Introduction to Programming Paradigms and Core Language Design Issues

Vinita Mishra

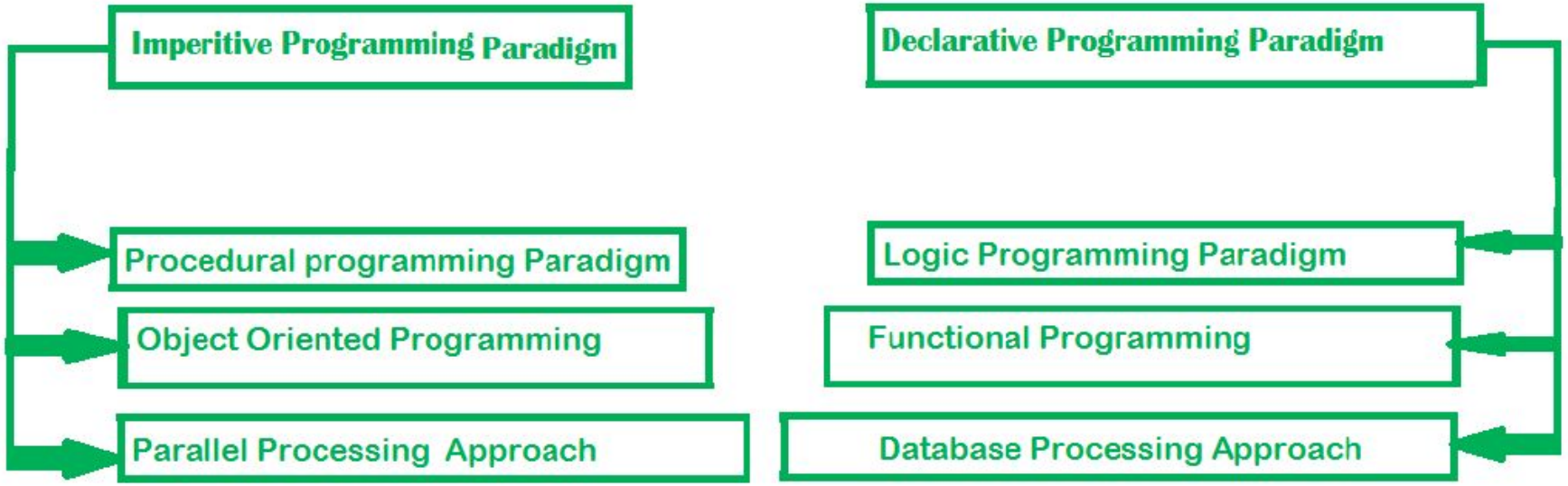
Introduction to different programming paradigms

Paradigm can be termed as method to solve some problem or do some task. Programming paradigm is an approach to solve problem using some programming language

It is also 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

Programming Paradigms



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.

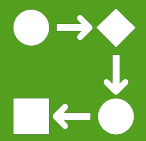
It works by changing the program state through assignment statements. It performs step by step task by changing state.

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.

Vinita Mishra

Imperative programming paradigm



Advantage:

- Very simple to implement
- It contains loops, variables etc.

Disadvantage:

- Complex problem cannot be solved
- Less efficient and less productive
- Parallel programming is not possible

Imperative programming paradigm

Vinita Mishra

Imperative programming is divided into three broad categories: Procedural, OOP and parallel processing

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.

E.g. C language

Imperative programming paradigm

```
#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;
}
```

Vinita Mishra

Imperative programming paradigm

Vinita Mishra

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

Imperative programming paradigm

Vinita Mishra

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

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 code is actually doing.
- It just declare 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.

Declarative programming paradigm

Vinita Mishra

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

Declarative programming paradigm

Vinita Mishra

Functional programming paradigms -
The functional programming paradigms has its roots in mathematics and it is language independent.

The key principal 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.

Declarative programming paradigm

Vinita Mishra

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.

Names, Scope, Memory, and Binding

Name:

- 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 code is actually doing.
- It just declare 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.

Binding

The terms STATIC and DYNAMIC are generally used to refer to things bound before run time and at run time, respectively

- “static” is a coarse term; so is "dynamic"

IT IS DIFFICULT TO OVERSTATE THE IMPORTANCE OF BINDING TIMES IN THE DESIGN AND IMPLEMENTATION OF PROGRAMMING LANGUAGES

Binding(association between name and object)

Binding Time is the point at which a binding is created

- language design time
  - program structure, possible type, control flow structure(i.e. if,if else ,for….)
  - Constructors,pointers….
- language implementation time
  - I/O, arithmetic overflow, stack size, type equality (if unspecified in design)
  - Eg int…what is size of int,float..what is precision of float
- program writing time
  - algorithms, names
- compile time
  - Time taken to convert HLL to ML
- link time
  - different module will be compiled separately and finally linked together at link time.
- load time
  - Once linked, file has to be loaded in memory for execution.

Binding

Implementation decisions (continued):

- run time(time for running the program)
  - value/variable bindings, sizes of strings
  - program start-up time
  - module entry time
  - elaboration time (point a which a declaration is first "seen")
  - procedure entry time
  - block entry time
  - statement execution time

BASIS FOR COMPARISON	STATIC BINDING	DYNAMIC BINDING
Event Occurrence	Events occur at compile time are "Static Binding".	Events occur at run time are "Dynamic Binding".
Information	All information needed to call a function is known at compile time.	All information need to call a function come to know at run time.
Advantage	Efficiency.	Flexibility.
Time	Fast execution.	Slow execution.
Alternate name	Early Binding.	Late Binding.
Example	Overloaded function call, overloaded operators.	Virtual function in C++, overridden methods in java.

Object lifetime

Binding lifetime:

- Time between creation and destruction of name to object binding

Objects lifetime:

- Time b/w creation and destruction of object
  - Name can be destroyed but object is still in the memory

Dangling reference:

- Binding to object that no longer live

Lifetime and Storage Management

Key events

- creation of objects
- creation of bindings
- references to variables (which use bindings)
- (temporary) deactivation of bindings
- reactivation of bindings
- destruction of bindings
- destruction of objects

The period of time from creation to destruction is called the LIFETIME of a binding

- If object outlives binding it's **garbage**
- If binding outlives object it's a **dangling reference**

The textual region of the program in which the binding is *active* is its scope

Storage Management

Static:

Variables whose value will be same throughout the program is called static variable eg.global variables in heap

Stack:

based on LIFO order Eg, subroutine calls i.e when function is called, it gets stored in form of LIFO

Heap:

arbitrary time data, data which comes at run time is stored in heap

Lifetime and Storage Management

Storage Allocation mechanisms(how the objects are stored in the memory)

- Static
  - Given absolute address retained throughout program exection e.g global variables
- Stack
  - Allocated and reallocated in LIFO order
  - Applicable for subroutine…function call
- Heap(most costliest aloocation)
  - Allocated and reallocated at arbiter times

Static allocation for

- code
- globals
- static or own variables
- explicit constants (including strings, sets, etc)
- scalars may be stored in the instructions

Static Allocation

- Objects whose value do not change during Program execution
- Statically allocated objects
  - Global variables
  - Instruction for machine language translation
  - Local variables that retain values b/w invocation
  - Numeric and string valued constants literals
  - A=B/14-7
  - Printf(“hello, world\n”)
- Debugging,dynamictype-checkinh,garbage collection, exception handling

Vinita Mishra

Static Allocation

- **Named constant** (e.g.const keyword)
  - Value is determined at compile time
  - Named constants with constant literals are called manifest constants or compile-time constants
  - **Compile time**
    - Syntax checking
  - Elaboration time
    - Translation of HLL to MLL
  - Run time
    - Execution of object code(MLL)

Static Allocation

Elaboration time constant:

Values may depend on other values not known until run time

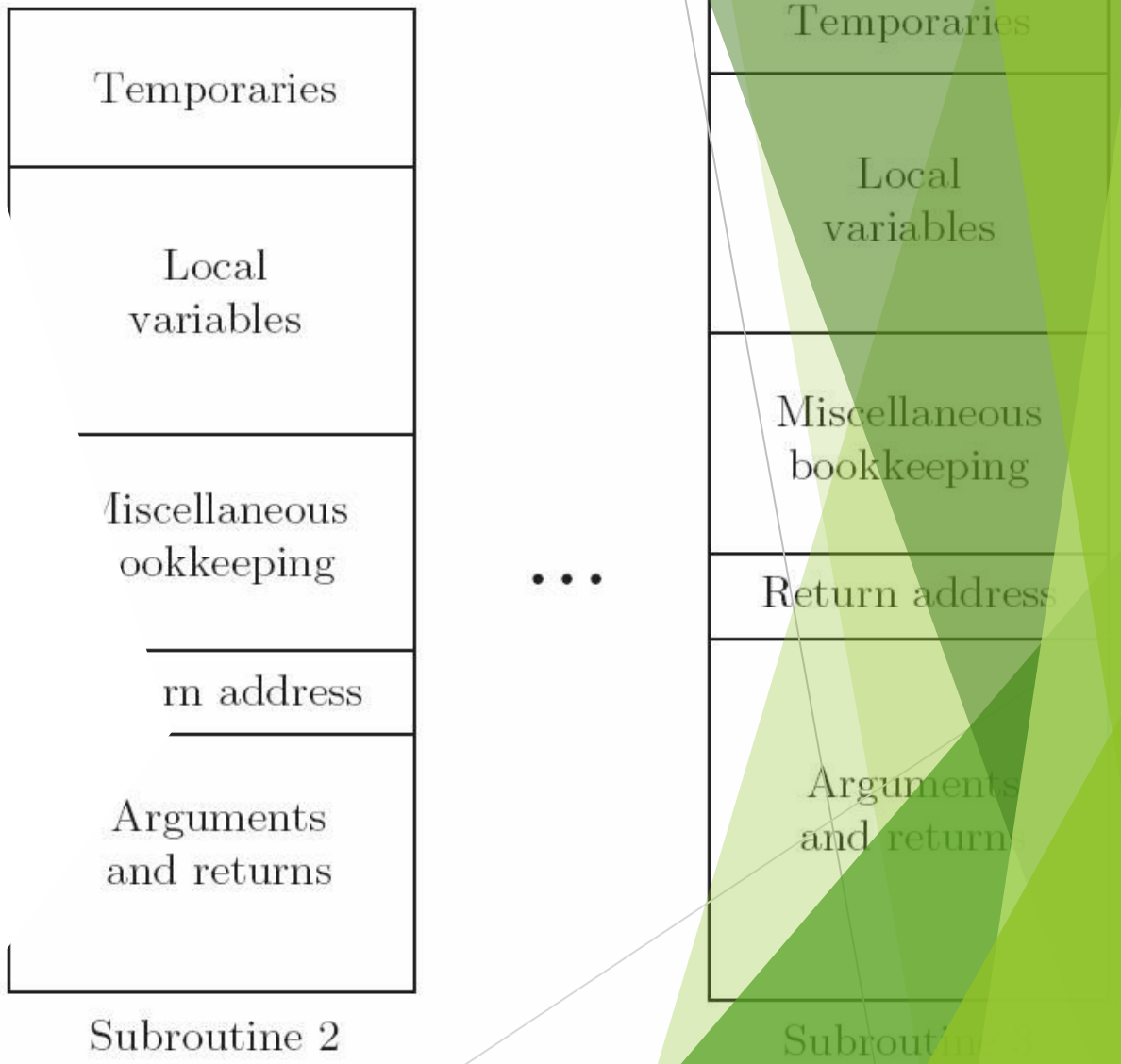
#define is example in c

C# provide both option

- Keyword-manifest constant
- Read-only-elaboration constant

Vinita Mishra

Lifetime and Storage Management



atic allocation of space for subroutines in

Stack based allocation

All subroutines are stack allocated

When a function is called, invokes multiple invocation

To avoid memory wasting,stack is used to allocated function

Vinita Mishra

Stack based allocation

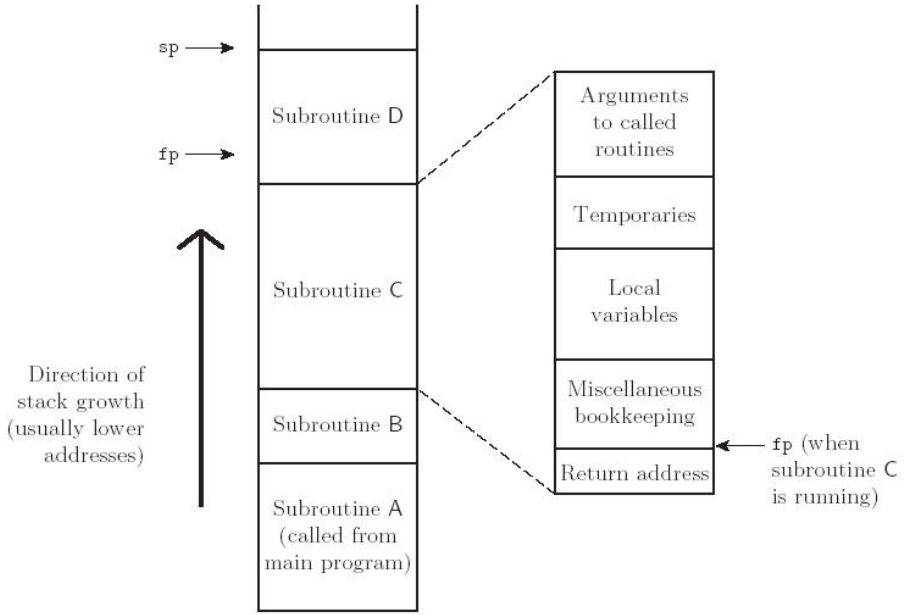


Figure 3.2: **Stack-based allocation of space for subroutines.** We assume here that subroutine A has been called by the main program, and that it then calls subroutine B. Subroutine B subsequently calls C, which in turn calls D. At any given time, the stack pointer (`sp`) register points to the first unused location on the stack (or the last used location on some machines), and the frame pointer (`fp`) register points to a known location within the frame (activation record) of the current subroutine. The relative order of fields within a frame may vary from machine to machine and compiler to compiler.

Lifetime and Storage Management

- Central stack for subroutine
  - Arguments to routines
  - parameters
  - local variables
  - Temporaries(temporary variables)
  - Book keeping(additional information is stored)
  - Return address(after subroutine is done, where it goes)

Lifetime and Storage Management

Maintenance of stack is done by subroutine calling sequence, prologue and epilogue

Calling sequence:

- Code executed by caller immediately before and after the call
- Combined operations of caller, prologue and epilogue

Prologue

- Code executed at the beginning

Epilogue:

- Code executed at the end

Example: Examine Stack for the C Program

```
int bar(int x)
{
        int z=5;
        return z;
}

int foo(int x)
{
        int y=3;
        x = x + y;
        y = bar(x);
        return x;
}

int main(int argc, char* argv[])
{
        int a=1, b=2, c=3;
        b = foo(a);
        printf("%d %d %d\n",a,b,c);
        return 0;
}
```

Memory Management Heap

- More unstructured
- Region of memory where sub blocks are allocated and deallocated dynamically
- Allocation and deallocation may happen in arbitrary order
  - Memory may become fragmented
  - Need for garbage collection
  - We will describe garbage collection algorithms later
- Heap Management
  - Often managed with a single linked list – the free list – of blocks not in use
    - First Fit?
    - Best Fit? (smallest block large enough to handle request)

Lifetime and Storage Management

- Heap for dynamic allocation

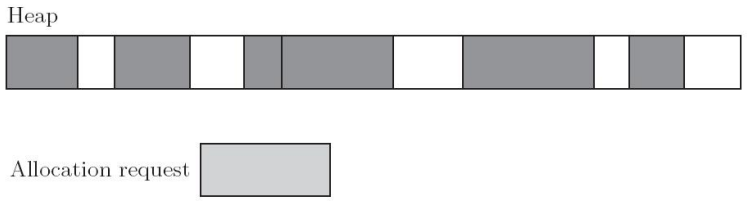


Figure 3.3: **External fragmentation.** The shaded blocks are in use; the clear blocks are free. While there is more than enough total free space remaining to satisfy an allocation request of the illustrated size, no single remaining block is large enough.

Scope Rules

A *scope* is a program section of maximal size in which no bindings change, or at least in which no re-declarations are permitted

In most languages with subroutines, we OPEN a new scope on subroutine entry:

- create bindings for new local variables,
- deactivate bindings for global variables that are re-declared (these variable are said to have a "hole" in their scope)
- make references to variables

Scope Rules

With STATIC (LEXICAL) SCOPE RULES, a scope is defined in terms of the physical (lexical) structure of the program

- The determination of scopes can be made by the compiler
- All bindings for identifiers can be resolved by examining the program
- Typically, we choose the most recent, active binding made at compile time
- Most compiled languages, C++ and Java included, employ static scope rules

Scope Rules

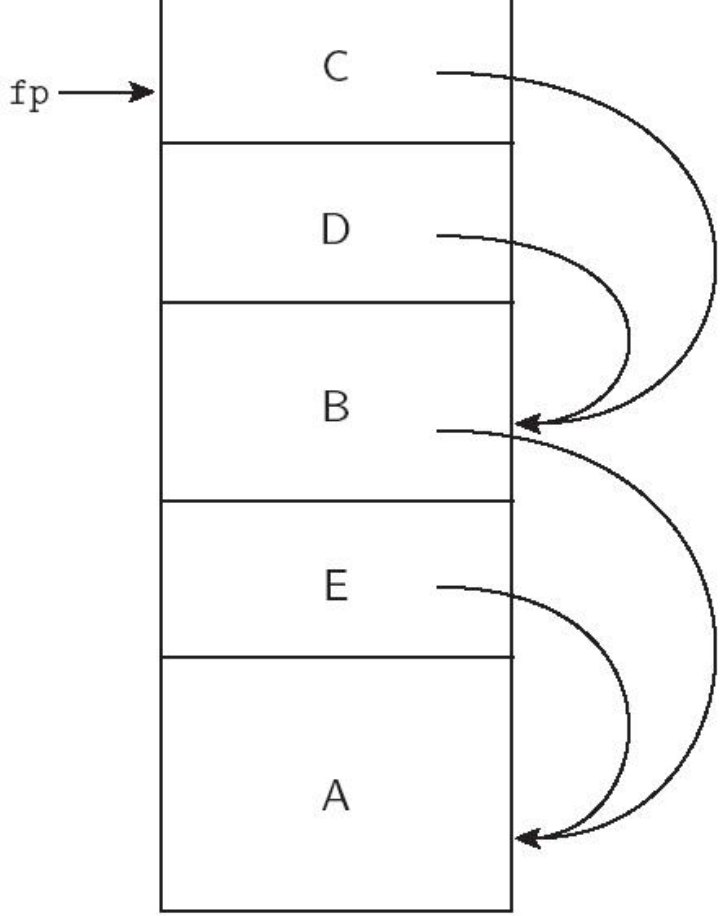
Note that the bindings created in a subroutine are destroyed at subroutine exit

Obvious consequence when you understand how stack frames are allocated and deallocated

The modules of Modula, Ada, etc., give you closed scopes without the limited lifetime

- Bindings to variables declared in a module are inactive outside the module, not destroyed
- The same sort of effect can be achieved in many languages with *own* (Algol term) or *static* (C term) variables

Scope Rules



. Subroutines A, B, C, D, and E are nested as sh
calls at run time is A, E, B, D, and C, then the s
wn on the right. The code for subroutine C can find
he frame pointer. It can find local objects of the surr
ts static chain once and then applying an offset. It ca
ing scope, A, by dereferencing its static chain twice and

Scope Rules

The key idea in **static scope rules** is that bindings are defined by the physical (lexical) structure of the program.

With **dynamic scope rules**, bindings depend on the current state of program execution

- They cannot always be resolved by examining the program because they are dependent on calling sequences
- To resolve a reference, we use the most recent, active binding made at run time

Binding of Referencing Environments

Accessing variables with dynamic scope:

- keep a stack (*association list*) of all active variables
  - When you need to find a variable, hunt down from top of stack
  - This is equivalent to searching the activation records on the dynamic chain

Binding of Referencing Environments

Accessing variables with dynamic scope:

- keep a central table with one slot for every variable name
  - If names cannot be created at run time, the table layout (and the location of every slot) can be fixed at compile time
  - Otherwise, you'll need a hash function or something to do lookup
  - Every subroutine changes the table entries for its locals at entry and exit (push / pop on a stack).

Type Systems, Type Checking, EqualityTesting and Assignment.

Type

A type, also known as a data type

Classifies into one of various types of data.

Semantic meaning of that structure, and how the values of that structure can be stored in memory.

If this sounds confusing, just think about Integers, Strings, Floats, and Booleans – those are all types. Types can be broken down into categories:

Type

Primitive types – these range based on language, but some common primitive types are integers, booleans, floats, and characters.

Composite types – these are composed of more than one primitive type, e.g. an array or record (not a hash, however). All composite types are considered **data structures**

Abstract types – types that do not have a specific implementation (and thus can be represented via multiple types), such as a hash, set, queue, and stack.

Other types – such as **pointers** (a type which holds as its value a reference to a different memory location) and functions.

Vinita Mishra

Purpose of types

To define what the program should do.

- e.g. read an array of integers and return a double

To guarantee that the program is meaningful.

- that it does not add a string to an integer
- that variables are declared before they are used

To document the programmer's intentions.

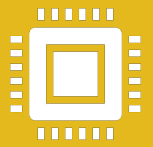
- better than comments, which are not checked by the compiler

To optimize the use of hardware.

- reserve the minimal amount of memory, but not more
- use the most appropriate machine instructions

Type Systems

A type system is a collection of rules that assign a property called type to various constructs in a computer program



reduces the number of bugs by verifying that data is represented properly throughout a program

E.g. variables, expressions, functions or modules

Type Systems

static type checking and dynamic type checking refer to two different **type systems**.

Vinita Mishra

Type Checking

Type Checking is the process of verifying and enforcing the constraints of types, and it can occur either at compile time (i.e. statically) or at runtime (i.e. dynamically).

Type checking is all about ensuring that the program is **type-safe** meaning that the possibility of type errors is kept to a minimum.

A type error is an erroneous program behavior in which an operation occurs (or trys to occur) on a particular data type that it's not meant to occur on.

This could be a situation where an operation is performed on an integer with the intent that it is a float, or even something such as adding a string and an integer together:

What belongs to type checking

Depending on language, the type checker can prevent

- application of a function to wrong number of arguments,
- application of integer functions to floats,
- use of undeclared variables in expressions,
- functions that do not return values,
- division by zero
- array indices out of bounds,
- nonterminating recursion,
- sorting algorithms that don't sort...

Static type Checking

A language is statically-typed if the type of a variable is known at **compile time** instead of at runtime.

Common examples of statically-typed languages include Ada, C, C++, C#, JADE, Java, Fortran, Haskell, ML, Pascal, and Scala.

Vinita Mishra

Static type Checking

Vinita Mishra

Advantages of Static Type Checking:

1) compiler saves information:- if that type of data is according to the operation then compiler saves that information for checking later operations which further no need of compilation.

2) checked execution paths: As static type checking includes all operations that appear in any program statement, all possible execution paths are checked, and further testing for type error is not needed. So no type tag on data objects at run-time are not required, and no dynamic checking is needed.

Dynamic Type Checking

Dynamic type checking is the process of verifying the type safety of a program at **runtime**.

Common dynamically-typed languages include Groovy, JavaScript, Lisp, Lua, Objective-C, PHP, Prolog, Python, Ruby, Smalltalk and Tcl.

Vinita Mishra

Dynamic Type Checking

Most type-safe languages include some form of dynamic type checking, even if they also have a static type checker.

The reason for this is that many useful features or properties are difficult or impossible to verify statically.

For example, suppose that a program defines two types, A and B, where B is a subtype of A. If the program tries to convert a value of type A to type B, which is known as **downcasting**, then the operation is legal only if the value being converted is actually a value of type B. Therefore, a dynamic check is needed to verify that the operation is safe..

Vinita Mishra

Advantages of Dynamic Type

It is much flexible in designing programs or we can say that the flexibility in program design.

In this no declarations are required.

In this type may be changed during execution.

In this programmer are free from most concern about data type.

Vinita Mishra

Disadvantage of Dynamic Type

1) difficult to debug: We need to check program execution paths for testing and in dynamic type checking, program execution path for an operation is never checked.

2) extra storage: Dynamic type checking need extra storage to keep type information during execution.

3) Seldom hardware support : As hardware seldom support the dynamic type checking so we have to implement in software which reduces execution speed.

Vinita Mishra

Equality Testing and Assignment

- The “**=**” is an assignment operator is used to assign the value on the right to the variable on the left.

- E.g.
  - a=10;
  - b=20;
  - c=‘y’;

```
// C program to demonstrate
// working of Assignment operators
#include <stdio.h>
Int main()
{
// Assigning value 10 to a
// using "=" operator
Int a=10;
Printf(“value of a is %d\n”,a);
return 0;
}
```

The ‘==’ operator checks whether the two given operands are equal or not. If so, it returns true. Otherwise it returns false.

```
5==5 This will return true.
```

```
// C program to demonstrate
// working of relational operators
#include <stdio.h>
int main()
{     int a = 10, b = 4;
      // equal to
      if (a == b)
            printf("a is equal to b\n");
      else
            printf("a and b are not equal\n");
      return 0; }
```

=	==
It is an assignment operator.	It is a relational or comparison operator.
It is used for assigning the value to a variable.	It is used for comparing two values. It returns 1 if both the values are equal otherwise returns 0.
Constant term cannot be placed on left hand side. Example: 1=x; is invalid.	Constant term can be placed in the left hand side. Example: 1==1 is valid and returns 1.

The differences can be shown in tabular form as follows:

Subroutines and Control Abstraction

Definitions

Function: subroutine that returns a value

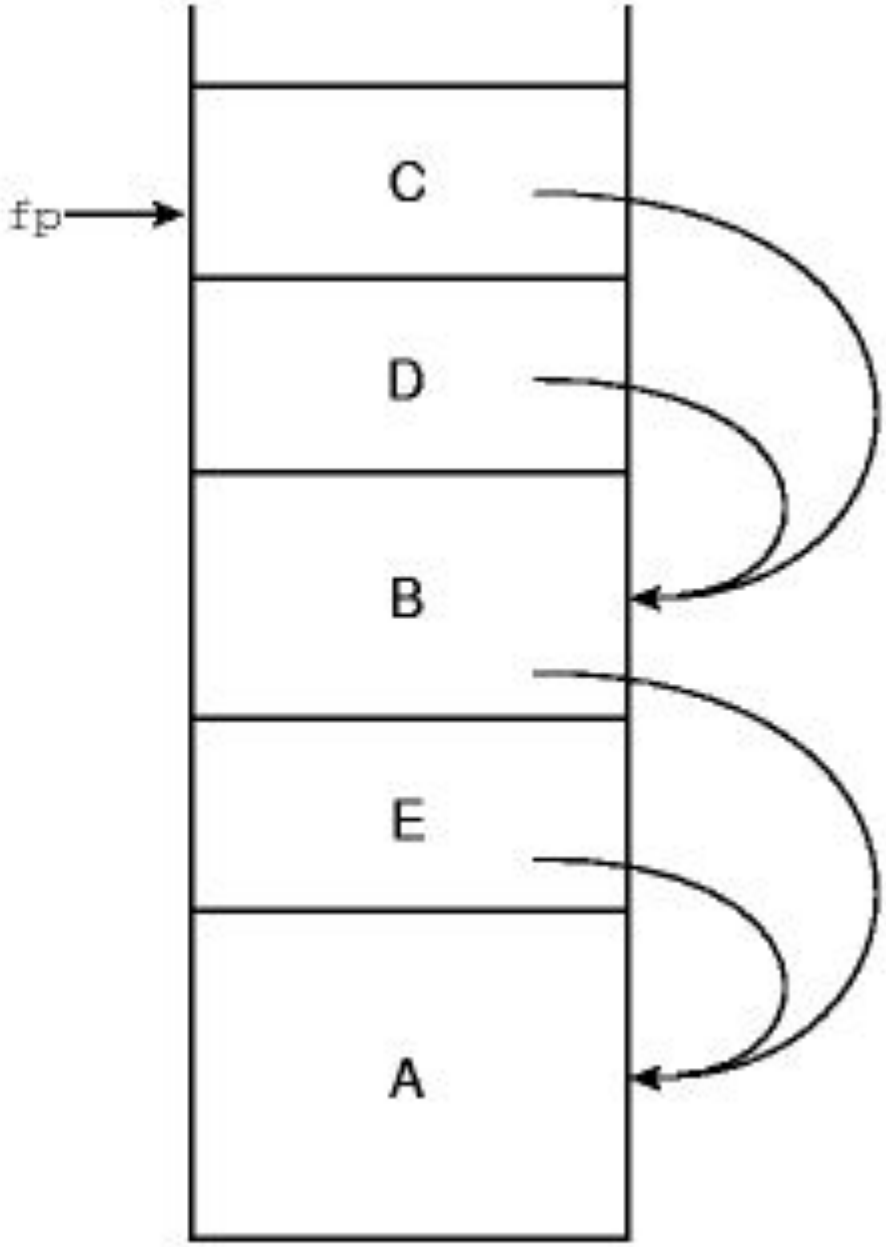
Procedure: subroutine that does not return a value

Review of Stack Layout

- Storage consumed by parameters and local variables can be allocated on a stack
- Activation record contains arguments and/or return values, bookkeeping info, local variables, and/or temporaries
- On return, stack frame is popped from stack

Review of Stack Layout (cont'd)

- Stack pointer: contains address of the first unused location at the top of the stack
- Frame pointer: contains address within frame
- Displacement addressing
- Objects with unknown size placed in variable-size area at top of frame



Review of Stack Layout (cont'd)

Static chain

- Each stack frame contains a reference to the lexically surrounding subroutine

Calling Sequences

- Calling sequence: code executed by the caller immediately before and after a subroutine call
- Prologue
- Epilogue

Calling Sequences (cont'd)

- Tasks on the way in (prologue)
  - Passing parameters, saving return address, changing program counter, changing stack pointer, saving registers, changing frame pointer
- Tasks on the way out (epilogue)
  - Passing return parameters, executing finalization code, deallocating stack frame, restoring saved registers and pc

In-Line Expansion

Allows certain subroutines to be expanded in-line at the point of call

Avoids several overheads (space allocation, branch delays, maintaining static chain or display, saving and restoring registers)

Implementations of In-line Expansion

C++ uses keyword *inline:*

```
inline int max( … ) { … }
```

Ada:

```
pragma inline (function_name);
```

Pragmas make suggestions to the compiler. Compiler can ignore suggestion.

In-line Expansion versus Macros

- in-line expansion
  - Semantically preferable
  - Disadvantages?
- macros
  - Semantic problems
  - Syntactic problems

Parameter Passing Basics

Parameters - arguments that control certain aspects of a subroutine's behavior or specify the data on which they are to operate

Global Variables are other alternative

Parameters increase the level of abstraction

More Basics

```
type small = 1..100;
var  a : 1..10;
     b : 1..1000;
procedure foo (var n : small);
begin
    n := 100;
    writeln (a);
end;
...
a := 2;
foo (b);     (* ok *)
foo (a);     (* static semantic error *)
```

- Formal Parameters - parameter name in the subroutine declaration
- Actual Parameters – values passed to a subroutine in a particular call

Call by Value

For P(X), two options are possible: Call by Value and Call by Reference

Call by Value - provides P with a copy of X's value

Actual parameter is assigned to the corresponding formal parameter when subroutine is called, and the two are independent from then on

Like creating a local or temporary variable

Call by Reference

Call by Reference – provide P with the address of X

The formal parameter refers to the same object as the actual parameter, so that changes made by one can be seen by the other

Language Specific Variations

Pascal: Call by Value is the default, the keyword `VAR` denotes Call by Reference

Fortran: all parameters passed by Reference

Smalltalk, Lisp: Actual Parameter is already a reference to the object

C: always passed by Value

Value vs. Reference

Pass by Value

- Called routine cannot modify the Actual Parameter

Pass by Reference

- Called routine can modify Actual Parameter

Generic subroutines and modules

Exception handling, Co-routines and Events.

Exceptions

Exceptions are an unexpected or unusual condition that arises during program execution.

Most common are various sorts of run-time errors (ex. encountering the end of a file before reading a requested value)

Examples include a user providing abnormal input, a file system error being encountered when trying to read or write a file, or a program attempting to divide by zero.

Handling Exceptions

- Recent languages provide exception-handling facilities where handlers are lexically bound to blocks of code.
- General rule is if an exception isn't handled in the current subroutine, then the subroutine returns and exception is raised at the point of call.
- Keeps propagating up dynamic chain until exception is handled.
- If not handled a predefined outermost handler is invoked which will terminate the program.

Main Handler Uses

1) Perform some operation that allows the program to recover from the exception and continue executing.

2)If recovery isn't possible, handler can print helpful message before termination

3)When exception occurs in block of code but can't be handled locally, it is important to declare local handler to clean up resources and then re-raise the exception to propagate back up.

Exception Propagation

```
try{
...
//protected block of code
...
}catch(end_of_file) {
...
}catch(io_error e) {
//handler for any io_error other than end_of_file
...
}catch(…) {
//handler for any exception not previously named
//… is a valid token in the case in C++, doesn't
//mean code has been left out.
}
```

Coroutines

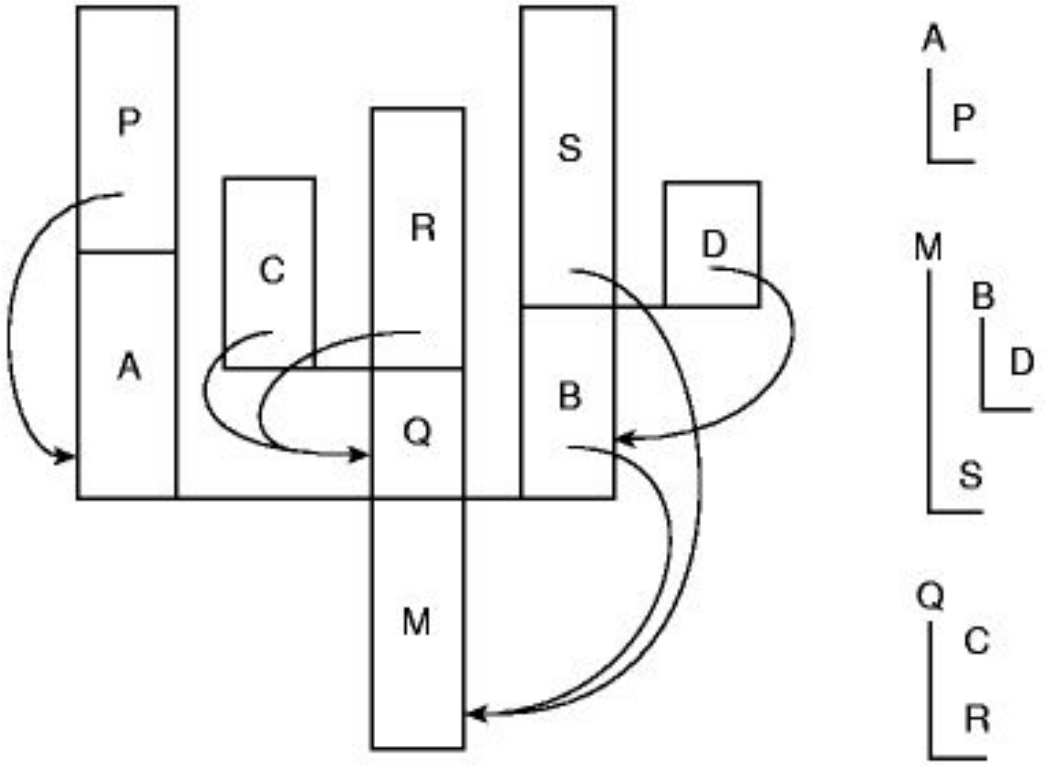
Coroutines

Coroutines are execution contexts that exist concurrently, but execute one at a time, and transfer control to each other explicitly, by name.

They can implement iterators and threads.

Stack Allocation

- Since they are concurrent, they can’t share a single stack because subroutine calls, and returns aren’t LIFO
- Instead the run-time system uses a *cactus stack* to allow sharing.



Transfer

```
transfer:
    push all registers other than sp (including ra)
    *current_coroutine := sp
    current_coroutine := r1        -- argument passed to transfer
    sp := *r1
    pop all registers other than sp (including ra)
    return
```

- To go from one coroutine to another the run-time system must change the PC, stack, and register contents. This is handled in the transfer operation.

Events

- Event driven programming is the programming paradigm in the field of computer science.
- In this type of programming paradigm, flow of execution is determined by the events like user clicks or other programming threads or query result from database.
- Events are handled by event handlers or event callbacks.

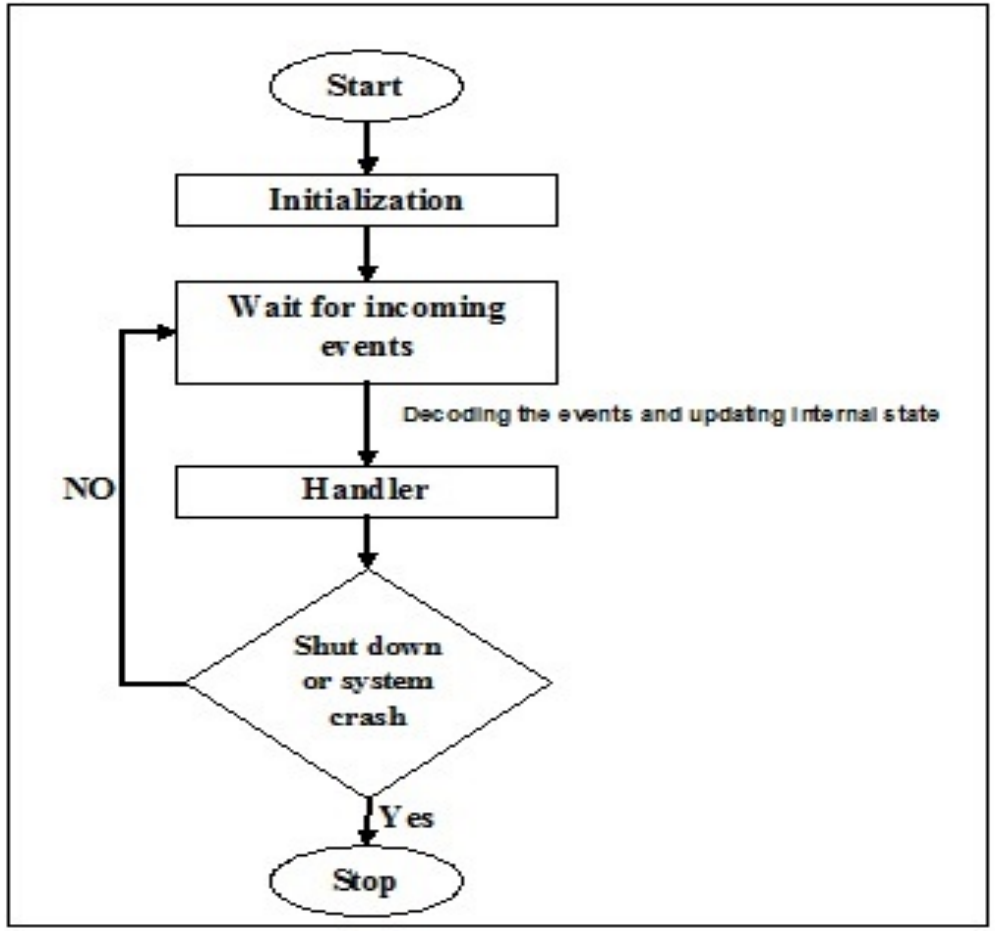
Event-driven programming focuses on events.

Eventually, the flow of program depends upon events.

Event-driven programming depends upon an event loop that is always listening for the new incoming events.

The working of event-driven programming is dependent upon events.

Once an event loops, then events decide what to execute and in what order. Following flowchart will help you understand how this works −


Start
Initialization
Wait for incoming events
Decoding the events and updating internal state
Handler
NO
Shut down or system crash
Yes
Stop

THANKYOU