BITS Pilani

Lecture 15
Pointer data type

Pointer Type

- A pointer type is a value that provides indirect access to elements of a known type.
- Declaration usage (in C)

```
int *p;
// p holds the address of an integer value
```

- Pointer data type
  - Range of values: Addresses and a NULL value
  - Operations: = (assignment), + (addition), - (subtraction), → (access), * (dereferencing)

Pointer as a variable

- Declared using some special ways
- In C language

```
int *p, *q;
struct node *r;
```

- In pascal

```
var  p:^integer;
```

- Grammar for C like pointer declaration

<declarationStmt>→<Type> <list>

<list>→ ID COMMA <List> | ID | STAR ID COMMA <LIST> | STAR ID

Example of Pointer usage

- In C language

```
int x;    //declaration of x
int *p;  // pointer declaration
...
x=15;    //initialization of x
p=&x;   //associating p to x
```

- Address of x: x301A
- Address of p:
  x303E
- p=address of x
- *p is the value 15 (dereferencing)

Address	Data (shown in decimal for convenience)
x3002	..
x3006	..
x300A	20
x300E	12
x3012	34
x3016	18
x301A	15
x302E	78
x3032	
x3036	
x303A	
x303E	
x3042	

Example C code

```c
#include <stdio.h>

int main()
{
        int x;  //declaration of x
        int *p; // pointer declaration
        x=15;   //initialization of x
        printf(" x= %d   address of location bound to x = %u contents of
            memory location bound to p = %u\n", x, &x, p);
        //printf("contents of location addressed by p\n", *p);// Produces
            garbage
        p=&x;   //associating p to x
        printf(" x= %d   address of location bound to x = %u contents of
            memory location bound to p = %u\n", x, &x, p);
        printf("contents of location addressed by p =%d\n", *p);
    return 0;
}
```

```
 x= 15   address of location bound to x = 3930661620 contents of memory location bound to p = 0
 x= 15   address of location bound to x = 3930661620 contents of memory location bound to p = 3930661620
contents of location addressed by p =15
```

Example Pascal code

```
program example1;
var
   x: integer;
   p: ^integer;

begin
   x := 20;
   writeln('x= ', x);

   p := @x;
   writeln('p points to a value: ', p^);

   p^ := 45;
   writeln('x= ', x);
   writeln('p points to a value: ', p^);
end.
```

```
x= 20
p points to a value: 20
x= 45
p points to a value: 45
```

Pointers

- Pointers are treated as first class citizens (can be used with integers with simple arithmetic)
- The static checking does not prevent them from being computed as infeasible address (as a result of simple arithmetic)
- A pointer can point to a value of a predefined type.

Use of pointer data type

- Indirect addressing
- Way to manage dynamic storage

Heap dynamic variables

- Variables that are dynamically allocated from the heap are called heap dynamic variables.
- These variables do not have identifiers associated with them and are called as **anonymous variables**.
- These variables are accessed only by the pointer variables, which themselves are bound to the location in the call stack and store the starting address of the dynamically allocated memory.
- In C,

```
struct node *p;
float *q;
p = (struct node *) malloc (sizeof(struct node)*37);
q = (float *) malloc (sizeof(float)*18);
```

Storage visualization for pointer to heap dynamic variable

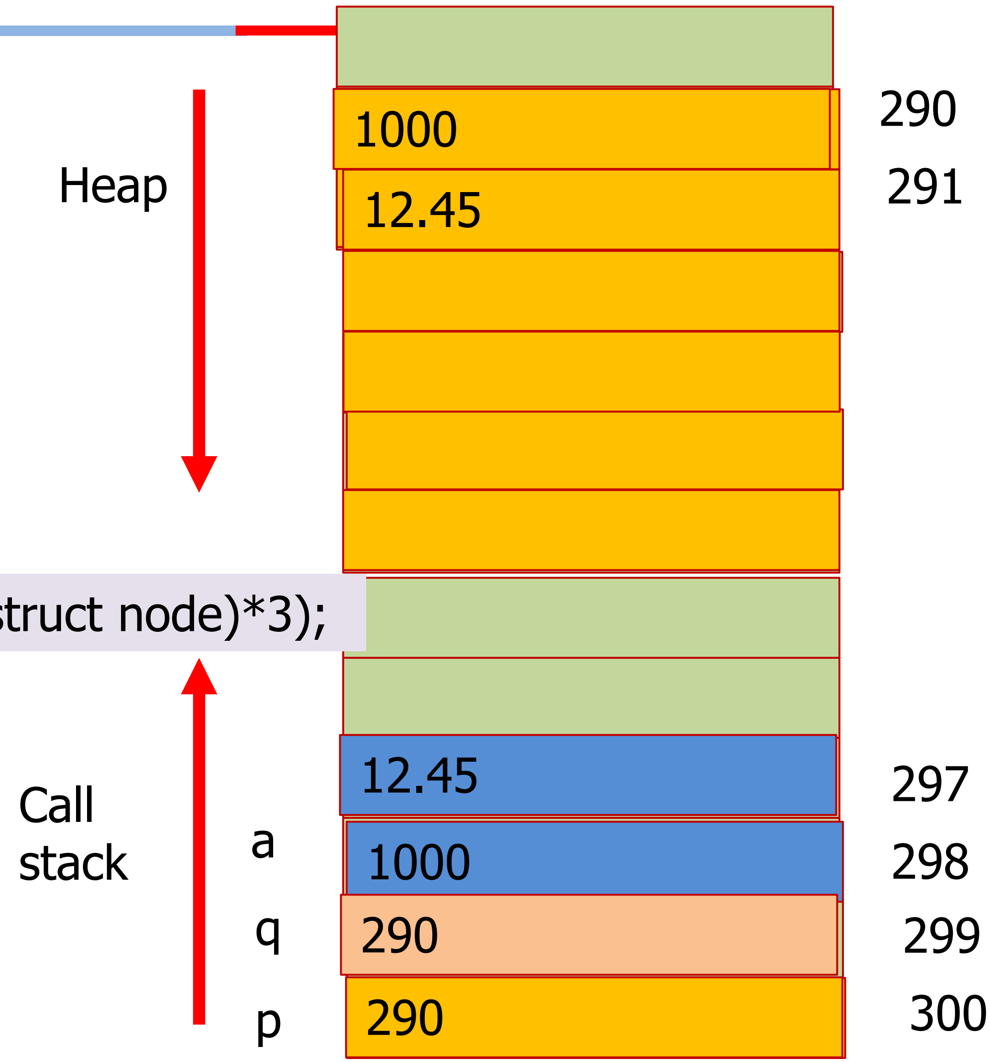
```
struct node {
      int x;
      float u;}; // assume 2
      locations used
Struct node *p, *q;
Struct node a;
```

```
p=(struct node *) malloc(sizeof(struct node)*3);
```

```
p->x = 1000;
P→u = 12.45;
```

```
a=*p;
```

```
q = p;
```



Pointer operations

- Assignment
  - Contents of pointer variable are populated with an address
  - If the pointer variable is heap dynamic and is used to manage the heap dynamic storage, then in-built functions are used to assign address value to the pointer variable.

    ```
    p = (struct node *) malloc (sizeof(struct node)*20);
    ```

  - If the pointer variable is used for indirect addressing to variables that are not heap dynamic, then special operator is needed to associate the address of a variable

    ```
    p = &x;
    *p=20;
    ```

Pointer operations

- Dereferencing: When a memory location is indirectly accessed by a pointer as in p=&x, then in order to access the data stored in memory bound to x, an explicit operator * is used.
- *p dereferences the pointer p to get value of the location whose address is stored in the location of p.
- Some language implicitly dereference based on the type of variable p which is pointer to integer for example. In FORTRAN 95+, it is implemented implicitly.

Dereferencing a record fields

```
struct node {
    int x, y;
    float z;
};
struct node *p;
```

Dereferencing the pointer

P→x

P→y

P→z

(*p).x

(*p).y

(*p).z

Problems in Pointers

- Dangling Pointer

The dangling pointer contains the address of the heap dynamic variable that has been deallocated.

```
Struct node *p, *q, *r;
p = (struct node *) malloc (sizeof(structnode)*30);        //allocation
q=p;
free(p);                               //deallocation
printf("%d\n", q->x);                           //using the dangling pointer
```

- Memory leak

```
p = (struct node *) malloc (sizeof(structnode)*30);        //M1
r = (struct node *) malloc (sizeof(structnode)*20);        //M2
p = r;    //causes memory M1 not reachable through p and r.
```

The anonymous heap dynamic variable is said to have been lost in this process.

Ada pointers

- Ada provides support to reduce possibilities of dangling pointers by not allowing user to explicitly deallocate.
- But it also has a feature that allows the user to deallocate explicitly by using a keyword 'Unchecked_Deallocation'
- Ada does not have inbuilt support to reduce the memory leaks.

Pointers

- Type checking prevents the pointer variables from pointing to wrong data type.
- Pointers have fixed size independent of what they point to.
- Pointer variable fits into a single machine location.

Design issues

- Which text segment is the pointer definition valid?
- Whether the pointer type definition survive across the functions or through out the program execution or not?
- What is the lifetime of the heap dynamic variable pointed to by the pointer variable?
- Should the pointer access be restricted for the type of variable it is pointing to?
- Should the pointer variable be used for indirect addressing or for storage management of the heap dynamic variable, or both?

Function types

- If

  R_type function_name (T1 arg1, T2 arg2)

  - the type of the function is defined as the function (T1, T2)→R_type
  - Type mismatch occurs in

    value = function_name(val1, val2)

    If either val1 is not of type T1

    Or val2 is not of type T2

    Or value is not of type R_type