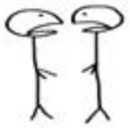
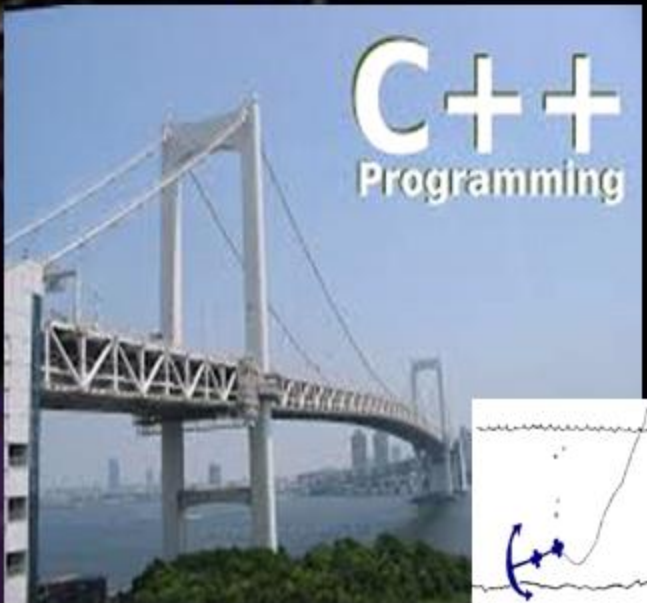
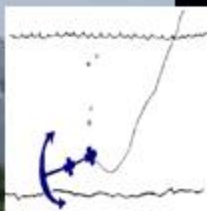
C++



New

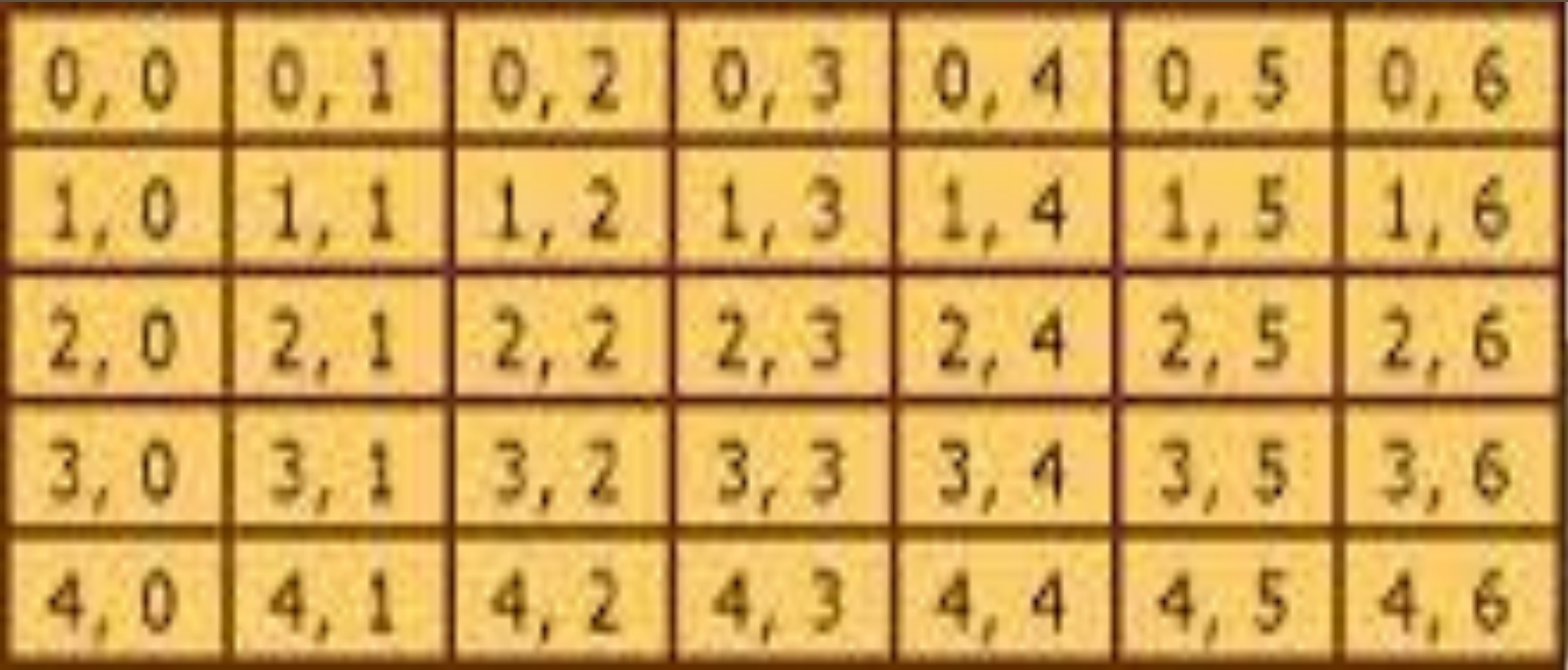
```
#include <iostream>
using namespace std;

int main(){
    cout<<"Hola Facebook\n";
    return 0;
}
```





Simple class artifacts

1. Built-in default constructor
2. Built-in copy constructor
3. Built-in assignment operator
4. Initialization lists



0, 0	0, 1	0, 2	0, 3	0, 4	0, 5	0, 6
1, 0	1, 1	1, 2	1, 3	1, 4	1, 5	1, 6
2, 0	2, 1	2, 2	2, 3	2, 4	2, 5	2, 6
3, 0	3, 1	3, 2	3, 3	3, 4	3, 5	3, 6
4, 0	4, 1	4, 2	4, 3	4, 4	4, 5	4, 6

Initializer lists

```
struct CAT{
  float first;
  int second;
};

CAT scalar = {0.36f, 14};
//One Object, with first=0. 36f and second=14

CAT anArray[] = {{13.4f, 3}, {43.28f, 29}, {5.934f, 17}}; //An
array of three Objects
```

in-class member initializers

C++03

```
class C
{
 int x;
public: C() : x(7)
{
}
};
```

C++11

```
class C {
 int x=7; //class member initializer
int y[5] {1,2,3,4};
string s("abc");
String s2
public:
C(){
}
};
```

initializer-list constructor

An *initializer-list constructor* is a constructor with std::initializer_list<T> as first argument,

without any additional arguments or

with additional arguments having default values.

```
class PointSequence
{
  public:
    PointSequence(initializer_list<double> args)
    {
      if (args.size() % 2 != 0)
      {
        throw invalid_argument("initializer_list should "
          "contain even number of elements.");
      }
      for (auto iter = args.begin(); iter != args.end(); ++iter)
        mVecPoints.push_back(*iter);
    }
    void dumpPoints() const
    {
      for (auto citer = mVecPoints.cbegin();
        citer != mVecPoints.cend(); citer += 2)
        {
          cout << "(" << *citer << ", " << *(citer+1) << ")" << en
        }
    }
  protected:
    vector<double> mVecPoints;
};
```

Uniform Initialization and std::initializer_list

C++03	C++11

int a[] = { 1, 2, 3, 4, 5 };
vector<int> v;
for(int i = 1; i <= 5; ++i) v.push_back(i);
	int a[] = { 1, 2, 3, 4, 5 };
	vector<int> v = { 1, 2, 3, 4, 5 };

map<int, string> labels;
labels.insert(make_pair(1, “Open”));
labels.insert(make_pair(2, “Close”));
labels.insert(make_pair(3, “Reboot”));
	map<int, string> labels {
	{ 1 , "Open" },
	{ 2 , "Close" },
	{ 3 , "Reboot" } };

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

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 initiailizer_list works*/
    for(auto it = begin(args); it != end(args); ++it)
      push_back(*it);
  }
//…
};
```

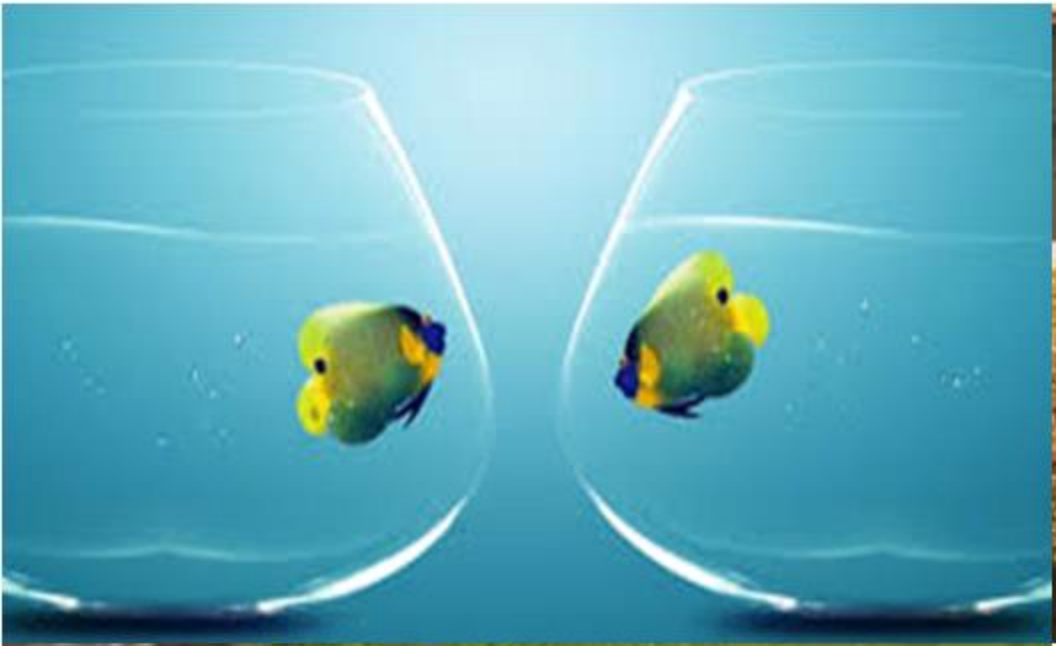
```
//what is initializer_list<T> ?
```

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;
```

Initializer-list constructor and Other Constructors



`std::initializer_list<>` a real type

The class `std::initializer_list<>` is a first-class C++11 standard library type.

It can be used in other places besides class constructors

```
void function_name(std::initializer_list<float> list);

function_name({1.0f, -3.45f, -0.4f});
```

Type inference

The Auto Keyword

auto for Type Declarations

auto variables have the type of their initializing expression:

```
auto x1 = 10;                        // x1: int

std::map<int, std::string> m;
auto i1 = m.begin();                 // i1: std::map<int, std::string>::iterator
```

const/volatile and reference/pointer adornments may be added:

```
const auto *x2 = &x1;                // x2: const int*

const auto& i2 = m;                  // i2: const std::map<int, std::string>&
```

To get a const_iterator, use the new cbegin container function:

```
auto ci = m.cbegin();                // ci: std::map<int, std::string>::const_iterator
```

- cend, crbegin, and crend exist, too.

auto for Type Declarations

Type deduction for auto is akin to that for template parameters:

```
template<typename T> void f(T t);
…
f(expr);                    // deduce t's type from expr
auto v = expr;              // do essentially the same thing for v's type
```

auto for Type Declarations

For variables *not* explicitly declared to be a reference:

- Top-level consts/volatiles in the initializing type are ignored.
- Array and function names in initializing types decay to pointers.

```
const std::list<int> li;
auto v1 = li;                    // v1: std::list<int>
auto& v2 = li;                   // v2: const std::list<int>&
float data[BufSize];
auto v3 = data;                  // v3: float*
auto& v4 = data;                 // v4: float (&)[BufSize]
```

Examples from earlier:

```
auto x1 = 10;                    // x1: int
std::map<int, std::string> m;
auto i1 = m.begin();             // i1: std::map<int, std::string>::iterator
const auto *x2 = &x1;            // x2: const int* (const isn't top-level)
const auto& i2 = m;              // i2: const std::map<int, std::string>&

auto ci = m.cbegin();            // ci: std::map<int, std::string>::const_iterator
```

auto for Type Declarations

auto can be used to declare multiple variables:

```
void f(std::string& s)
{
  auto temp = s, *pOrig = &s;        // temp: std::string,
  …                                  // pOrig: std::string*
}
```

Each initialization must yield the same deduced type.

```
auto i = 10, d = 5.0;                // error!
```

auto for Type Declarations

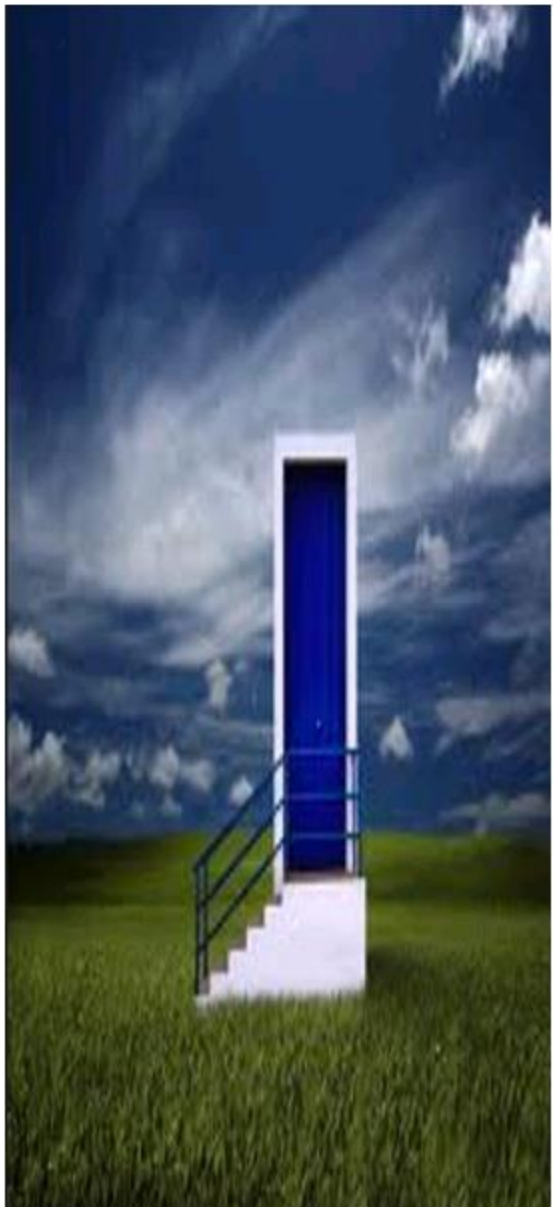
Both direct and copy initialization syntaxes are permitted.

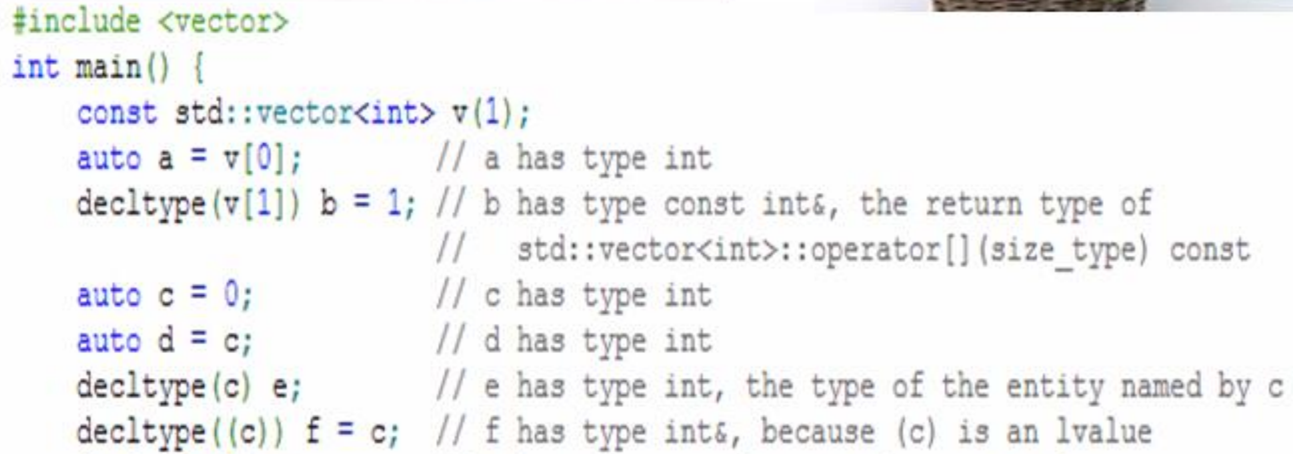
```
auto v1(expr);                  // direct initialization syntax
auto v2 = expr;                 // copy initialization syntax
```

For auto, both syntaxes have the same meaning.

decltype type specifier

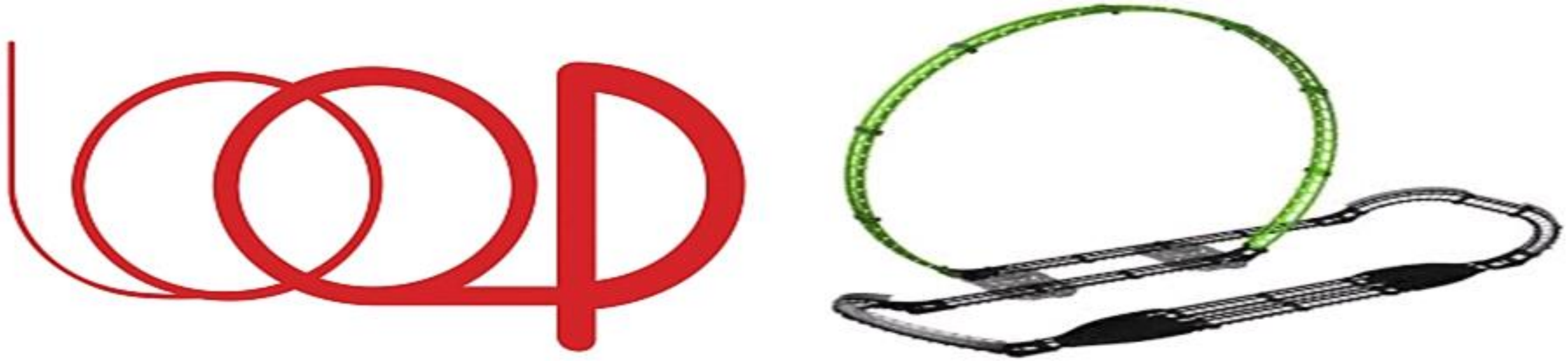
This can be used to determine the type of a expression.

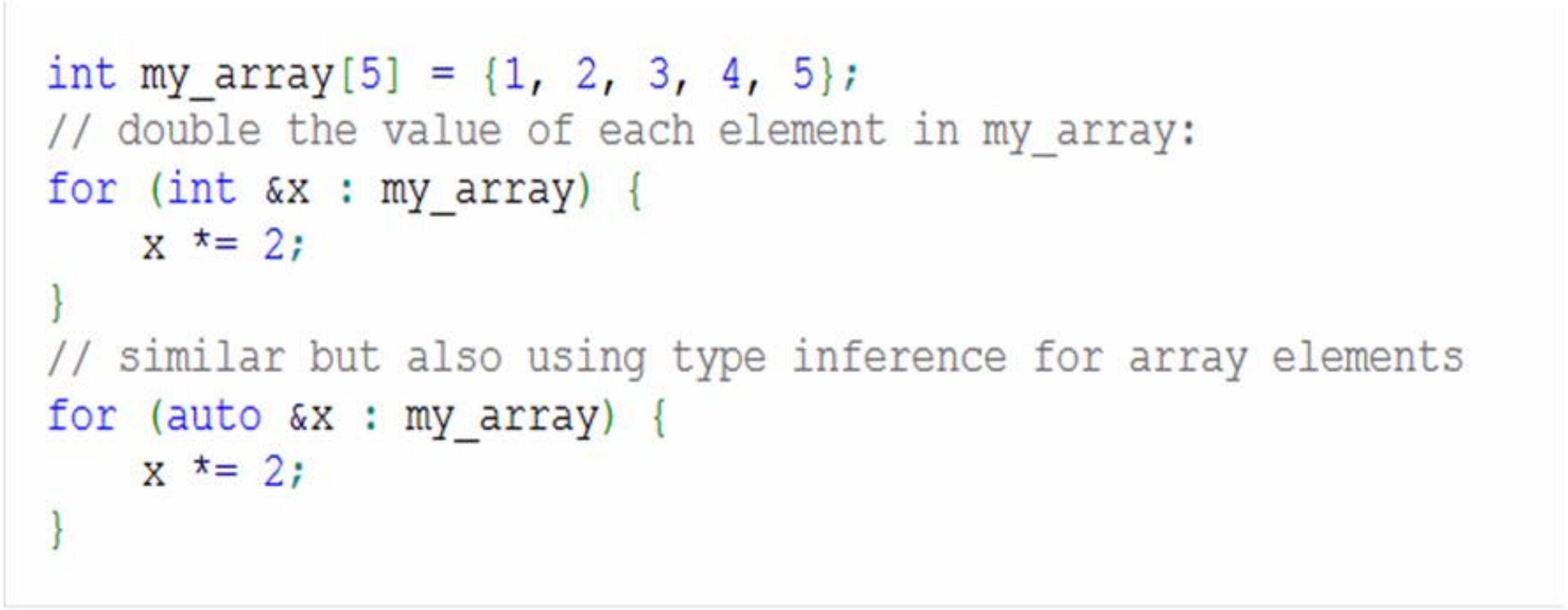




```
#include <vector>
int main() {
    const std::vector<int> v(1);
    auto a = v[0];          // a has type int
    decltype(v[1]) b = 1; // b has type const int&, the return type of
                          //   std::vector<int>::operator[](size_type) const
    auto c = 0;           // c has type int
    auto d = c;           // d has type int
    decltype(c) e;        // e has type int, the type of the entity named by c
    decltype((c)) f = c;  // f has type int&, because (c) is an lvalue
```

Range-based for loop





```
int my_array[5] = {1, 2, 3, 4, 5};
// double the value of each element in my_array:
for (int &x : my_array) {
    x *= 2;
}
// similar but also using type inference for array elements
for (auto &x : my_array) {
    x *= 2;
}
```

Allow for easy iteration over a range of elements

Range-Based for Loops

Looping over a container can take this streamlined form:

```
std::vector<int> v;
…
for (int i : v) std::cout << i;             // iteratively set i to every
                                            // element in v
```

The iterating variable may also be a reference:

```
for (int& i : v) std::cout << ++i;          // increment and print
                                            // everything in v
```

auto, const, and volatile are allowed:

```
for (auto i : v) std::cout << i;                 // same as above
for (auto& i : v) std::cout << ++i;              // ditto
for (volatile int i : v) someOtherFunc(i);       // or "volatile auto i"
```

Range-Based for Loops

Valid for any type supporting the notion of a *range*.

- Given object obj of type T, begin(obj) and end(obj) are valid.

Includes:

- All C++0x library containers.
- Arrays and valarrays.
- Initializer lists.
- Regular expression matches.
- Any UDT T where begin(T) and end(T) yield suitable iterators.

Range-Based for Loops

Examples:

```
std::unordered_multiset<std::shared_ptr<Widget>> msspw;
…
for (const auto& p : msspw) {
   std::cout << p << '\n';                          // print pointer value
}


short vals[ArraySize];
…
for (auto& v : vals) { v = -v; }
```

Range-Based for Loop Specification

```
for ( iterVarDeclaration : expression ) statementToExecute
```

is essentially equivalent to

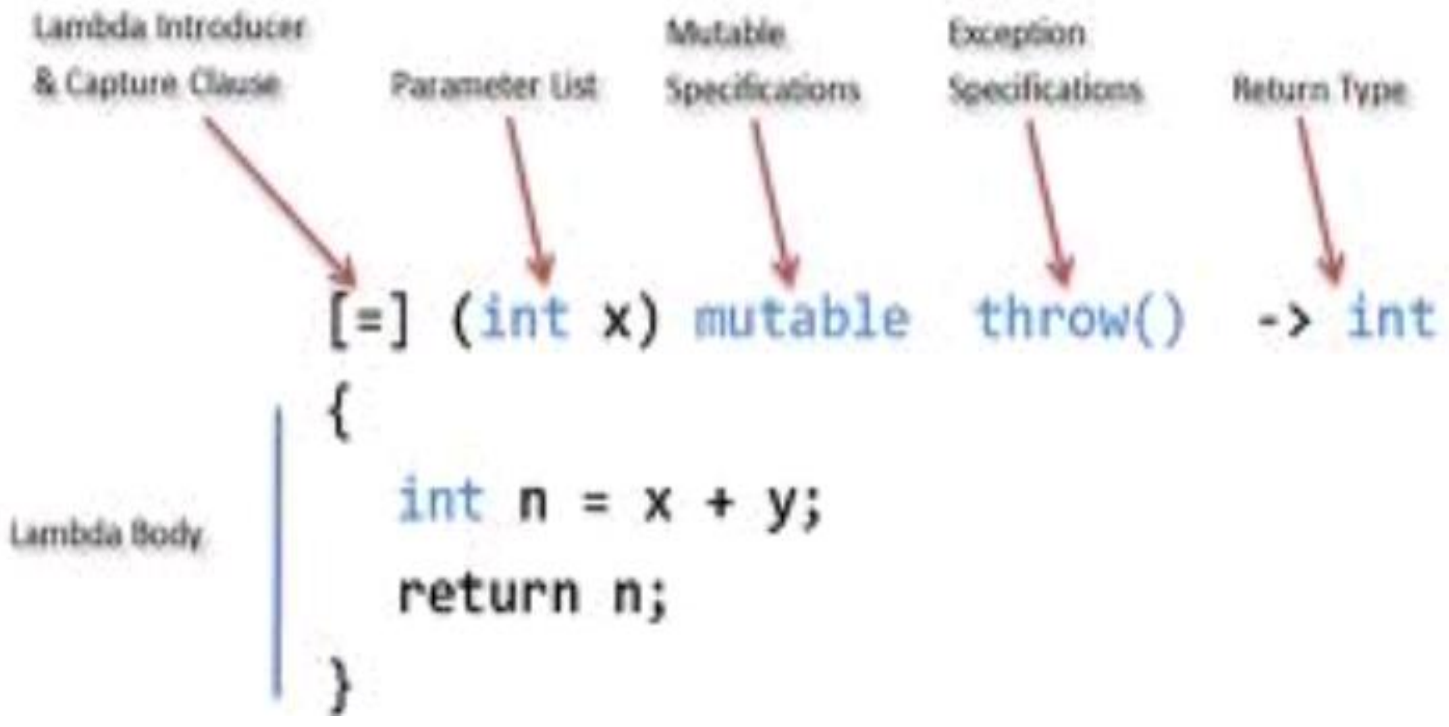
```
{
  auto&& range = expression;
  for (auto  b = begin(range), e = end(range);
       b != e;
       ++b ) {
    iterVarDeclaration = *b;
    statementToExecute
  }
}
```

Standardese somewhat more complex.

Range-Based for Loops

Range form valid only for for-loops.

- Not do-loops, not while-loops.



```
[=] (int x) mutable  throw()  -> int
{
   int n = x + y;
   return n;
}
```

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

Lambda Closures Implemented

[]	Capture nothing (or, a scorched earth strategy?)
[&]	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
[this]	Capture the this pointer of the enclosing class

lambdas/closures

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

Making Delegates with Lambdas

lambda with an empty capture specification
treated like a regular function and assigned to a function pointer.

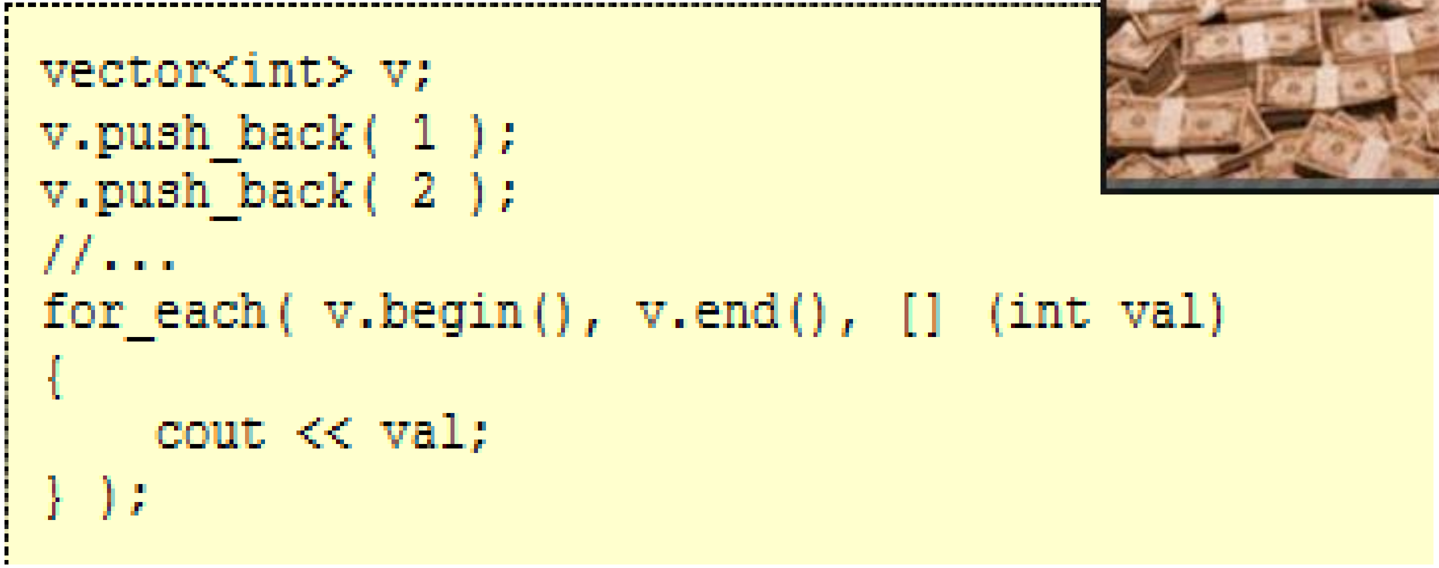
lambda with capture specification
treated differently(object state to be passed)
and assigned to a **std::function**.

recursive lambdas

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

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

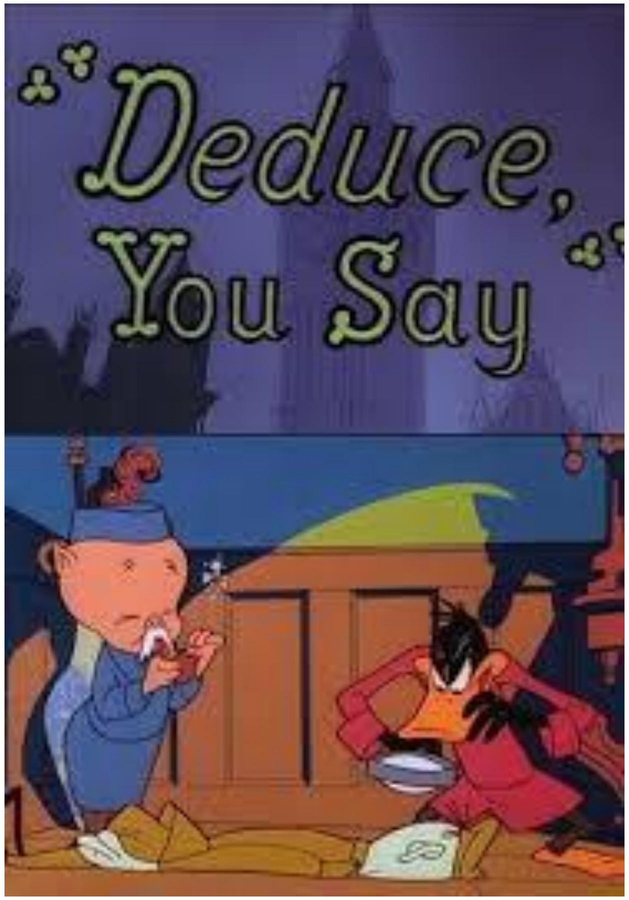
Lambda and the STL

STL algorithms package ***biggest beneficiarie***s of lambda functions



```
vector<int> v;
v.push_back( 1 );
v.push_back( 2 );
//...
for_each( v.begin(), v.end(), [] (int val)
{
    cout << val;
} );
```

Return Values in lambda



By default, if your lambda does not have a return statement, it defaults to void.

If you have a simple return expression, the compiler will deduce the type of the return value:

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;
}
```

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);
```

function objects

C++11(deprecated binders and adaptors)	C++11
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	**Function wrappers** function mem_fn bad_function_call **Bind** bind is_bind_expression is_placeholder _1, _2, _3, ... **Reference wrappers** reference_wrapper ref cref

delegating constructors

C++03

```
class A
{
  int a;
  void validate(int x)
  {
    if (5<x && x<=15) a=x; else throw Invalid(x);
  }
public:
  A(int x) { validate(x); }
  A() { validate(10); }
  A(string s)
  {
    int x = stoi(s);
    validate(x);
  }
```

C++11

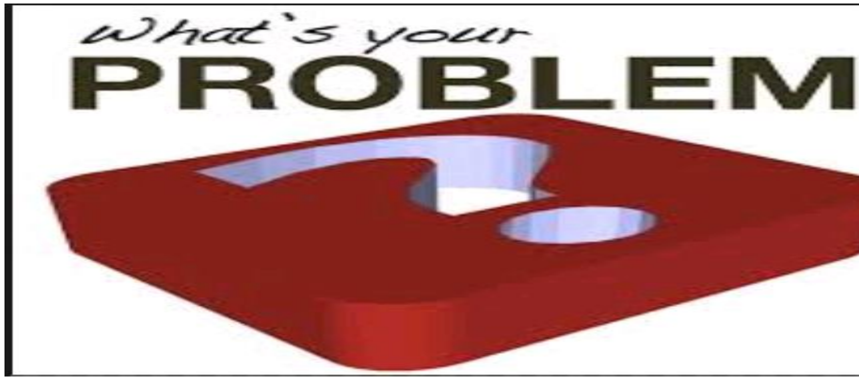
```
class A
{
  int a;
public:
  A(int x)
  {
    if (5<x && x<10) a=x; else throw Invalid(x);
  }
  A() : A(10){ }
  A(string s) : A(stoi(s)){ }
};
```

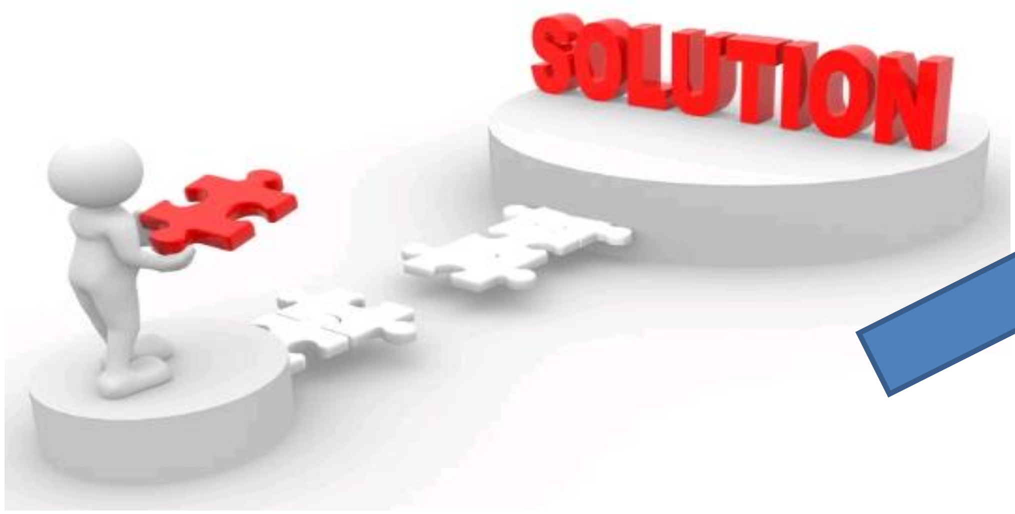
Explicit overrides and final

it is possible to accidentally create a new virtual function.

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

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

common problem, particularly when a user goes to modify the base class.





override
Final
Keywords

Null pointer constant

C++03

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

foo(NULL); //calls second foo
```

C++11

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

foo(nullptr); //calls first foo
```

nullptr

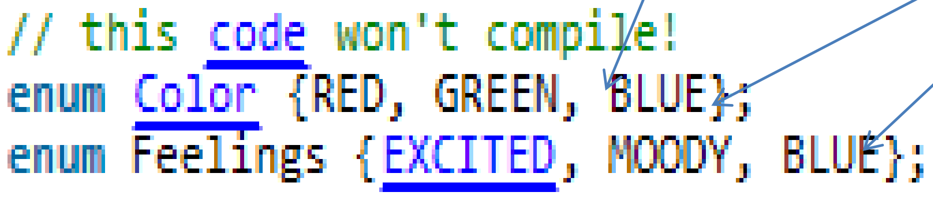
Unlike 0 and NULL, nullptr works well with forwarding templates:

```
template<typename F, typename P>        // make log entry, then
void logAndCall(F func, P param)        // invoke func on param
{
  …                                     // write log entry
  func(param);
}

void f(int* p);                         // some function to call


f(0);                                   // fine
f(nullptr);                             // also fine

logAndCall(f, 0);                       // error! P deduced as
                                        // int, and f(int) invalid

logAndCall(f, NULL);                    // error!

logAndCall(f, nullptr);                 // fine, P deduced as
                                        // std::nullptr_t, and
                                        // f(std::nullptr_t) is okay
```

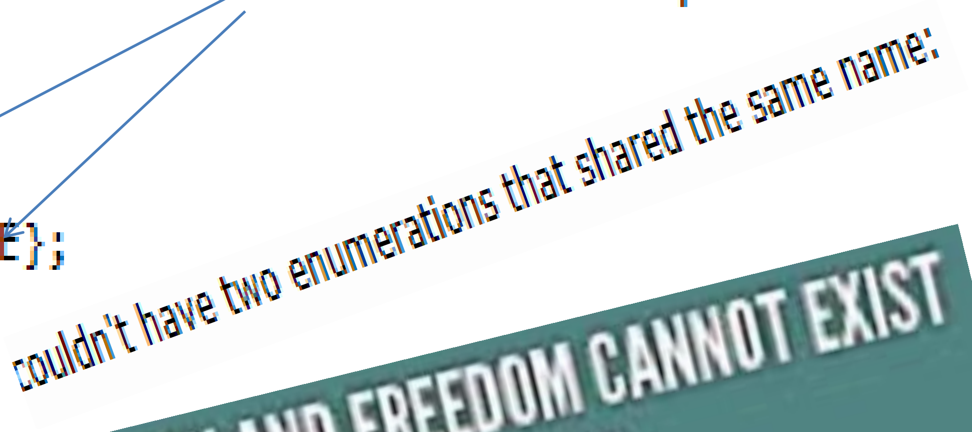
Strongly typed enums - enum classes

effectively integers

enum values were unscoped.

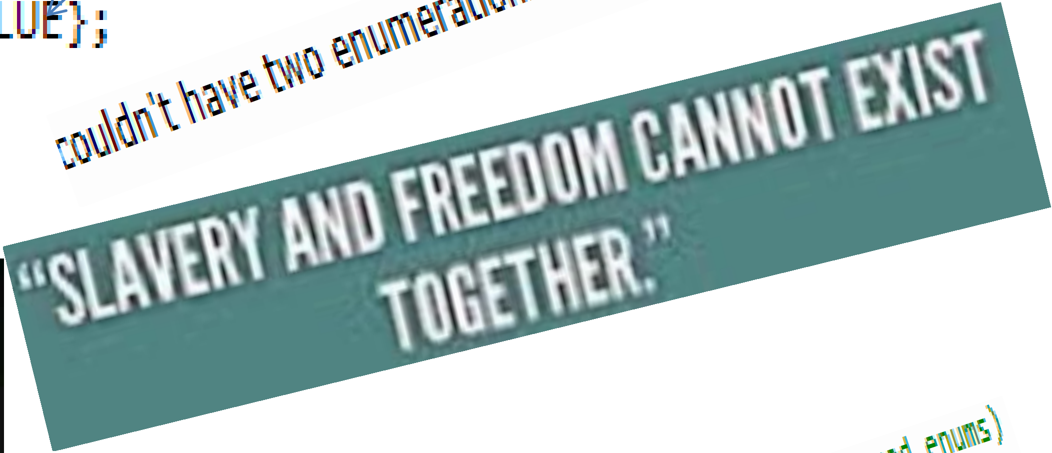


```
// this code won't compile!
enum Color {RED, GREEN, BLUE};
enum Feelings {EXCITED, MOODY, BLUE};
```

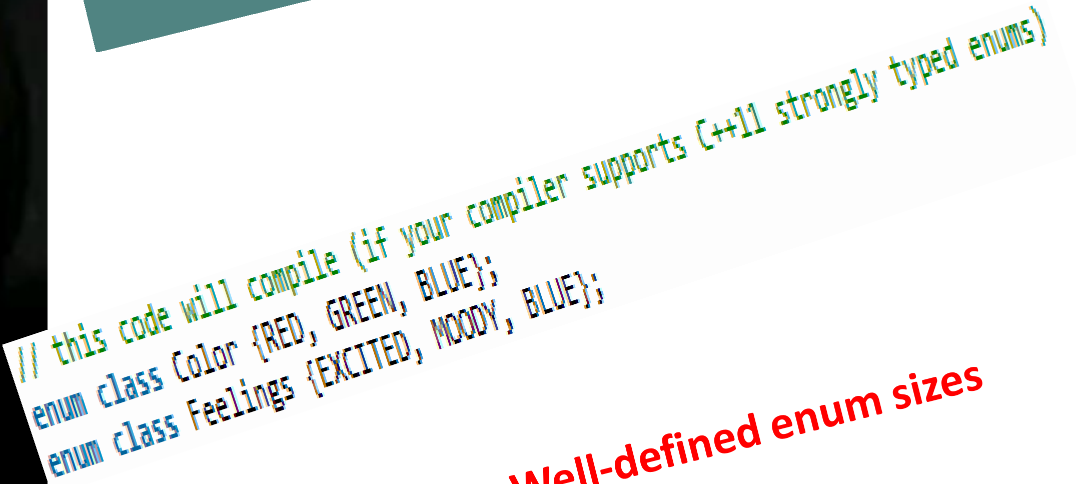


couldn't have two enumerations that shared the same name:

C++03, enumerations are not type-safe.



"SLAVERY AND FREEDOM CANNOT EXIST TOGETHER."



```
// this code will compile (if your compiler supports C++11 strongly typed enums)
enum class Color {RED, GREEN, BLUE};
enum class Feelings {EXCITED, MOODY, BLUE};
```

Well-defined enum sizes

“>>”as Nested Template Closer

“>>” now closes a nested template when possible:

```
std::vector<std::list<int>> vi1;      // fine in C++0x, error in C++98
```

The C++98 “extra space” approach remains valid:

```
std::vector<std::list<int> > vi2;     // fine in C++0x and C++98
```

For a shift operation, use parentheses:

- I.e., “>>” now treated like “>” during template parsing.

```
const int n = … ;                     // n, m are compile-
const int m = … ;                     // time constants

std::array<int, n>m?n:m > a1;         // error (as in C++98)

std::array<int, (n>m?n:m) > a2;       // fine (as in C++98)

std::list<std::array<int, n>>2 >> L1; // error in ’98: 2 shifts;
                                      // error in ’0x: 1st “>>”
                                      // closes both templates

std::list<std::array<int, (n>>2) >> L2; // fine in C++0x,
                                        // error in ’98 (2 shifts)
```

Right angle brackets:

```
map<int, vector<int>> _Map;
```

This is an error with earlier compilers as there is no space between >'s

Treated it as right shift operator.

But C++11 compilers will parse these multiple right angle brackets

Explicit conversion operators

C++03

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

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

C++03

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

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

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;
}
```

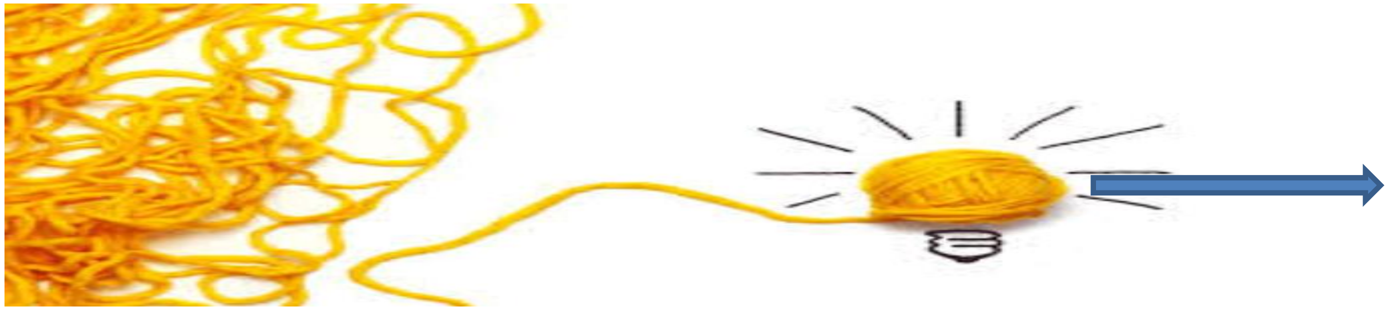
C++11

```
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;
}
```

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;
}
```

Alias templates



C++03	C++11

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

`template <int U, int V> class Type;`
 	
`typedef Type<42,36> ConcreteType;`
	`template <int U, int V> class Type;`
	
	`using ConcreteType = Type<42,36>;`

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

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

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

template <int U, int V> class Type;
 	
typedef Type<42,36> ConcreteType;
	template <int U, int V> class Type;
	
	using ConcreteType = Type<42,36>;

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

Unrestricted unions

unions cannot contain any objects that define a non-trivial constructor

```
#include <new> // Required for placement 'new'.

struct Point {
    Point() {}
    Point(int x, int y): x_(x), y_(y) {}
    int x_, y_;
};

union U {
    int z;
    double w;
    Point p; // Illegal in C++03; legal in C++11.
    U() {new(&p) Point();} // Due to the Point member, a constructor definition is now required.
};
```

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.
```

Parameter™

Pack

C++11

```
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');
```

Output

```
42,hello,2.3,a
```

raw string literals

C++03	C++11

string test="C:\\A\\B\\C\\D\\file1.txt";
cout << test << endl;
	string test=**R"(**C:\A\B\C\D\file1.txt**)"**;
	cout << test << endl;

C:\A\B\C\D\file1.txt	C:\A\B\C\D\file1.txt

string test;
test = "First Line.\nSecond line.\nThird Line.\n";
cout << test << endl;
	string test;
	test = **R"(**First Line.\nSecond line.\nThird Line.\n**)"**;
	cout << test << endl;

First Line. Second line. Third Line.	First Line.\nSecond line.\nThird Line.\n
	string test = **R"(**First Line. Second line. Third Line.**)"**; cout << test << endl;
	First Line. Second line. Third Line.

Unicode Support

Two new character types:

```
char16_t        // 16-bit character (if available);
                // akin to uint_least16_t
char32_t        // 32-bit character (if available);
                // akin to uint_least32_t
```

Literals of these types prefixed with u/U, are UCS-encoded:

```
u'x'            // 'x' as a char16_t using UCS-2
U'x'            // 'x' as a char32_t using UCS-4/UTF-32
```

C++98 character types still exist, of course:

```
'x'             // 'x' as a char
L'x'            // 'x' as a wchar_t
```

Unicode Support

There are corresponding string literals:

```
u"UCS-2 string literal"                // ⇒ char16_ts in UTF-16
U"UCS-4 string literal"                // ⇒ char32_ts in UCS-4/UTF-32
"Ordinary/narrow string literal"       // "ordinary/narrow" ⇒ chars
L"Wide string literal"                 // "wide" ⇒ wchar_ts
```

UTF-8 string literals are also supported:

```
u8"UTF-8 string literal"               // ⇒ chars in UTF-8
```

Code points can be specified via \u*nnnn* and \U*nnnnnnnn*:

```
u8"G clef: \U0001D11E"                          // )
u"Thai character Khomut: \u0E5B"                // ๛
U"Skull and crossbones: \u2620"                 // ☠
```

Unicode Support

There are std::basic_string typedefs for all character types:

```
std::string s1;                  // std::basic_string<char>
std::wstring s2;                 // std::basic_string<wchar_t>
std::u16string s3;               // std::basic_string<char16_t>
std::u32string s4;               // std::basic_string<char32_t>
```

Raw String Literals

String literals where “special” characters aren't special:

- E.g., escaped characters and double quotes:

```
std::string noNewlines(R"(\n\n)");
std::string cmd(R"(ls /home/docs | grep ".pdf")");
```

- E.g., newlines:

```
std::string withNewlines(R"(Line 1 of the string...
                            Line 2...
                            Line 3)");
```

“Rawness” may be added to any string encoding:

```
LR"(Raw Wide string literal \t (without a tab))"
u8R"(Raw UTF-8 string literal \n (without a newline))"
uR"(Raw UTF-16 string literal \\ (with two backslashes))"
UR"(Raw UTF-32 string literal \u2620 (without a code point))"
```

User-defined literals

```
constexpr long double operator"" _deg(long double deg)
{
		return deg*3.141592 / 180;
}
```

http://msdn.microsoft.com/en-us/library/hh567368.aspx

standard types

C++03	C++11

sizeof(int) == ?
sizeof(char) == 1 byte(== ? bits)
 	
sizeof(char) <= sizeof(short) <=
sizeof(int) <= sizeof(long)
	int8_t
	uint8_t
	int16_t
	uint16_t
	int32_t
	uint32_t
	int64_t
	uint64_t

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 long long int

In C++03, the largest integer type is long int

Resulted having size of 64 bits on some implementations and 32 bits on others

long long int is at least as Big as a long int

.

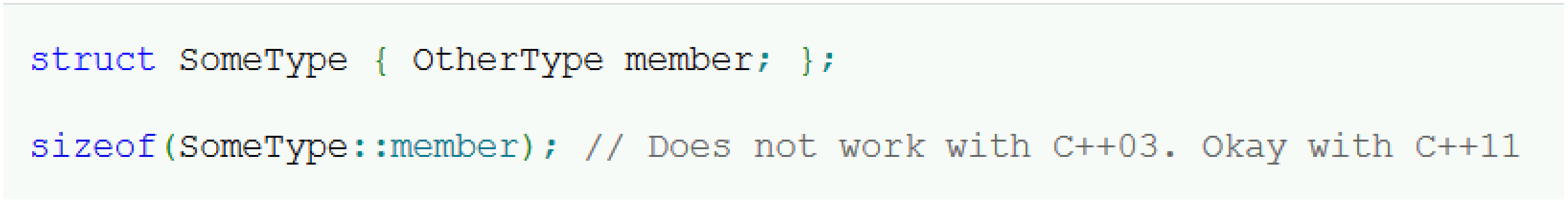
have no fewer than 64 bits.

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;
};
```

Sizeof works on members of classes without an explicit object



```
struct SomeType { OtherType member; };

sizeof(SomeType::member); // Does not work with C++03. Okay with C++11
```

object alignment

Alignment is a restriction on wich memory positions a value's first byte can be stored.

alignof operator takes a type and returns the power of 2 byte boundary

alignas specifier controls the memory alignment for a variable

```
alignas(float) unsigned char c[sizeof(float)]
```

std::tuple

C++11	**python**

tuple<int,float,string> t(1,2.f,”text”);
int x = **get**<0>(t);
float y = **get**<1>(t);
string z = **get**<2>(t);
	t = (1,2.0,’text’)
	x = t[0]
	y = t[1]
	z = t[2]

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;
	// packing values into tuple
	mytuple = (10, 2.6, 'a')
	// unpacking tuple into variables
	myint, _, mychar = mytuple

int a = 5;
int b = 6;
tie(b, a) = **make_tuple**(a, b);
	a = 5
	b = 6
	b,a = a,b

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;
```

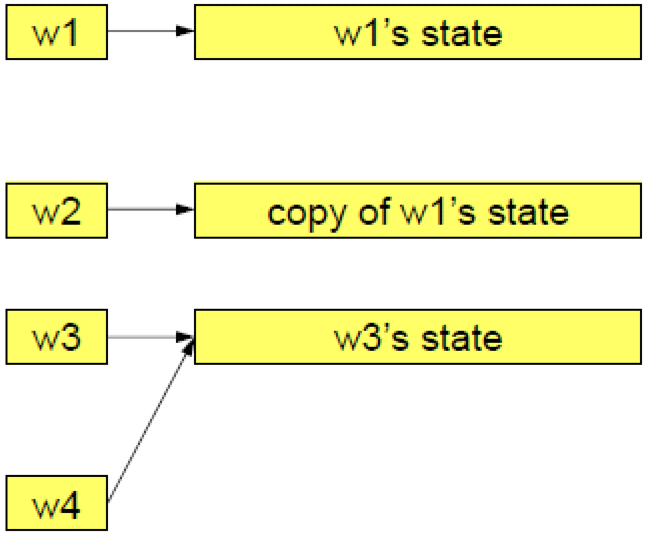
Copying vs. Moving

C++ has always supported copying object state:

- *Copy* constructors, *copy* assignment operators

C++0x adds support for requests to *move* object state:

```
Widget w1;
...
// copy w1's state to w2
Widget w2(w1);

Widget w3;
…
// move w3's state to w4
Widget w4(std::move(w3));
```

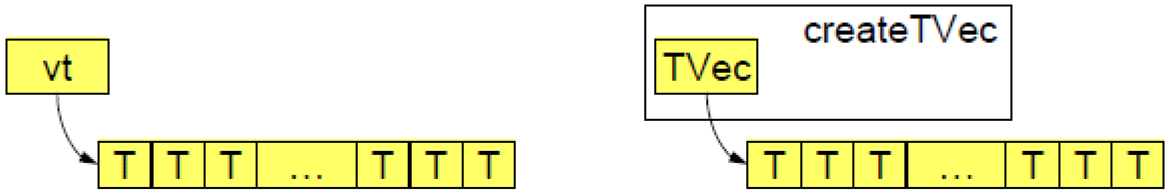


Note: w3 continues to exist in a valid state after creation of w4.

Copying vs. Moving

Temporary objects are prime candidates for moving:

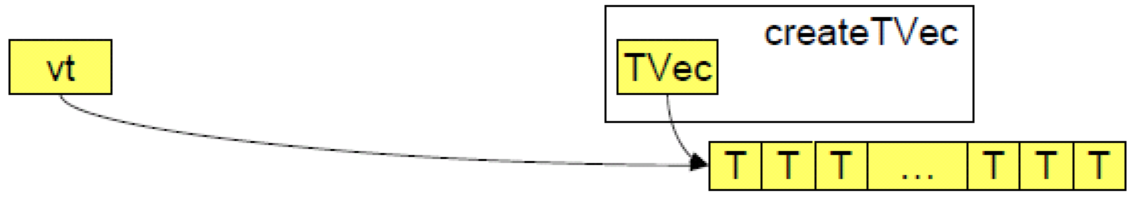
```
typedef std::vector<T> TVec;
TVec createTVec();                  // factory function
TVec vt;
…
vt = createTVec();                  // in C++98, copy return value to
                                    // vt, then destroy return value
```



Copying vs. Moving

C++0x turns such copy operations into move requests:

```
TVec vt;
…
vt = createTVec();                    // implicit move request in C++0x
```



Move Semantics

Detecting temporary objects with rvalue references

Move constructor and move assignment operator

Move constructor

The parameter is a non-const rvalue reference
Parameter pointers are set to NULL

Move Semantics

```cpp
class ArrayWrapper
{
    public:
        ArrayWrapper (int n)
            : _p_vals( new int[ n ] )
            , _size( n )
        {}
        // copy constructor
        ArrayWrapper (const ArrayWrapper& other)
            : _p_vals( new int[ other._size  ] )
            , _size( other._size )
        {
            for ( int i = 0; i < _size; ++i )
            {
                _p_vals[ i ] = other._p_vals[ i ];
            }
        }
        ~ArrayWrapper ()
        {
            delete [] _p_vals;
        }
    private:
    int *_p_vals;
    int _size;
};
```

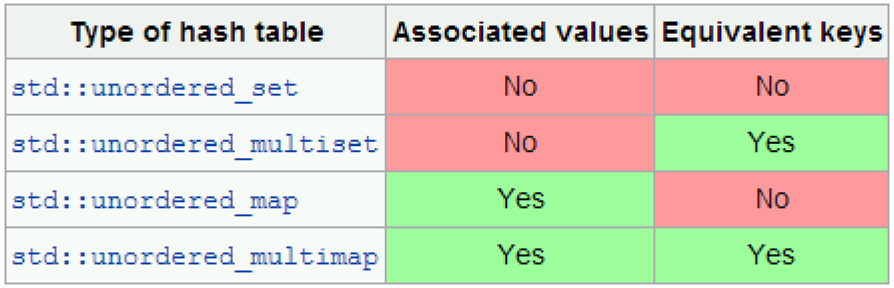
```cpp
class ArrayWrapper
{
public:
    // default constructor produces a moderately sized array
    ArrayWrapper ()
        : _p_vals( new int[ 64 ] )
        , _size( 64 )
    {}

    ArrayWrapper (int n)
        : _p_vals( new int[ n ] )
        , _size( n )
    {}

    // move constructor
    ArrayWrapper (ArrayWrapper&& other)
        : _p_vals( other._p_vals  )
        , _size( other._size )
    {
        other._p_vals = NULL;
        other._size = 0;
    }

    // copy constructor
    ArrayWrapper (const ArrayWrapper& other)
        : _p_vals( new int[ other._size  ] )
        , _size( other._size )
    {
        for ( int i = 0; i < _size; ++i )
        {
            _p_vals[ i ] = other._p_vals[ i ];
        }
    }
    ~ArrayWrapper ()
    {
        delete [] _p_vals;
    }

private:
    int *_p_vals;
    int _size;
};
```

Copying vs. Moving

Move semantics examined in detail later, but:

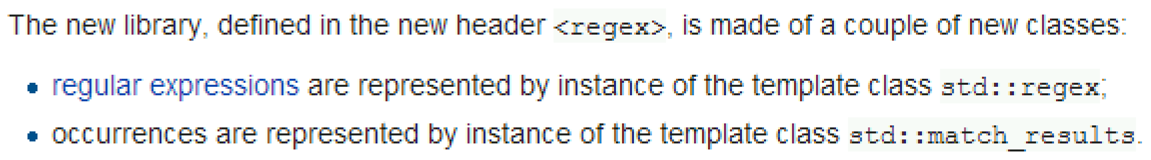
- **Moving a key new C++0x idea.**
  - Usually an optimization of copying.
- Most standard types in C++0x are *move-enabled*.
  - They support move requests.
  - E.g., STL containers.
- Some types are *move-only*:
  - Copying prohibited, but moving is allowed.
  - E.g., stream objects, `std::thread` objects, `std::unique_ptr`, etc.

Hash tables



Type of hash table	Associated values	Equivalent keys
`std::unordered_set`	No	No
`std::unordered_multiset`	No	Yes
`std::unordered_map`	Yes	No
`std::unordered_multimap`	Yes	Yes

Regular expressions



The new library, defined in the new header `<regex>`, is made of a couple of new classes:

- regular expressions are represented by instance of the template class `std::regex`;
- occurrences are represented by instance of the template class `std::match_results`.

using std::regex; regex reg2("[0-9]*"); // Match 0 or more digits.
 regex reg3("(\\+|-)?[0-9]+"); // Match digit string with // optional + or -

Regular expressions

- **\d**

Matches any digit character. This is equivalent to:

```
[0-9]
```

- **\D**

Matches any character other than a digit. This is equivalent to:

```
[^0-9]
```

- **\s**

Matches any whitespace character.

- **\S**

Matches any character other than a whitespace character.

- **\w**

Matches any digit, letter, or underscore.

- **[:alpha:]**

Any letter.

- **[:blank:]**

A space or tab character.

- **[:cntrl:]**

Any control character. (These are not printable.)

- **[:digit:]**

Any decimal digit.

- **[:graph:]**

Any printable character that is not a whitespace.

- **[:lower:]**

Any lowercase letter.

Main classes

These classes encapsulate a regular expression and the results of matching a regular expression within a target sequence of characters.

basic_regex (C++11)	regular expression object (class template)
sub_match (C++11)	identifies the sequence of characters matched by a sub-expression (class template)
match_results (C++11)	identifies one regular expression match, including all sub-expression matches (class template)

Algorithms

These functions are used to apply the regular expression encapsulated in a regex to a target sequence of characters.

regex_match (C++11)	attempts to match a regular expression to an entire character sequence (function template)
regex_search (C++11)	attempts to match a regular expression to any part of a character sequence (function template)
regex_replace (C++11)	replaces occurrences of a regular expression with formatted replacement text (function template)

Iterators

The regex iterators are used to traverse the entire set of regular expression matches found within a sequence.

regex_iterator (C++11)	iterates through all regex matches within a character sequence (class template)
regex_token_iterator (C++11)	iterates through the specified sub-expressions within all regex matches in a given string or through unmatched substrings (class template)

smart pointers

C++11 provides
`std::unique_ptr`,
`std::shared_ptr`
`std::weak_ptr`.

Concurrency guarantees

`std::auto_ptr` is deprecated.

unique_ptr

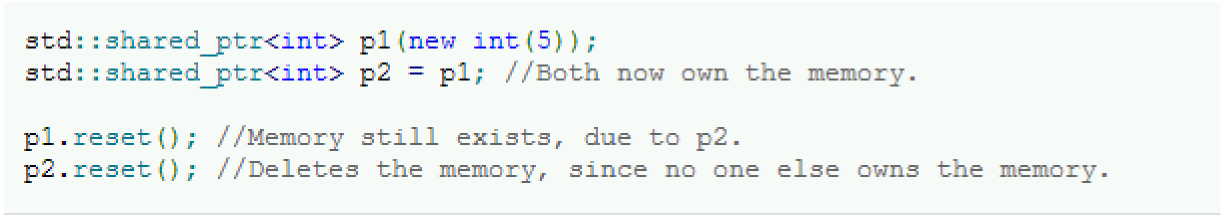
defined in the header `<memory>`

copy constructor and assignment operator
explicitly deleted;

```
std::unique_ptr<int> p1(new int(5));
std::unique_ptr<int> p2 = p1; //Compile error.
std::unique_ptr<int> p3 = std::move(p1); //Transfers ownership. p3 now owns the memory and p1 is rendered invalid.

p3.reset(); //Deletes the memory.
p1.reset(); //Does nothing.
```

shared_ptr

`std::shared_ptr` represents reference-counted ownership of a pointer.



```
std::shared_ptr<int> p1(new int(5));
std::shared_ptr<int> p2 = p1; //Both now own the memory.

p1.reset(); //Memory still exists, due to p2.
p2.reset(); //Deletes the memory, since no one else owns the memory.
```

weak_ptr

Shared_ptr circular references are potentially a problem.

To break up cycles, `std::weak_ptr` can be used to access the stored object.

object will be deleted if the only references to the object are `weak_ptr` references

`weak_ptr` therefore does not ensure that the object will continue to exist, but it can ask for the resource.

```
std::shared_ptr<int> p1(new int(5));
std::weak_ptr<int> wp1 = p1; //p1 owns the memory.

{
  std::shared_ptr<int> p2 = wp1.lock(); //Now p1 and p2 own the memory.
  if(p2) // As p2 is initialized from a weak pointer, you have to check if the memory still exists!
  {
    //Do something with p2
  }
} //p2 is destroyed. Memory is owned by p1.

p1.reset(); //Memory is deleted.

std::shared_ptr<int> p3 = wp1.lock(); //Memory is gone, so we get an empty shared_ptr.
if(p3)
{
  //Will not execute this.
}
```

Wrapper reference

```
g(func, std::ref(i));
```

Wrapper references are useful above all for function templates,

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

```
#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;
}
```

Output(may vary)

```
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

```cpp
#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;
}
```