C++11

Chuck Allison
Utah Valley University

The New C++

- New C++11 standard replaces C++98
- Many improvements on:
  - speed!
  - readability
  - high-level programming constructs
- Revisions are already planned for 2014 and 2017!

New Language Features

- Type inference with **auto** and **decltype**
- Aliases with **using** (superset of **typedef**)
- Compile-time initialization with **constexpr**
- Trailing return types
- Lambda Expressions
- Range-based for loops
- Uniform initialization syntax
- **static_assert**
- *rvalue* references
- Variadic templates
- **nullptr** constant

auto, decltype

```
int main() {
  enum {N = 3};

  array<int,N> a{10,20,30}; // uniform initialization syntax
  decltype(a) b;
  auto endb = copy_if(begin(a), end(a), begin(b),
                      bind(less<int>(),_1,15));
  copy(begin(b),endb,ostream_iterator<int>(cout,"\n"));   // 10
}
```

Also notice **array**, *global* **begin/end**, **bind**

Voldemort Types

- ```
  class X {
      class Y {};
  public:
      static Y foo();
  };
  ```

- ```
  auto y = X::foo();   // Unnamed type (locally)
  ```

From Anthony Williams, *C++ Concurrency in Action*

using

```
using intseq = int[5];
intseq a;
for (int i = 0; i < 5; ++i)
    a[i] = 5-i;
for (int i = 0; i < 5; ++i)
    cout << a[i] << ' ';    // 5 4 3 2 1
```

More readable than **typedef**

using and Templates

```
template<typename T>
using smap = map<string,T>;

int main() {
    smap<int> mymap {{"one",1},{"two",2}};
    for (const auto &p: mymap)
        cout << p.first << ',' << p.second << endl;
}

/* Output:
one,1
two,2
*/
```

Note range-based **for** loop and new initialization syntax

constexpr

- **const** = “I won’t change this”
- **constexpr** = “I will evaluate this at compile time”
  - if I can
- `constexpr size_t size = sizeof(int) + sizeof(Foo);`

Trailing Return Type (->)

The expression is *not* evaluated

```
int f(int x) {
    return x*x;
}

template<typename T1, typename T2>
auto min(T1 t1, T2 t2)->decltype(t1+t2) { // Promote to wider type
    return t1 < t2 ? t1 : t2;
}

int main() {
    cout << f(3) << endl;                      // 10
    auto x = min(1.1, 2);
    cout << x << ", " << typeid(x).name() << endl;  // 1.1, d
}
```

Lambda Expressions

```
int main() {
  enum {N = 3};

  array<int,N> a{10,20,30};
  decltype(a) b;
  auto endb = copy_if(begin(a), end(a), begin(b),
                 →   [](int x){return x < 15;});
  copy(begin(b),endb,ostream_iterator<int>(cout,"\n"));   // 10
}
```

Lambda Expressions

- Create function objects (anonymous functions)
- Can capture environment
  - a type of closure (stores copies or references in fun. object)
- Syntax:
  - **[**<capture>**] (**<args>**) {**<body>**} ->** <return type>
  - Return type is optional if body is a single **return** statement
    - Otherwise defaults to **void**
  - The capture directive links to the *calling environment*

Lambda Without Capture

```
int main() {
    vector<int> v{1,2,3,4,5};
    transform(begin(v), end(v), begin(v), [](int n){return n + 10;});
    copy(begin(v), end(v), ostream_iterator<int>(cout, " "));
    cout << endl;

    sort(begin(v), end(v), [](int m, int n){return m > n;});
    copy(begin(v), end(v), ostream_iterator<int>(cout, " "));
    cout << endl;
}

/* Output:
11 12 13 14 15
15 14 13 12 11
*/
```

Lambda With Capture ("Closures")

```
function<int(int)> addx(int x) {
    return [x](int y)->int {return x+y;};
}

int main() {
    auto f = addx(10);
    cout << f(3) << endl;  // 13
}
```

[=] captures *everything* visible by value
[&] captures *everything* by reference (careful!)
Can capture individual variables: **[=x,&y]**…
Globals aren't (and don't need to be) captured

Notice the generalized **function** signature declarator.

Recursive Lambdas

```
int main() {
    function<int(int)> fib =
        [&fib](int n) {return n < 2 ? n : fib(n-1) + fib(n-2);};
    for (auto n: {0,1,2,3,4,5})
        cout << fib(n) << endl;
}

/* Output:
0
1
1
2
3
5
*/
```

Captures the function name

Capturing **this**

```
class Foo {
    string s;
public:
    Foo(const string& s) : s(s) {}
    function<string(void)> greet() {
        return [this]{return "hello " + s;};
    }
};

int main() {
    Foo f("Larry");
    auto g = f.greet();
    cout << g() << endl;     // hello Larry
}
```

this->s

Mutable Lambdas

- Because in the generated function object, **operator()** is **const**

```
void algo(vector<int>& v) {  // Function from Stroustrup 4th
    int count = v.size();
    generate(begin(v), end(v),
        [count]() mutable {return --count;}); // local mods
}

int main() {
    vector<int> v(10);
    algo(v);
    copy(begin(v),end(v),ostream_iterator<int>(cout,"  "));
    cout << endl;
}

/* Output
9 8 7 6 5 4 3 2 1 0
*/
```

Range-based **for** and Higher-dim Arrays

```
int main() {
    int a[][3] = {{1,2,3},{4,5,6},{7,8,9}};
    for (const auto &row: a) {
        cout << "sizeof(row): " << sizeof(row) << endl;
        for (const auto &x: row)
            cout << "sizeof(" << x << "): “
                 << sizeof(x) << endl;
    }
}
```

/ Output:*
sizeof(row): 12
sizeof(1): 4
sizeof(2): 4
sizeof(3): 4
sizeof(row): 12
sizeof(4): 4
sizeof(5): 4
sizeof(6): 4
sizeof(row): 12
sizeof(7): 4
sizeof(8): 4
sizeof(9): 4
**/*

Uniform Initialization Syntax

- Can use {…} for all initializations

-
```
int a[]{1,2,3};
vector<int> v{1,2,3};
v = {4,5,6};                    // Assigning from an initializer_list
v.insert(end(v),{7,8,9});  // Append
Foo foo{"one",1};
map<string,int> m{{"one",1},{"two",2}};
 f({1,2,3});
```

static_assert

```
static_assert(CHAR_BIT == 8,"8 bits per byte required");
static_assert(sizeof(char*) == 4,"32-bit architecture required");

const int N = 10;

void f() {
    static_assert(N > 2,"N must be > 2");
    int a[N];
}

/* Output on 32-bit machine:
/Users/chuck/UVU/3370/CourseCode/static_assert.cpp:4:1: error:
static assertion failed: "32-bit architecture required"
*/
```

Illustrative Example

- See *compose5.cpp*
  - composes an arbitrary number of like-typed, unary functions
  - **auto f = f1(f2(…(fn(x)…))**
- Illustrates:
  - **std::bind**
  - **std::function<…>**
  - **auto**
  - lambda expressions
  - function objects
  - **std::accumulate**
  - **using**

rvalue References

- vs. traditional **lvalue** references
  - use **T&&** syntax
- only bind to *temporaries*
- Purpose:
  - move semantics (efficiency via “stealing resources”)
  - perfect forwarding (type preservation/collapsing)

Move Constructor and Assignment

```
class C {
public:
    C() = default;
    C(const C&) {cout << "copy constructor\n";}
    C(C&&) {cout << "move constructor\n";}
    C& operator=(const C&) {cout << "copy assignment\n";}
    C& operator=(C&&) {cout << "move assignment\n";}
};
```

Note **=default**.
There is also **=delete**.

Example Continued…

```
C g() {return C();}

void f(C) {}

int main() {
    C c;
    C c2(c);
    c = c2;
    c = g();
    f(c);
    f(g()); // Optimized out
    f(std::move(g()));
}
```

/ Output:*
copy constructor
copy assignment
move assignment
copy constructor
move constructor
**/*

move returns an *rvalue* reference to its argument

See *mstring.cpp* for a larger example

Perfect Forwarding

- **Named variables** are always **lvalues** in their scope
  - Even if their passed arguments were rvalue references
  - Any runtime value with a name is an lvalue
    - Because it has an address
- But rvalue references should be forwarded unchanged!
  - Use **g(std::forward<T>(x))**
- **std::forward<T>(x) == static_cast<T&&>(x)**

Reference Collapsing

- References to references are *collapsed*
  - A local parameter resolves to the 3rd column below...
- Rules (argument-type | parameter-type | collapsed-type):
  - A& & -> A&
  - A& && -> A&
  - A&& & -> A&
  - A&& && -> A&&
  - (i.e., an lvalue usage anywhere trumps the result)
- These rules are mainly for templates...

Special Template Rule

- && is a universal receiving reference qualifier
- `template<class T> void function foo(T&& t);`
  - If **foo** is passed an lvalue, then **t** is a **T&**
  - If **foo** is passed an rvalue, then **t** is a **T&&**
  - (again, an lvalue reference wins)

Variadic Templates

- Allows variable-length argument lists of mixed types
- Accomplished via templates with variable-length *type lists*
  - And *parameter packs*

```
void display() {}       // To stop the recursion

template<typename T, typename... Rest>
void display(T head, Rest... rest) {
    cout << typeid(T).name() << ": " << head << endl;
    display(rest...);  // parameter pack
}

int main() {
    display("one",1,2.0);
}
```

Output:

```
PKc: one
i: 1
d: 2
```

New Library Features

- New containers: **array**, **forward_list**, hashed sets and maps
- Generalized currying with **bind**
- String conversions functions (**stoi**, **stod**, **stof**, …)
- Tuples
- New smart pointers (**unique_ptr**, **shared_ptr**, **weak_ptr**)
- Concurrency with threads, atomics, and futures
- Regular expressions
- Generalized **begin/end** wrappers
- Generalized callable type: **function**

std::array

```
template<size_t N>
void add1(array<int,N>& a) {
    for (int& n: a)
        ++n;    // Modifies array in place
}

int main() {
    array<int,5> a{1,2,3,4};
    add1(a);
    for (auto n: a)
        cout << n << ' ';       // 2 3 4 5 1
    cout << endl;
}
```

std::bind and Member Functions

```
class Foo {
    int x;
public:
    Foo(int n) : x(n) {}
    int f() const {return x;}
    int g(int y) const {return x+y;}
    void display() const {cout << x << endl;}
};
```

See next slide...

Example Continued…

```
int main() {
    Foo obj(5);
    auto f1 = bind(&Foo::f,_1);     // “Unbound method”
    cout << f1(obj) << endl;        // 5
    auto f2 = bind(&Foo::g,obj,_1); // “Bound method”
    cout << f2(3) << endl;          // 8

    array<Foo,3> a = {Foo(1),Foo(2),Foo(3)};
    for_each(a.begin(),a.end(),bind(&Foo::display,_1));

    vector<Foo*> v = {new Foo(4), new Foo(5)};
    for_each(v.begin(),v.end(),bind(&Foo::display,_1)); // Just works!
    for_each(v.begin(),v.end(),[](Foo* p){delete p;});  // Clean-up
}
```

String Conversions

```
int main() {
    string n("10");
    string x("7.2");

    cout << stoi(n) << endl;    // 10
    cout << stol(n) << endl;    // 10
    cout << stoul(n) << endl;   // 10
    cout << stoll(n) << endl;   // 10
    cout << stoull(n) << endl;  // 10

    // Real to int
    cout << stoi(x) << endl;    // 7
```

```
// Reals
cout << stof(n) << endl;    // 10
cout << stod(n) << endl;    // 10
cout << stold(n) << endl;   // 10

// Different bases
n = "1011";
cout << stoi(n,0,2) << endl; // 11
n = "1F";
cout << stoi(n,0,16) << endl;// 31
}
```

Tuples

```
using MyTuple = tuple<int,string>;

MyTuple incr(const MyTuple& t) {
    return MyTuple(get<0>(t)+1, get<1>(t) + "+one");
}

int main() {
    MyTuple tup0(1,"text");
    auto tup1 = incr(tup0);
    cout << get<0>(tup1) << ' ' << get<1>(tup1) << endl;

    auto tup2 = make_tuple(2,string("text+one"));
    assert(tup1 == tup2);

    int n;
    string s;
    tie(n,s) = incr(tup2);     // Tuple assignment
    cout << n << ' ' << s << endl;
}
```

std::unique_ptr

- Only allows *single ownership* of a pointer
- Uses **move semantics** to transfer ownership
  - copying and assignment not allowed
- Automatically calls **delete** when scope is exited
- Can provide a *custom deleter*
  - if **delete** is not appropriate
- Can have containers of **unique_ptr**

unique_ptr Example 1

```
class Trace {
    int x;
public:
    Trace() : x(5) { cout << "ctor\n"; }
    ~Trace() { cout << "dtor\n"; }
    int get() const { return x; }
};

int main() {
    unique_ptr<Trace> p(new Trace);
    cout << p->get() << '\n';
}

/* Output:
ctor
5
dtor
*/
```

unique_ptr Example 2

```
class Foo {
public:
    Foo(){}
    ~Foo() {
        cout << "destroying a Foo\n";
    }
};

int main() {
    vector<unique_ptr<Foo> > v;
    v.push_back(unique_ptr<Foo>(new Foo));
    v.push_back(unique_ptr<Foo>(new Foo));
    v.push_back(unique_ptr<Foo>(new Foo));

}

/* Output:
destroying a Foo
destroying a Foo
destroying a Foo
*/
```

unique_ptr Example 3

```
class Foo {
public:
    Foo(){}
    ~Foo() {
        cout << "destroying a Foo\n";
    }
};

int main() {
    unique_ptr<Foo[]> p(new Foo[3]); // Arrays just work
}

/* Output:
destroying a Foo
destroying a Foo
destroying a Foo
*/
```

A Custom Deleter

```
void deleter(FILE* f) {
    fclose(f);
    cout << "FILE* closed\n";
}

int main() {
    // The following uses a deleter, but no wrapper class!
    FILE* f = fopen("deleter1.cpp", "r");
    assert(f);
    unique_ptr<FILE, void(*)(FILE*)> anotherFile(f,&deleter);

    // Could just do this instead (but there would be no trace)
    FILE* f2 = fopen("deleter1.cpp", "r");
    assert(f2);
    unique_ptr<FILE, int(*)(FILE*)> the3rdFile(f2,&fclose);
}

/* Output:
FILE* closed
*/
```

std::shared_ptr

A reference-counted pointer

```
struct Foo {
    int x;
};

void g(shared_ptr<Foo> p) {
    cout << p.use_count() << '\n';
    cout << p->x << '\n';
    p->x = 30;
}

void f(shared_ptr<Foo> p) {
    cout << p.use_count() << '\n';
    p->x = 20;
    g(p);
}
```

```
int main() {
    shared_ptr<Foo> p(new Foo);
    cout << p.use_count() << '\n';
    p->x = 10;
    f(p);
    cout << p.use_count() << '\n';
    cout << p->x << '\n';
}

/* Output:
1
2
3
20
1
30
*/
```

Automatic Cleanup with **shared_ptr**

```
class Resource {
    // Note: no public constructors!
    Resource() = default;
public:
    static Resource* Create() {     // Factory method
        Resource* p = new Resource;
        return p;
   }
   ~Resource() { cout << "Resource destroyed\n"; }
};

// A Client uses a Resource
class Client
{
public:
    Client(shared_ptr<Resource> p) : pRes(p){}
    ~Client() { cout << "Client object destroyed\n"; }

private:
    shared_ptr<Resource> pRes;
};
```

Example Continued…

```
int main() {
    // Create a Resource to be shared:
    shared_ptr<Resource> pR(Resource::Create());
    cout << pR.use_count() << endl;      // count is 1

    // Use the Resource in 2 clients:
    Client b1(pR);
    cout << pR.use_count() << endl;      // count is 2
    Client b2(pR);
    cout << pR.use_count() << endl;      // count is 3

    // b2.~Client() will reduce count to 2.
    // b1.~Client() will reduce count to 1.
    // pR.~shared_ptr<Resource>() will reduce the count to 0.
    // ...after which the Resource will self-destruct.
}
/* Output:
1
2
3
Client object destroyed
Client object destroyed
Resource destroyed
*/
```

Concurrency

- 3 levels of concurrency support
- 1) Threads
  - *Typical stuff*: condition variables (**wait/notify**), mutexes and locks, **try_lock**, multiple locks, **join**, thread-local data
- 2) Atomics
  - For low-level, lock-free programming (tricky!)
  - Intended for library implementers
- 3) Tasks
  - Higher-level support for task-based programming
  - **future**s, tasks as functions (**async**)

Task-Based Concurrency

- User specifies separable *tasks*
  - Callable entities (functions, methods, function objects, lambdas)
  - Users don't see underlying threads/synchronization
- The system provides a mechanism for automatically launching threads to run the tasks and for retrieving results at a later ("future") time

std::async

- Immediately returns a **future**
- Call **.get( )** later to get the result
  - Will block until the task has completed
- ```
  auto futr = async(f,arg1,arg2,…);
  …
  auto result = futr.get();
  ```
- The system decides whether to spawn a separate thread or to run the task synchronously
  - Unless you hint otherwise (**async/deferred** flags)
- See *async.cpp*
  ```

New Class-Related Features

- In-class initializers
- Inheritance control with **final** and **override**
- Delegating/Forwarding constructors
- Inheriting constructors
- Copy control with =**default**, =**delete**
- Move semantics with *rvalue references*

Template Method Example

- See *templatemethod.cpp*
- Illustrates:
  - **=default** for virtual destructors
  - **override**
  - **final**

Delegating Constructors

```
class Sales_data {
public:
    // nondelegating constructor initializes members from corresponding arguments
    Sales_data(std::string s, unsigned cnt, double price):
            bookNo(s), units_sold(cnt), revenue(cnt*price) { }
    // remaining constructors all delegate to another constructor
    Sales_data(): Sales_data("", 0, 0) {}
    Sales_data(std::string s): Sales_data(s, 0,0) {}
    Sales_data(std::istream &is): Sales_data()
                                    { read(is, *this); }
    // other members as before
};
```

Source: Lippman, C++ Primer, 5th Ed.

Inheriting Constructors

- Appropriate when derived classes add no data

```
struct A {
        A(initializer_list<int>){}
};

struct B : A {
        using A::A;     // Required to declare b below
};

int main() {
        B b{4,5,6};
}
```

Code

```
// compose5.cpp
template<typename Fun>
class Composer {
private:
    const vector<Fun>& funs;
public:
    Composer(vector<Fun>& fs) : funs(fs) {}
    using T = typename Fun::result_type;
    T operator()(T x) const {
        auto apply = [](T sofar, Fun f){return f(sofar);};
        return accumulate(funs.rbegin(),funs.rend(),x,apply);
    }
};
struct g {
    double operator()(double x) {
    return x*x;
    }
};

int main() {
    auto f = bind(divides<double>(),_1,2.0);
    using Fun = function<double(double)>;
    vector<Fun> funs{f,g(),[](double x){return x+1;}};
    Composer<Fun> comp(funs);
    cout << comp(2.0) << endl;//4.5
}
```

```
// mstring.cpp (page 1)
class String {
    char* data;
public:
    String(const char* p = "") {
        cout << "1-arg ctor: " << p << endl;
        strcpy(data=new char[strlen(p)+1],p);
    }
    String(const String& s) : String(s.data) {  // Delegating ctor
        cout << "copy ctor: " << s.data << endl;
    }
    String(String&& s) : data(s.data) {
        // Steal resources
        cout << "move ctor: " << data << endl;
        s.data = nullptr;                         // Zero-out temporary
    }
    String& operator=(const String& s) {
        cout << "copy assignment: " << s.data << endl;
        if (this != &s) {
            char* new_data = new char[strlen(s.data)+1];
            strcpy(new_data,s.data);
            delete [] data;
            data = new_data;
        }
        return *this;
    }
```

```
// mstring.cpp (page 2)
    String& operator=(String&& s) {
        cout << "move assignment: " << s.data << endl;
        std::swap(data,s.data);   // Steal resources via swapping
        return *this;
    }
    ~String() {
        cout << "destroying: " << (data ? data : "nullptr")
             << endl;
        delete [] data;
    }
    friend void print(const vector<String>&);
};

void print(const vector<String>& v) {
    cout << "<print>\n";
    for (const auto& x: v)
        cout << x.data << endl;
    cout << "</print>\n";
}
```

```
// mstring.cpp (page 3)
int main() {
    String s{"hello"};
    vector<String> v;
    v.reserve(3);
    v.push_back(String("every"));
    v.push_back(String("little"));
    v.push_back(String("thing"));
    cout << v.size() << endl;
    print(v);
}
```

```
/* Output:
1-arg ctor: hello
1-arg ctor: every
move ctor: every
destroying: nullptr
1-arg ctor: little
move ctor: little
destroying: nullptr
1-arg ctor: thing
move ctor: thing
destroying: nullptr
3
<print>
every
little
thing
</print>
destroying: thing
destroying: little
destroying: every
destroying: hello
*/
```

```
// async.cpp (page 1)

// Find number of hardware threads available (8 on my hardware)
auto nthreads = thread::hardware_concurrency();

using Iter = typename vector<double>::iterator;
double accum_block(Iter b, Iter e, size_t i) {
    return accumulate(b,e,0.0);
}

double concurrent_sum(Iter start, Iter stop) {
    vector<future<double>> tasks(nthreads); // Worker tasks
    // Launch tasks
    auto block_size = (stop - start) / nthreads;
    for (int i = 0; i < nthreads-1; ++i) {
        tasks[i] =
            async(accum_block,start+i*block_size,start+(i+1)*block_size,i);
    }
    // Sum last block
    double sum = accumulate(start+(nthreads-1)*block_size,stop,0.0);
    // Wait for tasks
    for (int i = 0; i < nthreads-1; ++i)
        sum += tasks[i].get();
    return sum;
}
```

```
int main() {
    cout << "Number of threads: " << nthreads << endl;
    vector<double> v;
    size_t n = 100000000;
    v.reserve(n);
    generate_n(back_inserter(v),n,[](){return rand()/1000.0;});
    auto start = chrono::high_resolution_clock::now();
    cout << accumulate(begin(v),end(v),0.0) << endl;
    auto stop = chrono::high_resolution_clock::now();
    cout << chrono::duration<double>(stop - start).count() << endl;

    cout << endl;
    start = chrono::high_resolution_clock::now();
    cout << concurrent_sum(begin(v),end(v)) << endl;
    stop = chrono::high_resolution_clock::now();
    cout << chrono::duration<double>(stop - start).count() << endl;
}

/* Output:
Number of threads: 8
1.07381e+14
0.48395
1.07381e+14
0.137339
*/
```

```
// Templatemethod.cpp (page 1)
class IBase {
public:
    virtual ~IBase() = default;
    virtual void theAlgorithm() = 0;
};
class Base : public IBase {
    void fixedop1() {
        cout << "fixedop1\n";
    }
    void fixedop2() {
        cout << "fixedop2\n";
    }
public:
    void theAlgorithm() override final {
        fixedop1();
        missingop1();
        fixedop2();
        missingop2();
    }
protected:
    virtual void missingop1() = 0;
    virtual void missingop2() = 0;
};
```

```
// Templatemethod.cpp (page 2)
class Derived : public Base {
    void missingop1() override {
        cout << "missingop1\n";
    }
    void missingop2() override {
        cout << "missingop2\n";
    }
};

int main() {
    Derived d;
    d.theAlgorithm();
}

/* Output:
Fixedop1
Missingop1
Fixedop2
Missingop2
*/
```