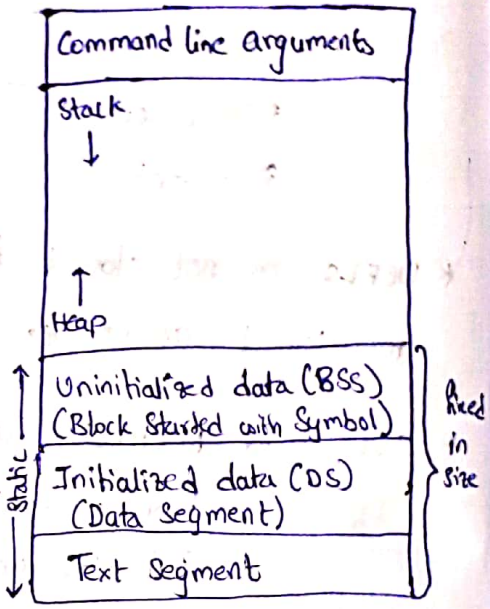
27/05/20

Programming Languages

Memory organization

Memory type	contents	Allocation time
Stack	• Local variables • Formal parameters • Return addresses • Function calls	Runtime
Static	Static variables Global variables Function code String Constants	Compile time (load time)
Heap	• Runtime allocated memory by malloc(), calloc() realloc()	Runtime



→ BSS contains uninitialized global variables

→ DS contains initialized global variables

→ Text segment contains function code & static variables of function

Storage Classes:

Storage Class	Storage Area	Default value	Lifetime	Scope
Auto	Stack	garbage	function	function
Register	Register	garbage	function	function
static	Static part	0	~~function~~ program	function
global	Static part	0	program	program

Note:

→ The 'extern' keyword is used to tell compiler that the variable will be declared as global elsewhere in the program.

→ During compilation choice of memory layout is ~~fin~~ finalized.

→ Whenever a function is called it ~~acce~~ activation record is loaded into stack. The activation record contains auto variables, formal parameters, Return address and function calls, temporaries & access links.

→ Static memory & Heap memory is deleted as soon as control returns to the operating system.

→ Accessing address of Register variable is ~~not~~ not possible and declaring a variable doesn't guarantee for its storage in register unless a register is free.

→ Static variables & Global variables can be be initialized only with constants but not expressions.

→ For declaration statements of static, global, extern variables no action is performed at runtime.

→ If there is a global variable & a local variable of same name, then local variable is given priority.

Pointers:

→ Consider
```
int a;
int *p;
p=&a;
```

~~Accessi~~ Accessing content of 'a': a, *p, *(&a)

Accessing address of 'a': &a, p

Accessing address of 'p': &p

→ A pointer cannot be assigned with an integer value. However, we can assign 0.

int *P=0 (or) int *P=NULL both are same.

Pointer Assignment:

```
int a,b;
int *p,*q;
p=&a; // p points to a
q=&b; // q points to b
p=q; // p also points to b. This is pointer assignment.
```

Passing address of a variable:

```
void f(int *p)
{
  ≡
}
```

Valid calls:

```
f(&a); // a is int type
f(p); // p is int* type
```

Returning address of a variable:

```
int* f()
{
  ≡
  return ___;
}
```

Valid returns:

```
return &a; // a is int type
return p; // p is int* type
```

Note:

→ Every pointer holds base address of the variable it is pointing to. Based the datatype of pointer, the pointer decides how many bytes are to be accessed from the base address irrespective of the type of variable it is pointing to.

Eg:

```
int a;
char *c=(char*)&a;
```

∴ Now *c access only the starting byte of a.

Refer Q22 & Q23 for more clarity

void Pointer:

→ This pointer is used to store address of any datatype.

→ But we cannot dereference a void pointer.

→ No arithmetic operations are possible on void pointer.

→ However address in void pointer can be assigned to other pointers by type casting.

Problems in Pointers:

i) Dangling pointer problem:

A pointer pointing to a memory location that has been deleted is called dangling pointer.

Eg:
```
int main()
{
    int *ptr = (int*) malloc(sizeof(int));
    :
    free(ptr);
    // Now ptr is dangling pointer;
    ptr = NULL; // ptr is, now, not a dangling pointer;
}
```

Eg:
```
int* fun()
{
    int x = 5;
    return &x;
}
int main()
{
    int *p = fun();
    // p is a dangling pointer
    :
}
```

Eg:
```
void main()
{
    int *p;
    {
        int a = 10;
        p = &a;
    }
    // p is a dangling pointer
}
```

(ii) Uninitialized pointer problem : (or) wild pointer :

```
void main()
{
    int *p; // p is uninitialized
    *p = 10; // accessing and changing contents in some unknown address
    :
}
```

(iii) Null pointer Dereference

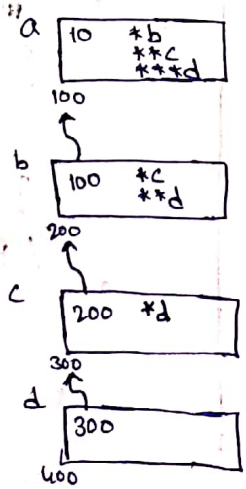
```
void main()
{
    int *p = NULL;
    *p = 10; // Accessing content from nothing
    :
}
```

(iv) Memory Leakage Problem :

```
void main()
{
    int *p;
    p = (int*) malloc (10 * sizeof(int));
    :
    p = (int*) malloc (10 * sizeof(int)); // Now old memory block in heap
    :                                     has no reference, but still it
}                                         remains allocated in heap.
```

Pointer to pointer :

```
int *a, *b, **c, ***d;
a = 10;
b = &a;
c = &b;
d = &c;
```



Dereference (or) Indirection Operation (*)

→ It is a unary operator which operates on an address.

→ Its operation results in contents present in the address.

Pointer to array element:

→ Every array name itself contains the base address of array.

→ Consider `int a[] = {1,2,3};`

```
int *p;
p = a;  (or)  p = &a[0]
```

→ assignment of array element to pointer.

Accessing array element: `a[i], *(a+i), *(i+a), i[a]`

`p[i], *(p+i), *(i+p), i[p]`

Accessing array elements' address: `&a[i], (a+i), (i+a), &i[a]`

`&p[i], (p+i), (i+p), &i[p]`

→ Note that 'a' is a constant pointer and hence it cannot be modified.

→ p[-i] evaluates *(p-i). So is a[-i].

Pointer to one-dimensional array:

Declaration: `int (*p)[3];`

It means p is a pointer which points to a one dimensional array ~~of size~~ containing 3 elements.

Thus, here, incrementing 'p' will increament its address by 3 * sizeof(int).

Thus incrementing will result in pointing to the next dimension.

assigning: int x[4][3];

int (*p)[3];

p=x; // Assignment.

(a) Now on wherever we use x, we can also use p.

Accessing array element: ~~x[i][j]~~,

x[i][j], (*(x+i))[j], *(*(x+i)+j), *(x[i]+j)

p[i][j], (*(p+i))[j], *(*(p+i)+j), *(p[i]+j)

Accessing address of an array element:

&x[i][j], &(*(x+i))[j], *(x+i)+j

&p[i][j], &(*(p+i))[j], *(p+i)+j

Accessing address of i^{th} array or i^{th} dimension:

x[i], &x[i], *(x+i), x+i

p[i], &p[i], *(p+i), p+i

Address of array:

~~x, &x, *x, x[0], &x[0], &~~

x, &x, *x, x[0], &x[0], &x[0][0]

p, *p, p[0], &p[0], &p[0][0]

valid assignments:

p=x;

p=&x[0]

p=&((&x[0])[0]);

p=x+1

p=&x[1];

p=&((&x[1])[0]);

Array of pointers:

→ Here we declare ~~array~~ array of pointers. For every element of array P[i], same rules are applied as pointer 'p'.

→ Consider

```
int x[4][3];
int *P[4];// To store base address of each one dimensional
          array x[0], x[1], x[2], x[3] we need four
          pointer.
```

Assignment:

```
int *P[4] = {x[0], x[1], x[2], x[3]};
        (or)
int *P[4] = {&x[0][0], &x[1][0], &x[2][0], &x[3][0]};
        (or)
int *P[4] = {*x, *(x+1), *(x+2), *(x+3)};
        (or)
int *P[4];
P[0]=x[0]; P[1]=x[1]; P[2]=x[2]; P[3]=x[3];
```

Accessing array element (x[i][j])

P[i][j], (*(P+i))[j], *(*(P+i)+j), *(P[i]+j)

Accessing address of an array element:

&P[i][j], &(*(P+i))[j], *(P+i)+j

~~Accessing~~

→ ~~A~~ Array of pointers can be initialized in the case of strings.

Eg: static char *s[3] = {"gowtham", "kranthi", "girish"};

However, this isn't possible for other datatypes.

→ A two dimensional array can be implemented either by pointer to one dimensional array or array of pointers.

→ The major difference b/w these two is,

* In pointer to 1D array we have only one pointer which points to required location and it can point only to specified size of array.

* In array of pointers we have specified number of pointers where each pointer can point to any sized array.

* In pointer to 1D array, the assigned pointer can be increamented and make pointer to the next dimension.

* In array of pointers increamenting pointer doesn't make any sense as it is the array name which is constant. However array elements can be incremented and make point to the next element of the array they are pointing to.

Pointer to strings:

String constant: It is a group of characters enclosed within double quotes. A string constant is always stored in static part during compile time.

* The string constant always returns its base address.
* The compiler adds null character at the end for a string constant.

→ `char *c = "India";`

Here "India" is stored in static part at the time of compilation. The base address of is returned at runtime and c points

to the base address.

* Here the string is stored as constant and hence no modifications are allowed.

→ char c[] = {'I', 'n', 'd', 'i', 'a', '\0'};

(or)

c[] = "India";

↳ Here null character should be added explicitly.

* Here memory is allocated for c during runtime and characters are stored in the array.
* Now we can modify the contents also.

* char *p = c;

Here 'p' acts as a pointer to string. and is used same as pointer to ~~an~~ array element.

Now if we write a stmt

p = "cat";

p doesn't change contents of c, but it points to the base address returned by "cat" which is in static part.

→ strlen() takes string pointer as a parameter and returns an unsigned integer.

→ When we use individual character like 'a', they are not considered a string. But '\0' is considered a string.

Eg: ~~strlen~~ strlen('a'); can't be used.

strlen('\0'); can be used.

Pointer to Structures:

→ Consider

```
struct student
{
    int idno;
    char s[5];
};
:
struct student s1;
struct student *p; // structure pointer declaration
p= &s1; // assignment
```

Accessing members:

s1.idno	s1.s
p→idno	p→s
(*p).idno	(*p).s

Array of structures:

```
struct student a[3]= { 10, "sri", 20, "sai", 30, "raj" };
```

* like all arrays, structure array name also contains its address.

* struct student *p= a; // pointer to structure array element.

Here increment p, causes p to point to the next structure element in the array.

* accessing arrays:

```
a[i].idno
(p+i)→idno   (or)   *(p+i).idno
p[i].idno
```

Note :

Consider 'ptr' is pointer to a structure element and 'z' is a member of the structure.

1) ++ptr → z ≅ ++(ptr → z)

This statement accesses z first and increaments 'z' in the structure variable pointed by 'ptr'.

2) (++ptr) → z

This statement increaments ptr first.
Thus ptr points to next structure variable (in arrays)
Now 'z' of the next variable is accessed.

3) ptr++ → z ≅ (ptr++) → z

This stmt first access the content of z in the variable pointed by 'ptr'.
Later ptr is increamented.

4) Consider 'z' is a pointer.

*ptr → z++ ≅ *(ptr → z++) ≅ *(p→a)++

Here content in 'z' of the variable pointed by 'ptr' is accessed firset.

Now indirection (*) operator is operated on pointer 'z'.

Later z is increamented i.e., z is pointed to next.

However the stmt

(*p→a)++

increments the content of a[0] in the structure variable pointed by p.

Self Referential structure : It is a structure containing a pointer variable pointing to its own type.

Pointer to function:

→ A function name always contains the base address of the function.

Declaration of pointer to function:

```
return_type (*ptr) (argument list);
```

Eg: `void (*p)(int, int);`

p can point to functions like:

`void fun(int x, int y)` etc.

Remember that if P is not enclosed in parenthesis

`void *p(int, int)`

denotes a function declaration which returns a void pointer

Assigning pointer to function:

```
P = fun;
  (or)
P = &fun;
```

} No paranthesis are allowed.

Accessing the function:

```
fun(2,3);
P(2,3);
(*P)(2,3);
```

} Both are valid

Array of function pointers:

declaration: `return_type (*ptr[size])(arg list);`

Eg: `int (*P[3])(int, int);`

Let
```
int add(int x, int y)
int sub(int x, int y)
int mul(int x, int y)
```
be 3 functions

```
P[0] = add;          P[0] = &add
P[1] = sub;   (or)   P[1] = &sub;
P[2] = mul;          P[2] = &mul;
```

} assignment.

P[0](3,2)
*(P[0])(3,2)
((P+i))(3,2)
} accessing.

→ array of function pointers can also be initialized

i.e., int (*P[3])(int,int) = {add, sub, mul};

03/06/20

Recursion:

→ There is nothing to write about recursion.
practice more problem to ~~get~~ improve speed in calculations.

Static scope & Dynamic Scope:

(i) Static scope:

Referencing environment of a statement in a statistically scoped language is the collection of all local variables and all other ancestor variables which are visible in the statement.

(ii) Dynamic scope:

Refering (enci) environment of a statement in a dynamically scoped language is the collection of all local variables and all other active subprogram variables which are visible in the statement.

* In both the scopes, the recent ancestor is given more importances.

Note:

→ In pascal language or psuedocode,

we need to see the syntax of declaration of variable and assume it is followed throughout the program.

Consider the two programs

```
var x: integer
program main()
{
    var y: integer
    x=3;
    y=5;
}
```

Here

var x: datatype is the syntax for declaration.

So 'x' in line1 refers to the global variable

```
x=10;
program main()
{
    y=6;
    x=3;
}
```

Here

x=value; is the syntax for declaration

So 'x' in line1 is considered local variable and doesn't refer global variable 'x'.

→ In pascal language's syntax,

* procedure (var x: datatype)

Here x is considered address variable

i.e., pass by reference

* procedure (x: datatype)

Here x is considered value variable

i.e., pass by value.

Parameter Passing Techniques

There are two types of parameter passing techniques:

(i) Call by value:

Here actual parameters always carry their values and these values are copied into formal parameters. Modifications done to formal parameter in called function doesn't affect the values of actual parameters in calling function.

(ii) Call by Reference

In call by reference, actual parameters' address are passed to formal parameters and thus formal parameters act as an alias to their corresponding actual parameter. Modifications done to formal parameters reflect in actual parameters.

Temporories:

When an expression is passed as a parameter, a temporary memory location is created and the result is stored in it. Temporaries are present in activation record of the calling function.

Access link:

Every activation record of a function must have access link to the most recent activation record of a function which lexically contains the current function.

→ This access link concept is used in static scoping as access links provide access to the ancestor function's activation record.