Course Review

Optimized C++

Ed Keenan

13 March 2019

13.0.5.17.7   Mayan Long Count

DePaul University

Goals

- Movies
  - Why – reason behind the madness
  - Associations
- Topics
- Exam
- Contact

Are we done yet?

Why did I play movies?

- Break
  - Waiting for students to roll in
- Entertaining
  - At least to me
- Learning and Memorization
  - These stupid videos will trigger a memory
    - A thought in the future
    - Please email me in years to come when one of these video goes through your mind. (and it will)

Albania

- TV Show Cheers
  - Episode showing in 1985
    - 31 years ago
  - Teacher's pet episode
    - http://www.youtube.com/watch?v=-F_tT-q8EF0
- To this day,
  - I recall material from this TV episode
    - *Albania boarders on the Adriatic*

Movies

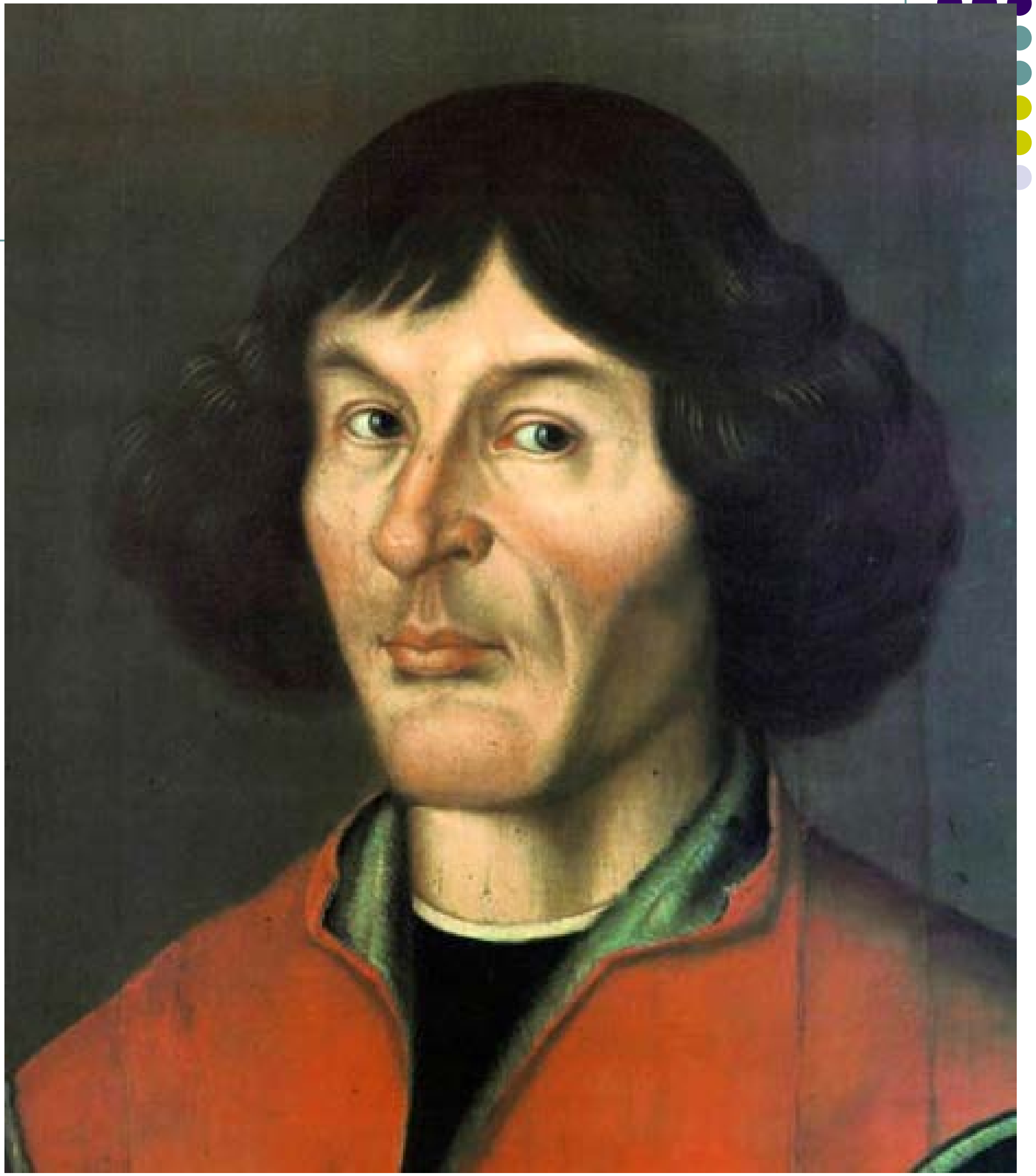
- Welcome to the world of Gentlemen, Gentlemen
  - Cadillac DTS: diner
    - http://www.youtube.com/watch?v=DoQXao7Zjpg
  - Season Programmers

Movies

- McDLT
  - Hot side ***HOT***, Cold side ***COLD***
    - http://www.youtube.com/watch?v=zL2c6NSVAvA
  - Hot / Cold data structures

Photo

- Who knows about Copernicus?
  - He’s Dead
  - Know
    - Linked Lists
    - Pointers



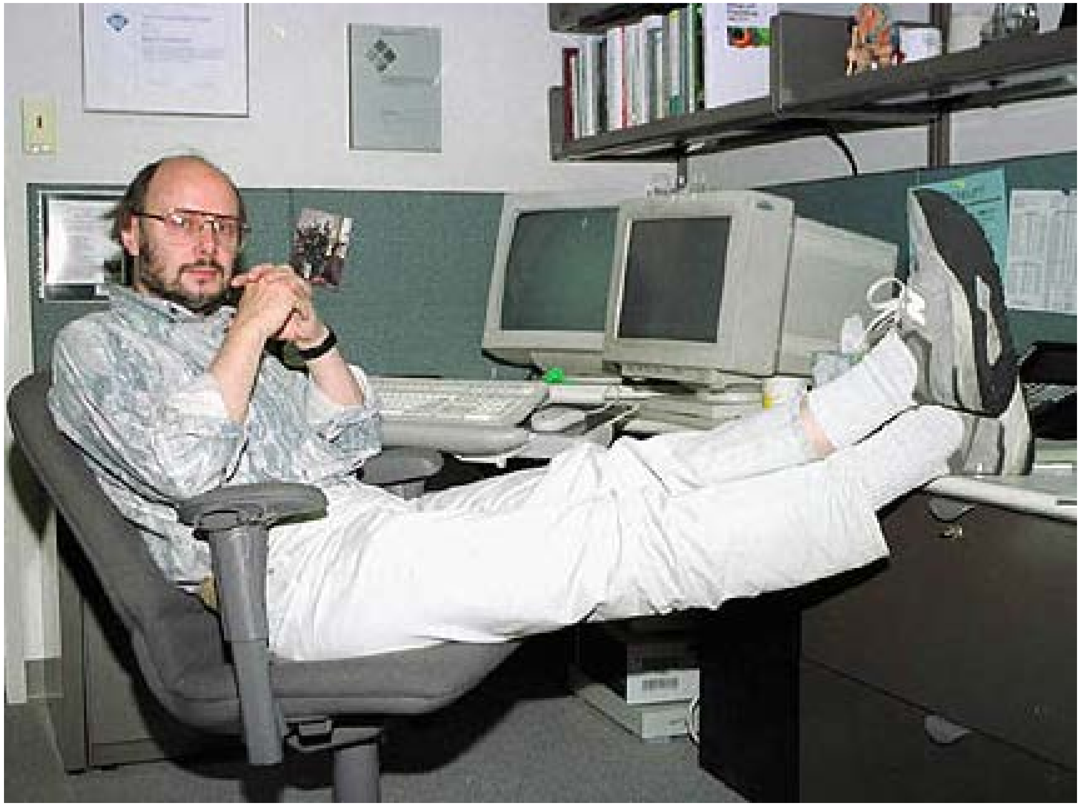
Movies

- Fight Club
  - 8 Rules of Fight Club
    - http://www.youtube.com/watch?v=fbMa4MGFCOg
  - Rules of Code Review
    - Learn the 8 rules of fight club

Movies

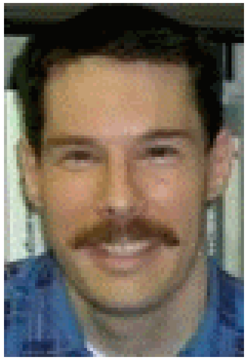
- Learn Self Defense
  - Memory system
    - http://www.youtube.com/watch?v=yyV1OuJyT4I&t=1m50s
  - You Make a Plan
    - Stick with it no matter what

Photo



Bjarne Stroustrup
C++

Photo



Herb Sutter
Compiler / C++ expert

Photo

Scott Meyers
C++ Expert

Effective C++,STL
Series of books

Movies

- Pot Noodles
  - Welsh Miners
    - http://www.youtube.com/watch?v=JrNuT9sn0Gc
  - Noodling in the back of Stroustrup book
    - Proxy Objects

Movies

- Single Instruction Multiple Data (SIMD)
  - How to Stack
    - http://www.youtube.com/watch?v=lcIb-tBUG5U
  - Fast Drink
    - http://www.youtube.com/watch?v=EmZGzoRXPP0

Movies

- Load in Place
  - Windows 8 _ Make up - Beautiful and Fast TV Commercial
    - http://www.youtube.com/watch?v=ZAxdKPKdlsI
  - I’m Batman
    - I'm batman commercial – Snickers
    - http://www.youtube.com/watch?v=JoX-HkOcEuE

Movies

- Monty Hall Problem
  - Reference
    - https://www.youtube.com/watch?v=T5QYTrDReTo
  - Hitler Optimizes Particle System
    - http://captiongenerator.com/1161732/HitlerOptimizesParticleSystemFall2018take2

Goal of this class

- Introduce you to Optimization
  - Started giving you tools for your toolbox
  - Growth of knowledge
    - Began class at one level
    - Hopefully leave here at a higher level
- It’s not the destination by the journey!
  - Keep learning
  - It’s never over
- What’s next
  - Talked with Addison Wesley
    - Loved the idea of the class
    - Wants me to turn it into a book
    - Maybe you’ll see your name in the acknowledgements…

Final Exam

- Movie / Concept association
- Data Layout
- Alignment
- C++ operators
- Hot / Cold data structures
- Pointers
- Optimization strategies
- Photo recognition
- Memory overload
- Return Value
- Proxy objects
- Memory System
- Overloading
- Implicit conversion
- STL issues
- Perforce basics
- File system
- Load in Place
- Debugging
- Const correctness

Final Exam

- Basics
  - Debugging
  - Overloading
  - Pointers
  - C Strings
  - Inheritance
  - vTables
  - STL
  - Templates
- Don’t know how much
  - But all is fair game
  - Make sure you understand the basics

Final Exam

- Excluded material
  - No PC
  - No Phones
  - No other reference material
- 3 hours
  - Shouldn't take that long
  - Particle competition while you test
- ***NOTE:***
  - You must pass the final exam
  - 20% of your grade
  - *It's not an easy exam*

Contact Information

- It's been a REAL pleasure guiding you through this material
  - Seriously!
- Stay in touch
  - Emails, Visits, lunches…
  - My door is always open
    - Ed Keenan
    - ekeenan2@cdm.depaul.edu

DePaul University

Thank You!

- Questions?

C++ 11

Optimized C++

Ed Keenan

DePaul University

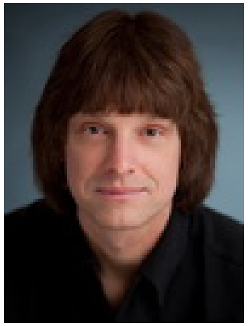
Goals

- New C++ features
  - Better or worse
    - You judge

Last class?

References:

- Internet articles
  - *http://www.codeproject.com/Articles/570638/Ten-Cplusplus-Features-Every-Cplusplus-Developer*
- Lecture series
  - Stroustrup
    - http://www.stroustrup.com/C++11FAQ.html
  - Meyers
    - http://www.aristeia.com/C++11.html



C++ 11

- Why did we not focus on C++ 11?
  - We need to understand the Basics
  - You can't appreciate the new features without understanding the fundamentals
- Not all compilers fully support C++ 11
  - Do not use or rely on new features until there is wide use
  - Wisdom

Keyword: auto

- Originally used for storage
  - ***auto, static, register***
  - ***auto*** – meant on the stack
- C++ 11
  - ***auto*** – type inference
    - placeholder for a type
    - compiler it has to deduce the actual type of a variable that is being declared from its initializer
  - used when declaring variables in different scopes
    - namespaces, blocks or initialization statement
  - Cannot be use for return type

auto

```
auto i = 42;          // i is an int
auto l = 42LL;        // l is an long long
auto p = new foo();   // p is a foo*
```

- less code
  - well maybe…
    - I’m not 100% sold on it…

auto

- Old way

```
std::map<std::string, std::vector<int>> map;
std::map<std::string, std::vector<int>>::iterator it= map.begin();
for(it; it != map.end(); ++it)
{
}
```

- New way

```
std::map<std::string, std::vector<int>> map;
for(auto it = begin(map); it != end(map); ++it)
{
}
```

- Simplify writing, removing the need for typedefs

Keyword: nullptr

- In C++
  - 0 used to be the value of null pointers
- Drawbacks
  - Implicit conversion to integral types
- ***nullptr*** denotes a value of type
  - std::nullptr_t that represents the null pointer literal
- Implicit conversions
  - ***nullptr*** to null pointer value of any pointer type
  - any pointer-to-member types
    - Conversion to ***bool*** (as false).
  - No implicit conversion to integral types exist

nullptr

```
void foo(int* p) {}
void bar(std::shared_ptr<int> p) {}
```

- For backward compatibility 0 is still a valid null pointer value
  - Yes – my code doesn't break!

```
int* p1 = NULL;
int* p2 = nullptr;
if(p1 == p2)
{
}

foo(nullptr);
bar(nullptr);

bool f = nullptr;

// error: A native nullptr can
// only be converted to bool
// or, using reinterpret_cast,
// to an integral type
int i = nullptr;
```

Range-based for loops

- C++11
  - Modified the idea of "***foreach***" in iterations
- Now possible to iterate
  - over C-like arrays
  - initializer lists
  - anything begin() and end() functions are overloaded
- Useful
  - elements of a collection/array
  - don't care about indexes, iterators or number of elements

Range-based for loops

- What does this do?
  - Look at the colon in the for loop

```
std::map<std::string, std::vector<int>> map;
std::vector<int> v;
v.push_back(1);
v.push_back(2);
v.push_back(3);
map["one"] = v;

for(const auto& kvp : map)
{
  std::cout << kvp.first << std::endl;

  for(auto v : kvp.second)
  {
     std::cout << v << std::endl;
  }
}
```

Range-based for loops

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

- What does this do?
  - Look at the colon in the for loop
  - What is the value of arr[] after this loop?

Overriding virtual functions

- Isn’t a mandatory mechanism to mark virtual methods as overridden in derived classes.
- Virtual keyword is optional
  - makes reading code a bit harder
  - Need to look through hierarchy to check if the method is virtual
  - Programmers should use virtual keyword on derived classes to make the code easier to read.

Overriding: Scenario 1

- D::f
  - Intended to override B::f
  - signature different
- B::f
  - another method with the same name
  - Its overloaded not overrided

```
class B
{
public:
    virtual void f(short)
    {std::cout << "B::f" << std::endl;}
};

class D : public B
{
public:
    virtual void f(int)
    {std::cout << "D::f" << std::endl;}
};
```

Overriding: Scenario 2

- Subtle error:
  - parameters are the same
  - method in the base class is const
  - derived is not.
- D::f
  - Intended to override B::f
  - signature different
- B::f
  - another method with the same name
  - Its overloaded not overrided

```
class B
{
public:
   virtual void f(int) const
   {std::cout << "B::f " << std::endl;}
};

class D : public B
{
public:
   virtual void f(int)
   {std::cout << "D::f" << std::endl;}
};
```

Keyword: override

- Use the keyword ***override*** to indicate that a method is supposed to be an override of a virtual method in a base class
- Now the class D:f causes an error

```
class B
{
public:
    virtual void f(short)
    {std::cout << "B::f" << std::endl;}
};

class D : public B
{
public:
    virtual void f(int) override
    {std::cout << "D::f" << std::endl;}
};
```

Keyword: final

- To make a method impossible to override any more (down the hierarchy) mark it as ***final***
- That can be in the base class, or any derived class.
- If it's in a derived classes you can use both the override and final specifiers.
- F:f is an error

```
class B
{
public:
   virtual void f(int)
   {std::cout << "B::f" << std::endl;}
};

class D : public B
{
public:
   virtual void f(int) override final
   {std::cout << "D::f" << std::endl;}
};

class F : public D
{
public:
   virtual void f(int) override
   {std::cout << "F::f" << std::endl;}
};
```

Strongly-typed enums

- Old C++ enums drawbacks
  - they export their enumerators in the surrounding scope
  - can lead to name collisions
    - if two different enums in the same have scope define enumerators with the same name
    - implicitly converted to integral types
    - cannot have a user-specified underlying type.
- C++ 11
  - strongly-typed enums
    - enum class keywords
  - no longer export their enumerators in the surrounding scope
  - no longer implicitly converted to integral types
  - can have a user-specified underlying type

Strongly-typed enums

```
enum class Options
{
    None,
    One,
    All
};

Options o = Options::All;
```

- Now everything is
  - scoped,
  - auto completed
  - safe
- Great addition

static_assert

```
int *p;

// runtime error
assert(p != nullptr);

// compile time error
static_assert( sizeof(int) == 4, "wrong size integers");
```

- Why did C++ 11 add ***static_assert*** didn't we already have assert?
  - assert() – for runtime checking
  - static_assert() – for compile time checking
    - Static – stands for static analysis

static_assert

- Why is this useful?
  - You can use this for protection with your templates
  - Remember templates are programmatically compiled code
    - So static analysis is available by the compiler
- More useful when used with type traits.
  - provide information about types at compile time
  - <type_traits> header
    - helper classes,
    - compile-time constants,
    - type traits classes
      - get type information,
    - type transformation classes
      - getting new types by applying transformation on existing types

smart_pointers

- In my opinion
  - the best part of C++ is direct control of the memory,
  - if you do not want this…
    - use C# or Objective-C or Java
- The progression towards managed memory continues…
  - ***auto_ptr*** is obsolete and should no longer be used
  - Now its ***smart_pointers***
- Smart pointers have reference counting and auto release
  - It tries to help in facilitating these functions
  - Again – I can argue,
    - design a encapsulated systems and these issues go away
- Three headed hydra of smart pointers:
  - ***unique_ptr***
  - ***shared_ptr***
  - ***weak_ptr***

unique_ptr

- ***unique_ptr*** is a container for a raw pointer
  - ***unique_ptr*** explicitly prevents copying of its contained pointer
  - ***std::move*** function can be used to transfer ownership of the contained pointer to another ***unique_ptr***
  - ***unique_ptr*** cannot be copied because its copy constructor and assignment operators are explicitly deleted
- should be used
  - when ownership of a memory resource does not have to be shared
    - it doesn't have a copy constructor
  - can be transferred to another ***unique_ptr***
    - move constructor exists

shared_ptr

- ***shared_ptr*** is a container for a raw pointer
  - It maintains reference-counted ownership of its contained pointer in cooperation with all copies of the ***shared_ptr***
  - The object referenced by the contained raw pointer will be destroyed when and only when all copies of the ***shared_ptr*** have been destroyed.
- Should be used
  - when ownership of a memory resource should be shared
    - hence the name

weak_ptr

- ***weak_ptr*** is a container for a raw pointer.
  - It is created as a copy of a ***shared_ptr***
  - holds a reference to an object managed by a ***shared_ptr***, but does ***not*** contribute to the reference count
- Should be used
  - to break dependency cycles
    - think of a tree where the parent holds an owning reference (***shared_ptr***) to its children, but the children also must hold a reference to the parent; if this second reference was also an owning one, a cycle would be created and no object would ever be released

smart_pointers: examples

```
std::unique_ptr<int> p1(new int(5));
std::unique_ptr<int> p2 = p1; //Compile error.

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

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

smart_pointers: examples

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

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

smart_pointers: examples

```
std::shared_ptr<int> p1(new int(5));
std::weak_ptr<int> wp1 = p1; //p1 owns the memory.
{
  std::shared_ptr<int> p2 = wp1.lock(); //Now p1 and p2 own the memory.
  if(p2) // As p2 is initialized from a weak pointer,
  {      // you have to check if the memory still exists!
    //Do something with p2
  }
} //p2 is destroyed. Memory is owned by p1.

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

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

Non-member begin and end

- C++ 11 introduces non-member begin(x) and end(x)
  - Instead of using x.begin() and x.end() in containers.
  - begin(x) and end(x) are extensible and can be adapted to work with all container types even arrays
- You can refactor your own non-member begin(x) and end(x) overloads for that type and then you can traverse collections of that type using the same coding style above as for STL containers.

Non-member begin and end

```
vector<int> v;
int a[100];

// C++98
sort( v.begin(), v.end() );
sort( &a[0], &a[0] + sizeof(a)/sizeof(a[0]) );

// C++11
sort( begin(v), end(v) );
sort( begin(a), end(a) );
```

Lamdas

- Anonymous functions, called ***lambda***, have been added to C++ and quickly rose to prominence.
  - http://en.cppreference.com/w/cpp/language/lambda
- Feature borrowed from functional programming, that in turned enabled other features
- Use lambdas wherever following are expected
  - function object
  - functor
  - std::function

Lamdas

```
int x;
int y;

// C++98: write a naked loop (using std::find_if is impractically difficult)
vector<int>::iterator i = v.begin(); // because we need to use i later
for( ; i != v.end(); ++i )
{
    if( *i > x && *i < y )
        break;
}

// C++11: use std::find_if
auto i = find_if( begin(v), end(v), [=](int i) { return i > x && i < y; } );
```

Move and &&

- The rvalue reference can be used to easily add move semantics to an existing class.
  - copy constructor and assignment operator can be overloaded based on whether the argument is an lvalue or an rvalue
- When the argument is an rvalue
  - the author of the class knows that he has a unique reference to the argument
- C++11 has introduced the concept of rvalue references (specified with ***&&***) to differentiate a reference to an lvalue or an rvalue.
  - An lvalue is an object that has a name, while an rvalue is an object that does not have a name (a temporary object).
  - The move semantics allow modifying rvalues
    - immutable and indistinguishable from const T& types
- Uses ***std::move()***

rvalue

- **rvalue**
  - An rvalue is an expression that is either a prvalue or an xvalue.
- **prvalue (since C++11)**
  - A *prvalue* ("pure" rvalue) is an expression that identifies a temporary object (or a subobject thereof) or is a value not associated with any object.
- **Xvalue (since C++11)**
  - An *xvalue* is an expression that identifies an "eXpiring" object, that is, the object that may be moved from.
  - The object identified by an xvalue expression may be a nameless temporary, it may be a named object in scope, or any other kind of object, but if used as a function argument, xvalue will always bind to the rvalue reference overload if available

lvalue

- **lvalue**
  - An *lvalue* is an expression that identifies a non-temporary object or a non-member function.
- **glvalue**
  - A glvalue ("generalized" lvalue) is an expression that is either an lvalue or an xvalue
  - A glvalue may be implicitly converted to prvalue with lvalue-to-rvalue, array-to-pointer, or function-to-pointer implicit conversion.
  - A glvalue may be polymorphic: the dynamic type of the object it identifies is not necessarily the static type of the expression.

Move and &&

Good Reference

- [***http://thbecker.net/articles/rvalue_references/section_01.html***](http://thbecker.net/articles/rvalue_references/section_01.html)

```
class Derived
: public Base
{
    std::vector<int> vec;
    std::string name;
    // ...
public:
    // ...
    // move semantics
    Derived(Derived&& x)              // rvalues bind here
        : Base(std::move(x)),
          vec(std::move(x.vec)),
          name(std::move(x.name)) { }

    Derived& operator=(Derived&& x)   // rvalues bind here
    {
        Base::operator=(std::move(x));
        vec  = std::move(x.vec);
        name = std::move(x.name);
        return *this;
    }
    // ...
};
```

Initializer Lists

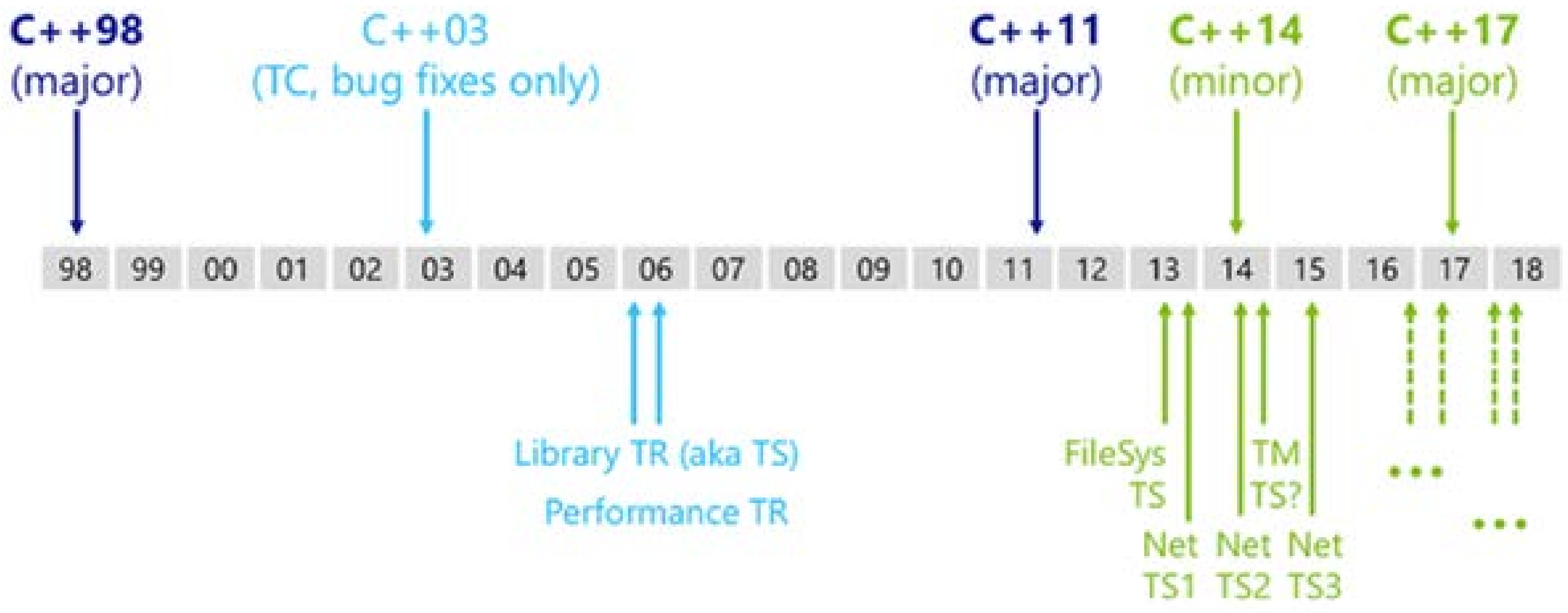
- local variable non-POD or auto
  - Continue using = syntax without extra { } braces.
- You would have used ( ) parentheses when constructing an object
  - prefer using { } braces instead
- Avoids problems:
  - narrowing conversions (e.g., float to int)
  - uninitialized POD member variables or arrays
  - occasional C++98 surprise that your code compiles but actually declares a function rather than a variable because of a declaration ambiguity in C++'s grammar – what Scott Meyers famously calls "C++'s most vexing parse."
    - Dig at Meyers by Sutter

Initializer Lists

```
// C++98
rectangle       w( origin(), extents() );
complex<double> c( 2.71828, 3.14159 );
int             a[] = { 1, 2, 3, 4 };
vector<int>     v;
for( int i = 1; i <= 4; ++i ) v.push_back(i);

// C++11
rectangle       w   { origin(), extents() };
complex<double> c   { 2.71828, 3.14159 };
int             a[] { 1, 2, 3, 4 };
vector<int>     v   { 1, 2, 3, 4 };
```

C++11 and Beyond



DePaul University

Lambda functions

- C++14 generic lambdas :
  - auto lambda = [](auto x, auto y) {return x + y;};
- C++11
  - auto lambda = [](int x, int y) {return x + y;};
- std::move function can capture a variable in a lambda expression by moving the object instead of copying or referencing it
  - std::unique_ptr ptr(new int(10));
  - auto lambda = [value = std::move(ptr)] {return *value;};

Constant Expressions

- C++11 is a function which can be executed at compile time to produce a value to be used where a constant expression is required
  - Instantiating a template with an integer argument.
  - constexpr functions only contain a single expression,
- C++14 relaxes those restrictions
  - allowing conditional statements such as if and switch, and also allowing loops, including range-based for loops

Type deduction

- C++14 allows return type deduction for all functions,
  - thus extending C++11 that only allows it for lambda functions:
    - auto DeducedReturnTypeFunction();
- Since C++14 is a strongly-typed language
  - If a function's implementation has multiple return statements, they must deduce the same type.
  - Return type deduction can be used in forward declarations, but the function definitions must be available to the translation unit that uses them before they can be used.
  - Return type deduction can be used in recursive functions, but the recursive call must be preceded by at least one return statement allowing to deduce the return type.

Thank You!

- Questions?