Allocation

```
new NumExpr(7)
```

```
... delete ...?
```

Detecting Leaks

```
$ leaks -atExit -- ./msdscript --interp
1+2
3

Process:         msdscript [97927]
....
leaks Report Version: 4.0
Process 97927: 222 nodes malloced for 20 KB
Process 97927: 7 leaks for 128 total leaked bytes.

    7 (128 bytes) << TOTAL >>

      3 (64 bytes) ROOT LEAK: <AddExpr 0x1267040a0> [32]
         1 (16 bytes) <NumExpr 0x126704080> [16]
         1 (16 bytes) <NumExpr 0x126704090> [16]

      1 (16 bytes) ROOT LEAK: <NumExpr 0x1267040f0> [16]
      1 (16 bytes) ROOT LEAK: <NumVal 0x1267040c0> [16]
      1 (16 bytes) ROOT LEAK: <NumVal 0x1267040d0> [16]
      1 (16 bytes) ROOT LEAK: <NumVal 0x1267040e0> [16]
```

Avoid Allocation?

```
{
    NumExpr seven(7);

    seven.print(std::cout);

    // automatically deleted by leaving block
}
```

Avoid Allocation?

```
{
    NumExpr seven(7);
    VarExpr x("x");
    AddExpr plus(&seven, &x);

    plus.print(std::cout);

    // all deleted by leaving block
}
```

Avoid Allocation?

```
{
    NumExpr seven(7), five(5);
    VarExpr x("x");
    AddExpr plus(&seven, &x);

    plus.subst("x", &five)->print(std::cout);

    // result of Add::subst still leaks
 }
```

Avoid Allocation?

Try to never use *****

```
class AddExpr{
   Expr lhs;
   Expr rhs;
   AddExpr(Expr lhs, Expr rhs);
   Expr subst(std::string name, Expr repla);
   Val interp();
};
```

Avoid Allocation?

Try to never use *****

```
class AddExpr{
  Expr lhs;
  Expr rhs;
  AddExpr(Expr lhs, Expr rhs);
  Expr subst(std::string name, Expr repla);
  Val interp();
};
```

Reserves a fixed amount of space

Avoid Allocation?

Try to never use *

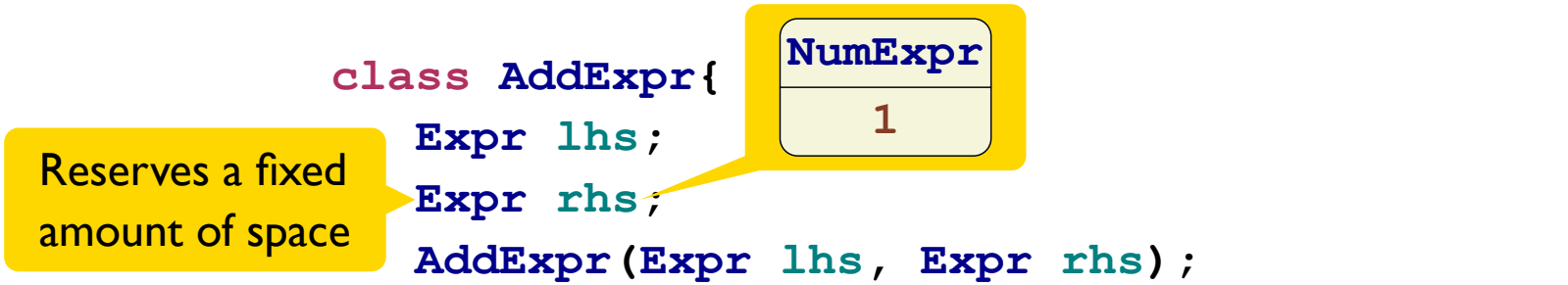
```
class AddExpr{
  Expr lhs;
  Expr rhs;
  AddExpr(Expr lhs, Expr rhs);
  Expr subst(std::string name, Expr repla);
  Val interp();
};
```

1

Reserves a fixed amount of space

Avoid Allocation?

Try to never use *




```
class AddExpr{
  Expr lhs;
  Expr rhs;
  AddExpr(Expr lhs, Expr rhs);
  Expr subst(std::string name, Expr repla);
  Val interp();
};
```

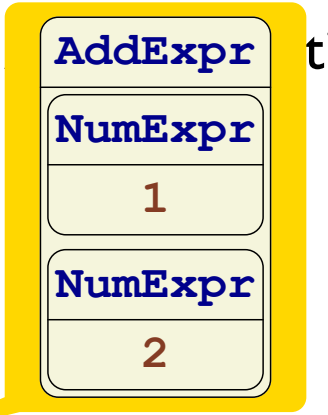
Avoid Allocation?

Try to never use *

Reserves a fixed amount of space

```
class AddExpr{
  Expr lhs;
  Expr rhs;
  AddExpr(Expr lhs, Expr rhs);
  Expr subst(std::string name, Expr repla);
  Val interp();
};
```

tion?



Try to never use *

Reserves a fixed amount of space

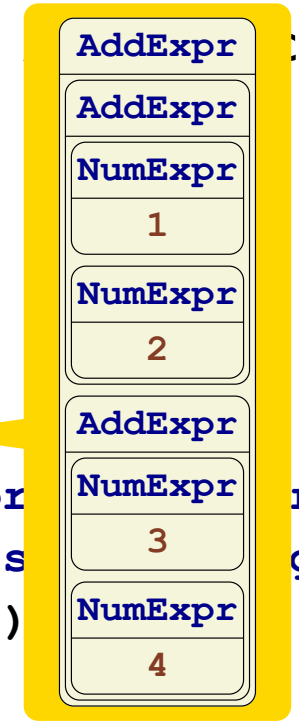
```
class AddExpr{
  Expr lhs;
  Expr rhs;
  AddExpr(Expr lhs, Expr rhs);
  Expr subst(std::string name, Expr repla);
  Val interp();
};
```

ation?

Try to never use *

```
class AddExpr{
  Expr lhs;
  Expr rhs;
  AddExpr(Expr                r rhs);
  Expr subst(s                g name, Expr repla);
  Val interp()
};
```

Reserves a fixed amount of space



You can't make general trees without pointers

Explicit Deallocation

```
CHECK( (new AddExpr(new VarExpr("x"), new NumExpr(3)))
        ->subst("x", new NumVal(3))
        ->equals(AddExpr(new NumExpr(3), new NumExpr(3)) );
```

```
Expr *x_e = new VarExpr("x");
Expr *three_e = new NumExpr(3);
Expr *add_x_e = new AddExpr(x_e, three_e);

Expr *result_e = add_x_e->subst("x", three_e);

Expr *add_three_e = new AddExpr(three_e, three_e);
CHECK( result_e->equals(add_three_e) );

delete add_three_e;
delete result_e;
delete add_x_e;
delete three_e;
delete x_e;
```

Explicit Deallocation

```
CHECK( (new AddExpr(new VarExpr("x"), new NumExpr(3)))
        ->subst("x", new NumVal(3))
        ->equals(AddExpr(new NumExpr(3), new NumExpr(3)) );
```

```
Expr *x_e = new VarExpr("x");
Expr *three_e = new NumExpr(3);
Expr *add_x_e = new AddExpr(x_e, three_e);

Expr *result_e = add_x_e->subst("x", three_e);

Expr *add_three_e = new AddExpr(three_e, three_e);
CHECK( result_e->equals(add_three_e) );

delete add_three_e;
delete result_e;
delete add_x_e;
delete three_e;
delete x_e;
```

What if there are new subexpressions?

Deallocating Trees

```
Expr *ten_e = new NumExpr(10);
Expr *five_e = new NumExpr(5);
Expr *add_e = new AddExpr(ten_e, five_e);

....

delete add_e;
delete five_e;
delete ten_e;
```

Deallocating Trees

```
Expr *ten_e = new NumExpr(10);
Expr *five_e = new NumExpr(5);
Expr *add_e = new AddExpr(ten_e, five_e);

....

delete add_e;
delete five_e;
delete ten_e;
```

Make **`AddExpr`** responsible for subexpressions?

Deallocating Trees

```
AddExpr::~AddExpr()
{
  delete lhs;
  delete rhs;
}
```

```
Expr *add_e = new AddExpr(new NumExpr(10),
                          new NumExpr(5));

....

delete add_e;
```

Deallocating Trees

```
AddExpr::~AddExpr()
{
  delete lhs;
  delete rhs;
}
```

```
Expr *add_e = new AddExpr(new VarExpr("x"),
                          new NumExpr(5));
Expr *three_e = new NumExpr(3);
Expr *result_e = add_e->subst("x", three_e);
....
delete result_e;
delete add_e;
delete three_e;
```

Deallocating Trees

```
AddExpr::~AddExpr()
{
  delete lhs;
  delete rhs;
}
```

```
Expr *add_e = new AddExpr(new VarExpr("x"),
                          new NumExpr(5));
Expr *three_e = new NumExpr(3);
Expr *result_e = add_e->subst("x", three_e);
....
delete result_e;
delete add_e;
delete three_e;
```

deletes `NumExpr(5)` again?

No Sharing Ever?

```
Expr *NumExpr::subst(std::string var, Val *new_val)
{
  return NumExpr(rep); // instead of `this`
}
```

```
Expr *FunExpr::subst(std::string var, Val *new_val)
{
  if (var == formal_arg)
     return this;
  else
     return FunExpr(var, body->subst(var, new_val));
}
```

No Sharing Ever?

```
Expr *NumExpr::subst(std::string var, Val *new_val)
{
  return NumExpr(rep); // instead of `this`
}
```

```
Expr *FunExpr::subst(std::string var, Val *new_val)
{
  if (var == formal_arg)
     return new FunExpr(var, body);
  else
     return FunExpr(var, body->subst(var, new_val));
}
```

No Sharing Ever?

```
Expr *NumExpr::subst(std::string var, Val *new_val)
{
  return NumExpr(rep); // instead of `this`
}
```

```
Expr *FunExpr::subst(std::string var, Val *new_val)
{
  if (var == formal_arg)
     return new FunExpr(var, body->clone());
  else
     return FunExpr(var, body->subst(var, new_val));
}
```

Workable, but tedious and arguably inefficient

Allowing Sharing: Reference Counting

Idea: keep track of the number of references to an object

```
class Expr {
  int refcount;
  void ref() { refcount++; }
  void unref()
  {
     refcount--;
     if (refcount == 0) delete this;
  }
  ....
};
```

Allowing Sharing: Reference Counting

Idea: keep track of the number of references to an object

```
AddExpr::AddExpr(Expr *lhs, Expr *rhs)
{
  lhs->ref();
  rhs->ref();
  this->lhs = lhs;
  this->rhs = rhs;
}

AddExpr::~AddExpr()
{
  lhs->unref();
  rhs->unref();
}
```

Allowing Sharing: Reference Counting

Idea: keep track of the number of references to an object

```
Expr *add_e = new AddExpr(new VarExpr("x"),
                          new NumExpr(5));
add_e->ref();

Expr *three_e = new NumExpr(3);
three_e->ref();

Expr *result_e = add_e->subst("x", three_e);
result_e->ref();
....
result_e->unref();
three_e->unref();
add_e->unref();
```

Allowing Sharing: Reference Counting

Idea: keep track of the number of references to an object

```
Expr *add_e = new AddExpr(new VarExpr("x"),
                          new NumExpr(5));
add_e->ref();

Expr *three_e = new NumExpr(3);
three_e->ref();

Expr *result_e = add_e->subst("x", three_e);
result_e->ref();
....
result_e->unref();
three_e->unref();
add_e->unref();
```

Workable,
probably more efficient,
definitely more tedious

Resource Acquisition is Initialization (RAII)

Idea: Make a wrapper object that refs and unrefs

```
class ExprBox {
   Expr *expr;
   ExprBox(Expr *expr) {
     expr->ref();
     this->expr = expr;
   }
   ~ExprBox() {
     expr->unref();
   }
}
```

Resource Acquisition is Initialization (RAII)

Idea: Make a wrapper object that refs and unrefs

```
{
  ExprBox add_e(new AddExpr(new VarExpr("x"),
                            new NumExpr(5)));
  ExprBox three_e(new NumExpr(3));

  ....

  // add_e and three_e are deleted here
}
```

Resource Acquisition is Initialization (RAII)

Idea: Make a wrapper object that refs and unrefs

```
{
  ExprBox add_e(new AddExpr(new VarExpr("x"),
                            new NumExpr(5)));
  ExprBox three_e(new NumExpr(3));

  ....
  ExprBox res_e(add_e.expr->subst("x", three_e.expr));
  ....

  // add_e, three_e, and res_e are deleted here
}
```

Wrappers as Fields

```
class AddExpr : public Expr {
  ExprBox lhs;
  ExprBox rhs;
  ....
};
```

```
AddExpr::AddExpr(Expr *lhs, Expr *rhs)
{
  this->lhs.set(lhs);
  this->rhs.set(rhs);
}

AddExpr::~AddExpr() { }
```

Wrappers as Fields

```
class ExprBox {
   Expr *expr;
   ExprBox() { expr = NULL; }
   ExprBox(Expr *expr) { set(expr); }
   void set(Expr *expr) {
     expr->ref();
     this->expr = expr;
   }
   ~ExprBox() {
     if (expr != NULL)
       expr->unref();
   }
}
```

Reference Counting via Explicit RAII

Remaining problems:

- Must create **`ExprBox`**, **`ValBox`**, ...
- Must add **`refcount`**, **`ref`**, and **`unref`** to all objects
- Must use **`.expr`** to access **`Expr`** in **`ExprBox`**
- Must use **`.set`** to assign **`Expr`** in **`ExprBox`**
- Not-yet-boxed objects are in a dangerous "floating" state

Smarter use of C++ can address theses problems

Predefined C++ Support for Reference Counting

`std::shared_ptr<`*T*`>`

Type of a box that holds *T**, can be used like *T**

Use as type instead of *T**

Predefined C++ Support for Reference Counting

`std::shared_ptr<`*T*`>` `std::shared_ptr<Expr>` is like `ExprBox`

Type of a box that holds *T**, can be used like *T**

Use as type instead of *T**

`std::make_shared<`*T*`>(`*arg*`, ... `*arg*`)`

Creates a box that holds a new *T* with the given *arg*s

Use as expression instead of `new` *T*`(`*arg*`, ... `*arg*`)`

Objects of type *T* **do not** need a `refcount` field, because it's stored in the `std::shared_ptr`

Using Shared Pointers

```
{
  std::shared_ptr<Expr> add_e
    = std::make_shared<AddExpr>(std::make_shared<VarExpr>("x"),
                                std::make_shared<NumExpr>(5)));
  std::shared_ptr<Expr> three_e = std::make_shared<NumExpr>(3);

  ....
  add_e->subst("x", three_e); // unused result deleted
  ....
  // add_e and three_e are deleted here
}
```

Using Shared Pointers

```
class AddExpr : public Expr {
  std::shared_ptr<Expr> lhs;
  std::shared_ptr<Expr> rhs;
  ....
};
```

```
AddExpr::AddExpr(std::shared_ptr<Expr> lhs,
                 std::shared_ptr<Expr> rhs)
{
  this->lhs = lhs;
  this->rhs = rhs;
}
```

Casting Shared Pointers

`std::dynamic_pointer_cast<`*T*`>(e)`

Casts to a shared *T**

Use as expression instead of `dynamic_cast<`*T**`>(e)`

Don't use the `get` method of `std::shared_ptr`, because that loses the reference count

Casting Shared Pointers

```
bool NumExpr::equals(std::shared_ptr<Expr> other_expr) {
  std::shared_ptr<NumExpr> other_num_expr
    = std::dynamic_pointer_cast<NumExpr>(other_expr);
  if (other_num_expr == NULL)
    return false;
  else
    return rep == other_num_expr->rep;
}
```

Returning This Object

```
std::shared_ptr<Expr> NumExpr::subst(std::string name,
                                     std::shared_ptr<Expr> repla) {
  return this; // does not work
}
```

Option 1: don't return `this`

```
std::shared_ptr<Expr> NumExpr::subst(....) {
  return std::make_shared<NumExpr>(rep);
}
```

Returning This Object

```
std::shared_ptr<Expr> NumExpr::subst(std::string name,
                                     std::shared_ptr<Expr> repla) {
  return this; // does not work
}
```

Option 2: use `std::enable_shared_from_this<Expr>` and `shared_from_this()`

```
class Expr : public std::enable_shared_from_this<Expr> {
  ....
};
```

```
std::shared_ptr<Expr> NumExpr::subst(....) {
  return shared_from_this();
}
```

Refactoring Summary

old	*new*
`T *`	`std::shared_ptr<T>`
`new T(arg, ...)`	`std::make_shared<T>(arg, ...)`
`dynamic_cast<T*>(arg)`	`std::dynamic_pointer_cast<T>(arg)`
`class T { .... };`	`class T : public std::enable_shared_from_this<T> { .... };`
`this` when not followed by `->`	`shared_from_this()`

Refactoring Summary

old	*new*
`Expr *`	`std::shared_ptr<Expr>`
`new Expr(arg, ...)`	`std::make_shared<Expr>(arg, ...)`
`dynamic_cast<Expr*>(arg)`	`std::dynamic_pointer_cast<Expr>(arg)`
`class Expr { .... };`	`class Expr : public std::enable_shared_from_this<Expr> { .... };`
`this`	`shared_from_this()`

when not followed by `->`

Conversion via Macros

pointer.h

```
#define USE_PLAIN_POINTERS 1
#if USE_PLAIN_POINTERS

# define NEW(T)     new T
# define PTR(T)     T*
# define CAST(T)    dynamic_cast<T*>
# define CLASS(T)   class T
# define THIS       this

#else

# define NEW(T)     std::make_shared<T>
# define PTR(T)     std::shared_ptr<T>
# define CAST(T)    std::dynamic_pointer_cast<T>
# define CLASS(T)   class T : public std::enable_shared_from_this<T>
# define THIS       shared_from_this()

#endif
```

Conversion via Macros

pointer.h

```
#define USE_PLAIN_POINTERS 1
#if USE_PLAIN_POINTERS

# define NEW(T)     new T
# define PTR(T)     T*
# define CAST(T)    dynamic_cast<T*>
# define CLASS(T)   class T
# define THIS       this

#else

# define NEW(T)     std::make_shared<T>
# define PTR(T)     std::shared_ptr<T>
# define CAST(T)    std::dynamic_pointer_cast<T>
# define CLASS(T)   class T : public std::enable_shared_from_this<T>
# define THIS       shared_from_this()

#endif
```

1 ⇒ leak

0 ⇒ no leak

Conversion via Macros

pointer.h

```
#define USE_PLAIN_POINTERS 1
#if USE_PLAIN_POINTERS

# define NEW(T)     new T
# define PTR(T)     T*
# define CAST(T)    dynamic_cast<T*>
# define CLASS(T)   class T
# define THIS       this

#else

# define NEW(T)     std::make_shared<T>
# define PTR(T)     std::shared_ptr<T>
# define CAST(T)    std::dynamic_pointer_cast<T>
# define CLASS(T)   class T : public std::enable_shared_from_this<T>
# define THIS       shared_from_this()

#endif
```

Using 1 lets you refactor part of your program at a time, then swicth to 0

Macro Refactoring Summary

old	*new*
`T *`	`PTR(T)`
`new T(arg, ...)`	`NEW(T)(arg, ...)`
`dynamic_cast<T>(arg)`	`CAST(T)(arg)`
`class T { .... };`	`CLASS(T) { .... };`
`this`	`THIS`

when not followed by `->`

Conversion via Macros

```
{
  PTR(Expr) add_e = NEW(AddExpr)(NEW(VarExpr)("x"),
                                 NEW(NumExpr)(5));
  PTR(Expr) three_e = NEW(NumExpr)(3);

  ....
  add_e->subst("x", three_e);
  ....
}
```

Conversion via Macros

```
class AddExpr : public Expr {
  PTR(Expr) lhs;
  PTR(Expr) rhs;
  ....
};
```

```
AddExpr::AddExpr(PTR(Expr) lhs, PTR(Expr) rhs)
{
  this->lhs = lhs;
  this->rhs = rhs;
}
```

Conversion via Macros

```
bool NumExpr::equals(PTR(Expr) other_expr) {
  PTR(NumExpr) other_num_expr = CAST(NumExpr)(other_expr);
  if (other_num_expr == NULL)
    return false;
  else
    return rep == other_num_expr->rep;
}
```

Conversion via Macros

Advantages:

- Less verbose
- Helps incremental conversion
- Supports future changes

Disadvantage:

- Language becomes a variant of C++