04-10 Discussion Notes  
PIC 10B  
Spring 2018

1 List Initialization

It is sometimes cumbersome to place specific data into containers. For example, say we wish to place the names “Bob”, “Alice”, “Jane” into a vector. One way to do this is as follows:

```cpp
1 std::vector<std::string> names;
2 names.push_back("Bob");
3 names.push_back("Alice");
4 names.push_back("Jane");
```

Not only is this code inelegant and redundant, it is also inefficient. It would be much better if we could directly initialize the vector to contain the desired names at construction time. This is possible using list-initialization:

```cpp
1 std::vector<std::string> names{
2   "Bob", "Alice", "Jane"
3};
```

In this code, the expression {...} is called a braced-init-list, and signifies to the compiler that we want to initialize the vector using list-initialization. What list-initialization actually does depends on the object being initialized, but in the case of an STL container list std::vector it’s simple: the std::vector class contains a constructor that accepts an std::initializer_list object, so the compiler casts the braced-init-list into an std::initializer_list and passes the result into the aforementioned constructor. (An std::initializer_list is a lightweight class that contains an array of objects.) The constructor then creates an std::vector containing the data in the std::initializer_list.

Do not mix up the initializer-lists used in constructors with the list initialization discussed in this section. Despite the similar terminology and syntax, they are two completely separate concepts.

2 Some syntactic sugar

2.1 Range-based for loops

Range-based for loops are syntactic sugar in the C++ language that allows us to conveniently iterate through a container. Indeed, the loops in the following code snippet are equivalent

```cpp
1 std::list<std::string> container{"thanks", "for", "reading!"); // add strings to an unordered_set
2
3 for (std::list<std::string>::iterator it = std::begin(container); it != std::end(container); ++it) {
4   *it += " "; // add a space to each string
5 }
6
7 for (std::string& word : container) {
8   word += " "; // add a space to each string
9 }
```
We can see that the code for the range-based for-loop is much more natural looking, and this is because range-based for loops allow us to directly access the values stored in a container without iterators. Range-based for-loops consist of a declaration before the colon and an expression after the color specifying the container through which to iterate. Here, we declare word to be a reference because we wish to modify the values in the container. However, we are not limited to references as the code below shows:

```cpp
for (std::string word : container) {
    std::cout << word << "\n";
}
for (const std::string& word : container) {
    std::cout << word << "\n";
}
```

### 2.2 The auto keyword

In the first code snippet of the previous section, we had to write the long string of characters `std::list<std::string>::iterator` in order to define an iterator in the for-loop. The awkward length of the string of characters is because the iterator class is nested within the `std::list` class, and this kind of nested structure is very common in C++ so we will be forced to type out long strings of characters frequently. Ameliorating for this unfortunate set of affairs, the designers of C++ added a keyword `auto` that can replace type specifiers such as the long string above. When `auto` is used, the compiler will automatically deduce what type a variable should be based upon the expression being assigned to the variable.

```cpp
std::string fun() {  
    return "Hello world!";
}
int main() {  
    auto x = 5; // compiler deduces x should be an int  
    auto y = fun(); // compiler deduces y should be an std::string from the return type of fun  
}
```

As mentioned, `auto` is particularly useful when having to deal with iterators because of their long names.

```cpp
std::list<std::string> container{"thanks", "for", "reading!");  // add strings to an unordered_set
for (auto it = std::begin(container) ; it != std::end(container) ; ++it) {  
    *it += " "; // add a space to each string
}
```

One can also specify whether `auto` should be const or of reference type, as we do in the range-based for loops below

```cpp
for (auto& word : container) {  
    word += " "; // word is of type std::string&
}
for (const auto& word : container) {  
    std::cout << word << "\n"; // word is of type const std::string&
}
for (auto word : container) {  
    word += " "; // word is of type std::string. Note this code does not change the values in container
```
3 Default values

Function parameters can be given default values as follows

```cpp
std::string fun(int x, double y = 3.14, std::string z = "Hello world!") {
    return z+y+x;
}

int main() {
    std::string a = fun(1); // a = "Hello world!3.141"
    std::string b = fun(1,2.75); // b = "Hello world!2.751"
    std::string c = fun(1,2.75,"Goodbye world!"); // c = "Goodbye world!2.751"
}
```

As seen above

1. Parameters with default values must be listed after parameters without default values.
2. Parameters passed in a function call (sometimes called arguments) are matched left to right with parameters in the function definition (sometimes called formal parameters), and any unmatched formal parameters are given their default values.

4 using and typedef

`using` and `typedef` are used to give typenames easier to remember “nicknames.” They do exactly the same thing, but have slightly different syntax. We focus on the `using` keyword, and consider three ways it is commonly used:

```cpp
using std::cout;
using intveciter = std::vector<int>::iterator;
using namespace std;
```

The code above has the following effects:

1. Line 1: allows for unqualified name lookup of the symbol std::cout. This means that we can simply use cout instead of std::cout.
2. Line 2: creates an alias for the symbol std::vector<int>::iterator. This means we can use intveciter in place of std::vector<int>::iterator.
3. Line 3: allows for unqualified name lookup of any name in the namespace std. Unqualified means it is unnecessary to use the “qualification” std::.

5 Compile-time computations

In C++ it is sometimes convenient, and sometimes required, to evaluate an expression at compile-time rather than at runtime. The keyword `constexpr` is used in C++ to tell the compiler that
a function should be evaluated at compile-time and to tell the compiler that a variable should be assigned a value at compile-time. However, not all expressions can be evaluated at compile-time: any expression depending on aspects of the runtime environment, such as the current time or user input, cannot be evaluated at runtime, so really constexpr asks the compiler to do a compile-time evaluation, but the compile may turn you down.

```cpp
#include <array>

constexpr unsigned factorial(unsigned n) {
    return (n <= 1) ? 1 : n * factorial(n - 1); // compute factorial recursively. The strange syntax is a ternary statement.
}

int main() {
    constexpr unsigned sz = factorial(15);
    std::array<int, sz> arr; // template arguments need to be evaluated at compile-time so the compiler knows which classes to make available
}
```

In the above code, we have a constexpr function that recursively calculates the factorial of a number. We use this function as a template argument to the std::array class. Template arguments must be known at compile-time, so we are forced to use constexpr here. A non-recursive example is given below,

```cpp
#include <array>

constexpr unsigned getSize(unsigned n) {
    return n % 7;
}

int main() {
    constexpr unsigned sz = getSize(106);
    std::array<int, sz> arr; // template arguments need to be evaluated at compile-time so the compiler knows which classes to make available
}
```

### 6 Passing by reference vs passing by copy

We’ve been told many times that passing by reference is more efficient for large objects because using references avoids copying large amounts of data from one scope to another scope in our code. It may be pretty obvious, but if we do wish to copy data, then we should avoid references and pass by copy instead. An example is given below

```cpp
// efficient
std::string addAy(std::string x) {
    return x + "ay";
}

// inefficient
std::string addAy(const std::string& x) {
    std::string y(x);
    return y + "ay";
}
```
Here, we wish to modify the value passed into the function without modifying the variable used as an argument to the function. Thus, in this case, it makes sense to pass by value since passing by value creates a copy, which we would do anyway as seen on line 8.

7 The const keyword and pointers

When using pointers, the const keyword can either refer to the object being pointed to or the pointer itself. We call a pointer to a const variable a “pointer to const,” and a pointer that cannot itself be moved a “const pointer.” A pointer to const believes the variable it points to is const, which means that it cannot be used to modify the variable it points to. A const pointer is forced to point to the same position in memory for its lifetime: it cannot be moved. Examples are given below:

```c
int main() {
    int num = 5; // x is a pointer to an int
    int* x = &num; // ptr1 is a pointer that believes the variable it points to is const
    int const* ptr1 = x; // same as previous
    int* const ptr3 = x; // ptr3 is a const pointer to an int
    int const* const ptr4 = x; // ptr4 is a const pointer to const
}
```

We summarize some facts below:

1. const pointers cannot be moved, so operations such as incrementing (++) or decrementing (–) are invalid
2. pointers to const cannot modify the data pointed to, so one cannot dereference and change the value pointed to.
3. a pointer to const can point to a non-const variable; however, a regular pointer cannot point to a const variable.