C++'s syntactic sugars

Dimitar Mirchev @DVMirchev

Definition

Syntactic sugar is syntax within a programming language that is designed to make things easier to read or to express.

C - operator[]

```
a[i] is syntactically equivalent to *(a + i)
```

C - operator[]

```c
int a[5];
a[2] = 1;
```

C - operator[]

```c
int a[5];
a[2] = 1;
3[a] = 5;
```

C++11 range-based for loop

for (*range_declaration* **:** *range_expression* **)**
    Loop_statement

is syntactically equivalent to

C++11 range-based for loop

```
for ( range_declaration : range_expression )
    Loop_statement
```

is syntactically equivalent to

```
{
  auto && __range = range_expression ;
  for (auto __begin = begin_expr, __end = end_expr;
        __begin != __end; ++__begin) {
    range_declaration = *__begin;
    loop_statement
  }
}
```

C++11 range-based for loop

- If *range_expression* is an expression of **built-in array** type
  - a. ***begin_expr*** is __range
  - b. ***end_expr*** is (__range + __bound)

- If *range_expression* is a class type C that has a member named **begin** and **end**
  - a. ***begin_expr*** is __range.begin()
  - b. ***end_expr*** is __range.end();

- Otherwise
  - a. ***begin_expr*** is begin(__range)
  - b. ***end_expr*** is end(__range)

C++11 range-based for loop

```
for ( range_declaration : range_expression )
      Loop_statement
```

is syntactically equivalent to

```
{
  auto && __range = range_expression ;
  for (auto __begin = begin_expr, __end = end_expr;
        __begin != __end; ++__begin) {
     range_declaration = *__begin;
     loop_statement
  }
}
```

C++17 range-based for loop

```
for ( range_declaration : range_expression )
        Loop_statement
```

`from C++17 is syntactically equivalent to`

```
{
  auto && __range = range_expression ;
  auto __begin = begin_expr ;
  auto __end = end_expr ;
  for ( ; __begin != __end; ++__begin) {
    range_declaration = *__begin;
    loop_statement
  }
}
```

C++17 structural binding

```
auto [x, y, z] = expression;
```

Case 1, built-in array:

```
auto __a = expression;

auto& x = __a[0]; // does not imply an actual reference
auto& y = __a[1];
auto& z = __a[2];
```

C++17 structural binding

```
auto [x, y, z] = expression;
```

Case 2, get<> for std::tuple and std::array:

```
auto __a = expression;

tuple_element<0, decltype(E)>::type& x = get<0>(__a);
tuple_element<1, decltype(E)>::type& y = get<1>(__a);
tuple_element<2, decltype(E)>::type& z = get<2>(__a);
```

C++17 structural binding

```
auto [x, y, z] = expression;
```

Case 3, public data for C-style structs and std::pair:

```
auto __a = expression;

auto& x = __a.mem1;
auto& y = __a.mem2;
auto& z = __a.mem3;
```

C++17 structural binding examples

```
tuple<T1, T2, T3>  f();
auto [x, y, z] = f(); // types are: T1, T2, T3


struct mystruct { int i; string s; double d; };
mystruct s = { 1, "xyzzy"s, 3.14 };
auto [x, y, z] = s; // types are: int, string,
double
```

C++17 structural binding examples

```
auto tuple = std::make_tuple(1, 'a', 2.3);
auto& [ i, c, d ] = tuple;


for (auto&& [first,second] : mymap) {
    // use first and second
}
```

C++17 If statement with initializer

```
if constexpr(optional) ( init-statement condition )
      statement-true
else
      statement-false
```

Is equivalent to

```
{
  init_statement
  if constexpr(optional) ( condition )
     statement-true
  else
     statement-false
}
```

C++17 If statement with initializer

```
{
    auto p = m.try_emplace(key, value);
    if (!p.second) {
        FATAL("Element already registered");
    } else {
        process(p.second);
    }
}
```

is equivalent to:

```
if (auto p = m.try_emplace(key, value); !p.second) {
    FATAL("Element already registered");
} else {
    process(p.second);
}
```

C++17 If statement with initializer

```
auto it = m.find(10);
if (it != m.end()) {
    return it->size();
} // "it" is leaked into the ambient scope.
```

```
if (auto it = m.find(10); it != m.end()) {
    return it->size();
} // "it" is destructed and undefined
```

C++17 If statement with initializer

```
if (std::lock_guard<std::mutex> lock(mx); shared_flag) {
    unsafe_ping();
    shared_flag = false;
}

 if (auto it = m.find(10); it != m.end()) {
    return it->size();
}

if (status_code c = bar(); c != SUCCESS) {
    return c;
}
```

C++17 switch statement with initializer

```cpp
switch (Foo x = make_foo(); x.status())
{
    case Foo::FINE: /* ... */
    case Foo::GOOD: /* ... */
    case Foo::NEAT: /* ... */
    default: /* ... */
}
```

C++ Lambdas

```
[](X& item){ item.DoTheJob(); }
std::for_each(par, items.begin(), items.end(), [](X& item){ item.DoTheJob(); });
```

Sort of translates into:

```
class _CompilerGeneratedNotReadable_
{
public:
    void operator() (X& item) const
    {
        item.DoTheJob();
    }
}
std::for_each(par, items.begin(), items.end(), _CompilerGeneratedNotReadable_{});
```

C - operator[]

```
[multiplier, &sum](X& item){ sum += item.Width() * multiplier; }
```

Sort of translates into:

```
class _CompilerGeneratedNotReadable_
{
public:
    _CompilerGeneratedNotReadable_(int& s, int m) : sum_[s}, multiplier_{m} {}
    void operator() (X& item) const
    {
        sum_ += item.Width() * multiplier_;
    }
private:
    int& sum_;
    int multiplier_;
}
```

Links

- Syntactic sugar
- P0184R0 Generalizing the Range-Based For Loop
- P0305R0 If statement with initializer
- P0217R2 Proposed wording for structured bindings
- Demystifying C++ lambdas
- C++17 If statement with initializer
- C++17 Structured Bindings
- How the new range-based for loop in C++17 helps Ranges TS?
- N4296 Working Draft, Standard for Programming Language C++
- What the ISO C++ committee added to the C++17 working draft at the Oulu 2016 meeting