Overloading the Member Access Operator

Sebastian Redl

Direct Member Access Operator

a.k.a. Dot Operator

foo.bar

Direct Member Access Operator

a.k.a. Dot Operator

foo.bar

- Part 1: Language Feature
- Part 2: Clang Implementation

Why?

Boost.Flyweight

Boost.Flyweight

```
<input type="text" name="f1">
<input type="text" name="f2">
<input type="radio" name="f3">
<input type="radio" name="f4">
<input type="submit">
```

Boost.Flyweight

```
<input type="text" name="f1">
<input type="text" name="f2">
<input type="radio" name="f3">
<input type="radio" name="f4">
<input type="submit">
```

Boost.Flyweight

```
<input type="text" name="f1">
<input type="text" name="f2">
<input type="radio" name="f3">
<input type="radio" name="f4">
<input type="submit">
```

Boost.Flyweight

```
struct element {
  string name;
  map<string, string> attributes;
};
```

Boost.Flyweight

```
struct element {
  string name;
  map<string, string> attributes;
};
```

Share memory of typically repeated parts:

```
struct element {
  flyweight<string> name;
  map<flyweight<string>,
      string> attributes;
};
```

Boost.Flyweight

```
if (e.name == "input")
```

Still works

Boost.Flyweight

```
if (e.name == "input")
```

Still works

```
e.name.size()
```

Doesn't work anymore

Boost.Flyweight

```
if (e.name == "input")
```

Still works

```
e.name.size()
```

Doesn't work anymore

```
e.name.get().size()
```

Have to do this

Boost.Flyweight

```
if (e.name == "input")
```

Still works

```
e.name.size()
```

Doesn’t work anymore

```
e.name.get().size()
```

Have to do this

Take the member access ...

... and push it to another object!

Boost.Bind

Boost.Lambda

Boost.Phoenix

Std.Bind

Binders & Lambdas

```
boost::bind(
  &vector<int>::push_back,
  _1, _2)
```

Binders & Lambdas

```
std::bind(
  static_cast<
    void (vector<int>::*)
           (const int&)
  >(&vector<int>::push_back),
  _1, _2)
```

Binders & Lambdas

```
std::bind(
  static_cast<
    void (vector<int>::*)
          (const int&)
  >(&vector<int>::push_back),
  _1, _2)
```

Binders & Lambdas

```
[](vector<int>& v, int i) {
    v.push_back(i);
}
```

Binders & Lambdas

```
[](vector<int>& v, int i) {
    v.push_back(i);
}
```

Binders & Lambdas

```
[](auto& c, auto i) {
    c.push_back(i);
}
```

Binders & Lambdas

```
bind(_1.push_back, _2)
```

Binders & Lambdas

```
bind(_1.push_back, _2)
```

Store member access and apply it later

JSON

JSON

```
{
  "answer": 42,
  "question": {
    "calculator": {
      "name": "Earth",
      "status”: "Harmless",
      "inhabitants": [
        {name: "Arthur Dent",
         origin: "Earth"}
      ]
    }
  }
}
```

JSON

```
// Javascript
let everything =
  JSON.parse(...);
alert(everything.question
  .calculator.inhabitants[0]
  .name);
```

JSON

```
// C++
auto everything =
  json::parse(...);
cout << everything["question"]
  ["calculator"]
  ["inhabitants"][0]["name"]
  << endl;
```

JSON

```
// C++?
auto everything =
  json::parse(...);
cout << everything.question
  .calculator.inhabitants[0]
  .name << endl;
```

JSON

```
// C++?
auto everything =
  json::parse(...);
cout << everything.question
  .calculator.inhabitants[0]
  .name << endl;
```

Use the name of the member at runtime

How?

The Old Way

Design & Evolution of C++

The Old Way

```
T operator ->();
```

Repeat the access on the result

```
x.foo -> x.operator->()->foo
```

The Old Way

```
T operator .();
```

Repeat the access on the result

```
x.foo -> x.operator.().foo
```

The Old Way

```
struct element {
  flyweight<string> name;
  map<flyweight<string>,
      string> attributes;
};

e.name.size()
```

The Old Way

```
template <typename T>
class flyweight {
public:
  const T& operator .();
};
```

The Old Way

```
std::bind(_1.push_back, _2)
```

The Old Way

```
template <int I>
class placeholder {
public:
  ??? operator .();
};
```

The Old Way

+ Simple
+ Easy to implement
— Very limited, no expression templates

My Experiment

Treat Name as an Argument

Attempt #1

```
MemberType operator .(??? Name);
```

Attempt #1

```
MemberType operator .(??? Name);
```

But we need the name at compile time!

Attempt #2

```
template <??? Name>
MemberType operator .();
```

Attempt #2

```
template <??? Name>
MemberType operator .();
```

How do we pass the name?

Identifier String

Identifier	String
+ Simple	— Complex

Identifier	String
+ Simple	— Complex
— Limited	+ Versatile

Identifier	String
+ Simple	— Complex
— Limited	+ Versatile
— New syntax	+ Existing syntax

Identifier	String
+ Simple	— Complex
— Limited	+ Versatile
— New syntax	+ Existing syntax
+ Operator names	— Canonicalization for operators

Attempt #2

```
template <__tstring Name>
MemberType operator .();
```

Can specialize, add SFINAE overloads, etc.

```
x.foo -> x.operator.<"foo">()
```

Attempt #2: JSON

```
class json {
public:
  template <__tstring Name>
  json operator .() {
    return (*this)[Name];
  }

  json operator [](const char* name)
  {
    return member_map[name];
  }
};
```

Attempt #2: Proxy

```
template <typename T>
class data_proxy {
  T t;
public:
  template <__tstring Name>
  auto operator .()
    -> ??? {
    return ???;
  }
};
```

Attempt #2: Proxy

Need a way to use __tstring to
access member in another object.

Attempt #2: Proxy

```
template <typename T>
class data_proxy {
  T t;
public:
  template <__tstring Name>
  auto operator .()
    -> decltype(t.*Name) {
    return t.*Name;
  }
};
```

.* with a __tstring means
"access member with this name"

Attempt #2: Proxy

```
struct Foo {
  int i;
  double d;
  std::string s;
};

data_proxy<Foo> fp;
fp.i = 3;
fp.d = 3.14;
fp.s = "Hello";
```

When?

When?

Old system: every lookup, implicit or explicit, is redirected.

When?

Old system: every lookup,
implicit or explicit, is redirected.

New system: only explicit member access!

When?

```
struct Y {
  template <__tstring Name>
  int operator .();
  int operator +(int);
  operator int();
};
struct X {
  int i;
  void func(Y y, Y* p);
  template <__tstring Name>
  int operator .();
};
```

When?

```
void X::func(Y y, Y* p) {
  y.abc;
  p->abc;
  y + 5;
  int j = y;
  this->i;
  i = 0;
}
```

When?

```
void X::func(Y y, Y* p) {
   y.abc;        // yes
   p->abc;
   y + 5;
   int j = y;
   this->i;
   i = 0;
}
```

When?

```
void X::func(Y y, Y* p) {
   y.abc;        // yes
   p->abc;       // yes, (*p).abc
   y + 5;
   int j = y;
   this->i;
   i = 0;
}
```

When?

Possible interpretations:

::operator +(y, 5) (built-in)

y.operator +(5)

but the latter would be

y.operator .<"operator+">()(5)

Every operator overload would exist.

When?

```
void X::func(Y y, Y* p) {
   y.abc;      // yes
   p->abc;     // yes, (*p).abc
   y + 5;      // no
   int j = y;
   this->i;
   i = 0;
}
```

When?

Interpretation:

```
y.operator int()
```

becomes

```
y.operator.("operator int")()
```

Every conversion would exist!

When?

```
void X::func(Y y, Y* p) {
   y.abc;      // yes
   p->abc;     // yes, (*p).abc
   y + 5;      // no
   int j = y; // please no
   this->i;
   i = 0;
}
```

When?

```
void X::func(Y y, Y* p) {
   y.abc;      // yes
   p->abc;     // yes, (*p).abc
   y + 5;      // no
   int j = y;  // please no
   this->i;    // yes
   i = 0;
}
```

When?

Unqualified lookup order:

1. Block scopes outward to function
2. Class scopes up the hierarchy
3. Namespace scopes outward to global

When?

```
int global;
void X::fn() {
  global = 0;
}
```

When?

```
int global;
void X::fn() {
  global = 0;
}
```

Class scope lookup finds operator .

```
int global;
void X::fn() {
  this->operator .<"global">() = 0;
}
```

When?

```
int global;
void X::fn() {
  global = 0;
}
```

Have to use explicit qualification

```
int global;
void X::fn() {
  ::global = 0;
}
```

When?

```
void X::fn() {
  int local;
  local = 0; // oops
}
```

When?

```
void X::fn() {
  int local;
  local = 0; // oops
}
```

Class scope lookup finds operator .

```
void X::fn() {
  int local;
  this->operator .<"local">() = 0;
}
```

When?

```
void X::func(Y y, Y* p) {
    y.abc;       // yes
    p->abc;      // yes, (*p).abc
    y + 5;       // no
    int j = y;  // please no
    this->i;     // yes
    i = 0;       // definitely not
}
```

Strings

Strings

```
template <__tstring S>
void fn() {
  std::cout << S << '\n';
}


fn<"one">();
```

Strings

```
template <__tstring S>
void fn() {
  std::cout << S << '\n';
}


fn<"one">();
```

All the usual template stuff works!

Strings

```
template <>
void fn<"one">() {
  std::cout << "the one\n";
}


fn<"one">();
```

Specialization

Strings

```
template <__tstring S>
void fn2(some_struct<S> s) {
  std::cout << S << '\n';
}


fn2(some_struct<"one">());
```

Deduction

Strings

- Only allowed as template parameter
- Argument can be other __tstring or literal
- Instantiated by replacing with literal
- Overload resolution gives trouble

Strings

```
template <unsigned N>
void print(char (&s)[N]);

template <__tstring S>
void fn() {
  print(S);
}
```

Type of S is not dependent, compiler wants to resolve overload at template definition time. But what is N?

Strings

__tstring needs an implicit conversion to a type that depends on the actual value of the object.

This is something new.

Strings

__tstring needs an implicit conversion to a type that depends on the actual value of the object.

This is something new.

My hacky solution: make conversion explicit.

Strings

```
template <unsigned N>
void print(char (&s)[N]);

template <__tstring S>
void fn() {
  print(S.c_str);
}
```

Type of the pseudo-member expression S.c_str is dependent.
S.c_str instantiates to string literal too.

Operator Names

Operator Names

```
struct S {
  template <__tstring Name>
  int operator .();
};

s.operator *();
```

What is passed to operator .?
"operator *" or "operator*"?

Operator Names

No current solution, compiler would probably crash.
This is a big argument in favor of a name type.

Escaping

Escaping

```
struct json {
  template <__tstring Name>
  operator .();

  json operator [](int i);

  optional<string> as_string();
};
```

How to access as_string?

Escaping

Option 1: Escaped Names

```
j..to_string();
(&j)->to_string();
this->to_string(); // private
to_string();       // private
j.*(&json::to_string)();
__escape(j).to_string();
```

Escaping prevents invoking dot operator.

Escaping

Option 2: Nodot Pattern

```
struct json_nodot {
  optional<string> as_string();
};

struct json : json_nodot {
  typedef json_nodot nodot_type;

  template <__tstring Name>
  json operator .();
};
```

Escaping

Option 2: Nodot Pattern (ctd.)

```
template <typename T>
typename T::nodot_type&
  nodot(T& t) { return t; }

json j;
nodot(j).to_string();
```

Built-in	Nodot
+ Automatic	+ User choice

Built-in	Nodot
+ Automatic	+ User choice
+ No code	— Boilerplate

Built-in	Nodot
+ Automatic	+ User choice
+ No code	— Boilerplate
— New syntax	+ Library solution

Built-in	Nodot
+ Automatic	+ User choice
+ No code	— Boilerplate
— New syntax	+ Library solution
— Hard to find intuitive syntax	— Requires inheritance

Escaping

Lazyness wins.

I have not implemented escaping.

Use Case Patterns

- Proxy
- Expression Template
- Fake Members

Proxy

- Boost.Flyweight
- shared_ref (shared_ptr with ref syntax)
- Boost.value_initialized
- copy_on_write
- locked_ref (holds mutex lock)
- many more ...

Proxy

Lots of boilerplate.
Overload dot operator.
Overload all other operators.

Proxy

Build a library that contains the boilerplate.
Specify actual behavior with a policy.

Simple Proxy

```
struct policy_archetype {
  typedef some wrapped_type;

  // const/non-const as needed
  const wrapped_type&
    access() const;
};
```

Flyweight Proxy

```
template <typename T>
class flyweight_policy {
  const T* shared;
public:
  typedef T wrapped_type;

  template <typename... Args>
  flyweight_policy(Args&&... args)
    : shared(factory().get(
      std::forward<Args>(args)...))
  {}
```

Flyweight Proxy (ctd.)

```
  const T& access() const {
    return *shared;
  }
};

template <typename T>
using flyweight =
  proxy<flyweight_policy<T>>;
```

Simple Proxy Implementation

```
template <typename Policy>
class proxy {
  using T = Policy::wrapped_type;
  Policy policy;

public:
  template <typename... Args>
  proxy(Args&&... args)
    : policy(
        std::forward<Args>(args)...)
  {}
```

Simple Proxy Implementation (ctd.)

```
  template <__tstring Member>
  auto operator .()
    -> decltype(do_access<Member>(
                     policy.access()))
  {
    auto& ref = policy.access();
    return do_access<Member>(ref);
  }
  // const overload, same code
  // other operator overloads
};
```

Member Function Dilemma

What happened to ref.*Member?
Member functions happened!

Member Function Dilemma

Remember this?

```
template <typename T>
class data_proxy {
  T t;
public:
  template <__tstring Name>
  auto operator .()
    -> decltype(t.*Name) {
    return t.*Name;
  }
};
```

Member Function Dilemma

What happens when I do this?

```
struct some {
  void foo();
};

data_proxy<some> sp;
sp.foo();
```

Member Function Dilemma

Instantiates to this:

```
class data_proxy<some> {
  some t;
public:
  auto operator .<"foo">()
    -> decltype(t.foo) {
    return t.foo;
  }
};
```

Error: member function must be called.

Simple Proxy Implementation (ctd.)

```
template <__tstring Member,
          typename T>
auto do_access(T& ref)
    -> decltype((ref.*Member)) {
  return ref.*Member;
}
```

Overload removed by SFINAE
for member functions.

Simple Proxy Implementation (ctd.)

```
template <__tstring Member,
          typename T>
auto do_access(T& ref)
    -> simple_call_proxy<T&,Member>
{ return ref; }
```

Simple Proxy Implementation (ctd.)

```
template <__tstring Member,
          typename T>
auto do_access(T& ref)
    -> typename enable_if_c<
__is_bound_function(ref.*Member),
simple_call_proxy<T&,Member>
        >::type
{ return ref; }
```

__is_bound_function(expr) is true
if expr is access to member function

Simple Proxy Implementation (ctd.)

```
template <typename Ref,
          __tstring Member>
class simple_call_proxy {
  Ref ref;
public:
  simple_call_proxy(Ref ref)
    : ref(ref) {}

  template <typename... Args> auto
  operator ()(Args&&... args) const
  {
    return (ref.*Member)(
      FORWARD(args)...);
  }
};
```

Simple Proxy Implementation

```
template <typename Ref, __tstring Member>
class simple_call_proxy {
  Ref ref;
public:
  simple_call_proxy(Ref ref) : ref(ref) {}

  template <typename... Args> auto
  operator ()(Args&&... args) const
    -> decltype((ref.*Member)(std::forward<Args>(args)...))
  {
    return (ref.*Member)(std::forward<Args>(args)...);
  }
};

template <__tstring Member, typename T>
auto do_access(T& ref) -> decltype((ref.*Member)) {
  return ref.*Member;
}
template <__tstring Member, typename T>
auto do_access(T& ref)
    -> typename enable_if_c<__is_bound_function(ref.*Member),
                            simple_call_proxy<T&,Member>>::type
{ return ref; }

template <typename Policy>
class proxy {
  using T = Policy::wrapped_type;
  Policy policy;

public:
  template <typename... Args>
  proxy(Args&&... args)
    : policy(std::forward<Args>(args)...)
  {}

  template <__tstring Member>
  auto operator .()
    -> decltype(do_access<Member>(policy.access()))
  {
    auto& ref = policy.access();
    return do_access<Member>(ref);
  }
  template <__tstring Member>
  auto operator .() const
    -> decltype(do_access<Member>(policy.access()))
  {
    auto& ref = policy.access();
    return do_access<Member>(ref);
  }
};
```

Not shown: other operator overloads.

Proxy Complexity Level

- Only notify on access

Proxy Complexity Level

- Only notify on access
- Pass member name to access function

Proxy Complexity Level

- Only notify on access
- Pass member name to access function
- Notify when access begins and ends (temporaries get destroyed)

Proxy Complexity Level

- Only notify on access
- Pass member name to access function
- Notify when access begins and ends (temporaries get destroyed)
- Detailed notifications, member call begins (supply arguments), ends (supply return value) or throws (supply exception)

Expression Template

- Lambdas
- Regex Named Captures (Boost.Xpressive)
- Named Parameters (Boost.Parameter)
- New DSL opportunities ...

Lambdas

```
bind(_1.push_back, _2)
```

Lambda Implementation

```
template <unsigned N>
struct placeholder {
  template <__tstring Member>
  access_expression<placeholder, Member>
  operator .() const {
    return {*this};
  }

  template <typename... Args> auto
  operator ()(Args&&... args) const {
    return select<N>(FORWARD(args)...);
  }
};
```

Lambda Implementation (ctd.)

```
template <typename Base,
          __tstring Member>
class access_expression {
  Base b;

public:
  access_expression(const Base& b)
    : b(b) {}

  template <typename... Args> auto
  operator ()(Args&&... args) const {
    return do_access<Member>(
        b(FORWARD(args)...));
  }
};
```

Lambdas

Sufficient for Data Member Access Lambda

```
struct data {
  int i;
};

vector<data> lots = get_data();
vector<int> ints(lots.size());
std::transform(lots.begin(), lots.end(),
               ints.begin(), _1.i);
```

Proper bind needs a lot of boilerplate

Lambdas

Ideally, extend Boost.Proto to support this.

Aside

Debugging forwarding functions is hard!

Debugging Forwarding Functions

```
void f(int) {}

struct st {};

void test() {
  st s;
  f(s);
}
```

Debugging Forwarding Functions

```
   f(s);

dbgfwd.cpp:13:3: error:
   no matching function for call to 'f'
note: candidate function not viable:
   no known conversion from 'st'
   to 'int' for 1st argument
void f(int) {}
```

Debugging Forwarding Functions

```
void f(int) {}

template <typename T>
auto bla(T t) -> decltype(f(t)) {
  return f(t);
}

struct st {};

void test() {
  st s;
  bla(s);
}
```

Debugging Forwarding Functions

```
  bla(s);

dbgfwd.cpp:15:3: error:
  no matching function for call to 'bla'
note: candidate template ignored:
  substitution failure [with T = st]:
  no matching function for call to 'f'
auto bla(T t) -> decltype(f(t))
```

But why? Which functions does it try?

Debugging Forwarding Functions

- Isolate failing use of forwarding functions
- Substitute return type with what I expect
- Look at the new error from template body

Debugging Forwarding Functions

```
void f(int) {}

template <typename T>
auto bla(T t) -> void
                  /*decltype(f(t))*/
{
  return f(t);
}

struct st {};

void test() {
  st s;
  bla(s);
}
```

Debugging Forwarding Functions

```
  bla(s);

dbgfwd.cpp:13:3: error:
  no matching function for call to 'f'
note: in instantiation of function
  template specialization 'bla<st>'
note: candidate function not viable:
  no known conversion from 'st'
  to 'int' for 1st argument
void f(int) {}
```

Debugging Forwarding Functions

I don’t have a better solution.

Someone please help me! ;-)

Boost.Parameter

Currently uses tag structs

```
namespace tag { struct index; }
boost::parameter::keyword<tag::index>&
  _index = ...;

template <typename ArgumentPack>
int print_index(
    ArgumentPack const& args) {
  std::cout << args[_index];
}

print_index(_index = 1);
```

Boost.Parameter

Could do this instead:

```
template <typename ArgumentPack>
int print_index(
    ArgumentPack const& args) {
  std::cout << args.index;
}

print_index(p.index = 1);
```

Boost.Parameter

Key idea:

```
template <typename Tag> class keyword;
->
template <__tstring Tag> class keyword;

args.index -> args[keyword<"index">]

p.index -> keyword<"index">
```

Fake Members

- JSON and other data formats
- Language Interop (Boost.Python)
- Properties without Overhead
- Tuple with Named Members
- Use your imagination!

Named Tuple Members

```
using value_type = tuple<
  n<"key", int>,
  n<"value", string>>;

value_type v;
v.key = 0;
v.value = "zero";
cout << get<0>(v) << ' ' << get<1>(v);
```

Named Tuple Members

```
template <int I> struct int_ {};
template <__tstring S>
struct string_ {};

template <int MyIndex, typename... Es>
class tuple_base;

template <int MyIndex>
class tuple_base<MyIndex> {
protected:
  void do_get() {}
};
```

Named Tuple Members

```
template <int MyIndex, __tstring Name,
          typename T, typename... Es>
class tuple_base<MyIndex, n<Name, T>,
                 Es...>
  : public tuple_base<MyIndex + 1,
                      Es...>
{
public:
  template <typename A, typename... As>
  tuple_base(A&& a, As&&... as)
    : base(FORWARD(as)...),
      data_(FORWARD(a))
  {}
```

Named Tuple Members

```
protected:
  using base::do_get;
  T& do_get(int_<MyIndex>)
    { return data_; }
  T& do_get(string_<Name>)
    { return data_; }

private:
  T data_;
};
```

Named Tuple Members

```
template <typename... Es>
class tuple_nodot :
    private tuple_base<0, Es...> {
  using base::do_get;

  template <int I>
  auto get() {
    return do_get(int_<I>());
  }

  template <__tstring N>
  auto nget() {
    return do_get(string_<N>());
  }
};
```

Named Tuple Members

```
template <typename... Es>
class tuple : public tuple_nodot<Es...>
{
public:
  using nodot_type = tuple_nodot<Es...>;

  template <__tstring Query>
  auto operator .() const
    return nodot(*this)
      .template nget<Query>();
  }
};
```

Named Tuple Members

```
template <int N, typename... Es>
auto get(tuple<Es...>& t) {
  return nodot(t).template get<N>();
}

template <__tstring Query,
          typename... Es>
auto nget(tuple<Es...>& t) {
  return nodot(t)
    .template nget<Query>();
}
```

Fake Members

Use Cases Vary Widely, Little Commonality

Why Not?

Why Not?

- You can do really, really weird stuff with it
- Very hard to use - heavy metaprogramming
- Limited to members that can be represented by strings and returned by a function - no member templates or types

Member Templates

How to translate this?

```
s.fn<int>();
```

Must know whether s.fn is a template
to disambiguate grammar.

Lessons Learned

Lessons Learned

- You learn a lot from implementing a feature
- You learn even more from using it
- Lack of uniformity hurts metaprogramming
  - Unutterable types (member closure)
  - Overload sets
  - Template names

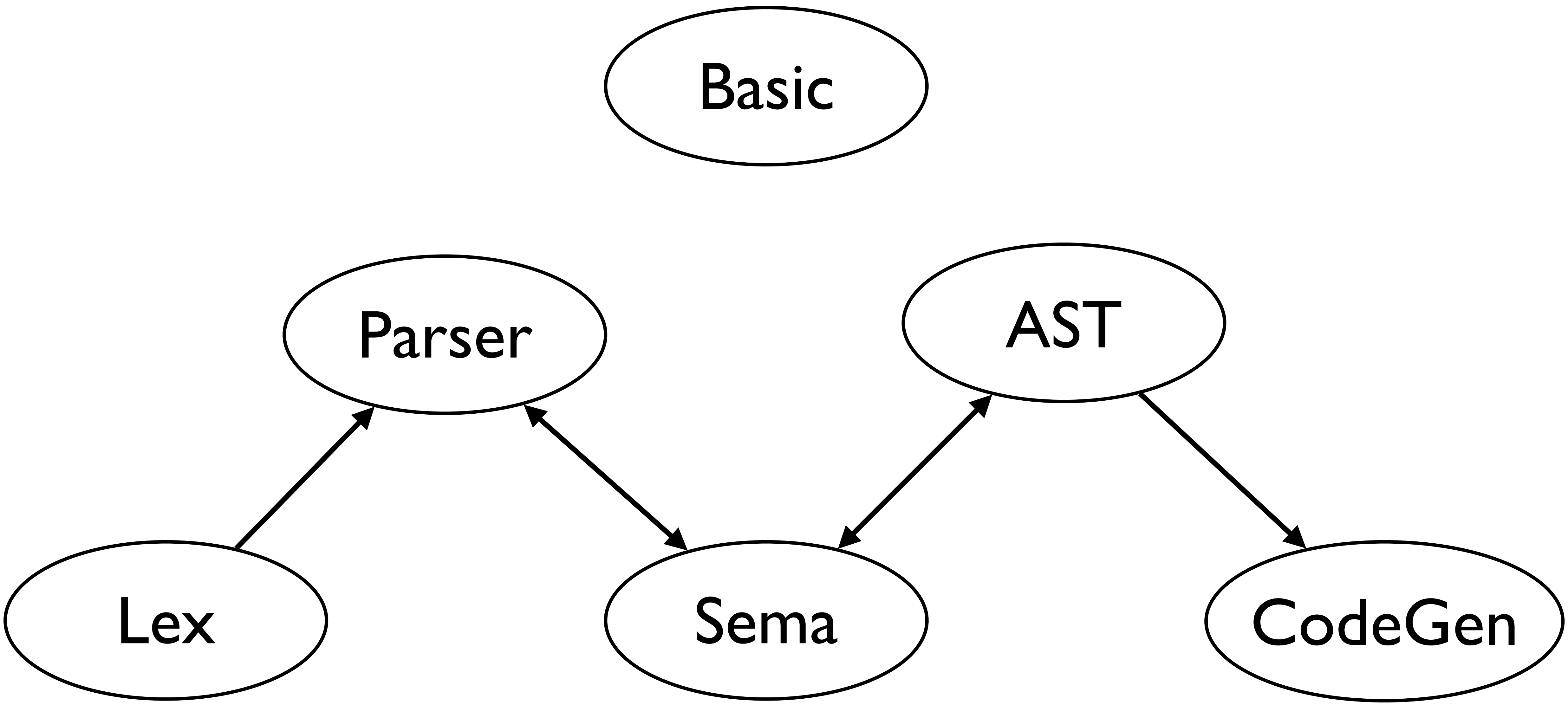
Questions?

Next: Implementation

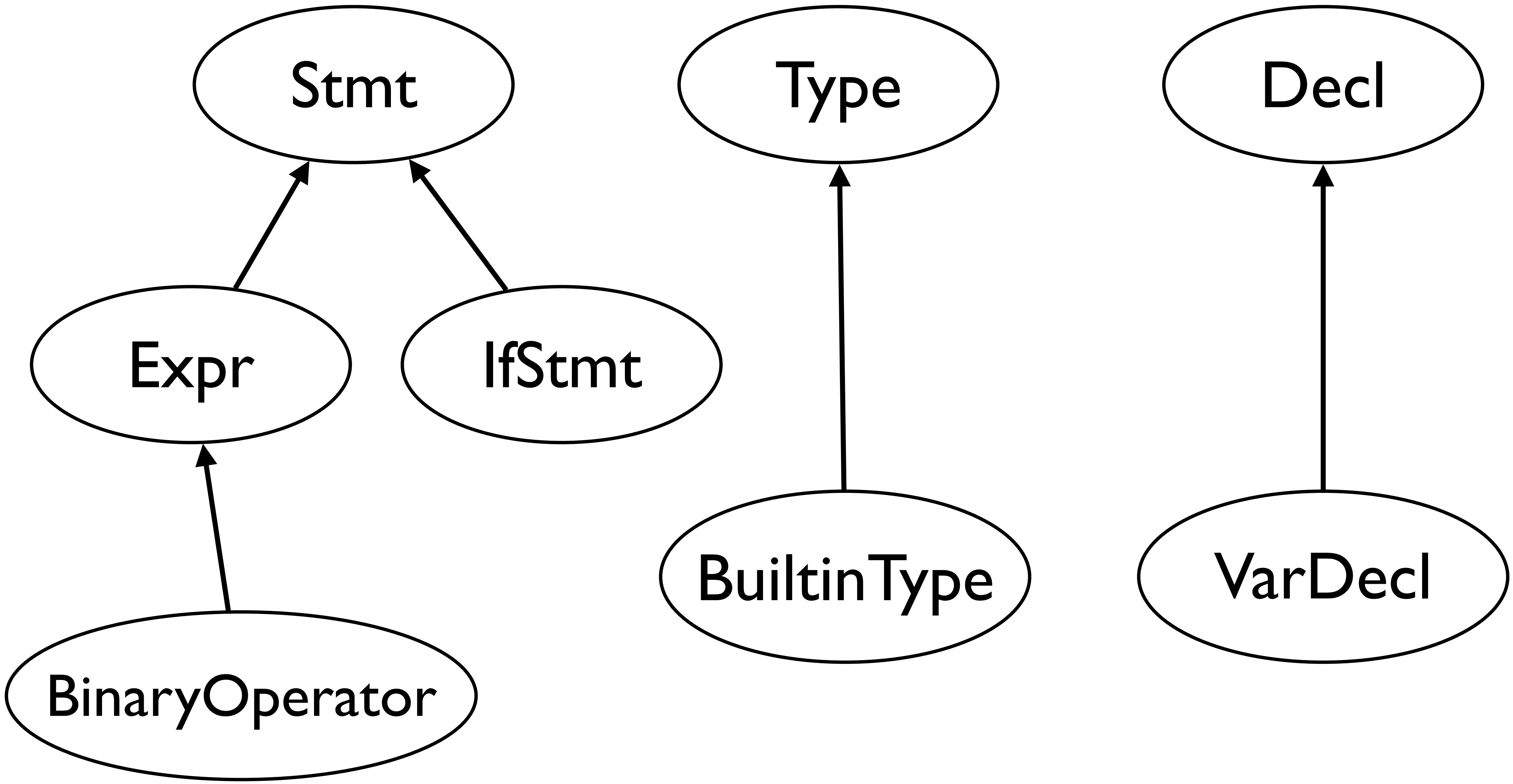
Part 2: Implementation

Here Be Clang Code

Clang Architecture



Clang AST Architecture



Implementing __tstring

- Recognize __tstring as keyword
- Parse __tstring as a type
- Allow __tstring as a template argument
- Pseudo-member c_str

Recognize __tstring as keyword

include/clang/Basic/TokenKinds.def:

```
KEYWORD(__tstring            , KEYCXX11)
```

Parse __tstring as a type

- Add parser structure representation
- Recognize __tstring when parsing declarations
- Add AST representation (BuiltinType)
- Turn parser structure into AST

Adding a Type

- Template instantiation
- Name mangling
- Serialization (PCH & Modules)

Allow __tstring as a template argument

- Allow as a non-type argument type
- Allow strings as arguments
- Instantiate __tstring parameters
- Implement specialization lookup
- Implement argument deduction

Pseudo-member c_str

- Create PseudoMemberExpr for AST
  - Template instantiation
  - Serialization
  - ...
- Handle member lookup into __tstring

Implementing the Dot Operator

- Make operator . overloadable
- Inject operator . into name lookup
- Extend operator .* to handle strings
- Implement __is_bound_function

Make operator . overloadable

include/clang/Basic/OperatorKinds.def:

```
OVERLOADED_OPERATOR(
  Period, // Enumeration Name
  ".",    // Spelling (for printing)
  period, // Parser Token
  true,   // Unary Operator
  false,  // Binary Operator
  true)   // Member-only
```

Make operator . overloadable

Semantic Validation of Operator Function

```
bool Sema::
  CheckOverloadedOperatorDeclaration
    (FunctionDecl *FnDecl) {
  // ...
  if (Op == OO_Period) {
    // Operator . must be a template
    // with one __tstring parameter.
    // I haven't actually implemented this.
  }
}
```

Inject operator . into Name Lookup

Sema::ActOnMemberAccessExpr

```
if (FunctionTemplateDecl *periodOperator =
        findPeriodOperator(...)) {
  return callPeriodOperator(...);
} else {
  // Normal Lookup
}
```

~100 lines of code in the helper functions

Extend operator .* to handle strings

Sema::CreateBuiltinBinOp

```
switch (Opc) {
// ...
case BO_PtrMemD:
case BO_PtrMemI: {
  if (rhsType->isTStringType() ||
      isCharArray(rhsType))
    return accessByStaticString(...);
  // normal pointer-to-member access
}
// ...
}
```

Extend operator .* to handle strings

accessByStaticString

```
StringLiteral *lit =
    cast<StringLiteral>(rhs);
IdentifierInfo *memberII =
    &ctx.Idents.get(lit->getString());
return ActOnMemberAccessExpr(memberII,...);
```

Full implementation is ~40 lines of code

Implement __is_bound_function

include/clang/Basic/TokenKinds.def

```
KEYWORD(__is_bound_function, KEYCXX11)
```

include/clang/Basic/ExpressionTraits.h

```
enum ExpressionTrait {
  // ...
  ET_IsBoundFunction
};
```

Implement __is_bound_function

Parser::ParseCastExpression

```
switch (Tok.getKind()) {
  // lots and lots of cases
case tok::kw___is_bound_function:
  return ParseExpressionTrait();
```

ExpressionTraitFromTokKind

```
case tok::kw___is_bound_function:
    return ET_IsBoundFunction;
```

Implement __is_bound_function

EvaluateExpressionTrait

```
case ET_IsBoundFunction:
  return E->getType() == ctx.BoundMemberTy;
```

Sema::BuildExpressionTrait

Prevent error from not immediately calling member function.

Questions?

Thank your for your attention!