1. Explain list data with examples.

- **Python Lists**
  - –The list data type also serves as Python's arrays
  - –Unlike Scheme, Common LISP, ML, and F#, Python's lists are **mutable**
  - –Elements can be of **any type**
  - –Create a list with **an assignment**

```
myList = [3, 5.8, "grape"]
```

- **Python Lists** (continued)
  - –List elements are **referenced with subscripting**, with indices beginning at zero

```
x = myList[1]
```
Sets x to 5.8

  - –List elements can be **deleted with** del

```
del myList[1]
```

  - –List Comprehensions – **derived from set notation**

```
[x * x for x in range(6) if x % 3 == 0]
```

range(12) **creates** [0, 1, 2, 3, 4, 5, 6]

Constructed list: [0, 9, 36]

2. What do you mean by pointer? Explain pointer arithmetic with example

Range of values that consists of memory addresses: A *pointer* type variable has a range of values that consists of memory addresses and a special value, *nil*

- **Indirect addressing:** Provide the power of indirect addressing
- **manage dynamic memory:** Provide a way to manage dynamic memory
- **access a location dynamically created :** A pointer can be used to access a location in the area where storage is dynamically created (usually called a *heap*)

```
float stuff[100];
float *p;
p = stuff;
```

`*(p+5)` is equivalent to `stuff[5]` and `p[5]`

`*(p+i)` is equivalent to `stuff[i]` and `p[i]`

3. What are the main problems with pointer? Explain dangling pointer with example

Dangling pointers (dangerous)

A pointer points to a heap-dynamic variable that has been deallocated

•Lost heap-dynamic variable

An allocated heap-dynamic variable that is no longer accessible to the user program (often called *garbage*)

- Pointer `p1` is set to point to a newly created heap-dynamic variable
- Pointer `p1` is later set to point to another newly created heap-dynamic variable
- The process of losing heap-dynamic variables is called *memory leakage*

```
void createDanglingPointer() {
    int* ptr = new int(42); // Dynamically allocate memory for an integer
    cout << "Value: " << *ptr << endl;

    delete ptr; // Deallocate the memory

    // ptr is now a dangling pointer
    cout << "Accessing dangling pointer: " << *ptr << endl; // Undefined behavior
}

int main() {
    createDanglingPointer();
    return 0;
}
```

A memory block is dynamically allocated using `new`, and the pointer `ptr` points to it.

The memory is deallocated using `delete`.

After deletion, `ptr` still holds the address of the now-deallocated memory, making it a dangling pointer.

Accessing `*ptr` after deletion results in undefined behavior.

4. How dangling pointer can be solved?

•Tombstone: extra heap cell that is a pointer to the heap-dynamic variable

–The actual pointer variable points only at tombstones

–When heap-dynamic variable de-allocated, tombstone remains but set to nil

–Costly in time and space

Locks-and-keys: Pointer values are represented as (key, address) pairs

–Heap-dynamic variables are represented as variable plus cell for integer lock value

–When heap-dynamic variable allocated, lock value is created and placed in lock cell and key cell of pointer

5. What do you mean by type checking? Explain coercion rules with example.

Type Checking

- Generalize the concept of operands and operators to include subprograms and assignments
- *Type checking* is the activity of ensuring that the operands of an operator are of compatible types
- A *compatible type* is one that is either legal for the operator, or is allowed under language rules to be implicitly converted, by compiler- generated code, to a legal type
  - This automatic conversion is called a *coercion*.
- A *type error* is the application of an operator to an operand of an inappropriate type

Strong Typing (continued)

- Coercion rules strongly affect strong typing--they can weaken it considerably (C++ versus Ada)
- Although Java has just half the assignment coercions of C++, its strong typing is still far less effective than that of Ada

If all type bindings are static, nearly all type checking can be static

•If type bindings are dynamic, type checking must be dynamic

•A programming language is *strongly typed* if type errors are always detected

•Advantage of strong typing: allows the detection of the misuses of variables that result in type errors

6.Explain different types of operators and precedence and associativity.

Answer:

Arithmetic Expressions: Operators

- A unary operator has one operand
- A binary operator has two operands
- A ternary operator has three operands

Arithmetic Expressions: Operator Precedence Rules

- The *operator precedence rules* for expression evaluation define the order in which "adjacent" operators of different precedence levels are evaluated
- Typical precedence levels
  - parentheses
  - unary operators
  - ** (if the language supports it)
  - *, /
  - +, -

Arithmetic Expressions: Operator Associativity Rule

- The *operator associativity rules* for expression evaluation define the order in which adjacent operators with the same precedence level are evaluated
- Typical associativity rules
  - Left to right, except **, which is right to left
  - Sometimes unary operators associate right to left (e.g., in FORTRAN)
- APL is different; all operators have equal precedence and all operators associate right to left
- Precedence and associativity rules can be overriden with parentheses

7. Explain the conditional expressions with example.

Answer:

Arithmetic Expressions: Conditional Expressions

- Conditional Expressions
  - C-based languages (e.g., C, C++)
  - An example:

```
average = (count == 0)? 0 : sum / count
```

  - Evaluates as if written as follows:

```
if (count == 0)
   average = 0
else
   average = sum /count
```

8. Explain the referential transparency or common subexpression elimination with example.

Answer:

Referential Transparency

- A program has the property of *referential transparency* if any two expressions in the program that have the same value can be substituted for one another anywhere in the program, without affecting the action of the program

```
result1 = (fun(a) + b) / (fun(a) - c);
temp = fun(a);
result2 = (temp + b) / (temp - c);
```

If `fun` has no side effects, `result1 = result2`
Otherwise, not, and referential transparency is violated

9. What do you mean by operator over loading? Explain with example.

Answer:

Overloaded Operators

- Use of an operator for more than one purpose is called *operator overloading*
- Some are common (e.g., + for **`int`** and **`float`**)
- Some are potential trouble (e.g., `*` in C and C++)
  - Loss of compiler error detection (omission of an operand should be a detectable error)
  - Some loss of readability

Overloaded Operators (continued)

- C++, C#, and F# allow user-defined overloaded operators
  - When sensibly used, such operators can be an aid to readability (avoid method calls, expressions appear natural)

10. Explain different types of type conversions with example.

Answer:

Type Conversions

- A *narrowing conversion* is one that converts an object to a type that cannot include all of the values of the original type e.g., **`float`** to **`int`**
- A *widening conversion* is one in which an object is converted to a type that can include at least approximations to all of the values of the original type
  e.g., **`int`** to **`float`**

Type Conversions: Mixed Mode

- A *mixed-mode expression* is one that has operands of different types
- A *coercion* is an implicit type conversion

11. Explain different types of assignment statements (conditional targets and multiple assignments)

Answer:

Assignment Statements: Conditional Targets

- Conditional targets (Perl)

```
($flag ? $total : $subtotal) = 0
```

Which is equivalent to

```
if ($flag){
  $total = 0
} else {
  $subtotal = 0
}
```

Multiple Assignments

- Perl, Ruby, and Lua allow multiple-target multiple-source assignments

```
($first, $second, $third) = (20, 30, 40);
```

Also, the following is legal and performs an interchange:

```
($first, $second) = ($second, $first);
```

12. Write down different types of control statement

Answer:

a) **Conditional control statement:** control flow based on conditions
Example: If. If-else, Switch

b) **Iterative/ Loop control statement:** repeat the blocks of code unit the condition is true.
For loop- executes a block of code a set number of time.
While loop- executes a block of code as long as a condition is true.
Do-While loop- execute a block of code at least once and then continues to execute until the condition is true.

c) **Jump statement:** allows a program to transfer control to a different part of the program,
Break-exits a loop par maturely, continue-skips the current iteration and proceeds the next iteration loop, return- exits a function and optionally returns a value to the caller.

13. Explain nested if-then-else statements with example

Nested if-then-else statements are conditional statements that are nested within another if-then or if-then-else statement. This allows for more complex decision-making by evaluating multiple conditions in a hierarchical manner.

```
if (x > 0) {
    printf("Positive number\n");

    if (x > 5) {
        printf("Greater than 5\n");
    } else {
        printf("Less than or equal to 5\n");
    }
} else {
    printf("Non-positive number\n");
}
```

14. How is a “switch-case” statement converted to “if-then-else” statement? Explain with example

int choice = 2;
switch (choice) {
case 1:
printf("Option 1\n");
break;
case 2:
printf("Option 2\n");
break;
default:
printf("Invalid option\n");
break;
}
 	
if (choice == 1) {
printf("Option 1\n");
} else if (choice == 2) {
printf("Option 2\n");
} else {
printf("Invalid option\n");
}
	A "switch-case" statement can be converted to an equivalent "if-then-else" statement by using multiple "if-else" statements. Each case in the "switch-case" statement corresponds to a condition in the "if-else" statements.

15. Explain different types of loop statements in C/C++ with example.

1. For loop: A for loop allows for executing a block of code repeatedly for a fixed number of iterations.
   Example:

```
for (int i = 0; i < 5; i++) {
 printf("%d ", i) }
```

2. While loop: A while loop repeatedly executes a block of code as long as a given condition remains true. It checks the condition before each iteration.
   Example:

```
int i = 0;
while (i < 5) {
   printf("%d ", i);
   i++; }
```

3. Do-while loop: A do-while loop is similar to a while loop, but it checks the condition after executing the block of code. This guarantees that the code is executed at least once.
   Example:

```
int i = 0;
do {
   printf("%d ", i);
   i++;
} while (i < 5);
```

16.What are the uses of “break” and “continue” statement in C/C++? Explain with example

In C/C++, the "break" and "continue" statements are used to control the flow of execution within loops

Break statement: The "break" statement is used to terminate the execution of a loop

```
for (int i = 1; i <= 5; i++) {
   if (i == 3) {
      break; }
```

```
   printf("%d ", i);
}
```

Continue statement: The "continue" statement is used to skip the current iteration of a loop and move to the next iteration.

```
for (int i = 1; i <= 5; i++) {
   if (i == 3) {
      continue;
   }
   printf("%d ", i);
}
```

17. Define subprogram, subprogram call and parameter profile.

Basic Definitions

- A *subprogram definition* describes the interface to and the actions of the subprogram abstraction
  - In Python, function definitions are executable; in all other languages, they are non-executable
  - In Ruby, function definitions can appear either in or outside of class definitions. If outside, they are methods of `Object`. They can be called without an object, like a function
  - In Lua, all functions are anonymous
- A *subprogram call* is an explicit request that the subprogram be executed
- A *subprogram header* is the first part of the definition, including the name, the kind of subprogram, and the formal parameters
- The *parameter profile* (aka *signature*) of a subprogram is the number, order, and types of its parameters
- The *protocol* is a subprogram's parameter profile and, if it is a function, its return type

18. Differentiate between formal parameter and actual parameter.

- A *formal parameter* is a dummy variable listed in the subprogram header and used in the subprogram
- An *actual parameter* represents a value or address used in the subprogram call statement

Actual/Formal Parameter Correspondence

- **Positional**
  - **The binding of actual parameters to formal parameters is by position: the first actual parameter is bound to the first formal parameter and so forth**
  - **Safe and effective**
- Keyword
  - The name of the formal parameter to which an actual parameter is to be bound is specified with the actual parameter
  - *Advantage*: Parameters can appear in any order, thereby avoiding parameter correspondence errors
  - *Disadvantage*: User must know the formal parameter's names

19. Differentiate between procedure and function.

Procedures and Functions

- There are two categories of subprograms
  - *Procedures* are collection of statements that define parameterized computations
  - *Functions* structurally resemble procedures but are semantically modeled on mathematical functions
    - They are expected to produce no side effects
    - In practice, program functions have side effects

20. Differentiate between pass by value, pass by variable and pass by name

Pass-by-Value (In Mode)

- The value of the actual parameter is used to initialize the corresponding formal parameter
  - Normally implemented by copying
  - Can be implemented by transmitting an access path but not recommended (enforcing write protection is not easy)
  - *Disadvantages* (if by physical move): additional storage is required (stored twice) and the actual move can be costly (for large parameters)
  - *Disadvantages* (if by access path method): must write-protect in the called subprogram and accesses cost more (indirect addressing)

Pass-by-Reference (Inout Mode)

- Pass an access path
- Also called pass-by-sharing
- Advantage: Passing process is efficient (no copying and no duplicated storage)
- Disadvantages
  - Slower accesses (compared to pass-by-value) to formal parameters
  - Potentials for unwanted side effects (collisions)
  - Unwanted aliases (access broadened)

```
fun(total, total);  fun(list[i], list[j]); fun(list[i], i);
```

Pass-by-Name (Inout Mode)

- By textual substitution
- Formals are bound to an access method at the time of the call, but actual binding to a value or address takes place at the time of a reference or assignment
- Allows flexibility in late binding
- Implementation requires that the referencing environment of the caller is passed with the parameter, so the actual parameter address can be calculated

21. Explain call semantics and return semantics.

Implementing "Simple" Subprograms

- Call Semantics:
  - Save the execution status of the caller
  - Pass the parameters
  - Pass the return address to the called
  - Transfer control to the called

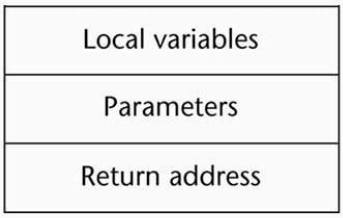
Implementing "Simple" Subprograms (continued)

- Return Semantics:
  - If pass-by-value-result or out mode parameters are used, move the current values of those parameters to their corresponding actual parameters
  - If it is a function, move the functional value to a place the caller can get it
  - Restore the execution status of the caller
  - Transfer control back to the caller
- Required storage:
  - Status information, parameters, return address, return value for functions, temporaries

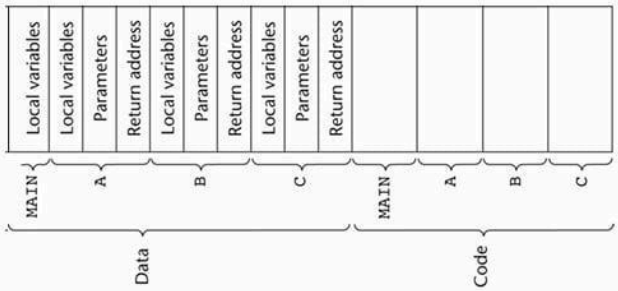
22. What do you mean by activation record? Explain with example.

- **The format, or layout, of the non-code part of an executing subprogram is called an *activation record***
- An *activation record instance* is a concrete example of an activation record (the collection of data for a particular subprogram activation)

An Activation Record for "Simple" Subprograms



Code and Activation Records of a Program with "Simple" Subprograms



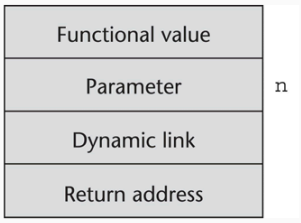
23. Explain the activation record of recursive factorial function

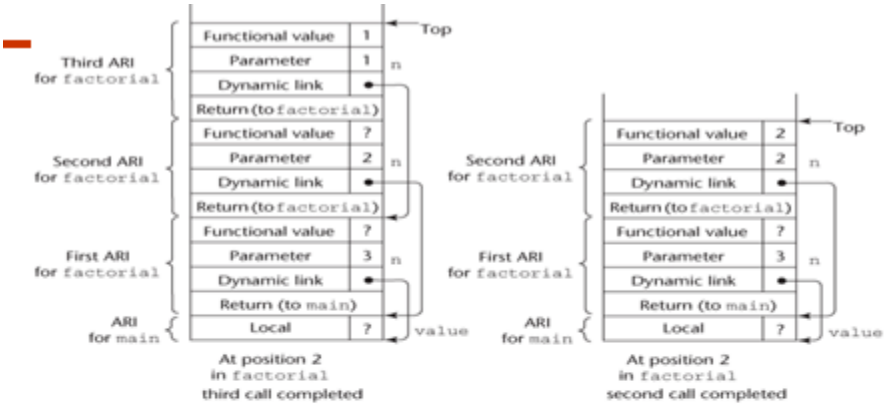
An Example With Recursion

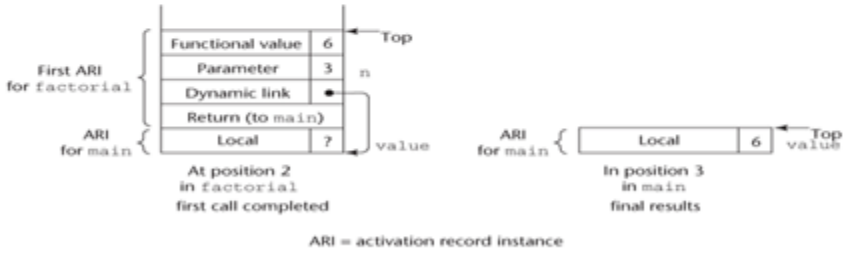
- The activation record used in the previous example supports recursion

```
int factorial (int n) {
    <------------------------------1
  if (n <= 1) return 1;
  else return (n * factorial(n - 1));
    <------------------------------2
}
void main() {
  int value;
  value = factorial(3);
    <------------------------------3
}
```

Activation Record for `factorial`

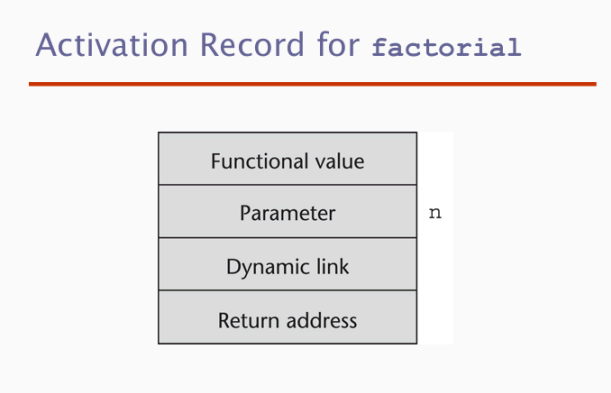






ARI = activation record instance

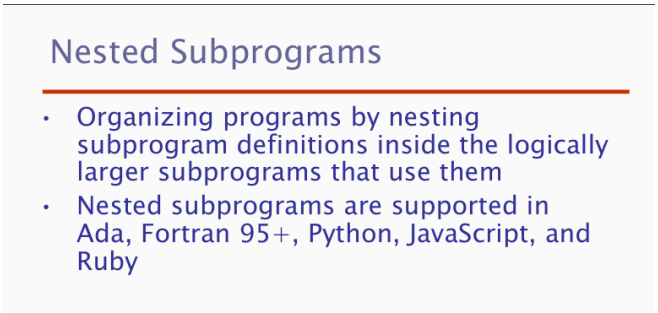
24. Explain the activation record of recursive Fibonacci series function.




```
fibonacci(3)
 ├─ fibonacci(2)
 │    ├─ fibonacci(1) --> 1 (base case)
 │    └─ fibonacci(0) --> 0 (base case)
 └─ fibonacci(1) --> 1 (base case)
```

```python
def fibonacci(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

25. Explain the nested subprogram with an example



ExampleSome non-C-based static-scoped languages (e.g., Fortran 95+, Ada, Python, JavaScript, Ruby, and Lua) use **stack-dynamic local variables and allow subprograms to be nested**
All variables that can be non-locally accessed reside in some activation record instance in the stack

```
#include <stdio.h>

void outerFunction() {
   printf("This is the outer function.\n");

   void innerFunction() {
      printf("This is the inner function.\n");
   }

   innerFunction();
}
```

```
int main() {
   outerFunction();


   return 0;
}
```

26. What do you mean by abstract data type? What are the conditions of ADT?

•**An *abstraction* is a view or representation of an entity that includes only the most significant attributes.An *abstract data type* is a user-defined data type that satisfies the following two conditions:**

1)**The representation of objects of the type is hidden from the program units that use these objects.**

2)**The declarations of the type and the protocols of the operations on objects of the type are contained in a single syntactic unit.**

27. What are the advantages of ADT?

–Reliability--by hiding the data representations, user code cannot directly depend on the representation, allowing the representation to be changed without affecting user code.

–Reduces the range of code and variables of which the programmer must be aware

–Name conflicts are less likely

Advantages of the second condition

–Provides a method of program organization

–Aids modifiability (everything associated with a data structure is together)

–Separate compilation

28. What do you mean by encapsulation? How can information be hidden in C/C++? Explain with example

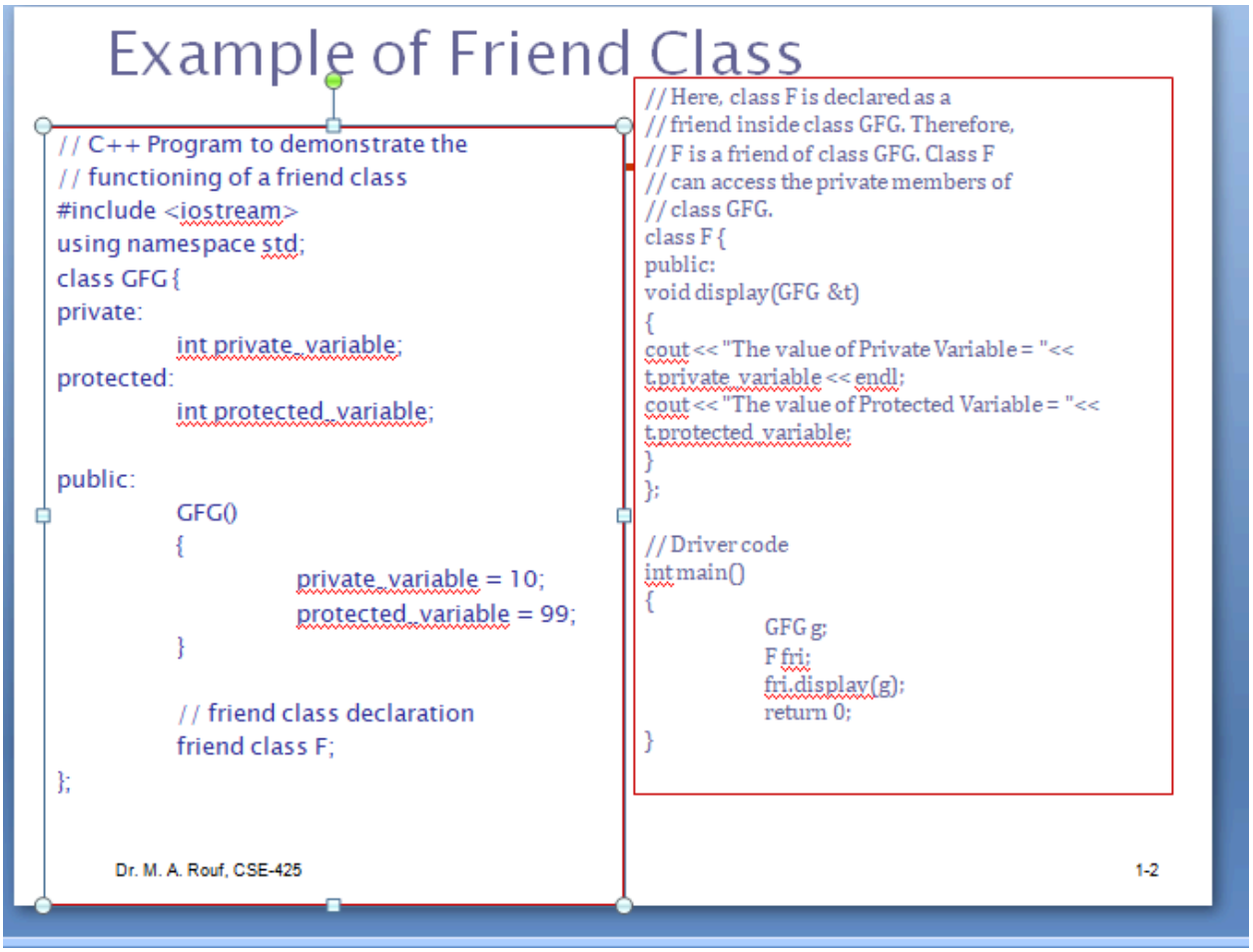
Encapsulation is a programming principle that involves packaging **variables, methods or function**s into a single unit.

- Large programs have two special needs:
  - Some means of organization, other than simply division into subprograms
  - Some means of partial compilation (compilation units that are smaller than the whole program)
- Obvious solution: a grouping of **subprograms that are logically related into a unit t**hat can be separately compiled (compilation units)
- Such collections are called *encapsulation*

Private clause for **hidden entities**

Public clause for **interface entities**

Protected clause for **inheritance**



29. What are the uses of constructor and destructor?

•Constructors:

–Functions to initialize the data members of instances (they *do not* create the objects)

–May also allocate storage if part of the object is heap-dynamic

–Can include parameters to provide parameterization of the objects

–Implicitly called when an instance is created

–Can be explicitly called

–Name is the same as the class name

•Destructors

–Functions to cleanup after an instance is destroyed; usually just to reclaim heap storage

–Implicitly called when the object's lifetime ends

–Can be explicitly called

–Name is the class name, preceded by a tilde (~)

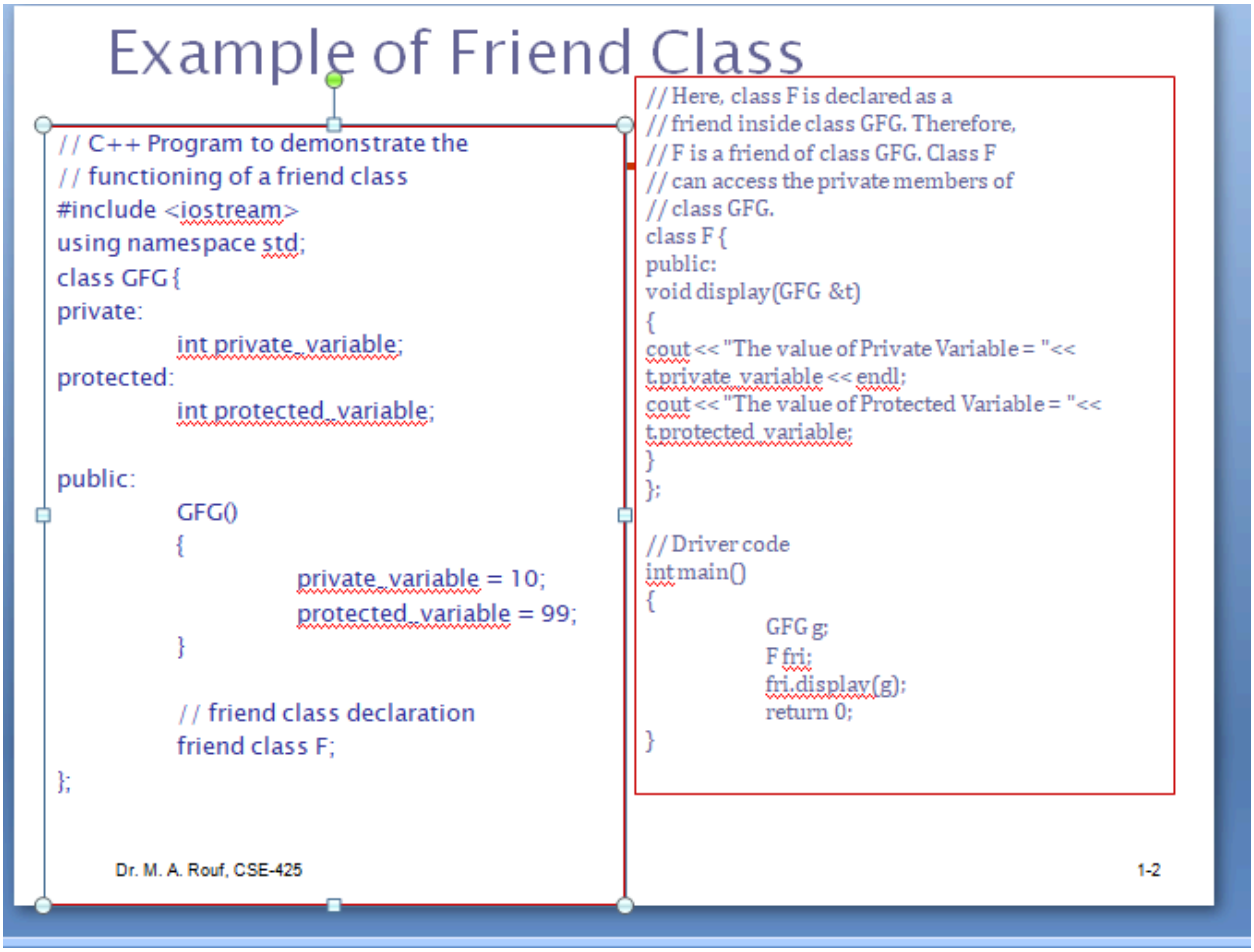
30. Write down the Stack ADT in C++?

```
class Stack {
private:
int *stackPtr, maxLen, topPtr;
public:
Stack() { // a constructor
stackPtr = new int [100];
maxLen = 99;
topPtr = -1;
};
~Stack () {delete [] stackPtr;};
void push (int number) {
if (topSub == maxLen)
cerr << "Error in push - stack is full\n";
else stackPtr[++topSub] = number;
};
void pop () {…};
int top () {…};
int empty () {…};
```

31. What do you mean by friend function/class?

- Friend functions or classes - to provide access to private members to some unrelated units or functions

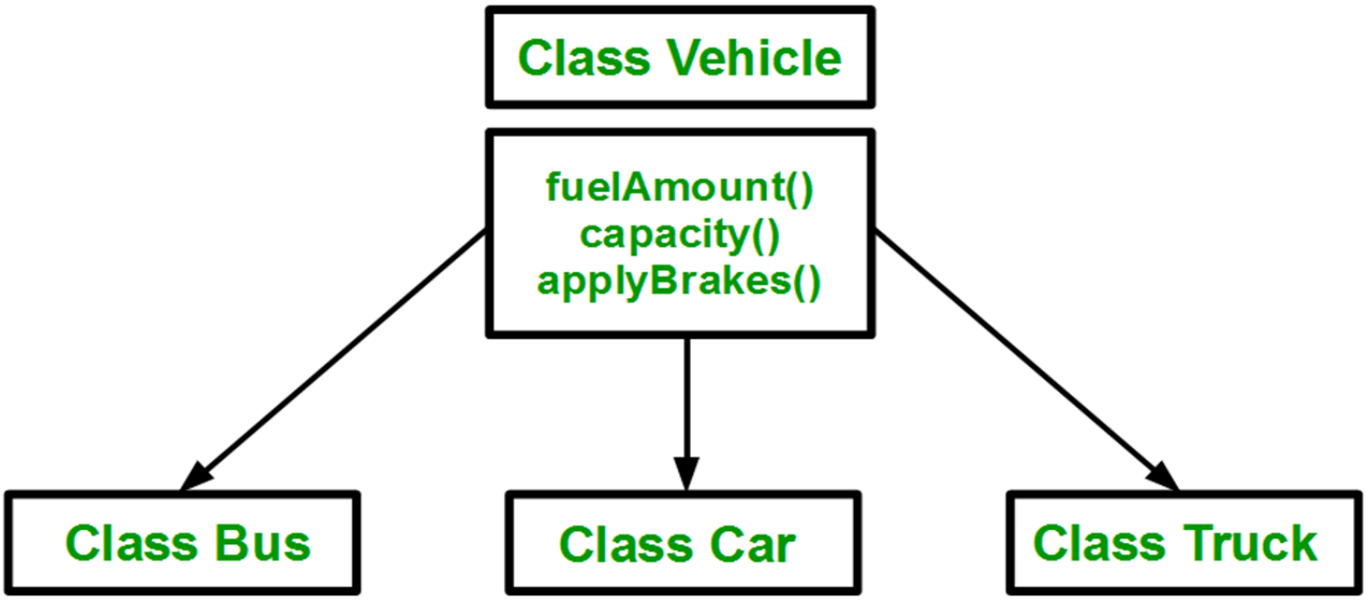
–Necessary in C++



32. What do you mean by inheritance in C++?

Inheritance **allows new classes defined in terms of existing ones**, by allowing them to inherit common parts

Inheritance addresses both of the above concerns--reuse ADTs after minor changes and define classes in a hierarchy


Class Vehicle
fuelAmount()
capacity()
applyBrakes()
Class Bus
Class Car
Class Truck

```
class vehicle
{
... .. ...
};
class Bus: public vehicle
{
... .. ...
};
class Car: public vehicle
{
... ... ...
};
class Truck : public vehicle
{
}
```

33. What do you mean by abstract method/class?

•An *abstract method* is one that does not **include a definition (it only defines a protocol)**

•An *abstract class* is one that includes a**t least one virtual method**

•An abstract **class cannot be instantiated**

Any class inheriting the abstract class must provide an implementation for the abstract method**.**

34. Differentiate between subclass and superclass?

•Three ways a class can differ from its parent:

1**. Access Restrictions of variables and methods:**The parent class can define **some of its variables or methods** to have private access, which means **they will not be visible in the subclass**

2. The subclass can **add variables and/or methods to those** inherited from the parent

3. The subclass **can modify the behavior of one or more of its inherited methods.**

Aspect	Superclass (Parent Class)	Subclass (Child Class)
Definition	A base class that provides common variables and methods.	A class that inherits from the superclass.
Access	Contains variables and methods that may have private, protected, or public access.	Can access **public** and **protected** members of the superclass.
Behavior	Defines default behavior and common attributes.	Can **add**, **extend**, or **override** behavior.
Inheritance	Does **not inherit** from any other class (if it's the top-level class).	Inherits variables and methods from the superclass.
Customization	Provides general functionality.	Can add new methods/variables or modify existing ones.
Example	`Animal` (general class).	`Dog` (specific class inheriting from `Animal`).

35. What do you mean by single and multiple inheritance? Explain with example

•**Single and Multiple inheritance** allows a new class to inherit from two or more classes where single inheritance Inherits from a single parent class.

•**Disadvantages of multiple inheritance:**

–**Language and implementation complexity** (in part due to name collisions) multiple inheritance ,where in single inheritance its Simple and easy to understand.

–**Potential inefficiency** - dynamic binding costs more with multiple inheritance (but not much)

•Advantage:

–Sometimes it is quite convenient and valuable

Aspect	Single Inheritance	Multiple Inheritance
Definition	Inherits from a single parent class.	Inherits from two or more parent classes.
Complexity	Simple and easy to understand.	More complex due to ambiguity and collisions.
Name Collision	No chance of name conflicts.	Can cause conflicts if methods have the same name.
Efficiency	More efficient and faster.	Slightly less efficient due to dynamic binding.
Example	`Dog` inherits from `Animal`.	`Horse` inherits from `Animal` and `Vehicle`.

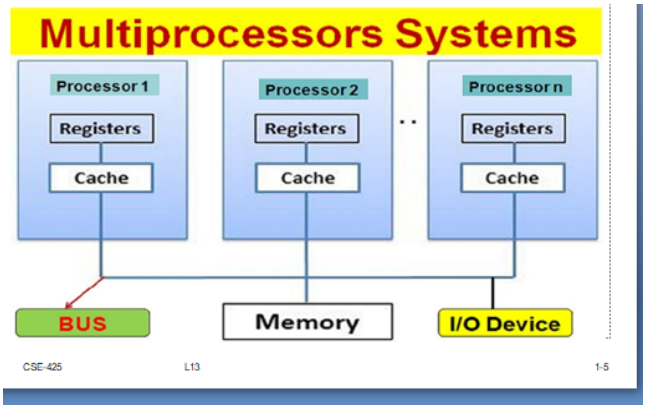
36. Differentiate between operator and functional overloading? Explain with example.

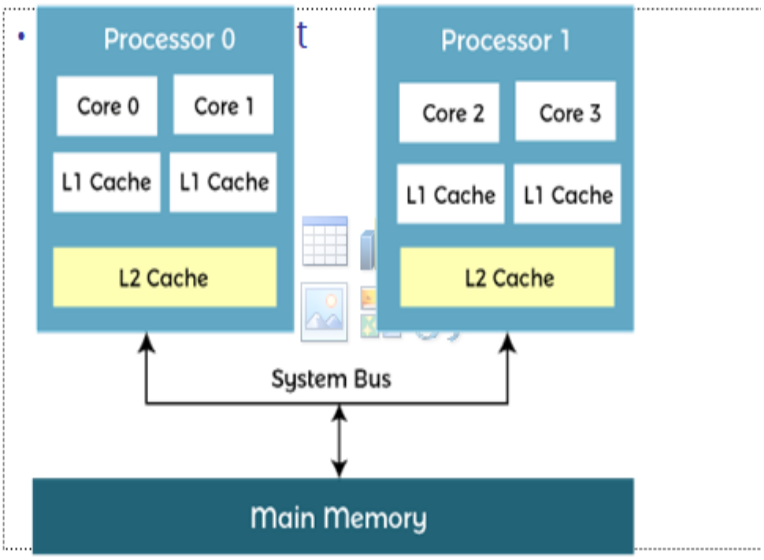
```python
class Number:
    def __init__ (self, value):
        self.value = value

    # Overload the + operator
    def __add__(self, other):
        return Number(self.value + other.value)

    def __str__(self):
        return str(self.value)

# Create objects
num1 = Number(10)
num2 = Number(20)

# Use overloaded + operator
result = num1 + num2

print("Sum:", result)  # Output: Sum: 30
```

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

// Function Overloading
void add(int a, int b) {
    cout << "Sum (int): " << a + b << endl;
}

void add(double a, double b) {
    cout << "Sum (double): " << a + b << endl;
}

int main() {
    add(10, 5);     // Calls int version
    add(3.5, 2.5);  // Calls double version
    return 0;
}
```

Aspect	Operator Overloading	Function Overloading
Definition	Redefining the behavior of an **operator** for user-defined data types.	Defining **multiple functions** with the same name but different parameters.
Purpose	Allows operators like `+`, `-`, `*` to work with objects.	Allows the same function name to perform different tasks based on input types.
Syntax	Uses the keyword `operator`.	Same function name but with different signatures (parameters).
Example Operators	`+`, `-`, `*`, `==`, `++`, etc.	Any user-defined functions.
Example Use Case	Combine two objects using `+` or compare objects.	Handle different data types or arguments.

38. Differentiate between multicore and multi-processing system





Difference Between Multicore and Multi-Processing System

---

1. Multicore System

- **A multicore system has a single processor with multiple cores (processing units) integrated into it.**
- **Each core can execute tasks independently and in parallel, improving performance.**

Key Features:

- **Cores share the same cache memory and resources (L1, L2 cache).**
- **Examples: Dual-core, Quad-core processors.Diagram Explanation (Multicore):**
- **Processor 0 and Processor 1 have multiple cores (Core 0, Core 1, Core 2, Core 3).**

- **Each core has its own cache (L1) but may share a larger L2 cache.**
- **Connected via the system bus to main memory.**

---

2. Multi-Processing System

- **A multi-processing system uses multiple independent processors (CPUs), each with its own resources.**
- **Processors work together to execute multiple processes simultaneously.**

Key Features:

- **Processors may or may not share memory (depending on architecture).**
- **Increases system throughput (parallel processing).**
- **Example: Servers or clusters with multiple CPUs.**

Diagram Explanation (Multi-Processing):

- **Multiple processors (Processor 1, Processor 2, Processor 3) each with their own registers and cache.**
- **All processors are connected to a common bus, memory, and I/O devices.**

39. Differentiate between single cycle processor and pipeline processor

Single-cycle processors:

- Single-cycle processors execute each instruction in a single clock cycle.
- Each instruction goes through all stages of the processor (fetch, decode, execute, memory access, write back) in a single cycle.
- Simple and easy to understand, but can be slower for complex instructions.

Pipeline processors:

- Pipeline processors divide the execution of instructions into multiple stages and process multiple instructions simultaneously.
- Each stage of the pipeline performs a specific operation on an instruction.
- Instructions are fetched, decoded, executed, and written back in overlapping fashion.

40. Differentiate between logical concurrency and physical concurrency

–***Physical concurrency*** **-**

1.Multiple independent processors

2. (multiple threads of control)

–***Logical concurrency*** **-**

1.The appearance of physical concurrency is presented by **time-sharing one processo**r

2.(software can be designed as if there were multiple threads of control)

41.Differentiate between process and thread.

Process

1.Heavyweight tasks execute in their own address space which is called **Process**

2.Process means any program is in execution.

3.The process takes more time to terminate.

Thread

1.Thread means a segment of a process

2.•*Lightweight tasks* all run in the same address space – more efficient which is called Thread

3.The process takes less time to terminate.

42. What do you mean by task synchronization? Explain different types of task synchronization

•A mechanism that controls the order in which tasks execute

•*Cooperation*: Task A must wait for task B to complete some specific activity before task A can continue its execution,

e.g., the producer-consumer problem

•*Competition*: Two or more tasks must use some resources that cannot be simultaneously used, e.g., a shared counter

–Competition is usually provided by mutually exclusive access (approaches are discussed later)

43. What do you mean by task scheduling? Explain different states of the task

Synchronization requires a mechanism for delaying task execution. The program which is responsible for controlling the task execution is

called scheduler. Scheduler maps the task execution into available processor

- ***New*** - created but not yet started
- ***Ready*** - ready to run but not currently running (no available processor)
- ***Running -*** Executing
- ***Blocked*** - has been running, but cannot now continue (usually waiting for some event to occur)
- ***Dead*** - no longer active in any sense

44. Explain different design issues of concurrency.

- 1.Competition and cooperation synchronization*
- 2. task scheduling:Controlling task scheduling
- 3.application influence:How can an application influence task scheduling
- 4. tasks start and end :How and when tasks start and end execution
- 5. tasks create:How and when are tasks created

* The most important issue

45. Explain the deadlock in task/process scheduling.

- ***Liveness*** is a characteristic that a program unit may or may not have

- In ***sequential code***, it means the unit will eventually complete its execution
- In a ***concurrent environment,*** a task can easily lose its liveness
- If all tasks in a concurrent environment lose their liveness, it is called *deadlock*

46. Explain Java Threads with an example

- The concurrent units in Java are methods named `run`

–A `run` method code can be in concurrent execution with other such methods

–The process in which the `run` methods execute is called a *thread*

```
class myThread extends Thread
  public void run () {…}
}
…
Thread myTh = new MyThread ();
myTh.start();
```