Some Elements of C++ 2011

Roland Levillain

EPITA Research and Development Laboratory (LRDE)

2012

Outline

Outline

Outline

Outline

Outline

Outline

Outline

Literal null pointer

- Null pointer in C++ 1998/2003 : 0 (not NULL)
- 0 is also the zero literal (constant).
- Overloading ambiguities:

```
void foo(short i);
void foo(float* p);

int main() {
  foo(0);  // foo(short) or foo(float*)?
}
```

- New C++ 2011 keyword: **nullptr**: null pointer literal.
- Ambiguity solved:

```
  foo(nullptr);
```

Simplifying Nested Templates

- In C++ 1998/2003, this code is not valid:

```
std::vector<std::pair<int, float>> v;
```

- The '>>' is seen as '**operator**>>'!
- Valid syntax:

```
std::vector< std::pair<int, float> > v;
```

- New interpretation in C++ 2011: '>>' are two closing angle brackets.

New type-related uses of `using`

- Defining a type alias (typedef) in C++ 1998/2003:

```
typedef std::list<std::string> stringlist;
typedef int (*intfun)(int);
```

- Alternative syntax in in C++ 2011:

```
using stringlist = std::list<std::string>;
using intfun = int (*)(int);
```

- **using** statements can be parameterized:

```
template<unsigned n, typename T>
class vec;

template<typename T>
using triplet = vec<3, T>;
```

Shortening Initializations

- Initializing containers is sometimes cumbersome:

```
std::list<int> values;
values.push_back (1);
values.push_back (2);
values.push_back (3);
// ...
```

- C++ 2011 extends the C array initialization syntax to any container:

```
std::list<int> values = { 1, 2, 3, 4, 5 };
```

or

```
std::list<int> values { 1, 2, 3, 4, 5 };
```

Constructing with an Initializer List

- These *initializer lists* are passed as

  ```
  std::initializer_list<T>
  ```

  objects.
- New category of constructor:

  ```
  container<T>::container (std::initializer_list<T> l)
  {
    insert (begin (), l.begin (), l.end ());
  }
  ```

Shortening and Generalizing Iterations on Containers

- Classic iterator-based traversal:

```
std::list<int> values = { 1, 2, 3, 4, 5 };
int sum = 0;
for (std::list<int>::const_iterator i = values.begin();
        i != values.end(); ++i)
  sum += *i;
```

- Foreach-like C++ 2011 syntactic sugar:

```
int sum = 0;
for (const int& i : values)
  sum += i;
```

Extensibility of Range-Based for Loops

- Also works with non class-based containers:

```
int values[5] = {1, 2, 3, 4, 5};
int sum = 0;
for (const int& i : values)
  sum += i;
```

- Mechanism based on *non-member* begin() and end() functions.
- Can be extended to any type T, by adding the following new functions, returning const (resp. mutable) iterators:
  - begin(**const** T&) (resp. begin(T&))
  - end(**const** T&) (resp. end(T&))

Automatic Type Deduction

- C++ can be verbose:

```
std::map<int, std::string>* dict =
  new std::map<int, std::string>;
very_long_type<int, float, double> v = foo (42);
```

- New C++ 2011 keyword for automatic type deduction : **auto** (actually, not really new).

```
auto dict = new std::map<float, std::string>;
auto v = foo (42);
```

- Can be **const**-qualified and used with ‘&’:

```
const auto& w = bar (51);
```

Changing the Default Behavior of the Compiler

- Making a type non-copyable and not directly constructible:

```
class Foo
{
private:
  Foo (const Foo&);                  // Only declared.
  Foo & operator= (const Foo&);      // Only declared.

  Foo();

public:
  static const Foo& instance ();
};


Foo::Foo () {}
const Foo& Foo::Foo& instance () { static Foo f; return f; }
```

Changing the Default Behavior of the Compiler

- In C++ 2011, with keywords **default** and **delete**:

```
class Foo
{
  Foo (const Foo&) = delete;
  Foo & operator= (const Foo&) = delete;

public:
  static const Foo& instance ();

private:
  Foo() = default;
};


const Foo& Foo::Foo& instance () { static Foo f; return f; }
```