STL: Function Objects

Miro Jurišić
meeroh@meeroh.org

Introduction

- Function objects aka. functors
- Objects which behave like functions
- `operator()`
- STL algorithms use functors
- More flexible than plain functions
- Can store additional data

Example: Functors in STL

```
vector <int>        v1;
vector <int>        v2;

transform (
    v1.begin (),
    v1.end (),
    v2.begin (),
    negate);
```

Standard Functors: Operators

- `plus`, `minus`, `multiplies`, `divides`, `modulus`, `negate` (unary)
- `equal_to`, `not_equal_to`, `greater`, `less`, `greater_equal`, `less_equal`
- `logical_and`, `logical_or`, `logical_not` (unary)

Standard Functors: Binders

- Binders convert 2-argument functors to 1-argument functors
- `bind1st` binds to the first argument
- `bind2st` binds to the second argument

```
find_if (
    v.begin (),
    v.end (),
    bind2nd (greater <int> (), 2));

find_if (
    v.begin (),
    v.end (),
    bind2nd (less_equal <int> (), 3));
```

Standard Functors: Adaptors

- Adaptors adapt real functions into functors
- `ptr_fun` converts a pointer to a function to a functor
- `mem_fun` and `mem_fun_ref` convert a pointer to a member function to a functor

```
class Object {
    public:
        Result function (Argument a);
};


vector <Object>         v1;
vector <Argument>       v2;
vector <Result>         v3;

transform (v1.begin (), v1.end (), v2.begin (), v3.begin (),
    mem_fun_ref (&Object::function));
```

Make Your Functors Adaptable

- A functor is adaptable if it works with STL adapters
- STL adapters require functors to have some typedefs
- `argument_type`, `first_argument_type`, `second_argument_type`, `result_type`
- Make your functors adaptable by inheriting from `std::unary_function` or `std::binary_function`
- Remove `const` and `&` from reference parameters

Example: Simple Functor

```
class StringLengthCompare:
    public std::unary_function <bool, string> {

    public:
        EqualToInt (int inCompareLength):
            mCompareLength (inCompareLength)
        {
        }

        bool operator () (const string& inCompare) const
        {
            return mCompareLength == inCompare.size ();
        }

    private:
        int mCompareLength;
};
```

```
StringLengthCompare     find3 (3);

find_if (container.begin(), container.end(), find3);
```

Make Your Functors Pure Functions

- STL makes little guarantees about copying your functors
- Using stateful functors is not correct and not portable
- Stateful functors often do not work as you expect
- Make your `operator() const`

Example: Bad (Stateful) Functor

```
class FindNth:
    public std::unary_function <bool, string> {

    public:
        EqualToInt (int inIndex):
            mIndex (inIndex),
            mCount (0)
        {
        }

        bool operator () (const string& inCompare)
        {
            if (mCount == mIndex) {
                mCount++;
                return true;
            } else {
                mCount++;
                return false;
            }
        }
```

```
    private:
        int mIndex;
        int mCount;
};



FindNth     find3rd (2);

container.erase (
    remove_if (
        container.begin(),
        container.end(),
        find3rd),
    container.end ());
```

Make Your Functors Suited For Pass-By-Value

- STL passes functors by value
- Design your functors with that in mind
- No expensive copy constructors and assignment operators
- No polymorphism
- Use a pointer wrapper in your functor if you need large or polymorphic objects