C Programming Language

B. C. Dhara

Department of Information Technology

Jadavpur University

Chapter 1
Introduction

Programming?

- A set (finite) of related instruction to solve a particular problem.
- Example:
  - $(a-b)^2$ => K=a-b; result=k*k
  - Celsius and Fahrenheit conversion [c/5 = (F-32)/9]
    - T1=F-32
    - T2=T1/9
    - result=T2*5

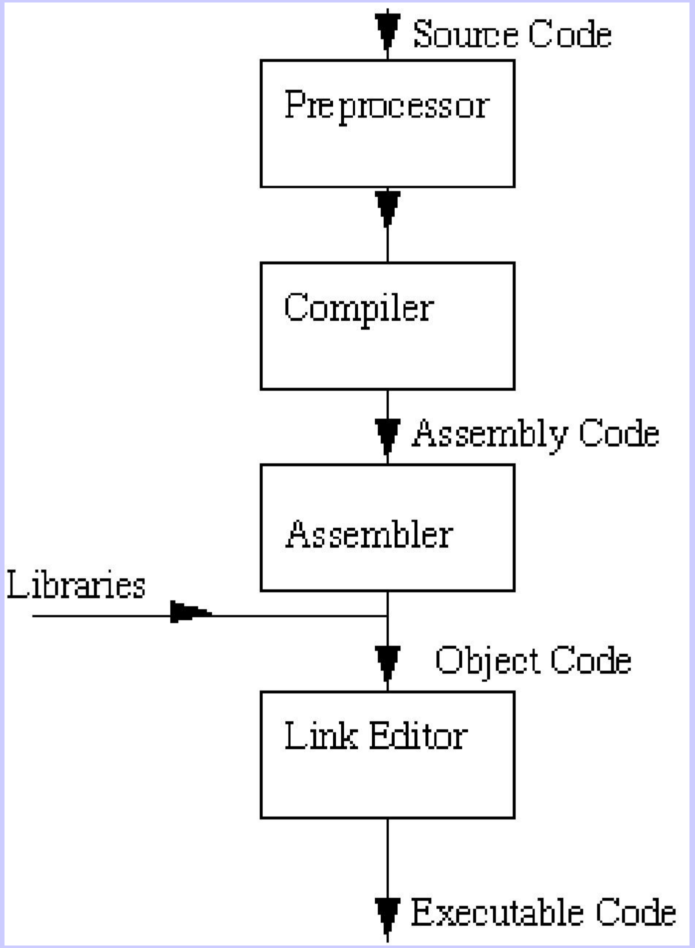
Programming

Designing a program means defining the logical flow of a program, what the program will take as input and what it will produce as output

The various steps involved in program development are,

- Analyzing or Defining the problem
- Developing a solution technique (Algorithm)
- Coding
- Documenting the problem
- Compiling and Running the program
- Testing and Debugging
- Maintenance

The C Compilation Model



Programming

- Compiling is a process in which the source program instructions are translated into a form that is suitable for execution by the computer.
- The compiler does the translation after examining each instruction for its correctness. The translation results in the creation of object code. If necessary, Linking is also done.
- Linking is the process of putting together other program files and functions that are required by the program.
- After compilation, the program can be executed. During execution, the executable object code is loaded into the computer memory and the program instructions are executed

Testing and Debugging

- **Testing is the process of executing a program with the deliberate intent of finding errors.**
- **Some programmers use the terms “testing” and “debugging” interchangeably, but careful programmers distinguish between the two activities.**
- **Testing is a means of detecting errors. Debugging is a means of diagnosing and correcting their root causes.**

What is a programming language?

- It is a vocabulary and set of grammatical rules for instructing a computer to perform specific tasks.
- Usually refers to high-level languages, such as BASIC, C, C++, COBOL, FORTRAN, Ada, and Pascal

Why study Programming Language?

- To improve a one’s ability to use programming languages effectively & efficiently
- To improve one’s ability to develop better & smarter algorithms
- To improve one’s knowledge on useful programming constructs
- To enable a better choice of programming languages which would tailor to one’s specific requirement.
- To make it easier to learn & design a new language

History of C Language

- ALGOL60 was developed by an International committee in year the1960
- Combined Programming Language was developed at Cambridge University in the year 1963.
- Basic Combined Programming language was developed by Martin Richards at Cambridge University in 1967.
- In 1970, B language was developed by Ken Thompson at Bell Labs.
- In 1972, C language was developed by Dennis Ritchie at AT & T Bell labs.

Introduction

- Lower level language (LLL)
- Higher level language (HLL)
- Hello world

```
    /* The Hello World program */
    # include <stdio.h>
main()
   {
     printf(“Hello, World”);
   }
```

- printf is a library function that prints the outputs
- #include<stdio.h> tells complier to include information about the standard input/output library

How to run this program?

```
cc hello.c
./a.out
```

Introduction

```
/* The Hello World
    program */
    # include <stdio.h>
main()
      {
        printf(“Hello”);
       printf(“,  World”);
       printf(“\n”);
     }
```

Escape sequence provides a general and extensible mechanism for representing ***hard-to-type or invisible characters***

Escape sequences

\n newline

\t horizontal Tab

\v vertical Tab

\b backspace

\r carriage Return

\\ backslash

\’ single quotation

\” double quotation

\0 Null character

\? Question mark

Assignment: print your first name at first line, middle name (if any) at second line and last name at third line.

A simple program

```
#include<stdio.h>
main()
{
int a;
int b;
int c;
a=10;
b=20;
c=a+b;
printf(“%d”,c);
}
```

```
#include<stdio.h>
main()
{
int a, b, c;     /* variables*/
a=10;
b=20;
c=a+b;
printf(“%i\n”,c);
printf(“%u”,c);
}
```

statements must be terminated with ”;”

Character Set

00	01	02	03	04	05	06	07
08	09	0A	0B	0C	0D	0E	0F
10	11	12	13	14	15	16	17
18	19	1A	1B	1C	1D	1E	1F
20	21 !	22 "	23 #	24 $	25 %	26 &	27 '
28 (	29)	2A *	2B +	2C ,	2D -	2E .	2F /
30 0	31 1	32 2	33 3	34 4	35 5	36 6	37 7
38 8	39 9	3A :	3B ;	3C <	3D =	3E >	3F ?
40 @	41 A	42 B	43 C	44 D	45 E	46 F	47 G
48 H	49 I	4A J	4B K	4C L	4D M	4E N	4F O
50 P	51 Q	52 R	53 S	54 T	55 U	56 V	57 W
58 X	59 Y	5A Z	5B [	5C \	5D]	5E ^	5F _
60 `	61 a	62 b	63 c	64 d	65 e	66 f	67 g
68 h	69 i	6A j	6B k	6C l	6D m	6E n	6F o
70 p	71 q	72 r	73 s	74 t	75 u	76 v	77 w
78 x	79 y	7A z	7B {	7C |	7D }	7E ~	7F

Data types

- Fundamental data types are data types implemented at the lowest level, i.e., those which are used for actual data representation in memory.
- Since the fundamental data types are implemented at the machine level, their storage requirement is machine-dependent.
- The available data types are
  - int
  - float
  - double
  - char
  - In addition, some modifiers or qualifiers (signed, unsigned, long, etc.)
  - The qualifier signed and unsigned may be applied to char or int

Data types (Contd…)

Type	Bytes	Range
short int	2	-32,768 - +32,767
unsigned short int	2	0 - +65,535
unsigned int	4	0 - +4,294,967,295
int	4	-2,147,483,648 - +2,147,483,647
signed char	1	-128 - +127
unsigned char	1	0 - +255
float	4	-3.4e38 – +3.4e38
double	8	-1.7e308 – +1.7e308
long double	10	-1.7e4932 - +1.7e4932

Derived Data Types

Derived data types, a combination of primitive data types. They are used to represent a collection of data. They are

- Arrays
- Structure
- Unions
- Typedef
- Enumerated
- Pointers

- C language is case sensitive

Data types

- In **C**, all variables must be declared before they are used, usually at the beginning of a function, before any executable statements.
- A declaration announces the properties of variables; it consists of a **type** name, and a list of variables.
- The type of a variable actually determines the type of operations that can be performed on a variable of that type.
- The definition of a variable will assign storage for the variable and define the type of data that will be held in the location.

C identifiers

- In C programming, identifiers are names given to C entities, such as
  - variables,
  - functions,
  - structures etc.
- Identifier are created to give unique name to C entities to identify it during the execution of program.

Rules to define identifier

- Rules to define identifier:
  - considering only letters and digits
  - underscore "_" in the alphabet set
  - first character must be a letter/_
  - don't begin with "_", as library routines often use such names (suggestion)
  - There is no bar on the length of the identifier

- The identifier name should be related with the data it represents.
- Example: a, b, b0 dasf012 , 1s, dg-df,

Keywords

- They have specified meaning; user cannot use them as variable

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

Introduction

- Comments

```
/* A sample C program */
  # include <stdio.h>



    main()
    {
      int a=10, b=20; /* declaration of variables*/
      int c;
      c = a+b;
      printf("%d+%d = %d \n",a,b,c);
    }
```

```
Output: 10+20=30
```

Defining Data

Data Definition	Data Type	Memory Defined	Size	Value Assigned
char x, y	char	x	1 byte	-
		y	1 byte	-
int a, m=22;	int	m	4 bytes	22
		a	4 bytes	-
float num	float	num	4 bytes	-
float num1= 9.67	float	num1	4 bytes	9.67

C Environment

- The **C** environment assumes the keyboard to be the standard input device referred to as **stdin**

- The VDU is assumed to be the standard output device referred to as **stdout**

- The VDU also serves as the standard error device, and is referred to as **stderr**

Input/Output Functions

- **C** supports Input/Output operations through functions written in **C**, and that are part of the standard **C** library along with other functions.
- I/O functions are:

– formatted	printf and scanf
– unformatted	character and string functions

printf function (next slide example of printf)

```
#include<stdio.h>
main()
{
int a;
int b;
int c;
a=10;
b=20;
c=a+b;
printf("%d",c);
}
```

- printf() is a function used to print the value of a variable on the screen
- printf("format string", <list of variables>);

"%c",	char
"%d %i",	int
"%f ,%g",	float
" %lf",	double
"%ld"	long signed int
"%lu"	long unsigned int
"%Lf"	long double
"%u"	unsigned int

Formatted Output

- An example:
- printf(“%c\n”, var);
- The conversion characters and their meanings are:

Conversion character	Meaning
d	The data is converted to decimal
c	The data is treated as a character
s	The data is a string, and characters from the string are printed until a null character is reached, or until the specified number of characters have been exhausted
f	The data is output as float or double with a default precision of 6

Formatted Output

- Between the % character and the conversion character, there may be:

A minus sign	Implying left adjustment of data
A digit	Implying the minimum width in which the data is to be output. If the data has larger number of characters than the specified width, the width occupied by the output is larger. If the data consists of fewer characters than the specified width, it is padded to the right (if minus sign is specified), or to the left (if no minus sign is specified). If the digit is prefixed with a zero, the padding is done with zeroes instead of blanks
A period	Separates the width from the next digit.
A digit	Specifying the precision, or the maximum number of characters to be output
l	To signify that the data item is a long integer, and not an integer.

Formatted Output

Format String	Data	Output
%2d	4	| 4|
%2d	224	|224|
%03d	8	|008|
%-2d	4	|4 |
%f	22.44	|22.440000|
“%0.2f”	12.3412	|12.34|
“%10.5f”	12.3412	| 12.34120 |
“%10.5f”	12.3412	| 12.34120 |
“%1.5f”	12.3412	| 12.34120 |
“%2.2”	12.3412	|12.34|
“%5.2”	12.3412	|12.34|
“%6.2”	12.3412	| 12.34|
“%6.6”	12.3412	|12.341200|

Formatted Input

- The function **scanf( )** is used for formatted input, and provides many of the conversion facilities of **printf( ).**

- The **scanf( )** function reads and converts characters from standard input according to the format string, and stores the input in memory locations specified by the other arguments.

scanf function

- scanf(), allows to enter data through keyboard.
- The general form of scanf() is
  - scanf(“format string”, list of addresses of the variables);
  - scanf(“%d%f%d”,&a,&b,&c);
  - supplied values must be separated by blank(s), tab(s), or newline(s)
  - Don’t include the escape sequences in the format string

Formatted Input

- #include<stdio.h>
- main( )
- {
-   char name;
-   int age = 0;
-   char gender = ' ';
-   scanf ("%c %c %d", &name, &gender, &age); ?????????????
-   fflush(stdin);
-   printf("% c %c %d", name, gender, age);
- }

Unformatted I/O functions

- Character input through:
  - int getch(void)   // read the character when typed (without waiting for return key)
  - int getche(void)   // same as getch(), but this function echo the typed char
  - int getchar(void)   // wait for return key and same as getche()
  - int getc(FILE *stream);
  - int fgetc(FILE *stream);

- Character output through:
  - putch(int c);
  - putchar(int c)
  - putc(int c, FILE *stream);
  - fputc(int c, FILE *stream);

Function **fflush( )** clears the buffer

Unformatted I/O functions

- string input through:
  - char *gets(char *s);
  - char *fgets(char *s, int n, FILE *fp);

- string output through:
  - int fputs(const char *s, FILE *fp);
  - int puts(const char *s);

Character-Based I/O: An Example

- #include <stdio.h>
- main( )
- {
-   int ch;
- fflush(stdin);
- ch = getchar( );
- putchar(ch);
-   }

Character-Based I/O: An Example

```
•   #include <stdio.h>
•   main( )
•   {
•     char ch;
•     ch = getchar( );
•     fflush(stdin);
•     putchar(ch);
•     }
```

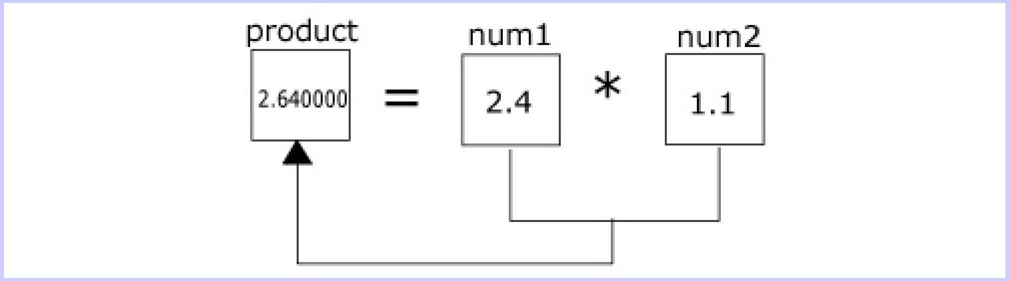
Example

```
#include <stdio.h>
int main()
{
    int num;
     printf("Enter a integer: ");
    scanf("%d",&num);
    printf("You entered: %d",num);
    return 0;
}
```

```
Enter a integer: 25
 You entered: 25
```

C Program to Multiply two Floating Point Numbers

```
#include <stdio.h>
 int main( )
{
float num1, num2, product;
printf("Enter two numbers: ");
scanf("%f %f",&num1,&num2); /* Stores the two floating point numbers
     entered by user in variable num1 and num2 respectively */
product=num1*num2; /* Performs multiplication and stores it */
printf("Product: %f",product);
return 0;
}
Enter two numbers: 2.4  1.1
 Sum: 2.640000
```



C Program to Find ASCII Value of a Character

```
#include <stdio.h>
int main()
{
char c;
printf("Enter a character: ");
scanf("%c",&c); /* Takes a character from user */
 printf("ASCII value of %c = %d",c,c);
return 0;
}
Enter a character: G
ASCII value of G = 71
```

Problems

- C Program to Find Quotient and Remainder of Two Integers Entered by User
- C Program to Swap Two Numbers

C constants

- Integer
- Real number { float, double}
- Character For example: 'a', 'l', 'm', 'F' etc.
- String
- Symbolic #define pi 3.14159
- Logical [true, nonzero value; false, zero value]
- Enumerated enum color {yellow, green, black, white}; {0, 1, 2, etc. by default}

  – The const qualifier is used to tell C that the variable value can not change after initialization.

  – const float pi=3.14159; // pi cannot be changed at a later time // within the program.

Operators

- Arithmetic
- Logical/Relational
- Bitwise
- Assignment
- Odds and ends!

Arithmetic operator

• +	addition or unary plus
• -	subtraction or unary minus
• *	multiplication
• /	division
• %	remainder after division(modulo division)

Binary Operators

- Binary operators as the name suggests, works on two operands.

- The binary operators in C (as in other programming languages) are:
  - The add (**+**) operator
  - The subtract (**-**) operator.
  - The multiply (*****) operator
  - The divide (**/**) operator
  - The modulo (**%**) operator

Binary Operators

- Examples of binary operators:
- int x, y, z;
- x = 27;
- y = x % 5; /* y set to 2 */
- z = x / 5 /* z set to 5 and not 5.4 */

Unary Operators

- Unary operators, as the name suggests, works on one operand or variable.
- The ++ and the -- operators are examples of unary operators.
- When these operators prefix an operand, they are referred to as **prefix operators**, and when they are suffixed to an operand, they are referred to as **postfix operators**.
- Prefix and postfix operators, when used in an expression, can have totally different results, and hence it is necessary to know their workings.

Unary Operators

- Incrementing and decrementing by 1 is such a ubiquitous operation that C offers the prefix and postfix implementations of the ++ and the -- operators.
- The expression i = i + 1; can also be written as **i++**.
- When used as a standalone statement, writing i++, or ++i does not make any difference at all.
- **It is only when they are used as part of an expression that the prefix and the postfix operators can have totally different results.**

Unary Operators

- Consider the following two statements:
- total = i++;
- total = ++i;

- The first statement **total = i++;** is equivalent to:
- total = i; i=i+1;
- This is equivalent to assigning to total the current value of i, and then incrementing i.

Unary Operators

- The second statement total = ++i; is equivalent to:
- i = i + 1;
- total = i;

- This is equivalent to first incrementing the value of i, and then assigning the incremented value of i to total.

- The same principle holds true when working with the unary -- operators.

Assignment Operators

- Compound assignment statements help in simplifying, and writing simple code.
- So far, we have been writing statements such as:
  sum = sum + num
- The same could be simplified as sum += num;
- The same applies to the other binary operators.

Assignment Operators

- sum = sum – num; can be written as sum -= num;
- x = x * y; can be written as x *= y;
- x = x / y; can be written as x /= y;
- Compound assignment operators can be very useful if the identifiers used for the operands is very long.

Assignment Operators

Operator	Example	Same as
=	a=b	a=b
+=	a+=b	a=a+b
-=	a-=b	a=a-b
*=	a*=b	a=a*b
/=	a/=b	a=a/b
%=	a%=b	a=a%b

Type Conversions

- When an operator has operands of different types, they are converted to a common type according to a small number of rules.
- In general, the only automatic conversions are those that convert a “narrower” operand into a “wider” one without losing information, such as converting an integer into floating point

Type Conversions

- Expressions that might lose information, like assigning a longer integer type to a shorter, or a floating-point type to an integer, may draw a warning, but they are not illegal.
- A **char** is just a small integer, so chars may be freely used in arithmetic expressions. This permits considerable flexibility in certain kinds of character transformations.
- Implicit arithmetic conversions work much as expected. In general, if an operator like +, or * that takes two operands (a binary operator) has operands of different types, the “lower” type is *promoted* to the “higher” type before the operation proceeds.

Type Conversion

The following informal set of rules will suffice:

- If either operand is long double, convert the other to long double.
- Otherwise, if either operand is double, convert the other to double.
- Otherwise, if either operand is float, convert the other to float.
- Otherwise, convert char and short to int.
- Then, if either operand is long, convert the other to long.

- **Conversions take place across assignments; the value of the right side is converted to the type of the left, which is the type of the result.**

Type Conversions

- Consider the following assignments:
- int i;
- char c;
- i = c;

- In the aforesaid assignment, the data type on the right (**char**) is converted to the data type on the left (**int**), which is the type of the result.

- If x is float and i is int, then x = i and i = x both cause conversions; float to int causes truncation of any fractional part. When a double is converted to float, whether the value is rounded, or truncated is implementation-dependent.

Explicit Type Conversion

- Finally, explicit type conversions can be forced in any expression, with a unary operator called a **cast**.
- In the construction (*type name*) *expression,* the *expression* is converted to the named type by the conversion rules above.
- The precise meaning of a **cast** is as if the *expression* were assigned to a variable of the specified type, which is then used in place of the whole construction.

Explicit Type Conversion

- Consider the following example:
- int i, j;
- double d;
- d = i / j; /* double assigned the result of the division of two integers */

- The problem with the above assignment is that the fractional portion of the above division is lost, and d is effectively assigned the integer quotient of the two integers i and j.

Explicit Type Conversion

- To resolve this, the variable i can be **typecast** into a double as in: d = (double)i / j;

- int i is converted to a double using the explicit cast operator. Since one of the operands is a double, the other operand (j) is also converted to a double, and the result is a double.

- The result is assigned to a double with no side effects. **The cast operator can be done only on the right-hand side of an assignment.**

Relational Operator

Operator	Meaning of Operator	Example
= =	Equal to	5==3 returns false (0)
>	Greater than	5>3 returns true ($\neq$ 0)
<	Less than	5<3 returns false (0)
!=	Not equal to	5!=3 returns true($\neq$ 0)
>=	Greater than or equal to	5>=3 returns true (1)
<=	Less than or equal to	5<=3 return false ($\neq$ 0)

Logical Operators

Operator	Meaning of Operator	Example
&&	Logical AND	If c=5 and d=2 then, ((c==5) && (d>5)) returns false
||	Logical OR	If c=5 and d=2 then, ((c==5) || (d>5)) returns true.
!	Logical NOT	If c=5 then, !(c==5) returns false

Bitwise Operators

Operators	Meaning of operators
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
~	Bitwise complement
<<	Shift left
>>	Shift right

- Bitwise operators should be used only with unsigned integer operands

Bitwise Operators

```
#include <stdio.h>
main()
{ unsigned int a = 60; /* 60 = 0011 1100 */
 unsigned int b = 13; /* 13 = 0000 1101 */
int c = 0;
c = a & b;    /* 12 = 0000 1100 */
printf("Line 1 - Value of c is %d\n", c );
c = a | b; /* 61 = 0011 1101 */
 printf("Line 2 - Value of c is %d\n", c );
 c = a ^ b; /* 49 = 0011 0001 */
 printf("Line 3 - Value of c is %d\n", c );
 c = ~a;  /*-61 = 1100 0011 */
printf("Line 4 - Value of c is %d\n", c );
c = a << 2; /* 240 = 1111 0000 */
printf("Line 5 - Value of c is %d\n", c );
 c = a >> 2; /* 15 = 0000 1111 */
printf("Line 6 - Value of c is %d\n", c );
}
```

```
Line 1 - Value of c is 12
 Line 2 - Value of c is 61
 Line 3 - Value of c is 49
 Line 4 - Value of c is -61
 Line 5 - Value of c is 240
 Line 6 - Value of c is 15
```

Comma Operator

- int a=1, b=2, c=3, i=0; // commas act as separators
- i = a, b; // stores a into i. Equivalent to (i = a), b; ‘,’ is operator
- i = (a, b); // stores b into i; comma operator executed first
- i = (a += 2, a + b);
  - // increases a by 2, then stores a+b = 3+2 into i
  - // ... a=3, b=2, c=3, i=5
- i = a += 2, a + b;
  - // increases a by 2, then stores a to i, and discards unused a + b rvalue. //Equivalent to (i = (a += 2)), a + b; ... a=5, b=2, c=3, i=5

Comma Operator

- i = a, b, c;
  - // stores a into i, discarding the unused b and c rvalues // ... a=5, b=2, c=3, i=5
- i = (a, b, c);
  - // stores c into i, discarding the unused a and b rvalues ... a=5, b=2, c=3, i=3
- return a=4, b=5, c=6;
  - // returns 6, not 4, since comma operator sequence points following the keyword 'return' are considered a single expression evaluating to rvalue of final subexpression c=6
- return 1, 2, 3; // returns 3, not 1, for same reason as previous example
- return(1), 2, 3; // returns 3, not 1, still for same reason as above.

sizeof () operator

- The sizeof() operator returns the size of its operand in bytes

```
#include <stdio.h>
int main()
{
int a; float b; double c; char d;
 printf("Size of int: %d bytes\n",sizeof(a));
printf("Size of float: %d bytes\n",sizeof(b));
 printf("Size of double: %d bytes\n",sizeof(c));
 printf("Size of char: %d byte\n",sizeof(d));
return 0;
}
```

```
Size of int: 4 bytes
Size of float: 4 bytes
Size of double: 8 bytes
Size of char: 1 byte
```

C Program to Find Size of int, float, double and char of Your System

Precedence & Associativity of Operators

- 2+20* 5 = (2+20)*5 or 2+(20*5) ?
- 2-20-5= ? (2-20)-5 or 2-(20-5)   ?
- 20/2/10   ((20/2)/10) or (20/(2/10)) ?

- If more than one operators are involved in an expression then, C language has predefined rule of priority of operators. This rule of priority of operators is called operator precedence.

- Associativity indicates in which order two operators of same precedence (priority) executes.

Order of Precedence

From high priority to low priority the order for all C operators is:

• **( ), [ ], -> , .**	**(Association left->right)**
• **!, ~, unary +/-, sizeof () ++ - -, &, * (pointer)**	**(Association right->left)**
• *** / %**	**(Association left->right)**
• **+ -**	**(Association left->right)**
• **<<, >>**	**(Association left->right)**
• **<, <=, >=, >**	**(Association left->right)**
• **= = !=**	**(Association left->right)**
• **&**	**(Association left->right)**
• **^**	**(Association left->right)**
• **|**	**(Association left->right)**
• **&&**	**(Association left->right)**
• **||**	**(Association left->right)**
• **?:**	**(Association right->left)**
• **= += -=**	**(Association right->left)**
• **, (comma)**	**(Association right->left)**

C statements

- Four types of instructions in C:
  - Type declaration statement
  - Input/output statement
  - Arithmetic statement
  - Control statement

Control statements

- Enable us to specify the order in which the instructions of a program are to be executed
  - Determine the **flow of control** in a program
- if [else] statements
- switch-case statements
- Loop statements {for, while, do-while}

If statements

- if (the condition is true) // non-zero value
  - then execute the statement
- Example

10% discount offered if the purchased quantity is more than 1000. Quantity and price per item given as input. Write a program to calculate the total expenses.

```
main()
{
    int a, b;
    scanf(“%d%d”,&a,&b);
    if(a<b) printf(“second number is larger \n”);
    if(a>b) printf((“first number is larger \n”);
    if(a==b) printf(“both are equal\n”);
}
```

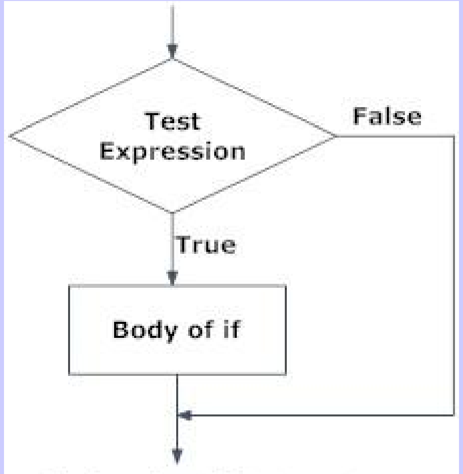


Fig: Operation of if statement

Example

- 10% discount offered if the purchased quantity is more than 1000. Quantity and price per item given as input. Write a program to calculate the total expenses.

```
main()
{
    int qty;
    float cost, rate;
    scanf("%d%f",&qty,&rate);
    cost=rate*qty;
    if(qty>1000) cost*=0.9;
    printf("total cost=%f\n",cost);
}
```

Multiple statements

```
if (yr_of_service>5)
{
    bonus=3000;
    others=500;
    total = salary+bonus+others;
    printf("%d",total);
}
```

whenever a group of statements they are grouped by `{‘ and `}’ pair; called ***compound statement*** or ***block***

if-else statement

```
if(expression)
      statement;
else
      statement;

#include <stdio.h>
main( )
 {
   int number;
    scanf("%d", &number);
 if (number%2 ==0)
     puts("it is even");
else printf("it is odd");
}
```

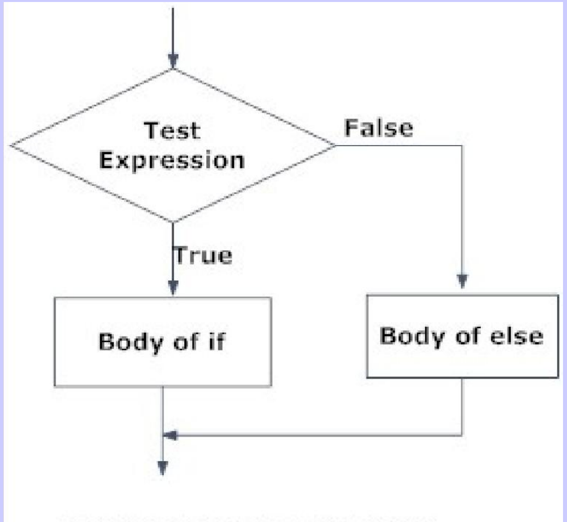


Fig: Operation of if...else statement

if-else statement

```
#include <stdio.h>
main( )
 {
   int digit;
    scanf(“%d”, &digit);
 if (digit ==0)
     puts("You entered the number 0");
 if (digit ==1)
     puts("You entered the number 1");
 if (digit ==2)
     puts("You entered the number 2");
}
```

```
#include <stdio.h>
main( )
 {
   int digit;
    scanf(“%d”, &digit);
 if (digit ==0)
     puts("You entered the number 0");
 else if (digit ==1)
     puts("You entered the number 1");
 else if (digit ==2)
     puts("You entered the number 2");
}
```

if-else-if statement

```
if (number>0)
    if( number%5==0)
          printf("do the operation1 \n");
    else printf("do the operation2 \n");
```

use braces to force proper association

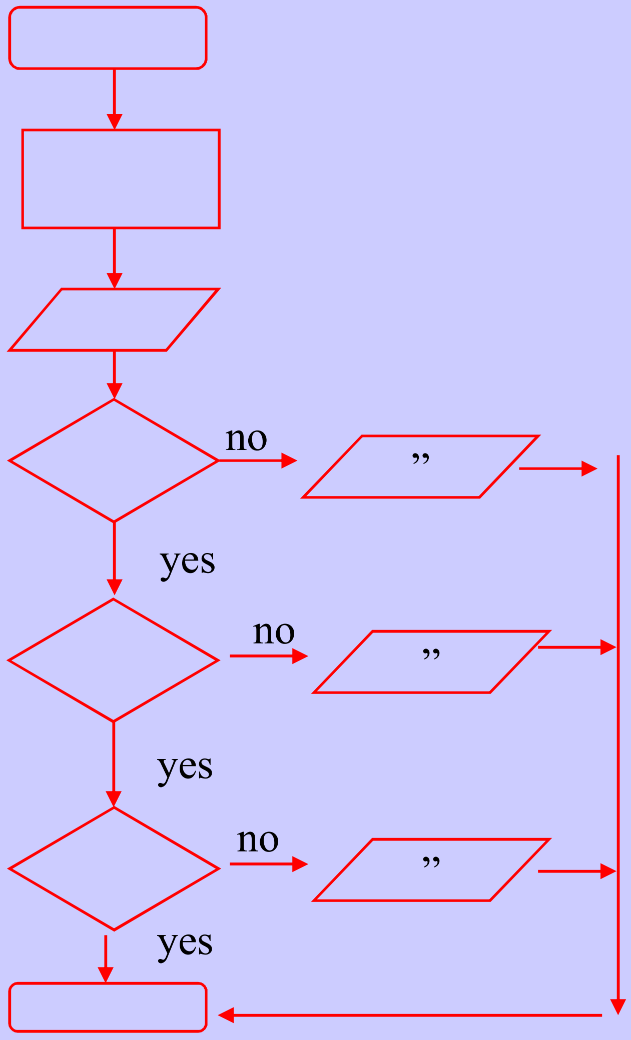
```
if (total>=60)
    printf("1st div\n");
if(total>=50 && total<60)
    printf("2nd div\n");
if(total>=40 && total<50)
    printf("3rd div\n");
if(total<40)
    printf("fails\n");
```

Cascading if-else Construct

```
• #include <stdio.h>
• main( )
•  {
•    int chr;
•     chr = getchar( );
•     if (chr == '0')
•      puts("You entered the number 0");
•     else if (chr == '1')
•      puts("You entered the number 1");
•    else if (chr == '2')
      puts("You entered the number 2");
```

Cascading if-else Construct

- else if (chr == '3')
- puts("You entered the number 3");
- else if (chr == '4')
- puts("You entered the number 4");
- else if (chr == '5')
- puts("You entered the number 5");
- else if (chr == '6')
- puts("You entered the number 6");
- else if (chr == '7')
- puts("You entered the number 7");
- else if (chr == '8')
- puts("You entered the number 8");
- else if (chr == '9')
- puts("You entered the number 9");
- else
- puts("You did not enter a number");
- }



Nested if Construct: Example

- The following program determines whether the character entered is an uppercase or a lowercase alphabet.

```
#include<stdio.h>
main( )
{
  char ch;
  ch = getchar( );
  fflush(stdin);
  if (ch >= 'A')
          if (ch <= 'Z')
                    puts ("Its an Uppercase alphabet");
          else if (ch >= 'a')
                    if (ch <= 'z')
     puts ("It’s a lowercase alphabet");
          else
             puts ("Input character > z");
       else
             puts ("Input character greater than Z but less than a");
     else
          puts ("Input character less than A");
}
```

Using Braces to Improve Readability

- #include< stdio.h>
- main( )
- {
-   char ch;
-   ch = getchar( );
-   fflush(stdin);
-   if (ch >= ‘A’)
-    {
-     if (ch <= ‘Z’)
-       puts (Its an Uppercase alphabet”);
-     else if (ch >= ‘a’)
-             {
-               if (inp <= ‘z’)
-                 puts (“It’s a lowercase alphabet”);
-               else
-                puts (“Input character > z”);
-             }
- else
-        puts (“Input character greater than Z but less than a”);
-     }
-   else
-        puts (“Input character less than A”);
- }

Conditional operators (Ternary operator)

```
 if (a>b) max=a;
    else max=b;
```

```
max=(a>b) ? a : b;
```

The general format

```
    expression1 ? expression2 : expression3;
```

This is ternary operator

Right to left associative

```
max= (a>b ? (a>c ? a : c) : (b > c ? b : c));
```

Conditional Operator

- expression1?expression2:expression3
  - If the test condition expression1 is true, expression2 is returned and if false expression3 is returned.
- Conditional operator takes three operands and consists of two symbols ? and : .
- Conditional operators are used for decision making in C
- c=(c>0)?10:-10;
  - If c is greater than 0, value of c will be 10 but, if c is less than 0, value of c will be -10.

The switch-case statements

- The switch-case conditional construct is a more structured way of testing for multiple conditions (cascading, or multiple if statement).
- A switch-case statement makes for better readability, and hence makes code better understandable.

```
switch(expression)
{
case const-expr: statement;
case const-expr: statement;
default: statements;
}
```

Example

```
main( )
 {   int chr;  chr = getchar( );
     switch (chr)
            {           case ‘A’:
                        case ‘E’:
                        case ‘I’:
                        case ‘O’:
                        case ‘U’: number_of_vowels++;
                                   break;
                        case ‘ ’ :  number_of_spaces++;
                                   break;
                        default :  number_of_consonants++;
              }
     }
```

Example

- main( )
- {   int chr; chr = getchar( );
- switch (chr)
- {case '0' : puts("you entered 0");   break;
- case '1'   : puts("you entered 1");   break;
- case '2'   : puts("you entered 2");   break;
- case '3'   : puts("you entered 3");   break;
- case '4'   : puts("you entered 4");   break;
- case '5'   : puts("you entered 5");   break;
- case '6'   : puts("you entered 6");   break;
- case '7'   : puts("you entered 7");   break;
- case '8'   : puts("you entered 8");   break;
- case '9'   : puts("you entered 9");   break;
- default   : puts ("You did not enter a number");

  }

}

The switch-case Conditional Construct

- The **case** and **default** statements can occur in any order.
- The **default** statement is optional, and is used for handling situations where none of the **case** statements are evaluated as true.
- After the action for a particular **case** value is specified, the **break** statement is used to bring control out of the **switch** block
- If the **break** statement is not specified after the statements associated with each **case** statement, then all the statements associated with the other **case** statements will be executed.

Loop in C

- Like to print all the numbers between 1 and 10
- main()
- {
- int count=1;
- printf("%d\n", count++);
- printf("%d\n", count++);
- printf("%d\n", count++);
- printf("%d\n", count++);
- printf("%d\n", count++);
- printf("%d\n", count++);
- printf("%d\n", count++);
- printf("%d\n", count++);
- printf("%d\n", count++);
- printf("%d\n", count++);
- }

If 1000?

Loops in C (for loop)

- main()
- { int count;
- for (count=1 ; count <= 10 ; count++)
  printf("%d\n", count);
- }
- count=1 is the initialization.
- count <= 10 expression. The for statement continues to loop while this statement remains true
- count++ increment or decrement.
- printf("%d\n", count) the statement to execute.

For loop

- Repeating several lines of code

```
main()
 {
   int count, sqr;

   for ( count=1 ; count <= 10 ; count++)
   {
     sqr=count * count;
     printf( " The square of");
     printf( " %2d", count);
     printf( " is %3d\n", sqr);
   }
 }
```

More details

- for (expression_1 ; expression_2 ; expression_3)

  statement ;

  or

  {block of statements}

- 1.Executes expression_1.
- 2.Evaluates expression_2, if false goto 6
- 3.Executes statement.
- 4.Executes expression_3.
- 5. goto 2
- 6. End of loop

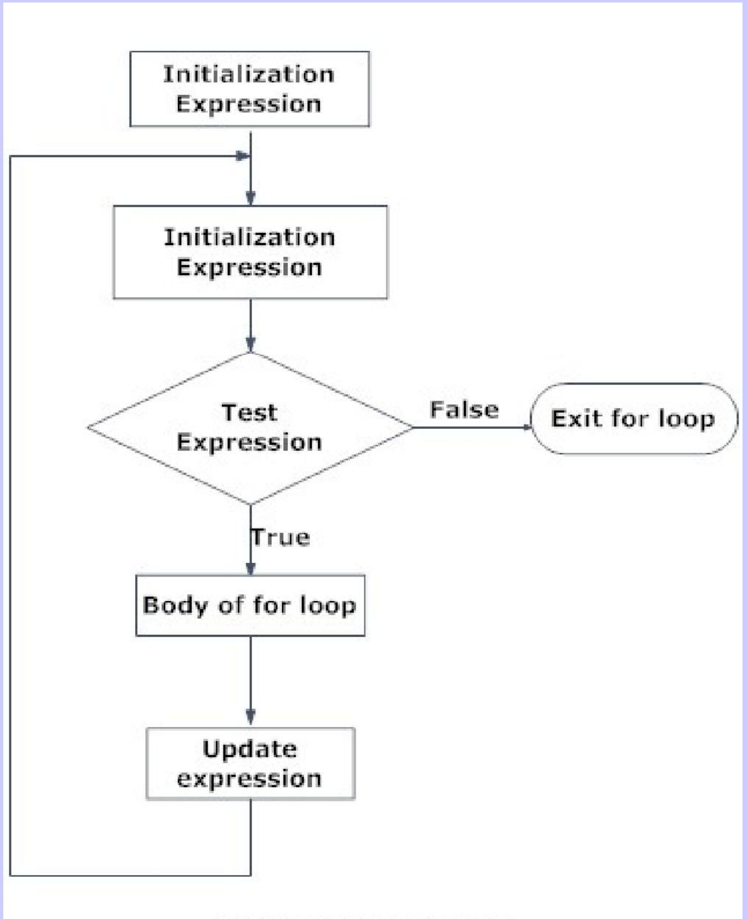


Figure: Flowchart of for loop

For loop (Contd…)

- Any of the three expressions can be missing, if the first or third is missing, it is ignored. If the second is missing, it is assumed to be TRUE.
- for (x=0 ; ((x>3) && (x<9)); x++)
- for (; ((x>3) && (x<9)); x++)
- for (; ((x>3) && (x<9));)
- for (x=0, y=4; ((x>3) && (y<9)); x++, y+=2)
- for (x=0, y=4, z=4000;   z ;   z/=10)
- for (; ;);
- for(; ;) puts(" Linux rules!");

C Program to Find Factorial of a Number

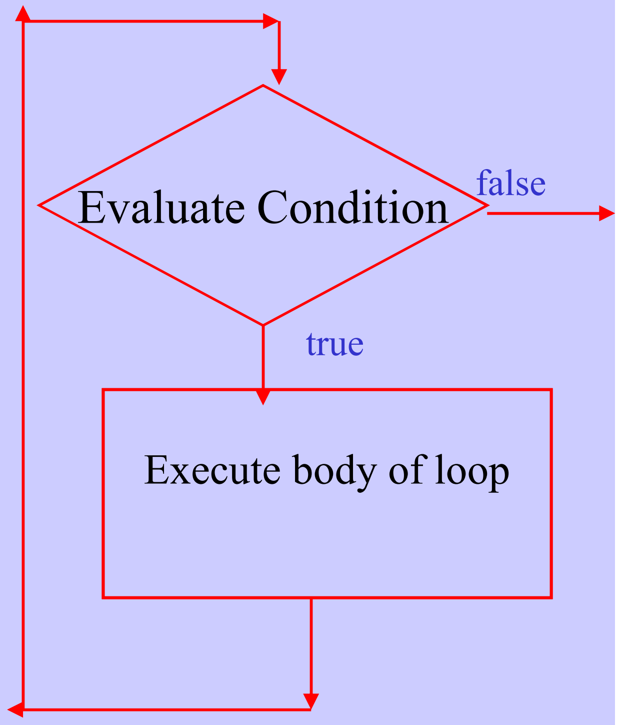
```
#include <stdio.h>
int main()
{
int n, count;
unsigned long  int factorial=1;
printf("Enter an integer: "); scanf("%d",&n);
if ( n< 0) printf("Error!!! Factorial of
   negative number doesn't exist.");
 else
     {
     for(count=1;count<=n;++count) {
    factorial*=count;
         }
      printf("Factorial = %lu",factorial);
    }
return 0;
}
```

- Enter an integer: -5
- Error!!! Factorial of negative number doesn't exist.

- Enter an integer: 10
  Factorial = 3628800

Iterative Constructs - The while Loop

- In a **while** loop, the loop condition is written at the top followed by the body of the loop.
- Therefore, the loop condition is evaluated first, and if it is true, the loop body is executed.
- After the execution of the loop body, the condition in the **while** statement is evaluated again. This repeats until the condition becomes false.

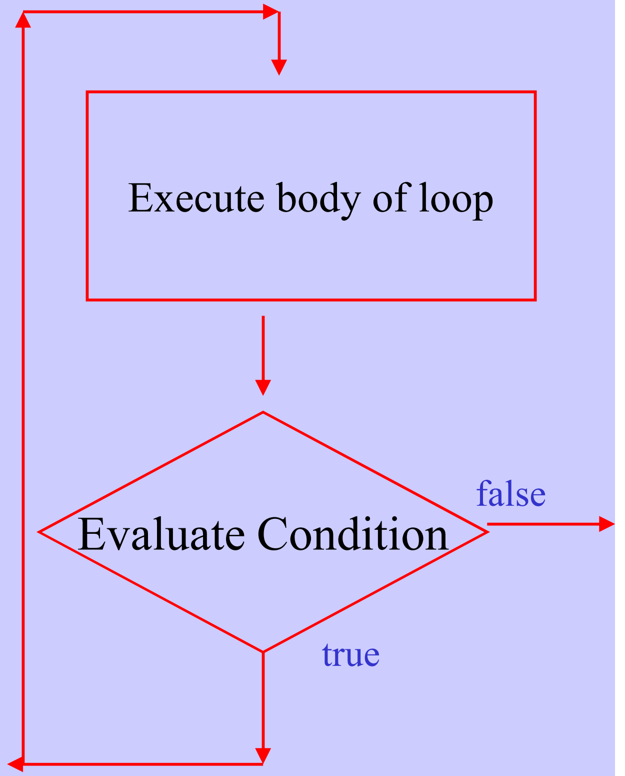


Iterative Constructs - The while Loop

- #include <stdio.h>
- /* function to accept a string and display it 10 times */
- main( )
- {
-   int count=1,sqr;
- while (counter <= 10)
-    {
-               sqr=count * count;
-               printf(" The square of");
-               printf(" %2d", count);
-               printf(" is %3d\n", sqr);
              count++;
-    }
- }

Iterative Constructs – The do…while loop

- In this loop construct, the body of the loop comes first followed by the loop condition at the end.
- Therefore, when this loop construct is used, the body of the loop is guaranteed to execute at least once.
- The loop is entered into straightaway, and after the first execution of the loop body, the loop condition is evaluated.
- Subsequent executions of the loop body would be subject to the loop condition evaluating to true.



Iterative Constructs – The do…while loop

```
/* This is an example of a do-while loop */
#include <stdio.h>
main()
{
 int i;
 i = 0;
 do {
        printf("The value of i is now %d\n",i);
        i = i + 1;
     } while (i < 5);
}
```

do-while example

```
•   #include <stdio.h>
•   /* function to accept a string and display it 10 times */
•   main( )
•   {
•     int count=1,sqr;
•   do  {
•                 sqr=count * count;
•                 printf( " The square of");
•                 printf( " %2d", count);
•                 printf( " is %3d\n", sqr);
                 count++;
•      } while (counter <=10);
•   }
```

Nested loop

- main()
- {
- int i, j, sum;

  for(i=1;i<=3;i++)

    for(j=1;j<=2;j++)

      printf("i=%d+\tj=%d\t=%d\n",i,j,i+j);
- }

Control of Loop Execution

- A loop construct, whether while, or do-while, or a for loop continues to iteratively execute until the loop condition evaluates to false.
- But there may be situations where it may be necessary to exit from a loop even before the loop condition is reevaluated after an iteration.
- The **break** statement is used to exit early from all loop constructs (while, do-while, and for).
- **break** causes **exit** from **innermost loop or switch** statement

Control of Loop Execution

- The **continue** statement used in a loop causes all subsequent instructions in the loop body (coming after the continue statement) to be skipped.
- Control passes back to the top of the loop
- In case of a **continue** statement in a **for** loop construct, control passes to the reinitialization part of the loop, after which the loop condition is evaluated again.

Control of Loop Execution

```
main()
{
int n,   i=2;
scanf(“%d”,&n);
if(n>1)
    while(1)
    {
     if(n%i==0) { printf(“composite number”);break;}
     i++;
    }
}
```

Control of Loop Execution

```
main()
{
int n,      i=2;
scanf("%d",&n);
if(n>1)
     while(n)
     {
      if(n%i==0) { printf("composite number");break;}
      if(n%2==0) {n/=2; continue;}
      else n--;
      i++;
     }
}
```

goto statement

- main()
- {
- int n;
- scanf(“%d”,&n);
- if (n<0) goto noop;
- n=n*n;
- printf(“%d\n”,n);
- /* exit(0); or goto stop*/
- noop:
-         printf(“number is negative\n”);
- }

goto statement

- main()
- {
- int n,m,k,i,j;
- int temp,sum=0;
- scanf(“%d%d%d”,&n,&m,&k);
- for(i=0;i<100;i++)
- for(j=0;j<50;j++)
- {

  temp=n+m+k;

  If(temp==0) goto out;

  sum+=temp*temp;
- }
- out:
-         printf(“out of loop && sum=%d\n”,sums);
- }

goto statement

- Use of goto
  - Programs become unreliable, unreadable, and hard to debug
  - It is almost always possible to write a code without goto
  - Avoid goto

Exercise

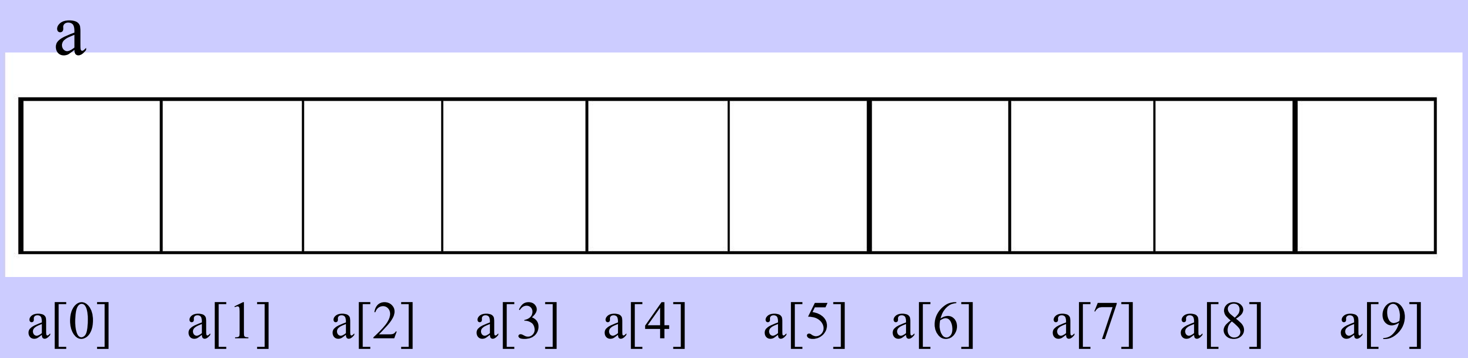
- Check Whether a Character is Vowel or consonant
- C Program to Find the Largest Number Among Three Numbers Entered by User
- C program to Find all Roots of a Quadratic equation
- C Program to Check Whether the Entered Year is Leap Year or not
- C Program to Check Whether a Number is Positive or Negative or Zero.
- C Program to Checker Whether a Character is an Alphabet or not
- C Program to Find Sum of Natural Numbers
- C Program to Find Factorial of a Number
- C Program to Display Fibonacci Series
- C Program to Find HCF of two Numbers
- C Program to Find LCM of two numbers entered by user
- C Program to Count Number of Digits of an Integer
- C Program to Reverse a Number
- C program to Calculate the Power of a Number
- C Program to Check Whether a Number is Palindrome or Not
- C Program to Check Whether a Number is Prime or Not
- C Program to Display Prime Numbers Between Two Intervals
- C program to Check Armstrong Number
- C Program to Display Armstrong Number Between Two Intervals
- C program to Display Factors of a Number

Chapter 2
Arrays

One-Dimensional Arrays

- Array is one of the simplest data structure in computer programming.
- An array can be defined as a collection of elements of identically typed data items that are stored contiguously in memory.
- Each array element shares the same name as the array name, but distinguishes itself from other elements in the array using a subscript, or an index.
- The subscript, or the index of each array element is determined based on the number of offset positions it is from the starting position. The starting offset is taken as 0.

Array Representation

- The declaration int a[10]; defines an array a of size 10, as a block of 10 contiguous elements in memory.
- 

Program using array

- Main()
- {
- int I,a[10];
- for(I=0;I<10;I++)

  scanf("%d",&a[I]);
- for(I=0;I<10;I++)

  printf("%d ",a[I]);

  }

- Main()

{
- int I,a[10];
- for(I=0;I<10;I++)

  scanf("%d",&a[I]);
- for(I=0;I<10;I++)

  {

  Temp=a[I];

  Temp=Temp*Temp;

  a[I]=Temp;

  printf("%d ",a[I]);

  }

}

Array Initialization

- Since an array is a set of elements located at contiguous memory locations, its initialization involves moving element by element, or one data at a time into the array.
- An array is initialized by specifying each element in an initialization list separated by commas.
- The size of the array, in this case, may or may not be specified. The number of elements stored in the array determines its size.

Array Initialization Syntax

#include<stdio.h>

- main( )
- {
- char array1[5] = {‘A’, ‘R’, ‘R’, ‘A’, ‘Y’};
- char dayofweek[ ] = {‘M’, ‘T’, ‘W’, ‘T’, ‘F’, ‘S’, ‘S’, ‘\0’};
- float values[ ] = {100.56, 200.33, 220.44, 400.22, 0};
- int a[10]={1,2,3,4,5}; ???????
- }

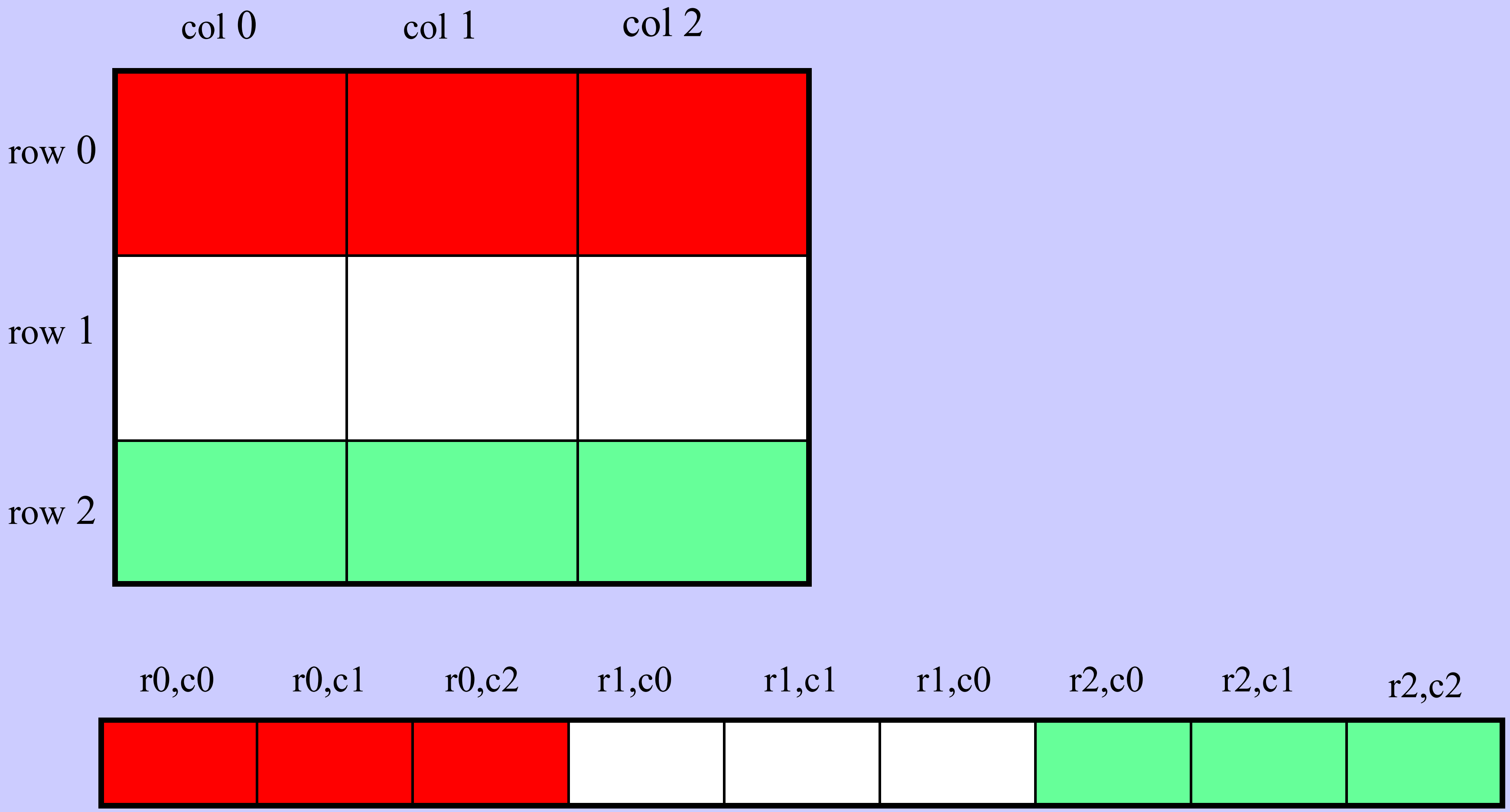
Array Initialization Using a for Loop

```
#include<stdio.h>
main( )
{
  int i, num[50];
  for (i = 0; i < 50; i = i + 1)
     {
       num[i] = i + 1;
       printf("%d\n",i);
      }
 }
```

Two-Dimensional Arrays

- While a one-dimensional array can be visualized in one-dimension as either a row of elements, or a column of elements, a two-dimensional array needs to be visualized along two dimensions, i.e., along rows and columns.
- To be precise, a two-dimensional array can be represented as an array of **m rows** by **n columns**.
- A general way of representing or visualizing a two-dimensional array is in the form of a two-dimensional grid. But, in memory, even a two-dimensional array is arranged contiguously as an array of elements.

Two-dimensional Arrays



Declaring a Two-Dimensional Array

- Declaring a two-dimensional array involves two indices, or two subscripts. There will be an extra set of square brackets[ ] to indicate the second subscript, or the second index.

- int rp_array[3][3];

- /* this array would have nine elements starting at rp_array[0][0], rp_array[1][1]…….and going on till rp_array[2][2] */

Initializing Two-Dimensional Arrays

- int rp_array[3][3] = {0,0, 0,
- 0, 0, 0,
- 0, 0, 0
- };

- To improve the legibility of the initialization, it can be written as: int rp_array[3][3] = { {0,0, 0},
- {0, 0, 0},
- {0, 0, 0}
- };

Initializing Two-Dimensional Arrays Using the for Loop

- /* r_counter is for referring to the rth region */
- /* p_counter is for referring to the pth product */
- for (r_counter = 0; r_counter < 3; r_counter ++)
- {
-    for (p_counter = 0; p_counter < 3; p_counter ++)
-     {
-       rp_array[r_counter][p_counter] = 0;
-      }
-   }

Input to Two-Dimensional Arrays

- for (r_counter = 0; r_counter < 3; r_counter ++)
- {
-   for (p_counter = 0; p_counter < 3; p_counter ++)
-    {
-      scanf(“%d”, &rp_array[r_counter][p_counter]);
-    }
- }

Processing Two-Dimensional Arrays

```
• /* Determine total  sales using the for loop */
• for (r_counter = 0; r_counter < 3; r_counter ++)
•  {
•    for (p_counter = 0; p_counter < 3; p_counter ++)
•     {
•       total_sales += rp_array[r_counter][p_counter];
•     }
•  }
```

Processing Two-Dimensional Arrays

- /* Determine percentage of individual sales data against total sales */
- for (r_counter = 0; r_counter < 3; r_counter ++)
- {
-    for (p_counter = 0; p_counter < 3; p_counter ++)
-     {
-       rp_array_perc[r_counter][p_counter] = ((float) 100 *
-       rp_array[r_counter][p_counter]) / total_sales ;
-     }
-   }
- }

Problems

1. Given a list of numbers
   1. Find the largest one
   2. Find the smallest one
   3. Find the average
   4. Find the frequency of occurrence
   5. Arrange the numbers in ascending order
   6. Arrange the numbers in descending order
   7. Decide whether a number ‘k’ is present or not
2. Write a C program to add two matrices
3. Write a C program to multiply two matrices
4. Write a C program to check whether the matrix is symmetric or not.
5. Write a C program to do 1.1, 1.2, 1.3 and 1.4 for a given 2D array.

Chapter 4
Pointers

Pointer – Definition

- When a variable is declared (such as int i = 22) in a program, space is reserved in memory for the variable i.
- To be precise, each variable is assigned a particular memory location referenced by its address.
- In our case, the integer i would be stored at a specific memory location having the address, say, 1000.

Pointer – Definition

- The declaration of int i = 22 can be conceptualized as follows:

i variable name

$\boxed{22}$ value

1000 address

i is a named location in memory that can hold an integer and having the address 1000

Pointer – Definition

- In C, it is possible to manipulate a variable either by its name, or by its address. The address of a variable can be stored in another variable (**called a pointer variable)**
- **A pointer can therefore be defined as a variable that holds the address of another variable.**
- If you like to define a pointer to hold the address of an integer variable, then you must define the pointer as a pointer to an integer.

Pointer – Declaration & Initialization

- **It is in this sense that a pointer is referred to as a derived data type.**
- **The type of the pointer is based on the type of the data type it is pointing to.**
- For the earlier declaration of the integer variable i, its pointer can be declared as follows:
- int i, *pointer_to_an_integer;
- i = 22;
- pointer_to_an_integer = &i; /* initializing the integer pointer variable (pointer_to_an_integer) with the address of the integer variable i. */

Pointer – Declaration & Initialization

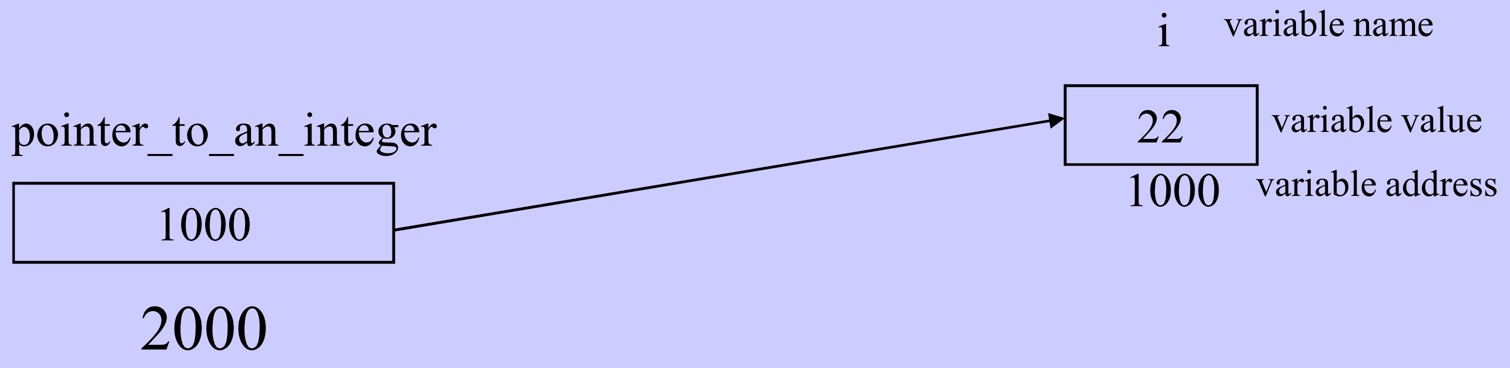
- In the preceding declaration, int *pointer_to_an_integer; the ‘*’ operator preceding the variable name marks it out as a pointer declaration, and the data type of the pointer variable begins the pointer declaration.

- It is obvious from the preceding code that the ‘&’ operator is used to return the address of a variable. The returned address is stored into a pointer variable of the appropriate type.

- **It is critical, as a good C programmer, to initialize a pointer as soon as it is declared.**

Pointer – Declaration & Initialization

- the integer pointer variable (pointer_to_an_integer) being initialized with the address of the integer variable i.
- pointer_to_an_integer = &i;



Dereferencing a Pointer

- **Returning the value pointed to by a pointer is known as pointer dereferencing. In this purpose the * operator is used.**

```
• #include<stdio.h>
• main( )
• {
•   int x, y, *pointer;
•   x = 22;
•   pointer = &x;
•   y = *pointer; /* obtain whatever pointer is pointing to */
• }
```

Common errors in pointer

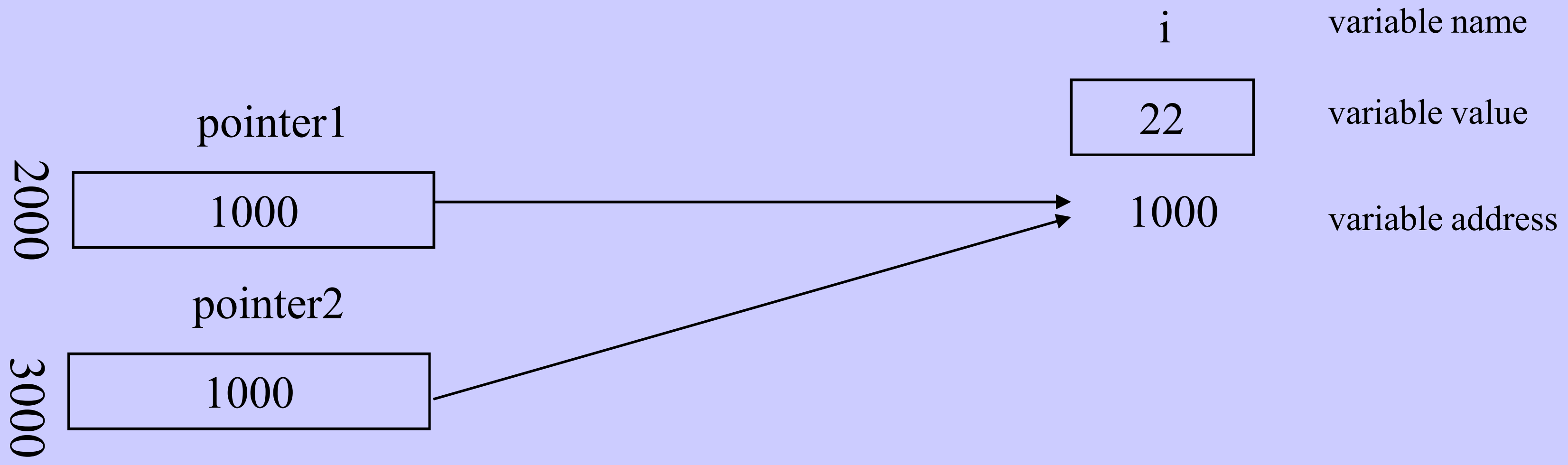
- Suppose, the programmar want pointer pc to point to the address of c. Then,
- int c, *pc;
- pc=c;
- /* pc is address whereas, c is not an address. */
- *pc=&c;
- /* &c is address whereas, *pc is not an address. */

Pointers - Variations on a Theme

- pointer = &x; /* make pointer point to x */
- y = *ptr + 1; /* return the value of the variable pointed to by pointer, add 1 to it, and store the result in y */
- *ptr = 0; /* set the value of the variable pointed to by pointer to 0. In this case, the variable x is set to 0 through its pointer */

Pointers - Variations on a Theme

- Consider the following code snippet:
- int x, y, *pointer1, *pointer2;
- pointer1 = &x /* return the address of x into pointer1 */
- pointer2 = pointer1; /* assign the address in pointer1 to pointer2. Hence, both the pointer variables, pointer1 and pointer2, point to the variable x. */



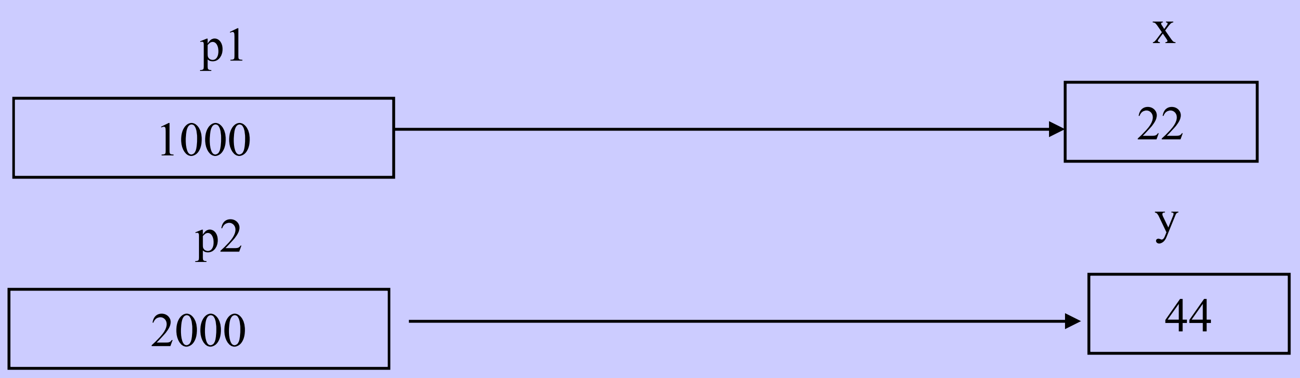
Pointers - Variations on a Theme

- Consider the following code snippet:
- int x, y, *p1, *p2;
- x = 22;
- y = 44;
- p1 = &x; /* p1 points to x */
- p2 = &y /* p2 points to y */
- *p1 = *p2 /* make whatever p1 was pointing to (variable x and hence the value 22) equivalent to whatever p2 is pointing to. */

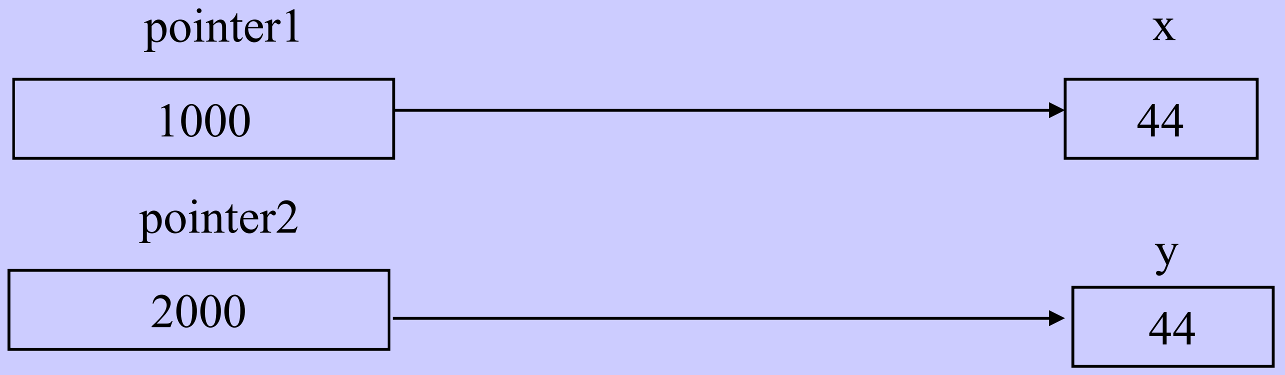
- Hence, both x and y now have the value 44.

Pointers - Variations on a Theme

Before *p1 = *p2



After *p1 = *p2



Printing Using Pointers

- Consider the following code snippet:

```
#include <stdio.h>
main( )
{
  int x, *p;
  x = 26;
  p = &x;
  printf("The value of x is %d\n", x);
  printf("The value of x is %d\n", *p);
  printf("The value of x is %d\n", *(&x));
  printf(“address of x=%u\n", p);
  printf(“address of x=%u\n", &x);
 }
```

Example

- main()
- {
- int j=54;
- float b=3.24;
- char *jj, *bb;
- jj=&j; bb=&b;
- printf(“address contained in jj=%u and that of j=%u\n”,jj,&j);
- printf(“address contained in bb=%u and that of b=%u\n”,bb,&b);
- printf(“value at the address contained in jj=%d %c\n”,*jj, *jj);
- printf(“value at the address contained in bb=%f %c\n”,*bb,*bb);
- jj=jj+1;
- printf(“value at the address contained in jj=%d %c\n”,*jj, *jj);
- }

Example

```
main()
{
      int * ptr , i=5;
      ptr= &i;
      printf("%u %u %u\n",&i, ptr, ptr+sizeof(int));
      ++*ptr;
     printf("value of i=%d\n",*ptr);
      (*ptr)++;
      printf("value of i=%d\n",*ptr);
      ptr ++;
      printf("%u\n",ptr);
     printf("value of %u  %d  %u  %d\n",&i, i, ptr,*ptr);
}
```

Pointer to pointer

- Pointer is a derived data type.
- We can define a pointer, which can store address of this derived data type.
- Example
- main()
- {
- int **ptr, *p, j=3;
- p=&j; ptr=&p;
- printf("address of j:%u %u %u\n",&j, p, *ptr);
- printf("address of p=%u %u\n",&p, ptr);
- printf("address of ptr=%u\n",&ptr);
- printf("value of j=%d %d %d\n",j,*p,**ptr);
- }

Pointers and array

- The address of the 1'st element of an array is represented by its name
- Example

int a[5] ;

a => &a[0]

a+1 => &a[0] + 1

(a+i) ; i varies from 0 to 4 ; traverses through the address of each element in the array 'a'

a[i]

a+i (i=0) => &a[0]
a+i (i=1) => &a[1]
a+i (i=2) => &a[2]
a+i (i=3) => &a[3]
a+i (i=4) => &a[4]

*(a+i) this returns ith value

*(i+a) same as above

i[a]

Example

- main()
- {
- int a[10]={0,1,2,3,4,5,6,7,8,9};
- int *p,i;
- p=a;
- for(i=0;i<10;i++,p++)
- printf("%d %d %d %d %d\n",a[i],i[a],*(a+i),*(i+a),*p);
- }

Pointer Arithmetic

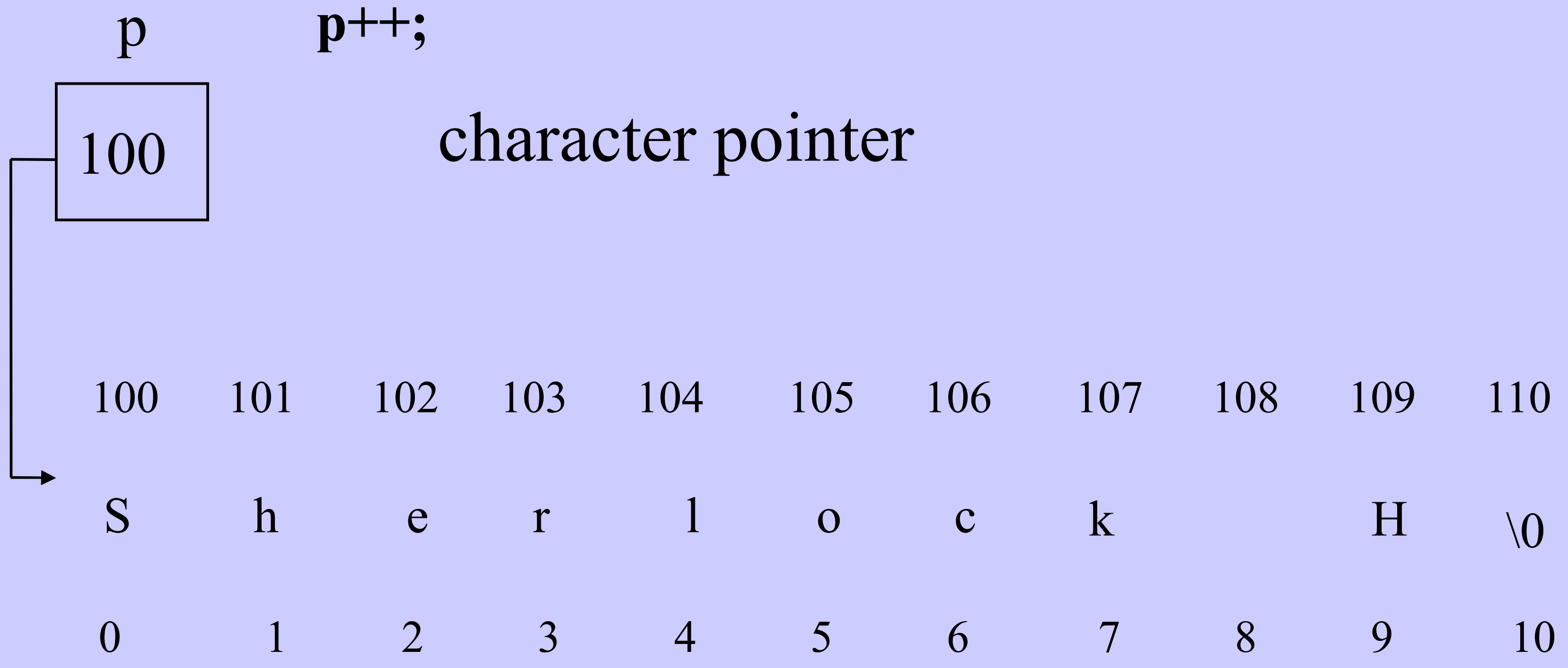
- Addition/subtraction of a number to a pointer
  - ptr++; ptr=ptr+4; ptr=ptr-5;

- The followings operations are not possible
  - Addition of two pointers
  - Multiplication of a pointer by a number
  - Dividing a pointer by a number

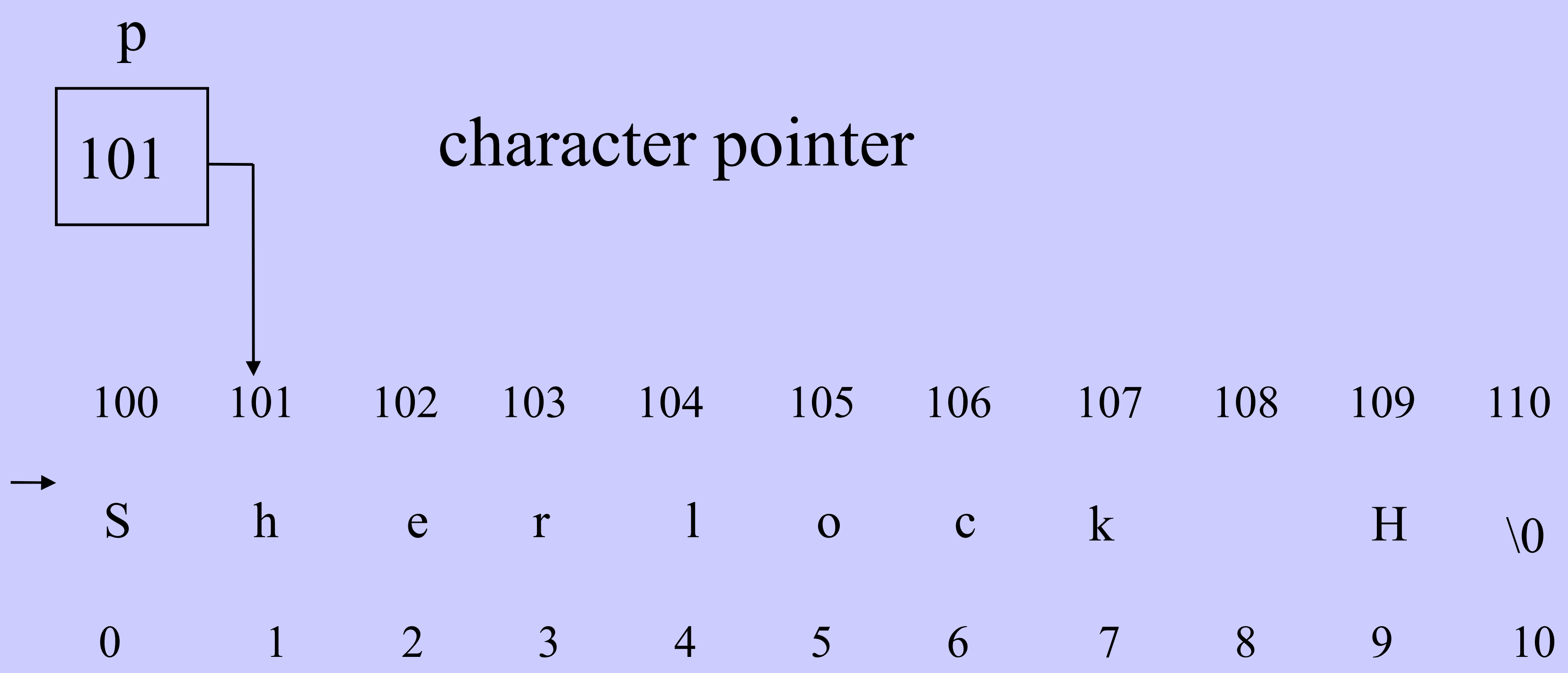
Pointer Arithmetic

p++;

character pointer



Pointer Arithmetic



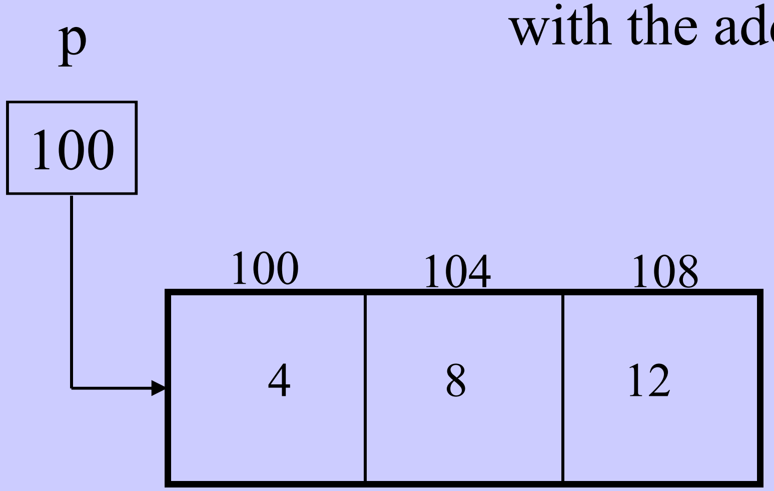
Pointer Arithmetic

- But will the statement p++; always make **p** point to the next memory location.
- The answer is an emphatic **No**.
- **This depends upon the data type that p is pointing to.**
- Consider the following piece of code:
- #include <stdio.h>
- main( )
- {

Pointer Arithmetic

- int int_array[3] = {4, 8, 22}; /* int_array is a constant pointer */
- int * p;
- p = int_array;
- printf (“%d”, p);
- p++;
- printf(“%d”, p);
- }

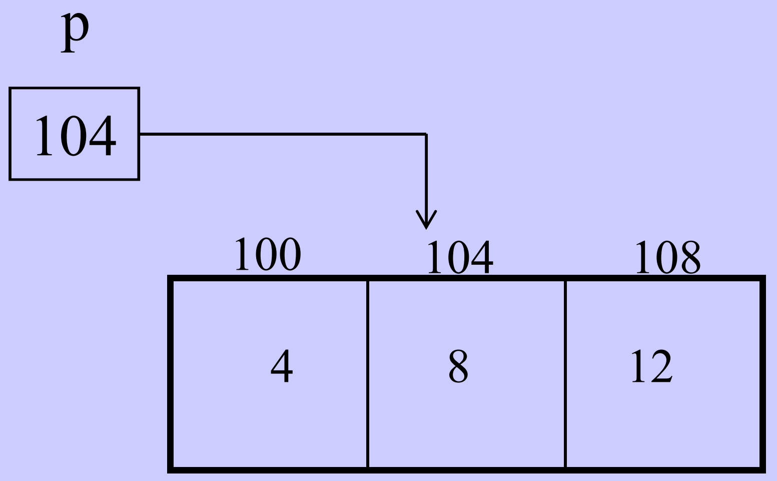
The value of p afer being initialized with the address of int_array



integer pointer

Pointer Arithmetic

The value of p after pointer arithmetic performed on p, i.e., p++



integer pointer

Pointer Arithmetic

- The key point to note is that when pointers are incremented by 1, the size of the data type to which it is pointing to (4 bytes in our case, since an integer needs 4 bytes of storage) is taken into account.

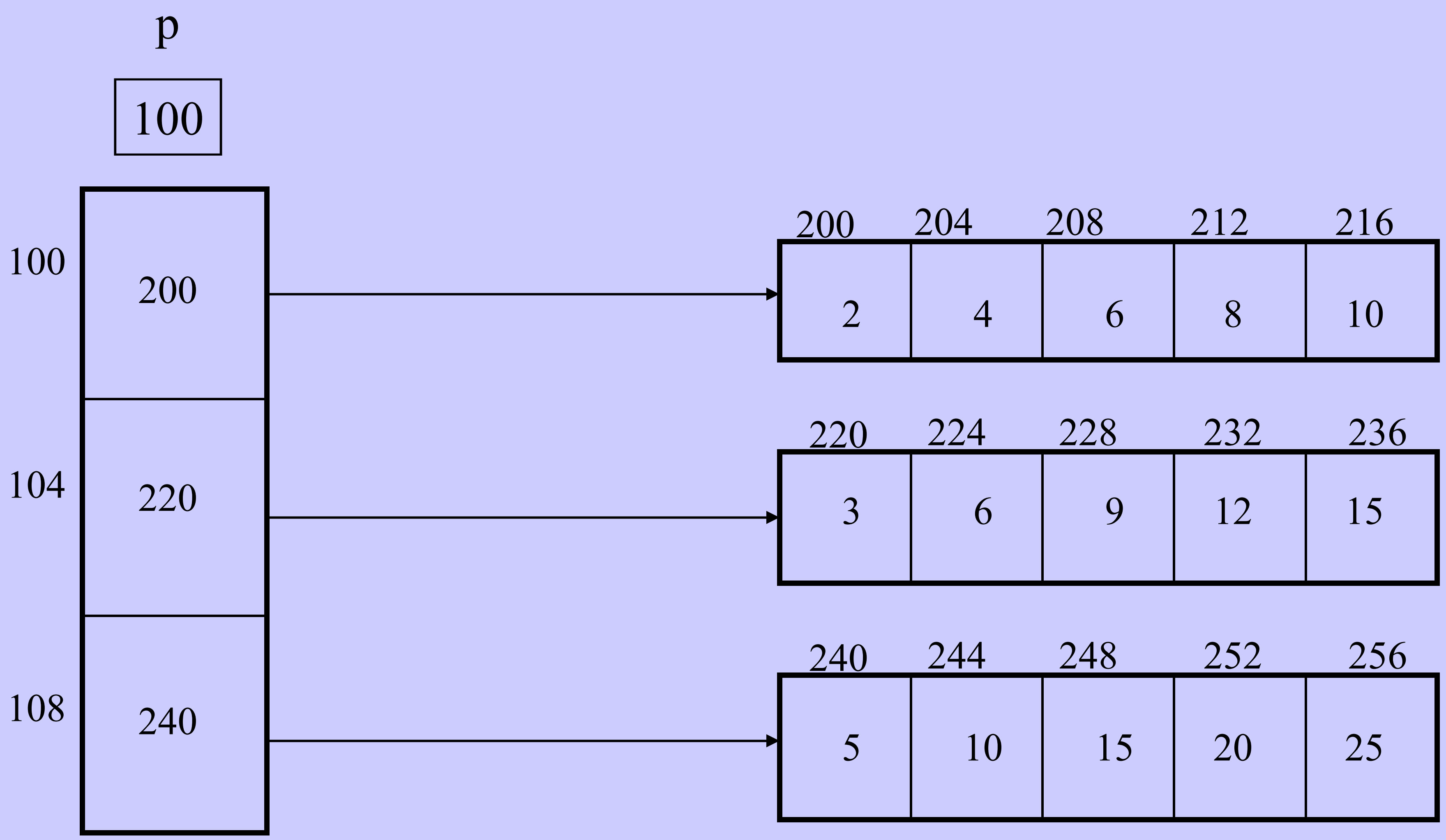
- To summarize, the operation p++; will result in the following computation:
- **New address of p = old address of p + size of data type**

Pointer Arithmetic

- Consider the following declaration of a two-dimensional integer array:
- int p[3][5] = {
- { 2, 4, 6, 8, 10},
- { 3, 6, 9, 12, 15},
- { 5, 10, 15, 20, 25}
- };

- The aforesaid declaration declares an array of three integer pointers, each pointing to the first element of an array of 5 integers. This can be visualized as follows:

Pointer Arithmetic



Pointer Arithmetic

- Here, p points to the first element of the array of pointers.
- *p equals p[0], i.e., it returns the address 200. This address points to the element at offset [0,0], i.e., element 2.
- Therefore, *(*p) returns the value 2.
- Since p is a pointer to another pointer, incrementing p by 1 makes it point to the next element in the array of pointers, i.e., it now points to the element containing the address 220.

Pointer Arithmetic

- Hence, *(p + 1) returns the address 220.
- Therefore, * (*(p + 1)) returns the value at this address, i.e., the element with the value 3. In other words, the element at the offset [1,0].
- The following table gives various pointer expressions, and their values:

Pointer Arithmetic

Pointer Expression	Resulting Address	Variable	Value
*(*p)	200	p[0][0]	2
*(*p+1)	204	p[0][1]	4
((p + 1))	220	p[1][0]	3
((p+1)+1)	224	p[1][1]	6
((p+1)+1)+1	224	p[1][1] + 1	6 + 1 = 7

2-D array and pointer

```
•  main()
•  {     int a[3][4];
   for(i=0;i<3;i++)
•    for(j=0;j<4;j++)   scanf("%d", &a[i][j]);
•  printf("%u ",&a[i][j]);
•  printf("%d",sizeof(a));
•  printf("%d",sizeof(a[0]));
•  printf("%d",sizeof(a[2]));
•  printf("%u  %u  %u %u %u %u\n",a,a[0],a[1],a[2],a[3]);
•  for(i=0;i<3;i++)
•     for(j=0;j<4;j++)
•  printf("%d %d %d %d\n",a[i][j],*(a[i]+j), *(*(a+i)+j));
•  }
```

2-D array and pointer

- main()
- {
- int a[3][4];
- int *p,*q,*r;
- p=a[0];q=a[1];r=a[2];
- for(i=0;i<4;i++,p++)
- printf(“%d ”,*p); printf(“\n”);
- for(i=0;i<4;i++,q++)
- printf(“%d ”,*q); printf(“\n”);
- for(i=0;i<4;i++,r++)
- printf(“%d ”,*r); printf(“\n”);
- }

- main()
- {
- int a[3][4];
- int *p[3];
- p[0]=a[0];p[1]=a[1];
- p[2]=a[2];
- for(i=0;i<4;i++,p[0]++)
- printf(“%d ”,*p[0]); printf(“\n”);
- for(i=0;i<4;i++,p[1]++)
- printf(“%d ”,*p[1]); printf(“\n”);
- for(i=0;i<4;i++,p[2]++)
- printf(“%d ”,*p[2]); printf(“\n”);
- }

2-D array and pointer

- main()
- {
- int a[3][4];
- int *p[3];
- p[0]=a[0];p[1]=a[1];
- p[2]=a[2];
- for(i=0;i<4;i++,p[0]++)
- printf("%d ",*p[0]);
  printf("\n");
- for(i=0;i<4;i++,p[1]++)
- printf("%d ",*p[1]);
  printf("\n");
- for(i=0;i<4;i++,p[2]++)
- printf("%d ",*p[2]);
  printf("\n");
- }

- main()
- {
- int a[3][4];
- int *p[3];
- p[0]=a[0];p[1]=a[1];
- p[2]=a[2];
- for(j=0;j<3;j++)
- {   for(i=0;i<4;i++,p[j]++)
-         printf("%d ",*p[j]);
  printf("\n");
- }
- }

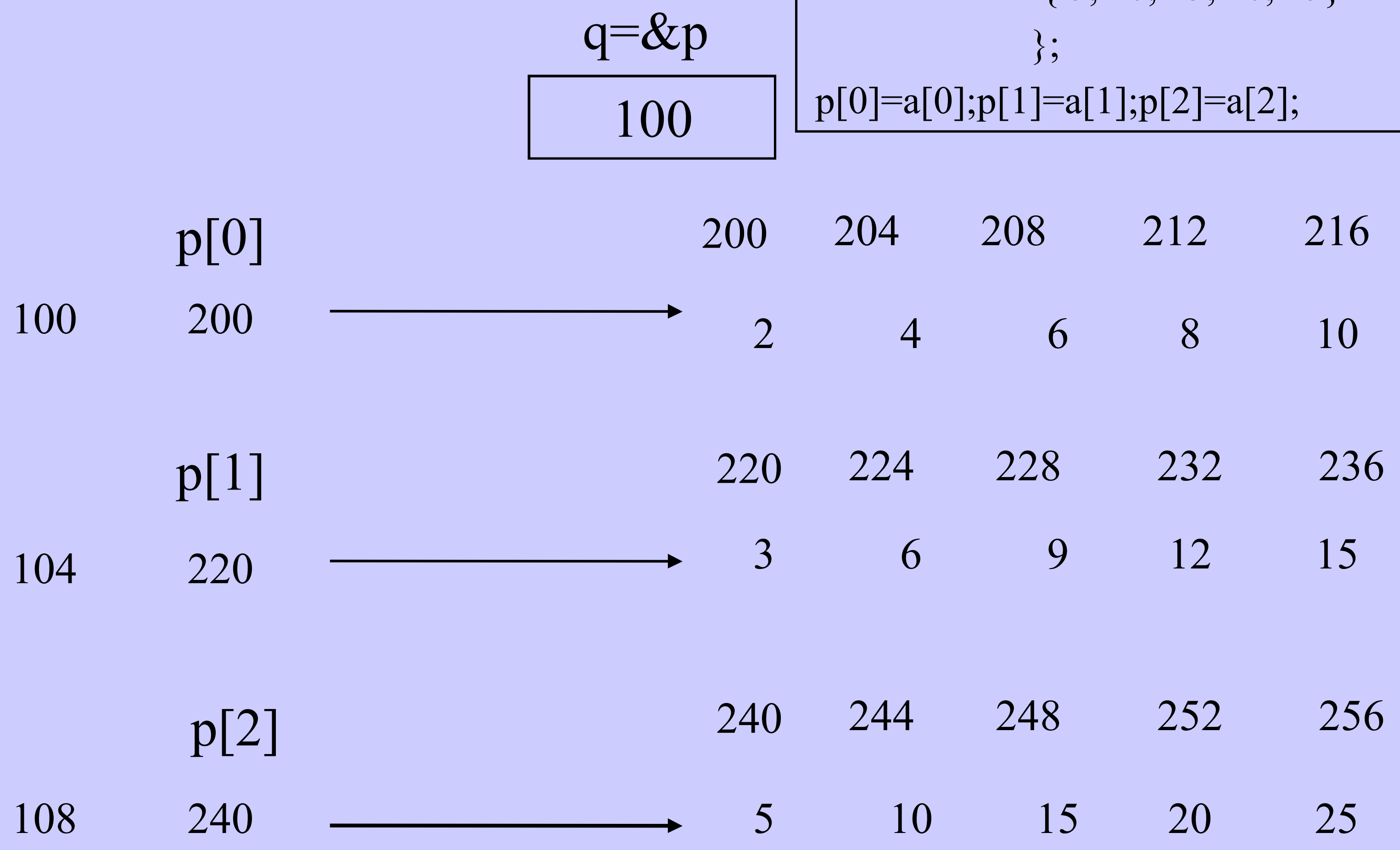
2-D array and pointer

- main()
- {
- int a[3][4];
- int *p[3];
- p[0]=a[0];p[1]=a[1];
- p[2]=a[2];
- for(j=0;j<3;j++)
- {   for(i=0;i<4;i++,p[j]++)
-         printf("%d ",*p[j]);
- printf("\n");
- }
- }

- main()
- {
- int a[3][4];
- int *p[3];
- int **q;
- p[0]=a[0];p[1]=a[1];
- p[2]=a[2];
- q=&p;
- for(j=0;j<3;j++)
- {   for(i=0;i<4;i++)
-         printf("%d ",q[j][i]);
- printf("\n");
- }

2-D array and pointer

```
Int a[3][5]={
              { 2, 4, 6, 8, 10},
              { 3, 6, 9, 12, 15},
              { 5, 10, 15, 20, 25}
             };
p[0]=a[0];p[1]=a[1];p[2]=a[2];
```



Dynamic Memory allocation

- main()
- { int number[1000];
- printf(“number of numbers?”);
- scanf(“%d”, &n); // if n << 1000 ????
- }

- If the data is less than 1000 bytes we are wasting memory.
- If the data is greater than 1000 bytes the program is going to crash.

- **malloc()** allows to allocate exactly the correct amount of memory.

Dynamic Memory allocation

- Library: stdlib.h
- Prototype: void *malloc(size_t number_of_bytes);
- It returns a pointer of type void * that is the start in memory of the reserved portion of size number_of_bytes. If memory cannot be allocated a NULL pointer is returned.
- The void * pointer can be converted to any type, by type casitng.
- size_t indicates **unsigned type of number of bytes**
- Syntax: char * String;
- String = (char *) malloc(1000);

Dynamic Memory allocation

- int *ip; // interger 1-D array with 100 spaces
- ip = (int *) malloc(100*sizeof(int));
- if(ip==NULL) printf("fail to allocate memory);
- else
- {
- do
- }

Dynamic Memory allocation

- Two other functions
  - void *calloc(size_t num_elements, size_t element_size};
- ip = (int *) calloc(100, sizeof(int));

  - void *realloc(void *ptr, size_t new_size);
- ip = (int *) realloc(ip, 50);
- Calloc: allocates a region of memory large enough to hold "n elements" of "size" bytes each. The allocated region is initialized to zero.
- Malloc: allocates "size" bytes of memory. If the allocation succeeds, a pointer to the block of memory is returned.
- If the previously allocated memory is insufficient or more than sufficient. Then, you can change memory size previously allocated using realloc().

Dynamic Memory allocation

- int * ptr1 = (int*) malloc(5 * sizeof(int));
- int * ptr2 = ptr1;
- // ptr1 may become a dangling pointer. In this case both ptr1 and ptr2 are pointing to the same memory location.

- ptr2 = (int*) realloc(ptr2, 10 * sizeof(int));   one example??????

- When realloc is called, the memory location pointed to by both the pointers may get deallocated (in case the contiguous space is not available just after the memory block). ptr2 will now point to the newly shifted location on the heap (returned by realloc), but ptr1 is still pointing to the old location (which is now deallocated).

- **Hence, ptr1 is a dangling pointer.**
- If pointer passed to realloc is null, then it will behave exactly like malloc.
- If the area is moved to new location then a free on the previous location is called.
- If contents will not change in the existing region. The new memory (in case you are increasing memory in realloc) will not be initialized and will hold garbage value.
- If realloc() fails the original block is left untouched; it is not freed or moved.

Dynamic Memory allocation

- **Free** (Free the memory allocated using malloc, calloc or realloc)
- free functions frees the memory on the heap, pointed to by a pointer. Signature of free function is
- void free(void* ptr);
- ptr must be pointing to a memory which is allocated using malloc, calloc or realloc.
- If ptr is called on a memory which is not on heap or on a dangling pointer, then the behavior is undefined.
- If ptr is NULL, then free does nothing and returns (So, its ok to call free on null pointers).
- int x = 2; int* ptr = &x;free(ptr); //UNDEFINED.
- int *ptr2; // UN initialized, hence dangling pointerfree(ptr2); //UNDEFINED
- int *ptr3 = NULL;free(ptr3); //OK.

Example (1-D)

- main()
- {
- int *a,n,i;
- printf(“How many elements?\n”);
- scanf(“%d”,&n);
- a=(int *)malloc(sizeof(int)*n);
- if(a!=NULL)

      for(i=0;i<n;i++) scanf(“%d”,&a[i]);
- }

Example (2-D)

- main()
- {
- int **a,r,c,i;
- printf("enter the dimension of the matrix\n");
- scanf("%d%d",&r,&c);
- a=(int **)malloc(sizeof(int *)*r);
- for(i=0;i<r;i++)
-   a[i]=(int *)malloc(sizeof(int)*c));
- for(i=0;i<r;i++)
-      for(j=0;j<c;j++)
-           scanf("%d",&a[i][j]);
- }

Memory deallocation

- void free (void *);

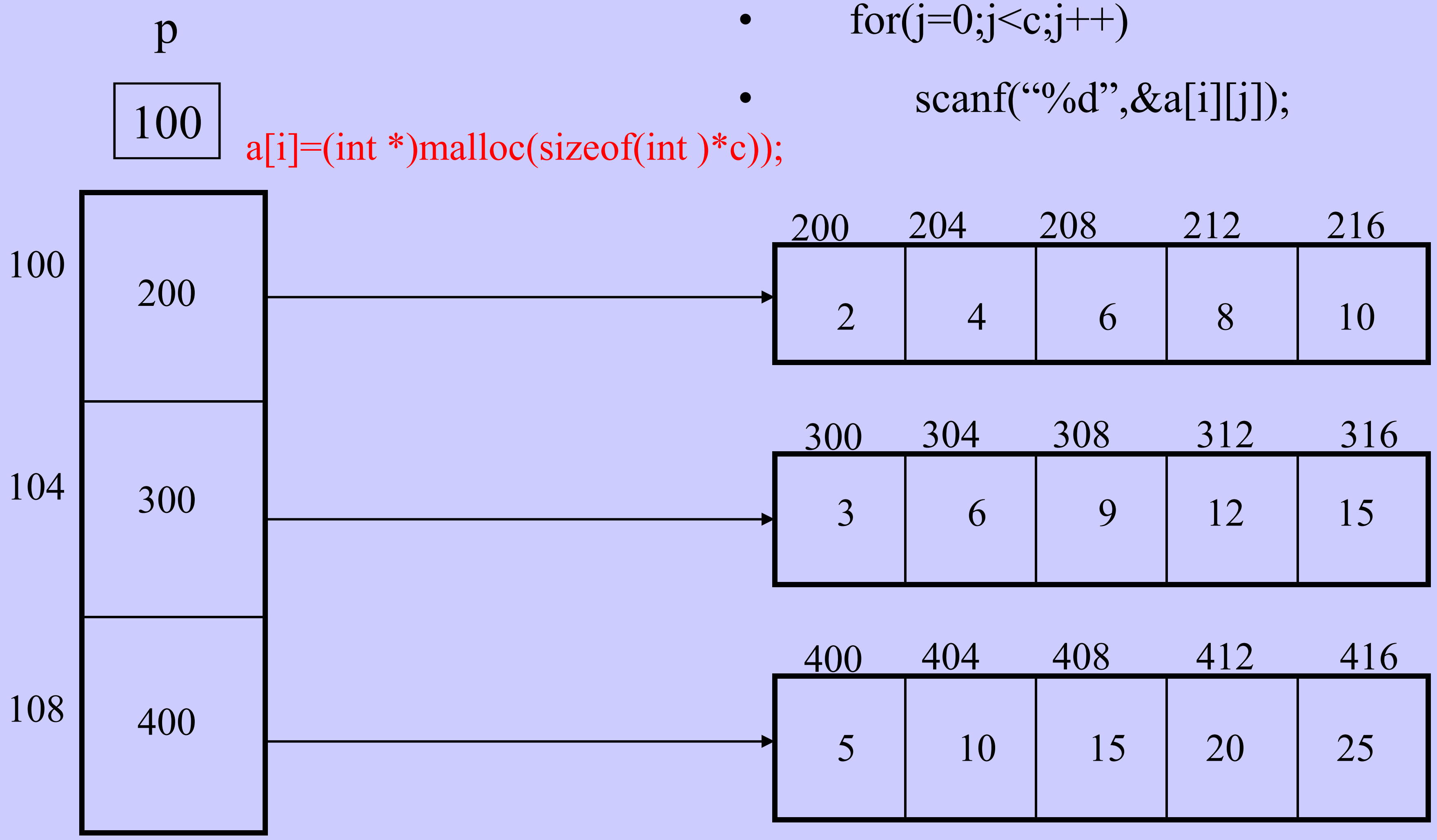
- for(i=0;i<r;i++) free(a[i]);
- free(a);

Pointer Arithmetic

r=3; c=5

- for(i=0;i<r;i++)
- for(j=0;j<c;j++)
- scanf("%d",&a[i][j]);

a[i]=(int *)malloc(sizeof(int)*c);



a=(int **)malloc(sizeof(int *)*r);

Function

Need for functions

As, programs become more complex and large, several problems be arises. Most common are:

- Algorithms for solving more complex problems become more difficult and ,hence difficult to design/implement
- As programs become larger and complex, debugging and testing becomes more difficult.
- As programs become larger and complex, more documentation is required to make the program understandable for people who will use and maintain the program
- Writing functions avoids rewriting the same code over and over.
- Readability of the program increases.

Functions

Defining a Function

A function is a self-contained block of statements to perform a specific task

In C, a program is a collection of well defined and interrelated functions. The general form of a function is as follows

```
return-type  function(type arg1, type arg2, …..)
{
     local variables Declaration;
          executable statement 1;
            executable statement 2;
                   :
                   :
          return(expression);
 }
```

Functions

Example 1: Function for finding the biggest of two integers

```
int  find_big(int  a, int b)
{
      if ( a > b) return a;
     else return b;
}
```

Functions

Example

Program for finding biggest of two integers using the function

```
        void main( )
            {
              int  num1, num2, big;
             int  find_big(int , int); // prototype declaration the function
             scanf(“%d%d”, &num1, &num2);
             big=find_big(num1,num2);
             printf(“ The biggest is : %d “,  big);
            }
      int  find_big(int  a, int b)
       {
                if ( a > b) return a;
                else return b;
       }
```

Functions

- main()
- {
- printf(“in main\n”); fun1(); fun2();fun3();
- printf(“in main\n”);
- }
- fun1()
- { printf(“in fun1\n”);}
- fun2()
- { printf(“in fun2\n”);}
- fun3()
- { printf(“in fun3\n”);}

Functions

- main()
- {void fun1(void);
- void fun2(void);
- void fun3(void);
- printf(“in main\n”);
- fun1(); fun2();fun3(); //function calling
- printf(“in main\n”);
- }
- void fun1(void)   // function definition
- { printf(“in fun1\n”);}
- void fun2(void)
- { printf(“in fun2\n”);fun3();}
- void fun3(void)
- { printf(“in fun3\n”);}

Functions

```
•   void fun1(void);
•   void fun2(void);
•   void fun3(void);
•   main()
•   {
•   printf(“in main\n”);
•   fun1(); fun2();fun3(); //function calling
•   printf(“in main\n”);
•   }
•   void fun1(void)             // function definition
•   { printf(“in fun1\n”);}
•   void fun2(void)
•   { printf(“in fun2\n”); fun3();}
•   void fun3(void)
•   { printf(“in fun3\n”);}
```

Functions

- Any c program must contain at least one function
- If only one function, it must be ***main().***
- If more than one function, **exactly one main().** Execution starts with main().
- No limit on the number of functions.
- After each function done its job, the control is returns to the function from which it was called.

Advantages of Function

- Functions therefore facilitate:
  - Reusability
  - Procedural abstraction

- By procedural abstraction, we mean that once a function is written, it serves as a **black box**. All that a programmer would have to know to invoke a function would be to know its name, and the parameters that it expects.

Function Parameters

- Function parameters are defined as part of a function header, and are specified inside the parentheses of the function.

- The reason that functions are designed to accept parameters is that you can embody generic code inside the function.

Function Parameters

- Consider the following printf( ) statement:
- printf(“%d”, i);

- This is actually a call to the printf( ) function with two pieces of information passed to it, namely, the format string that controls the output of the data, and the variable that is to be printed.

- The format string, and the variable name can be called the parameters to the function printf( ) in our example.

Function Parameters

- Function parameters are therefore a mechanism wherein values can be passed down to a function for necessary processing.
- It is important to remember that not all function need be defined to accept parameters, though most functions do.
- **The important concept of a function returning a value, and also the return type of a function.**

Example

Program for finding biggest of two integers using the function

```
void main( )
  {
    int  num1, num2, big;
   int  find_big(); // prototype declaration
the function
  //  scanf("%d%d", &num1, &num2);
   big=find_big(); sum=add();
   printf(" The biggest is : %d ",  big);
  }
```

```
int  find_big()
   {      int a, b;
           scanf("%d%d", &a, &b);
             if ( a > b) return a;
             else return b;
   }


int  add()
   {      int a, b;
           scanf("%d%d", &a, &b);
           return a+b;


   }
```

Functions

- Function prototype (declaration)
  - return_type function_name(type(1) argument(1),....,type(n) argument(n));
- Function call
  - function_name(argument(1),....argument(n));
- Function definition
  - return_type function_name(type(1) argument(1),..,type(n) argument(n)) { //body of function }
    - Return statement
    - return (expression);

      OR
    - return;

Functions

- int ncr(int,int);
- int fact(int);
- main()
- {
- int n,r;
- scanf(“%d%d”,&n,&r);
- if(r>=0 && r<=n)
- value=ncr(n,r);
- printf(“ncr=%d\n”,value);
- }

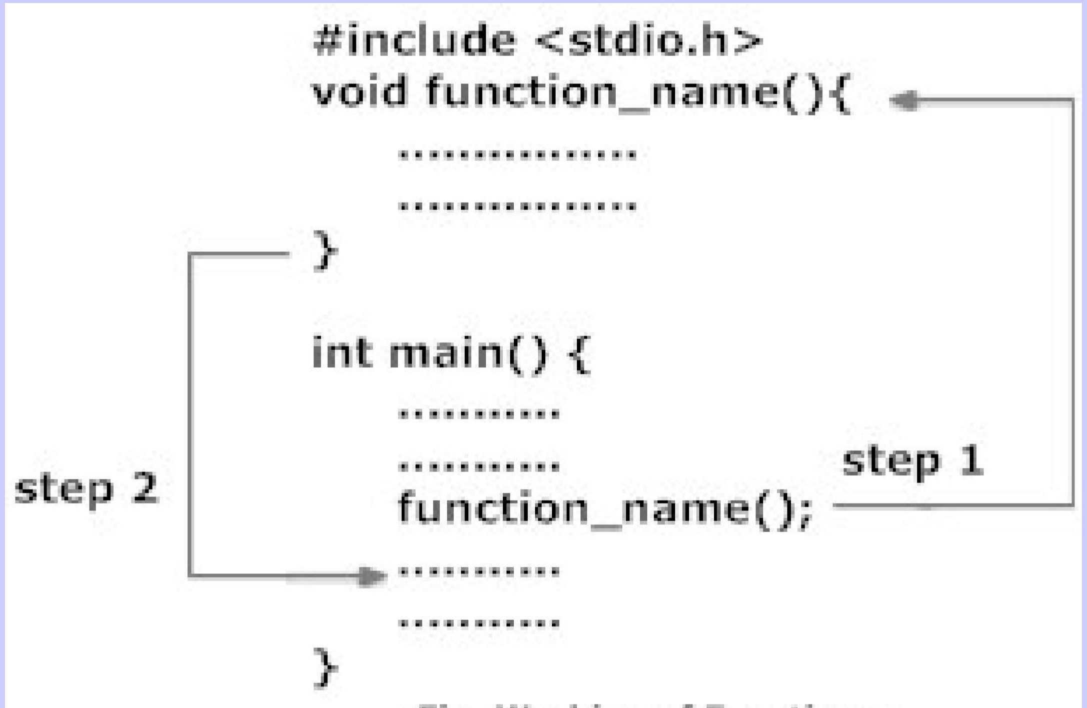
- int ncr(int a,int b)
- { int temp1,temp2,temp3;
- temp1=fact(a);
- temp2=fact(b);
- temp3=fact(a-b);
- return(temp1/(temp2*temp3));
- }

- int fact(int x);
- { int temp=1;
- while(x) {temp*=x; x--;}
- return temp;}

Invoking Functions

- In C, functions that have parameters are invoked in one of two ways:
  - Call by value
  - Call by reference

Working of functions



```
#include <stdio.h>
void function_name(){
    ................
    ................
}

int main() {
    ...........
    ...........
    function_name();
    ...........
    ...........
}
```

Fig: Working of Functions

Call by Value

```
void swap(int,int );
 main()
     { int a=10, b=20;
         swap(a, b);                // a and b are actual parameter
         printf(“ %d %d \n”,a,b);
     }
 void swap (int x, int y)       // x and y are formal parameter
    {  int temp = x;
        x=  y;
        y=temp;
    }
```

Call by Value

- In the preceding example, the function **main( )** declared and initialized two integers **a** and **b**, and then invoked the function **swap( )** by passing **a** and **b** as arguments to the function **swap( )**.
- The function **swap( )** receives the arguments a and b into its parameters **x** and **y**. In fact, the function **swap( )** receives a copy of the values of **a** and **b** into its parameters.
- The parameters of a function are local to that function, and hence, any changes made by the called function to its parameters affect only the copy received by the called function, and do not affect the value of the variables in the called function. This is the call by value mechanism.

Call by Reference

- Call by reference means that the called function should be able to refer to the variables of the calling function directly, and not create its own copy of the values in different variables.

- This would be possible only if the addresses of the variables of the calling function are passed down as parameters to the called function.

- In a call by reference, therefore, the called function directly makes changes to the variables of the calling function.

Call by Reference

- Consider the same swap( ) that we discussed earlier now rewritten using a call by reference.

```
void swap( int *, int *);
 main()
     { int a=10, b=20;
         swap(&a, &b);
         printf(“ %d %d \n”,a,b);
     }
void swap (int *x, int *y)
    {   int temp=*x;
         *x=*y;
         *y=temp;
    }
```

Returning a Value From a Function

- Just as data can be passed to a function through the function’s parameters at the time of call, the function can return a value back to the caller of the function.
- In C, functions can return a value through the return statement.
- The return statement not only returns a value back to the calling function, it also returns control back to the calling function.
- A function can return only one value, though it can return one of several values based on the evaluation of certain conditions

Value return by function

- int ncr(int,int);
- int fact(int);
- main()
- {
- int n,r;
- scanf(“%d%d”,&n,&r);
- if(r>=0 && r<=n)
- value=ncr(n,r);
- printf(“ncr=%d\n”,value);
- }

- int ncr(int a,int b)
- { int temp1,temp2,temp3;
- temp1=fact(a);
- temp2=fact(b);
- temp3=fact(a-b);
- return(temp1/(temp2*temp3));
- }

- int fact(int x);
- { int temp=1;
- while(x) {temp*=x; x--;}
- return temp;}

Functions returning pointers

- Int *fun(int *);
- main()
- {
- int a=100,*b;
- b=fun(&a);
- printf(“addresses: %u %u\n”, &a, b);
- }
- int *fun(int *x)
- {
- x++;
- return(x);
- }

Passing Arrays to Functions

- Arrays are inherently passed to functions through a call by reference.

- For example, if an integer array named **int_array** of 10 elements is to be passed to a function called **fn( )**, then it would be passed as:
- **fn(int_array);**

- **Recall that int_array is actually the address of the first element of the array, i.e., &int_array[0]. Therefore, this would be a call by reference.**

Passing Arrays to Functions

- The parameter of the called function **fn( )** would can be defined in one of three ways:

- **fn(int num_list[ ]);** or For static array

- **fn(int num_list[10]);** or ✗

- **fn(int *num_list)** For dynamic array

Pass 1-D array to a function

- void display (int a[], int);
- main()
- {int I,n;
- int a[10];
- scanf(“%d”,&n);
- for(I=0;I<n;I++)
- scanf(“%d”,&a[I]);
- display(a,n);
- }

- void display (int x[], int k)
- {int I;
- for(I=0;I<k;I++)
- printf(“%6d”,x[I]);
- }

Pass 2-D array to a function

- #define s 10
- void display (int a[][], int, int);
- main()
- {int I,n,m,j;
- int a[s][20];
- scanf("%d%d",&n,&m);
- for(I=0;I<n;I++)
-     for(j=0;j<m;j++)
-         scanf("%d",&a[I][j]);
- display(a,n,m);
- }

- void display (int x[][s], int k, int t)
- {int I,j;
- for(I=0;I<k;I++)
- { for(j=0;j<t;j++)
-         printf("%6d",x[I][j]);
-     printf("\n");
- }
- }

Function Prototype

- C assumes that a function returns a default value of **int** if the function does not specify a return type.
- In case a function has to return a value that is not an integer, then the function itself has to be defined of the specific data type it returns.
- Functions should be declared before they are used.

```
double pow( double x, double y);
double pow( double, double);
```

- A function prototype tells the compiler the number and data types of arguments to be passed to the function and the data type of the value that is to be returned by the function.

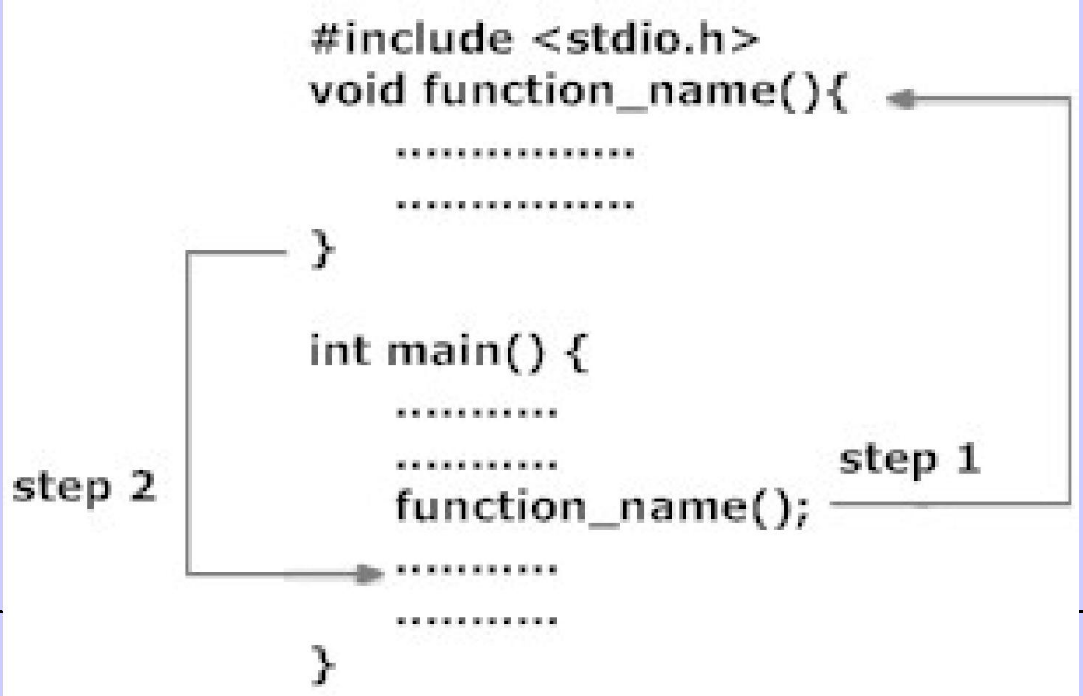
Function Prototype

- #include <stdio.h>
- main( )
- {
-   void add(float, float);
- float i, j, value;
- scanf("%f %f", &i, &j);
- fflush(stdin);
- add(i, j);
- printf("the total is %d\n", value);
- }

- void add(float a, float b)
- {
-   printf("%f", a + b);
-   return;
- }

Function Calls

- It is important for us to know what happens under the hood when a function executes.
- A logical extension of the aforesaid point is the situation that arises when a function calls another function.
- In short, we need to know the call mechanism, and the return mechanism.



```
#include <stdio.h>
void function_name(){
    ...............
    ...............
}

int main() {
    ..........
    ..........
    function_name();
    ..........
    ..........
}
```

Fig: Working of Functions

Function Calls – A Top Level Overview

- When a function call is encountered, it involves the following steps:

1. Each expression in the argument list is evaluated.
2. The value of the expression is converted, if necessary, to the type of the formal parameter, and that value is assigned to the corresponding formal parameter at the beginning of the body of the function.
3. The body of the function is executed.

Function Calls – A Top Level Overview

4. If the return statement includes an expression, then the value of the expression is converted, if necessary, to the type specified by the type specifier of the function, and that value is passed back to the calling function.
5. If no return statement is present, the control is passed back to the calling function when the end of the body of the function is reached. No useful value is returned.

Function Calls & The Runtime Stack

- The C language uses a stack-based runtime environment, which is also referred to as a runtime stack, or a call stack.

arguments

bookkeeping information
(return address)

local data

local temporaries

Tracing Function Calls

- int z;
- main( )
- {
-   int x;
-   fn_a( );
-   ….. . ← return instruction
- .
- .
- .
- }
-

```
fn_a( int m )
{
 int y;
 fn_b( );
 ….. ← return instruction
 .
 .
 .
}
```

```
fn_b( int n )
{
 int b;
  …..
 .
 .
 .
}
```

Evaluation of arg. list

- main()
- {
- int a=20, b=30;
- fun(a++, ++a, b--,--b);
- }

- fun(int x, int y, int z, int w)
- {
- printf(“%d %d %d %d\n”,x,y,z,w);
- }

Recursion

- A function call it self with some other realistic conditions

Recursion

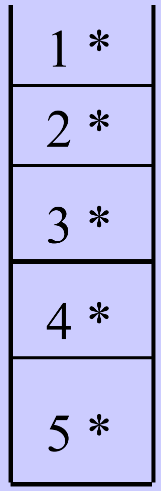
n! = n*(n-1)*….*3*2*1, if n>=1
=n*(n-1)!
= 1 if n=0

```
int fact(int n)
{
int prod, n;
prod = 1
for (x = n; x > 0; x--)
 prod *= x;
  return (prod)
 }
```

- 0! = 1
- 1! = 1
- 2! = 2 * 1
- 3! = 3 * 2 * 1
- 4! = 4 * 3 * 2 * 1

The Factorial Function

- Repeating this process, you have:
- 5!=5*4!
- 4!=4*3!
- 3!=3*2!
- 2!=2*1!
- 1!=1*0!
- 0! = 1



- This produces:
- 5! = 5 * 4! = 5 * 24 = 120
- 4! = 4 * 3! = 4 * 6 = 24
- 3! = 3 * 2! = 3 * 2 = 6
- 2! = 2 * 1! = 2 * 1 = 2
- 1! = 1 * 0! = 1 * 1 = 1
- 0! = 1

Properties of Recursive Definitions

- **For at least one argument or group of arguments, a recursive function f must be defined in terms that do not involve f.**
- **The successive call of the function moves closer to the base condition.**

Recursion

- A function which call it-self.
- This is possible due to runtime stack

```
• int fact (int n)
•  {
•   if(n==0) return 1;
•   else return (n*fact(n-1));
•  }
```

```
int fact( int n)
 {
   int x, y;
   if ( n == 0)
     return (1);
   else
    {
     x = n-1;
     y = fact(x);
     return ( n * y);
    }
  }
```

Recursion

- Gcd(a,b)=

Scope rules

- A scope is a region of the program where a defined variable can have its existence and beyond that variable cannot be accessed.
- There are three places where variables can be defined in C
  - Inside a function or a block is called **local** variable
  - Outside of all functions which is called **global** variable
  - In the definition of the function parameters which is called as **formal** parameters

Local Variables

- The variables declared inside a function are local to that function.
- It can be accessed only with in that function.
- Memories for the local variables are allocated only when the function is invoked and deallocated when the control returns to the calling function.
- These variables are also known to be automatic variables.
- Same variable can be used in two different functions.

Global Variables

- Global variable is defined outside all functions.
- A typical convention of defining global variables is before the main( ) function.
- A variable declared as global is visible to the function main( ) as well as to any other functions called from main( ).
- A variable defined as global has file scope, that is, it is visible to all the functions written within the scope of a program file.
- Global variable holds the value throughout the lifetime of the program
- A program can have same name for local and global variable and in that case local variable will get preference.

Local and Global Variables

Example

```
int  a;                    /* Global variable */
float  b;   /* Global variable */
void main()
{
      int  c;              /* Local variable*/
      a=10;
      b=1.2;
      c= 20;
             fun( );
             printf(" %d ", a);          /* prints 20 */
      }
 void fun()
 {
                 a+=10;
}
```

Local and Global Variables

Example :

```
int  a=100;            /* Global variable, default initialization 0
float  b;   /* Global variable */
void main()
{
     int  c;       /* Local variable*/
     int a;              // Local variable
     a=10;
     b=1.2;
     c= 20;
          fun( );
          printf(“ %d “, a);          /* prints 10, local variable */
       }
  void fun()
  {
       a+=30;        // now, a=130, global variable
}
```

Initialization of local and global variables

- When local variable is defined, not initialized by the system
  - Programmer has to initialized it.

- Global variables are initialized automatically, the default initial value is
  - int 0
  - float 0.0
  - double 0.0
  - char ‘\0’
  - pointer NULL

Nested Declarations

- Nested declarations can be treated in a similar way to temporary expressions, allocating them on the stack as the block is entered and deleting them on exit.

```
void p (int x, double y)
{  char a;
   int i;
   { double x;  // block A
     int y;
      …
    }
   {  double x; // block B
      int j;
      …
    }
    {  char *a; // block C
       int k;
       …
    }
}
```

Block structure

```
•   void main()
•       {
•           int a=10;              // Local variable
•           printf(" %d ", a);     // 10
•            {
•            int a=20;
•           printf("%d ",a)        // 20
•             }
•            { float a=15.5;
•               printf("f ",a)     // 15.5
•            }
•          printf("%d ",a);
•       }
•
```

Storage Qualifiers

- There two kinds of memory locations in computer: RAM and CPU registers.
- Storage class determines in which of these two storage locations the value will be stored.
- The storage qualifier determines the **lifetime** of the storage associated with the identified variable (length of time it retains the value in memory).
- What would be the initial value of a variable, if not specified.
- The **scope** of a name is the part of the program within which the can used. The storage class determine the scope of the variable.

Storage Qualifiers

- The storage qualifiers in C are:
- auto
- static
- extern
- register

automatic Variables

- **auto**matic variables are local to a block, and are discarded on exit from the block.
- Block implies a function block, a conditional block, or an iterative block.
- Declarations within a block create automatic variables if no storage class specification is mentioned, or if the **auto** specifier is used.
- Variable declarations, therefore, are by default, **auto**.

automatic Variables

- Stored in Memory
- If not initialized in the declaration statement, their initial value is unpredictable value often called garbage value
- Local (visible) to the block in which the variable is declared. If the variable is declared with in a function, then it is only visible to that function.
- It retains its value till the control remains in that block. As the execution of the block/function is completed, it is cleared and its memory destroyed.

automatic Variables

Example: Program illustrating auto variables

```
void main( )
{
    fun( );
    fun( );
    fun( );
 }
void fun( )
{
    auto int  b = 20;   /*Auto (Local) variable to fun() */
    b++;
    printf( " Calling Fun( ) :  %d \n", b);
}
```

The output of the above program is as follows

```
Calling Fun( ) :  21
Calling Fun( ) :  21
Calling Fun( ) :  21
```

Static Variables

- **Static** variables may be local to a block or external to all blocks, but in either case retain their values across exit from, and reentry to functions and blocks.

- Within a block, including a block that provides the code for a function, static variables are declared with the keyword **static**.

- Let us rewrite the code for the example involving auto variables to incorporate the declaration of static variables.

Static Variables

- Stored in Memory
- If not initialized in the declaration, it is initialized to zero.
- Local(Visible) to the block in which the variable is declared. If the variable is declared inside a function, then it is visible to that function. But, if the variable is declared outside the function, then it is visible to all functions following its declaration.
- It retains its value between different function calls. That is, when for the first a function is called, a static variable is created with initial value zero, and in subsequent calls it retains its present value.

Static Variables

Example: Program illustrating auto variables

```
void main( )
{
    fun( );
    fun( );
    fun( );
}
void fun( )
{
    static  int  b = 20;   /*Auto (Local) variable to fun() */
    b++;
    printf( " Calling Fun( ) :  %d \n", b);
}
```

The output of the above program is as follows

```
Calling Fun( ) :  21
Calling Fun( ) :  22
Calling Fun( ) :  23
```

Static and Global Variables – A Comparison

- From the preceding example, it seems that static variables are functionally similar to global variables in that they retain their value even after exiting from a function in which it has been defined as static.

- But an important difference is that while a global variable has file scope, and is therefore visible to all functions defined within that program file, a static variable is visible only to the function in which it has been defined as static.

Example: usage of static variable

```
• int main()
• {
• void show1(void);
• void show2(void);
• void show3(void);
• show1();show2();
• show3();show1();
• show2();show3();
• return;
• }
```

```
void show1(void)
{
static int x=10;
x+=5;
printf("%d\n",x);
}
void show2(void)
{
int x=10;
x+=5;
printf("%d\n",x);
}
```

```
void show3(void)
{
static int x;
x=10;
x+=5;
printf("%d\n",x);
}
```

```
15 15 15 20 15 15
```

Register Storage Class

- Stored in Processor registers, if register is available. If no register is available, the variable is stored in memory, and works as if its storage class is auto.
- If not initialized in the declaration, garbage value.
- Local(Visible) to the block in which the variable is declared. If the variable is declared inside a function, then it is visible to that function. But, if the variable is declared outside the function, then it is visible to all functions following its declaration.
- It retains its value till the control remains in that block. As the execution of the block/function is completed, it is cleared and its memory destroyed.

Register Storage Class

- It is not possible to take the address of a register variable, even it is not stored in register.
- This storage class for faster execution (frequently used variable, like counter of a loop).
- register storage class not supported for all data types
- Example

  register float a;

  register double b;

  register long c;

  The above declarations are wrong, though you may not get any error message, complier treat then as auto variable.

Extern Variables

- The Scope of external variables is global. External variables are declared outside all functions, usually in the beginning of the program file. As all the function will be defined after these variables declaration, these variables are visible to all functions of the program.
- This storage class is useful for a multi files program.
- Stored in memory
- If not initialized in the declaration, it is initialized to zero.

Extern Variables

- **Program a.c**
- int val /* global */
- main( )
- { //extern int value;
- printf(“Enter value”);
- scanf(“%d”, &val);
- compute( ); /* function call */
- }

- **Program b.c**
- compute( )
- {
- extern int val; /* implies that val is defined in another program containing a function to which the current function will be linked to at the time of compilation */
- }

Use of extern inside a function is optional as long as the variable is declared outside and above the function in the same source file.

Extern Variables

```c
#include <stdio.h>

int count ;
extern void write_extern();

main()
{
   write_extern();
}
```

```c
#include <stdio.h>

extern int count;

void write_extern(void)
{
   count = 5;
   printf("count is %d\n", count);
}
```

```
$gcc main.c write.c
```

Extern Variables

- From the preceding example, it is clear that though val is declared global in the function main( ) in program a.c, it has been declared again within the function compute( ), which is defined in its own program file, b.c.

- However, the qualifier extern has been added to its declaration.

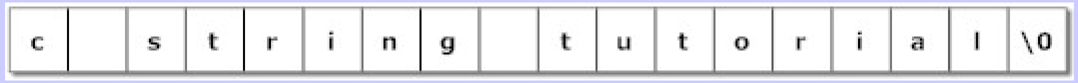
- The extern qualifier in the function compute( ) in b.c indicates to the compiler **(when a.c and b.c are compiled together)** that the variable used in this program file has been declared in another program file.

Which to use When

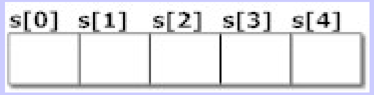
- If wants the value of a variable to persists between the function calls: use static
- If a variable used frequently you can declare as register.
- Extern storage for those variables which are being used in almost all functions.

String

- In C programming, array of character are called strings. A string is terminated by null character \0. For example:
- "c string tutorial" Here, "c string tutorial" is a string. When, compiler encounters strings, it appends null character at the end of string.



- char s[5];



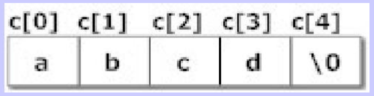
Strings

- For example, the following defines a string of 4 characters:

```
main( )
{       char name[5];
        name[0] = ‘A’;
        name[1] = ‘R’;
        name[2] = ‘U’;
        name[3] = ‘N’;
        name[4] = ‘\0’;
        return 0;
}
```

String

- char c[]="abcd";
- char c[5]="abcd";
- char c[]={'a','b','c','d','\0'};
- char c[5]={'a','b','c','d','\0'};



Strings

- To define a character array for storing a string of n characters, we would need to define a character array of size n+1 characters.

- This is because all character arrays are terminated by a **NULL** character (‘\0’).

- To define a character array called name for storing a ten-character name, we will define the array as:
- char name[11];

- where name[0] through name[9] will contain the characters comprising the name, and name[10] will store the **NULL** character.

Strings

- C doesn’t allow one array to be assigned to another, so we can’t write an assignment of the form.

```
name = “ARUN”;    /* illegal*/
```

- Instead we must use the standard library function strcpy to copy the string constant into the string variable.

```
strcpy(name,“ARUN”);
```

- To print a string we use printf with a special %s control character:

```
printf(“%s”,name);
```

Representation of a Character Array

name

a	b	c	d	e	f	g	h	i	j	\0
name[0]	name[1]	name[2]	name[3]	name[4]	name[5]	name[6]	name[7]	name[8]	name[9]	name[10]

printf(“%s”,name);   output: abcdefghij

Representation of a Character Array

name

a	b	c	d	e	f	\0	h	i	j	\0
name[0]	name[1]	name[2]	name[3]	name[4]	name[5]	name[6]	name[7]	name[8]	name[9]	name[10]

```
name[6]=‘\0’;
printf(“%s”,name);     output: abcdef
```

printf function to print string

```
#include<stdio.h>
main()
{
char
    a[15]="1234567890986";
printf("%s",a);
printf("%20s",a);
printf("%-20s%c",a,'a');
}
```

```
• 1234567890986
•        1234567890986
• 1234567890986       a
```

String-Based I/O

- #include <stdio.h>
- #include <conio.h>
- int main( )
- {
-   char str[11];
-   puts("Enter a string of maximum 10 characters");
-   gets(str);
-   fflush(stdin);
-   puts(str);
- return 0;
- }

String constant

• "good"	//string constant
• ""	//null string constant
• “ "	//string constant of six white space
• "x"	//string constant having single character.
• "Earth is round\n"	//prints string with newline

Array Manipulation Using Subscripts

```
•  #include<stdio.h>
•  /* displays each element of the array on a new line */
•  main( )
•   {
•     int i;
•     char array[11];
•     printf( "enter a string of maximum 10 characters\n");
•     gets(array);
•     fflush(stdin);
•     for (i = 0; array[i] != '\0'; i = i +1)
•      printf("Element %d is %c\n", i +1, array[i]);
•   }
```

C program to read line of text manually

```
#include <stdio.h>
 int main()
{
char name[30],ch;
 int i=0;
printf("Enter name: ");
while(ch!='\n') // terminates if user hit enter
{ ch=getchar();
name[i]=ch;
i++;
}
name[i]='\0'; // inserting null character at end
printf("Name: %s",name);
return 0;
}
```

Array Manipulation Using Subscripts

- /* this function finds the length of a character string */
- #include <stdio.h>
- main( )
- {
-   int i = 0;
-   char string[11];
-   printf(“Enter a string of maximum ten characters\n”);
-   gets(string);
-   fflush(stdin);
-   for(i =0; string[i] != ‘\0’; i = i + 1)
-     ;
-   printf(“The length of the string is %d \n”, i);
- }

Array Manipulation Using Subscripts

```
•   /* this function converts a string to upper case */
•   #include <stdio.h>
•   main( )
•   {
•     char string[51];
•     int i = 0;
•     printf("Enter a string of maximum 50 characters\n");
•     gets(string);
•     fflush(stdin);
•     while (string[i] != '\0')
•      {
•        if(string[i] >= 'a' && string[i] <= 'z')
•         { string[i] = string[i] - 32;   // string[i] = 'A' + string[i] – 'a';
•           i = i + 1;
•         }
•      }
•     printf("The converted string is %s\n", string); }
```

Two-Dimensional Character Arrays

- If you like to store an array of 11 names, with each name containing up to a maximum of 30 characters, you can declare it as a two-dimensional character array such as: char name[11][31];

- Here, name[0] through name[10] would store character strings representing names of 30 characters each.

- The first index represents the number of names, and the second index represents the maximum size of each name.

Printing Two-Dimensional Character Arrays

- main( )
- {
-   char team_india [11][30] = { “Akash Chopra”,
-   “Virendra Sehwag”,
-   “Rahul Dravid”
-   “Sachin Tendulkar”,
-   “V.V.S. Laxman”,
-   “Yuvraj Singh”,
-   “Ajit Agarkar”,
-   “Parthiv Patel”,
-   “Anil Kumble”,
-   “L. Balaji”,
-   “Irfan Pathan”
-   };

Printing Two-Dimensional Character Arrays

- int i;
- for(i = 0; i < 11; i ++)
- {
-    printf(“%s”, team_india[i]);
- }
- }

- Particular elements can also be printed from a two dimensional array by using the second subscript.

- Printf(“%c”, team_india[10][6] would print the character ‘P’ from the string “Pathan”

String Functions

- Main()
- { /*****   ****/
- char a[]=“abcd”,b[10]; int J;
- printf(“%d”,strlen(a));
- printf(“%d”,strlen(“abcd”);
- strcpy(b,a);
- printf(“a=%s b=%s\n”,a,b);
- J=strcmp(a,b);
- printf(“J=%d\n”,J);
- strcat(b, “xyz”);
- J=strcmp(b,a);
- printf(“J=%d\n”,J);
- }

One-dimensional Character Arrays Using Pointers

- What is string pointing to? Seemingly, it points to a string, but actually, string is pointing to a character at string[0].

- Recall that a string is just a sequence of characters terminated by a null character (‘\0’).

- When the string name is passed down as a parameter to a printf( ) function, it starts printing from the starting address of the string till it encounters a null character (‘\0’), which happens to be the string terminator.

Difference Between Pointers and Arrays

- There are subtle differences between pointers and arrays.

- Consider the following declaration:
- char string[10], *p;
- Both **string** and **p** are pointers to **char**.

Difference Between Pointers and Arrays

- However, **string[10]** has 10 bytes of contiguous storage allocated for it.
- Thus, during execution, **string is effectively a pointer with a constant address**, namely, the address **&string[0]**;
- **And this address cannot be changed during the life of the program**

Difference Between Pointers and Arrays

- However, p is a pointer variable that can point to one string at one point of time during the running of the program, and can also point to another string at another point of time during the running of the same program.

- p is a variable, and a variable by its very definition can hold different values at different points of time during program execution.

- **Therefore, we can conclude that a string is a pointer constant, whereas an explicit character pointer declaration is a character pointer variable.**

One-dimensional Character Arrays Using Pointers

- The one-dimensional character array declared earlier (char string[11]) can be alternately declared as:

- char *string; /* string is now a explicit pointer variable that can point to a character */
- char *string = “Hello”;
- printf(“%s”, string);

Two-dimensional Character Arrays Using Pointers

- Since a pointer to a character can be a pointer to a string, it follows therefore that a two-dimensional character array can be declared as an **array of character pointers.**

- Recall the declaration of the two dimensional character array that you used earlier to store 11 names, each of which could be a maximum of 30 characters. It was written as: **char team_india [11][30]**;

Two-dimensional Character Arrays Using Pointers

- This could be alternately declared as:
- **char *team_india[11];**

- **team_india** is now an array of 11 character pointers, each of which in turn can point to a string. A pointer to a character can be a pointer to a string as well.

- The flexibility with a declaration of an array of character pointers is that each of the character pointers may point to an array of unequal length, as against the declaration of a two-dimensional character array in which each string is of a fixed size.

String Handling Functions Using Pointers

- /* The following function determines the length of a string */
- #include <stdio.h>
- main( )
- {
-   char *message = “Virtue Alone Ennobles”;
-   char *p;
-   int count;
-   for (p = message, count = 0, p != ‘\0’, p++)
-    count++;
- printf(The length of the string is %d\n”, count);
- }

String Handling Functions Using Pointers

- /* The following functions compares two strings */
- #include<stdio.h>
- main( )
- {
- char *message1 = “This is a test”;
- char *message2 = “This is not a test”;
- char *p1, *p2;
- for(p1=message1, p2=message2; (*p1 = = *p2) && (*p1 != ‘\0’) && (*p2 != ‘\0’); p1++, p2++)
- if ((*p1 = = ‘\0’) && (*p2 = = ‘\0’))
- printf(“The two strings are identical\n”);
- else
- printf(“The two strings are not identical\n”);
- }

Standard String Handling Functions

- **strcmp( ) – compares two strings (that are passed to it as parameters) character by character using their ASCII values, and returns any of the following integer values.**

Return Value	Implication	Example
Less than 0	ASCII value of the character of the first string is less than the ASCII value of the corresponding character of the second string	i = strcmp("XYZ", "xyz")
Greater than 0	ASCII value of the character of the first string is less than the ASCII value of the corresponding character of the second string	i = strcmp("xyz", "XYZ")
Equal to 0	If the strings are identical	i = strcmp("XYZ", "XYZ")

Standard String Handling Functions

- **strcpy( ) – copies the second string to the first string, both of which are passed to it as arguments.**
- Example: strcpy(string1, “XYZ”) will copy the string “XYZ” to the string string1.
- If string1 were to contain any value, it is overwritten with the value of the second string.

Standard String Handling Functions

- **strcat( ) – appends the second string to the first string, both of which are passed to it as arguments.**

- Example: assume that the string realname contains the value “Edson Arantes Do Nascimento”. If one were to append the nickname “Pele” to the string realname, then it can be done as follows:
  - strcat(“Edson Arantes Do Nascimento”, “Pele”); will give the string “Edson Arantes Do Nascimento Pele”

Standard String Handling Functions

- **strlen( ) – This function returns the length of a string passed to it as an argument. The string terminator, i.e., the null character is not taken into consideration.**

- Example: i = strlen(“Johann Cryuff”); will return the value value 13 into i.

String to Numeric Conversion Functions

- **atoi( )** – This function accepts a string representation of an integer, and returns its integer equivalent.

- Example: i = atoi(“22”) will return the integer value 22 into the integer i.

- This function is especially useful in cases where main is designed to accept numeric values as command line arguments.

- It may be recalled that an integer passed as a command line argument to main( ) is treated as a string, and therefore needs to be converted to its numeric equivalent.

String to Numeric Conversion Functions

- **atof( )** - This function accepts a string representation of a number, and returns its **double** equivalent.

- For example, if the string str contains the value "1234", then the following statement:
- i = atoi(str); will cause i to have the value 1234.000000

- To use the function atof( ) in any function, its prototype must be declared at the beginning of the function where it is used:
- **double atof( )**

Problems

- C Program to Find the Frequency of Characters in a String
- C Program to Find the Number of Vowels, Consonants, Digits and White space in a String
- C Program to Reverse a String by Passing it to Function
- C program to Concatenate Two Strings
- C Program to Copy a String
- C Program to Remove all Characters in a String except alphabet
- C Program to Sort Elements in Lexicographical Order (Dictionary Order)

Passing Arguments to main( )

- Arguments are generally passed to a function at the time of call.
- Since **main( )** is the first function to be executed is it possible to pass arguments to **main( )?**
- Yes, command-line arguments.

Command-Line Arguments

- The function main( ) can receive arguments from the command line.
- Information can be passed to the function main( ) from the operating system prompt as command line arguments.
- The command line arguments are accepted into special parameters to main( ), namely, argc and argv. Their declarations are as follows:
- main(int argc, char * argv[ ])

Command Line Arguments

- **argc** provides a count of the number arguments in the command line.
- **argv** is an array of character pointers of undefined size that can be thought of as an array of pointers to strings.
- Since the element **argv[0]** contains the command itself, the value of **argc** is at least 1.

Command Line Arguments

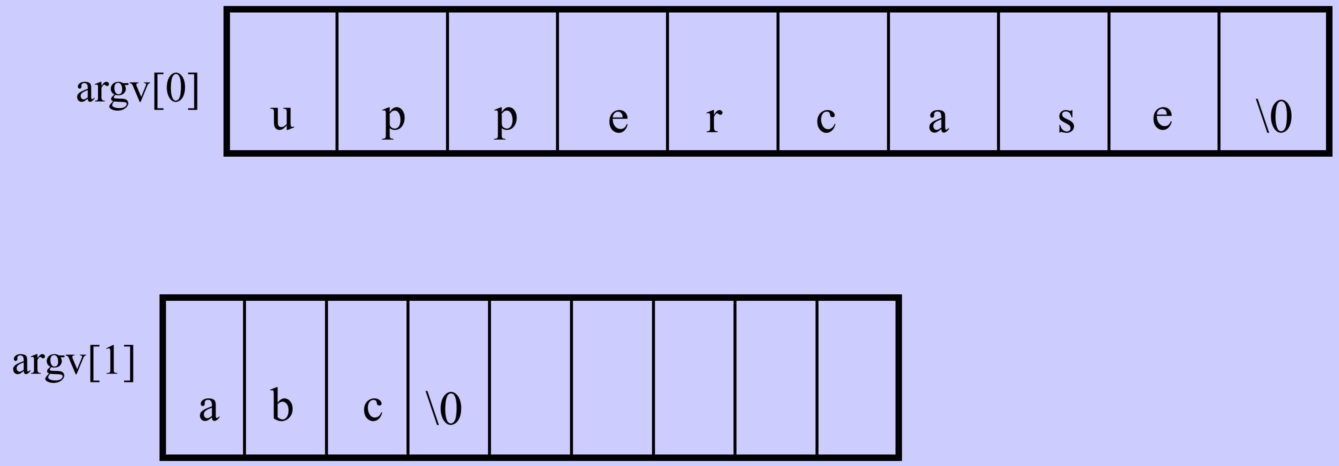
- Consider a program called uppercase which converts a string to uppercase. The program expects the string to be entered at the command prompt.

- It should be noted that if the string accepted as a command line argument has embedded spaces, it should be enclosed in quotation marks.

- Assume the program is run by entering the following command at the operating system prompt:
- Uppercase abc

Command-Line Arguments

argc = 2



Command Line Arguments

- The program uppercase.c can be coded as follows:

```
• #include <stdio.h>
• main(int argc, char * argv[ ])
• {
•   int i;
•   for (i = 0; argv[1][i] != ‘\0’, i++)
•    {
•     if ((argv[1][i] >= ‘a’) && (argv[1][i] <= ‘z’))
•      argv[1][i] = argv[1][i] – 32;
•    }
•    printf( “%s”, argv[1]);
•  }
•
```

Command Line Arguments

- Example : Program illustrating command line arguments

```
void main( int argc, char* argv[])
        {
           int  i;
           printf(“\n Total Number of Arguments = %d”,argc);
           for( i = 0; i < argc; i++)
           printf(“\nArgument number  %d = %s”,i , argv[i]);
   }
```

- Assume that this program is stored in a file test.c and got compiled. Now, in the operating system prompt if we type a command as

```
test  one  two  three  four
```

Command Line Arguments

Then the result will be

```
Number of Arguments  =  5
Argument number  0    =   test
Argument number  1    =   one
Argument number  2    =   two
Argument number  3    =   three
Argument number  4    =   four
```

Function Pointers

- Function Pointers are pointers, i.e. variables, which point to the address of a function.
- You must keep in mind, that a running program gets a certain space in the main-memory.
- Both, the executable compiled program code and the used variables, are put inside this memory.
- Thus a function in the program code is, like e.g. a character field, nothing else than an address.

Function Pointers

- It is only important how you, or better, your compiler/processor, interpret the memory a pointer points to.
- When you want to call a function *fn()* at a certain point in your program, you just put the call of the function *fn()* at that point in your source code.
- Then you compile your code, and every time your program comes to that point, your function is called.
- But what can you do, if you don't know at build-time which function has got to be called? Or, invoke functions at runtime.

Function Pointers

- You want to select one function out of a pool of possible functions.
- However you can also solve the latter problem using a switch-statement, where you call the functions just like you want it, in the different branches.
- **But there's still another way: Use a function pointer!**
- Consider the example on the following slide:

Declaring and Using Function Pointers

- How are function pointers used? As stated above they are typically used so one function can take in other functions as a parameter.

- Consider the following example:
- #include <stdio.h>
- int compute(int, int (*comp)(int));
- int doubleIt(int);
- int tripleIt(int);
- int main()
- {
-    int x;
-    x = compute(5, &doubleIt);
-    printf("The result is: %i\n", x);
-

Declaring and Using Function Pointers

```
•  x = compute( 5, &tripleIt );
•     printf("The result is: %i\n", x );
•     return 0;
•  }

•  int compute( int y, int (*comp)(int) )
•  {   return comp( y );
•  }
•  int doubleIt( int y )
•  {    return y*2;
•  }
•  int tripleIt( int y )
•  {   return y*3;
•  }
```

Declaring and Using Function Pointers

- Aside from **main( )**, this program has 3 functions.
- The functions **doubleIt( )** and **tripleIt( )** are just run of mill functions that take in an integer, either double or triple the value of the argument received, and return the value.
- The function of interest here is **compute( )**. It starts off normal. It returns an int, and the first parameter is an int.
- But the second parameter is pretty strange looking. The whole "**int (comp*)(int)**" is one parameter, which, in fact, is the format of a function pointer.

Declaring and Using Function Pointers

- Basically it says compute( ) wants a pointer to a function that takes one integer parameter, and returns an integer. And it is going to refer to this function through the parameter name **comp.**

- When **compute( )** is called, it uses its parameter **comp** just like a normal function. When a function pointer is passed to a function, it can be used like a normal function.

- When **compute( )** is called in **main( )**, the second parameter is given the name of another function with an **ampersand**.

Declaring and Using Function Pointers

- The **ampersand** is of course, the "**address of**" operator. So this is passing the address of **doubleIt( )** (and pointers are really just addresses). So this is how **compute( )** gets the pointer to **doubleIt( )**
- It is important to note however, that if the return type of the function **doubleIt( )** and it's parameter list did not match the ones for the function pointer **comp**, it could not be used.
- When you pass a function pointer, the functions header must match the header of the function pointer definition **exactly**, or it cannot be used.

Function Pointers

- You must keep in mind, that a running program gets a certain space in the main-memory.
- Like variables we can assign the address of a function to a pointer is called function pointer.
- Function Pointers are pointers, i.e. variables, which point to the address of a function.
- The definition of a pointer to a function must be same like the function prototype.
- **void (*fp)(void *);** same as free()
  - fp=free;
  - fp(a); or free(a);

Function Pointers

- int func(int, float);
- int (*fp)(int , float);

- fp=func;
- fp(3,4.5); or (*fp)(3,4.5);
- func(3,4.5);

Void Pointers

- There are times when you write a function but do not know the datatype of the returned value. When this is the case, you can use a void pointer.
- When declaring a pointer, If we don't know what kind of value a pointer will point to then we can declare a "void pointer".
- By declaring a void pointer, you are telling the compiler not to do error checking on the type of the pointer.
- In order to dereference the pointer later, you must type cast it into a known type so that the computer knows how much memory to read and how to interpret the data there.

Void Pointers

- void *calloc(size_t num_elements, size_t element_size};
- void *realloc(void *ptr, size_t new_size);
- void *malloc(size_t number_of_byte);
- void free(void *);

User-Defined Data Types

Structures – The Definition

- A structure is a collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling.
- Structures help to organize complicated data, particularly in large programs, because they permit a group of related variables to be treated as a unit instead of as separate entities.
- An example of a structure is the payroll record: an employee is described by a set of attributes such as name, address, social security number, salary, etc.

Structures – The Definition

- Some of these in turn could be structures: a name has several components, as does an address.
- Other examples of structures are: a point is a pair of coordinates, a rectangle is a pair of points, and so on.
- The main change made by the ANSI standard is to define structure assignment - structures may be copied and assigned to, passed to functions, and returned by functions.
- Automatic structures and arrays may now also be initialized.

Structures – Defining a Type

- When we declare a structure, we are **defining a type**.
- A structure declaration results in the definition of a **template or a blueprint for a user-defined data type**.
- Upon declaring a structure, the compiler identifies the structure declaration as a user-defined data type over and above the fundamental data types, or primitive data types built into the compiler.
- **A structure therefore is a mechanism for the extension of the type mechanism in the C language.**

Declaring a Structure

- The C language provides the struct keyword for declaring a structure. The following is a structure declaration for employee attributes.

- **struct empdata {**
- **int empno;**
- **char name[10];**
- **char job[10];**
- **float salary;**
- **};**

Structures

Declaring a Structure and its Variables

The syntax for declaring a structure is as follows

```
        struct  <struct-name> // the struct name is optional
    {
         type variable-name, variable-name,........;
         type variable-name, variable-name,........;
         type variable-name, variable-name,........;
            :
            :
         type variable-name, variable-name,........;
    }struct-variable, struct-variable...... ;
```

Declaring a Structure

- The keyword **struct** introduces a structure declaration, which is a list of declarations enclosed in braces.
- An optional name called a structure tag may follow the word struct, as with employee in the previous example.
- The tag names this kind of structure, and can be used subsequently as a shorthand for the part of the declaration in braces.
- **A struct declaration defines a type.**

Declaring a Structure - Conventions

- Furthermore, the same member names may occur in different structures, although as a matter of style one would normally use the same names only for closely related structure variables.

- Variables of a structure type may immediately follow the structure declaration, or may be defined separately as follows:

Declaring a Structure Variable

- **struct {**
- **int empno;**
- **char name[10];**
- **char job[10];**
- **float salary;**
- **} emprec1, emprec2;**

Declaring a Structure Variable

- **struct empdata {**
- **int empno;**
- **char name[10];**
- **char job[10];**
- **float salary;**
- **} emprec; /* emprec is a variable of structure type empdata */**

- **Or a structure can be declared separately as:**
- **struct empdata emprec;/* emprec is a variable of structure type empdata */**

Declaring a Structure

- Declaring a structure only defines a template or a blueprint for a data type. It does not therefore result in the allocation of memory.
- Memory is allocated only when a variable of a structure type is declared in the program.
- Till a variable of a structure type is created, a structure declaration is only identified as a type, and no memory is allocated.

Accessing Elements of a Structure

```
struct  employee_type
        {       int code;
                char name[20];
                int dept_code;
                float salary;
        } emp1={10, ``xyz”, 30, 1234.56}, emp2;
emp2.code=emp1.code;
strcpy(emp2.name,emp1.name);
emp2.dept_code=emp1.dept_code;
emp2.salary=emp1.salary;
emp2=emp1;
```

A sample program

- struct point
- {
- int x,y;
- }p1,p2;
- main()
- {double dist;
- scanf("%d%d%d%d\n",&p1.x,&p1.y,&p2.x,&p2.y);
- dist=sqrt((p1.x-p2.x)*(p1.x-p2.x) + (p1.y-p2.y)*(p1.y-p2.y));
- printf("distance between the given points is %lf\n", dist);
- }

The period(.) operator

- The period(.) operator is a member of the highest precedence group
- It will take precedence over the unary operators as well as various arithmetic, relational, logical and assignment operators
- Thus the expression ++variable.member is equivalent to ++(variable.member), i.e, the ++ operator will apply to the structure member, not the entire structure variable
- Similarly, the expression &variable.member stands for the address of the structure member, not the starting address of the structure variable

Passing Structures to Functions

- struct point
- {
- int x,y;
- }p1,p2;
- float distance(struct point, struct point);
- main()
- {double dist;
- scanf("%d%d%d%d\n",&p1.x,&p1.y,&p2.x,&p2.y);
- dist=distance(p1,p2);
- printf("distance between the given points is %lf\n", dist);
- }
- float distance(sturct point p1, sturct point p2)
- {
- return (sqrt((p1.x-p2.x)*(p1.x-p2.x) + (p1.y-p2.y)*(p1.y-p2.y)));
- }

Pointers and Structures

- As we can use pointers with other types, we can also use pointers for structure variables. We can declare pointer to a student structure as follows

```
struct  stud_type  *ptr;
```

- In this declaration, ptr is a pointer type variable, which can hold an address of a variable of the type stud_type.

Pointers and Structures

```
        struct  stud_type
         {
        int rollnum;
        char name[20];
        int semester;
        float avg;
                  };
 struct stud_type  stud={1001,"Aldina", 1, 98.23}, *ptr ;

 ptr = &stud;

(*pointer).field        (OR)        pointer -> field
```

So, We can access the fields of a stud through a pointer ptr as follows

```
printf(" %d\t %s\t%d\t%f ", ptr->rollnum, ptr->name, ptr->sem, ptr->avg);
```

Pointers and Structures

- A structure can include one or more pointers as member

```
main()
{
int n 1234, d=11; float s = 5152.5;
struct employee {
        int *code;
        char *name;
        int * dept_code;
        float *salary;
        }emp, *p = &emp;
```

Pointers and Structures

Check this with
*(p->code)

```
emp.code = &n;
emp.name ="Anup";
emp.dept_code = &d;
emp.salary = &s;
```

To access the members of the structure emp you have to use

```
*emp.code, emp.name, *emp.dept_code, *emp.salary
                or
*p->code, p->name, *p->dept_code, *p->salary
```

Passing Structures to Functions

```
#include<stdio.h>
struct salesdata
{
  int transaction_no;
  int salesman_no;
  int product_no;
  int units_sold;
  float value_of_sale;
 };
```

```
main( )
{   struct salesdata salesvar;
printf(“enter transaction number :”);
scanf(“%d”, &salesvar.transaction_no);
printf(“enter salesman number :”);
scanf(“%d”, &salesvar.salesman_no);
printf(“enter product number :”);
scanf(“%d”, &salesvar.product_no);
printf(“enter units sold :”);
scanf(“%d”, &salesvar.units_sold);
compute(&salesvar);
}
```

Passing Structures to Functions

- compute(salesdata *ptr)
- {
-   static float prod_unit_price[] = {10.0, 20.0, 30.0, 40.0};
-   ptr-> value_of_sale = (float)ptr-> units_sold * product_unit_price[ptr->product_no – 1];
- }

Array of Structures

- Just as it is possible to declare arrays of primitive data types, it should also be possible to declare arrays of structures as well.

- Consider the structure declaration for the employee details used earlier.
- **struct empdata {**
- **int empno;**
- **char name[10];**
- **char job[10];**
- **float salary;**
- **};**

Array of Structures

- If one were to define an array of structure variables, one would do so as follows:
- struct empdata employee_array[4];
- The rationale for declaring an array of structures becomes clear when one wants to improve I/O efficiency in a program.
- Once an array of structures is defined, it is possible to read in a block of records from a file using an appropriate function (fread( )), the details of which you will see shortly.

Nested Structure

- Example

```
struct  date
{
    int day;
    int month;
    int year;
};
```

```
struct employee_type
{
    int code;
    char name[20];
    struct date doj;
    int dept_code;
    float salary;
}emp1,emp2;
```

- In this example, if we want to access the year of joining of an employee of emp1, then we can do so by writing emp1.doj.year

Problems

- Store student records {name, roll, marks of sub1, sub2 …, subn} and rank the students according to their percentage etc.
- address

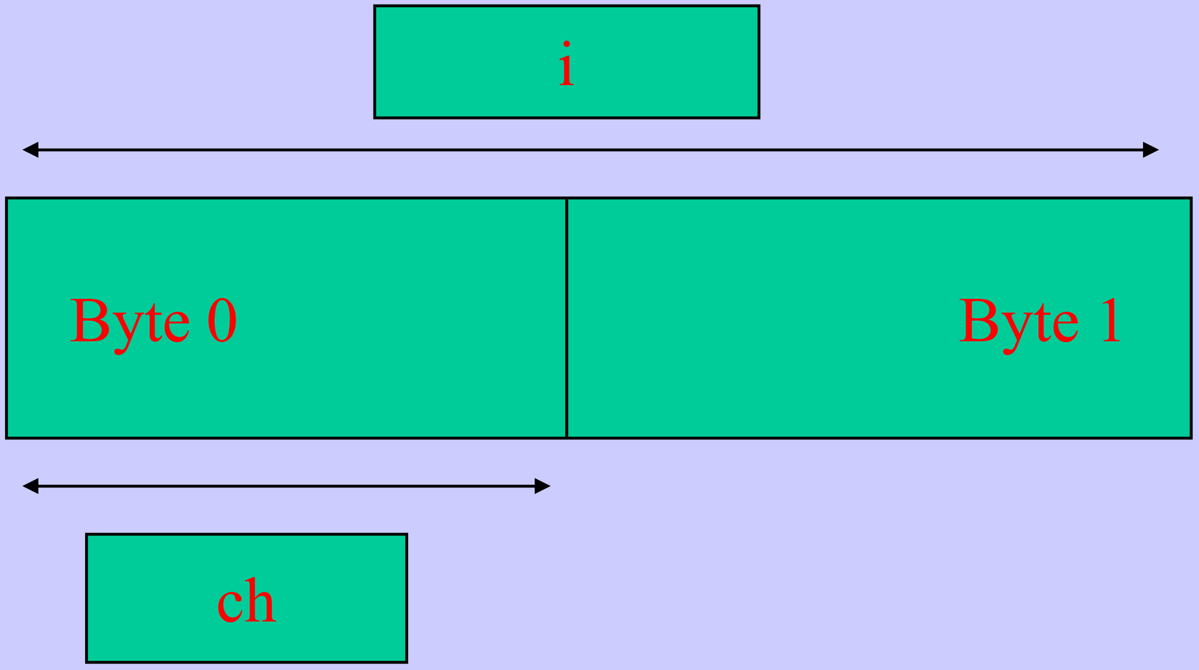
Union

- Can hold objects of different types and sizes at different times
- Syntax similar to structure but meaning is different
- All members of ***union*** share same storage space
- Only the last data member defined can be accessed
- Means of conserving memory
- ***union*** declaration similar to *struct* declaration

```
eg. union u_type {
          int i;
          char  ch;
        };
        union u_type cnvt;
```

Unions

- In cnvt, both integer i and character ch share the same memory location. Of course, **i** occupies 2 bytes (assuming 2-byte integers, and **ch** uses only one byte.



Unions

- To access a member of a union, use the same syntax that you would use for structures: the dot and arrow operators.
- If you are operating on the union directly, use the dot operator. If the union is accessed through a pointer, use the arrow operator.
- For example, to assign the integer 10 to element i of cnvt, write cnvt.i = 10;

Unions

- In the following code snippet, a pointer to cnvt is passed to a function:
- void func1(union u_type *un)
- {
- un->i = 10; /* assign 10 to cnvt using function */
- }

- Using a union can aid in the production of machine-independent (portable) code. Because the compiler keeps track of the actual size of the union members, no unnecessary machine dependencies are produced.

Unions

- Unions are used frequently when specialized type conversions are needed because you can refer to the data held in the union in fundamentally different ways.
- Consider the problem of writing a short integer to a disk file. The C standard library defines no function specifically designed to write a short integer to a file.
- While you can write any type of data to a file using fwrite( ), using fwrite( ) incurs excessive overhead for such a simple operation.

Unions

- However, using a union, you can easily create a function called putw( ), which represents the binary representation of a short integer to a file one byte at a time.

- The following example assumes that short integers are 2 bytes long. First, create a union consisting of one short integer and a 2-byte character array:
- union pw
- {
- short int i;
- char ch[2];
- };

Union

- union { int a; float b; char c;} union_var=97;
- By default the first variable (**a**) is initialized.

- union_var.b=99.99;
- union_var.a=34;
- union_var.c='x';

- It's important to note that the storage will only hold ONE value, looking at the three lines above, **union_var.a** overwrites **union_var.b** and then **union_var.c** overwrites **union_var.a**

Conversion using Union

- union conv{
- int a;
- float b;
- char c;
- };
- main()
- {
- union conv data;
- data.a=100;
- printf(“%f “,data.b);
- printf(“%c”,data.c);
- }

Size of Union

- #include<stdio.h>
- main()
- {
- union test
- {
- char name[30];
- int roll,s1,s2,s3,total;
- char grade;
- float percentage;
- int *ptr;
- }st1,*st2;
- printf("size of union is %d\n",sizeof(st1));}

- Output :
- size of union is 32
- it depends on the alignment that the compiler puts into the allocated space. So, unless you use some special option, the compiler will put padding into your union space.

Size of Union

- #include<stdio.h>
- main()
- {
- #pragma pack(push, 1)
- union test
- {
- char name[30];
- int roll,s1,s2,s3,total;
- char grade;
- float percentage;
- int *ptr;
- }st1,*st2;
- #pragma pack(pop)
- printf("size of union is %d\n",sizeof(st1));}

- Why do we need #pragma

- Output :
- size of union is 30

Unions

- Now, you can use pw to create the version of putw( ) shown in the following program:
- #include <stdio.h>
- union pw
- {
- short int i;
- char ch[2];
- };

- int putw(short int num, FILE *fp);

Unions

- int main (void)
- {
-    FILE *fp;
-    fp = fopen(“test.tmp”, “wb+”);
-    putw(1000, fp); /* write the value 1000 as an integer */
-    fclose(fp);
-    return 0;
- }

Unions

- int putw(short int num, FILE *fp)
- {
-   union pw word;
-   word.i = num;
-   fputc(word.ch[0], fp); /* write first half */
-   fputc(word.ch[1], fp); /* write second half */
- }

- Although putw( ) is called with a short integer, it can still use the standard function fputc( ) to write each byte in the integer to a disk file one byte at a time.

Bit Fields

```
#include <stdio.h>
#include <string.h>
struct
{
unsigned int age : 3;
} Age;
int main( )
{
Age.age = 4;
printf( "Sizeof( Age ) : %d\n", sizeof(Age) );
printf( "Age.age : %d\n", Age.age );
Age.age = 7;
```

Typedef Statements

- Creates synonyms (aliases) for previously defined datatypes
- Used to create shorter type names
- Format: *typedef* type new-type;
  - Example: *typedef struct* Card * CardPtr;

    defines a new type name CardPtr as a synonym for type *struct* Card *
- *typedef* does not create a new datatype
    - Only creates an alias

Typedef Statements

- main()
- { int money;
- money = 2;
- }

- main()

{

typedef int Pounds;

Pounds money = 2

}

- typedef enum {FALSE=0, TRUE} Boolean
- main ()
- { Boolean flag = TRUE;
- }

- typedef char *String;
- main()
- { String Text = "Thunderbird";
- printf("%s\n", Text);
- }

Typedef Statements

- typedef struct
- {
- int age;
- char *name
- } person;
- person people;

- typedef int Pounds, Shillings, Pennies, Dollars, Cents;

Enumeration

- Is a set of named integer constants that specify all the legal values a variable of that type can have.

- The keyword ***enum*** signals the start of an enumeration type.

- The general form for enumeration is
- enum enum-type-name { enumeration list } variable_list;

- enum coin { penny, nickel, dime, quarter, half_dollar, dollar};
- enum coin money;

Enumeration

- Given these declarations, the following types of statements are perfectly valid:
- money = dime;
- if (money = = quarter)
- printf(“Money is a quarter. \n”);

- The key point to understand about an enumeration is that each of the symbols stands for an integer value.

- As such, they may be used anywhere that an integer may be used.
- You can initialize the value of one or more of the symbols.

Enumeration

- Each symbol is given a value one greater than the symbol that precedes it. The value of the first enumeration symbol is 0. Therefore,
- printf(“%d %d”, penny, dime);
- displays 0 2 on the screen.

- You can specify the value of one or more of the symbols by using an initializer.

- Do this by following the symbol with an equal sign and an integer value.

Enumeration

- For example, the following code assigns the value of 100 to quarter:
- enum coin { penny, nickel, dime, quarter=100, half_dollar, dollar};
- Now, the values of these symbols are:

• penny	0
• nickel	1
• dime	2
• quarter	100
• half_dollar	101
• dollar	102

Enumeration

- One common but erroneous assumption about enumerations is that the symbols can be input and output directly. This is not the case.

- For example, the following code fragment will not perform as desired:
- money = dollar;
- printf(“%s”, money);

- Dollar is simply a name for an integer; it is not a string.

Enumeration

- For the same reason, you cannot use this code to achieve the desired results:
- /* this code is wrong */
- strcpy (money, “dime”);

- That is, a string that contains the name of a symbol is not automatically converted to that symbol.

- Actually creating code to input and output enumeration symbols is quite tedious (unless you are willing to settle for their integer values).

Enumeration

- For example, you need the following code to display, in words, the kind of coins that money contains:
- switch (money)
- {
-   case penny : printf(“penny”);
-                       break;
-   case nickel : printf(“nickel”);
-                       break;
-   case dime : printf(“dime”);
-                       break;
-   case quarter : printf(“quarter”);
-                       break;
-   case half_dollar : printf(“half_dollar”);
-                       break;
-   case dollar : printf(“dollar”);
-                       break;
- }

Enumeration

- enum DAY /* Defines an enumeration type */
- { saturday, /* Names day and declares a */
- sunday = 0, /* variable named workday with */
- monday, /* that type */
- tuesday,
- wednesday, /* wednesday is associated with 3 */
- thursday,
- friday } workday;

C Preprocessor

The C Preprocessor

- The ``preprocessor'' (macro processor) is a translation phase that is applied to your source code before the compiler proper gets its hands on it. Each preprocessor directive is identified by the # at the beginning of a line.
- **File inclusion:** inserting the contents of another file into your source file, as if you had typed it all in there.
- **Macro substitution**: replacing instances of one piece of text with another.
- **Conditional compilation**: Arranging that, depending on various circumstances, certain parts of your source code are seen or not seen by the compiler at all.

File inclusion

Examples:

#include <stdio.h>

This variant is used for system header files. It searches for a file named *file* in a standard list of system directories.

/usr/local/include

/usr/lib/gcc-lib/*target*/*version*/include

/usr/*target*/include

/usr/include

- #include "allconstants.h"

  This variant is used for header files of your own program. It searches for a file named file first in the directory containing the current file, then in the same directories used for <file>.

Why file inclusion

- The size of the program becomes very large, break it into a files.
- Some functions are widely used, put them in some file and include these files.

Macro Substitution

- A *macro* is a fragment of code which has been given a name.
- Macros are defined by **#define**
- Whenever the name is used, it is replaced by the contents of the macro.
- There are two kinds of macros.
  - *Object-like* macros
  - *function-like* macros

Object-like macros

- `#define' is followed by the name of the macro and then the token sequence
- #define BUFFER_SIZE 1024
- foo = (char *) malloc (BUFFER_SIZE);
  - foo = (char *) malloc (1024);
- The macro's body ends at the end of the `#define' line.
- You may continue the definition onto multiple lines, if necessary, using backslash-newline.
-
```
#define NUMBERS 1, \
                2, \
                3
```

```
int x[] = { NUMBERS };  =>  int x[] = { 1, 2, 3 };
```

Object-like macros

- #define TABLESIZE BUFSIZE
- #define BUFSIZE 1024

- TABLESIZE ==> BUFSIZE ==> 1024

The Preprocessor Phase

```
#include <stdio.h>
•   #define RG 3
•   #define PR 3
•   main( )
•   {
•      int r_counter, p_counter, rp_array[RG][PR], total_sales = 0;
•      float rp_array_perc[RG][PR];
•      /* initialization of rp_array using the for loop */
•     for (r_counter = 0; r_counter < RG; r_counter ++)
•     {
•      for (p_counter = 0; p_counter < PR; p_counter ++)
•       {
•         rp_array[r_counter][p_counter] = 0;
•       }
•     }
```

Function-like Macros

- #define min(X, Y) ((X) < (Y) ? (X) : (Y))
- x = min(a, b);
  - x = ((a) < (b) ? (a) : (b));
- y = min(1, 2);
  - y = ((1) < (2) ? (1) : (2));
- z = min(a + 28, *p);
  - z = ((a + 28) < (*p) ? (a + 28) : (*p));

Function-like Macros

- Macro can be used as function
  - Then which one will be used macro function or ordinary function?

Conditional compilation

- A *conditional* is a directive that instructs the preprocessor to select whether or not to include a chunk of code in the final token stream passed to the compiler.
- Why conditional preprocessor?
- A program may need to use different code depending on the machine or operating system it is to run on. In some cases the code for one operating system may be erroneous on another operating system
- A conditional whose condition is always false is one way to exclude code from the program but keep it as a sort of comment for future reference.

Enumeration

- #define MON 1
- main()
- {
- enum days {Jan=31, Feb=28, Mar=31, Apr=30, May=31, Jun=30, Jul=31, Aug=31, Sep=30, Oct=31, Nov=30, Dec=31};
- enum days month;
- printf("%d\n", month=Feb); / * This will return 28 */
- }

- An advantage of enum over #define is that it has scope; this means that the variable (just like any other) is only visible within the block it was declared within.

enum coding error

- enum People1 {Alex=0, Tracy, Kristian} Girls;
- enum People2 {William=0, Martin, Alex} Boys;

- #define FALSE 1
- main() {
- enum Boolean_t {FALSE=0, TRUE} Boolean; // 1 =0
- printf("False has a value of %d", FALSE);
- printf(" True has a value of %d", TRUE);
- }

enum and #define coding error

- enum Boolean_t {FALSE=0, TRUE} Boolean;
- #define FALSE 1
- main()
- {printf("False has a value of %d\n", FALSE);
- printf(" True has a value of %d\n", TRUE);
- }

- Results:

```
False has a value of 1

True has a value of 1
```

Conditional compilation

A conditional in the C preprocessor begins with a *conditional directive*: `#if', `#ifdef' or `#ifndef‘

- #ifdef *MACRO*
- *controlled text*
- #endif // end

- `#endif' always matches the nearest `#ifdef' (or `#ifndef', or `#if').
- #ifdef *MACRO*
- *controlled text*
- #esle
- #endif // end

- #ifndef *MACRO*
- *controlled text*
- #endif // end

- #ifndef *MACRO*
- *controlled text*
- #esle
- #endif // end

Conditional compilation

- #if *expression*
- *controlled text*
- #endif
- *expression* is a C expression of integer type,

- #if *expression*
- *text-if-true*
- #else
- *text-if-false*
- #endif /*

Conditional compilation

- #if X == 1
-
- #else   /* X != 1 */
-         #if X == 2
-         #else /* X != 2 */
-         #endif /* X != 2 */
- #endif /* X != 1 */

- #if X == 1
-    #elif X == 2
-
-            #else /* X != 2 and X != 1*/
- #endif /* X != 2 and X != 1*/
-

File Input/Output

Files

- A collection of logically related information
- Examples:
  - An employee file with employee names, designation, salary etc.
  - A product file containing product name, make, batch, price etc.
  - A census file containing house number, names of the members, age, sex, employment status, children etc.
- Two types:
  - Sequential file: All records are arranged in a particular order
  - Random Access file: Files are accessed at random

File Access

- The simplicity of file input/output in C lies in the fact that it essentially treats a file as a stream of characters, and accordingly facilitates input/output in streams of characters.
- Functions are available for character-based input/output, as well as string-based input/output from/to files.
- In C, there is no concept of a sequential or an indexed file. This simplicity has the advantage that the programmer can read and write to a file at any arbitrary position.

File Access

- The examples so far have involved reading from standard input, and writing to standard output, which is the standard environment provided by the operating system for all applications.
- You will now look at the process that a program needs to follow in order to access a file that is not already connected to the program.
- Before a file can be read from, or written into, a file has to be opened using the library function **fopen( )**

File Access

- The function fopen( ) takes a file name as an argument, and interacts with the operating system for accessing the necessary file, and returns a pointer of type FILE to be used in subsequent input/output operations on that file.

- This pointer, called the file pointer, points to a structure that contains information about the file, such as the location of a buffer, the current character position in the buffer, whether the file is being read or written, and whether errors, or end of file have occurred.

File Access

- This structure to which the file pointer point to, is of type FILE, defined in the header file <stdio.h>.

- The only declaration needed for a file pointer is exemplified by:
- FILE *fp;
- FILE *fopen(char *name, char *mode);
- fp = fopen(“file name”, “mode”);

- Once the function fopen( ) returns a FILE type pointer stored in a pointer of type FILE, this pointer becomes the medium through which all subsequent I/O can be performed.

File Access Modes

- When opening a file using fopen( ), one also needs to mention the mode in which the file is to be operated on. C allows a number of modes in which a file can be opened.

Mode	Access	Explanation
“r”	Read only mode	Opening a file in “r” mode only allows data to be read from the file
“w”	Write only mode	The “w” mode creates an empty file for output. If a file by the name already exists, it is deleted. Data can only be written to the file, and not read from it

File Access Modes

Mode	**Access**	**Explanation**
“a”	Append mode	Allows data to be appended to the end of the file, without erasing the existing contents. It is therefore useful for adding data to existing data files.
“r+”	Read + Write mode	This mode allows data to be updated in a file
“w+”	Write + Read mode	This mode works similarly to the “w” mode, except that it allows data to be read back after it is written.
“a+”	Read + Append mode	This mode allows existing data to be read, and new data to be added to the end of the file.

Character-based File I/O

- In C, character-based input/output from and to files is facilitated through the functions **fgetc( )** and **fputc( )**.
- These functions are simple extensions of the corresponding functions for input/output from/to the terminal.
- The only additional argument for both functions is the appropriate file pointer, through which these functions perform input/output from/to these files.

A File Copy Program

- #include<stdio.h>
- main( )
- {
-   FILE *fp1, *fp2;
-   fp1 = fopen(“source.dat”, “r”);
-   fp2 = fopen(“target.dat”, “w”);
-   char ch;
-   while ((ch = fgetc(fp1)) != EOF)
-    {
-      fputc (ch, fp2);
-    }
- fclose(fp1);
- fclose(fp2);
- }

Variation to Console-Based I/O

```
• #include<stdio.h>
• main( )
• {
•  char ch;
•   while ( (ch = fgetc( stdin)) != EOF)
•    {
•      fputc (ch, stdout);
•    }
• }
```

Nuggets on FILE Type Pointers

- The important point to note is that in C, devices are also treated as files. So, the keyboard and the VDU are also treated as files.

- It would be interesting here to know what **stdin, stdout, and stderr** actually are.

- **stdin**, **stdout**, and **stderr** are pointers of type **FILE** defined in **stdio.h**. **stdin** is a **FILE** type pointer to standard input, **stdout** is a **FILE** type pointer to standard output, and **stderr** is a **FILE** type pointer to the standard error device.

Nuggets on FILE Type Pointers

- In case fopen( ) is unsuccessful in opening a file (file may not exist, or has become corrupt), it returns a null (zero) value called NULL.

- NULL is defined in the header file stdio.h

- The NULL return value of fopen( ) can be used for error checking as in the following line of code:
- if ((fp = fopen(“a.dat”, “r”)) = = NULL)
- printf(“Error Message”);

The exit( ) function

- The **exit( )** function is generally used in conjunction with checking the return value of the **fopen( )** statement.

- If **fopen( )** returns a **NULL**, a corresponding error message can be printed, and program execution can be terminated gracefully using the **exit( )** function.

- if ((fp = fopen(“a.dat”, “r”)) = = NULL)
- {
-   printf(“Error Message”);
-   exit( );
- }

Line Input/Output With Files

- C provides the functions **fgets( )** and **fputs( )** for performing line input/output from/to files.

- The prototype declaration for fgets( ) is given below:
- **char* fgets(char *line, int maxline, FILE *fp);**

- The explanations to the parameters of fgets( ) is:
  - char* line – the string into which data from the file is to be read
  - int maxline – the maximum length of the line in the file from which data is being read
  - FILE *fp – is the file pointer to the file from which data is being read

Line Input/Output With Files

- **fgets( )** reads the next input line (including the newline) from file fp into the character array **line**;

- At most **maxline-1** characters will be read. The resulting line is terminated with **'\0'**.

- Normally **fgets( )** returns line; on end of file, or error it returns **NULL**.

Line Input/Output With Files

- For output, the function **fputs( )** writes a string (which need not contain a newline) to a file:

  – **int fputs(char *line, FILE *fp)**

- It returns **EOF** if an error occurs, and non-negative otherwise.

File Copy Program Using Line I/O

```
•   #define MAX 81;
•   #include<stdio.h>
•   main( )
•   {
•     FILE *fp1, *fp2;
•     fp1 = fopen( “source.dat”, “r”);
•     fp2 = fopen( “target.dat”, “w”);
•     char string[MAX];
•     while ( (fgets(string, MAX, fp1)) != NULL)
•     {
•        fputs (string, fp2);
•      }
•   fclose(fp1);
•   fclose(fp2);
•   }
```

Formatted File Input/Output

- C facilitates data to be stored in a file in a format of your choice. You can read and write data from/to a file in a formatted manner using **fscanf( )** and **fprintf( )**.

- Apart from receiving as the first argument the format specification (which governs the way data is read/written to/from a file), these functions also need the file pointer as a second argument.

- fscanf( ) returns the value EOF upon encountering end-of-file.

Formatted File Input/Output

- fscanf( ) assumes the field separator to be any white space character, i.e., a space, a tab, or a newline character.

- The statement printf(“The test value is %d”, i); can be rewritten using the function fprintf( ) as:
  - **fprintf(stdout, “The test value is %d”, x);**

- The statement scanf(“%6s%d, string, &i) can be rewritten using the function fscanf( ) as:
  - **fscanf(stdin, “%6s%d”, string, &i);**

Random Access

- Input from, or output to a file is effective relative to a position in the file known as the current position in the file.
- For example, when a file is opened for input, the current position in the file from which input takes place is the beginning of the file.
- If, after opening the file, the first input operation results in ten bytes being read from the file, the current position in the file from which the next input operation will take place is from the eleventh byte position.

Random Access

- It is therefore clear that input or output from a file results in a shift in the current position in the file.
- The current position in a file is the next byte position from where data will be read from in an input operation, or written to in an output operation.
- The current position advances by the number of bytes read or written.
- A current position beyond the last byte in the file indicates end of file.

Random Access

- For example, when a file is opened for input, the current position in the file from which input takes place is the beginning of the file.

- If, after opening the file, the first input operation results in ten bytes being read from the file, the current position in the file from which the next input operation will take place is from the eleventh byte position.

- This is sequential access, in which a file is opened, and you start reading bytes from the file sequentially till end of file. The same argument cab be extended to sequential write.

Random Access

- In sharp contrast to sequential access is random access that involves reading from any arbitrary position in the file, or writing to any arbitrary position in the file.
- Random access therefore mandates that we must have a mechanism for positioning the current position in the file to any arbitrary position in the file for performing input or output.
- To facilitate this, C provides the fseek( ) function, the prototype of which is as follows:

The fseek( ) Function

- The function **fseek( )** is used for repositioning the current position in a file opened by the function **fopen( )**.

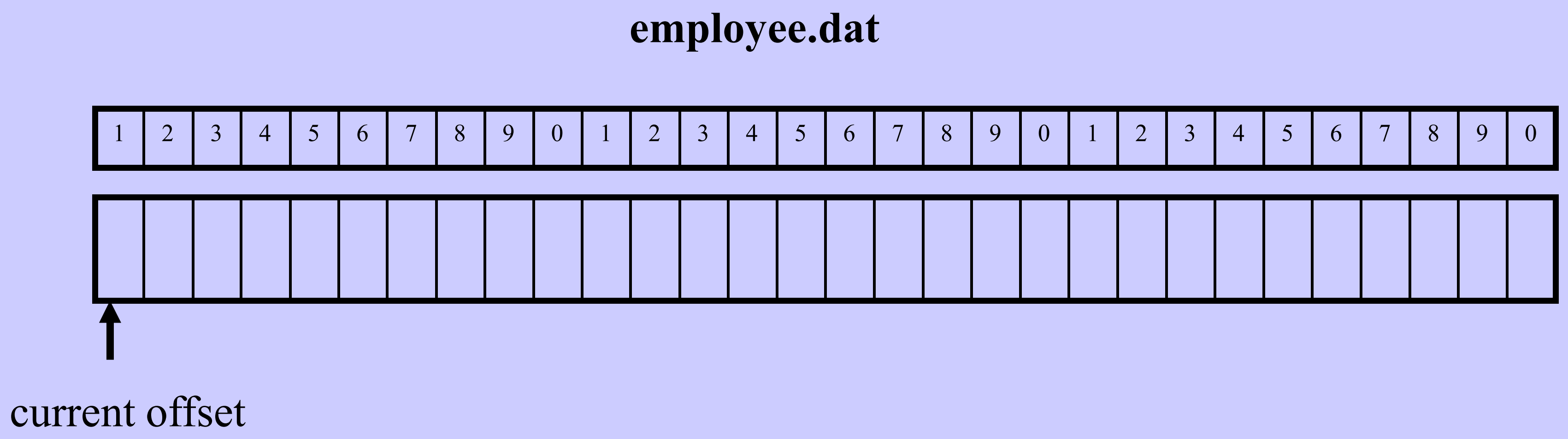
- **int rtn = fseek(file pointer, offset, from where)**
- where,
- **int rtn** is the value returned by the function fseek( ). fseek( ) returns the value 0 if successful, and 1 if unsuccessful.
- **FILE file-pointer** is the pointer to the file
- **long offset** is the number of bytes to be shifted from the position indicated by ‘from where’
- **int from-where** can have one of three values:

from beginning of file (represented as 0)	SEEK_SET
from current position (represented as 1)	SEEK_CUR
from end of file (represented as 2)	SEEK_END

The fseek( ) function

- Consider the following schematic representation of a file in terms of a string of 30 bytes. The file contains records and fields that are not delimited by any special character.

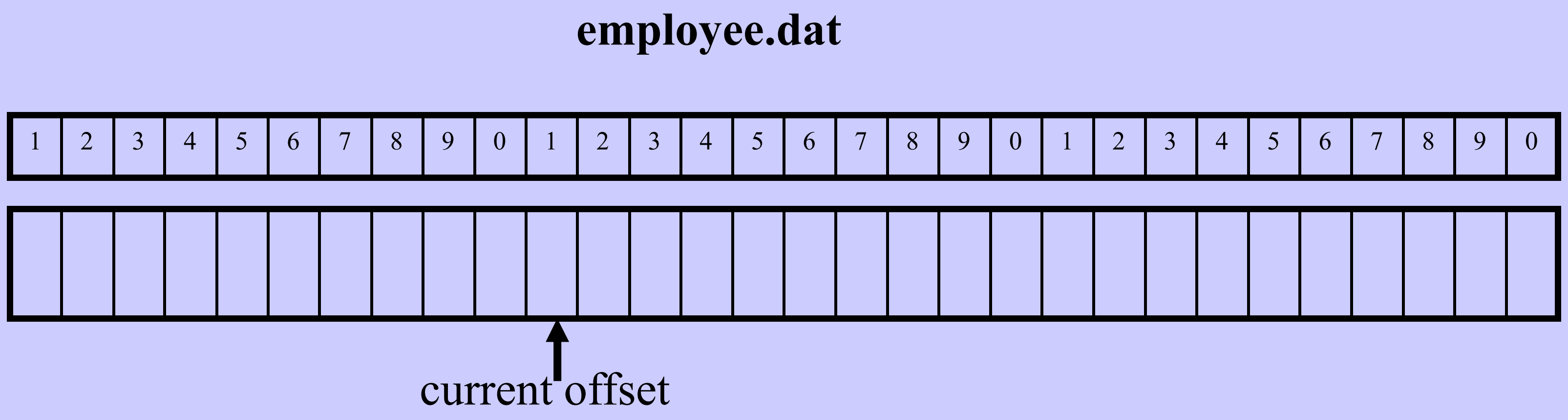
- fp = fopen(“employee.dat”, r)



The fseek( ) function

- Consider the following schematic representation of a file in terms of a string of 30 bytes. The file contains records and fields that are not delimited by any special character.

- fseek(fp, 10L, 0);

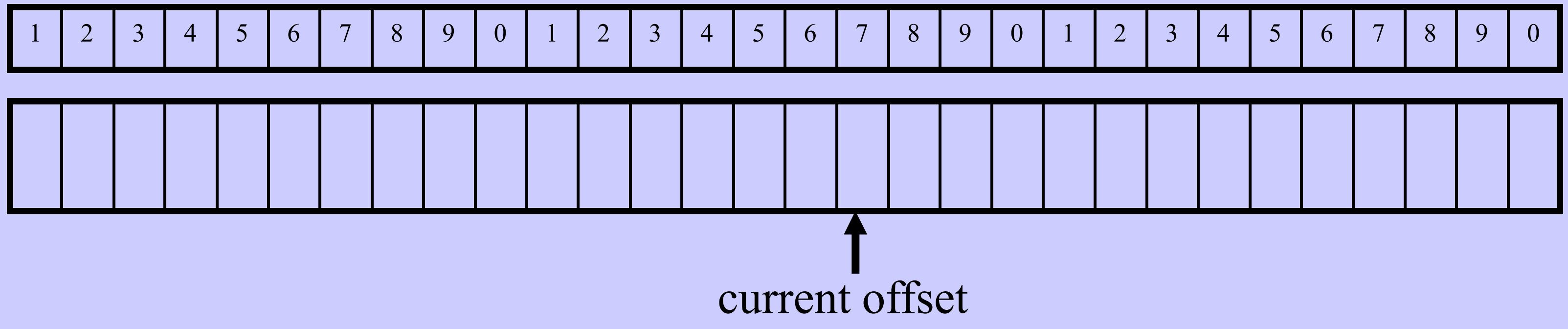


The fseek( ) function

- Consider the following schematic representation of a file in terms of a string of 30 bytes. The file contains records and fields that are not delimited by any special character.

- fgets(string, 7, fp);

employee.dat

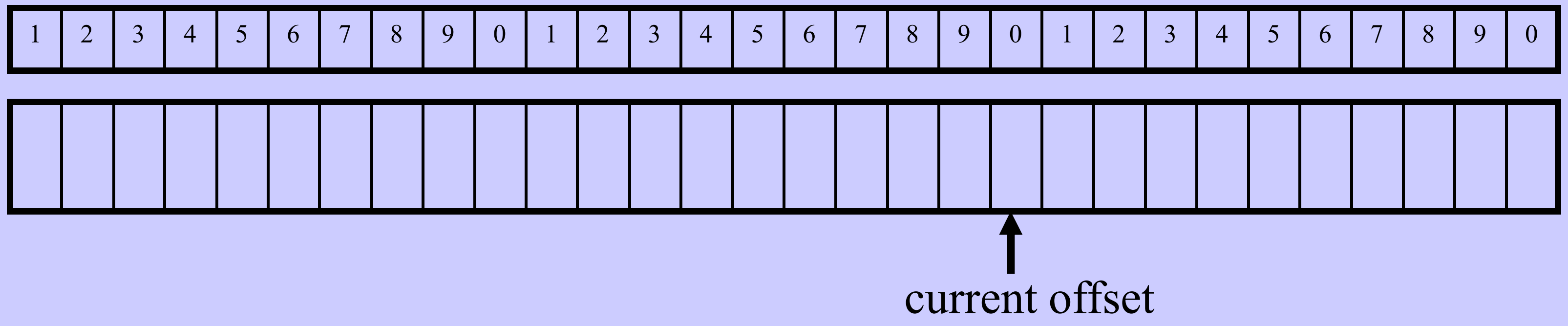


The fseek( ) function

- Consider the following schematic representation of a file in terms of a string of 30 bytes. The file contains records and fields that are not delimited by any special character.

- fseek(fp, -10L, 2)

employee.dat



The rewind( ) Function

- The function, rewind( ) is used to reposition the current position in the file (wherever it may be) to the beginning of the file.

- The syntax of the function rewind( ) is:
  - **rewind (file-pointer);**
  - **where file-pointer is the FILE type pointer returned by the function fopen( ).**

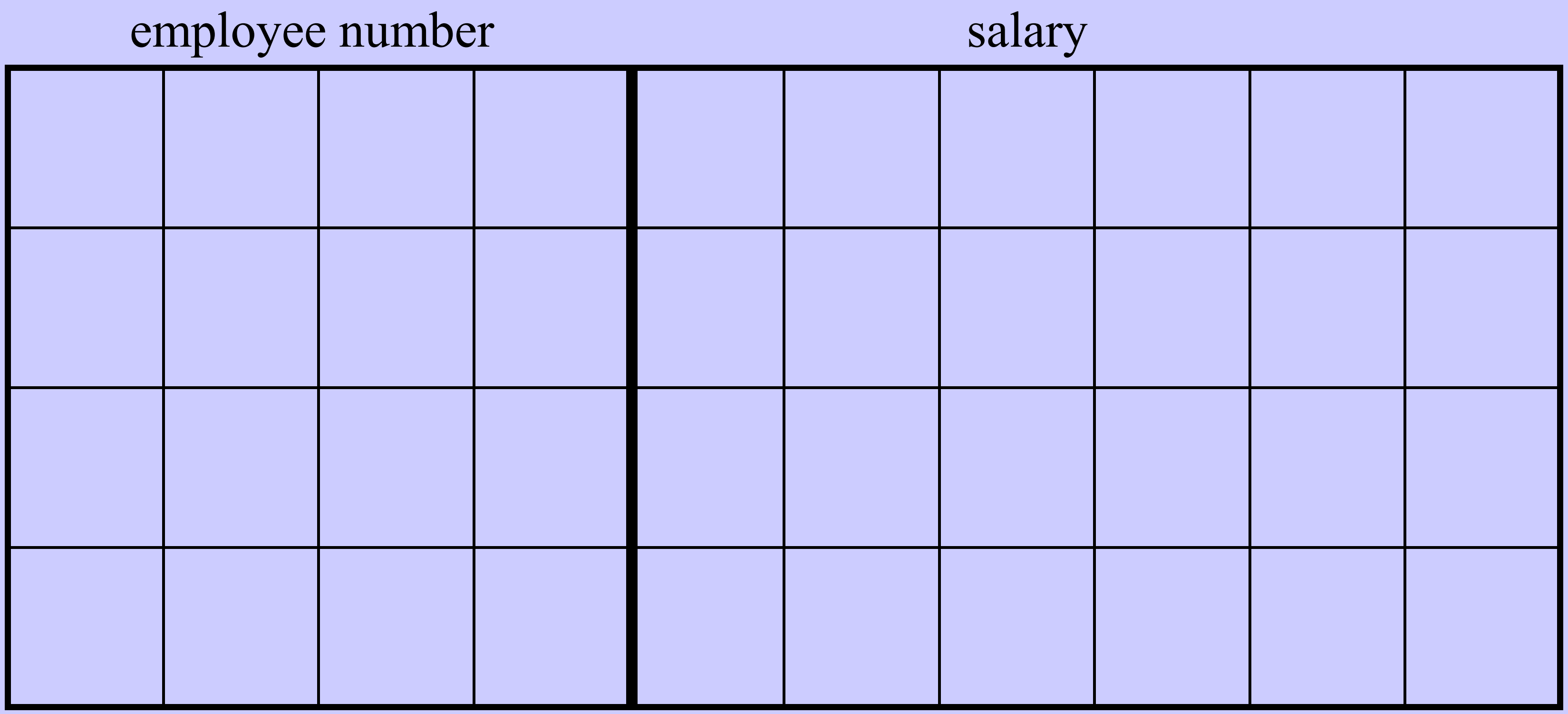
- After invoking rewind( ), the current position in the file is always the first byte position, i.e., at the beginning of the file.

Updating Data in a File

- A file that needs to be updated should be opened using fopen( ) in the “r+” mode, i.e., opening a file for read and write.
- The “r+” mode is useful for operations on files that need to be updated.
- Consider a file, “SALARY.DAT” which contains the following structure:

Updating Data in a File

Structure of SALARY.DAT



Updating Data in a File

- /* function to read the salary field (beginning at byte no. 5 and ending at byte 10) for each record in the file, and increase it by 100 */
- #include<stdio.h>
- main( )
- {
- FILE *fp;
- char empno[5], salary[7];
- double fsalary, atof( );
- long int pos = 4L, offset = 4L;
- /* open the file SALARY.DAT in read-write mode */
- if ((fp = fopen(“SALARY.DAT”, “r+”)) = = NULL)
-   {
-    printf(“Error opening file SALARY.DAT”);
-    exit( );
-   }

Updating Data in a File

- while((fseek(fp, offset, 1)) = = 0)
- {
-   fgets(salary, 7, fp);
-   f_salary = atof(salary) + 100;
-   sprintf(salary, “%6.2f”, f_salary); /*convert f_salary to a string */
-   fseek(fp, pos, 0); /* reposition at start of salary field */
-   fputs(salary, fp); /* update salary field
-   pos += 10; /* increment pos to starting byte of salary field for the next record */
- }
- printf(“The file SALARY.DAT has been updated”);
- fclose(fp);
- }

The ftell( ) and feof( ) Function

- The prototype declaration for the function ftell( ) is:
  - **long ftell(FILE *fp)**
- ftell returns the current file position for stream, or -1 on error.
- The prototype declaration for the function feof( ) is:
  - int feof(FILE *fp)
- feof returns non-zero if the end of file indicator for stream is set.

Writing Records On To a File

- The **fwrite( )** function allows a structure variable to be written on to a file.

- The following statement writes the structure variable salesvar on to a file “SALES.DAT, which is pointed to by the FILE type pointer fp:
- **fwrite(&salesvar, sizeof(struct salesdata), 1, fp);**

- The arguments to the function fwrite( ) are explained as follows:

Writing Structures To a File

- Here **&salesrec** is the address of the structure variable to be written to the file.
- The second parameter is the size of the data to be written, i.e., size of the structure **salesdata**. The parameter to the sizeof( ) operator is the structure label, or the structure type itself, and not a variable of the structure type. The sizeof( ) operator can be used to determine the size of any data type in C (fundamental as well as user-defined data types.
- The third parameter of **fwrite( )** is the number of structure variables to be written to the file. In our statement, it is 1, since only one structure variable is written to the file. In case, an array of 4 structure variables is to be written to a file using **fwrite( )**, the third parameter to **fwrite( )** should be **4**.
- The last parameter is the pointer to the file.

Reading Records from a File

- Records can be read from a file using **fread( )**. The corresponding read statement using **fread( )** for the earlier **fwrite( )** statement would be:
- **fread(&salesvar, sizeof(struct salesdata), 1, fp);**

- Here, the first parameter **&salesvar** is the address of the structure variable **salesvar** into which 1 record is to be read from the file pointed to by the FILE type pointer fp.

- The second parameter specifies the size of the data to be read into the structure variable.

Reading Records from a File

- fread( ) will return the actual number of records read from the file. This feature can be checked for a successful read.

- if ((fread(&salesvar, sizeof(struct salesdata), 1, fp)) != 1)
- error code;

- An odd feature of the **fread( )** function is that it does not return any special character on encountering end of file.

- Therefore, after every read using **fread( ),** care must be taken to check for end of file, for which the standard C library provides the **feof( )** function. It can be used thus:
- if(feof(fp))

Writing Records On To a File

- The **fwrite( )** function allows a structure variable to be written on to a file.

- The following statement writes the structure variable salesvar on to a file “SALES.DAT, which is pointed to by the FILE type pointer fp:
- **fwrite(&salesvar, sizeof(salesdata), 1, fp);**

- The arguments to the function fwrite( ) are explained as follows:

Writing Structures To a File

- Here **&salesvar** is the address of the structure variable to be written to the file.
- The second parameter is the size of the data to be written, i.e., size of the structure **salesdata**. The parameter to the sizeof( ) operator is the structure label, or the structure type itself, and not a variable of the structure type. The sizeof( ) operator can be used to determine the size of any data type in C (fundamental as well as user-defined data types).
- The third parameter of **fwrite( )** is the number of structure variables to be written to the file. In our statement, it is 1, since only one structure variable is written to the file. In case, an array of 4 structure variables is to be written to a file using **fwrite( )**, the third parameter to **fwrite( )** should be **4**.
- The last parameter is the pointer to the file.

Reading Records from a File

- Records can be read from a file using **fread( )**. The corresponding read statement using **fread( )** for the earlier **fwrite( )** statement would be:
- **fread(&salesvar, sizeof(salesdata), 1, fp);**

- Here, the first parameter **&salesvar** is the address of the structure variable **salesvar** into which 1 record is to be read from the file pointed to by the FILE type pointer fp.

- The second parameter specifies the size of the data to be read into the structure variable.

Reading Records from a File

- fread( ) will return the actual number of records read from the file. This feature can be checked for a successful read.

- if ((fread(&salesvar, sizeof(struct salesdata), 1, fp)) != 1)
- error code;

- An odd feature of the **fread( )** function is that it does not return any special character on encountering end of file.

- Therefore, after every read using **fread( ),** care must be taken to check for end of file, for which the standard C library provides the **feof( )** function. It can be used thus:
- if(feof(fp))

Variable-length Argument Lists

- How to write a function that processes a variable-length argument list in a portable way?
- Proper declaration of printf is `int printf(char *fmt, …)`
- The declaration … means that the number and the types of of these arguments may vary
- The declaration … can only appear at the end of an argument list

Variable-length Argument Lists

- The standard header <stdarg.h> needs to be included to process these arguments
- The header contains set of macro definitions that define how to step through an argument list
- The *type* va_list is used to declare a variable that will refer to each argument in turn
- The macro va_start initializes the variable (declared as type va_list) to point the first unnamed argument.
- The macro va_start must be called once before the variable is used.

Variable-length Argument Lists

- There must be at least one named argument and the final (last) named argument is used by va_start to get started.
- The macro va_arg returns one argument and steps the variable (of type va_list) to the next.
- The macro va_arg uses a type name to determine what type to return and how big a step to take.
- The macro va_end does whatever cleanup is necessary.

Variable-length Argument Lists

```
#include<stdarg.h>
void minprintf(char *fmt, ...)
{
va_list ap;/* points to each unnamed arg in turn*/
char *p, *sval;
int ival;
float dval;
va_start(ap, fmt); /* make ap point to 1st unnamed arg*/
for(p = fmt; *p; p++) {
 if(*p !='%')  {
   putchar(*p);
  continue;
}
switch(*++p)  {
case 'd':
ival = va_arg(ap, int);
printf("%d", ival);
break;
case 'f':
dval = va_arg(ap, float);
printf("%f", dval);
break;
default:
putchar(*p);
break;
}
va_end(ap); /*clean up when done*/
}
}
}
```

Variable-length Argument Lists

- The variable *ap* for “argument pointer”

- The statement *va_start(ap, fmt)* initilizes the *ap* with the last named argument *fmt.* It is required before accessing any unnamed argument.

- Thereafter, each execution of macro va_arg will produce a value that has the type and value of the next unnamed argument, and will also modify *ap* so the next use of va_arg returns the next argument.

Volatile qualifier

- The volatile keyword is intended to prevent the compiler from applying any optimizations on objects that can change in ways that cannot be determined by the compiler.

- volatile specifies a variable whose value may be changed by processes outside the current program
- One example of a volatile object might be a buffer used to exchange data with an external device:

Volatile qualifier

- int check_iobuf(void)
- { volatile int iobuf;
- int val;
- while (iobuf == 0) { val = iobuf; iobuf = 0; return(val); }
  if iobuf had not been declared volatile, the compiler would notice that nothing happens inside the loop and thus eliminate the loop
- const and volatile can be used together
  - An input-only buffer for an external device could be declared as const volatile (or volatile const, order is not important) to make sure the compiler knows that the variable should not be changed (because it is input-only) and that its value may be altered by processes other than the current program