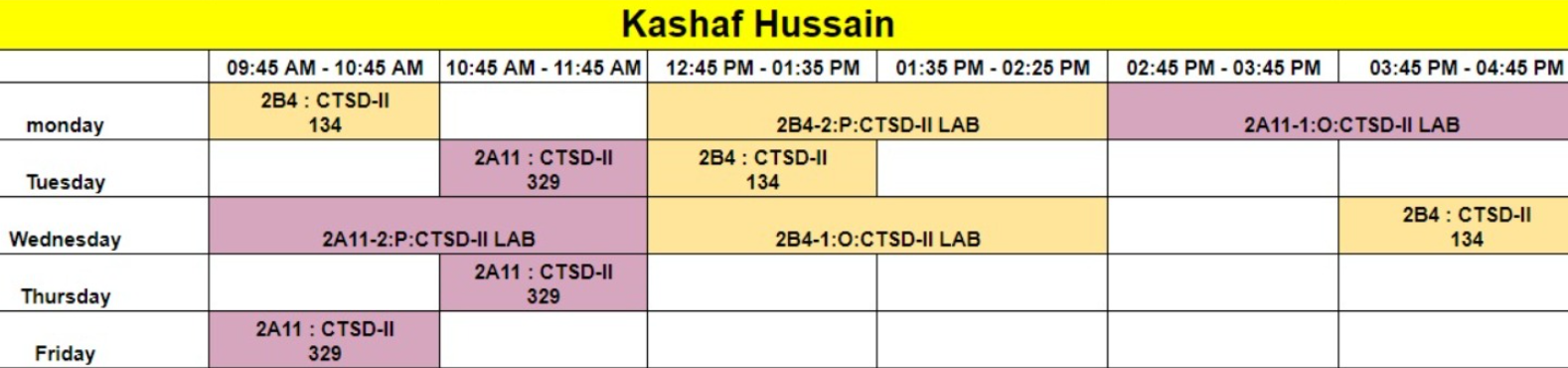
Parul University

Computational Thinking & Structured Design

**C & Data Structures**



**Dynamic Data Structures**

**Static data structures :** Memory requirement is known at the time of writing programs.

**Dynamic data structures :** Demand for memory is made at the time of running program.

**Data Structures :** Data may be organised in many different ways; The logical and mathematical model of a particular organization of data is called a **data structure**.

**NOTE :** Defining an array does fix a block of memory being reserved at the beginning of program execution, whereas this doesn’t occur if an array is represented in terms of a pointer variable. Therefore the use of a pointer variable to represent an array requires some type of initial memory assignment before the array elements are processed. This is known as **dynamic memory allocation**. The **C** function **malloc** is used to allocate memory at run time.

malloc function :

int \*p ;

p = (int \*) malloc (sizeof(int));

address type

reserve the area in memory

starting address

( int \* ) malloc (n \* sizeof( int ));

( int \* ) calloc (n, sizeof( int ));

( int \* ) calloc (10, sizeof( int ));

**Storage Management**

* The functions **malloc** and **calloc** obtain blocks of memory dynamically.

void \*malloc(size\_t n)

returns a pointer to n bytes of uninitialized storage, or NULL if the request cannot be satisfied.

void \*calloc(size\_t n, size\_t size)

returns a pointer to enough space for an array of n objects of the specified size, or NULL if the request cannot be satisfied. The storage is initialized to zero.

The pointer returned by malloc or calloc has the proper alignment for the object in question, but it must be cast into the appropriate type, as in

ip=(int\*)calloc(n,sizeof(int));

* free(p) frees the space pointed to by p, where p was originally obtained by a call to malloc or calloc. There are no restrictions on the order in which space is freed, but it is a ghastly error to free something not obtained by calling calloc or malloc.
* It is also an error to use something after it has been freed. A typical but incorrect piece of code is this loop that frees items from a list:

for(p=head; p!=NULL; p= p->next)

free(p);

The right way is to save whatever is needed before freeing:

for(p = head; p != NULL; p = q)

{

q = p->next;

free(p);

}

* **malloc()** allocates a memory block of given size (in bytes) and returns a pointer to the beginning of the block.
* **malloc()** doesn’t initialize the allocated memory. If you try to read from the allocated memory without first initializing it, then you will invoke undefined behavior, which usually means the values you read will be garbage values.
* **calloc()** allocates the memory and also initializes every byte in the allocated memory to **0**. If you try to read the value of the allocated memory without initializing it, you’ll get 0 as it has already been initialized to **0** by **calloc()**.

Memory value initialised to zero.

int \*p;

p = (int \*) malloc(10 \* sizeof( int ));

here **p** is not automatically assigned a memory block when it is defined as a **pointer variable**, though a block of memory large enough to store 10 integer quantities will be reserved in advance when **p** defined as an array i.e. int[10] ;

To assign sufficient memory for **pointer**

**p**, we use library function **“malloc”.**

p = (int \*) malloc ( 10 \* sizeof (int ));

* This function reserves a block of memory whose size (in bytes) is equivalent to 10 integer quantities.
* The function returns a **pointer to an integer**. This pointer indicates the beginning of the memory block. i.e. pointer **‘p’** will hold the starting address of allocated memory block. The **type casting** precede **malloc** function must be consistent with the data type of the pointer variable. Thus if **‘p’** were defined as a pointer to a double-precision quantity and we wanted enough memory to store 10 double-precision quantities, we would write :

p = ( double \*) malloc (10 \* sizeof( double)) ;

NOTE : A **double-precision** quantity is a 64-bit **floating-point number** that is accurate up to 16 decimal places. In double-precision each number takes up 64-bits while **single-precision** format uses 32-bits and **half-precision** is just 16-bits.

NOTE : if the declaration is to include the assignment of initial values, however then **’p’** must be defined as an array rather than a **pointer variable**.

For example :

int p[10] = { 1,2,3,4,5,6,7,8,9,10 };

or

int p[ ] = { 1,2,3,4,5,6,7,8,9,10 };

free(p);

it will free the memory area which was acquired by the pointer **‘p’** through **“malloc”** function.

Program: WAP to read ‘n’ numbers in a dynamic array.

p

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |

p+0 p+1 p+2 p+n-1

#include <stdio.h>

#include <stdlib.h>

int main()

{

int \*p ,i,n;

printf("Enter the no of elements :");

scanf("%d",&n);

p = (int \*) malloc(n\*sizeof(int));

for(i=0;i<n;i++)

{

printf("Enter the elements : ");

scanf("%d", p+i);

}

for(i=0;i<n;i++)

{

printf("%d\t", \*(p+i));

}

free(p);

return 0;

}

Program : WAP to read n-elements in a one dimensional array and sort them using pointer notation.

#include <stdio.h>

#include <stdlib.h>

void sorting(int n, int \*p);

int main()

{

int \*p ,i,n;

printf("Enter the no of elements :");

scanf("%d",&n);

p = (int \*) malloc(n\*sizeof(int));

for(i=0;i<n;i++)

{

printf("Enter the elements : ");

scanf("%d", p+i);

}

for(i=0;i<n;i++)

{

printf("%d\t", \*(p+i));

}

sorting(n,p);

for(i=0;i<n;i++)

{

printf("%d\t", \*(p+i));

}

void sorting(int n, int \*p)

{

int i,j,temp;

for(i=1;i<n-1;i++)

for(j=0;j<n-i;j++)

if(\*(p+j) > \*(p+j+1))

{

temp = \*(p+j);

\*(p+j) = \*(p+j+1);

\*(p+j+1) = temp;

}

}

// free(p);

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |

return 0;

}

NOTE : An important advantage of **dynamic memory allocation** is the ability to reserve as much as memory as may be required during program execution and then released this memory when it is no longer needed. The library function “malloc” and “free” are used for these purposes.

.

**Allocation of space for 2-D Array**

0 1 2 3

|  |
| --- |
|  |
|  |
|  |
|  |
|  |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  | +- ewe 0 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

p 0

p+1 1

p+2 2

p+3 3

p+4 4

* we assume 2D array as a one dimensional array whose data is the address of first row of the 2D array i.e. each cell of assumed one dimensional array contains the value which is the address of first element in each row.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |

Row 0 p

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |

Row 1 p+ 1

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |

Row 2 p +2

p+2 pointer to row 2 \*(p+2) \*(p+2)+4

\*(p+2) content of the pointer

i.e. (p+2) refers to the entire row \*(\*(p+2)+4)

Row 2 is one dimensional array hence

\*(p+2) is actually a pointer to the first element in row 2.

(\*(p+2)+4) is pointer to the 4th element (starting from 0 i.e. 5th element)

\*(\*(p+2)+4) is the content of this pointer.

\*\*q

x

\*p

Address of Address variable

Address

Here \*p contains the address of variable x and \*\*q contains the address of pointer \*p.

#include <stdio.h>

#include <stdlib.h>

int main() {

int \*\*q,m,n,i;

printf("\nEnter the no. of rows and columns : " );

scanf("%d%d",&m,&n);

q = (int \*\*) malloc(m\*sizeof(int \*));

pointer to an integer pointer

creating dynamic array

for(i=0;i<m;i++)

{

\*(q+i) = (int \*) malloc(n \* sizeof (int));

}

return 0;

}

Program : WAP to create 2D array.

Program : WAP to multiply two such dynamic arrays into the third array.

Program : WAP to store ‘n’ words in dynamic array and sort the words and print the arrays.

Program : WAP to create 2-D integer array and print the array.

#include <stdio.h>

#include <stdlib.h>

void print(int \*\*, int , int);

int main()

{

int \*\*p, r, c, i, j;

clrscr();

printf("Enter the no. of rows and cols : ");

scanf("%d%d",&r,&c);

// pointer to a integer pointer

p = (int \*\*) malloc(r \* sizeof( int \*));

for(i = 0; i<r; r++)

\*(p+i)=(int \*) malloc (c \* sizeof(int));

// entering the data in the matrix

for(i=0 ; i<r ; i++)

for(j=0;j<c;j++)

{

printf("Enter the data : ");

scanf("%d", (\*(p+i)+j));

}

print(p,r,c);

// getch();

return 0;

}

void print(int \*\*p, int r, int c)

{

int i,j;

for (i=0;i<r;i++)

{

printf("\n");

for(j=0;j < c ; j++)

printf("%d", \*(\*(p+i)+j));

}

}

**Preprocessor Directives**

**The C Preprocessor**

* C provides certain language facilities by means of a **preprocessor**, which is conceptually a separate first step in compilation. The two most frequently used features are **#include** and **#define.**
* **#include** is used include the contents of a file during compilation, and **#define**, to replace a token by an arbitrary **sequence of characters**. Other features described in this section include **conditional compilation** and **macros** with arguments.

**File Inclusion**

* File inclusion makes it easy to handle collections of **#defines** and declarations. Any source line of the form

#include "filename"

or

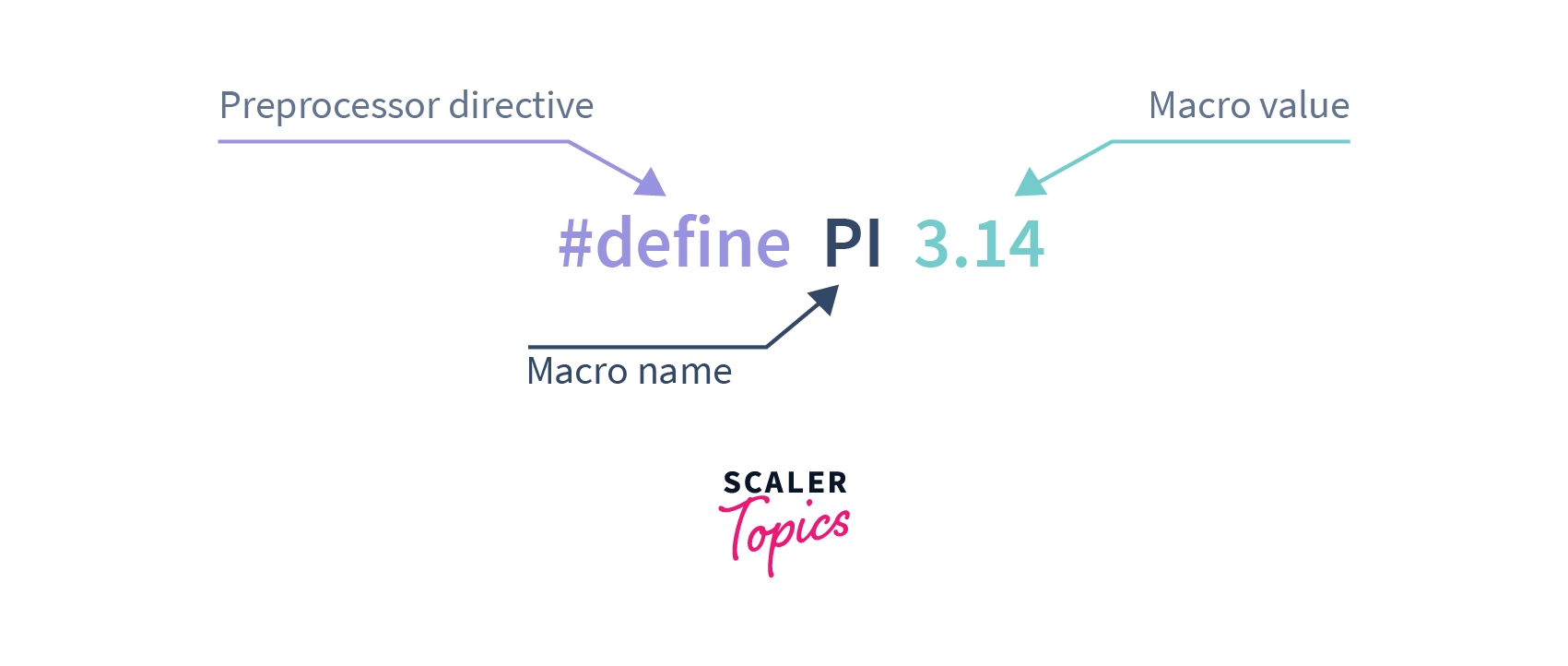
#include <filename>

is replaced by the contents of the file filename. If the filename is quoted, searching for the file typically begins where the source program was found; if it is not found there, or if the name is enclosed in < and >, searching follows an implementation-defined rule to find the file.

* There are often several #include lines at the beginning of a source file, to include common #define statements and extern declarations, or to access the **function prototype** declarations for **library functions** from headers like <stdio.h>. (Strictly speaking, these need not be files; the details of how headers are accessed are implementation-dependent.)
* #include is the preferred way to tie the declarations together for a large program. It guarantees that all the source files will be supplied with the same definitions and variable declarations, and thus eliminates a particularly nasty kind of bug. Naturally, when an included file is changed, all files that depend on it must be recompiled.

**Macro Substitution**

* The **macro** in C programminn is known as the small piece of code that can be replaced by the macro value.
* The **macro** is defined with the help of #define preprocessor directive and the macro doesn’t end with a semicolon(;). Macro is just a name given to certain values or expressions it doesn't point to any memory location.
* Whenever the compiler encounters the **macro**, it replaces the macro name with the macro value. Two macros could not have the same name.
* The syntax of the macro is as shown in the following figure. Here, we will have the three components:
* #define : Preprocessor Directive
* PI : Macro Name
* 3.14 : Macro Value



#include<stdio.h>

// This is macro definition

#define PI 3.14

void main()

{

// declaration and initialization of radius

int radius = 5;

// declaration and calculating the area

float area = PI \* (radius\*radius);

// Printing the area of circle

printf("Area of circle is %f", area);

}

* A definition has the form

*#define name replacement text*

* It calls for a **macro** substitution of the simplest kind-subsequent occurrences of the token name will be replaced by the ***replacement text***. The name in a **#define** has the same form as a variable name; the replacement text is arbitrary. Normally the replacement text is the rest of the line, but a long definition may be continued onto several lines by placing a **\** at the end of each line to be continued. The scope of a name defined with **#define** is from its point of definition to the end of the source file being compiled.
* Any name may be defined with any replacement text. For example,

#define forever for(;;)

/\* infinite loop \*/

* defines a new word, forever, for an infinite loop.
* It is also possible to define **macros** with arguments, so the replacement text can be different for different calls of the macro. As an example, define a **macro** called max:

#define max(A,B) ((A)>(B) ? (A): (B))

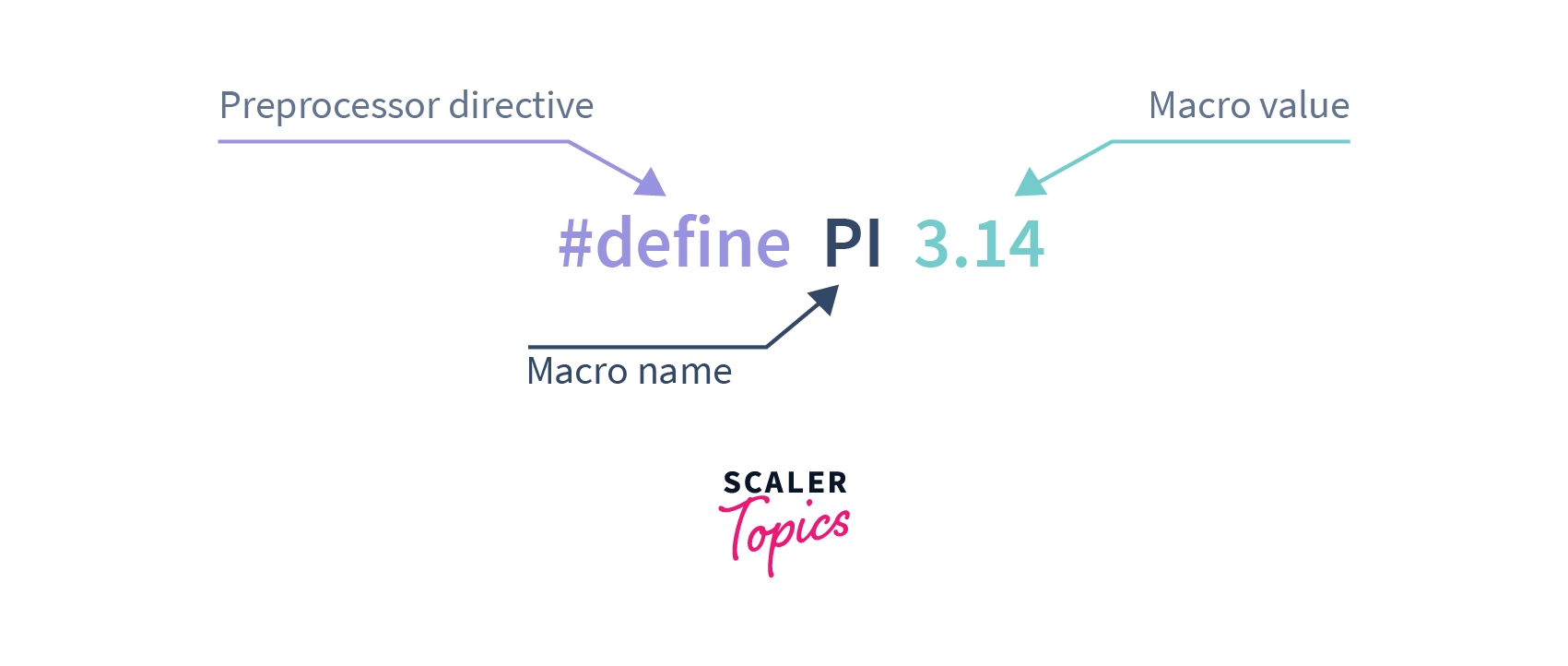
* Each occurrence of a formal parameter (here A or B) will be replaced by the corresponding actual argument. Thus the line

x = max(p+q, r+s);

will be replaced by the line

x = ((p+q) > (r+s) ? (p+q) : (r+s));

* So long as the arguments are treated consistently, this macro will serve for any data type; there is no need for different kinds of max for different data types, as there would be with functions.



**Conditional Compilation**

It is possible to control preprocessing itself with conditional statements that are evaluated during preprocessing. This provides a way to include code selectively, depending on the value of conditions evaluated during compilation.

The #if line evaluates a constant integer expression (which may not include sizeof, casts, or enum constants). If the expression is non-zero, subsequent lines until an #endif or #elif or #else are included. (The preprocessor statement #elif is like else if.) The expression defined(name) in a #if is 1 if the name has been defined, and 0 otherwise.

For example, to make sure that the contents of a file hdr.h are included only once, the contents of the file are surrounded with a conditional like this:

#if !defined(HDR)

#define HDR

/\* contents of hdr.h go here\*/

#endif

The first inclusion of hdr.h defines the name HDR; subsequent inclusions will find the name defined and skip down to the #endif. A similar style can be used to avoid including files multiple times. If this style is used consistently, then each header can itself include any other headers on which it depends, without the user of the header having to deal with the interdependence.