C++ Initialization Syntax

CIS-3012, C++ Programming

Vermont State University

Peter Chapin

Let’s Talk about C

- An object can be initialized as follows:

Type specifier → `int` ; Declarator → `x` ; Initializer (optional) → `42`

```
int x = 42;
```

- Without an initializer (“uninitialized”)…
  - … global variables are automatically initialized to 0, NULL, etc.
    - (global variables can only have constant expressions as initializers)
  - … local variables have indeterminate initial values.

C Array Initialization

- Basic rules:
  - Programmer provides an *initializer list* with one initializer for each element of the array.
  - Compiler can deduce the dimension of the array from the length of the initializer list.

```
int array_1[10] = { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 };
```

Compiler deduces the array's size by counting initializers.

```
int array_2[ ]  = { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 };
```

Extra array elements (at the end) are always zero-initialized.

```
int array_3[15] = { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 };
```

C Structure Initialization

- Basic Rules:
  - Again, the programmer provides an initializer list, except this time the initializer types might be different (to match the structure member types)

Structure definition (typically, but not necessarily, in a header file).

```
struct Example { int x; double y; const char *z; };
```

In C, you must prefix the name of a structure with “struct”. That isn’t necessary in C++.

```
struct Example x = {
    42, 3.14159, “Hello, World”
};
```

Designated Initializers

- Starting with C99 (and C++ 2020), you can use *designated initializers*.

```
struct Example { int x; double y; const char *z; };
```

```
struct Example x = {
    .y = 3.14159,
    .x = 42,
    .z = “Hello, World”
};
```

Notice that the initializers do not have to be in “declaration order.” More flexible, better documented.

Designated Initializers for Arrays

- It works for arrays too (as of C99 or C++ 2020):

```
int array[10] = {
    [0] = 2,
    [1] = 3,
    [3] = 7,
    [2] = 5
};
```

Initializers need not be in order.
Any elements with missing initializers always get zero-initialized.

Now C++…

- There are several different initialization syntaxes that can be used, each with its own special features and rules.
  1. C-style initialization (see previous slides). Note that designated initializers are **not** part of C++ until C++ 2020.
  2. Function-like initialization. This syntax is necessary for dealing with constructors taking multiple parameters.
  3. Uniform initialization syntax. Starting with C++ 2011, this syntax unifies C-style initialization, constructor parameters, and initialization lists into a single, unified syntax.

Why So Complicated?

- C++'s advanced features create situations where the C-style initialization syntax just isn't good enough.
- C++ 1998 added the function-like syntax to address some of the issues, but some issues remained.
- C++ 2011 added the uniform syntax to address the remaining issues.
- A school of thought says, therefore, that in modern C++ code, you should use the uniform syntax for **all** initializations.
  - But that turns out not to work 100%. There are some ambiguities, there is C compatibility, and sometimes the older syntaxes just look more natural.
  - Thus, *none of the syntaxes are deprecated*. <u>You should be familiar with all</u>.

Function-Like Initialization

- Here is how it looks:

```
int x = 42;    // Traditional (C-style) initialization.
int y( 42 );   // The same, but using function-like initialization.
```

- It's called "function-like" because it looks sort-of like calling a function.

Ambiguous?

- Consider this:

```
int x( 42 );  // The declaration of an object, initialized to 42.
int y( int ); // The declaration of a function returning int.
```

- The essential difference:
  - In a function declaration, the thing inside the parentheses is a list of *parameter declarations*.
  - In an object declaration, the thing inside the parentheses is a list of *expressions*.
- This tends to be less confusing in practice than it sounds.

What’s the Point?

- Function-like initializations exist for constructors with multiple parameters:

```
string separator(64, ‘*’);
```

- This uses function-like initialization to call a two-parameter constructor to create a string named `separator` consisting of 64 asterisk characters.
  - The C-style initialization syntax using the = sign can’t do this. Some sort of new syntax (relative to C) was needed.
  - The function-like initialization syntax is part of C++98.

With Dynamic Allocation

- A similar syntax can be used for dynamically allocated objects:

```
string *separator = new string(64, '*');
```

- Here `separator` is a *raw* pointer that points at the dynamically allocated object.
  - This is similar to Java syntax for creating dynamically allocated objects and initializing them (by calling a constructor).
  - As an aside: *using raw pointers in modern C++ is discouraged*. That's a subject for another slide deck!

Explicit Constructor Call

- It is possible to do this:

```
int x;           // A couple of ordinary looking declarations for context.
int y = 42;
string(64, '*'); // Explicitly construct an anonymous object of type string.
```

- Since the anonymous object has no name, this isn't very useful.
  - Although the string constructor and destructor still execute.
- But…

Using an Explicit Constructor Call, Part 1

- Passed as an argument:

```
// Declaration of a function (probably in a header file)
void do_something( const string &text );

// Call that function using an explicitly constructed temporary.
do_something( string( 64, '*' ) );
```

- It is important that the function takes its parameter as reference to const.
  - The compiler knows the function won't try to change the temporary.
  - The compiler **will not bind a non-const reference to a temporary**!

Using an Explicit Constructor Call, Part 2

- Returning a value:

```
string make_string( int x, int y )
{
    // …

    return string( 64, '*' );
}
```

- The function returns an explicitly constructed temporary.
  - In real life the constructor arguments would doubtless be the result of some “interesting” computation inside the function.
  - Note that the temporary is copied to the caller.

Temporaries?

- The compiler generates temporaries to hold explicitly constructed objects (although sometimes they can be removed by optimizations).

```
void do_something( const string &text );

// When you do:
// do_something( string( 64, '*' ) );

// … the compiler generates temporary with some internal name:

string t__103CF7( 64, '*' );
do_something( t__103CF7 );
```

Seems Complicated

- It isn't. Compiler generated temporaries are common and normal. Consider this example:

```
void do_something( int value );

// You write this:
do_somthing( x + y );

// The compiler does this:
int t__7A303C = x + y;
do_something( t__7A303C );

// For simple types like integers, the temporary is probably in a register
```

Copy Initialization

- If a class has a constructor that *can be called with one argument*…
  - (the phrase “can be called with one argument” is intended to cover constructors with multiple parameters but for which all but one have default arguments).
- … a C-style syntax can be used to initialize.

```
string name = “Jill”;

// This is the same as:
string t__xyzzy( “Jill” ); // Construct string from const char * argument.
string name( t_xyzzy );    // Copy the temporary string into the named string.
```

- Often the temporary can be “optimized away.”

Implicit Type Conversion

- This same idea allows you to do something like this:

```
// Declaration of a function (probably in a header file)
void do_something( const string &text );


// Create a temporary string from a const char *, etc.
do_something( "Jill" );
```

- *A constructor that can be called with one argument serves as an implicit type conversion from the type of the parameter to the type of the class.*

Implicit Type Conversion (and by the way…)

- This same idea allows you to do something like this:

```
// Declaration of a function (probably in a header file)
void do_something( const string &text );


// Create a temporary string from a const char *, etc.
do_something( "Jill" );
```

If you remove **const** here…

… this is an <u>error</u>!
The compiler won't bind a reference to non-const to a temporary!

Where We Are

- Everything I've shown so far works in C++ 1998.
- But there is still an issue with initializer lists for aggregate objects. Here is the C++ 1998 way to initialize a vector of 10 elements:

```
// The initial values:
int initial_primes[10] = { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 };

// Create the vector and copy the initial values into it.
vector<int> primes( 10 );
primes.insert( initial_primes, initial_primes + 10 );
```

- Gross!
    - (obviously)

C++ 2011 Initializer List Constructors

- In C++ 2011 this matter is fixed.
  - A special “initializer list” class is defined in the library.
  - Class designers can provide an “initializer list constructor” that takes an instance of the initializer list class.
  - When the compiler sees the programmer using an initializer list, it calls that constructor (if the initializer list constructor does not exist, it is an error).

```
vector<int> primes = { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 };
```

- Ahhh… much better.
- All C++ 2011 standard containers do this. *You can in your classes too*!

There’s More!

- Because of the previous rules, initializer list constructors can also do:

```
vector<int> special;

special = { 7, 42, 113 };

// The above is the same as:
vector<int> t__xyzzy = { 7, 42, 113 };
special = t__xyzzy;
```

And Still More!

- They can also do:

```
void do_something( const vector<int> &numbers );

do_something( { 7, 12, 113 } );

// The above is the same as:
vector<int> t__xyzzy = { 7, 42, 113 };
do_something( t__xyzzy );
```

The Uniform Syntax

- C++ 2011 also introduced the Uniform Initialization Syntax
  - Unifies all forms of initialization into a single syntax.
  - Applies stronger (safer) type conversion rules.
- The uniform syntax is signaled using *curly braces, but with no equal sign*.
  - It is not at all ambiguous with the older initialization syntaxes.
  - Personally, it took me some time to get used to the look!

The Uniform Syntax in Action

```
// Simple, scalar initialization (like in C).
int x{ 42 };

// Call constructor taking const char * parameter.
string name{ "Jill" };

// Call constructor taking two parameters.
string separator{ 64, '*' };

// Call the initializer list constructor.
vector<int> primes{ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 };
```

But There is This

- Ambiguities can still arise:

```
vector<int> numbers{ 10 };
```

- Is this…
  - … a call to the constructor that specifies the size of the vector or…
  - … an initializer list with a single initializer?
- Answer:
  - It is an initializer list with one initializer.
  - To get the other interpretation, use the function-like syntax: numbers(10).
  - The problem only arises because it's a vector of *integers*. A vector of strings doesn't have this ambiguity: strings{ 10 } can only be specifying the size.

Type Safety and the Uniform Syntax

- C-style initialization can be unsafe.

```
// Apostrophes as digit separators start with C++ 2014
long large_value = 10’000’000’000;  // Let’s assume 64-bit long integers.

// If plain integers are 32 bits, this initialization will fail.
int value = large_value; // Not an error, although a compiler warning is likely.
```

- C++ uniform syntax is safer.

```
int value{ large_value }; // Error! Programmer must explicitly cast.
```

- Everyone knows C’s rules are unsafe, but they can’t be changed without massive legacy code breakage.
  - *The uniform syntax is (was) entirely new, so new rules could be made for it.*