ECE 220

Lecture x001B - Final exam review

Tree traversal

- Last time, we printed a tree's boundary:
  - root + left boundary + leaves + right boundary
- Did this in counter-clockwise fashion
- Does the type of traversal matter for CW vs. CCW?
  - Then how do you print a tree in ***clockwise fashion?***

Tree traversal

- Standard traversals are:
  - Preorder: N-L-R
  - Inorder: L-N-R
  - Postorder: L-R-N

Will do CCW

- *Reversed* traversals are:
  - Preorder: N-R-L
  - Inorder: R-N-L
  - Postorder: R-L-N

Will do CW

Announcements

- Conflict exam policy (recap e-mails)
- HKN review session (**1230 - 1500 hrs, 12/14 in ECEB 1002**)
- Programming competition tomorrow at **7.00 pm in ECEB 1013**.
- Additional study material is on the course website
- Exam format
- Reminder: go.illinois.edu/flex

Topics to review …

<u>Part 1: LC-3</u>

- Assembly language programming & process
- Stacks, memory layout (arrays/structs)
- Memory-mapped I/O: input from keyboard, output to monitor
- TRAPs & Subroutines, Interrupts & Exceptions

<u>Part 2: C</u>

- Built-in data types, operators, scope
- Functions & run-time stack
- Pointers & arrays
- Recursion: searching, sorting, backtracking
- I/O: streams and buffers, read from / write to file
- User-defined data types: enum, struct, union
- Dynamic memory allocation
- Linked data structures: linked list (stack, queue) & trees

<u>Part 3: C++</u>

- Class (encapsulation, inheritance, abstraction)
- Pass by value /(const) reference / address
- Virtual function, operator overload, template (polymorphism)
- STL (vectors, lists, iterators, etc.)

Part 1 - LC3

- Address space: 216 locations, addressability: 16 bits
- General-purpose registers: R0, R1, ... R7
- Special-purpose register: PC, IR
- Input from keyboard: KBDR/KBSR
- Output to monitor: DDR/DSR
- Operate instructions: ADD, AND, NOT
- Data movement instructions: LD, LDI, LDR, LEA, ST, STR, STI
- Control instructions: BR, JSR/JSRR, JMP, RET, TRAP, RTI
- Condition codes: N (negative), Z (zero), P (positive)
- TRAPs: In, GETC, OUT, PUTS (uses R0; R7 is modified after call)
- Subroutines: callee-save vs. caller-save, nested subroutine needs to save R7
- Interrupts: external event, supervisor vs. user stack, RTI instruction
- Exceptions: internal event for handling errors
- Stack: FILO, overflow, underflow, R6 – stack ptr, R5 – frame ptr

Part 2 - C language

- Scope: local vs. global variables (determined by location of declaration)
- Storage class: static (retains value, global data area) vs. automatic (stack)
- Control structures: conditionals (if, if-else, switch); loops (for, while, do-while)
- Functions & run-time stack (C to LC-3)
- Pointer: address of a variable in memory
- Array: a list of values arranged sequentially in memory
- Pass by value vs. pass by reference (pointer)
- Pointer Array Duality (int array[10] = {1,2,3,4,5,6,7,8,9}; int *ptr = array;)
- Recursion: base case(s) & recursive case(s)
- File I/O: fopen, fclose, fscanf, fprintf
- Linked lists & trees (pointer, struct, dynamic memory allocation)

Part 3 - C++ language

- Class vs. struct: 4 features of OOP (polymorphism, inheritance, encapsulation, abstraction)
- Dynamic memory allocation: new & delete
- Basic I/O: std, cin, cout
- Pass by value vs. pass by address vs. pass by (const) reference
- Operator and function overloading
- Base class & derived Class: access identifier (public, protected, private)
- Virtual function & virtual function table: static vs. dynamic binding
- Function and class templates: separate type with container
- Big three: copy constructor (deep vs. shallow copy), destructor, copy assignment operator
- Implicit ‘this’ pointer: a pointer to the invoking object
- Vectors: dynamic arrays, elements are stored in consecutive locations
- Lists: doubly linked lists, elements are allocated individually
- Iterators: the mechanism used to minimize an algorithm’s dependency on the data structure on which it operates

Last new topic/information

- Dynamic dispatch. Recall Bruno the cat and his lunch?

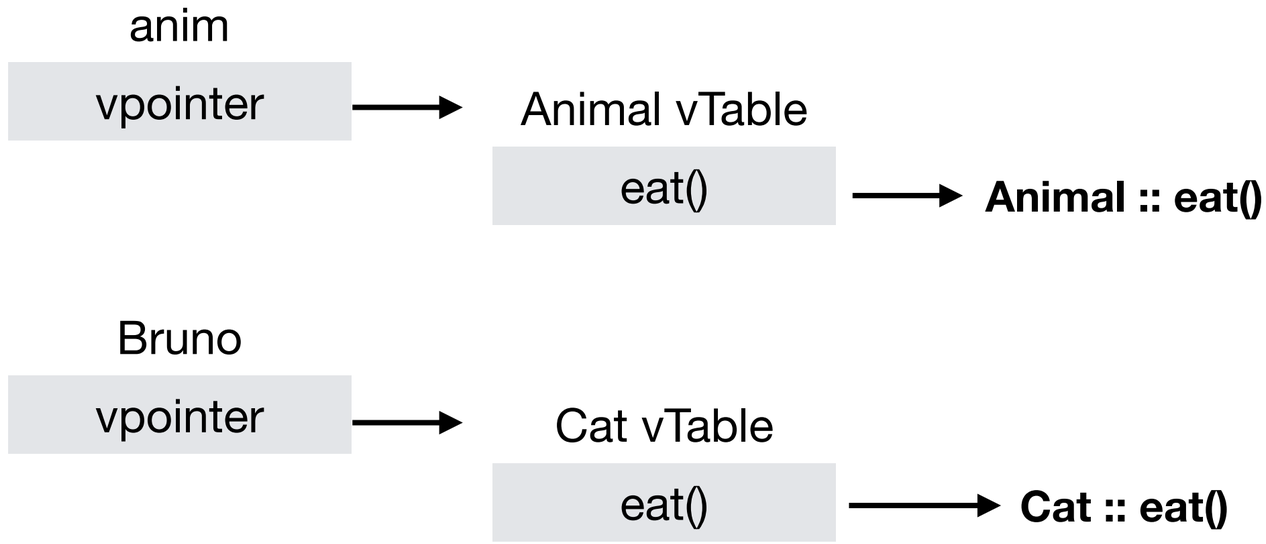
```
int main(){
  Animal *anim = new Animal();
  Cat *bruno = new Cat();
  anim->eat();
  bruno->eat();

  eat_lunch(anim);
  eat_lunch(bruno);
}
```

How is this accomplished?

```
#include <iostream>
using namespace std;

class Animal{
public:
  virtual void eat(){
    cout << "I'm eating generic food." << endl;
  }
};

class Cat : public Animal{
public:
  void eat(){
    cout << "I'm eating a mouse." << endl;
  }
};

void eat_lunch(Animal *a){
  a->eat();
}
```

Virtual functions

- Function to call determined at ***runtime.***
  - Called dynamic dispatch or linkage.
  - Commonly accomplished using virtual function/method table.
- Key idea(s):
  - You can define pointers to functions also (see Github).
  - For each class with virtual functions or deriving from a class with virtual functions, a ***vtable*** is maintained.
  - Compiler adds a pointer ***vpointer*** to this **vtable** to as data member to all objects.




```
void eat_lunch(Animal *a){
  a->eat();

Cat *bruno = new Cat();
eat_lunch(bruno);
```

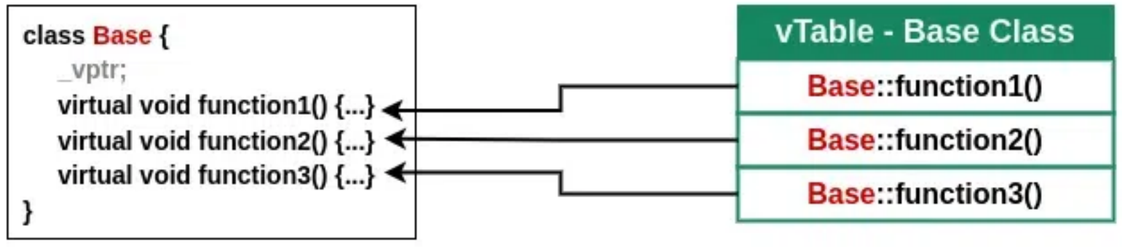
Another example

```cpp
// Base class
class Base {
public:
    virtual void function1(){
        cout << "Base function1()" << endl;
    }
    virtual void function2(){
        cout << "Base function2()" << endl;
    }
    virtual void function3(){
        cout << "Base function3()" << endl;
    }
};

// class derived from Base
class Derived1 : public Base {
public:
    // overriding function1()
    void function1(){
        cout << "Derived1 function1()" << endl;
    }
    // not overriding function2() and function3()
};
```

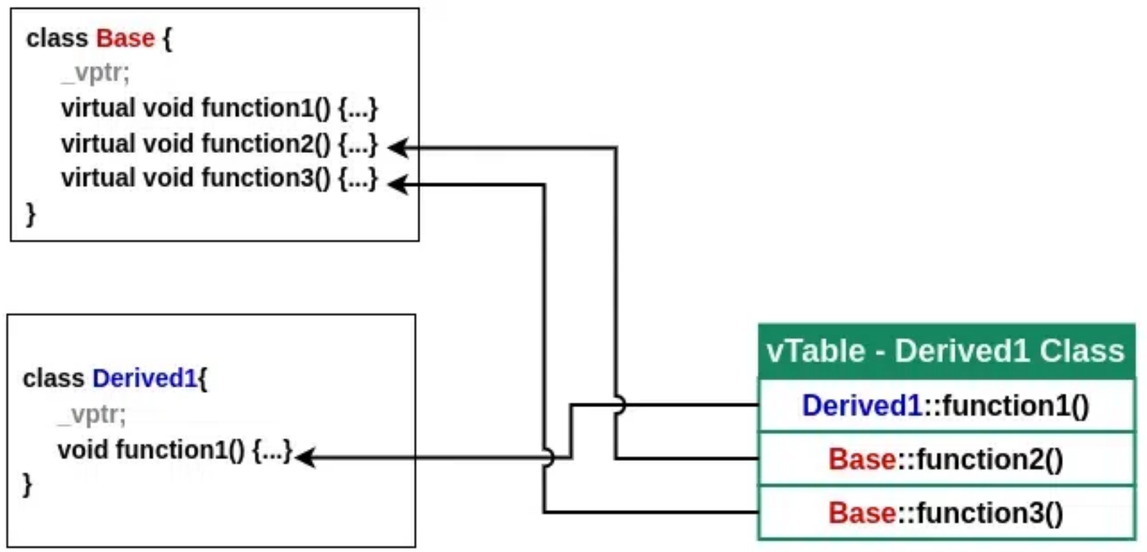
```cpp
// class derived from Derived1
class Derived2 : public Derived1 {
public:
    // again overriding function2()
    void function2(){
        cout << "Derived2 function2()" << endl;
    }
    // not overriding function1() and function3()
};

// driver code
int main(){
    // defining base class pointers
    Base* ptr1 = new Base();
    Base* ptr2 = new Derived1();
    Base* ptr3 = new Derived2();

    return 0;
}
```

Source: https://www.geeksforgeeks.org/vtable-and-vptr-in-cpp/

Another example

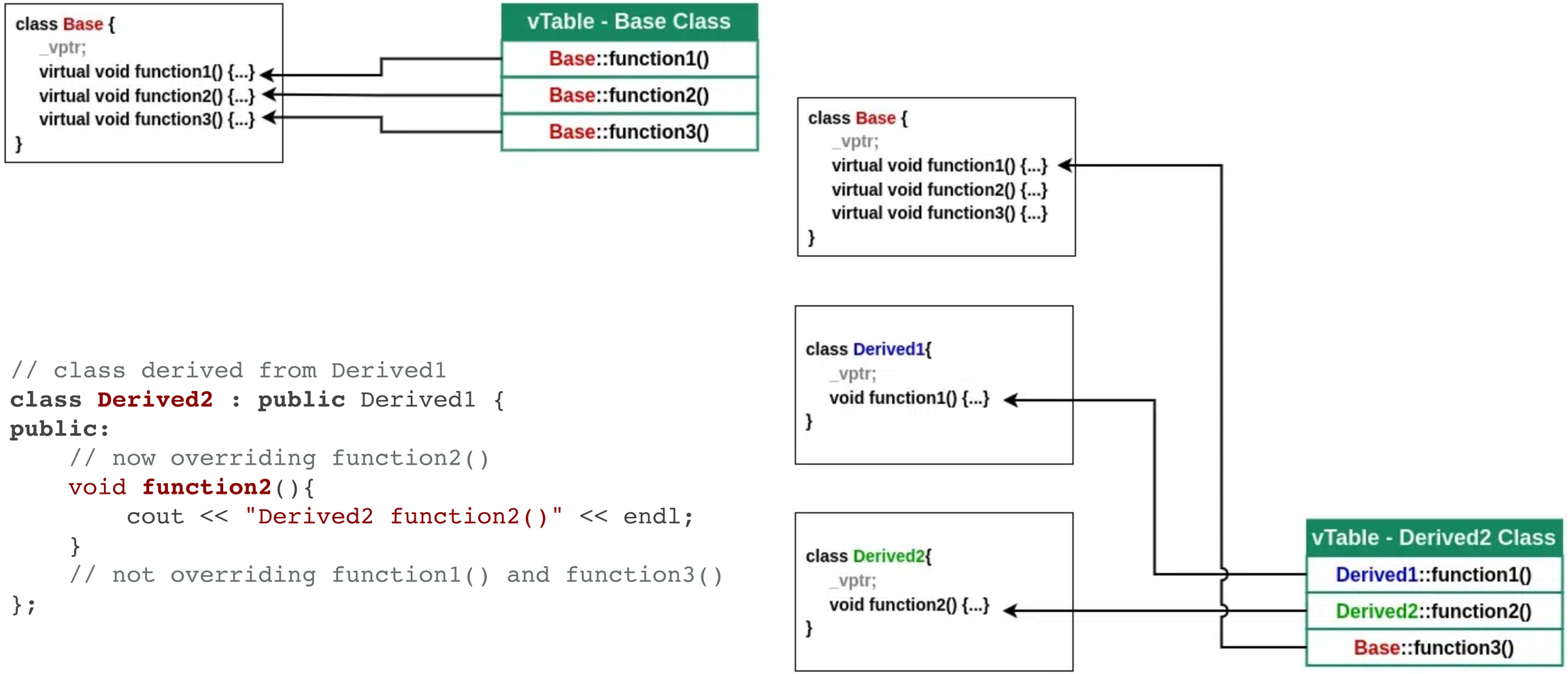



```
// class derived from Base
class Derived1 : public Base {
public:
    // overriding function1()
    void function1(){
        cout << "Derived1 function1()" << endl;
    }
    // not overriding function2() and function3()
};
```



Source: https://www.geeksforgeeks.org/vtable-and-vptr-in-cpp/

Another example




```
// class derived from Derived1
class Derived2 : public Derived1 {
public:
    // now overriding function2()
    void function2(){
        cout << "Derived2 function2()" << endl;
    }
    // not overriding function1() and function3()
};
```

Source: https://www.geeksforgeeks.org/vtable-and-vptr-in-cpp/

Practice material

- Let’s do some posted practice material
    - Circularly linked lists
    - Farey sequence (linked lists)
    - CPP exercsie
    - Tree to doubly linked list