

C++11

Alex Sinyakov,
Software Engineer at [AMC Bridge](#)
Twitter: @innochenti
E-mail: innochenti@gmail.com

PDF Slides: <http://j.mp/cpp11ref>

null pointer constant

C++03

```
void foo(char*);
```

```
void foo(int);
```

```
foo(NULL);
```

null pointer constant

C++03	C++11
<pre>void foo(char*); void foo(int); foo(NULL); //calls second foo</pre>	<pre>void foo(char*); void foo(int); foo(nullptr); //calls first foo</pre>

standard types

C++03

`sizeof(int) == ?`

`sizeof(char) == 1 byte(== ? bits)`

`sizeof(char) <= sizeof(short) <=`
`sizeof(int) <= sizeof(long)`

standard types

C++03	C++11
<code>sizeof(int) == ?</code> <code>sizeof(char) == 1 byte(== ? bits)</code> <code>sizeof(char) <= sizeof(short) <=</code> <code>sizeof(int) <= sizeof(long)</code>	<code>int8_t</code> <code>uint8_t</code> <code>int16_t</code> <code>uint16_t</code> <code>int32_t</code> <code>uint32_t</code> <code>int64_t</code> <code>uint64_t</code>

raw string literals

C++03	C++11
<pre>string test="C:\\A\\B\\C\\D\\file1.txt"; cout << test << endl;</pre>	<pre>string test=R"(C:\A\B\C\D\file1.txt)"; cout << test << endl;</pre>
C:\A\B\C\D\file1.txt	C:\A\B\C\D\file1.txt
<pre>string test; test = "First Line.\nSecond line.\nThird Line.\n"; cout << test << endl;</pre>	<pre>string test; test = R"(First Line.\nSecond line.\nThird Line.\n)"; cout << test << endl;</pre>
First Line. Second line. Third Line.	First Line.\nSecond line.\nThird Line.\n
	<pre>string test = R"(First Line. Second line. Third Line.)"; cout << test << endl;</pre>
	First Line. Second line. Third Line.

in-class member initializers

C++03

```
class A
{
public:
    A(): a(4), b(2),
        h("text1"), s("text2") {}
    A(int in_a) : a(in_a), b(2),
        h("text1"), s("text2") {}
    A(C c) : a(4), b(2),
        h("text1"), s("text2") {}
private:
    int a;
    int b;
    string h;
    string s;
};
```

C++11

```
class A
{
public:
    A() {}
    A(int in_a) : a(in_a) {}
    A(C c) {}
private:
    int a = 4;
    int b = 2;
    string h = "text1";
    string s = "text2";
};
```

delegating constructors

C++03

```
class A
{
    int a;
    void validate(int x)
    {
        if (0<x && x<=42) a=x; else throw bad_A(x);
    }
public:
    A(int x) { validate(x); }
    A() { validate(42); }
    A(string s)
    {
        int x = stoi(s);
        validate(x);
    }
};
```

C++11

```
class A
{
    int a;
public:
    A(int x)
    {
        if (0<x && x<=42) a=x; else throw bad_A(x);
    }
    A() : A(42){ }
    A(string s) : A(stoi(s)){ }
};
```


override

C++03

```
struct Base
{
    virtual void some_func(float);
};
```

```
struct Derived : Base
{
    virtual void some_func(int);
    //warning
};
```

C++11

```
struct Base
{
    virtual void some_func(float);
};
```

```
struct Derived : Base
{
    void some_func(int) override;
    //error
};
```

final

C++11

```
struct Base1 final {};  
struct Derived1 : Base1{  
    //error  
};
```

```
struct Base2{  
    virtual void f() final;  
};
```

```
struct Derived2 : Base2 {  
    void f(); //error  
};
```

Java

```
final class Base1 {}  
class Derived1 extends Base1 {  
    //error  
}
```

```
class Base2 {  
    public final void f(){};  
}
```

```
class Derived2 extends Base2 {  
    public void f(){}; //error  
}
```

static_assert

C++11

```
template<class T>
void f(T v){
    static_assert(sizeof(v) == 4, "v must have size of 4 bytes");
    //do something with v
}

void g(){
    int64_t v; // 8 bytes
    f(v);
}
```

vs2010/2012 output:

```
1>d:\main.cpp(5): error C2338: v must have size of 4 bytes
```

type traits

C++11	Output
<pre>#include <type_traits> #include <iostream> using namespace std; struct A { }; struct B { virtual void f(){} }; struct C : B {}; int main() { cout << "int:" << has_virtual_destructor<int>::value << endl; cout << "int:" << is_polymorphic<int>::value << endl; cout << "A: " << is_polymorphic<A>::value << endl; cout << "B: " << is_polymorphic::value << endl; cout << "C: " << is_polymorphic<C>::value << endl; typedef int mytype[][24][60]; cout << "(0 dim.): " << extent<mytype,0>::value << endl; cout << "(1 dim.): " << extent<mytype,1>::value << endl; cout << "(2 dim.): " << extent<mytype,2>::value << endl; return 0; }</pre>	<pre>int:0 int:0 A: 0 B: 1 C: 1 (0st dim.): 0 (1st dim.): 24 (2st dim.): 60</pre>

auto

C++03

```
map<string,string>::iterator it = m.begin();  
double const param = config["param"];  
singleton& s = singleton::instance();
```

C++11

```
auto it = m.begin();  
auto const param = config["param"];  
auto& s = singleton::instance();
```

Prefer using **auto** in the following cases:

```
auto p = new T();
```

Here is T in the expression. No need to repeat it again.

```
auto p = make_shared<T>(arg1);
```

The same as above.

```
auto my_lambda = [](){};
```

If you need to store lambda you may use **auto** or **std::function**

```
auto it = m.begin();
```

Instead of: `map<string,list<int>::iterator>::const_iterator it = m.cbegin();`

<http://programmers.stackexchange.com/questions/180216/does-auto-make-c-code-harder-to-understand>

decltype

C++11

```
int main(){
    int i = 4;
    const int j = 6;
    const int& k = i;
    int a[5];
    int *p;

    int var1;
    int var2;
    int var3;
    int& var4 = i;
    //
    const int var5 = 1;
    const int& var6 = j;
    int var7[5];
    int& var8 = i;
    int& var9 = i;

    return 0;
}
```



C++11

```
int main(){
    int i = 4;
    const int j = 6;
    const int& k = i;
    int a[5];
    int *p;
    //decltype is an operator for querying the type of an expression.
    //similarly to the sizeof operator, the operand of decltype is unevaluated.
    decltype(i) var1;
    decltype(1) var2;
    decltype(2+3) var3;
    decltype(i=1) var4 = i; //there is no assignment i to 1
    // i == 4 as before
    decltype(j) var5 = 1;
    decltype(k) var6 = j;
    decltype(a) var7;
    decltype(a[3]) var8 = i;
    decltype(*p) var9 = i;

    return 0;
}
```

suffix return type syntax

C++11

```
template<class T, class U>  
??? add(T x, U y)  
//return type???  
{  
    return x+y;  
}
```

suffix return type syntax

C++11

```
template<class T, class U>  
??? add(T x, U y)  
//return type???  
{  
    return x+y;  
}
```

```
template<class T, class U>  
decltype(x+y) add(T x, U y)  
//scope problem  
{  
    return x+y;  
}
```


suffix return type syntax

C++11

```
template<class T, class U>
??? add(T x, U y)
//return type???
{
    return x+y;
}
```

```
template<class T, class U>
decltype(x+y) add(T x, U y)
//scope problem
{
    return x+y;
}
```

```
template<class T, class U>
decltype(*(T*)(0)+*(U*)(0)) add(T x, U y)
// ugly!
{
    return x+y;
}
```

suffix return type syntax

C++11

```
template<class T, class U>
??? add(T x, U y)
//return type???
{
    return x+y;
}
```

```
template<class T, class U>
decltype(x+y) add(T x, U y)
//scope problem
{
    return x+y;
}
```

```
template<class T, class U>
decltype(*(T*)(0)+*(U*)(0)) add(T x, U y)
// ugly!
{
    return x+y;
}
```

```
template<class T, class U>
auto add(T x, U y) -> decltype(x+y)
{
    return x+y;
}
```

suffix return type syntax

C++03

```
struct LinkedList
{
    struct Link { /* ... */ };
    Link* erase(Link* p);
    // ...
};

LinkedList::Link* LinkedList::erase(Link* p)
{ /* ... */ }
```

C++11

```
struct LinkedList
{
    struct Link { /* ... */ };
    Link* erase(Link* p);
    // ...
};

auto LinkedList::erase(Link* p) -> Link*
{ /* ... */ }
```

std::function

C++11

```
int sum(int a, int b) { return a + b; }
```

```
function<int (int, int)> fsum = &sum;
```

```
fsum(4,2);
```

std::function

C++11

```
struct Foo
{
    void f(int i){}
};
```

```
function<void(Foo&, int)> fmember = mem_fn(&Foo::f);
```

```
Foo foo;
fmember(foo, 42);
```

std::function

C++11

```
struct Foo
{
    void f(int i){}
};
```

```
Foo foo;
```

```
function<void(int)> fmember = bind(&Foo::f, foo, _1);
```

```
fmember(42);
```

std::bind

C++11	output
<pre>float div(float a, float b){ return a/b; } cout << "6/1" << div(6,1); cout << "6/2" << div(6,2); cout << "6/3" << div(6,3);</pre>	<pre>6/1 = 6 6/2 = 3 6/3 = 2</pre>
<pre>function<float(float, float)> inv_div = bind(div, _2, _1); cout << "1/6" << inv_div(6,1); cout << "2/6" << inv_div(6,2); cout << "3/6" << inv_div(6,3);</pre>	<pre>1/6 = 0.166 2/6 = 0.333 3/6 = 0.5</pre>
<pre>function<float(float)> div_by_6 = bind(div, _1, 6); cout << "1/6" << div_by_6 (1); cout << "2/6" << div_by_6 (2); cout << "3/6" << div_by_6 (3);</pre>	<pre>1/6 = 0.166 2/6 = 0.333 3/6 = 0.5</pre>

std::bind

C++11

//Practical usage

```
linear_congruential_engine<uint64_t, 1103545, 123, 21478> generator(1127590);  
uniform_int_distribution<int> distribution(1,6);  
int rnd = distribution(generator);
```

//Let's make things a little bit easier:

```
auto dice = bind( distribution, generator );  
int rnd = dice()+dice()+dice();
```


function objects

C++11(deprecated binders and adaptors)

unary_function,
binary_function,
ptr_fun,
pointer_to_unary_function,
pointer_to_binary_function,
mem_fun,
mem_fun_t,
mem_fun1_t
const_mem_fun_t
const_mem_fun1_t
mem_fun_ref
mem_fun_ref_t
mem_fun1_ref_t
const_mem_fun_ref_t
const_mem_fun1_ref_t
binder1st
binder2nd
bind1st
bind2nd

C++11

Function wrappers

function
mem_fn
bad_function_call

Bind

bind
is_bind_expression
is_placeholder
_1, _2, _3, ...

Reference wrappers

reference_wrapper
ref
cref

lambdas

C++03

```
struct functor
{
    int &a;
    functor(int& _a)
        : a(_a)
    {
    }
    bool operator()(int x) const
    {
        return a == x;
    }
};

int a = 42;
count_if(v.begin(), v.end(), functor(a));
```

lambdas

C++03

```
struct functor
{
    int &a;
    functor(int& _a)
        : a(_a)
    {
    }
    bool operator()(int x) const
    {
        return a == x;
    }
};

int a = 42;
count_if(v.begin(), v.end(), functor(a));
```

C++11

```
int a = 42;
count_if(v.begin(), v.end(), [&a](int x){ return
x == a;});
```

C++14

//possible C++14 lambdas
count_if(v.begin(),v.end(),[&a](auto x)x == a);

<http://isocpp.org/blog/2012/12/an-implementation-of-generic-lambdas-request-for-feedback-faisal-vali>

lambdas/closures

C++11	test scope	lambda scope
<pre>void test() { int x = 4; int y = 5; [&]() { x = 2; y = 2; }(); [=]() mutable { x = 3; y = 5; }(); [=, &x]() mutable { x = 7; y = 9; }(); }</pre>	<pre>x=4 y=5 x=2 y=2 x=2 y=2 x=7 y=2</pre>	<pre>x=2 y=2 x=3 y=5 x=7 y=9</pre>
<pre>void test() { int x = 4; int y = 5; auto z = [=]() mutable { x = 3; ++y; int w = x + y; return w; }; z(); z(); z(); }</pre>	<pre>x=4 y=5 x=4 y=5 x=4 y=5 x=4 y=5</pre>	<pre>//closure //x,y lives inside z x=3 y=6 w=9 x=3 y=7 w=10 x=3 y=8 w=11</pre>

recursive lambdas

```
function<int(int)> f = [&f](int n)
{
    return n <= 1 ? 1 : n * f(n - 1);
};
```

```
int x = f(4); //x = 24
```

std::tuple

C++11	python
<pre>tuple<int,float,string> t(1,2.f,"text"); int x = get<0>(t); float y = get<1>(t); string z = get<2>(t);</pre>	<pre>t = (1,2.0,'text') x = t[0] y = t[1] z = t[2]</pre>
<pre>int myint; char mychar; tuple<int,float,char> mytuple; // packing values into tuple mytuple = make_tuple (10, 2.6, 'a'); // unpacking tuple into variables tie(myint, ignore, mychar) = mytuple;</pre>	<pre>// packing values into tuple mytuple = (10, 2.6, 'a') // unpacking tuple into variables myint, _, mychar = mytuple</pre>
<pre>int a = 5; int b = 6; tie(b, a) = make_tuple(a, b);</pre>	<pre>a = 5 b = 6 b,a = a,b</pre>

std::tuple/std::tie(for lexicographical comparison)

C++03

```
struct Student
{
    string name;
    int classId;
    int numPassedExams;

    bool operator<(const Student& rhs) const
    {
        if(name < rhs.name)
            return true;

        if(name == rhs.name)
        {
            if(classId < rhs.classId)
                return true;

            if(classId == rhs.classId)
                return numPassedExams < rhs.numPassedExams;
        }

        return false;
    }
};

set<Student> students;
```

C++11

```
struct Student
{
    string name;
    int classId;
    int numPassedExams;

    bool operator<(const Student& rhs) const
    {
        return tie(name, classId, numPassedExams) <
            tie(rhs.name, rhs.classId, rhs.numPassedExams);
    }
};

set<Student> students;
```

Uniform Initialization and std::initializer_list

C++03	C++11
<pre>int a[] = { 1, 2, 3, 4, 5 }; vector<int> v; for(int i = 1; i <= 5; ++i) v.push_back(i);</pre>	<pre>int a[] = { 1, 2, 3, 4, 5 }; vector<int> v = { 1, 2, 3, 4, 5 };</pre>
<pre>map<int, string> labels; labels.insert(make_pair(1, "Open")); labels.insert(make_pair(2, "Close")); labels.insert(make_pair(3, "Reboot"));</pre>	<pre>map<int, string> labels { { 1, "Open" }, { 2, "Close" }, { 3, "Reboot" } };</pre>
<pre>Vector3 normalize(const Vector3& v) { float inv_len = 1.f/ length(v); return Vector3(v.x*inv_len, v.y*inv_len, v.z*inv_len); }</pre>	<pre>Vector3 normalize(const Vector3& v) { float inv_len = 1.f/ length(v); return {v.x*inv_len, v.y*inv_len, v.z*inv_len}; }</pre>
<pre>Vector3 x = normalize(Vector3(2,5,9)); Vector3 y(4,2,1);</pre>	<pre>Vector3 x = normalize({2,5,9}); Vector3 y{4,2,1};</pre>

std::initializer_list

C++11

```
vector<int> v = { 1, 2, 3, 4, 5 }; //How to make this works?
```

```
vector<int> v = { 1, 2, 3, 4, 5 };  
//vector(initializer_list<T> args) is called
```

```
template<class T>  
class vector{  
    vector(initializer_list<T> args)  
    { /*rude, naive implementation to show how ctor  
with initializer_list works*/  
        for(auto it = begin(args); it != end(args); ++it)  
            push_back(*it);  
    }  
    //...  
};
```

```
//what is initializer_list<T> ?
```

initializer_list<T> is a lightweight proxy object that provides access to an array of objects of type T.

A std::initializer_list object is automatically constructed when:

```
vector<int> v{1,2,3,4,5}; //list-initialization
```

```
v = {1,2,3,4,5}; //assignment expression
```

```
f({1,2,3,4,5}); //function call
```

```
for (int x : {1, 2, 3}) //ranged for loop
```

```
    cout << x << endl;
```

Uniform Initialization

C++11

//but wait!!! How then does this work??

```
struct Vector3{  
    float x,y,z;  
    Vector3(float _x, float _y, float _z)  
        : x(_x), y(_y), z(_z){}  
    //I don't see ctor with std::initializer_list!  
};  
  
Vector3 normalize(const Vector3& v){  
    float inv_len = 1.f/ length(v);  
    return {v.x*inv_len, v.y*inv_len, v.z*inv_len};  
}
```

```
Vector3 x = normalize({2,5,9});  
Vector3 y{4,2,1};
```

The answer is:

now you can use `{}` instead of `()`

But what about following case:

```
struct T {  
    T(int,int);  
    T(initializer_list<int>);  
};
```

T foo {10,20}; // calls initializer_list ctor

T bar (10,20); // calls first constructor

Initializer-list constructors **take precedence over other constructors** when the initializer-list constructor syntax is used!

So, be careful! Consider following example:

vector<int> v(5); // v contains five elements {0,0,0,0,0}

vector<int> v{5}; // v contains one element {5}

Uniform Initialization

C++11

Uniform initialization solves many problems:

Narrowing

```
int x = 6.3; //warning!  
int y {6.3}; //error: narrowing  
int z = {6.3}; //error: narrowing  
vector<int> v = { 1, 4.3, 4, 0.6 }; //error: double to int narrowing
```

“The most vexing parse” problem

```
struct B{  
    B(){}  
};  
  
struct A{  
    A(B){}  
    void f(){}  
};  
  
int main(){  
    A a(B()); //this is function declaration!  
    a.f(); //compile error!  
    return 0;  
}
```

```
struct B{  
    B(){}  
};  
  
struct A{  
    A(B){}  
    void f(){}  
};  
  
int main(){  
    A a{B()}; //calls B ctor, then A ctor. Everything is ok.  
    a.f(); //calls A::f  
    return 0;  
}
```

Uniform Initialization and `std::initializer_list`

```
// Don't mix std::initializer_list with auto  
int n;  
auto w(n);    // int  
auto x = n;   // int  
auto y {n};   // std::initializer_list<int>  
auto z = {n}; // std::initializer_list<int>
```

using

C++03

```
typedef int int32_t; // on windows  
typedef void (*Fn)(double);
```

```
template <int U, int V> class Type;
```

```
typedef Type<42,36> ConcreteType;
```

```
template<int V>  
typedef Type<42,V> MyType;  
//error: not legal C++ code
```

```
MyType<36> object;
```

```
template<int V>  
struct meta_type{  
    typedef Type<42, V> type;  
};  
typedef meta_type<36>::type MyType;  
MyType object;
```

using

C++03	C++11
<pre>typedef int int32_t; // on windows typedef void (*Fn)(double);</pre>	<pre>using int32_t = int; // on windows using Fn = void (*)(double);</pre>
<pre>template <int U, int V> class Type;</pre>	<pre>template <int U, int V> class Type;</pre>
<pre>typedef Type<42,36> ConcreteType;</pre>	<pre>using ConcreteType = Type<42,36>;</pre>
<pre>template<int V> struct meta_type{ typedef Type<42, V> type; }; typedef meta_type<36>::type MyType; MyType object;</pre>	<pre>template <int V> using MyType = Type<42, V>; MyType<36> object;</pre>

explicit conversion operators

C++03

```
struct A { A(int){}; };  
void f(A){};
```

```
int main(){  
    A a(1);  
    f(1); //silent implicit cast!  
    return 0;  
}
```

explicit conversion operators

C++03

```
struct A {explicit A(int){}; };  
void f(A){};  
  
int main(){  
    A a(1);  
    f(1); //error: implicit cast!  
    return 0;  
}
```


explicit conversion operators

C++03

```
struct A {  
    A(int) {}  
};  
  
struct B {  
    int m;  
    B(int x) : m(x) {}  
    operator A() { return A(m); }  
};  
  
void f(A){}  
  
int main(){  
    B b(1);  
    A a = b; //silent implicit cast!  
    f(b); //silent implicit cast!  
    return 0;  
}
```

C++11

```
struct A {  
    A(int) {}  
};  
  
struct B {  
    int m;  
    B(int x) : m(x) {}  
    explicit operator A() { return A(m); }  
};  
  
void f(A){}  
  
int main(){  
    B b(1);  
    A a = b; //error: implicit cast!  
    f(b); //error: implicit cast!  
    return 0;  
}
```

explicit conversion operators

C++03

```
struct A {  
    A(int) {}  
};  
  
struct B {  
    int m;  
    B(int x) : m(x) {}  
    operator A() { return A(m); }  
};  
  
void f(A){}  
  
int main(){  
    B b(1);  
    A a = b; //silent implicit cast!  
    f(b); //silent implicit cast!  
    return 0;  
}
```

C++11

```
struct A {  
    A(int) {}  
};  
  
struct B {  
    int m;  
    B(int x) : m(x) {}  
    explicit operator A() { return A(m); }  
};  
  
void f(A){}  
  
int main(){  
    B b(1);  
    A a = static_cast<A>(b);  
    f(static_cast<A>(b));  
    return 0;  
}
```

control of defaults: default and delete

C++11

```
class A
{
    A& operator=(A) = delete; // disallow copying
    A(const A&) = delete;
};
```

```
struct B
{
    B(float); // can initialize with a float
    B(long) = delete; // but not with long
};
```

```
struct C
{
    virtual ~C() = default;
};
```

enum class - scoped and strongly typed enums

C++03

```
enum Alert { green, yellow, red };  
//enum Color{ red, blue };  
//error C2365: 'red' : redefinition
```

```
Alert a = 7; // error (as ever in C++)  
int a2 = red; // ok: Alert->int conversion  
int a3 = Alert::red; // error
```

C++11

```
enum class Alert { green, yellow, red };  
enum class Color : int{ red, blue };
```

```
Alert a = 7; // error (as ever in C++)  
Color c = 7; // error: no int->Color conversion  
int a2 = red; // error  
int a3 = Alert::red; //error  
int a4 = blue; // error: blue not in scope  
int a5 = Color::blue; //error: not Color->int  
conversion  
Color a6 = Color::blue; //ok
```

user-defined literals

C++03	C++11
<pre>123 // int 1.2 // double 1.2F // float 'a' // char 1ULL // unsigned long long</pre>	<pre>1.2_i // imaginary 123.4567891234_df // decimal floating point (IBM) 101010111000101_b // binary 123_s // seconds 123.56_km // not miles! (units) Speed v = 100_km/1_h; int operator "" _km(int val){ return val; }</pre> <p>Practical usage:</p> <p>http://www.codeproject.com/Articles/447922/Application-of-Cplusplus11-User-Defined-Literals-t</p>

Move Semantics

C++03

```
typedef vector<float> Matrix;  
  
//requires already created C  
void Mul(const Matrix& A, const Matrix& B, Matrix& C);  
  
//need to manage lifetime manually using new/delete  
void Mul(const Matrix& A, const Matrix& B, Matrix* C);  
  
//please, don't forget to call delete  
Matrix* operator*(const Matrix& A, const Matrix& B);  
  
//no need to manage lifetime manually, but adds some  
//performance and abstraction penalty  
shared_ptr<Matrix> operator* (const Matrix& A, const  
Matrix& B);
```

Move Semantics

C++03

```
typedef vector<float> Matrix;  
  
//requires already created C  
void Mul(const Matrix& A, const Matrix& B, Matrix& C);  
  
//need to manage lifetime manually using new/delete  
void Mul(const Matrix& A, const Matrix& B, Matrix* C);  
  
//please, don't forget to call delete  
Matrix* operator*(const Matrix& A, const Matrix& B);  
  
//no need to manage lifetime manually, but adds some  
//performance and abstraction penalty  
shared_ptr<Matrix> operator* (const Matrix& A, const  
Matrix& B);
```

C++11

```
typedef vector<float> Matrix;  
  
//Cool syntax, no abstraction or performance  
//penalty! Thanks to move semantics!  
Matrix operator*(const Matrix& A, const Matrix& B);  
  
  
  
  
  
  
  
  
  
Matrix A(10000);  
Matrix B(10000);  
Matrix C = A * B;
```

Move Semantics

C++11

```
typedef vector<float> Matrix;

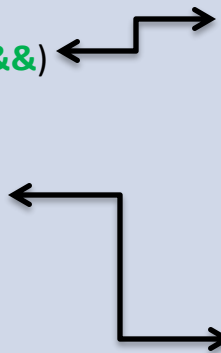
Matrix operator*(const Matrix& A, const Matrix& B);
{
    Matrix ret(A.size()); //ret.data = 0x0028fabcd
    //ret.size = 100000
    //matrix multiplication algorithm
    //
    return ret; //vector<float>(vector<float>&&)
    //C.data = ret.data, C.size = ret.size
    //ret.data = nullptr, ret.size = 0
} //~vector<float>()
//delete ret.data; //”delete nullptr,” is ok.

Matrix A(10000);
Matrix B(10000);
Matrix C = A * B;

//C.data = 0x0028fabcd
//C.size = 100000
```

```
template<class T>
class vector
{
    T* data;
    size_t size;

public:
    vector(vector<T>&& rhs)
        : data(rhs.data)
        , size(rhs.size)
    {
        rhs.data = nullptr;
        rhs.size = 0;
    }
    ~vector()
    {
        delete[] data;
    }
    //...
};
```



Move Semantics

C++03

```
typedef vector<float> BigObj;
void f(BigObj&); //reference to lvalue

//test1
BigObj x = createBigObject(); //BigObj(const BigObj&)

f(x); //BigObj(const BigObj&)
f(createBigObject()); //BigObj(const BigObj&)

//test3
BigObj createBigObject()
{
    BigObj object(100000); //value
    return object; //BigObj(const BigObj&)
} //~BigObj

BigObj x = createBigObject();
```

C++11

```
typedef vector<float> BigObj;
void f(BigObj&&); //reference to rvalue
void f(BigObj&); //reference to lvalue

//test1
BigObj x = createBigObject();

f(x); //BigObj(const BigObj&)
f(createBigObject()); //BigObj(BigObj&&)

//test2
BigObj x = createBigObject();
f(move(x)); // move makes from input value – rvalue.

//test3
BigObj createBigObject()
{
    BigObj object(100000); //prvalue
    return object; //BigObj(BigObj&&)
} //~BigObj

BigObj x = createBigObject();
```

constexpr

C++03

```
template<int N>
struct Fib{
    enum {
        value = Fib<N-1>::value + Fib<N-2>::value
    };
};
```

```
template<> struct Fib<1>{
    enum { value = 1 };
};
```

```
template<> struct Fib<0> {
    enum { value = 0 };
};
```

```
cout << Fib<15>::value;
```

C++11

```
constexpr int Fib(int n)
{
    return n<=2 ? 1 : Fib(n-1)+Fib(n-2);
}
```

```
cout << Fib(15); //compile time
```

```
int a = 15;
cout << Fib(a); //runtime
```

range-for, begin, end

C++03

```
vector<int> v;  
for( vector<int>::iterator i = v.begin(); i !=  
v.end(); ++i )  
    total += *i;  
  
sort( v.begin(), v.end() );  
  
int a[] = {1,2,3,4,5};  
sort( &a[0], &a[0] + sizeof(a)/sizeof(a[0]));
```

C++11

```
vector<int> v;  
for( auto d : v )  
    total += d;  
  
sort( begin(v), end(v) );  
  
int a[] = {1,2,3,4,5};  
sort( begin(a), end(a) );
```

Memory management

(unique_ptr is safe replacement for unsafe deprecated auto_ptr)

C++11

```
unique_ptr<int> p1(new int(42));
unique_ptr<int> p2 = p1; //Compile error. Only "move" operation is possible.
unique_ptr<int> p3 = move(p1); //Transfers ownership. p3 now owns the memory and p1 is nullptr.
p3.reset(); //Deletes the memory.
p1.reset(); //Does nothing.
```

```
unique_ptr<int> createUniqueResource()
{
    unique_ptr<int> ret( new int(42) );
    return ret; //no need to move(ret);
}
```

```
F* OpenFile(char* name);
void CloseFile(F*);
/* custom deleter */
```

```
unique_ptr<F, function<decltype(CloseFile)>> file(OpenFile("text"), CloseFile);
file->read(1024);
```

Memory management

(shared_ptr = ref(+weak) thread safe counter)

```
void test()
{
    shared_ptr<int> p( new int(42) );           ref count = 1, weak count = 0
    {
        shared_ptr<int> x = p;                  ref count = 2, weak count = 0
        {
            shared_ptr<int> y = p;              ref count = 3, weak count = 0
        }                                       ref count = 2, weak count = 0
    }                                           ref count = 1, weak count = 0
    // use weak_ptr to break reference-count cycles
    weak_ptr<int> wp = p;                      ref count = 1, weak count = 1 – note ref count is still 1
    shared_ptr<int> ap = wp.lock();             ref count = 2, weak count = 1
    {
        shared_ptr<int> y = ap;                ref count = 3, weak count = 1
    }                                           ref count = 2, weak count = 1
}                                               ap dtor: ref count = 1, weak count = 1
                                              wp dtor: ref count = 1, weak count = 0
                                              p dtor:  ref count = 0, weak count = 0 - destroy p!
```

Variadic templates

C++03

```
void f();  
template<class T>  
void f(T arg1);  
template<class T, class U>  
void f(T arg1, U arg2);  
template<class T, class U, class Y>  
void f(T arg1, U arg2, Y arg3);  
template<class T, class U, class Y, class Z>  
void f(T arg1, U arg2, Y arg3, Z arg4);  
  
f("test",42,'s',12.f);  
//... till some max N.
```

C++11

```
template <class ...T>  
void f(T... args);
```

```
f("test",42,'s',12.f);
```

Variadic templates

C++11	C++11(call sequence)
<pre>template<class T> void print_list(T value) { cout<<value<<endl; } template<class First, class ...Rest> void print_list(First first, Rest ...rest) { cout<<first<<","; print_list(rest...); } print_list(42,"hello",2.3,'a');</pre>	<pre>print_list(first = 42, ...rest = "hello",2.3,'a') 42 print_list(first = "hello", ...rest = 2.3,'a') hello print_list(first = 2.3, ...rest = 'a') 2.3 print_list(value ='a') //trivial case a</pre> <div>Output</div> <pre>42,hello,2.3,a</pre>

Tuple definition using variadic templates

C++11

```
template<class... Elements>  
class tuple;
```

```
template<>  
class tuple<> {};
```

```
template<class Head, class... Tail>  
class tuple<Head, Tail...> : private tuple<Tail...>  
{  
    Head head;  
    //  
};
```

“LISP-style” definition:
A tuple is either:

- An empty tuple, or
- A pair (head, tail) where head is the first element of the tuple and tail is a tuple containing the rest(...) of the elements.

Variadic templates

C++11

```
template<int... Elements> struct count;
```

```
template<> struct count<>
{
    static const int value = 0;
};
```

```
template<int T, int... Args>
struct count<T, Args...>
{
    static const int value = 1 +
        count<Args...>::value;
};
```

```
//call
int x = count<0,1,2,3,4>::value;
```

Haskell

```
count [] = 0
count (T:Args) = 1 + count Args
```

```
//call
count [0,1,2,3,4]
```

Variadic templates(sizeof... operator)

C++11

```
template<int... Elements> struct count;
```

```
template<> struct count<>
{
    static const int value = 0;
};
```

```
template<int T, int... Args>
struct count<T, Args...>
{
    static const int value = 1 +
        count<Args...>::value;
};
```

```
//call
int x = count<0,1,2,3,4>::value;
```

```
template<int... Elements>
struct count
```

```
{
    static const int value = sizeof...(Elements);
};
```

```
/*
sizeof...() – return the number elements in
a parameter pack
*/
```

```
//call
int x = count<0,1,2,3,4>::value;
```

std::string

Interprets a signed integer value in the string:

```
int    stoi( const std::string& str, size_t *pos = 0, int base = 10 );  
long   stol( const std::string& str, size_t *pos = 0, int base = 10 );  
long long stoll( const std::string& str, size_t *pos = 0, int base = 10 );
```

Interprets an unsigned integer value in the string:

```
unsigned long   stoul( const std::string& str, size_t *pos = 0, int base = 10 );  
unsigned long long stoull( const std::string& str, size_t *pos = 0, int base = 10 );
```

Interprets a floating point value in a string:

```
float    stof( const std::string& str, size_t *pos = 0 );  
double   stod( const std::string& str, size_t *pos = 0 );  
long double stold( const std::string& str, size_t *pos = 0 );
```

Converts a (un)signed/decimal integer to a string/wstring:

```
to_string  
to_wstring
```

std::array

C++03	C++11
<pre>char arr1[] = "xyz"; //'\0' is added to the end int arr2[] = {2112, 90125, 1928};</pre>	<pre>array<char, 3> arr1 = {'x', 'y', 'z'}; array<int, 3> arr2 = {2112, 90125, 1928};</pre>
<pre>int* x = arr2; //ok</pre>	<pre>int* x = arr2; //error x = arr2.data(); //ok</pre>
<pre>cout << sizeof(arr1) - 1 << endl; cout << sizeof(arr2) / sizeof(int) << endl;</pre>	<pre>cout << arr1.size() << endl; cout << arr2.size() << endl;</pre>
<pre>arr2[-42] = 36; //oops</pre>	<pre>arr2.at(-42) = 36; //throws std::out_of_range exception</pre>
<p>http://stackoverflow.com/questions/6111565/now-that-we-have-stdarray-what-uses-are-left-for-c-style-arrays</p>	

std::vector

C++03	C++11
<pre>void c_style_(int* x){} void test(){ vector<int> v; if(!v.empty()) c_style_(&v[0]); if(!v.empty()) c_style_(&v.front()); if(!v.empty()) c_style_(&*v.begin()); }</pre>	<pre>void c_style_(int* x){} void test(){ vector<int> v; c_style_(v.data()); }</pre>
<pre>vector<int> v; v.push_back(1); //capacity = 1 v.reserve(20); //capacity = 20 vector<int>(v).swap(v); //capacity = 1 //very intuitive!</pre>	<pre>vector<int> v; v.push_back(1); //capacity = 1 v.reserve(20); //capacity = 20 v.shrink_to_fit(); //capacity = 1</pre>
<pre>struct Some_type{ Some_type(int _x, int _y, int _z) : x(_x), y(_y), z(_z){} int x,y,z; }; vector<Some_type> v; v.push_back(Some_type(1,2,3));</pre>	<pre>struct Some_type{ Some_type(int _x, int _y, int _z) : x(_x), y(_y), z(_z){} int x,y,z; }; vector<Some_type> v; v.emplace_back(1,2,3);</pre>

STL

`std::regex`

```
bool equals = regex_match("subject", regex("(sub)(.*)") );
```

`std::chrono`

```
auto start = high_resolution_clock::now();  
some_long_computations();  
auto end = high_resolution_clock::now();  
cout<<duration_cast<milliseconds>(end-start).count();
```

`std::ratio`

```
using sum = ratio_add<ratio<1,2>, ratio<2,3>>;  
  
cout << "sum = " << sum::num << "/" << sum::den;  
cout << " (which is: " << ( double(sum::num) / sum::den ) << ")" << endl;
```

Output: sum = 7/6 (which is: 1.166667)

STL

New algorithms:

`std::all_of`, `std::none_of`, `std::any_of`,
`std::find_if_not`, `std::copy_if`, `std::copy_n`,
`std::move`, `std::move_n`, `std::move_backward`,
`std::shuffle`, `std::random_shuffle`,
`std::is_partitioned`, `std::partition_copy`,
`std::partition_point`, `std::is_sorted`,
`std::is_sorted_until`, `std::is_heap_until`,
`std::min_max`, `std::minmax_element`,
`std::is_permutation`, `std::iota`

Threads and memory model

Threading support:

thread, mutex, condition variable,
future/promise, package task

Memory model:

atomic, fence

Difference between `std::thread` and `boost::thread`:

<http://stackoverflow.com/questions/7241993/is-it-smart-to-replace-boostthread-and-boostmutex-with-c11-equivalents>

std::thread

C++11

```
#include <thread>
#include <iostream>

int main()
{
    using namespace std;
    thread t1([](){
        cout << "Hi from
        thread" << endl;});

    t1.join();
    return 0;
}
```

Java

```
public class TestThread {

    public static void main(String[] args) throws
        InterruptedException {
        Thread t1 = new Thread(new Runnable() {
            public void run() {
                System.out.println("Hi from thread");
            }
        });
        t1.start();

        t1.join();
    }
}
```

std::mutex

C++11

Output(may vary)

```
#include <iostream>
#include <thread>
//version without mutex!!!
using namespace std;

void run(size_t n){
    for (size_t i = 0; i < 5; ++i){
        cout << n << ": " << i << endl;
    }
}

int main(){
    thread t1(run, 1);
    thread t2(run, 2);
    thread t3(run, 3);

    t1.join();
    t2.join();
    t3.join();

    return 0;
}
```

```
1: 0
1: 1
1: 2
1: 3
1: 4
23: 0
3: 1
3: 2
3: 3
3: 4
: 0
2: 1
2: 2
2: 3
2: 4
```

std::mutex

C++11

```
#include <iostream>
#include <thread>
#include <mutex>

using namespace std;

mutex m;

void run(size_t n){
    m.lock();
    for (size_t i = 0; i < 5; ++i){
        cout << n << ": " << i << endl;
    }

    m.unlock();
}

int main(){
    thread t1(run, 1);
    thread t2(run, 2);
    thread t3(run, 3);

    t1.join();
    t2.join();
    t3.join();

    return 0;
}
```

Output(is defined within run)

```
1: 0
1: 1
1: 2
1: 3
1: 4
2: 0
2: 1
2: 2
2: 3
2: 4
3: 0
3: 1
3: 2
3: 3
3: 4
```

std::lock_guard+std::mutex

C++11

```
#include <iostream>
#include <thread>
#include <mutex>

using namespace std;

mutex m;

void run(size_t n){
    m.lock();
    for (size_t i = 0; i < 5; ++i){
        cout << n << ": " << i << endl;
    }

    m.unlock();
}

int main(){
    thread t1(run, 1);
    thread t2(run, 2);
    thread t3(run, 3);

    t1.join();
    t2.join();
    t3.join();

    return 0;
}
```

```
#include <iostream>
#include <thread>
#include <mutex>

using namespace std;

mutex m;

void run(size_t n){
    lock_guard<mutex> lm(m); //ctor – m.lock(), dtor – m.unlock()
    for (size_t i = 0; i < 5; ++i){
        cout << n << ": " << i << endl;
    }
}

int main(){
    thread t1(run, 1);
    thread t2(run, 2);
    thread t3(run, 3);

    t1.join();
    t2.join();
    t3.join();

    return 0;
}
```

std::async

C++11

```
#include <iostream>
#include <future>

using namespace std;

int Fib(int n){
    return n<=2 ? 1 : Fib(n-1)+Fib(n-2);
}

int calc1(){ return Fib(30); }

int calc2(){ return Fib(40); }

int main()
{
    // start calc1() asynchronously
    future<int> result1 = async(calc1);
    // call calc2() synchronously
    int result2 = calc2();
    // wait for calc1() and add its result to result2
    int result = result1.get() + result2;
    cout << "calc1()+calc2(): " << result << endl;
    return 0;
}
```

Deprecated idioms

C++11

Now that we have C++11, we can use new features instead of following idioms:

[nullptr](#)

[Move Constructor](#)

[Safe bool](#)

[Shrink-to-fit](#)

[Type Safe Enum](#)

[Requiring or Prohibiting Heap-based Objects](#)

[Type Generator](#)

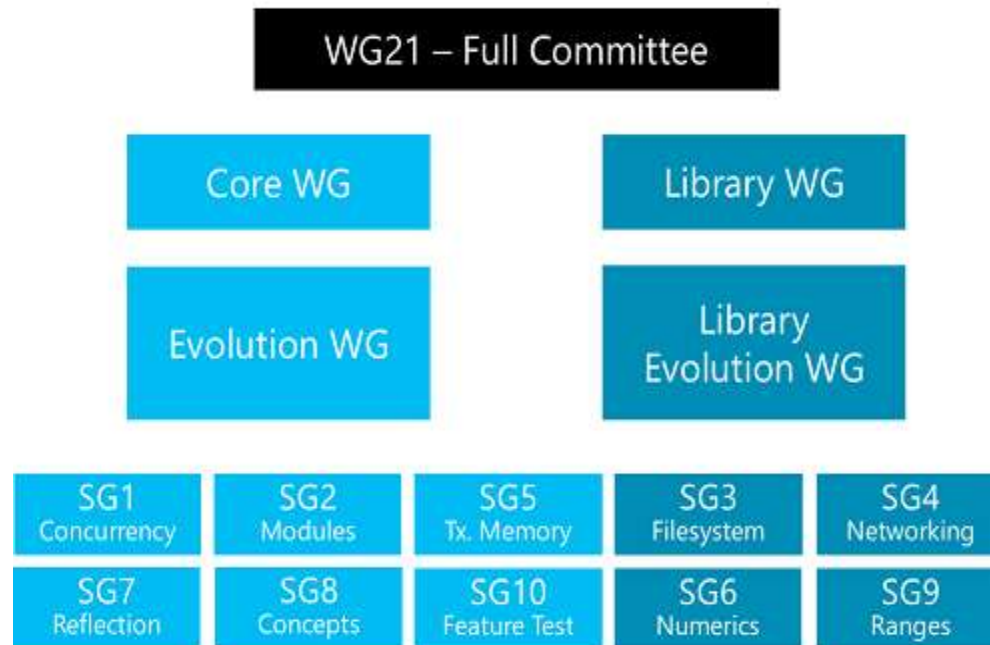
[Final Class](#)

[address_of](#)

<http://stackoverflow.com/questions/9299101/what-c-idioms-are-deprecated-in-c11>

C++11 compiler support

gcc	icc	msvc(with NOV CTP)	ibm xlc	clang
38/39 Not implemented: threads, regex	27/39 Full STL support	24/39 Full STL support(without init. list)	17/39 Not implemented: threads, regex	37/39 Full STL support



links

<http://www.isocpp.org>

<http://www.cplusplus.com>

<http://www.stroustrup.com/C++11FAQ.html>

<http://channel9.msdn.com/Events/GoingNative/GoingNative-2012/Keynote-Bjarne-Stroustrup-Cpp11-Style>

<http://channel9.msdn.com/Events/Build/BUILD2011/TOOL-835T>

<http://channel9.msdn.com/posts/C-and-Beyond-2011-Herb-Sutter-Why-C>

<http://channel9.msdn.com/Events/Lang-NEXT/Lang-NEXT-2012/-Not-Your-Father-s-C->

<http://cpprocks.com/cpp11-stl-additions/>

<http://cpprocks.com/c11-a-visual-summary-of-changes/#!prettyPhoto>

<http://wiki.apache.org/stdcxx/C++0xCompilerSupport>