Advantages of OO Programming

This section covers three major sections in Final Exams. Gary asked students to write down the advantages and disadvantages of different programming languages, design principles, and homework solutions. Memorizing these sections guaranteed at least 50% of the final. The rest will be the same as previous quizzes. *Created by Yilin and M.*

- **General**
  - **natural:**
    - designing objects like the real world is most simplistic
  - **privacy/encapsulation:**
    - reduce the scope of the implementation to only in needed
  - **re-use:**
    - reduced cost, time, faster-to-market, increased reliability when code is reused in multiple applications. Eg: container behaves differently depending on the items it holds.
  - **division of labor:**
    - team members can all work independently on their pieces.
  - **extensibility:**
    - Code written today can solve problems tomorrow.

- **C: w/ void pointers and function pointers**
  - **fast:**
    - solution used simplistic programming features. The least language complexity such that compiled code could be the simplest at runtime.
  - **smallest executable:**
    - solution used simplistic programming features. Least language features.
  - **most portable:**
    - C is the most mature language with the most compilers. The oldest language with the most compilers for most platforms.
  - **destructor method took parameter:**
    - reset the pointer holding data to 0 in one operation. function pointer implementation of virtual functions: use for logical decisions aswell perform functional tasks. Allowing us to give up authority and access at the same time.
  - **least magic :**
    - all code to execute was present and expected.

- **C++/Java w/ base class and derived class**
  - **objected-oriented syntax:**
    - rich language features (inheritance, virtual functions, virtual methods …) better assist software engineer
  - **references:**
    - compiler enforces never null and no validity checks
    - Power of pointers without pointer syntax (c++).
  - **operator overloading:**
    - re-use operators for user-defined types (increased design choices of software engineer)

- **const methods:**
  - compiler enforce design of when objects should not change
- **Rich APIs (Java):**
  - You can use APIs to give your program capabilities rather than implementing from scratch.

- **<u>Java w/ Generics:</u>**
  - **homogeneous containers:**
    - compiler can enforce one type in each container.
  - **no cast of generic object:**
    - item returned is a known type making casting unnecessary.
  - **homogeneous container was a compile only check :**
    - code generated used base class/derived class and virtual tables, smaller executable than templates.

- **<u>C++ w/ templates (generics):</u>**
  - **used with primitives types -**
    - a polymorphic container can hold ints, longs, floats
  - **polymorphism extended -**
    - polymorphic friends. More design choices for engineers.
  - **virtual friends -**
    - friends can now be polymorphic
  - **virtual constructors -**
    - containers can now create polymorphic objects, knows the size and all fields
  - **no need for a base class -**
    - deriving from an abstract parent is not necessary.
  - **Instantiation an unknown object in the container:**
    - Memory management is simplified.
  - **normal parameters -**
    - parameters to methods of the object to insert are expected not of an abstract class.

- **<u>C++ w/ diskfiles:</u>**
  - **persistent storage:**
    - objects lasted longer than program execution/most like real databases/ don't have to enter all data when program begins
  - **container was not limited by memory constraints (more efficient use of memory):**
    - storage was limited only my disk space - disks are larger than memory. No mem
  - **repeatability in testing:**
    - hard/tedious test cases can be easily reproduced saving time and resulting in faster program solutions
  - **TNodes were all on RTS:**
    - no memory leaks possible, implementing most efficient memory via RTS.
  - **Flat data:**
    - read and write any object with one function call.

- ## Disadvantages:
- **General:**
    - **high learning curve:**
        - much time is spent in analysis before code can be written.
    - **much magic:**
        - programmer can be unaware due to so much being hidden
    - **overkill:**
        - an object-oriented solution may exceed the needs of the problem
    - **more development time*:**
        - bugs in OO code are often more difficult to resolve

- **C:**
    - **Lack of OO language features -**
        - software engineer must provide infrastructure present in Java and C++.
    - Least compile time checking - due to void pointers, errors could go uncovered until Run-time.
    - **Most syntactically complex -**
        - differences in object/pointer field access, function pointer as parameters and data fields.
    - **Too many pointers -**
        - parameters to most methods were pointers each needing a validity check
    - **Too many void pointers -**
        - even objects to insert into the container needed were void pointers.

- **C++/Java w/ base class and derived class :**
    - **Not natural -**
        - objects to insert must be derived from abstract parent.
        - parameters to objects to are of abstract parent.
    - **Polymorphism limited -**
        - no virtual constructors or virtual friends.
    - **Container is heterogeneous -**
        - as long as a class were derived from Base, it could be inserted. (Hash table can be inserted into hash table)
    - ***Too much magic -***
        - *destructors, implicit casting overloading and operator overloading calls were not always obvious. (C++ only)*

- **Java w/ Generics**
    - **Code generated doesn't retain type information -**
        - size, fields, everything about object stored in the container is lost.
    - **Object going in the container still must be derived from base class -**
        - parameters to methods still need to match base class.
    - **Generics is compile-time only operation**
        - Still have to use virtual table and function pointers for polymorphism. (Pointer behind the scene).

- Indirection of virtual table adds overhead -
    - direct method calls don't have this overhead.
- **Explanation: polymorphism is achieved through virtual table behind the scene. Every object has to find its own function through virtual table, and then call the function. In c++, every objects is independent. Every object has its own function.**
    - **Eg: HashTable.**

- C++ w/ Templates
    - **Compiler checks were more verbose than they needed to be -**
        - more straightforward explanation may be possible.
    - **Extra syntax**
        - (template class <Whatever> prefix), must be present before every block of code.
    - **Code less portable -**
        - compilers still work with different standards.
    - **Largest executable -**
        - one set of object code generated for each use of the template Container.

- C++ w/ Templates and Diskfiles:
    - **Most code -**
        - one line of C++/templates became two lines with the disk file.
    - **May be too slow -**
        - performance due to disk accesses (Read and Write).
    - **Wasted space in disk file -**
        - TNodes existing in file that are not in the tree (Nodes being removed are still in the datafile).
    - **Not best for security -**
        - analysis of disk file can reveal information, even though data was deleted.
    - **Debugging was difficult -**
        - had to rely on viewing octal dump to resolve problems.
    - **Highest complexity of memory management -**
        - you had to track whether or not an item was synchronized to disk.

Final Information:

Part 1: [20%] one page of questions from quiz 2 - quiz 5; (no bit manipulation)
Part 2: [20%] HashTable question: probe sequence, fair share, ordered hashing
Part 3: [20%] Tree: memory, disk, drawing, comparison, octal dump.
Part 4: [20%] Efficiency
Part 5: [20%] C++ (1 page) language features;
OO design (1 page) advantages and disadvantages

Quiz3

- Efficiency of stack, linked list (push, pop, top, view)
- Polymorphic generic containers: change behavior depend on object it holds. Objects loses identity when inserted into a container, regains identity when removed from container. Restrictions on objects.
- Singly, doubly and circular doubly linked list.
- Insert, removing achieved through same set of codes.
- Arrow points to top of the Node really points to upper left corner of that Node
- List, Node, objects allocated in Heap. (nameless, accessible by starting with a named pointer)
- Stack(first in, last out) and queue(first in, first out)
- HW5(linked list): data stored using void pointer, manipulation achieved through function pointers(copy, delete, is_greater_than, write).
- HW5 Driver1 Vs Driver2: driver1 allocates MyRec on Heap(multiple) , driver2 allocates on RTS(only one reusable myRec object). Display five times - copy is not made
- MyRec simple representative of any future objects. Object - to test; methods - to conform with constraints. Test Calc - 1+2
- Copy func not found in Driver1 and driver2; **new_Myrec** is not a constraint method.
- Driver -> stack -> list -> Node -> myRec; layers
- insert_Node pointer parameter, the Node before. (null - empty list)
- Constraints methods: myRec, determination, first lines.
- Delete func from remove.
- Design pattern: facade, Algorithm: delegation
- Stack codes for top(view), Push(insert), Pop(remove_list)
- Insertion: integrate and attach
- Copy func, delete func
- Methods work based on parameters and returns; layers design
- Typeof; optional/mandatory tag
- HashTable: insert and search for where it belongs; array based; prime size; commonly lookup using key.
- **Index**: ASCII_Sum % table_size; **increment**: ASCII_Sum % (tablesize -1) + 1; **next**: (current + increment) % table_size
- HashCode(ASCII_sum), probe sequence, fair share algorithm, ordered hashing.
- HW6 Base pointer: (long) *element; * table[index] = *element;

Quiz4:

- Efficiency of binary tree (log2n); spindly, <u>bushy (bushy is prefered)</u>
- Root, parent, left and right, child, siblings, ancestors, descendants, leaf
- Height (1+ tallest child, 0 for leaf, -1 for non existent node); balance: leftH - rightH;
- Search path, (in - alphabetic , pre - to copy , post - to delete - order traversal)
- HW7 inserting from a to z: linked list

- Lazy remove: height, balance unchanged.
- Recursion (stack overflow, tail recursion, less code, more memory), loop.
- Return address on RTS; recursion (parameters and local variables), stack trace; program counter
- Hw6,7: UCSDStudent had to provide definitions for the virtual functions of Base, be a public derivation from Base.
- Virtual(default in java, keyword in C++), final(default in C++, keyword in Java); abstract class and methods.
- Reference in C++(establish at the time of declaration, can't be reset)
- Scope resolution: define member method outside its class definition, access static data or static methods.
- Operator overloading. (work on objects, member methods)
- Friend
- Cin, cout, global object of iostream on data section

Quiz5:

- Guaranteed Initialization: const, reference, parameters required for object data field; OPT primitive; CAN'T array; same order
- Mandatory semicolon end of class definition
- Const in method parameters and calling object. (input, output paramters)
- Generic: <Instantiation>
- Child in charge -> PointerInParent;
- Predecessor, successor
- SHB: current TNode deallocate before SHAB returns; start PIP is this; end PIP is the deleted TNode's replacement.
- elementTnode in remove: not in the tree, on the runtime stack
- Default constructors
- Access right: public, protected, private derivation of public, protected, private sections of parents
- HW9: TNodes allocated on Runtime Stack, no delete to avoid memory leaks; this_postion, positionInParent
- Serialization: read, write, flat.
- Code translation.
- Check input from file or from keyboard: reading from file, EOF; cin always for while loop, cout always for results
- Binary tree for a heap data structure: complete(as far left as possible), heap order; remove from root, last Node inserted is final leaf

Index: parant (index-1)/2; left child (index*2)+1; right child(index*2)+2
Union object

Advantages and Disadvantages Summary Two

OOD

Pros:

1. Natural: designing objects like the real word is most simplistic.
2. Privacy, encapsulation: reduce the scope of implementation to only in needed
3. Reusable: different parts of code can be used for multiple times. Save time, faster to market. Polymorphism
4. Division of Labor
5. Extensibility

Cons:

1. Overkill. Sometimes object-oriented solution exceeds the needs of application
2. High learning curve, spend too much time designing
3. Much magic: too much is hidden
4. Longer development time. Harder to debug

C with void and function pointers

Pros:

1. Fast and smallest executable: use the least language features.
2. Most portable: supported by most compilers
3. Destructor takes in two parameters: reset pointer holding data in one operation
4. Least magic: all code to execute was present and expected.
5. *Function pointer implementation of virtual table: use for logical decisions as well performed function tasks.*

Cons:

1. Lack of OOP language features: engineer must provide infrastructure present in Java and C++
2. Least compile time checking: void pointers, errors may occur at runtime
3. Syntax is very complicated: differences in object field access, function parameters pointers
4. Too many pointers: need validity checks. Void pointers: those are items inserted into the list

C++/Java with base class and derived class

Pros:

1. Object oriented syntax: rich language features, more choices for engineers.

2. Reference: compiler enforces never null, no validity checks
3. Operator overloading: reusable operators for user-defined types. (increase design choices)
4. Const methods: compiler enforce design of when objects should not change.
5. Garbage collectors: no need to worry about memory deallocation
6. Rich APIs: more capabilities rather than implementing from scratch

Cons:

1. Not natural: must derive from base; parameters to objects are of abstract parents
2. Limited polymorphism: no virtual constructors or virtual friends
3. Containers are heterogenous - anything derive from base can be inserted
4. Too much magic: destructors, implicit casting/operator overloading. Calls were not always obvious.

Java with Generics

Pros:

1. Homogenous containers - compiler can enforce one type in each container
2. No casting of generic objects - know the type of objects
3. Only a compile time check - code generated used base class and virtual table, smaller executable than C++ with template

Cons:

1. Code generate doesn't retain type information - size, fields, and information.
2. Objects going to container must derive from base - parameters still match abstract parents
3. Indirection of virtual tables. Rather than direct function calls

C++ with Template

Pros:

1. Used with primitive data types - long, chars
2. Polymorphism extended: polymorphic friends
3. Virtual friend and constructor - containers can now create objects, knowing this size and fields.
4. No need for a base class. Normal parameters: parameters are expected, not of an abstract class

Cons:

1. Compiler checks are more verbose than they needed to be
2. Extra syntax (template class<> prefix) must be present for every block
3. Code less portable - compiler still work with different standards
4. Largest executable - one set of object code generated for each use of the template container

C++ with diskfile

Pros

1. Persistent storage: data last longer than program execution/like real data base

2. Containers not limited by memory constraints: only by disk space (disk larger than memory)
3. Repeatability in testing: no need to renter everything
4. Most efficient use of memory; TNodes all on RTS, memory leak not possible
5. Flat data: read and write sequentially with one function calls

Cons:

1. No secured. Analysis of disk file can reveal information
2. Not efficient, may be slow: read and write from disk is slower than heap and rts access.
3. Most codes: two extra lines of codes
4. Waste space on dikse
5. Highest complexity of memory management;
6. Hard to debug: rely on viewing octal dump.

Other Summary

Quizzes (2 - 5):

- **Q#2:**
- Cannot dereference a void pointer;
- The unit of allocation for the parameter sent to "malloc" is ***bytes;***
- **0** is all type;
- new_Stack function in hw3:
    - Initialize the stack
    - Allocate
- Calling free gives up the authority
- Setting the pointer to null gives up the access
- In C, 3 illegal declarations:
    - Return a function
    - Return an array
    - Return an array of functions

- **Q#3:**
- In using a polymorphic generic container, the container changes its behavior based on the object it holds
- Hw3 is not generic container because it holds long
- Hw4 is not generic container because it holds char
- Hw5 is generic container using MyRec
- In our linked list of hw5, the data is stored using void pointers.
- The MyRec object structure definition of hw5 was created with what two design goals
    - All test objects to be simple
    - Representative for future objects
- What is the one method of MyRec from hw5 that is not called polymorphically
    - new_myRec
- a struct in C defines an object containing only data fields and is similar to a class without methods in Java or C++

- If a user runs driver2, inserts 5 items, and when displayed, the last item inserted is displayed 5 times, what is the likely cause?
    - The copy was not made
- In what three areas in the code for hw5 are the constraint methods determined?
    - List struct definition
    - New list constructor method
    - new stack constructor method
- Describe the first lines of code in all constraint methods in hw5.
    - Declaration of the pointer of the true type; initialize by casting
- What is the word in English to describe the field of an object that is known regardless of whether the operation is lookup or insert?
    - Key
- For the case of duplicate insertion in a hash table, how should the programmer decide how duplicates should be processed?
    - Look to the application

- **Q#4**
- A binary tree is so named due to what data fields? -
    - Left and right
- What is the term for the address in the text section of memory where execution resumes when a function call completes?-
    - Return address
- T or F: Before a function execution begins, the address of the next instruction is pushed on the RTS.
    - T
- What happens when a valid function pointer is stored into the PC (program counter). Pc - holds the address of programs to be executed
    - Function starts to execute ing
- Are functions in C++ by default virtual or final? - final
- What can't you do using an abstract class? - instantiate
- Secret algorithm is most likely defined outside the class definition
- return statement most likely existing in every overloaded<< method of class ostream: return *this;

- **Q#5:**
- const parameter cannot be changed in a function call
- const calling objects cannot changed in a function call
- If instantiation argument is existing type, then the code is for user of generic container, otherwise it is for the implementation of a generic container
- Output parameters are expected to change
- To be inserted into the tree in hw8, an object must provide all the methods will be called
- constructors for data allocated objects are called before main is called.
- destructors for data allocated objects are called after main execution is completed.
- ParentTNode and Tree are parameters of TNode constructor for initializing the data fields of TNode.
- What is the evaluation result of the following expression: cout << "Hello World\n"; cout (<< bit shift operation; result of action: Hello World show up on the screen)
- Serialization: reading or writing an object byte by byte.

- Iterator: the way to sequentially access or traverse each object in a container or collection.
- To make a heap data structure, we need to have a complete tree and maintain the heap order.
- The sizeof a union object in C is the sizeof its largest data field. T (union structure: only allocate the largest whatever lists there) (pros: change type without casting)
- **declaration in C**
    - Start reading the declaration at the name or the innermost set of parenthesis
        - Reading forward until you have to go back
            - Grouping parenthesis
            - End of your declaration
        - Until you have accounted for each symbol

Class_Review (06/ 05/ 17):

- In your C++ Drivers for hw9, you could have had the code:

```
//(A)                //(B)                //(C)
ifstream * is;       ifstream * is;       ifstream * is;

if (*is == cin) {    if (is == &cin) {    if (!*is) {
      ...                  ...                  ...
}                    }                    }
```

(15 pts) What is the purpose of the above code?

A) Answer: To check if input is coming from keyboard or file

B) Answer: To check if input is coming from keyboard or file

C) Answer: To check if input had reached EOF

(15 pts) What features of C++ (not C) allow the above to execute correctly?

- This question just asks for an enumeration, not an explanation. However in order for you to understand the enumeration, I am providing an explanation.

- **namespace declaration**
    - In the above examples, "cin" is a global variable of type "istream." In order to say "cin" with the std:: scope resolution operator, the code would have had to have a namespace declaration.
- **Inheritance**
    - In part B, In order to make a comparison for equality for two pointers of different types (istream, ifstream), just as in your assignments, the only way the compiler allows the expression is if one class is derived from the other. "ifstream" must be derived from "istream."
- **Operator overloading**
    - In part A, you are calling the public overloaded operator== member function. As in all your assignments for overloaded== functions, the parameter was passed by reference. Again due to our experience with the operator== functions, we can take an educated guess that the compiler would enforce both objects to remain constant during the comparison.
- **constant member function**

- In part C, due to C++'s hierarchy of implicit type conversion, this either a call to the public member function, operator void *, or the operator ! public member function. We can also assume that this is a constant member function, defined in a parent class of ifstream.

- **Operator overloading**
    - Lastly, as with many of your similar functions, operator == and casting overloading, these function were very short, defined within the class definition. Due to our experience, we can reasonably assume the same implementation for the functions above.
- **Constructors and destructors**
    - All member functions defined within class definitions are "inline" expanded by the C++ compiler. Constructors and destructors would have been called to create and destroy the stream objects (cin and *is). Therefore, the answers are:

  Classes, reference, inheritance, comment, access restrictions, streams

    - A) Answer: classes, inheritance, member functions, comment,
      access restrictions (public), references, operator overloading,
      constant member functions, streams, inline functions, namespace,
      constructors/destructors.
      Constant parameters were a recent addition to C.

      B) Answer: classes, inheritance, streams, comment, namespace,
      constructors/destructors.

      C) Answer: classes, inheritance, member functions, comment,
      access restrictions (public), casting overloading, streams, inline functions,
      constructors/destructors.

(15 pts) What member functions, if any, of istream are called in the above?
A) Answer: operator ==

B) Answer: none

C) Answer: operator void * or operator !