Programming with C++

COMP2011: Some New Features in C++11

Cecia Chan
Cindy Li
Brian Mak

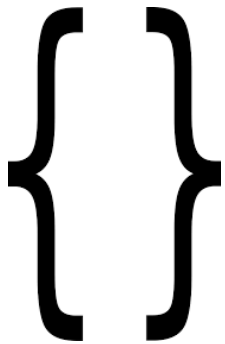
Department of Computer Science & Engineering
The Hong Kong University of Science and Technology
Hong Kong SAR, China

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-class 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

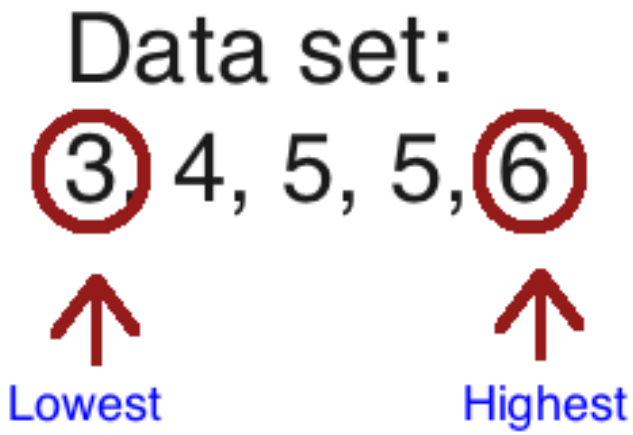
```
1   #include <iostream>     /* File: initializer2.cpp */
2   using namespace std;
3
4   int main()
5   {
6       const char s1[] = "Steve Jobs";
7       const char s2[] {"Bill Gates"};
8       const char s3[] = {'h', 'k', 'u', 's', 't', '\0'};
9       const char s4[] {'h', 'k', 'u', 's', 't', '\0'};
10
11      cout << "s1 = " << s1 << endl;
12      cout << "s2 = " << s2 << endl;
13      cout << "s3 = " << s3 << endl;
14      cout << "s4 = " << s4 << endl;
15      return 0;
16  }
```

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
  - class objects
  - other aggregate structures

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

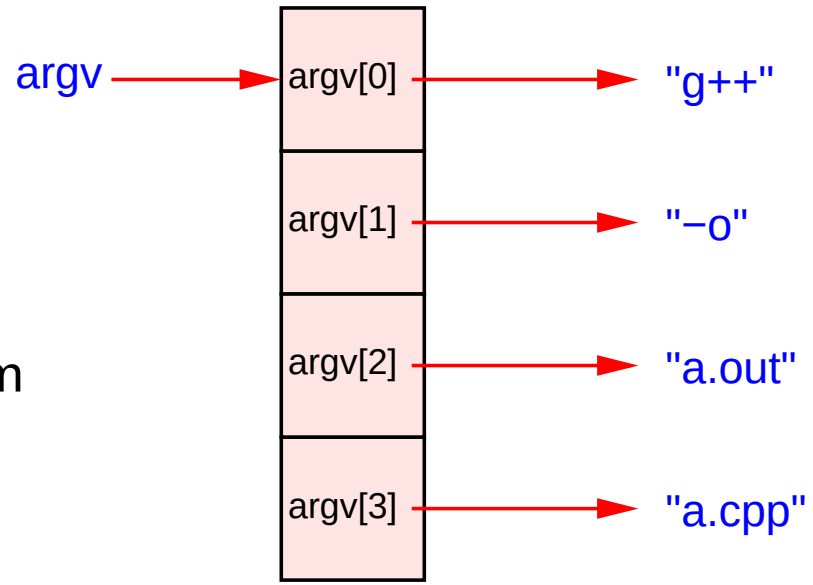
Arguments for the main( ) Function

main( ) Function Arguments

- Up to now, you write the main function header as `int main( )` or `int main(void)`.
- In fact, the general form of the main function allows variable number of arguments (overloaded function).

```
int main(int argc, char** argv)
int main(int argc, char* argv[ ])
```

- argc gives the actual number of arguments.
- argv is an array of char*, each pointing to a character string.
- e.g. `g++ -o a.out a.cpp` calls the main function of the g++ program with 3 additional commandline arguments. Thus, argc = 4, and



Example: Operations of a Dynamic 2D Array using argv

```
#include <iostream> /* File: 2d-dynamic-array-main-with-argv.cpp */
using namespace std;
int** create_matrix(int, int);
void print_matrix(const int* const*, int, int);
void delete_matrix(int**, int, int);

int main(int argc, char** argv)
{
    if (argc != 3)
    { cerr << "Usage: " << argv[0] << " #rows #columns" << endl; return -1; }

    int num_rows = atoi(argv[1]);
    int num_columns = atoi(argv[2]);
    int** matrix = create_matrix(num_rows, num_columns);

    // Dynamic array elements can be accessed like static array elements
    for (int j = 0; j < num_rows; ++j)
        for (int k = 0; k < num_columns; ++k)
            matrix[j][k] = 10*(j+1) + (k+1);

    print_matrix(matrix, num_rows, num_columns);
    delete_matrix(matrix, num_rows, num_columns);
    matrix = nullptr; // Avoid dangling pointer
    return 0;
} /* g++ 2d-dynamic-array-main-with-argv.cpp 2d-dynamic-array-functions.cpp */
```