student@ubuntu:~$./COMPUTER_SCIENCE_BASICS

Notes from second year and earlier:
Various Sources, Incomplete, Updated: June 3

What is Computer Science?

Computer Science is the study of the theory of computation, and the design of software system.

1. Introduction to Software Engineering:

What is a class?

- user defined data type
- holds its own data members and member functions
- can be used to create an instance of that class

Access Modifiers:	
Define: public	- everybody can use
Define: protected	- can be accessed by derived classes (inherited attributes from parent)
Define: private	- can only be accessed in the class they are defined or initialized.

Static Declarations

When we declare a member of a class as static it means no matter how many objects of class are there is only one copy of the static member

Using Static: Example (C++)

```
//Example: id, nextId - assign each Customer a unique id
```

```
//In header:

private:
          static int nextId;
          int id;

//In source:

int Customer::nextId=100;

public Customer:: Customer(...): id(nextId++){ ... }
```

const

- const - constant, unchanged

        - enforces documentation

When to use:

- When a variable is not to be changed (if passed as parameter or declaration

of variable)

- const function:

- can always be called

- non-const can only be called by non-const objects

- **const functions: function will not make changes to the object**

Define Pass By Reference

- a copy of the address of the actual parameter is stored. Use pass by reference when you are changing the parameter passed in by the client program

Define Pass By Value

- pass by value means that you are making a copy in memory of the actual parameters value is passed in, a copy of the contents of the actual parameters

- any changes you make will not affect the original

- the copy is stored on the stack frame

Define Constructor

- a member function of a class that initializes objects of a class

Constructor Example (C++):

```
// Member functions definitions including constructor
Line::Line(int len) {
   cout << "Normal constructor allocating ptr" <<endl;

   // allocate memory for the pointer;
   ptr = new int;
   *ptr = len;
}
```

Define Default Constructor

- a constructor, that has no parameters, (or default parameters)
- does not take any arguments
- the compiler will automatically provide default constructor implicitly
- the default value of variable is 0 in case of automatic initialization

Define Copy Constructor

A copy constructor is a member function which initializes an object using another object of the same class.

Generally has prototype classname(const classname &old_obj);

Example: Copy Constructor (C++)

```
Line::Line(const Line &obj) {
   cout << "Copy constructor allocating ptr." <<endl;
   ptr = new int;
   *ptr = *obj.ptr; // copy the value
}
```

Define Conversion Constructor

- single parameter that is declared without the function specifier explicit

- The compiler uses conversion constructors to convert objects from one type of the first parameter to the type of the conversion constructors class

Define Destructor

- called when the lifetime ends

- frees resources that the object may have acquired during its lifetime

Example: Linked List (C++) Header and Construtor

```
//In Header:

class LinkedList{
   class Node{
      friend class LinkedList;
      Type * data;
       Node *next;
  }

private:
```

```
Node *head;
}

//In Source:

LinkedList::LinkedList(): head(0){}
```

Define Encapsulation

- wrapping up data and information into a single unit (like a class or an object)

- Model, View, Controller

Define Least Privilege

Only give privileges to things that need privileges

Define Friendship

- can access protected and private members of other class which is declared as friend

Define Object Oriented Composition

The collection of attributes that make up an object

Ex. : A student is made up of:

- name
- student number
- gpa

Define Instructor

Creating new classes from existing classes, they inherit functions and properties

Example: Inheritance Syntax (C++)

```
//In child:
//header:
class aChild : public Parent {
}
//source (constructor)
aChild::aChild(): aParent(attribute ... ) {}
```

Define Polymorphism

- able to assign a different meaning or usage to something in different contexts

- or to allow a variable, function, operator, or object to have more than one forms

Abstract Classes C++

- normally a base class cannot be used to instantiate (instance) objects and can be too general to be their own objects

A class is made abstract by declaring one of more of its virtual functions to be pure virtual.

Example:

```
virtual void draw() const=0; //pure virtual function
```

Virtual And Pure Virtual Functions:

Pure virtual functions do not provide implementations.

Derived classes must override all base-class pure virtual functions with concrete implementations of those functions. If not, the derived class will be abstract as well.

Pure virtual functions are used when it does not make sense for the base class to have a function implementation.

Attempting to instantiate an object of an abstract class, or a

derived class without overriding pure virtual functions will result in compilation error.

(Abstract Classes) can also have data members and concrete functions, which are subject to normal rules of inheritance.

Virtual functions have an implementation, pure virtual functions do not have an implementation.

virtual - option of overriding base class function

pure virtual - base class function must be overridden.

virtual functions - functions that have a definition but can also be overridden by derived classes.

Example: Virtual Functions (C++)

```
//In base class: Animal.h
virtual void nameAnimal()=0;

//In derived class: Cat.h

virtual inline void nameAnimal(){ cout<<"A CAT \n"; }
inline void animalType(){ cout<<"base \n";}

//In derived class: TabbyCat.h

inline void nameAnimal(){ cout<<"A TABBY cat \n";}
inline void animalType(){ cout<<"derived \n";}

//In main class:

Cat c;
TabbyCat tc;

//virtual function, binded at runtime
c.nameAnimal();  // would print A CAT
tc.nameAnimal(); //  would print A TABBY cat

//non virtual function, binded at compile time
c.animalType(); //  would print base
```

```
tc.animalType(); //  would print base
```

Define Virtual Destructor

- create a virtual destructor (a destructor with the keyword virtual) in the base class.

- this makes all derived class destructors virtual.

- when the destructor is called, the destructor for the derived class and the base class is called.

- the base class destructor automatically executes after the derived-class destructor

Define Overloading

To specify more than one definition of a function name or operator in the same scope

Define: Function Overloading

- multiple definitions of a function

- functions must differ in return type, or number of parameters

Example:

```
int Person::agreedTo(int a);
//can be overloaded as:
bool Person::agreedTo(int a);
//but not:
int Person::agreedTo(bool a);
```

Operator Overloading:

- you can redefine or overload built in operators
- must have keyword operator

- :: , . , * , ?: can not be overloaded

- ex: void Person::operator+=(int b);

Cascading Function Calls:

cascading member-function calls - invoking multiple functions in the same statement

Example: Cascading Function Calls

```
//In main class:
//Time object:

Time t;

//cascading function calls
t.setHour(12).setMinute(1).setSecond(45);

//In Time.cc
Time &Time::setHour(int h){hour = h; return *this;}
Time &Time::setMinute(int m){minute=m; return *this;}
Time &Time::setSecond(int s){second=s; return *this;}
```

```
/*This works because the dot operator associates left to right, so the
program first evaluates setHour, and returns a reference to t as a value of
the function call. So like it would evaluate like this:*/

t.setHour(12).setMinute(1).setSecond(45);
           &t.setMinute(1).setSecond(45);
                       &t.setSecond(45);
                                   &t;
```

Cascading Stream Insertion Functions

- Using multiple stream insertion operators

Example:

```
std::cout << "Sum is " << number1 + number2 << std::
```

Define Stack

Stack is used for memory allocation

- variables are stored in memory
- Allocation is dealt with when the program is compiled

Define Call Stack

- a stack data structure that stores information about active subroutines and at what point they finish, and returns control when they are finished

- called at runtime, until the execution of the current statement

Define Frame Stack

- The first function in the frame stack is main()

- Function parameters are stored in frame stack

- Includes the return address, argument variables, local variables, saved copies modified by the subprogram to be restored

- Activation record, the collection of all data on the stack associated with one subprogram call

Why use a template

- only one definition for multiple types

- less code, more efficient

Class Templates:

- Everything should be defined in header or else you get errors since there are

multiple definitions

- must declare the class as a template class: template <class T>

- All objects using the class would be used as type T

STL Exceptions

- out_of_range

- invalid_argument - invalid argument passed to function

- length_error - attempt to locate to long (in a string, ...)

- bad_alloc - failed attempt to alloc with new allocator because not enough memory

Example: Out of Range Exception

```
try{
 integers.at(100) = 77;

}catch(out_of_range &outOfRange){
  cout << "Exception "<< outOfRange;
}
```

Data Types Memory Usage

Type	Size (Bytes)
bool	1
char	1
int	2

short	2
long	4
float	4
double	8
pointer	4 (OS dependent)

C Double Pointers

- pass variables by reference
- Ex: someFunction(**myVar);