What is Exception Handling?

Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

Advantage of Exception Handling

The core advantage of exception handling is **to maintain the normal flow of the application**. An exception normally disrupts the normal flow of the application; that is why we need to handle exceptions. Let's consider a scenario:

```
statement 1;
statement 2;
statement 3;
statement 4;
statement 5;//exception occurs
statement 6;
statement 7;
statement 8;
statement 9;
statement 10;
```

Suppose there are 10 statements in program and an exception occurs at statement 5; the rest of the code will not be executed, i.e., statements 6 to 10 will not be executed. However, when we perform exception handling, the rest of the statements will be executed. That is why we use exception handling.

One of the advantages of C++ over C is Exception Handling. Exceptions are runtime anomalies or abnormal conditions that a program encounters during its execution. There are two types of exceptions: a)Synchronous, b)Asynchronous (i.e., exceptions which are beyond the program's control, such as disc failure, keyboard interrupts etc.).

C++ provides the following specialized keywords for this purpose:

try: Represents a block of code that can throw an exception.

catch: Represents a block of code that is executed when a particular exception is thrown.

throw: Used to throw an exception. Also used to list the exceptions that a function throws but doesn't handle itself.

Why Exception Handling?

The following are the main advantages of exception handling over traditional error handling:

***1)** Separation of Error Handling code from Normal Code:* In traditional error handling codes, there are always if-else conditions to handle errors. These conditions and the code to handle errors get mixed up with the normal flow. This makes the code less readable and maintainable. With try/catch blocks, the code for error handling becomes separate from the normal flow.
***2)** Functions/Methods can handle only the exceptions they choose:* A function can throw many exceptions, but may choose to handle some of them. The other exceptions, which are thrown but not caught, can be handled by the caller. If the caller chooses not to catch them, then the exceptions are handled by the caller of the caller.
In C++, a function can specify the exceptions that it throws using the throw keyword. The caller of this function must handle the exception in some way (either by specifying it again or catching it).
***3)** Grouping of Error Types:* In C++, both basic types and objects can be thrown as exceptions. We can create a hierarchy of exception objects, group exceptions in namespaces or classes and categorize them according to their types.

C++ Exceptions:
When executing C++ code, different errors can occur: coding errors made by the programmer, errors due to wrong input, or other unforeseeable things.

When an error occurs, C++ will normally stop and generate an error message. The technical term for this is: C++ will throw an exception (error).

C++ try and catch:
Exception handling in C++ consists of three keywords: try, throw and catch:

The try statement allows you to define a block of code to be tested for errors while it is being executed.

The throw keyword throws an exception when a problem is detected, which lets us create a custom error.

The catch statement allows you to define a block of code to be executed if an error occurs in the try block.

The try and catch keywords come in pairs:

We use the try block to test some code: If the value of a variable “age” is less than 18, we will throw an exception, and handle it in our catch block.

In the catch block, we catch the error if it occurs and do something about it. The catch statement takes a single parameter. So, if the value of age is 15 and that’s why we are throwing an exception of type int in the try block (age), we can pass “int myNum” as the parameter to the catch statement, where the variable “myNum” is used to output the value of age.

If no error occurs (e.g. if age is 20 instead of 15, meaning it will be greater than 18), the catch block is skipped.

Exception Handling in C++

1) The following is a simple example to show exception handling in C++. The output of the program explains the flow of execution of try/catch blocks.

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

int main()
{
int x = -1;

// Some code
cout << "Before try \n";
try {
    cout << "Inside try \n";
    if (x < 0)
    {
            throw x;
            cout << "After throw (Never executed) \n";
    }
}
catch (int x ) {
    cout << "Exception Caught \n";
}

cout << "After catch (Will be executed) \n";
return 0;
}
```

Output:

```
Before try
Inside try
Exception Caught
After catch (Will be executed)
```

2) There is a special catch block called the ‘catch all’ block, written as catch(…), that can be used to catch all types of exceptions. For example, in the following program, an int is thrown as an exception, but there is no catch block for int, so the catch(…) block will be executed.

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

int main()
{
        try {
        throw 10;
        }
        catch (char *excp) {
                cout << "Caught " << excp;
        }
        catch (...) {
                cout << "Default Exception\n";
        }
        return 0;
}
```

Output:

```
Default Exception
```

3) Implicit type conversion doesn't happen for primitive types. For example, in the following program, 'a' is not implicitly converted to int.

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

int main()
{
        try {
        throw 'a';
        }
        catch (int x) {
                cout << "Caught " << x;
        }
        catch (...) {
                cout << "Default Exception\n";
        }
        return 0;
}
```

Output:

```
Default Exception
```

4) If an exception is thrown and not caught anywhere, the program terminates abnormally. For example, in the following program, a char is thrown, but there is no catch block to catch the char.

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

int main()
{
        try {
        throw 'a';
        }
        catch (int x) {
```

```
        cout << "Caught ";
    }
    return 0;
}
```

Output:

```
terminate called after throwing an instance of 'char'

This application has requested the Runtime to terminate it in an
unusual way. Please contact the application's support team for
more information.
```

We can change this abnormal termination behavior by writing our own unexpected function.
5) A derived class exception should be caught before a base class exception. See this for more details.
6) Like Java, the C++ library has a standard exception class which is the base class for all standard exceptions. All objects thrown by the components of the standard library are derived from this class. Therefore, all standard exceptions can be caught by catching this type
7) Unlike Java, in C++, all exceptions are unchecked, i.e., the compiler doesn’t check whether an exception is caught or not (See this for details). So, it is not necessary to specify all uncaught exceptions in a function declaration. Although it’s a recommended practice to do so. For example, the following program compiles fine, but ideally the signature of fun() should list the unchecked exceptions.

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

// This function signature is fine by the compiler, but not recommended.
// Ideally, the function should specify all uncaught exceptions and function
// signature should be "void fun(int *ptr, int x) throw (int *, int)"
void fun(int *ptr, int x)
{
    if (ptr == NULL)
        throw ptr;
    if (x == 0)
        throw x;
    /* Some functionality */
}

int main()
{
    try {
    fun(NULL, 0);
    }
    catch(...) {
        cout << "Caught exception from fun()";
        }
        return 0;
}
```

Output:

```
Caught exception from fun()
```

Module 1: Introduction to Programming Paradigms & Core Language Design Issues

Names, Bindings, Type Checking, and Scopes

Topics

- Introduction
- Names
- Variables
- The Concept of Binding
- Type Checking
- Strong Typing
- Scope
- Scope and Lifetime
- Referencing Environments
- Named Constants
- Variable Initialization

Names, Bindings, Type Checking, and Scopes

Introduction

- Imperative languages are abstractions of von Neumann architecture
  - Memory: stores both instructions and data
  - Processor: provides operations for modifying the contents of memory
- Variables characterized by attributes
  - Type: to design, must consider scope, lifetime, type checking, initialization, and type compatibility

Names

Design issues for names:

- Maximum length?
- Are connector characters allowed?
- Are names case sensitive?
- Are special words reserved words or keywords?

Name Forms

- A **name** is a string of characters used to identify some entity in a program.
- If too short, they cannot be connotative
- Language examples:
  - FORTRAN I: maximum 6
  - COBOL: maximum 30
  - FORTRAN 90 and ANSI C: maximum 31
  - Ada and Java: **no limit**, and all are significant
  - C++: **no limit**, but implementers often impose a length limitation because they do not want the **symbol table** in which identifiers are stored during compilation to be too large and also to simplify the maintenance of that table.
- Names in most programming languages have the same form: a letter followed by a string consisting of letters, digits, and (_).
- Although the use of the _ was widely used in the 70s and 80s, that practice is far less popular.
- C-based languages (C, C++, Java, and C#), replaced the _ by the “camel” notation, as in myStack.

- Prior to Fortran 90, the following two names are equivalent:

```
Sum Of Salaries  // names could have embedded spaces
SumOfSalaries    // which were ignored
```

- Case sensitivity
  - Disadvantage: readability (names that look alike are different)
    - worse in C++ and Java because predefined names are mixed case (e.g. **IndexOutOfBoundsException**)
    - In C, however, exclusive use of lowercase for names.
  - C, C++, and Java names are case sensitive ➔ rose, Rose, ROSE are distinct names “What about Readability”

Special words

- An aid to readability; used to delimit or separate statement clauses
- A **keyword** is a word that is special only in certain contexts.
- **Disadvantage**: poor readability. Compilers and users must recognize the difference.
- A **reserved word** is a special word that **cannot** be used as a user-defined name.
- As a language design choice, reserved words are **better** than keywords.
- Ex: In Fortran, one could have the statements

```
Integer Real     // keyword "Integer" and variable "Real"
Real Integer     // keyword "Real" and variable "Integer"
```

Variables

- A variable is an abstraction of a memory cell(s).
- Variables can be characterized as a sextuple of attributes:
    - Name
    - Address
    - Value
    - Type
    - Lifetime
    - Scope

Name

- Not all variables have names: **Anonymous**, heap-dynamic variables

Address

- The memory address with which it is associated
- A variable name may have different addresses at different places and at different times during execution.

  // `sum in sub1 and sub2`

- A variable may have **different** addresses at **different** times during execution. If a subprogram has a local var that is allocated from the run time **stack** when the subprogram is called, different calls may result in that var having different addresses.

  // `sum in sub1`

- The address of a variable is sometimes called its ***l*-value** because that is what is required when a variable appears in the **left** side of an assignment statement.

Aliases

- If **two variable** names can be used to access **the same memory location**, they are called **aliases**
- Aliases are created via **pointers**, **reference variables**, C and C++ **unions.**
- Aliases are harmful to readability (program readers must remember **all** of them)

Type

- Determines the **range** of values of variables and the set of **operations** that are defined for values of that type; in the case of floating point, type also determines the precision.
- For example, the int type in Java specifies a value range of -2147483648 to 2147483647, and arithmetic operations for addition, subtraction, multiplication, division, and modulus.

Value

- The value of a variable is the contents of the memory cell or cells associated with the variable.
- Abstract memory cell - the physical cell or collection of cells associated with a variable.
- A variable’s value is sometimes called its ***r*-value** because that is what is required when a variable appears in the **right** side of an assignment statement.

The Concept of Binding

- The ***l*-value** of a variable is its **address**.
- The ***r*-value** of a variable is its **value**.
- A **binding** is an association, such as between an attribute and an entity, or between an operation and a symbol.
- **Binding time** is the time at which a binding takes place.
- Possible binding times:
    - *Language design time*: bind operator symbols to operations.
        - For example, the asterisk symbol (*) is bound to the multiplication operation.
    - *Language implementation time*:
        - A data type such as **int** in C is bound to a **range** of possible values.
    - *Compile time*: bind a variable to a **particular data type** at compile time.
    - *Load time*: bind a variable to a **memory cell** (ex. C **static** variables)
    - *Runtime*: bind a **nonstatic** local variable to a memory cell.

Binding of Attributes to Variables

- A binding is **static** if it first occurs **before** run time and remains unchanged throughout program execution.
- A binding is **dynamic** if it first occurs **during** execution or can change during execution of the program.

Type Bindings

- If static, the type may be specified by either an **explicit** or an **implicit** declaration.

Variable Declarations

- An **explicit declaration** is a program statement used for declaring the types of variables.
- An **implicit declaration** is a **default** mechanism for specifying types of variables (the first appearance of the variable in the program.)
- Both explicit and implicit declarations create static bindings to types.
- FORTRAN, PL/I, BASIC, and Perl provide implicit declarations.
- EX:
    - In **Fortran**, an identifier that appears in a program that is not explicitly declared is implicitly declared according to the following convention:
      I, J, K, L, M, or N or their lowercase versions is **implicitly** declared to be Integer type; otherwise, it is implicitly declared as Real type.
    - **Advantage**: writability.

  - **Disadvantage**: reliability suffers because they prevent the compilation process from detecting some typographical and programming errors.
  - In Fortran, vars that are accidentally left undeclared are given default types and unexpected attributes, which could cause subtle errors that, are difficult to diagnose.
- Less trouble with **Perl**: Names that begin with $ is a scalar, if a name begins with @ it is an array, if it begins with %, it is a hash structure.
  - In this scenario, the names `@apple` and `%apple` are unrelated.
- In **C and C++**, one must distinguish between declarations and definitions.
  - **Declarations** specify types and other attributes but do **no** cause allocation of storage. Provides the type of a var defined external to a **function** that is used in the function.
  - **Definitions** specify attributes and cause storage allocation.

Dynamic Type Binding (JavaScript and PHP)

- Specified through an assignment statement
- Ex, JavaScript

```
list = [2, 4.33, 6, 8];     ➔ single-dimensioned array
list = 47;                  ➔ scalar variable
```

- Advantage: **flexibility** (generic program units)
- Disadvantages:
    - **High cost** (dynamic type checking and interpretation)
        - Dynamic type bindings must be implemented using pure interpreter **not** compilers.
        - Pure interpretation typically takes at least **ten times** as long as to execute equivalent machine code.
    - **Type error detection by the compiler is difficult** because **any** variable can be assigned a value of **any** type.
        - Incorrect types of right sides of assignments are not detected as errors; rather, the type of the left side is simply changed to the incorrect type.
        - Ex:

```
i, x  ➔ Integer
y     ➔ floating-point array
i = x ➔ what the user meant to type
i = y ➔ what the user typed instead
```

        - **No error** is detected by the compiler or run-time system. i is simply changed to a floating-point array type. Hence, the result is erroneous. In a static type binding language, the compiler would detect the error and the program would not get to execution.

Type Inference (ML, Miranda, and Haskell)

- Rather than by assignment statement, types are determined from the context of the reference.
- Ex:

```
fun circumf(r) = 3.14159 * r * r;
      The argument and functional value are inferred to
      be real.

fun times10(x) = 10 * x;
      The argument and functional value are inferred to
      be int.
```

Storage Bindings & Lifetime

- **Allocation** - getting a cell from some pool of available cells.
- **Deallocation** - putting a cell back into the pool.
- The **lifetime** of a variable is the time during which it is bound to a particular memory cell. So the lifetime of a var begins when it is bound to a specific cell and ends when it is unbound from that cell.
- Categories of variables by lifetimes: **static**, **stack-dynamic**, **explicit heap-dynamic**, and **implicit heap-dynamic**

Static Variables:

- bound to memory cells before execution begins and remains bound to the same memory cell throughout execution.
- e.g. all FORTRAN 77 variables, C static variables.
- **Advantages**:
    - **Efficiency**: (direct addressing): All addressing of static vars can be direct. No run-time overhead is incurred for allocating and deallocating vars.
    - **History-sensitive**: have vars retain their values between separate executions of the subprogram.
- **Disadvantage**:
    - Storage **cannot** be shared among variables.
    - Ex: if two large arrays are used by two subprograms, which are never active at the same time, they cannot share the same storage for their arrays.

Stack-dynamic Variables:

- Storage bindings are created for variables when their declaration statements are elaborated, but whose types are statically bound.
- Elaboration of such a declaration refers to the storage allocation and binding process indicated by the declaration, which takes place when execution reaches the code to which the declaration is attached.
- Ex:
    - The variable declarations that appear at the beginning of a **Java method** are elaborated when the method is invoked and the variables defined by those declarations are deallocated when the method completes its execution.
- Stack-dynamic variables are allocated from the **run-time stack**.
- If scalar, all attributes except address are statically bound.
- Ex:
    - Local variables in C subprograms and Java methods.
- **Advantages**:
    - Allows recursion: each active copy of the recursive subprogram has its own version of the local variables.
    - In the absence of recursion it conserves storage b/c all subprograms share the same memory space for their locals.

- **Disadvantages**:
    - Overhead of allocation and deallocation.
    - Subprograms cannot be history sensitive.
    - Inefficient references (indirect addressing) is required b/c the place in the stack where a particular var will reside can only be determined during execution.
- In Java, C++, and C#, variables defined in **methods** are by **default** stack-dynamic.

Explicit Heap-dynamic Variables:

- Nameless memory cells that are allocated and deallocated by explicit directives “run-time instructions”, specified by the programmer, which take effect during execution.
- These vars, which are allocated from and deallocated to the heap, can only be referenced through pointers or reference variables.
- The **heap** is a collection of storage cells whose organization is highly disorganized b/c of the unpredictability of its use.
- e.g. dynamic objects in C++ (via **new** and **delete**)

```
int *intnode;
…
intnode = new int; // allocates an int cell
…
delete intnode;  // deallocates the cell to which
                 // intnode points
```

- An explicit heap-dynamic variable of int type is created by the new operator.
- This operator can be referenced through the pointer, intnode.
- The var is deallocated by the **delete** operator.
- Java, all data except the primitive scalars are **objects**.
- Java objects are explicitly heap-dynamic and are accessed through **reference variables**.
- Java uses **implicit garbage collection**.
- Explicit heap-dynamic vars are used for dynamic structures, such as linked lists and trees that need to grow and shrink during execution.
- **Advantage**:
    - Provides for dynamic storage management.
- **Disadvantage**:
    - Inefficient “Cost of allocation and deallocation” and unreliable “difficulty of using pointer and reference variables correctly”

Implicit Heap-dynamic Variables:

- Bound to heap storage only when they are assigned value. Allocation and deallocation caused by **assignment statements**.

- All their attributes are bound every time they are assigned.
    - e.g. all variables in APL; all strings and arrays in Perl and JavaScript.
- **Advantage**:
  - Flexibility allowing generic code to be written.
- **Disadvantages**:
    - Inefficient, because all attributes are dynamic “run-time.”
    - Loss of error detection by the compiler.

Type Checking

- **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**.
- Ex: an **int** var and a **float** var are added in Java, the value of the **int** var is coerced to **float** and a floating-point is performed.
- A **type error** is the application of an operator to an operand of an inappropriate type.
- Ex: in C, if an **int** value was passed to a function that expected a **float** value, a type error would occur (compilers **didn't** check the types of parameters)
- If all type bindings are static, nearly all type checking can be static.
- If type bindings are dynamic, type checking must be dynamic and done at run-time.

Strong Typing

- A programming language is strongly typed if type errors are **always** detected. It requires that the types of all operands can be determined, either at compile time or run time.
- Advantage of strong typing: allows the detection of the misuses of variables that result in type errors.
- **Java and C#** are strongly typed. Types can be explicitly cast, which would result in type error. However, there are no implicit ways type errors can go undetected.
- The coercion rules of a language have an important effect on the value of type checking.
- Coercion results in a loss of part of the reason of strong typing – error detection.
- Ex:

```
int a, b;
float d;
a + d;    // the programmer meant a + b, however
```

– The compiler would not detect this error. Var `a` would be coerced to **float**.

Scope

- The scope of a var is the range of statements in which the var is visible.
- A var is **visible** in a statement if it can be referenced in that statement.
- **Local var** is local in a program unit or block if it is declared there.
- **Non-local var** of a program unit or block are those that are visible within the program unit or block but are not declared there.

Static Scope

- Binding names to non-local vars is called **static scoping**.
- There are two categories of static scoped languages:
    - Nested Subprograms.
    - Subprograms that can't be nested.
- Ada, and JavaScript allow **nested** subprogram, but the C-based languages do not.
- When a compiler for static-scoped language finds a reference to a var, the attributes of the var are determined by finding the statement in which it was declared.
- Ex: Suppose a reference is made to a var **x** in subprogram **Sub1**. The correct declaration is found by first searching the declarations of subprogram Sub1.
- If no declaration is found for the var there, the search continues in the declarations of the subprogram that declared subprogram Sub1, which is called its **static parent**.
- If a declaration of x is not found there, the search continues to the next larger enclosing unit (the unit that declared Sub1's parent), and so forth, until a declaration for x is found or the largest unit's declarations have been searched without success. ➔ an undeclared var error has been detected.
- The static parent of subprogram Sub1, and its static parent, and so forth up to and including the main program, are called the static **ancestors** of Sub1.

Ex: Ada procedure:

```
Procedure Big is
   X : Integer;
   Procedure Sub1 is
      Begin      -- of Sub1
      …X…
      end;       -- of Sub1
   Procedure Sub2 is
      X Integer;
      Begin      -- of Sub2
      …X…
      end;       -- of Sub2
   Begin         -- of Big
   …
   end;          -- of Big
```

- Under static scoping, the reference to the var X in Sub1 is to the X declared in the procedure Big.
- This is true b/c the search for X begins in the procedure in which the reference occurs, Sub1, but no declaration for X is found there.
- The search thus continues in the static parent of Sub1, Big, where the declaration of X is found.
- Ex: Skeletal C#

```
void sub()
{
  int count;
  …
  while (…)
  {
     int count;
     count ++;
     …
   }
   …
}
```

- The reference to count in the while loop is to that loop's local count. The count of sub is **hidden** from the code inside the while loop.
- A declaration for a var effectively hides any declaration of a var with the same name in a larger enclosing scope.
- C++ and Ada allow access to these "hidden" variables
  - In Ada: Main.X
  - In C++: class_name::name

Blocks

- Allows a section of code to have its own local vars whose scope is minimized.
- Such vars are **stack dynamic**, so they have their storage allocated when the section is entered and deallocated when the section is exited.
- From ALGOL 60:
- Ex:

  C and C++:

```
for (...)
{
  int index;
  ...
}
```

  Ada:

```
declare LCL : FLOAT;
begin
...
end
```

Dynamic Scope

- The scope of variables in APL, SNOBOL4, and the early versions of LISP is dynamic.
- Based on **calling sequences** of program units, not their textual layout (temporal versus spatial) and thus the scope is determined at **run time**.
- References to variables are connected to declarations by searching back through the chain of subprogram calls that forced execution to this point.
- Ex:

```
Procedure Big is
   X : Integer;
   Procedure Sub1 is
      Begin      -- of Sub1
      …X…
      end;       -- of Sub1
   Procedure Sub2 is
      X Integer;
      Begin      -- of Sub2
      …X…
      end;       -- of Sub2
   Begin         -- of Big
   …
   end;          -- of Big
```

- Big calls Sub1
  - The dynamic parent of Sub1 is Big. The reference is to the X in **Big**.
- Big calls Sub2 and Sub2 calls Sub1
  - The search proceeds from the local procedure, Sub1, to its caller, **Sub2**, where a declaration of X is found.
- Note that **if static scoping** was used, in either calling sequence the reference to X in Sub1 would be to **Big's X**.

Scope and Lifetime

- Ex:
```
void printheader()
{
…
}    /* end of printheader */
void compute()
{
   int sum;
   …
   printheader();
}    /* end of compute */
```

- The **scope** of sum in contained within compute.
- The **lifetime** of sum extends over the time during which printheader executes.
- Whatever storage location sum is bound to before the call to printheader, that binding will continue during and after the execution of printheader.

Referencing environment

- It is the collection of all names that are visible in the statement.
- In a **static-scoped language**, it is the local variables plus all of the visible variables in all of the enclosing scopes.
- The referencing environment of a statement is needed while that statement is being compiled, so code and data structures can be created to allow references to non-local vars in both static and dynamic scoped languages.
- A subprogram is active if its execution has begun but has not yet terminated.
- In a **dynamic-scoped language**, the referencing environment is the local variables plus all visible variables in all active subprograms.
- Ex, Ada, **static-scoped language**

```
procedure Example is
   A, B : Integer;
   …
   procedure Sub1 is
      X, Y : Integer;
      begin     -- of Sub1
      …                            🡸 1
       end      -- of Sub1
   procedure Sub2 is
      X : Integer;
      …
      procedure Sub3 is
         X : Integer;
         begin  -- of Sub3
         …                         🡸 2
         end;   -- of Sub3
    begin  -- of Sub2
    …                              🡸 3
     end;   { Sub2}
begin
   …                               🡸 4
end;        {Example}
```

- The referencing environments of the indicated program points are as follows:

Point	*Referencing Environment*
1	X and Y of Sub1, A & B of Example
2	X of Sub3, (X of Sub2 is hidden), A and B of Example
3	X of Sub2, A and B of Example
4	A and B of Example

- Ex, **dynamic-scoped language**
- Consider the following program; assume that the only function calls are the following: *main* calls *sub2*, which calls *sub1*

```
void sub1( )
{
  int a, b;
   …                    🡐 1
}       /* end of sub1 */
void sub2( )
{
  int b, c;
   …                    🡐 2
  sub1;
}       /* end of sub2 */
void main ( )
{
  int c, d;
   …                    🡐 3
  sub2( );
}       /* end of main */
```

- The referencing environments of the indicated program points are as follows:

Point	***Referencing Environment***
1	a and b of sub1, c of sub2, d of main
2	b and c of sub2, d of main
3	c and d of main

Named Constants

- It is a var that is bound to a value only at the time it is bound to storage; its value **can't** be change by assignment or by an input statement.
- Ex, Java

```
**final** int LEN = 100;
```

- **Advantages**: readability and modifiability

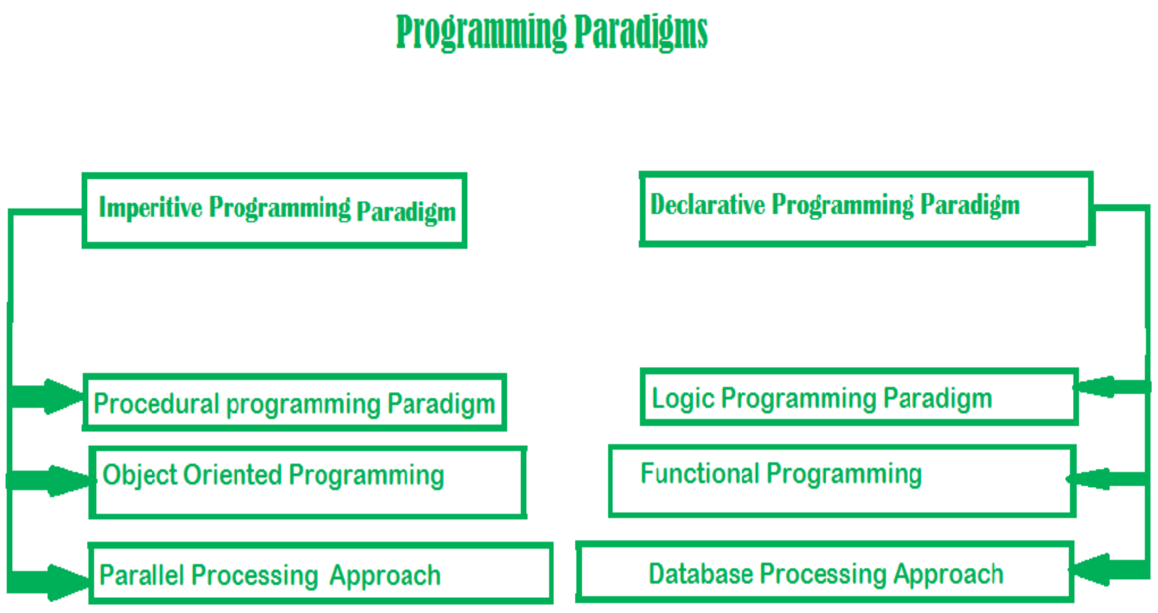
Variable Initialization

- The binding of a variable to a value at the time it is bound to storage is called initialization.
- Initialization is often done on the declaration statement.
- Ex, Java

```
int sum = 0;
```

Introduction of Programming Paradigms

Paradigm can also be termed as method to solve some problem or do some task. Programming paradigm is an approach to solve problem using some programming language or also we can say it is a method to solve a problem using tools and techniques that are available to us following some approach. There are lots for programming language that are known but all of them need to follow some strategy when they are implemented and this methodology/strategy is paradigms. Apart from varieties of programming language there are lots of paradigms to fulfill each and every demand.


Programming Paradigms
Imperitive Programming Paradigm
Declarative Programming Paradigm
Procedural programming Paradigm
Object Oriented Programming
Parallel Processing Approach
Logic Programming Paradigm
Functional Programming
Database Processing Approach

1. **Imperative programming paradigm:**

   It is one of the oldest programming paradigm. It features close relation to machine architecture. It is based on Von Neumann architecture. It works by changing the program state through assignment statements. It performs step by step task by changing state. The main focus is on how to achieve the goal. The paradigm consist of several statements and after execution of all the result is stored.

Advantage:

1. Very simple to implement
2. It contains loops, variables etc.

Disadvantage:

1. Complex problem cannot be solved
2. Less efficient and less productive
3. Parallel programming is not possible

Examples of **Imperative** programming paradigm:

C : developed by Dennis Ritchie and Ken Thompson
Fortan : developed by John Backus for IBM
Basic : developed by John G Kemeny and Thomas E Kurtz

```
// average of five number in C


int marks[5] = { 12, 32, 45, 13, 19 } int sum = 0;

float average = 0.0;

for (int i = 0; i < 5; i++) {

    sum = sum + marks[i];

}

average = sum / 5;
```

Imperative programming is divided into three broad categories: Procedural, OOP and parallel processing. These paradigms are as follows:

- **Procedural programming paradigm –**
  This paradigm emphasizes on procedure in terms of under lying machine model. There is no difference in between procedural and imperative approach. It has the ability to reuse the code and it was boon at that time when it was in use because of its reusability.

Examples of **Procedural** programming paradigm:

C : developed by Dennis Ritchie and Ken Thompson

C++ : developed by Bjarne Stroustrup

Java : developed by James Gosling at Sun Microsystems

ColdFusion : developed by J J Allaire

Pascal : developed by Niklaus Wirth

```
//C++ program to find factorial of number
#include <iostream>
using namespace std;
int main()
{
   int i, fact = 1, num;
   cout << "Enter any Number: ";
   cin >> number;
   for (i = 1; i <= num; i++) {
      fact = fact * i;
   }
   cout << "Factorial of " << num << " is: " << fact << endl;
   return 0;
}
```

Then comes OOP,

- **Object oriented programming –**
  The program is written as a collection of classes and object which are meant for communication. The smallest and basic entity is object and all kind of computation is performed on the objects only. More emphasis is on data rather procedure. It can handle almost all kind of real life problems which are today in scenario.

Advantages:

- Data security
- Inheritance
- Code reusability
- Flexible and abstraction is also present

Examples of **Object Oriented** programming paradigm:

Simula : first OOP language

Java : developed by James Gosling at Sun Microsystems

C++ : developed by Bjarne Stroustrup

Objective-C : designed by Brad Cox

Visual Basic .NET : developed by Microsoft

Python : developed by Guido van Rossum

Ruby : developed by Yukihiro Matsumoto

Smalltalk : developed by Alan Kay, Dan Ingalls, Adele Goldberg

- Java

```
import java.io.*;


class GFG {
    public static void main(String[] args)
    {
        System.out.println("GfG!");
        Signup s1 = new Signup();
        s1.create(22, "riya", "riya2@gmail.com", 'F', 89002);
    }
}


class Signup {
    int userid;
    String name;
    String emailid;
    char sex;

    long mob;


    public void create(int userid, String name,

                       String emailid, char sex, long mob)

    {

       System.out.println("Welcome to

                GeeksforGeeks\nLets create your account\n");

       this.userid = 132;

       this.name = "Radha";

       this.emailid = "radha.89@gmail.com";

       this.sex = 'F';

       this.mob = 900558981;

       System.out.println("your account has been created");

    }

  }
```

Parallel processing Approach

Parallel processing is the processing of program instructions by dividing them among multiple processors. A parallel processing system posses many numbers of processor with the objective of running a program in less time by dividing them. This approach seems to be like divide and conquer. Examples are NESL (one of the oldest one) and C/C++ also supports because of some library function.

2. Declarative programming paradigm:

It is divided as Logic, Functional, Database. In computer science the *declarative* *programming* is a style of building programs that expresses logic of computation without talking about its control flow. It often considers programs as theories of some logic.It may simplify writing parallel programs. The focus is on what needs to be done rather how it should be done basically emphasize on what code is actually doing. It just declares the result we want rather how it has be produced. This is the only difference between imperative (how to do) and declarative (what to do) programming paradigms. Getting into deeper we would see logic, functional and database.

- **Logic programming paradigms –** It can be termed as abstract model of computation. It would solve logical problems like puzzles, series etc. In logic programming we have a knowledge base which we know before and along with the question and knowledge base which is given to machine, it produces result. In normal programming languages, such concept of knowledge base is not available but while using the concept of artificial intelligence, machine learning we have some models like Perception model which is using the same mechanism. In logical programming the main emphasize is on knowledge base and the problem. The execution of the program is very much like proof of mathematical statement, e.g., Prolog

sum of two number in prolog:

```
  predicates
  sumoftwonumber(integer, integer)
clauses

  sum(0, 0).
   sum(n, r):-
        n1=n-1,
        sum(n1, r1),
        r=r1+n
```

Functional programming paradigms –

The functional programming paradigms has its roots in mathematics and it is language independent. The key principle of this paradigms is the execution of series of mathematical functions. The central model for the abstraction is the function which are meant for some specific computation and not the data structure. Data are loosely coupled to functions.The function hide their implementation. Function can be replaced with their values without changing the meaning of the program. Some of the languages like perl, javascript mostly uses this paradigm.

Examples of **Functional** programming paradigm:
JavaScript : developed by Brendan Eich

Haskell : developed by Lennart Augustsson, Dave Barton

Scala : developed by Martin Odersky

Erlang : developed by Joe Armstrong, Robert Virding

Lisp : developed by John Mccarthy

ML : developed by Robin Milner

Clojure : developed by Rich Hickey

The next kind of approach is of Database.

Database/Data driven programming approach

This programming methodology is based on data and its movement. Program statements are defined by data rather than hard-coding a series of steps. A database program is the heart of a business information system and provides file creation, data entry, update, query and reporting functions. There are several programming languages that are developed mostly for database application. For example SQL. It is applied to streams of structured data, for filtering, transforming, aggregating (such as computing statistics), or calling other programs. So it has its own wide application.

```
CREATE DATABASE databaseAddress;

CREATE TABLE Addr (
   PersonID int,
   LastName varchar(200),
   FirstName varchar(200),
   Address varchar(200),
       City varchar(200),
   State varchar(200)
);
```

Storage Allocation

Storage allocation is a process by which computer programs and services are assigned with physical or virtual memory space.Storage allocation is the process of reserving a partial or complete portion of computer memory for the execution of programs and processes. Storage allocation is achieved through a process known as memory management.

Static Allocation

Static allocation is an allocation procedure that is used for the allocation of all the data objects at compile time. In this type of allocation, allocation of data objects is done at compile time only. As in static allocation, the compiler decides the extent of storage which cannot be changed with time, hence it is easy for the compiler to know the addresses of these data objects in the activation records at a later stage. Static allocation is implemented in FORTRAN.

Examples of static variables are:

- Globally declared variables
- Print statements
- Numeric literals and string valued constants

Properties of Static Allocation

- Memory allocation is done during compile time.
- Memory cannot be changed while executing a program.
- The static memory allocation is fast and saves running time.
- It is less efficient as compared to Dynamic memory allocation.
- The allocation process is simple.

Disadvantages of Static Memory allocation

- This allocation method leads to memory wastage.
- Memory cannot be changed while executing a program.
- Exact memory requirements must be known.
- If memory is not required, it cannot be freed.

2. Stack Based Allocation

- The allocation happens on contiguous blocks of memory. The stack allocation is a runtime storage management technique. The activation records are pushed and popped as activations begin and end respectively.
- The size of memory to be allocated is known to the compiler and whenever a function is called, its variables get memory allocated on the stack and whenever the function call is over, the memory for the variables is de-allocated.
- A programmer does not have to worry about memory allocation and de-allocation of stack variables. This kind of memory allocation is also known as **Temporary memory allocation** because as soon as the method finishes its execution all the data belongs to that method flushes out from the stack automatically.
- Means, any value stored in the stack memory scheme is accessible as long as the method hasn't completed its execution and currently in running state.

Disadvantages of Stack Allocation

- Stack memory is of limited size.
- The total of size of the stack must be defined before.
- If too many objects are created then it can lead to stack overflow.
- Random accessing is not possible in stack.

Static allocation Vs Stack allocation

S.No.	Static Allocation	Stack Allocation
1.	Static Allocation does not makes data structures and objects dynamically.	Stack allocation makes data structures and objects dynamically.
2.	In static allocation, allocation of all data objects is performed at compile time.	While in stack allocation, allocation of data objects is performed at run time.
3.	It does not support recursive procedures.	It supports recursive procedures.
4.	Static allocation is not able to manage the allocation of memory at run time.	Stack allocation use stack to manage the allocation of memory at run time.
5.	In static allocation, at compile time the data object names are fixed.	In stack allocation, index and registers performs the memory addressing.
6.	This strategy is easy and simple in implementing.	This strategy is slower than static allocation.

Heap Allocation

Heap allocation is an allocation procedure in which the heap is used to manage the allocation of memory. Heap helps in managing dynamic memory allocation. In heap allocation, the creation of dynamic data objects and data structures is also possible as same as stack allocation. Heap allocation overcomes the limitation of stack allocation. It is possible to retain the value of variables even after the activation record in heap allocation strategy which is not possible in stack allocation. It maintains a linked list for the free blocks and reuses the deallocated space using the best fit.

The difference between Static Allocation and Heap Allocation is as follows:

S.No.	Static Allocation	Heap Allocation
1.	Static allocation allocates memory on the basis of the size of data objects.	Heap allocation makes use of heap for managing the allocation of memory at run time.
2.	In static allocation, there is no possibility of the creation of dynamic data structures and objects.	In heap allocation, dynamic data structures and objects are created.
3.	In static allocation, the names of the data objects are fixed with storage for addressing.	Heap allocation allocates a contiguous block of memory to data objects.
4.	Static allocation is a simple, but not efficient memory management technique.	Heap allocation does memory management inefficiently way.
5.	The static allocation strategy is faster in accessing data as compared to heap allocation.	While heap allocation is slow in accessing as there is a chance of creation of holes in reusing the free space.
6.	Static allocation is inexpensive, it is easy to implement.	While heap allocation is comparatively expensive.
7.	Static memory allocation is preferred in an array.`	Heap memory allocation is preferred in the linked list.
8.	Example : • int i; • float f;	Example : • p = malloc(sizeof(int));

Stack Allocation

The allocation happens on contiguous blocks of memory. We call it a stack memory allocation because the allocation happens in the function call stack. The size of memory to be allocated is known to the compiler and whenever a function is called, its variables get memory allocated on the stack. And whenever the function call is over, the memory for the variables is de-allocated. This all happens using some predefined routines in the compiler. A programmer does not have to worry about memory allocation and de-allocation of stack variables. This kind of memory allocation also known as Temporary memory allocation because as soon as the method finishes its execution all the data belongs to that method flushes out from the stack automatically. Means, any value stored in the stack memory scheme is accessible as long as the method hasn't completed its execution and currently in running state.

Parameter	STACK	HEAP
Basic	Memory is allocated in a contiguous block.	Memory is allocated in any random order.
Allocation and De-allocation	Automatic by compiler instructions.	Manual by the programmer.
Cost	Less	More
Implementation	Easy	Hard
Access time	Faster	Slower
Main Issue	Shortage of memory	Memory fragmentation
Locality of reference	Excellent	Adequate
Safety	Thread safe, data stored can only be accessed by owner	Not Thread safe, data stored visible to all threads
Flexibility	Fixed-size	Resizing is possible
Data type structure	Linear	Hierarchical

Parameter	STACK	HEAP
Preferred	Static memory allocation is preferred in array.	Heap memory allocation is preferred in the linked list.
Size	Small than heap memory.	Larger than stack memory.

Garbage Collection

Garbage collection is a memory management technique. It is a separate automatic memory management method which is used in programming languages where manual memory management is not preferred or done. In the manual memory management method, the user is required to mention the memory which is in use and which can be deallocated, whereas the garbage collector collects the memory which is occupied by variables or objects which are no more in use in the program. Only memory will be managed by garbage collectors, other resources such as destructors, user interaction window or files will not be handled by the garbage collector.

Few languages need garbage collectors as part of the language for good efficiency. These languages are called as garbage-collected languages. For example, Java, C# and most of the scripting languages needs garbage collection as part of their functioning. Whereas languages such as C and C++ support manual memory management which works similar to the garbage collector. There are few languages that support both garbage collection and manually managed memory allocation/deallocation and in such cases, a separate heap of memory will be allocated to the garbage collector and manual memory.

What is the Type-System?

When learning a programming language, the first few sections of the introductory course usually introduce the various data types of the language, and they are inseparable from the subsequent programming development. This is enough to show the importance of types to a programming language.Types in programming languages are broadly classified and can be divided into **built-in types** represented by int and float, and **abstract types** represented by class and function. The most striking feature that distinguishes these types is that we can only use its specific operations on a specific type.It's a system focused on managing types, it's a logical system consisting of a set of rules that assign properties called types to various structures of a computer program, such as variables, expressions, functions, or modules.

What can a Type-System do?

1. Define the program type to ensure the security of the program.
2. It can improve the readability of the code and improve the abstraction level of the code, rather than the low-level inefficient implementation.
3. It is beneficial to compiler optimization. After specifying a type, the compiler can align it with the corresponding byte, thereby generating efficient machine instructions.

As can be seen from the above points, the type of a variable can determine a specific meaning and purpose, which is extremely beneficial to our programming.

What is the nature of types in a program?

The type system can be said to be a tool because the program ultimately runs machine code. In the world of machine code, there are no types, those instructions just deal with immediate data or memory.

So the type is essentially an abstraction of memory, and different types correspond to different memory layouts and memory allocation strategies.

For more information on memory management check out my previous article:

Memory Management Every Programmer Should Know

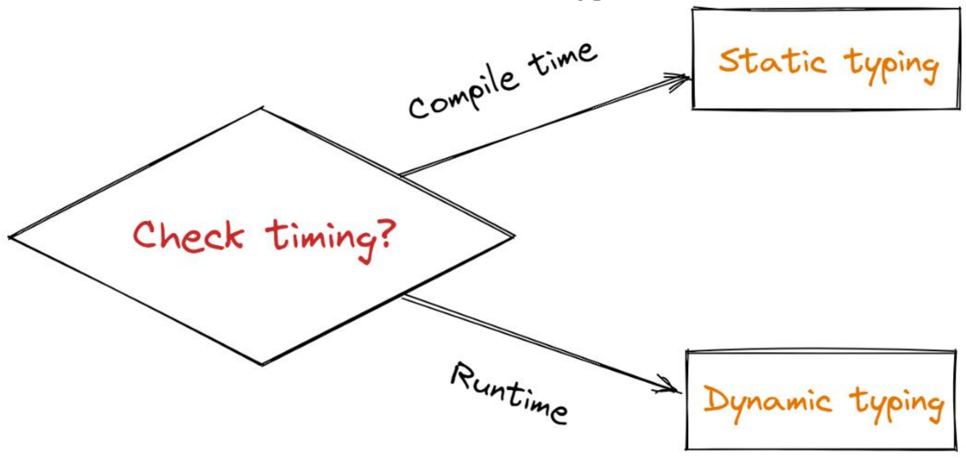
Static vs. dynamic typing

We often hear this question, but the difference between the two is when the type checking happens.

- The static type system determines the types of all variables at compile-time and throws exceptions in case of incorrect usage.

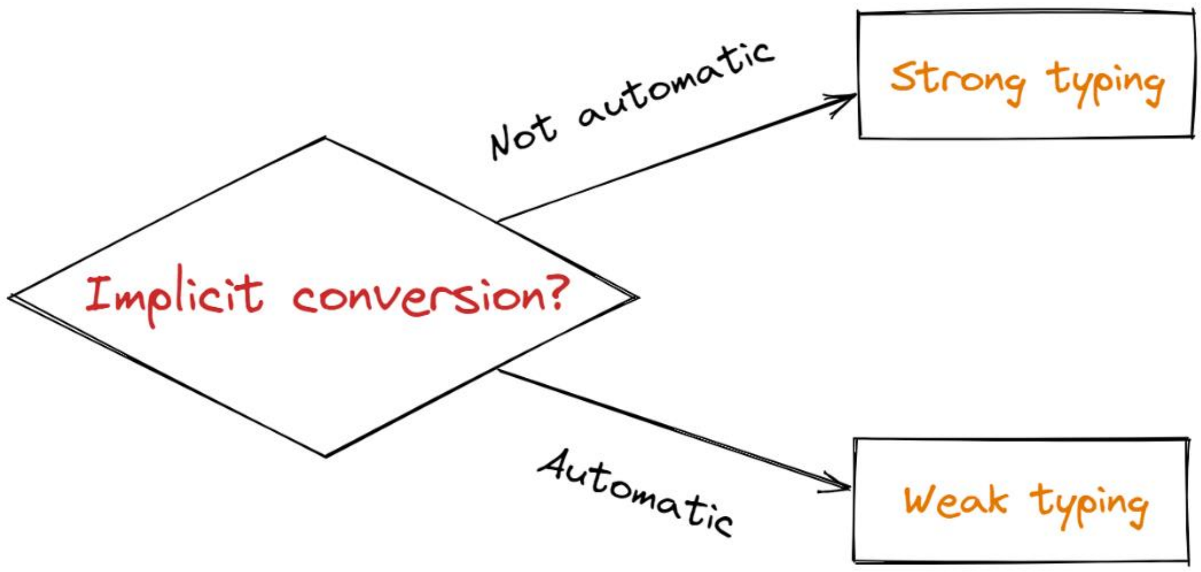
- The dynamic type system determines the variable type at runtime, throws an exception if there is an error, and may crash the program without proper handling.

The early type error reporting of the static type system ensures the security of large-scale application development, while the disadvantage of the dynamic type system is that there is no type checking at compile-time, and the program is not safe enough. Only a large number of unit tests can be used to ensure the robustness of the code. But programs that use the dynamic type system are easy to write and don't take a lot of time to make sure the type is correct.



Strong vs. Weak typing

There is no authoritative definition of the difference between strong typing and weak typing. Most of the early discussions on strong typing and weak typing can be summarized as the difference between static typing and dynamic typing. But the prevailing saying is that strong types tend to not tolerate implicit type conversions, while weak types tend to tolerate implicit type conversions. In this way, a strongly typed language is usually type-safe, that is, it can only access the memory it is authorized to access in the permitted ways.



Type Checking

To do *type checking* a compiler needs to assign a type expression to each com-ponent of the source program. The compiler must then determine that these type expressions conform to a collection of logical rules that is called the *type system* for the source language.

Type checking has the potential for catching errors in programs. In principle, any check can be done dynamically, if the target code carries the type of an element along with the value of the element. A *sound* type system eliminates the need for dynamic checking for type errors, because it allows us to determine statically that these errors cannot occur when the target program runs. An implementation of a language is *strongly typed* if a compiler guarantees that the programs it accepts will run without type errors.

Besides their use for compiling, ideas from type checking have been used to improve the security of systems that allow software modules to be imported and executed. Java programs compile into machine-independent bytecodes that include detailed type information about the operations in the bytecodes. Im-ported code is checked before it is allowed to execute, to guard against both inadvertent errors and malicious misbehavior.

1. Rules for Type Checking

Type checking can take on two forms: synthesis and inference. *Type synthesis* builds up the type of an expression from the types of its subexpressions. It requires names to be declared before they are used. The type of *E1* + E_2 is defined in terms of the types of *E1* and E_2 • A typical rule for type synthesis has the form

$$\begin{array}{l}\textbf{if } f \text{ has type } s \rightarrow t \textbf{ and } x \text{ has type } s,\\ \textbf{then } \text{expression } f(x) \text{ has type } t\end{array} \qquad (6.8)$$

Here, f and x denote expressions, and s ->• t denotes a function from s to t. This rule for functions with one argument carries over to functions with several arguments. The rule (6.8) can be adapted for E1 +E2 by viewing it as a function application add(Ei,E2).Q

Type inference determines the type of a language construct from the way it is used. Looking ahead to the examples in Section 6.5.4, let null be a function that tests whether a list is empty. Then, from the usage null(x), we can tell that x must be a list. The type of the elements of x is not known; all we know is that x must be a list of elements of some type that is presently unknown. Variables representing type expressions allow us to talk about unknown types. We shall use Greek letters a, (3, • • • for type variables in type expressions.

A typical rule for type inference has the form

$$\begin{array}{l}\textbf{if } f(x) \text{ is an expression,}\\ \textbf{then } \text{for some } \alpha \text{ and } \beta,\ f \text{ has type } \alpha \rightarrow \beta \textbf{ and } x \text{ has type } \alpha\end{array} \qquad (6.9)$$

Type inference is needed for languages like ML, which check types, but do not require names to be declared.

We shall use the term "synthesis" even if some context information is used to determine types. With overloaded functions, where the same name is given to more than one function, the context of E1 + E2 may also need to be considered in some languages.

In this section, we consider type checking of expressions. The rules for checking statements are similar to those for expressions. For example, we treat the conditional statement "if *(E)* S;" as if it were the application of a function *if* to *E* and *S.* Let the special type *void* denote the absence of a value. Then function *if* expects to be applied to a *boolean* and a *void;* the result of the application is a *void.*

EQUALITY TESTING AND ASSIGNMENT

➢ For simple, primitive data types such as integers, floating-point numbers, or characters, equality testing and assignment are relatively straightforward operations.

➢ It Consider for example the problem of comparing two character strings.

➢ Should the expression s = t determine whether s and t are aliases for one another?

➢ occupy storage that is bit-wise identical over its full length? contain the same sequence of characters?

➢ would appear the same if printed?

➢ In many cases the definition of equality boils down to the distinction between lvalues and r-values:

➢ In the presence of references, should expressions be considered equal only if they refer to the same object, or also if the objects to which they refer are in some sense equal?

➢ The first option (refer to the same object)is known as a shallow comparison. The second (refer to equal objects) is called a deep comparison is difficult for more complicated for abstract data types.

For complicated data structures (e.g., lists or graphs) a deep comparison may require recursive traversal. In imperative programming languages, assignment operations may also be deep or shallow.

➢ Under a reference model of variables, a shallow assignment a := b will make a refer to the object to which b refers. A deep assignment will create a copy of the object to which 8b9 refers, and make 8a9 refer to the copy.

➢ Under a value model of variables, a shallow assignment will copy the value of b into a, but if that value is a pointer, then the objects to which the pointer(s) refer will not be copied.

➢ Scheme, for example, has three general-purpose equality-testing functions: (eq? a b) ; do a and b refer to the same object?

➢ (eqv? a b) ; are a and b known to be semantically equivalent? (equal? a b) ; do a and b have the same recursive structure?

➢ (eq? a b) ; do a and b refer to the same object? ➢ eq.? is pointer comparison.it returns T iff its arguments literally refer to same object in memory (eqv? a b) ; are a and b known to be semantically equivalent?

➢ Eqv? Is comparing numbers. Required to detect the equality of values of the same discrete type, stored in different locations.

➢ (equal? a b) ; do a and b have the same recursive structure?

➢ Returns T if its arguments have same structure.

➢ The eqv? predicate is <less discriminating= than eq?, in the sense that eqv?

will never return false when eq? returns true. For structures (lists), eqv?

returns false if its arguments refer to different root cons cells.

➢ The equal? predicate recursively traverses two lists to see if their internal

structure is the same and their leaves are eqv?.

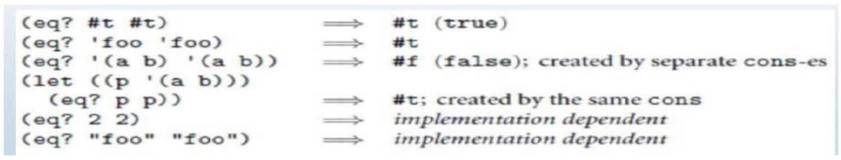
➢ The equal? predicate may lead to an infinite loop if the programmer has

used the imperative features of Scheme to create a circular list

➢ In any particular implementation, numeric, character, and string tests will

always work the same way; if (eq? 2 2) returns true, then (eq? 37 37) will also

return true .



```
(eq? #t #t)              ⟹  #t (true)
(eq? 'foo 'foo)          ⟹  #t
(eq? '(a b) '(a b))      ⟹  #f (false); created by separate cons-es
(let ((p '(a b)))
   (eq? p p))            ⟹  #t; created by the same cons
(eq? 2 2)                ⟹  implementation dependent
(eq? "foo" "foo")        ⟹  implementation dependent
```

A type system consists of a mechanism for defining types and associating them with language constructs A set of rules for

(i) Type equivalence : When do two objects have the same type?

(ii) Type compatibility : Where can objects of a given type be used?

(iii) Type synthesis/inference : How do you determine the type of an expression from its parts or, in some cases, from its context (i.e., the type of the whole is used to determine the type of the parts).

The synthesis/inference terminology is not standardized. Some texts, e.g., 3e, use type inference both for determining the type of the whole from the type of its parts, and for determining the type of the parts from the type of the whole. Other texts, e.g., the Dragon book, use type synthesis for the former and type inference for the latter.

Some languages are untyped (e.g., B the predecessor of C); we will have little to say about those beyond saying that B actually had one datatype, the computer word.

Types must be assigned to those constructs that can have values or that can refer to objects that have values. These include.

o Literal constants (e.g., 5.8, "hello").

o Named constants (e.g. static final int x = 3).

o Variables.

o Subroutines (not always, but having signatures is nice)

o More complicated expressions made up of these.

Type Equivalence

When are two types equivalent? There are two schools: name equivalence and structural equivalence. In (strict) name equivalence two type definitions are always distinct; Thus the four types on the right T1,...,T4 are all distinct.

- In structural equivalence, types are equivalent if they have the same structure so types T3 and T4 are equivalent and aggregates of those two types could be assigned to each other. Similarly, Ti, T2, and integer are equivalent under structural equivalence.

```
type  T1 is new integer
type  T2 is new integer
type  T3 is record x:integer;y:integer ;end record;
type  T4 is record x:integer;y:integer ;end record;
subtype S1 is integer
```

Type Compatibility

- A value must have a type compatible with the context in which the value is used. For most languages this notion is significantly weaker than equivalence; that is, there may be several

non-equivalent types that can legally occur at a given context.

-We first need to ask what are the contexts in which only a type compatible with the expected

type can be used, Three important contexts are

-Assignment statements : The type of the left hand side Ohs) must be compatible with the

type of the value computed by the rhs.

-Subroutine calls : The types of the arguments must be compatible with the types of the

corresponding parameters.

-Built-in operations : This situation is very similar to a subroutine call. For example, the

operands of and must have types compatible with Boolean. Many of these operators, e.g., +, are overloaded (defined below) so there may be several types with which the operand types may be compatible. For example the operand types of + may be compatible with integer or with real.

Type Inference/Synthesis

In this section we emphasize type synthesis, i.e, determining the type of an expression given the type of its constituents. Next lecture, when we study ML, we will encounter type inference, where the type of the constituents in determined by the type of the expression.