C++ 11 Features

C++: Procedural, Object Oriented Programming, and Functional

Pei-yih Ting

NTOU CS

Hey, do I really know you?

C++ code snippet

```
auto glambda = [](auto a, auto&& b) { return a < b; };
bool b = glambda(3, 3.14);

auto vglambda = [](auto printer) {
    return [=](auto&&... ts) {
        printer(std::forward<decltype(ts)>(ts)...);
        return [=] { printer(ts...); };
    };
};
auto p = vglambda([](auto v1, auto v2, auto v3)
                        { std::cout << v1 << v2 << v3; });
auto q = p(1, 'a', 3.14); // outputs 1a3.14
q();
```

Huh, C++??

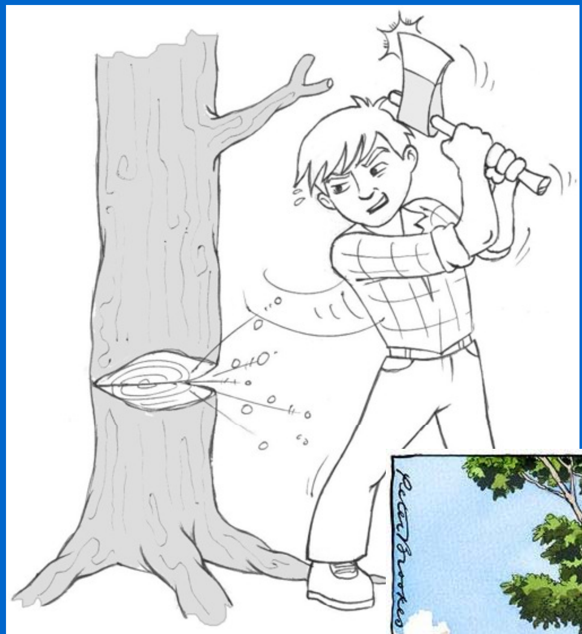
What happened?

"C++11 feels like a new language." – Bjarne Stroustrup

- Look at the bright side, the features in this example are majorly for incorporating functional programming paradigm to make C++ a real monster.
  - At least the object-oriented part is largely unchanged
    No panic! Many features are syntax sugars for you.
- **OK, I lied a little. Move semantics and thread support are important.**

Functional Thinking

- You shall not master STL <algorithm>, <numeric>, <functional> and many C++11 added syntaxes by your procedural or object-oriented intuitions!!!



Can you figure out the way to use a chainsaw in place of an ax if what you have ever seen is an ax in working!!!!

Paradigm shift!!!

Functional Thinking (cont'd)

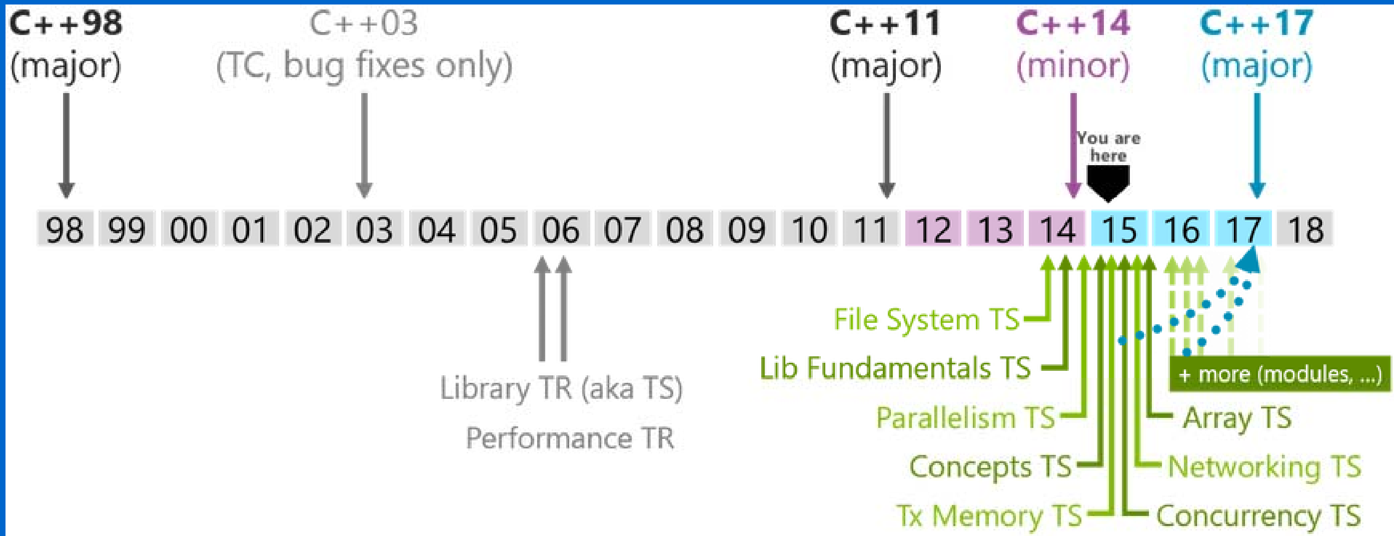
- *Functional Thinking: Paradigm over Syntax*, Neal Ford, 2014
- *Becoming Functional: Steps for transforming to a functional programmer*, Joshua Backfield, 2014
  - Cede control to the language/runtime (automatic garbage collection, automatic parallelism, iteration of containers, control of iterations)
  - Prefer higher-level abstractions and key data structures (highly optimized operations), customized with function objects
  - Common building blocks: filter, fold/reduce, map, closures
  - Avoid mutable state (the moving parts)
  - Stop thinking of low-level details of implementation and start focusing on the problem domain and on the intermediate results of each steps (gradual transformation of input data toward the overall result)

  - Scala, Clojure, Java8, Groovy, Functional Java, Ruby on Rail, Python

C++ Evolution

starting from **C**, **Simula**, **ALGOL 68**, **Ada**, **CLU**, **ML**

- 1979: **C with Classes** by Bjarne Stroustrup
- 1983: **C++**
- 1998: **C++98** – ISO/IEC 14882:1998, first standardization
- 2003: **C++03** – ISO/IEC 14882:2003, few corrections
- 2007: **C++TR1** – ISO/IEC TR 19768:2007
- 2011: **C++11** – ISO/IEC 14882:2011 (formerly **C++0x**)
- 2014: **C++14**, minor revision
- Scheduled: **C++17**, major revision



Aims of C++11 Effort

from C++11 FAQ by Bjarne Stroustrup

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

- C++
  - is a better C
  - supports ① data abstraction
    ② OOP
    ③ generic programming
- C++11 effort tried to strengthen
  - Make C++ a better language for system programming and library building
  - Make C++ easier to teach and learn
    - Through increased uniformity, stronger guarantees, and facilities supporting novices

Specific Design Aims

from C++11 FAQ

- Maintain stability and compatibility
- Prefer libraries to language extensions
- Prefer generality to specialization
- Support both experts and novices
- Increase type safety
- Improve performance and ability to work directly with hardware
- Fit into the real world

null pointer constant

C++98, C++03	C++11

void foo(char*);
void foo(int);
 	
foo(NULL); // calls second foo
	void foo(char*);
	void foo(int);
	
	foo(**nullptr**); // calls first foo

Note: not just a numeric constant, it has its own type - decltype(nullptr) is std::nullptr_t

standard types

C++98, C++03	C++11
sizeof(int) == ?	**#include <cstdint>**
sizeof(char) == ? 1 byte	**using namespace std;**
sizeof(wchar_t) == ?	**int8_t**
sizeof(char) ≤ sizeof(short) ≤ sizeof(int) ≤ sizeof(long)	**uint8_t**
	int16_t
	uint16_t
	int32_t
	uint32_t
	int64_t
	uint64_t

Note: C99 stdint.h

Raw string literals

C++98, C++03	C++11
string test="C:\\path\\file1.txt"; cout << test << endl;	string test=**R"(**C:\path\file1.txt**)"**; cout << test << endl;
C:\path\file1.txt	C:\path\file1.txt
string test="1st\n2nd\n3rd";	string test=**R"(**1st\n2nd\n3rd**)"**;
1st 2nd 3rd	1st\n2nd\n3rd
	string test=**R"(**1st 2nd 3rd**)"**; cout << test << endl;
	1st 2nd 3rd

Type inference: auto

C++98,C++03	C++11
map<string,string>::iterator it = m.begin();	auto it = m.begin();
const double param = config["param"];	auto const param = config["param"];
singleton& s = singleton::instance();	auto& s = singleton::instance();

To use or not to use?! Readability issue?

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

prefer using auto
auto p = new T(); // the class name T is in the expression, no need to repeat it.
auto p = make_share<T>(arg1); // share_ptr<T>
auto my_lambda = [](){}; // std::function<void(void)>, all the definition is here
auto it = m.begin();
// instead of map<string, list<int>::iterator>::const_iterator it = m.cbegin();

Type inference (cont'd)

- Function argument is insufficient to specify the template

```
template <typename BuiltType, typename Builder>
void makeAndProcessObject (const Builder& builder) {
   BuiltType val = builder.makeObject();
   // …
}
MyObjBuilder builder;
makeAndProcessObject<MyObj>(builder);
```

- Simplified design

```
template <typename Builder>
void makeAndProcessObject (const Builder& builder) {
   auto val = builder.makeObject();
   // …
}
MyObjBuilder builder;
makeAndProcessObject(builder);
```

Type inference (cont'd)

- Function argument is insufficient to specify the template

```
template <typename BuiltType, typename Builder>
Buildtype makeAndProcessObject (const Builder& builder) {
    BuiltType val = builder.makeObject();
    // …
}
MyObjBuilder builder;
MyObj obj = makeAndProcessObject<MyObj>(builder);
```

- Not working design

```
template <typename Builder>
auto makeAndProcessObject (const Builder& builder) { // compiler error
    auto val = builder.makeObject();
    // …
}
MyObjBuilder builder;
MyObj obj = makeAndProcessObject(builder);
```

Suffix return type

- Traditional function definition:

```
int multiply(int x, int y);
```

- new (optional) suffix return type (trailing return type):

```
auto multiply(int x, int y) -> int;
```

- What do you get from this new syntax?

```
class Person {
public:
    enum PersonType {Adult, Child, Senior};
    …
    PersonType getPersonType();
};
Person::PersonType Person::getPersonType() {
    …
}
```

required

```
auto Person::getPersonType() -> PersonType () {
    …
}
```

decltype(): auto’s not evil twin

- auto: declare a variable with its *type* compiler infers from the context
- decltype: extract the *type* from a variable

```
int x = 3;
decltype(x) y = x; // auto y = x;  or  int y = x;
```

- **auto** + **decltype()** + **suffix return type**

  ==> Working design

```
template <typename Builder>
auto makeAndProcessObject (const Builder& builder) ->
                              decltype(builder.makeObject() ) {
   auto val = builder.makeObject();
   // …
}
MyObjBuilder builder;
MyObj obj = makeAndProcessObject(builder);
```

decltype + Suffix return type

```
template <typename T, typename U>
??? add(T x, U y) {
    return x+y;        // depends on T and U,
}                      // the type might be T or U
```

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

```
template <typename T, typename U>
decltype(*(T*)nullptr+*(U*)nullptr)  add(T x, U y) {  // works but ugly
    return x+y;
}
```

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

References, Pointers, Const

- How does auto handle references?

```
int &foo();
…
auto bar = foo(); // int& or int?
```

auto defaults to by-value for references

- Might not be what you want!! But you can declare it explicitly!

```
int &foo();
…
auto bar1 = foo();              // int bar1 = foo();
auto &bar2 = foo();            // int &bar2 = foo();
const auto &bar3 = foo(); // const int &bar3 = foo();
```

```
const int *foo();
…
auto bar1 = foo;                              // const int *bar1 = foo;
const auto *const bar2 = foo(); // const int *const bar2 = foo;
```

More decltype

```
int add(int a, int b) { return a+b; }
int main() {
    int i = 4, *p;
    const int j = 6, &k = i;
    int a[5];
    decltype(i) var1;                 // int var1;
    decltype(0) var2;                 // int var2;
    decltype(2+3) var3;               // int var3;
    decltype(i=1) var4 = i;           // int var4 = i;
    decltype(j) var5 = 1;             // const int j var5 = 1;
    decltype(k) var6 = j;             // const int &var6 = j;
    decltype(a) var7;                 // int var7[5];
    decltype(a[3]) var8 = i;          // int var8 = i;
    decltype(*p) var9 = i;            // int var9 = i;
    decltype(add) *fptr = add;  (*fptr)(i, j);
    return 0;
}
```

const decltype(i) var1; // error

i=1 is not evaluated, i is still 4

in-class member initializers

C++98, C++03	C++11

class A {
public:
  A(): a(4), s("test1") { }
  A(int _a): a(_a), s("test1") { }
  A(C c): a(4), s("test1") { }
private:
  int a;
  string s;
};
	class A {
	public:
	  A() { }
	  A(int _a): a(_a) { }
	  A(C c) { }
	private:
	  int a **= 4**;
	  string s **= "test1"**;
	};

Note: prior to C++11, only const static is allowed in-class initialization

class A { … const static int cs=1000; … };

delegating constructors

C++98, C++03	C++11

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

override

C++98, C++03	C++11

class Base {
public:
    virtual void foo();
    void bar();
    virtual void baz() const;
};
class Derived : public Base {
public:
    void foo(); // **Overriding**
    void bar(); // **Hiding**: Base::bar and
      // Derived::bar are not virtual
    void baz(); // **Hiding**:
      // Derived::baz does not override
      // Base::baz (signature mismatch)
};
	Java @override annotation
	
	class Base {
	public:
	    virtual void foo();
	    void bar();
	    virtual void baz() const;
	};
	class Derived : public Base {
	public:
	    void foo() **override**; // Compile OK:
	    void bar() **override**; // Compile **Error**:
	      // Base::bar is not virtual
	    void baz() **override**; // Compile **Error**:
	      // Derived::baz does not override
	      // Base::baz (signature mismatch)
	};

final

JAVA	C++11

final class Base1{}
 	
class Derived1 extends Base1 {
// compile error
}
 	
class Base2 {
    public **final** void f() {};
}
 	
class Derived2 extends Base2 {
    public void f() {}; // compile error
}
	class Base1 **final** {};
	
	class Derived1: public Base1 {
	// compile error
	};
	
	class Base2 {
	public:
	    virtual void f() **final**;
	};
	
	class Derived2: public Base2 {
	public:
	    void f(); // compile error
	};

control of defaults: default and delete

C++11

```
class A {
public:
    A() = default;                              // deleted otherwise
    A(const A&) = delete;                       // disallow copy
    A& operator=(const A &) = delete;           // disallow assignment
};
class B {
public:
    B(float);               // can initialize with a float
    B(int) = delete;        // but not with an int
};
class C {
public:
    virtual ~C() = default; // better efficiency?
};
```

type_traits

C++11	value
#include <type_traits>	
using namespace std;	
struct A {};	
struct B { virtual void f(){} };	
struct C: B {};	
has_virtual_destructor<int>::value	0
is_polymorphic<int>::value	0
is_polymorphic<A>::value	0
is_polymorphic<B>::value	1
is_polymorphic<C>::value	1
typedef int mytype[][24][60];	
extent<mytype,0>::value	0
extent<mytype,1>::value	24
extent<mytype,2>::value	60
is_copy_constructible <A>::value	1
is_nothrow_move_constructible <B>::value	1

std::function

- Class template **std::function** is a general-purpose polymorphic function wrapper. Instances can store, copy, and invoke any Callable ***target*** – ***function pointers***, ***lambda expressions***, ***bind expressions***, ***functors***, ***pointers to member functions*** and pointers to data members (?).

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

```
int sum(int a, int b) { return a+b; }
function<int (int, int)> fsum1 = &sum; cout << fsum1(4,2); << endl;
function<decltype(sum)> fsum2 = &sum; cout << fsum2(4,2) << endl;
int (*fsum3)(int, int) = &sum; cout << fsum3(4,2) << endl;
decltype(sum) *fsum4 = &sum; cout << fsum4(4,2) << endl;
```

```
typedef int FN(int, int);  typedef int (*FPTR)(int, int);
function<FN> fsum5 = &sum; cout << fsum5(4,2) << endl;
FN *fsum6 = &sum; cout << fsum6(4,2) << endl;
FPTR fsum7 = &sum; cout << fsum7(4,2) << endl;
```

```
function<int (int, int)> (*fsumptr) = &fsum1;  cout << (*fsumptr)(4,2) << endl;
decltype(sum) fsum8; // a function prototype, FN fsum8; int fsum8(int,int);
```

std::function for member

```
struct Foo {
    void f(double d, int i)
        { cout << "d=" << d << " i=" << i << endl; }
} foo, *ptr = &foo;
```

```
#include <functional>
using namespace std;
using namespace std::placeholders;
                                   // _1, _2, …
```

<table>
<tr><th>C++98, C++03</th><th>C++11</th></tr>
<tr><td>

```
void (Foo::*fmember1)(double, int)
    = &Foo::f;
(foo.*fmember1)(1.2, 789);
(ptr->*fmember1)(1.2, 789);

typedef void (Foo::*FP)(double, int);
FP fmember2 = &Foo::f;
(foo.*fmember2)(2.2, 789);
```

</td><td>

```
function <void (Foo&, double, int)> fmember1 =
    mem_fn(&Foo::f); fmember1(foo, 1.2, 31);
function <void (Foo*, double, int)> fmember2 =
    mem_fn(&Foo::f); fmember2(&foo, 2.2, 31);
function <void (double, int)> fmember3 =
    bind(&Foo::f, foo, _1, _2); fmember3(3.2, 31);
//function <void (double)> fmember4 =
auto fmember4 =
    bind(&Foo::f, &foo, _1, 123); fmember4(4.2);
```

</td></tr>
</table>

Currying/Partial application in functional programming

std::bind

C++11	output

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); std::placeholders::_1
	
	6/1 = 6
	6/2 = 3
	6/3 = 2

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);
	
	1/6 = 0.166
	2/6 = 0.333
	3/6 = 0.5

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);
	
	1/6 = 0.166
	2/6 = 0.333
	3/6 = 0.5

function<float(void)> oneSixth = bind(div, 1, 6); cout << "1/6 = " << oneSixth();	 1/6 = 0.166

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

lambda closure

C++98, C++03	C++11
see code below	see code below

C++98, C++03

```
#include <vector>
#include <iostream>
using namespace std;
class functor {
public:
    functor(int& _a): a(_a) { }
    bool operator()(int x) const {
        return a == x;
    }
private:
    int &a;
};
int main() {
    vector<int> v = {1, 2, 42, 3, 42};
    int a = 42;
    cout << count_if(v.begin(), v.end(),
        functor(a)) << endl;
}
```

C++11

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

In functional programming paradigm, a **closure** is a function that carries an implicit binding to all the variables referenced within it. In other words, the function encloses a context around the things it references.

Lambdas are good friends of STL algorithms - functional

lambda closure (cont'd)

C++11	test scope	lambda scope

void test() {
    int x = 4;
    int y = 5;
    [&](){x = 2;y = 2;}();
    [=]() mutable{x = 3;y = 5;}();
    [=,&x]() mutable{x = 7;y = 9;}();
}
	
	x=4
	y=5
	x=2 y=2
	x=2 y=2
	x=7 y=2
		
		x=2 y=2
		x=3 y=5
		x=7 y=9

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

Misc. lambda

- Recursive lambda

```
function<int (int)> f = [&f](int n) {
    return n <= 1 ? 1 : n * f(n-1);
}
int x = f(4); // x = 24
```

- Specifying return type

```
[](){ return 1; } // compiler deduces the type
[]() -> int { return 1; } // suffix return type
```

- Capture specification

[]	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
[a,b,&c]	Capture a, b by making a copy, c by reference
[this]	Capture the this pointer of the enclosing class

- lambda is implemented as a functor class with operator() overloaded
  lambda with an empty capture is simplified as a regular function

Example

```
#include <string>
#include <vector>
using namespace std;
class AddressBook {
public:
    // using template to ignore the difference btw functors, function pointers or lambda
    template<typename Func> vector<string> findMatchingAddresses(Func func) {
        vector<string> results;
        for (auto itr = _addresses.begin(), end = _addresses.end(); itr != end; ++itr)
            if (func(*itr)) results.push_back( *itr );
        return results;
    }
private:
    vector<string> _addresses;
};
```

a template has to be declared in a header file

put in cpp file

```
vector<string> findMatchingAddresses
        (function<bool(const string&)> func)
```

```
AddressBook global_address_book;
vector<string> findAddressesFromOrgs() {
    return global_address_book.findMatchingAddresses(
            [] (const string& addr) { return addr.find( ".org" ) != string::npos; });
}
```

std::tuple

python	C++11

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

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

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

Uniform Initialization and std::initializer_list<T>

C++98, 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);
struct X {
    int a[4], b;
    X():b(5) { for (int i=0; i<4; ++i) a[i] = i; }
};
	int a[] {1,2,3,4,5}; // int a[] = {1,2,3,4,5}
	int *b **{new int[5] {1,2,3,4,5}}**;
	vector<int> v **{1,2,3,4,5}**;
	vector<int> *w = **new vector<int> {1,2,3}**;
	struct X {
	    int a[4], b;
	    X():a**{1,2,3,4}**,b**{5}**{}
	};

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

Vector3 normalize(const Vector3& v) {
    float inv_len = 1.f / v.length();
    return Vector3(v.x*inv_len, v.y*inv_len,
                    v.z*inv_len);
}
	Vector3 normalize(const Vector3& v) {
	    float inv_len = 1.f / v.length();
	    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 z = Vector3(4,2,1);	Vector3 x = normalize(**{2,5,9}**); Vector3 y **{4,2,1}**; Vector3 z = **{4,2,1}**;

std::initializer_list<T>

- vector<int> v {1, 2, 3, 4, 5}; // initializer_list<int> object

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

- initializer_list<T> is a lightweight proxy object that provides access to an array of objects of type T. An std::initializer_list object is constructed from the braced-init-list {1,2,3,4,5} and converted to a vector with this ctor
- Examples:

```
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;
```

Behind the scenes

- How does the previous example work?

```
struct Vector3 {
    Vector3(float x, float y, float z): m_x(x), m_y(y), m_z(z) {}
    float length() const { return sqrt(m_x*m_x+m_y*m_y+m_z*m_z); }
    float m_x, m_y, m_z;
};
Vector3 normalize(const Vector3& v) {
    float inv_len = 1.f / v.length();
    return {v.m_x*inv_len, v.m_y*inv_len, v.m_z*inv_len};
}
Vector3 x = normalize({2,5,9});
Vector3 y{4,2,1};
Vector3 z = {4,2,1};
```

There is no initializer_list ctor!

Vector3(float,float,float)

Vector3(float,float,float)

Vector3(float,float,float)

Vector3(float,float,float),
Vector3(const Vector3&)

- 1. If there is an initializer_list ctor, construct an initializer_list object from a braced-init-list, invoke that ctor
  2. try to matched the braced-init-list with every ctor that have the same number of arguments and best matched types

Uniform Initialization

- Initializer-list ctor has higher precedence

```
struct T {
   T(int,int);
   T(initializer_list<int>);
};
T foo {10,20}; // calls T(initializer_list<int>)
T bar (10,20); // calls T(int,int)
```

```
int n;
auto w(n); // int
auto x = n; // int
auto y {n}; // initializer_list<int>
auto z = {n};
```

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

- Most vexing parse

```
struct B { B(){} };
struct A { A(B){} void f(){} };
B test() { return B(); }
int main() {
   A a(B()); // A a(B (*fp)());
   a(test);
}
A a(B (*fp)()) { return A((*fp)()); }
```

```
struct B { B(){} };
struct A { A(B){} void f(){} };
int main() {
   A a1((B())); a1.f();
   A a2 = B(); a2.f();
   A a3(B{}); a3.f();
   A a4{B()}; a4.f();
   A a5({}); a5.f();
}
```

typedef vs. using

C++98, C++03	C++11
`typedef int int32_t;` **`typedef void (*Fn)(double);`**	**`using int32_t = int;`** **`using Fn = void (*)(double);`**

`template <int U, int V>`
`class Type {};`
`typedef Type<42,36> ConcreteType;`
`ConcreteType object1;`
	`template <int U, int V>`
	`class Type {};`
	`using ConcreteType = Type<42,36>;`
	`ConcreteType object1;`

`template<int V>`
`typedef Type<42,V> MyType1;`
`//error: template declaration of 'typedef'`
`//MyType1<36> object;`
	`template <int V>`
	`using MyType = Type<42, V>;`
	`MyType<36> object2;`

`template<int V>`
`struct meta_type {`
`    typedef Type<42, V> type;`
`};`
`typedef meta_type<36>::type MyType;`
`MyType object2;`
	Type alias

explicit conversion operators

C++98, C++03	C++11
struct A { A(int){}; };	struct A { A(int){}; };
struct B { explicit B(int){};	struct B { explicit B(int){};
operator A() {return A(0);}};	**explicit** operator A() {return A(0);}};
void f(A) {}	void f(A) {}
void g(B) {}	void g(B) {}
int main() {	int main() {
A a(1);	A a(1);
f(1); **// silent implicit cast!**	f(1); **// silent implicit cast!**
B b(2);	B b(2);
g(2); **// compiler error: implicit cast**	g(2); **// compiler error: implicit cast**
A a = b; // B->A, A’s copy ctor	**A a = b; // error, implicit cast!**
// silent implicit cast!	**f(b); // error, implicit cast!**
f(b); // B->A, silent implicit cast!	**A a = static_cast<A>(b);**
return 0;	**f(static_cast<A>(b));**
}	return 0;
	}

Eunm class – scoped and strongly typed enums

Underlying type

C++98, C++03	C++11

enum Alert { green, yellow, **red** };
enum Color { **red**, blue };
// error: 'red' : redefinition
Alert a = 7; // error (as ever in C++)
int a2 = red; // ok: Alert->int conversion
int a3 = Alert::red; // error
	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: red not declared
	int a3 = Alert::red; // error
	int a4 = blue; // error: blue not declared
	int a5 = Color::blue;
	// error: not Color->int conversion
	Color a6 = Color::blue; // ok

User-defined literals

C++98, C++03	C++11

123 // int
1.2 // double
1.2F // float
'a' // char
1ULL // unsigned long long
	1.2_i // imaginary
	123.4567891234_df // decimal floating point
	101010111000101_b // binary
	123_s // seconds
	123.56_km // kilometers
	Speed v = 100_km/1_h;
	double operator "" _km(double val) {
	    return 1000.*val;
	}

Application of C++11 User-Defined Literals to Handling Scientific Quantities, Number Representation and String Manipulation

http://www.codeproject.com/Articles/447922/Application-of-Cplusplus11-User-Defined-Literals-t

lvalue vs. rvalue reference

- **lvalue** is an expression that refers an object and persists beyond a simple expression; an lvalue’s address can be taken, a locator value
- an expression is an **rvalue** if it results in a temporary object

```
int a;
a = 1; // expression a is an lvalue, 1 is an rvalue, a=1 is an lvalue
(a=1)=2; // a will be 2
```

```
int x;
int& getRef() {
    return x;
}
getRef() = 4; // getRef() is an lvalue
int *ptr = &getRef();
```

```
int x = 1;
++x = 10; // ++x is an lvalue
```

```
int x;
x + 10; // x+10 is an rvalue
```

C++11

rvalue
lvalue
xvalue
glvalue
prvalue

- **reference** vs. **rvalue reference**:

```
string getName() {
    return "Alex";
}
```

```
string name = getName();
string& name2 = name;
string& name = getName(); // error
const string& name = getName();
string&& name = getName();
const string&& name = getName();
```

Move Semantics

<table>
<tr><th>C++98, C++03</th><th>C++11</th></tr>
<tr><td>typedef vector<float> Matrix;
// 4 ways to handle multiplication result C
void Mul(const Matrix& A, const Matrix& B,
Matrix& C);
// require already created C</td><td rowspan="4">typedef vector<float> Matrix;

Matrix operator*(const Matrix& A,
const Matrix& B);

Matrix A(10000);
Matrix B(10000);
….
Matrix C = A * B;
// no need to manage lifetime of C or
// to delete the returned matrix
// no abstraction or performance penalty</td></tr>
<tr><td>void Mul(const Matrix& A, const Matrix& B,
Matrix* C);
// need to manage lifetime of C, new/delete</td></tr>
<tr><td>Matrix*
operator*(const Matrix& A, const Matrix& B);
// need to delete the returned matrix C</td></tr>
<tr><td>shared_ptr<Matrix>
operator*(const Matrix& A, const Matrix& B);
// no need to manage lifetime manually with
// performance and abstraction penalty</td></tr>
</table>

Move Semantics with rvalue reference

C++11

```
typedef vector<float> Matrix;
Matrix operator*(const Matrix& A,
                 const Matrix& B) {
    Matrix ret(A.size());
    // ret.data = 0x0028fabc
    // ret.size = 100000
    // matrix multiplication => ret.data[i]
    return ret;
    // C.data = ret.data, C.size = ret.size;
    // ret.data = nullptr, ret.size = 0;
} // ~vector<float>()
// delete ret.data;
Matrix A(10000);
Matrix B(10000);
Matrix C = A * B; // vector<float> &&
// C.data = 0x0028fabc, C.size = 100000
```

```
template<typename T>
class vector {
    T* data;
    size_t size;                  rvalue reference
public:
    vector(vector<T>&& rhs): // move ctor
        data(rhs.data), size(rhs.size) {
        rhs.data = nullptr;
        rhs.size = 0;
    }
    ~vector() {
        delete[] data;
    }
    …
};
```

Move Semantics

C++98, C++03	C++11
typedef vector<float> BigObj;	typedef vector<float> BigObj;
void f(BigObj&); // bind to lvalue	void f(BigObj&); // bind to lvalue
	void f(BigObj&&); // bind to rvalue
// test1	// test1
BigObj x = createBigObject(); // copy ctor	BigObj x = createBigObject(); // **move ctor**
f(x);	f(x); // void f(BigObj&);
f(createBigObject()); // compile error	f(createBigObject()); // void f(BigObj&&)
	// test2
	BigObj x = createBigObject(); // **move ctor**
	f(move(x)); // cast lvalue to rvalue
// test3	// test3 // void f(BigObj&&)
BigObj createBigObject() {	BigObj createBigObject() {
BigObj x(100000);	BigObj x(100000);
return x; // copy ctor to construct tmp	return x; // **move ctor to construct tmp**
} // dtor of x	} // dtor of x
BigObj y = createBigObject(); // copy ctor	BigObj y = createBigObject(); // move ctor

Some details

- Without optimization, vc2010, cl /EHsc test.cpp

```
BigObj createBigObject() {
    BigObj x(100000);
    return x; // move ctor to construct a temporary object, a local variable x
} // dtor of x                                is treated as an rvalue in the return statement
BigObj y = createBigObject(); // move ctor to construct y
// dtor of the temporary object
```

- Return value optimization (RVO) - no copy no move : activated when returns a local variable by value
  gcc4.9.2, g++ -std=c++11 test.cpp -o test.exe
  vc2010, cl /EHsc **/O2** test.cpp

```
BigObj createBigObject() {
    BigObj x(100000);
    return x;
}
BigObj y = createBigObject(); // move ctor to construct y
// dtor of x
```

Move Semantics with Member Objects

lvalue expressions inside the move ctor

C++11

`#include <utility> // move()`
`class Wrapper {`
`public:`
`    Wrapper(Wrapper&& src) // move ctor`
`        : m_data(src.m_data),`
`          m_meta(std::move(src.m_meta)){`
`            src.m_data = nullptr;`
`    }`
`    …`
`private:`
`    int *m_data;`
`    Meta m_meta;`
`};`
	`class Meta {`
	`public:`
	`    Meta(Meta&& src) // move ctor`
	`        : m_name(std::move(src.m_name)),`
	`          m_size(src.m_size) {`
	`    }`
	`    …`
	`private:`
	`    string m_name;`
	`    int m_size;`
	`};`

std::move() casts an lvalue expression to an rvalue expression

Generalized constant Expression: constexpr

- Allows programs to take advantage of compile-time computation – programming at compile time
- Before a function, a member function, or a variable (object) with initialization
- constexpr implies const automatically

```
constexpr int multiply(int x, int y) { return x*y; }
const int val = multiply(10,10);
```

```
class Circle {                    ----------------inside Circle.h
public:
   constexpr Circle(int x, int y, int radius):
      m_x(x),m_y(y),m_radius(radius) {}
   constexpr double getArea() {
      return m_radius * m_radius * (atan(1.)*4.);
   }
private: int m_x, m_y, m_radius;
};
const Circle c(0,0,10);
const double area = c.getArea();
```

- Strict rules for defining a **constexpr** function
  1. Only a single return statement (ctor must have empty body, except typedef, static_assert, using)
  2. Only call other constexpr functions
  3. Only reference const global variables (member variables of a const object is OK)

constexpr

C++98, C++03

```
template<int N>
struct Fib {
    enum {
        value = Fib<N-1>::value + Fib<N-2>::value
    };
};
template <> struct Fib<2> {
    enum { value = 1 };
};
template <> struct Fib<1> {
    enum { value = 1 };
};
cout  << Fib<15>::value; // compile time
int a; Fib<a>; // error, a cannot  appear in a
                    //         constant-expression
```

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
```

template metaprogramming: https://en.wikipedia.org/wiki/Template_metaprogramming

static_assert

C++11

```
template <typename T>
void f(T v) {
    static_assert(sizeof(v)==4, "v must have size of 4 bytes");
    …
}
void g() {
    int64_t v; // 8 bytes
    f(v);
}
```

vs2010/2012 output: Error C2338: v must have size of 4 bytes

gcc4.9.2 output: error: static assertion failed: v must have size of 4 bytes

Range-based for, begin, end

C++98, C++03	C++11
`#include <vector>`	`#include <vector>`
`#include <algorithm>`	`#include <algorithm>`
`vector<int> v;`	`vector<int> v;`
`vector<int>::iterator iter;`	
`for (iter= v.begin(); iter != v.end(); ++iter)`	`for (auto d: v)`
`    total += *iter;`	`    total += d;`
`sort(v.begin(), v.end());`	`sort(begin(v), end(v));`
`int a[] = {3,2,1,4,5};`	`int a[] = {3,2,1,4,5};`
`for (int i; i<sizeof(a)/sizeof(int); i++) a[i]++;`	`for (auto &i: a) i++;`
`sort(a, a+sizeof(a)/sizeof(int));`	`sort(begin(a), end(a));`

- Strings, arrays, and all STL containers can be iterated over with the ranged-based for loop

Range-based for

- Make your own data structure (container) ready to be a range
- Provide the following 5 member or stand-alone functions
  1. begin() and end() for the container
  2. operator*, operator!=, prefix operator++ for the iterator
- Example:

```
class IntVector;
class Iter {
public:
   Iter(const IntVector *p_vec, int pos):
      _pos(pos), _p_vec(p_vc) {}
   bool operator!=(const Iter& rhs) const
      { return _pos != rhs._pos; }
   int operator*() const;
   const Iter& operator++()
      { ++_pos;  return *this; }
private:
   int _pos;
   const IntVector *_p_vec;
};
```

```
class IntVector {
public:
   int get(int idx) const { return _data[idx]; }
   void set(int idx, int val) { _data[idx] = val; }
   Iter begin() const { return Iter(this,0); }
   Iter end() const { return Iter(this, 100); }
private:
   int _data[100];
};
```

```
int Iter::operator*() const
   { return _p_vec->get(_pos); }
```

```
IntVector v;
for (int i=0; i<100; ++i) v.set(i,i);
for (auto i: v) cout << i << endl;
```

Smart Pointers

- ✧ C++98, C++03's smart pointer: auto_ptr
- ✧ C++11's smart pointers: unique_ptr, shared_ptr and weak_ptr
- ✧ **Problems**: In a medium/large-scale object-oriented application project developed by a group of programmers, efficient memory usages might NOT be as critical as in a hardware driver or in a heavily-used part of operating system. But **memory leakages** are lethal to a software that is to be executed on devices which keep running for weeks/months. People tend to forget/evade the labor of cleaning up. Many programmers are expecting C++ standard to adopt something like garbage collector in Java. Is it possible to let go the control of memory usage while keeping the efficiency of C++?
- ✧ **Goals**: behave like built-in (raw) pointers but manage object's lifetime such that a programmer can either "new without worrying about when to delete" or "no naked new"

unique_ptr

- unique_ptr is an enhancement of C++03's auto_ptr, manages the resource with unique ownership -- only one smart pointer owns the resource at a time, when the smart pointer is destroyed, the owned resource is automatically released.
- **auto_ptr** implements copy and assignment with implicit ownership transfer due to the lack of *move semantics* in C++98/03. The compiler allow you to pass an auto_ptr by value to a function, the original auto_ptr would lose the ownership and the managed resource is going to be deleted as the function exits unless another auto_ptr is returned back. auto_ptr cannot manage an array and cannot be used in a container.
- **unique_ptr** disables copy and assignment, allow only move and move assignment with move semantics, support management of array resource, and can be used in a container

unique_ptr (cont'd)

```
#include <memory>
using namespace std;
class Thing { … };
void foo() {
    unique_ptr<Thing> p(new Thing);
    p->do_something();
    some_other_things(); // might throw an exception
} // p is destroyed; dtor deletes the Thing
```

```
void foo(unique_ptr<Thing> q) { … } // compiler error
void foo(unique_ptr<Thing>& q) { … }
void foo(unique_ptr<Thing>&& q) { … } // ownership transfer with temporary rvalue
```

```
unique_ptr<Thing> create() { unique_ptr<Thing> local(new Thing); return local; }
unique_ptr<Thing> p(create()); // move ctor to construct p and then dtor of local
p = create(); // move assignment (delete original p), and owns the 2nd Thing, dtor
```

```
unique_ptr<Thing> p2; p2 = p; // compiler error
unique_ptr<Thing> p2; p2 = move(p); // explicit move assignment
unique_ptr<Thing> p3(p); // compiler error
unique_ptr<Thing> p3(move(p)); // explicit move ctor
```

Important Practices with SP

- If smart pointers are used in a project, several important restrictions should be complied with.

  1. Only use smart pointers to manage objects allocated with new operator. Do not point to objects on the stack.
  2. Only one manager object for each managed object. A new object should be given to shared_ptr or unique_ptr immediately. Any other shared_ptr/weak_ptr/unique_ptr are copied or assigned from the first shared_ptr/unique_ptr.
  3. Avoid using raw pointers on those objects being managed by smart pointers (do not use shared_ptr::get()); otherwise it is too easy to have problems with dangling pointers or double deletions.
  4. Do not new/delete a shared_ptr/weak_ptr/unique_ptr

unique_ptr for Array

```
class Bar {
public:
    Bar(float b): bb(b) {
        cout << "Bar(float) b="
                << b << endl;
    }
    ~Bar() {
        cout << "~Bar()" << endl;
    }
    float bb;
};
```

```
unique_ptr<int []> foo(new int[5]);
for (int i=0; i<5; ++i) foo[i] = i*i;
for (int i=0; i<5; ++i)
  cout << "foo[" << i << "] = " << foo[i] << endl;
```

```
foo[0] = 0
foo[1] = 1
foo[2] = 4
foo[3] = 9
foo[4] = 16
```

deleter

```
unique_ptr<Bar* [], function<void(Bar**)>>
    boo(new Bar*[4], [](Bar** boo_ptr) {
        for (int i=0; i<4; ++i) delete boo_ptr[i];
        delete[] boo_ptr;
    });

for (int i=0; i<4; ++i) boo[i] = new Bar(i * 0.5f);
for (int i=0; i<4; ++i)
    cout << "bar[" << i << "] = " << boo[i]->bb << endl;
```

```
Bar(float) b=0
Bar(float) b=0.5
Bar(float) b=1
Bar(float) b=1.5
bar[0] = 0
bar[1] = 0.5
bar[2] = 1
bar[3] = 1.5
~Bar()
~Bar()
~Bar()
~Bar()
```

Misc unique_ptr

- File managing

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

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

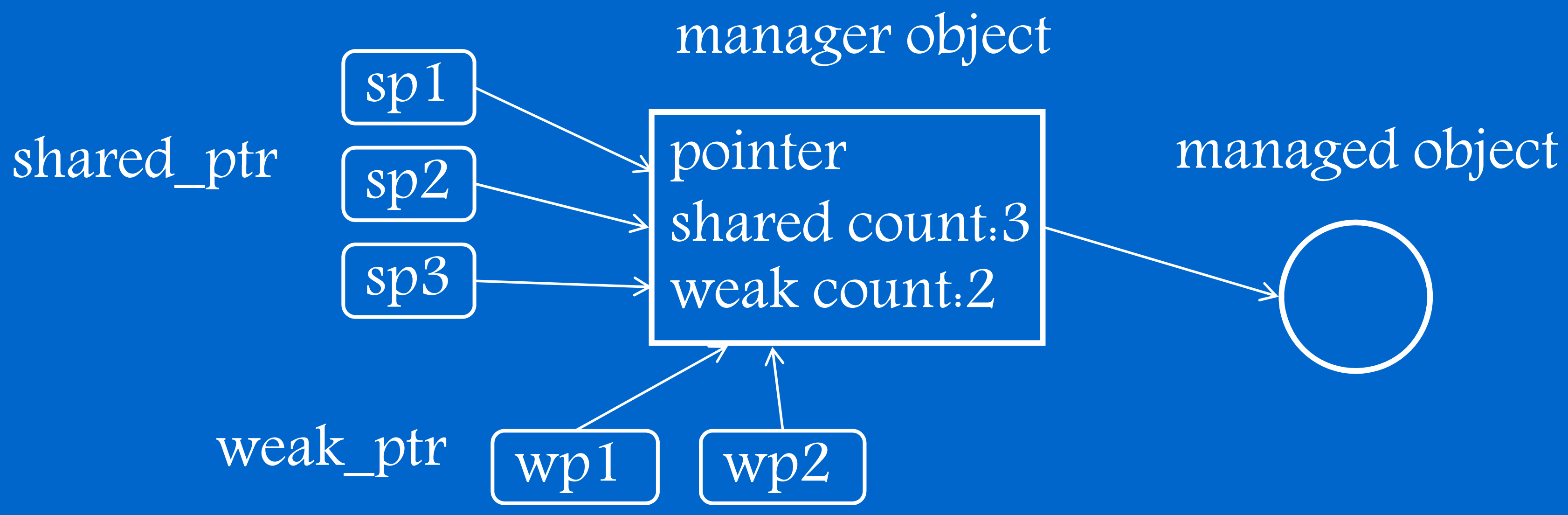
- Allocate the managed object and the manager object with a single new

```
auto p = make_unique<int []>(5); // C++14
for (int i=0; i<5; ++i) p[i] = i;
```

Also make “no naked new” possible

shared_ptr and weak_ptr

- Management of resources with *unique ownership* sacrifices a great deal of efficiency provided by pointers, i.e. a single copy of resource can be referred to by many pointers without physical copying the resource.
- shared_ptr and weak_ptr manage the sharing of resources with reference counts



- weak_ptr observes an object but does not influence its lifetime

share count and weak count

```
void test()
{
    shared_ptr<int> p(new int(42));             share count=1, weak count=0
    {
        shared_ptr<int> x = p;                  share count=2, weak count=0
        {
            shared_ptr<int> y = p;              share count=3, weak count=0
        }                                       share count=2, weak count=0
    }
    // weak_ptr is usd to break reference-count cycles
    weak_ptr<int> wp = p;                       share count=1, weak count=1
    shared_ptr<int> ap = wp.lock();             share count=2, weak count=1
    {
        shared_ptr<int> y = ap;                 share count=3, weak count=1
    }                                           share count=2, weak count=1
}                                     ap dtor   share count=1, weak count=1
                                      wp dtor   share count=1, weak count=0
                                      p dtor    share count=0, weak count=0
                                      the managed int is deleted
```

shared_ptr

```
#include <memory>
using namespace std;
class Thing { … void do_something(); … };
ostream& operator<< (ostream&, const Thing &) { … }
void foo() {
    shared_ptr<Thing> p(new Thing);
    p->do_something();
    some_other_things(); // might throw an exception
} // p is destroyed; dtor deletes the Thing
```

```
shared_ptr<Thing> foo(shared_ptr<Thing> q) { … } // call by value and return by value
```

```
void foo() {
    shared_ptr<Thing> p1(new Thing);          // 1st Thing
    shared_ptr<Thing> p2 = p;                 // p1 and p2 shares the ownership
    shared_ptr<Thing> p3(new Thing);          // 2nd Thing
    p1 = foo(p2);                             // p1 may no longer point to 1st Thing
    p3->do_something();   cout << *p2 << endl;
    p1.reset();                               // decrement count, delete if last
    p2 = nullptr;
} // p3 is destroyed; dtor deletes the Thing
```

shared_ptr (cont'd)

- No implicit conversion btw a raw pointer and a shared_ptr

```
Shared_ptr<Thing> sp(new Thing);
Thing *raw_ptr = sp; // compile error, raw_ptr = sp.get(); is ok but not a good idea
sp = raw_ptr; // compile error
```

- shared_ptr can refer to classes in a class hierarchy in the same way as built-in pointers

```
Derived* dp1 = new Derived;
Base *bp1 = dp1, *bp2(dp1);
```

```
shared_ptr<Derived> dp1(new Derived);
shared_ptr<Base> bp1 = dp1, bp2(dp1);
```

- Casting shared_ptrs

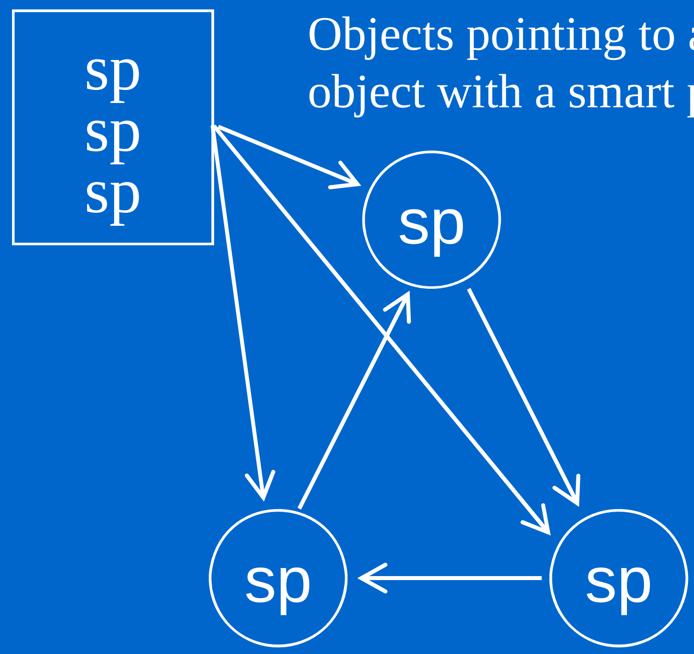
```
shared_ptr<Base> base_ptr(new Base);
shared_ptr<Derived> derived_ptr;
// if static_cast<Derived *>(base_ptr.get()) is valid, so is the following
derived_ptr = static_pointer_cast<Derived>(base_ptr);
```

- Testing and comparing shared_ptrs
  - ==, !=, < behaves like operators between built-in pointers
  - if (sp) will test true if sp is pointing to an object

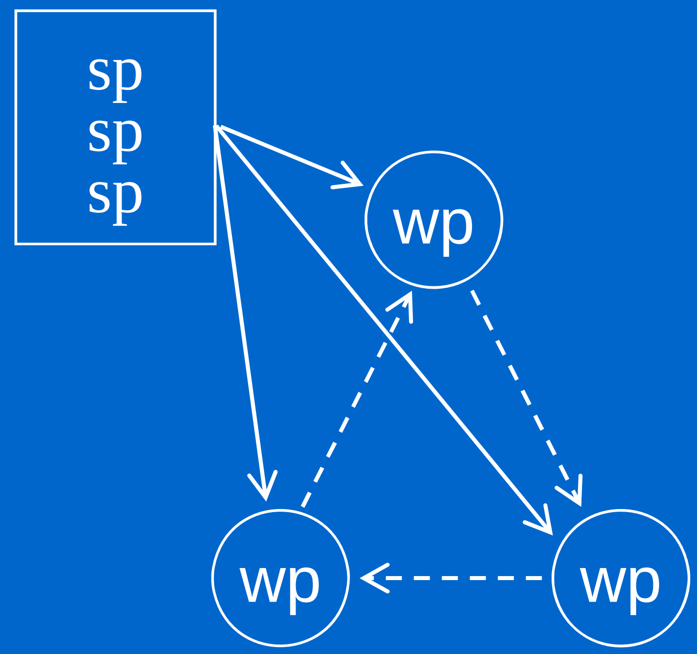
Cycle of objects

✧ A problem with reference-counted smart pointers is that if there is a ring, or cycle of objects that have smart pointers to each other, they keep each other alive – they won’t get deleted even if no other objects are pointing to them from outside of the ring.

container of
smart pointers



Objects pointing to another
object with a smart pointer



Variadic templates

C++98, C++03	C++11
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);	template <class ...T> void f(T... args) {} f("test", 42, 's',12.f);

Variadic templates

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');
```

parameter pack

unpack param

Compiler expansion

```
print_list(first=42, …rest="hello",2.3,'a')
    print_list(first="hello", …rest=2.3,'a')
        print_list(first=2.3, …rest='a')
            print_list(value='a')
```

Output

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

Variadic templates

<table>
<tr><th>C++11</th><th>C++11</th></tr>
<tr><td>

```
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;
};

cout << count<0,1,2,3,4>::value << endl;
```

</td><td>

```
template<int... Elements>
struct count1 {
    static const int value =
                              sizeof...(Elements);
};

// sizeof…() return the number elements
// in a parameter pack

cout << count1<0,1,2,3,4>::value << endl;
```

</td></tr>
</table>

Tuple definition using variadic templates

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

template<> class tuple<> {};

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

MyTuple<int,double,char> x;
```

std::string

- string => signed integer

```
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);
```

- string => unsigned integer

```
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);
```

- string => floating point

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

- (un)signed/decimal integer => string/wstring

```
to_string
to_wstring
```

std::array

- **std::array<T>** is a very thin wrapper around C++ arrays, with the primary purpose of hiding the pointer from the user of the class. It encapsulate a statically-sized array which cannot grow or shrink. It provides much of the STL-related functionality of std::vector<T> and also stores the size (length).

C++98, C++03	C++11
char arr1[] = "xyz"; // '\0' at the end int arr2[] = {2112, 90125, 1928};	array<char, 3> arr1 = {'x', 'y', 'z'}; array<int, 3> arr2 = {2112, 90125, 1928};

int* x = arr2; //ok	int* x = arr2; // error
	x = arr2.data(); // ok

cout << sizeof(arr1) - 1 << endl; cout << sizeof(arr2) / sizeof(int) << endl;	cout << arr1.size() << endl; cout << arr2.size() << endl;
arr2[-42] = 36; // oops	arr2.at(-42) = 36; arr2[-42] = 36; //throws std::out_of_range exception
for (i=0; i<3; i++) arr2[i] = 0;	foreach (begin(arr2),end(arr2),[](int &x){x=0;} for (int &x:arr2) { x=0; }

std::regex

- Prior to C++11: vc2008, header <regex>, namespace std::tr1
- C++11: vc2010, g++ 4.9.2, header <regex>, namespace std
- **6 regular expression flavors**: ECMAScript, basic(POSIX BRE), extended(POSIX ERE), grep, egrep, awk
- **Defining regular expression objects**
  regex re1("\\S+"); // one or more non-white space chars
  regex re2("Regular Expressions", regex_constants::icase);
  regex re3("(\\w{7,})"); // 7 or more words, i.e. [_[:alnum:]]
  regex re4("Name:(?:\\s*([^;]*);)"); // "Name: Wilson Yung;"
  regex re5("(\\d+)-(\\d+)"); // "1234-567"
- **3 basic operations**, greedy
  regex_match(subject, results, target): match entire subject string
  regex_search(subject, results, target): match partial subject string
  regex_replace(subject, results, target, replacement): replace all matched partial subject strings

regex_match, regex_search

- **bool regex_match(string subject, match_results<T> result, regex target)**
  whether target can match the **entire** subject string

```
string input = "-9876";
smatch result;
regex integer("[\\+-]?[[:digit:]]+");
If (regex_match(input, result, integer))
    cout<<"integer " << result.position() << ' ' << result.length() << endl;
```

integer 0 5

- **bool regex_search(string subject, match_results<T> result, regex target)**
  whether target can match **any part** of the subject string

```
string input = " abc -987";
smatch result;
If (regex_search(input, result, regex("[\\+-]?\\d+");))
    cout<<"integer " << result.position() << ' ' <<
              result.length() << "'" << result.str() << "'" << endl;
```

integer 5 4 "-987"

"(\\d+)-(\\d+)"
on 1234-567
smatch::str(i)
are matched
subpatterns:
1234 and 567

regex_iterator

- Get the matched results one-by-one

typedef regex_iterator<string::const_iterator> **sregex_iterator**;

```
string s = "There are some oddities in the perspective "
           "with which we see the world.";
regex word_re("\\S+"); // "(\\S+)" has the same result
auto words_begin = sregex_iterator(s.begin(), s.end(), word_re);
auto words_end = sregex_iterator();

cout << "Found " << distance(words_begin, words_end) << " words\n";

const int N = 6;
cout << "Words longer than " << N << " characters: ";
for (sregex_iterator i = words_begin; i != words_end; ++i) {
   string match_str = i->str();
   if (match_str.size() > N) cout << match_str << ' ';
}
```

```
Found 13 words
Words longer than 6 characters:  oddities  perspective
```

regex_replace

```
try
{
    regex long_word_regex("(\\w{6,})");
    string new_s = regex_replace(s, long_word_regex, "[$&]");
    cout << new_s << '\n';
}
catch (regex_error& e)
{
    // Syntax error in the regular expression
}
```

There are some [oddities] in the [perspective] with which we see the world.

$& or $0 is the whole matched string

$1, …, $9 are the strings matched by the first 9 capture groups

Misc. Utilities in STL

- std::chrono,   #include <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,   #include <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)

Misc. Utilities (cont'd)

```
#include <iostream> // cout
#include <ctime> // time_t, ctime
#include <ratio> // ratio
#include <chrono> // duration, system_clock::now, time_point, to_time_t
int main() {
    using std::chrono::system_clock;
    std::chrono::duration<int,std::ratio<60*60*24>> one_day(1);
    system_clock::time_point today = system_clock::now();
    system_clock::time_point tomorrow = today + one_day;
    std::time_t tt;

    tt = system_clock::to_time_t(today);
    std::cout << "today is: " << ctime(&tt);

    tt = system_clock::to_time_t(tomorrow);
    std::cout << "tomorrow will be: " << ctime(&tt);

    return 0;
}
```

```
today is: Sat Mar 12 08:41:17 2016
tomorrow will be: Sun Mar 13 08:41:17 2016
```

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

Thread Starting

- Starts a thread with a regular function

```
#include <thread>
#include <iostream>
void my_thread_func() {
    std::cout<<"hello"<<std::endl;
}
int main() {
    std::thread t(my_thread_func);
    t.join();
}
```

- Starts a thread with a function object

```
#include <thread>
#include <iostream>
class Functor {
public:
    void operator()() const {
        std::cout << "hello" << std::endl;
    }
};
int main() {
    std::thread t((Functor()));
    t.join();
}
```

- Callable objects – regular function, functor, lambda function
- The function object is copied into the private storage accessible to the new thread

Thread Starting w/ Parameters

- Pass arguments through ctor

```
class Greeting {
   std::string message;
public:
   explicit Greeting(std::string const& _message): message(_message) {}
   void operator()() const { std::cout << message << std::endl; }
};
int main() {
   std::thread t(Greeting("goodbye")); t.join();
}
```

- Pass arguments to a regular function using bind

```
#include <functional>
void greeting(std::string const& message) {
   std::cout << message << std::endl;
}
int main() {
   std::thread t(std::bind(greeting, "hi!")); t.join();
}
```

Thread Starting w/ Parameters

- Pass arguments directly through the thread ctor

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

void write_sum(int x, int y){
   std::cout << x << " + “ << y << " = “ << (x+y) << std::endl;
}

int main() {
   std::thread t(write_sum,123,456);
   t.join();
}
```

  - The arguments are copied into the private storage accessible to the new thread

Thread Starting (cont'd)

- Starts with a member function, arguments by value

<thread>, <iostream>, <string>

```
class SayHello {
public:
    void greeting(const string& message) {  cout << message << endl; }
};
int main() {
    SayHello x;
    std::thread t(&SayHello::greeting, &x, "goodbye");
    t.join();
}
```

- Because object **x** is not copied to the thread's local storage, you need to ensure that **x** outlives the thread.
- shared_ptr is an alternative that uses the heap-allocated object and the reference-counted pointer to ensure the object stays around as long as the thread does.

```
int main() {
    std::shared_ptr<SayHello> p(new SayHello);
    std::thread t(&SayHello::greeting, p, "goodbye");
    t.join();
}
```

Thread Starting (cont'd)

- Starts with a function, arguments by reference

```
class PrintThis {
public:
   void operator()() const {
      std::cout << "this="
         << this << std::endl; }
};
int main() {
   PrintThis x;
   x();
   std::thread t1(std::ref(x));
   t1.join();
   std::thread t2(x);
   t2.join();
}
```

```
this=0x22fe2f
this=0x22fe2f
this=0x9d5ac8
```

```
#include <thread>
#include <iostream>
#include <functional>
void increment(int& i) { ++i; }
int main() {
   int x=42;
   std::thread t(increment, std::ref(x));
   t.join();
   std::cout << "x=" << x << std::endl;
}
```

Quick comparison

<table>
<tr><th>C++11</th><th>Java</th></tr>
<tr><td><pre>#include <thread>
#include <iostream>
int main() {

      std::thread t([]() {
         std::cout <<
         "Hi from thread\n”;});

      t.join();
      return 0;
}</pre></td><td><pre>import …
public class TestThread {
   public static void main(String[] args)
              throws InterruptedException {
      Thread t = new Thread(new Runnable() {
         public void run() {
            System.out.println("Hi from thread");
         } });
      t.start();
      t.join();
   }
}</pre></td></tr>
</table>

Resource Coordination

- 3 threads compete for CPU resource w/ and w/o coordination

```
#include <thread>

void run(int n) {
   for (int 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: 03: 0
3: 1
3: 2
3: 3
3: 4

1: 1
1: 22: 0
2: 1
2: 2
2: 3
2: 4

1: 3
1: 4
```

lose control of CPU in the middle

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

mutex m;
void run(int n) {
   m.lock();
   for (int 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;
}
```

```
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
```

Coordinate memory access

- To protect a shared counter from being read/write by two threads at the same time, you can use std::lock_guard<> to ensure that the mutex is locked for either an increment or a query operation, and make sure that the mutex is unlocked even in the case of exception (i.e. exception-safe)

```
std::mutex m;
unsigned counter=0;
unsigned increment() {
    std::lock_guard<std::mutex> lk(m);
    return ++counter;
}
unsigned query() {
    std::lock_guard<std::mutex> lk(m);
    return counter;
}
```

- lock_guard<mutex> owns the lock from construction to destruction. No ownership transfer and no deferred locking.

Flexible Locking w/ unique_lock()

Functionalities:

- **default ctor**: without an associated mutex
- **deferred-locking ctor**: with an associated unlocked mutex
- **try-lock ctor**: tries to lock a mutex, unlocked if failed
- Cooperate with **std::timed_mutex:** tries to acquire a lock for either a specified time period or until a specified point in time, otherwise leaves the mutex unlocked
- **lock():** lock the associated mutex
- **try_lock()**, **try_lock_for()** and **try_lock_until()**
- **unlock()**: unlock the associated mutex
- **owns_lock()**: check whether the instance owns the lock
- **release()**: release the association with the mutex
- **move ctor** and **move assignment operator**: transfer ownership between instances

std::unique_lock<>

- Unlock the mutex temporarily

```
std::mutex m;
std::vector<std::string> strings_to_process;

void update_strings()
{
    std::unique_lock<std::mutex> lk(m);
    if (strings_to_process.empty())
    {
        lk.unlock();
        std::vector<std::string> local_strings=load_strings();
        lk.lock();
        strings_to_process.insert(strings_to_process.end(),
                                  local_strings.begin(),local_strings.end());
    }
}
```

Dead Lock

- Sometimes it is necessary to hold locks on more than one mutex, because you need to perform an operation on two distinct data, each of which is protected by its own mutex.
- E.g. a bank transfer between two accounts

Bank A, account 1

❶ Account 1 locked, tries to lock account 2

$$

at the same time

$$

❷ Account 2 locked, tries to lock account 2

Bank B, account 2

```
class account {
public:
   void transfer(account& from,account& to,
                          currency_value amount) {
      lock_guard<mutex> lock_from(from.m);
      lock_guard<mutex> lock_to(to.m);
      from.balance -= amount;
      to.balance += amount;
   }
private:
   std::mutex m;
   currency_value balance;
};
```

std::lock()

- To avoid a deadlock, we have to lock the two mutexes together, so that one of the threads acquires) both locks
- std::lock() – locks a number of mutexes at a time (variadic template)

```
void transfer(account& from, account& to, currency_value amount) {
    std::lock(from.m, to.m);
    std::lock_guard<std::mutex> lock_from(from.m, std::adopt_lock);
    std::lock_guard<std::mutex> lock_to(to.m, std::adopt_lock);
    from.balance -= amount;
    to.balance += amount;
}
```

or

```
void transfer(account& from,account& to, currency_value amount) {
    std::unique_lock<std::mutex> lock_from(from.m, std::defer_lock);
    std::unique_lock<std::mutex> lock_to(to.m, std::defer_lock);
    std::lock(lock_from, lock_to);
    from.balance -= amount;
    to.balance += amount;
}
```

std::async and future

- **async()** takes a callable object and arguments, copy the arguments to thread's local storage, spawns a thread to execute the thread function if resource is sufficient (depends on implementation)
- Can specify the launch policy - std::launch::async (starts immediately) or std::launch::deferred (starts as future<T>.get() is called)
  async(policy,function,args)

C++11

```
#include <iostream>
#include <future>
using namespace std;
int Fib(int n) {
   return n<=2?1:Fib(n-1)+Fib(n-2);
}
int main() {
   future<int> result1 = async(Fib, 30);
   int result2 = Fib(40);
   int result = result1.get() + result2;
   cout << "Fib(30) + Fib(40)= "
          << result << endl;
   return 0;
}
```

Divide and Conquer

- std::sync can be used to easily parallelize simple algorithms.

```
#include <iostream>
#include <vector>
#include <numeric>          // accumulate
#include <future>           // async and future
template <typename RAIter>
int parallel_sum(RAIter beg, RAIter end) {
    auto len = end - beg; int sum;
    if (len < 1000) return std::accumulate(beg, end, 0);
    RAIter mid = beg + len/2;
    auto handle = std::async(std::launch::async, parallel_sum<RAIter>, mid, end);
    try { sum = parallel_sum(beg, mid); } catch (...) { handle.wait(); throw; }
    return sum + handle.get();
}
int main() {
    std::vector<int> v(10000, 1);
    std::cout << "The sum is " << parallel_sum(v.begin(), v.end()) << '\n';
}
```

The sum is 10000

C++0x Supports in VC10,VC11

C++11 Core Language Features	VC10	VC11
Rvalue references v0.1, v1.0, v2.0, v2.1, v3.0	v2.0	v2.1*
ref-qualifiers	No	No
Non-static data member initializers	No	No
Variadic templates v0.9, v1.0	No	No
Initializer lists	No	No
static_assert	Yes	Yes
auto v0.9, v1.0	v1.0	v1.0
Trailing return types	Yes	Yes
Lambdas v0.9, v1.0, v1.1	v1.0	v1.1
decltype v1.0, v1.1	v1.0	v1.1**
Right angle brackets	Yes	Yes
Default template arguments for function templates	No	No
Expression SFINAE	No	No

C++0x Supports in VC10,VC11

C++11 Core Language Features	VC10	VC11
Alias templates	No	No
Extern templates	Yes	Yes
nullptr	Yes	Yes
Strongly typed enums	Partial	Yes
Forward declared enums	No	Yes
Attributes	No	No
constexpr	No	No
Alignment	TR1	Partial
Delegating constructors	No	No
Inheriting constructors	No	No
Explicit conversion operators	No	No
char16_t and char32_t	No	No
Unicode string literals	No	No

C++0x Supports in VC10,VC11

C++11 Core Language Features	VC10	VC11
Raw string literals	No	No
Universal character names in literals	No	No
User-defined literals	No	No
Standard-layout and trivial types	No	Yes
Defaulted and deleted functions	No	No
Extended friend declarations	Yes	Yes
Extended sizeof	No	No
Inline namespaces	No	No
Unrestricted unions	No	No
Local and unnamed types as template arguments	Yes	Yes
Range-based for-loop	No	Yes
override and final v0.8, v0.9, v1.0	Partial	Yes
Minimal GC support	Yes	Yes
noexcept	No	No

Supporting C++11/14/17 in VC

C++11 Core Language Features	VS10	VS12	VS13	VS15
Rvalue references v0.1, v1.0, v2.0, v2.1, v3.0	v2.0	v2.1*	v2.1*	v3.0
ref-qualifiers	No	No	No	Yes
Non-static data member initializers	No	No	Yes	Yes
Variadic templates v0.9, v1.0	No	No	Yes	Yes
Initializer lists	No	No	Yes	Yes
static_assert	Yes	Yes	Yes	Yes
auto v0.9, v1.0	v1.0	v1.0	v1.0	Yes
Trailing return types	Yes	Yes	Yes	Yes
Lambdas v0.9, v1.0, v1.1	v1.0	v1.1	v1.1	Yes
decltype v1.0, v1.1	v1.0	v1.1**	v1.1	Yes
Right angle brackets	Yes	Yes	Yes	Yes
Default template arguments for function templates	No	No	Yes	Yes
Expression SFINAE	No	No	No	No
Alias templates	No	No	Yes	Yes
Extern templates	Yes	Yes	Yes	Yes
nullptr	Yes	Yes	Yes	Yes
Strongly typed enums	Partial	Yes	Yes	Yes
Forward declared enums	No	Yes	Yes	Yes

C++11 Core Language Features	VS10	VS12	VS13	VS15
Attributes	No	No	No	Yes
constexpr	No	No	No	Yes
Alignment	TR1	Partial	Partial	Yes
Delegating constructors	No	No	Yes	Yes
Inheriting constructors	No	No	No	Yes
Explicit conversion operators	No	No	Yes	Yes
char16_t/char32_t	No	No	No	Yes
Unicode string literals	No	No	No	Yes
Raw string literals	No	No	Yes	Yes
Universal character names in literals	No	No	No	Yes
User-defined literals	No	No	No	Yes
Standard-layout and trivial types	No	Yes	Yes	Yes
Defaulted and deleted functions	No	No	Yes*	Yes
Extended friend declarations	Yes	Yes	Yes	Yes
Extended sizeof	No	No	No	Yes
Inline namespaces	No	No	No	Yes
Unrestricted unions	No	No	No	Yes
Local and unnamed types as template arguments	Yes	Yes	Yes	Yes
Range-based for-loop	No	Yes	Yes	Yes
override and final v0.8, v0.9, v1.0	Partial	Yes	Yes	Yes
Minimal GC support	Yes	Yes	Yes	Yes

C++11 Core Language Features	VS10	VS12	VS13	VS15
noexcept	No	No	No	Yes

C++11 Core Language Features: Concurrency	VS10	VS12	VS13	VS15
Reworded sequence points	N/A	N/A	N/A	Yes
Atomics	No	Yes	Yes	Yes
Strong compare and exchange	No	Yes	Yes	Yes
Bidirectional fences	No	Yes	Yes	Yes
Memory model	N/A	N/A	N/A	Yes
Data-dependency ordering	No	Yes	Yes	Yes
Data-dependency ordering: function annotation	No	No	No	Yes
exception_ptr	Yes	Yes	Yes	Yes
quick_exit	No	No	No	Yes
Atomics in signal handlers	No	No	No	No
Thread-local storage	Partial	Partial	Partial	Yes
Magic statics	No	No	No	Yes

C++11 Core Language Features: C99	VS10	VS12	VS13	VS15
__func__	Partial	Partial	Partial	Yes
C99 preprocessor	Partial	Partial	Partial	Partial
long long	Yes	Yes	Yes	Yes
Extended integer types	N/A	N/A	N/A	N/A

VC Supports of C++14 Core Language Features

C++14 Core Language Features	**VS10**	**VS12**
Tweaked wording for contextual conversions	Yes	Yes
Binary literals	No	Yes
auto and decltype(auto) return types	No	Yes
init-captures	No	Yes
Generic lambdas	No	Yes
Variable templates	No	No
Extended constexpr	No	No
NSDMIs for aggregates	No	No
Avoiding/fusing allocations	No	No
[[deprecated]] attributes	No	No
Sized allocation	No	Yes
Digit separators	No	Yes

VC Supports of C++17 Core Language Features

C++17 Proposed Core Language Features	VS13	VS15
New rules for auto with braced-init-lists	No	No
Terse static assert	No	No
typename in template template-parameters	No	No
Removing trigraphs	Yes	Yes
Nested namespace definitions	No	No
N4259 std::uncaught_exceptions()	No	No
N4261 Fixing qualification conversions	No	No
N4266 Attributes for namespaces and enumerators	No	No
N4267 u8 character literals	No	No
N4268 Allowing more non-type template args	No	No
N4295 Fold expressions	No	No
await/resume	No	Yes

C++ Compiler Supports

http://en.cppreference.com/w/cpp/compiler_support

feature document C++version	HP aCC	EDG eccp	**GCC**	Intel C++	**MSVC**	IBM XLC++	Sun/Oracle C++	**Embarcadero C++ Builder**	Digital Mars C++	**Clang**	Cray	Portland Group (PGI)
constexpr N2235 C++11			4.6		14.0 vc2005			Yes		3.1		

compiler version