CPSC 5011: Object-Oriented Concepts

Lecture 2: OO Type & Class Design

Preservation of State, Abstraction, Encapsulation

Notion of Type

Set of values and associated operations

- Built-in, aka primitives
  - integer, real, character, logical
  - Compiler invokes appropriate operator
    - Algorithm may differ by type

    => Operator overloaded (e.g. +)
- User-defined types
  - Composites (union, record type)
- Abstract Data Type (ADT)
  - Separation of interface and implementation

OO Type Definition

Collection of Objects (values)

AND Set of Associated Operations

- Integer +, -, /, *, %, <, <=, ==, !=, >, >=
- Date -, <, <=, ==, !=, >, >=

  *, / not meaningful for dates

  + may be meaningful for mixed types

- Definition Driven by Coupling & Cohesion

Type Checking

Verify that operations performed on a type are meaningful (legal)

- Static (C++, C#,Java, Eiffel, Ada)) -- efficient
  - Type is associated with identifier -- `int x;`
  - Compiler verifies (no RT overhead)
- Dynamic (Smalltalk, Python) -- flexible
  - Type associated with value

    ```
    value = 10; value = “hello”; value = x;
    ```
  - Type tag checked at RT

Type Casting

- Implicit Type Casting (Coercion)

```
float    p = 3.14159;
int      i = 10;
p += i;          // i promoted (real temp) then added
i   = p;         // p demoted (truncated temp)
```

- Explicit Type Casting

```
float    p = 3.14159;
int      i = 10;
p += float(i);  // i promoted (real temp) then added
i   = int(p);   // p demoted (truncated temp)
```

Types in Software Design

- Procedural (Structured Programming)
  - Functional decomposition
  - Pass/return types
  - Little persistent connection between functions and data
- Modular Programming
  - A module is a set of procedures with its related data

  $\Rightarrow$ reduced maintenance cost (if highly cohesive)
- Design must consider
  - Interface of module
  - Private versus public

Types in Software Design

- ADT – increase cohesion
  - Implementation separate from interface
  - ➢ Application programmer dependent on interface NOT implementation
  - ➢ No change in application code when implementation changes
- OO = ADT +
  - Encapsulation
  - Type extension (inheritance – increases coupling)
  - Substitutability (polymorphism)

Dual Perspective supported by ADT

- *External use by an application programmer (client)*
- *Internal definition by a ADT designer*
- Separating form and function
  - development of reusable, testable code
- Application programmer dependent
  - only on the interface
  - NOT the implementation
- Internal changes impact only ADT designer
  - no change should be required in application code
  - application programmer codes only to the interface

OO Data Abstraction

- Class Construct
  - Encapsulated ADT
  - Private Implementation
  - Public interface
  - External dependencies minimized
  - Supported in modern OOPL (C++, Java, C#,…)
- Example: Queue
  - Interface: enQ(), deQ(), isEmpty(), isFull(), clear()
  - Implementation: circular array OR linked list
  - Client's use of Queue not affected by internal implementation

Class Terminology

- A **class method (member function)** is a function declared in class scope
- An **instance of a class (object)** is an allocation/ declaration of that class definition.
- Member functions must distinguish between multiple instances of the class

⇒ **this** pointer (Example 2.6)

  - holds the address of the active object
  - an implicit parameter passed with each member function invocation

- **Internal state**
  - Value of internal data members at a given point

Object Definitions

- Self-contained entity (data & operations encapsulated)
- Instance of an ADT/class that has
  - Internal state
    - Value of all fields (properties)
    - Class should control state => minimize set/get
  - Behavior
    - Public functionality
    - State changes so triggered should be consistent
  - Identity
    - Name (static type checking)
    - Value (dynamic type checking)

OO Tenets (Principles)

- Abstraction
- Encapsulation
- Information Hiding

Central Class Design Principle:

Control of state

Abstraction

- Implementation details abstracted away
- Application programmer need not know/care how type stored or manipulated
  - just as with built-in types
- Application programmer not responsible for
  - Initializing object
  - Maintaining consistent state of object
  - Proper manipulation
  - Bounds checking
- Incompetent/malicious programmer cannot subvert type

Information Hiding

- Idealization of Abstraction
  - Theory compromised by compiler's need to know size
  - Information hiding not fully realized
  - C++ (.h files contain private data declarations)
  - Java (private data and public interface in same file)
- Implementation hidden
  - only interface is public
- Application programmer has no knowledge of implementation details
  - application code dependent on interface only
- Robust
  - external forces cannot put object into invalid state
- Extensible

Encapsulation

- Wrap up type definition in one class (capsule) => data protected
- Low coupling
  - no extraneous functionality or data
  - no external manipulation of data
- High cohesion
  - all needed data AND functionality contained within type definition
- Maintainable

Violation of Encapsulation

```
class Mole
{
   int hidden;
 public:
   …      // constructors, etc.
   int getHidden()     { return hidden;}
   int& aliased()      { return hidden;}
};
```

Violation of Encapsulation

```
class Mole
{
   int hidden;
 public:
   …      // constructors, etc.
   int getHidden()     { return hidden;}
   int& aliased()      { return hidden;}
};

Mole     x, y(7), z(12);     // initialize 3 Mole objects

int&     gotcha = y.aliased();
gotcha++;

cout << x.getHidden() << y.getHidden() << z.getHidden();
```

Class Design

- Type Definition
  - No memory allocated until instantiated
- Accessibility
  - public, protected, private
- Data
  - Instance data
    - exists for every object instantiated
  - Class-wide global(s)
    - Static member(s)
    - One copy for ENTIRE class
    - Independent of objects instantiated

Class Design Functionality

- Constructor(s)
  - Set initial state
  - Allocate resources
  - No need for `initialize()`
- Accessors
  - `get()` – should be `const`
  - Controlled peek inside class
- Mutators -- minimize
  - `set()`
  - Controlled alteration of state
  - Check values
    - discard out-of-bounds values
    - provide default values

Class Design Functionality

- Private Utility functions
  - Supports reuse within class (functional decomposition)
- Protected Interface
  - Utility to be inherited by child classes
- Public Interface
  - Type of class
  - Minimum functionality needed by application programmer
- Cleanup
  - Bookkeeping (static data members)
  - C++ -- destructor (deallocate resources)
  - Java – `finalize()` – called only when GC collects dead object

Table 5.1 Types of Functions defined in class construct

Function	***Intent***	***Use***
Constructor	Set object in initial valid state Initialize data Allocate resources	Explicit with new operator Implicit in C++ (stack objects)
Destructor	Release resources Bookkeeping details	Language dependent
Accessor	View data values	Depends on accessibility
Mutator	Change data values Preserve validity of state	Depends on accessibility
Private utility	Preserve data dependencies Manage resources	Internal to class
Public interface	Support type definition Provide needed utility	Unrestricted Type related

Standard Class Design

- Class design encapsulates
  - data
  - associated functionality
- Class design should
  - Control internal state
  - Control access to data and state transitions
- Class methods include
  - accessors, mutators, private utility functions, constructors
  - C++, possibly: destructors, copy constructors, overloaded assignment
- Type driven public functionality
  - That which is essential to external manipulation of the type

Why Constructors?

- Constructors remove object initialization responsibility from client.
- Objects ‘automatically’ placed in a valid, initial state upon instantiation.
- Failure to initialize does not mean that the data has no value
  - Uninitialized data assumes whatever residual bit string value that resides in the allocated memory
  - Some languages (Java, C#) zero out fields
- A class without constructor yields objects in an indeterminate or ‘valueless’ initial state.

Why Private Utility Methods?

- Provide functional decomposition
- Improve readability and maintainability
- Encourage consistent behavior
- Encapsulation controls accessibility
  - application programmers do not have direct access to such methods.

Why Minimize Set/Get?

- Control access to object's internal data
- Promote software maintainability
  - client programs to interface, not implementation
- If object's internal state known
  - Feasible to code in a manner dependent on such internal details
  - Compromise maintainability
- Accessors (gets) expose part or all of an object state
- Mutators change object state
  - should be conditional
  - class designer should determine when it is appropriate to reject a change made through a call to a mutator.

When to define Destructor?

- C++ destructors are resource managers
- Destructors release internally allocated heap memory
- Destructors may also track number of active objects, decrement reference counts, close files, etc.

- Not available in Java
  - Though a finalize() method is recommend to aid garbage collection
- Not effectively used in C#