C++11

S. M. Niaz Arifin, PhD

Computer Science and Engineering
University of Notre Dame
Indiana, USA
Overview: C++11

- Design goals
- Features
  - auto, decltype
  - nullptr
- Strongly-typed enum
- Range-based for statements
- Lambda functions
  - Examples
- Smart Pointers
C++11

• a version of the standard for C++
• approved by International Organization for Standardization (ISO) on August 2011
  – replacing C++03
• name follows the tradition of naming language versions by the publication year
• changes to the libraries
• makes several additions to the core C++ language
Design goals

• maintain stability and compatibility with C++98 and C
• prefer introducing new features via the standard library
  – rather than extending the core language
• improve C++ to facilitate systems and library design
  – rather than introduce new features useful only to specific applications
Design goals

• increase type safety by providing safer alternatives
• increase performance, ability to work directly w/ hardware
• implement zero-overhead principle
  – further support needed by some utilities must be used only if the utility is used
• make C++ easy to teach and to learn
C++11 features: **auto**

- **before**: *auto* was a keyword used for storage duration specification
- **now**: a placeholder for a type, telling the compiler it has to deduce the actual type of a variable
- it can be used when:
  - declaring variables in different scopes
    - namespaces, blocks
  - initialization statement of for loops
C++11 features: auto

• declaring variables:
  – type of the variable that is being declared will be automatically deduced from its initializer

auto i = 42;    // i is an int
auto l = 42LL;  // l is an long long
auto p = new foo(); // p is a foo*
C++11 features: auto

• example: STL vector class

```cpp
for (std::vector<int>::const_iterator itr = myvector.cbegin(); itr != myvector.cend(); ++itr){
    ...
}
```

• with auto:

```cpp
for (auto itr = myvector.cbegin(); itr != myvector.cend(); ++itr){
    ...
}
```
C++11 features: auto

- example: STL map class
- map = an associative array
  - key names are associated with particular values
- grade_list
  - student’s name as a key, and grade as the data

```cpp
std::map<string, char> grade_list;
grade_list["John"] = 'B';
grade_list["Mary"] = 'A';
```

- grade_list with auto:

  ```cpp
  for(auto it = begin(grade_list); it != end(grade_list); ++it) {
    ...
  }
  ```
C++11 features: `decltype`

- can be used to determine the type of an expression at compile-type
  ```cpp
dcltype(5) x; // x will be type int because 5 is an int
dcltype(x) y = 6; // y will be type int because x is an int
  ```

- auto and decltype will NOT always deduce the same type
- generally, if you need a type for a variable you are going to initialize, use auto
- if you need the type for something that is not a variable (e.g., a return type), use decltype
auto and decltype

- auto can be used in place of the return type of function
- the function must have a trailing return type, specified by decltype
- auto instructs the compiler to look for the return type at the end of the function
auto and decltype

• example:

```cpp
template <typename T1, typename T2>
auto compose(T1 t1, T2 t2) -> decltype(t1 + t2)
{
    return t1 + t2;
}
```

```cpp
auto v = compose(2, 3.14); // type is double, value = 5.14
```

```cpp
std::cout << "type of v: " << typeid(b).name() << "\n" // prints double
```

• the return type of function `compose` is the return type of operator + that sums values of types T1 and T2
C++11 features: `nullptr`

- Before: 0 acted as both a constant integer and as the null pointer constant
- Odd, because:
  ```
  int *p = 1;  // illegal, can't assign an int to an int* variable
  int *q = 0;  // legal, 0 has a special meaning as a null pointer
  ```
- C++11 defines `nullptr`
  - A new reserved identifier
- Serves as a distinguished null pointer constant
- Not an integer
- Can not be converted to an integer
Strongly-typed `enum` classes

- traditional enums in C++ have drawbacks
- they are implicitly converted to integral types
- not type safe - they are treated as integers even when the enumeration types are distinct
int main()
{
    enum Color
    {
        RED,
        BLUE
    };

    enum Fruit
    {
        BANANA,
        APPLE
    };

    Color a = RED;
    Fruit b = BANANA;

    if (a == b) // The compiler will compare a and b as integers
        cout << "a and b are equal" << endl; // and find they are equal!
}

• C++ compares a and b as integers
• in the example, a does equal b since they both default to integer 0

• C++11: strongly-typed enums:

if (a == b) // compile error here, as the compiler doesn't know how to compare different types Color and Fruit
Range-based **for** statements

- before: stepping through all the values of a sequence required a lot of code
- ex.: using the iterator syntax:

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

- C++11: the **auto** keyword:

  ```
  for (auto itr = myvector.begin(); itr != myvector.end(); ++itr)
  ```

- still a lot...
Range-based for statements

• even simpler syntax to iterate through sequences
• range-based for statement
• aka “for each”
  
  ```cpp
  for (auto x: myvector) // x is read-only
  {
      cout << x;
  }
  ```

• translate this as: for each value of x in myvector
Lambda Functions

• originates in the lambda calculus
• aka a lambda expression
• a lambda is an unnamed function useful for short snippets of code
  – often impossible to reuse and/or
  – not worth naming
• you can write inline in your source code
• create quick-and-easy, unnamed functions
Lambda Functions

• basic syntax:

```
[ captures ] (parameters) ->
returnTypesDeclaration { lambdaStatements; }
```

• [ captures ]:
  – the capture clause specifies which outside variables are available for the lambda function

• ( parameters ):
  – optional parameters list
  – omit if a function takes zero arguments
Lambda Functions

• basic syntax:
  
  \[
  \text{[ captures ] \ (parameters) \ -> \ returnTypesDeclaration \{ \ lambdaStatements; \}}
  \]

• \( \rightarrow \) returnTypeDeclaration :
  
  – optional return type
  – compilers can deduce the return type when you have zero or one return statement

• \( \{ \ \lambda \lambda \lambda \text{Statements;} \} \) :
  
  – the lambda body
  – statements within the lambda body
  
  • can access the captured variables and the parameters
Lambda Functions: Examples

• Example 1: print something

```cpp
int main()
{
    auto lambda = []() { cout << "Code within a lambda function" << endl; }; 
    lambda();
}
```

• doesn’t capture any variable, takes zero arguments, and doesn’t have a return statement

• some components are optional, so the following is equivalent:

```cpp
auto lambda = [](void) -> void { cout << "Code within a lambda function" << endl; }; 
```
Lambda Functions: Examples

• Example 2: sum two variables

```cpp
int main()
{
    auto sum = [](int x, int y) { return x + y; };
    cout << sum(5, 2) << endl;
    cout << sum(10, 5) << endl;
}
```

• doesn’t capture any variable, takes two int arguments, and returns an int
Lambda Functions: Examples

- Example 3: count the number of elements in an int vector that are greater than 5
  - use count_if in the Standard Templates Library (STL)
  - non-lambda version:

```cpp
bool is_greater_than_5(int value) // returns a bool
{
  return (value > 5);
}

int main()
{
  vector<int> numbers { 1, 2, 3, 4, 5, 10, 15, 20, 25, 35, 45, 50 };  
  auto greater_than_5_count =
    count_if(numbers.begin(), numbers.end(), is_greater_than_5);

  cout << "Number of elements greater than 5 = " << greater_than_5_count << endl;
}
```
Lambda Functions: Examples

• Example 3: count the number of elements in an int vector that are greater than 5
• use of a lambda function removes the need of the additional function:

    int main()
    {
        vector<int> numbers { 1, 2, 3, 4, 5, 10, 15, 20, 25, 35, 45, 50 };
        auto greater_than_5_count =
            count_if(numbers.begin(), numbers.end(), [](int x) { return (x > 5); });
        cout << "Number of elements greater than 5 = "
            << greater_than_5_count << endl;
    }
Lambda Functions

• lambda capture clause: [ captures ]
• consider a test of divisibility
  – whether each number is divisible by another number
  – leaving no remainder
• let, int values saved in the numbers vector
Lambda Functions

• lambda capture clause: [ captures ]
• use a for_each with a lambda function
• because the user will specify different values for the divisor, we want to declare it as an int outside of the lambda function
• we also want to capture divisor by value to make it available inside of the lambda body
• within the lambda capture clause, specify the variable name (divisor) that we want to capture
Lambda Functions

- lambda capture clause: [ captures ]

```cpp
int main()
{
    int divisor = 3; // user may introduce different values
    vector<int> numbers { 1, 2, 3, 4, 5, 10, 15, 20, 25, 35, 45, 50 };
    for_each(numbers.begin(), numbers.end(), [divisor] (int y) {
        if (y % divisor == 0) {
            cout << y << endl;
        }
    });
}
```

- capture `divisor` by value to make it available `inside` of the lambda body
Lambda Functions

• lambda capture clause: [captures]
• let, we also want to compute the sum of all the numbers that are divisible by divisor
• use as [captures]: [divisor, &sum]
• the lambda captures:
  – divisor by value and
  – sum by reference (&)
Lambda Functions

• lambda capture clause: `[divisor, &sum]`

```cpp
int main()
{
    int sum = 0;
    int divisor = 3;
    vector<int> numbers { 1, 2, 3, 4, 5, 10, 15, 20, 25, 35, 45, 50 };\
    for_each(numbers.begin(), numbers.end(), [divisor, &sum] (int y) {\n        if (y % divisor == 0) {
            cout << y << endl;
            sum += y;
        }
    });
    cout << sum << endl;
}
```

• if the value is divisible by `divisor` leaving no remainder, the value is added to the `sum`
Smart Pointers

• objects that behave like built-in pointers, but also manage objects that you create with new
• no need to worry about when and whether to delete them
  – automatically delete the managed object at the appropriate time
• syntactically almost like a built-in pointer
• dynamically-allocated objects
  – who owns the object
  – what code is responsible for deleting it
Smart Pointers

• without smart pointers: where in the code we place the `delete` ...
  – memory leaks, seg faults, undefined behavior
• smart pointers make it easier to implement ownership correctly
  – smart pointer `destructor`
  – place where the object is deleted
Smart Pointers

• built-in/raw pointers:

```cpp
void Foo( )
{
    int* iPtr = new int[5];

    //manipulate the memory block
    ... 
    delete[ ] iPtr;
}
```

• in many cases, you will never reach the point where the memory is released
Smart Pointers

• C++ 11 has a new set of smart pointers, each has its own purpose:

1. shared_ptr
2. unique_ptr
3. weak_ptr
Smart Pointers

• `shared_ptr`
• shared ownership
• multiple shared pointers can refer to a single object
• when the last shared pointer goes out of scope, memory is released automatically
Smart Pointers

- **shared_ptr**
- **creation:**
  ```cpp
class MyObject { ... };
shared_ptr<MyObject> sptr1(new MyObject);
```
- **destruction:**
  releases the associated resource by calling `delete` by default
- **usage:**
  ```cpp
  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

- **unique_ptr**
  - should be used when ownership of a memory resource does **not** have to be shared
  - prevents copying of its contained pointer
  - can be transferred to another unique_ptr
    - using the `std::move` function
- **usage:**
  ```cpp
  std::unique_ptr<int> p1(new int(5));
  std::unique_ptr<int> p2 = p1; // Compile error
  std::unique_ptr<int> p3 = std::move(p1); // Transfers ownership:
      p3 now owns the memory and p1 is invalid
  ...
  p3.reset(); // Deletes the memory
  p1.reset(); // Does nothing
  ```
Smart Pointers

- **weak_ptr**
  - holds a reference to an object managed by a `shared_ptr`, but does **not** contribute to the reference count
  - used to break dependency **cycles**
  - let, a tree where the parent holds an owning reference (`shared_ptr`) to its children
  - the children also must hold a reference to the parent
  - if this second reference was also an owning one, a cycle would be created
    – no object would ever be released
  - hence, child-to-parent must be a `weak_ptr`
Thank You!