BITS Pilani
Hyderabad Campus

innovate achieve lead

Principles of Programming Languages (CS F301)

Chapter 5

Jabez Christopher
Assistant Professor
Department of CSIS

Scope

The scope of a variable is the range of statements in which the variable is visible.

A variable is local in a program unit or block if it is declared there

STATIC SCOPE (*a.k.a* lexical scoping)

scope of a variable can be statically determined prior to execution

two categories of static-scoped languages: nested vs. non-nested

If a suitable declaration is not found in the static parent the search continues in to static ancestors (enclosing subprogram)

Scope

The scope of a variable is the range of statements in which the variable is visible.

Blocks (a section in the code)

storage is allocated when the section is entered and deallocated when the section is exited

compound statement - a statement sequence surrounded by matched braces

Scope

The scope of a variable is the range of statements in which the variable is visible.

GLOBAL SCOPE

Definitions can appear outside the functions

visible to those all functions.

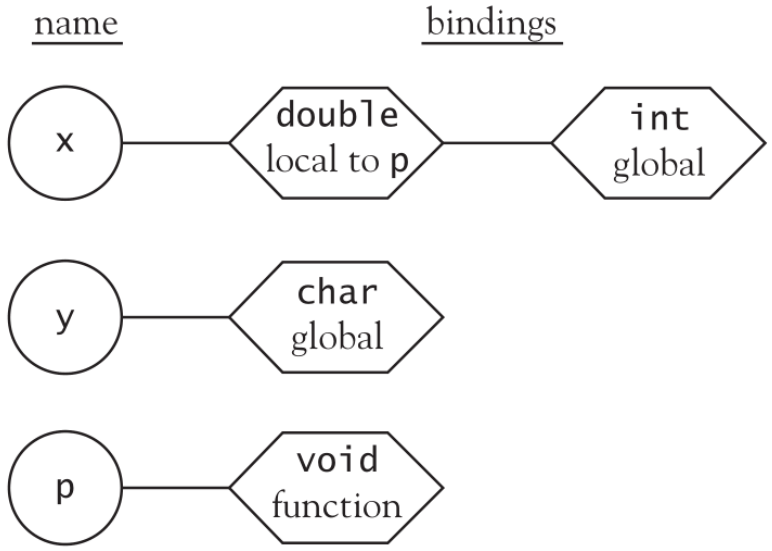
extern qualifier - global variable that is defined after a function can be made visible in the function by declaring it to be external

Scope Resolution Operator (`::`) – Enables the access of a global variable that is hidden by a local with the same name

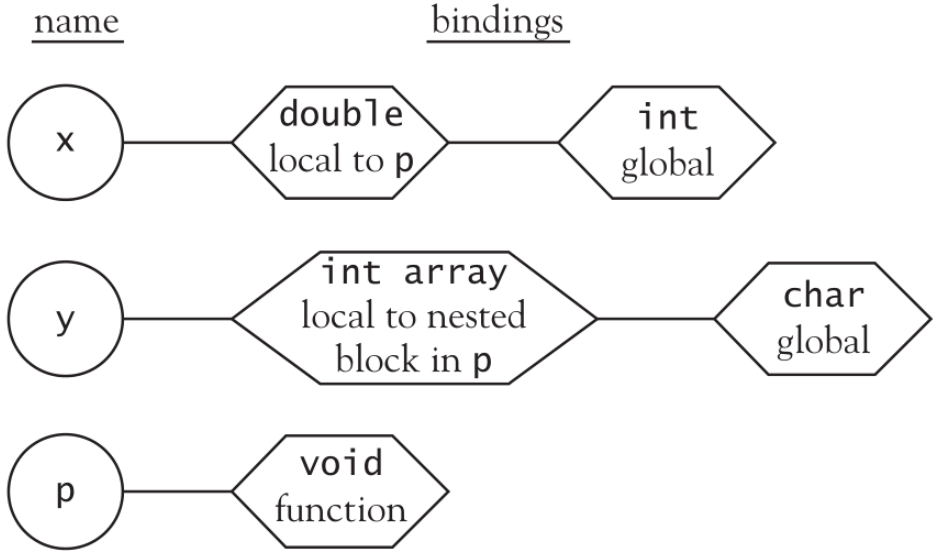
Static Scoping


```
(1)  int x;
(2)  char y;
(3)  void p(){
(4)     double x;
(5)     ...
(6)     {    int y[10];
(7)                ...
(8)     }
(9)     ...
(10) }
(11) void q(){
(12)    int y;
(13)      ...
(14) }
(15) main(){
(16)    char x;
(17)    ...
(18) }
```

Symbol table structure **at line 5**



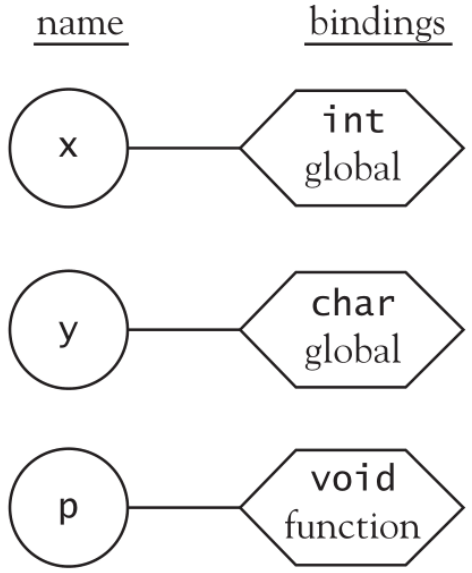
Symbol table structure **at line 7**



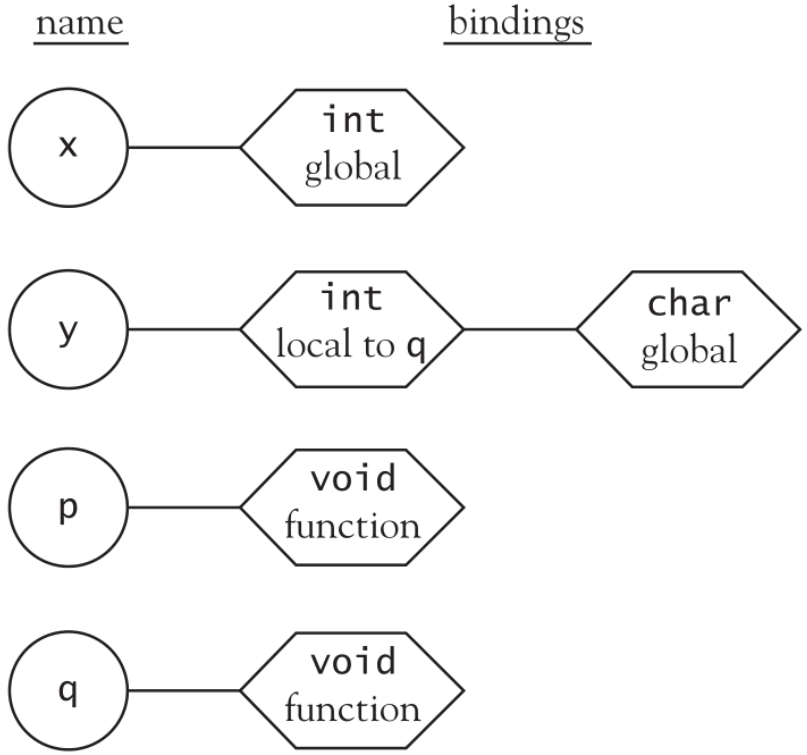
Static Scoping


```
(1)  int x;
(2)  char y;
(3)  void p(){
(4)     double x;
(5)     ...
(6)     {    int y[10];
(7)                ...
(8)     }
(9)     ...
(10) }
(11) void q(){
(12)    int y;
(13)      ...
(14) }
(15) main(){
(16)    char x;
(17)    ...
(18) }
```

Symbol table structure at **line 10**



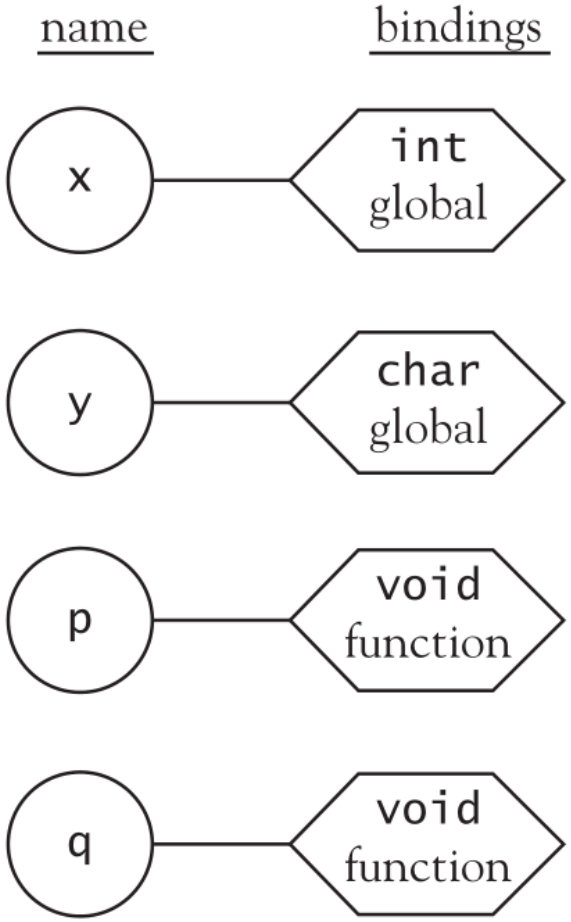
Symbol table structure at **line 13**



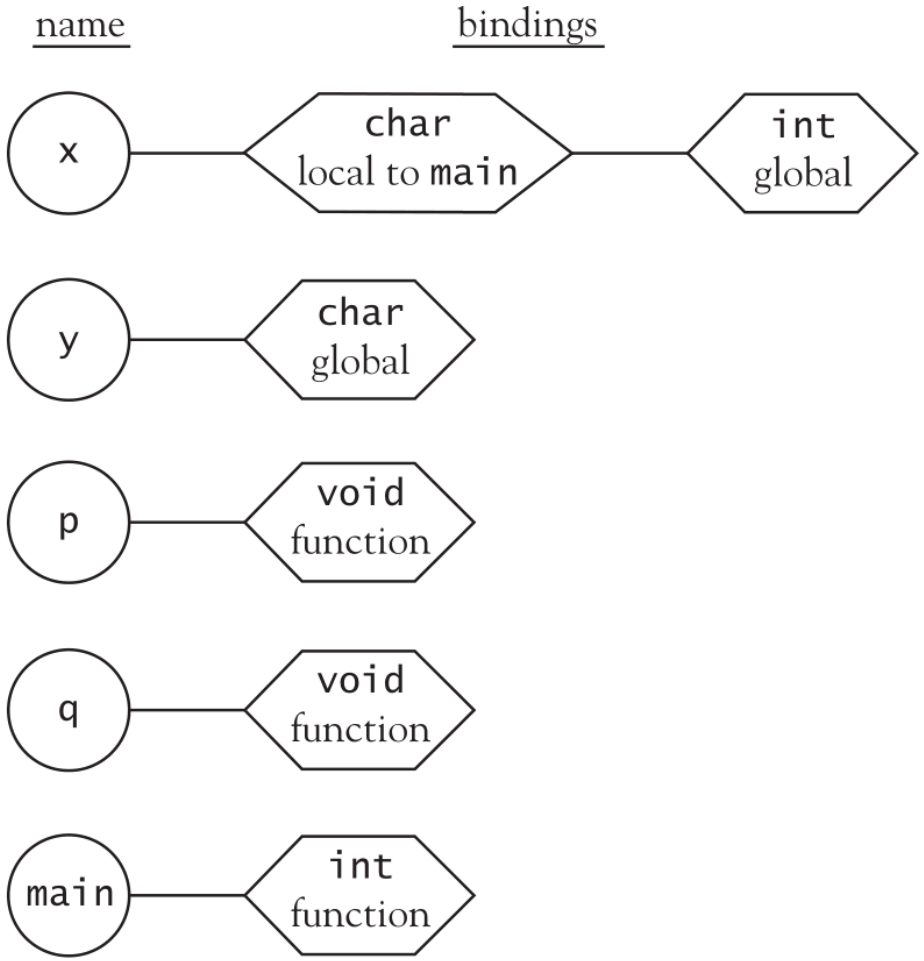
Static Scoping


```
(1)  int x;
(2)  char y;
(3)  void p(){
(4)     double x;
(5)     ...
(6)     {    int y[10];
(7)                ...
(8)     }
(9)     ...
(10) }
(11) void q(){
(12)    int y;
(13)      ...
(14) }
(15) main(){
(16)    char x;
(17)    ...
(18) }
```

Symbol table structure at line 14



Symbol table structure at line 17



Dynamic Scoping

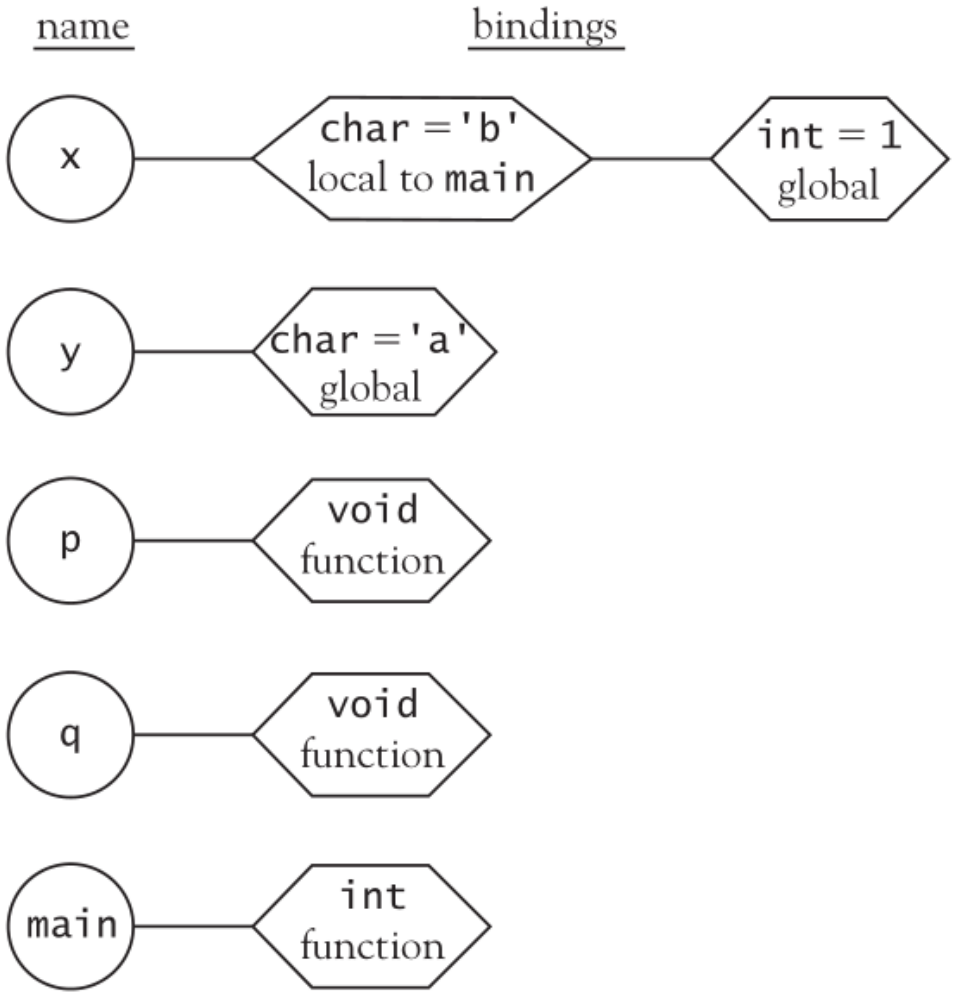
```
(1)  #include <stdio.h>

(2)  int x = 1;
(3)  char y = 'a';
(4)  void p(){
(5)       double x = 2.5;
(6)       printf("%c\n",y);
(7)       { int y[10];
(8)       }
(9)  }

(10) void q(){
(11)      int y = 42;
(12)      printf("%d\n",x);
(13)      p();
(14) }

(15) main(){
(16)      char x = 'b';
(17)      q();
(18)      return 0;
(19) }
```

Symbol table structure at line 17



Dynamic Scoping

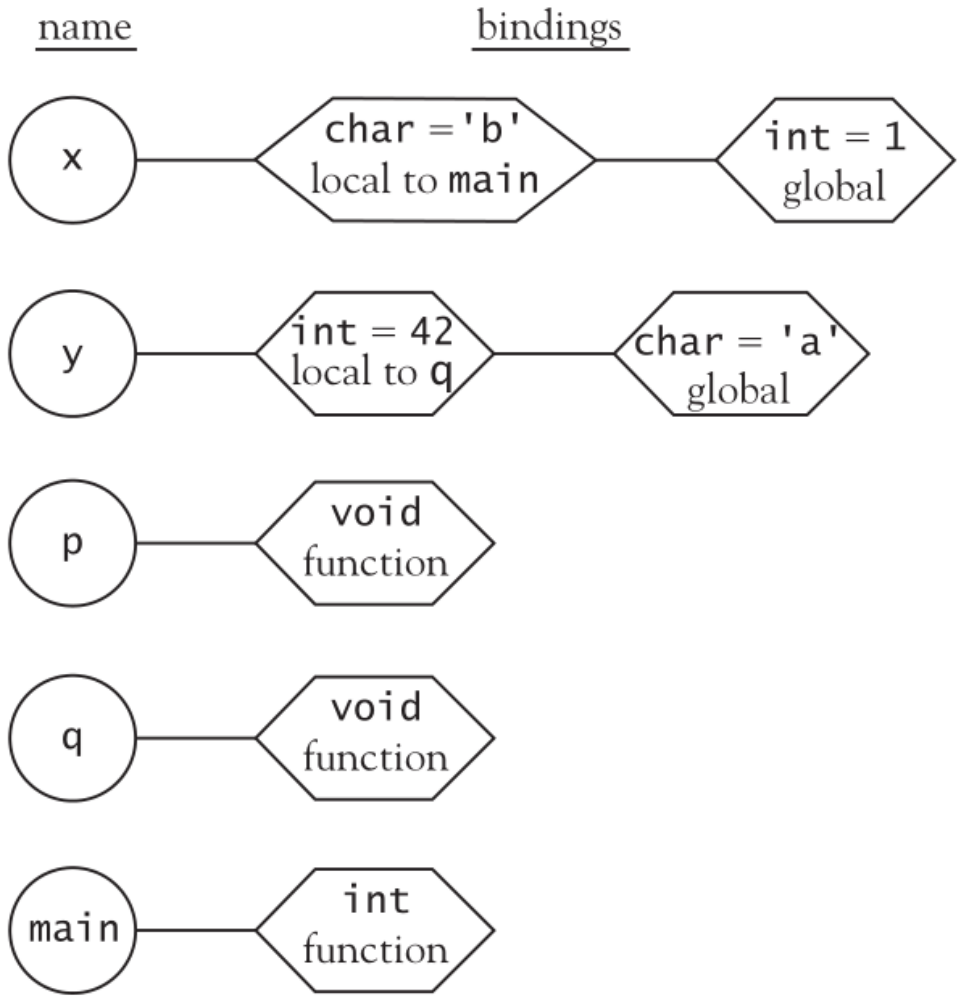
```
(1)  #include <stdio.h>

(2)  int x = 1;
(3)  char y = 'a';
(4)  void p(){
(5)       double x = 2.5;
(6)       printf("%c\n",y);
(7)       { int y[10];
(8)       }
(9)  }

(10) void q(){
(11)      int y = 42;
(12)      printf("%d\n",x);
(13)      p();
(14) }

(15) main(){
(16)      char x = 'b';
(17)      q();
(18)      return 0;
(19) }
```

Symbol table structure at line 12



Dynamic Scoping

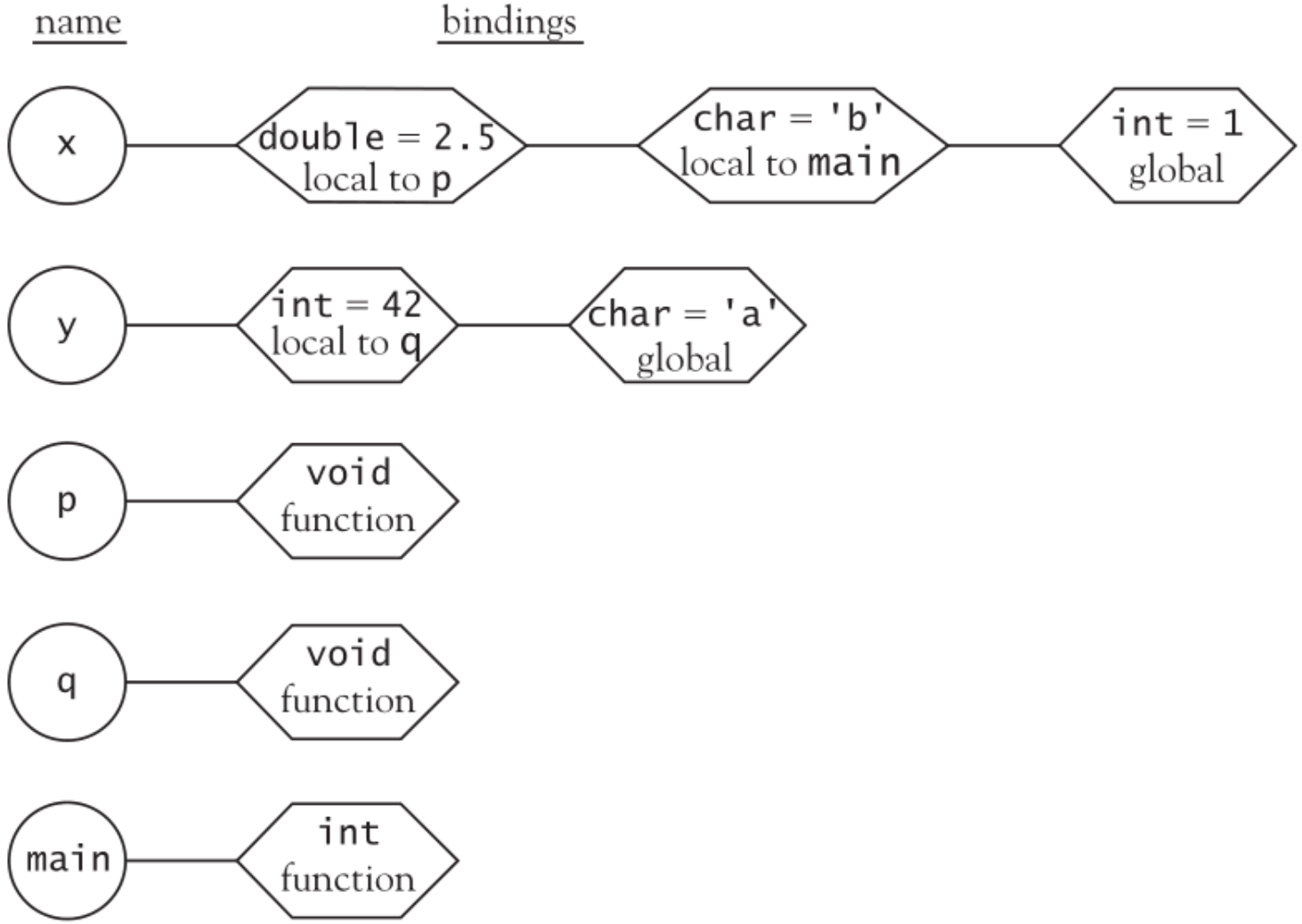
```
(1)  #include <stdio.h>

(2)  int x = 1;
(3)  char y = 'a';
(4)  void p(){
(5)       double x = 2.5;
(6)       printf("%c\n",y);
(7)       { int y[10];
(8)       }
(9)  }

(10) void q(){
(11)      int y = 42;
(12)      printf("%d\n",x);
(13)      p();
(14) }

(15) main(){
(16)      char x = 'b';
(17)      q();
(18)      return 0;
(19) }
```

Symbol table structure at line 6



Referencing Environments

referencing environment of a statement is the collection of all variables that are visible in the statement.

```
g = 3;  # A global

def sub1():
    a = 5;  # Creates a local
    b = 7;  # Creates another local
    ...  <-------------------------------- 1
  def sub2():
    global g;  # Global g is now assignable here

c = 9;  # Creates a new local
...  <-------------------------------- 2
def sub3():
  nonlocal c:  # Makes nonlocal c visible here
  g = 11;  # Creates a new local
  ...  <-------------------------------- 3
```

Named Constants

A named constant is a variable that is bound to a value only once.

The binding of a variable to a value at the time it is bound to storage is called initialization

- **Pros**:
  - Readability
  - Reliability

BITS Pilani
Hyderabad Campus

innovate achieve lead

Principles of Programming Languages (CS F301)

Ch. 6 - DATA TYPES

Jabez Christopher
Assistant Professor
Department of CSIS

INTODUCTION

Data type defines a collection of data values and a set of predefined operations on those values.

User-defined types provide improved readability & modifiability through the use of meaningful names for types.

Abstract data type: interface of a type, which is visible to the user, is separated from the representation and set of operations on values of that type, which are hidden from the user.

Type system of a programming language defines how a type is associated with each expression in the language and includes its rules for type equivalence and type compatibility.

Descriptors are collection of attributes of a variable.

Primitive Data Types

- Types supported directly in hardware of the machine
  - **Integer:** byte, short, int, long, signed, unsigned
  - **Floating Point:** single and double precision
    - stored in 3 parts: sign bit, exponent, mantissa
  - **Complex:** numbers that contain an imaginary part
    - available in languages like Common Lisp, FORTRAN and Python
  - **Decimal:** BCD (2 decimal digits per byte)
    - for business processing whereby numbers (dollars and cents) are stored as 1 digit per ½ byte instead of using an int format
    - popularized in COBOL, also used in PL/I and C#
  - **Boolean:** 1-bit value (usually referenced as true or false)
    - available in C++, Java, Pascal, Ada, Lisp, but not C
    - in Lisp, the values are T or NIL
    - for hardware convenience, these are often stored in 1 byte or 1 word!
  - **Character:** ASCII or Unicode
    - IBM mainframes use a different code called EBCDIC
    - Java, Javascript C# support Unicode

Character String Types

A character string type is one in which the values consist of sequences of characters.

A substring reference is a reference to a substring of a given string.

String data is usually stored in arrays of single characters

Some Lib. functions support NULL character.

Common Lib. Functions: strcpy, strcat , strcmp, strlen

Parameters and return values for most of the string manipulation functions are char pointers that point to arrays of char

Various implementation approaches: Java, Python, perl etc..

String Types

String Length Options

Static length string

Limited dynamic length strings

Dynamic length strings

String Descriptors

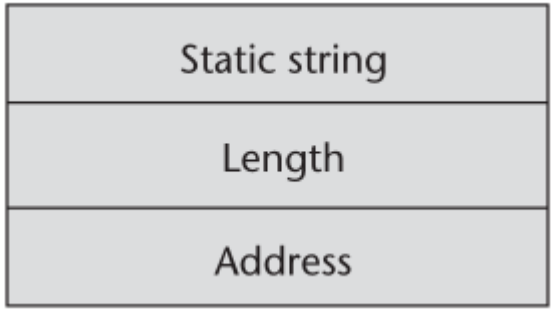
Approaches to support the dynamic allocation

Linked list

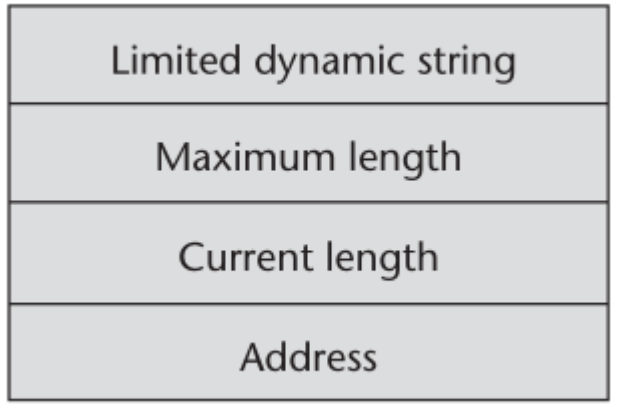
Arrays of pointers to individual characters

Adjacent storage cells

Compile-time descriptor for static strings



Run-time descriptor for limited dynamic strings



Enumeration Types

A Enumeration types provide a way of defining and grouping collections of named constants, which are called enumeration constants.

The enumeration values are coerced to int when they are put in integer context. This allows their use in any numeric expression.

Pros:

Readability is enhanced very directly:

Named values are easily recognized

Syntax of Enumeration Types

- Enumeration-controlled loops:

```
for today := mon to fri do begin ...
```

- Enumerations to be used to index arrays

```
var daily_attendance : array [weekday] of integer;
```

- Enumerations as declare a collection of constants

```
const sun =0;mon =1;tue =2;wed =3;thu =4;fri =5;sat= 6;
```

- Enumeration In C Two approaches.

```
enum weekday{sun,mon,tue,wed,thu,fri,sat};
```

is essentially equivalent to

```
typedef int weekday;
const weekday sun = 0, mon = 1, tue = 2, wed = 3, thu =
4, fri = 5, sat = 6
```

Array Types

Homogeneous aggregate of data elements

References to individual array elements are specified using subscript expressions

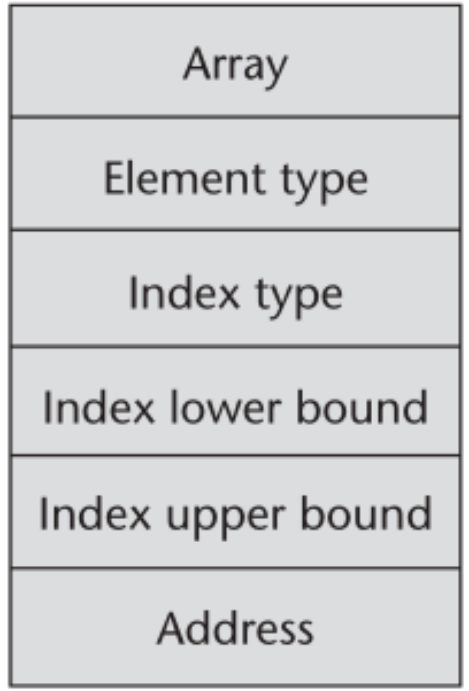
Pointers and references are restricted to point a single type.

Type of the subscripts

Taxonomy:

- static array
- fixed stack-dynamic array
- stack-dynamic array
- fixed heap-dynamic array
- heap-dynamic array

Compile-time descriptor for single-dimensioned arrays



Associative array

An unordered collection of data elements that are indexed by an equal number of values called keys

In Perl, associative arrays are called hashes

The key value is placed in braces and the hash name is replaced by a scalar variable name

Elements can be added using assignment operator and deleted using delete operator.

These are analogous to Dictionaries in Python

Record Types

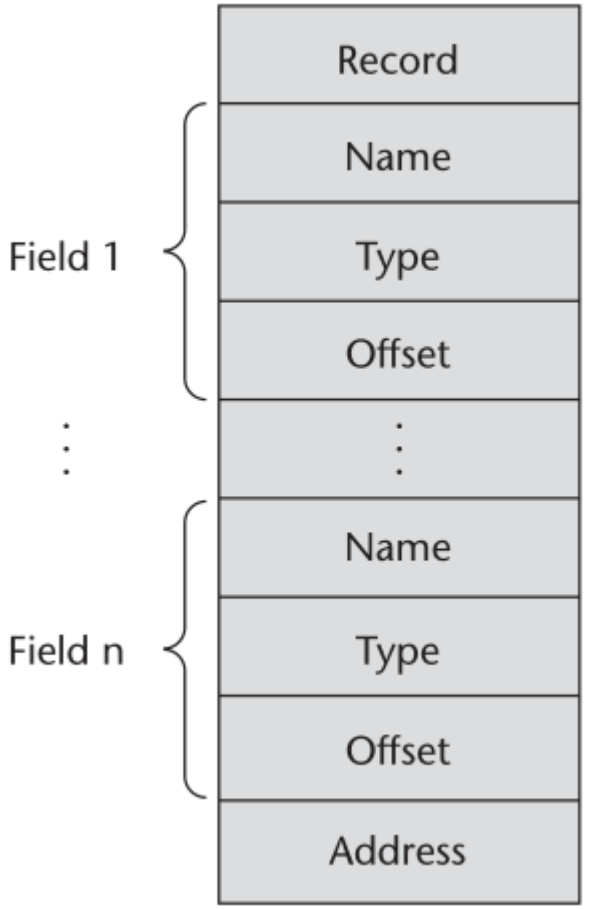
Record is an aggregate of data elements

Identified by names and accessed through offsets from the beginning of the structure.

Analogous to struct datatype in C-based Languages.

Dot notation is used for field references

A compile-time descriptor for a record

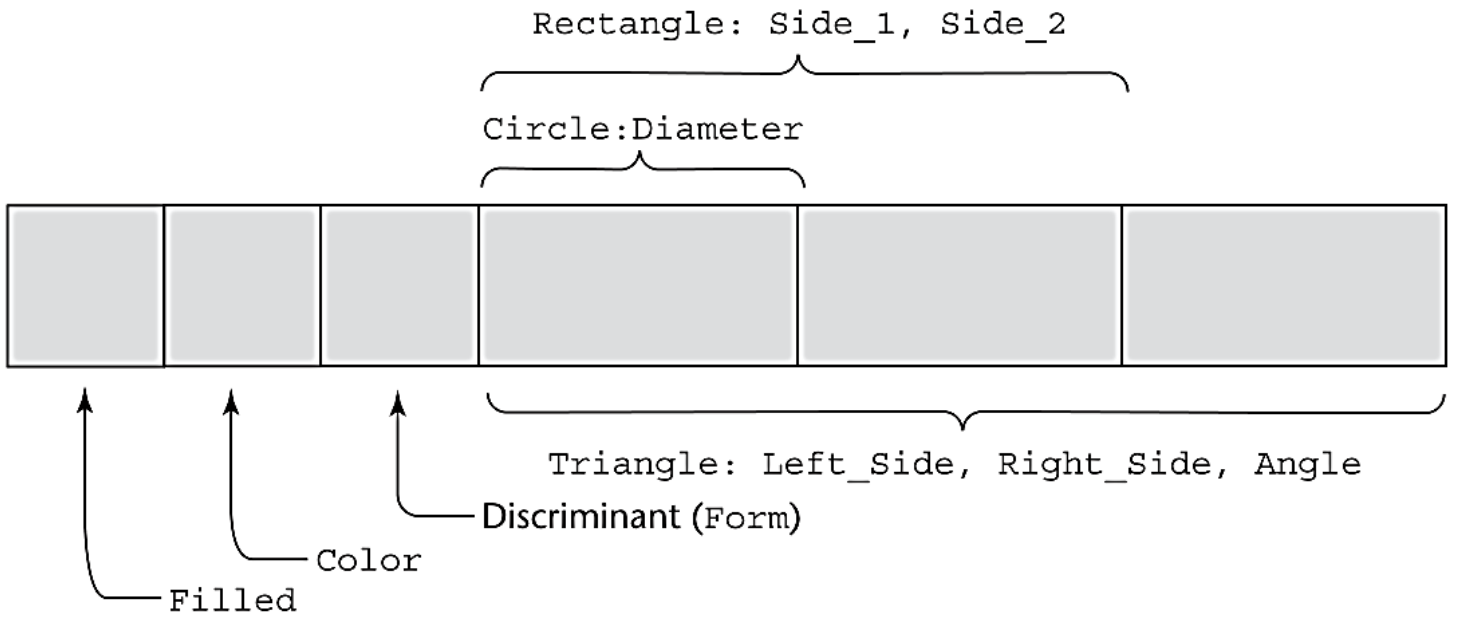


Union

```
type Shape is (Circle, Triangle, Rectangle);
type Colors is (Red, Green, Blue);
type Figure (Form : Shape) is
  record
    Filled : Boolean;
    Color : Colors;
    case Form is
      when Circle =>
        Diameter : Float;
      when Triangle =>
        Left_Side : Integer;
        Right_Side : Integer;
        Angle : Float;
      when Rectangle =>
        Side_1 : Integer;
        Side_2 : Integer;
    end case;
  end record;
```

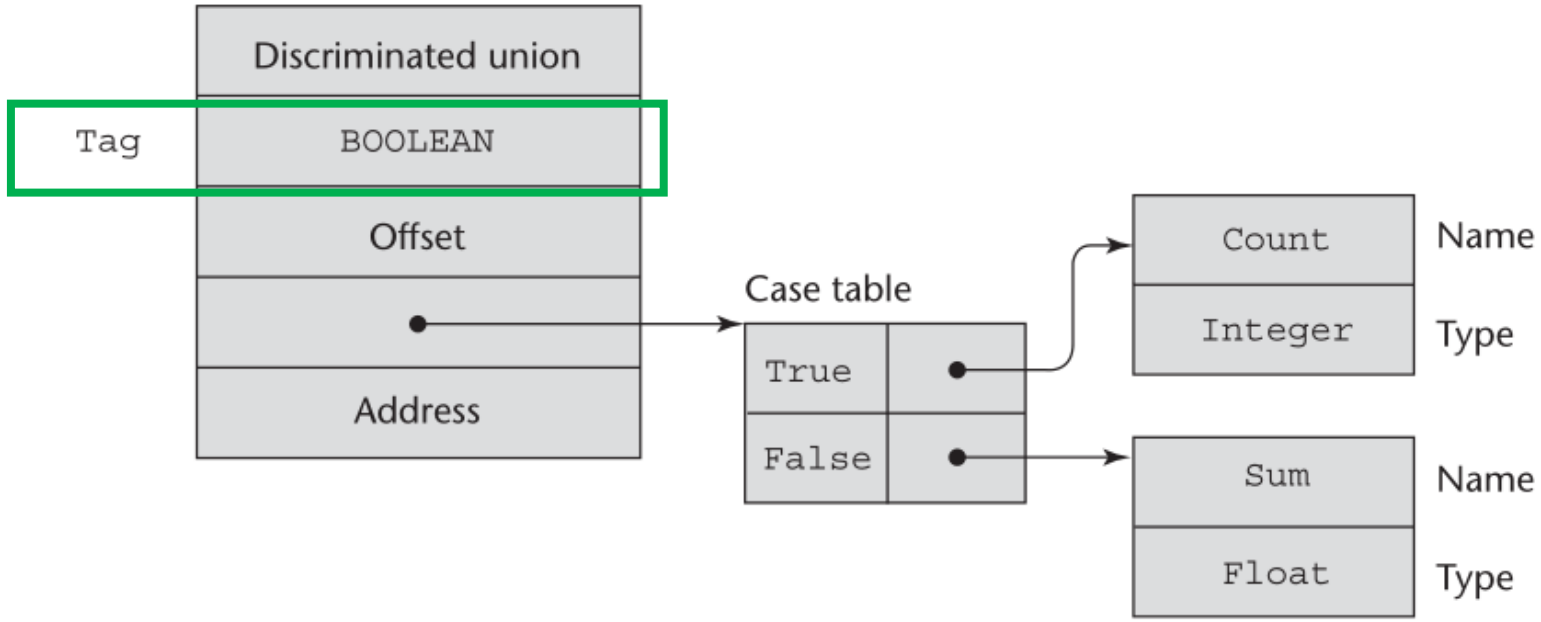
```
Figure_1 : Figure;
Figure_2 : Figure(Form => Triangle);

Figure_1 := (Filled => True,
             Color => Blue,
             Form => Rectangle,
             Side_1 => 12,
             Side_2 => 3);
```



Union

```
type Node (Tag : Boolean) is
  record
  case Tag is
       when True => Count : Integer;
       when False => Sum : Float;
  end case;
 end record;
```



Unions: Example – Info about employees


```
If grade=HSK(Highly skilled)
hobby name
credit card number
If Grade=SSK(Semi skilled)
Vehicle no.
Distance from Company
```

We can use a single structure for it but then it would lead to wastage of memory coz either hobby name & credit card no. or vehicle no. & distance from com. is used at a time.
Both of them are never used simultaneously.
Union inside structure can be used effectively:

```
struct info1
{
char hobby[10];
int crcardno;
};
struct info2
{
char vehno[10];
int dist;
};
```

```
union info
{
struct info1 a;
struct info2 b;
};
struct emp
{
char n[20];
char grade[4];
int age;
union info f;
};
struct emp e;
```

Tuple Types

The elements of a tuple can be referenced with indexing in brackets

Tuples can be catenated with the plus (+) operator

They can be deleted with the del statement.

F# also has tuples

```
let tup = (3, 5, 7);;
let a, b, c  = tup;;
```

```
# Python code to test that
# tuples are immutable

tuple1 = (0, 1, 2, 3)
tuple1[0] = 4
print(tuple1)
```

Mutable Objects : These are of type list, dict, set
Custom classes are generally mutable.

```
# Python code to test that
# lists are mutable
color = ["red", "blue", "green"]
print(color)

color[0] = "pink"
color[-1] = "orange"
print(color)
```

List Types

Lists in LISP and Scheme are delimited by parentheses and use no commas

`(A B C D)` and `(A (B C) D)`

Data and code have the same form

As data, `(A B C)` is literally what it is

As code, `(A B C)` is the function `A` applied to the parameters `B` and `C`

The interpreter needs to know which a list is, so if it is data, we quote it with an apostrophe

`'(A B C)` is data

List Types

List Operations in Scheme

- `CAR` returns the first element of its list parameter

  `(CAR '(A B C))` returns `A`

- `CDR` returns the remainder of its list parameter after the first element has been removed

  `(CDR '(A B C))` returns `(B C)`

- `CONS` puts its first parameter into its second parameter, a list, to make a new list

  `(CONS 'A (B C))` returns `(A B C)`

- `LIST` returns a new list of its parameters

  `(LIST 'A 'B '(C D))` returns `(A B (C D))`

List Types

C-based languages

- `int list [] = {1, 3, 5, 7}`
- `char *names [] = {"Mike", "Fred", "Mary Lou"};`

Ada

```
- List : array (1..5) of Integer :=
      (1 => 17, 3 => 34, others => 0);
```

Python

- `list = [x ** 2 for x in range(12) if x % 3 == 0]`
- `print (list)`
- **Output ➔ [0, 9, 36, 81]**

Pointer and Reference Types

- A pointer type variable has a range of values that consists of memory addresses and a special value, *null.*
- Provide the power of indirect addressing.
- Provide a way to manage dynamic memory.
- A pointer can be used to access a location in the area where storage is dynamically created (usually called a heap).

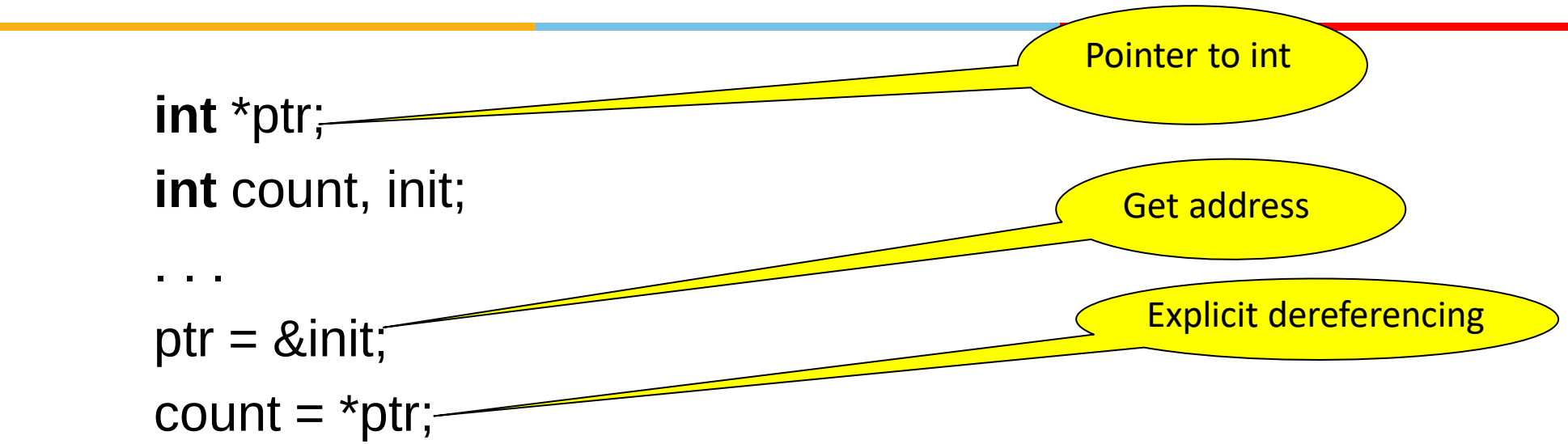
Design Issues

- What are the scope and lifetime of a pointer variable?
- What is the lifetime of a heap -dynamic variable?
- Are the pointers restricted to the type of value they point to?
- Are pointers used dynamic storage management

Pointer Operations

- Two fundamental operations: assignment and dereferencing.
- Assignment is used to set a pointer variable's value to some useful address.
- Dereferencing yields the value stored at the location represented by the pointer's value.
- Dereferencing can be explicit or implicit.

Pointers in C and C++




```
int *ptr;
int count, init;
. . .
ptr = &init;
count = *ptr;
```

```
//POINTER ARITHMETIC
float stuff[100];
float *p;
p = stuff;
```

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

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

Problems with pointers

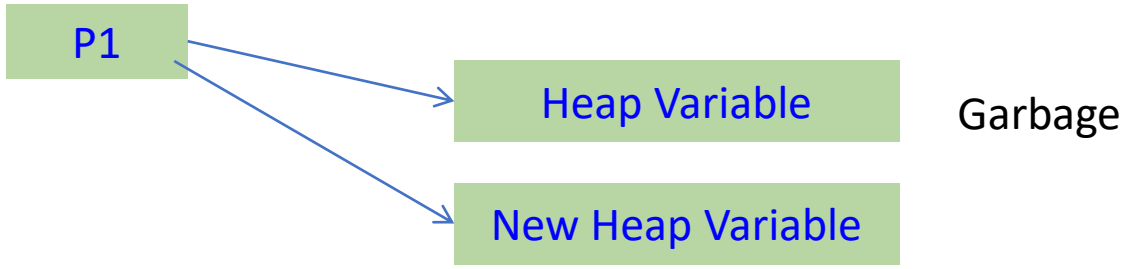
A **lost heap-dynamic variable** is an allocated heap-dynamic variable that is no longer accessible to the user program.

Pointer p1 is set to point to a newly created heap-dynamic variable.

p1 is later set to point to another newly created heap-dynamic variable.

The first heap-dynamic variable is now inaccessible, or lost.

This is sometimes called **memory leakage**.



- A **dangling pointer**, or **dangling reference**, is a pointer that contains the address of a heap-dynamic variable that has been deallocated.

- A new heap-dynamic variable is created and pointer p1 is set to point at it.
- Pointer p2 is assigned p1's value.
- The heap-dynamic variable pointed to by p1 is explicitly deallocated (possibly setting p1 to nil), but p2 is not changed by the operation.
- P2 is now a dangling pointer. If the deallocation operation did not change p1, both p1 and p2 would be dangling.

There are **three** different ways where Pointer acts as dangling pointer
1/3: **De-allocation of memory**

```
// Deallocating a memory pointed by ptr causes  dangling pointer
#include <stdlib.h>
#include <stdio.h>
int main()
{
        int *ptr = (int *)malloc(sizeof(int));

        // After below free call, ptr becomes a
        // dangling pointer
        free(ptr);

        // No more a dangling pointer
        ptr = NULL;
}
```

There are **three** different ways where Pointer acts as dangling pointer 2/3: **Function Call**

// The pointer pointing to local variable becomes dangling when local variable is not static.

```
#include<stdio.h>

int *fun()
{
        // x is local variable and goes out of
        // scope after an execution of fun() is
        // over.
        int x = 5;

        return &x;
}
```

```
// Driver Code
int main()
{
        int *p = fun();
        fflush(stdin);

        /* p points to something which is
        not valid anymore */
        printf("%d", *p);
        return 0;
}
```

The problem doesn’t appear (or p doesn’t become dangling) if x is a static variable.
// The pointer pointing to local variable doesn't become dangling when local variable is static.

```
#include<stdio.h>

int *fun()
{
        // x now has scope throughout the program
        static int x = 5;

        return &x;
}
```

```
int main()
{
        int *p = fun();
        fflush(stdin);

        // Not a dangling pointer as it points
        // to static variable.
        printf("%d",*p);
}
```

There are **three** different ways where Pointer acts as dangling pointer
3/3: **Variable goes out of scope**

```
void main()
{
  int *ptr;
  .....
  .....
  {
     int ch;
     ptr = &ch;
  }
  .....
  // Here ptr is dangling pointer
}
```

VOID POINTER

A pointer that points to some data location in storage, which doesn't have any specific type.

Void refers to the type

```
#include<stdlib.h>

int main()
{
        int x = 4;
        float y = 5.5;

        //A void pointer
        void *ptr;
        ptr = &x;

        // (int*)ptr - does type casting of void
        // *((int*)ptr) dereferences the typecasted
        // void pointer variable.
        printf("Integer variable is = %d", *( (int*) ptr) );

        // void pointer is now float
        ptr = &y;
        printf("\nFloat variable is= %f", *( (float*) ptr) );

        return 0;
}
```

NULL POINTER

NULL Pointer is a pointer which is pointing to nothing. In case, if we don't have address to be assigned to a pointer, then we can simply use NULL.

```
#include <stdio.h>
int main()
{
        // Null Pointer
        int *ptr = NULL;

        printf("The value of ptr is %u", ptr);
        return 0;
}
```

1.NULL vs Uninitialized pointer
2. NULL vs Void Pointer

1.An uninitialized pointer stores an undefined value. A null pointer stores a defined value, but one that is defined by the environment to not be a valid address for any member or object.

2.Null pointer is a value, while void pointer is a type

WILD POINTER

A pointer which has not been initialized to anything (not even NULL) is known as wild pointer. The pointer may be initialized to a non-NULL garbage value that may not be a valid address.

```
int main()
{
        int *p; /* wild pointer */
        int x = 10;
        p = &x; // p is not a wild pointer now
        return 0;
}
```

Generally, compilers warn about the wild pointer.

Solutions to the Dangling-Pointer Problem

- *Tombstone*: extra heap cell that is a pointer to the heap-dynamic variable
  - The actual pointer variable points only at tombstones
  - When heap-dynamic variable de-allocated, tombstone remains but set to nil
  - **Costly in time and space**

dangling–pointer

This approach prevents a pointer from ever pointing to a deallocated variable

Solutions to the Dangling-Pointer Problem

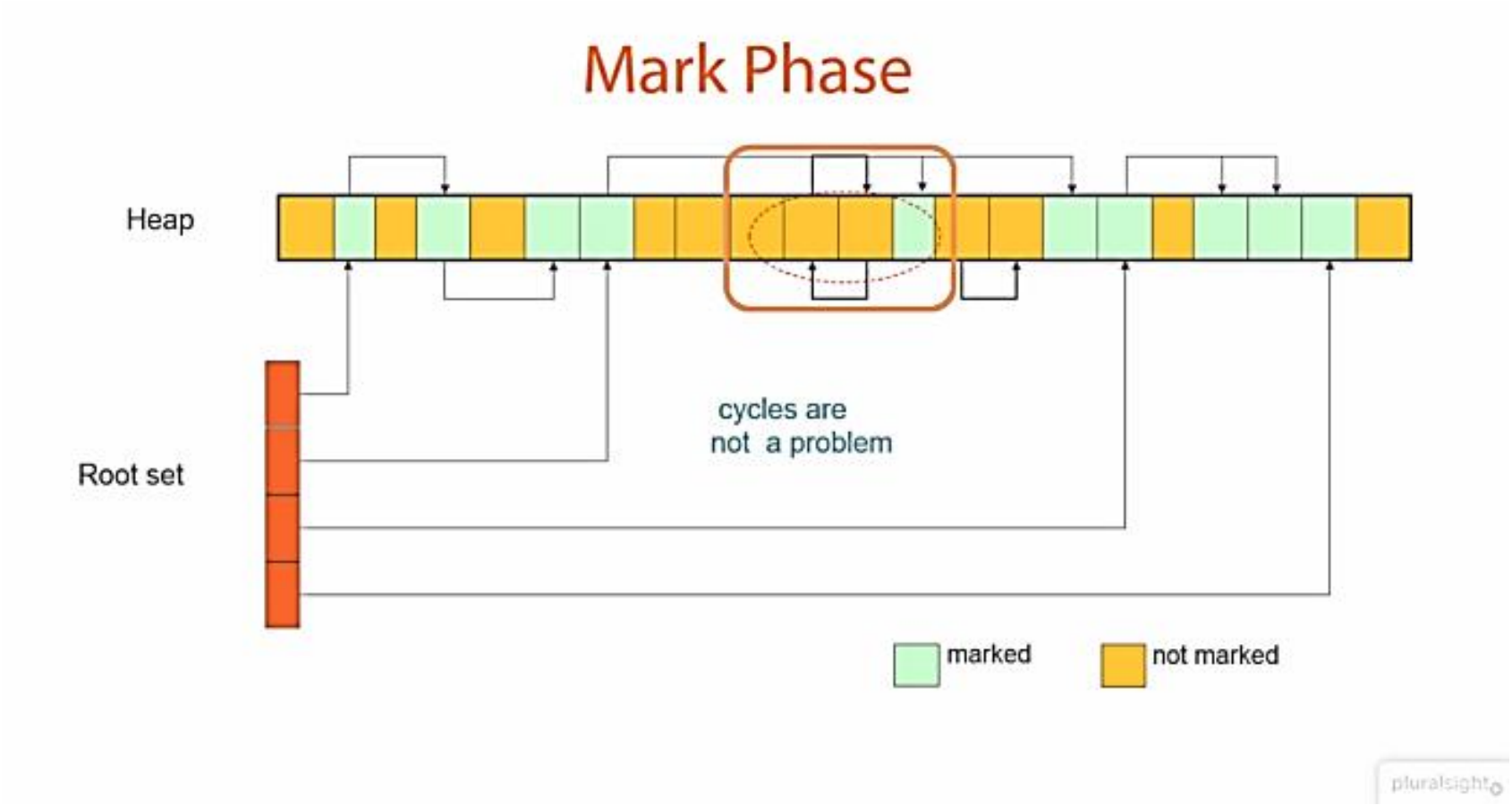
An alternative to tombstones is the **locks-and-keys approach** used in the implementation of **UW-Pascal** (Fischer and LeBlanc, 1977, 1980).

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

- Heap-dynamic variables are represented as variable plus cell for integer lock value
- When heap-dynamic variable allocated, lock value is created and placed in lock cell and key cell of pointer

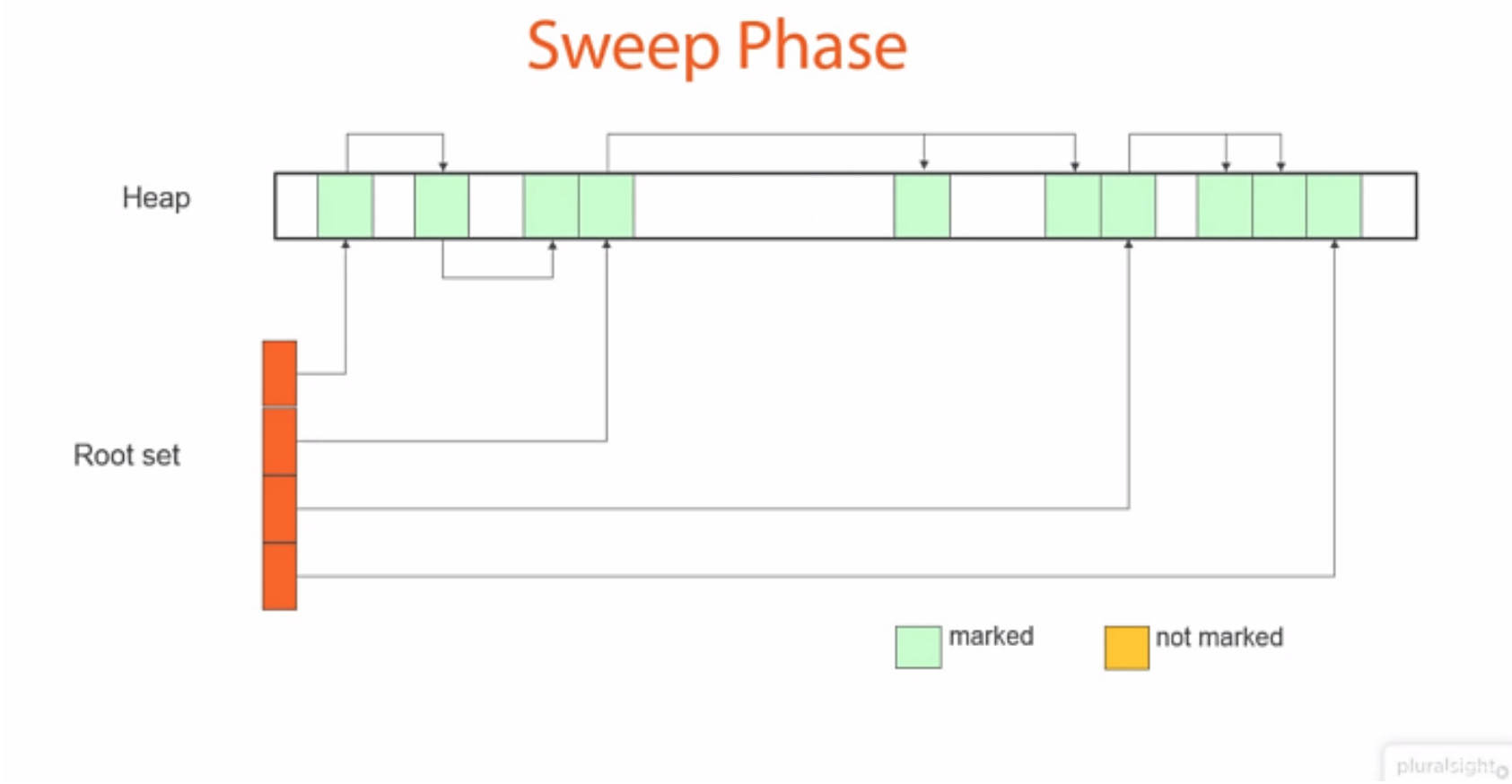
Solutions to the Dangling-Pointer Problem

GARBAGE COLLECTION

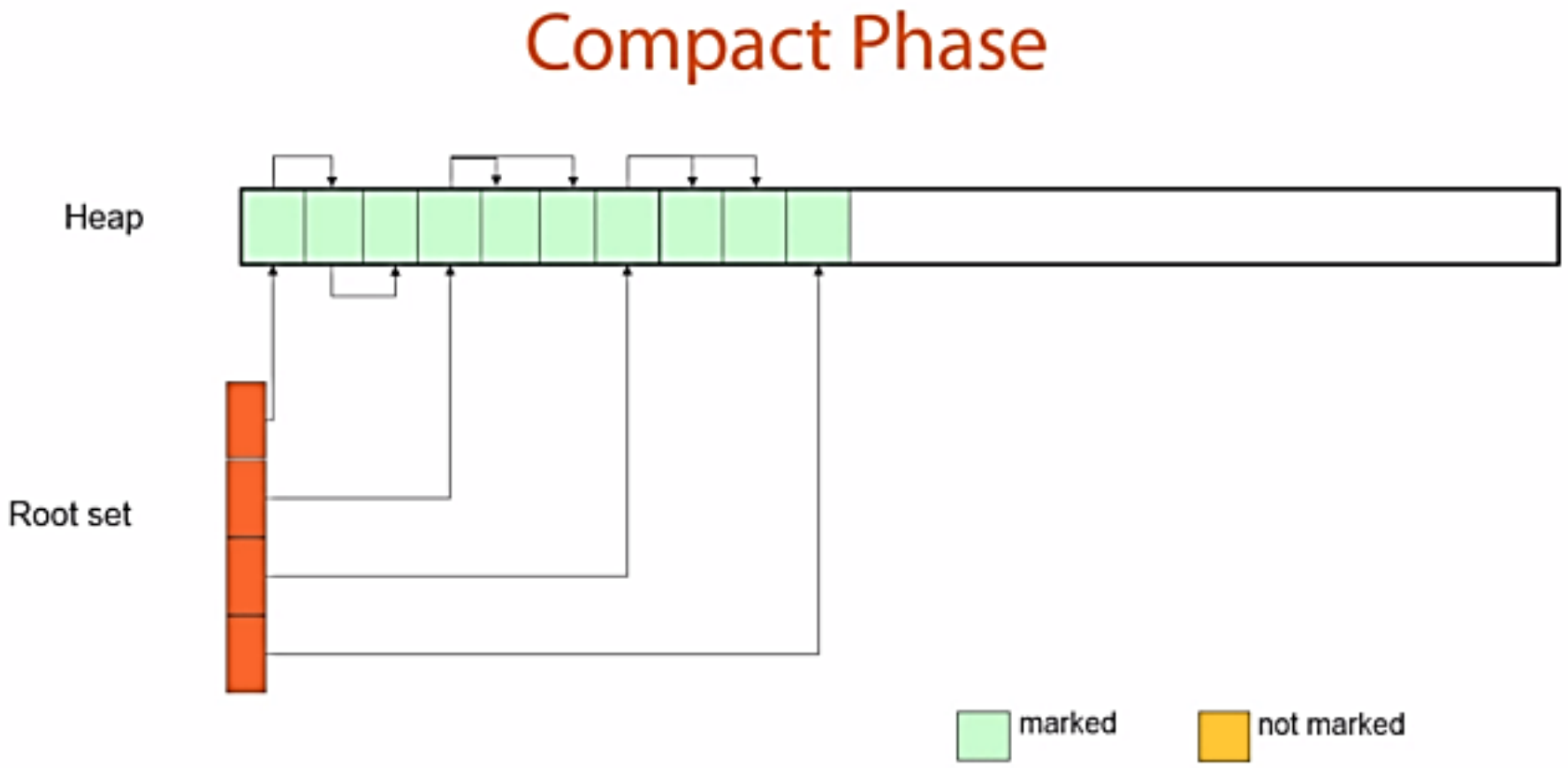


Solutions to the Dangling-Pointer Problem

GARBAGE COLLECTION



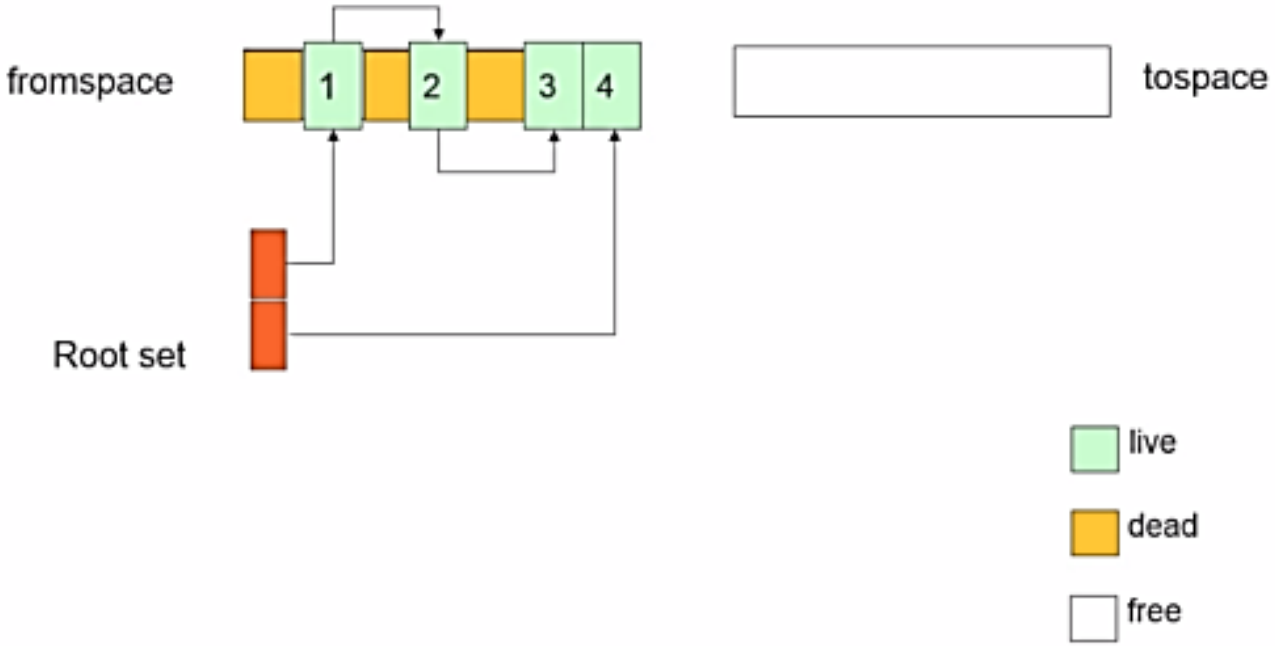
GARBAGE COLLECTION

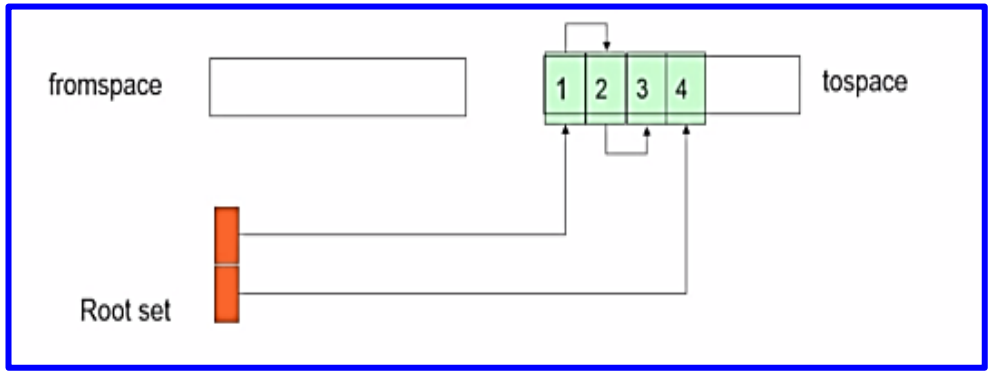


Solutions to the Dangling-Pointer Problem

GARBAGE COLLECTION

Copying Fromspace to Tospace





```
{
int b[3][3]={
        1,2,3,
        4,5,6,
        7,8,9
        };
                cout<<"content of b:\nstart addr of 2D array:  "<<b<<endl;
                cout<<"content of b[0]:\nstart addr of 1st 1D array: "<<b[0]<<endl;
                cout<<"content of b[0][0]:\nfirst value of 2D array \n(or)\n first
                                        value in 1D array b[0]: "<<b[0][0]<<endl;
                cout<<"address of b[0][0]:\n"<<&b[0][0]<<endl;
                cout<<"size of b, 2D array"<<sizeof(b)<<endl;
                cout<<"size of b[0], 1st 1D array"<<sizeof(b[0])<<endl;
                cout<<"size of b[1], 2nd 1D array"<<sizeof(b[1])<<endl;
                cout<<"size of an element b[0][0]"<<sizeof(b[0][0])<<endl;
                cout<<"==============**************============\n";
                cout<<"start address of first 2d array b[0]\t"<<b[0]<<endl;
                cout<<"adding 1 to b\t"<<b+1<<endl;
                cout<<"start address of second 2d array b[1]\t"<<b[1]<<endl;
                cout<<"adding 2 to b\t"<<b+2<<endl;
                cout<<"start address of third 2d array b[2]"<<b[2]<<endl;
                cout<<"value in b[2][2]->  "<<b[2][2]<<endl;
                cout<<"address of b[2][2]->  "<<&b[2][2]<<endl;
                cout<<"address of b[2]->  "<<b[2]<<endl;
                cout<<"****address of b[2]->  "<<&b[2]<<endl;
                cout<<"access b[2][2]  using b[2] base addr\t"<<b[2]+2<<endl;
                for(int i=2;i>=0;i--)
                        for(int j=2;j>=0;j--)
                        {  cout<<b[i][j]<<"\t"<<*((b+i)+j)<<endl; }
                }
```

BITS Pilani
Hyderabad Campus

innovate achieve lead

Principles of Programming Languages (CS F301)

SUBPROGRAMS

Jabez Christopher
Assistant Professor
Department of CSIS

INTRODUCTION

- A mechanism in a programming language for abstracting a group of actions or computations.
- Each subprogram has a single entry point.
- The calling program unit is suspended during the execution of the called subprogram, which implies that there is only one subprogram in execution at any given time.
- Control always returns to the caller when the subprogram execution terminates.
- It consists of subprogram definition, subprogram call and subprogram header.
- The parameter profile of a subprogram contains the number, order, and types of its formal parameters.
- The protocol of a subprogram is its parameter profile plus, if it is a function, its return type.
- Functions return values and procedures do not

INTRODUCTION

- A subprogram can gain access to the data in 2 ways:
  - direct access to nonlocal variables
  - through parameter passing
- Parameter passing is more flexible than direct access to nonlocal variables.
- The parameters in the subprogram header are called formal parameters.
- At procedure call, the list of parameters to be bound to formal parameters is called actual parameters.
- Two types: positional parameters & keyword parameters.
- formal parameters can have default values – (Python, Ruby, C++, Fortran 95+ Ada, and PHP)

Local Referencing Environments

Variables that are defined inside subprograms are called local Variables

Local variables can be either static or stack dynamic.

Disadvantages of stack-dynamic local variables:

- Cost of the time required to allocate, initialize, deallocate
- Accesses to stack-dynamic local variables must be indirect

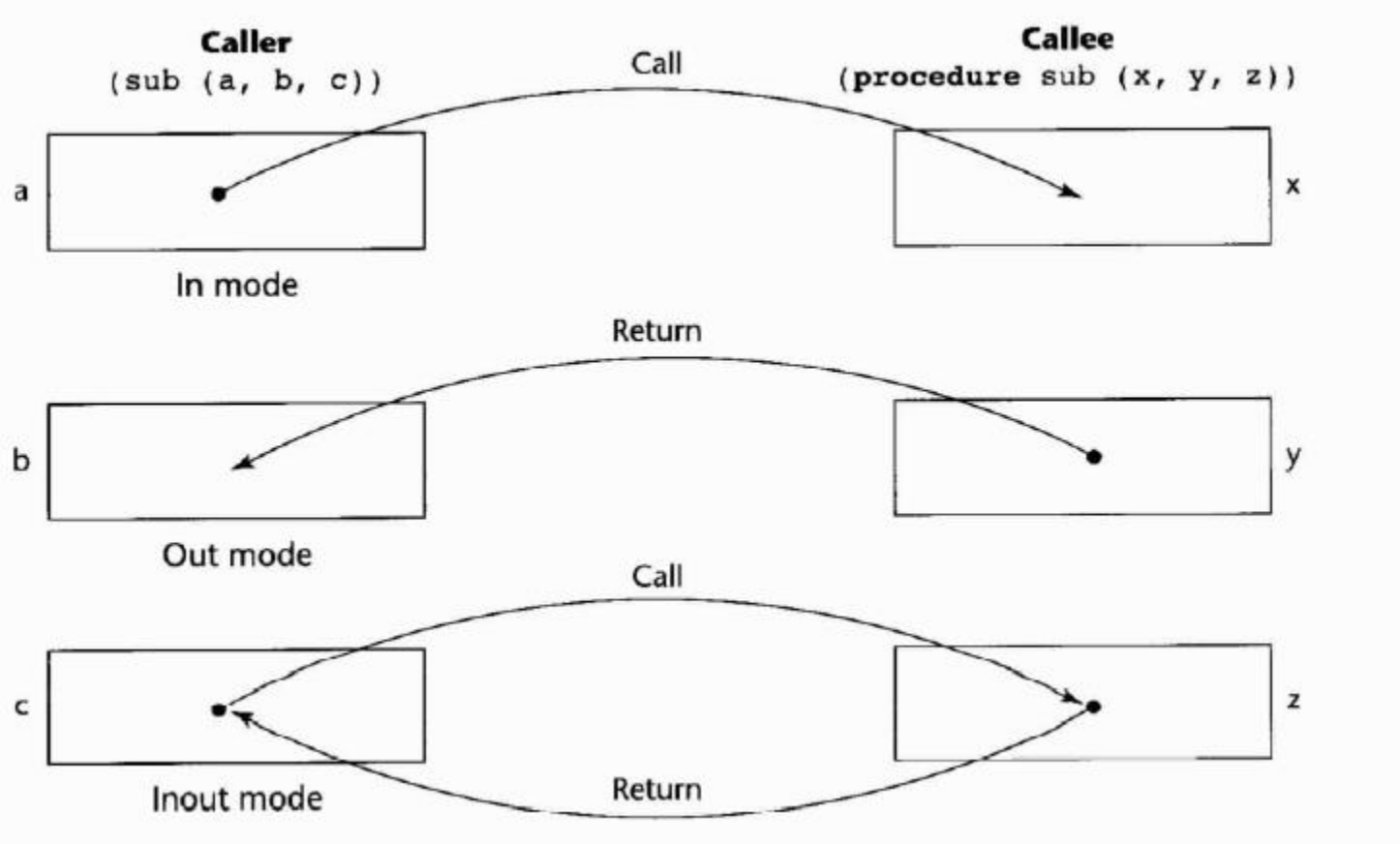
The methods of C++, Java, and C# have only stack-dynamic local variables.

Nested Subprograms

Gives ability to create a hierarchy of both logic and scopes

Recent Languages JavaScript, Python, Ruby, and Lua allow nesting subprograms.

Semantic Models of Parameter Passing



Implementation Models of Parameter Passing

- Pass-by-Value (In Mode)
- Pass-by-Result (Out Mode)
- Pass-by-Value-Result (InOut Mode)
- Pass-by-Reference (InOut Mode, but also In, or Out)
- Pass-by-Name (InOut Mode)

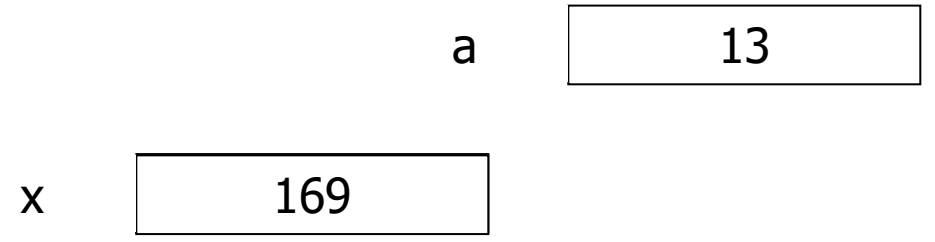
Call-by-value

- Evaluate the actual parameters, assign them to corresponding formal parameters.
- Execute the function body.
- Example Languages: C

Call-by-Value

```
int callee(int x)
{
    x = x * x;
}
```

```
Calling_func()
{
   int a=13;
   callee(a);
   .....
}
```



Call-by-value-result

- Evaluate the actual parameters, assign them to corresponding formal parameters
- Execute the function body.
- The final value of the formals are copied back out to the locations of the actuals.
- Example Languages: Legal ADA
- Support three kinds of parameters
  - **in** parameters
  - **out** parameters
  - **in out** parameters

Call-by-Value-Result

```
int callee(inout int x)
{
    x=x * x;
}
```

```
Calling_func()
{
   int a=13;
   callee(a);
   .....
}
```

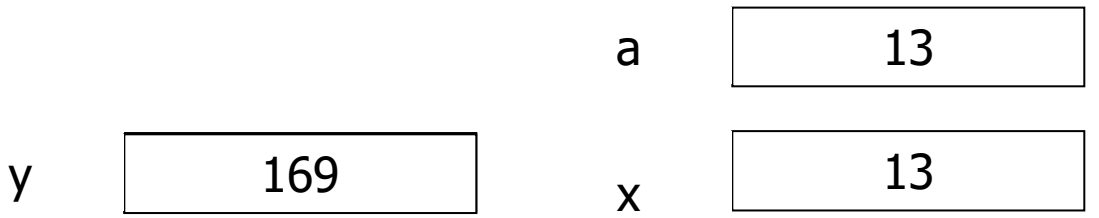
Call-by-Constant Value

- Evaluate the actual parameters, assign them to corresponding formal parameters. The formal parameter is a constant value which cannot be modified.
- Execute the function body.

Call-by-Constant Value

```
int callee(int x)
{
    int y;
    y = x * x;
}
```

```
Calling_func()
{
   int a=13;
   callee(a);
   .....
}
```



Call-by-reference

- Evaluate actual parameters, which must have l-values assign these l-values to l-values of corresponding formal parameters.
- Execute the body.
- Call-by-reference is explicit in C, whereas implicit in C++;

Call-by-Reference

```
int callee(int *x)
{
    (*x)=(*x) * (*x);
}
```

```
Calling_func()
{
   int a=13;
   callee(&a);
   .....
}
```

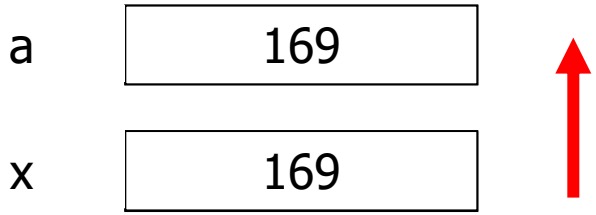
a	169
x	Address of a

Call-by-Result

```
int callee(out int x)
{
    x=x * x;
}
```

```
Calling_func()
{
   int a=13;
   callee(a);
   .....
}
```

In this case x is local variable and is not Initialized but during evaluation the value is initialized



Call by name

- Substitute formal parameters in the body of the procedure by actual parameter expressions.
- If any of the actual parameters have the same name as a local variable used in the called procedure body, then the local variable will first be renamed before the substitution.

Call by name

```
void f(by-name int a, by-name int b)
{
  b=5;
  b=a;
}
```

```
Calling_func(){
    int i = 3;
    f(i+1,i);
}
```

a	i+1
b	i

i	3

BITS Pilani
Hyderabad Campus

innovate achieve lead

Principles of Programming Languages (CS F301)

Implementing SUBPROGRAMS

Jabez Christopher
Assistant Professor
Department of CSIS

Implementing “Simple” Subprograms Parts

- Two separate parts:
- the actual code (which never changes), and
- the non-code part (local variables and data that can change)

- The format, or layout, of the non-code part of an executing subprogram is called an activation record

- An activation record instance is a concrete example of an activation record (the collection of data for a particular subprogram activation)

Semantics

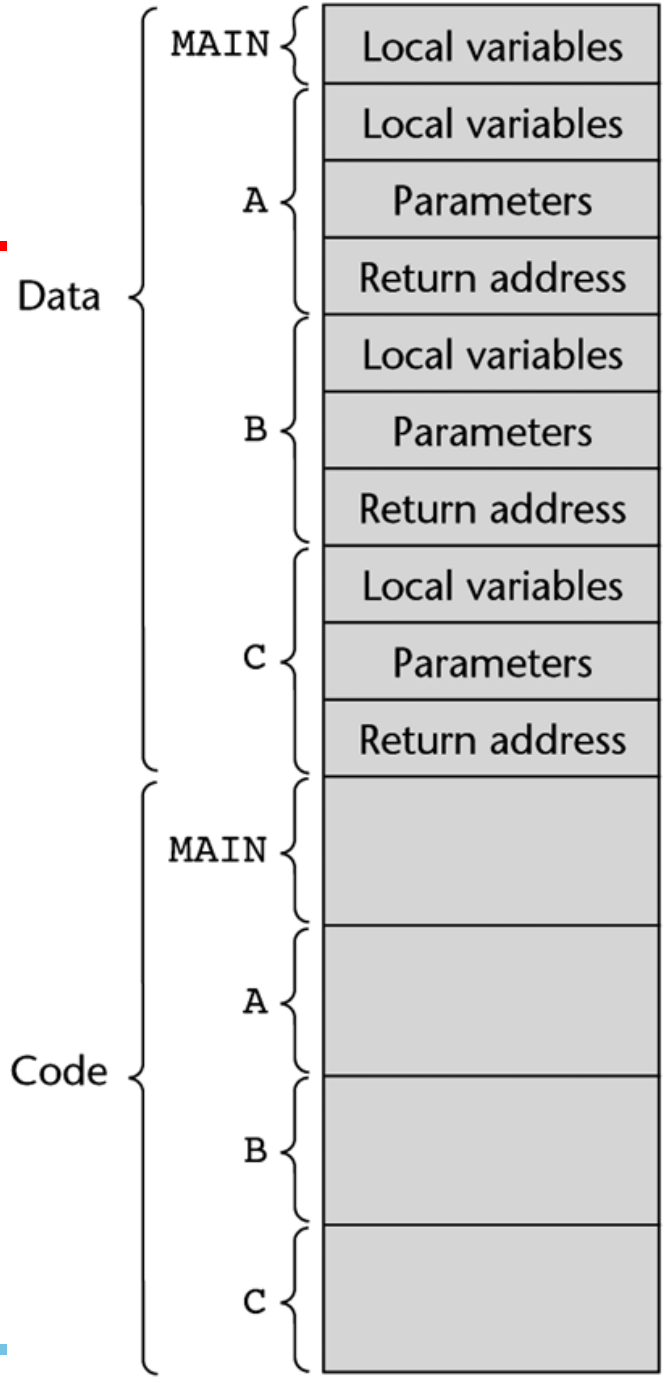
Call Semantics:

1. Save the execution status of the current program unit.
2. Compute and pass the parameters.
3. Pass the return address to the called.
4. Transfer control to the called.

Return Semantics:

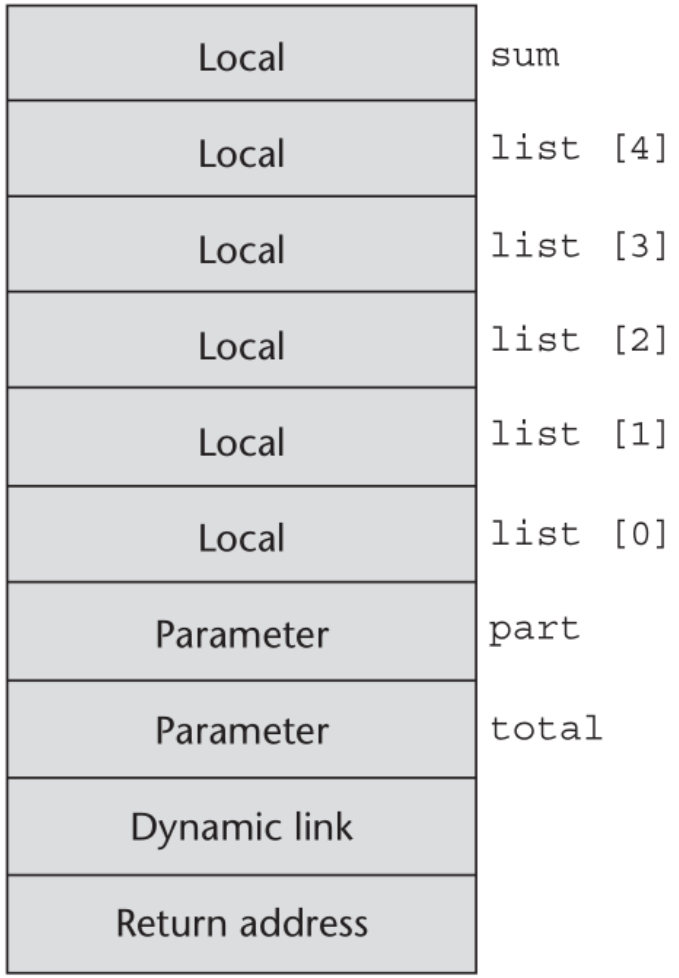
1. If there are pass-by-value-result or out-mode parameters, the current values of those parameters are moved to or made available to the corresponding actual parameters.
2. If the subprogram is a function, the functional value is moved to a place accessible to the caller.
3. The execution status of the caller is restored.
4. Control is transferred back to the caller.

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



An Activation Record for “Simple” Subprograms


```
void sub(float total, int part) {
  int list[5];
  float sum;
  . . .
}
```



Activation Records

- The activation record format is static, but its size may be dynamic
- The *dynamic link* points to the top of an instance of the activation record of the caller
- An activation record instance is dynamically created when a subprogram is called
- Activation record instances reside on the run-time stack
- The *Environment Pointer* (EP) must be maintained by the run-time system. It always points at the base of the activation record instance of the currently executing program unit

Activation Records

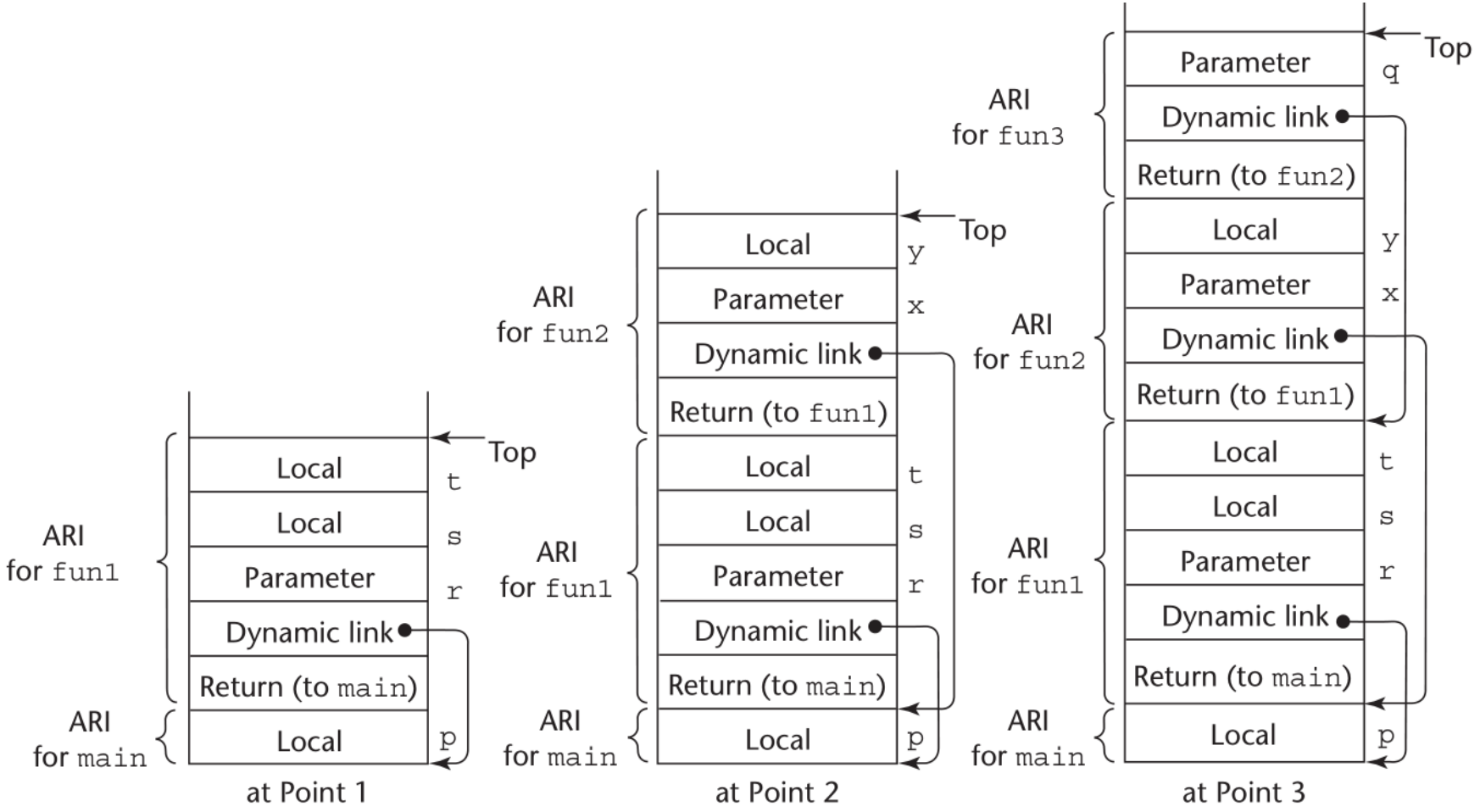
Stack contents for three points in a program

```
void fun1(float r) {
  int s, t;
  ...      <-------- 1
  fun2(s);
  ...
}

void fun2(int x) {
  int y;
  ...      <-------- 2
  fun3(y);
  ...
}

void fun3(int q) {
  ...      <-------- 3
}

void main() {
  float p;
  ...
  fun1(p);
  ...
}
```



Nested Subprograms

Static Chains

A chain of static links that connect certain activation record instances in the stack.

Dynamic chain

Collection of dynamic links present in the stack at a given time; this represents the dynamic history of how execution got to its current position

Local offset

References to local variables can be represented in the code as offsets from the beginning of the activation record of the local scope, whose address is stored in the EP.

Static depth

An integer associated with a static scope that indicates how deeply it is nested in the outermost scope.

Chain offset/ Nesting depth

The difference between the static depth of the subprogram containing the reference to X and the static depth of the subprogram containing the declaration for X

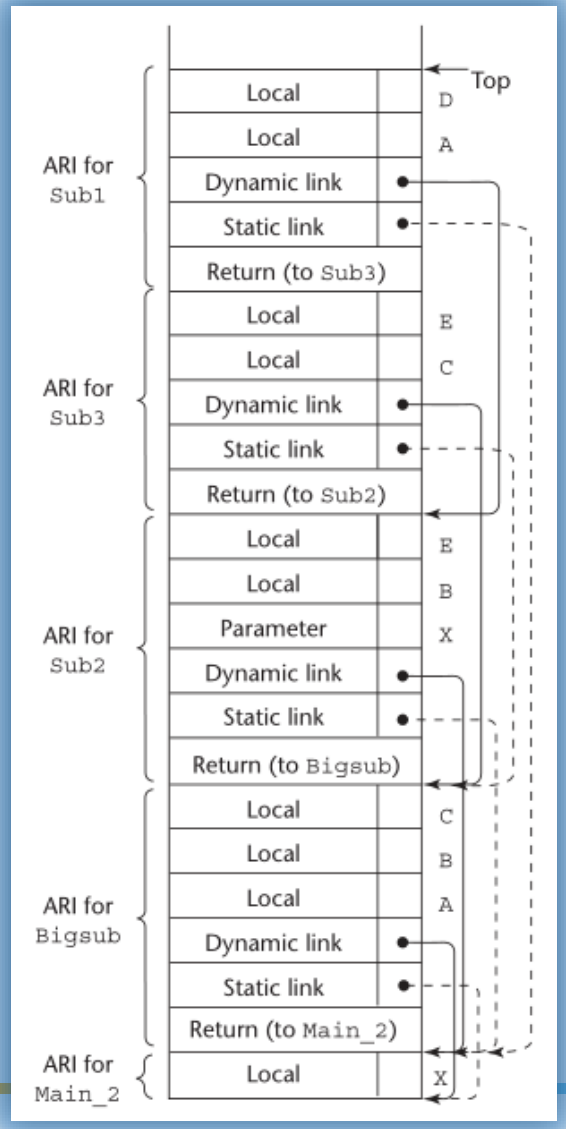
Issues in Nesting

Only variables that are declared in static ancestor scopes are visible and can be accessed

A subprogram is callable only when all of its static ancestor subprograms are active.

```
procedure Main_2 is
  X : Integer;
  procedure Bigsub is
    A, B, C : Integer;
    procedure Sub1 is
      A, D : Integer;
      begin  -- of Sub1
      A := B + C;    <-------------- 1
      . . .
    end;  -- of Sub1
    procedure Sub2(X : Integer) is
      B, E : Integer;
      procedure Sub3 is
        C, E : Integer;
        begin  -- of Sub3
        . . .
        Sub1;
        . . .
        E := B + A; <-------------- 2
      end;  -- of Sub3
      begin  -- of Sub2
      . . .
      Sub3;
      . . .
      A := D + E;    <-------------- 3
    end;  -- of Sub2
    begin  -- of Bigsub
    . . .
    Sub2(7);
    . . .
  end;  -- of Bigsub
  begin  -- of Main_2
  . . .
  Bigsub;
  . . .
end;  -- of Main_2
```

Stack contents at position 1 in the program Main_2



The references to the variable *A* at points 1, 2, and 3 Represented as (*chain offset, local offset pair*)

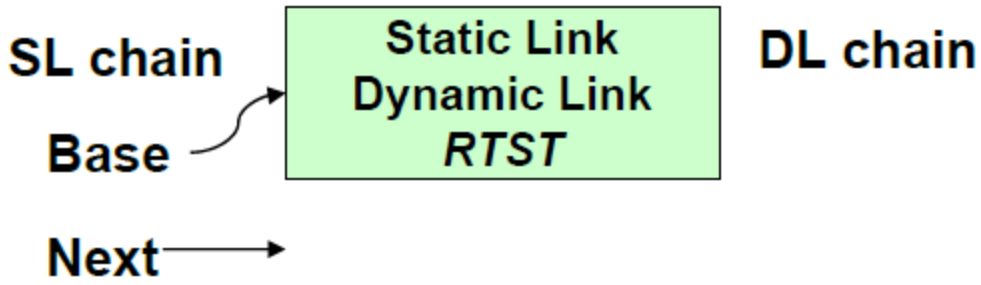
- (0, 3) (local)
- (2, 3) (two levels away)
- (1, 3) (one level away)

Allocation of Activation Records

```
RTST()
{
    P()
    {
        Q() {R();} //definition Q
        R() {Q();} //definition R

    R(); // call to R
    }
    P(); //call to P
}
```

RTST -> P -> R -> Q -> R

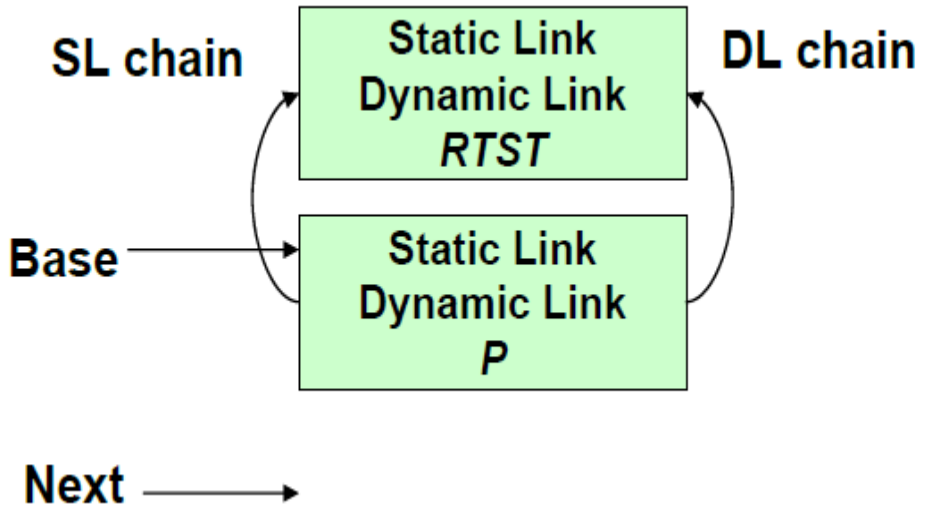


Activation records are created at procedure entry time and destroyed at procedure exit time

Allocation of Activation Records


```
RTST()
{
    P()
    {
        Q() {R();}
        R() { Q();}
    R();
    }
    P();
}
```

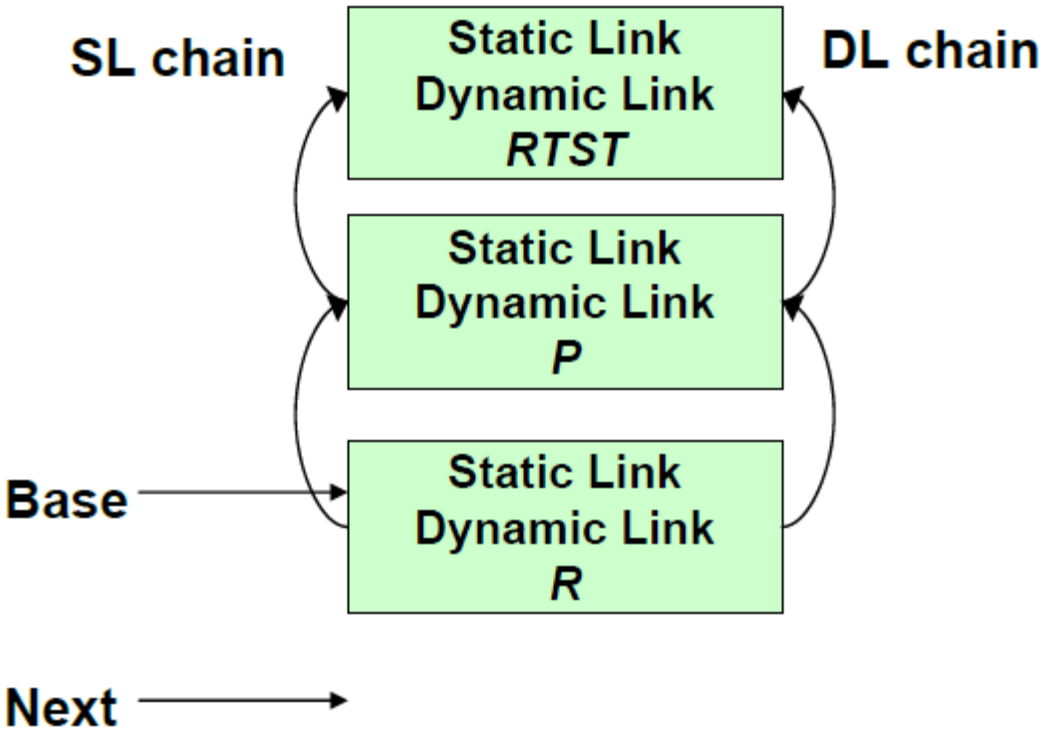
RTST -> P -> R -> Q -> R



Allocation of Activation Records

```
RTST()
{
    P()
    {
        Q() {R();}
        R() { Q();}
    R();
    }
    P();
}
```

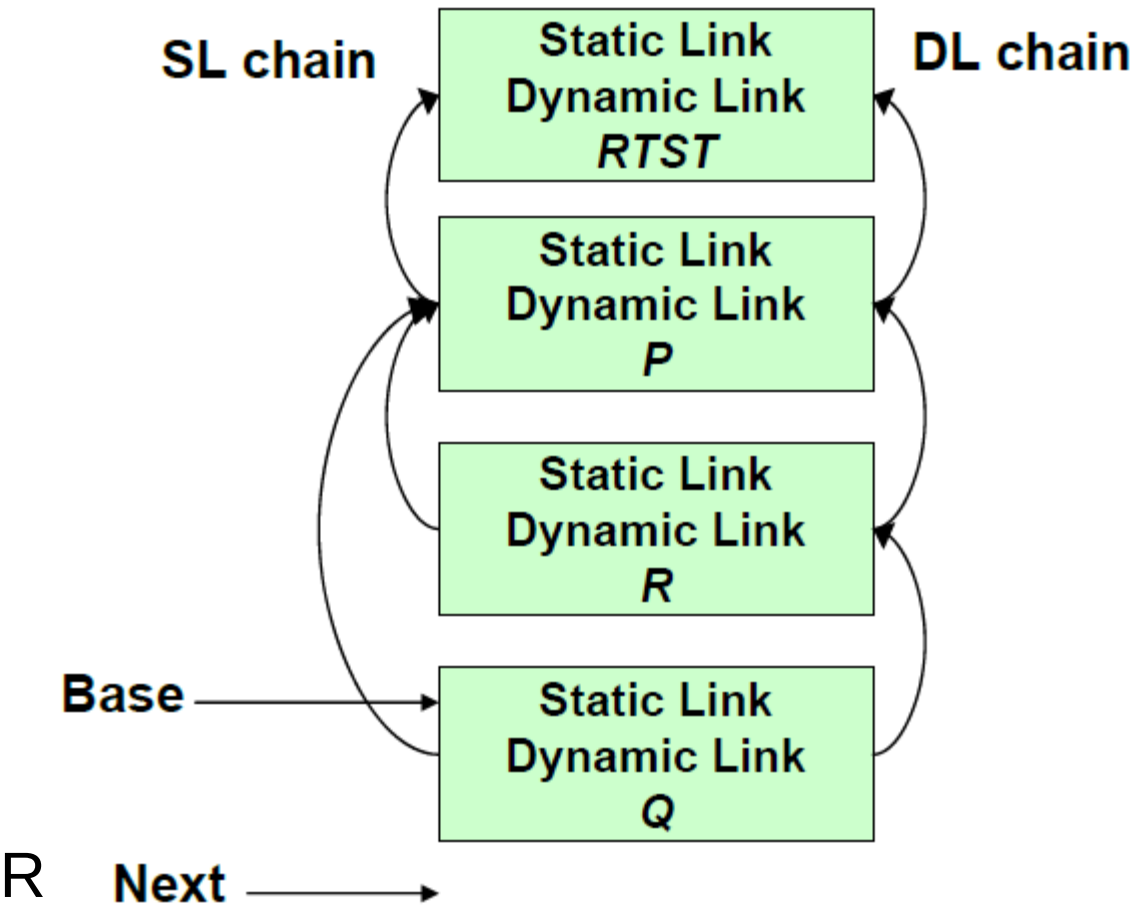
RTST -> P -> R -> Q -> R



Allocation of Activation Records

```
RTST()
{
    P()
    {
        Q() {R();}
        R() { Q();}
    R();
    }
    P();
}
```

RTST -> P -> R -> Q -> R

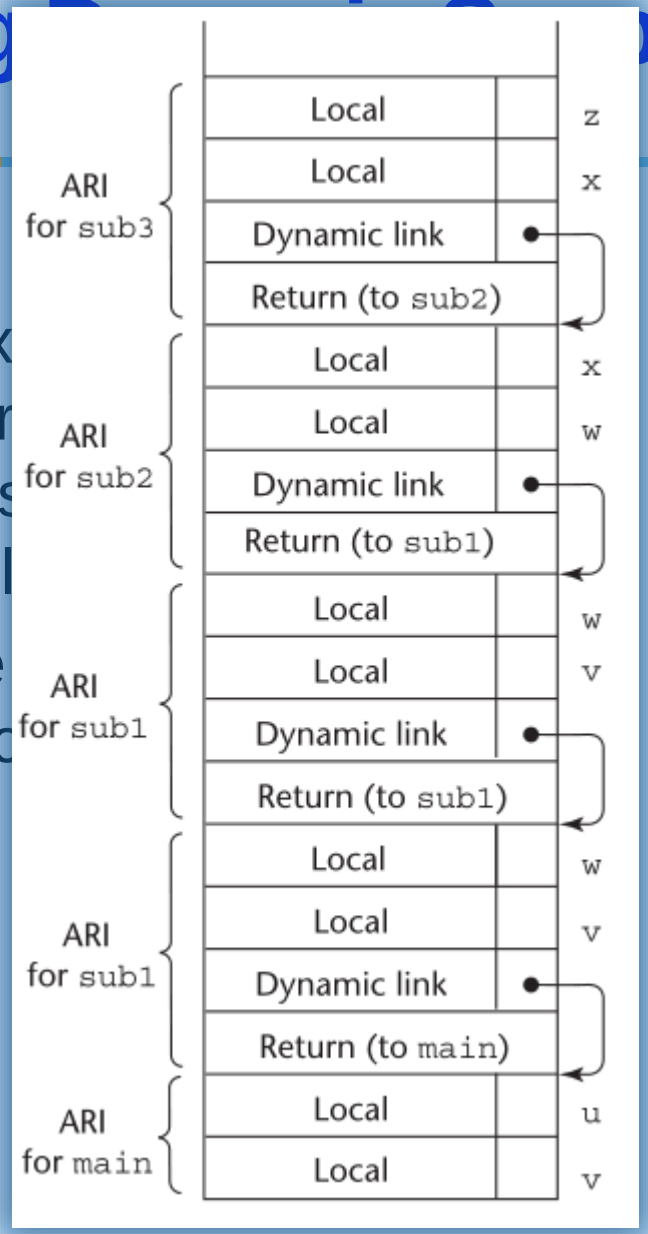


Implementing Dynamic Scoping

Deep Access

Dynamic chain is ex
is needed to refer
nonlocal variables
dynamic-scoped l

Access may require
deep into the stac



main calls sub1
sub1 calls sub1
sub1 calls sub2
sub2 calls sub3

In sub 3

- The reference to *'u'* is found by searching all of the activation record instances on the stack, because the only existing variable with that name is in main .
- Requires following four dynamic links and examining 10 variable names.

```
void sub3() {
  int x, z;
  x = u + v;
  ...
}

void sub2() {
  int w, x;
  ...
}

void sub1() {
  int v, w;
  ...
}

void main() {
  int v, u;
  ...
}
```

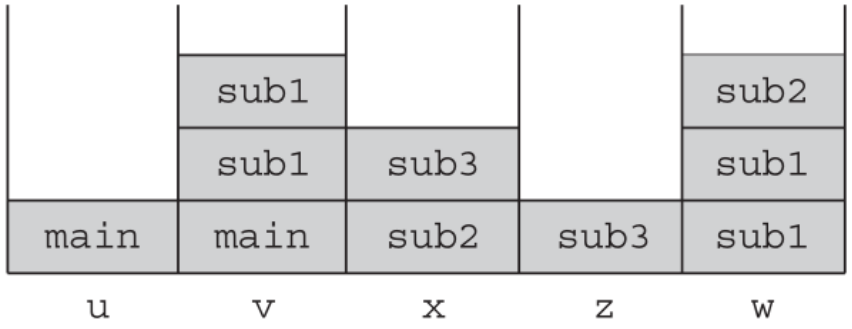
Implementing Dynamic Scoping

Shallow Access

Has a separate stack for each variable name in a complete program.

Allows fast references to variables, but maintaining the stacks at the entrances and exits of subprograms is costly.

main calls sub1
sub1 calls sub1
sub1 calls sub2
sub2 calls sub3




```
void sub3() {
  int x, z;
  x = u + v;
  ...
}

void sub2() {
  int w, x;
  ...
}

void sub1() {
  int v, w;
  ...
}

void main() {
  int v, u;
  ...
}
```

Logic Programming
PROLOG

Jabez Christopher
Assistant Professor
Department of CSIS

Chapter 16 Topics

- Introduction
- A Brief Introduction to Predicate Calculus
- Predicate Calculus and Proving Theorems
- An Overview of Logic Programming
- The Origins of Prolog
- The Basic Elements of Prolog
- Deficiencies of Prolog
- Applications of Logic Programming

Introduction

- Programs in logic languages are expressed in a form of symbolic logic
- Use a logical inferencing process to produce results
- *Declarative* rather that *procedural*:
  - Only specification of *results* are stated (not detailed *procedures* for producing them)

Proposition

- A logical statement that may or may not be true
  - Consists of objects and relationships of objects to each other

Symbolic Logic

- Logic which can be used for the basic needs of formal logic:
  - Express propositions
  - Express relationships between propositions
  - Describe how new propositions can be inferred from other propositions
- Particular form of symbolic logic used for logic programming called *predicate calculus*

Object Representation

- Objects in propositions are represented by simple terms: either constants or variables
- *Constant*: a symbol that represents an object
- *Variable*: a symbol that can represent different objects at different times
  - Different from variables in imperative languages

Compound Terms

- *Atomic propositions* consist of compound terms
- *Compound term*: one element of a mathematical relation, written like a mathematical function
  - Mathematical function is a mapping
  - Can be written as a table

Parts of a Compound Term

- Compound term composed of two parts
  - Functor: function symbol that names the relationship
  - Ordered list of parameters (tuple)
- Examples:

```
student(jon)
like(seth, OSX)
like(nick, windows)
like(jim, linux)
```

Forms of a Proposition

- Propositions can be stated in two forms:
  - *Fact*: proposition is assumed to be true
  - *Query*: truth of proposition is to be determined
- Compound proposition:
  - Have two or more atomic propositions
  - Propositions are connected by operators

Logical Operators

Name	Symbol	Example	Meaning
negation	$\neg$	$\neg a$	not a
conjunction	$\cap$	$a \cap b$	a and b
disjunction	$\cup$	$a \cup b$	a or b
equivalence	$\equiv$	$a \equiv b$	a is equivalent to b
implication	$\supset$ $\subset$	$a \supset b$ $a \subset b$	a implies b b implies a

Quantifiers

Name	Example	Meaning
universal	∀X.P	For all X, P is true
existential	∃X.P	There exists a value of X such that P is true

Clausal Form

- Too many ways to state the same thing
- Use a standard form for propositions
- *Clausal form*:
  - $B_1 \cup B_2 \cup \ldots \cup B_n \subset A_1 \cap A_2 \cap \ldots \cap A_m$
  - means if all the As are true, then at least one B is true
- *Antecedent*: right side
- *Consequent*: left side

Predicate Calculus

- A use of propositions is to discover new theorems that can be inferred from known axioms and theorems
- *Resolution*: an inference principle that allows inferred propositions to be computed from given propositions

Resolution

- *Unification*: finding values for variables in propositions that allows matching process to succeed
- *Instantiation*: assigning temporary values to variables to allow unification to succeed
- After instantiating a variable with a value, if matching fails, may need to *backtrack* and instantiate with a different value

Proof by Contradiction

- *Hypotheses*: a set of pertinent propositions
- *Goal*: negation of theorem stated as a proposition
- Theorem is proved by finding an inconsistency

Theorem Proving

- Basis for logic programming
- When propositions used for resolution, only restricted form can be used
- *Horn clause* - can have only two forms
  - *Headed*: single atomic proposition on left side
  - *Headless*: empty left side (used to state facts)
- Most propositions can be stated as Horn clauses

Overview of
Logic Programming

- Declarative semantics
  - There is a simple way to determine the meaning of each statement
  - Simpler than the semantics of imperative languages
- Programming is nonprocedural
  - Programs do not state now a result is to be computed, but rather the form of the result

Example: Sorting a List

- Describe the characteristics of a sorted list, not the process of rearranging a list

sort(old_list, new_list) $\subset$ permute (old_list, new_list) $\cap$ sorted (new_list)

sorted (list) $\subset \forall_j$ such that $1 \leq j < n$, list(j) $\leq$ list (j+1)

Terms

- *Term*: a constant, variable, or structure
- *Constant*: an atom or an integer
- *Atom*: symbolic value of Prolog
- Atom consists of either:
  - a string of letters, digits, and underscores beginning with a lowercase letter
  - a string of printable ASCII characters delimited by apostrophes

Terms: Variables and Structures

- *Variable*: any string of letters, digits, and underscores beginning with an uppercase letter
- *Instantiation*: binding of a variable to a value
  - Lasts only as long as it takes to satisfy one complete goal
- *Structure*: represents atomic proposition

  functor (*parameter list*)

Fact Statements

- Used for the hypotheses
- Headless Horn clauses

```
female(shelley).
male(bill).
father(bill, jake).
```

Rule Statements

- Used for the hypotheses
- Headed Horn clause
- Right side: *antecedent* (***if*** part)
  - May be single term or conjunction
- Left side: *consequent* (***then*** part)
  - Must be single term
- *Conjunction*: multiple terms separated by logical AND operations (implied)

Example Rules

```
ancestor(mary,shelley):- mother(mary,shelley).
```

- Can use variables (*universal objects*) to generalize meaning:

```
parent(X,Y):- mother(X,Y).
parent(X,Y):- father(X,Y).
grandparent(X,Z):- parent(X,Y), parent(Y,Z).
```

More Examples @

https://youtu.be/tSgjTnEt4io

Goal Statements

- For theorem proving, theorem is in form of proposition that we want system to prove or disprove – *goal statement*
- Same format as headless Horn

```
man(fred)
```

- Conjunctive propositions and propositions with variables also legal goals

```
father(X, mike)
```

Inferencing Process of Prolog

- Queries are called goals
- If a goal is a compound proposition, each of the facts is a subgoal
- To prove a goal is true, must find a chain of inference rules and/or facts. For goal Q:

  P_2 :- P_1
  P_3 :- P_2
  …
  Q :- P_n

- Process of proving a subgoal called matching, satisfying, or resolution

Approaches

- *Matching* is the process of proving a proposition
- Proving a subgoal is called *satisfying* the subgoal
- *Bottom-up resolution, forward chaining*
  - Begin with facts and rules of database and attempt to find sequence that leads to goal
  - Works well with a large set of possibly correct answers
- *Top-down resolution, backward chaining*
  - Begin with goal and attempt to find sequence that leads to set of facts in database
  - Works well with a small set of possibly correct answers
- Prolog implementations use backward chaining

Subgoal Strategies

- When goal has more than one subgoal, can use either
  - Depth-first search: find a complete proof for the first subgoal before working on others
  - Breadth-first search: work on all subgoals in parallel
- Prolog uses depth-first search
  - Can be done with fewer computer resources

Backtracking

- With a goal with multiple subgoals, if fail to show truth of one of subgoals, reconsider previous subgoal to find an alternative solution: *backtracking*
- Begin search where previous search left off
- Can take lots of time and space because may find all possible proofs to every subgoal

Simple Arithmetic

- Prolog supports integer variables and integer arithmetic
- `is` operator: takes an arithmetic expression as right operand and variable as left operand

```
A is B / 17 + C
```

- Not the same as an assignment statement!
  - The following is illegal:

```
Sum is Sum + Number.
```

Example

```
speed(ford,100).
speed(chevy,105).
speed(dodge,95).
speed(volvo,80).
time(ford,20).
time(chevy,21).
time(dodge,24).
time(volvo,24).
distance(X,Y) :-    speed(X,Speed),
                    time(X,Time),
                    Y is Speed * Time.
```

A query: `distance(chevy, Chevy_Distance).`

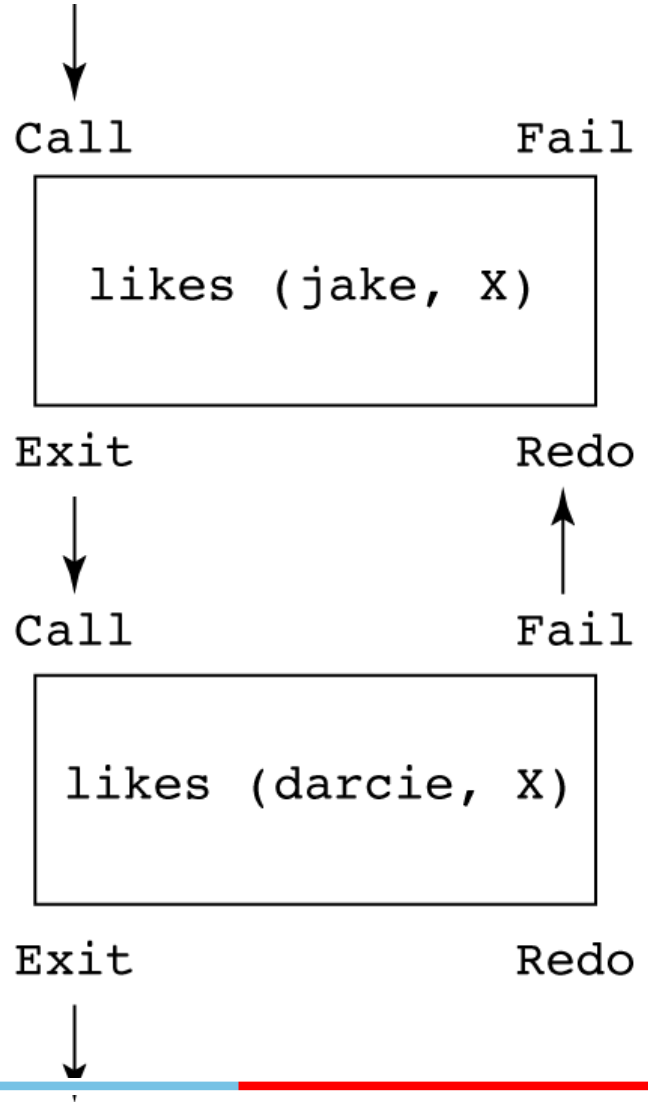
Trace

- Built-in structure that displays instantiations at each step
- *Tracing model* of execution - four events:
  - *Call* (beginning of attempt to satisfy goal)
  - *Exit* (when a goal has been satisfied)
  - *Redo* (when backtrack occurs)
  - *Fail* (when goal fails)

Example

```
likes(jake,chocolate).
likes(jake, apricots).
likes(darcie, licorice).
likes(darcie, apricots).

trace.
likes(jake, X), likes(darcie, X).
(1) 1 Call: likes(jake, _0)?
(1) 1 Exit: likes(jake, chocolate)
(2) 1 Call: likes(darcie, chocolate)?
(2) 1 Fail: likes(darcie, chocolate)
(1) 1 Redo: likes(jake, _0)?
(1) 1 Exit: likes(jake, apricots)
(3) 1 Call: likes(darcie, apricots)?
(3) 1 Exit: likes(darcie, apricots)
X = apricots
```



Deficiencies of Prolog

- Resolution order control
  - In a pure logic programming environment, the order of attempted matches is nondeterministic and all matches would be attempted concurrently
- The closed-world assumption
  - The only knowledge is what is in the database
- The negation problem
  - Anything not stated in the database is assumed to be false
- Intrinsic limitations
  - It is easy to state a sort process in logic, but difficult to actually do—it doesn't know how to sort

Applications of Logic Programming

- Relational database management systems
- Expert systems
- Natural language processing

Summary

- Symbolic logic provides basis for logic programming
- Logic programs should be nonprocedural
- Prolog statements are facts, rules, or goals
- Resolution is the primary activity of a Prolog interpreter
- Although there are a number of drawbacks with the current state of logic programming it has been used in a number of areas