C++ Standard Library Type `initializer_list<>`

The material in this handout is collected from the following references:

- Section 6.2.6 of the text book C++ Primer.
- Various section of reference book The C++ Standard Library: A Tutorial and Reference 2nd Edition

Additional examples and explanations on C++ enumerations can be found at this page from Microsoft.

Introduction

To support the concept of initializer list for user-defined types, C++11 provides class template `std::initializer_list<>` which is presented in header file `<initializer_list>`. This library type represents an array of values of a specified type and can be used to support initializations by a sequence of values or in any other place where you want to process a sequence of values. One important application of `<initializer_list>`s is to write functions that take a varying number of arguments of the same type. For example:

```
// process a sequence of values of the same type
template <typename T>
void print_list(std::initializer_list<T> vals) {
  for (T const& x : vals) {
    std::cout << x << '\n';
  }
}

// pass a list of int values to print_list
print_list({1, 2, 3, 4});
// pass a list of double values to print_list
print_list({1.1, 2.2, 3.3, 4.4, 5.5});
// pass a list of char values to print_list
print_list({'h', 'e', 'l', 'l', 'o'});
```

Implementation Model

The general model to understand the semantics of a `{}`-list is this: if the `{}`-list is used to construct an `initializer_list` object, each list element is used to initialize an element of the underlying array of the `initializer_list`. Elements are typically copied from the `initializer_list` to wherever we use them. Consider:

```
std::vector<double> v{1, 2, 3.14};
```

The standard library `vector` has an initializer-list constructor, so the initializer-list is interpreted as a temporary constructed and used like this:

```
double const tmp_arr[] { double{1}, double{2}, 3.14 };
initializer_list<double> const tmp_lst(tmp_arr,
sizeof(tmp_arr)/sizeof(double));
std::vector<double> v(tmp_lst);
```

That is, the compiler constructs an array containing the initializer converted to the desired type which is `double`. This array is passed to `vector`'s initializer-list constructor as an `initializer_list`. The initializer-list constructor then copies the values from the array into its own data structure. Note that an `initialzier_list` is a small object of about size two words, so passing it by value makes sense.

`initializer_list` is Immutable

Note that the underlying array is immutable, so there is no way within the standard's rules that the meaning of a `{}`-list can change between two uses. This means a container taking elements from an `initializer_list` must use a copy operation, rather than a move operation. Consider:

```
void foo() {
  std::initializer_list<int> lst{1, 2, 3};
  std::cout << *lst.begin() << '\n'; // ok
  *lst.begin() = 2; // error!!! lst is immutable
}
```

Lifetime of `initialzier_list`

The life of a `{}`-list and its underlying array is determined by the scope in which it is used. When used to initialize a variable of type `initializer_list<T>`, the list lives as long as the variable. When used in an expression [including as an initializer to a variable of some other type, such as `vector<T>`], the list is destroyed at the end of its full expression.

Inside `<initializer_list>`

As explained previously, an object of type `initializer_list<E>` provides access to an array of objects of type `E`. In `<initializer_list>`, we find `initializer_list` declared as:

```
namespace std {
  template<class E> class initializer_list {
  public:
    typedef E value_type;
    typedef const E& reference;
    typedef const E& const_reference;
    typedef size_t size_type;
    typedef const E* iterator;
    typedef const E* const_iterator;
    initializer_list() noexcept;
    size_t size() const noexcept();  // number of elements
    const E* begin() const noexcept; // first element
    const E* end() const noexcept;   // one past the last element
  };

  // initializer-list range access
  template<class E> const E* begin(initializer_list<E> il) noexcept;
  template<class E> const E* end(initializer_list<E> il) noexcept;
```

```
}
```

Unfortunately, `initializer_list` doesn't offer a subscript operator. If you want to use `[]` rather than `*`, subscript a pointer:

```
void foo(std::initializer_list<int> lst) {
  for (size_t i{}; i < lst.size(); ++i) {
    std::cout << lst[i] << '\n'; // error
  }

  const int *p = lst.begin();
  for (size_t i{}; i < lst.size(); ++i) {
    std::cout << p[i] << '\n'; // ok
  }
}
```

As shown earlier, an `initializer_list` can also be used by a range-`for`. For example:

```
void foo2(std::initializer_list<int> lst) {
  for (int x : lst) std::cout << x << '\n';
}
```

`initializer_list` Constructors

A constructor that takes a single argument of type `std::initializer_list` is called an initializer-list constructor. An initializer-list constructor is used to construct objects using a `{}`-list as its initializer value. Standard library containers have initializer-list containers, assignments, etc. Consider:

```
std::vector<double> vd{1.0, 2.0, 3.456};
std::list<std::pair<std::string, std::string>> capitals {
  {"Kenya", "Nairobi"}, {"Taiwan", "Taipei"}, {"Vietnam", "Hanoi"}
};
std::map<std::vector<std::string>, std::vector<int>> years {
  {{"Maurice", "Vincent", "Wilkes"}, {1913, 1945, 1951, 1967, 2000}},
  {{"Martin", "Richards"}, {1982, 2003, 2007}},
  {{"David", "John", "Wheeler"}, {1927, 1947, 1951, 2004}}
};
```

The mechanism for accepting a `{}`-list is a function [often a constructor] taking an argument of type `std::initializer_list<T>`. For example:

```
void foo(std::initializer_list<int>);
foo({1, 2});
foo({1, 23, 345, 5678});
foo({}); // the empty list
foo{1, 2}; // error: missing function call operator ()!!!

years.insert({{"Bill", "Clinton"}, {1940, 1955, 1980}});
```

`initializer_list` Constructor Disambiguation

When you've several constructors for a class, the usual overload resolution rules are used to select the right one for a given set of arguments. For selecting a constructor, default and initializer-lists take precedence. Consider:

```
struct X {
  X(std::initializer_list<int>);
  X();
  X(int);
};

X x0{}; // empty list: default ctor or initializer-list ctor?
        // answer: default ctor
X x1{1}; // one integer: an int argument ctor or a list of one element ctor?
         // answer: the initializer-list ctor
```

The rules are:

- If either a default constructor or an initializer-list constructor could be invoked, *prefer the default constructor*. The reason for this is simple common sense: pick the simplest constructor possible. If you define an initializer-list constructor to do something with an empty list that differs from what the default constructor does, your class definition has a design error.
- If both an initializer-list constructor and an ordinary constructor could be invoked, *prefer the initializer-list constructor*. This rule is necessary to avoid different resolutions based on different numbers of elements. Consider `std::vector`:

  ```
  std::vector<int> v1{1};       // one element
  std::vector<int> v2{1, 2};    // two elements
  std::vector<int> v3{1, 2, 3}; // three elements

  std::vector<std::string> vs1{"one"};
  std::vector<std::string> vs2{"one", "two"};
  std::vector<std::string> vs3{"one", "two", "three"};
  ```

  In every case, the initializer-list constructor is used. If we really want to invoke the constructor taking one or two `int` arguments, we must use the `()` notation:

  ```
  std::vector<int> v1(1);    // one element with default value 0
  std::vector<int> v2(1, 2); // one element with value 2
  ```

  Here is another example:

```
struct S {
  S(int, int);
  S(std::initializer_list<int>);
};

S p(77, 5);      // calls S::S(int, int)
S q{77, 5};      // calls S::S(initializer_list<int>)
S r{77, 5, 22}; // calls S::S(initializer_list<int>)
S s = {77, 5};  // calls S::S(initializer_list<int>)
```

Without the initializer-list constructor, the constructor taking two `int`s would be called to initialize `q` and `s`, while the initialization of `r` would be invalid.

`explicit` keyword for Constructors

Because of initializer-lists, `explicit` now also becomes relevant for constructors taking more than one argument. So, you can now disable automatic type conversions from multiple values, which is also used when an initialization uses the `=` syntax:

```
struct S {
  S(int, int);
  explicit S(std::initializer_list<int>);
};

S x(77, 5);         // ok
S y{77, 5};         // ok
S z{77, 5, 22};     // ok
S v = {77, 5};      // ok - implicit type conversion is allowed
S w = {77, 5, 42}; // error - due to explicit keyword in ctor
```

Use of `initializer-list`s

Function `add_int` with a variable number of `int` parameter would have been implemented in C/C++98 like this:

```
#include <cstdarg>

int add_int(int how_many, ...) {
  va_list args;
  va_start(args, how_many);
  int sum{};
  for (int i{}; i < how_many; ++i) {
    sum += va_arg(args, int);
  }
  return sum;
}
```

A function with an `initializer_list<T>` parameter can access it as a sequence using member functions `size` or `begin` and `end`. Function `add_int` can now be implemented with more safety and generality using `initializer_list<T>`:

```
template <typename T>
T add(std::initializer_list<T> lst) {
  T sum {T()};
  for (size_t i{}; i != lst.size(); ++i) {
    // initializer_list doesn't provide subscript operator ...
    sum += lst.begin()[i]; // assuming operator+= is implemented for type T
  }
  return sum;
}
```

The subscript operator can be avoided by using iterators:

```
template <typename T>
T add(std::initializer_list<T> lst) {
  T sum {T()};
  for (T const* p = lst.begin(); p != lst.end(); ++p) {
    sum += *p; // assuming operator+= is implemented for type T
  }
  return sum;
}
```

Or, the function could also be implemented more simply using the range-`for` statement:

```
template <typename T>
T add(std::initializer_list<T> lst) {
  T sum {T()};
  for (T const& elem : lst) {
    sum += elem;
  }
  return sum;
}
```

Note that the elements of an `initializer_list` are immutable - don't even think about trying to modify their values. For example:

```
int foo(std::initializer_list<int> x, int val) {
  *x.begin() = val; // error: attempt to change the value of
                    // an initializer-list element
  return *x.begin(); // ok
}
```

Because `initializer_list` elements are immutable, we cannot apply a move constructor to them.

An `initializer_list<T>` is passed by value. That is required by the function overload resolution rules and doesn't impose an overhead because an `initializer_list<T>` object is just a small handle of about two words to specify a pointer to the array of `T`s and the array's size.