

**SAVITRIBAI PHULE PUNE UNIVERSITY**

**SE(ELECTRONICS/E & TC )**

**2019 COURSE**

**204184: DATA STRUCTURES**

**UNIT I**

**INTRODUCTION TO C PROGRAMMING**

**DR. G R PATIL**

**PROFESSOR & HEAD**

**DEPT OF E&TC ENGG**

**AIT PUNE**

# Syllabus

Unit I	Introduction to C Programming	(08 Hrs)
<b>C Fundamentals:</b> Constants, Variables and Keywords in C, Operators, Bitwise Operations, Decision Control and Looping Statements.		
<b>Arrays &amp; Pointers:</b> Arrays, Functions, Recursive Functions, Pointers, String Manipulations, Structures, Union, Enumeration, MACROS.		
<b>File Handling:</b> File Operations- Open, Close, Read, Write and Append		
<b>Mapping of Course Outcomes for Unit I</b>	<b>CO1:</b> After successful completion of course, Student will be able to solve mathematical problems using C programming language.	

# Syllabus & CO-PO Mapping

## CO-PO mapping:

Course Outcome	Blooms Taxonomy Level	After successful completion of the course students will be able to	Mapping with Syllabus Unit	PO MAPPING
CO204184.1	2	Solve mathematical problems using C programming language.	1	1, 2, 3

## Justification of CO-PO mapping:

MAPPING	LEVEL	JUSTIFICATION
CO1-PO1 (Engineering knowledge)	1	Design the program logic using appropriate mathematical Knowledge.
CO1-PO2 (Problem analysis)	1	Analyze the problem with different problem-solving ways,
CO1-PO3 (Design/development of solutions)	1	Program development using suitable logic.

# Text & References

S#	TOPIC	REFERENCE BOOK
UNIT I: Introduction to C and Algorithm		
1	C Fundamentals: Constants, Variables and Keywords in C, Operators, Bitwise Operations, Decision Control and Looping Statements.	R2 (25-30) R2 (52-61) R2(61) R2 (114-189)
2	Arrays & Pointers: Arrays, Functions, Recursive Functions, Pointers, String Manipulations, Structures, Union, Enumeration, MACROS.	R2 (190-209) R2 (262-295) R2 (351-378) R2 (229-250) R2 (317-337) R2 (445-449)
3	File Handling: File Operations- Open, Close, Read, Write and Append	R2 (389-397)

**UNIT I**  
**INTRODUCTION TO C PROGRAMMING**

**TOPIC 1: C FUNDAMENTALS**

**DR. G R PATIL**

**PROFESSOR & HEAD**

**DEPT OF E&TC ENGG**

**AIT PUNE**

# **1.1 CONSTANTS, VARIABLES & KEYWORDS, OPERATORS & BITWISE OPERATIONS**

# Tokens in C

- Keywords
  - ▣ These are reserved words of the C language. For example `int`, `float`, `if`, `else`, `for`, `while` etc.
- Identifiers
  - ▣ An Identifier is a sequence of letters and digits, but must start with a letter. Underscore ( `_` ) is treated as a letter. Identifiers are case sensitive. Identifiers are used to name variables, functions etc.
  - ▣ Valid: `Root`, `_getchar`, `__sin`, `x1`, `x2`, `x3`, `x_1`, `If`
  - ▣ Invalid: `324`, `short`, `price$`, `My Name`
- Constants
  - ▣ Constants like `13`, `'a'`, `1.3e-5` etc.

# Tokens in C

## □ String Literals

- A sequence of characters enclosed in double quotes as "...". For example "13" is a string literal and not number 13. 'a' and "a" are different.

## □ Operators

- Arithmetic operators like +, -, \*, /, % etc.
- Logical operators like ||, &&, ! etc. and so on.

## □ White Spaces

- Spaces, new lines, tabs, comments ( A sequence of characters enclosed in /\* and \*/ ) etc. These are used to separate the adjacent identifiers, keywords and constants.



# Basic Data Types

## □ Integral Types

- ▣ Integers are stored in various sizes. They can be signed or unsigned.

### ▣ Example

Suppose an integer is represented by a byte (8 bits). Leftmost bit is sign bit. If the sign bit is 0, the number is treated as positive.

Bit pattern 01001011 = 75 (decimal).

The largest positive number is 01111111 =  $2^7 - 1 = 127$ .

Negative numbers are stored as two's complement or as one's complement.

-75 = 10110100 (one's complement).

-75 = 10110101 (two's complement).

# Basic Data Types

## □ Integral Types

- **char**                      Stored as 8 bits. Unsigned 0 to 255.  
Signed -128 to 127.
- **short int**                Stored as 16 bits. Unsigned 0 to 65535.  
Signed -32768 to 32767.
- **int**                        Same as either short or long int.
- **long int**                Stored as 32 bits. Unsigned 0 to  
4294967295.  
Signed -2147483648 to 2147483647

# Basic Data Types

- Floating Point Numbers
  - Floating point numbers are rational numbers. Always signed numbers.
  - **float** Approximate precision of 6 decimal digits .
    - Typically stored in 4 bytes with 24 bits of signed mantissa and 8 bits of signed exponent.
  - **double** Approximate precision of 14 decimal digits.
    - Typically stored in 8 bytes with 56 bits of signed mantissa and 8 bits of signed exponent.
- One should check the file limits.h to what is implemented on a particular machine.

# Constants

## □ Numerical Constants

- ▣ Constants like 12, 253 are stored as **int type**. No **decimal point**.
- ▣ 12L or 12l are stored as **long int**.
- ▣ 12U or 12u are stored as **unsigned int**.
- ▣ 12UL or 12ul are stored as **unsigned long int**.
- ▣ Numbers with a decimal point (12.34) are stored as **double**.
- ▣ Numbers with exponent ( $12e-3 = 12 \times 10^{-3}$ ) are stored as **double**.
- ▣ 12.34f or 1.234e1f are stored as **float**.
- ▣ These are not valid constants:  
25,000 7.1e 4                      \$200 2.3e-3.4 etc.

# Constants

## □ Character and string constants

- `'c'` , a single character in single quotes are stored as char. Some special character are represented as two characters in single quotes.  
`'\n'` = newline, `'\t'` = tab, `'\\'` = backlash, `'\"'` = double quotes.  
Char constants also can be written in terms of their ASCII code.  
`'\060'` = `'0'` (Decimal code is 48).
- A sequence of characters enclosed in double quotes is called a string constant or string literal. For example  
"Charu"  
"A"  
"3/9"  
"x = 5"

# Variables

## □ Naming a Variable

- Must be a valid identifier.
- Must not be a keyword
- Names are case sensitive.
- Variables are identified by only first 32 characters.
- Library commonly uses names beginning with \_.
- Naming Styles: Uppercase style and Underscore style
- **lowerLimit lower\_limit**
- **incomeTax income\_tax**

# Declarations

## □ Declaring a Variable

- ▣ Each variable used must be declared.

- ▣ A form of a declaration statement is

**`data-type var1, var2, ...;`**

- ▣ Declaration announces the data type of a variable and allocates appropriate memory location. No initial value (like 0 for integers) should be assumed.

- ▣ It is possible to assign an initial value to a variable in the declaration itself.

**`data-type var = expression;`**

- ▣ Examples

**`int sum = 0;`**

**`char newLine = '\n';`**

**`float epsilon = 1.0e-6;`**

# Standard input/output

16

- The C language itself does not have any special *statements* for performing input/output (I/O) operations; all I/O operations in C must be carried out through *function calls*.
- These functions are contained in the standard C library.
- `#include <stdio.h>`
- Formatted I/O: `scanf()`, `printf()`
- Character I/O: `getchar()`, `putchar()`



# Brief overview - `scanf()`

17

- `scanf(control string, list of arguments)`
- Control string: contains format characters
  - ▣ Important: **match** the **number** and **type** of format characters with the number and type of following arguments !
  - ▣ Format characters: as for `printf`
- Arguments: variable names prefixed with the address operator (&)
- Example:
- `scanf ("%i %i", &x, &y) ;`

# How scanf works

18

- ❑ **Scanf:** searches the input stream for a value to be read, bypasses any leading *whitespace* characters
- ❑ `scanf ("%f %f", &value1, &value2);`
- ❑ `scanf ("%f%f", &value1, &value2);`
- ❑ **In both cases, user can type:**
  - `3 <space> 5 <enter>`
  - `<space> 3 <space> <space> 5 <space> <enter>`
  - `3 <enter> 5 <enter>`
- ❑ **The exceptions:** in the case of the `%c` format characters—the next character from the input, no matter what it is, is read
- ❑ `scanf ("%f %c %f", &value1, &op, &value2);`
  - `3 <space> + <space> 5 <enter>`
  - `3 <space> + <space> <space> <enter> 5 <enter>`
- ❑ `scanf ("%f%c%f", &value1, &op, &value2);`
  - `3+5<enter>`
  - Not OK:** `3 <space> +5<enter>`  $\Rightarrow$  `op` would take the value `<space>`, character `+` would remain as input for `value2` !

# How scanf works

19

- When scanf reads in a particular value: reading of the value terminates when a character that is not valid for the value type being read is encountered.
- `scanf ("%f%c%f", &value1, &op, &value2);`  
`3+5<enter>`
- Any nonformat characters that are specified in the format string of the scanf call are expected on the input.
- `scanf ("%i:%i:%i", &hour, &minutes, &seconds);`  
`3:6:21<enter>`  
`3<space>:<space>6<space>:<space>21<enter>`  
`3<space>6<space>21<space> => NOT OK !`
- The next call to scanf picks up where the last one left off.
- `scanf ("%f", &value1);`
- **User types:** `7 <space> 8 <space> 9 <enter>`
- **7 is stored in value1, rest of the input chars remain waiting in buffer**
- `scanf ("%f", &value2);`
- **8 from buffer is stored in value2, rest of the input remains waiting**

# getchar () and putchar ()

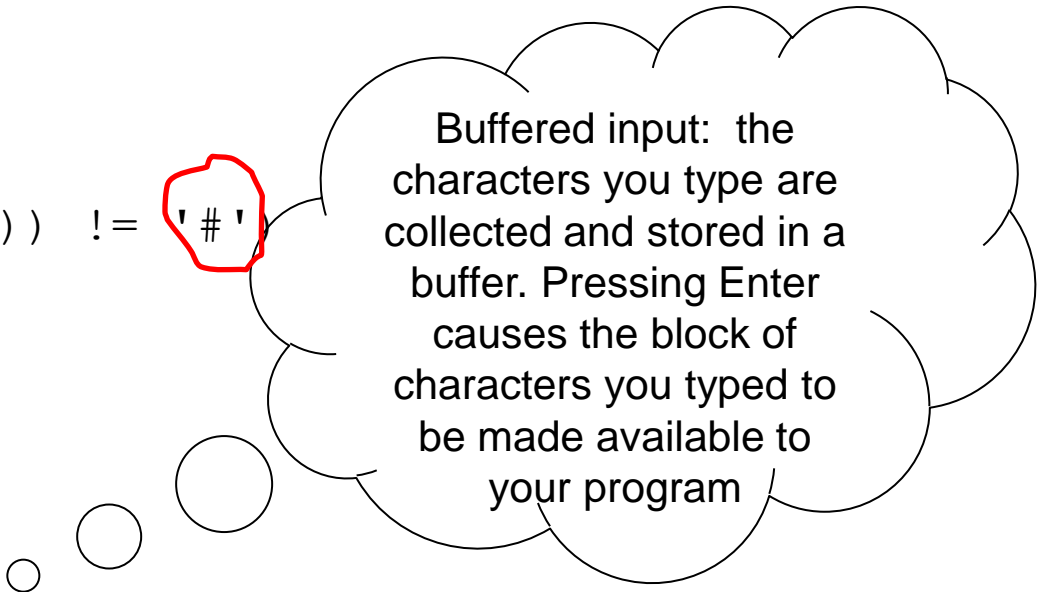
20

- The simplest input mechanism is to read one character at a time from the *standard input*, with `getchar`
- To display a character: `putchar`

# Example: getchar()

21

```
#include <stdio.h>
int main(void) {
    char ch;
    while ((ch = getchar()) != '#')
        putchar(ch);
    return 0;
}
```



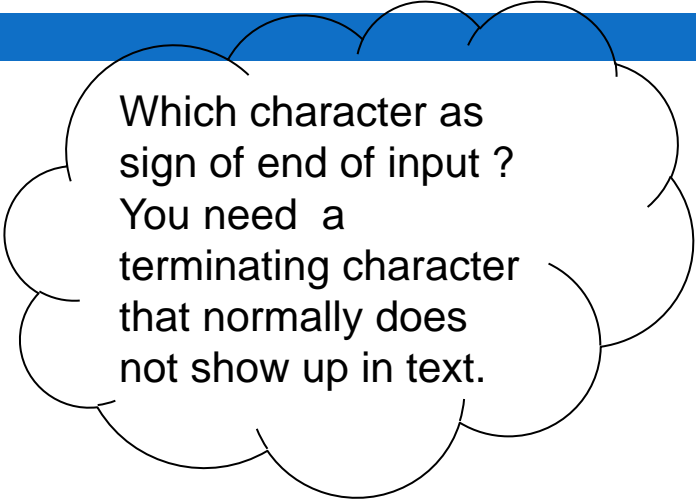
Buffered input: the characters you type are collected and stored in a buffer. Pressing Enter causes the block of characters you typed to be made available to your program

```
Hello ! I am<enter>
Hello ! I am
Joe from #3.<enter>
Joe from
```

# Terminating keyboard input

22

```
#include <stdio.h>
int main(void) {
    int ch;
    while ((ch = getchar()) != EOF)
        putchar(ch);
    return 0;
}
```



Which character as  
sign of end of input ?  
You need a  
terminating character  
that normally does  
not show up in text.

getchar returns the next input character each time it is called,  
or EOF when it encounters end of file.

EOF is a symbolic constant defined in <stdio.h>. (The value is typically -1)

EOF from the keyboard: Ctrl+Z

# Exercise: getchar()

23

```
/* Read characters from input over several lines until EOF.  
   Count lines and characters in input */
```

```
#include <stdio.h>  
int main(void) {  
    int  c,  nl, nc;  
    nl = 0;  
    nc = 0;  
    while ((c = getchar()) != EOF) {  
        nc++;  
        if (c == '\n')  
            nl++;  
    }  
    printf("Number of lines in input: %d\n", nl);  
    printf("Number of characters in input: %d\n", nc);  
    return 1;  
}
```

# Global and Local Variables

## □ Global Variables

- These variables are declared outside all functions.
- Life time of a global variable is the entire execution period of the program.
- Can be accessed by any function defined below the declaration, in a file.

```
/* Compute Area and Perimeter of a
   circle */
#include <stdio.h>
float pi = 3.14159; /* Global */

main() {
    float rad;      /* Local */

    printf( "Enter the radius " );
    scanf("%f" , &rad);

    if ( rad > 0.0 ) {
        float area = pi * rad * rad;
        float peri = 2 * pi * rad;

        printf( "Area = %f\n" , area );
        printf( "Peri = %f\n" , peri );
    }
    else
        printf( "Negative radius\n");

    printf( "Area = %f\n" , area );
}
```



# Global and Local Variables

## □ Local Variables

- These variables are declared inside some functions.
- Life time of a local variable is the entire execution period of the function in which it is defined.
- Cannot be accessed by any other function.
- In general variables declared inside a block are accessible only in that block.

```
/* Compute Area and Perimeter of a
   circle */
#include <stdio.h>
float pi = 3.14159; /* Global */

main() {
    float rad;      /* Local */

    printf( "Enter the radius " );
    scanf("%f" , &rad);

    if ( rad > 0.0 ) {
        float area = pi * rad * rad;
        float peri = 2 * pi * rad;

        printf( "Area = %f\n" , area );
        printf( "Peri = %f\n" , peri );
    }
    else
        printf( "Negative radius\n");

    printf( "Area = %f\n" , area );
}
```

# Operators

## □ Arithmetic Operators

□  $+$ ,  $-$ ,  $*$ ,  $/$  and the modulus operator  $\%$ .

□  $+$  and  $-$  have the same precedence and associate left to right.

$$3 - 5 + 7 = (3 - 5) + 7 \neq 3 - (5 + 7)$$

$$3 + 7 - 5 + 2 = ((3 + 7) - 5) + 2$$

□  $*$ ,  $/$ ,  $\%$  have the same precedence and associate left to right.

□ The  $+$ ,  $-$  group has lower precedence than the  $*$ ,  $/$ ,  $\%$  group.

$$3 - 5 * 7 / 8 + 6 / 2$$

$$3 - 35 / 8 + 6 / 2$$

$$3 - 4.375 + 6 / 2$$

$$3 - 4.375 + 3$$

$$-1.375 + 3$$

$$1.625$$

# Operators

## □ Arithmetic Operators

- % is a modulus operator.  $x \% y$  results in the remainder when  $x$  is divided by  $y$  and is zero when  $x$  is divisible by  $y$ .
- Cannot be applied to float or double variables.
- Example

```
if ( num % 2 == 0 )  
    printf("%d is an even number\n", num) ;  
else  
    printf("%d is an odd number\n", num) ;
```

# Type Conversions

- ▣ The operands of a binary operator must have the same type and the result is also of the same type.

- ▣ Integer division:

```
c = (9 / 5) * (f - 32)
```

The operands of the division are both int and hence the result also would be int. For correct results, one may write

```
c = (9.0 / 5.0) * (f - 32)
```

- ▣ In case the two operands of a binary operator are different, but compatible, then they are converted to the same type by the compiler. The mechanism (set of rules) is called **Automatic Type Casting**.

```
c = (9.0 / 5) * (f - 32)
```

- ▣ It is possible to force a conversion of an operand. This is called **Explicit Type casting**.

```
c = ((float) 9 / 5) * (f - 32)
```

# Automatic Type Casting

1. char and short operands are converted to int
2. Lower data types are converted to the higher data types and result is of higher type.
3. The conversions between unsigned and signed types may not yield intuitive results.

4. Example

`float f; double d; long l;`

`int i; short s;`

`d + f` f will be converted to double

`i / s` s will be converted to int

`l / i` i is converted to long; long result

## Hierarchy

Double

float

long

Int

Short and  
char

# Explicit Type Casting

- ▣ The general form of a type casting operator is
- ▣ (type-name) expression
- ▣ It is generally a good practice to use explicit casts than to rely on automatic type conversions.
- ▣ Example  
$$C = (\text{float}) 9 / 5 * (f - 32)$$
- ▣ float to int conversion causes truncation of fractional part
- ▣ double to float conversion causes rounding of digits
- ▣ long int to int causes dropping of the higher order bits.

# Precedence and Order of evaluation

**Table 3.8 Summary of C Operators**

OPERATOR	DESCRIPTION	ASSOCIATIVITY	RANK
( )	Function call	Left to right	1
[ ]	Array element reference		
+	Unary plus	Right to left	2
-	Unary minus		
++	Increment		
--	Decrement		
!	Logical negation		
~	Ones complement		
*	Pointer reference (indirection)		
&	Address	Left to right	3
sizeof	Size of an object		
(type)	Type cast (conversion)		
*	Multiplication	Left to right	4
/	Division		
%	Modulus		
+	Addition	Left to right	4
-	Subtraction		

# Precedence and Order of evaluation

OPERATOR	DESCRIPTION	ASSOCIATIVITY
<< >>	Left shift Right shift	Left to right
< <= > >=	Less than Less than or equal to Greater than Greater than or equal to	Left to right
== !=	Equality Inequality	Left to right
&	Bitwise AND	Left to right
^	Bitwise XOR	Left to right
	Bitwise OR	Left to right
&&	Logical AND	Left to right
	Logical OR	Left to right
?:	Conditional expression	Right to left
= *= /= %= += -= &= *=  = <<= >>=	Assignment operators	Right to left
,	Comma operator	Left to right

Dr G R Patil, AIT Pune



# Operators

## □ Relational Operators

- `<`, `<=`, `>`, `>=`, `==`, `!=` are the relational operators. The expression

`operand1 relational-operator operand2`

takes a value of 1(int) if the relationship is true and 0(int) if relationship is false.

## ■ Example

```
int a = 25, b = 30, c, d;
```

```
c = a < b;
```

```
d = a > b;
```

value of c will be 1 and that of d will be 0.

# Operators

## □ Logical Operators

- ▣ `&&`, `||` and `!` are the three logical operators.
- ▣ `expr1 && expr2` has a value 1 if `expr1` and `expr2` both are nonzero.
- ▣ `expr1 || expr2` has a value 1 if `expr1` and `expr2` both are nonzero.
- ▣ `!expr1` has a value 1 if `expr1` is zero else 0.

## ▣ Example

- ▣ 

```
if ( marks >= 40 && attendance >= 75 )  
    grade = 'P'
```
- ▣ 

```
If ( marks < 40 || attendance < 75 )  
    grade = 'N'
```

# Operators

## □ Assignment operators

▣ The general form of an assignment operator is

▣  $v \text{ op} = \text{exp}$

▣ Where  $v$  is a variable and  $op$  is a binary arithmetic operator.  
This statement is equivalent to

▣  $v = v \text{ op } (\text{exp})$

▣  $a = a + b$   
 $+= b$

can be written as

$a$

▣  $a = a * b$   
 $*= b$

can be written as

$a$

▣  $a = a / b$   
 $/= b$

can be written as

$a$

▣  $a = a - b$   
 $= b$

can be written as

$a -$

# Operators

## □ Increment and Decrement Operators

- ▣ The operators `++` and `--` are called increment and decrement operators.
- ▣ `a++` and `++a` are equivalent to `a += 1`.
- ▣ `a--` and `--a` are equivalent to `a -= 1`.
- ▣ `++a op b` is equivalent to `a ++; a op b;`
- ▣ `a++ op b` is equivalent to `a op b; a++;`
- ▣ Example

Let `b = 10` then

`(++b)+b+b = 33`

`b+(++b)+b = 33`

`b+b+(++b) = 31`

`b+b*(++b) = 132`

# Bitwise Operators

37

- Binary representation of integer numbers
- Operations on bits
  - The Bitwise AND Operator
  - The Bitwise Inclusive-OR Operator
  - The Bitwise Exclusive-OR Operator
  - The Ones Complement Operator
  - The Left Shift Operator
  - The Right Shift Operator
- Review: Operators: The complete table of operator's precedence

# Binary representation of integers

38

- Positive integers:



*Sign bit*

*most  
significant or  
high-order bit*

*least  
significant or  
low-order bit*

$$1 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 = 5$$

# Binary representation of integers

39

- Negative integers:



Sign bit

most  
significant or  
high-order bit

least  
significant or  
low-order bit

Steps to convert a negative number from decimal to binary: Ex: -5

1. add 1 to the value:  $-5+1=-4$
2. express the absolute value of the result in binary:  $4=00000100$
3. then “complement” all the bits:  $11111011=-5$

Steps to convert a negative number from binary to decimally: Ex: 11111011

1. complement all of the bits:  $00000100$
2. convert the result to decimal:  $4$
3. change the sign of the result, and then subtract 1:  $4-1=-5$

# Bit operators

40

- `&` Bitwise AND
- `|` Bitwise Inclusive-OR
- `^` Bitwise Exclusive-OR
- `~` Ones complement
- `<<` Left shift
- `>>` Right shift
- All the operators, with the exception of the ones complement operator `~`, are binary operators and as such take two operands.
- Bit operations can be performed on **any type of integer value** in C—be it short, long, long long, and signed or unsigned—and on characters, but **cannot be performed on floating-point values**.



# Bitwise AND

41

b1	b2	b1&b2
0	0	0
0	1	0
1	0	0
1	1	1

```
int a=25;
```

```
int b=77;
```

```
printf("%x %x %x",a,b,a&b);
```

0	0	0	0	0		0	0	0	1	1	0	0	1
---	---	---	---	---	--	---	---	---	---	---	---	---	---

a=0x19

0	0	0	0	0		0	1	0	0	1	1	0	1
---	---	---	---	---	--	---	---	---	---	---	---	---	---

b=0x4d

0	0	0	0	0		0	0	0	0	1	0	0	1
---	---	---	---	---	--	---	---	---	---	---	---	---	---

a&b=0x9

# Exercise: even or odd

42

- testing if an integer is even or odd, using bitwise operations:
- the rightmost bit of any odd integer is 1 and of any even integer is 0.

```
int n;  
if ( n & 1 )  
    printf("odd");  
else  
    printf("even");
```

# Bitwise Inclusive-OR (OR)

43

b1	b2	b1 b2
0	0	0
0	1	1
1	0	1
1	1	1

```
int a=25;  
int b=77;
```

```
printf("%x %x %x",a,b,a|b);
```

0	0	0	0	0		0	0	0	1	1	0	0	1
---	---	---	---	---	--	---	---	---	---	---	---	---	---

a=0x19

0	0	0	0	0		0	1	0	0	1	1	0	1
---	---	---	---	---	--	---	---	---	---	---	---	---	---

b=0x4d

0	0	0	0	0		0	1	0	1	1	1	0	1
---	---	---	---	---	--	---	---	---	---	---	---	---	---

a|b=0x5d

# Bitwise vs boolean !

44

- AND: bitwise: &, boolean: &&
- OR: bitwise: |, boolean: ||

```
int a,b;  
a=1;  
b=2;  
printf("a:%i    b:%i    &:%i    &&:%i    |:%i    ||:%i \n",
```

# Bitwise Exclusive-OR (XOR)

45

b1	b2	b1^b2
0	0	0
0	1	1
1	0	1
1	1	0

```
int a=25;
```

```
int b=77;
```

```
printf("%x %x %x", a, b, a^b);
```

0	0	0	0	0		0	0	0	1	1	0	0	1
---	---	---	---	---	--	---	---	---	---	---	---	---	---

a=0x19

0	0	0	0	0		0	1	0	0	1	1	0	1
---	---	---	---	---	--	---	---	---	---	---	---	---	---

b=0x4d

0	0	0	0	0		0	1	0	1	0	1	0	0
---	---	---	---	---	--	---	---	---	---	---	---	---	---

a^b=0x54

# The Ones Complement Operator

46

b1	~b1
0	1
1	0

```
int a=25;
```

```
printf("%x %x", a, ~a);
```

0	0	0	0	0		0	0	0	1	1	0	0	1	a=0x19
1	1	1	1	1		1	1	1	0	0	1	1	0	~a=0xfffffe6

# Usage: Masking bits

47

- **A mask** is a bit pattern with some bits set to on (1) and some bits to off (0).
- **Example:**
- `int MASK = 2; // 00000010, with only bit number 1 nonzero.`
- **Masking bits:**
- `flags = flags & MASK;`
- all the bits of `flags` (except bit 1) are set to 0 because any bit combined with 0 using the `&` operator yields 0. Bit number 1 will be left unchanged. (If the bit is 1, `1 & 1` is 1; if the bit is 0, `0 & 1` is 0.) This process is called "using a mask" because the zeros in the mask hide the corresponding bits in `flags`.
- you can think of the 0s in the mask as being opaque and the 1s as being transparent. The expression `flags & MASK` is like covering the `flags` bit pattern with the mask; only the bits under `MASK`'s 1s are visible

# Usage: Turning bits on

48

- Turn on (set to 1) particular bits in a value while leaving the remaining bits unchanged.
- Example: an IBM PC controls hardware through values sent to ports. To turn on a device (i.e., the speaker), the driver program has to turn on the 1 bit while leaving the others unchanged.
- This can be done with the bitwise OR operator.
- The MASK has bit 1 set to 1, other bits 0
- `int MASK = 2; // 00000010, with only bit number 1 nonzero.`
- The statement
- `flags = flags | MASK;`
- sets bit number 1 in `flags` to 1 and leaves all the other bits unchanged. This follows because any bit combined with 0 by using the `|` operator is itself, and any bit combined with 1 by using the `|` operator is 1.



# Usage: Turning bits off

49

- Turn off (set to 0) particular bits in a value while leaving the remaining bits unchanged.
- Suppose you want to turn off bit 1 in the variable `flags`.
- The MASK has bit 1 set to 1, other bits 0
- `int MASK = 2; // 00000010, with only bit number 1 nonzero.`
- The statement:
- `flags = flags & ~MASK;`
- Because MASK is all 0s except for bit 1, `~MASK` is all 1s except for bit 1. A 1 combined with any bit using `&` is that bit, so the statement leaves all the bits other than bit 1 unchanged. Also, a 0 combined with any bit using `&` is 0, so bit 1 is set to 0 regardless of its original value.

# Usage: Toggling bits

50

- Toggling a bit means turning it off if it is on, and turning it on if it is off.
- To toggle bit 1 in `flag`:
- The `MASK` has bit 1 set to 1, other bits 0
- `int MASK = 2; // 00000010, with only bit number 1 nonzero.`
- `flag = flag ^ MASK;`
- You can use the bitwise EXCLUSIVE OR operator to toggle a bit.
- The idea is that if `b` is a bit setting (1 or 0), then `1 ^ b` is 0 if `b` is 1 and is 1 if `b` is 0. Also `0 ^ b` is `b`, regardless of its value.
- Therefore, if you combine a value with a mask by using `^`, values corresponding to 1s in the mask are toggled, and values corresponding to 0s in the mask are unaltered.

# Usage: Checking value of a bit

51

- For example: does `flag` has bit 1 set to 1?
- you must first mask the other bits in `flag` so that you compare only bit 1 of `flag` with `MASK`:
  - `if ((flag & MASK) == MASK)`
  -

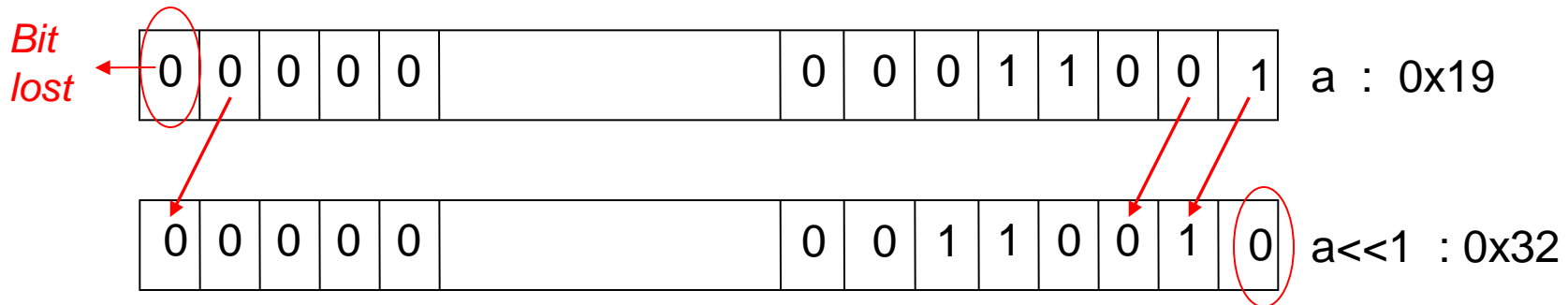
# The Left Shift Operator

52

- the bits contained in the value are shifted to the left a number of places (bits)
- Bits that are shifted out through the high-order bit of the data item are lost,
- 0s are always shifted in through the low-order bit of the value.

```
int a=25;
```

```
printf("%x %x", a, a<<1);
```



# Exercise: multiply with 2

53

- Left shifting has the effect of multiplying the value that is shifted by two.
- Multiplication of a value by a power of two: left shifting the value the appropriate number of places

```
int a;  
scanf("%i", &a);  
printf("a multiplied with 2 is %i \n", a<<1);  
printf("a multiplied with 4 is %i \n", a<<2);
```

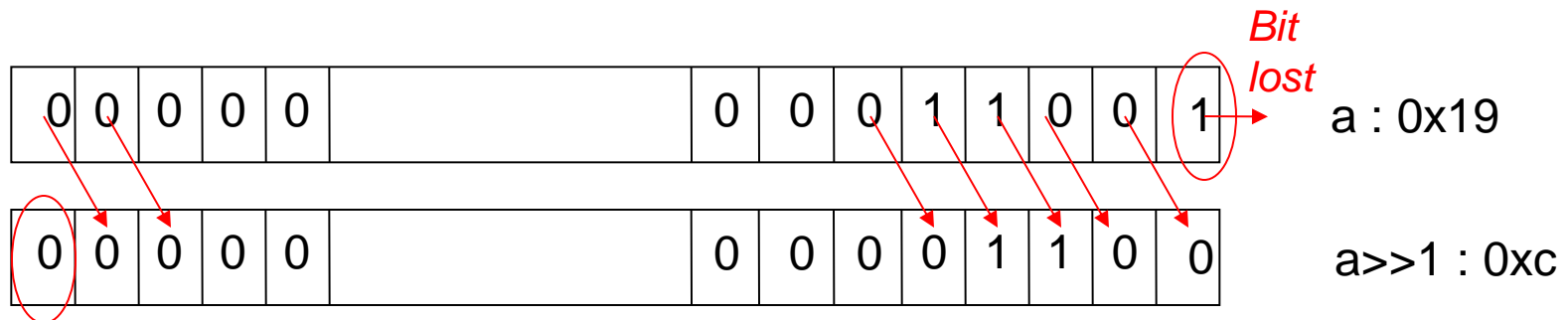
# The Right Shift Operator

54

- Shifts the bits of a value to the right.
- Bits shifted out of the low-order bit of the value are lost.
- Right shifting an **unsigned value or a signed positive value** always results in **0s being shifted in on the left**

```
int a=25;
```

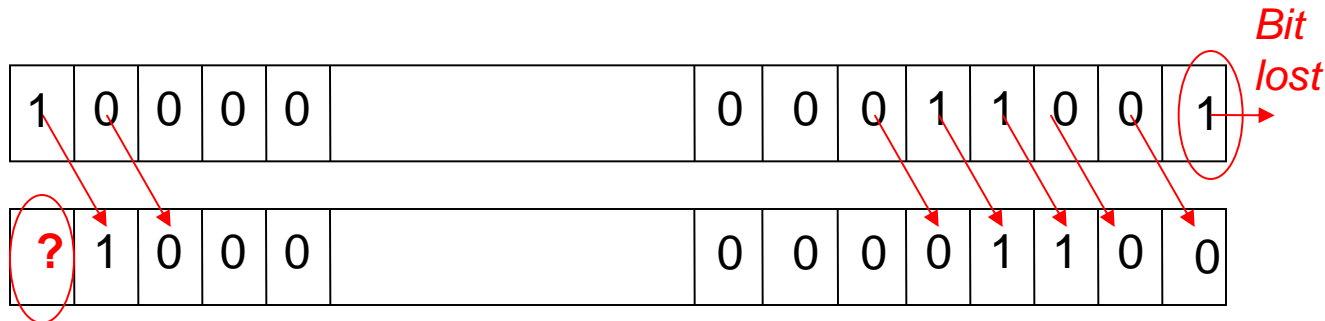
```
printf("%x %x", a, a>>1);
```



# Right shift of negative integers

55

- If the sign bit is 1, on some machines 1s are shifted in, and on others 0s are shifted in.
- This former type of operation is known as an *arithmetic* right shift, whereas the latter is known as a *logical* right shift.
- Never make any assumptions about whether a system implements an arithmetic or a logical right shift. A program that shifts signed values right might work correctly on one system but fail on another due to this type of assumption.



# Example: extract groups of bits

56

- Example: an unsigned long value is used to represent color values, with the low-order byte holding the red intensity, the next byte holding the green intensity, and the third byte holding the blue intensity.
- You then want to store the intensity of each color in its own unsigned char variable.

```
unsigned char BYTE_MASK = 0xff;
unsigned long color = 0x002a162f;
unsigned char blue, green, red;
red = color & BYTE_MASK;
green = (color >> 8) & BYTE_MASK;
blue = (color >> 16) & BYTE_MASK;
printf("%x %x %x\n", red, green, blue);
```

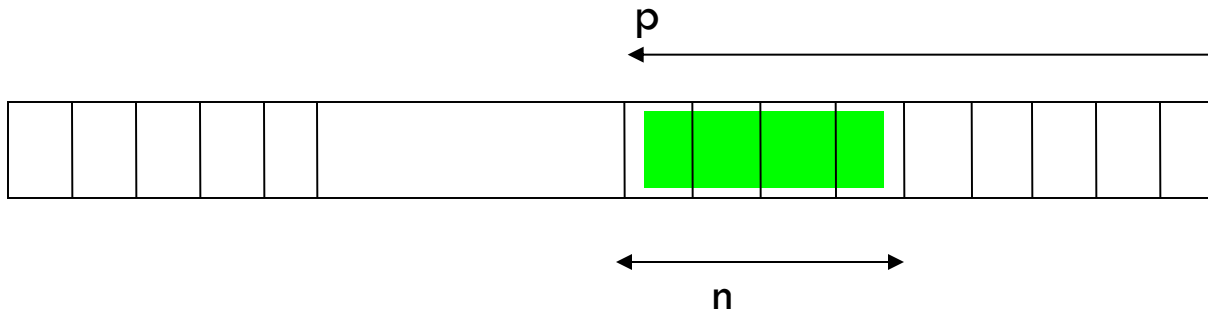


# Exercise: getbits

57

- `getbits(x, p, n)` : returns the (right adjusted)  $n$ -bit field of  $x$  that begins at position  $p$ . We assume that bit position 0 is at the right end and that  $n$  and  $p$  are positive values.

```
/* getbits: get n bits from position p */
unsigned getbits(unsigned x, int p, int n)
{
    return (x >> (p+1-n)) & ~ (~0 << n);
}
```



# Exercise: printbits

58

- `printbits(x, p, n)` : prints the `n`-bit field of `x` that begins at position `p`. We assume that bit position 0 is at the right end and that `n` and `p` are positive values.

```
void printbits(unsigned x, int p, int n)
{
    int i;
    unsigned mask=1<<p;
    for (i=0; i<n; i++) {
        if (x&mask) printf("1");
        else printf("0");
        mask=mask>>1;
    }
}
```

# Review: Operators in C

59

Table A.5 Summary of C Operators

Operator	Description	Associativity
()	Function call	Left to right
[]	Array element reference	
->	Pointer to structure member reference	
.	Structure member reference	
-	Unary minus	Right to left
+	Unary plus	
++	Increment	
--	Decrement	
!	Logical negation	
~	Ones complement	
*	Pointer reference (indirection)	
&	Address	
sizeof	Size of an object	
(type)	Type cast (conversion)	

*	Multiplication	
/	Division	Left to right
%	Modulus	
+	Addition	Left to right
-	Subtraction	
<<	Left shift	Left to right
>>	Right shift	
<	Less than	
<=	Less than or equal to	Left to right
>	Greater than	
=>	Greater than or equal to	

Table A.5 Continued

Operator	Description	Associativity
==	Equality	Left to right
!=	Inequality	
&	Bitwise AND	Left to right
^	Bitwise XOR	Left to right
	Bitwise OR	Left to right
&&	Logical AND	Left to right
	Logical OR	Left to right
?:	Conditional	Right to left
=		
*= /= %=		
+= -= &=	Assignment operators	Right to left
^=  =		
<<= >>=		
,	Comma operator	Right to left

## 1.2 DECISION CONTROL

# Outline

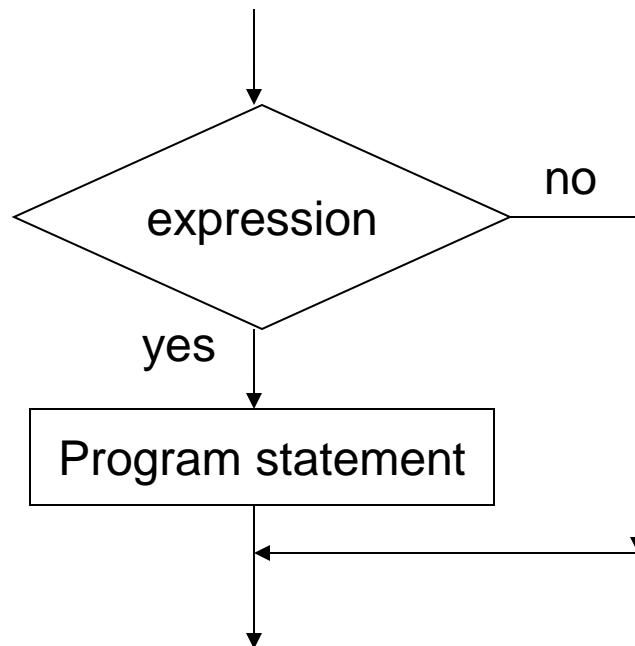
63

- **Making Decisions [chap 6 – Kochan]**
  - ▣ The `if` Statement
  - ▣ The `if-else` Construct
  - ▣ Logical Operators
  - ▣ Boolean Variables
  - ▣ Nested `if` Statements
  - ▣ The `else if` Construct
  - ▣ The `switch` Statement
  - ▣ The Conditional Operator
- **Character Input/Output**

# The `if` statement

64

```
if ( expression )  
    program statement
```



If expression is true (non-zero), executes statement.  
If gives you the choice of executing statement or skipping it.



# Example - if

65

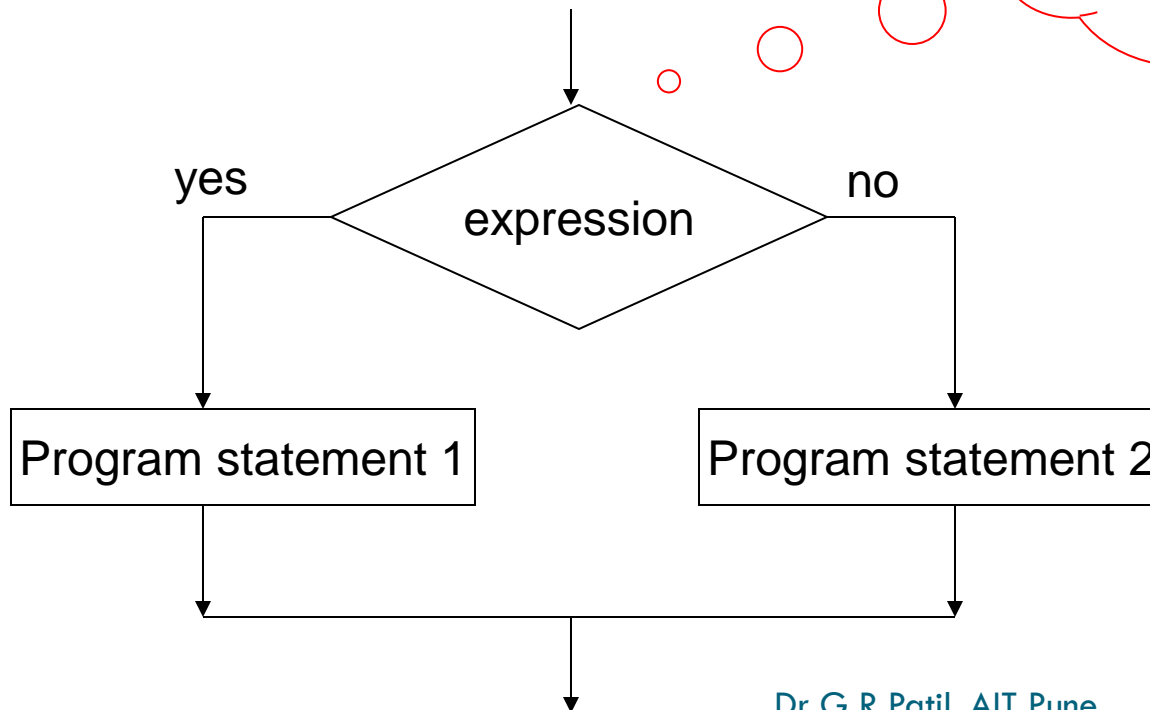
```
// Program to calculate the absolute value of an integer
int main (void)
{
    int number;
    printf ("Type in your number: ");
    scanf ("%i", &number);
    if ( number < 0 )
        number = -number;
    printf ("The absolute value is %i\n", number);
    return 0;
}
```

# The if-else statement

66

```
if ( expression )  
    program statement 1  
else  
    program statement 2
```

if-else statement:  
enables you to  
choose between  
two statements



# Example: if-else

67

```
// Program to determine if a number is even or odd
#include <stdio.h>
int main ()
{
    int number_to_test, remainder;
    printf ("Enter your number to be tested: ");
    scanf ("%i", &number_to_test);
    remainder = number_to_test % 2;
    if ( remainder == 0 )
        printf ("The number is even.\n");
    else
        printf ("The number is odd.\n");
    return 0;
}
```

# Attention on `if-else` syntax !

68

```
if ( expression )  
    program statement 1  
else  
    program statement 2
```

```
if ( remainder == 0 )  
    printf ("The number is even.\n");  
else  
    printf ("The number is odd.\n");
```

```
if ( x == 0 );  
    printf ("The number is zero.\n");
```

In C, the ; is part  
(end) of a statement !  
You have to put it  
also before an `else`  
!

Syntactically OK  
(void statement  
on if) but  
probably a  
semantic error !

# Example: compound relational test

69

```
// Program to determine if a year is a leap year
#include <stdio.h>
int main (void)
{
    int year, rem_4, rem_100, rem_400;
    printf ("Enter the year to be tested: ");
    scanf ("%i", &year);
    rem_4 = year % 4;
    rem_100 = year % 100;
    rem_400 = year % 400;
    if ( (rem_4 == 0 && rem_100 != 0) || rem_400 == 0 )
        printf ("It's a leap year.\n");
    else
        printf ("It's not a leap year.\n");
    return 0;
}
```

# Logical operators

70

Operator	Symbol	Meaning
AND	& &	X && y is true if BOTH x and y are true
OR		X    y is true if at least one of x and y is true
NOT	!	!x is true if x is false

Logical values as operands or in tests: true = non-zero, false=zero

Logical values returned as results of expressions: true = 1, false=zero

Example: 5 || 0 is 1

# Example

71

- Program to generate a table of all prime numbers up to 50

# Boolean variables

72

```
// Program to generate a table of prime numbers
#include <stdio.h>
int main (void) {
    int p, d;
    int isPrime;
    for ( p = 2; p <= 50; ++p ) {
        isPrime = 1;
        for ( d = 2; d < p; ++d )
            if ( p % d == 0 )
                isPrime = 0;
        if ( isPrime != 0 )
            printf ("%i ", p);
    }
    printf ("\n");
    return 0;
}
```

**A *flag*:** assumes only one of two different values. The value of a flag is usually tested in the program to see if it is “on” (TRUE) or “off” (FALSE)

**Equivalent test:**  
(more in C-style):  
if (isPrime)



# Boolean variables

73

```
// Program to generate a table of prime numbers - rewritten
#include <stdio.h>
#include <stdbool.h>
int main (void) {
    int p, d;
    bool isPrime;
    for ( p = 2; p <= 50; ++p ) {
        isPrime = true;
        for ( d = 2; d < p; ++d )
            if ( p % d == 0 )
                isPrime = false;
        if ( isPrime )
            printf ("%i ", p);
    }
    printf ("\n");
    return 0;
}
```

# Precedence of operators

74

## Precedence

!, ++, --, (type)

\*, /, %

+, -

<, <=, >, >=, ==, !=

&&

||

=

Example for operator precedence:

`a > b && b > c || b > d`

Is equivalent to:

`((a > b) && (b > c)) || (b > d)`

# Testing for ranges

75



```
if (x >= 5 && x <= 10)
    printf("in range");
```

```
if (5 <= x <= 10)
    printf("in range");
```

# Testing for ranges

76

**YES**

```
if (x >= 5 && x <= 10)
    printf("in range");
```

**NO!**

```
if (5 <= x <= 10)
    printf("in range");
```

Syntactically correct, but semantically an error !!!

Because the order of evaluation for the `<=` operator is left-to-right, the test expression is interpreted as follows:

$(5 \leq x) \leq 10$

The subexpression  $5 \leq x$  either has the value 1 (for true) or 0 (for false). Either value is less than 10, so the whole expression is always true, regardless of  $x$  !

# Testing for character ranges

77

```
char ch;
scanf("%c", &ch);
if (ch >= 'a' && ch <= 'z')
    printf("lowercase char\n");
if (ch >= 'A' && ch <= 'Z')
    printf("uppercase char\n");
if (ch >= '0' && ch <= '9')
    printf("digit char\n");
```

- ❑ This works for character codes such as ASCII, in which the codes for consecutive letters are consecutive numbers. However, this is not true for some codes (i.e. EBCDIC)
- ❑ A more portable way of doing this test is to use functions from `<ctype.h>`  
`islower(ch)`, `isupper(ch)`, `isdigit(ch)`

# Other operations on characters

78

```
char ch='d';  
ch=ch+1; o o o
```

c will be the  
next letter 'e'

```
char ch='d';  
ch=ch+'A'-'a'; o o o
```

c will be the  
corresponding  
uppercase letter  
'D'

- This works for character codes such as ASCII, in which the codes for consecutive letters are consecutive numbers.
- A more portable way: `<ctype.h>: toupper(c), tolower(c)`

# Nested if statements

79

```
if (number > 5)
    if (number < 10)
        printf("1111\n");
    else printf("2222\n");
```

```
if (number > 5) {
    if (number < 10)
        printf("1111\n");
}
else printf("2222\n");
```

Rule: an else goes with the most recent if, unless braces indicate otherwise

# Example: else-if

80

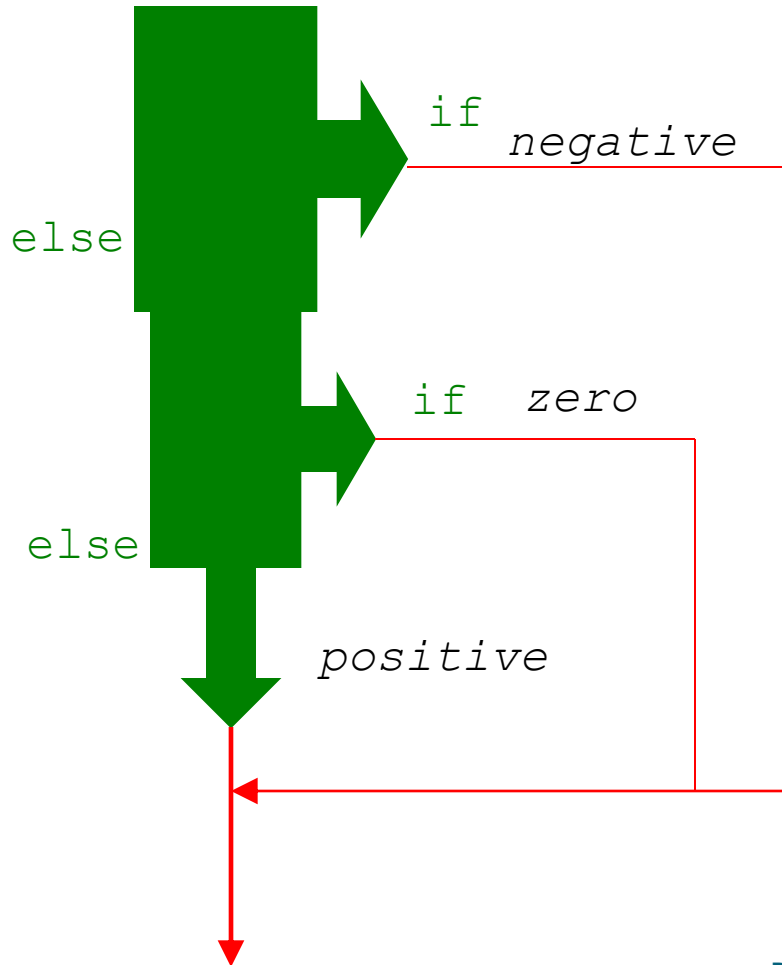
```
// Program to implement the sign function
#include <stdio.h>
int main (void)
{
    int number, sign;
    printf ("Please type in a number: ");
    scanf ("%i", &number);
    if ( number < 0 )
        sign = -1;
    else if ( number == 0 )
        sign = 0;
    else // Must be positive
        sign = 1;
    printf ("Sign = %i\n", sign);
    return 0;
}
```



# Multiple choices – else-if

81

```
int number;
```



```
if ( expression 1)
    program statement 1
else if ( expression 2)
    program statement 2
else
    ○ program statement 3
    ○
    ○
```

Program style: this unindented formatting improves the readability of the statement and makes it clearer that a three-way decision is being made.

# Example: else-if

82

```
// Program to categorize a single character
// that is entered at the terminal
#include <stdio.h>
int main (void)
{
    char c;
    printf ("Enter a single character:\n");
    scanf ("%c", &c);
    if ( (c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z') )
        printf ("It's an alphabetic character.\n");
    else if ( c >= '0' && c <= '9' )
        printf ("It's a digit.\n");
    else
        printf ("It's a special character.\n");
    return 0;
}
```

# Example – multiple choices

83

```
/* Program to evaluate simple expressions of the form
number operator number */
#include <stdio.h>
int main (void) {
    float value1, value2;
    char operator;
    printf ("Type in your expression.\n");
    scanf ("%f %c %f", &value1, &operator, &value2);
    if ( operator == '+' )
        printf (".2f\n", value1 + value2);
    else if ( operator == '-' )
        printf (".2f\n", value1 - value2);
    else if ( operator == '*' )
        printf (".2f\n", value1 * value2);
    else if ( operator == '/' )
        printf (".2f\n", value1 / value2);
    else printf ("Unknown operator.\n");
    return 0;}
```

# Example - switch

84

```
/* Program to evaluate simple expressions of the form
value operator value */
#include <stdio.h>
int main (void) {
    float value1, value2;
    char operator;
    printf ("Type in your expression.\n");
    scanf ("%f %c %f", &value1, &operator, &value2);
    switch (operator) {
        case '+': printf (".2f\n", value1 + value2); break;
        case '-': printf (".2f\n", value1 - value2); break;
        case '*': printf (".2f\n", value1 * value2); break;
        case '/':
            if ( value2 == 0 ) printf ("Division by zero.\n");
            else printf (".2f\n", value1 / value2);
            break;
        default: printf ("Unknown operator.\n"); break;
    }
    return 0;
}
```

# The switch statement

85

```
switch ( expression )
{
    case value1:
        program statement
        program statement
        ...
        break;
    case value2:
        program statement
        program statement
        ...
        break;
    ...
    case valuen:
        program statement
        program statement
        ...
        break;
    default:
        program statement
        program statement
        ...
        break; }
```

The *expression* is successively compared against the values *value1*, *value2*, ..., *valuen*. If a case is found whose value is equal to the value of *expression*, the program statements that follow the case are executed.

The switch test expression must be one with an integer value (including type char) (No float !).  
The case values must be integer-type constants or integer constant expressions (You can't use a variable for a case label !)

# The switch statement (cont)

86

Break can miss !

Statement list on  
a case can miss  
!

```
switch (operator)
{
    ...
    case '*':
    case 'x':
        printf (("%.2f\n", value1 * value2);
        break;
    ...
}
```

# The conditional operator

87

*condition ? expression1 : expression2*

*condition* is an expression that is evaluated first.

If the result of the evaluation of *condition* is TRUE (nonzero), then *expression1* is evaluated and the result of the evaluation becomes the result of the operation. If *condition* is FALSE (zero), then *expression2* is evaluated and its result becomes the result of the operation

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

Equivalent to:

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

# LOOPING STATEMENTS OR CONTROL STRUCTURES: LOOPS



# Lecture 3: Outline

89

- Program Looping
  - ▣ The `for` Statement
  - ▣ Relational Operators
  - ▣ Nested `for` Loops
  - ▣ Increment Operator
  - ▣ Program Input
  - ▣ `for` Loop Variants
  - ▣ The `while` Statement
  - ▣ The `do` Statement
  - ▣ The `break` Statement
  - ▣ The `continue` Statement

# Executing a program

90

- Program = list of statements
  - ▣ **Entrypoint**: the point where the execution starts
  - ▣ **Control flow**: the order in which the individual statements are executed

```
Statement1  
Statement2  
Statement3  
Statement4  
Statement5  
Statement6  
Statement7  
Statement8
```

# Structure of a C program

91

Entry point of a C  
program

```
#include <stdio.h>
int main (void)
{
    int value1, value2, sum;
    value1 = 50;
    value2 = 25;
    sum = value1 + value2;
    printf ("The sum of %i and %i is %i\n", value1, value2, sum);
    return 0;
}
```

Sequential  
flow of control

# Controlling the program flow

92

- Forms of controlling the program flow:
  - Executing a sequence of statements
  - Repeating a sequence of statements (until some condition is met) (looping)
  - Using a test to decide between alternative sequences (branching)

```
Statement1  
Statement2  
Statement3  
Statement4  
Statement5  
Statement6  
Statement7  
Statement8
```

# Program Looping

93

- Looping: doing one thing over and over
- Program loop: a set of statements that is executed repetitively for a number of times
- Simple example: displaying a message 100 times:

```
printf("hello !\n");  
printf("hello !\n");  
printf("hello !\n");  
  
...  
printf("hello !\n");  
printf("hello !\n");
```

```
Repeat 100 times  
    printf("hello !\n");
```

**Program looping:** enables you to develop concise programs containing repetitive processes that could otherwise require many lines of code !

# The need for program looping

94

Example problem: computing triangular numbers.

(The  $n$ -th triangular number is the sum of the integers from 1 through  $n$ )

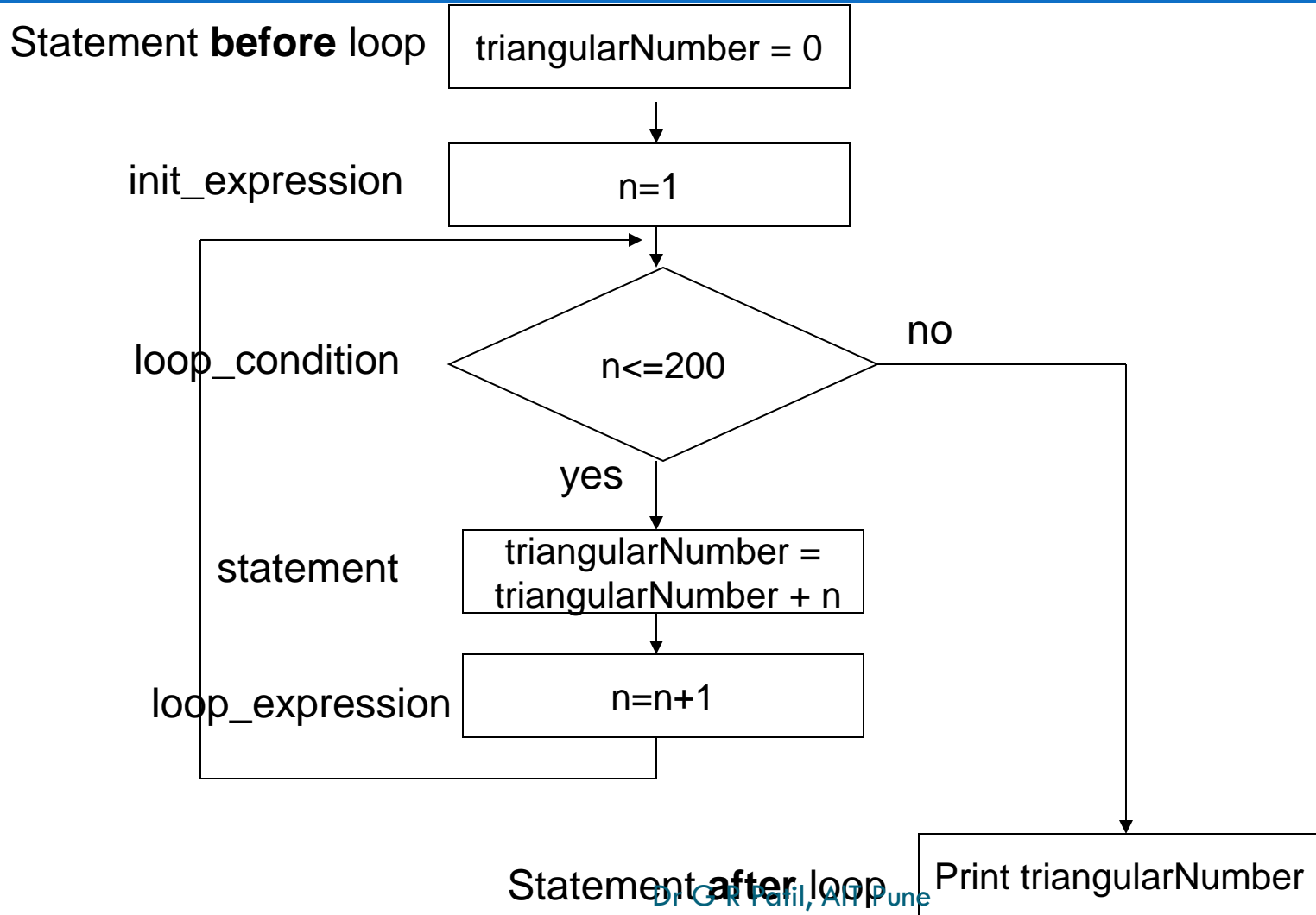
```
#include <stdio.h>
int main (void) {
    int triangularNumber;
    triangularNumber = 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8;
    printf ("The eighth triangular number is %i\n",
            triangularNumber);
    return 0;
}
```

*What if we have to compute the 200-th (1000-th, etc) triangular number ?*

In C: 3 different statements for looping: for, while, do

# Example – 200<sup>th</sup> triangular number

95



# Example - for

96

```
/* Program to calculate the 200th triangular number
Introduction of the for statement */

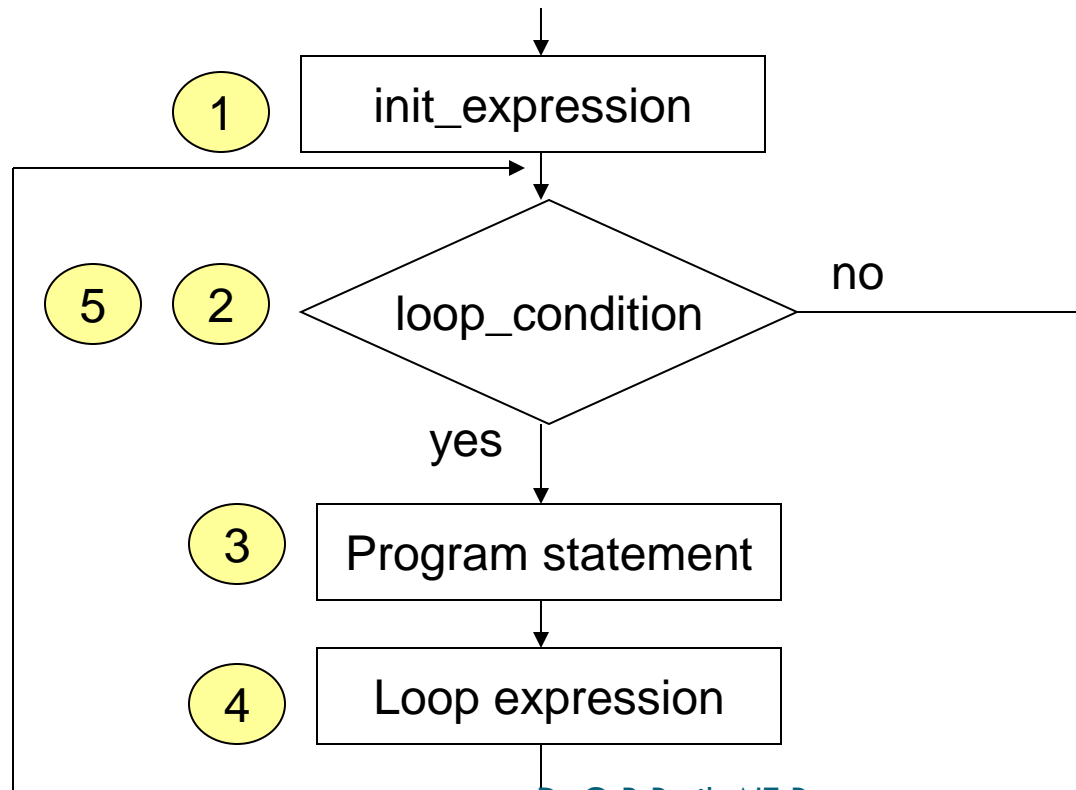
#include <stdio.h>
int main (void)
{
    int n, triangularNumber;
    triangularNumber = 0;
    for ( n = 1; n <= 200; n = n + 1 )
        triangularNumber = triangularNumber + n;
    printf ("The 200th triangular number is %i\n",
                                                    triangularNumber);
    return 0;
}
```



# The for statement

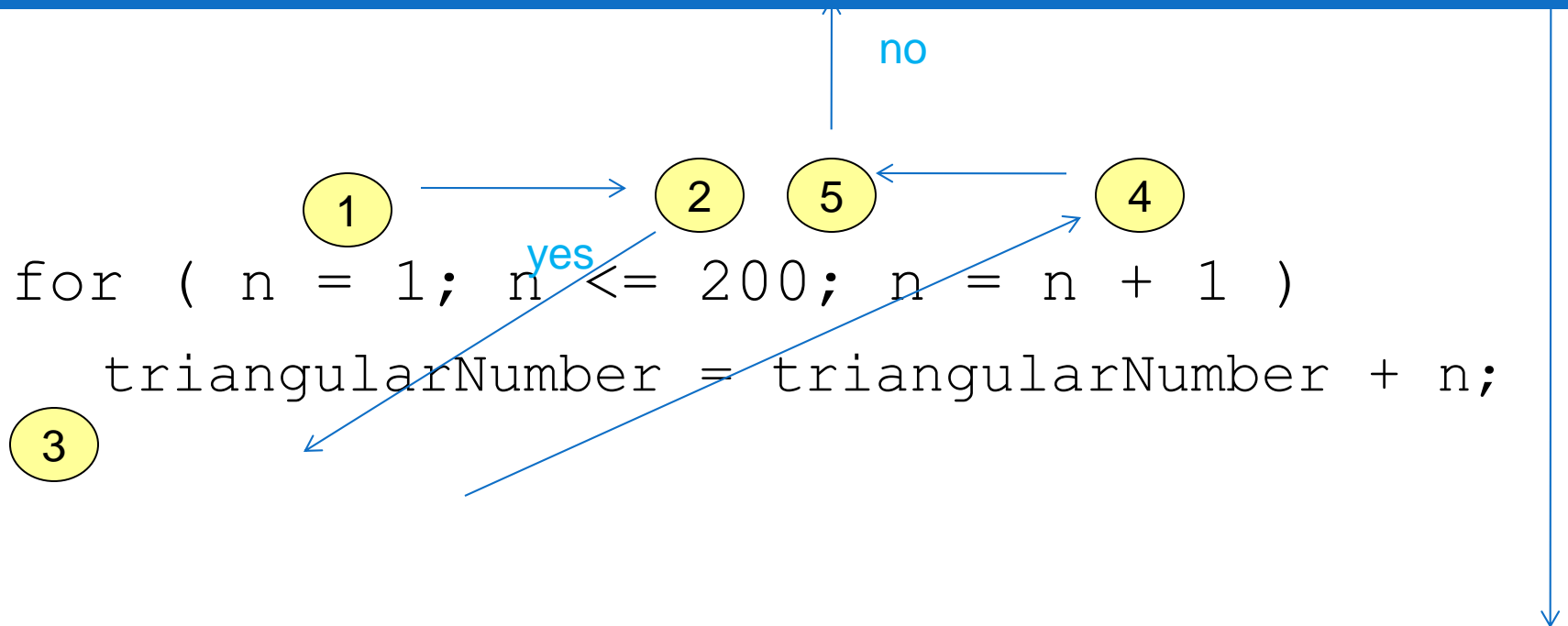
97

```
for ( init_expression; loop_condition; loop_expression )  
    program statement
```



# The for statement

98



# How for works

99

- The execution of a for statement proceeds as follows:
  1. The initial expression is evaluated first. This expression usually sets a variable that will be used inside the loop, generally referred to as an *index* variable, to some initial value.
  2. The looping condition is evaluated. If the condition is not satisfied (the expression is false – has value 0), the loop is immediately terminated. Execution continues with the program statement that immediately follows the loop.
  3. The program statement that constitutes the body of the loop is executed.
  4. The looping expression is evaluated. This expression is generally used to change the value of the index variable
  5. Return to step 2.

# Infinite loops

100

- It's the task of the programmer to design correctly the algorithms so that loops end at some moment !

```
// Program to count 1+2+3+4+5
#include <stdio.h>
int main (void)
{
    int i, n = 5, sum =0;
    for ( i = 1; i <= n; n = n + 1 ) {
        sum = sum + i;
        printf ("%i %i %i\n", i , sum, n);
    }
    return 0;
}
```

**What is wrong here  
?**

**Does the loop end?**

# Relational operators

101

Operator	Meaning
==	Is equal to
!=	Is not equal to
<	Is less than
<=	Is less or equal
>	Is greater than
>=	Is greater or equal

The relational operators have lower precedence than all arithmetic operators:  
 $a < b + c$  is evaluated as  $a < (b + c)$

ATTENTION ! Do not confuse:  
the “is equal to” operator == and the “assignment” operator =

ATTENTION when comparing floating-point values !  
Only < and > comparisons make sense !

# Example – for with a body of 2

102

```
// Program to generate a table of triangular numbers
#include <stdio.h>
int main (void)
{
    int n, triangularNumber;
    printf ("TABLE OF TRIANGULAR NUMBERS\n\n");
    printf (" n Sum from 1 to n\n");
    printf ("--- -----\n");
    triangularNumber = 0;
    for ( n = 1; n <= 10; ++n ) {
        triangularNumber += n;
        printf (" %i %i\n", n, triangularNumber);
    }
    return 0;
}
```

The *body* of the loop  
consists in a *block* of 2  
statements

Dr G R Patil, AIT Pune

# Increment operator

103

- Because addition by 1 is a very common operation in programs, a special operator was created in C for this.
- *Increment operator*: the expression  $++n$  is equivalent to the expression  $n = n + 1$ .
- *Decrement operator*: the expression  $--n$  is equivalent to the expression  $n = n - 1$
- Increment and decrement operators can be placed in front (*prefix*) or after (*postfix*) their operand.
- The *difference between prefix and postfix*:
- Example: if  $n=4$ :
  - $a=n++$  leads to  $a=4, n=5$
  - $a=++n$  leads to  $a=5, n=5$

# Program input

104

```
#include <stdio.h>
int main (void)
{
    int n, number, triangularNumber;
    printf ("What triangular number do you want? ");
    scanf ("%i", &number);
    triangularNumber = 0;
    for ( n = 1; n <= number; ++n )
        triangularNumber += n;
    printf ("Triangular number %i is %i\n", number,
        triangularNumber);

    return 0;
}
```

It's polite to display a message before

Reads integer from keyboard

**Scanf:** similar to printf: first argument contains format characters, next arguments tell where to store the values entered at the keyboard  
More details -> in a later chapter !



# Nested loops

105

```
#include <stdio.h>
int main (void)
{
    int n, number, triangularNumber, counter;
    for ( counter = 1; counter <= 5; ++counter ) {
        printf ("What triangular number do you want? ");
        scanf ("%i", &number);
        triangularNumber = 0;
        for ( n = 1; n <= number; ++n )
            triangularNumber += n;
        printf ("Triangular number %i is %i\n\n", number,
                triangularNumber);
    }
    return 0;
}
```

Remember indentations!

# for loop variants

106

- Multiple expressions (*comma between...*)  
`for(i=0 , j=10 ; i<j ; i++ , j--)`
- Omitting fields (*semicolon have to be still...*)  
`i=0;`  
`for( ; i<10 ; i++ )`
- Declaring variables  
`for(int i=0 ; i<10 ; i++ )`

# The while statement

107

```
while ( expression )  
    program statement
```

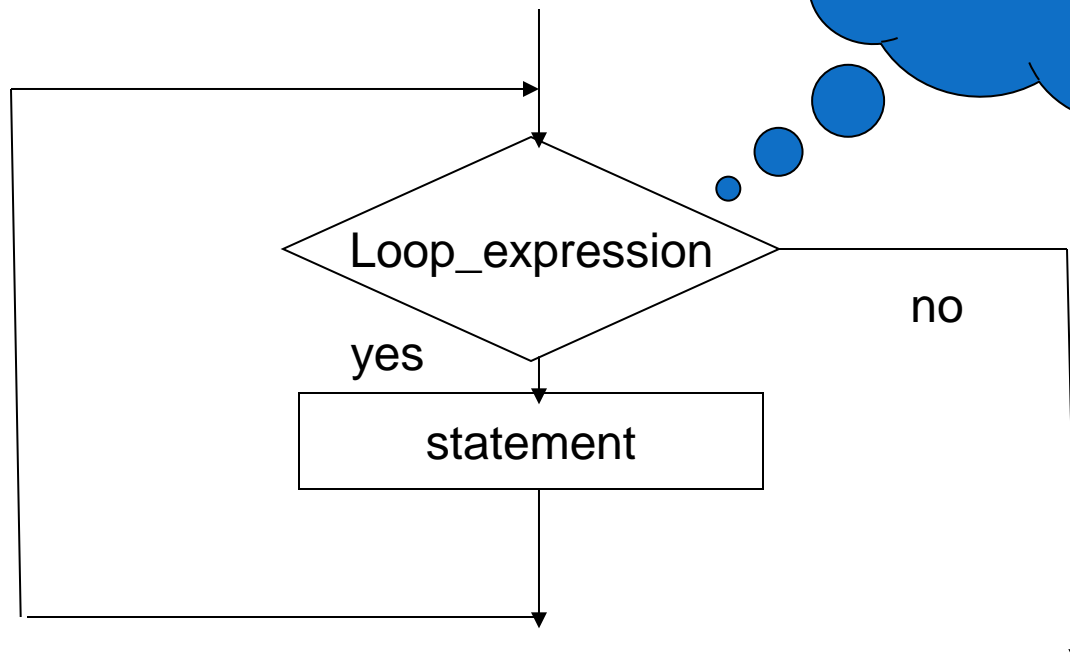
```
while ( number <= 0 ) {  
    printf ("The number must be >0");  
    printf ("Give a new number:  ");  
    scanf ("%i", &number);  
}
```

# The while statement

108

```
while ( expression )  
    program statement
```

Loop with the test  
in the beginning !  
Body might never  
be executed !



# Example:

- A program to find the greatest common divisor of two nonnegative integer values ...

# Example - while

110

```
/* Program to find the greatest common divisor
of two nonnegative integer values */
#include <stdio.h>
int main (void)
{
    int u, v, temp;
    printf ("Please type in two nonnegative integers.\n");
    scanf ("%i%i", &u, &v);
    while ( v != 0 ) {
        temp = u % v;
        u = v;
        v = temp;
    }
    printf ("Their greatest common divisor is %i\n", u);
    return 0;
}
```

# Example:

- A program to print out the digits of a number in reverse order ...

# Example - while

112

```
// Program to reverse the digits of a number
#include <stdio.h>
int main (void)
{
    int number, right_digit;
    printf ("Enter your number.\n");
    scanf ("%i", &number);
    while ( number != 0 ) {
        right_digit = number % 10;
        printf ("%i", right_digit);
        number = number / 10;
    }
    printf ("\n");
    return 0;
}
```



# Example – while not quite OK !

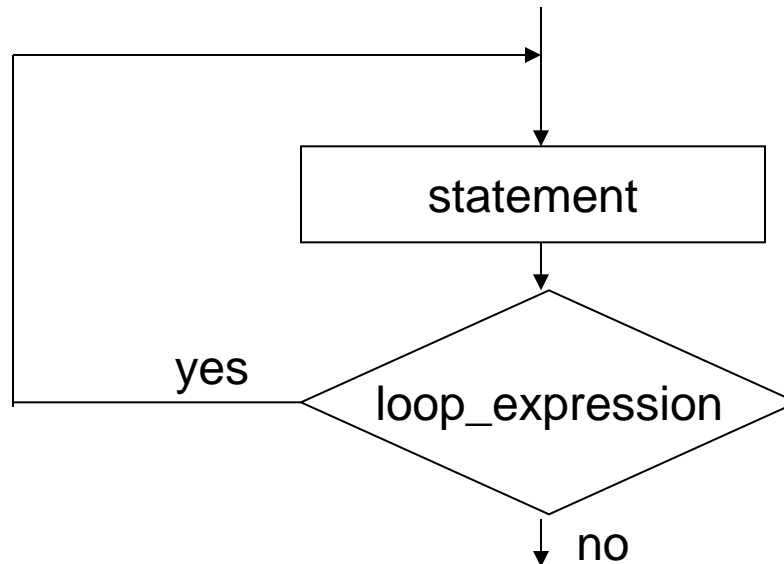
```
// Program to reverse the digits of a number
#include <stdio.h>
int main (void)
{
    int number, right_digit;
    printf ("Enter your number.\n");
    scanf ("%i", &number);
    while ( number != 0 ) {
        right_digit = number % 10;
        printf ("%i", right_digit);
        number = number / 10;
    }
    printf ("\n");
    return 0;
}
```

What happens if you enter  
number=0 ?

# The do statement

114

```
do  
    program statement  
while ( loop_expression );
```



Loop with the test  
at the end !  
Body is executed  
at least once !

# Example – do while

115

```
// Program to reverse the digits of a number
#include <stdio.h>
int main ()
{
    int number, right_digit;
    printf ("Enter your number.\n");
    scanf ("%i", &number);
    do {
        right_digit = number % 10;
        printf ("%i", right_digit);
        number = number / 10;
    }
    while ( number != 0 );
    printf ("\n");
    return 0;
}
```

# Which loop to choose ?

116

- Criteria: Who determines looping
  - ▣ Entry-condition loop -> for, while
  - ▣ Exit-condition loop -> do
- Criteria: Number of repetitions:
  - ▣ Indefinite loops -> while
  - ▣ Counting loops -> for
- In C, you can actually rewrite any while as a for and viceversa !

# Example: while vs for

117

```
#include <stdio.h>
int main (void)
{
    int count = 1;
    while ( count <= 5 ) {
        printf ("%i\n", count);
        ++count;
    }
    return 0;
}
```

```
#include <stdio.h>
int main (void)
{
    int count;
    for ( count=1; count<=5;
          count++ ) {
        printf ("%i\n", count);
    }
    return 0;
}
```

# The break Statement

118

- ❑ Can be used in order to immediately exiting from a loop
- ❑ After a break, following statements in the loop body are skipped and execution continues with the first statement after the loop
- ❑ If a break is executed from within nested loops, only the innermost loop is terminated

# The break statement

119

- Programming style: don't abuse break !!!

...

```
while ( number != 0 ) {  
    // Statements to do something in loop  
    printf("Stop, answer 1:  ");  
    scanf ("%i", &answer);  
    if(answer == 1)  
        break; // very bad idea to do this  
}
```

# The continue statement

120

- Similar to the break statement, but it does not make the loop terminate, just skips to the next iteration



# The continue statement

121

## Continue also not so good style!!!

...

```
while ( number != 0 ) {  
    // Statements to do something in loop  
    printf("Skip next statements answer 1: ");  
    scanf ("%i", &answer);  
    if(answer == 1)  
        continue; // not so good idea..  
    // Statements to do something in loop  
    // If answer was 1 these statements are  
    // not executed. They are skipped.  
    // Go straight to the beginning of while  
}
```

**UNIT I**  
**INTRODUCTION TO C PROGRAMMING**  
**TOPIC 2: ARRAYS, (FUNCTIONS) &**  
**POINTERS**

**DR. G R PATIL**  
**PROFESSOR & HEAD**  
**DEPT OF E&TC ENGG**  
**AIT PUNE**

## 2.1 ARRAYS

# Outline

124

- Arrays
  - ▣ The concept of array
  - ▣ Defining arrays
  - ▣ Initializing arrays
  - ▣ Character arrays
  - ▣ Multidimensional arrays
  - ▣ Variable length arrays

# The concept of array

125

- Array: a set of ordered data items
- You can define a variable called  $x$ , which represents not a *single* value, but an entire *set of values*.
- Each element of the set can then be referenced by means of a number called an *index* number or *subscript*.
- Mathematics: a subscripted variable,  $x_i$ , refers to the *i*th element  $x$  in a set
- C programming: the equivalent notation is  $x[i]$

# Declaring an array

126

- Declaring an array variable:
  - Declaring the **type of elements** that will be contained in the array—such as int, float, char, etc.
  - Declaring the **maximum number of elements** that will be stored inside the array.
    - The C compiler needs this information to determine how much memory space to reserve for the array.)
    - This must be a **constant integer value**
- The range for valid index values in C:
  - First element is at index 0
  - Last element is at index [size-1]
  - It is the task of the programmer to make sure that array elements are referred by indexes that are in the valid range ! The compiler cannot verify this, and it comes to severe runtime errors !

# Arrays - Example

127

## Example:

```
int values[10];
```

Declares an array of 10 elements of type `int`

Using Symbolic Constants for array size:

```
#define N 10
```

```
...
```

```
int values[N];
```

## Valid indexes:

```
values[0]=5;
```

```
values[9]=7;
```

## Invalid indexes:

```
values[10]=3;
```

```
values[-1]=6;
```

In memory: elements of an array are stored  
at consecutive locations

values [0]

values [1]

values [2]

values [3]

values [4]

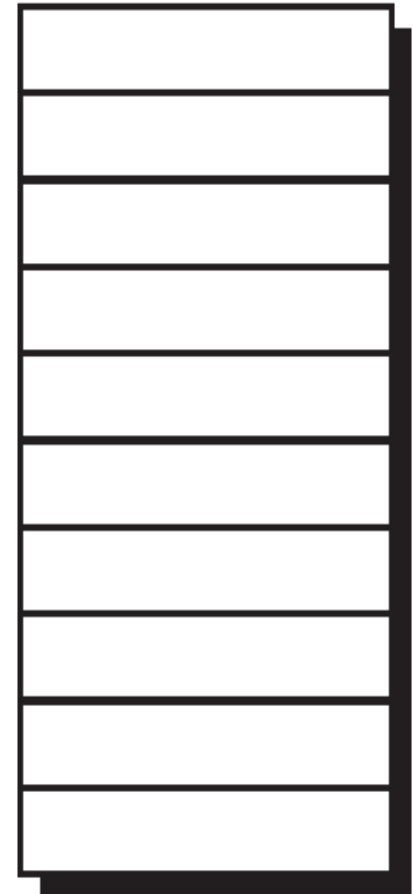
values [5]

values [6]

values [7]

values [8]

values [9]



# Arrays - Example

128

```
#include <stdio.h>
#define N 6
int main (void)
{
    int values[N];
    int index;
    for ( index = 0; index < N; ++index ) {
        printf("Enter value of element #%i \n", index);
        scanf("%i", &values[index]);
    }
    for ( index = 0; index < N; ++index )
        printf("values[%i] = %i\n", index, values[index]);
    return 0;
}
```

Using symbolic constants for array size makes program more general

Typical loop for processing all elements of an array



# What goes wrong if an index goes out of range ?

129

```
#include <stdio.h>
#define NA    4
#define NB    7
int main (void) {
    int b[NB],a[NA];
    int index;
    for ( index = 0; index < NB; ++index )
        b[index]=10+index;
    for ( index = 0; index < NA+2; ++index )
        a[index]=index;

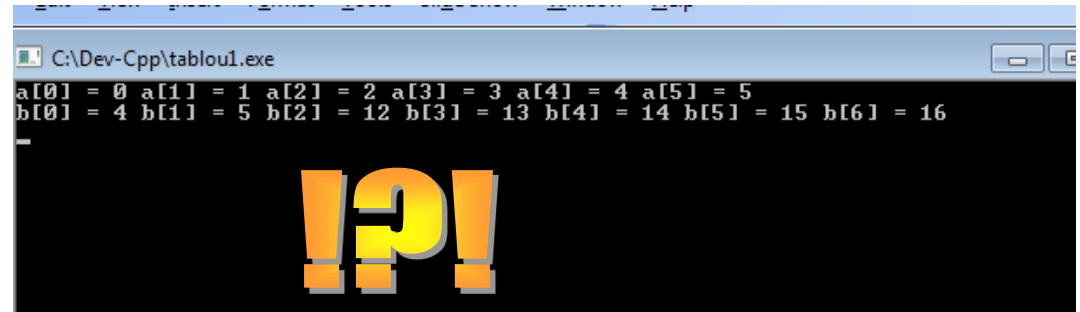
    for ( index = 0; index < NA+2; ++index )
        printf ("a[%i] = %i ", index, a[index]);
    printf("\n");
    for ( index = 0; index < NB; ++index )
        printf ("b[%i] = %i ", index, b[index]);
    printf("\n");
    return 0;}
```

# What goes wrong if an index goes out of range ?

130

```
#include <stdio.h>
#define NA    4
#define NB    7
int main (void) {
    int b[NB],a[NA];
    int index;
    for ( index = 0; index < NB; ++index )
        b[index]=10+index;
    for ( index = 0; index < NA+2; ++index )
        a[index]=index;

    for ( index = 0; index < NA+2; ++index )
        printf ("a[%i] = %i ", index, a[index]);
    printf("\n");
    for ( index = 0; index < NB; ++index )
        printf ("b[%i] = %i ", index, b[index]);
    printf("\n");
    return 0;
}
```



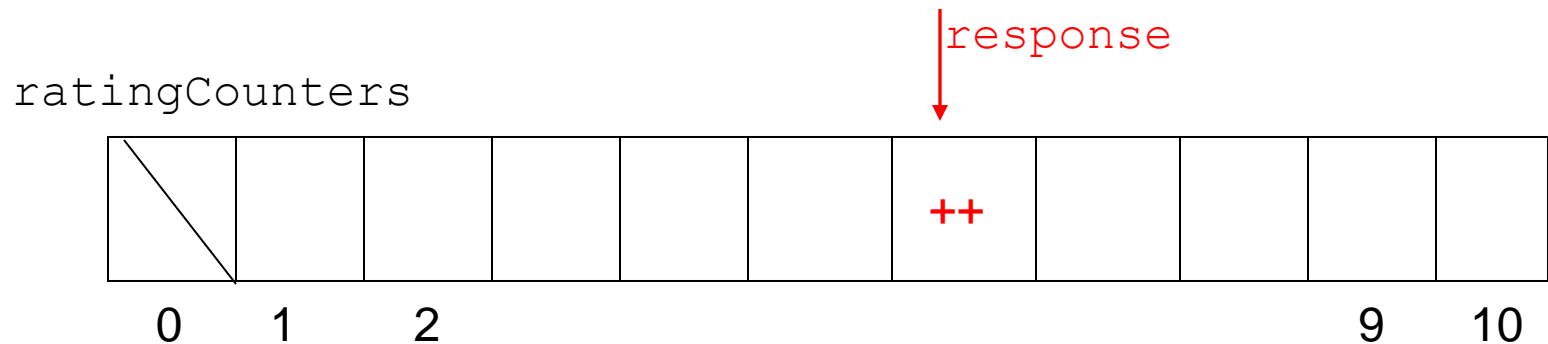
# Exercise

131

- Suppose you took a survey to discover how people felt about a particular television show and you asked each respondent to rate the show on a scale from 1 to 10, inclusive. After interviewing 5,000 people, you accumulated a list of 5,000 numbers. Now, you want to analyze the results.
- One of the first pieces of data you want to gather is a table showing the distribution of the ratings: you want to know how many people rated the show a 1, how many rated it a 2, and so on up to 10.
- Develop a program to count the number of responses for each rating.

# Exercise: Array of counters

132



`ratingCounters[i]` = how many persons rated the show an i

# Exercise: Array of counters

133

```
#include <stdio.h>
int main (void) {
    int ratingCounters[11], i, response;
    for ( i = 1; i <= 10; ++i )
        ratingCounters[i] = 0;
    printf ("Enter your responses\n");
    for ( i = 1; i <= 20; ++i ) {
        scanf ("%i", &response);
        if ( response < 1 || response > 10 )
            printf ("Bad response: %i\n", response);
        else
            ++ratingCounters[response];
    }
    printf ("\n\nRating Number of Responses\n");
    printf ("-----\n");
    for ( i = 1; i <= 10; ++i )
        printf ("%4i%14i\n", i, ratingCounters[i]);
    return 0;
}
```

# Exercise: Fibonacci numbers

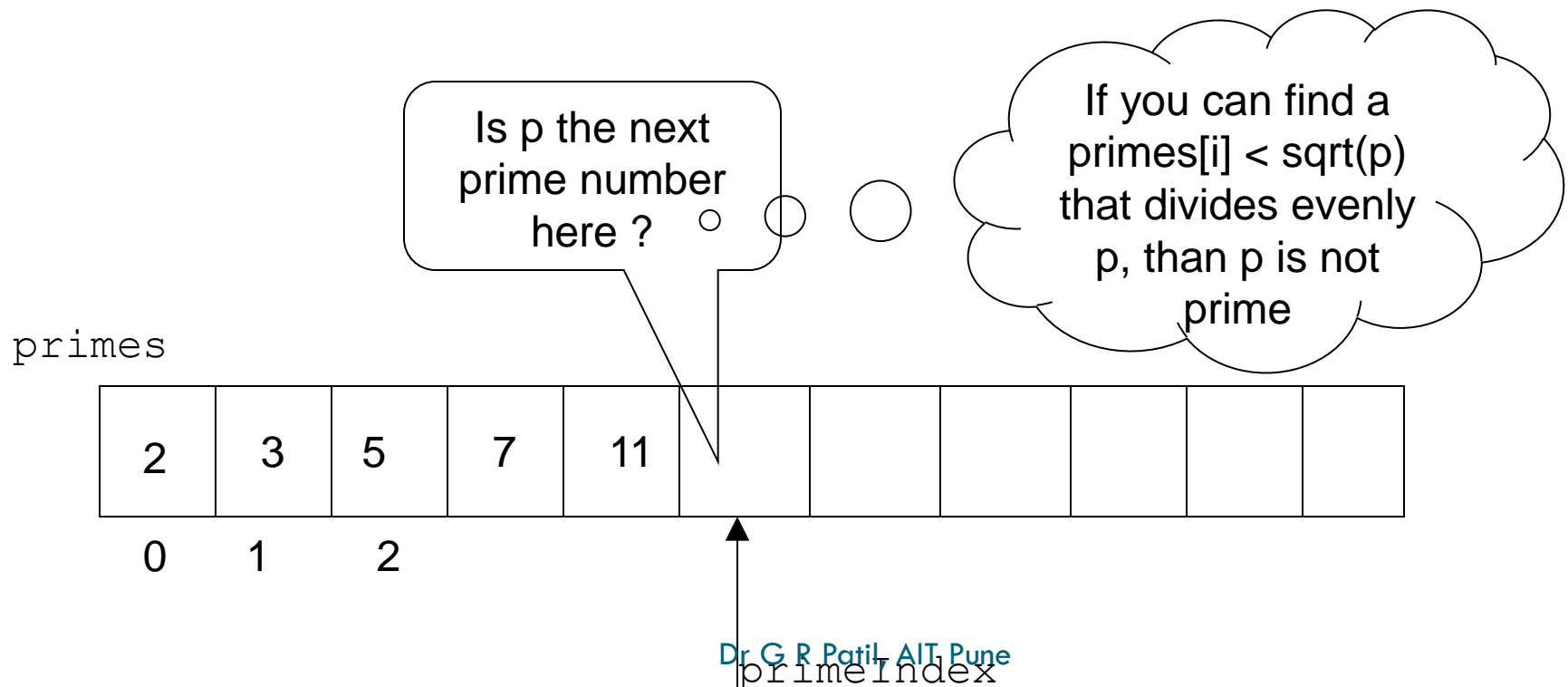
134

```
// Program to generate the first 15 Fibonacci numbers
#include <stdio.h>
int main (void)
{
    int Fibonacci[15], i;
    Fibonacci[0] = 0; // by definition
    Fibonacci[1] = 1; // ditto
    for ( i = 2; i < 15; ++i )
        Fibonacci[i] = Fibonacci[i-2] + Fibonacci[i-1];
    for ( i = 0; i < 15; ++i )
        printf ("%i\n", Fibonacci[i]);
    return 0;
}
```

# Exercise: Prime numbers

135

- An improved method for generating prime numbers involves the notion that a number  $p$  is prime if it is not evenly divisible by any other prime number
- Another improvement: a number  $p$  is prime if there is no prime number smaller than its square root, so that it is evenly divisible by it



# Exercise: Prime numbers

136

```
#include <stdio.h>
#include <stdbool.h>
// Modified program to generate prime numbers
int main (void) {
    int p, i, primes[50], primeIndex = 2;
    bool isPrime;
    primes[0] = 2;
    primes[1] = 3;
    for ( p = 5; p <= 50; p = p + 2 ) {
        isPrime = true;
        for ( i = 1; isPrime && p / primes[i] >= primes[i]; ++i )
            if ( p % primes[i] == 0 )
                isPrime = false;
        if ( isPrime == true ) {
            primes[primeIndex] = p;
            ++primeIndex;
        }
    }
    for ( i = 0; i < primeIndex; ++i )
        printf ("%i ", primes[i]);
    printf ("\n");
    return 0;
}
```



# Initializing arrays

137

- `int counters[5] = { 0, 0, 0, 0, 0 };`
- `char letters[5] = { 'a', 'b', 'c', 'd', 'e' };`
- `float sample_data[500] = { 100.0, 300.0, 500.5 };`
- The C language allows you to define an array without specifying the number of elements. If this is done, the size of the array is determined automatically based on the number of initialization elements: `int counters[] = { 0, 0, 0, 0, 0 };`

# Character arrays

138

```
#include <stdio.h>
int main (void)
{
    char word[] = { 'H', 'e', 'l', 'l', 'o', '!' };
    int i;
    for ( i = 0; i < 6; ++i )
        printf ("%c", word[i]);
    printf ("\n");
    return 0;
}
```

a special case of character arrays: the character *string* type => in a later chapter

# Example: conversion to base b

139

- Convert a number from base 10 into a base b, b in range [2..16]
- Example: convert number from base 10 into base b=2

	Number	Number % 2	Number / 2
Step1:	10	0	5
Step2:	5	1	2
Step3:	2	0	1
Step4:	1	1	0

# Example: Base conversion using arrays

140

```
// Program to convert a positive integer to another base
#include <stdio.h>
int main (void)
{
    const char baseDigits[16] = {
        '0', '1', '2', '3', '4', '5', '6', '7',
        '8', '9', 'A', 'B', 'C', 'D', 'E', 'F' };
    int convertedNumber[64];
    long int numberToConvert;
    int nextDigit, base, index = 0;
    // get the number and the base
    printf ("Number to be converted? ");
    scanf ("%ld", &numberToConvert);
    printf ("Base? ");
    scanf ("%i", &base);
```

# Example continued

141

```
// convert to the indicated base
do {
    convertedNumber[index] =
        numberToConvert % base;
    ++index;
    numberToConvert = numberToConvert /
        base;
}
while ( numberToConvert != 0 );
// display the results in reverse order
printf ("Converted number = ");
for (--index; index >= 0; --index ) {
    nextDigit = convertedNumber[index];
    printf ("%c", baseDigits[nextDigit]);
}
printf ("\n");
return 0;
}
```

# Multidimensional arrays

142

- C language allows arrays of any number of dimensions
- Two-dimensional array: matrix

```
int M[4][5]; // matrix, 4 rows, 5 columns  
M[i][j] - element at row i, column j
```

```
int M[4][5] = {  
    { 10, 5, -3, 17, 82 },  
    { 9, 0, 0, 8, -7 },  
    { 32, 20, 1, 0, 14 },  
    { 0, 0, 8, 7, 6 }  
};  
int M[4][5] = { 10, 5, -3, 17, 82, 9, 0, 0, 8, -7, 32, 20,  
    1, 0, 14, 0, 0, 8, 7, 6 };
```

# Example: Typical matrix processing

143

```
#define N 3
#define M 4
int main(void) {
    int a[N][M];
    int i,j;
    /* read matrix elements */
    for(i = 0; i < N; i++)
        for(j = 0; j < M; j++) {
            printf("a[%d][%d] = ", i, j);
            scanf("%d", &a[i][j]);
        }
    /* print matrix elements */
    for(i = 0; i < N; i++) {
        for(j = 0; j < M; j++)
            printf("%5d", a[i][j]);
        printf("\n");
    }
    return 0;
}
```

# Example: Dealing with variable numbers of elements

144

```
#include <stdio.h>
#define NMAX 4

int main(void) {
    int a[NMAX];
    int n;
    int i;
    printf("How many elements(maximum %d)?\n", NMAX);
    scanf("%d", &n);
    if (n > NMAX) {
        printf("Number too big !\n");
        return 1;
    }
    for(i = 0; i < n; i++)
        scanf("%d", &a[i]);
    for(i = 0; i < n; i++)
        printf("%5d", a[i]);
    printf("\n");
    return 0;
}
```



# Variable length arrays

145

- ❑ A feature introduced by C99
- ❑ It was NOT possible in ANSI C !
- ❑ `int a[n];`
- ❑ The array `a` is declared to contain `n` elements. This is called a *variable length array* because *the size of the array is specified by a variable and not by a constant expression*.
- ❑ The value of the variable must be known at runtime when the array is created => the array variable will be declared later in the block of the program
- ❑ Possible in C99: variables can be declared anywhere in a program, as long as the declaration occurs before the variable is first used.
- ❑ A similar effect of variable length array could be obtained in ANSI C using *dynamic memory allocation* to allocate space for arrays while a program is executing.

# Example: Variable length arrays

146

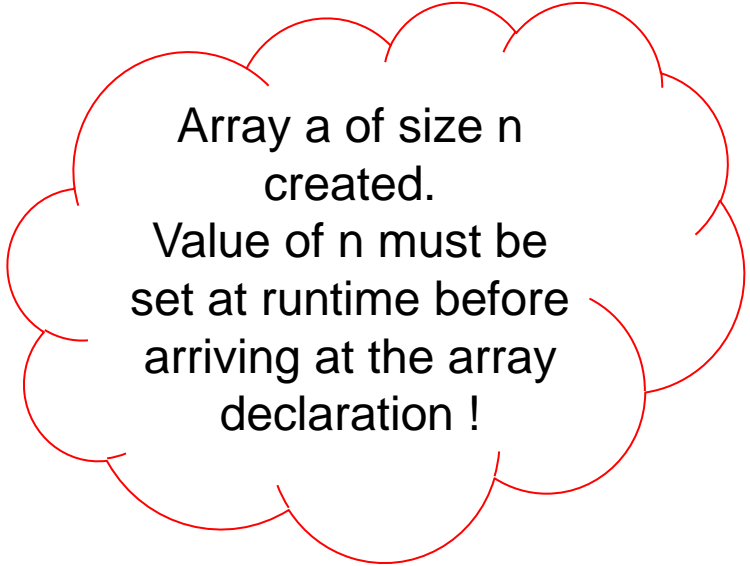
```
#include <stdio.h>

int main(void) {
    int n;
    int i;

    printf("How many elements do you have ? \n");
    scanf("%d", &n);

    int a[n];

    for(i = 0; i < n; i++)
        scanf("%d", &a[i]);
    for(i = 0; i < n; i++)
        printf("%5d", a[i]);
    printf("\n");
    return 0;
}
```



Array a of size n  
created.  
Value of n must be  
set at runtime before  
arriving at the array  
declaration !

# Example: variable length arrays

147

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

```
int main(void) {
```

```
    int n;
```

```
    int i;
```

```
    n=7;
```

```
    int a[n];
```

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

```
        a[i]=i
```

```
    n=20;
```

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

```
        a[i]=2*i;
```

```
    printf("\n");
```

```
    return 0;
```

```
}
```

Array a  
created of size  
7

Wrong!

Variable-length array  
does NOT mean that  
you can modify the  
length of the array  
after you create it !  
Once created, a VLA  
keeps the same size  
!

## 2.2 FUNCTIONS

# Outline

149

## □ Functions

- ▣ Defining a Function
- ▣ Arguments and Local Variables
  - Automatic Local Variables
- ▣ Returning Function Results
- ▣ Declaring a Function Prototype
- ▣ Functions and Arrays
  - Arrays as parameters
  - Sorting Arrays
  - Multidimensional Arrays
- ▣ Global Variables
- ▣ Automatic and Static Variables
- ▣ Recursive Functions

# What is a function

150

- ❑ A function in C: is a self-contained unit of program code designed to accomplish a particular task.
- ❑ The concept has some equivalent in all high-level programming languages: *functions, subroutines, and procedures*
- ❑ The use of a function: a "black box"
  - ❑ defined in terms of the information that goes in (its input) and the value or action it produces (its output).
  - ❑ what goes on inside the black box is not your concern, unless you are the one who has to write the function.
  - ❑ Think on how you used functions printf, scanf, getchar !
- ❑ What kind of “output” comes out from a function black box ?
  - ❑ Some functions find a **value** for a program to use. Example: getchar() returns to the program the next character from the standard input buffer.
  - ❑ Some functions cause an **action** to take place. Example: printf() causes data to be printed on the screen
  - ❑ In general, a function can both produce actions and provide values.

# Defining a function

151

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

```
void printMessage (void)
{
    printf ("Programming is fun.\n");
}
```

*Function  
Definition*

*-occurs ONE time for all  
-outside other functions*

```
int main (void)
{
    printMessage ();
    printMessage ();
    return 0;
}
```

*Function*

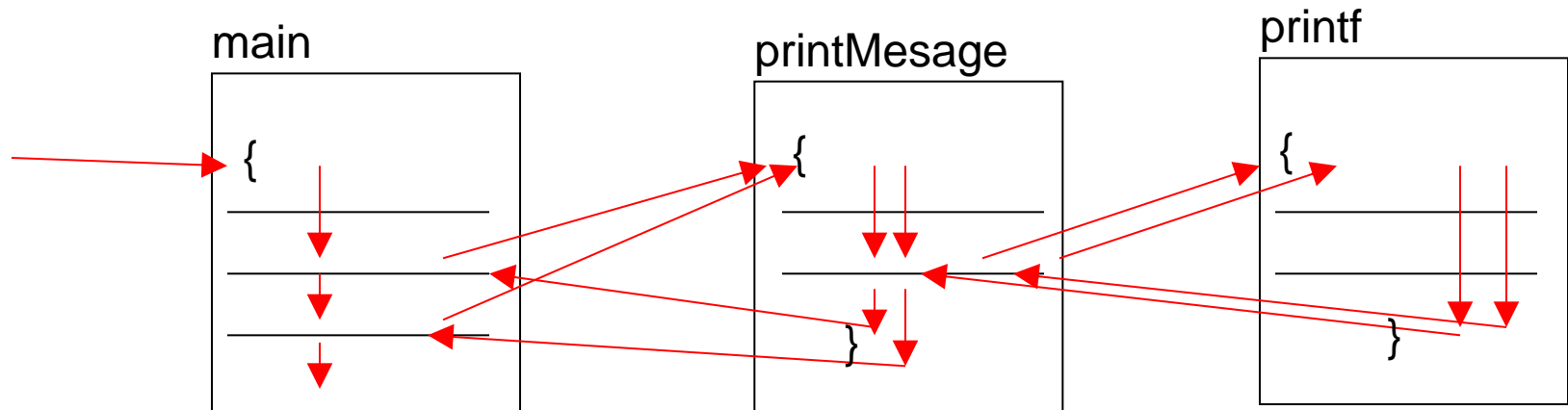
*calls (invocations)*

*-occurs ANY (0-N) times*

*-statement inside (other) functions body*

# Transfer of control flow

152



When a function call is executed, program execution is transferred directly to the indicated function. After the called routine is finished (as signaled by the closing brace) the program *returns* to the calling routine, where program execution continues at the point where the function call was executed.



# Function definitions

153

General form of function definition:

```
return-type function-name(argument declarations)  
{  
    declarations and statements  
}
```

```
return-type           arguments  
void printMessage ( void )  
{  
    printf ("Programming is fun.\n");  
}
```

# Function prototype

154

- The first line of the function definition
- Contains everything that others (other functions) need to know about the function in order to use it (call it)
- `void printMessage (void)`
- `void calculateTriangularNumber (int n)`

## Function prototype

```
return-type function-name(argument declarations)  
{  
    declarations and statements  
}
```

# Function arguments

155

- arguments (parameters): a kind of input for the function blackbox
- **In the function definition:** *formal arguments* (formal parameters)
  - Formal parameter: a name that is used inside the function body to refer to its argument
- **In the function call:** *actual arguments* (actual parameters)
  - The actual arguments are values are assigned to the corresponding formal parameters.
  - The actual argument can be a constant, a variable, or an even more elaborate expression.
  - The actual argument is evaluated, and its *value is copied* to the corresponding formal parameter for the function.
    - Because the called function works with data copied from the calling function, the original data in the calling function is protected from whatever manipulations the called function applies to the copies.

# Example: arguments

156

```
// Function to calculate the nth triangular number
#include <stdio.h>
void calculateTriangularNumber (int n) formal argument
{
    int i, triangularNumber = 0; local variables
    for ( i = 1; i <= n; ++i )
        triangularNumber += i;
    printf ("Triangular number %i is %i\n", n, triangularNumber);
}
int main (void)
{
    calculateTriangularNumber (10); actual argument
    calculateTriangularNumber (20);
    calculateTriangularNumber (50);
    return 0;
}
```

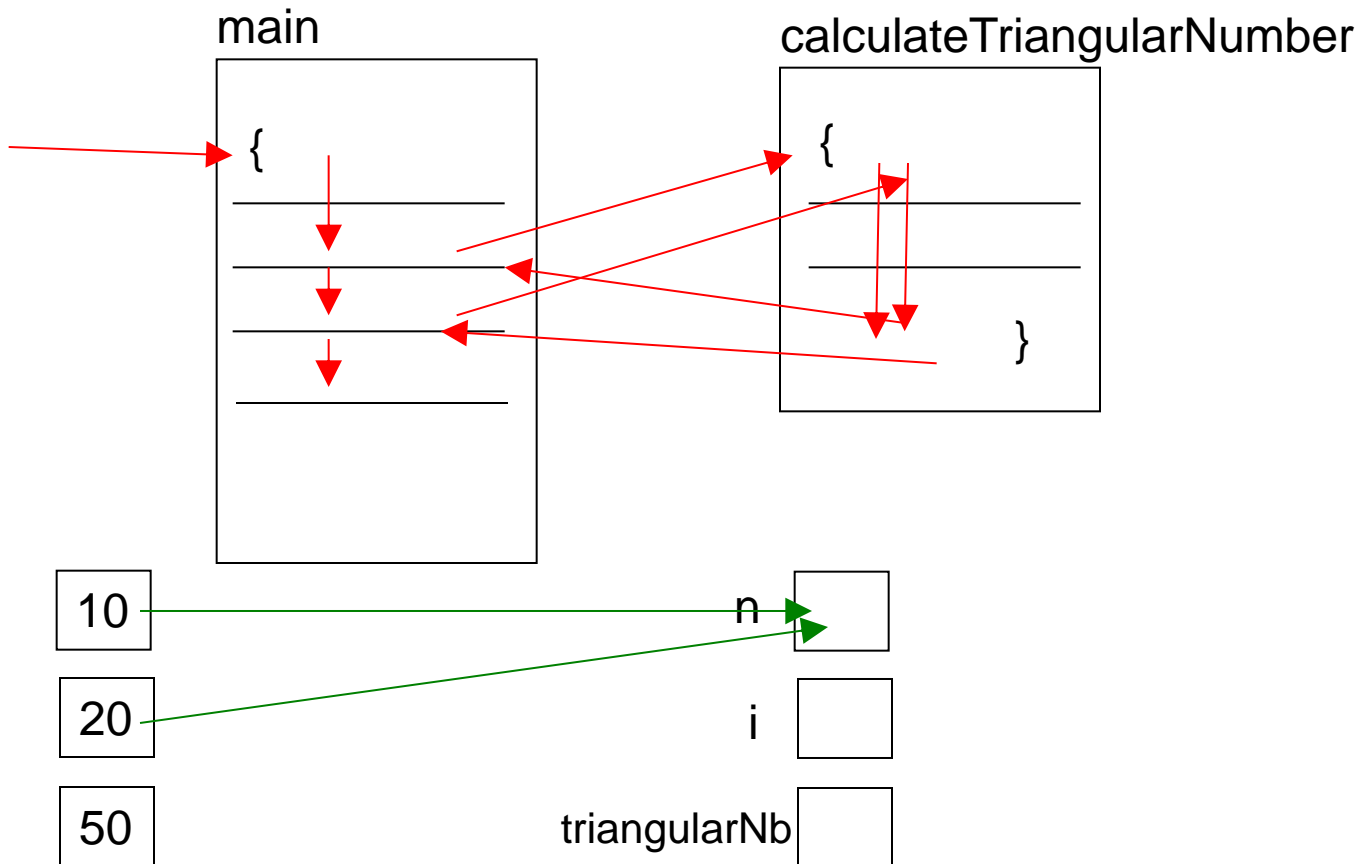
# Arguments and local variables

157

- Variables defined inside a function: *automatic local* variables
  - ▣ they are automatically “created” each time the function is called
  - ▣ their values are local to the function:
    - The value of a local variable can only be accessed by the function in which the variable is defined
    - Its value cannot be accessed by any other function.
    - If an initial value is given to a variable inside a function, that initial value is assigned to the variable each time the function is called.
- Formal parameters: behave like local variables, private to the function.
- **Lifetime**: Period of time when memory location is allocated
- **Scope**: Region of program text where declaration is visible
- Scope: local variables and formal parameters => only in the body of the function
  - ▣ Local variable *i* in function `calculateTriangularNumber` is different from a variable *i* defined in another function (including `main`)
  - ▣ Formal parameter *n* in function `calculateTriangularNumber` is different from a variable *n* defined in another function

# Automatic local variables

158



# Example: scope of local variables

159

```
#include <stdio.h>
void f1 (float x)  {
    int n=6;
    printf("%f \n", x+n);
}
int f2(void) {
    float n=10;
    printf("%f \n",n);
}
int main (void)
{
    int n=5;
    f1(3);
    f2();
    return 0;
}
```

# Arguments are passed by copying values !

160

- In a function call, the actual argument is evaluated, and its **value is copied** to the corresponding formal parameter for the function.
- ▣ Because the called function works with data copied from the calling function, the original data in the calling function is protected from whatever manipulations the called function applies to the copies



# Example: arguments

161

```
#include <stdio.h>
void gcd (int u, int v)
{
    int temp;
    printf ("The gcd of %i and %i is ", u, v);
    while ( v != 0 ) {
        temp = u % v;
        u = v;
        v = temp;
    }
    printf ("%i\n", u);
}
int main (void)
{
    gcd (150, 35);
    gcd (1026, 405);
    gcd (83, 240);
    return 0;
}
```

# Example: arguments are passed by copying values !

162

```
#include <stdio.h>
void gcd (int u, int v)
{
```

```
    int temp;
    printf ("The gcd of %i and %i is ", u, v);
```

```
    while ( v != 0 ) {
        temp = u % v;
```

```
        u = v;
        v = temp;
```

```
    }
```

```
    printf ("%i\n", u);
```

```
}
```

```
int main (void)
```

```
{
```

```
    int x=10,y=15;
```

```
    gcd (x, y);
```

```
    printf("x=%i y=%i \n",x,y);
```

```
    return 0;
```

```
}
```

The formal parameters u and v are assigned new values in the function

The actual parameters x and y are not changed !

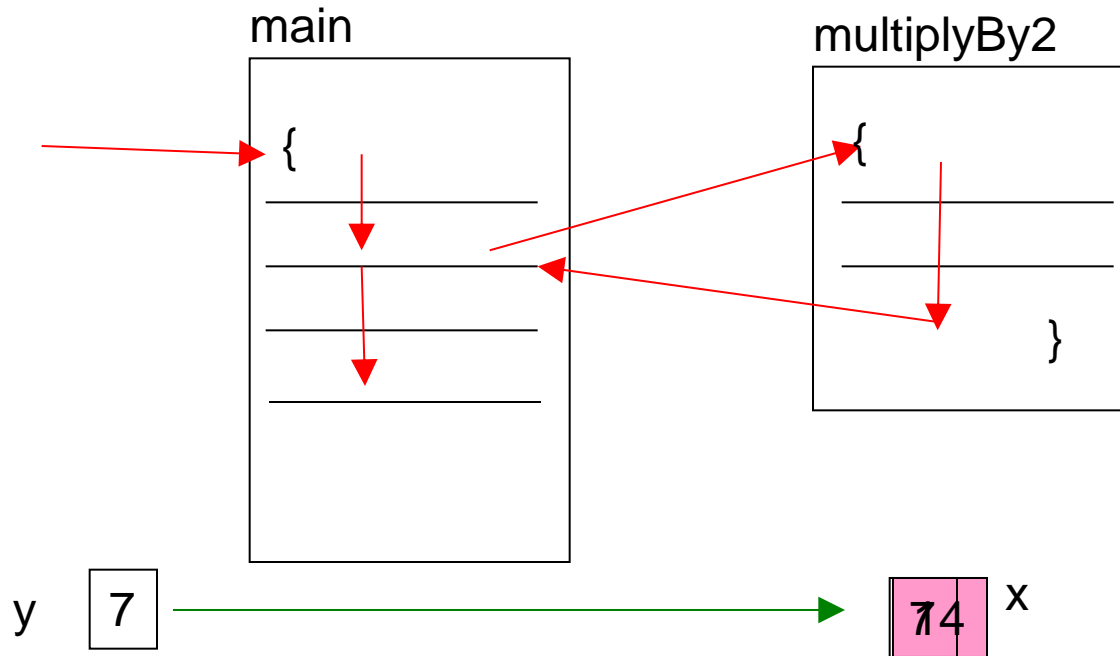
# Example: arguments are passed by copying values !

163

```
#include <stdio.h>
void multiplyBy2 (float x)
{
    printf("parameter at start: %.2f, at %p \n",x, &x);
    x*=2;
    printf("parameter at end: %.2f, at %p \n",x, &x);
}
int main (void)
{
    float y = 7;
    printf ("y before call: %.2f, at %p \n", y, &y);
    multiplyBy2 (y);
    printf ("y after call: %.2f, at %p \n", y, &y);
    return 0;
}
```

# Arguments by copying

164



# Returning function results

165

- A function in C can optionally return a single value
- `return expression;`
- The value of *expression* is returned to the calling function. If the type of *expression* does not agree with the return type declared in the function declaration, its value is automatically converted to the declared type before it is returned.
- A simpler format for declaring the return statement is as follows:
- `return;`
- Execution of the simple return statement causes program execution to be immediately returned to the calling function. This format can only be used to return from a function that does not return a value.
- If execution proceeds to the end of a function and a return statement is not encountered, it returns as if a return statement of this form had been executed. Therefore, in such a case, no value is returned.
- If the declaration of the type returned by a function is omitted, the C compiler assumes that the function returns an int !

# Return example

166

```
void printMessage (void)
{
    printf ("Programming is fun.\n");
    return;
}
```

# Example: function result

167

```
/* Function to find the greatest common divisor of two
nonnegative integer values and to return the result */
#include <stdio.h>
int gcd (int u, int v)
{
    int temp;
    while ( v != 0 ) {
        temp = u % v;
        u = v;
        v = temp;
    }
    return u;
}
int main (void)
{
    int result;
    result = gcd (150, 35);
    printf ("The gcd of 150 and 35 is %i\n", result);
    result = gcd (1026, 405);
    printf ("The gcd of 1026 and 405 is %i\n", result);
    printf ("The gcd of 83 and 240 is %i\n", gcd (83, 240));
    return 0;
}
```

# Function declaration

168

- a function prototype—a declaration that states the return type, the number of arguments, and the types of those arguments.
- Useful mechanism when the called function is defined after the calling function
- The **prototype** of the called function is everything the compiler needs in order to be able to **compile** the calling function
- In order to produce the **executable program**, of course that also the whole **definition** of the function body is needed, but this occurs later, in the process of **linking**



# Example: function declaration

169

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

```
void printMessage (void) ;
```

*Function declaration (prototype)  
(has to be before the calling function)*

```
int main (void)
```

```
{
```

```
    printMessage ();
```

```
    printMessage ();
```

```
    return 0;
```

```
}
```

*Function calls*

```
void printMessage (void)
```

```
{
```

```
    printf ("Programming is fun.\n");
```

```
}
```

*Function definition  
(can be after the  
calling function)*

# #include explained

170

- ❑ `#include <stdio.h>`
- ❑ `#include <filename>` is a preprocessor directive
- ❑ Preprocessor: a first step in the C compilation process
- ❑ Preprocessor statements are identified by the pound sign `#` that must be the first nonspace character of a line
- ❑ The `#include` directive will insert in place the contents of the specified file
- ❑ These files usually have names that end with `.h` (header files)
- ❑ Header files usually contain declarations and definitions that are used by several programs
- ❑ `<stdio.h>` contains the **declarations** for the standard input output functions `printf`, `scanf`, `getchar`, etc. This is why any program that uses these functions has to include `<stdio.h>`

# Function declaration style added in C99

171

Besides the ANSI C style for function declaration, C99 accepts also this style

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

```
int main (void)
```

```
{
```

```
    void printMessage (void) ;
```

```
    printMessage ();
```

```
    printMessage ();
```

```
    return 0;
```

```
}
```

*Function declaration  
(has to be before the function call  
can be inside a function body)*

*Function calls*

```
void printMessage (void)
```

```
{
```

```
    printf ("Programming is fun.\n");
```

```
}
```

*Function definition  
(can be after the  
calling function)*

# Examples: function declarations

172

In a function declaration you have to specify the argument type inside the parentheses, and not its name. You can optionally specify a “dummy” name for formal parameters after the type if you want.

```
int gcd (int u, int v);
```

Or

```
int gcd (int, int);
```

```
void calculateTriangularNumber (int n);
```

Or

```
void calculateTriangularNumber (int);
```

# Passing arrays as parameters

173

- A whole array can be one parameter in a function
- ***In the function declaration, you can then omit the specification of the number of elements contained in the formal parameter array.***
  - The C compiler actually ignores this part of the declaration anyway; all the compiler is concerned with is the fact that an array is expected as an argument to the function and not how many elements are in it.
- Example: a function that returns the minimum value from an array given as parameter
  - `int minimum (int values[10]);`
    - We must modify the function definition if a different array size is needed !
  - `int minimum (int values[]);`
    - Syntactically OK, but how will the function know the actual size of the array ?!
  - `int minimum (int values[], int numberOfElements);`

# Computing the minimum/maximum

174

```
int values[10];  
int minValue, i;  
minValue = values[0];  
for ( i = 1; i < 10; ++i )  
    if ( values[i] < minValue )  
        minValue = values[i];
```

```
#include <limits.h>  
  
int minValue, i;  
minValue = INT_MIN;  
for ( i = 0; i < 10; ++i )  
    if ( values[i] < minValue )  
        minValue = values[i];
```

# Example: Passing arrays as parameters

175

```
// Function to find the minimum value in an array
#include <stdio.h>
int minimum (int values[], int n) {
    int minValue, i;
    minValue = values[0];
    for ( i = 1; i < n; ++i )
        if ( values[i] < minValue )
            minValue = values[i];
    return minValue;
}
int main (void) {
    int scores[10], i, minScore;
    printf ("Enter 10 scores\n");
    for ( i = 0; i < 10; ++i )
        scanf ("%i", &scores[i]);
    minScore = minimum (scores);
    printf ("\nMinimum score is %i\n", minScore);
    return 0;
}
```

# Example: No size specified for formal parameter array

176

```
// Function to find the minimum value in an array
#include <stdio.h>
int minimum (int values[], int numberOfElements)
{
    int minValue, i;
    minValue = values[0];
    for ( i = 1; i < numberOfElements; ++i )
        if ( values[i] < minValue )
            minValue = values[i];
    return minValue;
}
int main (void)
{
    int array1[5] = { 157, -28, -37, 26, 10 };
    int array2[7] = { 12, 45, 1, 10, 5, 3, 22 };
    printf ("array1 minimum: %i\n", minimum (array1, 5));
    printf ("array2 minimum: %i\n", minimum (array2, 7));
    return 0;
}
```



# Array parameters are passed by reference

## !

177

- Parameters of non-array type: passed by copying values
- **Parameters of array type: passed by reference**
  - ▣ the entire contents of the array is *not* copied into the formal parameter array.
  - ▣ the function gets passed information describing *where* in the computer's memory the original array is located.
  - ▣ *Any changes made to the formal parameter array by the function are actually made to the original array passed to the function, and not to a copy of the array.*
  - ▣ *This change remains in effect even after the function has completed execution and has returned to the calling routine.*

# Example: Array parameters are passed by reference !

178

```
#include <stdio.h>
void multiplyBy2 (float array[], int n)
{
    int i;
    for ( i = 0; i < n; ++i )
        array[i] *= 2;
}
int main (void)
{
    float floatVals[4] = { 1.2f, -3.7f, 6.2f, 8.55f };
    int i;
    multiplyBy2 (floatVals, 4);
    for ( i = 0; i < 4; ++i )
        printf ( "%.2f ", floatVals[i] );
    printf ( "\n" );
    return 0;
}
```

# Sorting arrays

179

```
// Program to sort an array of integers
// into ascending order

#include <stdio.h>
void sort (int a[], int n)
{
    int i, j, temp;
    for ( i = 0; i < n - 1; ++i )
        for ( j = i + 1; j < n; ++j )
            if ( a[i] > a[j] ) {
                temp = a[i];
                a[i] = a[j];
                a[j] = temp;
            }
}
```

# Sorting arrays - continued

180

```
void sort (int a[], int n);

int main (void)
{
    int i;
    int array[16] = { 34, -5, 6, 0, 12, 100, 56, 22,
44, -3, -9, 12, 17, 22, 6, 11 };

    printf ("The array before the sort:\n");
    for ( i = 0; i < 16; ++i )
        printf ("%i ", array[i]);
    sort (array, 16);
    printf ("\n\nThe array after the sort:\n");
    for ( i = 0; i < 16; ++i )
        printf ("%i ", array[i]);
    printf ("\n");
    return 0;
}
```

# Multidimensional arrays and functions

181

- When declaring a single-dimensional array as a formal parameter inside a function, the actual dimension of the array is not needed; simply use a pair of empty brackets to inform the C compiler that the parameter is, in fact, an array.
- This does not totally apply in the case of multidimensional arrays. For a two-dimensional array, the number of rows in the array can be omitted, but the declaration *must* contain the number of columns in the array.

- **Valid examples:**

```
function(int array_values[100][50]);  
function(int array_values[][50]);
```

- **Invalid examples:**

```
function(int array_values[100][]);  
function(int array_values[][]);
```

# Example: multidimensional array as function parameter

182

*The number of  
columns must be specified !  
No generic matrix display function possible !*

```
void displayMatrix (int matrix[3][5])  
{  
    int row, column;  
    for ( row = 0; row < 3; ++row) {  
        for ( column = 0; column < 5; ++column )  
            printf ("%5i", matrix[row][column]);  
        printf ("\n");  
    }  
}
```

# Example: multidimensional variable length array as function parameter

183

*A generic matrix display function is possible with the variable length array feature.  
The rows and columns must be listed as arguments before the matrix itself.*

```
void displayMatrix (int nRows, int nCols,  
                   int matrix[nRows][nCols])  
{  
    int row, column;  
    for ( row = 0; row < nRows; ++row) {  
        for ( column = 0; column < nCols; ++column )  
            printf ("%5i", matrix[row][column]);  
        printf ("\n");  
    }  
}
```

# Global variables

184

- A **global variable** declaration is made *outside* of any function.
- It does not belong to any particular function. Any function in the program can then access the value of that variable and can change its value.
- The primary use of global variables is in programs in which many functions must access the value of the same variable. Rather than having to pass the value of the variable to each individual function as an argument, the function can explicitly reference the variable instead.
- There is a **drawback** with this approach: Because the function explicitly references a particular global variable, the generality of the function is somewhat reduced !
- Global variables do have **default initial values: zero**



# Example: global variables

185

```
#include <stdio.h>

int x;

void f1 (void) {
    x++;
}

void f2 (void) {
    x++;
}

int main(void) {
    x=7;
    f1();
    f2();
    printf("x=%i \n", x);
}
```

# Automatic and static variables

186

- Automatic local variables (the default case of local vars) :
  - ▣ an automatic variable disappears after the function where it is defined completes execution, the value of that variable disappears along with it.
  - ▣ the value an automatic variable has when a function finishes execution is *guaranteed* not to exist the next time the function is called.
  - ▣ The value of the expression is calculated and assigned to the automatic local variable *each* time the function is called.
- Static local variables:
  - ▣ If you place the word `static` in front of a variable declaration
  - ▣ “something that has no movement”
  - ▣ a static local variable—it does *not* come and go as the function is called and returns. This implies that the value a static variable has upon leaving a function is the same value that variable will have the next time the function is called.
  - ▣ Static variables also differ with respect to their initialization. A static, local variable is initialized only *once* at the start of overall program execution—and not each time that the function is called. Furthermore, the initial value specified for a static variable *must* be a simple constant or constant expression. Static variables also have default initial values of zero, unlike automatic variables, which have no default initial value.

# Example: Automatic and static variables

187

```
// Program to illustrate static and automatic variables
#include <stdio.h>
void auto_static (void)
{
    int autoVar = 1;
    static int staticVar = 1;
    printf ("automatic = %i, static = %i\n", autoVar, staticVar);
    ++autoVar;
    ++staticVar;
}
int main (void)
{
    int i;
    for ( i = 0; i < 5; ++i )
        auto_static ();
    return 0;
}
```

# Recursive functions

188

- C permits **a function to call itself**. This process is named **recursion**.
- Useful when the solution to a problem can be expressed in terms of successively applying the same solution to subsets of the problem
- Example: factorial: recursive definition:

- $$n! = n * (n-1)!$$

  
factorial(n)

  
factorial(n-1)

# Example: recursive function

189

```
// Recursive function to calculate the factorial of n
unsigned long int factorial (unsigned int n)
{
    unsigned long int result;
    if ( n == 0 )
        result = 1;
    else
        result = n * factorial (n - 1);
    return result;
}
```

factorial(3)=3 \* factorial(2); =6

factorial(2) = 2 \* factorial(1); =2

factorial(1)= 1 \* factorial(0); =1

factorial(0)= 1

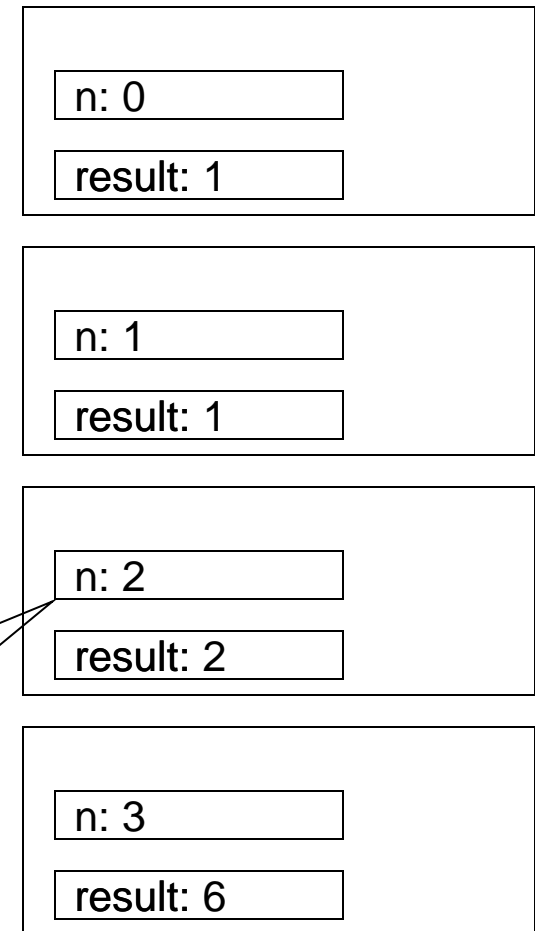
# Recursive function calls

190

- ❑ Each time any function is called in C—be it recursive or not—the function gets **its own set of local variables and formal parameters** with which to work !
- ❑ These local automatic variables are stored in a memory area called **stack**
- ❑ Each new function call pushes a new **activation record** on the stack
- ❑ This activation record contains its set of automatic local variables
- ❑ When a function call returns, its activation record is removed from the top of the stack

*The local variable `result` and the formal parameter `n` that exist when the factorial function is called to calculate the factorial of 3 are distinct from the variable `result` and the parameter `n` when the function is called to calculate the factorial of 2.*

Dr G R Patil, AIT Pune



# Example: recursive function calls

191

```
void up_and_down(int n)
{
    printf("Start call %d: n location %p\n", n, &n);
    if (n < 4)
        up_and_down(n+1);
    printf("End call %d: n location %p\n", n, &n);
}
```

...

```
up_and_down(0);
```

# Recursion pros and cons

192

- Tricky with recursion: programmer must make sure to get the recursion to an end at some time !
  - ▣ a function that calls itself tends to do so indefinitely unless the programming includes a conditional test to terminate recursion.
- Recursion often can be used where loops can be used. Sometimes the iterative solution is more obvious; sometimes the recursive solution is more obvious.
- Recursive solutions tend to be **more elegant** and **less efficient** than iterative solutions.



# Functions - Summary

193

- We distinguish between *Function definition*, *function declaration* and *Function call*
- *Function definition* general format:

```
returnType name ( type1 param1, type2 param2, ... )  
{  
    variableDeclarations  
    programStatement  
    programStatement  
    ...  
    return expression;  
}
```

- The function called *name* is defined, which returns a value of type *returnType* and has *formal parameters* *param1*, *param2*,... . The formal parameter *param1* is declared to be of type *type1*, *param2* is declared to be of type *type2*, etc.

# Function Definitions - Summary

194

- **Local variables** are typically declared at the beginning of the function, but that's not required. They can be declared anywhere, in which case their access is limited to statements appearing after their declaration in the function.
- If the function does not return a value, *returnType* is specified as **void**.
- If just void is specified inside the parentheses, the function takes no arguments.
- **Declarations for single-dimensional array arguments** do not have to specify the number of elements in the array. For multidimensional arrays, the size of each dimension except the first must be specified.

# Function Declaration - Summary

195

- Function declaration: a *prototype declaration* for the function, which has the following general format:  
*returnType name (type1, type2, ... );*
- This tells the compiler the function's return type, the number of arguments it takes, and the **type** of each argument (*names of formal parameters are not needed in function declaration !*)

# Function Calls - Summary

196

- *A function call is a statement:*
- `name ( arg1 , arg2, ... );`
- The function called *name* is called and the values *arg1*, *arg2*, ... are passed as arguments (**actual parameters**) to the function. If the function takes no arguments, just the open and closed parentheses are needed
- If you are calling a function that is defined after the call, or in another file, a function definition has to be present before !
- A function whose return type is declared as void causes the compiler to flag any calls to that function that try to make use of a returned value.
- In C, all arguments to a function are **passed by value**; therefore, their values cannot be changed by the function. Exception from this rule are arrays passed as parameters, they are **passed by reference**

## 2.3 POINTERS

# Outline

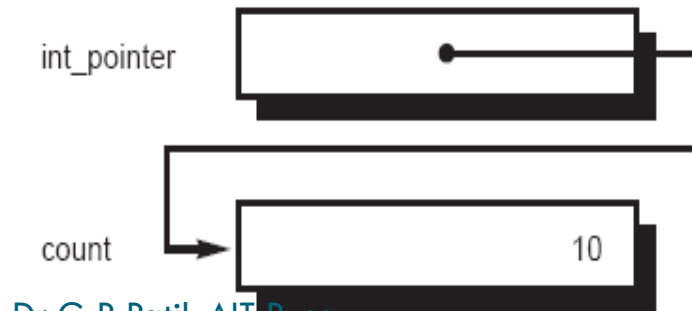
198

- Pointers
- Kernighan&Ritchie – chapter 6
- Steve Summit: Online C tutorial – chapters 10, 11
  - <http://www.eskimo.com/~scs/cclass/notes/sx10.html>
  - <http://www.eskimo.com/~scs/cclass/notes/sx11.html>
  - ▣ Pointers and Addresses
  - ▣ Pointers and Function Arguments
  - ▣ Pointers and Arrays
  - ▣ Pointer Arithmetics
  - ▣ Pointers and strings
  - ▣ Dynamic memory allocation
  - ▣ Pointer arrays. Pointers to pointers
  - ▣ Multidimensional arrays and pointers
  - ▣ Structures and pointers

# Pointers and addresses

199

- *a pointer is a variable whose value is a memory address*
- `int count = 10;`
- `int *int_pointer;`
- `int_pointer = &count;`
- The **address operator** has the effect of assigning to the variable `int_pointer`, not the value of `count`, but a *pointer* to the variable `count`.
- We say that `int_ptr` "points to" `count`
- The values and the format of the numbers representing memory addresses depend on the computer architecture and operating system. In order to have a portable way of representing memory addresses, we need a different type than integer !
- To print addresses: `%p`



# Lvalues and Rvalues

200

- There are two “values” associated with any variable:
  - ▣ An "lvalue" (left value) of a variable is the value of its address, where it is stored in memory.
  - ▣ The "rvalue" (right value) of a variable is the value stored in that variable (at that address).
- The lvalue is the value permitted on the left side of the assignment operator '=' (the address where the result of evaluation of the right side will be stored).
- The rvalue is that which is on the right side of the assignment statement
- $a = a + 1$



# Declaring pointer variables

201

```
type * variable_name;
```

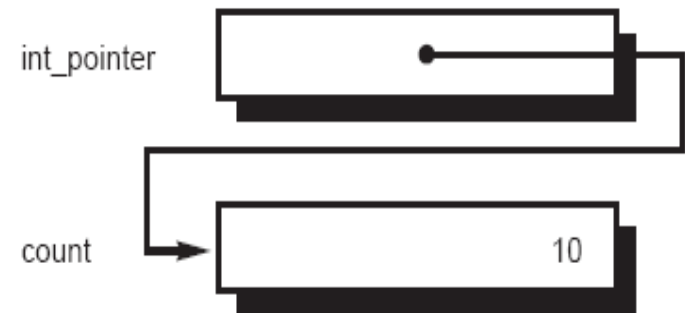
- it is not enough to say that a variable is a pointer. You also have to specify the *type of variable to which the pointer points !*
  - ▣ `int * p1; // p1 points to an integer`
  - ▣ `float * p2; // p2 points to a float`
- Exception: generic pointers (`void *`) indicate that the pointed data type is unknown
  - ▣ may be used with explicit type cast to any type (`type *`)
  - ▣ `void * p;`

# Indirection (dereferencing) operator \*

202

- To reference the contents of count through the pointer variable int\_pointer, you use the **indirection operator**, which is the asterisk \* as an unary prefix operator. `*int_pointer`
- If a pointer variable p has the type `t*`, then the expression `*p` has the type `t`

```
// Program to illustrate pointers
#include <stdio.h>
int main (void)
{
    int count = 10, x;
    int *int_pointer;
    int_pointer = &count;
    x = *int_pointer;
    printf ("count = %i, x = %i\n", count, x);
    return 0;
}
```



# Example: pointers

203

```
// Program to illustrate pointers
#include <stdio.h>
int main (void)
{
    int count = 10;
    int *ip;
    ip = &count;
    printf ("count = %i, *ip = %i\n", count, *ip);
    *ip=4;
    printf ("count = %i, *ip = %i\n", count, *ip);
    return 0;
}
```

# Using pointer variables

204

- The value of a pointer in C is meaningless until it is set pointing to something !

```
int *p;  
*p = 4;
```

**Severe runtime error !!!** the value 4 is stored in the location to which p points. But p, being uninitialized, has a random value, so we cannot know where the 4 will be stored !

- How to set pointer values:

- ▣ Using the address operator

```
int *p;  
int x;  
p = &x;  
*p = 4;
```

- ▣ Using directly assignments between pointer variables

```
int *p;  
int *p1;  
int x;  
p1 = &x;  
p = p1;  
*p = 4;
```

# NULL pointers

205

- Values of a pointer variable:
  - ▣ Usually the value of a pointer variable is a pointer to some other variable
  - ▣ Another value a pointer may have: it may be set to a *null pointer*
- A *null pointer* is a special pointer value that is known not to point anywhere.
- No other valid pointer, to any other variable, will ever compare equal to a null pointer !
- Predefined constant NULL, defined in <stdio.h>
- Good practice: test for a null pointer before inspecting the value pointed !

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

```
int *ip = NULL;
```

```
if(ip != NULL)    printf("%d\n", *ip);
```

```
if(ip )    printf("%d\n", *ip);
```

# const and pointers

206

- **With pointers, there are two things to consider:**
  - ▣ **whether the pointer will be changed**
  - ▣ **whether the value that the pointer points to will be changed.**
- Assume the following declarations:  

```
char c = 'X';  
char *charPtr = &c;
```
- If the pointer variable is always set pointing to c, it can be declared as a const pointer as follows:  

```
char * const charPtr = &c;  
*charPtr = 'Y'; // this is valid  
charPtr = &d;   // not valid !!!
```
- If the location pointed to by charPtr will not change *through the pointer variable charPtr*, that can be noted with a declaration as follows:  

```
const char *charPtr = &c;  
charPtr = &d;       // this is valid  
*charPtr = 'Y';    // not valid !!!
```

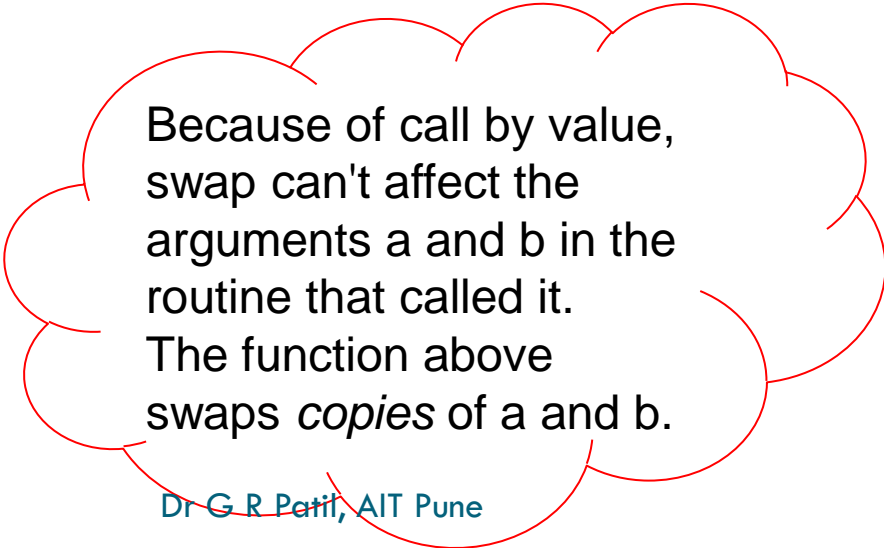
# Pointers and Function Arguments

207

- Recall that the C language passes arguments to functions by value (except arrays)
- there is no direct way for the called function to alter a variable in the calling function.

```
void swap(int x, int y) /* WRONG */  
{  
    int temp;  
    temp = x;  
    x = y;  
    y = temp;  
}
```

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



Because of call by value, swap can't affect the arguments a and b in the routine that called it. The function above swaps *copies* of a and b.

# Pointers and Function Arguments

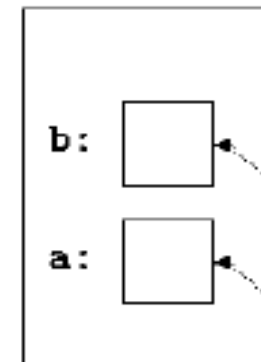
208

- *If it is necessary that a function alters its arguments, the caller can pass pointers to the values to be changed*
- Pointer arguments enable a function to access and **change** variables in the function that called it

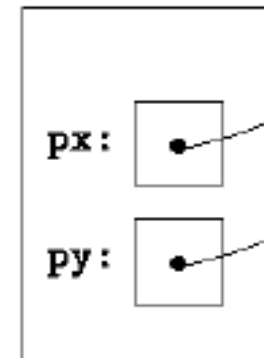
```
void swap(int *px, int *py)
/* interchange *px and *py */
{
    int temp;
    temp = *px;
    *px = *py;
    *py = temp;
}
```

```
int a=3, b=5;
swap(&a, &b);
```

in caller:



in swap:



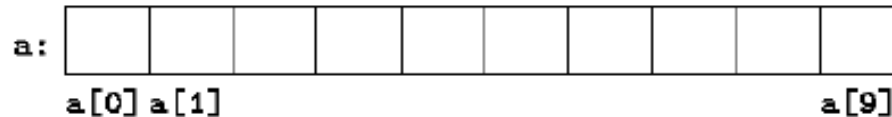


# Pointers and arrays

209

- In C, there is a strong relationship between pointers and arrays
- Any operation that can be achieved by array subscripting can also be done with pointers

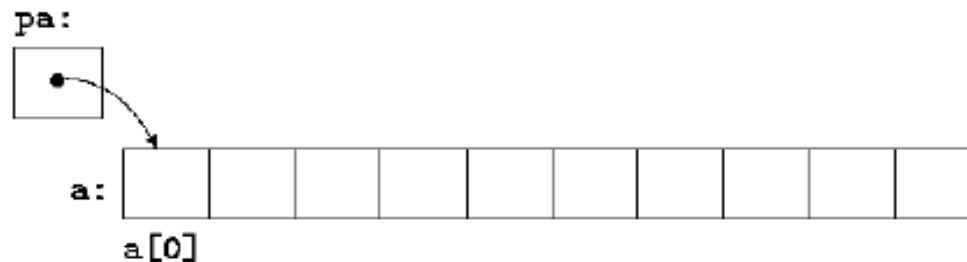
```
int a[10];
```



```
int *pa;  
pa=&a[0];
```

Or

```
pa=a;
```



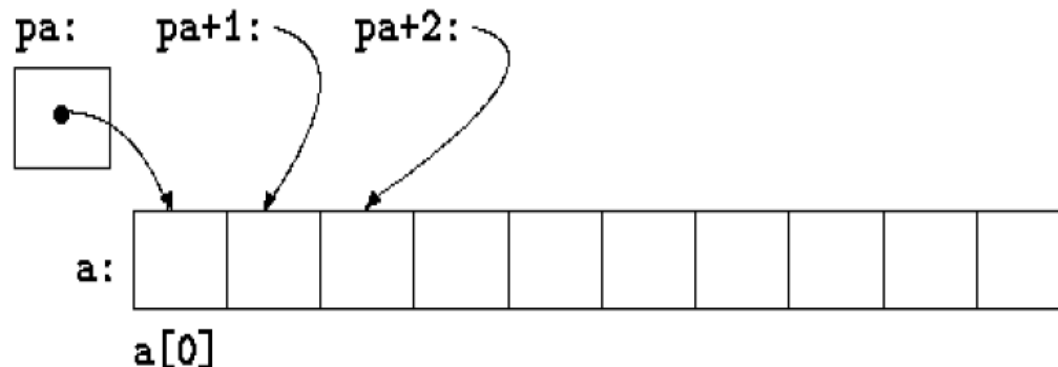
The value of a variable of type array is the address of element zero of the array.

The name of an array is a synonym for the location of the initial element.

# Pointers and Arrays

210

- If  $pa$  points to a particular element of an array, then by definition  $pa+1$  points to the next element,  $pa+i$  points  $i$  elements after  $pa$ , and  $pa-i$  points  $i$  elements before.
- If  $pa$  points to  $a[0]$ ,  $*(pa+1)$  refers to the contents of  $a[1]$ ,  $pa+i$  is the address of  $a[i]$ , and  $*(pa+i)$  is the contents of  $a[i]$ .
- The value in  $a[i]$  can also be written as  $*(a+i)$ . The address  $\&a[i]$  and  $a+i$  are also identical
- These remarks are true regardless of the type or size of the variables in the array  $a$  !



# Arrays are **constant** pointers

211

```
int a[10];  
int *pa;
```

```
pa=a;
```

```
pa++;
```

```
int a[10];  
int *pa;
```

```
a=pa;
```

```
a++;
```

OK. Pointers are variables that  
can be assigned or incremented

**Errors !!!**

The name of an array is a **CONSTANT** having as a value the location of the first element.

You cannot change the address where the array is stored !

An array's name is equivalent with a ***constant*** pointer

# Arrays as parameters

212

- When an array name is passed to a function, what is passed is the location of the first element. Within the called function, this argument is a local variable, and so an array name parameter is a pointer, that is, a variable containing an address.
- As formal parameters in a function definition,  $T\ s[]$  and  $T\ *s$  are equivalent, for any type  $T$ ; The latter is preferred because it says more explicitly that the variable is a pointer.
- Examples:
  - $f(\text{int } arr[]) \{ \dots \}$  is equivalent with  $f(\text{int } *arr) \{ \dots \}$
  - $f(\text{char } s[]) \{ \dots \}$  is equivalent with  $f(\text{char } *s) \{ \dots \}$

# Example: Arrays as parameters

213

```
void print1(int tab[], int N) {
    int i;
    for (i=0; i<N; i++)
        printf("%d ", tab[i]);
}

void print2(int tab[], int N) {
    int * ptr;
    for (ptr=tab; ptr<tab+N; ptr++)
        printf("%d ", *ptr);
}

void print3(int *tab, int N) {
    int * ptr;
    for (ptr=tab; ptr<tab+N; ptr++)
        printf("%d ", *ptr);
}

void print4(int *tab, int N) {
    int i;
    for (i=0; i<N; i++, tab++)
        printf("%d ", *tab);
}
```

*The formal parameter can be declared as array or pointer !*  
*In the body of the function, the array elements can be accessed through indexes or pointers !*

```
void main(void) {
    int a[5]={1,2,3,4,5};
    print1(a,5);
    print2(a,5);
    print3(a,5);
    print4(a,5);
}
```

# Example: Arrays as parameters

214

```
/* strlen: return length of string s */
int strlen(char *s)
{
    int n;
    for (n = 0; *s != '\0'; s++)
        n++;
    return n;
}
```

*The actual parameter can be declared as array or pointer !*

```
char array[100]="Hello, world";
char *ptr="Hello, world";

strlen("Hello, world"); /* string constant */
strlen(array); /* char array[100]; */
strlen(ptr); /* char *ptr; */
```

# Example: Arrays as parameters

215

```
int strlen(char *s)
{
    if (*s=='\0')
        return 0;
    else
        return 1 + strlen(++s);
}
```

*The recursive call gets as parameter the subarray starting with the second element*

It is possible to pass part of an array to a function, by passing a pointer to the beginning of the subarray.

For example, if `a` is an array, `f(&a[2])` and `f(a+2)` both pass to the function `f` the address of the subarray that starts at `a[2]`.

# Pointer arithmetic

216

- All operations on pointers take into account the size of the pointed type (`sizeof(T)`) !
- *Valid pointer operations:*
  - ▣ Assignment between pointers of the same type
  - ▣ Addition/ subtraction between a pointer and an integer
  - ▣ Comparison between two pointers that point to elements of the same array
  - ▣ Subtraction between two pointers that point to elements of the same array
  - ▣ Assignment or comparison with zero (NULL)



# Pointer arithmetic

217

- **Increment/decrement:** if  $p$  is a pointer to type  $T$ ,  $p++$  increases the value of  $p$  by  $\text{sizeof}(T)$  ( $\text{sizeof}(T)$  is the amount of storage needed for an object of type  $T$ ). Similarly,  $p--$  decreases  $p$  by  $\text{sizeof}(T)$ ;

```
T tab[N];
T * p;
int i;
p=&tab[i];
p++;    // p contains the address of tab[i+1];
```

- **Addition/subtraction with an integer:** if  $p$  is a pointer to type  $T$  and  $n$  an integer,  $p+n$  increases the value of  $p$  by  $n*\text{sizeof}(T)$ . Similarly,  $p-n$  decreases  $p$  by  $n*\text{sizeof}(T)$ ;

```
T tab[N];
T * p;
p=tab;
p=p+n;    // p contains the address of tab[n].
```

# Pointer arithmetic

218

- **Comparison of two pointers.**
- *If  $p$  and  $q$  point to members of the same array*, then relations like  $==$ ,  $!=$ ,  $<$ ,  $>=$ , etc., work properly.
  - For example,  $p < q$  is true if  $p$  points to an earlier element of the array than  $q$  does.
- Any pointer can be meaningfully compared for equality or inequality with zero.
- **Pointer subtraction :**
- *if  $p$  and  $q$  point to elements of the same array*, and  $p < q$ , then  $q - p + 1$  is the number of elements from  $p$  to  $q$  inclusive.
  
- The behavior is undefined for arithmetic or comparisons with pointers that do not point to members of the same array.

# Example: pointer subtraction

219

```
/* strlen: return length of string s */
int strlen(char *s)
{
    char *p = s;
    while (*p != '\0')
        p++;
    return p - s;
}
```

# Character pointers

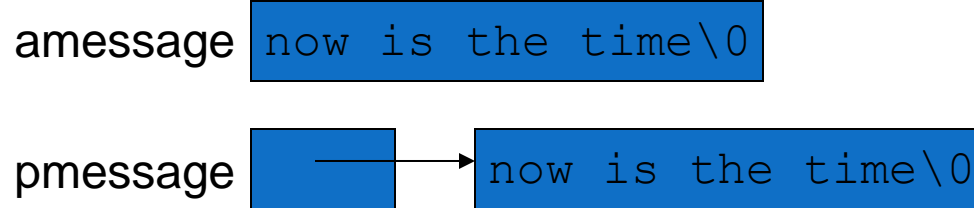
220

- A *string constant*: "I am a string"
- is an array of characters. In the internal representation, the array is terminated with the null character '\0' so that programs can find the end. The length in storage is thus one more than the number of characters between the double quotes.
- a string constant is accessed by a pointer to its first element
- `char *pmessage;`
- `pmessage = "now is the time";`
- assigns to pmessage a pointer to the character array. This is *not* a string copy; only pointers are involved !

# Character pointers

221

```
char amessage[] = "now is the time"; /* an array */  
char *pmessage = "now is the time"; /* a pointer */
```



- ❑ `amessage` is an array, just big enough to hold the sequence of characters and `'\0'` that initializes it. Individual characters within the array may be changed but `amessage` will always refer to the same storage.
- ❑ `pmessage` is a pointer, initialized to point to a string constant; the pointer may subsequently be modified to point elsewhere, but the result is undefined if you try to modify the string contents.

# Precedence of operators

222

- **\*p++** increments p after fetching the character that p points to

```
char *p="hello" ;  
printf("%c", *p++); // displays h
```

- **\*++p** increments p before fetching the character that p points to

```
char *p="hello" ;  
printf("%c", *++p); // displays e
```

# Example: Character pointers and functions

223

```
/* strcpy: copy t to s; array subscript version */
void strcpy(char s[], char t[])
{
    int i;
    i = 0;
    while ((s[i] = t[i]) != '\0')
        i++;
}
```

```
/* strcpy: copy t to s; pointer version */
void strcpy(char *s, char *t)
{
    int i;
    i = 0;
    while ((*s = *t) != '\0') {
        s++;
        t++;
    }
}
```

# Example: Character pointers and functions

224

```
/* strcpy: copy t to s; pointer version 2 */  
void strcpy(char *s, char *t)  
{  
    while ((*s++ = *t++) != '\0')  
        ;  
}
```

```
/* strcpy: copy t to s; pointer version 3 */  
void strcpy(char *s, char *t)  
{  
    while (*s++ = *t++)  
        ;  
}
```



# Exercise: What is the output ?

225

```
void swap_vectors(int *a, int *b, int n) {  
    int *t;  
    t=a;  
    a=b;  
    b=t;  
    print_vector(a,3);  
    print_vector(b,3);  
}
```

...

```
int a[3]={1,2,3};  
int b[3]={4,5,6};  
swap_vectors(a,b,3);  
print_vector(a,3);  
print_vector(b,3);
```

# Exercise: different versions of adding 2 vectors

226

```
void add_vectors1 (int a[], int b[], int r[], int n) {  
    int i;  
    for (i=0; i<n; i++)  
        r[i]=a[i]+b[i];  
}
```

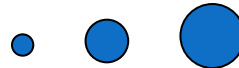
```
void add_vectors2 (int *a, int *b, int *r, int n) {  
    int i;  
    for (i=0; i<n; a++, b++, r++, i++)  
        *r=*a+*b;  
}
```

# Exercise: different versions of adding 2 vectors

227

```
int a[3]={1,2,3};  
int b[3]={4,5,6};  
int r[3];
```

```
add_vectors1(a,b,r,3);  
print_vector(r,3);  
add_vectors2(a,b,r,3);  
print_vector(r,3);
```



*OK, functions  
are equivalent*

# Exercise: different versions of adding 2 vectors

228

```
int * add_vector3(int *a, int *b, int n) {  
    int r[3];  
    int i;  
    for (i=0; i<n; i++, a++, b++)  
        r[i]=*a+*b;  
    return r;  
}
```

...

```
int a[3]={1,2,3};  
int b[3]={4,5,6};  
int * rr;  
rr=add_vector3(a,b,3);  
print_vector(rr,3);
```

**NO !!!**

***BIG MISTAKE !***  
*The function  
returns the  
address of a  
local variable !*

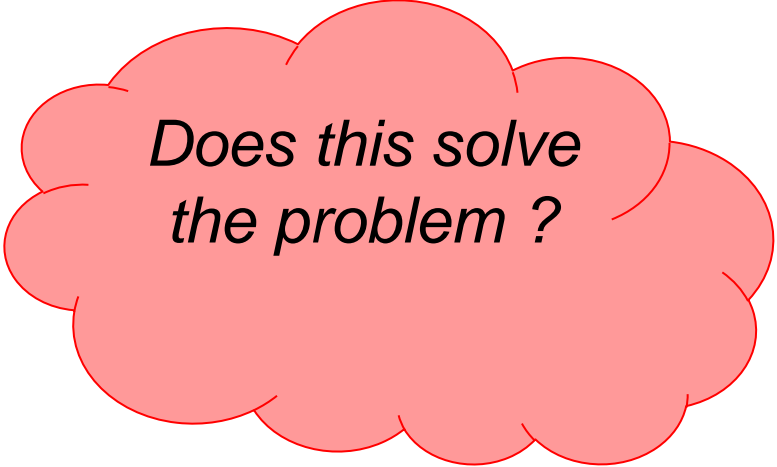
# Exercise: different versions of adding 2 vectors

229

```
int * add_vector3(int *a, int *b, int n) {  
    static int r[3];  
    int i;  
    for (i=0; i<n; i++, a++, b++)  
        r[i]=*a+*b;  
    return r;  
}
```

...

```
int a[3]={1,2,3};  
int b[3]={4,5,6};  
int * rr;  
rr=add_vector3(a,b,3);  
print_vector(rr,3);
```



*Does this solve  
the problem ?*

# Exercise: different versions of adding 2 vectors

230

```
int * add_vector3(int *a, int *b, int n) {  
    static int r[3];  
    int i;  
    for (i=0; i<n; i++, a++, b++)  
        r[i]=*a+*b;  
    return r;  
}
```

...

```
int a[3]={1,2,3};  
int b[3]={4,5,6};  
int c[3]={0,1,2};  
int d[3]={3,4,5};  
int * rr1, *rr2;  
rr1=add_vector3(a,b,3);  
rr2=add_vector3(c,d,3);  
print_vector(rr1,3);  
print_vector(rr2,3);
```

*Does this solve  
the problem ?*

**NO !**

# Dynamic memory allocation

231

- Variable definitions in C: the C compiler automatically allocates the correct amount of storage (1-n memory locations) where the variable will be stored  
-> this happens before the execution of the actual program starts
- Sometimes it can be useful to *dynamically* allocate storage while a program is running:
- Suppose we have a program that is designed to read in a set of data from input into an array in memory but we don't know how many data items there are until the program starts execution. We have three choices:
  - ▣ Define the array to contain the maximum number of possible elements at compile time.
  - ▣ Use a variable-length array to dimension the size of the array at runtime.
  - ▣ Allocate the array dynamically using one of C's memory allocation routines.

# malloc()

232

- `<stdlib.h>`
- `void * malloc(int n);`
- `malloc` allocates *n* bytes of memory and returns a pointer to them if the allocation was succesful, NULL otherwise
- Before using the pointer returned by `malloc`, it has to be checked if it is not NULL !!
- The pointer returned by `malloc` is of the generic type `void *`; it has to be converted to a concrete pointer type



# malloc() Example 1

233

```
#include <stdlib.h>

...

char *line;
int linelen;

printf ("How long is your line ?");
scanf ("%d",&linelen);

line = (char *) malloc(linelen);

/* incomplete here - malloc's return value not checked */

getline(line, linelen);
```

# malloc() Example 2

234

```
char *somestring="how are you";  
char *copy;  
...  
copy = (char *) malloc(strlen(somestring) + 1);  
  
/* incomplete -- malloc's return value not checked */  
  
strcpy(copy, somestring);
```

# Checking what malloc returns !

235

- When malloc is unable to allocate the requested memory, it returns a *null pointer*. Therefore, whenever you call malloc, it's vital to check the returned pointer before using it !

```
char * line = (char *) malloc(linelen); ;  
if(line == NULL) {  
    printf("out of memory\n");  
    return; // exits current function  
}
```

```
char * line = (char *) malloc(linelen);  
if(line == NULL) {  
    printf("out of memory\n");  
    exit(1); // exits all nested function calls,  
             // terminates program  
}
```

# Dynamic allocation of arrays

236

- Dynamic allocation of an array of N elements of type TIP:
- `TIP * p;`
- `p= (TIP *) malloc(N*sizeof(TIP)) ;`
- Pointer p will point to a memory block big enough to hold N elements of type TIP.
- Variable p can be used in the same way as if it was declared:
- `TIP p[N] ;`

# Example: dynamic allocation of arrays

237

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    int n;
    int * tab;
    int i;
    printf("Input number of elements: \n");
    scanf("%d", &n);
    if ((tab=(int *)malloc(n * sizeof(int)))==NULL) {
        printf("Memory allocation error !\n");
        exit(1);
    }
    for (i=0; i<n; i++)
        scanf("%d", &tab[i]);
    for (i=0; i<n; i++)
        printf("%d ", tab[i]);
    free(tab);
    return 1;
}
```

# Example: dynamic allocation of character strings

238

```
void main(void)
{
    char sir1[40];
    char *sir2;
    printf("Enter a string: \n");
    scanf("%40s", sir1);
    if ((sir2=(char *)malloc(strlen(sir1)+1))==NULL) {
        printf("memory allocation error !\n");
        exit(1);
    }
    strcpy(sir2, sir1);
    printf("The copy is: %s \n", sir2);
    .....
}
```

# Lifetime of dynamic allocated memory

239

- Memory allocated with malloc lasts as long as you want it to. It does not automatically disappear when a function returns, as automatic variables do:

```
void fct(void) {  
    int *p;  
    p=(int*) malloc(10*sizeof(int));  
    return;  
}
```



The memory area  
allocated here remains  
occupied also after the  
function call is ended !  
Only the pointer  
variable p accessing it  
disappears.

# Lifetime example

240

- Have a function create and return an array of values:

```
int * fct(void) {  
    int *p;  
    p=(int*) malloc(10*sizeof(int));  
    // fill p with values . . .  
    return p;  
}
```

**YES !**

```
int * fct(void) {  
    int p[10];  
    // fill p with values . . .  
    return p;  
}
```

**NO !**



# Exercise: different versions of adding 2 vectors

241

```
int * add_vector4(int *a, int *b, int n) {  
    int * r;  
    r=(int *) malloc(sizeof (int) * n);  
    if (r==NULL) exit(1);  
    int i;  
    for (i=0; i<n; i++, a++, b++)  
        r[i]=*a+*b;  
    return r;  
}
```

...

```
int a[3]={1,2,3};  
int b[3]={4,5,6};  
int * rr;  
rr=add_vector4(a,b,3);  
print_vector(rr,3);
```

# free()

242

- Dynamically allocated memory is deallocated with the free function. If `p` contains a pointer previously returned by `malloc`, you can call
- `free(p) ;`
- which will ``give the memory back" to the heap of memory from which `malloc` requests are satisfied.
- the memory you give back by calling `free()` is immediately usable by other parts of your program. (Theoretically, it may even be usable by other programs.)
- When your program exits, any memory which it has allocated but not freed should be automatically released by the operating system.
- Once you've freed some memory you must remember not to use it any more. After calling `free(p)` it is probably the case that `p` still points at the same memory. However, since we've given it back, it's now ``available," and a later call to `malloc` might give that memory to some other part of your program.

# Pointer arrays. Pointers to Pointers

243

- Since pointers are variables themselves, they can be stored in arrays just as other variables can

# Pointers vs Multidimensional arrays

244

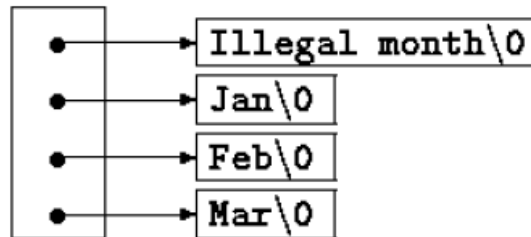
- `int a[10][20];`
- `int *b[10];`
- `a[3][4]` and `b[3][4]` are both syntactically legal references to an int.
- `a` is a true two-dimensional array: 200 int-sized locations have been set aside, and the conventional rectangular subscript calculation  $20 * \text{row} + \text{col}$  is used to find the element `a[row,col]`.
- For `b`, however, the definition only allocates 10 pointers and does not initialize them; initialization must be done explicitly, either statically or with code. Assuming that each element of `b` does point to a twenty-element array, then there will be 200 ints set aside, plus ten cells for the pointers. The important advantage of the pointer array is that the rows of the array may be of different lengths. That is, each element of `b` need not point to a 20 element vector; some may point to two elements, some to fifty, and some to none at all.

# Pointers vs Multidimens arrays

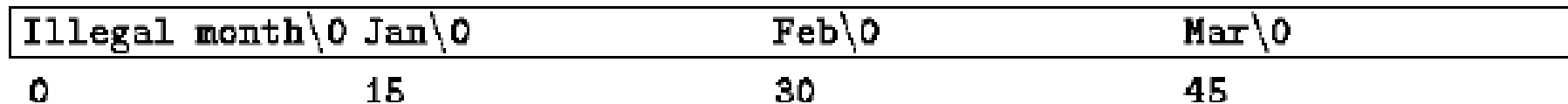
245

- ❑ `char *name[] = { "Illegal month", "Jan", "Feb", "Mar" };`
- ❑ `char aname[][15] = { "Illegal month", "Jan", "Feb", "Mar" };`

name:



aname:



# Program design example

246

- a program that will read an then sort a set of text lines into alphabetic order
- Requirements:
  - ▣ Program reads a number of lines, until EOF, but not more than MAXLINES=5000
  - ▣ Each line may have a different number of characters, but not more than MAXLEN=1000
  - ▣ Program sorts lines in alphabetic order
  - ▣ Program displays sorted lines
- How to approach program design problems:
  - ▣ Data structures
  - ▣ Algorithms
  - ▣ Functions

# Recall: Sorting algorithm

247

```
// sort an array of integers into ascending order
```

```
void sort (int a[], int n) {  
    int i, j, temp;  
    for ( i = 0; i < n - 1; ++i )  
        for ( j = i + 1; j < n; ++j )  
            if ( a[i] > a[j] ) {  
                temp = a[i];  
                a[i] = a[j];  
                a[j] = temp;  
            }  
}
```

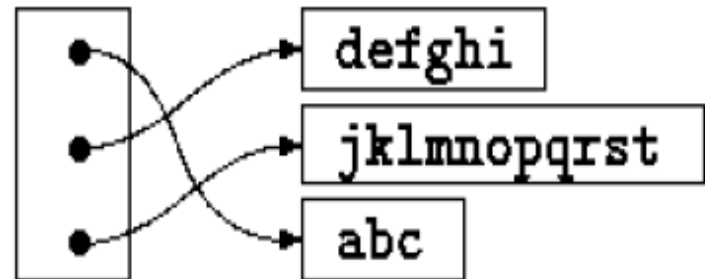
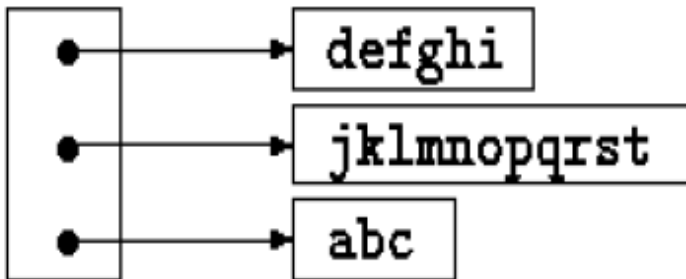
The sorting algorithms will work; except that now we have to deal with lines of text, which are of different lengths, and which, unlike integers, can't be compared or moved in a single operation (strcmp, strcpy).

We need a data representation that will cope efficiently and conveniently with variable-length text lines.

# Data structure: array of pointers

248

- Each line can be accessed by a pointer to its first character. The pointers themselves can be stored in an array.
- Two lines can be compared by passing their pointers to `strcmp`.
- When two out-of-order lines have to be exchanged, the pointers in the pointer array are exchanged, not the text lines themselves.

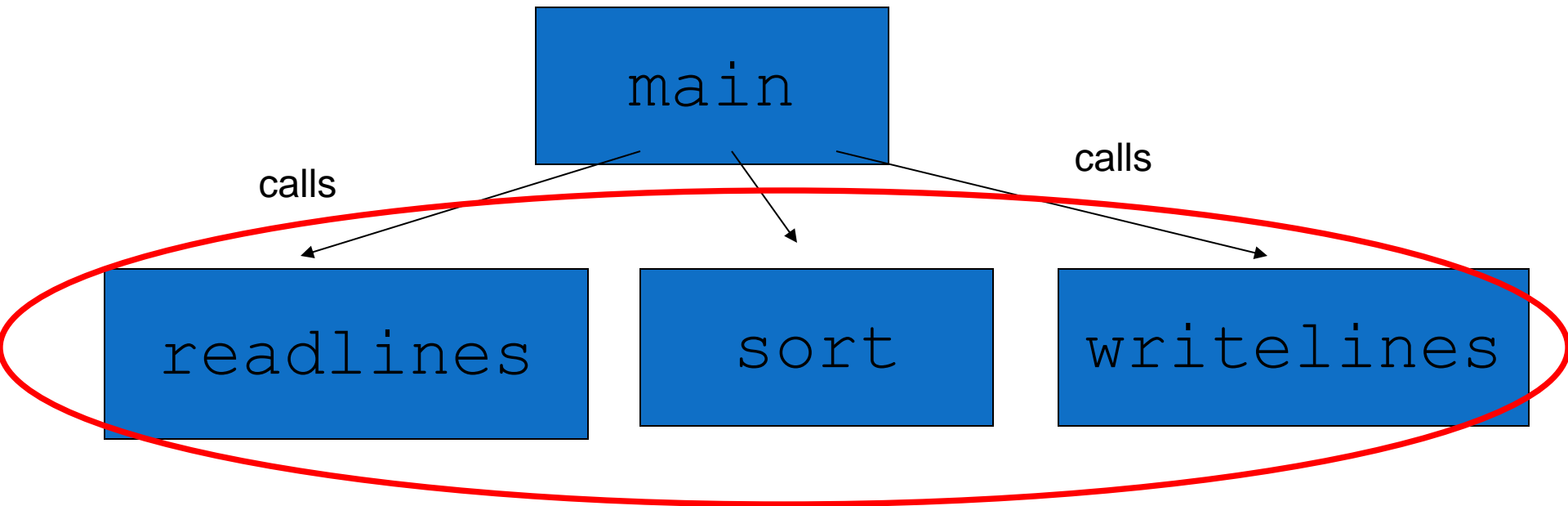


```
#define MAXLINES 5000    /* max number of lines to be sorted */  
char *lineptr[MAXLINES]; /* array of pointers to text lines */
```



# Top-down program refinement

249



1. Split problem (program) in subproblems (functions)
2. For each function:
  - ▣ Functionality (what it does)
  - ▣ Prototype
  - ▣ Contract: how exactly does it handle input arguments, which are its returned results and side-effects (on global variables)

# Designing the functions

250

- Readlines:
  - ▣ `int readlines(char *lineptr[], int nlines);`
  - ▣ Argument `lineptr`: array will be filled with pointers to the read lines
  - ▣ Argument `nlines`: maximum number of lines that can be read
  - ▣ Return value: number of lines that has been read or -1 if an error occurred
- Writelines:
  - ▣ `void writelines(char *lineptr[], int nlines);`
  - ▣ Writes `nlines` elements of array `lineptr`
- Sort
  - ▣ `void sort(char *lineptr[], int nlines);`

# Example: main program

251

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

#define MAXLINES 5000      /* max #lines to be sorted */
char *lineptr[MAXLINES]; /* pointers to text lines */

int readlines(char *lineptr[], int nlines);
void writelines(char *lineptr[], int nlines);
void sort(char *lineptr[], int nlines);

int main(void) {
    int nlines;      /* number of input lines read */
    if ((nlines = readlines(lineptr, MAXLINES)) >= 0) {
        sort(lineptr, nlines);
        writelines(lineptr, nlines);
        return 0;
    } else {
        printf("error: input too big to sort\n");
        return 1;
    }
}
```

# Implementing the functions

252

- If a function corresponds to a problem which is still complex, split it in other subproblems (functions)
- Top-down decomposition
- Stepwise refinement

# Example: decomposing `readlines()`

253

- Readlines:
  - ▣ repeatedly: reads a line from `stdin`, allocates a buffer of the exact line length and copies there the input, stores pointer to buffer in current element of `lineptr`
- `int getline(char *s, int lim);`
  - ▣ Reads a maximum of `lim-1` characters from `stdin`, until `\n` or EOF
  - ▣ Argument `s` must point to an array that is able to hold a maximum of `lim` characters
  - ▣ Returns the number of characters read
  - ▣ The characters read will be stored in array `s`; character `\n` is also included. Null character is appended.

# Example: readlines () implementation

254

```
#define MAXLEN 1000 /* max length of any input line */
int getline(char *, int);

/* readlines: read input lines */
int readlines(char *lineptr[], int maxlines) {
    int len, nlines;
    char *p, line[MAXLEN];
    nlines = 0;
    while ((len = getline(line, MAXLEN)) > 0)
        if (nlines >= maxlines ||
            ((p = (char *)malloc(len)) == NULL))
            return -1;
        else {
            line[len-1] = '\0'; /* delete newline */
            strcpy(p, line);
            lineptr[nlines++] = p;
        }
    return nlines;
}
```

# Example: getline

255

```
/* getline:  read a line into s, return length */
int getline(char s[],int lim)
{
    int c, i;
    for (i=0; i < lim-1 &&
           (c=getchar()) !=EOF && c!='\n'; ++i)
        s[i] = c;
    if (c == '\n') {
        s[i] = c;
        ++i;
    }
    s[i] = '\0';
    return i;
}
```

# Example: writelines

256

```
/* writelines:  write output lines */
void writelines(char *lineptr[], int nlines)
{
    int i;
    for (i = 0; i < nlines; i++)
        printf("%s\n", lineptr[i]);
}
```



# Example: sort

257

```
void sort (char* v[], int n)
{
    int i, j;
    char *temp;
    for ( i = 0; i < n - 1; ++i )
        for ( j = i + 1; j < n; ++j )
            if ( strcmp(v[i], v[j])>0 ) {
                /* swaps only pointers to strings */
                temp = v[i];
                v[i] = v[j];
                v[j] = temp;
            }
}
```

## 2.4 STRING MANIPULATIONS

# Outline

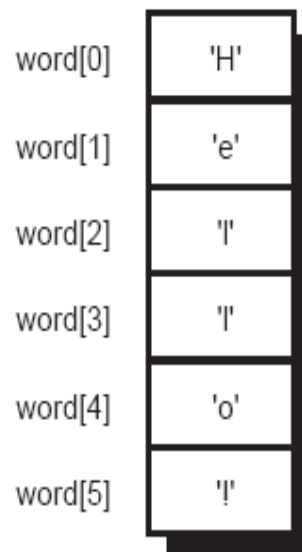
259

- Strings
  - ▣ Character Arrays/ Character Strings
  - ▣ Initializing Character Strings. The null string.
  - ▣ Escape Characters
  - ▣ Displaying Character Strings
  - ▣ Inputting Character Strings
  - ▣ String processing:
    - Testing Strings for Equality
    - Comparing Strings
    - Copying Strings
  - ▣ Functions in <string.h>
  - ▣ String to number conversion functions
  - ▣ Character Strings, Structures, and Arrays
  - ▣ Example: Simple dictionary program
    - Sorting the dictionary
    - A better search in sorted arrays

# Arrays of characters

260

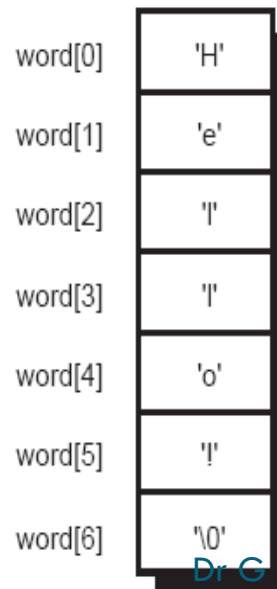
- `char word [] = { 'H', 'e', 'l', 'l', 'o', '!' };`
- To print out the contents of the array `word`, you run through each element in the array and display it using the `%c` format characters.
- To do processings of the word (copy, concatenate two words, etc) you need to have the actual length of the character array in a separate variable !



# Character strings

261

- A method for dealing with character arrays without having to worry about precisely how many characters you have stored in them:
- **Placing a special character at the end of every character string.** In this manner, the function can then determine for itself when it has reached the end of a character string after it encounters this special character.
- In the C language, the special character that is used to signal the end of a string is known as the *null* character and is written as `'\0'`.
- `char word [] = { 'H', 'e', 'l', 'l', 'o', '!', '\0' };`



# Example: string length

262

```
// Function to count the number of characters in a string
#include <stdio.h>

int stringLength (char string[]){
    int count = 0;
    while ( string[count] != '\0' )
        ++count;
    return count;
}

int main (void) {
    char word1[] = { 'a', 's', 't', 'e', 'r', '\0' };
    char word2[] = { 'a', 't', '\0' };
    char word3[] = { 'a', 'w', 'e', '\0' };
    printf ("%i %i %i\n", stringLength (word1),
            stringLength (word2), stringLength (word3));
    return 0;
}
```

# Example: const strings

263

```
// Function to count the number of characters in a string
#include <stdio.h>
```

```
int stringLength (const char string[]){
    int count = 0;
    while ( string[count] != '\0' )
        ++count;
    return count;
}
```

The function declares its argument as a const array of characters because it is not making any changes to the array

```
int main (void) {
    const char word1[] = { 'a', 's', 't', 'e', 'r', '\0' };
    const char word2[] = { 'a', 't', '\0' };
    const char word3[] = { 'a', 'w', 'e', '\0' };
    printf ("%i %i %i\n", stringLength (word1),
            stringLength (word2), stringLength (word3));
    return 0;
}
```

# Initializing character strings

264

- **Initializing a string:**

```
char word[] = "Hello!";
```

- **Is equivalent with:**

```
char word[] = { 'H', 'e', 'l', 'l', 'o', '!', '\0' };
```

- **The null string:** A character string that contains no characters other than the null character

```
char empty[] = "";
```

```
char buf[100] = "";
```

- **Initializing a very long string over several lines:**

```
char letters[] =  
{ "abcdefghijklmnopqrstuvwxyz\  
ABCDEFGHIJKLMNOPQRSTUVWXYZ" };
```

- **Adjacent strings are concatenated:**

```
char letters[] =  
{ "abcdefghijklmnopqrstuvwxyz"  
"ABCDEFGHIJKLMNOPQRSTUVWXYZ" };
```

```
printf ("Programming" " in C is fun\n");
```



# Strings vs Characters

265

- The string constant "x"
- The character constant 'x'
- Differences:
  1. 'x' is a basic type (char) but "x" is a derived type, an array of char
  2. "x" really consists of two characters, 'x' and '\0', the null character

# Escape characters

266

`\a` Audible alert  
`\b` Backspace  
`\f` Form feed  
`\n` Newline  
`\r` Carriage return  
`\t` Horizontal tab  
`\v` Vertical tab  
`\\` Backslash  
`\"` Double quotation mark  
`\'` Single quotation mark  
`\?` Question mark  
`\nnn` Octal character value *nnn*  
`\unnnn` Universal character name  
`\Unnnnnnnnn` Universal character name  
`\xnn` Hexadecimal character value *nn*

- the backslash character has a special significance
- other characters can be combined with the backslash character to perform special functions. These are referred to as *escape characters*.

# Examples: Escape characters

267

```
printf ("\aSYSTEM SHUT DOWN IN 5 MINUTES!!\n");  
  
printf ("%i\t%i\t%i\n", a, b, c);  
  
printf ("\\t is the horizontal tab character.\n");  
  
printf ("\"Hello,\" he said.\n");  
  
c = '\\';  
  
printf("\033\"Hello\"\\n");
```

# Displaying character strings

268

- Displaying a string: %s
- `printf ("%s\n", word);`

# Inputting character strings

269

- ❑ `char string[81];`
- ❑ `scanf ("%s", string); // NOT recommended !`
  - The `scanf` function can be used with the `%s` format characters to read in a string of characters up to a blank space, tab character, or the end of the line, whichever occurs first
  - The string terminator (null character) is appended to string
  - Note that unlike previous `scanf` calls, in the case of reading strings, the `&` is *not* placed before the array name !
  - If you type in more than 80 consecutive characters to the preceding program without pressing the spacebar, the tab key, or the Enter (or Return) key, `scanf` overflows the character array !
- ❑ Correct use of `scanf` for reading strings:
- ❑ `scanf ("%80s", string); // Safer version !`
  - If you place a number after the `%` in the `scanf` format string, this tells `scanf` the maximum number of characters to read.



# Example: string processing

270

```
#include <stdio.h>

void concat (char result[],
             const char str1[], const char str2[]);

int main (void)
{
    const char s1[] = "Test " ;
    const char s2[] = "works." ;
    char s3[20];
    concat (s3, s1, s2);
    printf ("%s\n", s3);
    return 0;
}
```

# Example: concatenate strings

271

```
// Function to concatenate two character strings
void concat (char result[], const char str1[],
              const char str2[])
{
    int i, j;
    // copy str1 to result
    for ( i = 0; str1[i] != '\0'; ++i )
        result[i] = str1[i];
    // copy str2 to result
    for ( j = 0; str2[j] != '\0'; ++j )
        result[i + j] = str2[j];
    // Terminate the concatenated string with a null character
    result [i + j] = '\0';
}
```

# Testing strings for equality

272

```
bool equalStrings (const char s1[], const char s2[])
{
    int i = 0;
    bool areEqual;
    while ( s1[i] == s2 [i] &&  s1[i] != '\0' && s2[i] != '\0' )
        ++i;
    if ( s1[i] == '\0' && s2[i] == '\0' )
        areEqual = true;
    else
        areEqual = false;
    return areEqual;
}
```

**!!! NOT: s1==s2**



# Alphabetically comparing strings

273

```
// Function to compare two character strings
int compareStrings (const char s1[], const char s2[])
{
    int i = 0, answer;
    while ( s1[i] == s2[i] && s1[i] != '\0' && s2[i] != '\0' )
        ++i;
    if ( s1[i] < s2[i] )
        answer = -1; /* s1 < s2 */
    else if ( s1[i] == s2[i] )
        answer = 0; /* s1 == s2 */
    else
        answer = 1; /* s1 > s2 */
    return answer;
}
```

**!!! NOT:  $s1 < s2$**

**!!! NOT:  $s1 > s2$**

Dr. R. Patil, AIT Pune

# Copying strings

274

```
void copyString(char dest[], char srs[]) {  
    int i;  
    i=0;  
    while ((dest[i] = srs[i]) != '\0')  
        i++;  
}
```

**!!! NOT: dest = srs**

# String functions

275

- ❑ The C library supplies several string-handling functions; You don't have to re-write them from scratch !
- ❑ ANSI C uses the `<string.h>` header file to provide the prototypes.
- ❑ Most frequently used functions: `strlen()`, `strcat()`, `strncat()`, `strcmp()`, `strncmp()`, `strcpy()`, and `strncpy()`.
  
- ❑ `#include <string.h>`
- ❑ `strcat (s1, s2)`
  - ▣ Concatenates the character string `s2` to the end of `s1`, placing a null character at the end of the final string. The function also returns `s1`.
- ❑ `strcmp (s1, s2)`
  - ▣ Compares strings `s1` and `s2` and returns a value less than zero if `s1` is less than `s2`, equal to zero if `s1` is equal to `s2`, and greater than zero if `s1` is greater than `s2`.
- ❑ `strcpy (s1, s2)`
  - ▣ Copies the string `s2` to `s1`, also returning `s1`.
- ❑ `strlen (s)`
  - ▣ Returns the number of characters in `s`, excluding the null character.

# String functions (cont.)

276

- `strncat (s1, s2, n)`
  - Copies `s2` to the *end* of `s1` until either the null character is reached or `n` characters have been copied, whichever occurs first. Returns `s1`.
- `strncmp (s1, s2, n)`
  - Performs the same function as `strcmp`, except that at most `n` characters from the strings are compared.
- `strncpy (s1, s2, n)`
  - Copies `s2` to `s1` until either the null character is reached or `n` characters have been copied, whichever occurs first. Returns `s1`.
- `strchr (s, c)`
  - Searches the string `s` for the last occurrence of the character `c`. If found, a pointer to the character in `s` is returned; otherwise, the null pointer is returned.
- `strstr (s1, s2)`
  - Searches the string `s1` for the first occurrence of the string `s2`. If found, a pointer to the start of where `s2` is located inside `s1` is returned; otherwise, if `s2` is not located inside `s1`, the null pointer is returned.

# Example: String functions

277

```
#include <stdio.h>

#include <string.h> /* provides strlen() prototype */

#define PRAISE "What a super marvelous name!"

int main(void) {
    char name[40];
    printf("What's your name?\n");
    scanf("%39s", name);
    printf("Hello, %s. %s\n", name, PRAISE);
    printf("Your name of %d letters occupies %d memory \n",
        strlen(name), sizeof name);
    return 0;
}
```

# Example: String functions

278

```
#include <stdio.h>

#include <string.h>

int main(void) {
    char string1[] = "this is";
    char string2[] = "a test";
    char string3[20] = "Hello, ";
    char string4[] = "world!";
    printf("%s\n", string3);
    strcat(string3, string4);
    printf("%s\n", string3);
    if(strcmp(string1, string2) == 0)
        printf("strings are equal\n");
    else printf("strings are different\n");
    return 0;
}
```

# String to number conversions

279

- Storing a number as a string means storing the digit characters
- Example: the number 213 can be stored in a character string array as the digits '2', '1', '3', '\0'. Storing 213 in numeric form means storing it as an int.
- C requires numeric forms for numeric operations, such as addition and comparison, but displaying numbers on your screen requires a string form because a screen displays characters. The printf() and scanf() functions, through their %d and other specifiers, convert numeric forms to string forms, and vice versa.
- C also has functions whose sole purpose is to convert string forms to numeric forms.

# String to number conversion functions

280

- `<stdlib.h>`
- `atoi(s)` converts string `s` to a type `int` value and returns it. The function converts characters until it encounters something that is not part of an integer.
- `atof()` converts a string to a type `double` value and returns it
- `atol()` converts a string to a type `long` value and returns it

```
// Using atoi
#include <stdio.h>
#include <stdlib.h>
int main (void) {
    printf ("%i\n", atoi("245"));
    printf ("%i\n", atoi("100") + 25);
    printf ("%i\n", atoi("13x5"));
    return 0;
}
```



# String to number conversion

## Do-it-yourself exercise

281

```
// Function to convert a string to an integer - my atoi
#include <stdio.h>
int strToInt (const char string[]) {
    int i, intValue, result = 0;
    for ( i = 0; string[i] >= '0' && string[i] <= '9'; ++i )
    {
        intValue = string[i] - '0';
        result = result * 10 + intValue;
    }
    return result;
}
int main (void) {
    printf ("%i\n", strToInt("245"));
    printf ("%i\n", strToInt("100") + 25);
    printf ("%i\n", strToInt("13x5"));
    return 0;
}
```

# Example: readLine

282

```
// Function to read a line of text from the terminal
void readLine (char buffer[])
{
    char character;
    int i = 0;
    do
    {
        character = getchar ();
        buffer[i] = character;
        ++i;
    }
    while ( character != '\n' );
    buffer[i - 1] = '\0';
}
```

# Example continued

283

```
#include <stdio.h>

void readLine (char buffer[]);

int main (void)
{
    int i;
    char line[81];
    for ( i = 0; i < 3; ++i )
    {
        readLine (line);
        printf ("%s\n\n", line);
    }
    return 0;
}
```

# Example: array of structures

## Example: a dictionary program

284

```
struct entry
{
char word[15];
char definition[50];
};

struct entry dictionary[100] =
{ { "aardvark", "a burrowing African mammal" },
  { "abyss", "a bottomless pit" },
  { "acumen", "mentally sharp; keen" },
  { "addle", "to become confused" },
  { "aerie", "a high nest" },
  { "affix", "to append; attach" },
  { "agar", "a jelly made from seaweed" },
  { "ahoy", "a nautical call of greeting" },
  { "aigrette", "an ornamental cluster of feathers" },
  { "ajar", "partially opened" } };
```

# Example: dictionary continued

285

```
int lookup (const struct entry dictionary[],
            const char search[], const int entries);

int main (void)
{
    char word[10];
    int entries = 10;
    int entry;
    printf ("Enter word: ");
    scanf ("%14s", word);
    entry = lookup (dictionary, word, entries);
    if ( entry != -1 )
        printf ("%s\n", dictionary[entry].definition);
    else
        printf ("The word %s is not in my dictionary.\n", word);
    return 0;
}
```

# Searching in array

286

```
#include <string.h>

// function to look up a word inside a dictionary
int lookup (const struct entry dictionary[],
            const char search[],const int entries)
{
    int i;
    for ( i = 0; i < entries; ++i )
        if ( strcmp(search, dictionary[i].word) )
            return i;
    return -1;
}
```

# Binary search

287

- ❑ **Binary Search Algorithm**
- ❑ Search  $x$  in **SORTED** array  $M$
- ❑ **Step 1:** Set  $low$  to  $0$ ,  $high$  to  $n - 1$ .
- ❑ **Step 2:** If  $low > high$ ,  $x$  does not exist in  $M$  and the algorithm terminates.
- ❑ **Step 3:** Set  $mid$  to  $(low + high) / 2$ .
- ❑ **Step 4:** If  $M[mid] < x$ , set  $low$  to  $mid + 1$  and go to step 2.
- ❑ **Step 5:** If  $M[mid] > x$ , set  $high$  to  $mid - 1$  and go to step 2.
- ❑ **Step 6:**  $M[mid]$  equals  $x$  and the algorithm terminates.

# Binary search

288

```
// Function to look up a word inside a dictionary
int lookup (const struct entry dictionary[],
            const char search[], const int entries)
{
    int low = 0;
    int high = entries - 1;
    int mid, result;
    while ( low <= high )
    {
        mid = (low + high) / 2;
        result = strcmp (dictionary[mid].word, search);
        if ( result == -1 )
            low = mid + 1;
        else if ( result == 1 )
            high = mid - 1;
        else
            return mid; /* found it */
    }
    return -1; /* not found */
}
```



## 2.5 STRUCTURES, UNIONS, ENUMERATIONS

# Outline

290

- Structures
  - ▣ Defining and using Structures
  - ▣ Functions and Structures
  - ▣ Initializing Structures. Compound Literals
  - ▣ Arrays of Structures
  - ▣ Structures Containing Structures and/or Arrays
  - ▣ Unions
- More on Data Types
  - ▣ Enumerated Data Types
  - ▣ The typedef Statement
  - ▣ Data Type Conversions

# The concept of structures

291

- ❑ Structure: a tool for grouping heterogenous elements together.
- ❑ Array: a tool for grouping homogenous elements together
- ❑ Example: storing calendar dates (day, month, year)
- ❑ Version1: using independent variables:
- ❑ `int month = 9, day = 25, year = 2004;`
- ❑ Using this method, you must keep track of three separate variables for each date that you use in the program—variables that are logically related. It would be much better if you could somehow group these sets of three variables together. This is what the structure in C allows you to do !

# Example: structures

292

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

Defines type `struct date`, with 3 **fields** of type `int`  
The names of the fields are local in the context of the structure.

A struct declaration defines a type: if not followed by a list of variables it reserves no storage; it merely describes a template or shape of a structure.

```
struct date today, purchaseDate;
```

Defines 3 variables of type `struct date`

```
today.year = 2004;
today.month = 10;
today.day = 5;
```

Accesses fields of a variable of type `struct date`

A member of a particular structure is referred to in an expression by a construction of the form *structurename.member*

# Example: determine tomorrow's date (Version 1)

293

```
// Program to determine tomorrow's date
#include <stdio.h>
int main (void)
{
    struct date
    {
        int month;
        int day;
        int year;
    };
    struct date today, tomorrow;
    const int daysPerMonth[12] = { 31, 28, 31, 30, 31, 30,
    31, 31, 30, 31, 30, 31 };

    printf ("Enter today's date (mm dd yyyy): ");
    scanf ("%i%i%i", &today.month, &today.day, &today.year);
```

## Example continued

294

```
if ( today.day != daysPerMonth[today.month - 1] ) {
    tomorrow.day = today.day + 1;
    tomorrow.month = today.month;
    tomorrow.year = today.year;
}
else if ( today.month == 12 ) { // end of year
    tomorrow.day = 1;
    tomorrow.month = 1;
    tomorrow.year = today.year + 1;
}
else { // end of month
    tomorrow.day = 1;
    tomorrow.month = today.month + 1;
    tomorrow.year = today.year;
}
printf ("Tomorrow's date is %i/%i/%i.\n", tomorrow.month,
                                              tomorrow.day, tomorrow.year );
return 0;
}
```

# Operations on structures

295

- Legal operations on a structure are :
  - ▣ copying it or assigning to it as a unit
    - this includes passing arguments to functions and returning values from functions as well.
  - ▣ taking its address with &
  - ▣ accessing its members.
  - ▣ structures may **not** be compared as units !
  - ▣ a structure may be initialized by a list of constant member values

# Example: determine tomorrow's date (Version 2)

296

```
// Program to determine tomorrow's date
#include <stdio.h>
#include <stdbool.h>
```

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

*Defines type struct date as a global type*

```
int numberOfDays (struct date d);
```

*Declares a function that takes a struct date as a parameter*

```
int main (void)
{
    struct date today, tomorrow;
    printf ("Enter today's date (mm dd yyyy): ");
    scanf ("%i%i%i", &today.month, &today.day, &today.year);
```



# Example continued

297

```
if ( today.day != numberOfDays (today) ) {
    tomorrow.day = today.day + 1;
    tomorrow.month = today.month;
    tomorrow.year = today.year;
}
else if ( today.month == 12 ) { // end of year
    tomorrow.day = 1;
    tomorrow.month = 1;
    tomorrow.year = today.year + 1;
}
else { // end of month
    tomorrow.day = 1;
    tomorrow.month = today.month + 1;
    tomorrow.year = today.year;
}
printf ("Tomorrow's date is %i/%i/%i.\n", tomorrow.month,
tomorrow.day, tomorrow.year);
return 0;
}
```

# Example continued

```
298 bool isLeapYear (struct date d);
```

```
// Function to find the number of days in a month
```

```
int numberOfDays (struct date d) {  
    int days;  
    const int daysPerMonth[12] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 },  
    if ( isLeapYear (d) == true && d.month == 2 )  
        days = 29;  
    else  
        days = daysPerMonth[d.month - 1];  
    return days;  
}
```

```
// Function to determine if it's a leap year
```

```
bool isLeapYear (struct date d) {  
    bool leapYearFlag;  
    if ( (d.year % 4 == 0 && d.year % 100 != 0) || d.year % 400 == 0 )  
        leapYearFlag = true; // It's a leap year  
    else  
        leapYearFlag = false; // Not a leap year  
    return leapYearFlag;  
}
```

# Example: determine tomorrow's date (Version 3)

299

```
// Program to determine tomorrow's date
#include <stdio.h>
#include <stdbool.h>
struct date
{
    int month;
    int day;
    int year;
};
```

*Declares a function that takes a struct date as a parameter and returns a struct date*

```
struct date dateUpdate (struct date today);

int main (void){
    struct date thisDay, nextDay;
    printf ("Enter today's date (mm dd yyyy): ");
    scanf ("%i%i%i", &thisDay.month, &thisDay.day,
                                                    &thisDay.year);

    nextDay = dateUpdate (thisDay);
    printf ("Tomorrow's date is %i/%i/%i.\n",nextDay.month,
                                                    nextDay.day, nextDay.year );
    return 0;
}
```

# Example continued

300

```
int numberOfDays (struct date d);

// Function to calculate tomorrow's date
struct date dateUpdate (struct date today) {
    struct date tomorrow;
    if ( today.day != numberOfDays (today) ) {
        tomorrow.day = today.day + 1;
        tomorrow.month = today.month;
        tomorrow.year = today.year;
    }
    else if ( today.month == 12 ) { // end of year
        tomorrow.day = 1;
        tomorrow.month = 1;
        tomorrow.year = today.year + 1;
    }
    else { // end of month
        tomorrow.day = 1;
        tomorrow.month = today.month + 1;
        tomorrow.year = today.year;
    }
    return tomorrow;
}
```

# Example: Update time by one second

301

```
// Program to update the time by one second
#include <stdio.h>
struct time
{
    int hour;
    int minutes;
    int seconds;
};

struct time timeUpdate (struct time now);

int main (void) {
    struct time currentTime, nextTime;
    printf ("Enter the time (hh:mm:ss): ");
    scanf ("%i:%i:%i", &currentTime.hour, &currentTime.minutes,
            &currentTime.seconds);
    nextTime = timeUpdate (currentTime);
    printf ("Updated time is %.2i:%.2i:%.2i\n", nextTime.hour,
            nextTime.minutes, nextTime.seconds );
    return 0;
}
```

# Example continued

302

```
// Function to update the time by one second
struct time timeUpdate (struct time now) {
    ++now.seconds;
    if ( now.seconds == 60 ) { // next minute
        now.seconds = 0;
        ++now.minutes;
        if ( now.minutes == 60 ) { // next hour
            now.minutes = 0;
            ++now.hour;
            if ( now.hour == 24 ) // midnight
                now.hour = 0;
        }
    }
    return now;
}
```

Parameters of a  
struct type are  
passed by copy  
of value

# Initializing structures. Compound literals

303

```
struct date today = { 7, 2, 2005 };  
struct time this_time = { 3, 29, 55 };  
today = (struct date) { 9, 25, 2004 };  
today = (struct date) { .month = 9, .day = 25, .year = 2004 };
```

# Arrays of structures

304

```
struct time { . . . };

struct time timeUpdate (struct time now);

int main (void) {
    struct time testTimes[5] =
        { { 11, 59, 59 }, { 12, 0, 0 }, { 1, 29, 59 },
          { 23, 59, 59 }, { 19, 12, 27 } };
    int i;
    for ( i = 0; i < 5; ++i ) {
        printf ("Time is %.2i:%.2i:%.2i", testTimes[i].hour,
                testTimes[i].minutes, testTimes[i].seconds);
        testTimes[i] = timeUpdate (testTimes[i]);
        printf (" ...one second later it's %.2i:%.2i:%.2i\n",
                testTimes[i].hour, testTimes[i].minutes,
                testTimes[i].seconds);
    }
    return 0;
}
```



testTimes[0]	.hour	11
	.minutes	59
	.seconds	59
testTimes[1]	.hour	12
	.minutes	0
	.seconds	0
testTimes[2]	.hour	1
	.minutes	29
	.seconds	59
testTimes[3]	.hour	23
	.minutes	59
	.seconds	59
testTimes[4]	.hour	19
	.minutes	12
	.seconds	27

# Structures containing structures

306

```
struct dateAndTime  
{  
    struct date sdate;  
    struct time stime;  
};
```

. . .

```
struct dateAndTime event;
```

```
event.sdate = dateUpdate (event.sdate);
```

```
event.sdate.month = 10;
```

# Structures containing arrays

307

```
struct month
{
    int numberOfDays;
    char name[3];
};

struct month aMonth;

. . .

aMonth.numberOfDays = 31;
aMonth.name[0] = 'J';
aMonth.name[1] = 'a';
aMonth.name[2] = 'n';
```

# Structures and pointers

308

- ❑ Structure pointers are just like pointers to ordinary variables. The declaration
- ❑ `struct point *pp;`
- ❑ says that `pp` is a pointer to a structure of type `struct point`. If `pp` points to a point structure, `*pp` is the structure, and `(*pp).x` and `(*pp).y` are the members.
- ❑ To use `pp`:
- ❑ `struct point origin, *pp;`
- ❑ `pp = &origin;`
- ❑ `printf("origin is (%d,%d)\n", (*pp).x, (*pp).y);`
- ❑ The parentheses are necessary in `(*pp).x` because the precedence of the structure member operator `.` is higher than `*`. The expression `*pp.x` means `*(pp.x)`, which is illegal here because `x` is not a pointer.

# Pointers to structures

309

- Pointers to structures are so frequently used that an alternative notation is provided as a shorthand. If `p` is a pointer to a structure, then referring to a particular member can be done by:
  - `p->member-of-structure`
  - Equivalent with `(*p).member-of-structure`
  - `printf("origin is (%d,%d)\n", pp->x, pp->y);`
  - Both `.` and `->` associate from left to right, so if we have
  - `struct rect r, *rp = &r;`
  - then these four expressions are equivalent:
    - `r.pt1.x`
    - `rp->pt1.x`
    - `(r.pt1).x`
    - `(rp->pt1).x`

# Precedence of operators

310

- The structure operators `.` and `->`, together with `()` for function calls and `[]` for subscripts, are at the top of the precedence hierarchy and thus bind very tightly.
- For example, given the declaration
- ```
struct {
```
- ```
    int len;
```
- ```
    char *str;
```
- ```
} *p;
```
- then `++p->len` increments `len`, not `p`, because the implied parenthesization is `++(p->len)`.
- Parentheses can be used to alter binding: `(++p)->len` increments `p` before accessing `len`, and `(p++)->len` increments `p` afterward. (This last set of parentheses is unnecessary.)
- `*p->str` fetches whatever `str` points to;
- `*p->str++` increments `str` after accessing whatever it points to (just like `*s++`)
- `(*p->str)++` increments whatever `str` points to
- `*p++->str` increments `p` after accessing whatever `str` points to.

# Unions

311

- ❑ Similar to **structures**.
- ❑ The syntax to declare/define a union is also similar to that of a structure.
- ❑ The only differences is in terms of storage.
- ❑ In **structure** each member has its own storage location, whereas all members of **union** uses a single shared memory location which is equal to the size of its largest data member.

```
struct st
{ int x;
  int y;
}s1;
S1.x=10; s1.y=20;
```

```
union un
{int x;
 int y;
}u1;
u1.x=10;
```

# Precedence of operators

312

- The structure operators `.` and `->`, together with `()` for function calls and `[]` for subscripts, are at the top of the precedence hierarchy and thus bind very tightly.
- For example, given the declaration
- `struct {`
- `int len;`
- `char *str;`
- `} *p;`
- then `++p->len` increments `len`, not `p`, because the implied parenthesization is `++(p->len)`.
- Parentheses can be used to alter binding: `(++p)->len` increments `p` before accessing `len`, and `(p++)->len` increments `p` afterward. (This last set of parentheses is unnecessary.)
- `*p->str` fetches whatever `str` points to;
- `*p->str++` increments `str` after accessing whatever it points to (just like `*s++`)
- `(*p->str)++` increments whatever `str` points to
- `*p++->str` increments `p` after accessing whatever `str` points to.



# Enumerated data type

313

- ❑ You can use the enumerated type to declare symbolic names to represent integer constants.
- ❑ By using the `enum` keyword, you can create a new "type" and specify the values it may have.
- ❑ Actually, enum constants are type `int`; therefore, they can be used wherever you would use an `int`.
- ❑ The purpose of enumerated types is to enhance the readability of a program.
- ❑ `enum primaryColor { red, yellow, blue };`
- ❑ `enum primaryColor myColor, gregsColor;`
- ❑ `myColor = red;`
- ❑ `if ( gregsColor == yellow ) ...`

# Enumerated data type

314

- ❑ The C compiler actually treats enumeration identifiers as integer constants. Beginning with the first name in the list, the compiler assigns sequential integer values to these names, starting with 0.
- ❑ `enum month thisMonth;`
- ❑ `...`
- ❑ `thisMonth = february;`
- ❑ the value 1 is assigned to `thisMonth` because it is the second identifier listed inside the enumeration list.
- ❑ If you want to have a specific integer value associated with an enumeration identifier, the integer can be assigned to the identifier when the data type is defined. Enumeration identifiers that subsequently appear in the list are assigned sequential integer values beginning with the specified integer value plus 1. For example, in the definition
- ❑ `enum direction { up, down, left = 10, right };`
- ❑ an enumerated data type `direction` is defined with the values `up`, `down`, `left`, and `right`. The compiler assigns the value 0 to `up` because it appears first in the list; 1 to `down` because it appears next; 10 to `left` because it is explicitly assigned this value; and 11 to `right` because it appears immediately after `left` in the list.

# Example: enum

315

```
// Program to print the number of days in a month
#include <stdio.h>
int main (void)
{
    enum month { january = 1, february, march, april, may,
june,july, august, september, october, november, december
};
    enum month aMonth;
    int days;
    printf ("Enter month number: ");
    scanf ("%i", &aMonth);
```

# Cont.

316

```
switch (aMonth ) {
    case january: case march: case may: case july:
    case august: case october: case december:
        days = 31; break;
    case april: case june: case september: case november:
        days = 30; break;
    case february:
        days = 28; break;
    default:
        printf ("bad month number\n");
        days = 0; break;
}
if ( days != 0 )
    printf ("Number of days is %i\n", days);
if ( amonth == february )
    printf ("...or 29 if it's a leap year\n");
return 0;
}
```

# The typedef statement

317

- C provides a capability that enables you to **assign an alternate name to a data type**. This is done with a statement known as `typedef`.

```
typedef type_description type_name;
```

- The statement  

```
typedef int Counter;
```
- defines the name `Counter` to be equivalent to the C data type `int`. Variables can subsequently be declared to be of type `Counter`, as in the following statement:  

```
Counter j, n;
```
- The C compiler actually treats the declaration of the variables `j` and `n`, shown in the preceding code, as normal integer variables.
- The main advantage of the use of the `typedef` in this case is in the added **readability** that it lends to the definition of the variables.
- the `typedef` statement **does not actually define a new type—only a new type name**.

# The typedef statement

318

- In forming a typedef definition, proceed as though a normal variable declaration were being made. Then, place the new type name where the variable name would normally appear. Finally, in front of everything, place the keyword typedef:
- `typedef char Linebuf [81];`
- defines a type called Linebuf, which is an array of 81 characters. Subsequently declaring variables to be of type Linebuf, as in
- `Linebuf text, inputLine;`

```
typedef struct
{
    int month;
    int day;
    int year;
} Date;
```

```
Date birthdays[100];
```

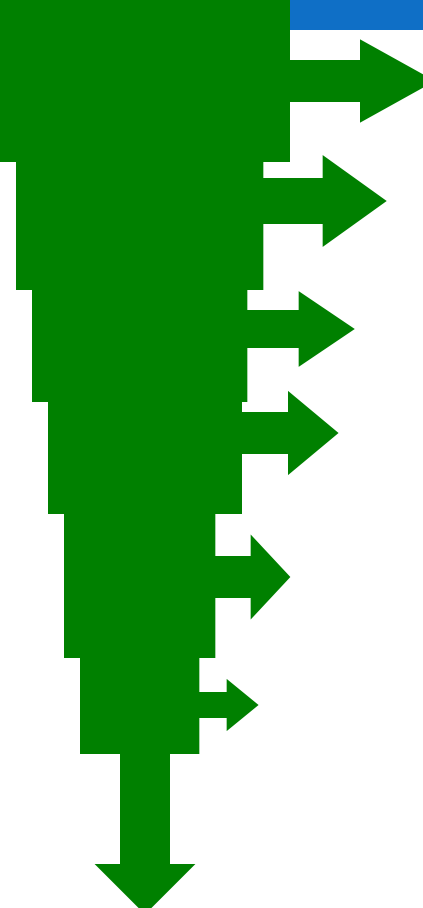
# Data type conversions

319

- *Expressions: Operands and operators*
- *Operands may be of different types; issue: how is the expression evaluated and what is the type of the result ?*
- sometimes conversions are implicitly made by the system when expressions are evaluated !
- The C compiler adheres to strict rules when it comes to evaluating expressions that consist of different data types.
- **Essence of automatic conversion rules: convert “smaller” type to “bigger” type**
- Example: the case examined in lecture 2 was with the data types `float` and `int`: an operation that involved a float and an int was carried out as a floating-point operation, ***the integer data item being automatically converted to floating point.***
- the ***type cast operator*** can be used to explicitly dictate a conversion.

# Rules for automatic conversions

320

- 
1. If either operand is of type `long double`, the other is converted to `long double`, and that is the type of the result.
  2. If either operand is of type `double`, the other is converted to `double`, and that is the type of the result.
  3. If either operand is of type `float`, the other is converted to `float`, and that is the type of the result.
  4. If either operand is of type `_Bool`, `char`, `short int`, or of an enumerated data type, it is converted to `int`.
  5. If either operand is of type `long long int`, the other is converted to `long long int`, and that is the type of the result.
  6. If either operand is of type `long int`, the other is converted to `long int`, and that is the type of the result.
  7. If this step is reached, both operands are of type `int`, and that is the type of the result.

Example: `f` is defined to be a float, `i` an int, `l` a long int, and `s` a short int variable: evaluate expression  $f * i + l / s$



# Sign extensions

321

- Conversion of a signed integer to a longer integer results in extension of the sign (0 or 1) to the left;
  - ▣ This ensures that a short int having a value of  $-5$  will also have the value  $-5$  when converted to a long int.
- Conversion of an unsigned integer to a longer integer results in zero fill to the left.

# Signed integer representation

322

- The representation of negative numbers: Most computers represent such numbers using the **two's complement notation**.
- Using this notation, the leftmost bit represents the *sign* bit. If this bit is 1, the number is negative; otherwise, the bit is 0 and the number is positive. The remaining bits represent the value of the number.
- The value  $w$  of an  $N$ -bit integer is given by the formula:

$$w = -a_{N-1}2^{N-1} + \sum_{i=0}^{N-2} a_i 2^i$$

# Two's complement example

323

$$w = -a_{N-1}2^{N-1} + \sum_{i=0}^{N-2} a_i 2^i$$

□  $N=8$

□  $W = 11111011$

□  $W = (-1) * 2^7 + 1 * 2^6 + 1 * 2^5 + 1 * 2^4 + 1 * 2^3 + 0 + 1 * 2^1 + 1 * 2^0 = -5$

# Two's complement example

324

- A convenient way to convert a negative number from decimal to binary:
  1. add 1 to the value
  2. express the absolute value of the result in binary
  3. and then “complement” all the bits; that is, change all 1s to 0s and 0s to 1s.
- Example: convert -5 to binary:
  1. First 1 is added, which gives -4;
  2. Value 4 expressed in binary is 00000100
  3. Complementing the bits produces 11111011.

# Signed & Unsigned types

325

- If in an expression appear both signed and unsigned operands, **signed operands are automatically converted to unsigned**
- Converting signed to unsigned:
  - ▣ No change in bit representation
  - ▣ Nonnegative values unchanged
  - ▣ **Negative values change into positive values !**
  - ▣ **Example:** `int x=-5; unsigned int ux=(unsigned int) x; printf("%ud \n",ux); // 4294966*62`
- Conversion surprises:

```
int a=-5;
unsigned int b=1;

if (a>b) printf("a is bigger than b");
else if (a<b) printf("b is bigger than a");
else printf("a equals b");
```

# Explicit conversions/casts

326

## □ Rules:

- Conversion of any value to a `_Bool` results in 0 if the value is zero and 1 otherwise.
- Conversion of a longer integer to a shorter one results in truncation of the integer on the left.
- Conversion of a floating-point value to an integer results in truncation of the decimal portion of the value. If the integer is not large enough to contain the converted floating-point value, the result is not defined, as is the result of converting a negative floating-point value to an unsigned integer.
- Conversion of a longer floating-point value to a shorter one might or might not result in rounding before the truncation occurs.

## 2.6 MACROS

# Introduction

- Preprocessing
  - ▣ Occurs before program compiled
    - Inclusion of external files
    - Definition of symbolic constants
    - Macros
    - Conditional compilation
    - Conditional execution
  - ▣ All directives begin with #
    - Can only have whitespace before directives
  - ▣ Directives not C++ statements
    - Do not end with ;



# The `#include` Preprocessor Directive

- **`#include`** directive
  - ▣ Puts copy of file in place of directive
    - Seen many times in example code
  - ▣ Two forms
    - **`#include <filename>`**
      - For standard library header files
      - Searches predesignated directories
    - **`#include "filename"`**
      - Searches in current directory
      - Normally used for programmer-defined files

# 19.2 The #include Preprocessor Directive

- Usage
  - Loading header files
    - `#include <Stdio.h>`
  - Programs with multiple source files
  - Header file
    - Has common declarations and definitions
    - Classes, structures, enumerations, function prototypes
    - Extract commonality of multiple program files

# The #define Preprocessor Directive: Symbolic Constants

## □ #define

### ■ Symbolic constants

- Constants represented as symbols
- When program compiled, all occurrences replaced

### ■ Format

- **#define *identifier replacement-text***
- **#define PI 3.14159**

### ■ Everything to right of identifier replaces text

- **#define PI=3.14159**
- Replaces PI with **"=3.14159"**
- Probably an error

### ■ Cannot redefine symbolic constants

# The #define Preprocessor Directive: Symbolic Constants

- Advantages
  - ▣ Takes no memory
- Disadvantages
  - ▣ Name not be seen by debugger (only replacement text)
  - ▣ Do not have specific data type
- **const** variables preferred

# The #define Preprocessor Directive:

## Macros

- Macro
  - ▣ Operation specified in **#define**
  - ▣ Intended for legacy C programs
  - ▣ Macro without arguments
    - Treated like a symbolic constant
  - ▣ Macro with arguments
    - Arguments substituted for replacement text
    - Macro expanded
  - ▣ Performs a text substitution
    - No data type checking

# The #define Preprocessor Directive:

## Macros

### □ Example

```
#define CIRCLE_AREA( x ) ( PI * ( x ) * ( x ) )  
area = CIRCLE_AREA( 4 );
```

becomes

```
area = ( 3.14159 * ( 4 ) * ( 4 ) );
```

### □ Use parentheses

■ Without them,

```
#define CIRCLE_AREA( x ) PI * x * x  
area = CIRCLE_AREA( c + 2 );
```

becomes

```
area = 3.14159 * c + 2 * c + 2;
```

which evaluates incorrectly

# The #define Preprocessor Directive: Macros

## □ Multiple arguments

```
#define RECTANGLE_AREA( x, y )  ( ( x ) * ( y ) )  
rectArea = RECTANGLE_AREA( a + 4, b + 7 );
```

becomes

```
rectArea = ( ( a + 4 ) * ( b + 7 ) );
```

# Predefined Symbolic Constants

## □ Five predefined symbolic constants

Symbolic constant	Description
<code>__LINE__</code>	The line number of the current source code line (an integer constant).
<code>__FILE__</code>	The presumed name of the source file (a string).
<code>__DATE__</code>	The date the source file is compiled (a string of the form " <b>Mmm dd yyyy</b> " such as " <b>Jan 19 2001</b> ").
<code>__TIME__</code>	The time the source file is compiled (a string literal of the form " <b>hh:mm:ss</b> ").



**UNIT I**  
**INTRODUCTION TO C PROGRAMMING**

**TOPIC 3: FILE HANDLING**

**DR. G R PATIL**  
**PROFESSOR & HEAD**  
**DEPT OF E&TC ENGG**  
**AIT PUNE**

# 5.1 OPEN, CLOSE, READ, WRITE AND APPEND

# Introduction

339

- A file is a container in computer storage devices used for storing data.
- **Why files are needed?**
  - When a program is terminated, the entire data is lost. Storing in a file will preserve your data even if the program terminates.
  - If you have to enter a large number of data, it will take a lot of time to enter them all. However, if you have a file containing all the data, you can easily access the contents of the file using a few commands in C.
  - You can easily move your data from one computer to another without any changes

# Introduction

340

## □ Types of Files

- When dealing with files, there are two types of files you should know about:
  - Text files
  - Binary files

## □ Text files

- Text files are the normal **.txt** files. You can easily create text files using any simple text editors such as Notepad.
- When you open those files, you'll see all the contents within the file as plain text. You can easily edit or delete the contents.
- They take minimum effort to maintain, are easily readable, and provide the least security and takes bigger storage space.

# Introduction

341

## □ Binary files

- Binary files are mostly the **.bin** files in your computer.
- Instead of storing data in plain text, they store it in the binary form (0's and 1's).
- They can hold a higher amount of data, are not readable easily, and provides better security than text files.

## □ Text versus Binary Mode

- Newlines:
  - Converted to carriage return-linefeed in text file
  - Not in Binary
- End of file :
  - Special ASCII char 26
  - Not in Binary they keep track from no of chars in directory entry of file
- Storage of numbers

# File Operations

342

- File Declaration
  - ▣ When working with files, you need to declare a pointer of type file. This declaration is needed for communication between the file and the program.
  - ▣ `FILE *file_ptr;`
- In C, you can perform four major operations on files, either text or binary:
  - ▣ Creating a new file
  - ▣ Opening an existing file
  - ▣ Closing a file
  - ▣ Reading from and writing information to a file

# File Operations

343

- Opening a file is performed using the `fopen()` function defined in the `stdio.h` header file.
- The syntax for opening a file in standard I/O is:

```
file_ptr = fopen("fileopen","mode");
```

For example,

```
fopen("E:\\CPRG\\abc.txt","w");  
fopen("E:\\CPRG\\emp.dat","rb");
```

# File Open Modes

344

Mode	Meaning of Mode	During Inexistence of file
r	Open for reading.	If the file does not exist, fopen() returns NULL.
rb	Open for reading in binary mode.	If the file does not exist, fopen() returns NULL.
w	Open for writing.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
wb	Open for writing in binary mode.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
a	Open for append. Data is added to the end of the file.	If the file does not exist, it will be created.
ab	Open for append in binary mode. Data is added to the end of the file.	If the file does not exist, it will be created.



# File Open Modes

345

Mode	Meaning of Mode	During Inexistence of file
r+	Open for both reading and writing.	If the file does not exist, fopen() returns NULL.
rb+	Open for both reading and writing in binary mode.	If the file does not exist, fopen() returns NULL.
w+	Open for both reading and writing.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
wb+	Open for both reading and writing in binary mode.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
a+	Open for both reading and appending.	If the file does not exist, it will be created.
ab+	Open for both reading and appending in binary mode.	If the file does not exist, it will be created.

# Closing a File

346

- The file (both text and binary) should be closed after reading/writing.
- Closing a file is performed using the `fclose()` function.  
`fclose(file_ptr);`
- Here, `file_ptr` is a file pointer associated with the file to be closed.

# Writing Characters into Text File

347

- **Syntax:**

**int fputc(int char, FILE \*file\_ptr)**

- **char:** character to be written.
- This is passed as its int promotion.
- **File\_ptr:** pointer to a FILE object that identifies the stream where the character is to be written.

# Writing Characters into Text File:Example

348

```
void main()
{
    int i = 0;  char s[80];  FILE *fp;
    fp = fopen("output.txt","w");
    if (fp == NULL) // Return if could not open file
        return 0;
    printf("Enter a string\n");
    gets(s);
    while(s[i]!='\0')
    {
        fputc(s[i], fp);
        i++;
    }
    fclose(fp);
}
```

# Reading Text File

349

- **Syntax:**

**int fgetc(FILE \*file\_ptr)**

- **file\_ptr:** pointer to a FILE object that identifies the stream on which the operation is to be performed.
- Returns EOF at the end of file

# Reading Text File

350

```
int main () {  
    FILE *fp; char ch;  
    fp = fopen("test.txt","r"); // open the file  
    if (fp == NULL) // Return if could not open file  
        return 0;  
    while (1) {  
        ch = fgetc(fp); // Taking input single character at a time  
        if (ch==EOF) // Checking for end of file or if (feof(fp))  
            break ;  
        printf("%c", ch);  
    } while(1);  
    fclose(fp);  
    return(0);  
}
```

# Assignment

351

- Copy one file into another
- Count Number of characters in a file
- Combine two files
- Count Number of words in a file

# Assignment

352

- ❑ Copy one file into another
- ❑ Count Number of characters in a file
- ❑ Combine two files
- ❑ Count Number of words in a file



# fgets() and fputs()

353

## □ Syntax :

**char \*fgets(char \*str, int n, FILE \*file\_ptr)**

**str** : Pointer to an array of chars where the string read is copied.

**n** : Maximum number of characters to be copied into str (including the terminating null-character).

**\*file\_ptr** : Pointer to a FILE object that identifies an input stream. stdin can

be used as argument to read from the standard input.

**returns** : the function returns str

# fgets() and fputs()

354

## □ Syntax :

**char \*fgets(char \*str, int n, FILE \*file\_ptr)**

**str** : Pointer to an array of chars where the string read is copied.

**n** : Maximum number of characters to be copied into str (including the terminating null-character).

**\*file\_ptr** : Pointer to a FILE object that identifies an input stream. stdin can be used as argument to read from the standard input.

**returns** : the function returns NULL when it finds end of file

## □ Syntax:

**int fputs(char \*str, int n, FILE \*file\_ptr)**

## □ Return Value:

On success, it returns 0. On error, it returns -1.

# fprintf()

355

## □ Syntax :

```
int fprintf(FILE *stream, const char *format [, argument, ...])
```

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

```
main(){
```

```
    FILE *fp;
```

```
    fp = fopen("file.txt", "w");//opening file
```

```
    fprintf(fp, "Hello file by fprintf...\n");//writing data into file
```

```
    fclose(fp);//closing file
```

```
}
```

# fscanf()

356

## □ Syntax:

```
int fscanf(FILE *stream, const char *format [, argument, ...])
```

The **return** value is EOF if an input failure occurs before any conversion, or the number of input items assigned if successful.

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

```
main(){
```

```
    FILE *fp;
```

```
    char buff[255]; //creating char array to store data of file
```

```
    fp = fopen("file.txt", "r");
```

```
    while(fscanf(fp, "%s", buff)!=EOF){
```

```
        printf("%s ", buff );
```

```
    }
```

```
    fclose(fp);
```

```
}
```

# Example

357

```
#include<stdio.h>
#include<conio.h>
struct emp{ char name[10]; int age;};
main(){
    struct emp e;
    FILE *p,*q;
    p = fopen("one.txt", "a");
    q = fopen("one.txt", "r");
    printf("Enter Name and Age");
    scanf("%s %d", e.name, &e.age);
    fprintf(p,"%s %d", e.name, e.age);
    fclose(p);
```

# Example

358

```
do
{
    fscanf(q,"%s %d", e.name, e.age);
    printf("%s %d", e.name, e.age);
} while(!eof(q));
}
```

# fwrite()

359

## □ Syntax:

**size\_t fwrite(const void \*ptr, size\_t size, size\_t nmemb, FILE \*stream)**

**ptr** - This is pointer to array of elements to be written

**size** - This is the size in bytes of each element to be written

**nmemb** - This is the number of elements, each one with a size of size bytes

**stream** - This is the pointer to a FILE object that specifies an output stream

```
int main () {
```

```
    FILE *fp;
```

```
    char str[] = "This is test String";
```

```
    fp = fopen( "file.txt" , "wb" );
```

```
    fwrite(str , 1 , sizeof(str) , fp );
```

```
    fclose(fp);
```

```
    return(0); }
```

# fwrite()

360

```
struct employee{
    char name[50];
    int age;} employee;
int main(){
    int n, i, chars;
    FILE *fp;
    fp = fopen("employee.dat", "wb");
    if(fp == NULL)
    {
        printf("Error opening file\n");
        exit(1);
    }
```



# fwrite()

361

```
printf("Enter the number of records you want to enter: ");
scanf("%d", &n);
for(i = 0; i < n; i++)
{
    printf("Name: ");
    gets(employee.name);
    printf("Age: ");
    scanf("%d", &employee.age);
    chars = fwrite(&employee, sizeof(employee), 1, fp);
    printf("Number of items written to the file: %d\n", chars);
}
fclose(fp);
return 0;
}
```

# fread()

362

## □ Syntax:

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream)
```

## Parameters:

**ptr** – This is the pointer to a block of memory with a minimum size of *size\*nmemb* bytes.

**size** – This is the size in bytes of each element to be read.

**nmemb** – This is the number of elements, each one with a size of **size** bytes.

**stream** – This is the pointer to a FILE object that specifies an input stream.

## □ Return Value

The total number of elements successfully read are returned as a `size_t` object, which is an integral data type. If this number differs from the `nmemb` parameter, then either an error had occurred or the End Of File was reached.

# fread()

363

```
struct employee{
    char name[50];
    int age;} employee;
int main ()
{
    FILE *infile;
    struct employee input;
    infile = fopen ("employee.dat", "rb");
    if (infile == NULL)
    {
        fprintf(stderr, "\nError opening file\n");
        exit (1);
    }
```

# fread()

364

```
while(fread(&input, sizeof(struct person), 1, infile))  
    printf ("Name = %s Age = %d \n", input.fname, input.age);  
fclose (infile);  
return 0;  
}
```

# fseek()

365

- **fseek function** changes the file position indicator for the stream pointed to by *stream*.

- **Syntax:**

```
int fseek(FILE *stream, long int offset, int whence);
```

- Parameters or Arguments

**stream:** The stream whose file position indicator is to be modified.

**Offset:** The offset to use (in bytes) when determining the new file position.

**Whence:** It can be one of the following values:

Value	Description
SEEK_SET	The new file position will be the beginning of the file plus <i>offset</i> (in bytes).
SEEK_CUR	The new file position will be the current position plus <i>offset</i> (in bytes).
SEEK_END	The new file position will be the end of the file plus <i>offset</i> (in bytes).

Returns 0 if successful

Dr G R Patil, AIT Pune

# fseek()

366

```
int main()
{
    FILE *fp;
    fp = fopen("test.txt", "r");
    // Moving pointer to end
    fseek(fp, 0, SEEK_END);
    // Printing position of pointer
    printf("%ld", ftell(fp));
    return 0;
}
```

# fseek()

367

```
#include <stdio.h>

int main () {
    FILE *fp;
    fp = fopen("file.txt","w+");
    fputs("This is tutorialspoint.com", fp);
    fseek( fp, 7, SEEK_SET );
    fputs(" C Programming Language", fp);
    fclose(fp);
    return(0);
}
```

# fseek()

368

```
#include <stdio.h>

struct employee{
    char name[50];
    int age;} employee;

int main(){
    int n, i, chars;
    FILE *fp;
    fp = fopen("employee.dat", "wb+");
    if(fp == NULL)
    {
        printf("Error opening file\n");
        exit(1);
    }
```



# fseek()

369

```
fseek(fp, 0, SEEK_END);  
printf("Name: ");  
gets(employee.name);  
printf("Age: ");  
scanf("%d", &employee.age);  
fwrite(&employee, sizeof(employee), 1, fp);  
fclose(fp);  
return 0;  
}
```

?

# Exercise

371

- ❑ Q1. Develop a Check writer program: Amount will entered by user in figures convert it into words. ( BT level 6)

E.g. Input : 32510.20

Output: Thirty two thousand five hundred and ten Rupees and twenty Paisa.

- ❑ Q2. Write a program to count number of words in a file. (BT level 6)
- ❑ Q3. Write a program to Perform various operations on matrices (BT level 6)
- ❑ Q4. Create and manage database of students using file. Provide a good user interface (BT level 6)

# Thank You