

# YESHWANTRAO CHAVAN COLLEGE OF ENGINEERING

DEPARTMENT OF COMPUTER SCIENCE AND  
ENGINEERING

SESSION 2021-22 ODD SEMESTER

## CSE2204 : Data Structures I

### III Semester

Objective	Course Outcome
<ol style="list-style-type: none"><li>1. To make students familiar with syntaxes and usages of various programming constructs of C language</li><li>2. To make student understand concept of abstract data types like stacks and queues</li><li>3. To make student understand file handling operations</li><li>4. To create thinking ability needed for implementation of programming logic with proper use of memory</li></ol>	<ol style="list-style-type: none"><li>1. To understand programming constructs like stack, queue, array, structures</li><li>2. To Apply appropriate data structures in problem solving.</li><li>3. To Implement various abstract data types and programming logic needed for solving given problem</li><li>4. Analyze the performance of operations performed on data structures.</li></ol>

# SYLLABUS

Unit No.	Contents	Max. Hrs.
1	Types and operations, Iterative constructs and loop invariants, Quantifiers and loops, Structured programming and modular design, Illustrative examples, Scope rules, parameter passing mechanisms, recursion, program stack and function invocations including recursion	6
2	Overview of arrays and array based algorithms - searching and sorting: merge sort, quick sort, Sparse matrices.	7
3	Structures (Records) and array of structures (records). Database implementation using array of records. Dynamic memory allocation and de allocation. Dynamically allocated single and multi-dimensional arrays, polynomial representation.	7
4	Concept of an Abstract Data Type (ADT), Lists as dynamic structures, operations on lists, implementation of linked list using arrays and its operations. Introduction to linked list implementation using self-referential-structures/pointers.	6
5	Stack, Queues and its operations. Implementation of stacks and queues using both array-based and pointer-based structures. Applications of stacks and queues.	6
6	File organization, examples of using file, file access methods , Hashing and collision resolution techniques	6

**TEXT BOOKS:**

<b>Sr. No</b>	<b>Title</b>	<b>Authors</b>	<b>Edition (Year of Publication)</b>	<b>Publisher</b>
1	Data Structures and Program Design in C	Robert Kruse, G. L. Tondo and B. Leung	latest edition	PHI-EEE
2	Fundamentals of Data Structures in C	Ellis Horowitz, Satraj Sahni and Susan Anderson-Freed	latest edition	W. H. Freeman and Company.
3	How to Solve it by Computer	R. G. Dromey	latest edition	Pearson Education

**Reference books:**

<b>Sr. No</b>	<b>Title</b>	<b>Authors</b>	<b>Edition (Year of Publication)</b>	<b>Publisher</b>
1	Data Structures with C	Seymour Lipschutz	Latest	TMH

# UNIT-1

Types and operations,  
Iterative constructs and loop invariants, Quantifiers  
and loops,  
Structured programming and modular design,  
Illustrative examples,  
Scope rules,  
parameter passing mechanisms,  
recursion,  
program stack and function invocations including  
recursion

# **C language preliminaries and control structures**

# Structured programming and Modular Design

- A technique for organizing and coding computer programs in which a hierarchy of modules is used, each having a single entry and a single exit point, and in which control is passed downward through the structure without unconditional branches to higher levels of the structure.
- Three types of control flow are used: sequential, test, and iteration.
- A style of programming usually associated with languages such as C, Fortran, Pascal and so on.
- Using structured programming techniques, a problem is often solved using a divide and conquer approach such as stepwise refinement.
- An initially large problem is broken into several smaller sub-problems. Each of these is then progressively broken into even smaller sub-problems, until the level of difficulty is considered to be manageable. At the lowest level, a solution is implemented in terms of data structures and procedures.

# **Basic structure of 'C' Language**

Various sections of C program are as follows:

Documentation section

Link section

Definition section

Global declaration section

main() function section

{

Declaration part

Executable part

}

Subprogram section

Function1

Function 2

-----

Function n



# Program structure

A sample C Program

```
#include<stdio.h>
int main()
{
    --other statements
}
```

## Header files

- The files that are specified in the include section is called as header file
- These are precompiled files that has some functions defined in them
- We can call those functions in our program by supplying parameters
- Header file is given an extension .h
- C Source file is given an extension .c

# **Main function**

- This is the entry point of a program
- When a file is executed, the start point is the main function
- From main function the flow goes as per the programmers choice.
- There may or may not be other functions written by user in a program
- Main function is compulsory for any c program

# Writing the first program

```
#include<stdio.h>
int main()
{
    printf("Hello");
    return 0;
}
```

- This program prints Hello on the screen when we execute it

# Running a C Program

- Type a program.
- Save it.
- Compile the program – This will generate an exe file (executable).
- Run the program (Actually the exe created out of compilation will run and not the .c file).
- In different compiler we have different option for compiling and running. We give only the concepts.

## Comments in C

- Single line comment
  - // (double slash)
  - Termination of comment is by pressing enter key
- Multi line comment
  - /\*....
  - .....\*/

This can span over to multiple lines

# Documentation Section

Comments:

Comments are used to document programs and improve readability

Compilers ignore the comments

Compilers do not provide any machine language object code for the comments

Comments help other programmers to understand your program

- In C a comment will start with /\* and ends with \*/

- Syntax: /\* Comments \*/

/\*This is a single line comment \*/

/\* This is a multiline

- comment in C \*/

/\*\*\*\*\*\*

\* This style of commenting is used for functions

\*\*\*\*\*/

- Only C style comments should be used

- The C++ style comments will start with a // and ends at the end of that line

# Character Set

- The characters in C are grouped into the following categories:
  1. Letters
  2. Digits
  3. Special Characters
  4. White Spaces

## ‘C’ Tokens

- Keywords
- Identifiers
- Constants: numeric and character
- Strings
- Operators
- Special Symbols

# Keywords

Keywords			
auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

# Data types in C

- Primitive data types
  - int, float, double, char
- Aggregate data types
  - Arrays come under this category
  - Arrays can contain collection of int or float or char or double data
- User defined data types
  - Structures and enum fall under this category.

## C has the following simple data types:

C type	Size (bytes)	Lower bound	Upper bound
char	1	—	—
unsigned char	1	0	255
short int	2	-32768	+32767
unsigned short int	2	0	65536
(long) int	4	$-2^{31}$	$+2^{31} - 1$
float	4	$-3.2 \times 10^{\pm 38}$	$+3.2 \times 10^{\pm 38}$
double	8	$-1.7 \times 10^{\pm 308}$	$+1.7 \times 10^{\pm 308}$



# Operators in C

- Assignment operator =
- Arithmetic operators +, -, \*, /, %
- Relational operators >, >=, <, <=, ==, !=
- Logical operators !, &&, ||
- Address operator &
- Increment and Decrement operators ++, --
- Compound Assignment Operators =, +=, -=, /=, \*=, %=
- **sizeof** operator

# Arithmetic Operators

C operation	Arithmetic operator	Algebraic expression	C expression
Addition	+	$f + 7$	<b>f + 7</b>
Subtraction	-	$p - c$	<b>p - c</b>
Multiplication	*	$bm$	<b>b * m</b>
Division	/	$x / y$	<b>x / y</b>
Modulus	%	$r \text{ mod } s$	<b>r % s</b>

- Rules of operator precedence:

Operator(s)	Operation(s)	Order of evaluation (precedence)
( )	Parentheses	Evaluated first. If the parentheses are nested, the expression in the innermost pair is evaluated first. If there are several pairs of parentheses “on the same level” (i.e., not nested), they are evaluated left to right.
*, /, or %	Multiplication, Division, Modulus	Evaluated second. If there are several, they are evaluated left to right.
+ or -	Addition Subtraction	Evaluated last. If there are several, they are evaluated left to right.

# Decision Making: Equality and Relational Operators

Standard algebraic equality operator or relational operator	C equality or relational operator	Example of C condition	Meaning of C condition
<i>Equality Operators</i>			
<b>=</b>	<b>==</b>	<b>x == y</b>	<b>x</b> is equal to <b>y</b>
<b>not =</b>	<b>!=</b>	<b>x != y</b>	<b>x</b> is not equal to <b>y</b>
<i>Relational Operators</i>			
<b>&gt;</b>	<b>&gt;</b>	<b>x &gt; y</b>	<b>x</b> is greater than <b>y</b>
<b>&lt;</b>	<b>&lt;</b>	<b>x &lt; y</b>	<b>x</b> is less than <b>y</b>
<b>&gt;=</b>	<b>&gt;=</b>	<b>x &gt;= y</b>	<b>x</b> is greater than or equal to <b>y</b>
<b>&lt;=</b>	<b>&lt;=</b>	<b>x &lt;= y</b>	<b>x</b> is less than or equal to <b>y</b>

```

1  /* Fig. 2.13: fig02_13.c
2     Using if statements, relational
3     operators, and equality operators */
4  #include <stdio.h>
5
6  int main()
7  {
8     int num1, num2;
9
10    printf( "Enter two integers, and I will tell you\n" );
11    printf( "the relationships they satisfy: " );
12    scanf( "%d%d", &num1, &num2 );    /* read two integers */
13
14    if ( num1 == num2 )
15        printf( "%d is equal to %d\n", num1, num2 );
16
17    if ( num1 != num2 )
18        printf( "%d is not equal to %d\n", num1, num2 );
19
20    if ( num1 < num2 )
21        printf( "%d is less than %d\n", num1, num2 );
22
23    if ( num1 > num2 )
24        printf( "%d is greater than %d\n", num1, num2 );
25
26    if ( num1 <= num2 )
27        printf( "%d is less than or equal to %d\n",
28                num1, num2 );

```

1. Declare variables

2. Input

3. if statements

4. Print

```
29
30     if ( num1 >= num2 )
31         printf( "%d is greater than or equal to %d\n",
32                 num1, num2 );
33
34     return 0;    /* indicate program ended successfully */
35 }
```

Exit main

```
Enter two integers, and I will tell you
the relationships they satisfy: 3 7
3 is not equal to 7
3 is less than 7
3 is less than or equal to 7
```

Program Output

```
Enter two integers, and I will tell you
the relationships they satisfy: 22 12
22 is not equal to 12
22 is greater than 12
22 is greater than or equal to 12
```

# Bitwise operators

- In addition to standard math operators, C has low-level bitwise operators
- `&` bitwise AND
- `|` bitwise OR
- `^` bitwise XOR
- `~` bitwise NOT (one's complement)
- `<<` left shift
- `>>` right shift

# Increment and Decrement Operators

Operator	Called	Sample expression	Explanation
++	preincrement	++a	Increment <b>a</b> by 1, then use the new value of <b>a</b> in the expression in which <b>a</b> resides.
++	postincrement	a++	Use the current value of <b>a</b> in the expression in which <b>a</b> resides, then increment <b>a</b> by 1.
--	predecrement	--b	Decrement <b>b</b> by 1, then use the new value of <b>b</b> in the expression in which <b>b</b> resides.
--	postdecrement	b--	Use the current value of <b>b</b> in the expression in which <b>b</b> resides, then decrement <b>b</b> by 1.

Fig. increment and decrement operators.

# Increment and Decrement Operators

Operator	Associativity	Type
++ --	right to left	unary
* / %	left to right	multiplicative
+ -	left to right	additive
< <= > >=	left to right	relational
== !=	left to right	equality
?:	right to left	conditional
= += -= *= /= %=	right to left	assignment
Fig. Precedence and associativity of the operators discussed so far.		



# Precedence

- An operator with higher precedence is done earlier (precedes) one with lower precedence
  - A *higher* precedence is indicated with a *lower* number; zero is the highest precedence
- Most of the time, operators with equal precedence are done left to right
  - Examples:  
 $3 + 4 * 5$  gives 23  
 $10 - 5 - 2$  gives 3
- Exceptions: unary operators, casts, assignment operators, the ternary operator: all done right to left

# Testing Conditions

- For a variable, zero = false, any non-zero number = true

- The Quick Switch:  
 $(x > y) ? x = 5 : x = 0$

- Equal / Not Equal / Less Than / Greater Than:

$x == y$

$x != y$

$x < y, x > y$

- Compound Expressions:

$(x == y) \&\& (x > z)$  AND

$(x != y) \parallel (x >= z)$  OR

Is equivalent to

if (  $x > y$  )

$x = 5;$

else

$x = 0;$

# Control Structure 1

- IF / IF ... ELSE

```
if (true )  
{  
    DoFirstThing();  
    DoSecondThing();  
}  
  
if (true)  
    DoSomething();  
else  
    DoSomethingElse();
```

## SWITCH

```
switch (key)  
{  
    case 'a':  
    case 'A':  
        DoFirstThing();  
        DoSecondThing();  
        break;  
    case 'b':  
        DoSomething();  
        break;  
    default:  
        break;  
}
```

# Sample program using if-else

```
#include<stdio.h>
int main()
{
int x,y;
printf("Enter value for x :");
scanf("%d",&x);
printf("Enter value for y :");
scanf("%d",&y);
if ( x > y )
{
printf("X is large number - %d\n",x); } else{ printf("Y is large number -
%d\n",y);
}
return 0;
}
```

# Sample program using switch

```
#include <stdio.h>
main()
{
int menu, numb1, numb2, total;
printf("enter in two numbers -->");
scanf("%d %d", &numb1, &numb2 );
printf("enter in choice\n");
printf("1=addition\n");
printf("2=subtraction\n");
scanf("%d", &menu );
switch( menu )
{
case 1:
total = numb1 + numb2;
break;

default:
Prinf(“ wrong choice”);
Break;

case 2: total = numb1 - numb2;
}
```

default:

```
printf("Invalid option selected\n");  
}  
if( menu == 1 )  
printf("%d plus %d is %d\n", numb1, numb2, total );  
else if( menu == 2 )  
printf("%d minus %d is %d\n", numb1, numb2, total );  
}
```

# Control Structure 2

- FOR

```
int i, j;  
for (i=0; i<5; i++)  
    for (j=5; j>0; j--) {  
        // i counts up  
        // j counts down  
        printf("%i %j\n", i, j);  
    }
```

- The “++” / “--” is shortcut used to increment / decrement value of int variables

- WHILE

```
int i = 0;  
int StayInLoop = 1;  
while ( StayInLoop ) {  
    i+=2;  
    // Make sure you have  
    // exit condition!  
    if ( i > 200 )  
        StayInLoop = 0;  
}
```

- “+=” increments by n

# Loops

- Are used to repeat certain set of statements if certain condition is true.
- Three types of loops in c :
  - Do while
  - While
  - For

## Increment and Decrement Operators

- Preincrement or predecrement operator
  - Increment of decrement operator placed before a variable
- Postincrement or postdecrement operator
  - Increment of decrement operator placed after a variable



# The do-while loop

- Condition checking is at the end of loop.
- Syntax :

*do*

{ <statements> ;

<statements> ;

.....;

} *while* (<condition>);

- The body of do-while loop is executed continuously ,till condition becomes false.
- do-while loop is also called as Odd loop.
- Even if the condition is false at the start, still the loop gets executed once.

# Do-while Loop Example

```
void main( )  
{  
    int i=1;  
    do  
    {  
        printf(“\n%d”,i);  
        i=i+1;  
    } while(i<=10);  
}
```

# The while loop

- Condition checking is at the starting of loop.

- Syntax :

```
while (<condition>)  
{ <statements> ;  
  <statements> ;  
  .....;  
}
```

- Body of while loop is executed, till the condition becomes false.
- Body of while loop is executed zero or more times.

# while Loop Example

```
void main( )  
{  
  int i=1;  
  while(i<=10)  
  {  
    printf("\n%d", i);  
    i=i+1;  
  }  
}
```

# The for loop

- Condition checking is at the starting of loop.
- Syntax :  

```
for (<intial status>; <repeat condition>; < action statements>)  
{  
    <statements> ;  
    <statements> ;  
    .....;  
}
```
- Body of loop is executed, till the repeat condition becomes false.

# for Loop Example

```
void main( )  
{  
    int i;  
    for(i=1;i<=10;i++)  
    {  
        printf("\n%d",i);  
    }  
}
```

# **Control transfer statements**

- break statement
- continue statement

# break statement

- Terminates innermost loop/switch
- From this statement, control is transferred to outside the loop/switch.
- Usage

## ■ Within loops :

Generally used in conjunction with conditional statements

## ■ Within Switch Case statements :

useful to break the switch statement



# break Example

```
void main( )
{
    int i, t,no;
    for( i=0; i<n; i++)
    {
        while(t!=10)
        {
            scanf("%d", &no);
            if(no==0)
                break;
            t--;
        } /*end of while*/
    }
```

# **continue statement**

- Terminates the current iteration of innermost loop.
- From this statement, control is transferred to condition part of innermost loop.

## continue Example

```
void main( )
{
    int i;
    for( i=0; i<50; i++)
    {
        if(i%5==0)
        {
            continue;
            printf("%d", i);
        } /*end of if*/
    } /*end of for*/
}
```

# Modular Programming with functions

## Modularity

Programmer-defined functions

Standard C library functions

## Examples

random number generation

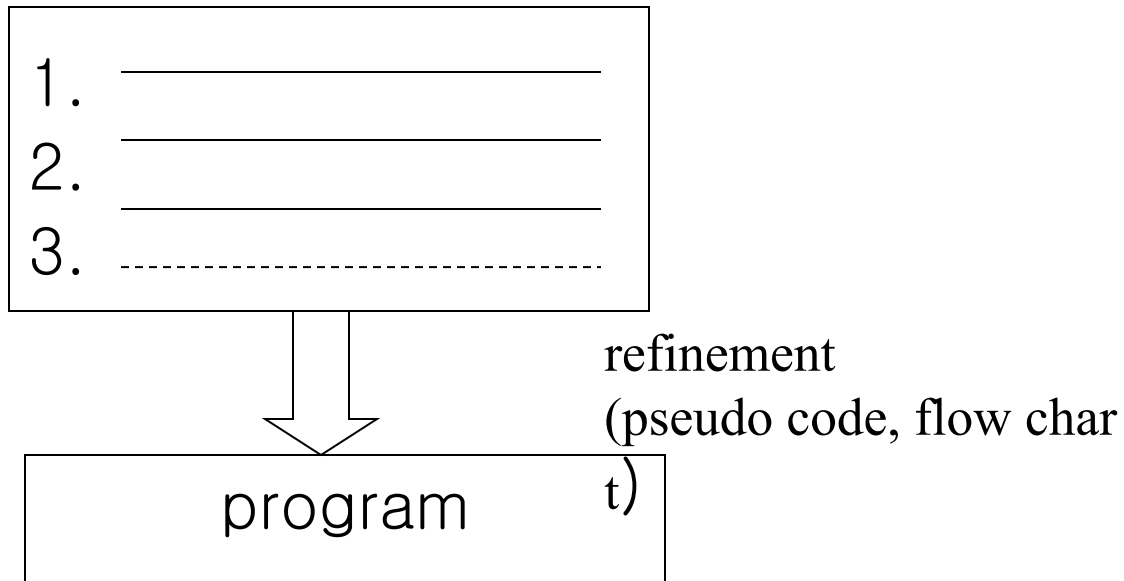
incremental search

## Macros

## Recursive functions

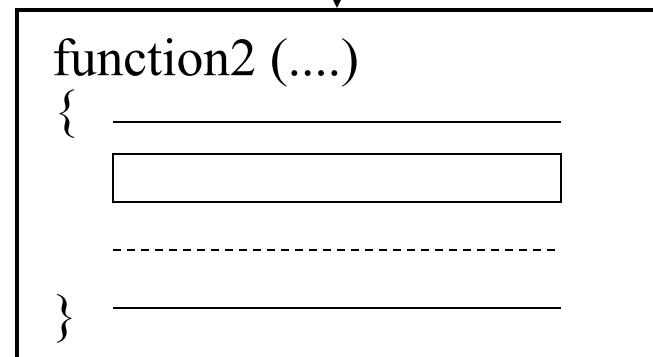
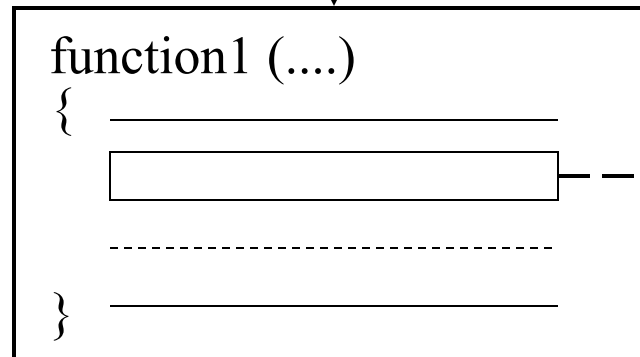
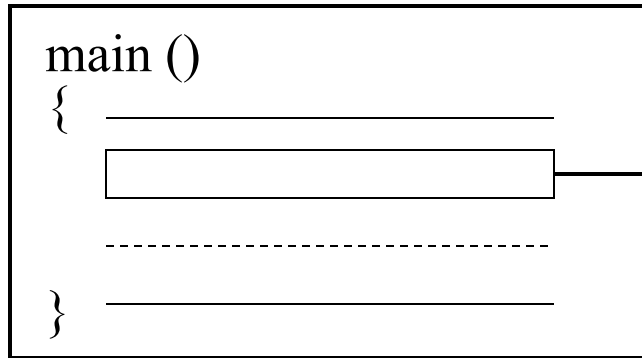
# Modularity

“Divide & Conquer.”  
Decomposition Outlines



What if a program is lengthy and complicated?

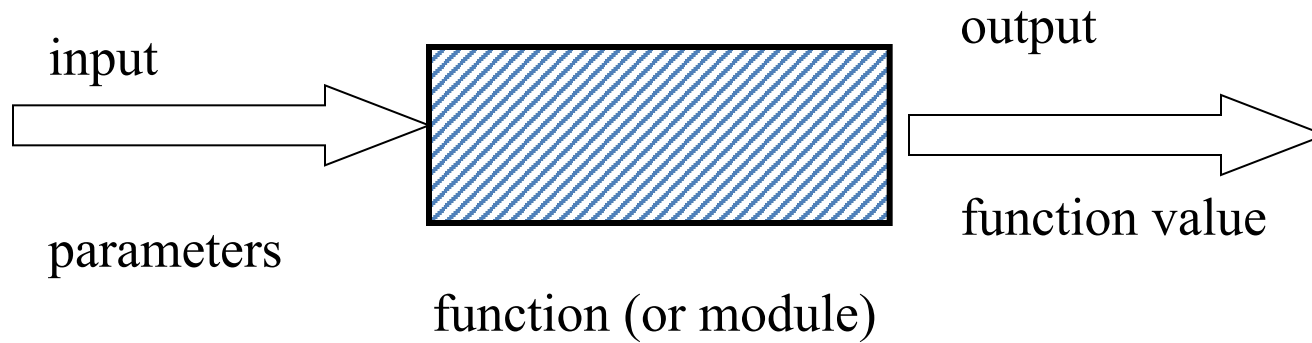
# Modular Programming



# Advantages

- Easier to write and understand
- Independently developed and tested
- Reusable. Once written and tested, a function can be used in other programs.
- Both the size of a program and its development costs reduced.

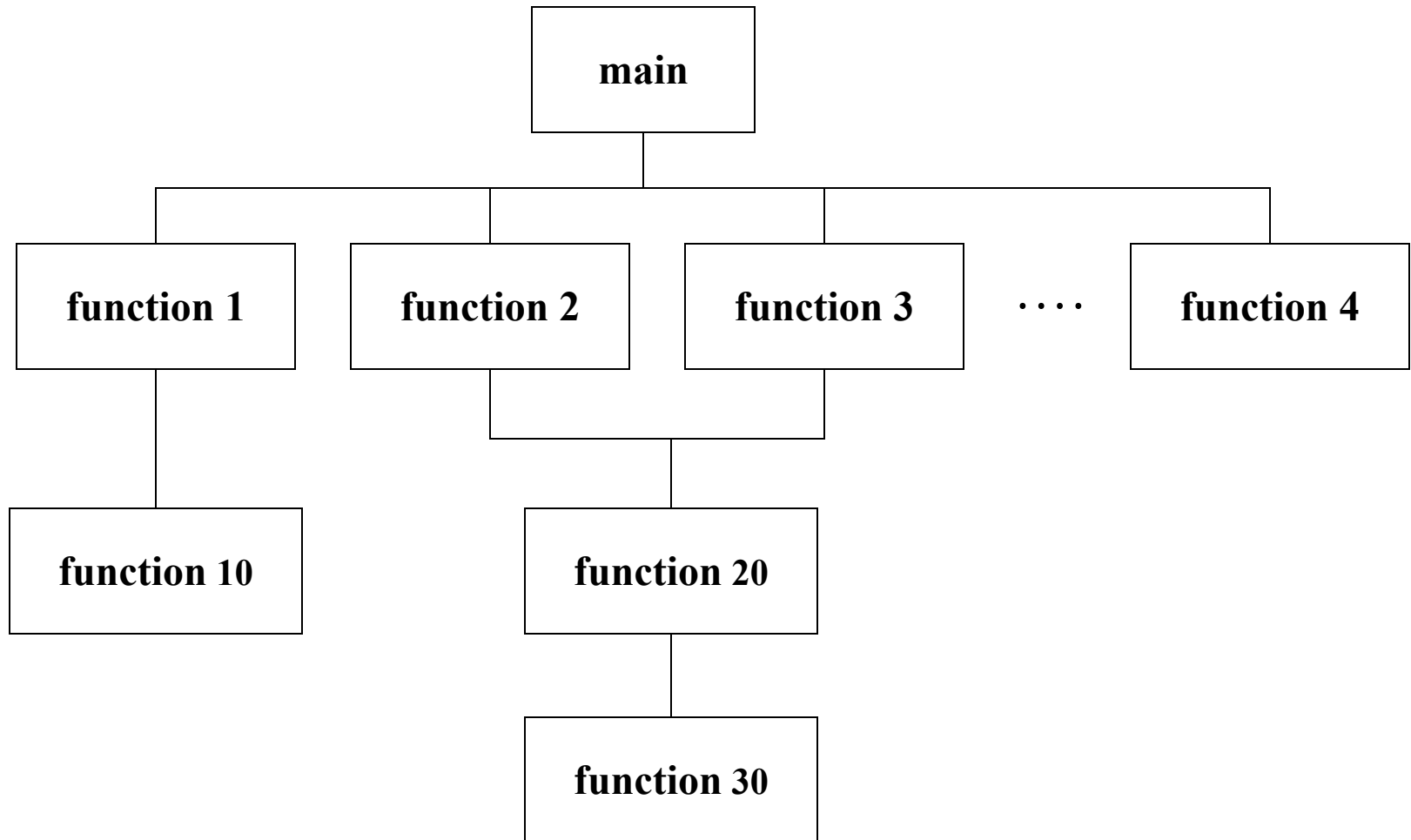
# Abstract



e.g. Standard C Lib. functions

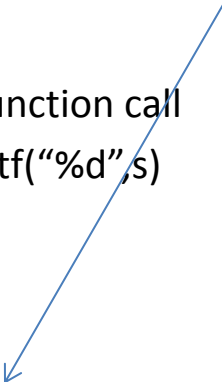


# Structured Charts(Modular Charts)



```
main()  
{ int s;  
s= sum(10,15);
```

```
// function call  
printf("%d",s)  
}
```



```
int sum( int a, int b) // function definition  
{  
int c ;  
c= a+b;  
if( a>0)  
return(c);  
else  
return(b);  
}
```

# Program Modules in C

- Functions
  - Modules in C
  - Programs combine user-defined functions with library functions
    - C standard library has a wide variety of functions
- Function calls
  - Invoking functions
    - Provide function name and arguments (data)
    - Function performs operations or manipulations
    - Function returns results

## **Function call analogy**

- Boss asks worker to complete task
  - Worker gets information, does task, returns result
  - Information hiding: boss does not know details

# Math Library Functions

- Math library functions
  - perform common mathematical calculations
  - **#include <math.h>**
- Format for calling functions
  - **FunctionName( *argument* );**
    - If multiple arguments, use comma-separated list
  - **printf( "%d", sqrt( 900 ) );**
    - Calls function **sqrt**, which returns the square root of its argument
    - All math functions return data type **double**
  - Arguments may be constants, variables, or expressions

# Functions

- Functions
  - Modularize a program
  - All variables declared inside functions are local variables
    - Known only in function defined
  - Parameters
    - Communicate information between functions
    - Local variables
- Benefits of functions
  - Divide and conquer
    - Manageable program development
  - Software reusability
    - Use existing functions as building blocks for new programs
    - Abstraction - hide internal details (library functions)
  - Avoid code repetition

# Functions

Three parts of function

- function declaration - prototype declaration/  
function signature
- function definition
- function call

# Function Prototypes

- Function prototype
  - Function name
  - Parameters – what the function takes in
  - Return type – data type function returns (default **int**)
  - Used to validate functions
  - Prototype only needed if function definition comes after use in program
  - The function with the prototype  
**int maximum( int, int, int );**
    - Takes in 3 **int**
    - Returns an **int**
- Promotion rules and conversions
  - Converting to lower types can lead to errors

# Function Definitions

- Function definition format

*return-value-type function-name( parameter-list )*  
*{*  
*declarations and statements*  
*}*

- Function-name: any valid identifier
- Return-value-type: data type of the result (default **int**)
  - **void** – indicates that the function returns nothing
- Parameter-list: comma separated list, declares parameters
  - A type must be listed explicitly for each parameter unless, the parameter is of type **int**



# Function Definitions

- Function definition format (continued)

*return-value-type function-name( parameter-list )*  
*{*  
*declarations and statements*  
*}*

- Declarations and statements: function body (block)
  - Variables can be declared inside blocks (can be nested)
  - **Functions can not be defined inside other functions**
- Returning control
  - If nothing returned **return;**
    - or, until reaches right brace
  - If something returned
    - **return** *expression*;

```

1  /* Fig. 5.4: fig05_04.c
2     Finding the maximum of three integers */
3  #include <stdio.h>
4
5  int maximum( int, int, int );    /* function prototype */
6
7  int main()
8  {
9     int a, b, c;
10
11     printf( "Enter three integers: " );
12     scanf( "%d%d%d", &a, &b, &c );
13     printf( "Maximum is: %d\n", maximum( a, b, c ) );
14
15     return 0;
16 }
17
18 /* Function maximum definition */
19 int maximum( int x, int y, int z )
20 {
21     int max = x;
22
23     if ( y > max )
24         max = y;
25
26     if ( z > max )
27         max = z;
28
29     return max;
30 }

```

1. Function  
prototype (3  
parameters)

2. Input values

2.1 Call function

3. Function  
definition

Program Output

```

Enter three integers: 22 85 17
Maximum is: 85

```

# Header Files

- Header files
  - Contain function prototypes for library functions
  - `<stdlib.h>` , `<math.h>` , etc
  - Load with `#include <filename>`  
`#include <math.h>`
- Custom header files
  - Create file with functions
  - Save as **filename.h**
  - Load in other files with `#include "filename.h"`
  - Reuse functions

# Calling Functions: Call by Value and Call by Reference

- Used when invoking functions
- Call by value
  - Copy of argument passed to function
  - Changes in function do not effect original
  - Use when function does not need to modify argument
  - Avoids accidental changes
- Call by reference
  - Passes original argument
  - Changes in function effect original
  - Only used with trusted functions

# Example of Call by Value

/\* Write a program to swap 2 integer numbers by using function \*/

```
#include<stdio.h>
```

```
#include<conio.h>
```

```
void swap(int,int);
```

/\* Function Declaration \*/

```
void main()
```

```
{
```

```
    int a=10,b=20;
```

```
    clrscr();
```

```
    swap(a,b);
```

/\* Function call \*/

```
    printf("\nOriginal values of a=%d & b=%d",a,b);
```

```
    getch();
```

```
}
```

(P..T.O.)

# Example of Call by Value

```
void swap(int x , int y)           /* Function Definition */
{
    int t;
    t=x;
    x=y;
    y=t;
    printf("\nSwapped values are x=%d y=%d",x,y);
}
```

# Example of Call by Reference

```
/* Program to swap two numbers using function call by reference */
```

```
#include<stdio.h>
```

```
#include<conio.h>
```

```
void swap(int * , int *);
```

```
/* Function Declaration */
```

```
void main()
```

```
{
```

```
    int a=10,b=20;
```

```
    clrscr();
```

```
    swap(&a,&b);
```

```
/* Function call */
```

```
    printf("\nSwapped values are a=%d b=%d",a,b);
```

```
    getch();
```

```
}
```

(P.T.O.)

# Example of Call by Reference

```
void swap(int *x ,int *y)           /* Function Definition */
{
    int t;
    t=*x;
    *x=*y;
    *y=t;
}
```



# Recursion

- Definition

Recursive function is a function which contains a call to itself.

- Why recursive function

Recursive function allows you to divide your complex problem into identical single simple cases which can handle easily.

This is also a well-known computer programming technique: divide and conquer.

- Warning of using recursive function

Recursive function must have at least one exit condition that can be satisfied. Otherwise, the recursive function will call itself repeatedly until the runtime stack overflows.

# Example of Recursion

/\* Write a program to calculate factorial of entered number using recursive function \*/

```
#include<stdio.h>
```

```
#include<conio.h>
```

```
int factorial (int n);
```

/\* Function Declaration \*/

```
void main()
```

```
{
```

```
    int fact, num;
```

```
    clrscr();
```

```
    printf(" enter the number");
```

```
    scanf("%d", &num);
```

```
    fact= factorial(num);
```

/\* Function call \*/

```
    printf(" factorial of number %d is %d", num, fact);
```

```
}
```

(P..T.O.)

# Example of Recursion

```
factorial (int n)                                /* Function Definition */
{
    if(n==1)
        return(1);
    else
        return(n* factorial(n-1));              /* Recursive call to function */
}
```