Generic handles

Memory management made easier

A cautionary note

- The programming technique that we are about to see is pretty specific to C++, because it relies on
  - destructors
  - templates
- However, the way we will develop the program is applicable to any language

The problem

- Remember the `Expr` classes?
  - Version 1
    - the user does memory management
    - leaks memory, never really satisfactory
  - Version 2
    - memory management in the implementation
    - somewhat intertwined with the rest of the code
- We are going to try to do better

Do better? How?

- Better correspondence between the code and the concepts it expresses
- More general
- Easier to follow once you understand it

Our first try (lecture 11)

- Our first try was a user-visible class hierarchy:

```
class Expr { /* … */ };
class IntExpr: public Expr { /* … */ };
class UnaryExpr: public Expr { /* … */ };
class BinaryExpr: public Expr { /* … */ };
```

- Advantage: Straightforward
- Disadvantage: Exposes memory management to users

Using the first try

```
IntExpr* three = new IntExpr(3);
IntExpr* four = new IntExpr(4);
IntExpr* five = new IntExpr(5);
UnaryExpr* negfive =
    new UnaryExpr(“-”, five);
BinaryExpr* twelve =
    new BinaryExpr(“*”, three, four);
BinaryExpr* seven =
    new BinaryExpr(“+”, negfive, twelve);

seven->print(cout);
```

Memory management woes

- There is no good place to delete
  - Sometimes, the user has to delete

```
Expr* e1 = new IntExpr(8);
Expr* e2 = new BinaryExpr("*", e1, e1);
```

  - But sometimes, it's impossible

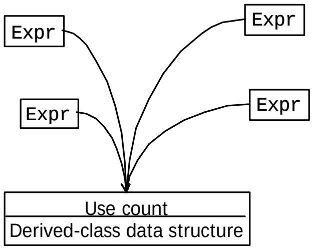
```
Expr* e =
    new BinaryExpr("*",
                   new IntExpr(3),
                   new IntExpr(4));
```

- Therefore, we need a better scheme

The second version (lecture 13)

- We renamed our base class `ExprBase`
- We added a use count to the `ExprBase` class
- We defined a use-counted handle class called `Expr`
  - An `Expr` object contains a pointer to `ExprBase`
  - The `Expr` class does memory management

The revised data structure



Outline of class hierarchy

```
class Expr { /* … */ };
class ExprBase { /* … */ };
class IntExpr: public ExprBase
    { /* … */ };
class UnaryExpr: public ExprBase
    { /* … */ };
class BinaryExpr: public ExprBase
    { /* … */ };
```

This approach is easier to use

```
Expr e(“*”,
       Expr(“-”, Expr(3)),
       Expr(“+”, Expr(4), Expr(5)));
e.print(cout);
```

However, there are still disadvantages

- A single class implements the user interface for the `Expr` hierarchy and use-counted memory management
- Each `Expr` object contains data related to the expression contents and to memory management

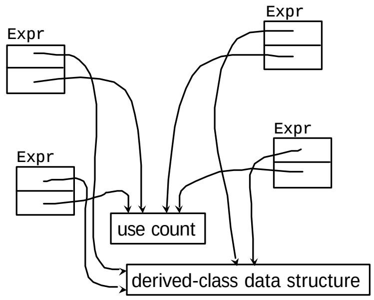
The source of the problems

- Class `Expr` is really a kind of container
  - Each `Expr` object contains a single expression node
- But it is an *intrusive* container
  - The bookkeeping information is intertwined with the data in the container element
- If we're going to keep the code separate, we'll want separate data, too

A new strategy

- Keep the use count separate from the expression node
  - Advantage: We can ignore what's in the expression nodes when we do memory management, and vice versa
  - Disadvantage: Probably slightly slower
- Put all the memory management in a separate class

The data structure



Let's think about it generically

- We have an inheritance hierarchy
- We want a handle class whose objects will
  - each identify an object from that hierarchy
  - manage memory for its object
  - not know the details of that object's type
- In effect, we want a generic handle

What properties should it have?

- The usual construct, copy, assign, and destroy operations
- A way of constructing a handle from an object of the target class
- A way of getting at the object to which the handle is attached

These handles act a lot like pointers

- They are sometimes called "counted pointers" or "smart pointers"
- We can use the `operator->` feature of C++ to make them look a lot like pointers
- It is hard to defend against deliberate misuse

`operator->` explained

- If `p` is a pointer, then `p->x` is defined as equivalent to `(*p).x`
- If p is not a pointer, then `p->x` is defined as `(p.operator->())->x`
- Note that this definition is recursive: `operator->` can return a class object as long as it is of a type with `operator->` defined

We can already start coding

```
template<class T> class Handle {
public:
  Handle();
  Handle(T*);
  Handle(const Handle&);
  Handle& operator=(const Handle&);
  ~Handle();
  T& operator*() const;
  T* operator->() const;
private:
  T* p;
  int* use;
};
```

We will want to cater to null handles

- If someone says
  `Handle<T> h;`
  we want to allow it, even though `h` doesn’t refer to anything useful (yet).
- We would like to avoid special cases in our use-counting code
- Therefore, *every* handle will have a use count, even if its pointer is 0

The default constructor

```
template<class T> Handle<T>::Handle():
  use(new int(1)), p(0) { }
```

Other constructors

- When we attach a handle to an object, we will be giving the handle the responsibility for deleting that object eventually:

```
{
  Handle<string>
    h(new string(“hello”));
  // …
}
```

Constructor definitions

```
template<class T>
Handle<T>::Handle(T* tp):
  use(new int(1)), p(tp) { }
template<class T>
Handle<T>::Handle(const Handle<T>& h):
  use(h.use), p(h.p) { ++*use; }
```

Destructor

```
template<class T>
Handle<T>::~Handle() {
  if (--*use == 0) {
    delete use;
    delete p;
  }
}
```

Assignment

- As usual, we will increment the use count on the right-hand side before we decrement our own:

```
template<class T> Handle<T>&
Handle<T>::operator=(const Handle<T>& h) {
  ++*h.use;
  if (--*use == 0) {
    delete use; delete p;
  }
  use = h.use; p = h.p;
  return *this;
}
```

The `*` and `->` operators

```
template<class T>
T& Handle<T>::operator*() const {
  return *p;
}
template<class T>
T* Handle<T>::operator->() const {
  return p;
}
```

How do we use it?

- It works a lot like a pointer, but it will delete objects for us:

```
// Pointer version
int* p(new int(42));
cout << *p << endl;
delete p;
// Handle version
Handle<int> h(new int(42));
cout << *h << endl;
// No delete
```

Interactions with inheritance

- Handles encapsulate pointers, which means that they can point to a base class in an inheritance hierarchy:

```
class B { virtual ~B(); /* … */ };
class D: public B { /* … */ };
Handle<B> h(new D);
```

Handles as data elements

- Because handles have copy and assignment defined, we can use them as elements of other data structures, almost as easily as if they were pointers

```
struct node {
  Handle<Thing> h;
  Handle<OtherThing> h2;
  // …
};
```

What do these handles do for us?

- They are an abstraction of the idea of use-counted memory allocation
- They behave a lot like pointers
- They allow us to structure our programs to separate the algorithmic part from the memory-management part
- We have to write the `Handle` template only once

Advantages of use counting

- We can manage resources other than memory
  - files
  - network connections
- Resources are deallocated as soon as they are no longer needed
  - no unused memory sitting around and waiting for the garbage collector

What don't they do for us?

- Use-counted memory allocation does not handle circular data structures
- There is some extra overhead in allocating the use counts separately
- They are not completely foolproof
  - Attaching a handle to an object not allocated by `new` is a recipe for disaster
  - You mustn't explicitly delete an object while there is still a handle attached to it

Homework (due Monday)

- Take the `Handle` template definition and the first version of the `Expr` class definition (both available from the course website) and merge them, modifying the `Expr` class hierarchy to use the `Handle` template.
- Add an appropriate definition of `operator<<` for `Expr` output