

# The new C++ (C++11)

“C++11 feels like a new language.” — B. Stroustrup

“C++11 feels like a new language.”

— Bjarne Stroustrup

# Characteristics of C++

*Claudia Seardine*

- Statically and strongly typed
- Multi-paradigm:
  - Procedural
  - Object-Oriented
  - Generic programming (i.e., template meta-programming)
- Compiled
- General-purpose

# Brief history of C++

Claudia Seardine

- 1969: Dennis Ritchie creates C programming language at Bell Labs to implement Unix
- 1979: Bjarne Stroustrup created "C with Classes" at Bell Labs
- 1983: Name changed to "C++"
- 1985: First edition of *"The C++ Programming Language"*
- 1991: Second edition of *"The C++ Programming Language"*
- 1998: First ISO standard (AKA "C++98")
- 2003: Second ISO standard (AKA "C++03")  
Actually it was the first standard with corrections.
- 2007: Technical report (AKA "TR1") about extensions to the standard library to be expected in the next standard

# And now...

*Claudia Seardine*

2011: Third C++ ISO standard

- Called "C++11"
- Previously known as "C++0x"

“C++11 feels like a new language.” — Bjarne Stroustrup

It changes style/idioms/guidance, but...  
it keeps backward compatibility

# Directives applied by the committee

*Claudio Scazzini*

- Maintain stability and compatibility with C++98 and possibly with C;
- Prefer introduction of new features through the standard library, rather than extending the core language;
- Prefer changes that can evolve programming technique;
- Improve C++ to facilitate systems and library design, rather than to introduce new features useful only to specific applications;
- Increase type safety by providing safer alternatives to earlier unsafe techniques;
- Increase performance and the ability to work directly with hardware;
- Provide proper solutions for real-world problems;
- Implement “zero-overhead” principle (additional support required by some utilities must be used only if the utility is used);
- Make C++ easy to teach and to learn without removing any utility needed by expert programmers.

# Outline

*Claudia Seardine*

- Core language:
  - Automatic type deduction: auto
  - Long long integers
  - Null pointer
  - Right-angle brackets
  - Range-based for
  - Strongly typed enumerations
  - Uniform initialization
  - Move constructors
  - Lambdas

- Standard library:
  - Fixed-size arrays
  - Smart pointers
  - Thread support
  - Tuples
  - Hash tables
  - More...

# Part I: Changes in the core language



# Automatic type deduction: auto

Claudia Seardine

- **auto** allows the compiler understand the type of a variable at compile time
- Called ***type inference***
- Useful for:
  - Reducing the verbosity of the code
  - Code where the type of a variable is automatically generated
    - E.g., return value of functions in template metaprogramming

# Automatic type deduction: auto (2)

*Claudia Seardine*

- Simple example:

```
std::vector v;  
for (std::vector<int>::const_iterator itr = v.begin();  
     itr != v.end(); ++itr)
```

can now be written as

```
std::vector v;  
for (auto itr = v.begin(); itr != v.end(); ++itr)
```

# Automatic type deduction: auto (3)

*Claudia Seardine*

- **auto** doesn't change the code's meaning
  - The code is still statically typed
- Use **auto** whenever possible (like **const** !)

# long long integers

*Claudia Seardine*

- C++11 allows **long long int** types:
  - At least as large as a **long int** and have no fewer than 64 bits

```
long long int i;  
std::cout << "Size of i is: " << sizeof(i) << std::endl;
```

```
cloud@mycomputer$ g++ --std=c++11 -Wall test.cpp -o test
```

```
cloud@mycomputer$ ./test
```

```
Size of i is 8
```

# nullptr

*Claudia Seardine*

- In the previous C++ standard, developers had 2 options to indicate a null pointer:
  - **NULL** (MACRO inherited from C)
  - **0** (recommended option for better type checking)
- Both interact poorly with function overloading:

```
void foo(char *);  
void foo (int);  
f(NULL); // which f is called ??
```

- If **NULL** is defined as **0** (usually in C++), the statement **foo (NULL)** will call **foo (int)**, which is not what the programmer meant!

# nullptr (2)

- Now the developer is given a special keyword **nullptr**
  - **nullptr**'s type is **nullptr\_t**

```
int* p = nullptr;
```

# decltype

- **decltype** can be used to know the type of an expression at compile-time

```
int some_int;  
decltype(some_int) other_integer_variable = 5;
```

# Right-angle brackets

```
list<vector<string>> lvs;
```

- Error in C++03 because there is no space between the two >s
- Fixed in C++11

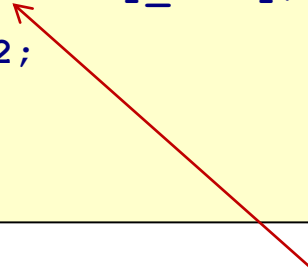


# Range-based for

- A range for statement allows you to iterate through a "range":

```
int my_array[5] = {1, 2, 3, 4, 5};  
for (int &x : my_array)  
    x *= 2;
```

```
vector<double> d;  
for (auto x : d)  
    cout << x << endl;
```



using a reference to allow us to  
change the value

- It works with:
  - C-style arrays
  - All standard containers
  - Any type that has begin() and end() that return iterators.

# Strongly typed enumerations

*Claudia Seardine*

- Issues with common enums:
  - They are not type-safe: they are effectively integers; this allows the comparison between two enum values of different enumeration types.
  - They export their enumerators to the surrounding scope, causing name clashes.

# Strongly typed enumerations (2)

*Claudia Seardine*

- Stronger type-checking on enums through **enum class**
- These enumerations are type safe as they cannot be converted or compared to integers

```
enum class Enumeration {Val1, Val2 = 3, Val3 = 100};
```

- The default underlying type is int . Different types can be explicitly set:

```
enum class Enumeration: unsigned int {Val1, Val2 = 3, Val3 = 100};
```

# Uniform initialization

- Fully uniform type initialization that works on any object

```
int* a = new int[3] { 1, 2, 0 };  
vector<string> vs={ "first", "second", "third"};
```

- In-class initialization of data members:

```
class C  
{  
    int a = 7;  // C++11 only  
public:  
    C();  
};
```

# Move constructors

*Claudia Seardine*

- In C++03, temporaries (AKA "rvalues", as they often lie on the right side of an assignment) were intended to never be modifiable — just as in C.
- Common problem: how to get a lot of data cheaply out of a function ?
  - Performance problem with C++03 is the costly and unnecessary deep copies that can happen implicitly when objects are passed by value.
  - Consider a function that returns a `std::vector<T>` : it can be stored only by creating a new `std::vector<T>` and copying all of the rvalue's data into it. Then the temporary and all its memory is destroyed.
- No solution with previous standard:
  - Copy ? Expensive!
  - Pre-allocated "results stack" ? Unacceptable
  - Pointer to a new'd object ? Error-prone + issues in design (who is in charge of deallocating the object ?)

# Move constructors (2)



Most  
important  
feature of  
C++11

- C++11 introduces:
  - a new category of reference types called *rvalue references*
  - "move constructors" and "move assignments"

```
class vector {  
    // ...  
    vector(const vector&);           // copy constructor  
    vector(vector&&);               // move constructor  
    vector& operator=(const vector&); // copy assignment  
    vector& operator=(vector&&);     // move assignment  
};
```

- The **&&** indicates an rvalue reference
- Unlike traditional copying, moving means that a target object "steals" the resources of the source object, leaving the source in an "empty" state

# Move constructors (3)

You will  
use pass-  
by-value  
more often

Example:

```
string flip (string s)
{
    reverse (begin(s), end(s));
    return s;
}
```

Move semantics

=

Semantics of pass-by-value (no pointers, declared lifetime)

+

Efficiency of pass-by-reference (no deep copying)

# Move constructors (4)

*Claudia Seardine*

- In the previous example, a move constructor of `std::vector<T>` can simply copy the pointer to the internal C-style array out of the rvalue into the new `std::vector<T>`, then set the pointer inside the rvalue to null.
- Since the temporary will never again be used, no code will try to access the null pointer, and because the pointer is null, its memory is not deleted when it goes out of scope.
  - => The operation not only forgoes the expense of a deep copy, but is safe and invisible.
- This mechanism can provide performance benefits to existing code without needing to make any changes outside the standard library
  - The type of the returned value of a function returning an object does not need to be changed explicitly to `&&` to invoke the move constructor, because temporaries are considered rvalues automatically



# Move constructors (5)

- Move constructors can also be invoked explicitly:

```
template<class T>
void swap(T& a, T& b) // "perfect swap" (almost)
{
    T tmp = move(a); // could invalidate a
    a = move(b);      // could invalidate b
    b = move(tmp);    // could invalidate tmp
}
```

# Move constructors (6)

*Claudia Seardine*


- The C++11 Standard Library uses move semantics extensively.
- Many algorithms and containers are now move-optimized, having better performance.

# Delegating constructors

*Claudia Seardine*

- A constructor may call another constructor of the same class

```
class M
{
    int x, y;
    char *p;
public:
    M(int v) x(v), y(0), p(new char [MAX]) {} // #1 target
    M(): M(0) {cout<<"delegating ctor"<<endl;} // #2 delegating
};
```



# Lambdas

*Claudia Seardine*

- C++11 provides the ability to create anonymous functions locally
- The functions are called "lambda functions".
- They make code more elegant and faster

# Lambdas (2)

- Typical form:

```
[capture] (arguments) ->return-type{body}
```

- The return type can be often omitted:

```
[capture] (arguments) {body}
```

- Example:

```
[] (int x, int y) { return x + y; }
```

# Lambdas (3)

*Claudia Seardine*

- Example: count how many uppercase letters a string contains

```
char s[]="Hello World!";  
int upperCase = 0; //modified by the lambda  
for_each(s, s+sizeof(s), [&UpperCase] (char c) {  
    if (isupper(c))  
        upperCase++;  
});
```

for\_each: applies function f to  
each of the elements in the  
range [first,last)

# Lambdas (4)

Claudia Seardine

- A lambda function can refer to identifiers declared outside the lambda function.
- The set of these variables is commonly called a "*closure*"
- Closures are defined between square brackets [ ]
- These variables can be captured by value or by reference.

Symbol	Meaning
[ ]	Any access to external variables is an error.
[x, &y]	x captured by value; y captured by reference
[&]	any external variable is implicitly captured by reference
[=]	any external variable is implicitly captured by value
[&, x]	x captured by value; other variables captured by reference
[= , &z]	z captured by reference; other variables captured by value

# Lambdas (5)

- Typical for loop:

```
sum = 0;
for (vector<int>::size_type i = 0; i < v.size(); ++i)
    sum += v[i];
```

- can now be written as

```
sum = 0;
for_each(v.begin(), v.end(), [&sum] (int x) {
    sum +=x;
});
```



## Part II: Changes in the standard library

# Standard types

Include `cstdint` to have:

- `std::int8_t`
- `std::int16_t`
- `std::int32_t`
- `std::uint8_t`
- `std::uint16_t`
- `std::uint32_t`
- ...

# Fixed-size arrays

*Claudia Scardine*

## array:

- Standard container
- Fixed-sized array of elements
- No space overhead
- It can be initialized with an initializer list
- It knows its size

```
array<int,6> a = { 1, 2, 3 };  
a[3] = 4;  
int x = a[5];    // x becomes 0 because default elements are zero initialized  
int* p2 = a.data(); // ok: get pointer to first element
```

“C++ is the best language for garbage collection principally because it creates less garbage.”

— Bjarne Stroustrup

# Introduction to smart pointers

- Typical example of memory leak with new/delete:

```
#include <iostream>

class C {
    long int array[100000];
public:
    C(){std::cout << "Constructor called" << std::endl;}
    ~C(){std::cout << "Destructor called" << std::endl;}
};

int main ()
{
    for (;;) {
        C* c = new C;
        // delete forgotten
    }
    return 0;
}
```

# Introduction to smart pointers (2)

*Claudia Seardine*

- Once compiled and run we obtain:

```
cloud@mycomputer$ g++ -Wall test.cpp -o test
cloud@mycomputer$ ./test
Constructor called
Constructor called
Constructor called
Constructor called
Constructor called
terminate called after throwing an instance of 'std::bad_alloc'
  what():  std::bad_alloc
```

# Smart pointers

- Same example with smart pointers:

```
#include <iostream>
#include <memory>

class C {
    long int array[100000];
public:
    C(){std::cout << "Constructor called" << std::endl;}
    ~C(){std::cout << "Destructor called" << std::endl;}
};

int main ()
{
    for (;;) {
        std::shared_ptr<C> c = std::make_shared<C>();
        // no need of delete
    }
    return 0;
}
```

# Smart pointers (2)

*Claudia Seardine*

- Once compiled and run we obtain:

```
cloud@mycomputer$ g++ --std=c++11 -Wall test.cpp -o test
```

```
cloud@mycomputer$ ./test
```

```
Constructor called
```

```
Destructor called
```

```
Constructor called
```

```
Destructor called
```

```
Constructor called
```

```
Destructor called
```

```
Constructor called
```

```
Destructor called
```

```
...
```



# Smart pointers (3)

- Another example of memory leak with exceptions:

```
for( vector<circle*>::iterator i = vw.begin(); i != vw.end(); ++i )  
    delete *i; // Not exception safe! Missing try/catch...
```

# Smart pointers (4)

- Yet another example of memory leak with exceptions:

```
class C {  
    long int array[100000];  
public:  
    C(){std::cout << "Constructor called" << std::endl;}  
    ~C(){std::cout << "Destructor called" << std::endl;}  
};  
  
void f (int n)  
{  
    C *c = new C;  
    if (n < 100)  
        throw std::runtime_error("Error!"); //Leak  
    delete c;  
}
```

# Smart pointers (5)

- Solution with smart pointers:

```
class C {  
    long int array[100000];  
public:  
    C(){std::cout << "Constructor called" << std::endl;}  
    ~C(){std::cout << "Destructor called" << std::endl;}  
};  
  
void f (int n)  
{  
    std::unique_ptr<C> c (new C());  
    if (n < 100)  
        throw std::runtime_error("Error!");  
    // no need of delete  
}
```

# Types of smart pointers: `unique_ptr`

*Claudia Seardine*

## `unique_ptr`:

- Semantics of strict ownership
- *"what `auto_ptr` should have been"*
- Uses include
  - providing exception safety for dynamically allocated memory
  - storing pointers in containers

Note: the usage of `auto_ptr` is now deprecated

# Types of smart pointers: shared\_ptr

*Claudia Seardine*

## **shared\_ptr:**

- Represents shared ownership: when two pieces of code needs access to some data but neither has exclusive ownership (in the sense of being responsible for destroying the object)
- Kind of counter : the pointed object is deleted when the counter goes to zero

# Types of smart pointers: weak\_ptr

*Claudia Seardine*

## `weak_ptr`:

- Often explained as what you need to break loops in data structures managed using `shared_ptrs`.
- Better to think of a `weak_ptr` as a pointer to something that
  1. you need access to (only) if it exists, and
  2. may get deleted (by someone else), and
  3. must have its destructor called after its last use (usually to delete a non-memory resource)

# Smart pointers and raw pointers

*Claudia Seardine*

- Always use
  - standard smart pointers, and
  - non-owning raw pointers
- Never use owning raw pointers and delete!

# Smart pointers: concurrency

*Claudia Seardine*

- Operations that change the reference count, due to copying or destroying `shared_ptr` or `weak_ptr` objects, do not provoke data race conditions.
- This means that multiple threads can safely store `shared_ptr` or `weak_ptr` objects that reference the same object.
- This only protects the reference count itself; it does not protect the object being stored by the smart pointer.



# Threads

```
void f (vector<double>&);    // function

struct F {
    vector<double>&v;
    F(vector<double>& vv): v(vv){}
    void operator(){}
};

void code (vector<double>& vec1, vector<double>& vec2)
{
    std::thread t1 {f, vec1};    // run f(vec1) on a separate thread
    std::thread t2 {F(vec2)};    // run F(vec2) () on a separate thread
    t1.join();
    t2.join();
    // use vec1 and vec2
}
```

# Threads with lambdas

*Claudia Seardine*

```
std::vector<std::thread> threads;
for (auto c : s){
    threads.push_back (std::thread([c] {
        std::this_thread::sleep_for(std::chrono::milliseconds(i*50));
        // ...
    }));
}
for (auto& t: threads)
    t.join();
```

# Asynchronous execution

*Claudia Seardine*

```
int myfunc(int arg)
{
    //...
    return ...
}
```

```
int main ()
```

```
{
    //...
    int a = ...;
    std::future<int> out = std::async(std::launch::async, myfunc, in);
    //...
    a += out.get();
}
```

Without this, code executed only  
when the future is got



# Mutual exclusion and synchronization

*Claudia Scardine*

- Synchronization:
  - `std::mutex`
  - `std::condition_variable`

```
std::mutex m;  
// ...  
m.lock();  
// manipulate shared data:  
m.unlock();
```

# Better mutual exclusion

- Exception-safe mutual exclusion:

```
void myfunc()  
{  
    std::unique_lock<std::mutex> lock (mutex_);  
}
```

# Tuples

```
tuple<string,int> t2("Kylling", 123);  
auto t = make_tuple(string("Herring"), 10, 1.23);  
string s = get<0>(t);  
int x = get<1>(t);  
double d = get<2>(t);
```

# Hash tables

- C++11 STL contains hash tables
- Called "unordered" to avoid issues in backward compliance
  - **map**: tree-based data structure
  - **unordered\_map**: hash-based data structure

```
map<string, string> phone;  
phone["Alex Lifeson"] = "+1 (416) 555-1212";
```

```
multimap<string, string> phone;  
phone["Neil Peart"] = "+1 (416) 555-1212";  
phone["Neil Peart"] = "+1 (905) 555-1234";
```

```
unordered_map<string, string> phone;  
phone["Alex Lifeson"] = "+1 (416) 555-1212";
```

```
unordered_multimap<string, string> phone;  
phone["Neil Peart"] = "+1 (416) 555-1212";  
phone["Neil Peart"] = "+1 (905) 555-1234";
```

# Time utilities: chrono

- C++11 contains time utilities:

```
auto now = std::chrono::system_clock::now();  
  
// Print calendar time (i.e., date + time):  
std::time_t result = std::time(NULL);  
std::cout << std::asctime(std::localtime(&result));
```



# Time utilities: chrono (2)

*Claudia Seardine*

```
std::chrono::time_point<std::chrono::system_clock> t1 =  
    std::chrono::system_clock::now();  
//...  
std::chrono::time_point<std::chrono::system_clock> t2 =  
    std::chrono::system_clock::now();  
int elapsed_seconds = std::chrono::duration_cast<std::chrono::seconds>  
    (t2-t1).count();
```

# Time utilities: chrono (3)

*Claudia Seardine*

```
auto now = std::chrono::system_clock::now();
std::time_t currTime = std::chrono::system_clock::to_time_t(now);
struct tm *currTm = std::localtime(&currTime);
std::cout << (1900 + currTm->tm_year) << "-"
            << currTm->tm_mon << "-"
            << currTm->tm_mday << "_"
            << currTm->tm_hour << "-"
            << currTm->tm_min << "-"
            << std::endl;
```

# C++11 contains much more...

*Claudia Seardine*

- Default and delete constructors
- Regular expressions
- Bind
- Random number generation
- Constant expressions
- Static assertions (compile-time)
- Variadic templates
- No exception propagation
- Algorithms improvements
- Condition variables

# Level of inclusion in compilers

*Claudia Seardine*

- Gcc:

- Initial support from gcc 4.6 (use `--std=c++0x`)
- More support from gcc 4.7 (use `--std=c++0x`)
- See: [http://gcc.gnu.org/gcc-4.7/cxx0x\\_status.html](http://gcc.gnu.org/gcc-4.7/cxx0x_status.html)

- Visual C++

- See: <http://blogs.msdn.com/b/vcblog/archive/2010/04/06/c-0x-core-language-features-in-vc10-the-table.aspx>
- See: <http://blogs.msdn.com/b/vcblog/archive/2011/09/12/10209291.aspx>

- Clang:

- See: [http://clang.llvm.org/cxx\\_status.html](http://clang.llvm.org/cxx_status.html)

# References

Claudia Seardine

- Wikipedia: <http://en.wikipedia.org/wiki/C%2B%2B11>
- B. Stroustrup: <http://www2.research.att.com/~bs/C++0xFAQ.html>
- H. Sutter: <http://herbsutter.com/elements-of-modern-c-style/>
- Going Native 2012 talks:  
<http://channel9.msdn.com/Events/GoingNative/GoingNative-2012>
- S. Meyers, *Overview of the New C++*  
[http://www.artima.com/shop/overview\\_of\\_the\\_new\\_cpp](http://www.artima.com/shop/overview_of_the_new_cpp)
- Interesting link:  
<http://www.softwarequalityconnection.com/2011/06/the-biggest-changes-in-c11-and-why-you-should-care/>

# Questions ?

*Claudia Scardine*



# Sorting vectors: using C and qsort

*Claudia Seardine*

We have to sort SIZE1 arrays of SIZE integers randomly generated...

```
int compare (const void * a, const void * b) {
    return ( *(int*)a - *(int*)b );
}

int main() {
    int i,k;
    for (i = 0; i < SIZE1; ++i){
        int v [SIZE2];
        for (k = 0; k < SIZE2; ++k)
            v[k] = rand();
        qsort(v, SIZE2, sizeof(int), compare);
    }
    return 0;
}
```

# Sorting vectors: using C++ and std::sort

*Claudia Seardine*

In C++ we have:

```
int main()
{
    for (int i = 0; i < SIZE1; ++i){
        std::vector<int> v;
        for (int k = 0; k < SIZE2; ++k)
            v.push_back(rand());
        std::sort(v.begin(), v.end());
    }
    return 0;
}
```

Which one is faster ??