Programming with C++

COMP2011: Some New Features in C++11

Cecia Chan
Cindy Li
Pedro Sander

Department of Computer Science & Engineering
The Hong Kong University of Science and Technology
Hong Kong SAR, China

A List of New Features in C++11

- uniform and general initialization using { }-list ⋆
- type deduction of variables from initializer: auto
  — NOT ALLOWED TO USE IN COMP2011
- prevention of narrowing ⋆
- generalized and guaranteed constant expressions: constexpr
- Range-for-statement ⋆
- null pointer keyword: nullptr ⋆
- scoped and strongly typed enums: enum_class
- rvalue references, enabling move semantics †
- lambdas or lambda expressions ⋆
- support for unicode characters
- long long integer type
- delegating constructors †
- in-cass member initializers †
- explicit conversion operators †
- override control keywords: override and final †

Part I

General Initialization Using { }-Lists

{ }

= and { } Initializer for Variables

- In the past, you always initialize variables using the assignment operator =.

Example: = Initializer

```
int x = 5;
float y = 9.8;
int& xref = x;
int a[] = {1, 2, 3};
```

- C++11 allows the more uniform and general curly-brace-delimited initializer list.

Example: { } Initializer

```
int x = {5};        // = here is optional
float y {9.8};
int& xref {x};
int a[] {1, 2, 3};
```

Initializer Example 1

```
1   #include <iostream>     /* File: initializer1.cpp */
2   using namespace std;
3
4   int main()
5   {
6       int w = 3.4;
7       int x1 {6};
8       int x2 = {8};       // = here is optional
9       int y {'k'};
10      int z {6.4};        // Error!
11
12      cout << "w = " << w << endl;
13      cout << "x1 = " << x1 << endl << "x2 = " << x2 << endl;
14      cout << "y = " << y << endl << "z = " << z << endl;
15
16      int& ww = w;
17      int& www {ww}; www = 123;
18      cout << "www = " << www << endl;
19      return 0;
20  }
```

```
initializer1.cpp:10:15: error: narrowing conversion of 6.4000000000000004e+0
from double to int inside { } [-Wnarrowing]
     int z {6.4};
               ^
```

Initializer Example 2

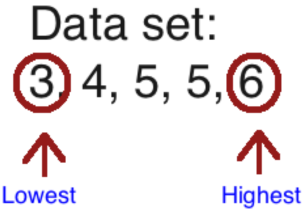
```cpp
#include <iostream>     /* File: initializer2.cpp */
using namespace std;

int main()
{
    const char s1[] = "Steve Jobs";
    const char s2[] {"Bill Gates"};
    const char s3[] = {'h', 'k', 'u', 's', 't', '\0'};
    const char s4[] {'h', 'k', 'u', 's', 't', '\0'};

    cout << "s1 = " << s1 << endl;
    cout << "s2 = " << s2 << endl;
    cout << "s3 = " << s3 << endl;
    cout << "s4 = " << s4 << endl;
    return 0;
}
```

Differences Between the = and { } Initializers

- The { } initializer is more restrictive: it doesn't allow conversions that lose information — narrowing conversions.
- The { } initializer is more general as it also works for:
  - arrays
  - other aggregate structures
  - class objects (we'll talk about that later)

Part II

Range-for-Statement



for-Statements

- In the past, you write a for-loop by
  - initializing an index variable,
  - giving an ending condition, and
  - writing some post-processing that involves the index variable.

Example: Traditional for-Loop

```
for (int k = 0; k < 5; ++k)
    cout << k*k << endl;
```

- C++11 adds a more flexible range-for syntax that allows looping through a sequence of values specified by a list.

Example: Range-for-Loops

```
for (int k : { 0, 1, 2, 3, 4 })
    cout << k*k << endl;

for (int k : { 1, 19, 54 }) // Numbers need not be successive
    cout << k*k << endl;
```

Range-for Example

```
#include <iostream>     /* File : range-for.cpp */
using namespace std;

int main()
{
    cout << "Square some numbers in a list" << endl;
    for (int k : {0, 1, 2, 3, 4})
        cout << k*k << endl;

    int range[] { 2, 5, 27, 40 };

    cout << "Square the numbers in range" << endl;
    for (int k : range)  // Won't change the numbers in range
        cout << k*k << endl;

    cout << "Print the numbers in range" << endl;
    for (int v : range) cout << v << endl;

    for (int& x : range) // Double the numbers in range in situ
        x *= 2;

    cout << "Again print the numbers in range" << endl;
    for (int v : range) cout << v << endl;
    return 0;
}
```

Part III

Local Anonymous Functions — Lambdas

Lambda Expressions (Lambdas)

Syntax: Lambda

[<capture-list>] (<parameter-list>) mutable →<return-type> { <body> }

- They are anonymous functions — functions *without* a name.
- They are usually defined locally inside functions, though global lambdas are also possible.
- The capture list (of variables) allows lambdas to use local variables that are already defined in the enclosing function.
  - [=]: capture all local variables by value.
  - [&]: capture all local variables by reference.
  - [variables]: specify only the variables to capture
  - global variables can always be used in lambdas without being captured. In fact, it is an error to capture them in a lambda.
- The return type
  - is void by default if there is no return statement.
  - is automatically inferred if there is a return statement.
  - may be explicitly specified by the → syntax.

Example: Simple Lambdas with No Captures

```
#include <iostream>     /* File : simple-lambdas.cpp */
using namespace std;

int main()
{
    // A lambda for computing squares
    int range[] =  { 2, 5, 7, 10 };
    for (int v : range)
        cout << [](int k) { return k * k; } (v) << endl;

    // A lambda for doubling numbers
    for (int& v : range) [](int& k) { return k *= 2; } (v);
    for (int v : range) cout << v << "\t";
    cout << endl;

    // A lambda for computing max between 2 numbers
    int x[3][2] = { {3, 6}, {9, 5}, {7, 1} };
    for (int k = 0; k < sizeof(x)/sizeof(x[0]); ++k)
        cout << [](int a, int b) { return (a > b) ? a : b; } (x[k][0], x[k][1])
             << endl;

    return 0;
}
```

Example: Lambdas with Captures

```
 1  #include <iostream>     /* File : lambda-capture.cpp */
 2  using namespace std;
 3  int main()
 4  {
 5      int sum = 0, a = 1, b = 2, c = 3;
 6
 7      for (int k = 0; k < 4; ++k) // Evaluate a quadratic polynomial
 8          cout << [=](int x) { return a*x*x + b*x + c; } (k) << endl;
 9      cout << "a = " << a << "\tb = " << b << "\tc = " << c << endl;
10
11      for (int k = 0; k < 4; ++k) // a and b are used as accumulators
12          cout << [&](int x) { a += x*x; return b += x; } (k) << endl;
13      cout << "a = " << a << "\tb = " << b << "\tc = " << c << endl;
14
15      for (int v : { 2, 5, 7, 10 }) // Only variable sum is captured
16          cout << [&sum](int x) { return sum += a*x; } (v) << endl; // Error!
17      cout << "sum = " << sum << endl;
18
19      return 0;
20  }
```

```
lambda-capture.cpp:16:47: error: variable 'a' cannot be implicitly captured
      in a lambda with no capture-default specified
        cout << [&sum](int x) { return sum += a*x; } (v) << endl;
                                              ^
```

Example: When Are Values Captured?

```
#include <iostream>     /* File : lambda-value-binding.cpp */
using namespace std;

int main()
{
    int a = 1, b = 2, c = 3;
    auto f = [=](int x) { return a*x*x + b*x + c; };

    for (int k = 0; k < 4; ++k)
        cout << f(k) << endl;
    cout << "a = " << a << "\tb = " << b << "\tc = " << c << endl;

    a = 11, b = 12, c = 13;
    for (int k = 0; k < 4; ++k)
        cout << f(k) << endl; // Will f use the new a, b, c?
    cout << "a = " << a << "\tb = " << b << "\tc = " << c << endl;

    return 0;
}
```

- The keyword auto allows one to declare a variable without a type which will be inferred automatically by the compiler.
- WARNING: You are not allowed to use auto in this course!

Example: When Are References Captured?

```
#include <iostream>     /* File : lambda-ref-binding.cpp */
using namespace std;

int main()
{
    int a = 1, b = 2, c = 3;
    auto f = [&](int x) { a *= x; b += x; c = a + b; };

    for (int k = 1; k < 3; f(k++))
        ;
    cout << "a = " << a << "\tb = " << b << "\tc = " << c << endl;

    a = 11, b = 12, c = 13;
    for (int k = 1; k < 3; f(k++)) // Will f use the new a, b, c?
        ;
    cout << "a = " << a << "\tb = " << b << "\tc = " << c << endl;

    return 0;
}
```

Question: What is the printout now?

Capture by Value or Reference

- When a lambda expression captures variables by value, the values are captured by copying only once at the time the lambda is defined.
- Capture-by-value is similar to pass-by-value.
- Unlike PBV, variables captured by value cannot be modified inside the lambda unless you make it mutable.

Examples

```
/* File: mutable-lambda.cpp*/
int a = 1, b = 2;

cout << [a](int x) { return a += x; } (20) << endl; // Error!
cout << [b](int x) mutable { return b *= x; } (20) << endl; // OK!
cout << "a = " << a << "\tb = " << b << endl;
```

- Similarly, capture-by-reference is similar to pass-by-reference.

Example: Mutable Lambda with Return

```
#include <iostream>     /* File : mutable-lambda-with-return.cpp */
using namespace std;

int main()
{
    float a = 1.6, b = 2.7, c = 3.8;

    // [&, a] means all except a are captured by reference; a by value
    auto f = [&, a](int x) mutable ->int { a *= x; b += x; return c = a+b; };

    for (int k = 1; k < 3; ++k)
        cout << "a = " << a << "\tb = " << b << "\tc = " << c
             << "\tf(" << k << ") = " << f(k) << endl;

    cout << "a = " << a << "\tb = " << b << "\tc = " << c << endl;
    return 0;
}
```

- One may mix the capture-default [=] or [&] with explicit variable captures as in [&, a] above.
- In this case, all variables but a are captured by reference while a is captured by value.