# CS 6015: Software Engineering

Spring 2024

Lecture 21: Memory Leaks – Smart Pointers

# Last Week

- Undefined behavior
- Profiling

# This Week

- Smart pointers
- Environments

# Memory leaks

- How to know if there are memory leaks in any program?
- When memory leaks happen?
- Any garbage collector in the language being used?

```
new NumExpr(4)

new NumExpr(5)

new AddExpr(new NumExpr(4), new NumExpr(5))
```

- Are we using delete? Destructors?

# Approaches to avoid memory leaks

- Create objects on stack and not the heap

```
{
  NumExpr Obj(7);
  Obj.print(std::cout);
  // automatically deleted by leaving block
}
```

- Another example that still works:

```
{
  NumExpr Obj(7);
  VarExpr x("x");
  AddExpr Obj_add(&Obj, &x);
  Obj_add.print(std::cout);
  // all deleted by leaving block
}
```

# Approaches to avoid memory leaks

- Create objects on stack and not the heap
- What about this one?

```
{
  NumExpr Obj1(7), Obj2(5); ;
  VarExpr x("x");
  AddExpr Obj_plus(&Obj1, &x);
  Obj_plus.subst("x", &Obj2)->print(std::cout);
   // still have new
  // result of Add::subst still leaks
}
```

# Approaches to avoid memory leaks

- Recreate the methods to not use pointers *

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

Reserves fixed amount of space

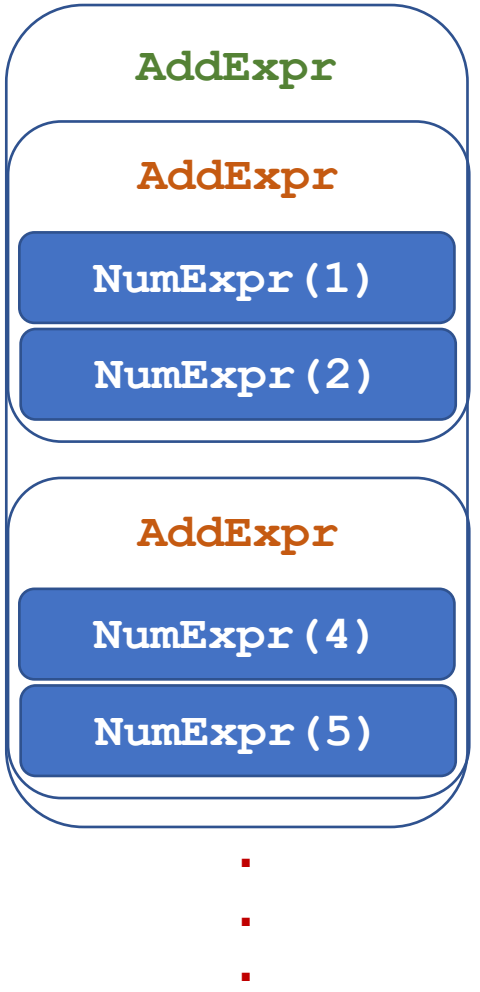
1

NumExpr(1)

# Approaches to avoid memory leaks

- Recreate the methods to not use pointers *

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




**You can't make general trees without pointers**

# Approaches to avoid memory leaks

- Allocate memory by **new** keyword and deallocate memory by **delete** keyword

```
void assign_pointer()
{
    int* p = new int(4);

    // function body

    //delete once done
    delete (p);
}
```

**Explicit deallocation**

- Call a destructor

# Approaches to avoid memory leaks

- Allocate memory by **new** keyword and deallocate memory by **delete** keyword
- How to handle that in our MSDscript?

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

**Do NOT use this direct new form anymore?**
**Replace each by a variable!?!**

# Approaches to avoid memory leaks

- Allocate memory by **`new`** keyword and deallocate memory by **`delete`** keyword
- How to handle that in our MSDscript?

```
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?

# Approaches to avoid memory leaks

- Allocate memory by **`new`** keyword and deallocate memory by **`delete`** keywod
- How to handle that in our MSDscript?

```
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 its sub expressions

**Implicit / indirect or tree deallocation**

# Approaches to avoid memory leaks

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

**Implicit / indirect or tree deallocation**

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

**Or**

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

# Approaches to avoid memory leaks

```
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 three_e

deletes three_e again

**Good but tricky**

**Prevent Sharing?**

# Approaches to avoid memory leaks

- Prevent sharing? Do not return `this`?

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

Expr *FunExpr::subst(std::string var, Val *new_val) {
        . . .
          return this;
        . . .
}
```

# Approaches to avoid memory leaks

- Prevent sharing? Do not return `this`?

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

Expr *FunExpr::subst(std::string var, Val *new_val) {
      . . .
         return new FunExpr(var, body);
      . . .
}
```

# Approaches to avoid memory leaks

- Prevent sharing? Do not return `this`?

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

Expr *FunExpr::subst(std::string var, Val *new_val) {
       . . .
          return new FunExpr(var, body ->clone());
        . . .

}
```

**Works but tedious and inefficient**

# Approaches to avoid memory leaks

- Allow sharing, but 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;
}

....
};
```

**Drawback**

- We need to call **`refs`** and **`unrefs`** before and after each allocation.
- Lacks automation

# Approaches to avoid memory leaks

- Resource Acquisition is Initialization (RAII)
  - Make a wrapper object that `refs` and `unrefs`
  - Little bit better than previous approaches

**Drawback**

- Create new classes for the new
- Create new methods
- Remember to wrap all objects

**Smarter and automatic approach**

# Wrapping up

Some Issues while trying to handle Memory Leaks :

- Dangling Pointers: When the object is de-allocated from memory without modifying the value of the pointer.
- Buffer Overflow: When a pointer is used to write data to a memory address that is outside of the allocated memory block.
- Undefined behavior: When trying to delete freed pointers
- Introduces overhead while tending to keep a reference to the shared pointers

Solution?

# Smart pointers

- Is a Wrapper class over a pointer
- Deallocate and free destroyed object memory
- Types:

  **`auto_ptr`** (*deprecated as of C++11)*

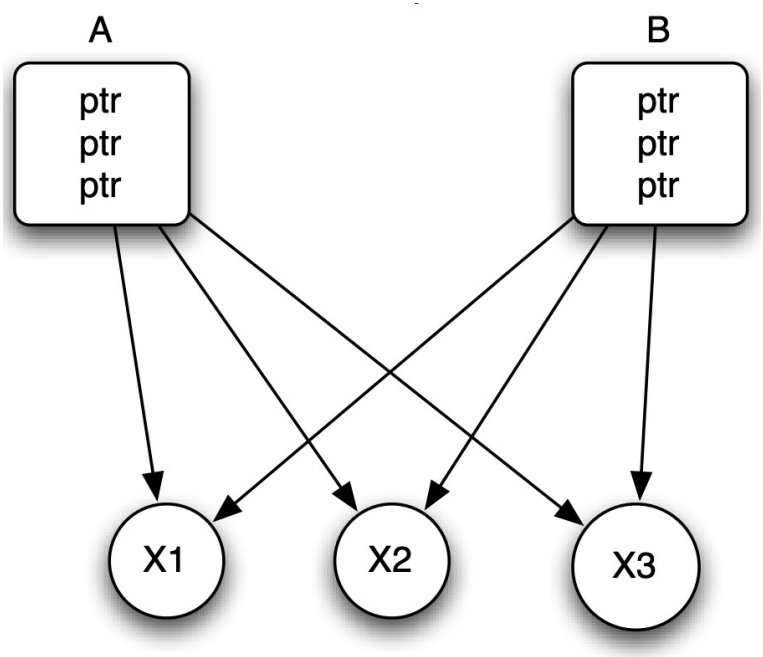
  **`unique_ptr`** (similar to **`auto_ptr`**, but with improved security)

  **`shared_ptr`** (our candidate for MSDScript)

  **`weak_ptr`** (similar to **`shared_ptr`**, but without a reference counter)

# Smart pointers: shared_ptr

- Maintains a **Reference Counter** using the ***use_count()*** **method**




- **How to use it?** `std::shared_ptr <T>`

# Smart pointers: shared_ptr

`std::shared_ptr<T>`


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)`

# MSDScript: using shared_ptr

```
{
    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
}
```

# MSDScript: using shared_ptr

```
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;
}
```

# MSDScript: using shared_ptr

`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

# MSDScript: using shared_ptr

```
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 == nullptr)
       return false;

    else
       return rep == other_num_expr->rep;
}
```

# MSDScript: using shared_ptr

```
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(std::string name,
                                     std::shared_ptr<Expr> repla)
{
        return std::make_shared<NumExpr>(rep);
}
```

# MSDScript: using shared_ptr

```
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();
}
```

# Using shared_ptr: 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` | `shared_from_this()` |

Not followed by ->

# Using shared_ptr: 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()` |

Not followed by ->

# Using shared_ptr: 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 -> Normal pointers
0 -> Smart pointers

# Using shared_ptr: 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` |

`this` — Not followed by `->`

# Using shared_ptr: Refactoring Summary

```
{
    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);

    ....
}
```

# Using shared_ptr: Refactoring Summary

```
class AddExpr : public Expr {

    PTR(Expr) lhs;
    PTR(Expr) rhs;

    ....

 };
```

```
AddExpr::AddExpr(PTR(Expr) lhs, PTR(Expr) rhs) {

    this->lhs = lhs;
    this->rhs = rhs;

}
```

# Using shared_ptr: Refactoring Summary

```
bool NumExpr::equals(PTR(Expr) other_expr) {

    PTR(NumExpr) other_num_expr = CAST(NumExpr)(other_expr);

    if (other_num_expr == nullptr)
        return false;
    else
        return rep == other_num_expr->rep;

}
```