

C-Programming

Mathematical Physics-I (Lab)

Prepared by

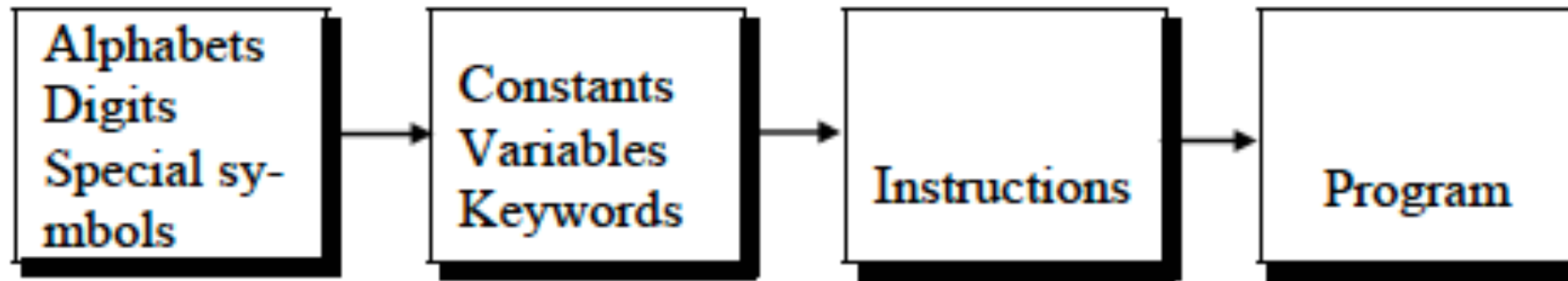
Dr. Ananya Phukan
Department of Physics
Mangaldai College

What is a program?

Steps in learning English language:



Steps in learning C:



Introduction to C

C is a middle-level procedure oriented programming language developed by **Dennis Ritchie** at AT's & T bell laboratories in the year **1972** in USA.

Four stages of C program:

- **Editing:** Writing the source code by using some IDE or editor
- **Preprocessing or libraries:** Already available routines
- **compiling:** translates or converts source to object code for a specific platform

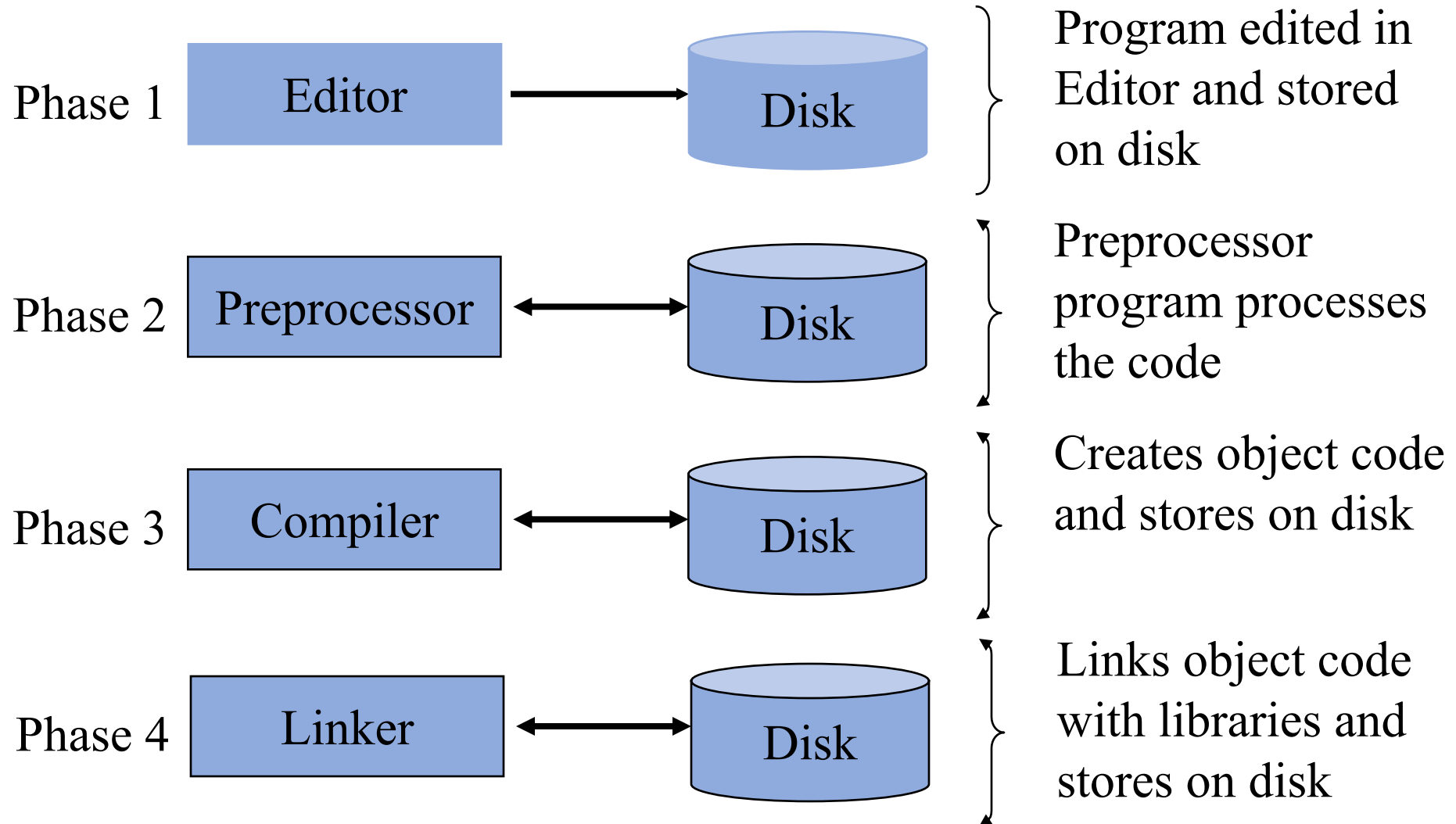
source code -> object code

- **linking:** resolves external references and produces the executable module

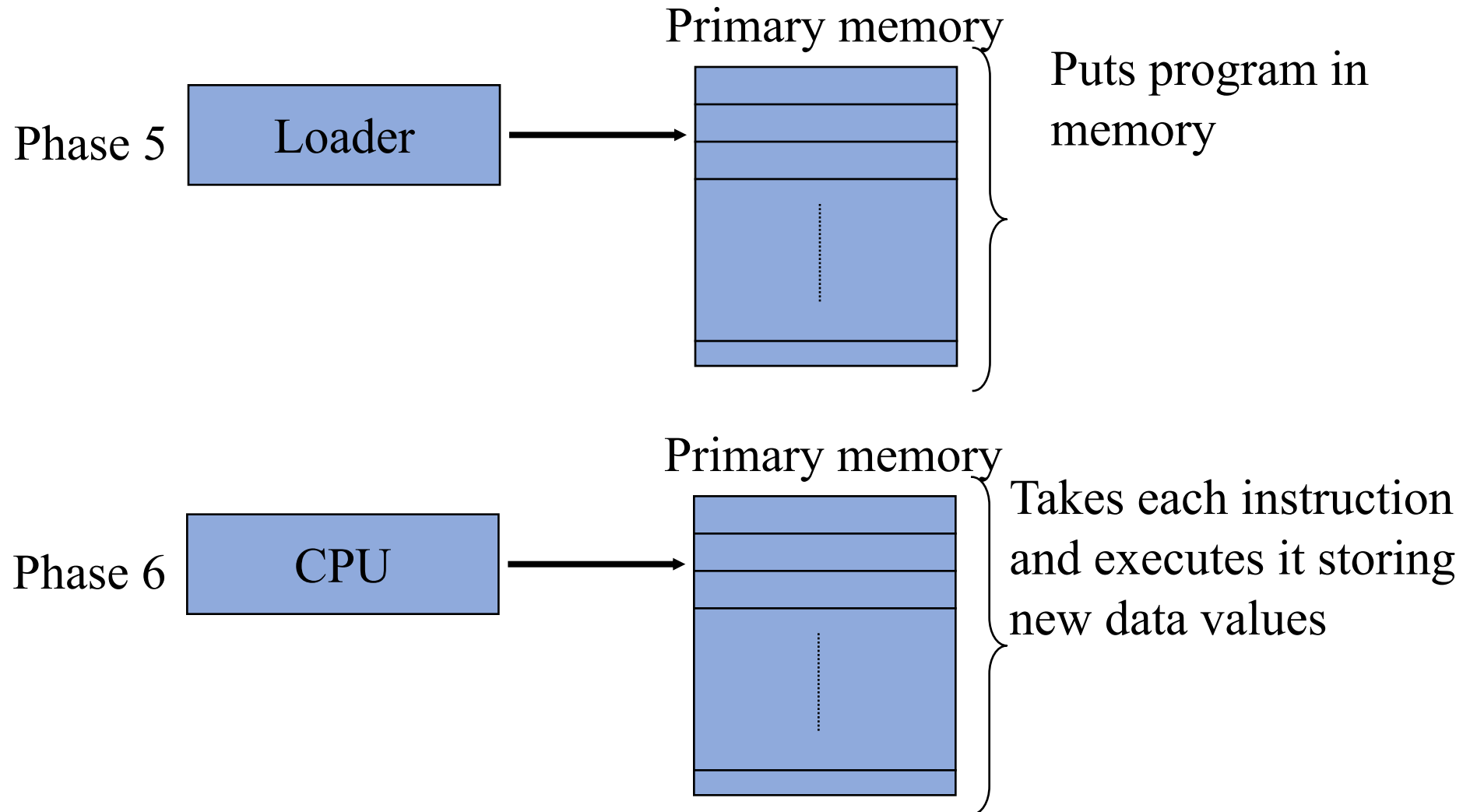
Basics of C Environment

- C systems consist of 3 parts
 - Environment
 - Language
 - C Standard Library
- Development environment has 6 phases
 - Edit
 - Pre-processor
 - Compile
 - Link
 - Load
 - Execute

Basics of C Environment



Basics of C Environment



The C Character Set

Alphabets	A, B,, Y, Z a, b,, y, z
Digits	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Special symbols	~ ' ! @ # % ^ & * () _ - + = \ { } [] : ; " ' < > , . ? /

Identifier

- Identifier refers to the name that is used to identify variables, functions and so on.

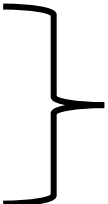
Example:

```
int a  
int b
```



a, b variables are identifiers

```
void sum()  
void fun()
```

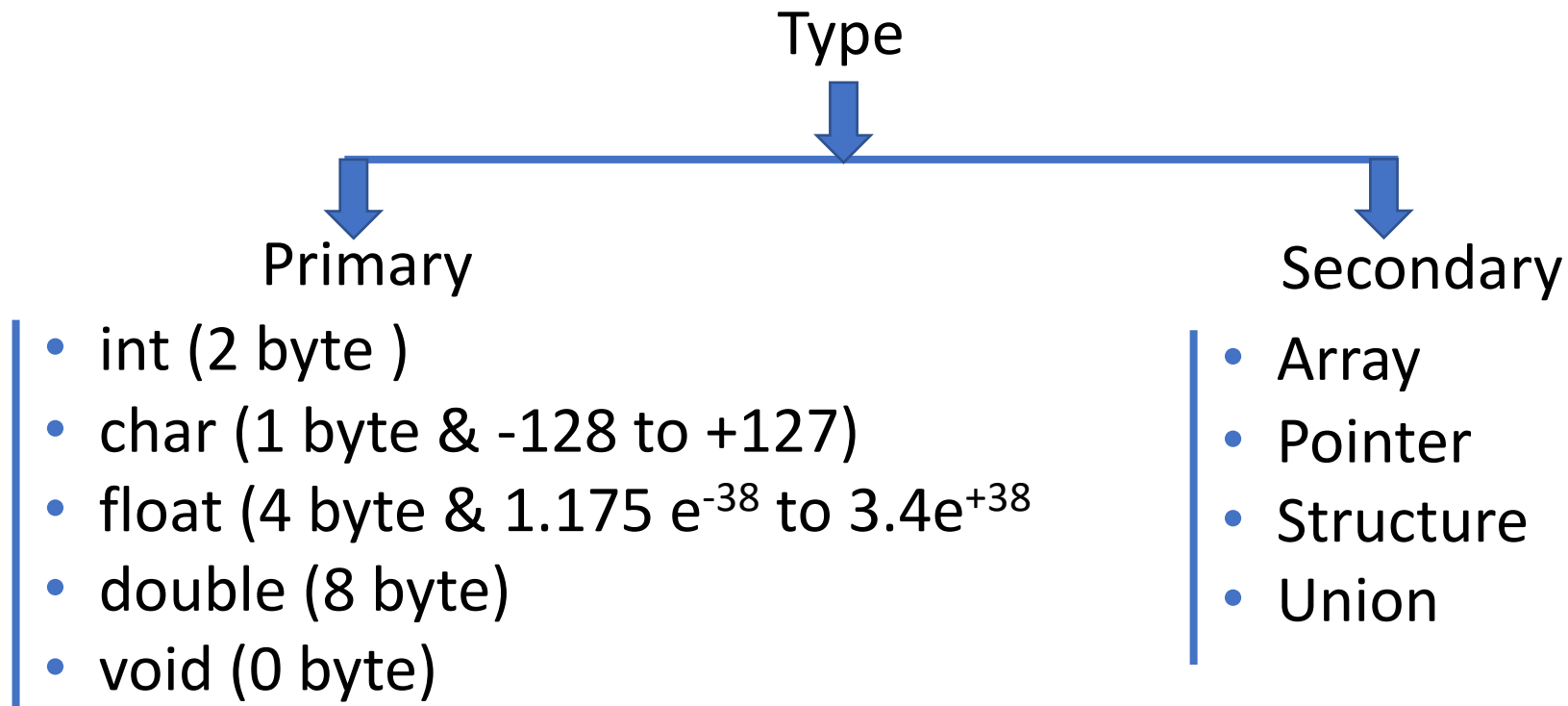


Functions sum() and fun() are identifiers

Data Type

- Data type specifies the size & type of values that can be stored in a variable.
- Examples:

int a= 10



C Constants

A constant is a value or an identifier whose value cannot be altered in a program.

■ Integer Constants

- It must have at least one digit with no decimal point.
- It can be either positive or negative
- The allowable range for integer constants is -32768 to 32767

Eg: 440, -21, +9000

■ Real Constants

- It must have at least one digit and a decimal point.
- It can be either positive or negative
- Range of real constants expressed in exponential form is -3.4×10^{38} to 3.4×10^{38}

Eg: +330.50, -68.99, 2.1×10^4 , -1.2×10^{-2}

C Constants

- **Character Constants**

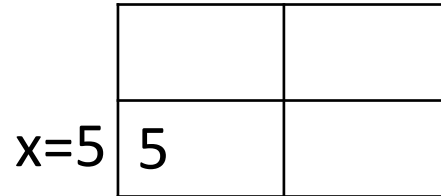
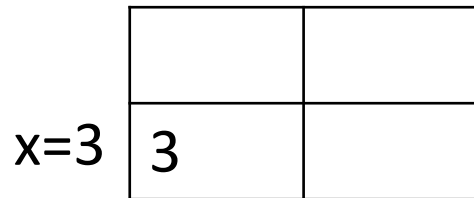
- A character constant is a single alphabet, a single digit or a single special symbol enclosed within single inverted commas.

- The maximum length of a character constant can be 1 character.

Eg: 'A', 'h', '5', '='

C Variables

- Variable is the name of **memory** location where we **store** data.
- These locations can contain integer, real or character constants.
- The variable values are not always same, a new value can overwrite the earlier value as shown below:



- The name of a variable can be composed of **letters**, **digits**, and the **underscore** character. It must begin with either a letter or an underscore. Upper and lowercase letters are distinct because C is case-sensitive.

C Variables

- Examples of type declaration statements:

Eg: `int si, x_hr;`

`float fourier;`

`char code;`

Keyword

- Keywords are the words whose meaning has already been explained to the C compiler.

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

Simple C Program

```
#include <stdio.h>
/* My first C program which prints Hello World */
int main (int argc, char *argv[])
{
    printf ("Hello World!\n");
    return 0;
}
```

Preprocessor

Comments are good

main() means "start here"

Library command

Return 0 from main means our program finished without errors

Brackets define code blocks

Simple C Program

Line 1: `#include <stdio.h>`

- As part of compilation, the C compiler runs a program called the **C preprocessor**. The preprocessor is able to add and remove code from your source file.
- In this case, the **directive #include** tells the preprocessor to include code from the file **stdio.h**.
- This file contains declarations for functions that the program needs to use. A declaration for the **printf** function is in this file.

Simple C Program

Line 2: `void main()`

- This statement declares the **main function**.
- A C program can contain many functions but must always have one main function.
- A function is a self-contained module of code that can accomplish some task.
- Functions are examined later.
- The "void" specifies the return type of main. In this case, nothing is returned to the operating system.

Simple C Program

Line 3: {

- This opening bracket denotes the start of the program.

Simple C Program

Line 4: `printf("Hello World From About\n");`

- **Printf** is a function from a standard C library that is used to print strings to the standard output, normally your screen.
- The compiler links code from these standard libraries to the code you have written to produce the final executable.
- The **"\n"** is a special format modifier that tells the **printf** to put a line feed at the end of the line.
- If there were another **printf** in this program, its string would print on the next line.

Simple C Program

Line 5: }

- This closing bracket denotes the end of the program.

- All output to screen is achieved using readymade library functions. One such function is **printf**.

- The general form of **printf()** function is,

printf ("<format string>", <list of variables>);

<format string> can contain,

%f for printing **real values**

%d for printing **integer values**

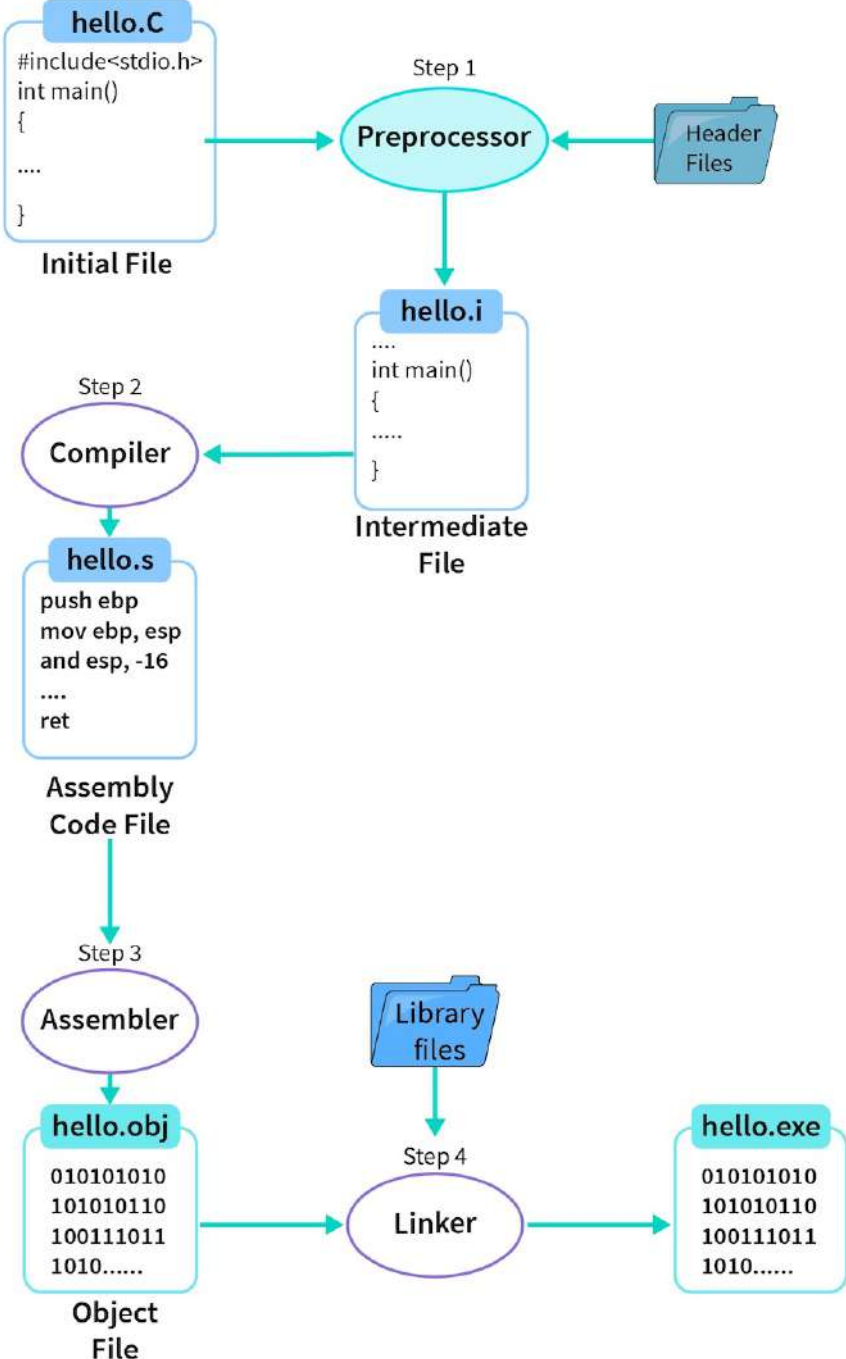
%c for printing **character values**

Compilation and Execution

- The C program is typed in a program called *editor*.
- After typing, it is converted to machine language (0's and 1's) before the machine can execute it with the help of a *compiler*.
- Compiler vendors provide an **Integrated Development Environment** (IDE) which consists of an Editor as well as the Compiler.

Eg: Turbo C, Turbo C++, Visual C++, gcc etc.

Compilation Process



C program-2

```
/* Calculation of simple interest */  
int main( )  
{  
    int p, n ;  
    float r, si ;  
    p = 1000 ;  
    n = 3 ;  
    r = 8.5 ;  
    si = p * n * r / 100 ; /* formula for simple interest */  
    printf ( "%f" , si ) ;  
}
```


Receiving Input

- A user can supply values of variables through the keyboard during execution by using a function called *scanf()*.
- *printf()* outputs the values to the screen whereas *scanf()* receives them from the keyboard.

```
/* Calculation of simple interest */
int main( )
{
    int p, n ;
    float r, si ;
    printf ( "Enter values of p, n, r" ) ;
    scanf ( "%d %d %f", &p, &n, &r ) ;
    si = p * n * r / 100 ;
    printf ( "%f" , si ) ;
}
```

& is an **'Address of'** operator. It gives the location number used by the variable in memory

Type Conversion in Assignments

- It may so happen that the type of the expression and the type of the variable on the left-hand side of the assignment operator may not be same. In such a case the value of the expression is promoted or demoted depending on the type of the variable on left-hand side of =.

- Example:

```
int i;  
i=3.5;
```

 } Since i is an integer, the value of i that is stored is 3.
Float is converted to integer.

- ```
float b;
b=30;
```

 } Since b is a float, the value of b that is stored is  
30.00000.  
Integer is converted to float.

# C Instructions

- There are basically three types of instructions in C:
  - (a) **Type Declaration Instruction**- To declare the type of variables used in a C program.

Eg: `int a; float rs; char name;`

`int i=11, j=2; float a=10.5, b=2.3+1.1*3.44;`

- (b) **Arithmetic Instruction**- To perform arithmetic operations between constants and variables.

Eg: `int i; float j, k, l, m, n;`

`i=567; j=0.00234;`

`k=l*m/n+2.3*5/2;`

# C Instructions

(c) **Control Instruction**- To control the sequence of execution of various statements in a C program. There are four types of control instructions in C.

- **Sequence Control Instruction**
- **Selection or Decision Control Instruction**
- **Repetition or Loop Control Instruction**
- **Case Control Instruction**

# Operator

- Operator is a symbol that tells the compiler to perform mathematical and logical task.
- Types of operator:
  1. Arithmetic operator (+, -, \*, ÷, %)
  2. Relational operator (<, >, <=, >=, ==, !=)
  3. Logical operator (&&, !)
  4. Increment / Decrement (++ , --)
    - pre-increment/pre-decrement (++a, --a)
    - post-increment/post-decrement (a++, a--)
  5. Ternary operator (?:)
  6. Assignment operator (=)

# Precedence order

- **Highest to lowest**

1. ()

2. \*, /, %

3. +, -

# Example

Algebra:

$$z = pr\%q+w/x-y$$

C:

```
z = p * r % q + w / x - y ;
```

Precedence:

1      2      4      3      5

# Example

Algebra:

$$a(b+c)+ c(d+e)$$

C:

$$a * ( b + c ) + c * ( d + e ) ;$$

Precedence:

3      1      5      4      2



| Algebraic Expression                                | C Expression                            |
|-----------------------------------------------------|-----------------------------------------|
| $a \times b - c \times d$                           | $a * b - c * d$                         |
| $(m + n) (a + b)$                                   | $(m + n) * (a + b)$                     |
| $3x^2 + 2x + 5$                                     | $3 * x * x + 2 * x + 5$                 |
| $\frac{a + b + c}{d + e}$                           | $(a + b + c) / (d + e)$                 |
| $\left[ \frac{2BY}{d+1} - \frac{x}{3(z+y)} \right]$ | $2 * b * y / (d + 1) - x / 3 * (z + y)$ |

# Decision Control in C

- When different sets of instructions are executed in different situations in a C program, then a ***decision control instruction*** is used by using:
  - (a) **if** statement
  - (b) **if-else** statement
  - (c) Conditional operators

# The *if* statement

- The general form of **if** statement looks like:

***if (this condition is true)***  
***execute this statement;***

| <b>this expression</b> | <b>is true if</b>               |
|------------------------|---------------------------------|
| $x == y$               | x is equal to y                 |
| $x != y$               | x is not equal to y             |
| $x < y$                | x is less than y                |
| $x > y$                | x is greater than y             |
| $x \leq y$             | x is less than or equal to y    |
| $x \geq y$             | x is greater than or equal to y |

# Example (*if* statement)

```
/* Demonstration of if statement */
main()
{
 int num ;

 printf ("Enter a number less than 10 ") ;
 scanf ("%d", &num) ;

 if (num <= 10)
 printf ("What an obedient servant you are !") ;
}
```

# The *if-else* statement

**Example:** In a company an employee is paid as under:

If his basic salary is less than Rs. 1500, then HRA = 10% of basic salary and DA = 90% of basic salary. If his salary is either equal to or above Rs. 1500, then HRA = Rs. 500 and DA = 98% of basic salary. If the employee's salary is input through the keyboard write a program to find his gross salary.

```
/* Calculation of gross salary */
main()
{
 float bs, gs, da, hra ;

 printf ("Enter basic salary ") ;
 scanf ("%f", &bs) ;

 if (bs < 1500)
 {
 hra = bs * 10 / 100 ;
 da = bs * 90 / 100 ;
 }
 else
 {
 hra = 500 ;
 da = bs * 98 / 100 ;
 }

 gs = bs + hra + da ;
 printf ("gross salary = Rs. %f", gs) ;
}
```

# Logical Operators

- C allows usage of three logical operators, namely,

**&& - AND**

**|| - OR**

**! - NOT**

- The operators **&&** and **||**, allow two or more conditions to be combined in an **if** statement.

# Example (Logical operators in if statement)

Q. The marks obtained by a student in 5 different subjects are input through the keyboard. The student gets a division as per the following rules:

Percentage above or equal to 60 - First division

Percentage between 50 and 59 - Second division

Percentage between 40 and 49 - Third division

Percentage less than 40 - Fail

Write a program to calculate the division obtained by the student.

Two ways to write the program:

1. With **Nested if-else** statement.
2. With **Logical operators**.



# && and || operators:

- The && and || are useful in the following programming situations:
  - (a) When it is to be tested whether a value falls within a particular range or not.
  - (b) When after testing several conditions the outcome is only one of the two answers (This problem is often called yes/no problem).

# The ! (NOT) Operator

- This operator **reverses the result** of the expression it operates on. For example, if the expression evaluates to a non-zero value, then applying ! operator to it results into a 0.

Eg: ! (y<10)

This means “not **y** less than 10”. In other words, if **y** is less than 10, the expression will be false, since ( **y < 10** ) is true. We can express the same condition as ( **y >= 10** ).

# Hierarchy of Operators Revisited

| Operators | Type                   |
|-----------|------------------------|
| !         | Logical NOT            |
| * / %     | Arithmetic and modulus |
| + -       | Arithmetic             |
| < > <= >= | Relational             |
| == !=     | Relational             |
| &&        | Logical AND            |
|           | Logical OR             |
| =         | Assignment             |

| Operands |          | Results |    |        |        |
|----------|----------|---------|----|--------|--------|
| x        | y        | !x      | !y | x && y | x    y |
| 0        | 0        | 1       | 1  | 0      | 0      |
| 0        | non-zero | 1       | 0  | 0      | 0      |
| non-zero | 0        | 0       | 1  | 0      | 1      |
| non-zero | non-zero | 0       | 0  | 1      | 1      |

# The Conditional Operators

- The conditional operators **?** and **:** are called **ternary operators** since they take three arguments. They form a kind of foreshortened if-then-else. Their general form is:

***expression 1 ? expression 2 : expression 3***

This expression says that,

*“if **expression 1** is true, then the value returned will be **expression 2**, otherwise the value returned will be **expression 3**”.*

# Example of conditional operators

```
1. int x, y;
 scanf("%d", &x);
 y=(x>5?3:4);
```



```
if (x>5)
 y=3;
else
 y=4
```

This statement will store 3 in **y** if **x** is greater than 5, otherwise it will store 4 in **y**.

```
2. char a;
 int y;
 scanf("%c",&a)
 y=(a>=65 && a<=90 ? 1 :0);
```



1 would be assigned to **y** if **a>=65 && a<=90** evaluates to true, otherwise 0 would be assigned

3. Apart from arithmetic statements, conditional operators are used as shown below:

```
eg: int i;
 scanf("%d", &i);
 (i==1 ? printf("Amit") : printf("Prakash"));
```

```
eg: char a='z';
 printf("%c",(a>='a' ? a : '!'));
```

4. The conditional operators can be nested as shown below:

```
int big, a, b, c;
```

```
big = (a > b ? (a > c ? 3 : 4) : (b > c ? 6 : 8));
```



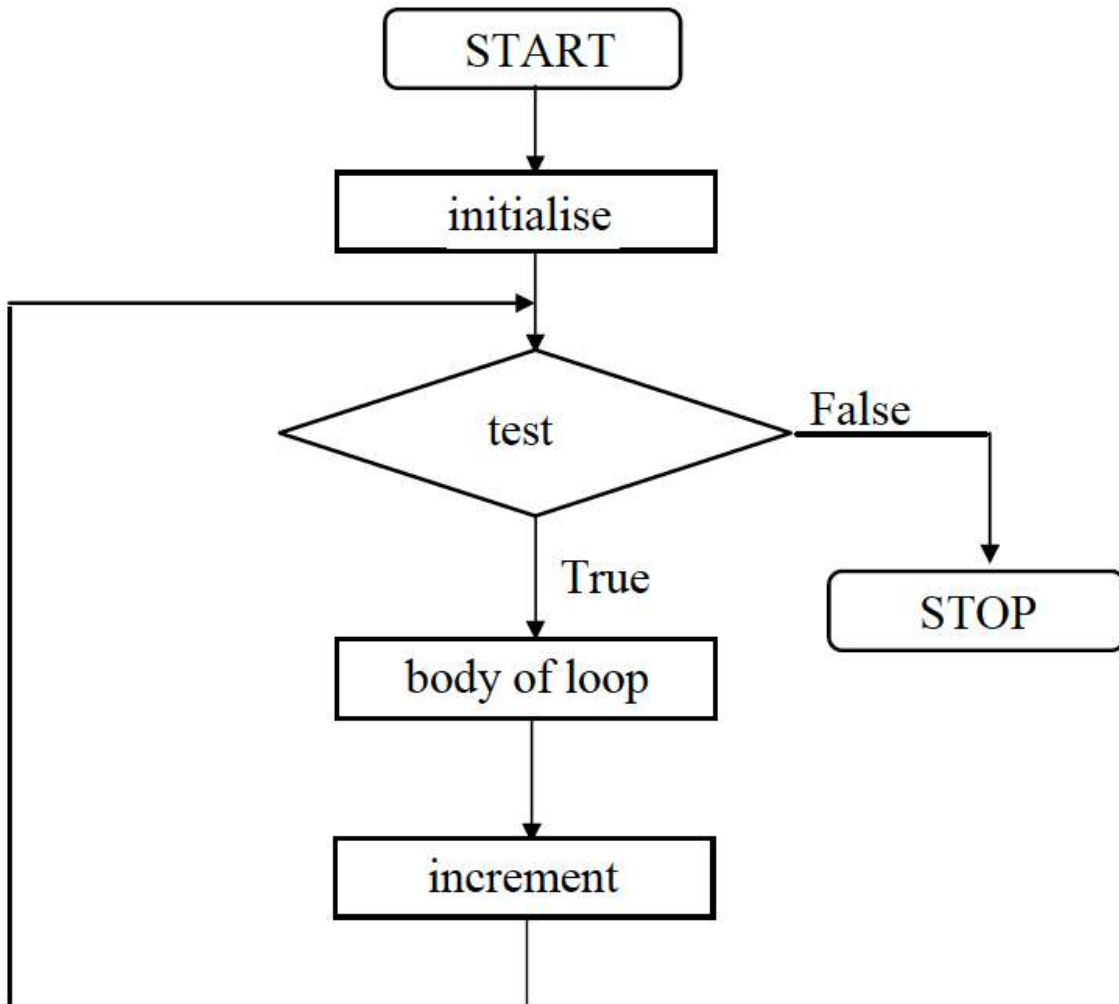
# The Loop Control Structure

- When some portion of a program is repeated for a specified number of times or performed until a particular condition is being satisfied, then this repetitive operation is done through a **loop control instruction**.
- **Three methods** to repeat a part of a program. They are:
  - (a) Using a **for statement**
  - (b) Using a **while statement**
  - (c) Using a **do-while statement**

# The *while* loop

- Calculation of simple interest for 3 sets of p, n and r.

```
int main()
{ int p, n, count;
 float r, si;
 count =1;
 while (count<=3)
 { printf ("Enter the values of p, n and r");
 scanf ("%d %d %f", &p, &n, &r);
 si = p*n*r/100;
 count = count +1;
 }
}
```



```
Initialise loop counter;
while (test loop counter using a condition)
{
 do this;
 and this;
 increment loop counter;
}
```

# while loop...

- The statements within the **while** loop would keep on getting executed till the condition being tested remain true. When the condition becomes false, the control passes to the first statement that follows the body of the while loop.
- Expressions are also used in place of the condition. The statements within the loop get executed as long as the expression evaluates to a non-zero value.
- The condition being tested may use relational or logical operators.

***while(i<=10),      while(i>=10&&j<=15),      while(j>10&&(b<15 | |c<20))***

# while loop...

- We can even decrement the loop counter and still manage to get the body of the loop executed repeatedly.

```
main()
{
 int i = 5 ;
 while (i >= 1)
 {
 printf ("\nMake the computer literate!") ;
 i = i - 1 ;
 }
}
```

# while loop...

- The statements within the loop may be a single line or a block of statements.
- The while must test a condition that will eventually become false, otherwise the loop would be executed forever, indefinitely.

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

Indefinite loop, since **i** remains equal to 1 forever

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

*Correct form*

# while loop...

- A loop counter can be a float also.

Example:

```
int main
{
 float a=10.0;
 while (a<=10.5)
 {
 printf("Rose is red");
 a=a+0.1;
 }
}
```

Consider a problem where numbers from 1 to 10 are to be printed on the screen.

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

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

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

```
(d) main()
{
 int i = 0;
 while (i++ < 10)
 printf ("%d\n", i);
}
```

```
(e) main()
{
 int i = 0;
 while (++i <= 10)
 printf ("%d\n", i);
}
```



# The *for* Loop

- The general form of **for** statement is :

```
for (initialise counter ; test counter ; increment counter)
{
 do this;
 and this;
 and this;
}
```

# The *for* Loop

```
/* Calculation of simple interest for 3 sets of p, n and r */
main ()
{
 int p, n, count ;
 float r, si ;

 for (count = 1 ; count <= 3 ; count = count + 1)
 {
 printf ("Enter values of p, n, and r ") ;
 scanf ("%d %d %f", &p, &n, &r) ;

 si = p * n * r / 100 ;
 printf ("Simple Interest = Rs.%f\n", si) ;
 }
}
```

```
for (i = 10 ; i ; i --)
 printf ("%d", i) ;
```

```
for (i < 4 ; j = 5 ; j = 0)
 printf ("%d", i) ;
```

```
for (i = 1 ; i <= 10 ; printf ("%d", i++) ;
```

```
for (scanf ("%d", &i) ; i <= 10 ; i++)
 printf ("%d", i) ;
```

# The *break* statement

We often come across situations where we want to jump out of a loop instantly, without waiting to get back to the conditional test. The keyword **break** allows us to do this. When **break** is encountered inside any loop, control automatically passes to the first statement after the loop. A **break** is usually associated with an **if**.

# The *continue* Statement

In some programming situations we want to take the control to the beginning of the loop, bypassing the statements inside the loop, which have not yet been executed. The keyword **continue** allows us to do this. When **continue** is encountered inside any loop, control automatically passes to the beginning of the loop.

A **continue** is usually associated with an **if**.

# The *do-while* loop

The **do-while** loop looks like:

```
do
{
 this;
 and this;
 and this;
} while(this condition is true);
```

# Difference between **while** and **do-while**

- This difference is the place where the condition is tested. The **while** tests the condition before executing any of the statements within the **while** loop. As against this, the **do-while** tests the condition after having executed the statements within the loop.
- This means that **do-while** would execute its statements at least once, even if the condition fails for the first time. The **while**, on the other hand will not execute its statements if the condition fails for the first time.

# Do-while loop

- **break** and **continue** are used with **do-while** just as they would be in a **while** or a **for** loop. A **break** takes you out of the **do-while** bypassing the conditional test. A **continue** sends you straight to the test at the end of the loop.

```
main()
{
 while (4 < 1)
 printf ("Hello there \n");
}
```

```
main()
{
 do
 {
 printf ("Hello there \n");
 } while (4 < 1);
}
```



# *Case Control* Structure

- The control statement that allows us to make a decision from the number of choices is called a **switch**, or more correctly a **switch-case-default**, since these three keywords go together to make up the control statement.

- It appears as:

```
switch (integer expression)
{
 case constant 1 :
 do this ;
 case constant 2 :
 do this ;
 case constant 3 :
 do this ;
 default :
 do this ;
}
```

# Switch Case Example

```
main()
{
 int i = 2 ;

 switch (i)
 {
 case 1 :
 printf ("I am in case 1 \n") ;
 case 2 :
 printf ("I am in case 2 \n") ;
 case 3 :
 printf ("I am in case 3 \n") ;
 default :
 printf ("I am in default \n") ;
 }
}
```

# Switch Case Example using break

```
main()
{
 int i = 2;

 switch (i)
 {
 case 1 :
 printf ("I am in case 1 \n");
 break ;
 case 2 :
 printf ("I am in case 2 \n");
 break ;
 case 3 :
 printf ("I am in case 3 \n");
 break ;
 default :
 printf ("I am in default \n");
 }
}
```

- It is not necessary to arrange the cases in ascending order always. You can put the cases in any order you please.
- Character values are also allowed to use in case and switch.
- At times we may want to execute a common set of statements for multiple **cases**.
- Every statement in a **switch** must belong to some **case** or the other. If a statement doesn't belong to any **case** the compiler won't report an error. However, the statement would never get executed.
- If we have no **default** case, then the program simply falls through the entire **switch** and continues with the next instruction (if any,) that follows the closing brace of **switch**.
- All that we can have after the case is an **int** constant or a **char** constant or an expression that evaluates to one of these constants. Even a **float** is not allowed.

- We can check the value of any expression in a **switch**. Thus the following **switch** statements are legal.

switch ( i + j \* k )

switch ( 23 + 45 % 4 \* k )

switch ( a < 4 && b > 7 )

- Expressions can also be used in cases provided they are constant expressions. Thus **case 3 + 7** is correct, however, **case a + b** is incorrect.
- The **break** statement when used in a **switch** takes the control outside the **switch**. However, use of **continue** will not take the control to the beginning of **switch** as one is likely to believe.