OBJECT-ORIENTED PROGRAMMING IN C

CSCI 5448
Fall 2012

Pritha Srivastava

Introduction

- **Goal:**
  - To discover how ANSI – C can be used to write object-oriented code
  - To revisit the basic concepts in OO like Information Hiding, Polymorphism, Inheritance etc…
- **Pre-requisites** – A good knowledge of pointers, structures and function pointers

Table of Contents

Information Hiding

- Data types - a set of values and operations to work on them
- OO design paradigm states – conceal internal representation of data, expose only operations that can be used to manipulate them
- Representation of data should be known only to implementer, not to user – the answer is ***Abstract Data Types***

Information Hiding

- Make a header file only available to user, containing
  - a **descriptor** pointer (which represents the user-defined data type)
  - **functions** which are operations that can be performed on the data type
- Functions accept and return generic (**void**) **pointers** which aid in hiding the implementation details

Information Hiding

- Example: **Set** of elements
- operations – **add, find and drop.**
- Define a header file **Set.h** (exposed to user)
- **Appropriate Abstractions** – Header file name, function name reveal their purpose
- Return type - void* helps in hiding implementation details

Set.h

Type Descriptor ↑

```
extern const void * Set;

void* add(void *set, const void *element);
void* find(const void *set, const void *element);
void* drop(void *set, const void *element);
int contains(const void *set, const void *element);
```

Set.c   Main.c - Usage

Information Hiding

- **Set.c** – Contains **implementation** details of Set data type (Not exposed to user)
- The pointer **Set** (in Set.h) is passed as an argument to add, find etc.

```
Set.c
struct Set { unsigned count; };
static const size_t _Set = sizeof(struct Set);
const void * Set = & _Set;
```

Externed in Set.h

```
void* add (void *_set, void *_element)
{
    struct Set *set = _set;
    struct Object *element = _element;
    if ( !element-> in)
    {
        element->in = set;
    }
    else
        assert(element->in == set);
        ++set->count; ++element->count;
    return element;
}
find(), drop(), contains() etc …
```

Set.h   Main.c - Usage

Information Hiding

- Set is a pointer, **NOT** a data type
- Need to define a mechanism using which variables of type Set can be declared
- Define a header file – **New.h**
- **new** – creates variable conforming to descriptor Set
- **delete** – recycles variable created

New.h

Takes in pointer 'Set'

```
void* new (const void* type, ...);
void delete (void *item);
```

Arguments with which to initialize the variable

New.c

Main.c - Usage

Information Hiding

- **New.c** – Contains implementations for new() and delete()

```
void* new (const void * type, ...)
{
    const size_t size = * (const size_t *)
type;
    void * p = calloc(1, size);
    assert(p);
    return p;
}
delete() …
```

New.h   Main.c - Usage

Information Hiding

- Need another data type to represent an **Object** that will be added to a Set
- Define a header file – **Object.h**

Object.h

Type Descriptor ↑

```
extern const void *Object;
```

Compares variables of type 'Object' ↑

```
int differ(const void *a, const void *b);
```

Object.c   Main.c - Usage

Information Hiding

- Object.c – Contains implementation details of Object data type (Not exposed to user)

```
struct Object { unsigned count; struct Set * in; };
static const size_t _Object = sizeof(struct Object);
const void * Object = & _Object;
```

Externed in Object.h

```
int differ (const void * a, const void * b)
{
return a != b;
}
```

Object.h Main.c - Usage

Information Hiding

- Application to demonstrate the **usage** of Set.h, Object.h & New.h

```
#include <stdio.h>
#include "New.h"
#include "Set.h"
#include "Object.h"
int main()
{
    void *s = new (Set);
    void *a = add(s, new(Object);
    void *b = add(s, new(Object));
    void *c = new(Object);

    if(contains(s, a) && contains(s,b))
        puts("OK");
    delete(drop(s, b));
    delete(drop(s, a));
}
```

Only header files given to user

Pointer 'Set' externed in Set.h

Pointer 'Object' externed in Object.h

Output:

OK

Set.h Set.c Object.h Object.c New.h New.c

Dynamic Linkage & Polymorphism

- A **generic function** should be able to invoke **type-specific functions** using the pointer to the object
- Demonstrate with an example how **function pointers** can be used to achieve this
- Introduce how **constructors, destructors** and other such generic functions can be defined and invoked dynamically

Dynamic Linkage & Polymorphism

- Problem:
  - Implement a String data type to be included/ added to a Set
  - Requires a dynamic buffer to hold data
- Possible Solution:
  - new() – can include memory allocation; but will have a chain of ‘if’ statements to support memory allocations and initializations specific to each data-type
  - Similar problems with delete() for reclamation of memory allocated

Dynamic Linkage & Polymorphism

- Elegant Solution:
  - Each object must be responsible for initializing and deleting its own resources (constructor & destructor)
  - new() – responsible for allocating memory for struct String & constructor responsible for allocating memory for the text buffer within struct String and other type-specific initializations
  - delete() – responsible for freeing up memory allocated for struct String & destructor responsible for freeing up memory allocated for text buffer within struct String

Dynamic Linkage & Polymorphism

- How to Locate the constructor & destructor within new() & delete() ?
- Define a table of function pointers which can be common for each data-type
- Associate this table with the data-type itself
- Example of table – Struct Class

```
struct Class {
/* Size of the object */
size_t size;

/* Constructor */
void * (* ctor) (void * self, va_list * app);

 /* Destructor */
 void * (* dtor) (void * self);

/* Makes a copy of the object self */
 void * (* clone) (const void * self);

/* Compares two objects */
int (* differ) (const void * self, const void * b);
};
```

Dynamic Linkage & Polymorphism

- struct Class has to be made a part of the data - type
- pointer to struct Class is there in the data - type String and Set

```
struct String {
const void * class;  /* must be first */
char * text;
};
struct Set {
const void * class; /* must be first */
...
};
```

Dynamic Linkage & Polymorphism

- struct Class pointer at the beginning of each Object is important, so that it can be used to locate the dynamically linked function (constructor & destructor) as shown
- new() & delete() can be used to allocate memory for any data-type

```
void delete (void * self)
{
    const struct Class ** cp = self;
    if (self && * cp && (* cp) —> dtor)
    self = (* cp) —> dtor(self);
   free(self);
}
```

```
void * new (const void * _class, ...)
{
    const struct Class * class = _class;
    void * p = calloc(1, class —> size);
    * (const struct Class **) p = class;
    if (class —> ctor)
    {
        va_list ap;
        va_start(ap, _class);
        p = class —> ctor(p, & ap);
        va_end(ap);
    }
    return p;
}
```

Allocate memory for p of size given in _class

Assign class at the beginning of the new variable p

Locate and invoke the dynamically linked constructor

Dynamic Linkage & Polymorphism

```
int differ (const void * self, const void * b)
{
    const struct Class * const * cp = self;
    assert(self && * cp && (* cp) —>differ);
    return (* cp) —> differ(self, b);
}
```

Dynamically linked function

```
size_t sizeOf (const void * self)
{
    const struct Class * const * cp = self;
    assert(self && * cp);
    return (* cp) —> size;
}
```

Variable which stores size in struct Class

- **Polymorphism:** differ() is a generic function which takes in arguments of any type (void *), and invokes the appropriate dynamically linked function based on the type of the object

- **Dynamic Linkage/ Late Binding:** the function that does the actual work is called only during execution
- **Static Linkage:** Demonstrated by sizeOf(). It can take in any object as argument and return its size which is stored as a variable in the pointer of type struct Class

Dynamic Linkage & Polymorphism

- Define a header file **String.h** which defines the abstract data type- String:

String.h

```
extern const void * String;
```

Dynamic Linkage & Polymorphism

- Define another header file **String.r** which is the representation file for String data-type

String.r

```
struct String {
    /* must be first */
    const void * class;
    char * text;
};
```

Dynamic Linkage & Polymorphism

- **String.c** – Initialize the function pointer table with the type-specific functions
- All the functions have been qualified with **static**, since the functions should not be directly accessed by the user, but only through new(), delete(), differ() etc. defined in New.h
- **static** – helps in encapsulation

```
String.c
#include "String.r"
static void * String_ctor (void * _self, va_list * app)
{ struct String * self = _self;
const char * text = va_arg(* app, const char *);
self —> text = malloc(strlen(text) + 1);
assert(self —> text);
strcpy(self —> text, text);
return self;
}
String_dtor (), String_clone(), String_differ () …
static const struct Class _String = {
sizeof(struct String),
String_ctor, String_dtor,
String_clone, String_differ
};
const void * String = & _String;
```

Dynamic Linkage & Polymorphism

- Add the generic functions – clone(), differ() and sizeOf() in New.h

New.h

```
void * clone (const void * self);
int differ (const void * self,
const void * b);
size_t sizeOf (const void * self);
```

Dynamic Linkage & Polymorphism

- Sample Application that demonstrates the usage
- Create variable ‘a’ of type String, clone it ‘aa’ and create another variable ‘b’ of type String and compare a, b

```
#include "String.h"
#include "New.h"
int main ()
{
    void * a = new(String, "a");
    * aa = clone(a);
    void * b = new(String, "b");
    printf("sizeOf(a) == %u\n", sizeOf(a));
    if (differ(a, b))
    puts("ok");
    delete(a), delete(aa), delete(b);
    return 0;
}
```

<u>Output :</u>

```
sizeOf(a) == 8
ok
```

Inheritance

- Inheritance can be achieved by including a structure at the beginning of another
- Demonstrate Inheritance by defining a **superclass Point** with rudimentary graphics methods like draw() and move() and then define a **sub-class Circle** that derives from Point

Inheritance

- Define a header file **Point.h** for the super-class Point
- It has the type descriptor pointer 'Point' and functions to manipulate it

Point.h

```
extern const void *Point;
void move (void * point, int dx, int dy);
```

Inheritance

- Define a second header file Point.r which is the representation file of Point

Point.r

```
struct Point {
    const void * class;
    int x, y; /* coordinates */
};
```

Inheritance

- The function pointer table is initialized in Point.c
- It contains implementations for dynamically linked functions
- Move() is not dynamically linked, hence not pre-fixed with static, so can be directly invoked by user

```
Point.c
static void * Point_ctor (void * _self, va_list * app)
{
    struct Point * self = _self;
    self —> x = va_arg(* app, int);
    self —> y = va_arg(* app, int);
    return self;
}
Point_dtor(), Point_draw() … etc
static const struct Class _Point = {
sizeof(struct Point), Point_ctor, 0, Point_draw
};
const void * Point = & _Point;
void move (void * _self, int dx, int dy)
{ struct Point * self = _self;
self —> x += dx, self —> y += dy;
}
```

Inheritance

- struct Class in New.r has been modified to contain draw() in place of differ()
- differ() in New.c has been replaced with draw()

New.r

```
struct Class {
size_t size;
void * (* ctor) (void * self, va_list * app);
void * (* dtor) (void * self);
void (* draw) (const void * self);
};
```

New.c

```
void draw (const void * self)
{ const struct Class * const * cp = self;
assert(self && * cp && (* cp) —> draw);
(* cp) —> draw(self);
}
```

Inheritance

- Circle is a class that derives from Point
- Inheritance can be achieved by placing a variable of type struct Point at the beginning of struct Class:

  struct Circle { const struct Point _; int rad; };

  - Just so that the user does not access the base class using the derived class pointer, the variable name is an almost hidden underscore symbol
  - 'const' helps to protect against invalid modification of the variable of type struct Point
- Radius is initialized in its constructor:

  self —> radius = va_arg(* app, int);

Inheritance

- The internal representation file of Circle – **Circle.r** is shown

```
Circle.r


struct Circle {
const struct Point _;
int rad;
};
```

Inheritance

- Circle.c contains the table of function pointers
- It contains the implementation of the dynamically linked functions
- draw() method has been over-ridden in this case

Circle.c

```
static void * Circle_ctor (void * _self, va_list * app)
{
    struct Circle * self =
    ((const struct Class *) Point) —> ctor(_self, app);
    self —> rad = va_arg(* app, int);
    return self;
}
 static void Circle_draw (const void * _self)
{
    const struct Circle * self = _self;
    printf("circle at %d,%d rad %d\n",
    x(self), y(self), self —> rad);
}
static const struct Class _Circle = {
sizeof(struct Circle), Circle_ctor, 0, Circle_draw
};
const void * Circle = & _Circle;
```

Inheritance

- Since the initial address of the sub-class always contains a variable of the superclass, the sub-class variable can always behave like the super-class variable
- Functionality of move() remains exactly the same for Point and Circle, hence we can look for code re-use
- Passing the sub-class variable to a function like move() is fine, since move() will be able to operate only on the super-class() part which is embedded in the sub-class
  - Struct Circle can be converted to struct Point by **up-conversion** and using void* as intermediate mechanisms

Inheritance

- Sub-classes inherit statically linked functions like move() from Super-class
  - Statically linked functions can not be over-ridden in a sub-class
- Sub-classes inherit dynamically linked functions like draw() also from super-class
  - Dynamically linked functions can be over-ridden in sub-class

Visibility and Access functions

- A data-type has three files:
  - **‘.h’ file** - contains declaration of abstract data type and other functions that can be accessed by the user; application can include this file & a sub-class’s .h file will include a super-class’s .h file
  - **‘.r’ file** - contains internal representation of the class; a sub-class’s .r file will include a super-class’s .r file
  - **‘.c’ file** - contains implementation of the functions belonging to the data – type; a sub-class’s .c file include its own .h and .r file and its super-class’s .h and .r file

Visibility and Access functions

- We have an almost invisible super-class variable '_' within the sub-class, but we need to make sure that the sub-class part does not access and make changes to the super-class part.
- We define the following macros for this purpose in Point.r:

```
#define x(p) (((const struct Point *)(p)) —> x)
#define y(p) (((const struct Point *)(p)) —> y)
```

- While accessing x and y of Point within Circle, 'const' prevents any assignment to x and y

Multiple Inheritance

- Can be achieved by including the structure variables of all the super-class objects
- The downside is that we need to perform address manipulations apart from up-cast (from a sub-class variable to a super-class) , to obtain the appropriate super-class object

Inheritance vs. Aggregation

- Inheritance is shown by having struct Circle contain struct Point at its starting address:

  `struct Circle { const struct Point _; int rad; };`

- Delegation can be achieved by the following mechanism:

  `struct Circle2 { struct Point * point; int rad; };`

  - Circle2 cannot re-use the methods of Point. It can just apply Point methods to the Point component, but not to itself

- We need to decide whether to use Inheritance or Delegation using the ‘is-a’ or ‘has-a’ test

Conclusion

- ANSI-C has all the language level – mechanisms to implement object-oriented concepts
  - Static keyword
  - Function pointers
  - Structures etc…
- The downside is that implementing object-oriented concepts in C is not very straightforward and can be complex in certain situations (Multiple inheritance)

References

- http://www.cs.rit.edu/~ats/books/ooc.pdf
- http://www.eventhelix.com/realtimemantra/basics/object_oriented_programming_in_c.htm
- http://stackoverflow.com/questions/2181079/object-oriented-programming-in-c