance has been
instantiated in another module!*/
extern template void myfunc<int>(int);


int main() {
    myfunc(1);
}
```

Constant Expressions



- provides more general constant expressions
- allows constant expressions involving user-defined types
- provides a way to guarantee that an initialization is done at compile time

Example 1

```
enum Flags { GOOD=0, FAIL=1, BAD=2, EOF=3 };

constexpr  int operator| (Flags f1, Flags f2)  { return Flags(int(f1)|int(f2)); }


void f(Flags x) {
    switch (x) {
        case BAD: /* ... */break;
        case EOF: /* ... */ break;
        case BAD|EOF: /* ... */ break;      //OK now return value is not const
        default: /* ... */ break;
    }
}
```

error:

```
void f(Flags x) {
    switch (x) {
        case bad_c(): /* ... */break;
        case eof_c(): /* ... */ break;
        case be_c(): /* ... */ break;
        default: /* ... */ break;
    }
}
```

```
constexpr int bad_c();
constexpr int eof_c();
constexpr int be_c();
```

Example 2

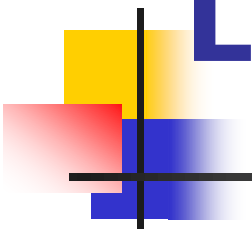
```
struct Point {
    int x,y;
    constexpr Point(int xx, int yy) : x(xx), y(yy) { }
};


int main() {
    constexpr Point origo(0,0);
    constexpr int z = origo.x;

    constexpr Point a[] = {Point(0,0), Point(1,1), Point(2,2) };
    constexpr int x = a[1].x; // x becomes 1
}
```

All evaluation can be done at compile time. Hence runtime efficiency is raised.

Lambda Function



- Also names as Lambda Expression.
- A mechanism for specifying a function object

Example 1 bool cmpInt(int a, int b) {return a < b;}

```
class CmpInt {
    bool operator()(const int a, const int b) const {
        return a < b;
    }
};

int main() {
    std::vector<int> items { 4, 3, 1, 2 };
    std::sort(items.begin(), items.end(), cmpInt ); //Function Pointer
    std::sort(items.begin(), items.end(),  CmpInt()); //Function Object (Functor)
    std::sort(items.begin(), items.end(),
          [](int a, int b) { return a < b; } //Lambda Function
    );
    return 0;
}
```

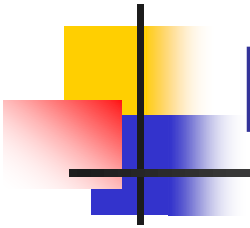
```
template <class RandomAccessIterator, class Compare>
  void sort (RandomAccessIterator first, RandomAccessIterator last, Compare comp) {
    //...
    if ( comp(*it1, *it2) )
    //...
}
```

```
std::function<bool(int, int)>f1(cmpInt);
std::function<bool(int, int)> f2(CmpInt);
std::function<bool(int, int)> f3([](int a, int b) { return a < b;} );
```

Example 2

```
vector<string> str_filter(vector<string> &vec, function<bool(string &)> matched){
    vector<string> result;
    for (string tmp : vec) {
        if (matched(tmp))
            result.push_back(tmp);
    }
    return result;
}

int main(){
    vector<string> vec = {"www.baidu.com", "www.kernel.org", "www.google.com"};
    string pattern = ".com";
    vector<string> filterd = str_filter(vec,
        [&](string &str) {
        if (str.find(pattern) != string::npos)
            return true;
        return false;
    });
}
```



Lambda capture

[]	Capture nothing
[&]	Capture any referenced variable by reference
[=]	Capture any referenced variable by making a copy
[=, &foo]	Capture any referenced variable by making a copy, but capture variable foo by reference
[bar]	Capture bar by making a copy; don't copy anything else

Delegating Constructor

```
#define MAX 256
class X {
    int a;
    void validate(int x) { if (0<x && x<=MAX) a=x; else throw bad_X(x); }
public:
    X(int x) { validate(x); }
    X() { validate(42); }
    // ...
};
```

```
class X {
    int a;
public:
    X(int x) { if (0<x && x<=max) a=x; else throw bad_X(x); }
    X() :X(42) { }
    // ...
};
```

X(int x = 42) ?

Uniform Initialization

```
//Old style initialization
vector<int> vec;
vec.push_back(1);
//…
```

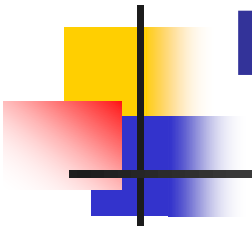
```
int arr[] = {1, 2, 3}; //OK
vector<int> vec = {1, 2, 3};  ?
A a= {1, 2, 3};        ?
```

```
//New style initialization
vector<int> vec = {1, 2, 3};
//Compiler will translate {} as initializer_list<int>
```

```
template class vector<T> {
    //..
    vector(initializer_list<T> list) {
        for (auto it = list.begin(); it != list.end(); ++it)
            push_back(*it);
    }
};
```

```
class A{
    int x, y, z;
    //Default generated by compiler
    A(initializer_list<int> list) {
        auto it = list.begin();
        x = *it++;
        y = *it++;
        z = *it;
    }
};


//Uniform Initialization achieved!
int arr[] = {1, 2, 3};
vector<int> vec = {1, 2, 3};
A  a = {1, 2, 3};
```



nullptr

- **nullptr** is a literal denoting the null pointer

```
void f(int);
void f(char*);
```

f(0); ?

```
f(0);          // call f(int)
f(nullptr);    // call f(char*)
```

```
f(NULL);    // call f(int)
```