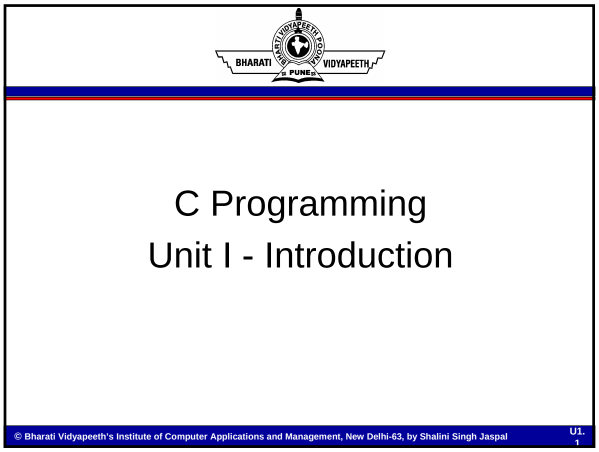
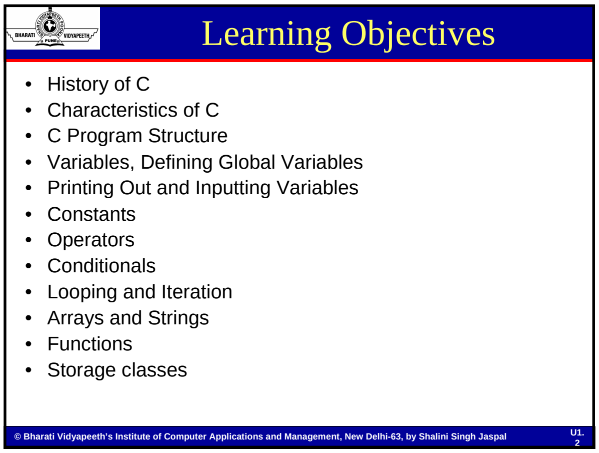
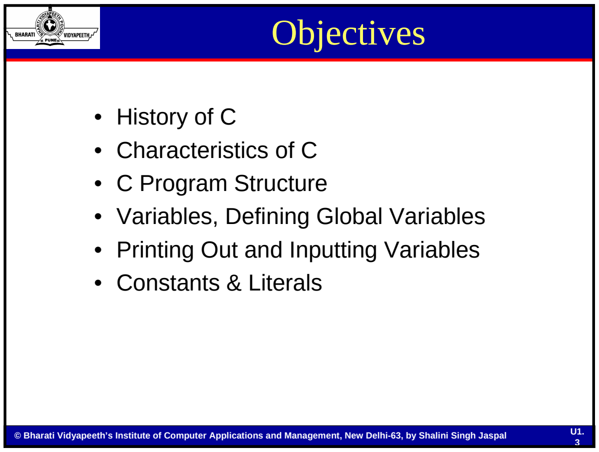


C Programming
Unit I - Introduction



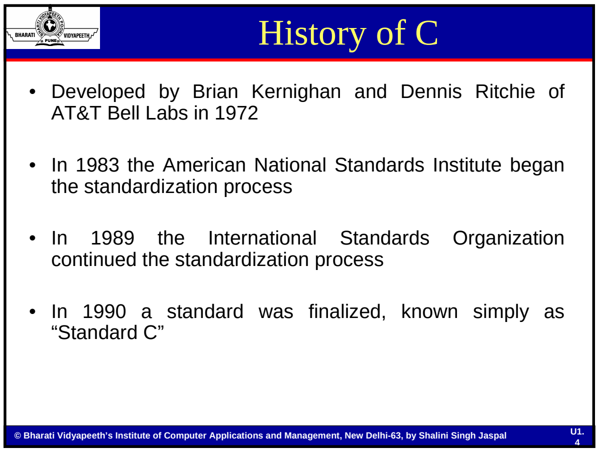
Learning Objectives

- History of C
- Characteristics of C
- C Program Structure
- Variables, Defining Global Variables
- Printing Out and Inputting Variables
- Constants
- Operators
- Conditionals
- Looping and Iteration
- Arrays and Strings
- Functions
- Storage classes



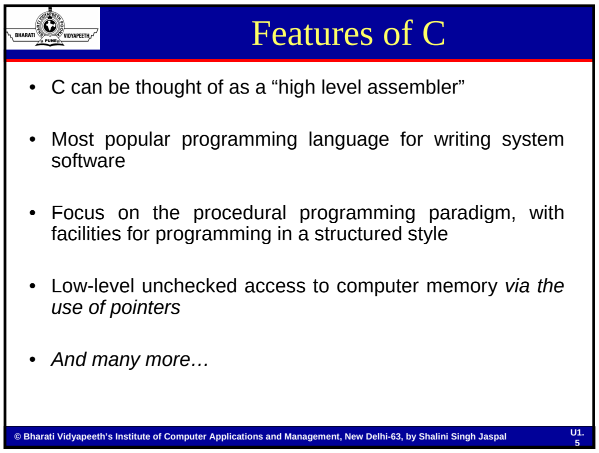
Objectives

- History of C
- Characteristics of C
- C Program Structure
- Variables, Defining Global Variables
- Printing Out and Inputting Variables
- Constants & Literals



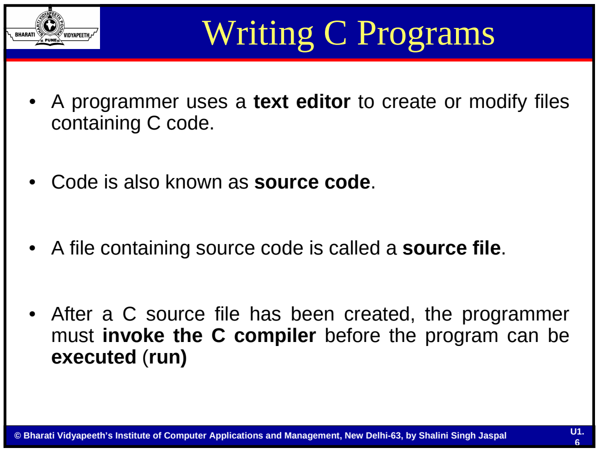
History of C

- Developed by Brian Kernighan and Dennis Ritchie of AT&T Bell Labs in 1972
- In 1983 the American National Standards Institute began the standardization process
- In 1989 the International Standards Organization continued the standardization process
- In 1990 a standard was finalized, known simply as “Standard C”



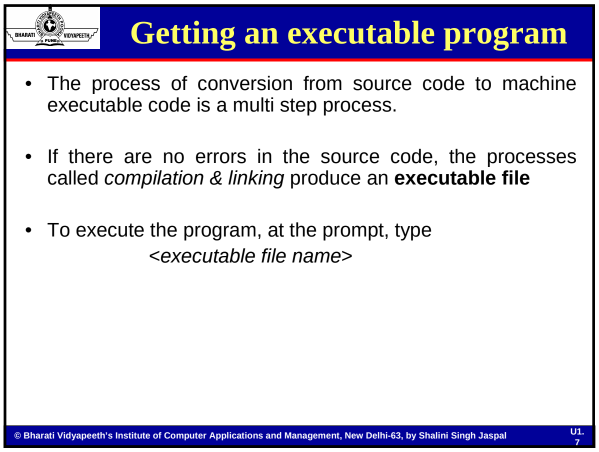
Features of C

- C can be thought of as a “high level assembler”
- Most popular programming language for writing system software
- Focus on the procedural programming paradigm, with facilities for programming in a structured style
- Low-level unchecked access to computer memory *via the use of pointers*
- *And many more…*



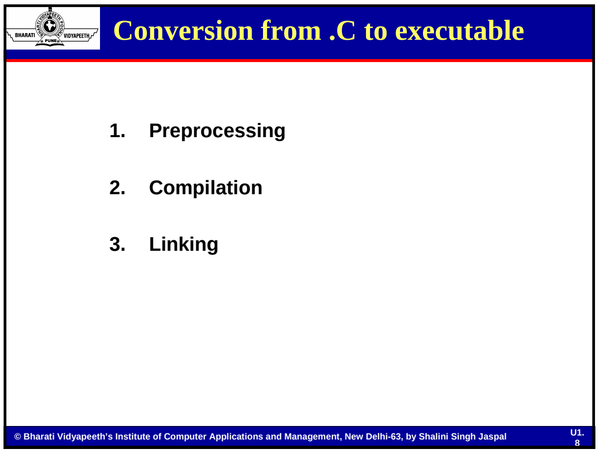
Writing C Programs

- A programmer uses a **text editor** to create or modify files containing C code.
- Code is also known as **source code**.
- A file containing source code is called a **source file**.
- After a C source file has been created, the programmer must **invoke the C compiler** before the program can be **executed** (**run)**



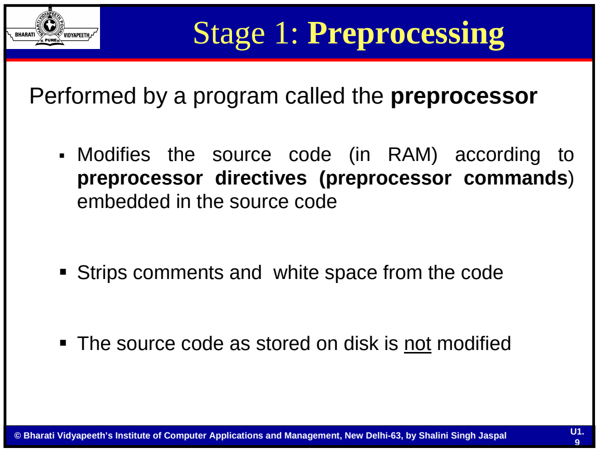
Getting an executable program

- The process of conversion from source code to machine executable code is a multi step process.
- If there are no errors in the source code, the processes called *compilation & linking* produce an **executable file**
- To execute the program, at the prompt, type
  <executable file name>



Conversion from .C to executable

1. **Preprocessing**
2. **Compilation**
3. **Linking**



Stage 1: **Preprocessing**

Performed by a program called the **preprocessor**

- Modifies the source code (in RAM) according to **preprocessor directives (preprocessor commands**) embedded in the source code
- Strips comments and white space from the code
- The source code as stored on disk is not modified

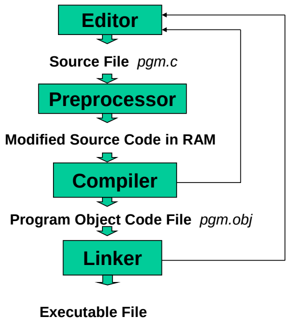
Stage 2: **Compilation**

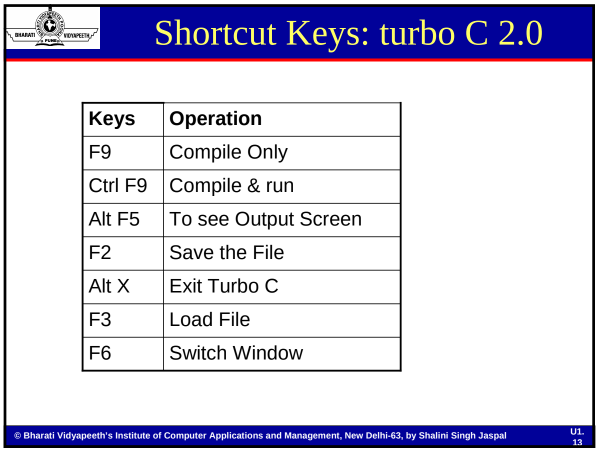
- Performed by a program called the **compiler.**
- Checks for **syntax errors** and **warnings**
- Translates the preprocessor-modified source code into **object code (machine code).**
- Saves the object code to a disk file (**.obj**)
- If any compiler errors are received, no object code file will be generated.
  - *An object code file will be generated if only warnings, not errors, are received.*

Stage 3: **Linking**

- Combines the program object code with other object code to produce the executable file.
- The other object code can come from the
  - Run-Time Library
  - other libraries
  - or object files that you have created.
- Saves the executable code (.exe) to a disk file.
- *If any linker errors are received, no executable file is generated.*

Developing a C Program

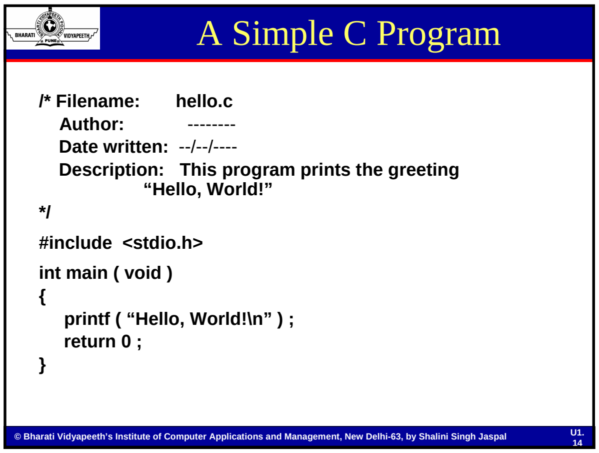




Shortcut Keys: turbo C 2.0

Keys	Operation
F9	Compile Only
Ctrl F9	Compile & run
Alt F5	To see Output Screen
F2	Save the File
Alt X	Exit Turbo C
F3	Load File
F6	Switch Window

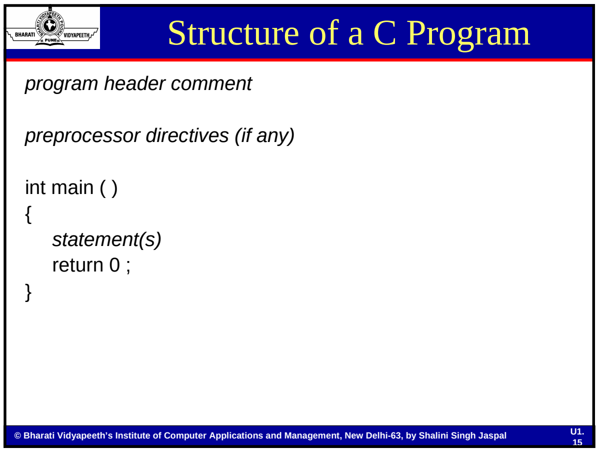
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 13



A Simple C Program

```
/* Filename:      hello.c
   Author:          --------
   Date written:  --/--/----
   Description:   This program prints the greeting
                  “Hello, World!”
*/
#include  <stdio.h>
int main ( void )
{
    printf ( “Hello, World!\n” ) ;
    return 0 ;
}
```

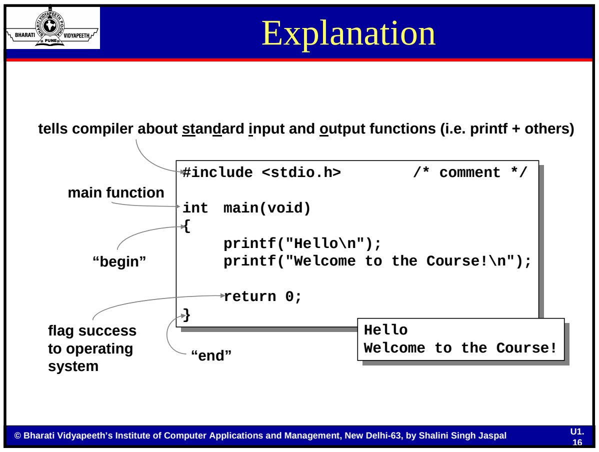
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 14



Structure of a C Program

```
program header comment

preprocessor directives (if any)

int main ( )
{
    statement(s)
    return 0 ;
}
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 15



Explanation

tells compiler about standard input and output functions (i.e. printf + others)

main function

"begin"

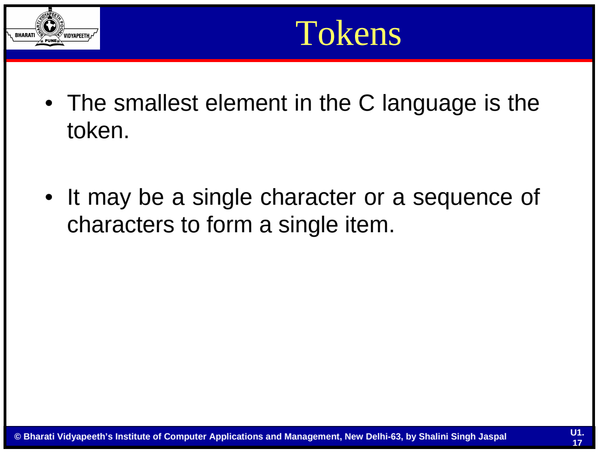
flag success to operating system

"end"

```
#include <stdio.h>        /* comment */

int  main(void)
{
     printf("Hello\n");
     printf("Welcome to the Course!\n");

     return 0;
}
```

```
Hello
Welcome to the Course!
```

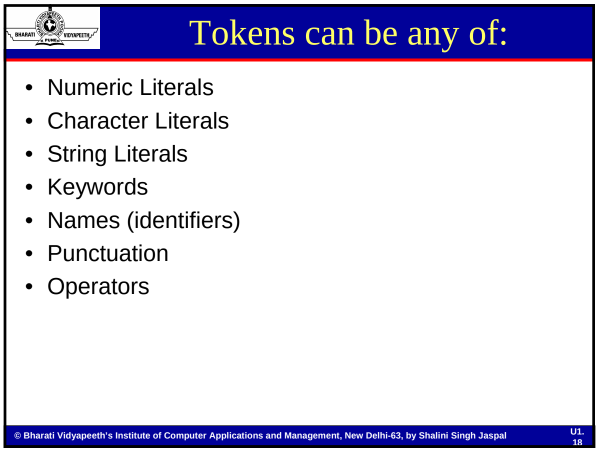
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 16



Tokens

- The smallest element in the C language is the token.
- It may be a single character or a sequence of characters to form a single item.

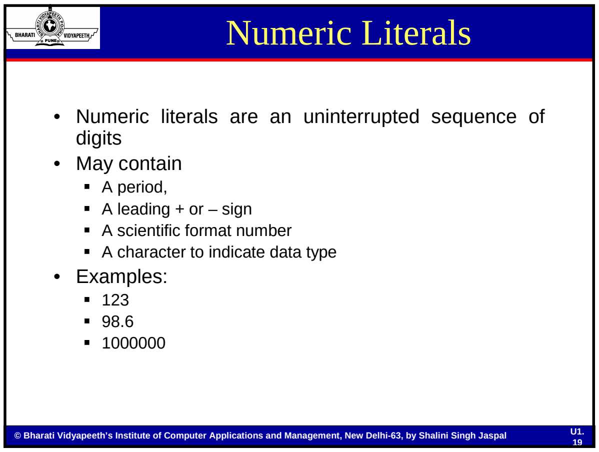
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 17



Tokens can be any of:

- Numeric Literals
- Character Literals
- String Literals
- Keywords
- Names (identifiers)
- Punctuation
- Operators

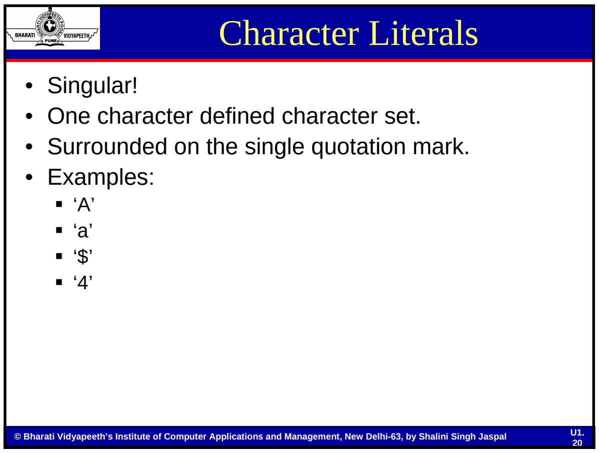
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 18



Numeric Literals

- Numeric literals are an uninterrupted sequence of digits
- May contain
  - A period,
  - A leading + or – sign
  - A scientific format number
  - A character to indicate data type
- Examples:
  - 123
  - 98.6
  - 1000000

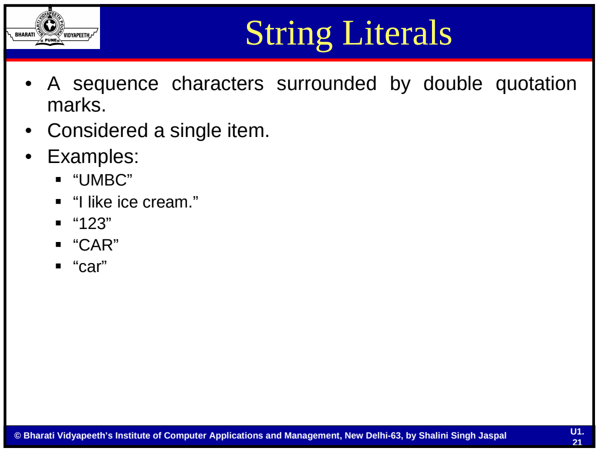
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 19



Character Literals

- Singular!
- One character defined character set.
- Surrounded on the single quotation mark.
- Examples:
  - ‘A’
  - ‘a’
  - ‘$’
  - ‘4’

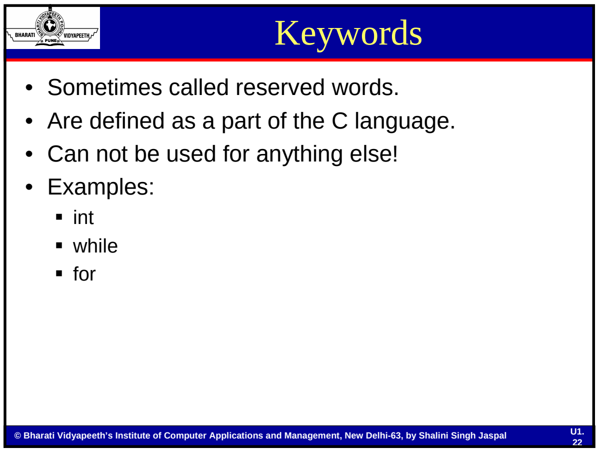
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 20



String Literals

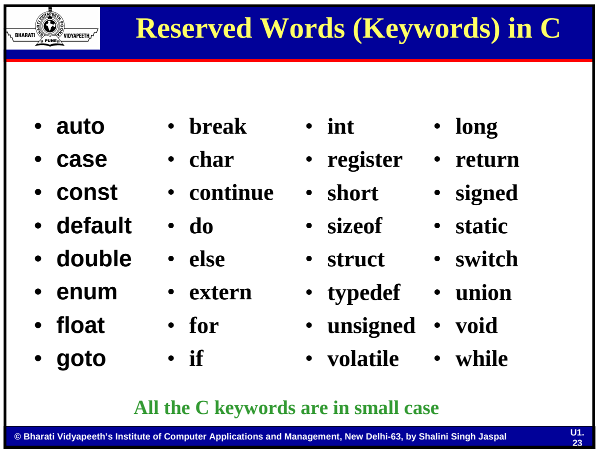
- A sequence characters surrounded by double quotation marks.
- Considered a single item.
- Examples:
  - “UMBC”
  - “I like ice cream.”
  - “123”
  - “CAR”
  - “car”

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 21



Keywords

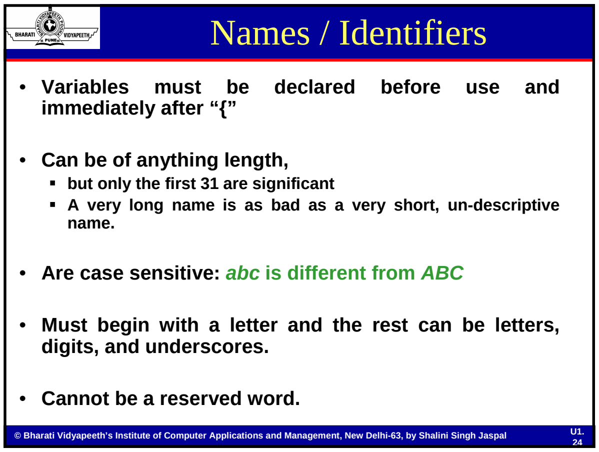
- Sometimes called reserved words.
- Are defined as a part of the C language.
- Can not be used for anything else!
- Examples:
  - int
  - while
  - for



Reserved Words (Keywords) in C

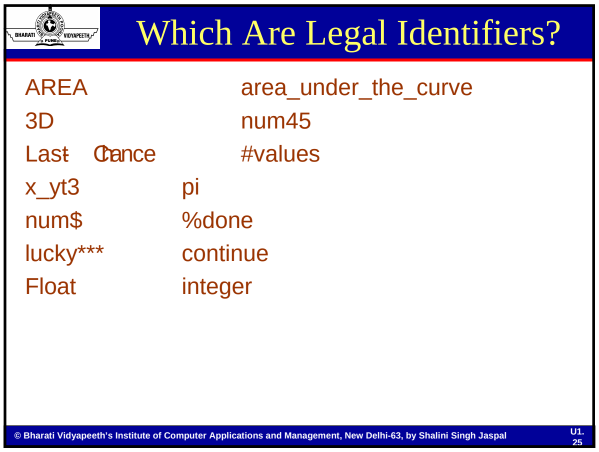
- auto
- case
- const
- default
- double
- enum
- float
- goto
- break
- char
- continue
- do
- else
- extern
- for
- if
- int
- register
- short
- sizeof
- struct
- typedef
- unsigned
- volatile
- long
- return
- signed
- static
- switch
- union
- void
- while

All the C keywords are in small case



Names / Identifiers

- **Variables must be declared before use and immediately after "{"**
- **Can be of anything length,**
  - **but only the first 31 are significant**
  - **A very long name is as bad as a very short, un-descriptive name.**
- **Are case sensitive: *abc* is different from *ABC***
- **Must begin with a letter and the rest can be letters, digits, and underscores.**
- **Cannot be a reserved word.**

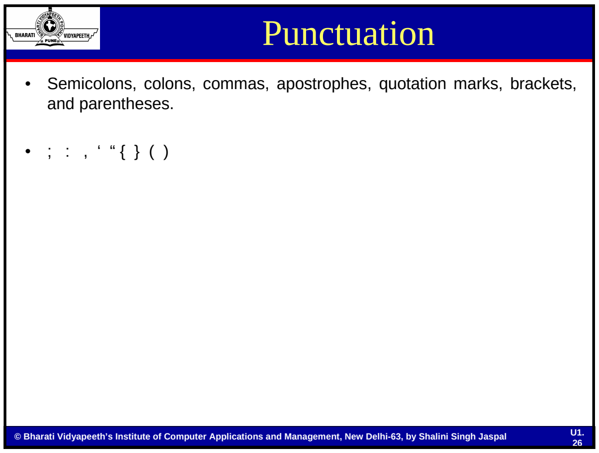


Which Are Legal Identifiers?

AREA
3D
Last- Chance
x_yt3
num$
lucky***
Float

area_under_the_curve
num45
#values
pi
%done
continue
integer

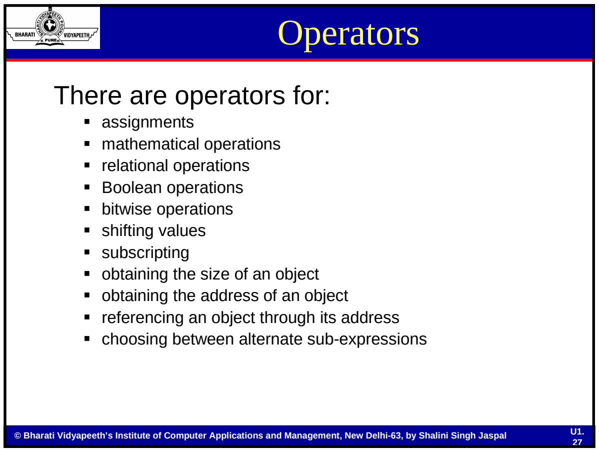
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1.25



Punctuation

- Semicolons, colons, commas, apostrophes, quotation marks, brackets, and parentheses.
- ; : , ‘ “ { } ( )

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1.26

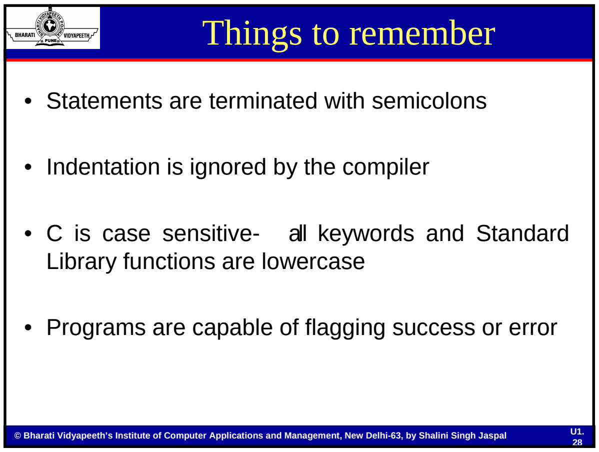


Operators

There are operators for:

- assignments
- mathematical operations
- relational operations
- Boolean operations
- bitwise operations
- shifting values
- subscripting
- obtaining the size of an object
- obtaining the address of an object
- referencing an object through its address
- choosing between alternate sub-expressions

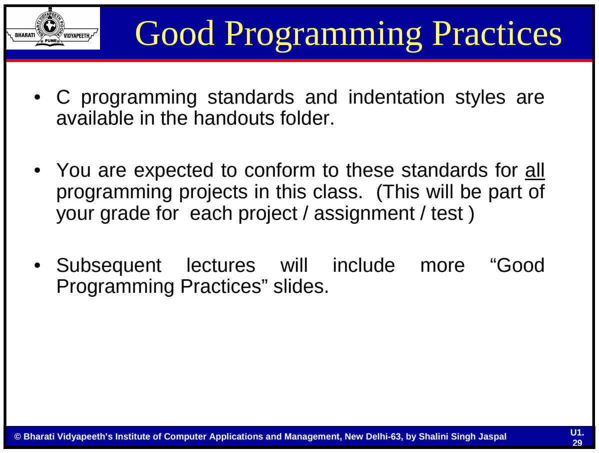
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1.27



Things to remember

- Statements are terminated with semicolons
- Indentation is ignored by the compiler
- C is case sensitive- all keywords and Standard Library functions are lowercase
- Programs are capable of flagging success or error

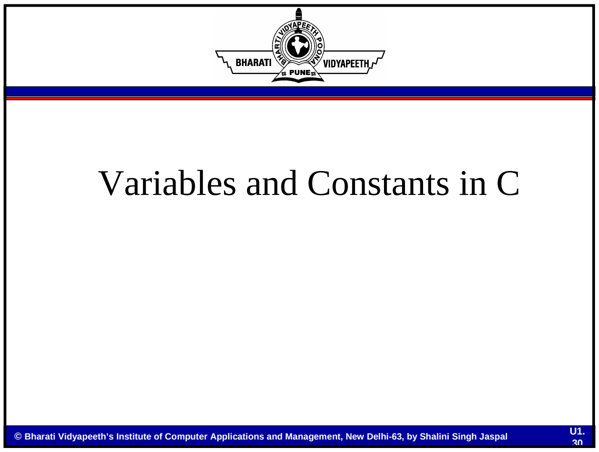
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 28



Good Programming Practices

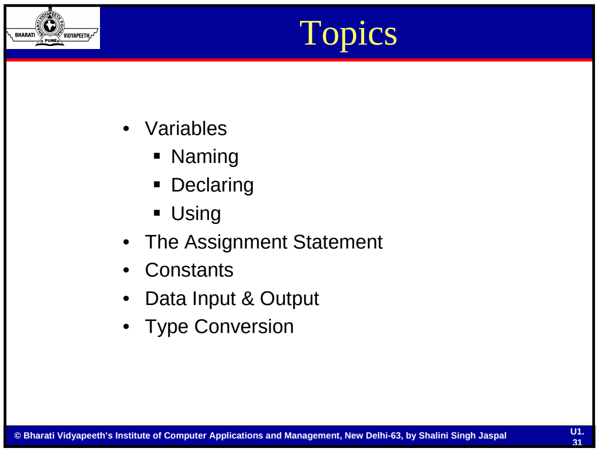
- C programming standards and indentation styles are available in the handouts folder.
- You are expected to conform to these standards for all programming projects in this class. (This will be part of your grade for each project / assignment / test)
- Subsequent lectures will include more "Good Programming Practices" slides.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 29



Variables and Constants in C

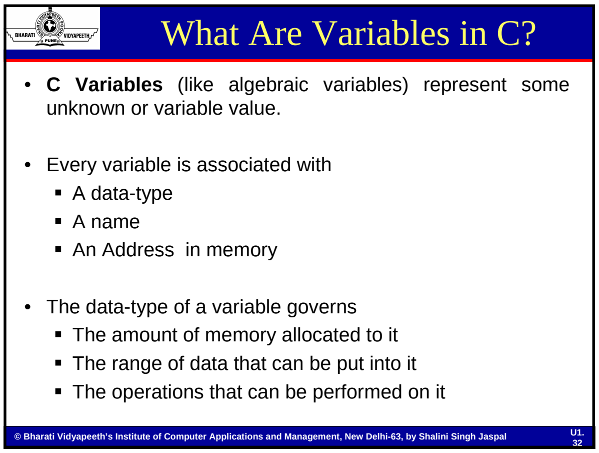
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 30



Topics

- Variables
  - Naming
  - Declaring
  - Using
- The Assignment Statement
- Constants
- Data Input & Output
- Type Conversion

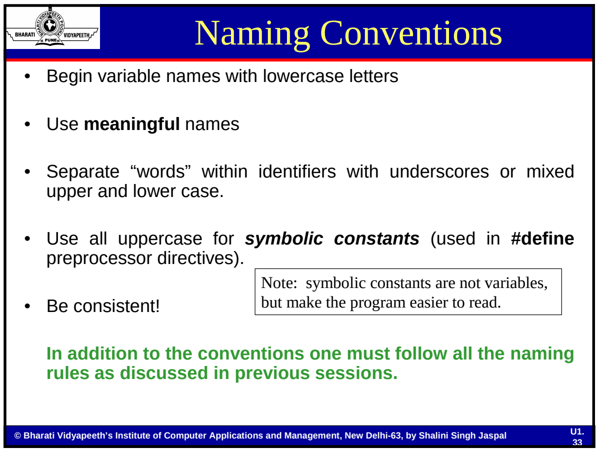
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 31



What Are Variables in C?

- **C Variables** (like algebraic variables) represent some unknown or variable value.
- Every variable is associated with
  - A data-type
  - A name
  - An Address in memory
- The data-type of a variable governs
  - The amount of memory allocated to it
  - The range of data that can be put into it
  - The operations that can be performed on it

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 32



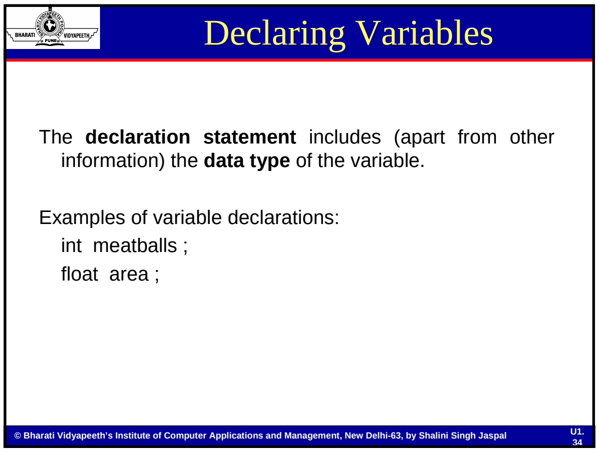
Naming Conventions

- Begin variable names with lowercase letters
- Use **meaningful** names
- Separate "words" within identifiers with underscores or mixed upper and lower case.
- Use all uppercase for ***symbolic constants*** (used in **#define** preprocessor directives).
- Be consistent!

Note: symbolic constants are not variables, but make the program easier to read.

In addition to the conventions one must follow all the naming rules as discussed in previous sessions.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 33

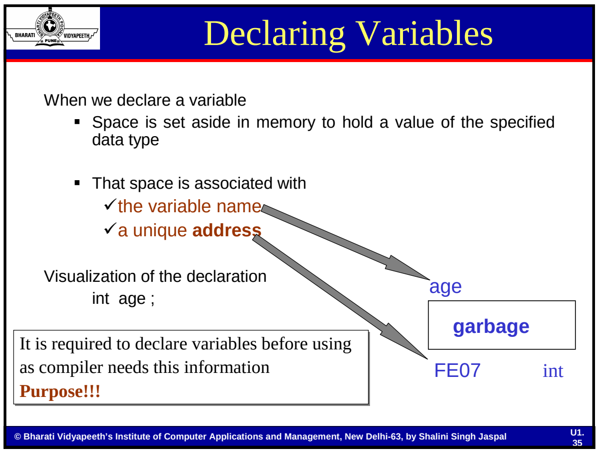


Declaring Variables

The **declaration statement** includes (apart from other information) the **data type** of the variable.

Examples of variable declarations:

```
int  meatballs ;
float  area ;
```



Declaring Variables

When we declare a variable

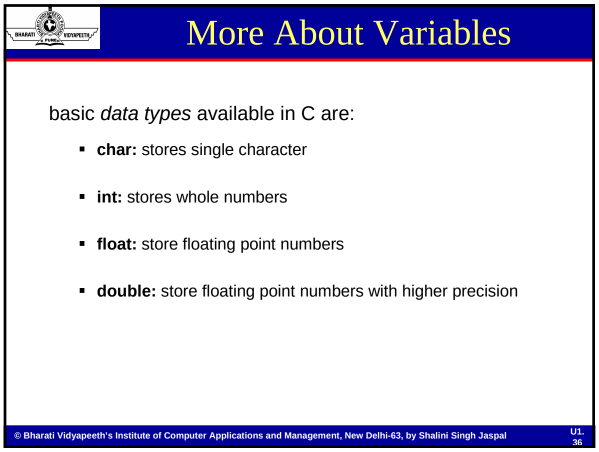
- Space is set aside in memory to hold a value of the specified data type
- That space is associated with
  - ✓the variable name
  - ✓a unique **address**

Visualization of the declaration

```
int  age ;
```

It is required to declare variables before using as compiler needs this information

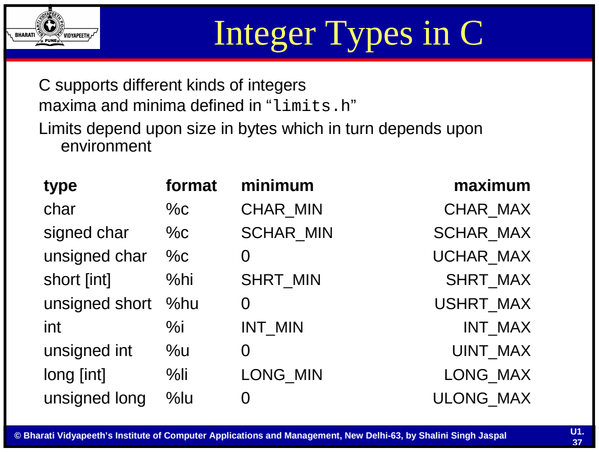
Purpose!!!



More About Variables

basic *data types* available in C are:

- **char:** stores single character
- **int:** stores whole numbers
- **float:** store floating point numbers
- **double:** store floating point numbers with higher precision



Integer Types in C

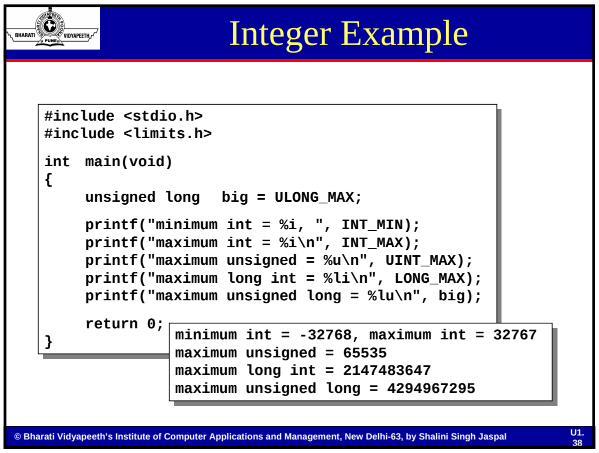
C supports different kinds of integers

maxima and minima defined in “`limits.h`”

Limits depend upon size in bytes which in turn depends upon environment

type	format	minimum	maximum
char	%c	CHAR_MIN	CHAR_MAX
signed char	%c	SCHAR_MIN	SCHAR_MAX
unsigned char	%c	0	UCHAR_MAX
short [int]	%hi	SHRT_MIN	SHRT_MAX
unsigned short	%hu	0	USHRT_MAX
int	%i	INT_MIN	INT_MAX
unsigned int	%u	0	UINT_MAX
long [int]	%li	LONG_MIN	LONG_MAX
unsigned long	%lu	0	ULONG_MAX

U1.37

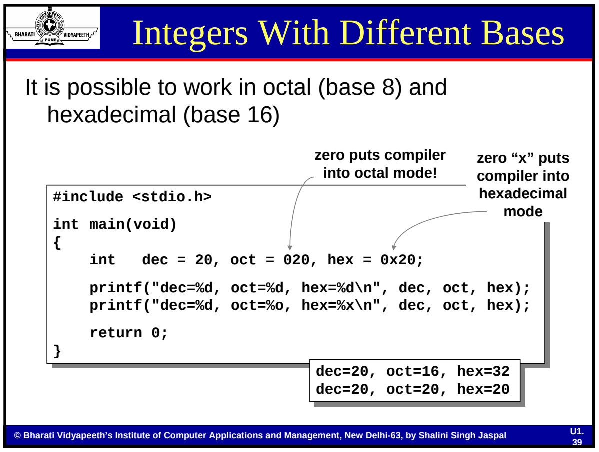


Integer Example

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

int  main(void)
{
     unsigned long   big = ULONG_MAX;

     printf("minimum int = %i, ", INT_MIN);
     printf("maximum int = %i\n", INT_MAX);
     printf("maximum unsigned = %u\n", UINT_MAX);
     printf("maximum long int = %li\n", LONG_MAX);
     printf("maximum unsigned long = %lu\n", big);

     return 0;
}
```

```
minimum int = -32768, maximum int = 32767
maximum unsigned = 65535
maximum long int = 2147483647
maximum unsigned long = 4294967295
```

U1.38



Integers With Different Bases

It is possible to work in octal (base 8) and hexadecimal (base 16)

zero puts compiler into octal mode!

zero “x” puts compiler into hexadecimal mode

```
#include <stdio.h>

int main(void)
{
    int   dec = 20, oct = 020, hex = 0x20;

    printf("dec=%d, oct=%d, hex=%d\n", dec, oct, hex);
    printf("dec=%d, oct=%o, hex=%x\n", dec, oct, hex);

    return 0;
}
```

```
dec=20, oct=16, hex=32
dec=20, oct=20, hex=20
```

U1.39

The **char** Data Type

- The **char** data type holds a single character.
  char ch;
- Example assignments:

  char grade, symbol;
  grade = ‘B’;
  symbol = ‘$’;
- The char is held as a one-byte integer in memory.
- The ASCII code is what is actually stored,
  - **so we can use them as characters or integers**, depending on our need.

The **char** Data Type (Contd.)

Use

scanf (“%c”, &ch) ;

to read a single character into the variable ch.

Use

printf(“%c”, ch) ;

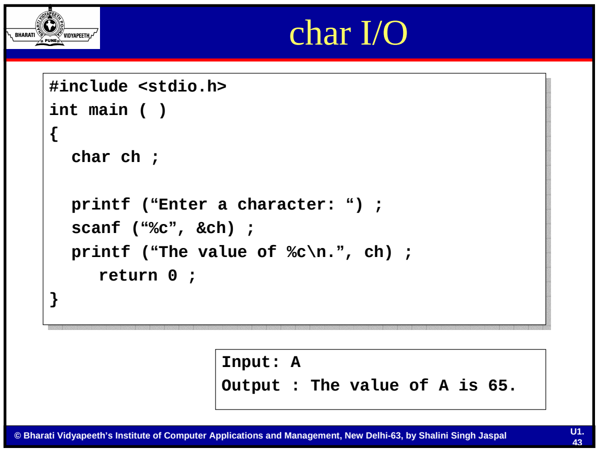
to display the value of a character variable.

Character Example

Note: print integer value of character

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

int main(void)
{
    char  lower_a = 'a';
    char  lower_m = 'm';

    printf("minimum char = %i, ", CHAR_MIN);
    printf("maximum char = %i\n", CHAR_MAX);

    printf("after '%c' comes '%c'\n", lower_a, lower_a + 1);
    printf("uppercase is '%c'\n", lower_m - 'a' + 'A');

    return 0;
}
```

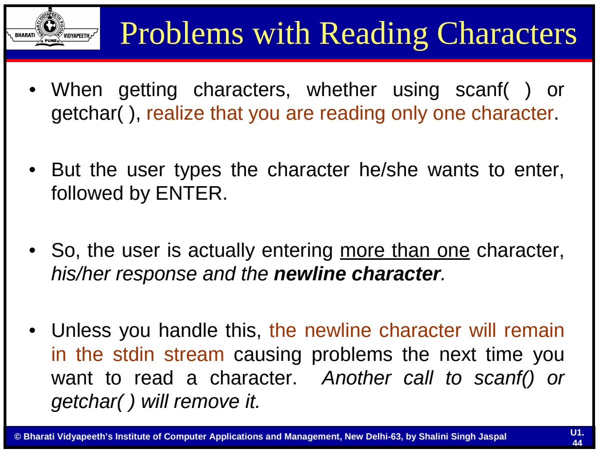
```
minimum char = 0, maximum char = 255
after 'a' comes 'b'
uppercase is 'M'
```



char I/O

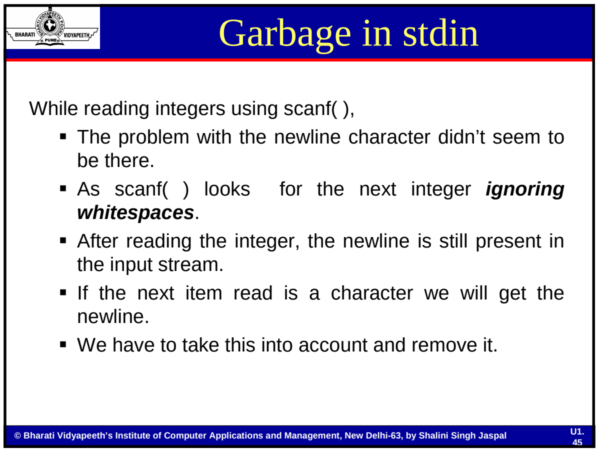
```
#include <stdio.h>
int main ( )
{
  char ch ;

  printf (“Enter a character: “) ;
  scanf (“%c”, &ch) ;
  printf (“The value of %c\n.”, ch) ;
     return 0 ;
}
```

```
Input: A
Output : The value of A is 65.
```



Problems with Reading Characters

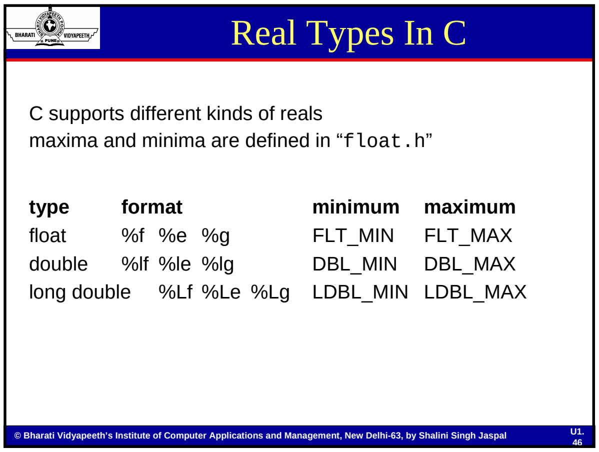
- When getting characters, whether using scanf( ) or getchar( ), realize that you are reading only one character.
- But the user types the character he/she wants to enter, followed by ENTER.
- So, the user is actually entering more than one character, *his/her response and the* ***newline character****.*
- Unless you handle this, the newline character will remain in the stdin stream causing problems the next time you want to read a character. *Another call to scanf() or getchar( ) will remove it.*



Garbage in stdin

While reading integers using scanf( ),

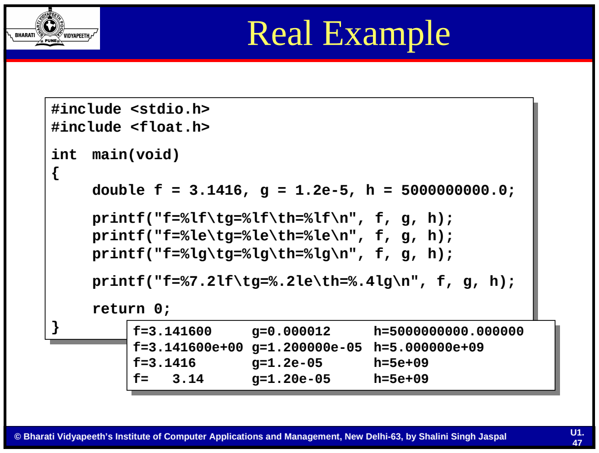
- The problem with the newline character didn’t seem to be there.
- As scanf( ) looks for the next integer ***ignoring whitespaces***.
- After reading the integer, the newline is still present in the input stream.
- If the next item read is a character we will get the newline.
- We have to take this into account and remove it.



Real Types In C

C supports different kinds of reals
maxima and minima are defined in “`float.h`”

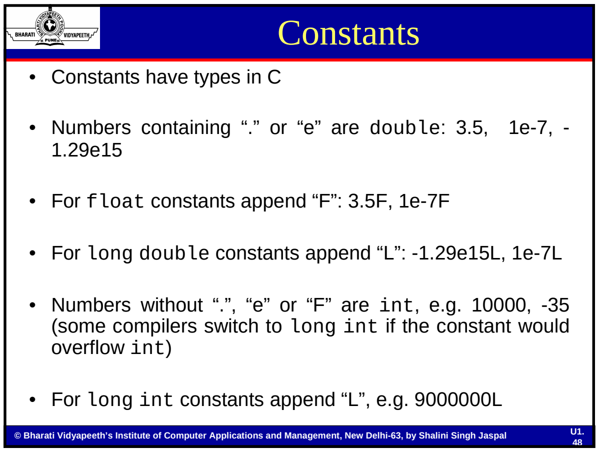
type	format	minimum	maximum
float	%f %e %g	FLT_MIN	FLT_MAX
double	%lf %le %lg	DBL_MIN	DBL_MAX
long double	%Lf %Le %Lg	LDBL_MIN	LDBL_MAX



Real Example

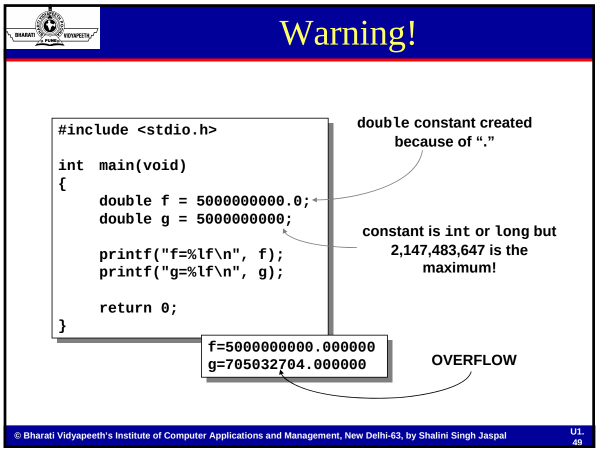
```
#include <stdio.h>
#include <float.h>

int  main(void)
{
     double f = 3.1416, g = 1.2e-5, h = 5000000000.0;

     printf("f=%lf\tg=%lf\th=%lf\n", f, g, h);
     printf("f=%le\tg=%le\th=%le\n", f, g, h);
     printf("f=%lg\tg=%lg\th=%lg\n", f, g, h);

     printf("f=%7.2lf\tg=%.2le\th=%.4lg\n", f, g, h);

     return 0;
}
```

```
f=3.141600      g=0.000012      h=5000000000.000000
f=3.141600e+00  g=1.200000e-05  h=5.000000e+09
f=3.1416        g=1.2e-05       h=5e+09
f=   3.14       g=1.20e-05      h=5e+09
```



Constants

- Constants have types in C
- Numbers containing “.” or “e” are `double`: 3.5, 1e-7, -1.29e15
- For `float` constants append “F”: 3.5F, 1e-7F
- For `long double` constants append “L”: -1.29e15L, 1e-7L
- Numbers without “.”, “e” or “F” are `int`, e.g. 10000, -35 (some compilers switch to `long int` if the constant would overflow `int`)
- For `long int` constants append “L”, e.g. 9000000L



Warning!

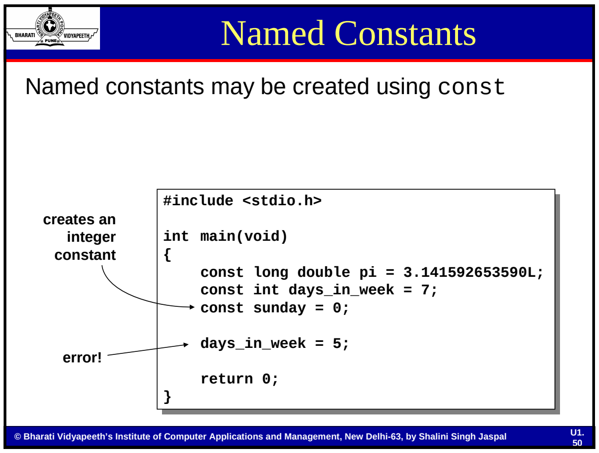
```
#include <stdio.h>

int  main(void)
{
     double f = 5000000000.0;
     double g = 5000000000;

     printf("f=%lf\n", f);
     printf("g=%lf\n", g);

     return 0;
}
```

double constant created because of "."

constant is int or long but 2,147,483,647 is the maximum!

```
f=5000000000.000000
g=705032704.000000
```

OVERFLOW



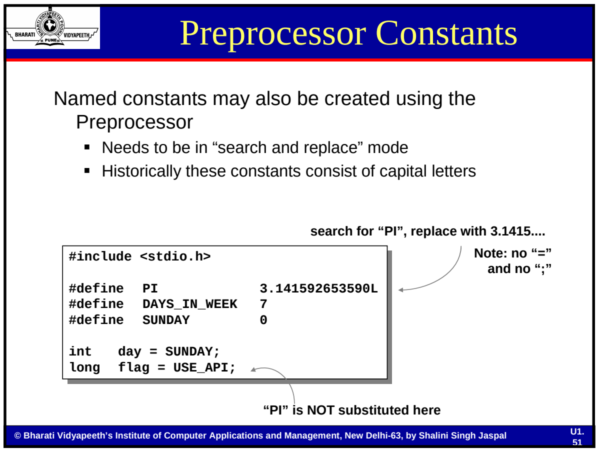
Named Constants

Named constants may be created using `const`

```
#include <stdio.h>

int main(void)
{
     const long double pi = 3.141592653590L;
     const int days_in_week = 7;
     const sunday = 0;

     days_in_week = 5;

     return 0;
}
```

creates an integer constant

error!



Preprocessor Constants

Named constants may also be created using the Preprocessor

- Needs to be in "search and replace" mode
- Historically these constants consist of capital letters

search for "PI", replace with 3.1415....

```
#include <stdio.h>

#define   PI              3.141592653590L
#define   DAYS_IN_WEEK    7
#define   SUNDAY          0

int    day = SUNDAY;
long   flag = USE_API;
```

Note: no "=" and no ";"

"PI" is NOT substituted here

Notes About Variables

- You must not use a variable until you somehow give it a value.
- You can not assume that the variable will have a value before you give it one.
  - Some compilers do, others do not! This is the source of many errors that are difficult to find.

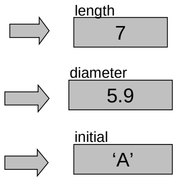
Using Variables: Initialization

Variables may be be given initial values, or **initialized**, when declared. Examples:

int length = 7 ;

float diameter = 5.9 ;

char initial = ‘A’ ;



Using Variables: Initialization (Contd.)

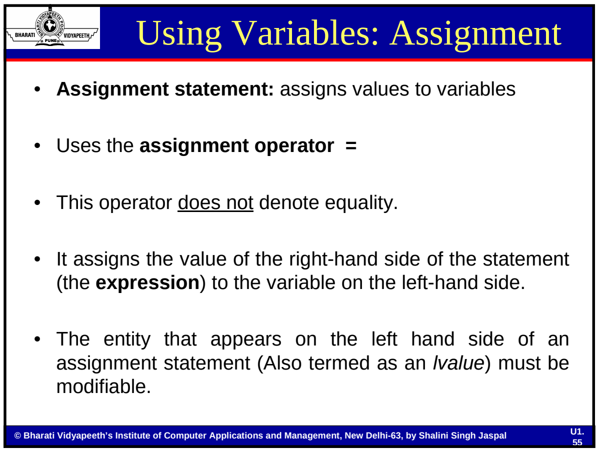
Do not “hide” the initialization

- put initialized variables on a separate line
- a comment is always a good idea

Example:

```
int height ;        /* rectangle height */
int width = 6 ;     /* rectangle width  */
int area ;          /* rectangle area   */
```

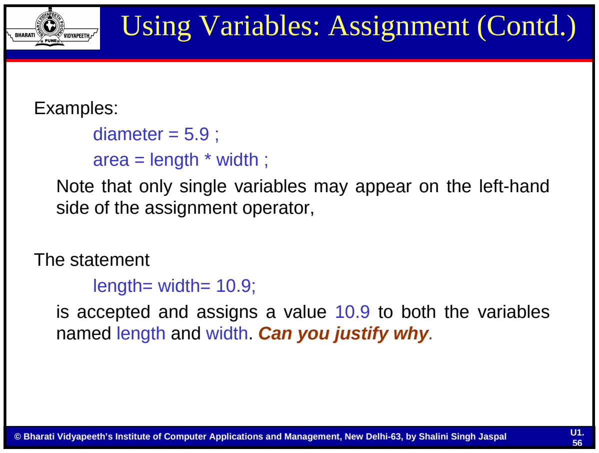
AVOID int height, width = 6, area ;



Using Variables: Assignment

- **Assignment statement:** assigns values to variables
- Uses the **assignment operator =**
- This operator does not denote equality.
- It assigns the value of the right-hand side of the statement (the **expression**) to the variable on the left-hand side.
- The entity that appears on the left hand side of an assignment statement (Also termed as an *lvalue*) must be modifiable.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 55



Using Variables: Assignment (Contd.)

Examples:

```
diameter = 5.9 ;
area = length * width ;
```

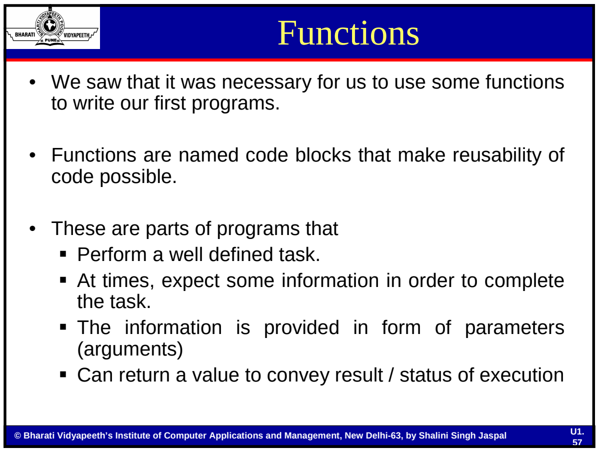
Note that only single variables may appear on the left-hand side of the assignment operator,

The statement

```
length= width= 10.9;
```

is accepted and assigns a value 10.9 to both the variables named length and width. ***Can you justify why.***

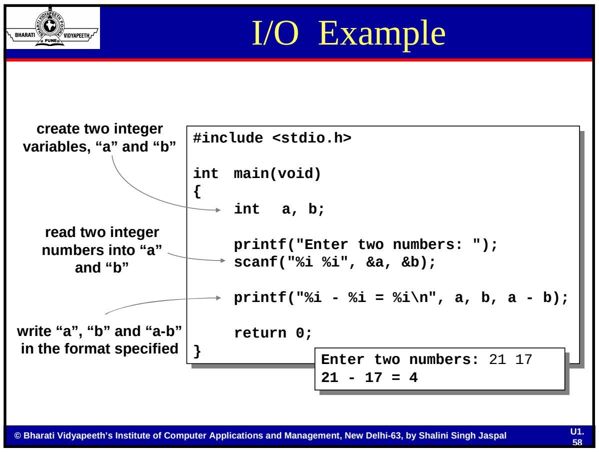
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 56



Functions

- We saw that it was necessary for us to use some functions to write our first programs.
- Functions are named code blocks that make reusability of code possible.
- These are parts of programs that
  - Perform a well defined task.
  - At times, expect some information in order to complete the task.
  - The information is provided in form of parameters (arguments)
  - Can return a value to convey result / status of execution

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 57



I/O Example

create two integer variables, “a” and “b”

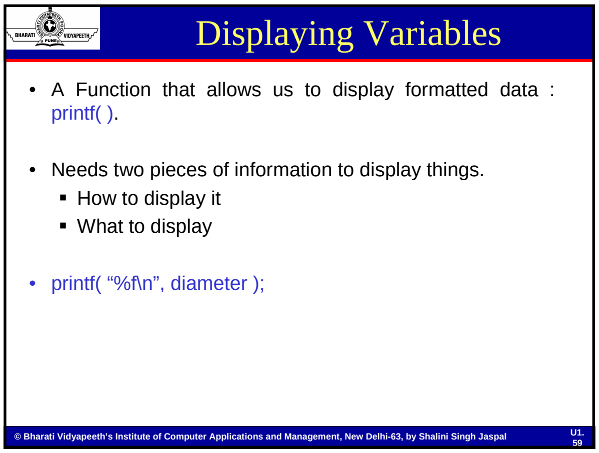
read two integer numbers into “a” and “b”

write “a”, “b” and “a-b” in the format specified

```
#include <stdio.h>

int  main(void)
{
     int   a, b;

     printf("Enter two numbers: ");
     scanf("%i %i", &a, &b);

     printf("%i - %i = %i\n", a, b, a - b);

     return 0;
}
```

```
Enter two numbers: 21 17
21 - 17 = 4
```

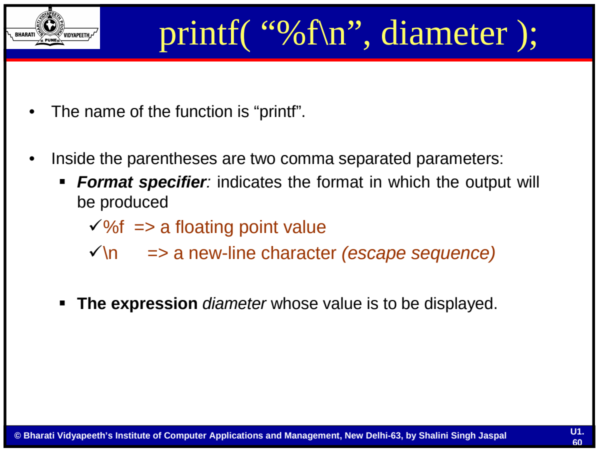
U1.58



Displaying Variables

- A Function that allows us to display formatted data : printf( ).
- Needs two pieces of information to display things.
  - How to display it
  - What to display
- printf(“%f\n”, diameter);

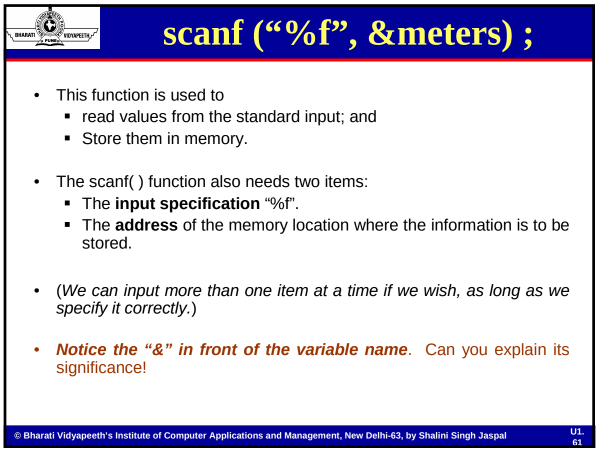
U1.59



printf(“%f\n”, diameter);

- The name of the function is “printf”.
- Inside the parentheses are two comma separated parameters:
  - ***Format specifier**:* indicates the format in which the output will be produced
    - ✓%f => a floating point value
    - ✓\n   => a new-line character *(escape sequence)*
  - **The expression** *diameter* whose value is to be displayed.

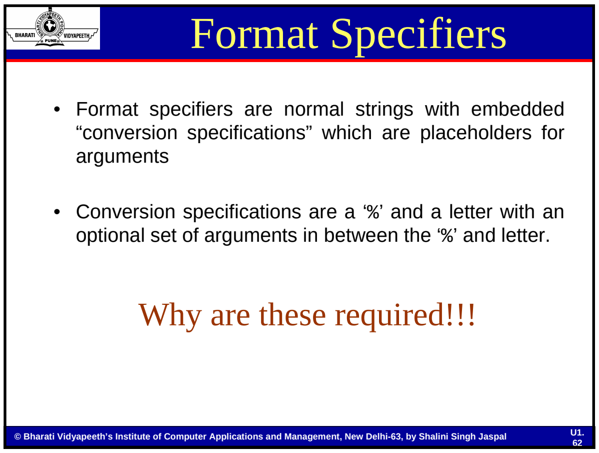
U1.60



scanf (“%f”, &meters) ;

- This function is used to
  - read values from the standard input; and
  - Store them in memory.
- The scanf( ) function also needs two items:
  - The **input specification** “%f”.
  - The **address** of the memory location where the information is to be stored.
- (*We can input more than one item at a time if we wish, as long as we specify it correctly.*)
- ***Notice the “&” in front of the variable name***. Can you explain its significance!

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 61

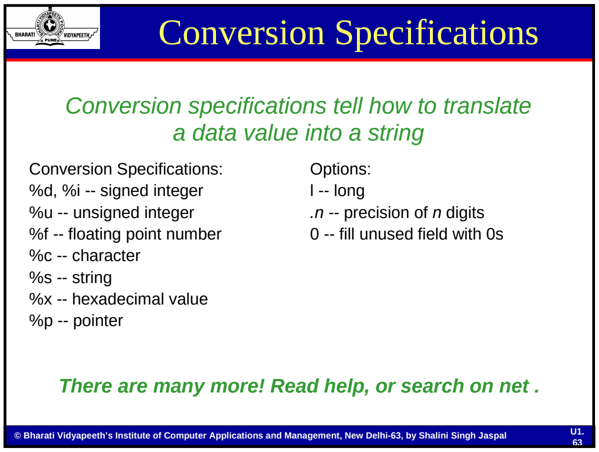


Format Specifiers

- Format specifiers are normal strings with embedded “conversion specifications” which are placeholders for arguments
- Conversion specifications are a ‘%’ and a letter with an optional set of arguments in between the ‘%’ and letter.

Why are these required!!!

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 62



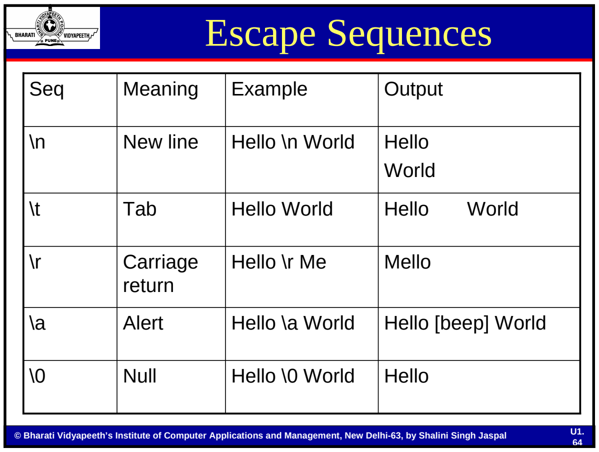
Conversion Specifications

Conversion specifications tell how to translate a data value into a string

Conversion Specifications:	Options:
%d, %i -- signed integer	l -- long
%u -- unsigned integer	.*n* -- precision of *n* digits
%f -- floating point number	0 -- fill unused field with 0s
%c -- character	
%s -- string	
%x -- hexadecimal value	
%p -- pointer	

There are many more! Read help, or search on net .

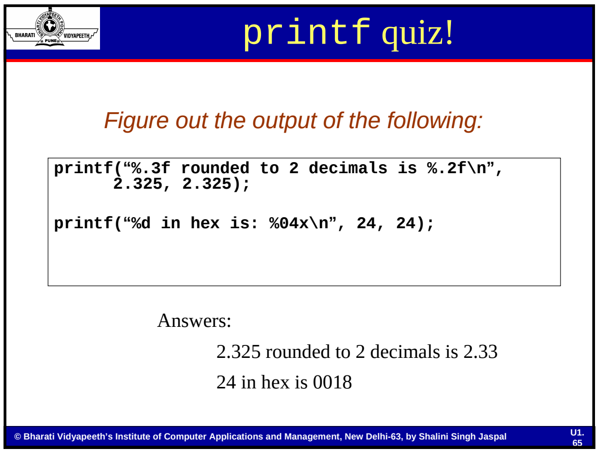
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 63



Escape Sequences

Seq	Meaning	Example	Output
\n	New line	Hello \n World	Hello World
\t	Tab	Hello World	Hello World
\r	Carriage return	Hello \r Me	Mello
\a	Alert	Hello \a World	Hello [beep] World
\0	Null	Hello \0 World	Hello

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 64



printf quiz!

Figure out the output of the following:

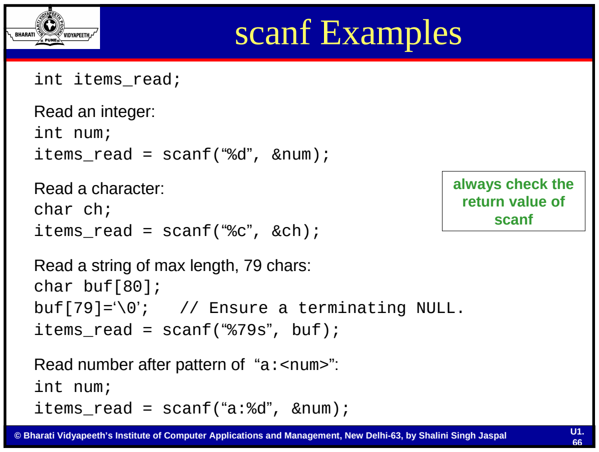
```
printf("%.3f rounded to 2 decimals is %.2f\n",
      2.325, 2.325);

printf("%d in hex is: %04x\n", 24, 24);
```

Answers:

2.325 rounded to 2 decimals is 2.33

24 in hex is 0018

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 65



scanf Examples

```
int items_read;
```

Read an integer:

```
int num;
items_read = scanf("%d", &num);
```

Read a character:

```
char ch;
items_read = scanf("%c", &ch);
```

always check the return value of scanf

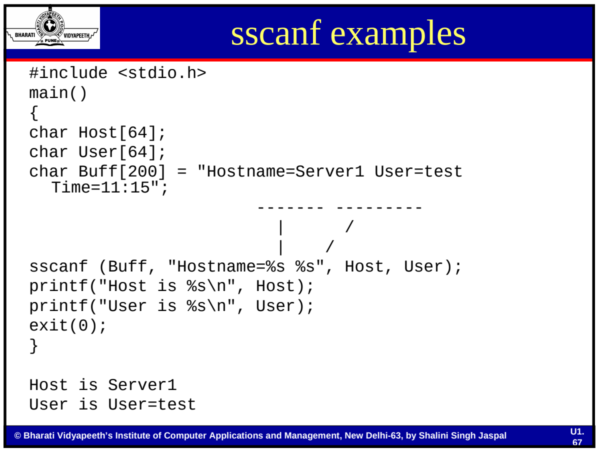
Read a string of max length, 79 chars:

```
char buf[80];
buf[79]='\0';   // Ensure a terminating NULL.
items_read = scanf("%79s", buf);
```

Read number after pattern of "a:<num>":

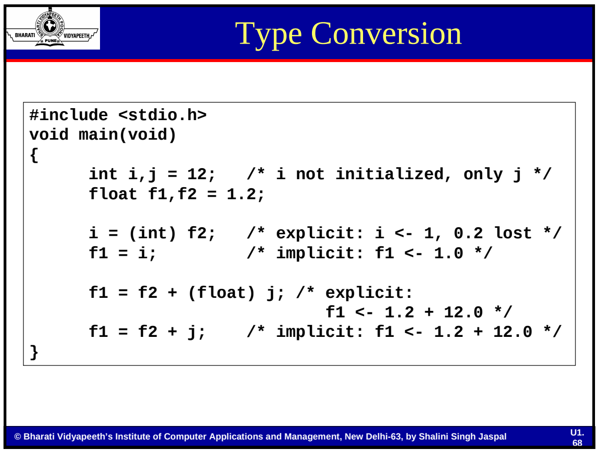
```
int num;
items_read = scanf("a:%d", &num);
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 66



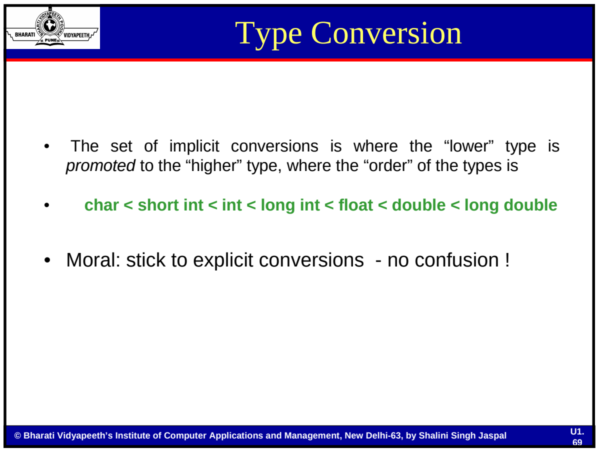
sscanf examples

```
#include <stdio.h>
main()
{
char Host[64];
char User[64];
char Buff[200] = "Hostname=Server1 User=test
  Time=11:15";
                       ------- ---------
                         |      /
                         |    /
sscanf (Buff, "Hostname=%s %s", Host, User);
printf("Host is %s\n", Host);
printf("User is %s\n", User);
exit(0);
}

Host is Server1
User is User=test
```



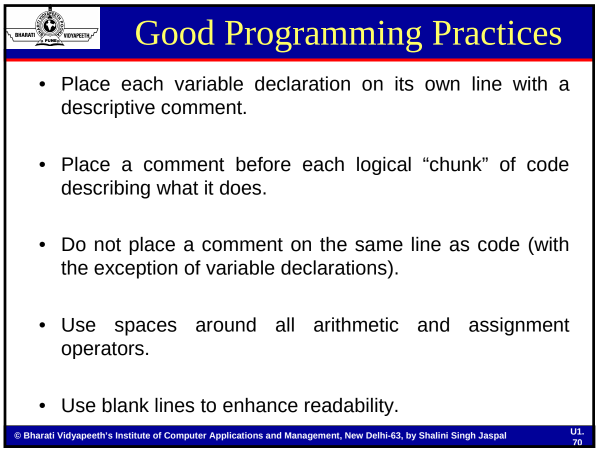
Type Conversion

```
#include <stdio.h>
void main(void)
{
      int i,j = 12;   /* i not initialized, only j */
      float f1,f2 = 1.2;

      i = (int) f2;   /* explicit: i <- 1, 0.2 lost */
      f1 = i;         /* implicit: f1 <- 1.0 */

      f1 = f2 + (float) j; /* explicit:
                              f1 <- 1.2 + 12.0 */
      f1 = f2 + j;    /* implicit: f1 <- 1.2 + 12.0 */
}
```



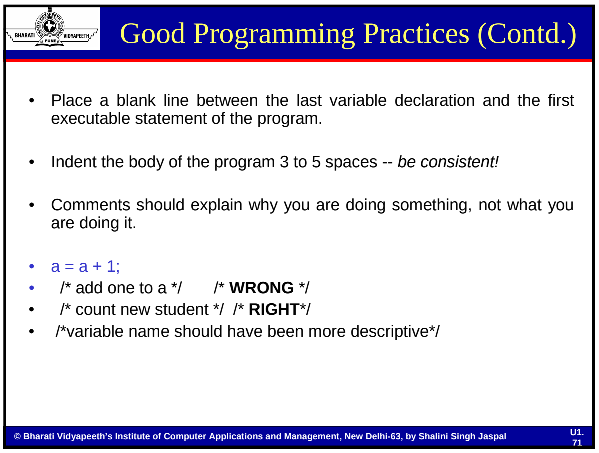
Type Conversion

- The set of implicit conversions is where the “lower” type is *promoted* to the “higher” type, where the “order” of the types is
- **char < short int < int < long int < float < double < long double**
- Moral: stick to explicit conversions - no confusion !



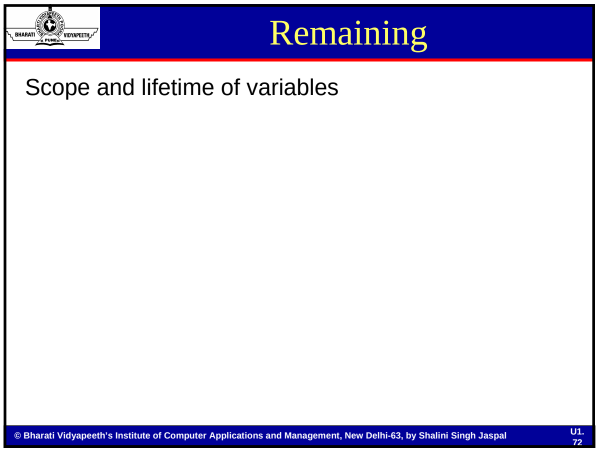
Good Programming Practices

- Place each variable declaration on its own line with a descriptive comment.
- Place a comment before each logical “chunk” of code describing what it does.
- Do not place a comment on the same line as code (with the exception of variable declarations).
- Use spaces around all arithmetic and assignment operators.
- Use blank lines to enhance readability.



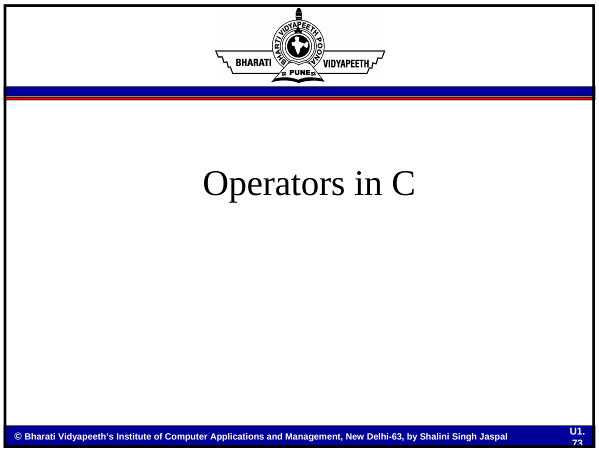
Good Programming Practices (Contd.)

- Place a blank line between the last variable declaration and the first executable statement of the program.
- Indent the body of the program 3 to 5 spaces -- *be consistent!*
- Comments should explain why you are doing something, not what you are doing it.
- a = a + 1;
- /* add one to a */ /* **WRONG** */
- /* count new student */ /* **RIGHT***/
- /*variable name should have been more descriptive*/

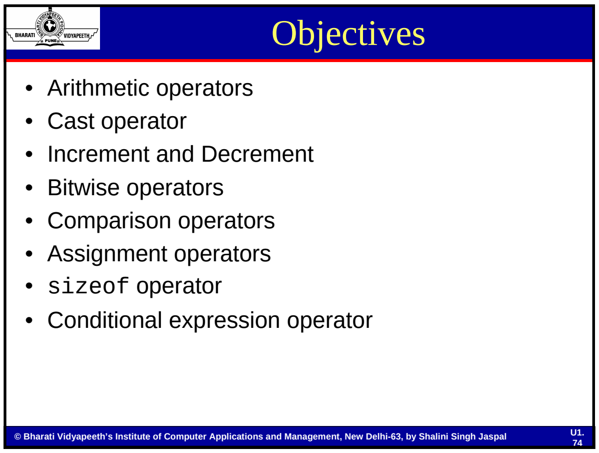


Remaining

Scope and lifetime of variables

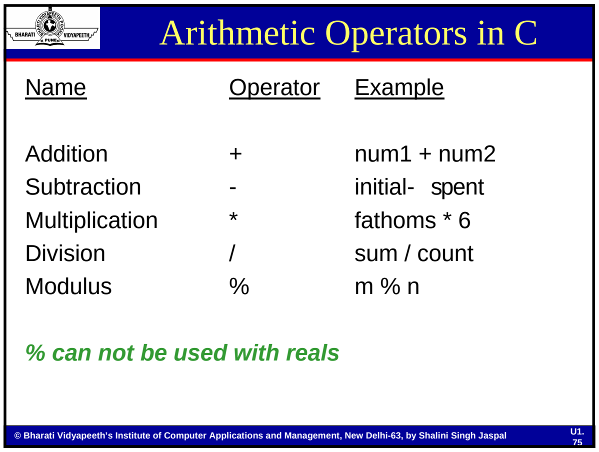


Operators in C



Objectives

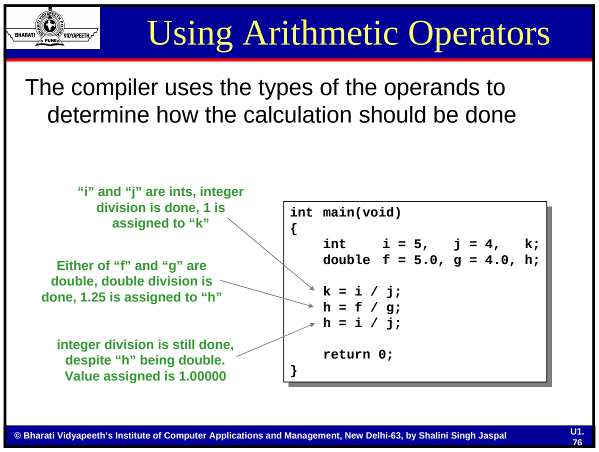
- Arithmetic operators
- Cast operator
- Increment and Decrement
- Bitwise operators
- Comparison operators
- Assignment operators
- `sizeof` operator
- Conditional expression operator



Arithmetic Operators in C

Name	Operator	Example
Addition	+	num1 + num2
Subtraction	-	initial- spent
Multiplication	*	fathoms * 6
Division	/	sum / count
Modulus	%	m % n

% can not be used with reals



Using Arithmetic Operators

The compiler uses the types of the operands to determine how the calculation should be done

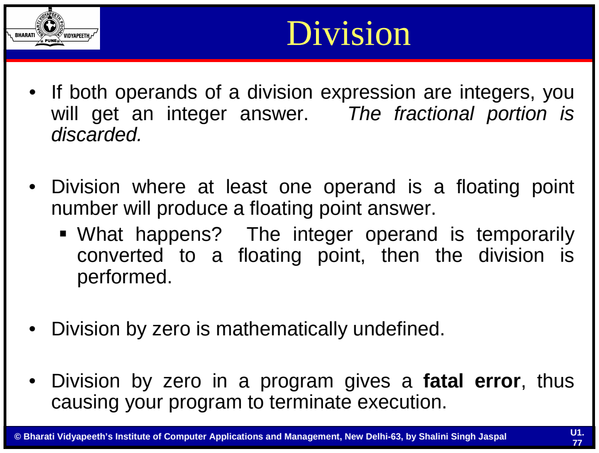
"i" and "j" are ints, integer division is done, 1 is assigned to "k"

Either of "f" and "g" are double, double division is done, 1.25 is assigned to "h"

integer division is still done, despite "h" being double. Value assigned is 1.00000

```
int  main(void)
{
    int    i = 5,   j = 4,   k;
    double  f = 5.0, g = 4.0, h;

    k = i / j;
    h = f / g;
    h = i / j;

    return 0;
}
```

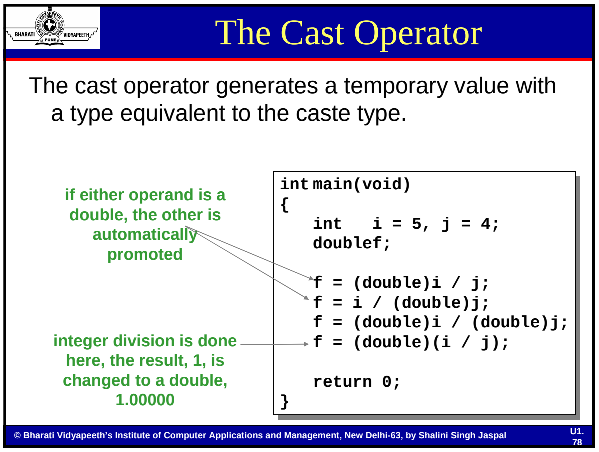
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 76



Division

- If both operands of a division expression are integers, you will get an integer answer. *The fractional portion is discarded.*
- Division where at least one operand is a floating point number will produce a floating point answer.
  - What happens? The integer operand is temporarily converted to a floating point, then the division is performed.
- Division by zero is mathematically undefined.
- Division by zero in a program gives a **fatal error**, thus causing your program to terminate execution.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 77



The Cast Operator

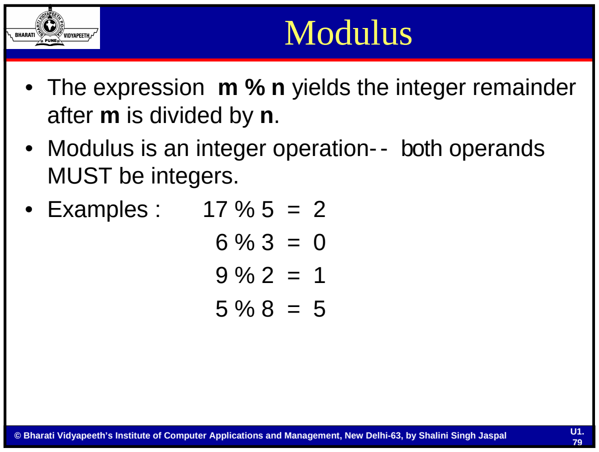
The cast operator generates a temporary value with a type equivalent to the caste type.

if either operand is a double, the other is automatically promoted

integer division is done here, the result, 1, is changed to a double, 1.00000

```
int main(void)
{
    int   i = 5, j = 4;
    doublef;

    f = (double)i / j;
    f = i / (double)j;
    f = (double)i / (double)j;
    f = (double)(i / j);

    return 0;
}
```

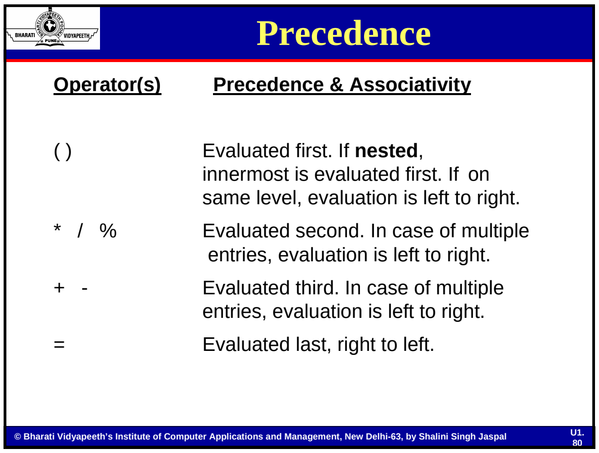
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 78



Modulus

- The expression **m % n** yields the integer remainder after **m** is divided by **n**.
- Modulus is an integer operation-- both operands MUST be integers.
- Examples : 17 % 5 = 2
  6 % 3 = 0
  9 % 2 = 1
  5 % 8 = 5

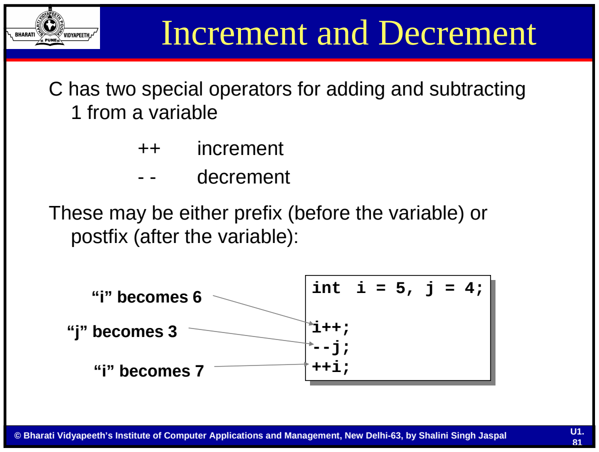
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 79



Precedence

Operator(s)	Precedence & Associativity
( )	Evaluated first. If **nested**, innermost is evaluated first. If on same level, evaluation is left to right.
* / %	Evaluated second. In case of multiple entries, evaluation is left to right.
+ -	Evaluated third. In case of multiple entries, evaluation is left to right.
=	Evaluated last, right to left.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 80



Increment and Decrement

C has two special operators for adding and subtracting 1 from a variable

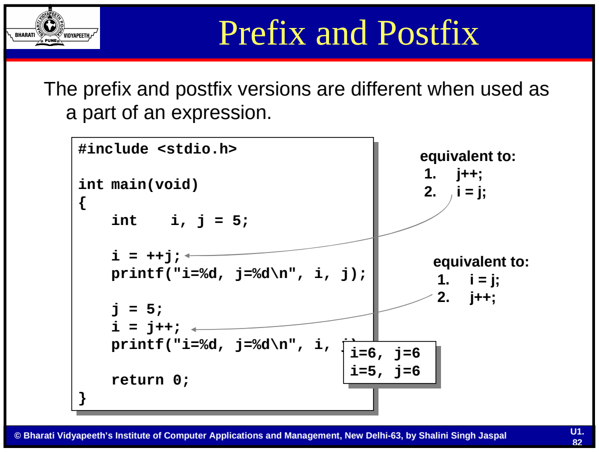
++	increment
- -	decrement

These may be either prefix (before the variable) or postfix (after the variable):

```
int  i = 5, j = 4;

i++;
--j;
++i;
```

"i" becomes 6
"j" becomes 3
"i" becomes 7

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 81



Prefix and Postfix

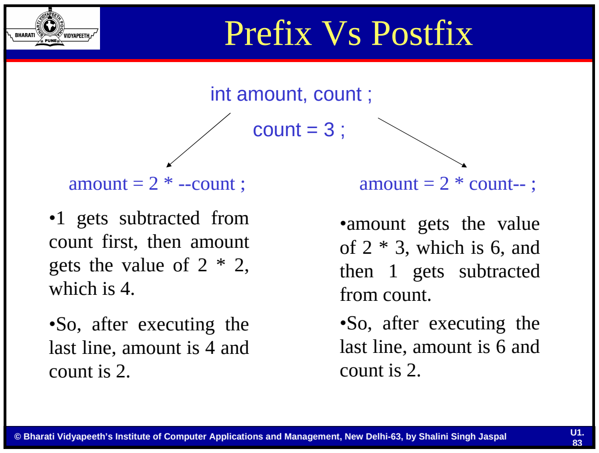
The prefix and postfix versions are different when used as a part of an expression.

```
#include <stdio.h>

int main(void)
{
    int    i, j = 5;

    i = ++j;
    printf("i=%d, j=%d\n", i, j);

    j = 5;
    i = j++;
    printf("i=%d, j=%d\n", i,

    return 0;
}
```

```
i=6, j=6
i=5, j=6
```

equivalent to:
1. j++;
2. i = j;

equivalent to:
1. i = j;
2. j++;



Prefix Vs Postfix

int amount, count ;

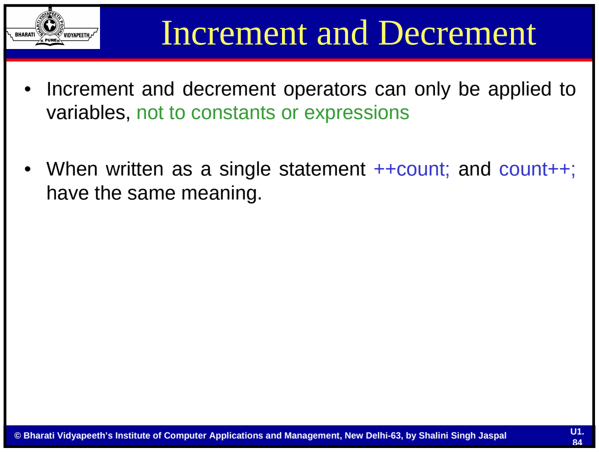
count = 3 ;

amount = 2 * --count ;

- 1 gets subtracted from count first, then amount gets the value of 2 * 2, which is 4.
- So, after executing the last line, amount is 4 and count is 2.

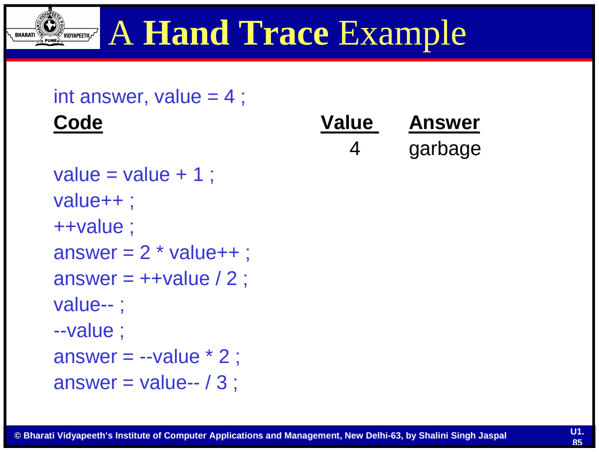
amount = 2 * count-- ;

- amount gets the value of 2 * 3, which is 6, and then 1 gets subtracted from count.
- So, after executing the last line, amount is 6 and count is 2.



Increment and Decrement

- Increment and decrement operators can only be applied to variables, not to constants or expressions
- When written as a single statement ++count; and count++; have the same meaning.

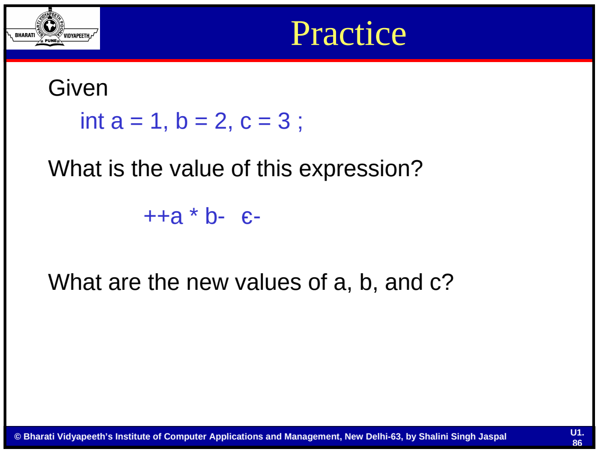


A Hand Trace Example

int answer, value = 4 ;

Code	Value	Answer
	4	garbage
value = value + 1 ;		
value++ ;		
++value ;		
answer = 2 * value++ ;		
answer = ++value / 2 ;		
value-- ;		
--value ;		
answer = --value * 2 ;		
answer = value-- / 3 ;		

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 85



Practice

Given

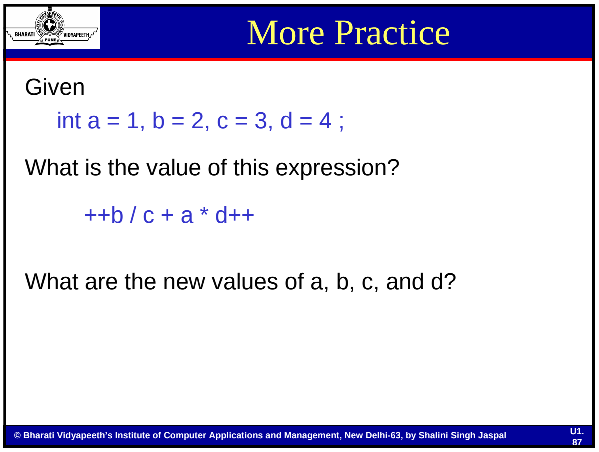
int a = 1, b = 2, c = 3 ;

What is the value of this expression?

++a * b- є-

What are the new values of a, b, and c?

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 86



More Practice

Given

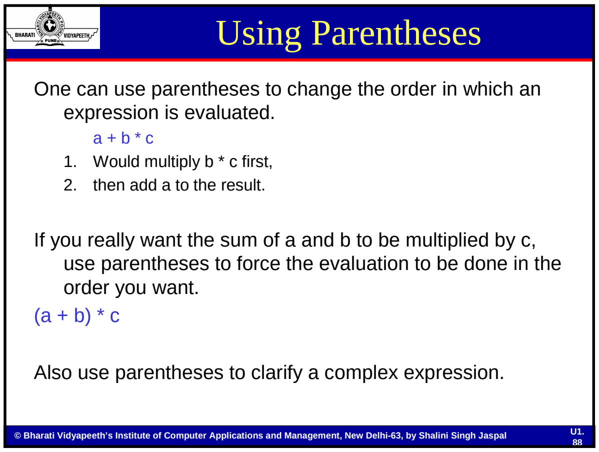
int a = 1, b = 2, c = 3, d = 4 ;

What is the value of this expression?

++b / c + a * d++

What are the new values of a, b, c, and d?

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 87



Using Parentheses

One can use parentheses to change the order in which an expression is evaluated.

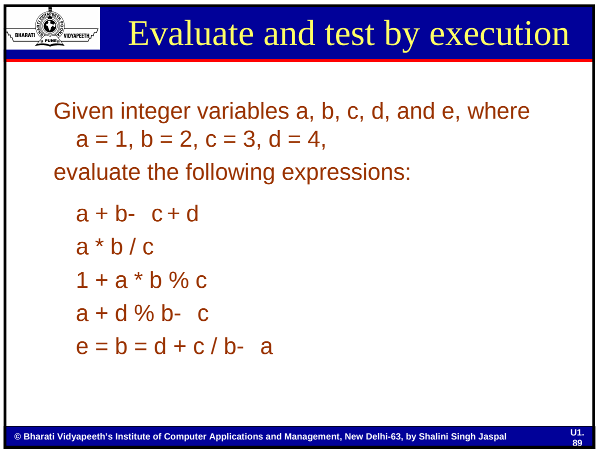
a + b * c

1. Would multiply b * c first,
2. then add a to the result.

If you really want the sum of a and b to be multiplied by c, use parentheses to force the evaluation to be done in the order you want.

(a + b) * c

Also use parentheses to clarify a complex expression.



Evaluate and test by execution

Given integer variables a, b, c, d, and e, where a = 1, b = 2, c = 3, d = 4,

evaluate the following expressions:

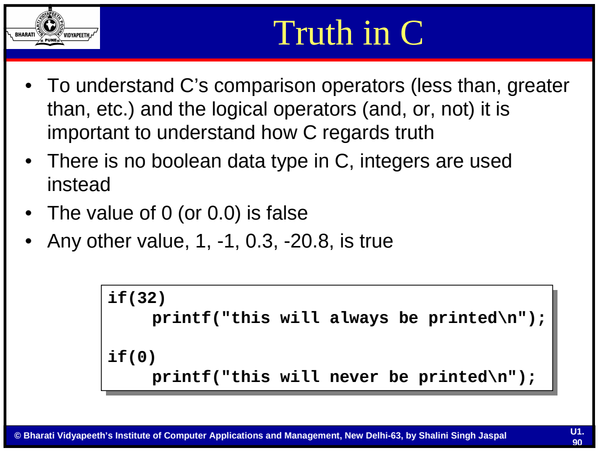
a + b- c + d

a * b / c

1 + a * b % c

a + d % b- c

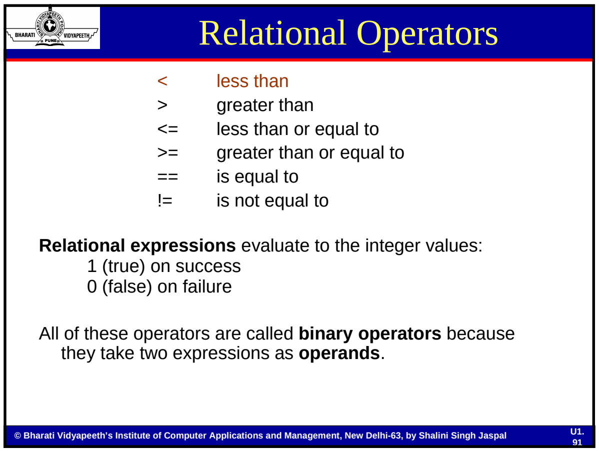
e = b = d + c / b- a



Truth in C

- To understand C’s comparison operators (less than, greater than, etc.) and the logical operators (and, or, not) it is important to understand how C regards truth
- There is no boolean data type in C, integers are used instead
- The value of 0 (or 0.0) is false
- Any other value, 1, -1, 0.3, -20.8, is true

```
if(32)
    printf("this will always be printed\n");

if(0)
    printf("this will never be printed\n");
```



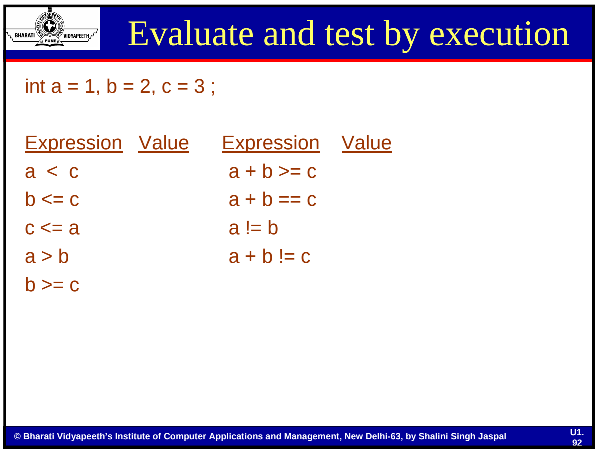
Relational Operators

<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
==	is equal to
!=	is not equal to

Relational expressions evaluate to the integer values:
1 (true) on success
0 (false) on failure

All of these operators are called **binary operators** because they take two expressions as **operands**.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1.91

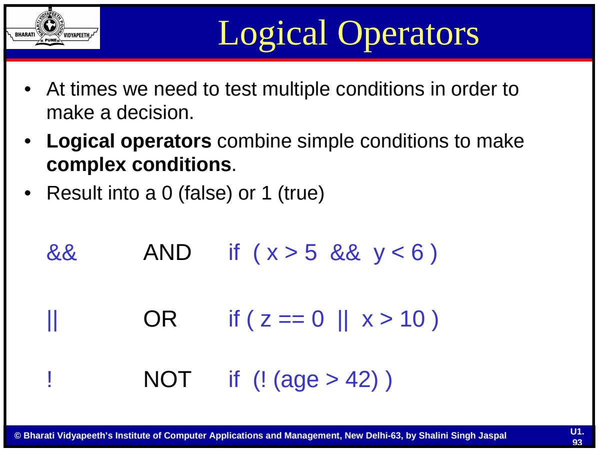


Evaluate and test by execution

int a = 1, b = 2, c = 3 ;

Expression	Value	Expression	Value
a < c		a + b >= c	
b <= c		a + b == c	
c <= a		a != b	
a > b		a + b != c	
b >= c			

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1.92

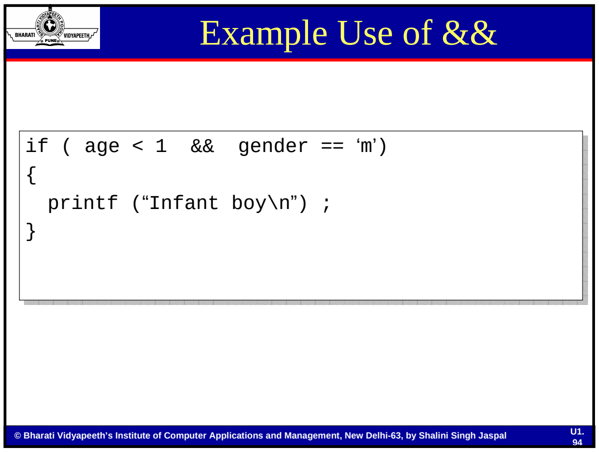


Logical Operators

- At times we need to test multiple conditions in order to make a decision.
- **Logical operators** combine simple conditions to make **complex conditions**.
- Result into a 0 (false) or 1 (true)

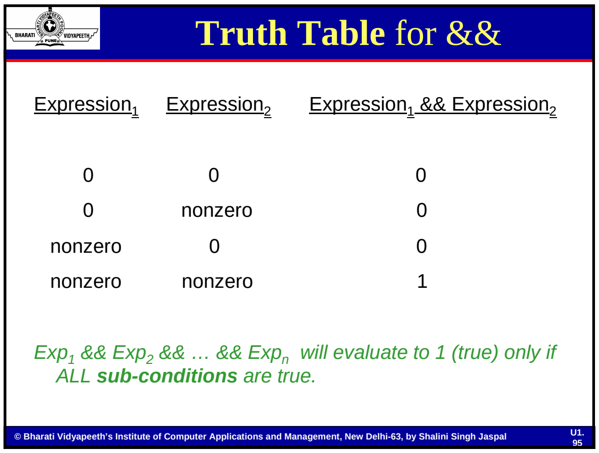
&&	AND	if (x > 5 && y < 6)
||	OR	if (z == 0 || x > 10)
!	NOT	if (! (age > 42))

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1.93



Example Use of &&

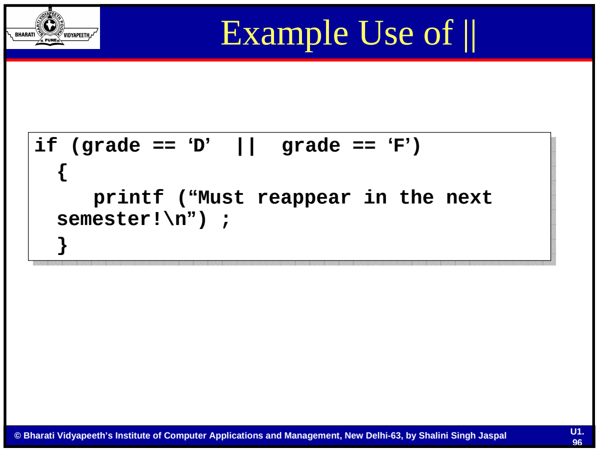
```
if ( age < 1  &&  gender == ‘m’)
{
  printf (“Infant boy\n”) ;
}
```



Truth Table for &&

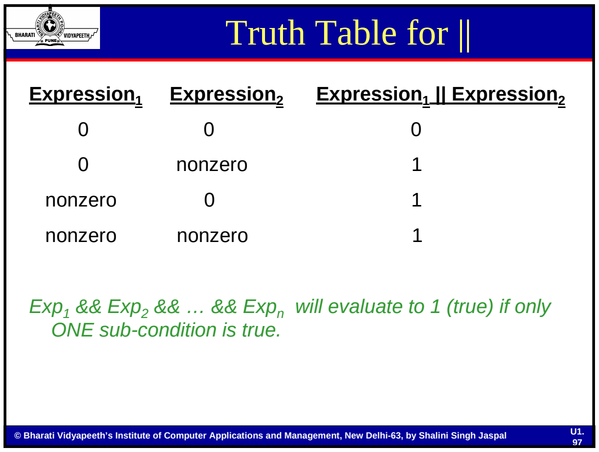
Expression$_1$	Expression$_2$	Expression$_1$ && Expression$_2$
0	0	0
0	nonzero	0
nonzero	0	0
nonzero	nonzero	1

*Exp_1 && Exp_2 && … && Exp_n will evaluate to 1 (true) only if ALL **sub-conditions** are true.*



Example Use of ||

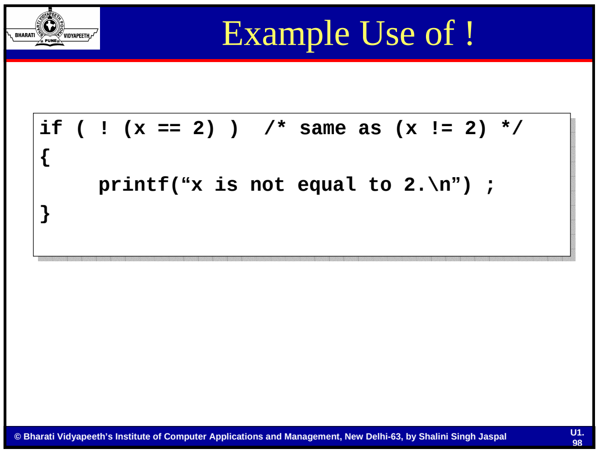
```
if (grade == ‘D’   ||  grade == ‘F’)
  {
     printf (“Must reappear in the next
  semester!\n”) ;
  }
```



Truth Table for ||

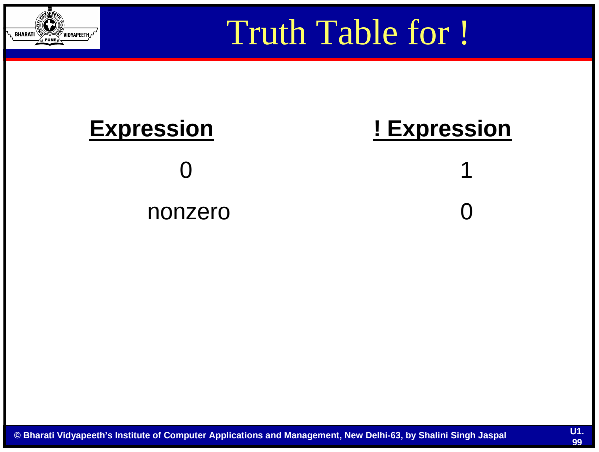
Expression_1	Expression_2	$\text{Expression}_1 \mid\mid \text{Expression}_2$
0	0	0
0	nonzero	1
nonzero	0	1
nonzero	nonzero	1

Exp_1 && Exp_2 && … && Exp_n will evaluate to 1 (true) if only ONE sub-condition is true.



Example Use of !

```
if ( ! (x == 2) )  /* same as (x != 2) */
{
     printf(“x is not equal to 2.\n”) ;
}
```



Truth Table for !

Expression	! Expression
0	1
nonzero	0

Practice

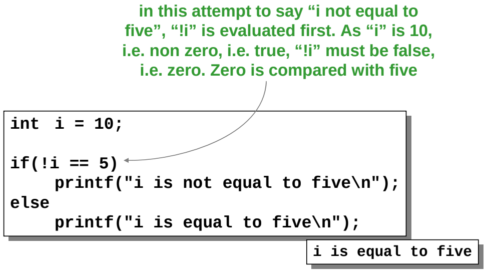
Given
int a = 5, b = 7, c = 17 ;

evaluate each expression as True or False.

1. c / b == 2
2. c % b <= a % b
3. b + c / a != c - a
4. (b < c) && (c == 7)
5. (c + 1-   b == 0) || (b = 5)

Warning!

Remember to use parentheses with conditions, otherwise your program may not mean what you think



in this attempt to say “i not equal to five”, “!i” is evaluated first. As “i” is 10, i.e. non zero, i.e. true, “!i” must be false, i.e. zero. Zero is compared with five

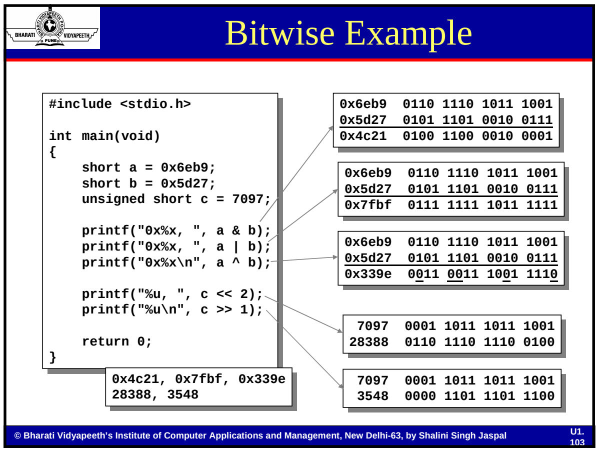
```
int  i = 10;

if(!i == 5)
    printf("i is not equal to five\n");
else
    printf("i is equal to five\n");
```

```
i is equal to five
```

Bitwise Operators

C has the following bit operators which may only be applied to integer types:

&	bitwise and
|	bitwise inclusive or
^	bitwise exclusive or
~	one’s compliment
>>	right shift
<<	left shift



Bitwise Example

```
#include <stdio.h>

int main(void)
{
    short a = 0x6eb9;
    short b = 0x5d27;
    unsigned short c = 7097;

    printf("0x%x, ", a & b);
    printf("0x%x, ", a | b);
    printf("0x%x\n", a ^ b);

    printf("%u, ", c << 2);
    printf("%u\n", c >> 1);

    return 0;
}
```

```
0x4c21, 0x7fbf, 0x339e
28388, 3548
```

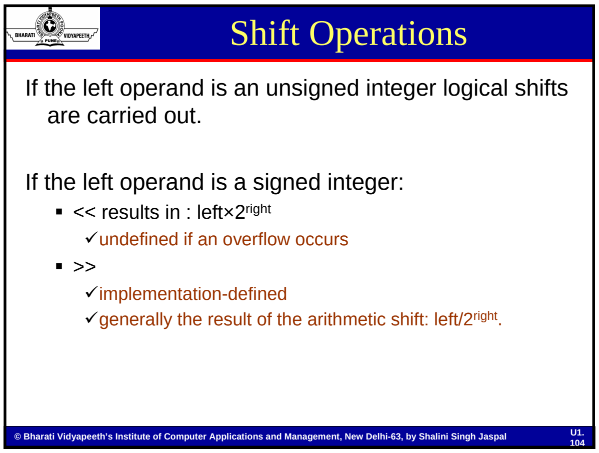
```
0x6eb9  0110 1110 1011 1001
0x5d27  0101 1101 0010 0111
0x4c21  0100 1100 0010 0001
```

```
0x6eb9  0110 1110 1011 1001
0x5d27  0101 1101 0010 0111
0x7fbf  0111 1111 1011 1111
```

```
0x6eb9  0110 1110 1011 1001
0x5d27  0101 1101 0010 0111
0x339e  0011 0011 1001 1110
```

```
 7097  0001 1011 1011 1001
28388  0110 1110 1110 0100
```

```
 7097  0001 1011 1011 1001
 3548  0000 1101 1101 1100
```

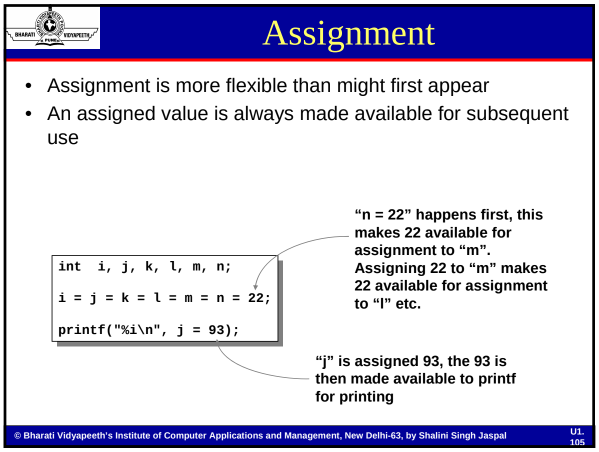


Shift Operations

If the left operand is an unsigned integer logical shifts are carried out.

If the left operand is a signed integer:

- << results in : left×2^{right}
  - ✓undefined if an overflow occurs
- >>
  - ✓implementation-defined
  - ✓generally the result of the arithmetic shift: left/2^{right}.



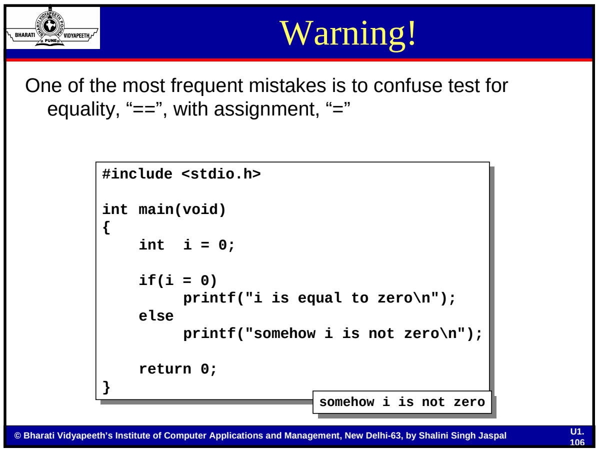
Assignment

- Assignment is more flexible than might first appear
- An assigned value is always made available for subsequent use

```
int  i, j, k, l, m, n;

i = j = k = l = m = n = 22;

printf("%i\n", j = 93);
```

"n = 22" happens first, this makes 22 available for assignment to "m". Assigning 22 to "m" makes 22 available for assignment to "l" etc.

"j" is assigned 93, the 93 is then made available to printf for printing

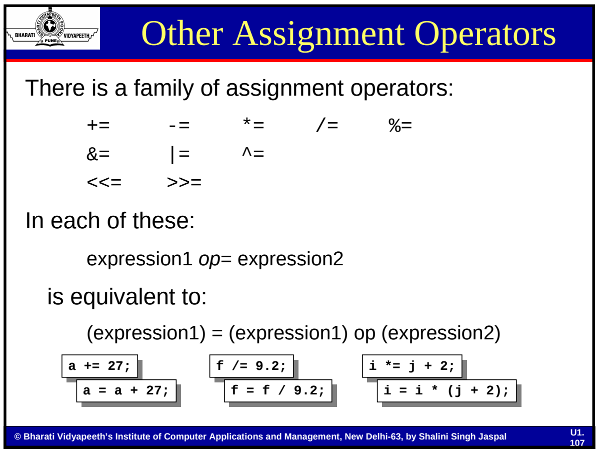


Warning!

One of the most frequent mistakes is to confuse test for equality, “==”, with assignment, “=”

```
#include <stdio.h>

int main(void)
{
    int  i = 0;

    if(i = 0)
         printf("i is equal to zero\n");
    else
         printf("somehow i is not zero\n");

    return 0;
}
```

```
somehow i is not zero
```



Other Assignment Operators

There is a family of assignment operators:

```
+=     -=     *=     /=     %=
&=     |=     ^=
<<=    >>=
```

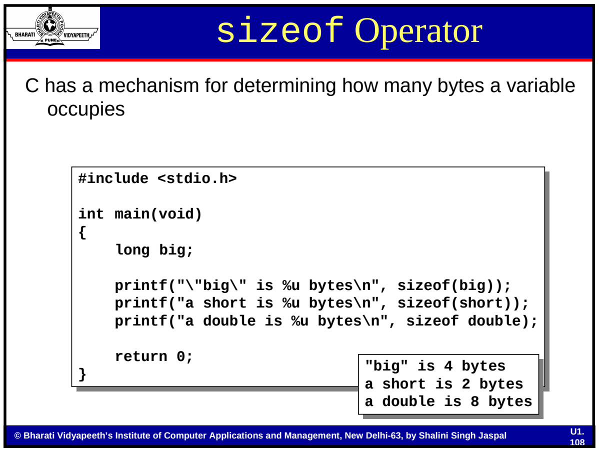
In each of these:

expression1 *op*= expression2

is equivalent to:

(expression1) = (expression1) op (expression2)

```
a += 27;          f /= 9.2;          i *= j + 2;
  a = a + 27;       f = f / 9.2;       i = i * (j + 2);
```

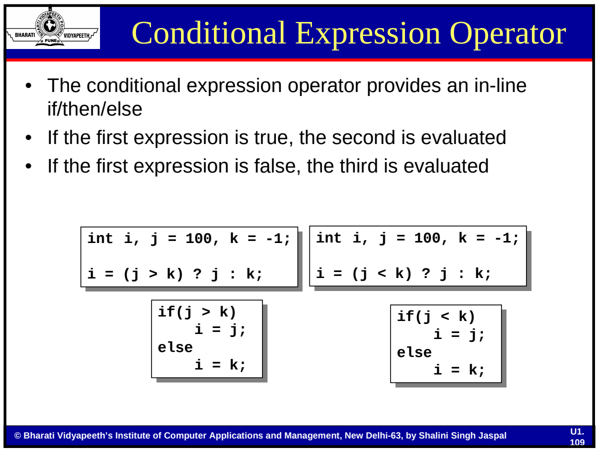


`sizeof` Operator

C has a mechanism for determining how many bytes a variable occupies

```
#include <stdio.h>

int main(void)
{
    long big;

    printf("\"big\" is %u bytes\n", sizeof(big));
    printf("a short is %u bytes\n", sizeof(short));
    printf("a double is %u bytes\n", sizeof double);

    return 0;
}
```

```
"big" is 4 bytes
a short is 2 bytes
a double is 8 bytes
```



Conditional Expression Operator

- The conditional expression operator provides an in-line if/then/else
- If the first expression is true, the second is evaluated
- If the first expression is false, the third is evaluated

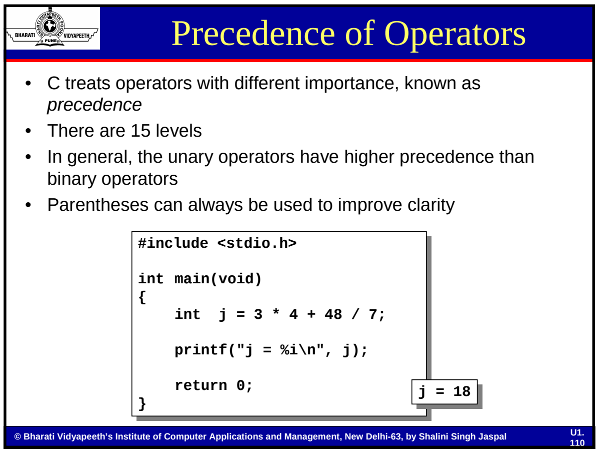
```
int i, j = 100, k = -1;

i = (j > k) ? j : k;
```

```
int i, j = 100, k = -1;

i = (j < k) ? j : k;
```

```
if(j > k)
    i = j;
else
    i = k;
```

```
if(j < k)
    i = j;
else
    i = k;
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 109



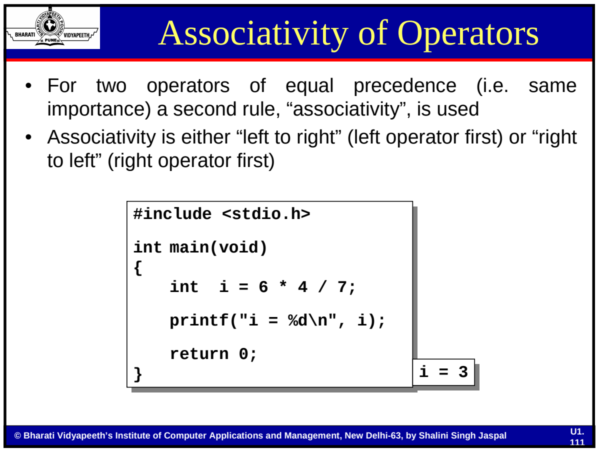
Precedence of Operators

- C treats operators with different importance, known as *precedence*
- There are 15 levels
- In general, the unary operators have higher precedence than binary operators
- Parentheses can always be used to improve clarity

```
#include <stdio.h>

int main(void)
{
    int  j = 3 * 4 + 48 / 7;

    printf("j = %i\n", j);

    return 0;
}
```

```
j = 18
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 110



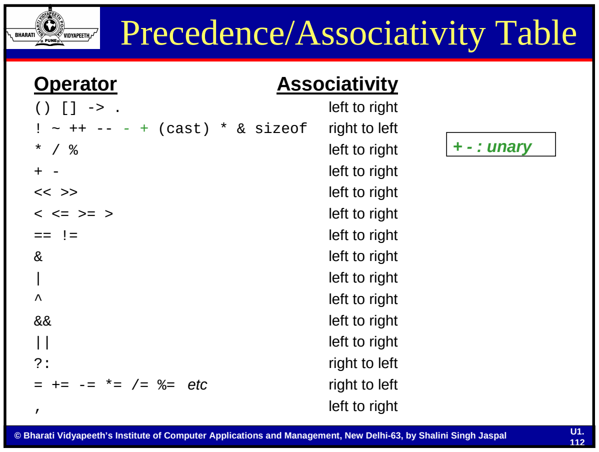
Associativity of Operators

- For two operators of equal precedence (i.e. same importance) a second rule, “associativity”, is used
- Associativity is either “left to right” (left operator first) or “right to left” (right operator first)

```
#include <stdio.h>

int main(void)
{
    int  i = 6 * 4 / 7;

    printf("i = %d\n", i);

    return 0;
}
```

```
i = 3
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 111

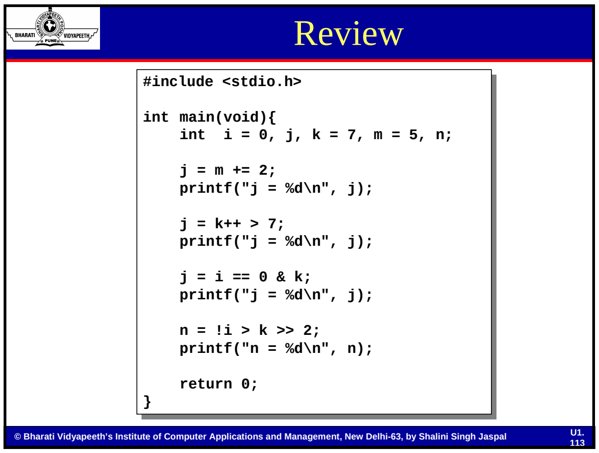


Precedence/Associativity Table

Operator	Associativity
`() [] -> .`	left to right
`! ~ ++ -- - + (cast) * & sizeof`	right to left
`* / %`	left to right
`+ -`	left to right
`<< >>`	left to right
`< <= >= >`	left to right
`== !=`	left to right
`&`	left to right
`|`	left to right
`^`	left to right
`&&`	left to right
`||`	left to right
`?:`	right to left
`= += -= *= /= %=` *etc*	right to left
`,`	left to right

+ - : unary

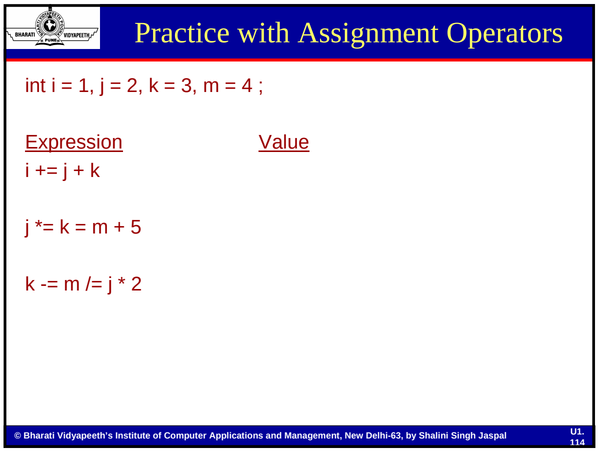
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 112



Review

```
#include <stdio.h>

int main(void){
    int  i = 0, j, k = 7, m = 5, n;

    j = m += 2;
    printf("j = %d\n", j);

    j = k++ > 7;
    printf("j = %d\n", j);

    j = i == 0 & k;
    printf("j = %d\n", j);

    n = !i > k >> 2;
    printf("n = %d\n", n);

    return 0;
}
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 113

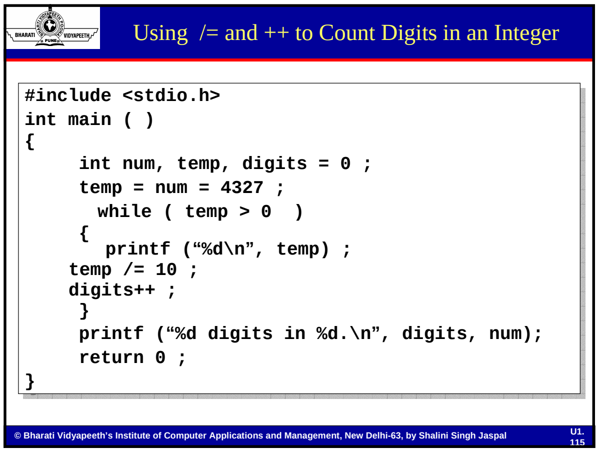


Practice with Assignment Operators

int i = 1, j = 2, k = 3, m = 4 ;

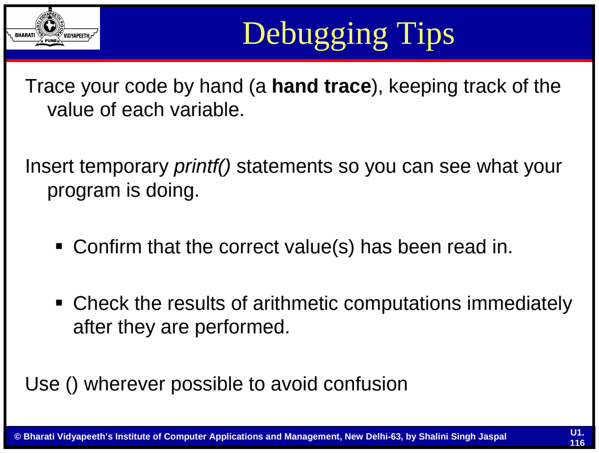
Expression	Value
i += j + k	
j *= k = m + 5	
k -= m /= j * 2	

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 114



Using /= and ++ to Count Digits in an Integer

```
#include <stdio.h>
int main ( )
{
     int num, temp, digits = 0 ;
     temp = num = 4327 ;
       while ( temp > 0  )
     {
        printf (“%d\n”, temp) ;
    temp /= 10 ;
    digits++ ;
     }
     printf (“%d digits in %d.\n”, digits, num);
     return 0 ;
}
```



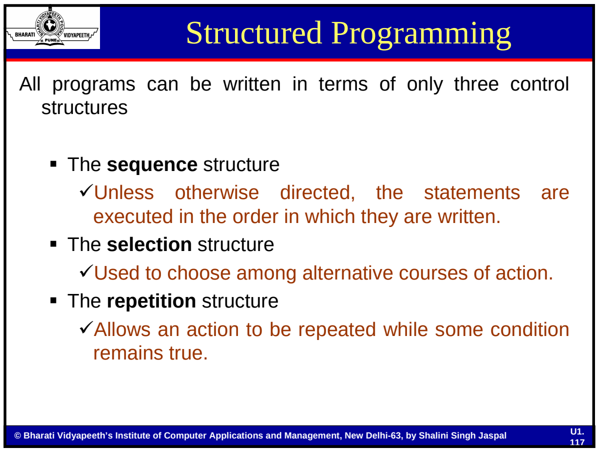
Debugging Tips

Trace your code by hand (a **hand trace**), keeping track of the value of each variable.

Insert temporary *printf()* statements so you can see what your program is doing.

- Confirm that the correct value(s) has been read in.
- Check the results of arithmetic computations immediately after they are performed.

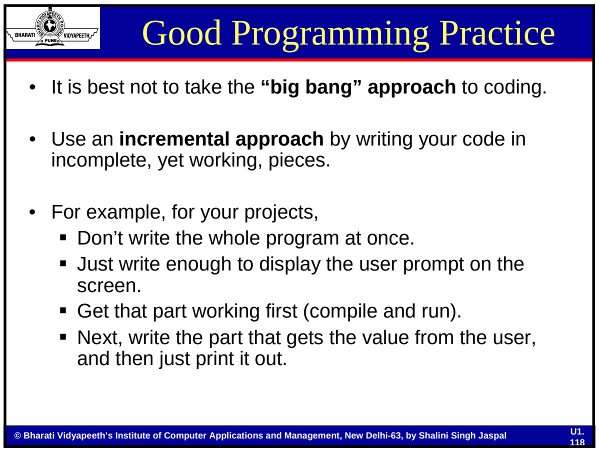
Use () wherever possible to avoid confusion



Structured Programming

All programs can be written in terms of only three control structures

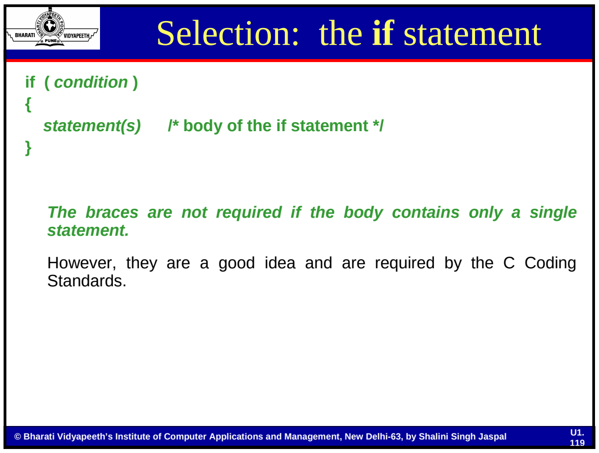
- The **sequence** structure
  - ✓Unless otherwise directed, the statements are executed in the order in which they are written.
- The **selection** structure
  - ✓Used to choose among alternative courses of action.
- The **repetition** structure
  - ✓Allows an action to be repeated while some condition remains true.



Good Programming Practice

- It is best not to take the **“big bang” approach** to coding.
- Use an **incremental approach** by writing your code in incomplete, yet working, pieces.
- For example, for your projects,
  - Don’t write the whole program at once.
  - Just write enough to display the user prompt on the screen.
  - Get that part working first (compile and run).
  - Next, write the part that gets the value from the user, and then just print it out.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 118



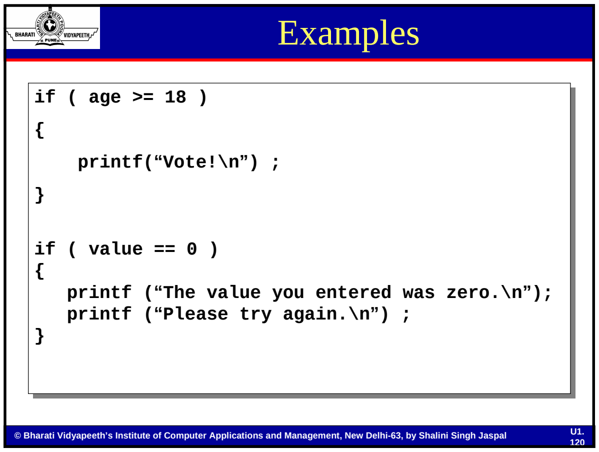
Selection: the **if** statement

```
if ( condition )
{
   statement(s)      /* body of the if statement */
}
```

The braces are not required if the body contains only a single statement.

However, they are a good idea and are required by the C Coding Standards.

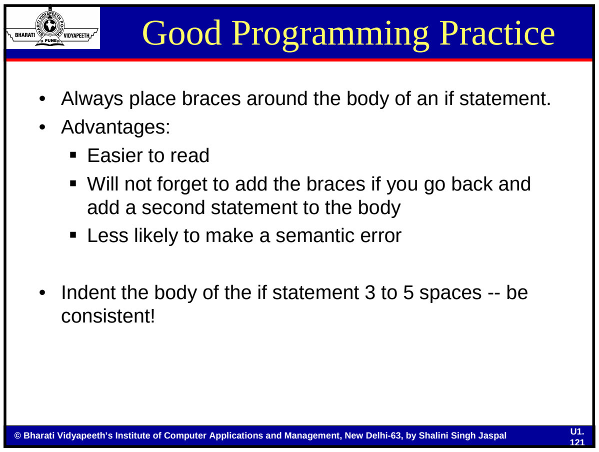
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 119



Examples

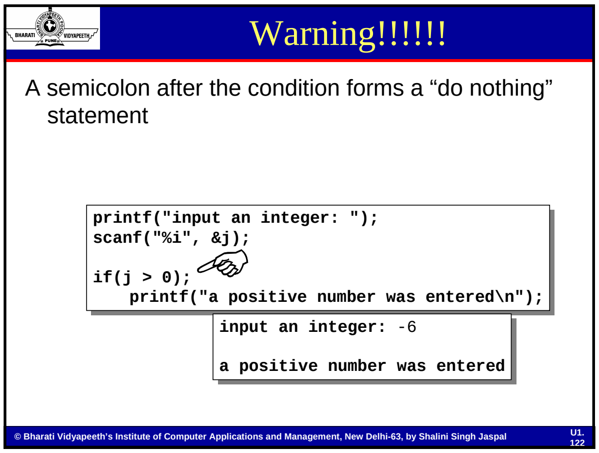
```
if ( age >= 18 )
{
    printf(“Vote!\n”) ;
}

if ( value == 0 )
{
   printf (“The value you entered was zero.\n”);
   printf (“Please try again.\n”) ;
}
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 120



Good Programming Practice

- Always place braces around the body of an if statement.
- Advantages:
  - Easier to read
  - Will not forget to add the braces if you go back and add a second statement to the body
  - Less likely to make a semantic error
- Indent the body of the if statement 3 to 5 spaces -- be consistent!

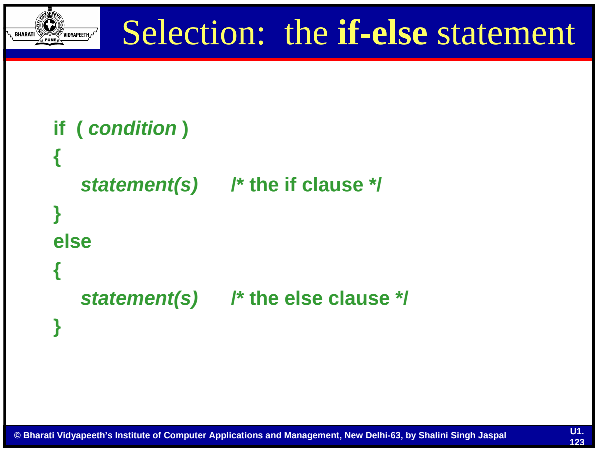


Warning!!!!!!

A semicolon after the condition forms a “do nothing” statement

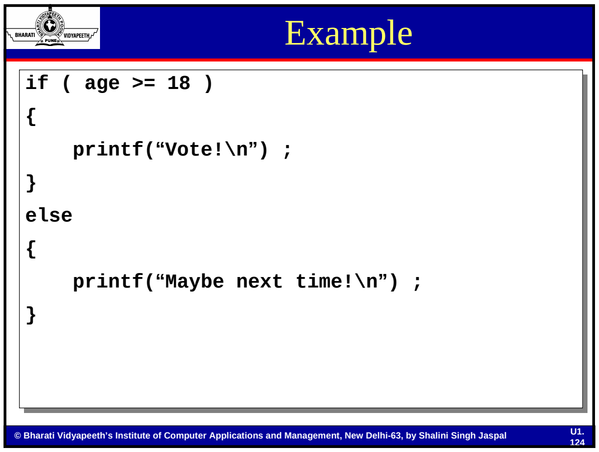
```
printf("input an integer: ");
scanf("%i", &j);

if(j > 0);
    printf("a positive number was entered\n");
```

```
input an integer: -6

a positive number was entered
```



Selection: the **if-else** statement

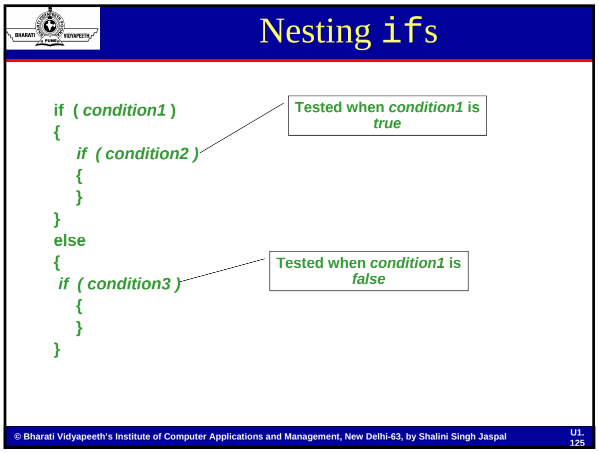
```
if  ( condition )
{
    statement(s)     /* the if clause */
}
else
{
    statement(s)     /* the else clause */
}
```



Example

```
if ( age >= 18 )
{
    printf(“Vote!\n”) ;
}
else
{
    printf(“Maybe next time!\n”) ;
}
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1.124



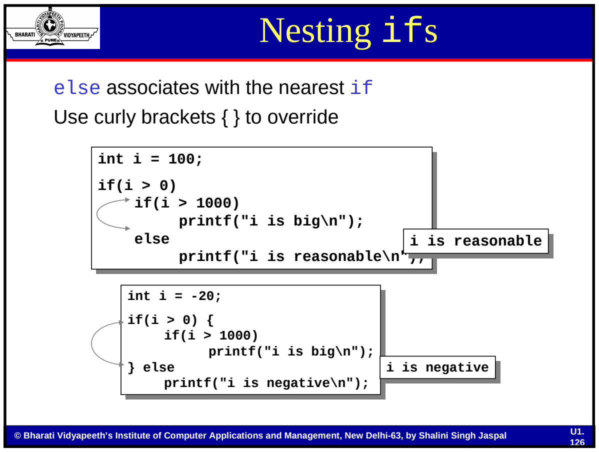
Nesting ifs

```
if ( condition1 )
{
   if ( condition2 )
   {
   }
}
else
{
 if ( condition3 )
   {
   }
}
```

Tested when *condition1* is *true*

Tested when *condition1* is *false*

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1.125



Nesting ifs

else associates with the nearest if

Use curly brackets { } to override

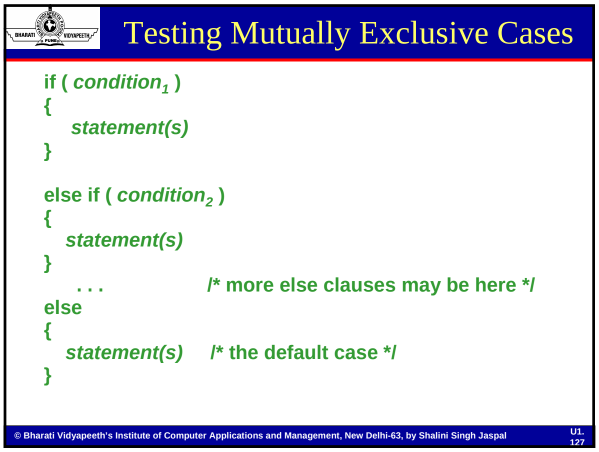
```
int i = 100;

if(i > 0)
    if(i > 1000)
         printf("i is big\n");
    else
         printf("i is reasonable\n");
```

```
i is reasonable
```

```
int i = -20;

if(i > 0) {
    if(i > 1000)
         printf("i is big\n");
} else
    printf("i is negative\n");
```

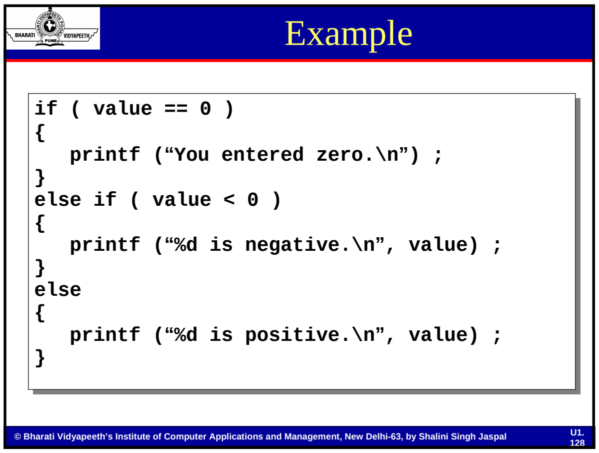
```
i is negative
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1.126



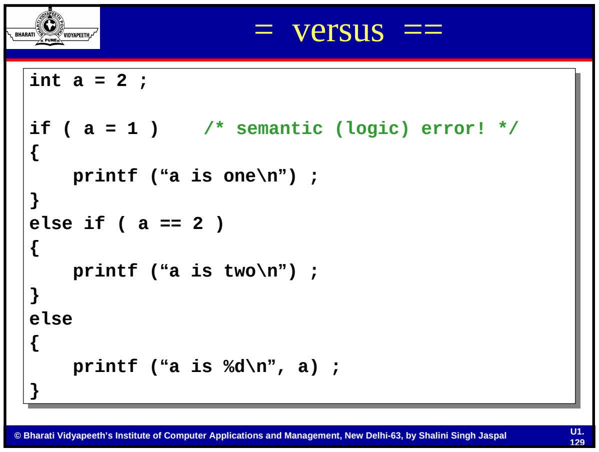
Testing Mutually Exclusive Cases

```
if ( condition_1 )
{
    statement(s)
}

else if ( condition_2 )
{
   statement(s)
}
    . . .              /* more else clauses may be here */
else
{
   statement(s)     /* the default case */
}
```



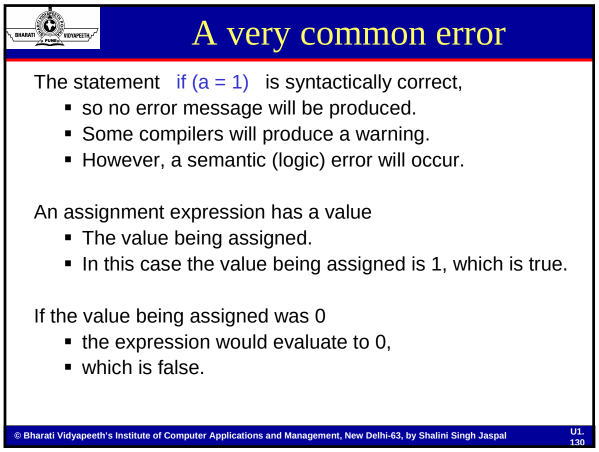
Example

```
if ( value == 0 )
{
   printf (“You entered zero.\n”) ;
}
else if ( value < 0 )
{
   printf (“%d is negative.\n”, value) ;
}
else
{
   printf (“%d is positive.\n”, value) ;
}
```



= versus ==

```
int a = 2 ;

if ( a = 1 )    /* semantic (logic) error! */
{
    printf (“a is one\n”) ;
}
else if ( a == 2 )
{
    printf (“a is two\n”) ;
}
else
{
    printf (“a is %d\n”, a) ;
}
```



A very common error

The statement if (a = 1) is syntactically correct,

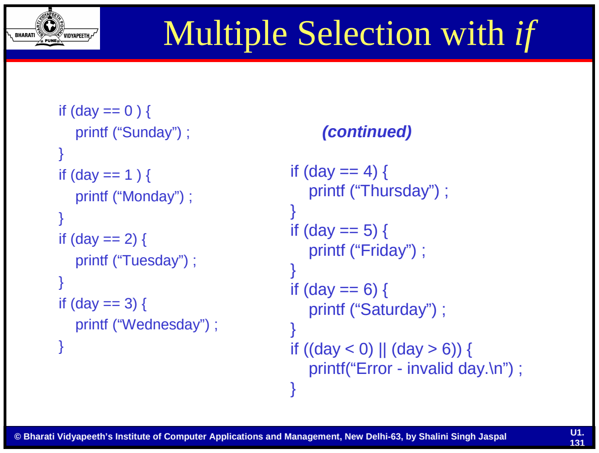
- so no error message will be produced.
- Some compilers will produce a warning.
- However, a semantic (logic) error will occur.

An assignment expression has a value

- The value being assigned.
- In this case the value being assigned is 1, which is true.

If the value being assigned was 0

- the expression would evaluate to 0,
- which is false.

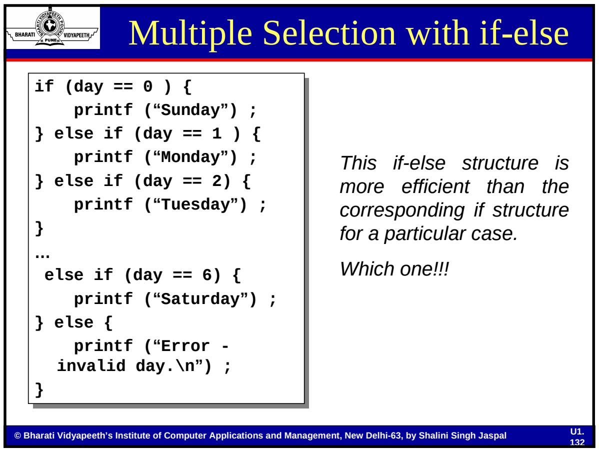


Multiple Selection with *if*

```
if (day == 0 ) {
    printf (“Sunday”) ;
}
if (day == 1 ) {
    printf (“Monday”) ;
}
if (day == 2) {
    printf (“Tuesday”) ;
}
if (day == 3) {
    printf (“Wednesday”) ;
}
```

(continued)

```
if (day == 4) {
    printf (“Thursday”) ;
}
if (day == 5) {
    printf (“Friday”) ;
}
if (day == 6) {
    printf (“Saturday”) ;
}
if ((day < 0) || (day > 6)) {
    printf(“Error - invalid day.\n”) ;
}
```

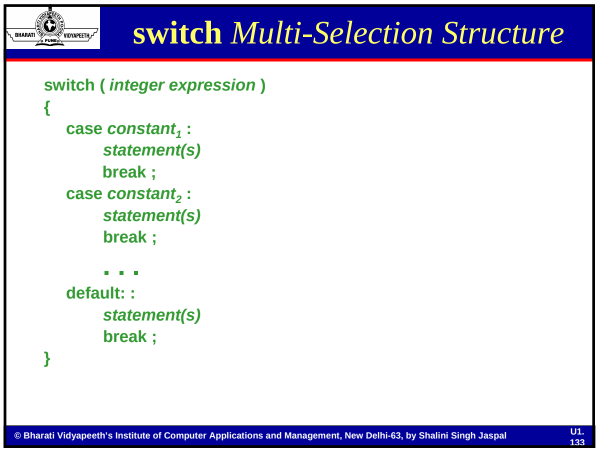


Multiple Selection with if-else

```
if (day == 0 ) {
    printf (“Sunday”) ;
} else if (day == 1 ) {
    printf (“Monday”) ;
} else if (day == 2) {
    printf (“Tuesday”) ;
}
…
 else if (day == 6) {
    printf (“Saturday”) ;
} else {
    printf (“Error -
  invalid day.\n”) ;
}
```

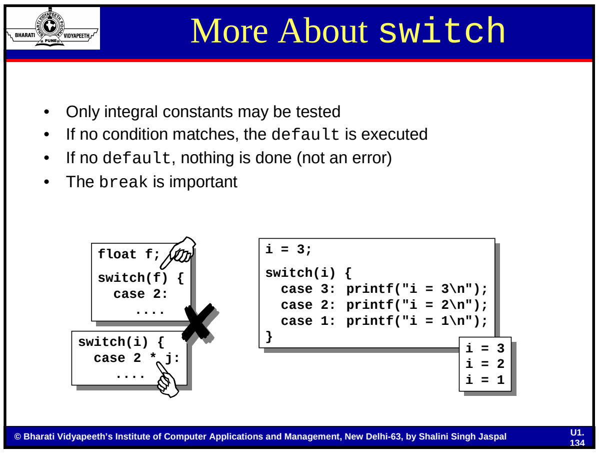
This if-else structure is more efficient than the corresponding if structure for a particular case.

Which one!!!



switch *Multi-Selection Structure*

switch (*integer expression*)
{
    case *constant$_1$* :
        statement(s)
        break ;
    case *constant$_2$* :
        statement(s)
        break ;
        . . .
    default: :
        statement(s)
        break ;
}



More About `switch`

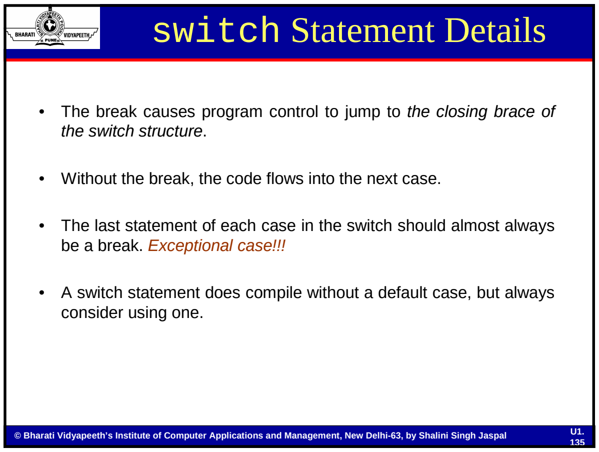
- Only integral constants may be tested
- If no condition matches, the `default` is executed
- If no `default`, nothing is done (not an error)
- The `break` is important

```
float f;
switch(f) {
  case 2:
    ....
```

```
switch(i) {
  case 2 * j:
    ....
```

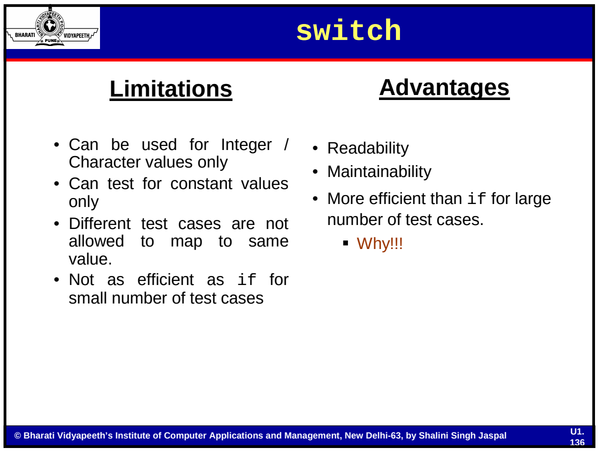
```
i = 3;
switch(i) {
  case 3: printf("i = 3\n");
  case 2: printf("i = 2\n");
  case 1: printf("i = 1\n");
}
```

```
i = 3
i = 2
i = 1
```



`switch` Statement Details

- The break causes program control to jump to *the closing brace of the switch structure*.
- Without the break, the code flows into the next case.
- The last statement of each case in the switch should almost always be a break. *Exceptional case!!!*
- A switch statement does compile without a default case, but always consider using one.



switch

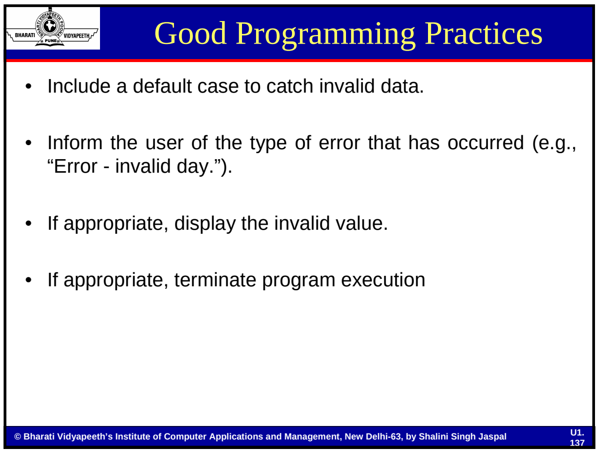
Limitations

- Can be used for Integer / Character values only
- Can test for constant values only
- Different test cases are not allowed to map to same value.
- Not as efficient as `if` for small number of test cases

Advantages

- Readability
- Maintainability
- More efficient than `if` for large number of test cases.
  - Why!!!

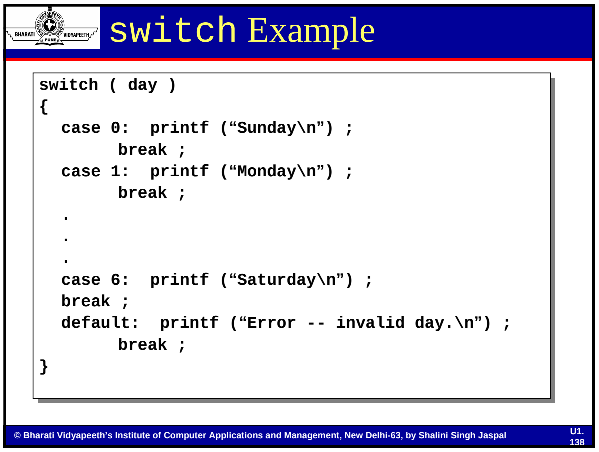
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 136



Good Programming Practices

- Include a default case to catch invalid data.
- Inform the user of the type of error that has occurred (e.g., “Error - invalid day.”).
- If appropriate, display the invalid value.
- If appropriate, terminate program execution

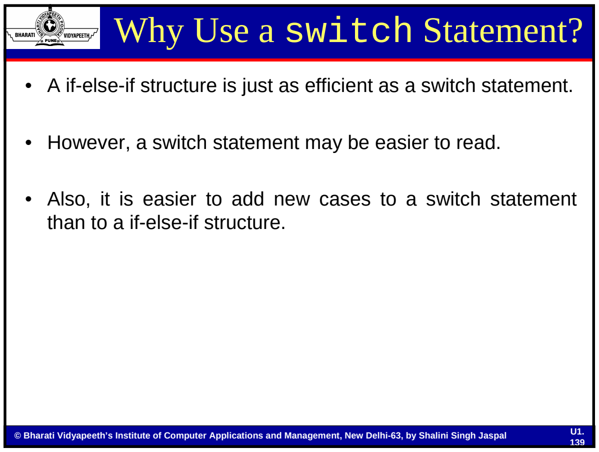
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 137



switch Example

```
switch ( day )
{
  case 0:  printf (“Sunday\n”) ;
        break ;
  case 1:  printf (“Monday\n”) ;
        break ;
  .
  .
  .
  case 6:  printf (“Saturday\n”) ;
  break ;
  default:  printf (“Error -- invalid day.\n”) ;
        break ;
}
```

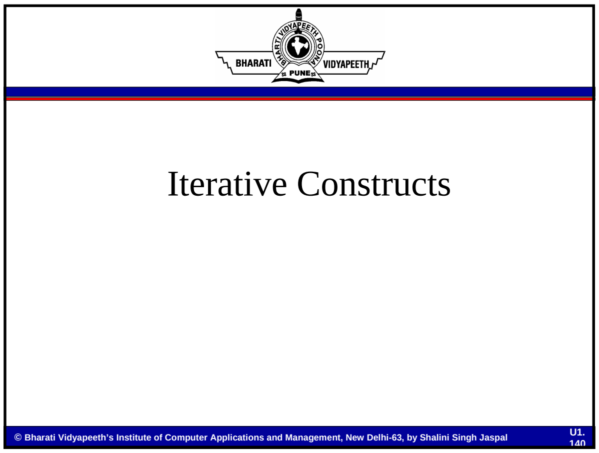
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 138



Why Use a `switch` Statement?

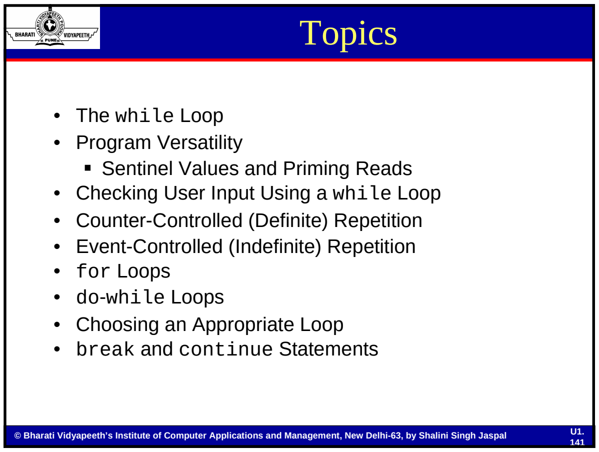
- A if-else-if structure is just as efficient as a switch statement.
- However, a switch statement may be easier to read.
- Also, it is easier to add new cases to a switch statement than to a if-else-if structure.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 139



Iterative Constructs

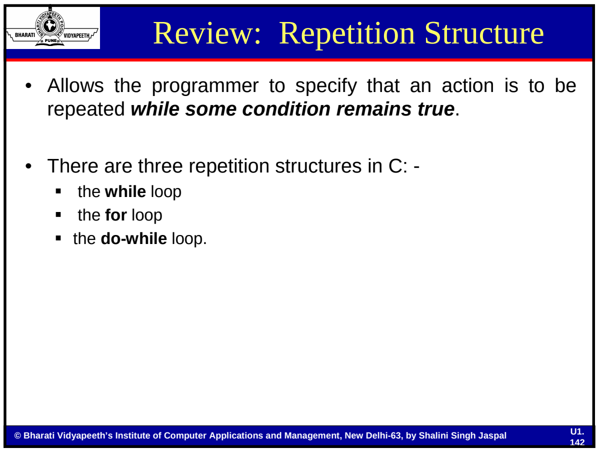
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 140



Topics

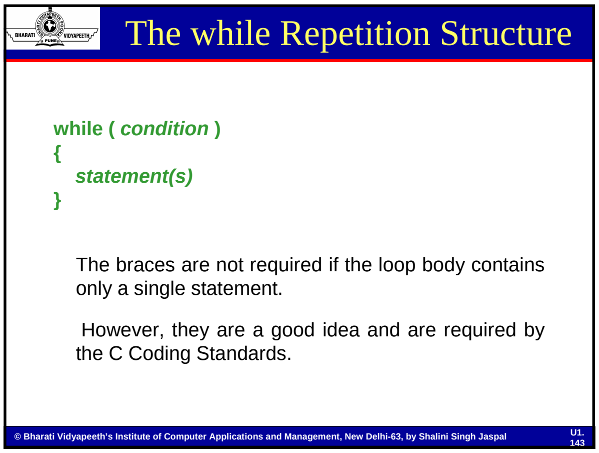
- The `while` Loop
- Program Versatility
  - Sentinel Values and Priming Reads
- Checking User Input Using a `while` Loop
- Counter-Controlled (Definite) Repetition
- Event-Controlled (Indefinite) Repetition
- `for` Loops
- `do-while` Loops
- Choosing an Appropriate Loop
- `break` and `continue` Statements

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 141



Review: Repetition Structure

- Allows the programmer to specify that an action is to be repeated ***while some condition remains true***.
- There are three repetition structures in C: -
  - the **while** loop
  - the **for** loop
  - the **do-while** loop.

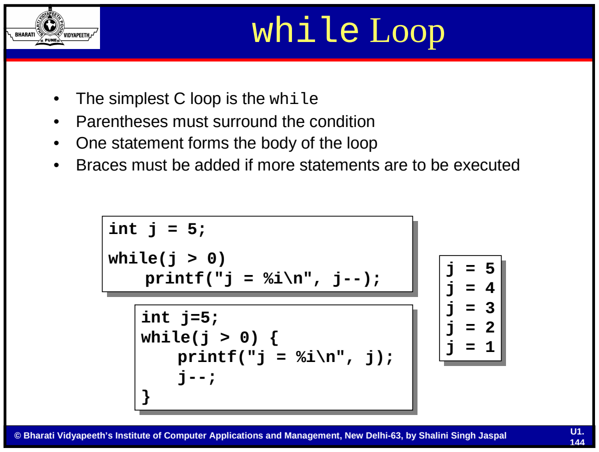


The while Repetition Structure

while (***condition*** **)**
{
   statement(s)
}

The braces are not required if the loop body contains only a single statement.

However, they are a good idea and are required by the C Coding Standards.



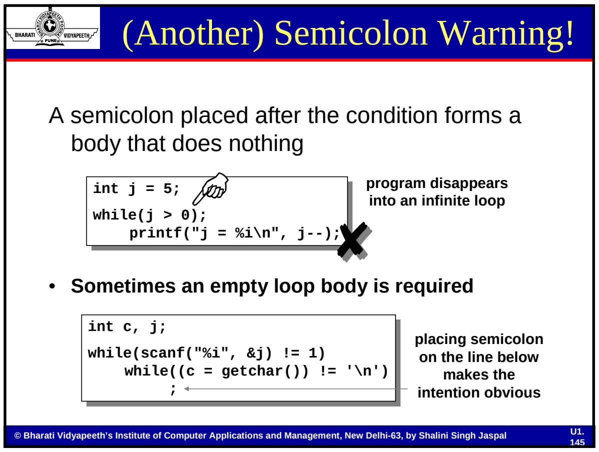
`while` Loop

- The simplest C loop is the `while`
- Parentheses must surround the condition
- One statement forms the body of the loop
- Braces must be added if more statements are to be executed

```
int j = 5;

while(j > 0)
    printf("j = %i\n", j--);
```

```
int j=5;
while(j > 0) {
    printf("j = %i\n", j);
    j--;
}
```

```
j = 5
j = 4
j = 3
j = 2
j = 1
```



(Another) Semicolon Warning!

A semicolon placed after the condition forms a body that does nothing

```
int j = 5;
while(j > 0);
    printf("j = %i\n", j--);
```

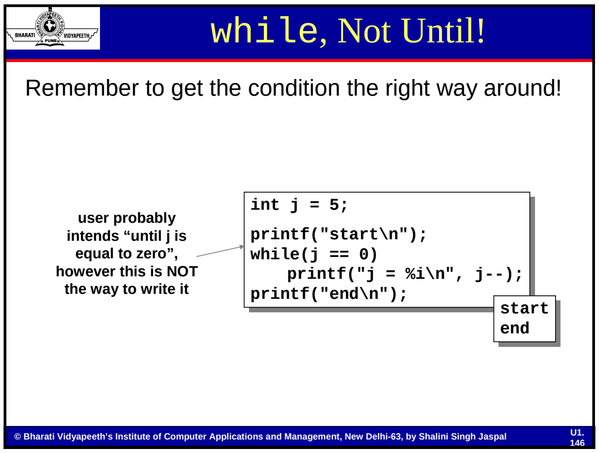
program disappears into an infinite loop

- **Sometimes an empty loop body is required**

```
int c, j;
while(scanf("%i", &j) != 1)
    while((c = getchar()) != '\n')
        ;
```

placing semicolon on the line below makes the intention obvious

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 145



`while`, Not Until!

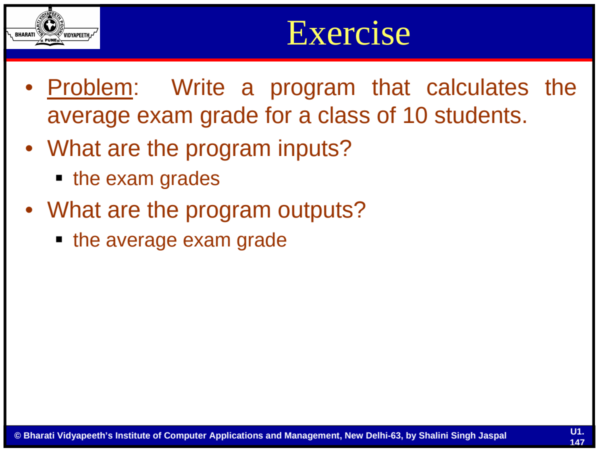
Remember to get the condition the right way around!

user probably intends "until j is equal to zero", however this is NOT the way to write it

```
int j = 5;
printf("start\n");
while(j == 0)
    printf("j = %i\n", j--);
printf("end\n");
```

```
start
end
```

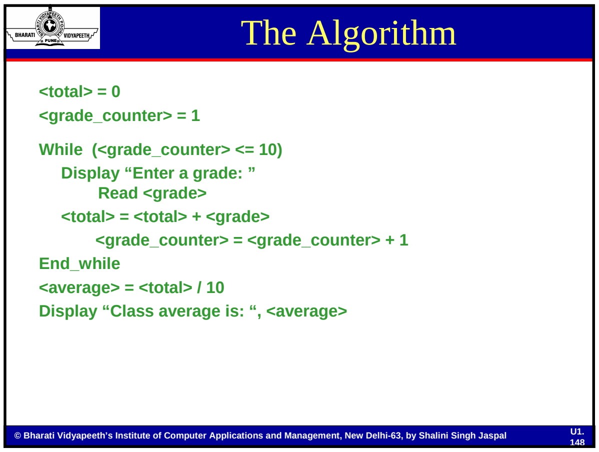
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 146



Exercise

- Problem: Write a program that calculates the average exam grade for a class of 10 students.
- What are the program inputs?
  - the exam grades
- What are the program outputs?
  - the average exam grade

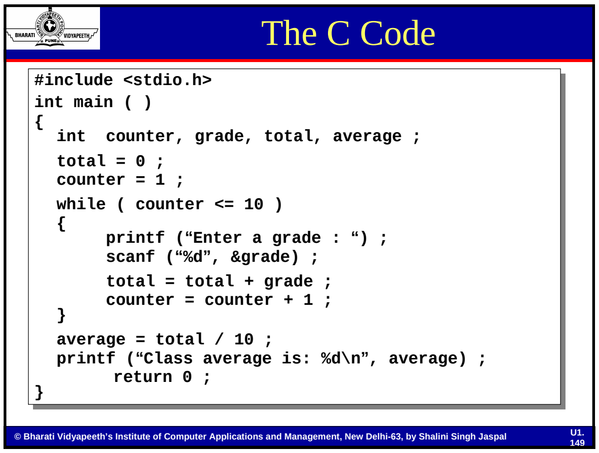
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 147



The Algorithm

```
<total> = 0
<grade_counter> = 1
While  (<grade_counter> <= 10)
    Display "Enter a grade: "
           Read <grade>
    <total> = <total> + <grade>
          <grade_counter> = <grade_counter> + 1
End_while
<average> = <total> / 10
Display "Class average is: ", <average>
```

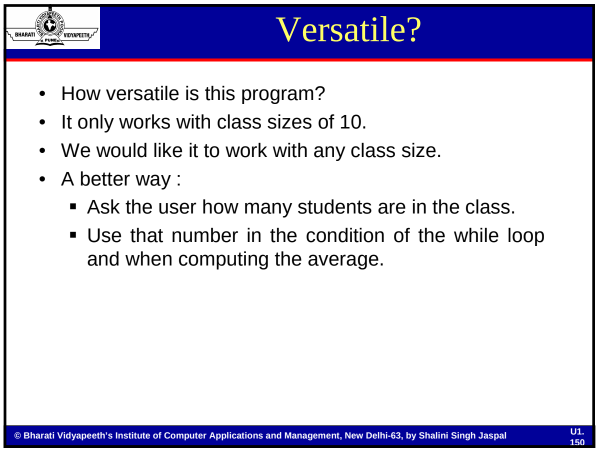
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1.148



The C Code

```
#include <stdio.h>
int main ( )
{
  int  counter, grade, total, average ;
  total = 0 ;
  counter = 1 ;
  while ( counter <= 10 )
  {
       printf ("Enter a grade : ") ;
       scanf ("%d", &grade) ;
       total = total + grade ;
       counter = counter + 1 ;
  }
  average = total / 10 ;
  printf ("Class average is: %d\n", average) ;
        return 0 ;
}
```

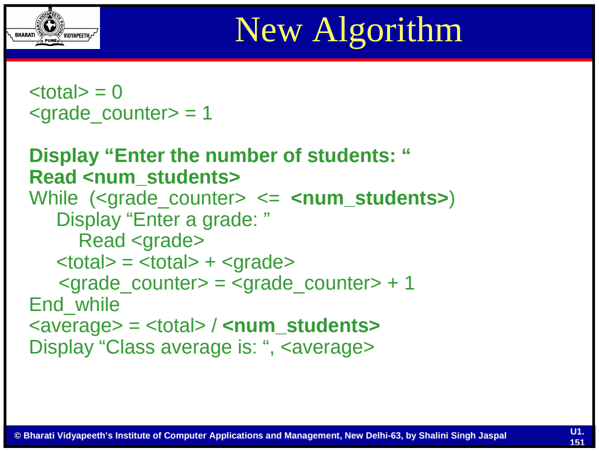
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1.149



Versatile?

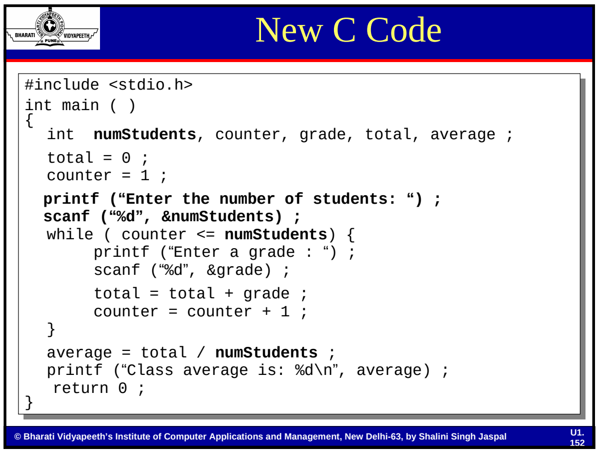
- How versatile is this program?
- It only works with class sizes of 10.
- We would like it to work with any class size.
- A better way :
  - Ask the user how many students are in the class.
  - Use that number in the condition of the while loop and when computing the average.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1.150



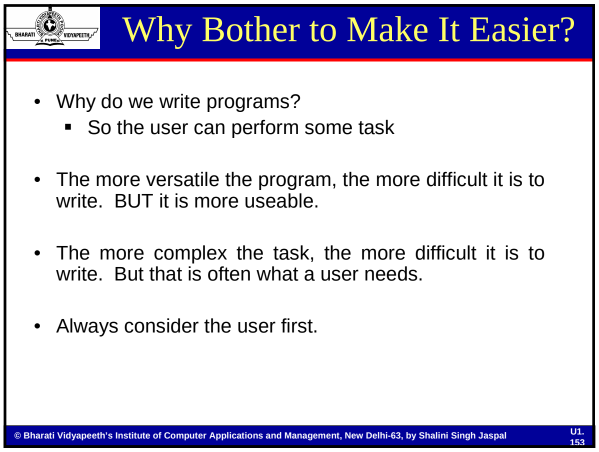
New Algorithm

```
<total> = 0
<grade_counter> = 1

Display "Enter the number of students: "
Read <num_students>
While  (<grade_counter>  <=  <num_students>)
    Display "Enter a grade: "
        Read <grade>
    <total> = <total> + <grade>
    <grade_counter> = <grade_counter> + 1
End_while
<average> = <total> / <num_students>
Display "Class average is: ", <average>
```



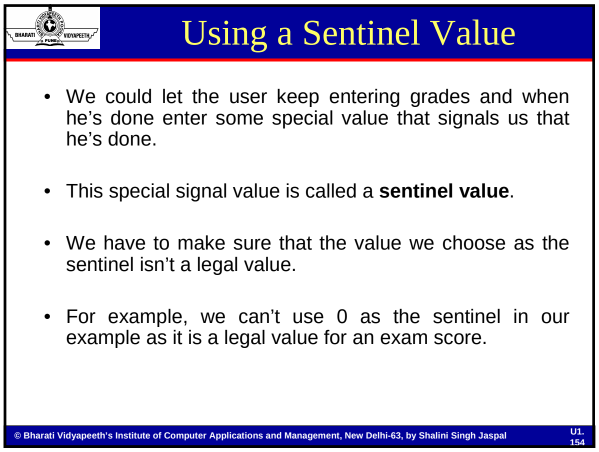
New C Code

```
#include <stdio.h>
int main ( )
{
   int  numStudents, counter, grade, total, average ;
   total = 0 ;
   counter = 1 ;
  printf ("Enter the number of students: ") ;
  scanf ("%d", &numStudents) ;
   while ( counter <= numStudents) {
         printf ("Enter a grade : ") ;
         scanf ("%d", &grade) ;
         total = total + grade ;
         counter = counter + 1 ;
   }
   average = total / numStudents ;
   printf ("Class average is: %d\n", average) ;
    return 0 ;
}
```



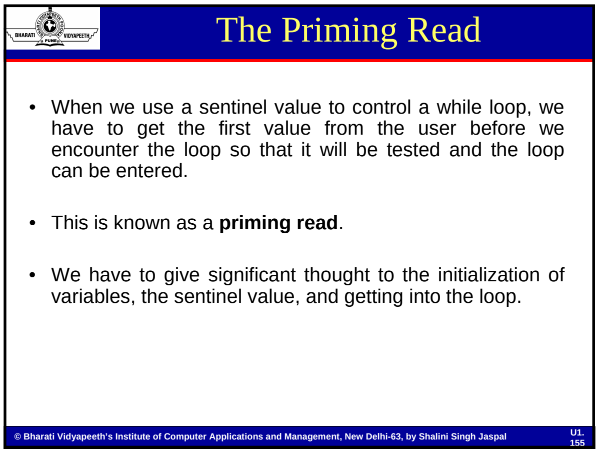
Why Bother to Make It Easier?

- Why do we write programs?
  - So the user can perform some task
- The more versatile the program, the more difficult it is to write. BUT it is more useable.
- The more complex the task, the more difficult it is to write. But that is often what a user needs.
- Always consider the user first.



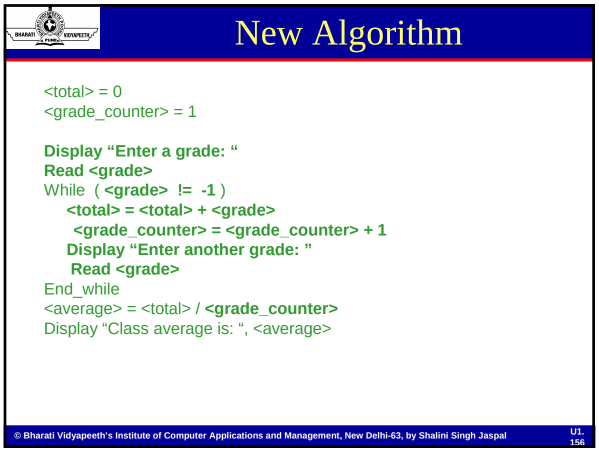
Using a Sentinel Value

- We could let the user keep entering grades and when he's done enter some special value that signals us that he's done.
- This special signal value is called a **sentinel value**.
- We have to make sure that the value we choose as the sentinel isn't a legal value.
- For example, we can't use 0 as the sentinel in our example as it is a legal value for an exam score.



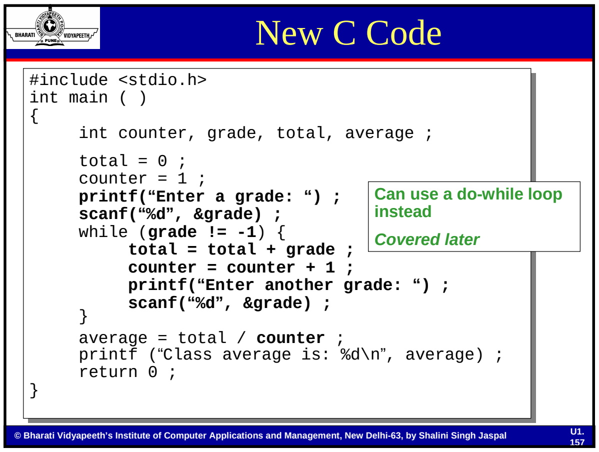
The Priming Read

- When we use a sentinel value to control a while loop, we have to get the first value from the user before we encounter the loop so that it will be tested and the loop can be entered.
- This is known as a **priming read**.
- We have to give significant thought to the initialization of variables, the sentinel value, and getting into the loop.



New Algorithm

```
<total> = 0
<grade_counter> = 1

Display "Enter a grade: "
Read <grade>
While  ( <grade>  !=  -1 )
    <total> = <total> + <grade>
     <grade_counter> = <grade_counter> + 1
    Display "Enter another grade: "
     Read <grade>
End_while
<average> = <total> / <grade_counter>
Display "Class average is: ", <average>
```

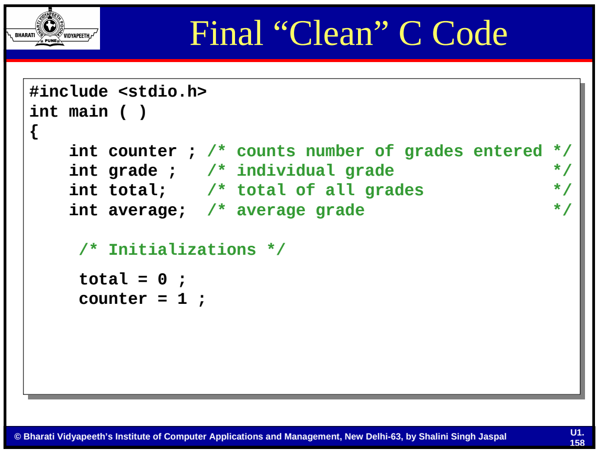


New C Code

```
#include <stdio.h>
int main ( )
{
     int counter, grade, total, average ;

     total = 0 ;
     counter = 1 ;
     printf(“Enter a grade: “) ;
     scanf(“%d”, &grade) ;
     while (grade != -1) {
          total = total + grade ;
          counter = counter + 1 ;
          printf(“Enter another grade: “) ;
          scanf(“%d”, &grade) ;
     }
     average = total / counter ;
     printf (“Class average is: %d\n”, average) ;
     return 0 ;
}
```

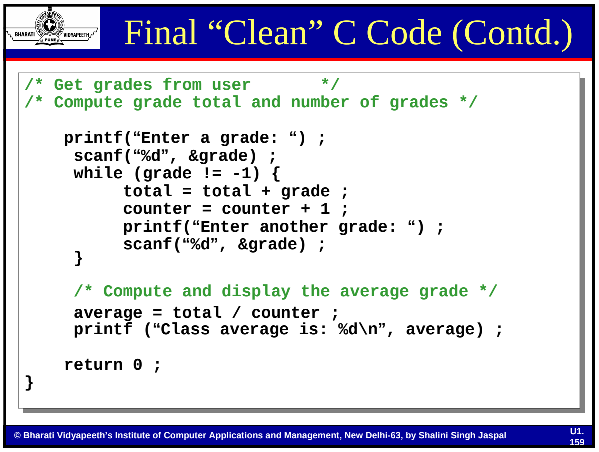
Can use a do-while loop instead

Covered later



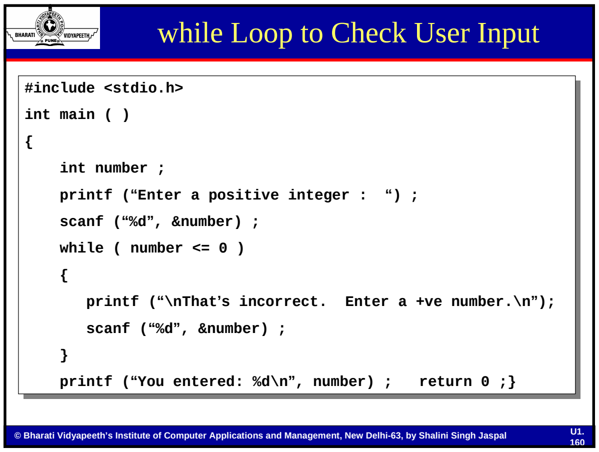
Final “Clean” C Code

```
#include <stdio.h>
int main ( )
{
    int counter ; /* counts number of grades entered */
    int grade ;   /* individual grade                */
    int total;    /* total of all grades             */
    int average;  /* average grade                   */

     /* Initializations */

     total = 0 ;
     counter = 1 ;
```



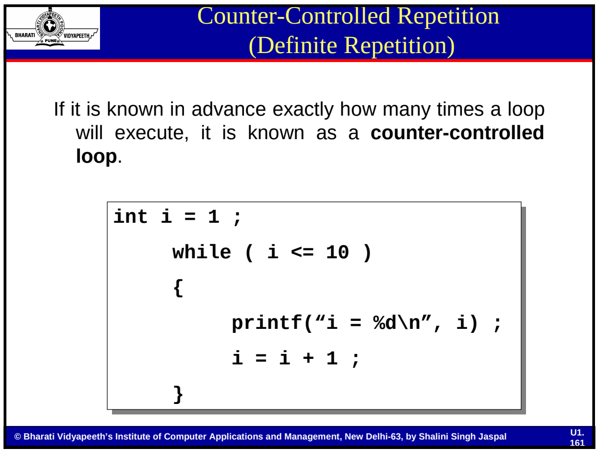
Final “Clean” C Code (Contd.)

```
/* Get grades from user       */
/* Compute grade total and number of grades */

    printf(“Enter a grade: “) ;
     scanf(“%d”, &grade) ;
     while (grade != -1) {
          total = total + grade ;
          counter = counter + 1 ;
          printf(“Enter another grade: “) ;
          scanf(“%d”, &grade) ;
     }

     /* Compute and display the average grade */
     average = total / counter ;
     printf (“Class average is: %d\n”, average) ;

    return 0 ;
}
```



while Loop to Check User Input

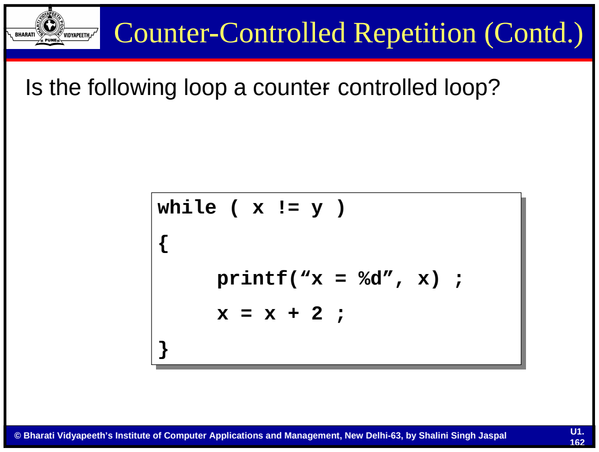
```
#include <stdio.h>
int main ( )
{
    int number ;
    printf (“Enter a positive integer :  “) ;
    scanf (“%d”, &number) ;
    while ( number <= 0 )
    {
       printf (“\nThat’s incorrect.  Enter a +ve number.\n”);
       scanf (“%d”, &number) ;
    }
    printf (“You entered: %d\n”, number) ;   return 0 ;}
```



Counter-Controlled Repetition (Definite Repetition)

If it is known in advance exactly how many times a loop will execute, it is known as a **counter-controlled loop**.

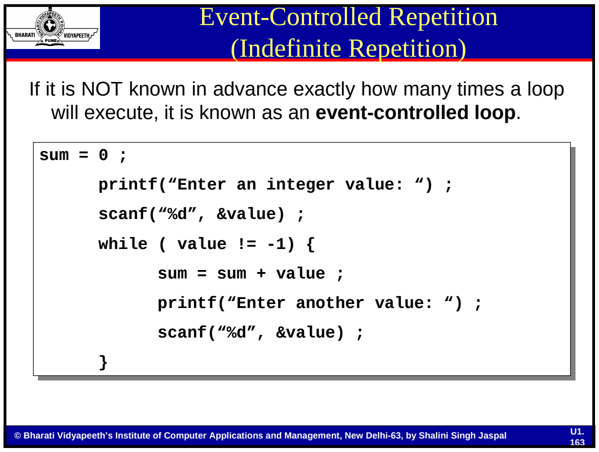
```
int i = 1 ;
      while ( i <= 10 )
      {
            printf(“i = %d\n”, i) ;
            i = i + 1 ;
      }
```



Counter-Controlled Repetition (Contd.)

Is the following loop a counter controlled loop?

```
while ( x != y )
{
      printf(“x = %d”, x) ;
      x = x + 2 ;
}
```

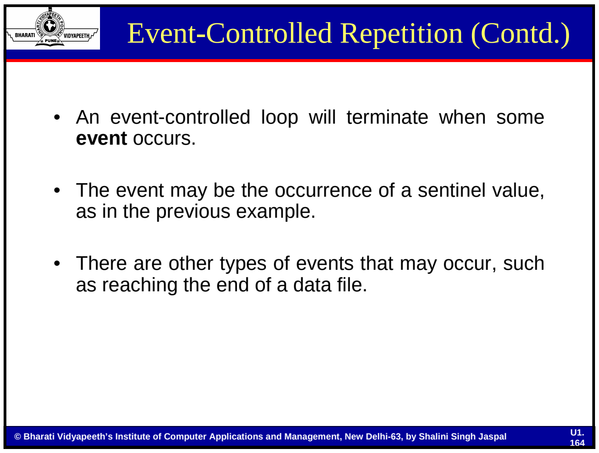


Event-Controlled Repetition (Indefinite Repetition)

If it is NOT known in advance exactly how many times a loop will execute, it is known as an **event-controlled loop**.

```
sum = 0 ;
      printf(“Enter an integer value: “) ;
      scanf(“%d”, &value) ;
      while ( value != -1) {
            sum = sum + value ;
            printf(“Enter another value: “) ;
            scanf(“%d”, &value) ;
      }
```

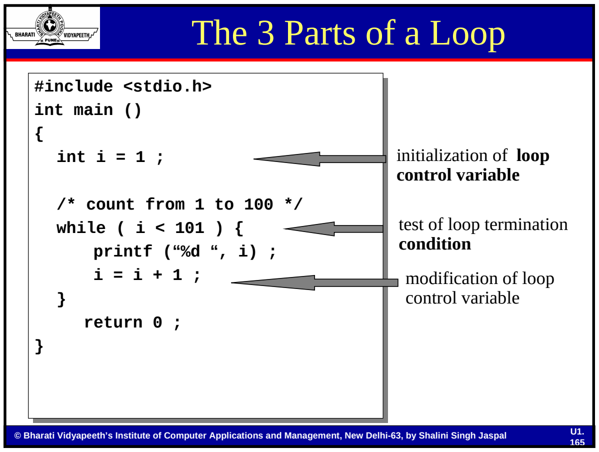
U1.163



Event-Controlled Repetition (Contd.)

- An event-controlled loop will terminate when some **event** occurs.
- The event may be the occurrence of a sentinel value, as in the previous example.
- There are other types of events that may occur, such as reaching the end of a data file.

U1.164

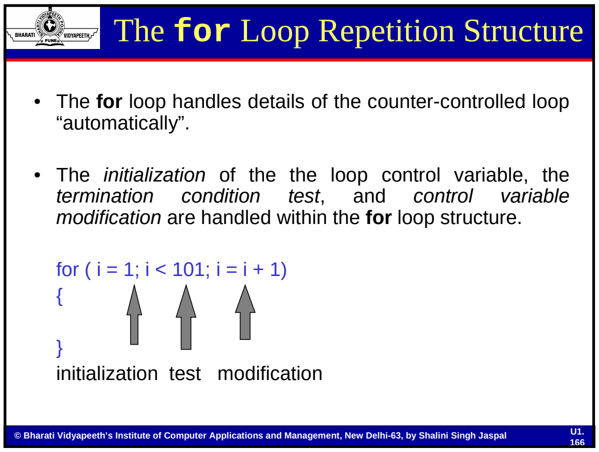


The 3 Parts of a Loop

```
#include <stdio.h>
int main ()
{
  int i = 1 ;

  /* count from 1 to 100 */
  while ( i < 101 ) {
      printf (“%d “, i) ;
      i = i + 1 ;
  }
     return 0 ;
}
```

initialization of **loop control variable**
test of loop termination **condition**
modification of loop control variable

U1.165

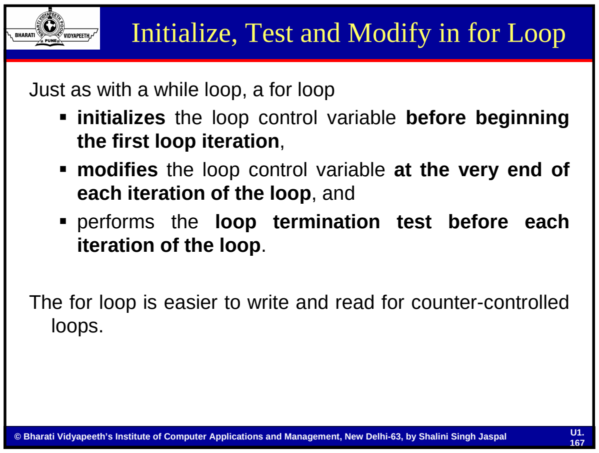


The for Loop Repetition Structure

- The **for** loop handles details of the counter-controlled loop “automatically”.
- The *initialization* of the the loop control variable, the *termination condition test*, and *control variable modification* are handled within the **for** loop structure.

```
for ( i = 1; i < 101; i = i + 1)
{

}
```

initialization test modification

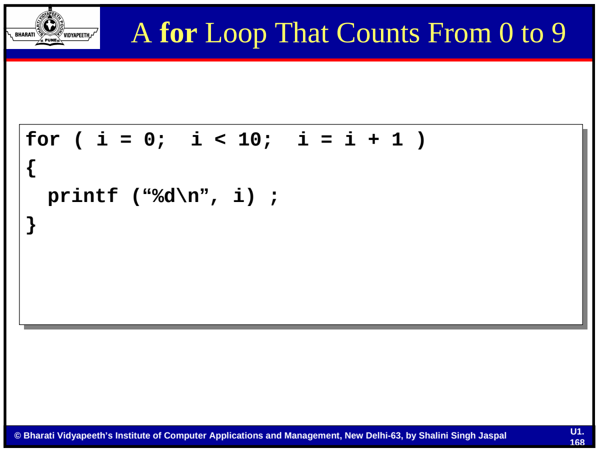


Initialize, Test and Modify in for Loop

Just as with a while loop, a for loop

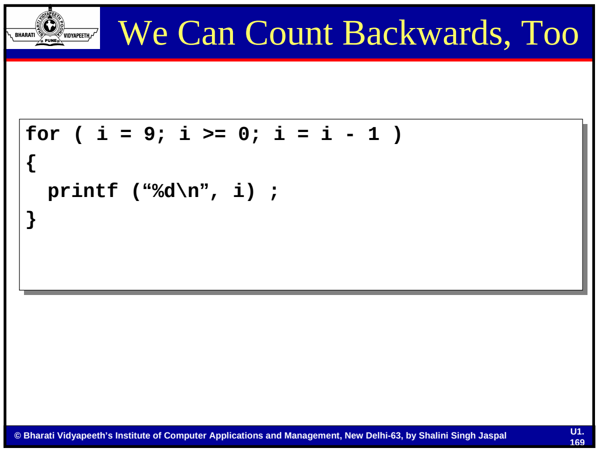
- **initializes** the loop control variable **before beginning the first loop iteration**,
- **modifies** the loop control variable **at the very end of each iteration of the loop**, and
- performs the **loop termination test before each iteration of the loop**.

The for loop is easier to write and read for counter-controlled loops.



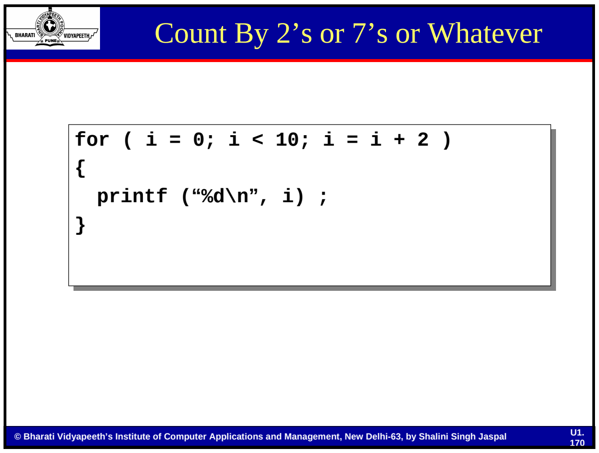
A for Loop That Counts From 0 to 9

```
for ( i = 0;  i < 10;  i = i + 1 )
{
  printf (“%d\n”, i) ;
}
```



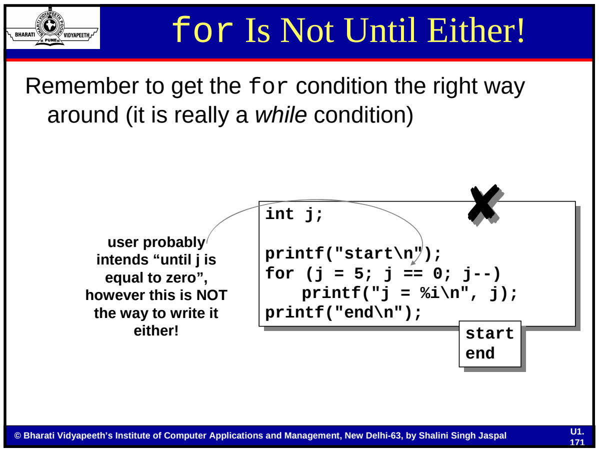
We Can Count Backwards, Too

```
for ( i = 9; i >= 0; i = i - 1 )
{
  printf (“%d\n”, i) ;
}
```



Count By 2’s or 7’s or Whatever

```
for ( i = 0; i < 10; i = i + 2 )
{
  printf (“%d\n”, i) ;
}
```



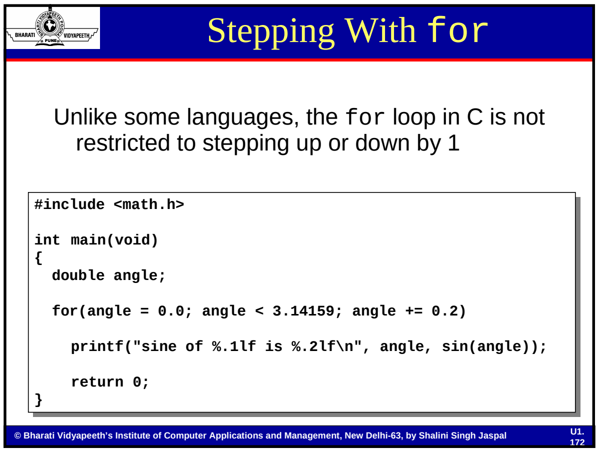
`for` Is Not Until Either!

Remember to get the `for` condition the right way around (it is really a *while* condition)

```
int j;

printf("start\n");
for (j = 5; j == 0; j--)
    printf("j = %i\n", j);
printf("end\n");
```

```
start
end
```

user probably intends “until j is equal to zero”, however this is NOT the way to write it either!

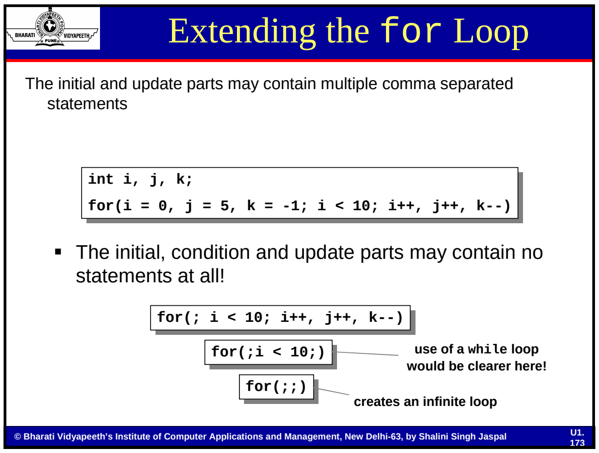


Stepping With `for`

Unlike some languages, the `for` loop in C is not restricted to stepping up or down by 1

```
#include <math.h>

int main(void)
{
  double angle;

  for(angle = 0.0; angle < 3.14159; angle += 0.2)

    printf("sine of %.1lf is %.2lf\n", angle, sin(angle));

    return 0;
}
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 172



Extending the `for` Loop

The initial and update parts may contain multiple comma separated statements

```
int i, j, k;
for(i = 0, j = 5, k = -1; i < 10; i++, j++, k--)
```

- The initial, condition and update parts may contain no statements at all!

```
for(; i < 10; i++, j++, k--)
```

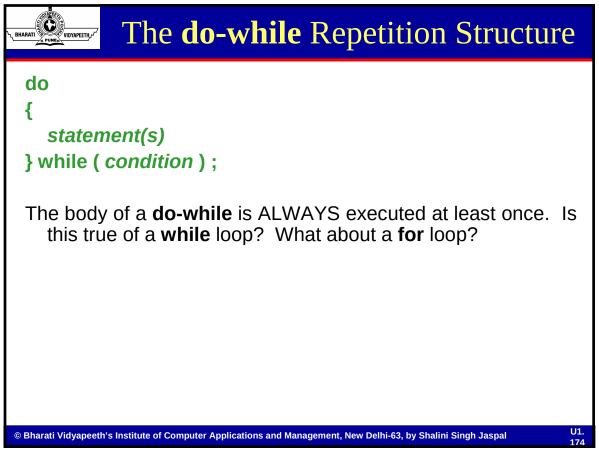
```
for(;i < 10;)
```

use of a `while` loop would be clearer here!

```
for(;;)
```

creates an infinite loop

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 173

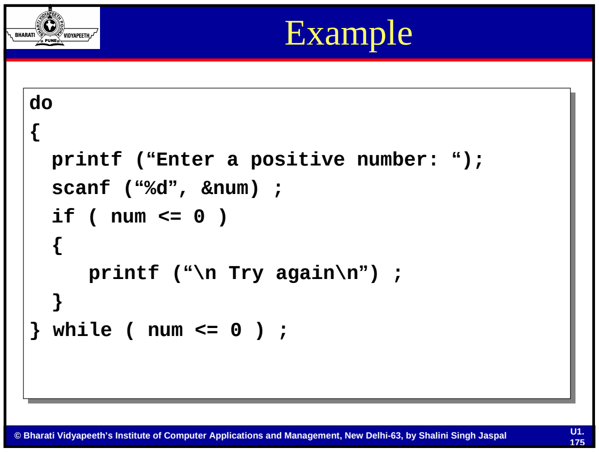


The **do-while** Repetition Structure

do
{
    statement(s)
} while (*condition*) ;

The body of a **do-while** is ALWAYS executed at least once. Is this true of a **while** loop? What about a **for** loop?

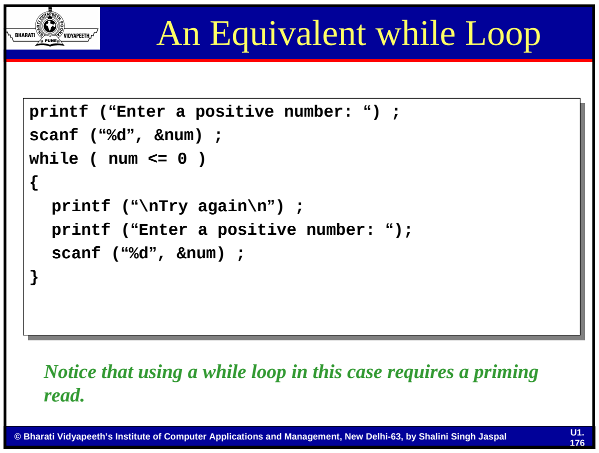
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 174



Example

```
do
{
  printf (“Enter a positive number: “);
  scanf (“%d”, &num) ;
  if ( num <= 0 )
  {
     printf (“\n Try again\n”) ;
  }
} while ( num <= 0 ) ;
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 175

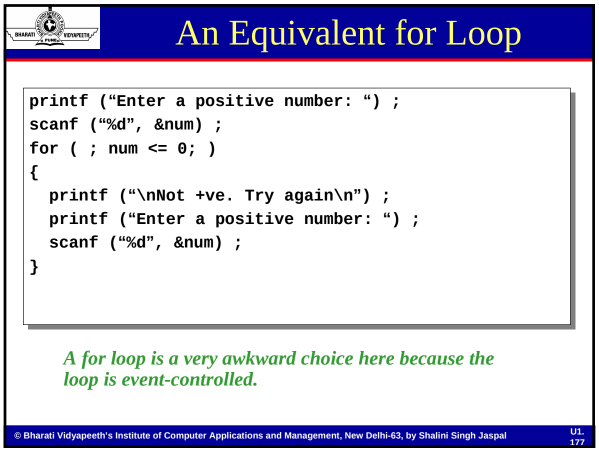


An Equivalent while Loop

```
printf (“Enter a positive number: “) ;
scanf (“%d”, &num) ;
while ( num <= 0 )
{
  printf (“\nTry again\n”) ;
  printf (“Enter a positive number: “);
  scanf (“%d”, &num) ;
}
```

Notice that using a while loop in this case requires a priming read.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 176

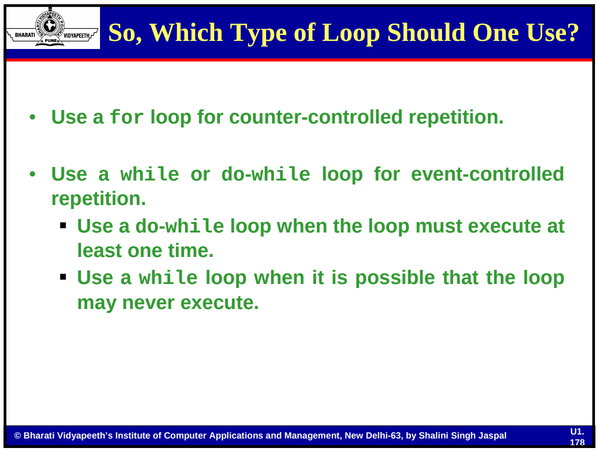


An Equivalent for Loop

```
printf (“Enter a positive number: “) ;
scanf (“%d”, &num) ;
for ( ; num <= 0; )
{
  printf (“\nNot +ve. Try again\n”) ;
  printf (“Enter a positive number: “) ;
  scanf (“%d”, &num) ;
}
```

A for loop is a very awkward choice here because the loop is event-controlled.

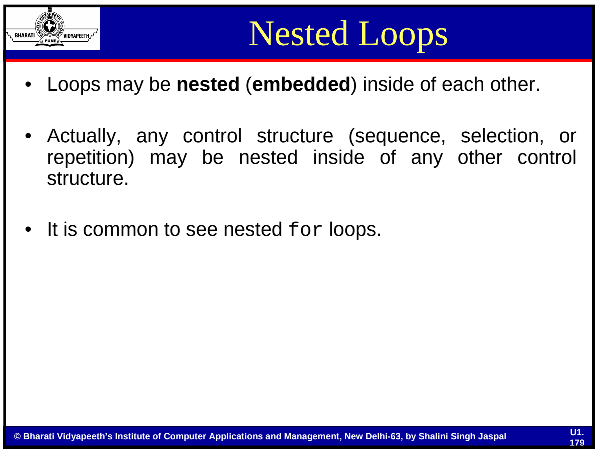
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 177



So, Which Type of Loop Should One Use?

- Use a `for` loop for counter-controlled repetition.
- Use a `while` or `do-while` loop for event-controlled repetition.
  - Use a `do-while` loop when the loop must execute at least one time.
  - Use a `while` loop when it is possible that the loop may never execute.

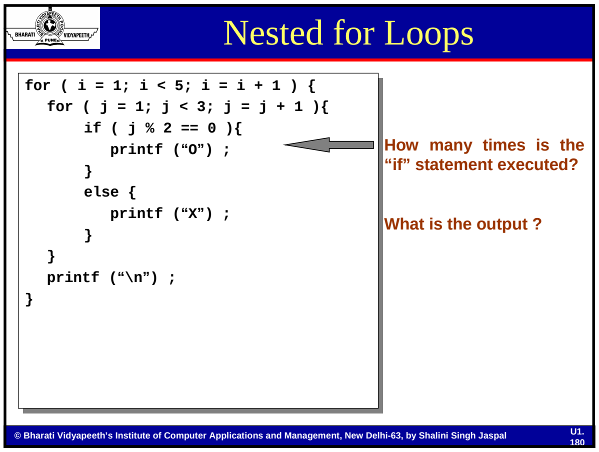
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 178



Nested Loops

- Loops may be **nested** (**embedded**) inside of each other.
- Actually, any control structure (sequence, selection, or repetition) may be nested inside of any other control structure.
- It is common to see nested `for` loops.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 179



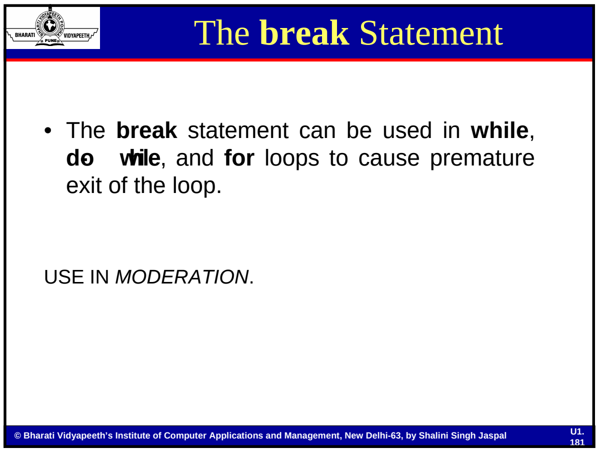
Nested for Loops

```
for ( i = 1; i < 5; i = i + 1 ) {
  for ( j = 1; j < 3; j = j + 1 ){
      if ( j % 2 == 0 ){
         printf (“O”) ;
      }
      else {
         printf (“X”) ;
      }
  }
  printf (“\n”) ;
}
```

How many times is the “if” statement executed?

What is the output ?

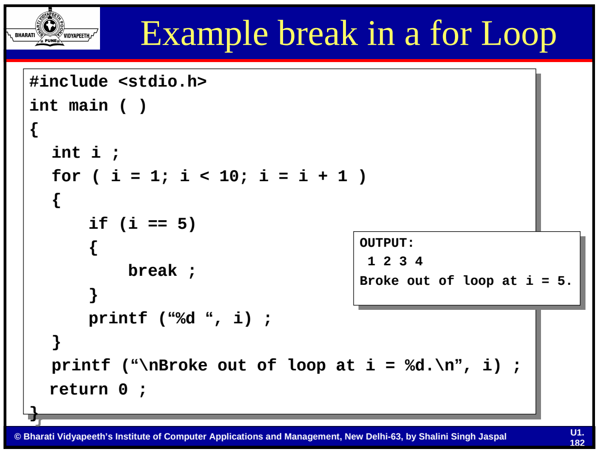
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 180



The **break** Statement

- The **break** statement can be used in **while**, **do while**, and **for** loops to cause premature exit of the loop.

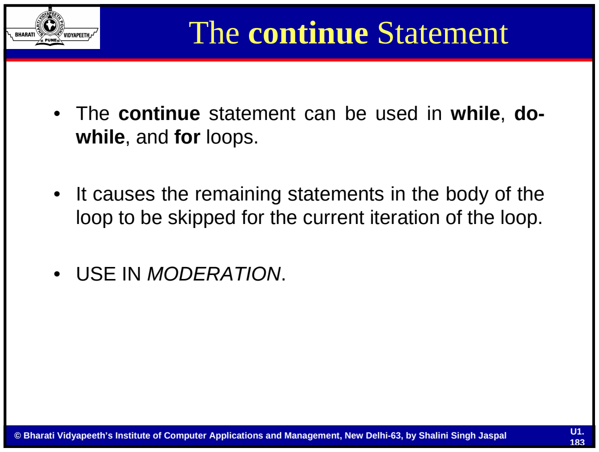
USE IN *MODERATION*.



Example break in a for Loop

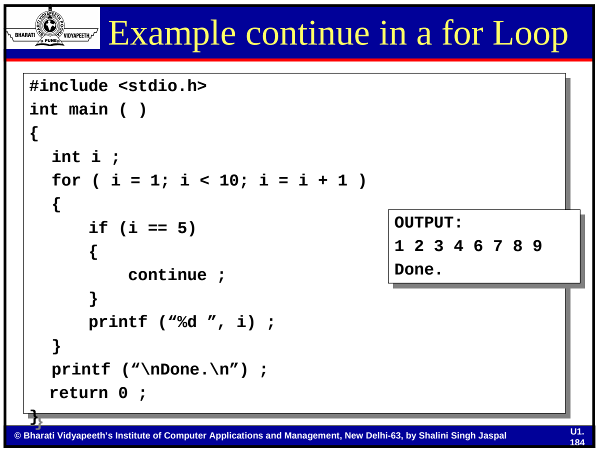
```
#include <stdio.h>
int main ( )
{
  int i ;
  for ( i = 1; i < 10; i = i + 1 )
  {
      if (i == 5)
      {
           break ;
      }
      printf (“%d “, i) ;
  }
  printf (“\nBroke out of loop at i = %d.\n”, i) ;
  return 0 ;
}
```

```
OUTPUT:
 1 2 3 4
Broke out of loop at i = 5.
```



The **continue** Statement

- The **continue** statement can be used in **while**, **do-while**, and **for** loops.
- It causes the remaining statements in the body of the loop to be skipped for the current iteration of the loop.
- USE IN *MODERATION*.

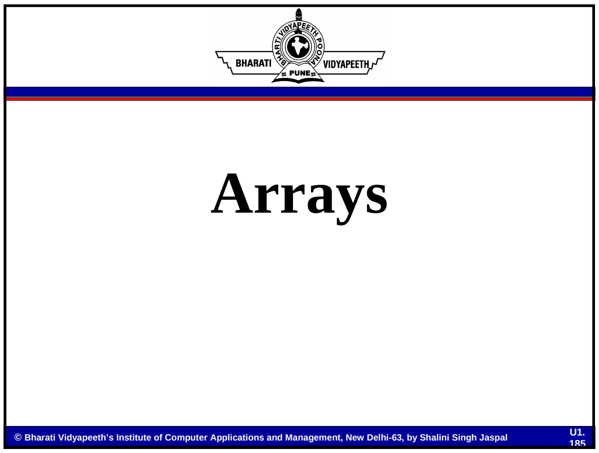


Example continue in a for Loop

```
#include <stdio.h>
int main ( )
{
   int i ;
   for ( i = 1; i < 10; i = i + 1 )
   {
        if (i == 5)
        {
             continue ;
        }
        printf (“%d ”, i) ;
   }
   printf (“\nDone.\n”) ;
   return 0 ;
}
```

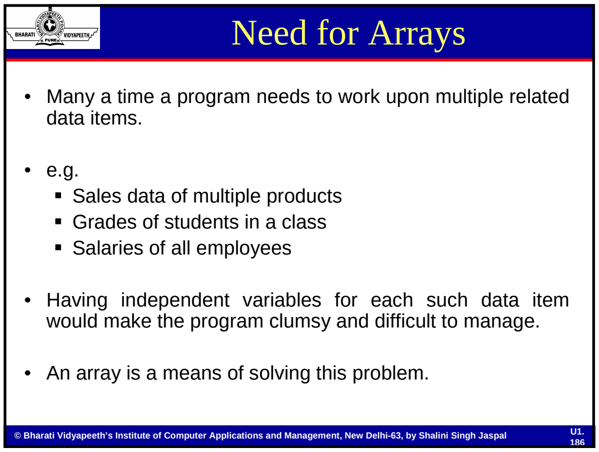
```
OUTPUT:
1 2 3 4 6 7 8 9
Done.
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1.184



Arrays

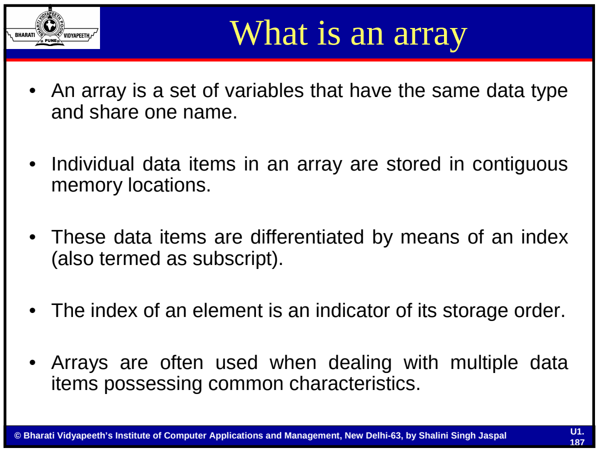
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1.185



Need for Arrays

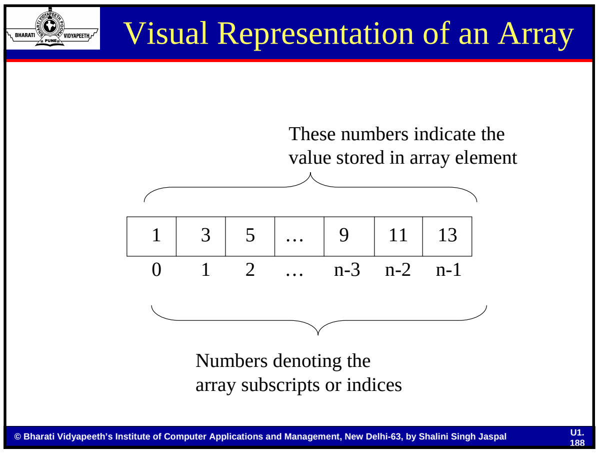
- Many a time a program needs to work upon multiple related data items.
- e.g.
  - Sales data of multiple products
  - Grades of students in a class
  - Salaries of all employees
- Having independent variables for each such data item would make the program clumsy and difficult to manage.
- An array is a means of solving this problem.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1.186



What is an array

- An array is a set of variables that have the same data type and share one name.
- Individual data items in an array are stored in contiguous memory locations.
- These data items are differentiated by means of an index (also termed as subscript).
- The index of an element is an indicator of its storage order.
- Arrays are often used when dealing with multiple data items possessing common characteristics.

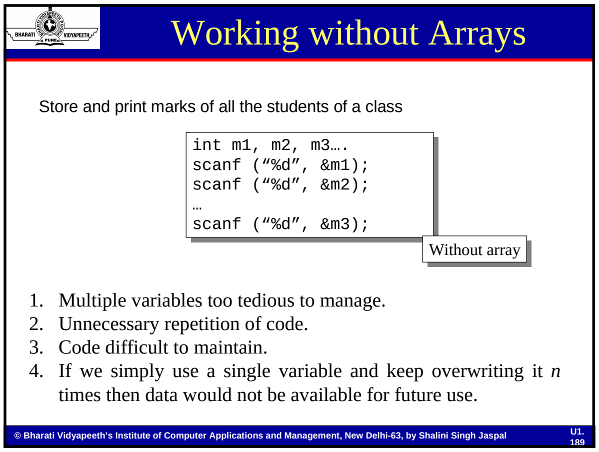


Visual Representation of an Array

These numbers indicate the value stored in array element

1	3	5	…	9	11	13
0	1	2	…	n-3	n-2	n-1

Numbers denoting the array subscripts or indices



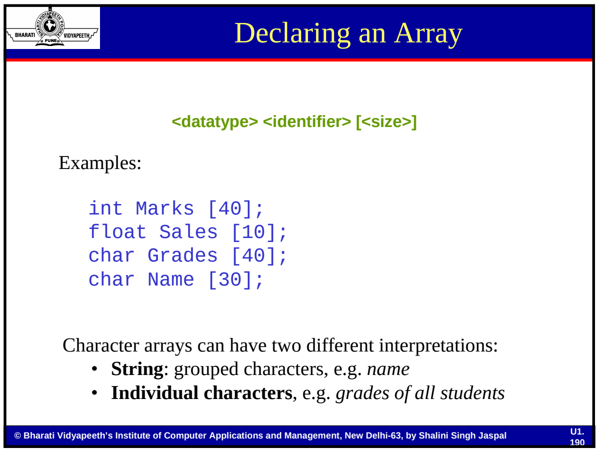
Working without Arrays

Store and print marks of all the students of a class

```
int m1, m2, m3….
scanf (“%d”, &m1);
scanf (“%d”, &m2);
…
scanf (“%d”, &m3);
```

Without array

1. Multiple variables too tedious to manage.
2. Unnecessary repetition of code.
3. Code difficult to maintain.
4. If we simply use a single variable and keep overwriting it *n* times then data would not be available for future use.



Declaring an Array

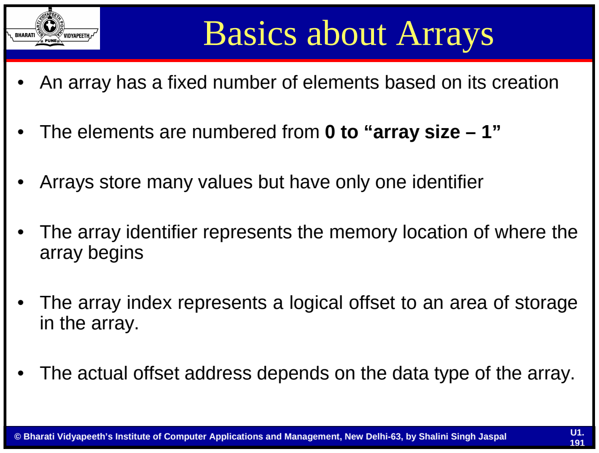
<datatype> <identifier> [<size>]

Examples:

```
int Marks [40];
float Sales [10];
char Grades [40];
char Name [30];
```

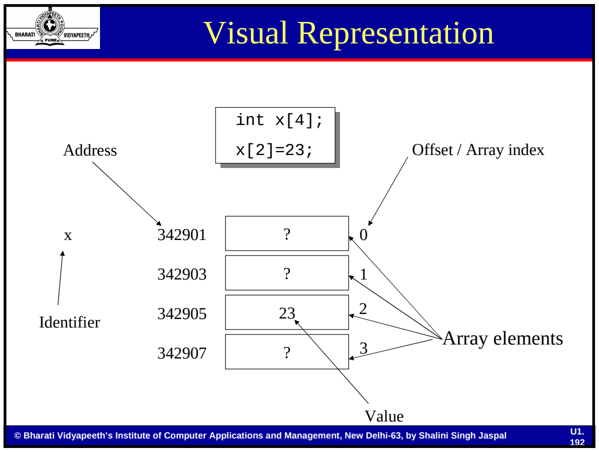
Character arrays can have two different interpretations:

- **String**: grouped characters, e.g. *name*
- **Individual characters**, e.g. *grades of all students*



Basics about Arrays

- An array has a fixed number of elements based on its creation
- The elements are numbered from **0 to "array size – 1"**
- Arrays store many values but have only one identifier
- The array identifier represents the memory location of where the array begins
- The array index represents a logical offset to an area of storage in the array.
- The actual offset address depends on the data type of the array.



Visual Representation

```
int x[4];
x[2]=23;
```

Address

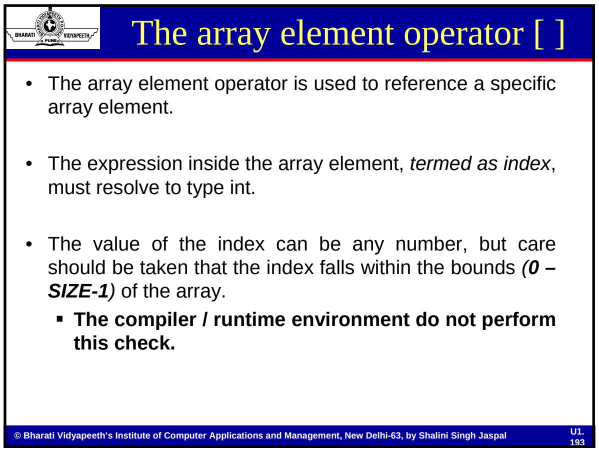
Offset / Array index

x	342901	?	0
	342903	?	1
	342905	23	2
	342907	?	3

Identifier

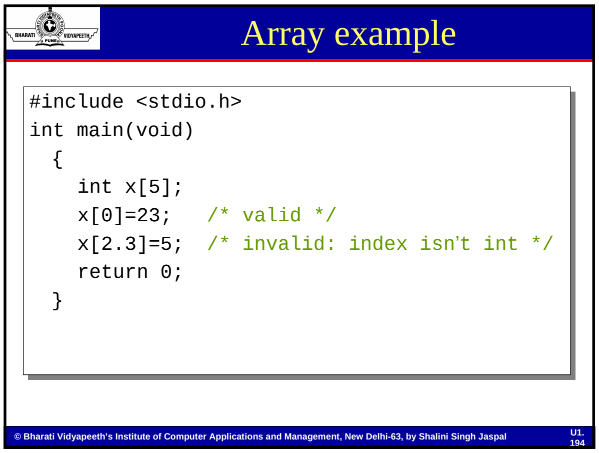
Array elements

Value



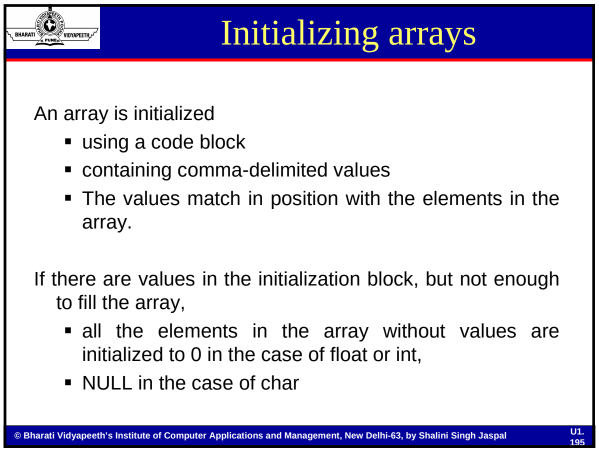
The array element operator [ ]

- The array element operator is used to reference a specific array element.
- The expression inside the array element, *termed as index*, must resolve to type int.
- The value of the index can be any number, but care should be taken that the index falls within the bounds ***(0 – SIZE-1)*** of the array.
  - **The compiler / runtime environment do not perform this check.**



Array example

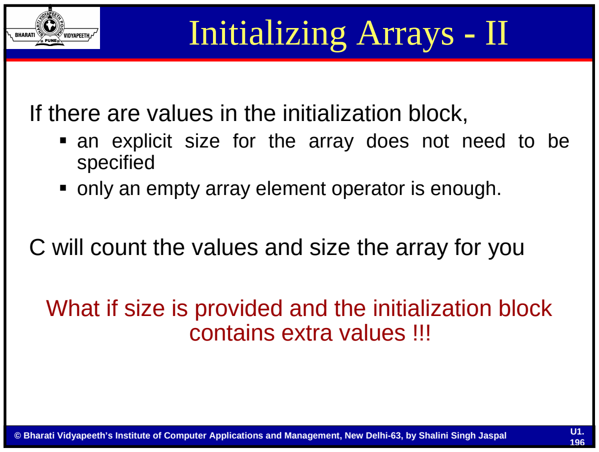
```
#include <stdio.h>
int main(void)
  {
    int x[5];
    x[0]=23;   /* valid */
    x[2.3]=5;  /* invalid: index isn't int */
    return 0;
  }
```



Initializing arrays

An array is initialized
- using a code block
- containing comma-delimited values
- The values match in position with the elements in the array.

If there are values in the initialization block, but not enough to fill the array,
- all the elements in the array without values are initialized to 0 in the case of float or int,
- NULL in the case of char



Initializing Arrays - II

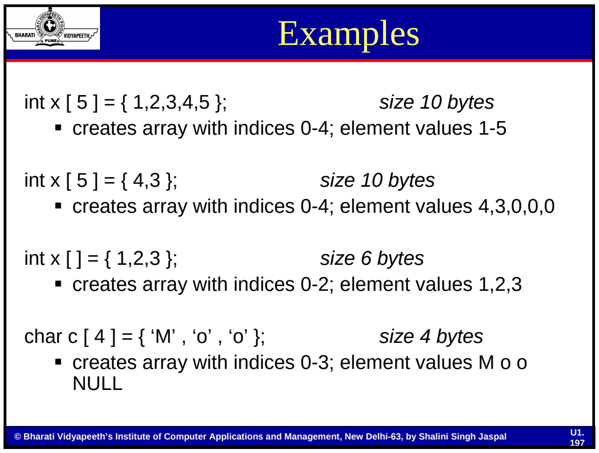
If there are values in the initialization block,

- an explicit size for the array does not need to be specified
- only an empty array element operator is enough.

C will count the values and size the array for you

What if size is provided and the initialization block contains extra values !!!

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 196



Examples

int x [5] = { 1,2,3,4,5 }; *size 10 bytes*

- creates array with indices 0-4; element values 1-5

int x [5] = { 4,3 }; *size 10 bytes*

- creates array with indices 0-4; element values 4,3,0,0,0

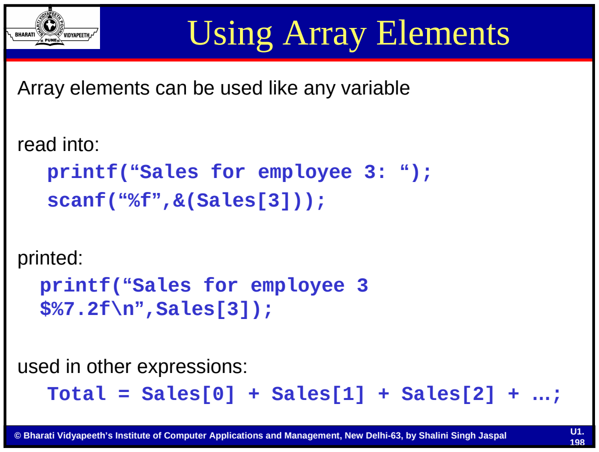
int x [ ] = { 1,2,3 }; *size 6 bytes*

- creates array with indices 0-2; element values 1,2,3

char c [4] = { ‘M’ , ‘o’ , ‘o’ }; *size 4 bytes*

- creates array with indices 0-3; element values M o o NULL

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 197



Using Array Elements

Array elements can be used like any variable

read into:

```
printf(“Sales for employee 3: “);
scanf(“%f”,&(Sales[3]));
```

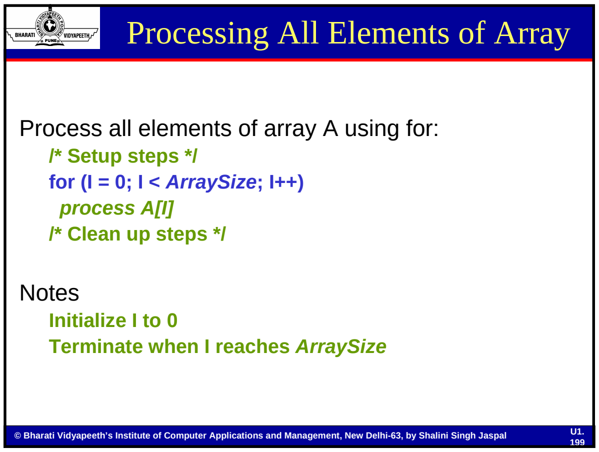
printed:

```
printf(“Sales for employee 3
$%7.2f\n”,Sales[3]);
```

used in other expressions:

```
Total = Sales[0] + Sales[1] + Sales[2] + …;
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 198



Processing All Elements of Array

Process all elements of array A using for:

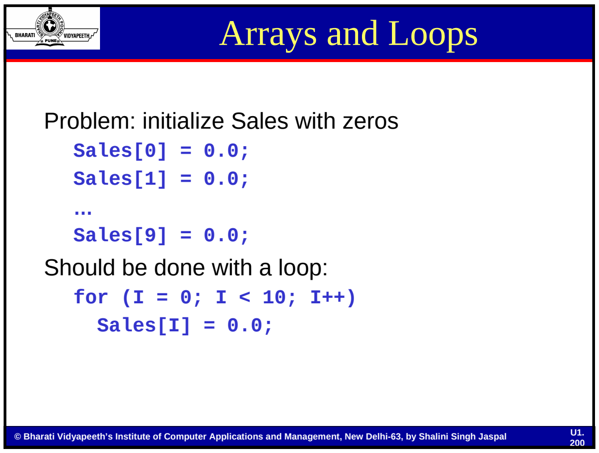
```
/* Setup steps */
for (I = 0; I < ArraySize; I++)
  process A[I]
/* Clean up steps */
```

Notes

Initialize I to 0

Terminate when I reaches *ArraySize*

U1. 199



Arrays and Loops

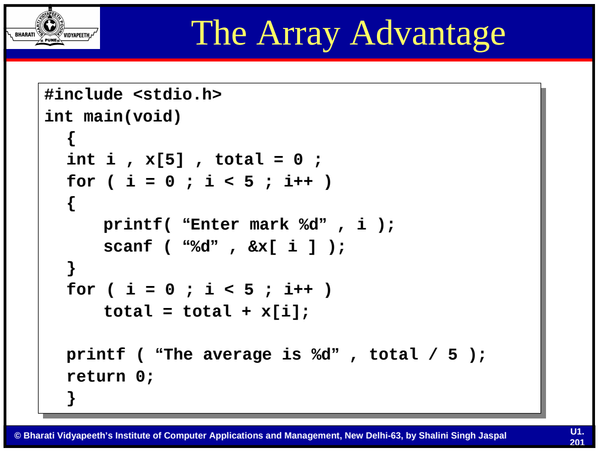
Problem: initialize Sales with zeros

```
Sales[0] = 0.0;
Sales[1] = 0.0;
…
Sales[9] = 0.0;
```

Should be done with a loop:

```
for (I = 0; I < 10; I++)
  Sales[I] = 0.0;
```

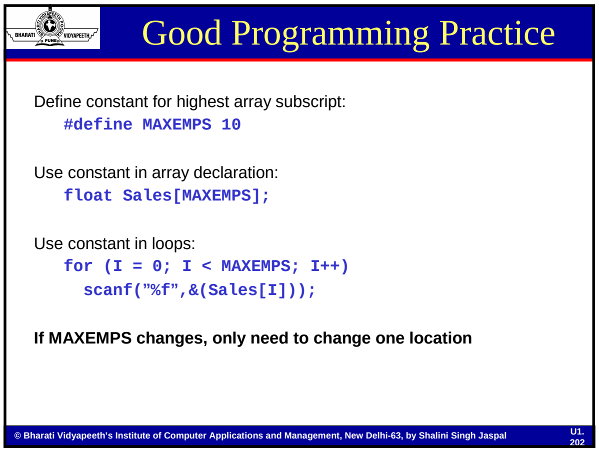
U1. 200



The Array Advantage

```
#include <stdio.h>
int main(void)
  {
  int i , x[5] , total = 0 ;
  for ( i = 0 ; i < 5 ; i++ )
  {
      printf( “Enter mark %d” , i );
      scanf ( “%d” , &x[ i ] );
  }
  for ( i = 0 ; i < 5 ; i++ )
      total = total + x[i];

  printf ( “The average is %d” , total / 5 );
  return 0;
  }
```

U1. 201



Good Programming Practice

Define constant for highest array subscript:

```
#define MAXEMPS 10
```

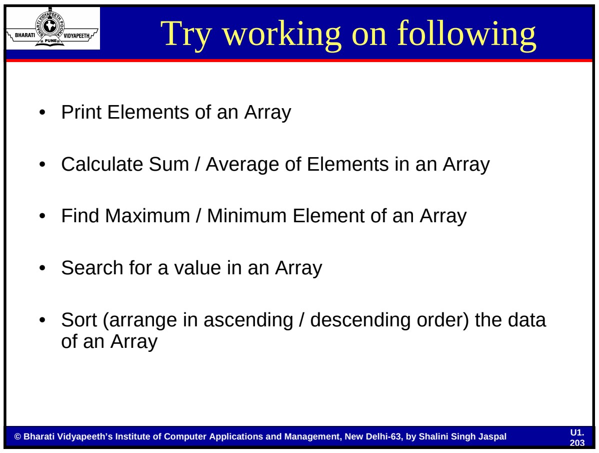
Use constant in array declaration:

```
float Sales[MAXEMPS];
```

Use constant in loops:

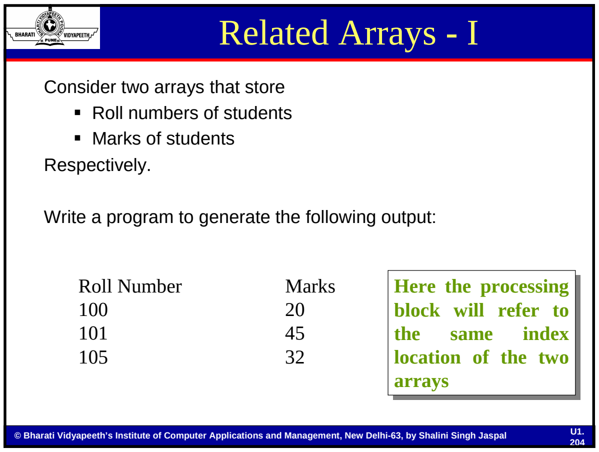
```
for (I = 0; I < MAXEMPS; I++)
  scanf(”%f”,&(Sales[I]));
```

If MAXEMPS changes, only need to change one location



Try working on following

- Print Elements of an Array
- Calculate Sum / Average of Elements in an Array
- Find Maximum / Minimum Element of an Array
- Search for a value in an Array
- Sort (arrange in ascending / descending order) the data of an Array



Related Arrays - I

Consider two arrays that store

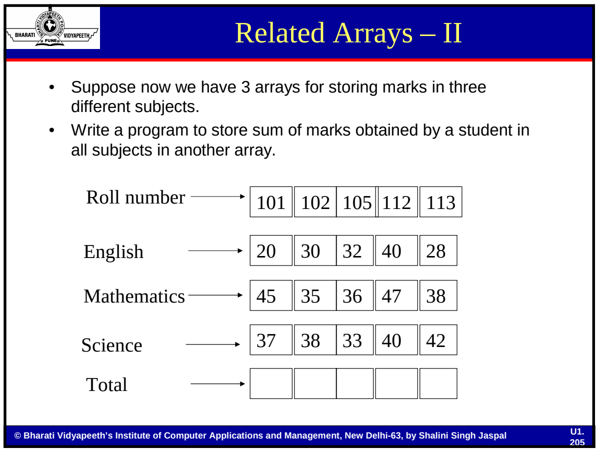
- Roll numbers of students
- Marks of students

Respectively.

Write a program to generate the following output:

```
Roll Number          Marks
100                  20
101                  45
105                  32
```

Here the processing block will refer to the same index location of the two arrays

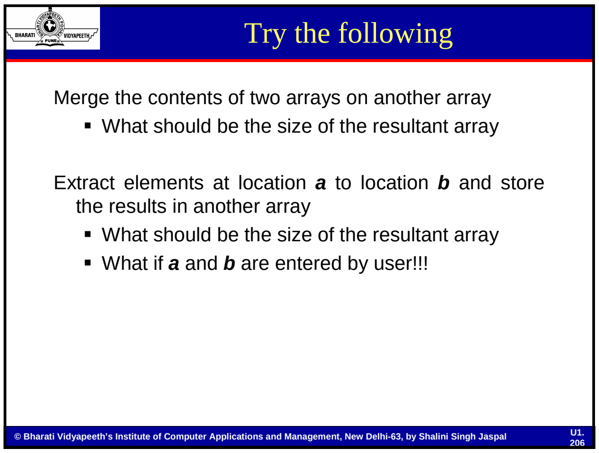


Related Arrays – II

- Suppose now we have 3 arrays for storing marks in three different subjects.
- Write a program to store sum of marks obtained by a student in all subjects in another array.

Roll number →	101	102	105	112	113
English →	20	30	32	40	28
Mathematics →	45	35	36	47	38
Science →	37	38	33	40	42
Total →					

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 205



Try the following

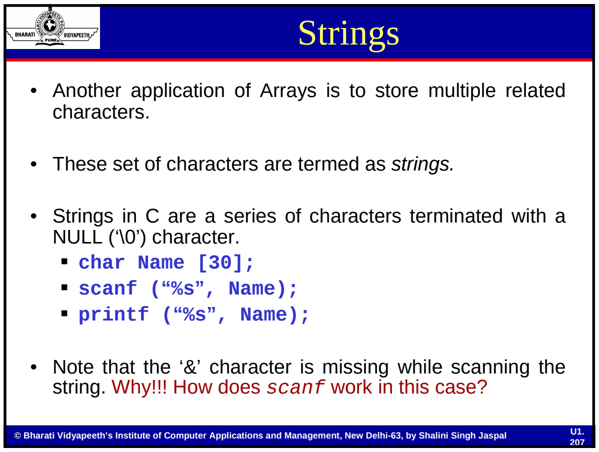
Merge the contents of two arrays on another array

- What should be the size of the resultant array

Extract elements at location ***a*** to location ***b*** and store the results in another array

- What should be the size of the resultant array
- What if ***a*** and ***b*** are entered by user!!!

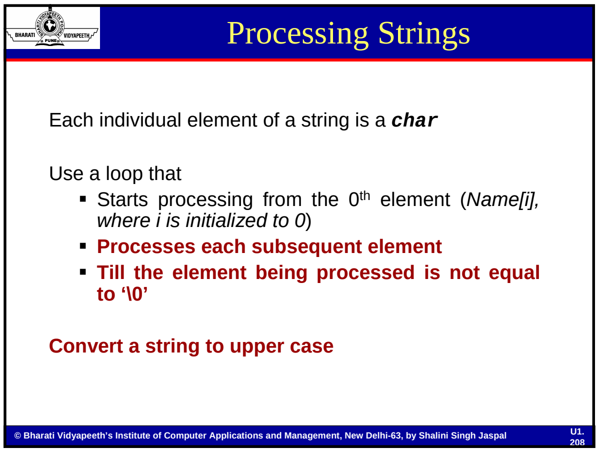
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 206



Strings

- Another application of Arrays is to store multiple related characters.
- These set of characters are termed as *strings.*
- Strings in C are a series of characters terminated with a NULL ('\0') character.
  - `char Name [30];`
  - `scanf ("%s", Name);`
  - `printf ("%s", Name);`
- Note that the '&' character is missing while scanning the string. Why!!! How does *`scanf`* work in this case?

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 207



Processing Strings

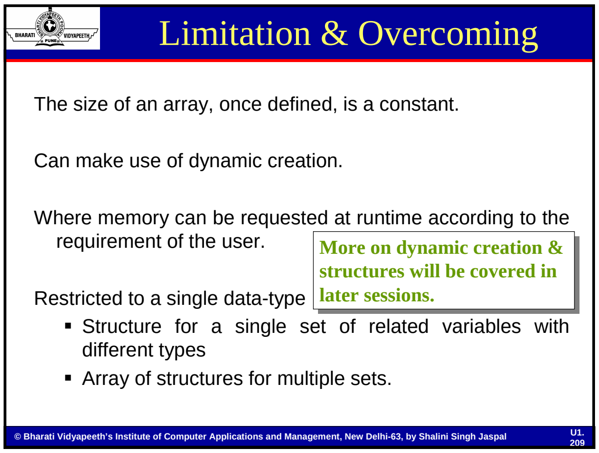
Each individual element of a string is a ***char***

Use a loop that

- Starts processing from the 0th element (*Name[i], where i is initialized to 0*)
- **Processes each subsequent element**
- **Till the element being processed is not equal to '\0'**

Convert a string to upper case

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1.208



Limitation & Overcoming

The size of an array, once defined, is a constant.

Can make use of dynamic creation.

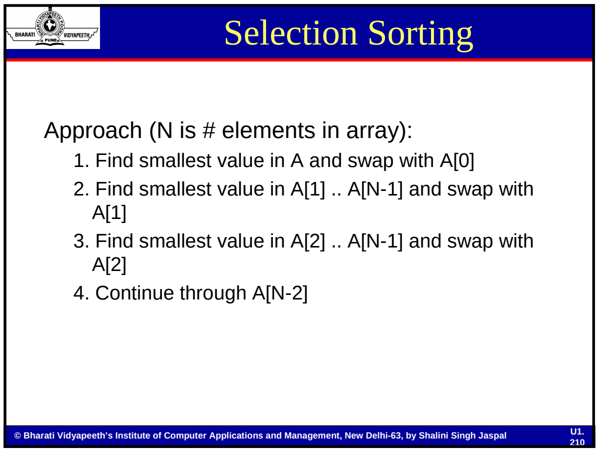
Where memory can be requested at runtime according to the requirement of the user.

Restricted to a single data-type

- Structure for a single set of related variables with different types
- Array of structures for multiple sets.

More on dynamic creation & structures will be covered in later sessions.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1.209

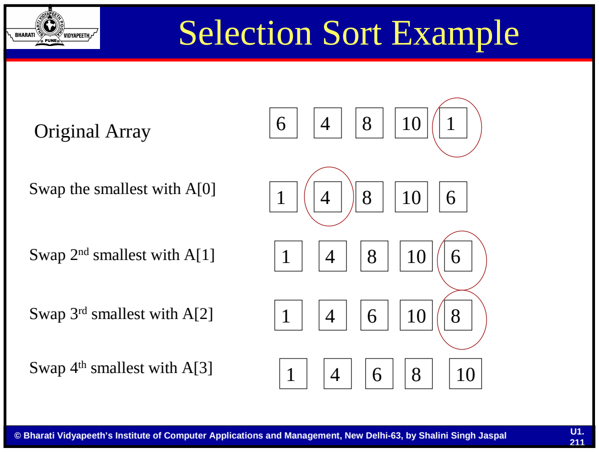


Selection Sorting

Approach (N is # elements in array):

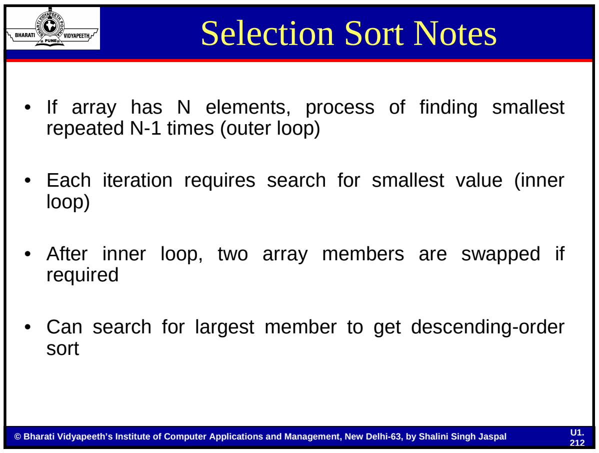
1. Find smallest value in A and swap with A[0]
2. Find smallest value in A[1] .. A[N-1] and swap with A[1]
3. Find smallest value in A[2] .. A[N-1] and swap with A[2]
4. Continue through A[N-2]

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1.210



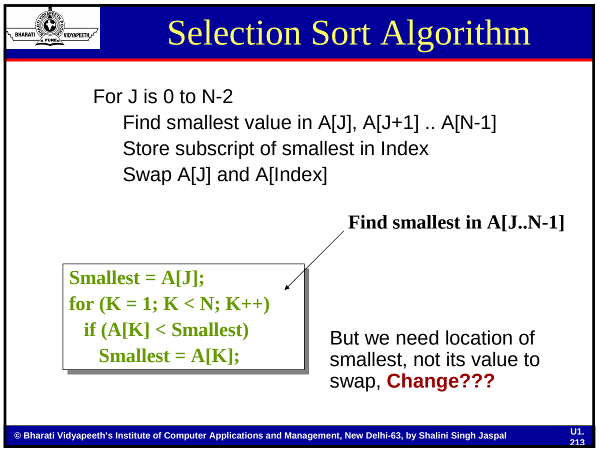
Selection Sort Example

Original Array	6	4	8	10	1
Swap the smallest with A[0]	1	4	8	10	6
Swap 2nd smallest with A[1]	1	4	8	10	6
Swap 3rd smallest with A[2]	1	4	6	10	8
Swap 4th smallest with A[3]	1	4	6	8	10



Selection Sort Notes

- If array has N elements, process of finding smallest repeated N-1 times (outer loop)
- Each iteration requires search for smallest value (inner loop)
- After inner loop, two array members are swapped if required
- Can search for largest member to get descending-order sort



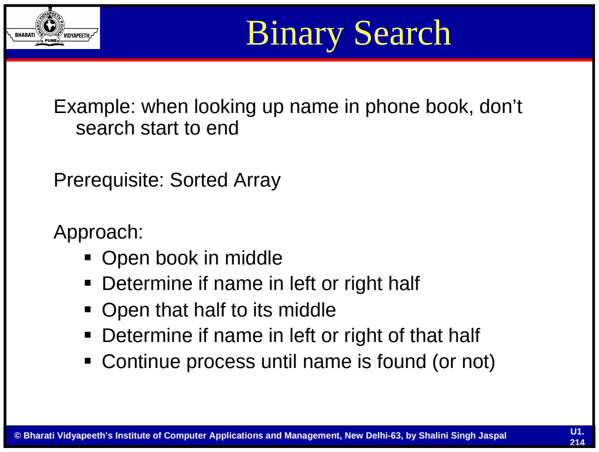
Selection Sort Algorithm

```
For J is 0 to N-2
    Find smallest value in A[J], A[J+1] .. A[N-1]
    Store subscript of smallest in Index
    Swap A[J] and A[Index]
```

Find smallest in A[J..N-1]

```
Smallest = A[J];
for (K = 1; K < N; K++)
   if (A[K] < Smallest)
      Smallest = A[K];
```

But we need location of smallest, not its value to swap, **Change???**



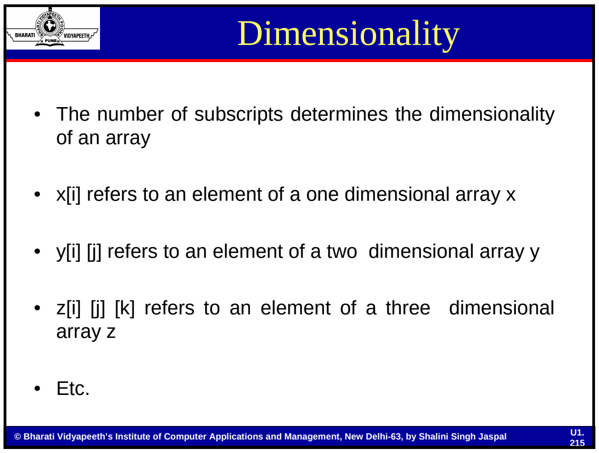
Binary Search

Example: when looking up name in phone book, don't search start to end

Prerequisite: Sorted Array

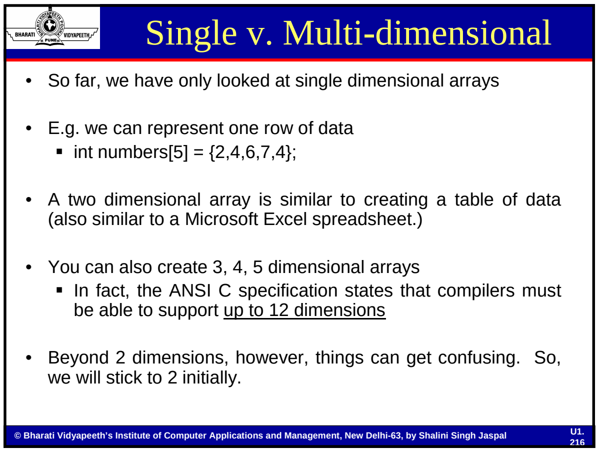
Approach:

- Open book in middle
- Determine if name in left or right half
- Open that half to its middle
- Determine if name in left or right of that half
- Continue process until name is found (or not)



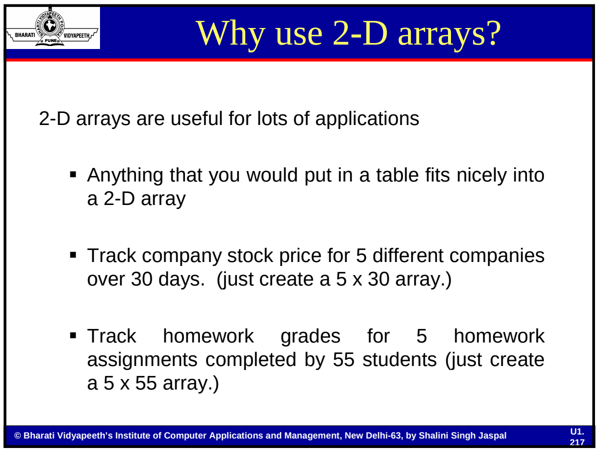
Dimensionality

- The number of subscripts determines the dimensionality of an array
- x[i] refers to an element of a one dimensional array x
- y[i] [j] refers to an element of a two dimensional array y
- z[i] [j] [k] refers to an element of a three dimensional array z
- Etc.



Single v. Multi-dimensional

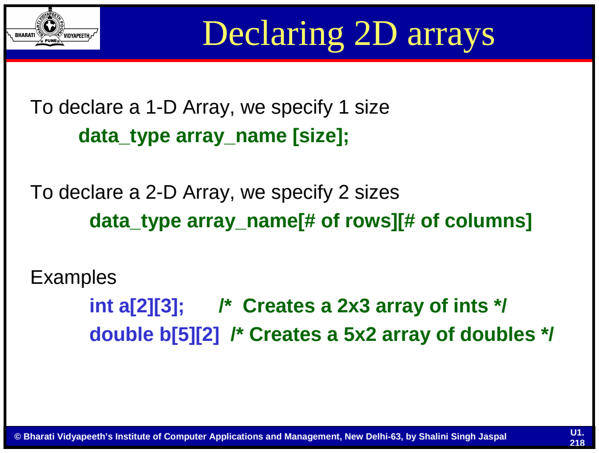
- So far, we have only looked at single dimensional arrays
- E.g. we can represent one row of data
  - int numbers[5] = {2,4,6,7,4};
- A two dimensional array is similar to creating a table of data (also similar to a Microsoft Excel spreadsheet.)
- You can also create 3, 4, 5 dimensional arrays
  - In fact, the ANSI C specification states that compilers must be able to support up to 12 dimensions
- Beyond 2 dimensions, however, things can get confusing. So, we will stick to 2 initially.



Why use 2-D arrays?

2-D arrays are useful for lots of applications

- Anything that you would put in a table fits nicely into a 2-D array
- Track company stock price for 5 different companies over 30 days. (just create a 5 x 30 array.)
- Track homework grades for 5 homework assignments completed by 55 students (just create a 5 x 55 array.)



Declaring 2D arrays

To declare a 1-D Array, we specify 1 size

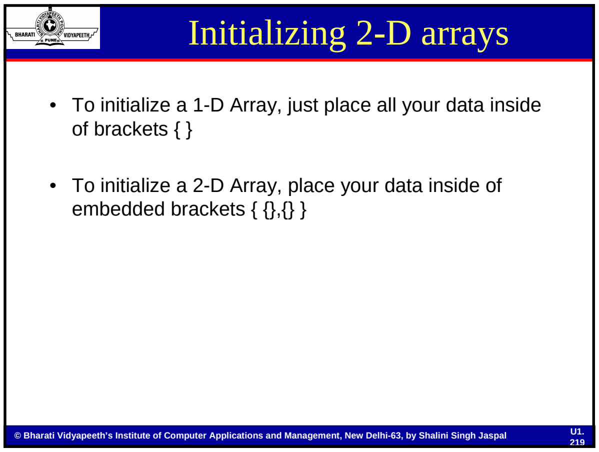
```
data_type array_name [size];
```

To declare a 2-D Array, we specify 2 sizes

```
data_type array_name[# of rows][# of columns]
```

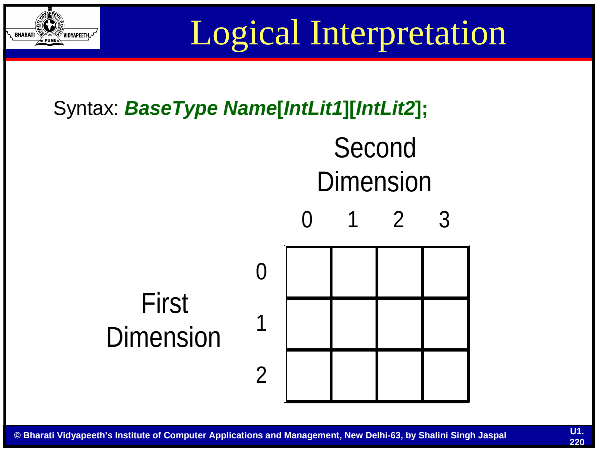
Examples

```
int a[2][3];      /*  Creates a 2x3 array of ints */
double b[5][2]  /* Creates a 5x2 array of doubles */
```



Initializing 2-D arrays

- To initialize a 1-D Array, just place all your data inside of brackets { }
- To initialize a 2-D Array, place your data inside of embedded brackets { {},{} }



Logical Interpretation

Syntax: ***BaseType Name*[*IntLit1*][*IntLit2*];**

Second Dimension

0 1 2 3

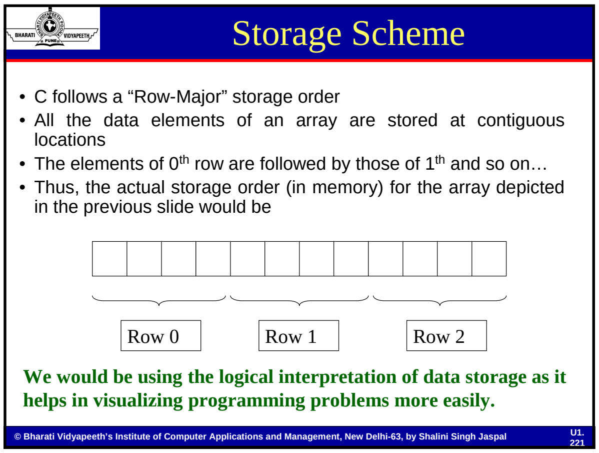
First Dimension

0

1

2

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 220



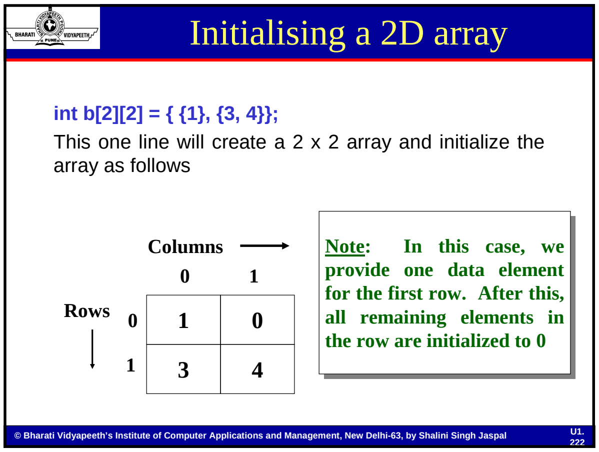
Storage Scheme

- C follows a "Row-Major" storage order
- All the data elements of an array are stored at contiguous locations
- The elements of 0th row are followed by those of 1th and so on…
- Thus, the actual storage order (in memory) for the array depicted in the previous slide would be

Row 0 Row 1 Row 2

We would be using the logical interpretation of data storage as it helps in visualizing programming problems more easily.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 221



Initialising a 2D array

int b[2][2] = { {1}, {3, 4}};

This one line will create a 2 x 2 array and initialize the array as follows

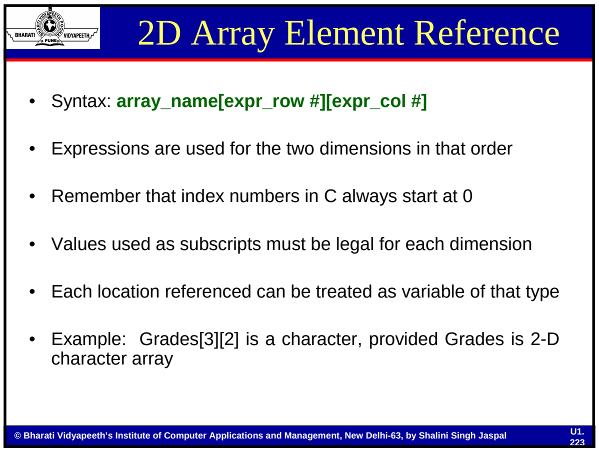
Columns →

Rows ↓

	0	**1**
0	**1**	**0**
1	**3**	**4**

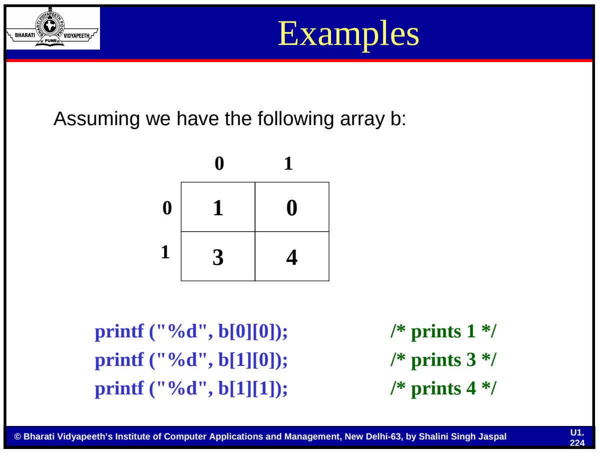
Note: In this case, we provide one data element for the first row. After this, all remaining elements in the row are initialized to 0

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 222



2D Array Element Reference

- Syntax: **array_name[expr_row #][expr_col #]**
- Expressions are used for the two dimensions in that order
- Remember that index numbers in C always start at 0
- Values used as subscripts must be legal for each dimension
- Each location referenced can be treated as variable of that type
- Example: Grades[3][2] is a character, provided Grades is 2-D character array

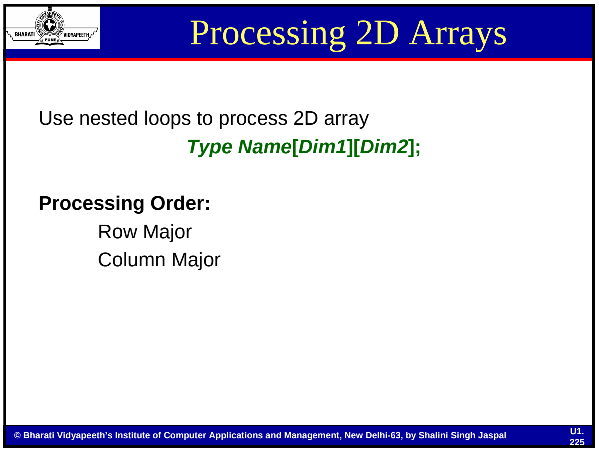


Examples

Assuming we have the following array b:

	0	1
0	1	0
1	3	4

```
printf ("%d", b[0][0]);          /* prints 1 */
printf ("%d", b[1][0]);          /* prints 3 */
printf ("%d", b[1][1]);          /* prints 4 */
```



Processing 2D Arrays

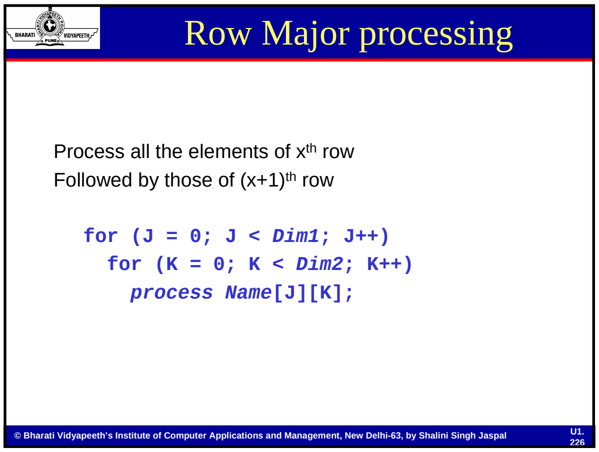
Use nested loops to process 2D array

***Type Name*[*Dim1*][*Dim2*];**

Processing Order:

Row Major

Column Major

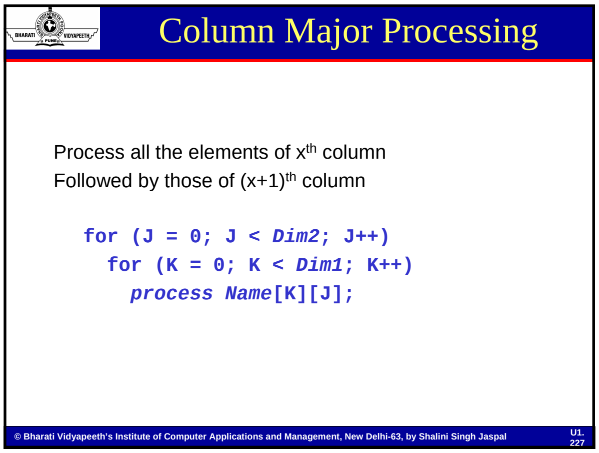


Row Major processing

Process all the elements of x^th row
Followed by those of (x+1)^th row

```
for (J = 0; J < Dim1; J++)
  for (K = 0; K < Dim2; K++)
    process Name[J][K];
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 226

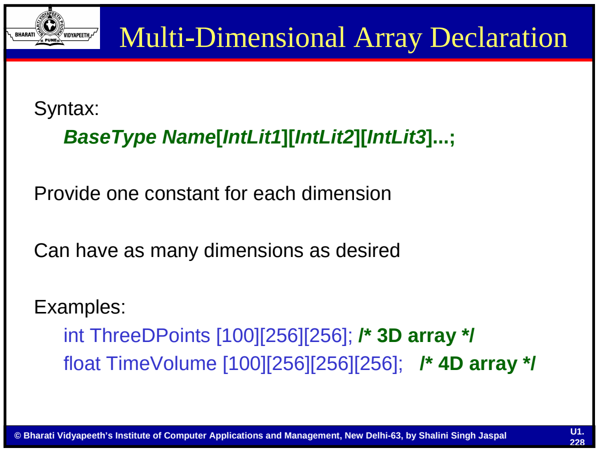


Column Major Processing

Process all the elements of x^th column
Followed by those of (x+1)^th column

```
for (J = 0; J < Dim2; J++)
  for (K = 0; K < Dim1; K++)
    process Name[K][J];
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 227



Multi-Dimensional Array Declaration

Syntax:

***BaseType Name*[*IntLit1*][*IntLit2*][*IntLit3*]...;**

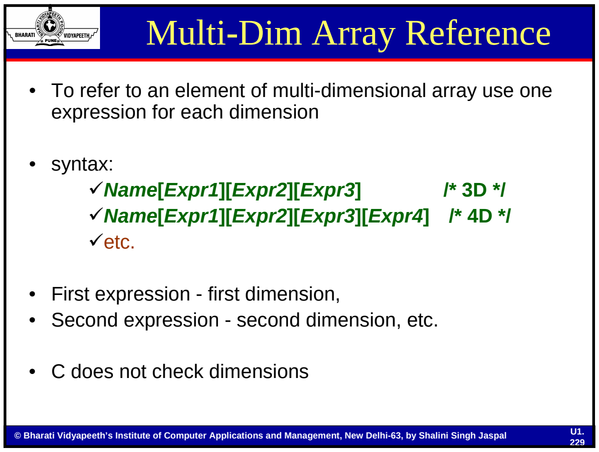
Provide one constant for each dimension

Can have as many dimensions as desired

Examples:

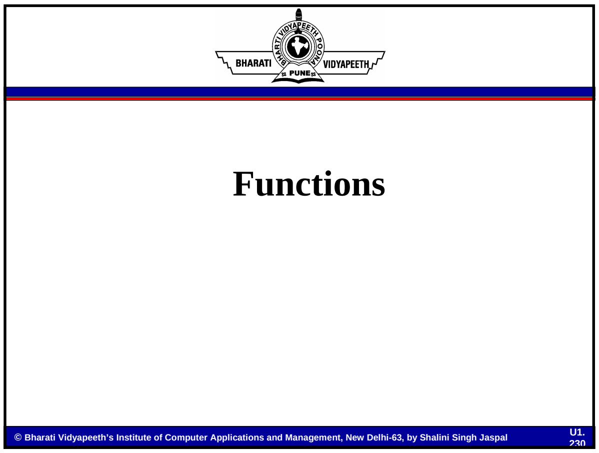
```
int ThreeDPoints [100][256][256]; /* 3D array */
float TimeVolume [100][256][256][256];   /* 4D array */
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 228

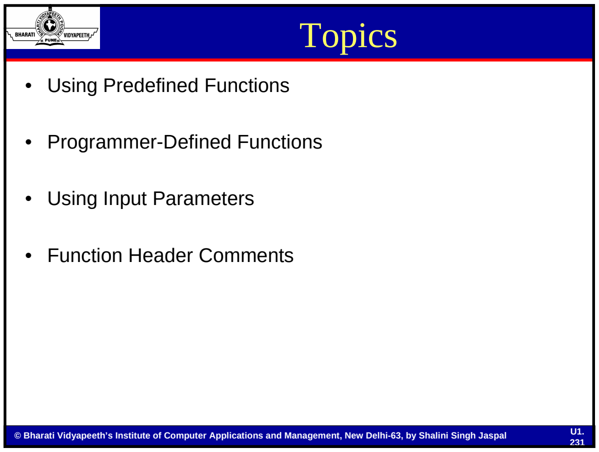


Multi-Dim Array Reference

- To refer to an element of multi-dimensional array use one expression for each dimension
- syntax:
  - ✓***Name[Expr1][Expr2][Expr3]*** **/* 3D */**
  - ✓***Name[Expr1][Expr2][Expr3][Expr4]*** **/* 4D */**
  - ✓etc.
- First expression - first dimension,
- Second expression - second dimension, etc.
- C does not check dimensions

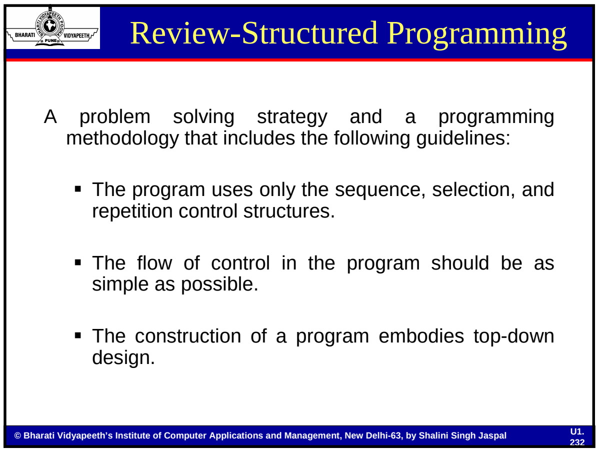


Functions



Topics

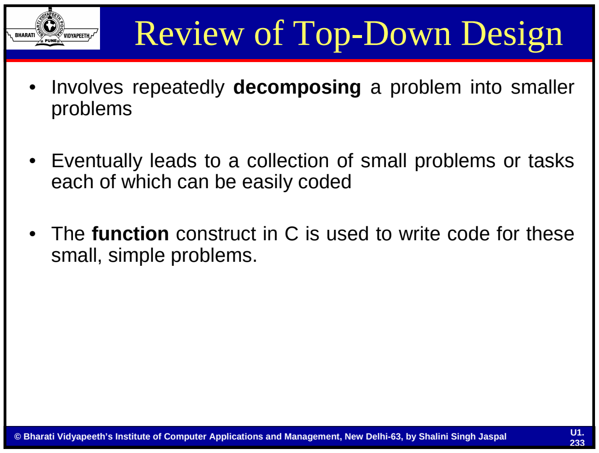
- Using Predefined Functions
- Programmer-Defined Functions
- Using Input Parameters
- Function Header Comments



Review-Structured Programming

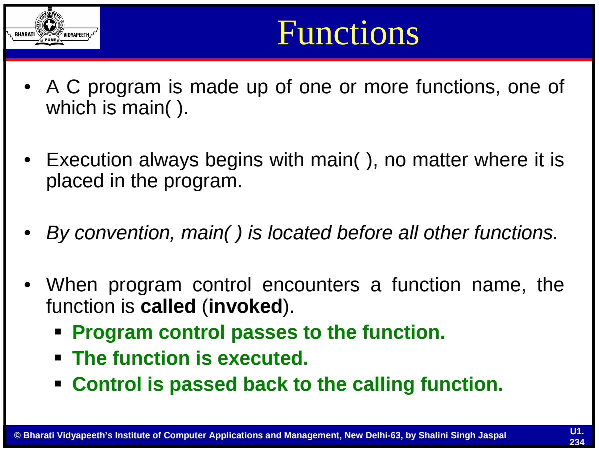
A problem solving strategy and a programming methodology that includes the following guidelines:

- The program uses only the sequence, selection, and repetition control structures.
- The flow of control in the program should be as simple as possible.
- The construction of a program embodies top-down design.



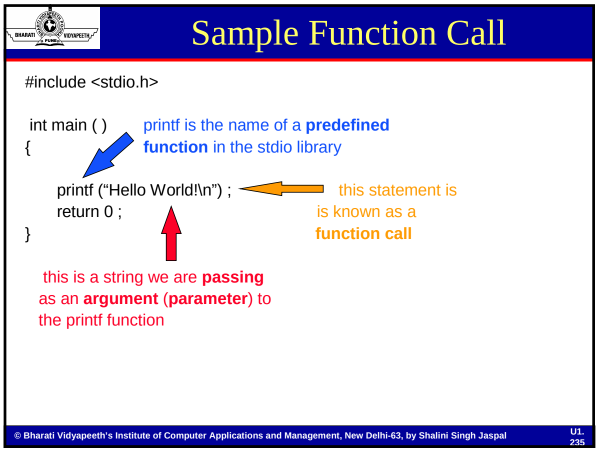
Review of Top-Down Design

- Involves repeatedly **decomposing** a problem into smaller problems
- Eventually leads to a collection of small problems or tasks each of which can be easily coded
- The **function** construct in C is used to write code for these small, simple problems.



Functions

- A C program is made up of one or more functions, one of which is main( ).
- Execution always begins with main( ), no matter where it is placed in the program.
- *By convention, main( ) is located before all other functions.*
- When program control encounters a function name, the function is **called** (**invoked**).
  - **Program control passes to the function.**
  - **The function is executed.**
  - **Control is passed back to the calling function.**



Sample Function Call

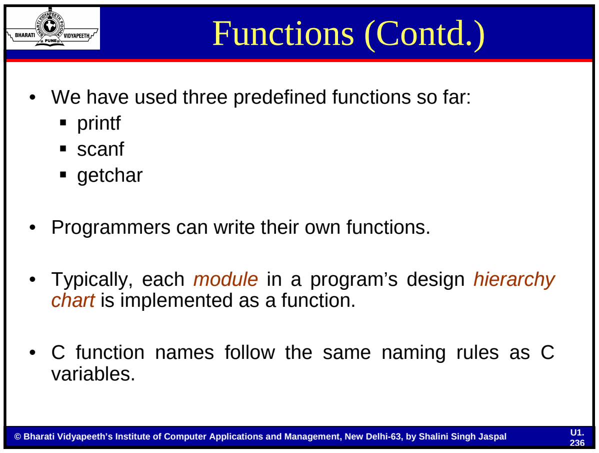
```
#include <stdio.h>

 int main ( )
{

     printf ("Hello World!\n") ;
     return 0 ;
}
```

printf is the name of a **predefined function** in the stdio library

this statement is is known as a **function call**

this is a string we are **passing** as an **argument** (**parameter**) to the printf function

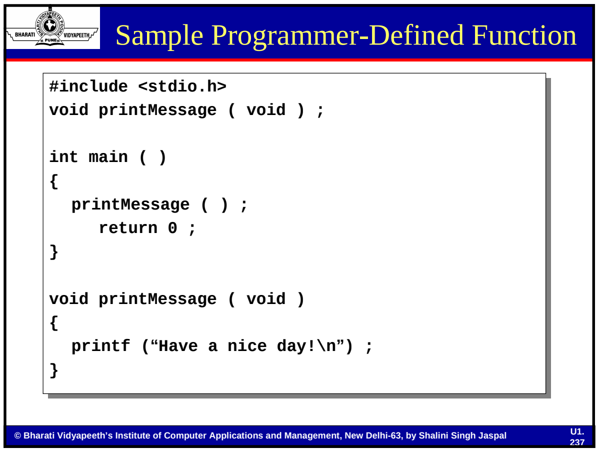
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 235



Functions (Contd.)

- We have used three predefined functions so far:
  - printf
  - scanf
  - getchar
- Programmers can write their own functions.
- Typically, each *module* in a program's design *hierarchy chart* is implemented as a function.
- C function names follow the same naming rules as C variables.

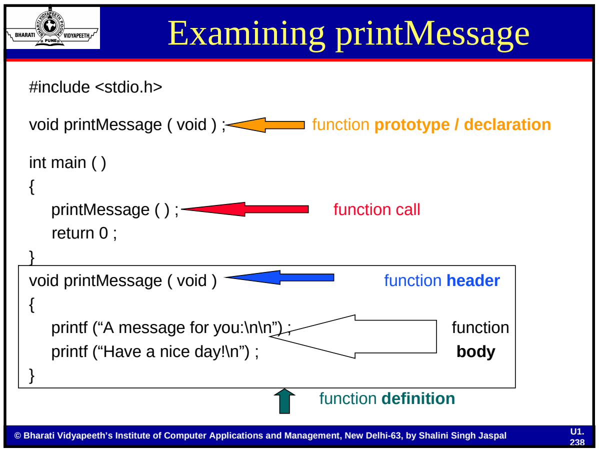
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 236



Sample Programmer-Defined Function

```
#include <stdio.h>
void printMessage ( void ) ;

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

void printMessage ( void )
{
   printf ("Have a nice day!\n") ;
}
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 237



Examining printMessage

```
#include <stdio.h>

void printMessage ( void ) ;

int main ( )
{
    printMessage ( ) ;
    return 0 ;
}
void printMessage ( void )
{
    printf (“A message for you:\n\n”) ;
    printf (“Have a nice day!\n”) ;
}
```

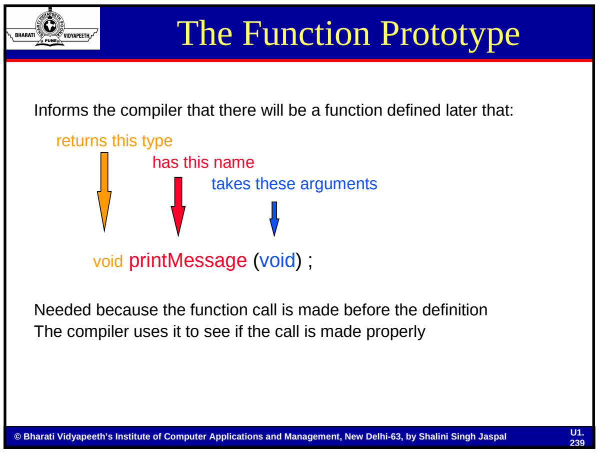
function **prototype / declaration**

function call

function **header**

function **body**

function **definition**



The Function Prototype

Informs the compiler that there will be a function defined later that:

returns this type

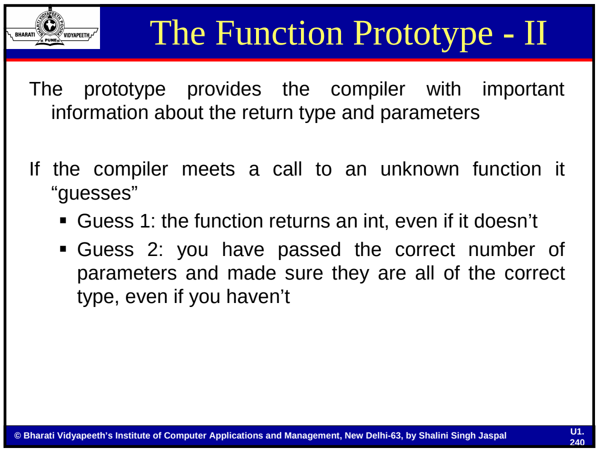
has this name

takes these arguments

void printMessage (void) ;

Needed because the function call is made before the definition

The compiler uses it to see if the call is made properly

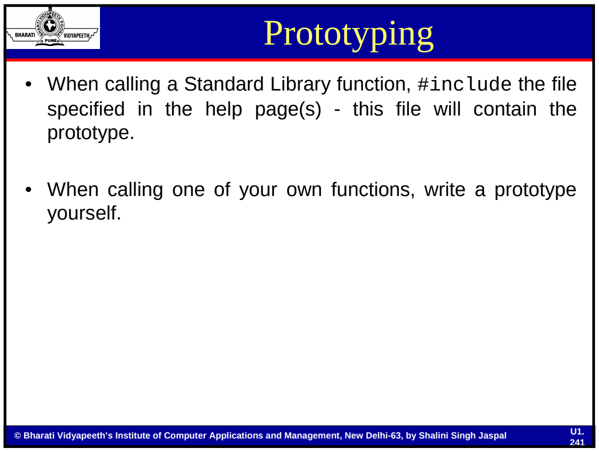


The Function Prototype - II

The prototype provides the compiler with important information about the return type and parameters

If the compiler meets a call to an unknown function it “guesses”

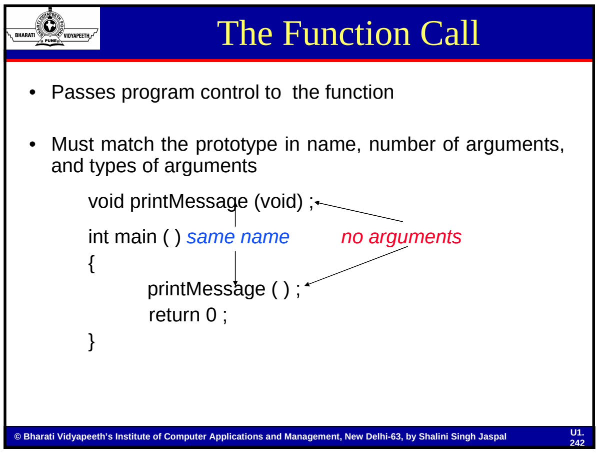
- Guess 1: the function returns an int, even if it doesn’t
- Guess 2: you have passed the correct number of parameters and made sure they are all of the correct type, even if you haven’t



Prototyping

- When calling a Standard Library function, `#include` the file specified in the help page(s) - this file will contain the prototype.
- When calling one of your own functions, write a prototype yourself.

U1.241

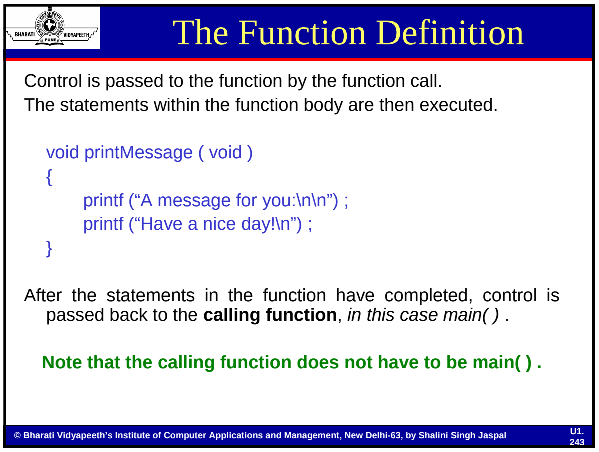


The Function Call

- Passes program control to the function
- Must match the prototype in name, number of arguments, and types of arguments

```
void printMessage (void) ;

int main ( ) same name          no arguments
{
        printMessage ( ) ;
        return 0 ;
}
```

U1.242



The Function Definition

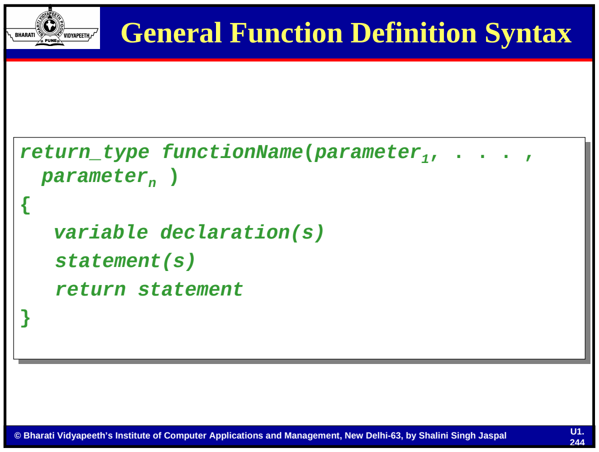
Control is passed to the function by the function call.
The statements within the function body are then executed.

```
void printMessage ( void )
{
     printf (“A message for you:\n\n”) ;
     printf (“Have a nice day!\n”) ;
}
```

After the statements in the function have completed, control is passed back to the **calling function**, *in this case main( )* .

Note that the calling function does not have to be main( ) .

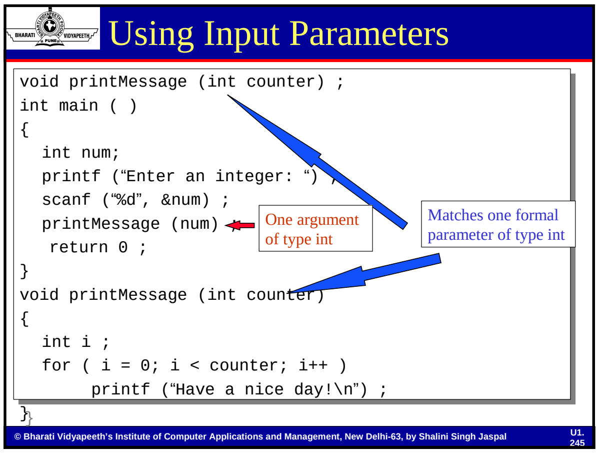
U1.243



General Function Definition Syntax

```
return_type functionName(parameter_1, . . . ,
  parameter_n )
{
   variable declaration(s)
   statement(s)
   return statement
}
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 244



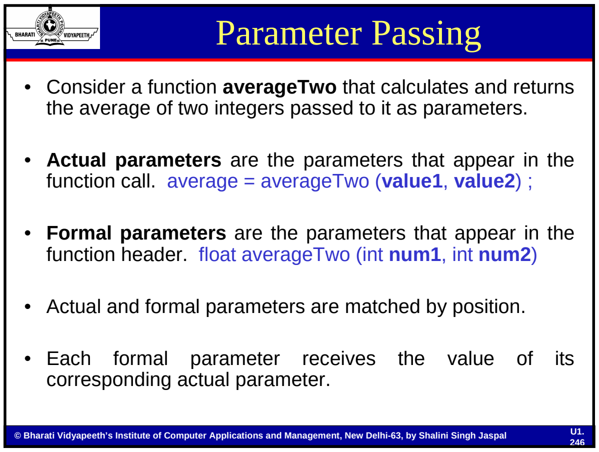
Using Input Parameters

```
void printMessage (int counter) ;
int main ( )
{
  int num;
  printf (“Enter an integer: “) ;
  scanf (“%d”, &num) ;
  printMessage (num) ;
   return 0 ;
}
void printMessage (int counter)
{
  int i ;
  for ( i = 0; i < counter; i++ )
       printf (“Have a nice day!\n”) ;
}
```

One argument of type int

Matches one formal parameter of type int

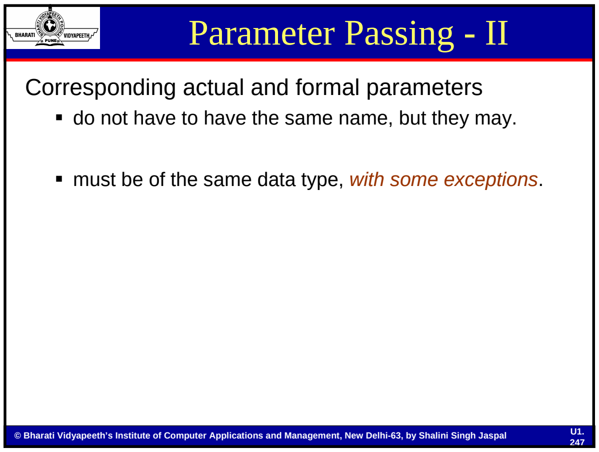
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 245



Parameter Passing

- Consider a function **averageTwo** that calculates and returns the average of two integers passed to it as parameters.
- **Actual parameters** are the parameters that appear in the function call. average = averageTwo (**value1**, **value2**) ;
- **Formal parameters** are the parameters that appear in the function header. float averageTwo (int **num1**, int **num2**)
- Actual and formal parameters are matched by position.
- Each formal parameter receives the value of its corresponding actual parameter.

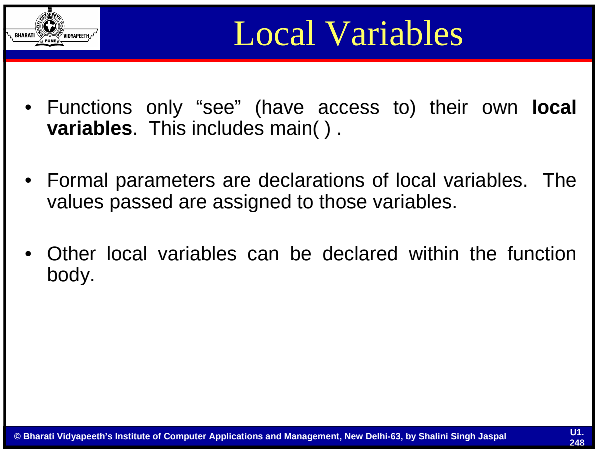
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 246



Parameter Passing - II

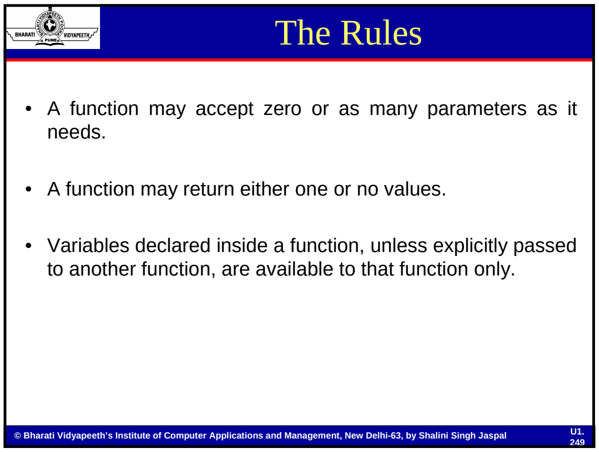
Corresponding actual and formal parameters

- do not have to have the same name, but they may.
- must be of the same data type, *with some exceptions*.



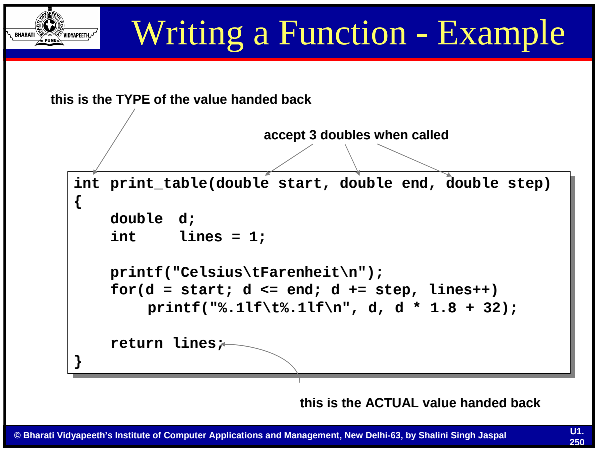
Local Variables

- Functions only "see" (have access to) their own **local variables**. This includes main( ) .
- Formal parameters are declarations of local variables. The values passed are assigned to those variables.
- Other local variables can be declared within the function body.



The Rules

- A function may accept zero or as many parameters as it needs.
- A function may return either one or no values.
- Variables declared inside a function, unless explicitly passed to another function, are available to that function only.



Writing a Function - Example

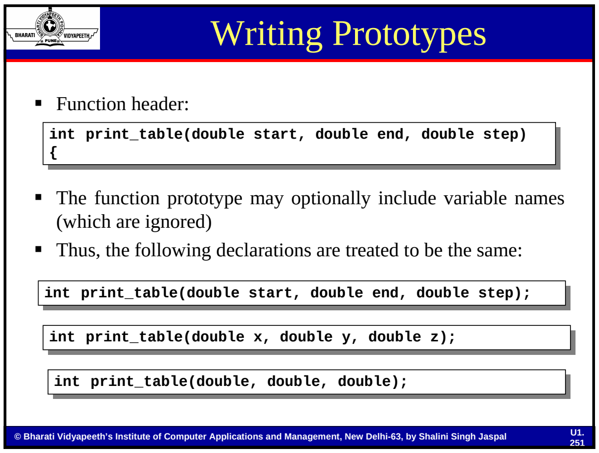
this is the TYPE of the value handed back

accept 3 doubles when called

```
int print_table(double start, double end, double step)
{
    double  d;
    int     lines = 1;

    printf("Celsius\tFarenheit\n");
    for(d = start; d <= end; d += step, lines++)
        printf("%.1lf\t%.1lf\n", d, d * 1.8 + 32);

    return lines;
}
```

this is the ACTUAL value handed back

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 250



Writing Prototypes

- Function header:

```
int print_table(double start, double end, double step)
{
```

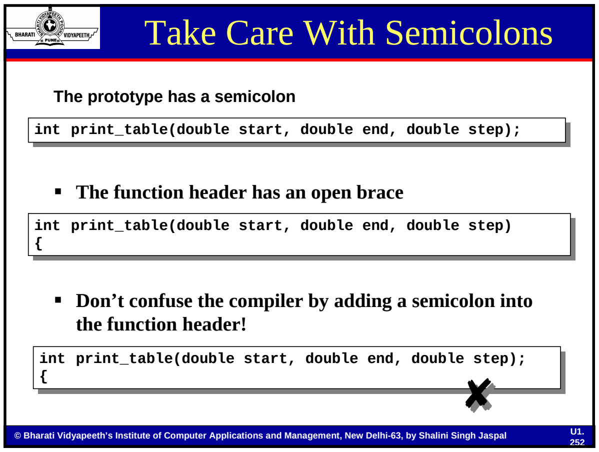
- The function prototype may optionally include variable names (which are ignored)
- Thus, the following declarations are treated to be the same:

```
int print_table(double start, double end, double step);
```

```
int print_table(double x, double y, double z);
```

```
int print_table(double, double, double);
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 251



Take Care With Semicolons

The prototype has a semicolon

```
int print_table(double start, double end, double step);
```

- **The function header has an open brace**

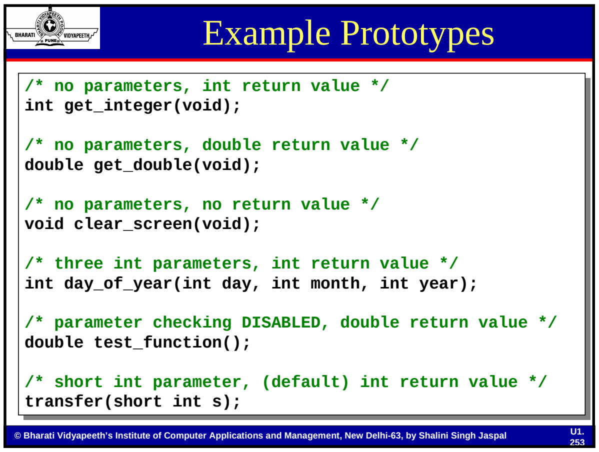
```
int print_table(double start, double end, double step)
{
```

- **Don’t confuse the compiler by adding a semicolon into the function header!**

```
int print_table(double start, double end, double step);
{
```

✘

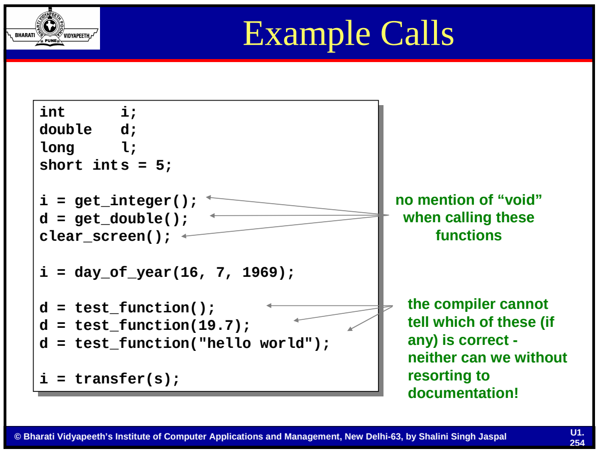
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 252



Example Prototypes

```
/* no parameters, int return value */
int get_integer(void);

/* no parameters, double return value */
double get_double(void);

/* no parameters, no return value */
void clear_screen(void);

/* three int parameters, int return value */
int day_of_year(int day, int month, int year);

/* parameter checking DISABLED, double return value */
double test_function();

/* short int parameter, (default) int return value */
transfer(short int s);
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 253



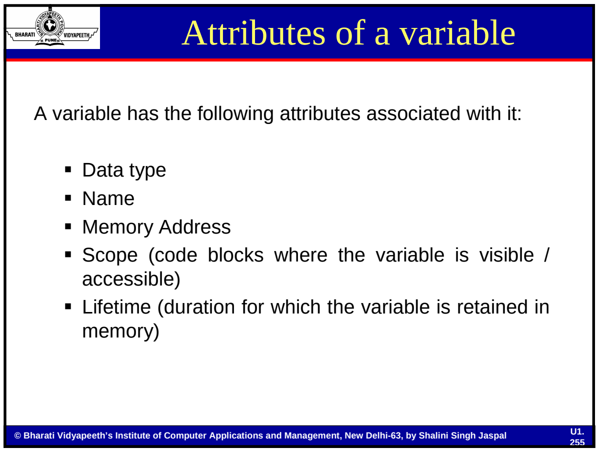
Example Calls

```
int      i;
double   d;
long     l;
short ints = 5;

i = get_integer();
d = get_double();
clear_screen();

i = day_of_year(16, 7, 1969);

d = test_function();
d = test_function(19.7);
d = test_function("hello world");

i = transfer(s);
```

no mention of “void” when calling these functions

the compiler cannot tell which of these (if any) is correct - neither can we without resorting to documentation!

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 254

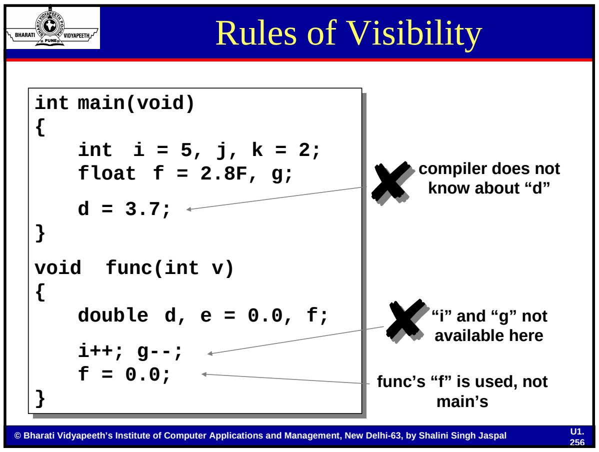


Attributes of a variable

A variable has the following attributes associated with it:

- Data type
- Name
- Memory Address
- Scope (code blocks where the variable is visible / accessible)
- Lifetime (duration for which the variable is retained in memory)

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 255



Rules of Visibility

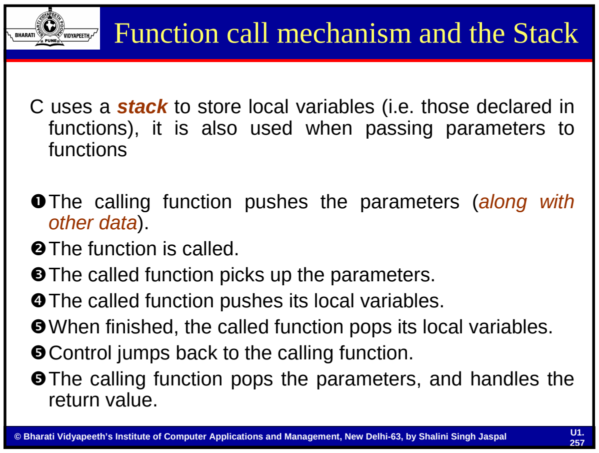
```
int main(void)
{
    int  i = 5, j, k = 2;
    float  f = 2.8F, g;

    d = 3.7;
}
void  func(int v)
{
    double  d, e = 0.0, f;

    i++; g--;
    f = 0.0;
}
```

compiler does not know about “d”

“i” and “g” not available here

func’s “f” is used, not main’s

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 256

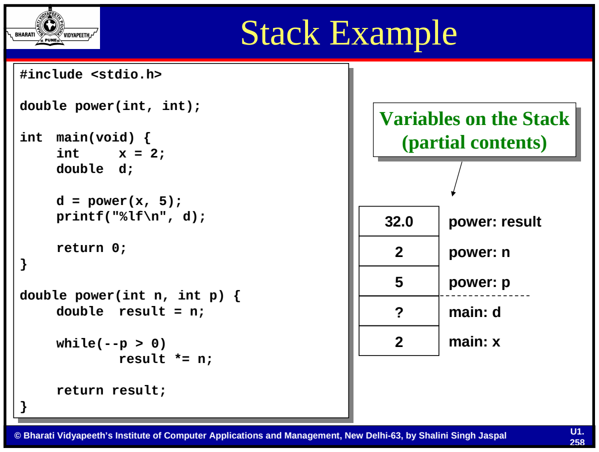


Function call mechanism and the Stack

C uses a ***stack*** to store local variables (i.e. those declared in functions), it is also used when passing parameters to functions

❶The calling function pushes the parameters (*along with other data*).
❷The function is called.
❸The called function picks up the parameters.
❹The called function pushes its local variables.
❺When finished, the called function pops its local variables.
❺Control jumps back to the calling function.
❺The calling function pops the parameters, and handles the return value.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 257



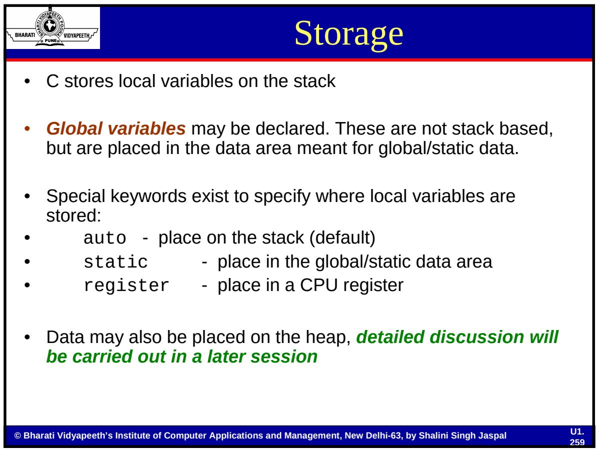
Stack Example

```
#include <stdio.h>

double power(int, int);

int  main(void) {
     int     x = 2;
     double  d;

     d = power(x, 5);
     printf("%lf\n", d);

     return 0;
}

double power(int n, int p) {
     double  result = n;

     while(--p > 0)
             result *= n;

     return result;
}
```

Variables on the Stack (partial contents)

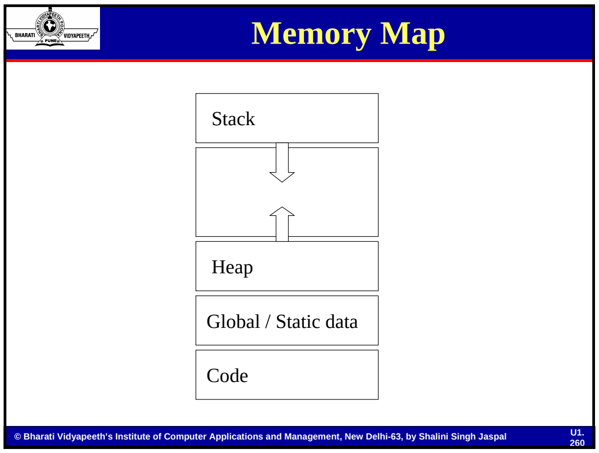
32.0	power: result
2	power: n
5	power: p
?	main: d
2	main: x

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 258

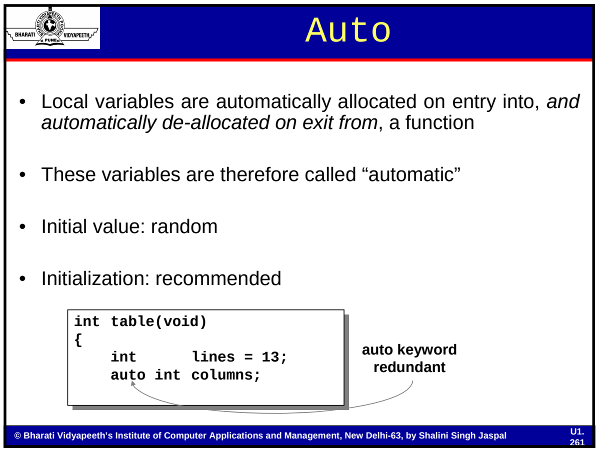


Storage

- C stores local variables on the stack
- ***Global variables*** may be declared. These are not stack based, but are placed in the data area meant for global/static data.
- Special keywords exist to specify where local variables are stored:
- `auto` - place on the stack (default)
- `static` - place in the global/static data area
- `register` - place in a CPU register
- Data may also be placed on the heap, ***detailed discussion will be carried out in a later session***



Memory Map

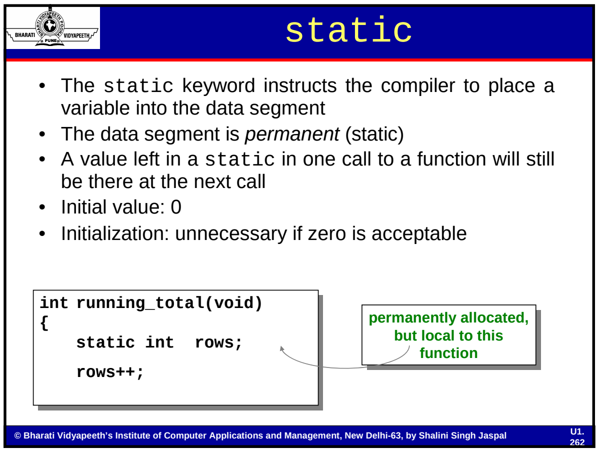


Auto

- Local variables are automatically allocated on entry into, *and automatically de-allocated on exit from*, a function
- These variables are therefore called "automatic"
- Initial value: random
- Initialization: recommended

```
int table(void)
{
    int      lines = 13;
    auto int columns;
```

auto keyword redundant

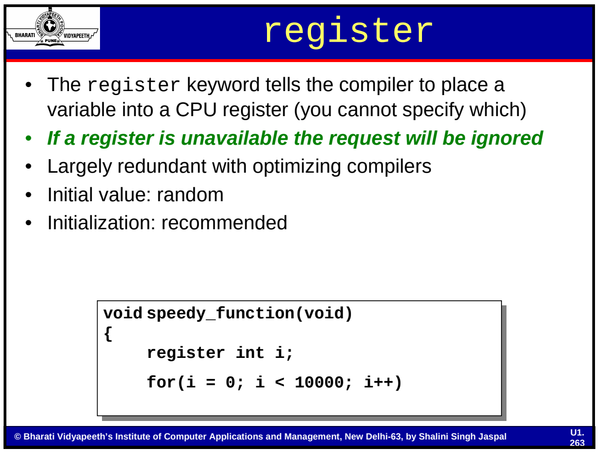


static

- The `static` keyword instructs the compiler to place a variable into the data segment
- The data segment is *permanent* (static)
- A value left in a `static` in one call to a function will still be there at the next call
- Initial value: 0
- Initialization: unnecessary if zero is acceptable

```
int running_total(void)
{
    static int  rows;

    rows++;
```

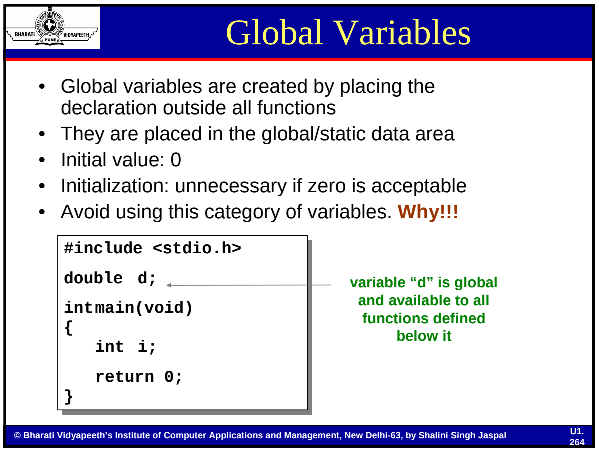
permanently allocated, but local to this function



register

- The `register` keyword tells the compiler to place a variable into a CPU register (you cannot specify which)
- ***If a register is unavailable the request will be ignored***
- Largely redundant with optimizing compilers
- Initial value: random
- Initialization: recommended

```
void speedy_function(void)
{
     register int i;

     for(i = 0; i < 10000; i++)
```

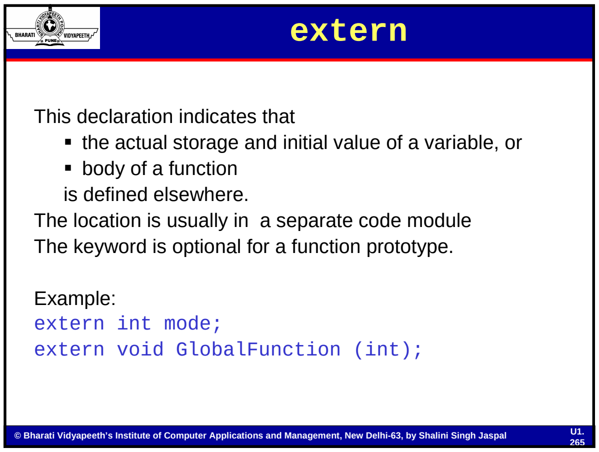


Global Variables

- Global variables are created by placing the declaration outside all functions
- They are placed in the global/static data area
- Initial value: 0
- Initialization: unnecessary if zero is acceptable
- Avoid using this category of variables. **Why!!!**

```
#include <stdio.h>

double  d;

intmain(void)
{
    int  i;

    return 0;
}
```

variable “d” is global and available to all functions defined below it



extern

This declaration indicates that

- the actual storage and initial value of a variable, or
- body of a function

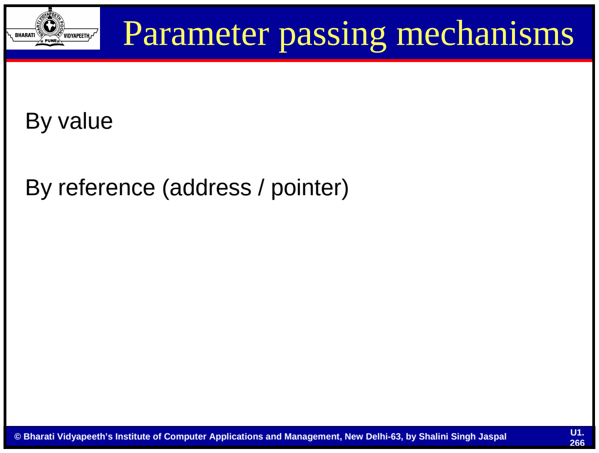
is defined elsewhere.

The location is usually in a separate code module

The keyword is optional for a function prototype.

Example:

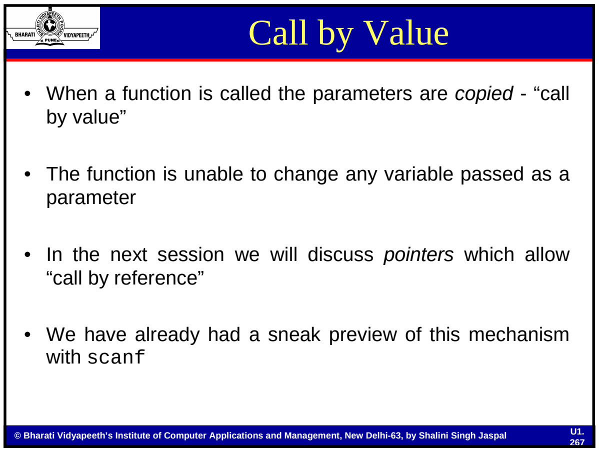
```
extern int mode;
extern void GlobalFunction (int);
```



Parameter passing mechanisms

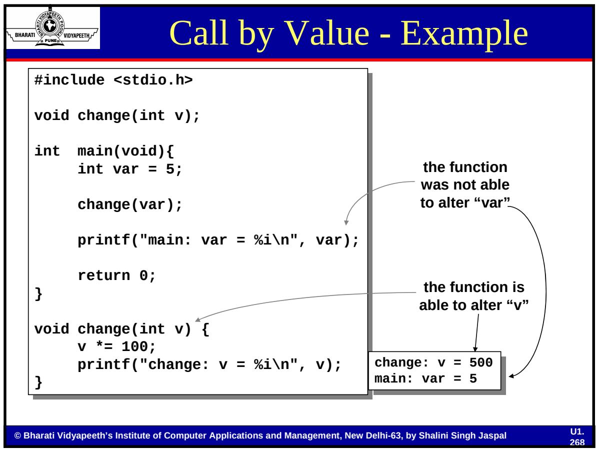
By value

By reference (address / pointer)



Call by Value

- When a function is called the parameters are *copied* - “call by value”
- The function is unable to change any variable passed as a parameter
- In the next session we will discuss *pointers* which allow “call by reference”
- We have already had a sneak preview of this mechanism with `scanf`



Call by Value - Example

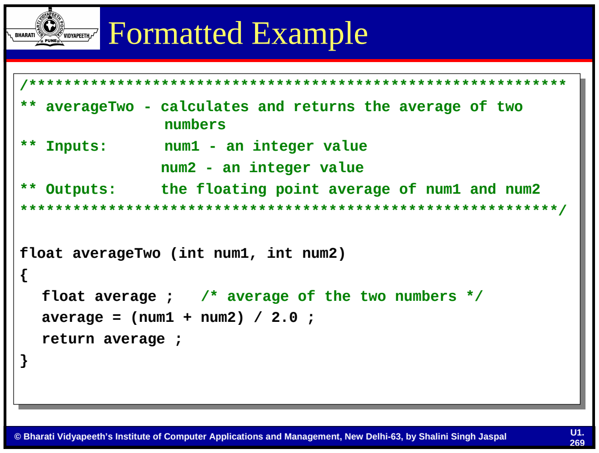
```
#include <stdio.h>

void change(int v);

int  main(void){
     int var = 5;

     change(var);

     printf("main: var = %i\n", var);

     return 0;
}

void change(int v) {
     v *= 100;
     printf("change: v = %i\n", v);
}
```

the function was not able to alter “var”

the function is able to alter “v”

```
change: v = 500
main: var = 5
```

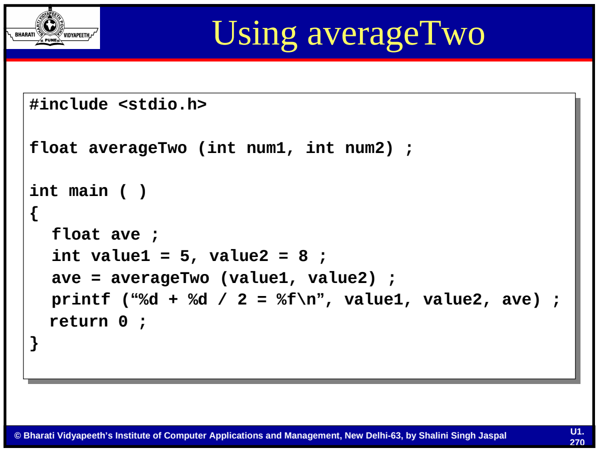
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 268



Formatted Example

```
/**************************************************************
** averageTwo - calculates and returns the average of two
                numbers
** Inputs:      num1 - an integer value
                num2 - an integer value
** Outputs:     the floating point average of num1 and num2
**************************************************************/

float averageTwo (int num1, int num2)
{
  float average ;   /* average of the two numbers */
  average = (num1 + num2) / 2.0 ;
  return average ;
}
```

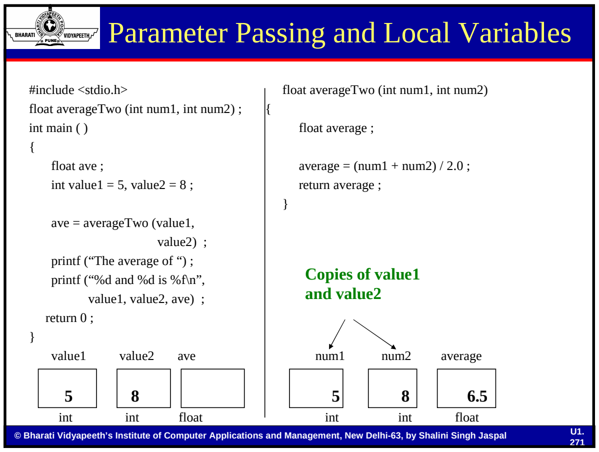
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 269



Using averageTwo

```
#include <stdio.h>

float averageTwo (int num1, int num2) ;

int main ( )
{
  float ave ;
  int value1 = 5, value2 = 8 ;
  ave = averageTwo (value1, value2) ;
  printf (“%d + %d / 2 = %f\n”, value1, value2, ave) ;
  return 0 ;
}
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 270



Parameter Passing and Local Variables

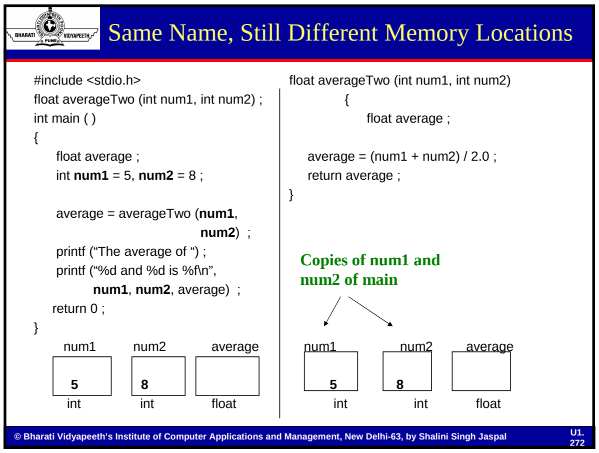
```
#include <stdio.h>
float averageTwo (int num1, int num2) ;
int main ( )
{
    float ave ;
    int value1 = 5, value2 = 8 ;

    ave = averageTwo (value1,
                      value2)  ;
    printf ("The average of ") ;
    printf ("%d and %d is %f\n",
            value1, value2, ave)  ;
   return 0 ;
}
```

```
float averageTwo (int num1, int num2)
{
    float average ;

    average = (num1 + num2) / 2.0 ;
    return average ;
}
```

Copies of value1 and value2

value1	value2	ave
5	8	
int	int	float

num1	num2	average
5	8	6.5
int	int	float



Same Name, Still Different Memory Locations

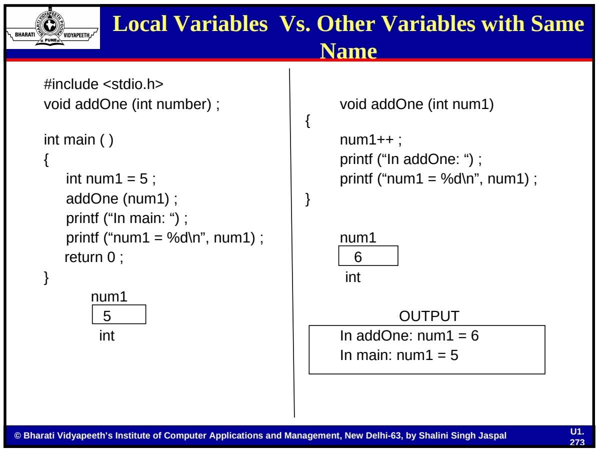
```
#include <stdio.h>
float averageTwo (int num1, int num2) ;
int main ( )
{
    float average ;
    int num1 = 5, num2 = 8 ;

    average = averageTwo (num1,
                          num2)  ;
    printf ("The average of ") ;
    printf ("%d and %d is %f\n",
            num1, num2, average)  ;
   return 0 ;
}
```

```
float averageTwo (int num1, int num2)
        {
            float average ;

    average = (num1 + num2) / 2.0 ;
    return average ;
}
```

Copies of num1 and num2 of main

num1	num2	average
5	8	
int	int	float

num1	num2	average
5	8	
int	int	float



Local Variables Vs. Other Variables with Same Name

```
#include <stdio.h>
void addOne (int number) ;

int main ( )
{
    int num1 = 5 ;
    addOne (num1) ;
    printf ("In main: ") ;
    printf ("num1 = %d\n", num1) ;
   return 0 ;
}
```

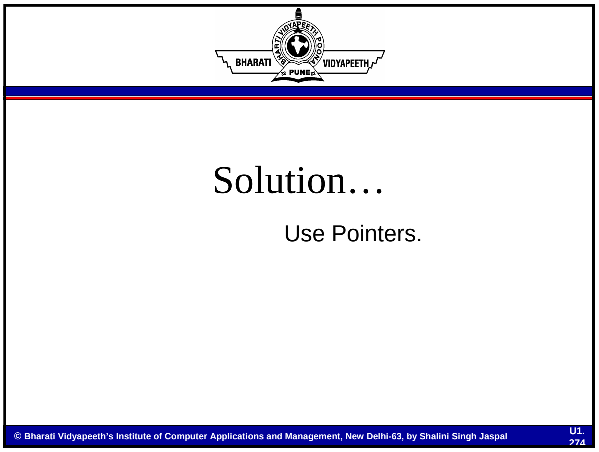
num1
5
int

```
void addOne (int num1)
{
    num1++ ;
    printf ("In addOne: ") ;
    printf ("num1 = %d\n", num1) ;
}
```

num1
6
int

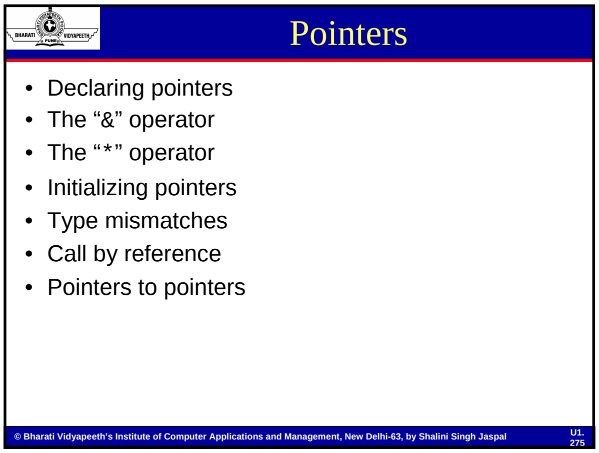
OUTPUT

```
In addOne: num1 = 6
In main: num1 = 5
```



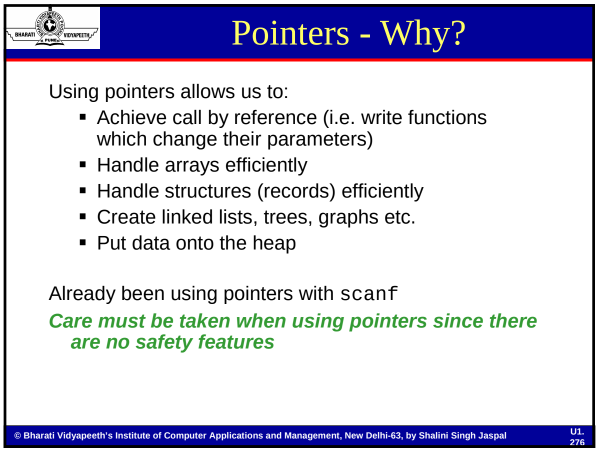
Solution…

Use Pointers.



Pointers

- Declaring pointers
- The “&” operator
- The “*” operator
- Initializing pointers
- Type mismatches
- Call by reference
- Pointers to pointers



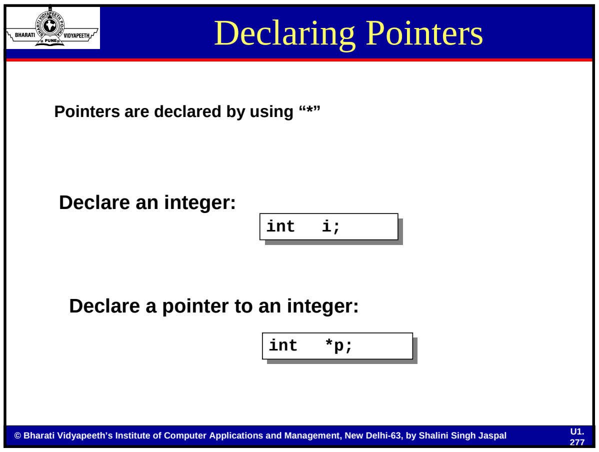
Pointers - Why?

Using pointers allows us to:

- Achieve call by reference (i.e. write functions which change their parameters)
- Handle arrays efficiently
- Handle structures (records) efficiently
- Create linked lists, trees, graphs etc.
- Put data onto the heap

Already been using pointers with `scanf`

Care must be taken when using pointers since there are no safety features



Declaring Pointers

Pointers are declared by using “*”

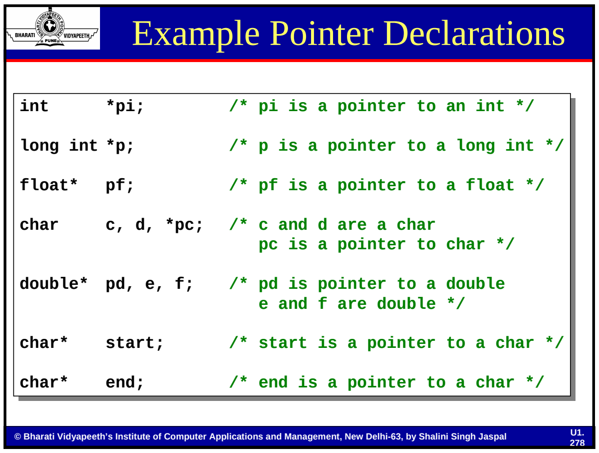
Declare an integer:

```
int   i;
```

Declare a pointer to an integer:

```
int   *p;
```

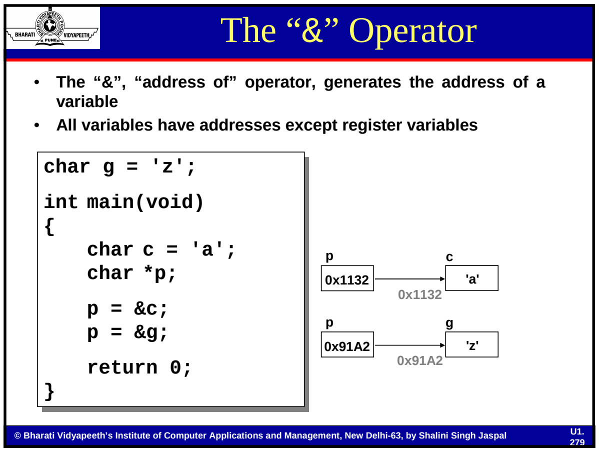
U1. 277



Example Pointer Declarations

```
int      *pi;         /* pi is a pointer to an int */

long int *p;          /* p is a pointer to a long int */

float*   pf;          /* pf is a pointer to a float */

char     c, d, *pc;   /* c and d are a char
                         pc is a pointer to char */

double*  pd, e, f;    /* pd is pointer to a double
                         e and f are double */

char*    start;       /* start is a pointer to a char */

char*    end;         /* end is a pointer to a char */
```

U1. 278



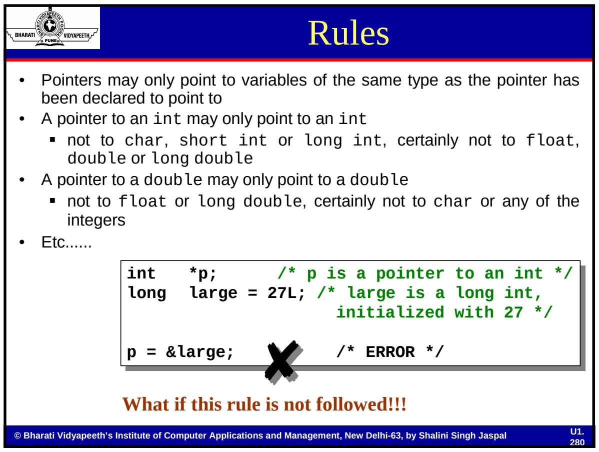
The “&” Operator

- The “&”, “address of” operator, generates the address of a variable
- All variables have addresses except register variables

```
char g = 'z';

int main(void)
{
    char c = 'a';
    char *p;

    p = &c;
    p = &g;

    return 0;
}
```

p
0x1132
c
'a'
0x1132
p
0x91A2
g
'z'
0x91A2

U1. 279



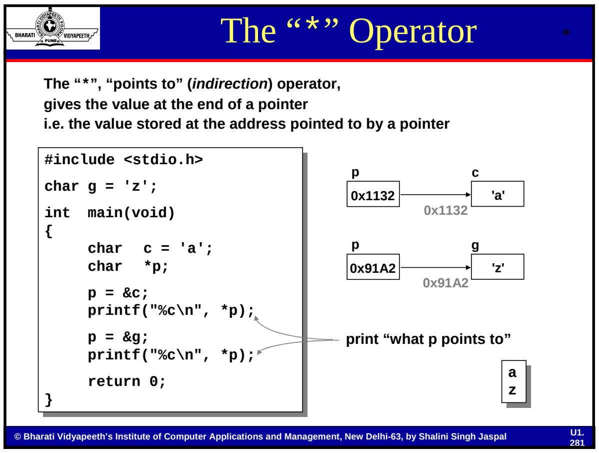
Rules

- Pointers may only point to variables of the same type as the pointer has been declared to point to
- A pointer to an `int` may only point to an `int`
  - not to `char`, `short int` or `long int`, certainly not to `float`, `double` or `long double`
- A pointer to a `double` may only point to a `double`
  - not to `float` or `long double`, certainly not to `char` or any of the integers
- Etc......

```
int    *p;       /* p is a pointer to an int */
long   large = 27L; /* large is a long int,
                      initialized with 27 */

p = &large;  ✘        /* ERROR */
```

What if this rule is not followed!!!

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 280



The “*” Operator

The “*”, “points to” (*indirection*) operator,
gives the value at the end of a pointer
i.e. the value stored at the address pointed to by a pointer

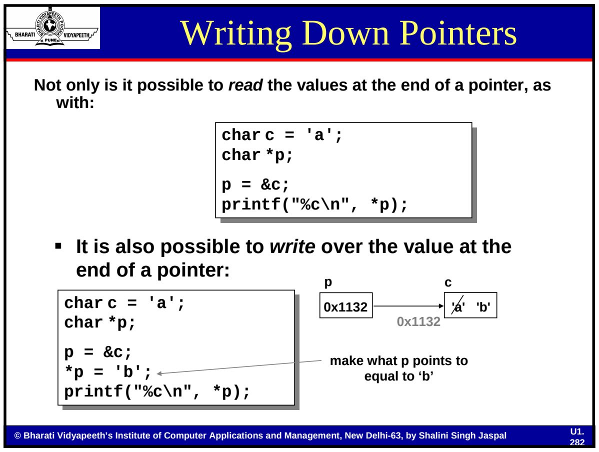
```
#include <stdio.h>

char g = 'z';

int  main(void)
{
     char   c = 'a';
     char   *p;

     p = &c;
     printf("%c\n", *p);

     p = &g;
     printf("%c\n", *p);

     return 0;
}
```

p
c
0x1132
'a'
0x1132
p
g
0x91A2
'z'
0x91A2

print “what p points to”

```
a
z
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 281



Writing Down Pointers

Not only is it possible to *read* the values at the end of a pointer, as with:

```
char c = 'a';
char *p;

p = &c;
printf("%c\n", *p);
```

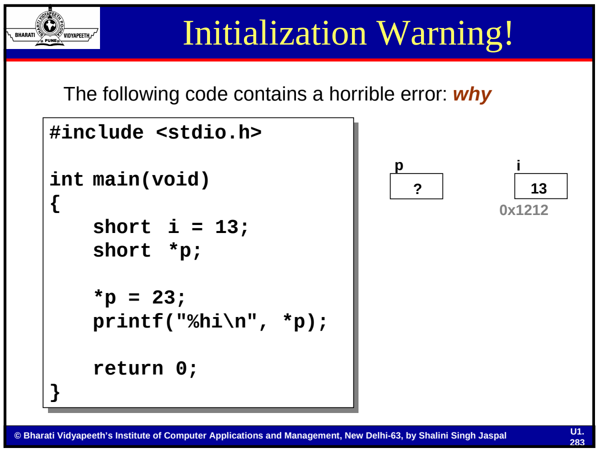
- It is also possible to *write* over the value at the end of a pointer:

```
char c = 'a';
char *p;

p = &c;
*p = 'b';
printf("%c\n", *p);
```

p
c
0x1132
~~'a'~~ 'b'
0x1132

make what p points to equal to ‘b’

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 282



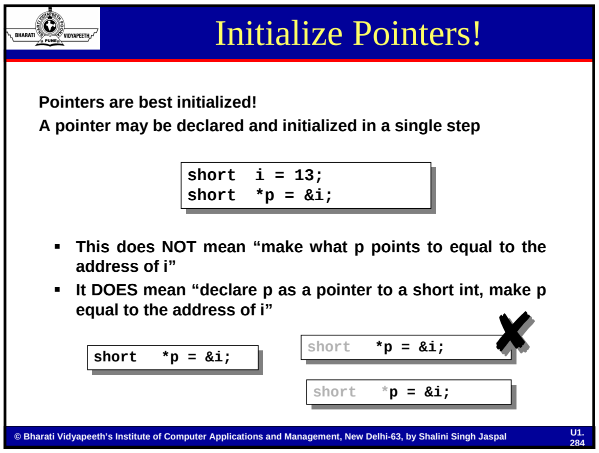
Initialization Warning!

The following code contains a horrible error: ***why***

```
#include <stdio.h>

int main(void)
{
    short  i = 13;
    short  *p;

    *p = 23;
    printf("%hi\n", *p);

    return 0;
}
```

p
?
i
13
0x1212

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 283



Initialize Pointers!

Pointers are best initialized!
A pointer may be declared and initialized in a single step

```
short  i = 13;
short  *p = &i;
```

- **This does NOT mean "make what p points to equal to the address of i"**
- **It DOES mean "declare p as a pointer to a short int, make p equal to the address of i"**

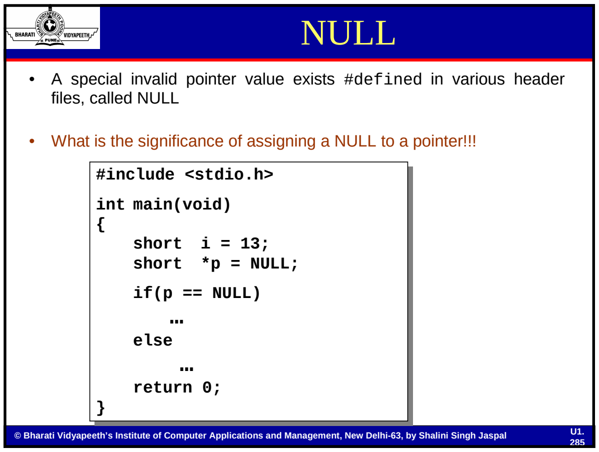
```
short   *p = &i;
```

```
short   *p = &i;
```

✗

```
short   *p = &i;
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 284

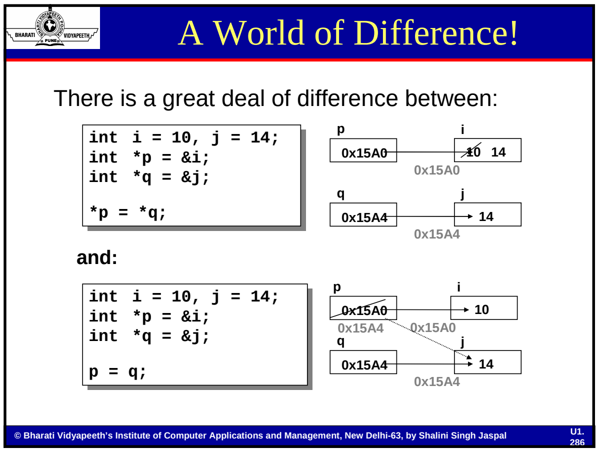


NULL

- A special invalid pointer value exists `#defined` in various header files, called NULL
- What is the significance of assigning a NULL to a pointer!!!

```
#include <stdio.h>
int main(void)
{
    short  i = 13;
    short  *p = NULL;
    if(p == NULL)
      …
    else
       …
    return 0;
}
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 285



A World of Difference!

There is a great deal of difference between:

```
int  i = 10, j = 14;
int  *p = &i;
int  *q = &j;

*p = *q;
```

p
0x15A0
i
~~10~~ 14
0x15A0
q
0x15A4
j
14
0x15A4

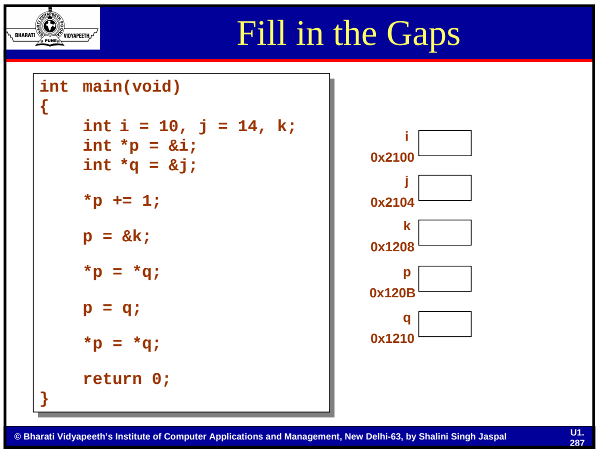
and:

```
int  i = 10, j = 14;
int  *p = &i;
int  *q = &j;

p = q;
```

p
~~0x15A0~~
0x15A4
i
10
0x15A0
q
0x15A4
j
14
0x15A4

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal

U1.
286



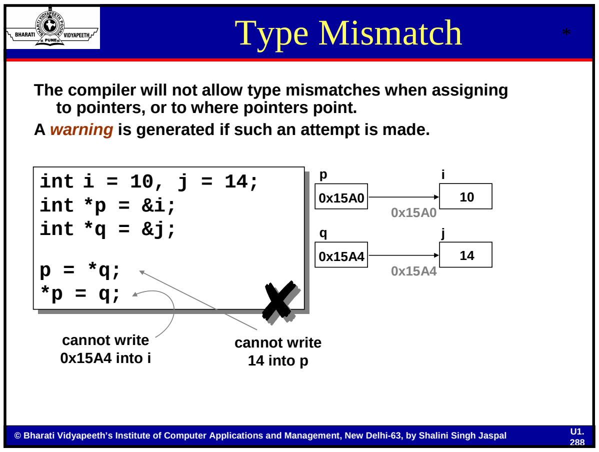
Fill in the Gaps

```
int  main(void)
{
     int i = 10, j = 14, k;
     int *p = &i;
     int *q = &j;

     *p += 1;

     p = &k;

     *p = *q;

     p = q;

     *p = *q;

     return 0;
}
```

i
0x2100
j
0x2104
k
0x1208
p
0x120B
q
0x1210

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal

U1.
287



Type Mismatch

The compiler will not allow type mismatches when assigning to pointers, or to where pointers point.

A *warning* is generated if such an attempt is made.

```
int i = 10, j = 14;
int *p = &i;
int *q = &j;

p = *q;
*p = q;
```

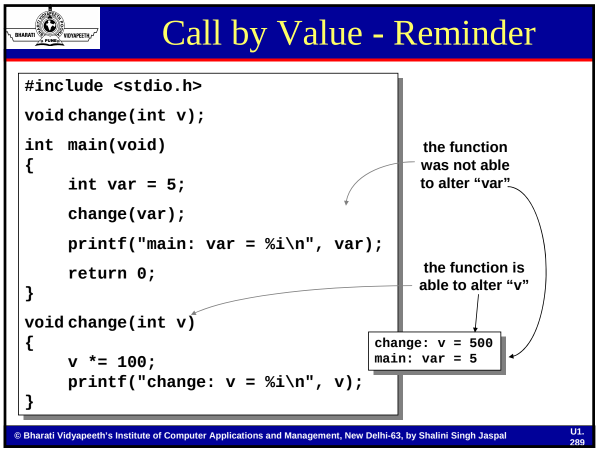
p
0x15A0
i
10
0x15A0
q
0x15A4
j
14
0x15A4
✘

cannot write
0x15A4 into i

cannot write
14 into p

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal

U1.
288



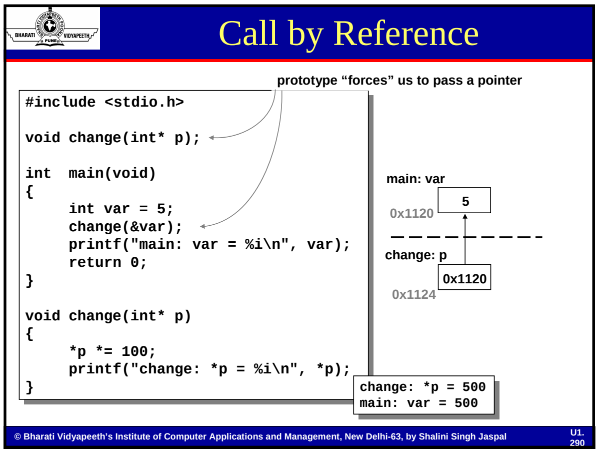
Call by Value - Reminder

```
#include <stdio.h>
void change(int v);
int  main(void)
{
     int var = 5;
     change(var);
     printf("main: var = %i\n", var);
     return 0;
}
void change(int v)
{
     v *= 100;
     printf("change: v = %i\n", v);
}
```

the function was not able to alter "var"

the function is able to alter "v"

```
change: v = 500
main: var = 5
```



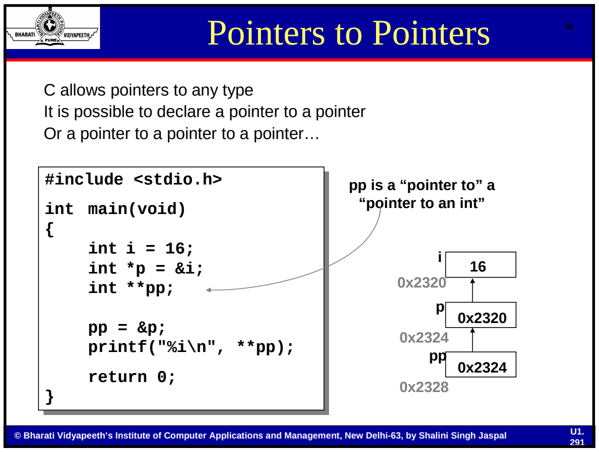
Call by Reference

prototype "forces" us to pass a pointer

```
#include <stdio.h>

void change(int* p);

int  main(void)
{
     int var = 5;
     change(&var);
     printf("main: var = %i\n", var);
     return 0;
}

void change(int* p)
{
     *p *= 100;
     printf("change: *p = %i\n", *p);
}
```

main: var
5
0x1120
change: p
0x1120
0x1124

```
change: *p = 500
main: var = 500
```



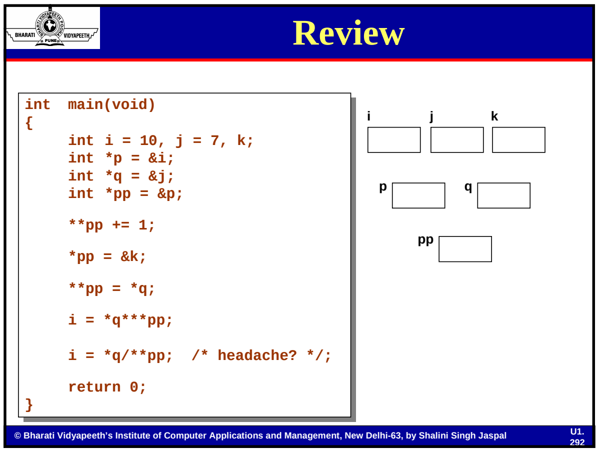
Pointers to Pointers

C allows pointers to any type
It is possible to declare a pointer to a pointer
Or a pointer to a pointer to a pointer…

```
#include <stdio.h>

int  main(void)
{
     int i = 16;
     int *p = &i;
     int **pp;

     pp = &p;
     printf("%i\n", **pp);

     return 0;
}
```

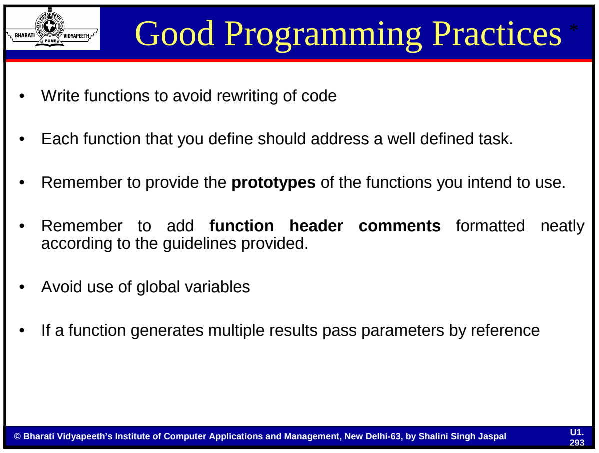
pp is a "pointer to" a "pointer to an int"

i
16
0x2320
p
0x2320
0x2324
pp
0x2324
0x2328



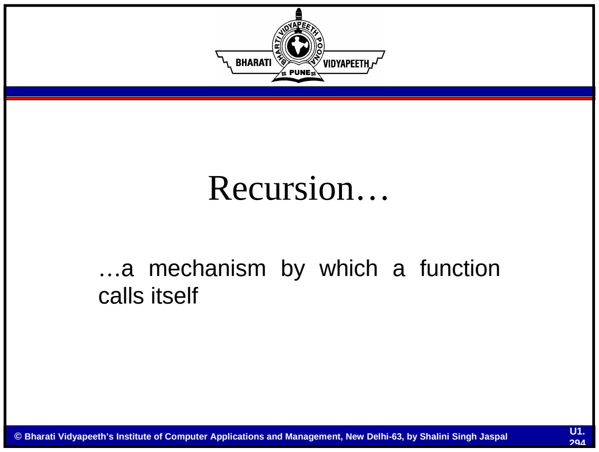
Review

```
int  main(void)
{
     int i = 10, j = 7, k;
     int *p = &i;
     int *q = &j;
     int *pp = &p;

     **pp += 1;

     *pp = &k;

     **pp = *q;

     i = *q***pp;

     i = *q/**pp;  /* headache? */;

     return 0;
}
```



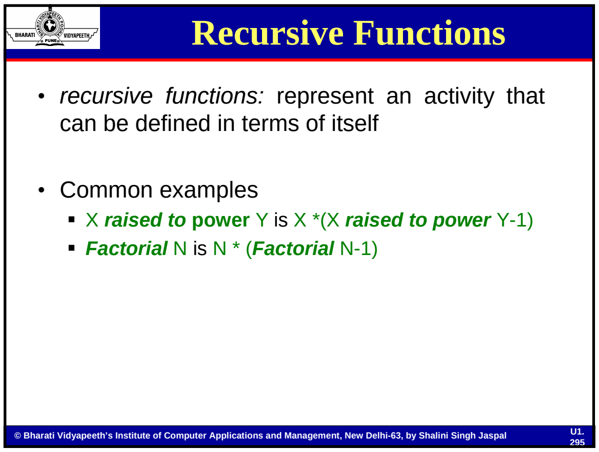
Good Programming Practices

- Write functions to avoid rewriting of code
- Each function that you define should address a well defined task.
- Remember to provide the **prototypes** of the functions you intend to use.
- Remember to add **function header comments** formatted neatly according to the guidelines provided.
- Avoid use of global variables
- If a function generates multiple results pass parameters by reference



Recursion…

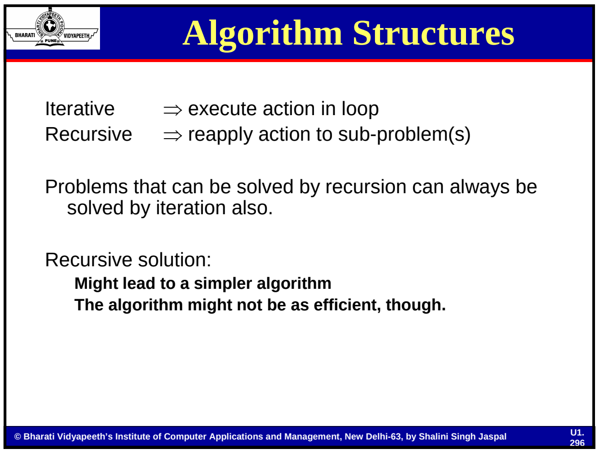
…a mechanism by which a function calls itself



Recursive Functions

- *recursive functions:* represent an activity that can be defined in terms of itself

- Common examples
  - X ***raised to* power** Y is X *(X ***raised to power*** Y-1)
  - ***Factorial*** N is N * (***Factorial*** N-1)



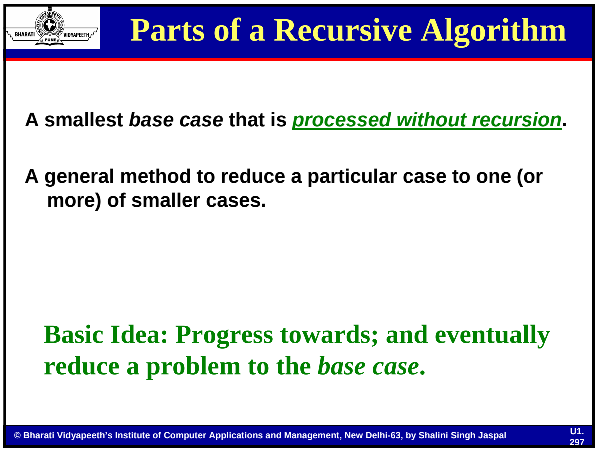
Algorithm Structures

Iterative ⇒ execute action in loop
Recursive ⇒ reapply action to sub-problem(s)

Problems that can be solved by recursion can always be solved by iteration also.

Recursive solution:

Might lead to a simpler algorithm
The algorithm might not be as efficient, though.

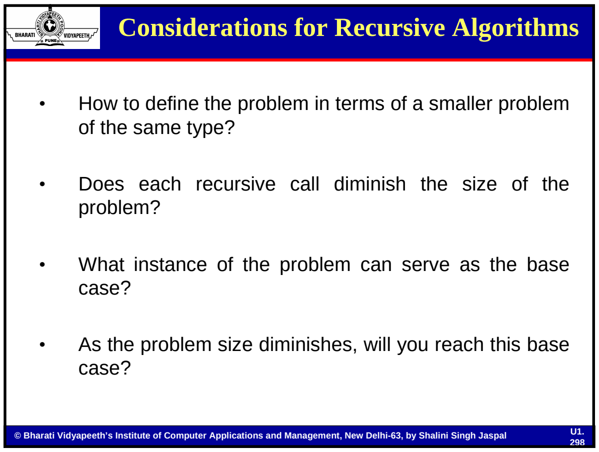


Parts of a Recursive Algorithm

A smallest *base case* that is *processed without recursion*.

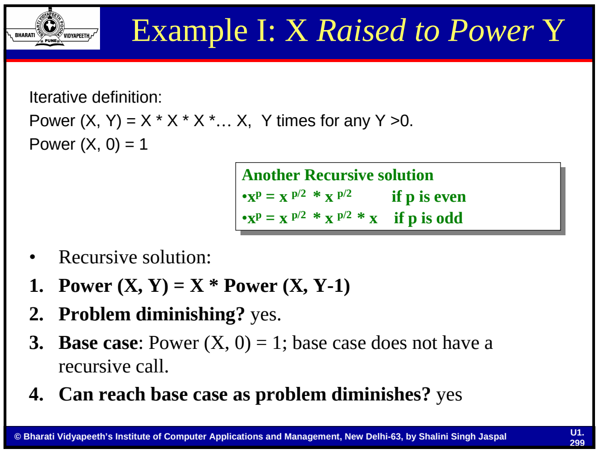
A general method to reduce a particular case to one (or more) of smaller cases.

Basic Idea: Progress towards; and eventually reduce a problem to the *base case*.



Considerations for Recursive Algorithms

- How to define the problem in terms of a smaller problem of the same type?
- Does each recursive call diminish the size of the problem?
- What instance of the problem can serve as the base case?
- As the problem size diminishes, will you reach this base case?



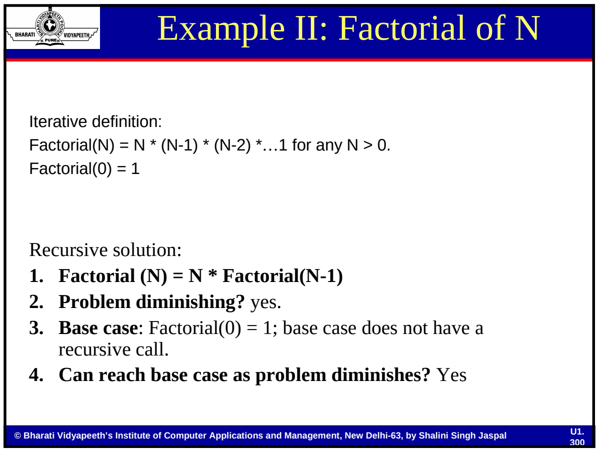
Example I: X *Raised to Power* Y

Iterative definition:
Power (X, Y) = X * X * X *… X, Y times for any Y >0.
Power (X, 0) = 1

Another Recursive solution
•$x^p = x^{p/2} * x^{p/2}$ if p is even
•$x^p = x^{p/2} * x^{p/2} * x$ if p is odd

- Recursive solution:

1. **Power (X, Y) = X * Power (X, Y-1)**
2. **Problem diminishing?** yes.
3. **Base case**: Power (X, 0) = 1; base case does not have a recursive call.
4. **Can reach base case as problem diminishes?** yes

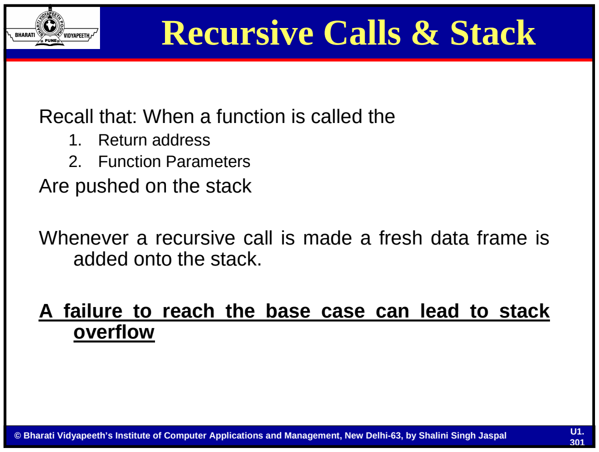


Example II: Factorial of N

Iterative definition:
Factorial(N) = N * (N-1) * (N-2) *…1 for any N > 0.
Factorial(0) = 1

Recursive solution:

1. **Factorial (N) = N * Factorial(N-1)**
2. **Problem diminishing?** yes.
3. **Base case**: Factorial(0) = 1; base case does not have a recursive call.
4. **Can reach base case as problem diminishes?** Yes



Recursive Calls & Stack

Recall that: When a function is called the

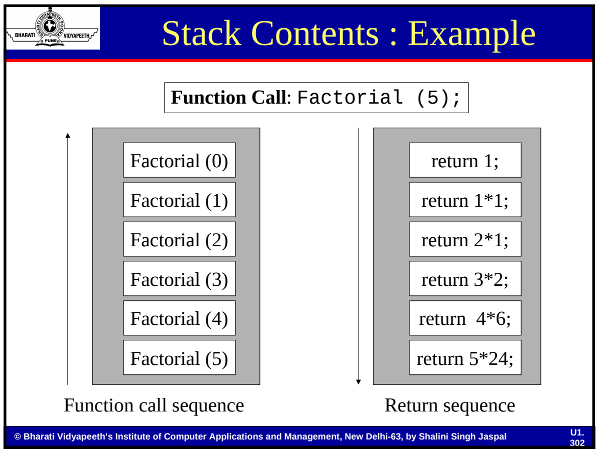
1. Return address
2. Function Parameters

Are pushed on the stack

Whenever a recursive call is made a fresh data frame is added onto the stack.

A failure to reach the base case can lead to stack overflow

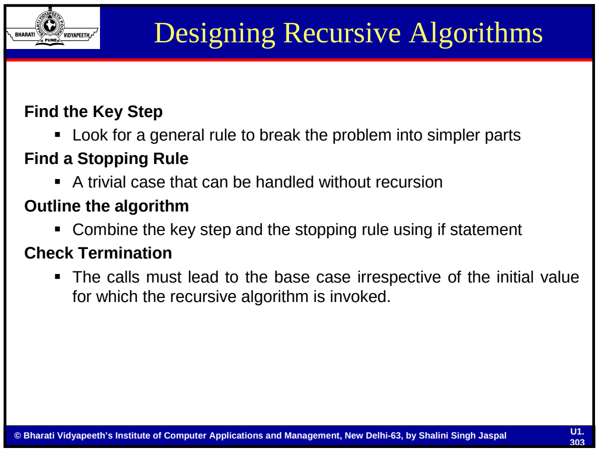
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 301



Stack Contents : Example

Function Call: `Factorial (5);`

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 302



Designing Recursive Algorithms

Find the Key Step

- Look for a general rule to break the problem into simpler parts

Find a Stopping Rule

- A trivial case that can be handled without recursion

Outline the algorithm

- Combine the key step and the stopping rule using if statement

Check Termination

- The calls must lead to the base case irrespective of the initial value for which the recursive algorithm is invoked.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 303

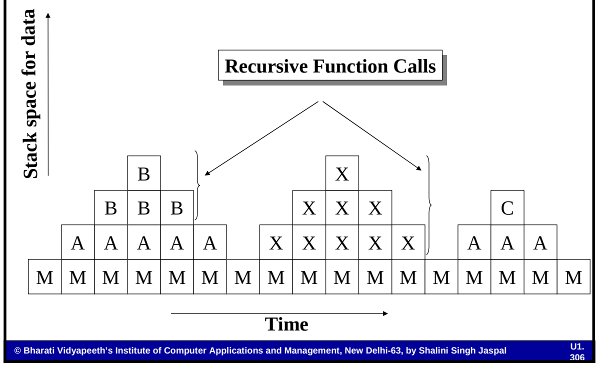
Discuss

⇒Sum of first *n* positive integers
⇒Fibonacci Series
⇒Recurrence Relations
⇒Binary Search
⇒Is Palindrome
⇒Reversing a string
⇒Towers of Hanoi

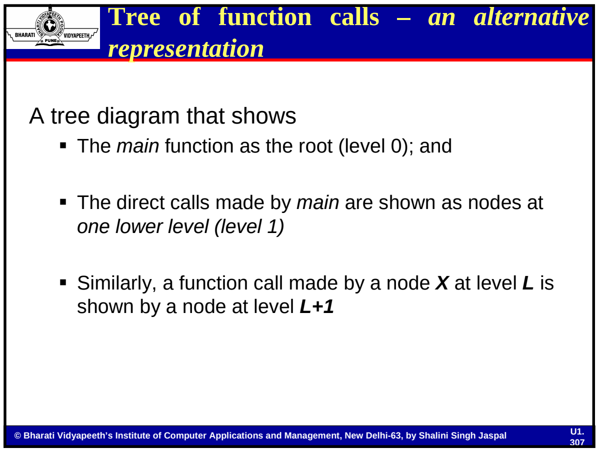
Function Calls

- **We know that function call activity follows the LIFO rule.**
- **A set of stack frames can be used to represent function calls.**
- **These stack frames would plot stack contents against time.**
- **A stack frame showing**
  - **a particular function stacked upon itself,**
  - **at a given point of time**
- **represents a recursive function call.**

A Sample Stack Frame



			B						X							
		B	B	B				X	X	X				C		
	A	A	A	A	A		X	X	X	X	X		A	A	A	
M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M

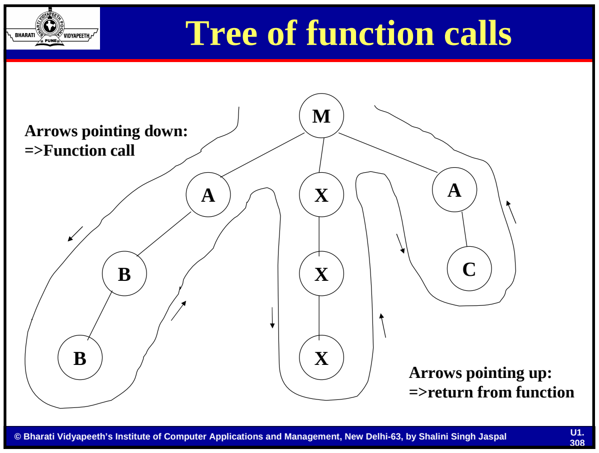


Tree of function calls – *an alternative representation*

A tree diagram that shows

- The *main* function as the root (level 0); and
- The direct calls made by *main* are shown as nodes at *one lower level (level 1)*
- Similarly, a function call made by a node ***X*** at level ***L*** is shown by a node at level ***L+1***

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 307

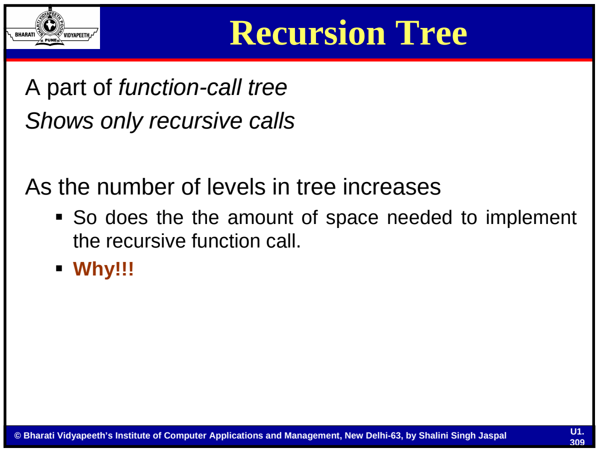


Tree of function calls

Arrows pointing down:
=>Function call

Arrows pointing up:
=>return from function

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 308



Recursion Tree

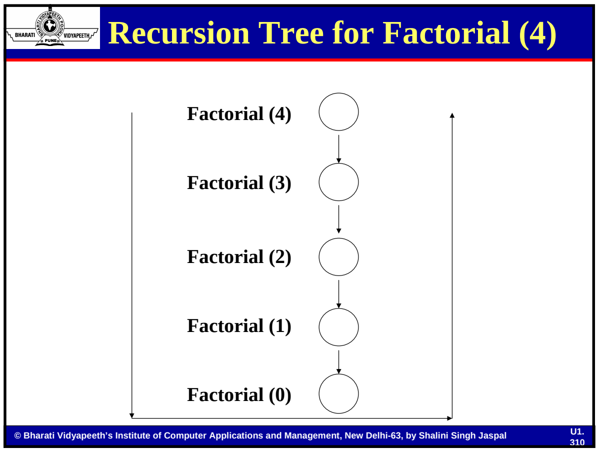
A part of *function-call tree*

Shows only recursive calls

As the number of levels in tree increases

- So does the the amount of space needed to implement the recursive function call.
- **Why!!!**

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 309



Recursion Tree for Factorial (4)

Factorial (4)

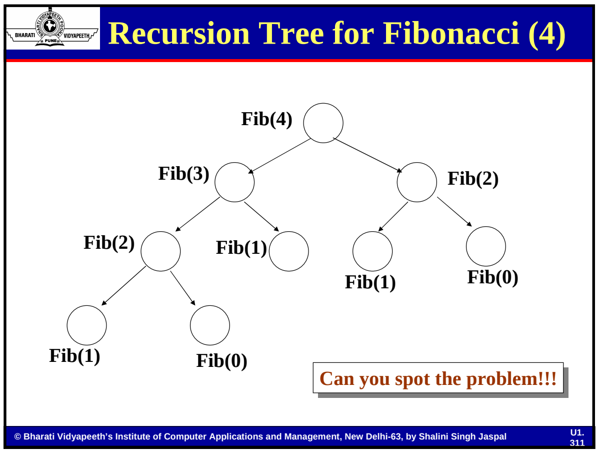
Factorial (3)

Factorial (2)

Factorial (1)

Factorial (0)

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1.310



Recursion Tree for Fibonacci (4)

Fib(4)

Fib(3)

Fib(2)

Fib(2)

Fib(1)

Fib(1)

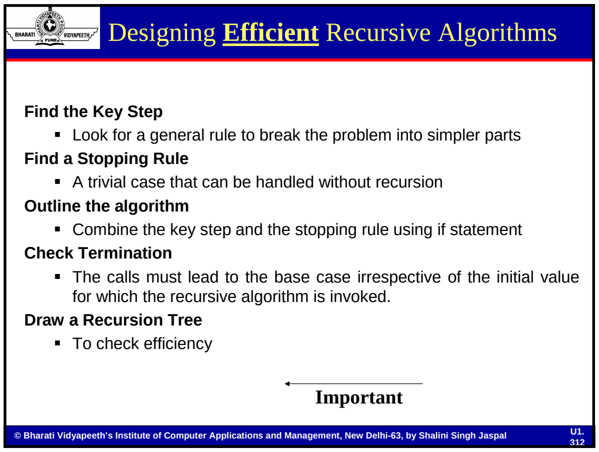
Fib(0)

Fib(1)

Fib(0)

Can you spot the problem!!!

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1.311



Designing **Efficient** Recursive Algorithms

Find the Key Step

- Look for a general rule to break the problem into simpler parts

Find a Stopping Rule

- A trivial case that can be handled without recursion

Outline the algorithm

- Combine the key step and the stopping rule using if statement

Check Termination

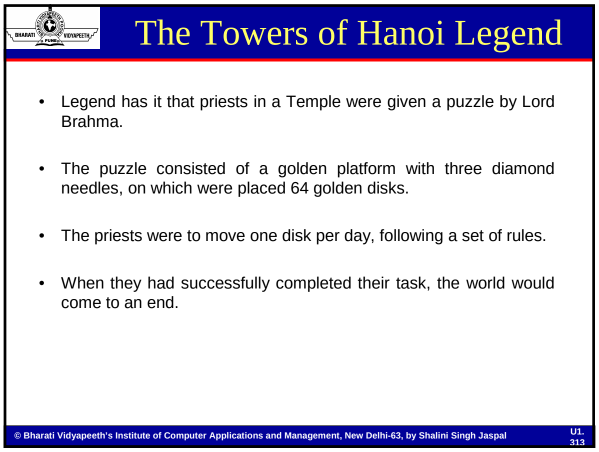
- The calls must lead to the base case irrespective of the initial value for which the recursive algorithm is invoked.

Draw a Recursion Tree

- To check efficiency

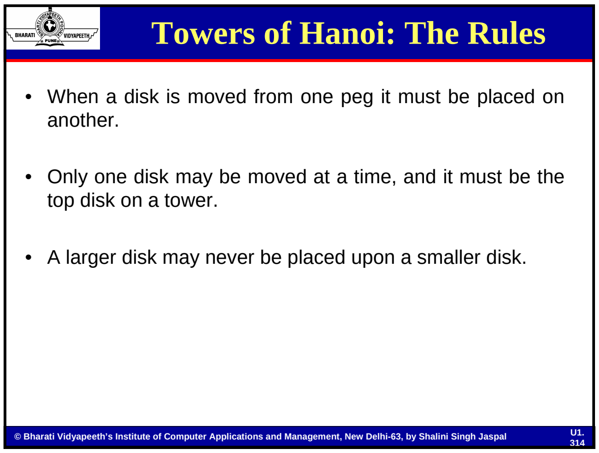
Important

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1.312



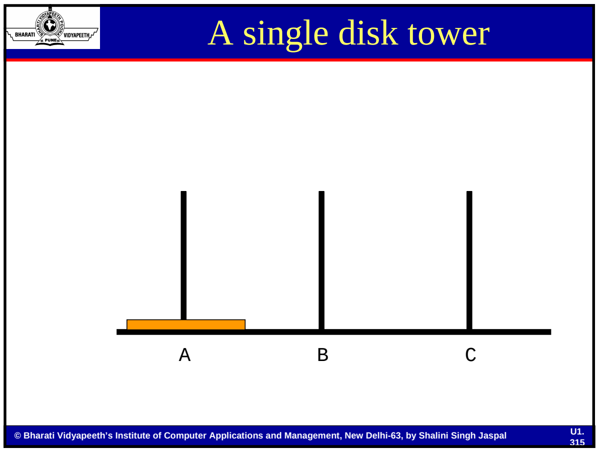
The Towers of Hanoi Legend

- Legend has it that priests in a Temple were given a puzzle by Lord Brahma.
- The puzzle consisted of a golden platform with three diamond needles, on which were placed 64 golden disks.
- The priests were to move one disk per day, following a set of rules.
- When they had successfully completed their task, the world would come to an end.

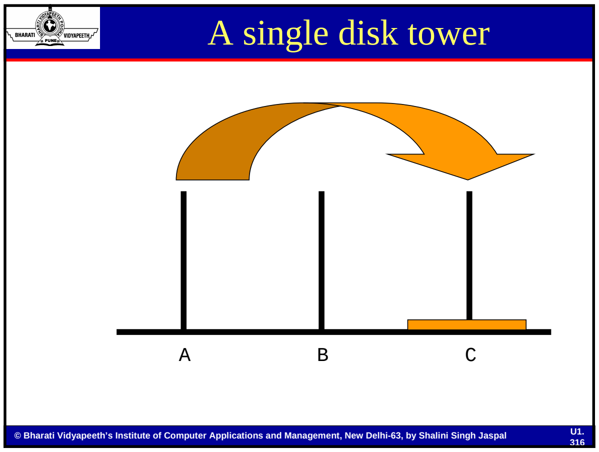


Towers of Hanoi: The Rules

- When a disk is moved from one peg it must be placed on another.
- Only one disk may be moved at a time, and it must be the top disk on a tower.
- A larger disk may never be placed upon a smaller disk.



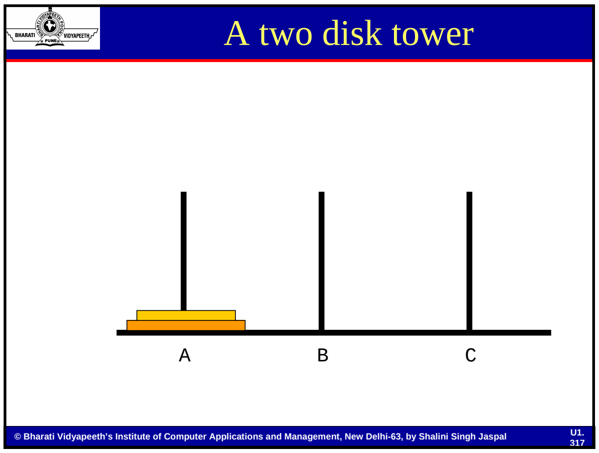
A single disk tower



A single disk tower

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal

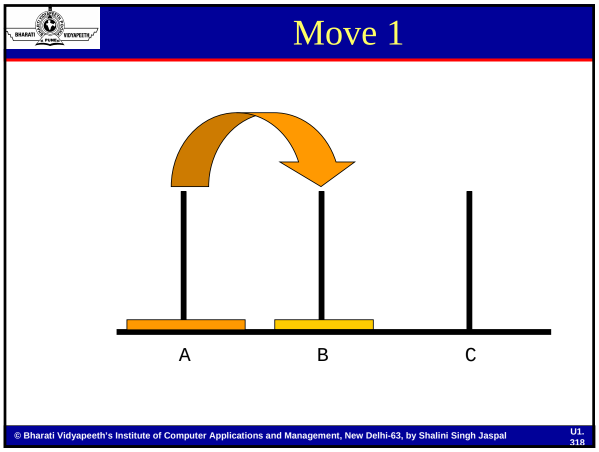
U1. 316



A two disk tower

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal

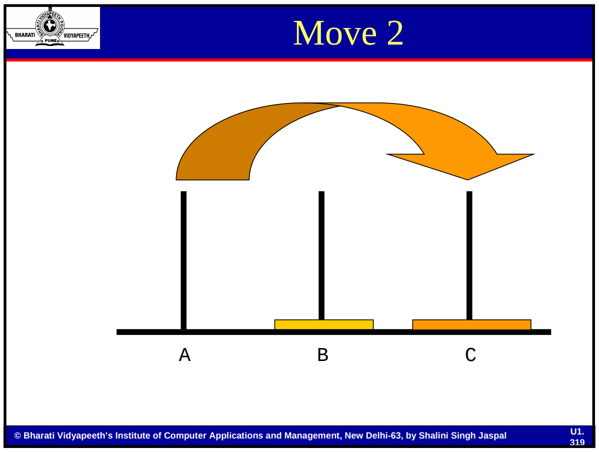
U1. 317



Move 1

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal

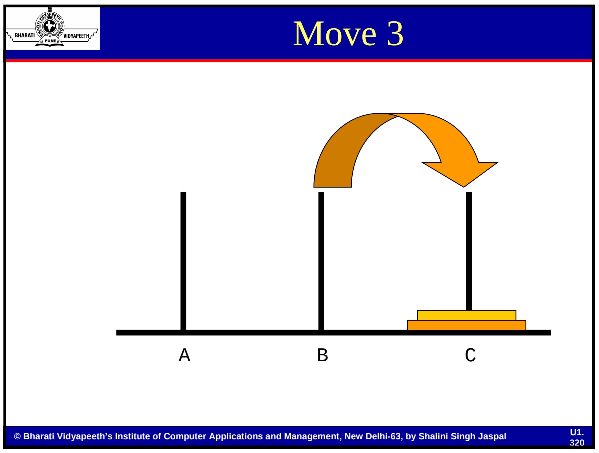
U1. 318



Move 2

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal

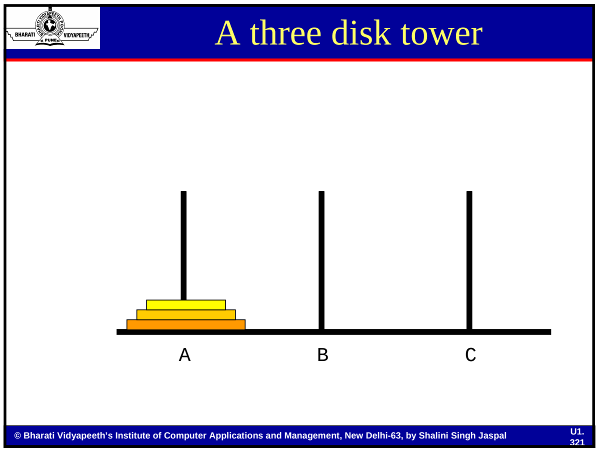
U1. 319



Move 3

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal

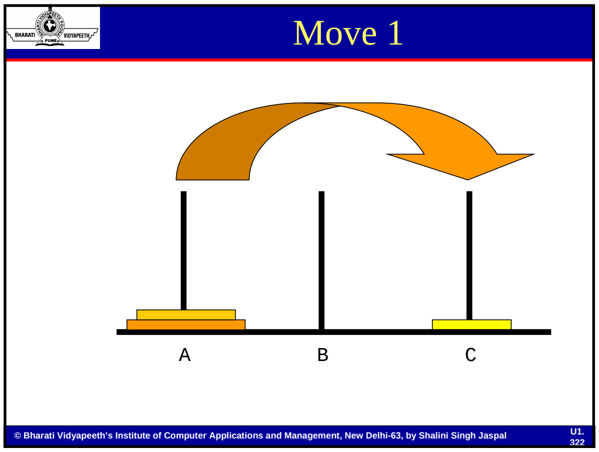
U1. 320



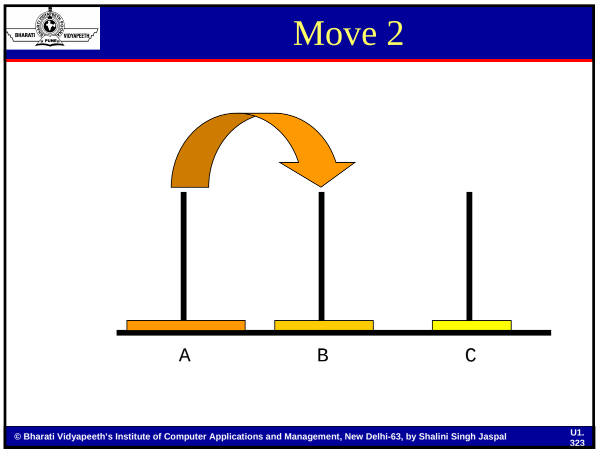
A three disk tower

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal

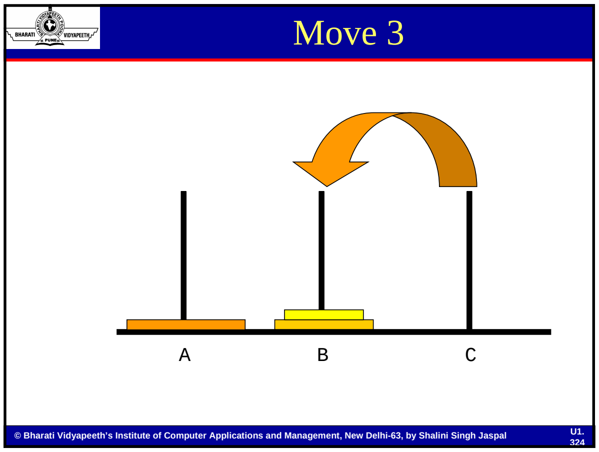
U1. 321



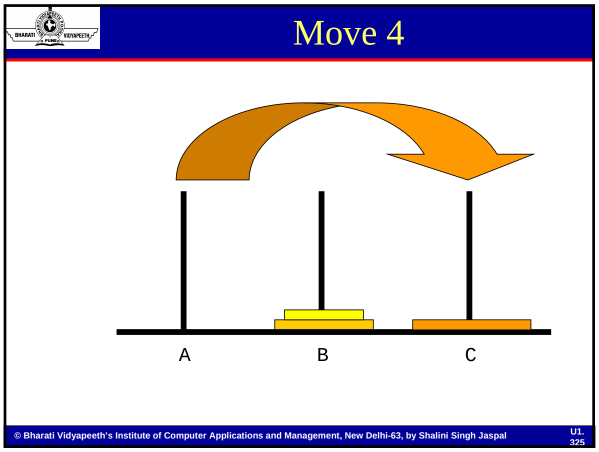
Move 1



Move 2



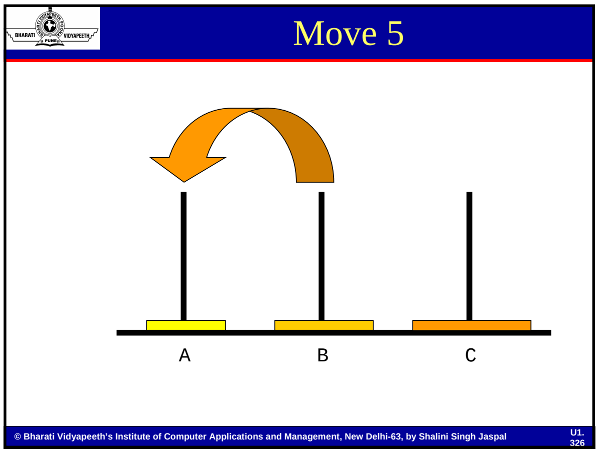
Move 3



Move 4

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal

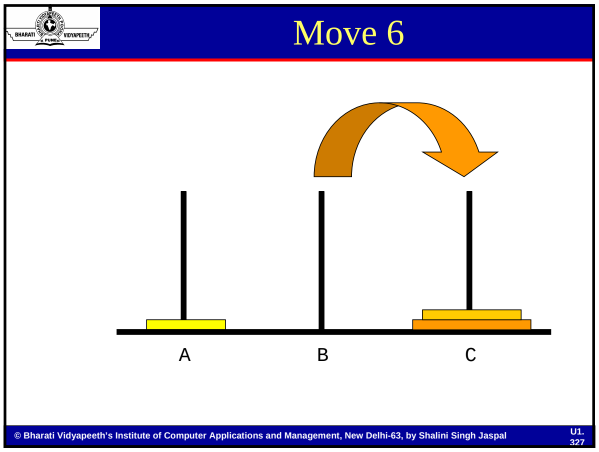
U1. 325



Move 5

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal

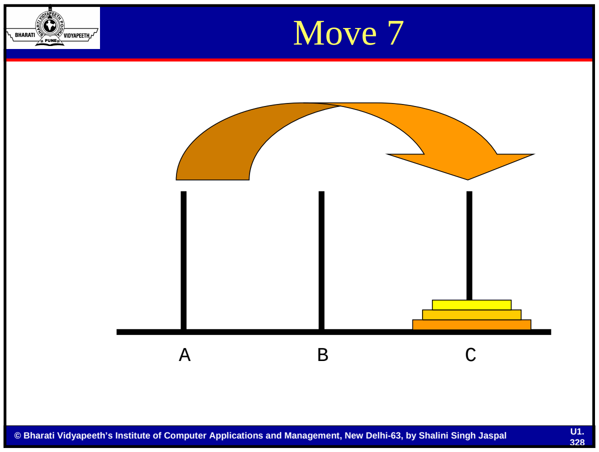
U1. 326



Move 6

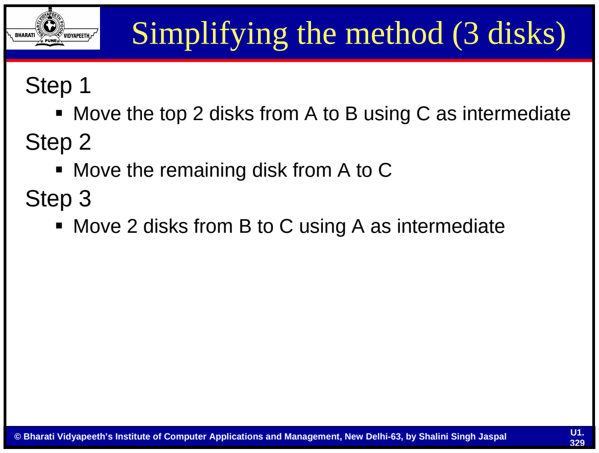
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal

U1. 327



Move 7

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1.328



Simplifying the method (3 disks)

Step 1

- Move the top 2 disks from A to B using C as intermediate

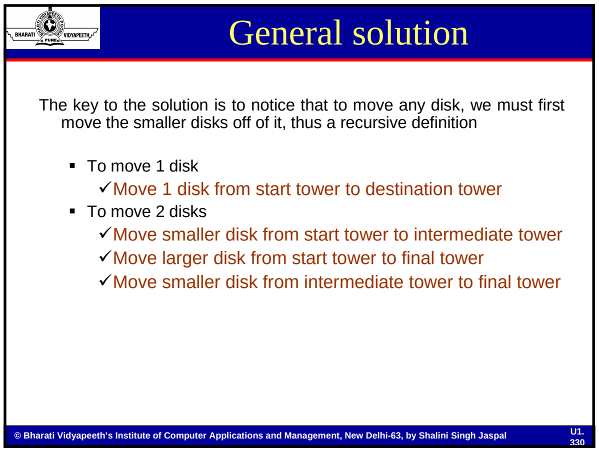
Step 2

- Move the remaining disk from A to C

Step 3

- Move 2 disks from B to C using A as intermediate

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1.329

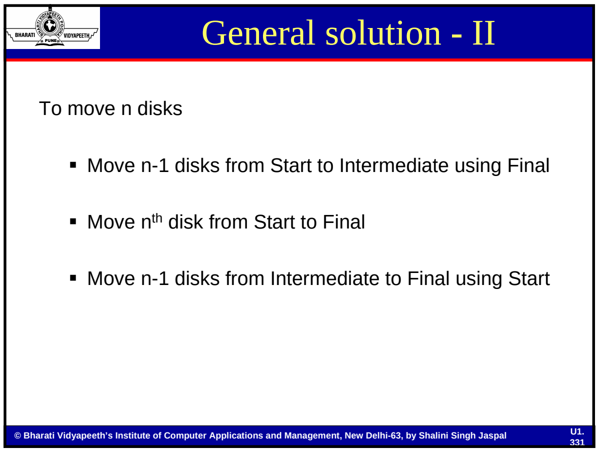


General solution

The key to the solution is to notice that to move any disk, we must first move the smaller disks off of it, thus a recursive definition

- To move 1 disk
  - ✓Move 1 disk from start tower to destination tower
- To move 2 disks
  - ✓Move smaller disk from start tower to intermediate tower
  - ✓Move larger disk from start tower to final tower
  - ✓Move smaller disk from intermediate tower to final tower

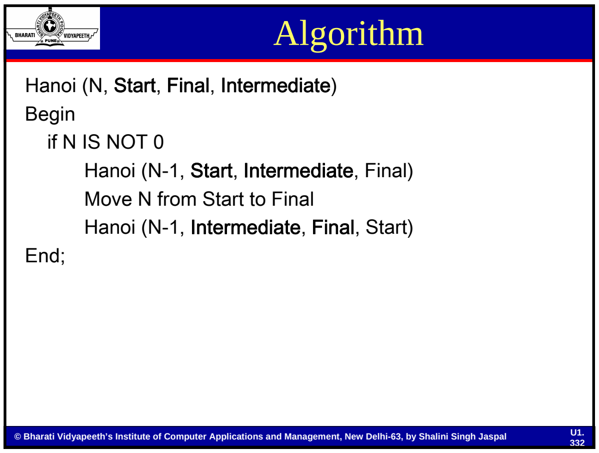
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1.330



General solution - II

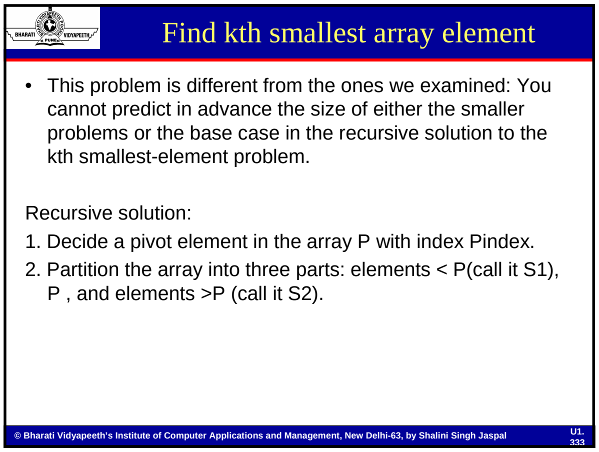
To move n disks

- Move n-1 disks from Start to Intermediate using Final
- Move nth disk from Start to Final
- Move n-1 disks from Intermediate to Final using Start



Algorithm

```
Hanoi (N, Start, Final, Intermediate)
Begin
   if N IS NOT 0
        Hanoi (N-1, Start, Intermediate, Final)
        Move N from Start to Final
        Hanoi (N-1, Intermediate, Final, Start)
End;
```

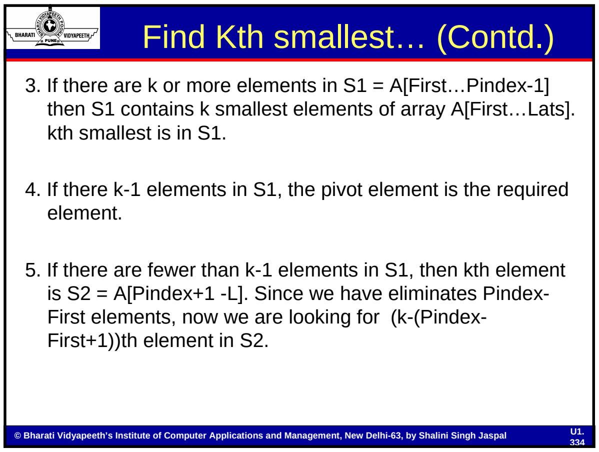


Find kth smallest array element

- This problem is different from the ones we examined: You cannot predict in advance the size of either the smaller problems or the base case in the recursive solution to the kth smallest-element problem.

Recursive solution:

1. Decide a pivot element in the array P with index Pindex.
2. Partition the array into three parts: elements < P(call it S1), P , and elements >P (call it S2).



Find Kth smallest… (Contd.)

3. If there are k or more elements in S1 = A[First…Pindex-1] then S1 contains k smallest elements of array A[First…Lats]. kth smallest is in S1.

4. If there k-1 elements in S1, the pivot element is the required element.

5. If there are fewer than k-1 elements in S1, then kth element is S2 = A[Pindex+1 -L]. Since we have eliminates Pindex-First elements, now we are looking for (k-(Pindex-First+1))th element in S2.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1. 334