

CS481 - Object-Oriented Programming

Implementation Hiding

- ❖ For true encapsulation
- ❖ Only interface visible
 - Not implementation details
- ❖ But, class declaration visible
 - Shows even "private" members

1

Better Hiding?

- ❖ Use an abstract base class
 - But we don't know how yet
- ❖ Use a handle class
 - Contains pointer to actual class
 - Declares interface member functions

2

Handle Classes

```
class actual;
class handle
{
  actual* p_act;
public:
  void  op1();
  void  op2();
};
```

"Cheshire cat"

Abstract
declaration

```
class actual
{
  int x;
  int y;
public:
  void  op1();
  void  op2();
};
```

3

Handle Classes

```
void handle::op1()
{
  p_act->op1();
}
```

Handle class
operation

```
void actual::op1()
{
  /* do something */
}
```

Actual
operation

4

Handle Class Use

- ❖ #include "..\handle.h"

```
main()
{
  handle h;
  h.op1();
  h.op2();
}
```

5

Handle Classes

p_act is initialized in the constructor for the handle class

```
class actual;
class handle
{
  actual* p_act;
public:
  void  op1();
  void  op2();
};
```

```
class actual
{
  int x;
  int y;
public:
  void  op1();
  void  op2();
};
```

6

CS481 - Object-Oriented Programming

Handle Classes

Also minimizes recompilation

```

class actual;
class handle
{
  actual* p_act;
public:
  void    op1();
  void    op2();
};
    
```

Why?

```

class actual
{
  int x;
  int y;
public:
  void    op1();
  void    op2();
};
    
```

Initialization

- ❖ Our C++ stack ADT
 - Declared like a normal variable
 - But must call "Init()" member
 - ◆ Bad news if we don't!
- ❖ Automatic initialization?
 - Compiler knows when created

Constructor

Also known as "ctor"

- ❖ Special member function
 - Must still be declared (in most cases)
- ❖ Name same as class name
- ❖ Automatically called
 - When instance (object) is created

Constructor Example

```

class Stack
{
  int    max_elem;
  int    current_elem;
  int*   elems;
public:
  Stack();
  ...
};
    
```

Constructor Example

```

class Stack
{
  int    max_elem;
  int    current_elem;
  int*   elems;
public:
  Stack();
  ...
};
    
```

For now, no arguments to constructor

Constructor Example

```

class Stack
{
  int    max_elem;
  int    current_elem;
  int*   elems;
public:
  Stack();
  ...
};
    
```

No return value possible!

CS481 - Object-Oriented Programming

Constructor ExamplePerforms function of `init()`

```
Stack::Stack()
{
    max_elem = STACK_SIZE_INCR;
    current_elem = -1;
    elems = (int*)
    malloc(max_elem * sizeof(int));

    assert(elems != NULL);
    assert(isEmpty());
}
```

13

Clean-up

❖ C++ stack ADT

- What if we forget to call "Free"?

◆ Memory leak! (cardinal sin of C?)

❖ Automatic clean-up?

- Use a destructor (like constructor)

◆ Also known as "dtor"

14

Destructor Example

```
class Stack
{
    int    max_elem;
    int    current_elem;
    int*   elems;
public:
    Stack();
    ~Stack();
    bool isEmpty() const;
    ...
};
```

15

Destructor Example

```
Stack::~~Stack()
{
    free(elems);
    // C++ will free the instance
    // storage.
}
```

16

Destructor Invocation

- ❖ What if multiple `auto` objects defined in a block?
- ❖ Destructors called when objects go out of scope, at end of block
- ❖ Destroyed in reverse order of creation (also globals at program end)

17

Default Constructor

❖ Constructor with no arguments

- not the same as the compiler-generated constructor (*synthesized*)
- sometimes called "null argument ctor"
- used for most common declarations
- necessary if there's an array of objects.

18

CS481 - Object-Oriented Programming

C++ Definitions

- ❖ In C, definitions at beginning
 - Of compound statement (block) - {}
- ❖ In C++, any time before use
 - Even after executable statements
- ❖ Why??
 - Convenience & constructor

19

C++ Definitions

```
void main()
{
    get_params(); // Executable
                  // statement.
    int    ii=10; //
    Stack  stack1; // Definitions
    stack1.push(ii);
}
```

20

C++ Definitions (for)

```
for (int j=0; j<100; j++)
{
    get_data(j);
}
for (int j=0; j<100; j++)
{
    put_data(j);
}
```

Loop variable (j) lifetime should be for single loop; same scoping as func params.

21

Aggregate Initialization

- ❖ Still supports the C init list:


```
int a[5] = {1, 2, 3, 4, 5};
```
- ❖ even with fewer initializers:


```
int b[10] = {}; //all are 0
```
- ❖ and automatic counting:


```
int c[] = {0, -1, -2};
```

22

Aggregate Initialization

- ❖ True Aggregates:


```
struct Aggie {int i;
              float f; char c;};
Aggie a1 = {1, 1.1, 'a'};
Aggie a[2] = {{2,2.2, 'b'}};
```
- ❖ 2nd element in a[] is all zeros.

23

Aggregate Objects

- ❖


```
struct Agg {
    float f;
    int i;
    Agg(float, int); //assigns
};
```

24

CS481 - Object-Oriented Programming

Aggregate Objects

```
❖ struct Agg {
    float f;
    int i;
public:
    Agg(float, int); //assigns
};           Constructor call
❖ Agg a2[] =
    {Agg(1.0,1),Agg(2.0,2)}; 25
```

Aggregate Objects

```
❖ Agg::Agg(float f2, int i2)
{
    f = f2;
    i = i2;
}
```

26

Function Call Syntax

```
❖ Used to create objects:
    int i(3); // int i = 3;
❖ Used to perform type casting:
    float x = (float) 2 * 6;
    float y = float(4 * 3); 27
```

