Uniform Initialization; Initializer Lists

ITP 435 – Spring 2016
Week 6, Lecture 2

Lecturer: Sanjay Madhav

Making Copy Constructors Unusable

- The old way of disallowing copy constructors was to declare them private, like:

```
class Test {
public:
   // Allow default constructor
   Test() { }
   // (Destructor is implicit)
private:
   // Disallow copy/assignment
   Test(const Test& other) { }
   Test& operator=(const Test& other) { }
};
```

Default/Delete Syntax

- In C++11, there is a more succinct way to describe the same:

```
class Test {
public:
   // Use default constructor for no parameters
   Test() = default;
   // (Destructor is implicit)
   // Delete copy constructor/assignment operator
   Test(const Test& other) = delete;
   Test& operator=(const Test& other) = delete;
};
```

- ***A deleted operator/constructor cannot be invoked!***

Basic Initialization in C++11

- There are way too many ways to initialize variables in C++11:

```
int w(0);       // Initializer in parenthesis

int x = 0;      // Initializer after =

int y{ 0 };     // Initializer in braces

int z = { 0 }; // Equals and braces!
```

Problems w/ Basic Initialization

- Not all forms of initialization work in all cases – for example, if the copy constructor is deleted…

```
// Clang error: Copying invokes deleted constructor
std::atomic<int> a1 = 5;

std::atomic<int> a2(5);       // Okay

std::atomic<int> a3{ 5 };     // Okay

std::atomic<int> a4 = { 5 }; // Okay
```

USC Viterbi School of Engineering University of Southern California

All of these forms actually work in Visual Studio, since it has very relaxed C++11 conformity

Initializing Non-Static Member Data

- In C++98, the best way to initialize member data is in the initializer list:

```
class Test {
public:
   Test()
      : x(20)
   { }
private:
   int x;
};
```

Initializing Non-Static Member Data, Cont'd

- In C++11, you are allowed to initialize non-static data at the point of declaration – provided you use either equals ***or*** braces (or both):

```
class Test {
public:
private:
   int x = 5;       // Works
   int y = { 25 }; // Works
   int z{ 125 };   // Works
};
```

USC Viterbi School of Engineering

University of Southern California

If you look at a constructed Test in the debugger, the member variables will be initialized to 5, 25, and 125!

Initializing Non-Static Member Data, Cont'd

- ***However***, parenthesis syntax is rejected:

```
class Test {
public:
private:
   int x = 5;       // Works
   int y = { 25 }; // Works
   int z{ 125 };   // Works
   int a(125);     // ERROR!
};
```

Uniform Initialization

- ***Uniform initialization*** is the idea of one initialization syntax that is valid wherever initialization is valid

- In order to maintain backwards compatibility in old code, the ***braces initialization*** was selected as the uniform initialization:

```
// This is a uniform initialization
int y{ 0 };

// This is (almost always) also a uniform initialization
int z = { 0 };
```

USC Viterbi School of Engineering

University of Southern California

Personally, I prefer the second syntax to the first

Difference Between Brace Syntaxes

- = { } will ignore explicit constructors…

```
class Test {
public:
   explicit Test(int i) : x(i) { }
private:
   int x;
};
```

- Then:

```
// Uniform Initialization - Works!
Test t1{ 5 };

// = { } ignores explicit constructors - ERROR! :(
Test t2 = { 5 };
```

Narrowing

- A novel aspect of uniform initialization is that it does not allow ***narrowing*** from a less constrained to a more constrained type

- Examples:

```
char c1{ 50 }; // Fine, 50 can fit in a char
char c2{ 1234 }; // ERROR - Can't narrow 1234 -> char


int i1{ 10 }; // Fine
int t2{ 10.0 }; // ERROR - Can't narrow float -> int
```

Aggregate Initialization

- You can also use the uniform initialization to initialize a class, struct, or union, provided that:
  - There's no private/protected members
  - No constructors (other than default/delete)
  - No base class
  - No virtual functions
  - No brace/equal initializers for non-static members

Aggregate Initialization, Cont'd

```
struct Test1 {
    int x;
    int y;
    int z;
};
```

- You can initialize each member via the uniform initialization (in the order in which they are declared)

```
// Initializes
// x = 50
// y = -50
// z = 25
Test1 t{ 50, -50, 25 };
```

Aggregate Initialization, Cont'd

- Also works when nesting aggregates that satisfy the conditions!

<table>
<tr><th>Declaration</th><th>Initialization</th></tr>
<tr><td><pre>struct Point {
   int x;
   int y;
   int z;
};

struct Test2 {
   Point topLeft;
   Point botRight;
};</pre></td><td><pre>Test2 t2{
   {5, 10, 15}, // Top left
   {2, 4, 6}, // Bottom right
};</pre></td></tr>
</table>

Uniform Initialization and STL Collections

- The great thing about uniform initialization is it works to initialize STL collections!

```
// Initialize a vector of even numbers
std::vector<int> v{ 2, 4, 6, 8, 10 };

// Initialize a list of odd numbers
std::list<int> l{ 1, 3, 5, 7, 9 };

// Create a pair
std::pair<std::string, int> p{ "Hello", 5 };
```

Vector of Pairs

- Works even with nesting!

```
// Vector of pairs
std::vector<std::pair<int, std::string>> months{
   { 1, "January" },
   { 2, "February" },
   { 3, "March" },
   // ...
};
```

- ***Q: How does this actually work?***

std::initializer_list

- Included in the header `<initializer_list>`
- It's a templated and lightweight proxy class to a temporary array
- Only supports the following operations:
  - Size
  - Begin and end iterators
  - That's it!
- This can be used to create an initializer_list constructor

Example

- Suppose we have a standardish dynamic array:

```
template <typename T>
class DynArray {
public:
   // Functions here ...
private:
   size_t mSize;
   T* mData;
};
```

USC Viterbi School of Engineering

University of Southern California

Example, Cont'd

- Some standard functions:

```
// Default Constructor
DynArray()
: mSize(10)
{
   mData = new T[mSize];
}
// Destructor
~DynArray() {
   delete[] mData;
}
// Constructor to specify initial size
DynArray(size_t size)
: mSize(size)
{
   mData = new T[mSize];
}
```

Example, Cont'd

- The initializer list constructor:

```
// Initializer list constructor
DynArray(const std::initializer_list<T>& list) {
   mSize = list.size();
   mData = new T[mSize];
   int i = 0;
   for (const T& val : list) {
      mData[i] = val;
      i++;
   }
}
```

Example, Cont'd

- Now we can call the initializer_list constructor using the uniform initialization syntax:

```
// Calls the initializer list constructor
DynArray<int> test{ 5, 10, 15, 20, 25 };
```

- A side effect is that if there is an initializer_list constructor, it'll always be preferred when using uniform initialization

Example, Cont'd

- A side effect is that if there is an initializer_list constructor, it'll ***always*** be preferred when using uniform initialization:

```
// This STILL calls the initializer list constructor
// (Creates a list of size 1 with the value 5)
DynArray<int> test2{ 5 };

// To call the constructor that takes the initial size
// only, you have to use the old syntax...
// (Creates a list of size 5 with no data)
DynArray<int> test3(5);
```

- This is the one thing to watch out for when using uniform initialization!

Something that's annoying…

- Why can't I just do:

```
// Test 2
someFunctionThatTakesAList({1, 2, 3});
```

- "In Python you could totally do this!" – Random Python programmer

std::initializer_list

- Great way to make a function that can take an arbitrary number of arguments, provided they are all the same type.

```
#include <initializer_list>
int addList(const std::initializer_list<int>& list)
{
   int retVal = 0;
   for (auto i : list)
   {
      retVal += i;
   }

   return retVal;
}
```

- Then later this function can be called like this:

```
// Outputs 33
std::cout << addList({1, 1, 2, 3, 5, 8, 13}) << std::endl;
```

std::initializer_list Constraint

- Remember, it only works if all elements in the list are the same type
- But this should usually be the case…
- And you can use pairs/tuples if you want to pack them in further

std::initializer_list with std::pair

```cpp
void printMonths(const std::initializer_list<std::pair<int,
                        std::string>> & list)
{
   for (auto i : list) {
      std::cout << i.first << ":" << i.second << std::endl;
   }
}

// Later...
printMonths({
   { 1, "January" },
   { 2, "February" },
   { 3, "March" },
   // ...
});
```

A more real use

- I use initializer lists in some of the code in ITP 439:

```
// Returns true if the current token matches one of the tokens
// in the list.
bool Parser::peekIsOneOf(const std::initializer_list<Token::Tokens>& list)
   noexcept
{
   for (Token::Tokens t : list)
   {
      if (t == peekToken())
      {
         return true;
      }
   }
   return false;
}
```

Pair

- std::pair is a pair of elements, it's not restricted to just maps
- You could make a vector like this, for example:

```
#include <vector>
// This vector stores pairs of month numbers and names
std::vector<std::pair<int, std::string>> v;
// Add January
v.push_back(std::pair<int, std::string>(1, "January");
// This would output 1,January
std::cout << v[0].first << "," << v[0].second;
```

Tuple

- std::tuple is like pair but you can have an unlimited number of elements

```
#include <vector>
#include <tuple>
// This vector stores tuples with month num, short name, long name
std::vector<std::tuple<int, std::string, std::string>> v;
// Add January
v.push_back(std::make_tuple(1, "Jan", "January"));
// This would output 1,Jan,January
std::cout << std::get<0>(v[0]) << "," << std::get<1>(v[0]) << "," <<
   std::get<2>(v[0]);
```