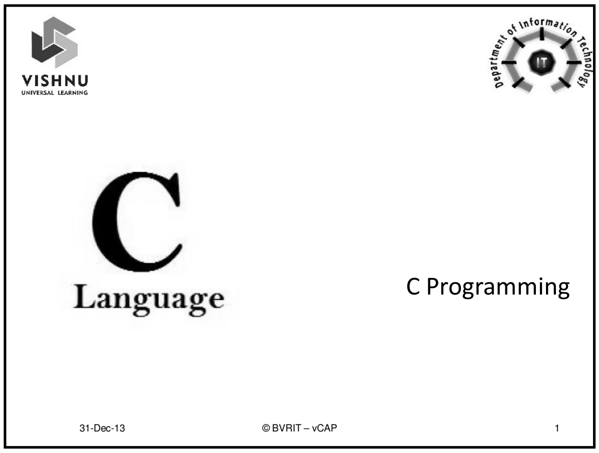
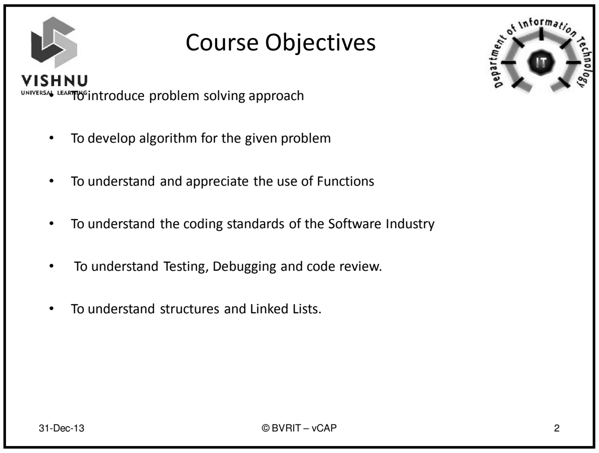


C Programming

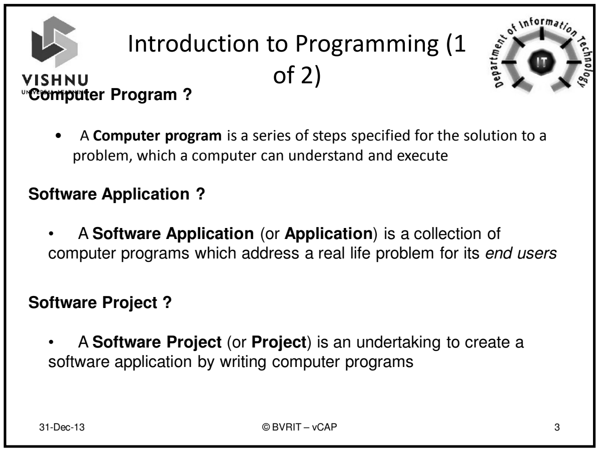
31-Dec-13 © BVRIT – vCAP 1



Course Objectives

- To introduce problem solving approach
- To develop algorithm for the given problem
- To understand and appreciate the use of Functions
- To understand the coding standards of the Software Industry
- To understand Testing, Debugging and code review.
- To understand structures and Linked Lists.

31-Dec-13 © BVRIT – vCAP 2



Introduction to Programming (1 of 2)

Computer Program ?

- A **Computer program** is a series of steps specified for the solution to a problem, which a computer can understand and execute

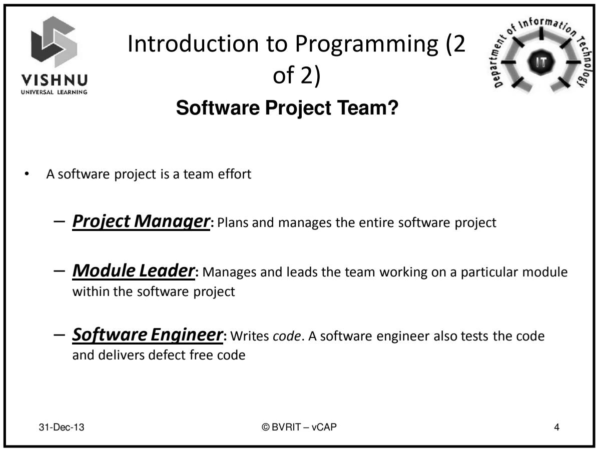
Software Application ?

- A **Software Application** (or **Application**) is a collection of computer programs which address a real life problem for its *end users*

Software Project ?

- A **Software Project** (or **Project**) is an undertaking to create a software application by writing computer programs

31-Dec-13 © BVRIT – vCAP 3

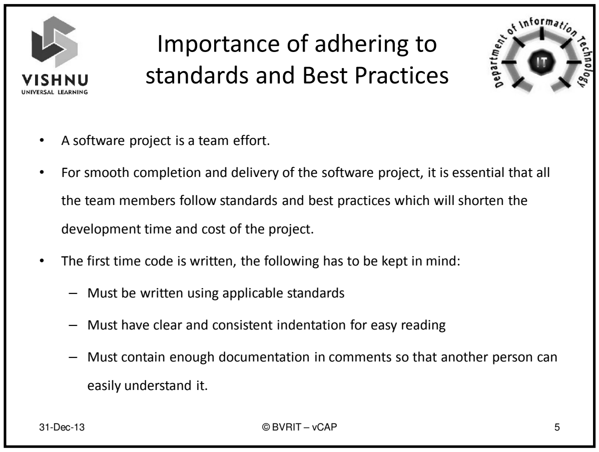


Introduction to Programming (2 of 2)

Software Project Team?

- A software project is a team effort
  - ***Project Manager***: Plans and manages the entire software project
  - ***Module Leader***: Manages and leads the team working on a particular module within the software project
  - ***Software Engineer***: Writes *code*. A software engineer also tests the code and delivers defect free code

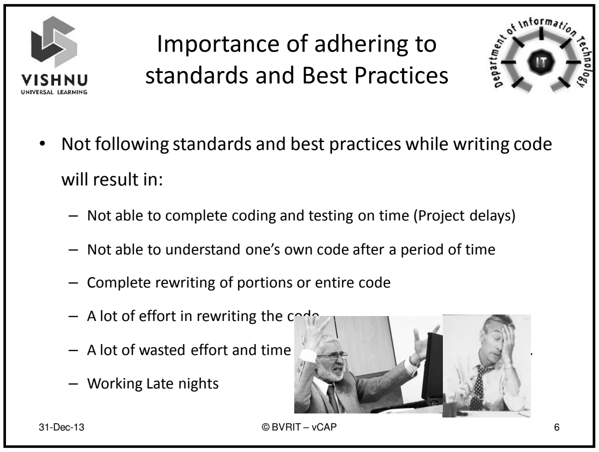
31-Dec-13 © BVRIT – vCAP 4



Importance of adhering to standards and Best Practices

- A software project is a team effort.
- For smooth completion and delivery of the software project, it is essential that all the team members follow standards and best practices which will shorten the development time and cost of the project.
- The first time code is written, the following has to be kept in mind:
  - Must be written using applicable standards
  - Must have clear and consistent indentation for easy reading
  - Must contain enough documentation in comments so that another person can easily understand it.

31-Dec-13 © BVRIT – vCAP 5



Importance of adhering to standards and Best Practices

- Not following standards and best practices while writing code will result in:
  - Not able to complete coding and testing on time (Project delays)
  - Not able to understand one's own code after a period of time
  - Complete rewriting of portions or entire code
  - A lot of effort in rewriting the code
  - A lot of wasted effort and time
  - Working Late nights

31-Dec-13 © BVRIT – vCAP 6

Comments in a C Program

- Comments are used to document programs and improve readability
- It is a very good practice to add comments to all the programs.
- In C Program, a comment will start with /* and ends with */
- Comments are ignored by the compiler during the compilation process.

Figure 2-2: Reviewing a program with no comments

Figure 2-3: Reviewing a program with comments

Comments in a C Program Contd.

.

- **Syntax:**

```
    /*   Comments    */
    /*This is a single line comment */

    /* This is a multiline
     * comment in C */

/*****************************************************
   * This style of commenting is used for functions
*****************************************************/
```

Comments in a C Program Contd.

.

- Note 1: The code in any line should not exceed 80 columns.
- Common Programming Errors and Guidelines:
  - (1) Forgetting to terminate the comment */
  - Comments should make the code accessible to the reader
  - Explain the code's intent in the heading
  - Keep the comments up to date (if you update the code, update the comment)
  - Don't comment bad code--fix it
  - Avoid useless comments

Naming Variables According to Standards

- Hungarian Notation: invented by Charles Simonyi from Microsoft
  - It is a good programming practice that a vriable name should also indicate its data type and its intended use.
  - Example: if there is a variable 'Age' which is of type integer, then it should be declared as 'iAge'.

Prefix	Data Type	Example
i	int and unsigned int	iTotalMarks
f	float	fAverageMarks
d	double	dSalary
l	long and unsigned long	lFactorial
c	signed char and unsigned char	cChoice
ai	Array of integers	aiStudentId
af	Array of float	afquantity
ad	Array of double	adAmount
al	Array of long integers	alSample
ac	Array of characters	acEmpName

Importance of Following Coding Standards

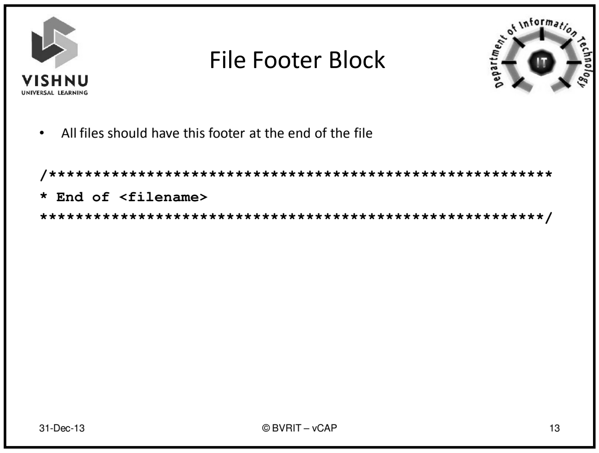
- The coding standards may differ from project to project
- This may also vary from customer to customer
- Every company prepares its own coding standards
- Adhering to coding standards has the following advantages:
  - Improves the readability of the program
  - Improves the clarity of the program
  - Makes a person to understand the program without any difficulty
  - Makes it easy to debug and maintain the program

File Header Block

- All *source* and *header files* must contain at the beginning of file, a section providing information about the source or the header file
- **Format**:

```
/*********************************************************
**
* File           : <filename>
* Description    : <description>
* Author         : <author> <company>
* Version        : <version number>
* Date           : <Date>
**********************************************************
*/
```

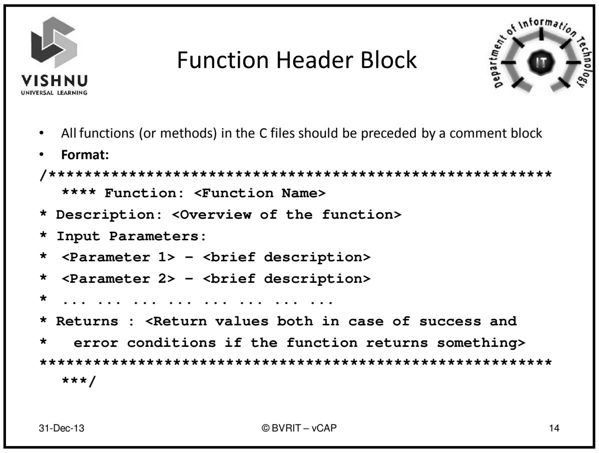
- Here the description should be a brief summary of what the code in the file does



File Footer Block

- All files should have this footer at the end of the file

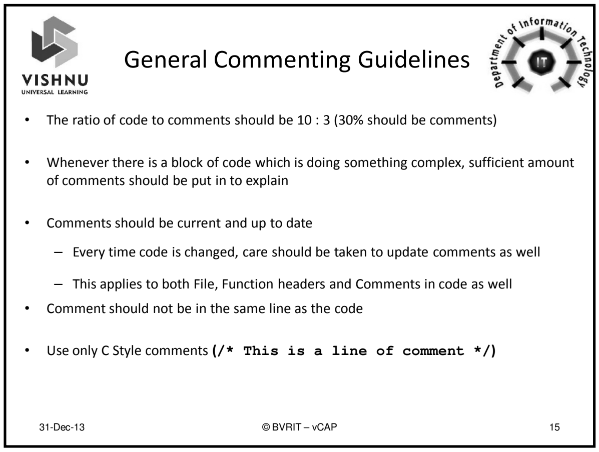
```
/************************************************************
* End of <filename>
************************************************************/
```



Function Header Block

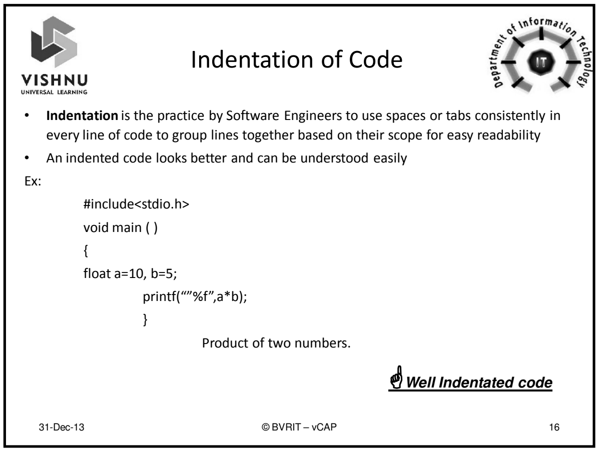
- All functions (or methods) in the C files should be preceded by a comment block
- **Format:**

```
/************************************************************
   **** Function: <Function Name>
* Description: <Overview of the function>
* Input Parameters:
*  <Parameter 1> – <brief description>
*  <Parameter 2> – <brief description>
*  ... ... ... ... ... ... ... ...
* Returns : <Return values both in case of success and
*   error conditions if the function returns something>
************************************************************
   ***/
```



General Commenting Guidelines

- The ratio of code to comments should be 10 : 3 (30% should be comments)
- Whenever there is a block of code which is doing something complex, sufficient amount of comments should be put in to explain
- Comments should be current and up to date
  - Every time code is changed, care should be taken to update comments as well
  - This applies to both File, Function headers and Comments in code as well
- Comment should not be in the same line as the code
- Use only C Style comments **`(/* This is a line of comment */)`**



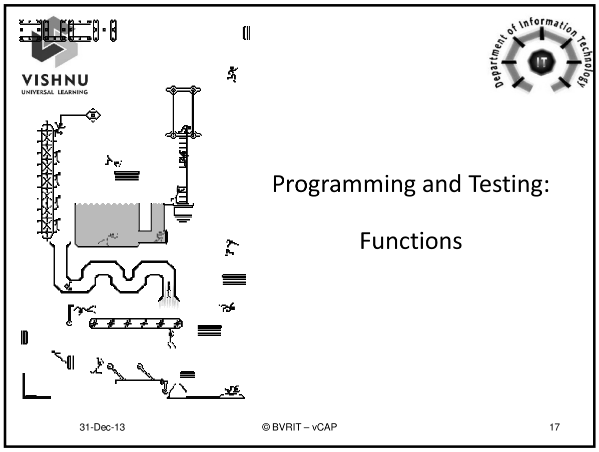
Indentation of Code

- **Indentation** is the practice by Software Engineers to use spaces or tabs consistently in every line of code to group lines together based on their scope for easy readability
- An indented code looks better and can be understood easily

Ex:

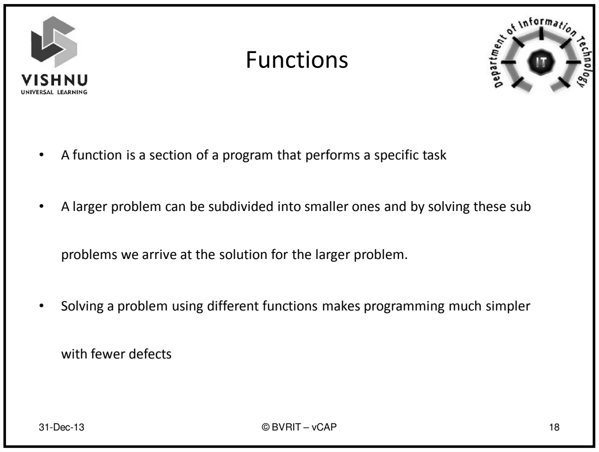
```
      #include<stdio.h>
      void main ( )
      {
      float a=10, b=5;
              printf("“”%f”,a*b);
              }
                      Product of two numbers.
```

☝ ***Well Indentated code***



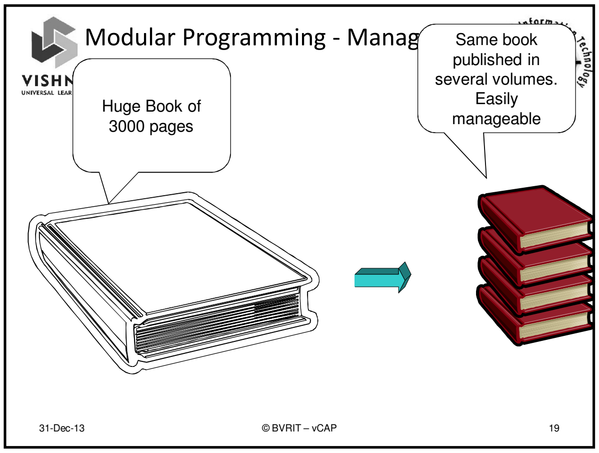
Programming and Testing:

Functions



Functions

- A function is a section of a program that performs a specific task
- A larger problem can be subdivided into smaller ones and by solving these sub problems we arrive at the solution for the larger problem.
- Solving a problem using different functions makes programming much simpler with fewer defects

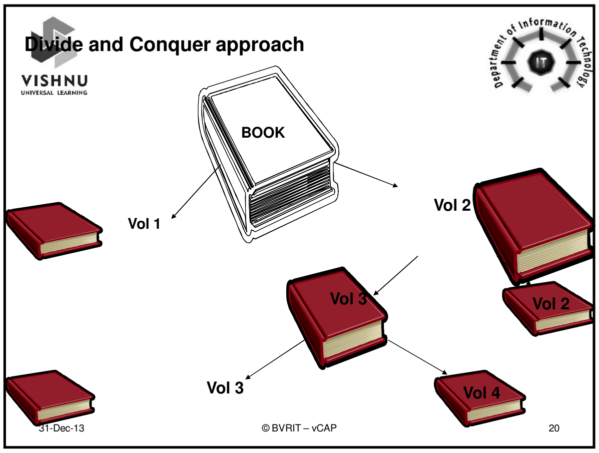


Modular Programming - Manag

Huge Book of 3000 pages

Same book published in several volumes. Easily manageable

31-Dec-13 ©BVRIT – vCAP 19



Divide and Conquer approach

BOOK

Vol 1

Vol 2

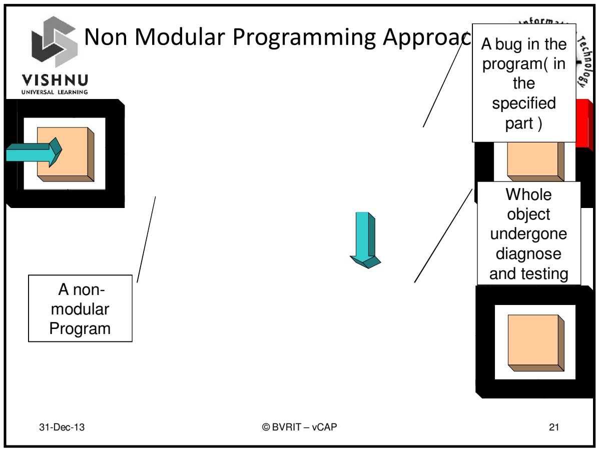
Vol 3

Vol 2

Vol 3

Vol 4

31-Dec-13 ©BVRIT – vCAP 20



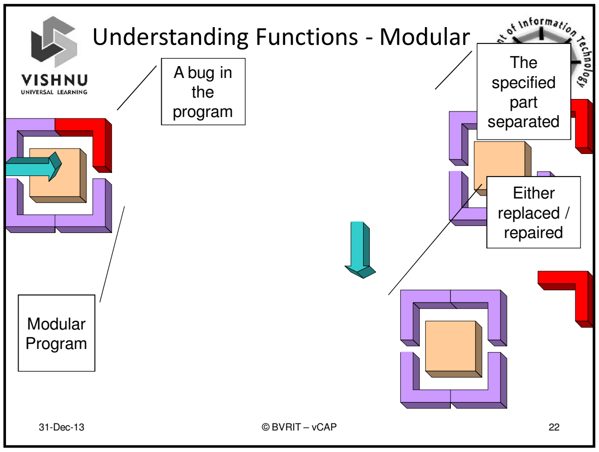
Non Modular Programming Approac

A bug in the program(in the specified part)

Whole object undergone diagnose and testing

A non-modular Program

31-Dec-13 ©BVRIT – vCAP 21



Understanding Functions - Modular

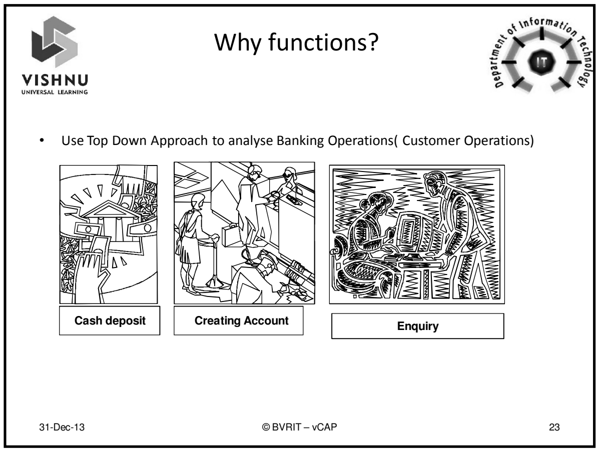
A bug in the program

The specified part separated

Either replaced / repaired

Modular Program

31-Dec-13 ©BVRIT – vCAP 22



Why functions?

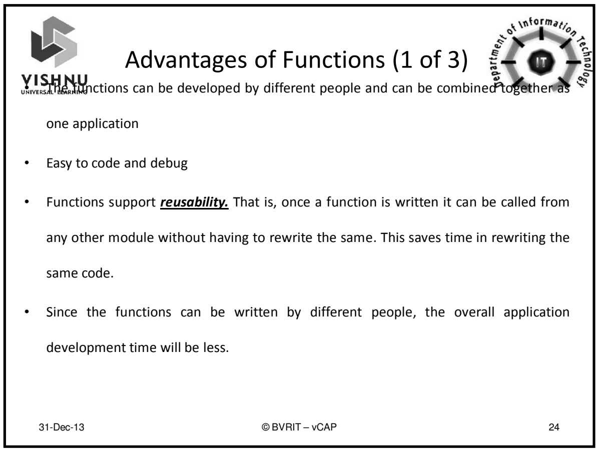
- Use Top Down Approach to analyse Banking Operations(Customer Operations)

Cash deposit

Creating Account

Enquiry

31-Dec-13 ©BVRIT – vCAP 23

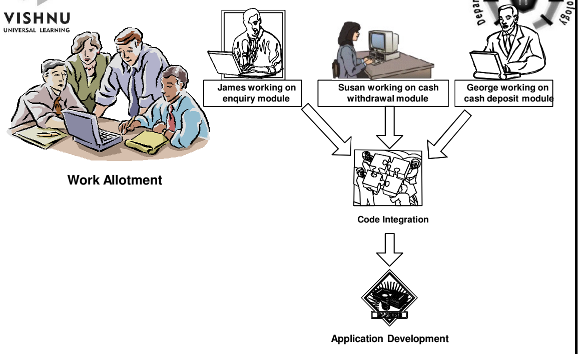


Advantages of Functions (1 of 3)

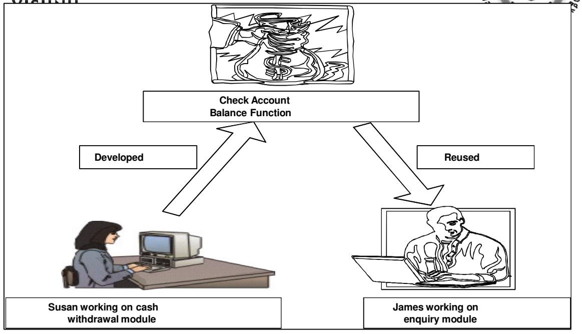
- The functions can be developed by different people and can be combined together as one application
- Easy to code and debug
- Functions support ***reusability.*** That is, once a function is written it can be called from any other module without having to rewrite the same. This saves time in rewriting the same code.
- Since the functions can be written by different people, the overall application development time will be less.

31-Dec-13 ©BVRIT – vCAP 24

Advantages of Functions (2 of 3)


Work Allotment
James working on enquiry module
Susan working on cash withdrawal module
George working on cash deposit module
Code Integration
Application Development

Advantages of Functions (3 of 3)


Check Account Balance Function
Developed
Reused
Susan working on cash withdrawal module
James working on enquiry module

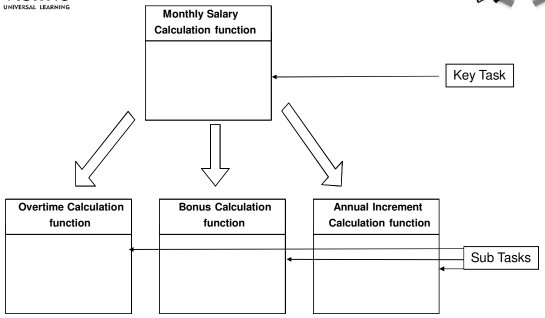
Identifying Functions

- The first step in solving a large problem is identification of sub problems.
- In C programming terms, the sub problems can be viewed as functions
- Once the functions are identified, solving the problem becomes easy.

Identifying Functions

- Problem statement
  - In an automobile company salaries are delayed every month due to the manual calculations of the pay roll.
  - Employee dissatisfaction.
  - Management decided to computerize the operation to remove this delay.
  - i) When an employee joins the company, he/she will be fixed with a monthly salary.
  - Ii) He /she can work overtime and the overtime amount will be added with the salary.
  - Bonus will be announced every year for all categories of employees.
  - In April, employees get the annual increment.

Identifying Functions


Monthly Salary Calculation function
Key Task
Overtime Calculation function
Bonus Calculation function
Annual Increment Calculation function
Sub Tasks

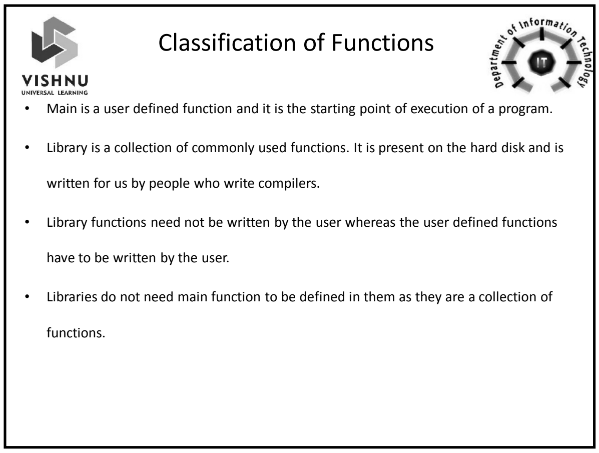
Classification of Functions

- Library functions
  - Defined in the language
  - Provided along with the compiler

Example: printf(), scanf() etc.

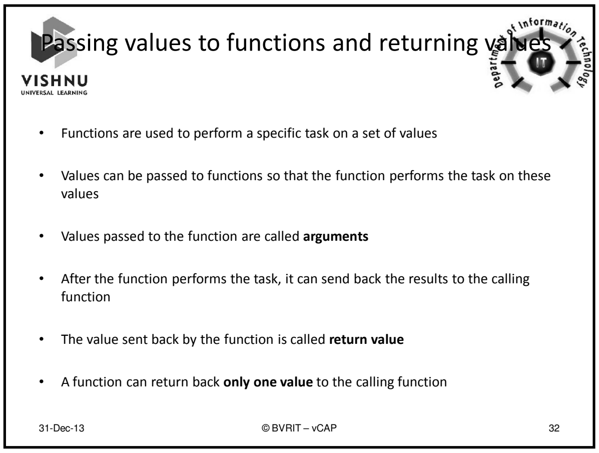
- User Defined functions
  - Written by the user

Example: main() or any other user-defined function



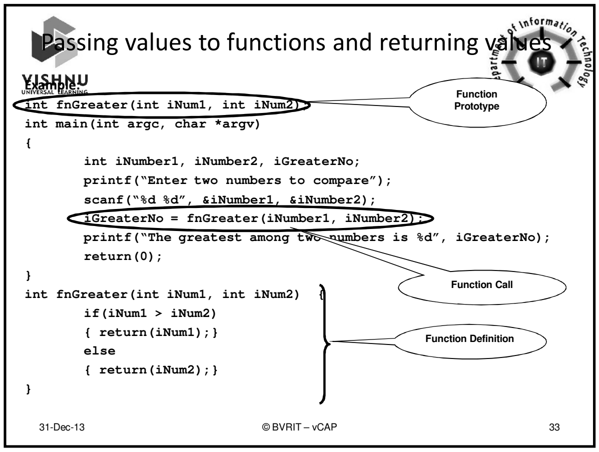
Classification of Functions

- Main is a user defined function and it is the starting point of execution of a program.
- Library is a collection of commonly used functions. It is present on the hard disk and is written for us by people who write compilers.
- Library functions need not be written by the user whereas the user defined functions have to be written by the user.
- Libraries do not need main function to be defined in them as they are a collection of functions.



Passing values to functions and returning values

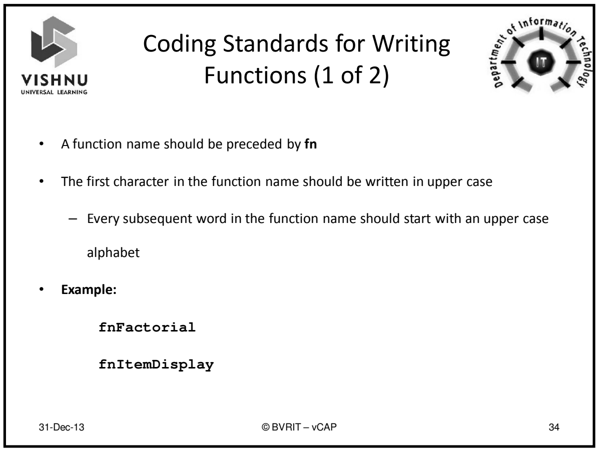
- Functions are used to perform a specific task on a set of values
- Values can be passed to functions so that the function performs the task on these values
- Values passed to the function are called **arguments**
- After the function performs the task, it can send back the results to the calling function
- The value sent back by the function is called **return value**
- A function can return back **only one value** to the calling function



Passing values to functions and returning values

Example:

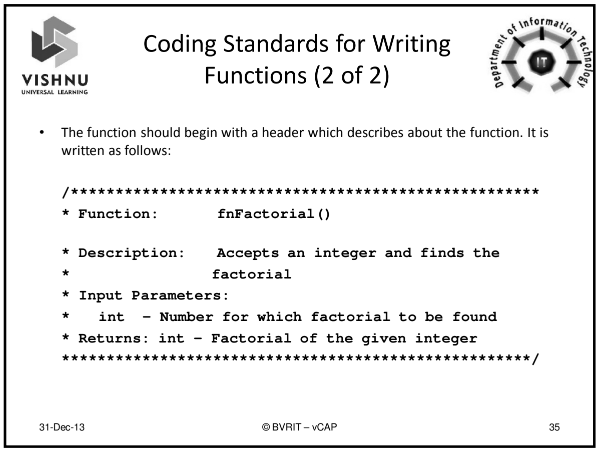
```
int fnGreater(int iNum1, int iNum2);
int main(int argc, char *argv)
{
        int iNumber1, iNumber2, iGreaterNo;
        printf(“Enter two numbers to compare”);
        scanf(“%d %d”, &iNumber1, &iNumber2);
        iGreaterNo = fnGreater(iNumber1, iNumber2);
        printf(“The greatest among two numbers is %d”, iGreaterNo);
        return(0);
}
int fnGreater(int iNum1, int iNum2)  {
        if(iNum1 > iNum2)
        { return(iNum1);}
        else
        { return(iNum2);}
}
```



Coding Standards for Writing Functions (1 of 2)

- A function name should be preceded by **fn**
- The first character in the function name should be written in upper case
  - Every subsequent word in the function name should start with an upper case alphabet
- **Example:**

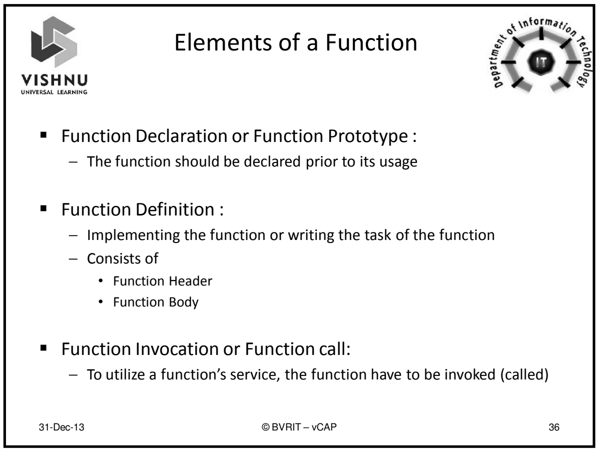
```
fnFactorial
fnItemDisplay
```



Coding Standards for Writing Functions (2 of 2)

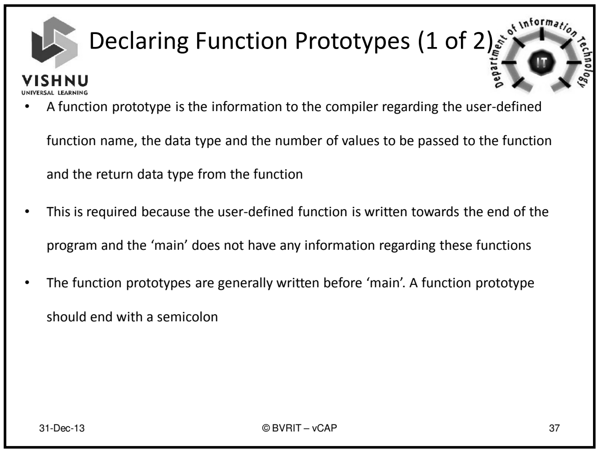
- The function should begin with a header which describes about the function. It is written as follows:

```
/*****************************************************
* Function:       fnFactorial()

* Description:    Accepts an integer and finds the
*                factorial
* Input Parameters:
*   int  - Number for which factorial to be found
* Returns: int - Factorial of the given integer
*****************************************************/
```



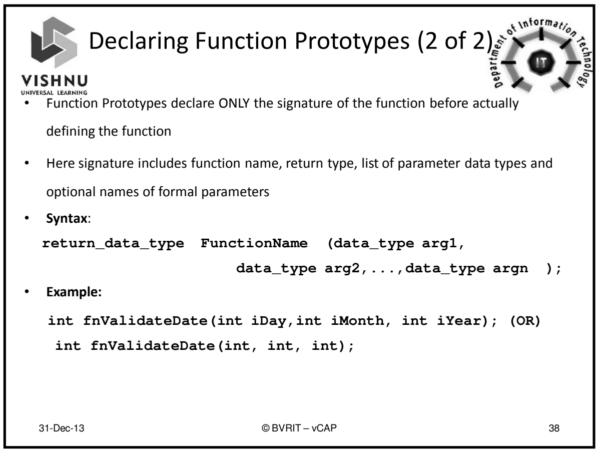
Elements of a Function

- Function Declaration or Function Prototype :
  - The function should be declared prior to its usage
- Function Definition :
  - Implementing the function or writing the task of the function
  - Consists of
    - Function Header
    - Function Body
- Function Invocation or Function call:
  - To utilize a function’s service, the function have to be invoked (called)



Declaring Function Prototypes (1 of 2)

- A function prototype is the information to the compiler regarding the user-defined function name, the data type and the number of values to be passed to the function and the return data type from the function
- This is required because the user-defined function is written towards the end of the program and the 'main' does not have any information regarding these functions
- The function prototypes are generally written before 'main'. A function prototype should end with a semicolon



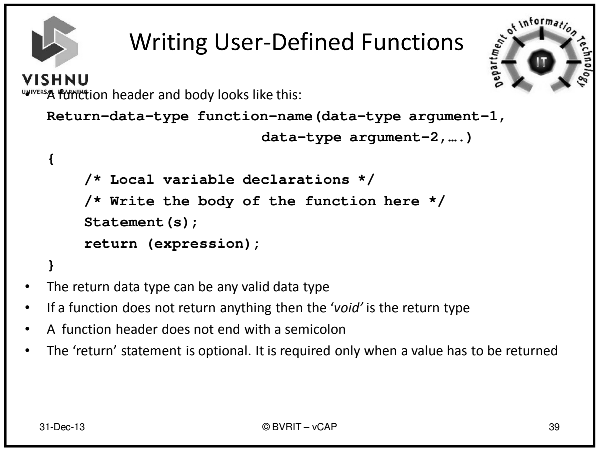
Declaring Function Prototypes (2 of 2)

- Function Prototypes declare ONLY the signature of the function before actually defining the function
- Here signature includes function name, return type, list of parameter data types and optional names of formal parameters
- **Syntax**:

```
return_data_type  FunctionName  (data_type arg1,
                         data_type arg2,...,data_type argn  );
```

- **Example:**

```
int fnValidateDate(int iDay,int iMonth, int iYear); (OR)
 int fnValidateDate(int, int, int);
```

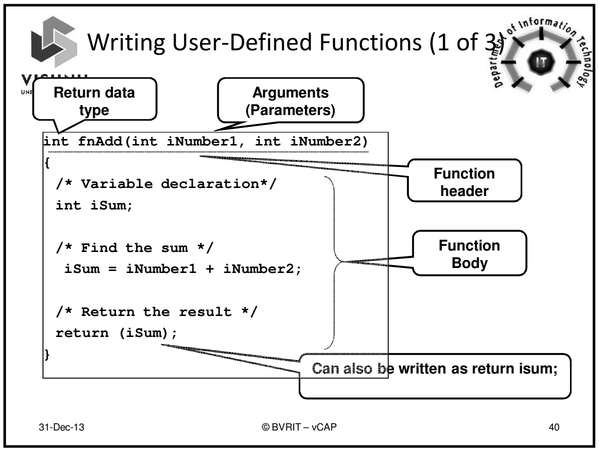


Writing User-Defined Functions

- A function header and body looks like this:

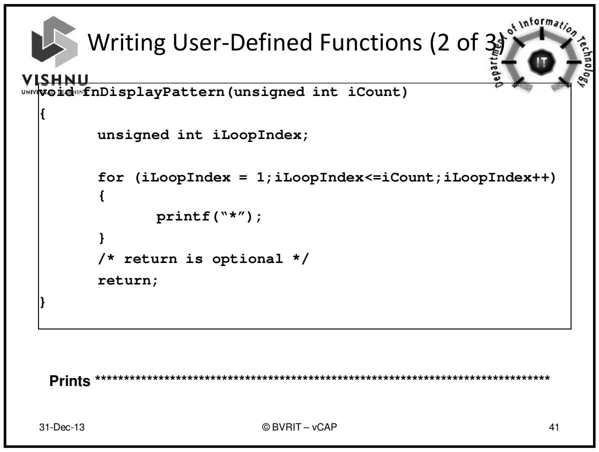
```
Return-data-type function-name(data-type argument-1,
                              data-type argument-2,….)
{
     /* Local variable declarations */
     /* Write the body of the function here */
     Statement(s);
     return (expression);
}
```

- The return data type can be any valid data type
- If a function does not return anything then the '*void'* is the return type
- A function header does not end with a semicolon
- The 'return' statement is optional. It is required only when a value has to be returned



Writing User-Defined Functions (1 of 3)

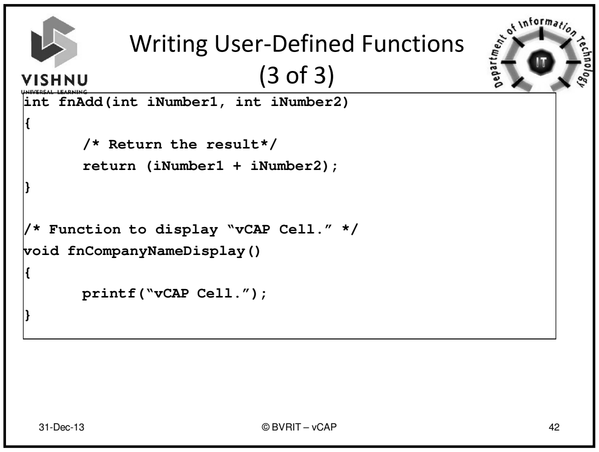
```
int fnAdd(int iNumber1, int iNumber2)
{
 /* Variable declaration*/
 int iSum;

 /* Find the sum */
  iSum = iNumber1 + iNumber2;

 /* Return the result */
 return (iSum);
}
```



Writing User-Defined Functions (2 of 3)

```
void fnDisplayPattern(unsigned int iCount)
{
      unsigned int iLoopIndex;

      for (iLoopIndex = 1;iLoopIndex<=iCount;iLoopIndex++)
      {
             printf("*");
      }
      /* return is optional */
      return;
}
```

Prints **



Writing User-Defined Functions (3 of 3)

```
int fnAdd(int iNumber1, int iNumber2)
{
      /* Return the result*/
      return (iNumber1 + iNumber2);
}

/* Function to display "vCAP Cell." */
void fnCompanyNameDisplay()
{
      printf("vCAP Cell.");
}
```

Returning values

- The result of the function can be given back to the calling functions
- ‘return’ statement is used to return a value to the calling function
- **Syntax:**

```
      return (expression) ;
```

- **Example:**

```
   return(iNumber * iNumber);
    return 0;
    return (3);
    return;
    return (10 * i);
```

Calling User-Defined Functions (1 of 2)

- A function is called by giving its name and passing the required arguments
- The constants can be sent as arguments to functions

```
/* Function is called here */
iResult = fnAdd(10, 15);
```

- The variables can also be sent as arguments to functions

```
int iResult,iNumber1=10, iNumber2=15;
/* Function is called here */
iResult = fnAdd(iNumber1, iNumber2);
```

Calling User-Defined Functions (2 of 2)

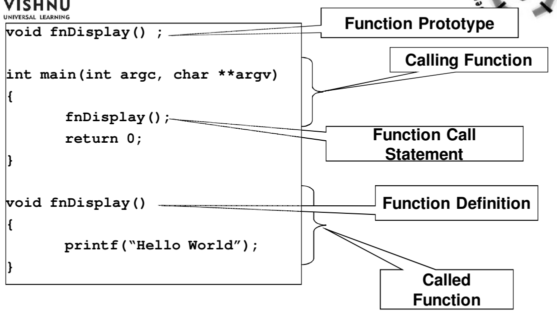
- Calling a function which does not return any value

```
/* Calling a function */
fnDisplayPattern(15);
```

- Calling a function that do not take any arguments and do not return anything

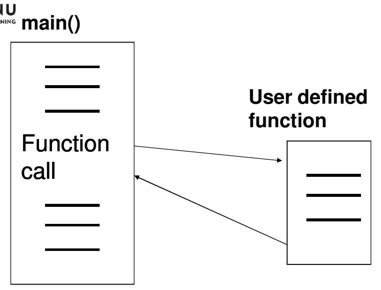
```
/* Calling a function */
fnCompanyNameDisplay();
```

Function Terminologies



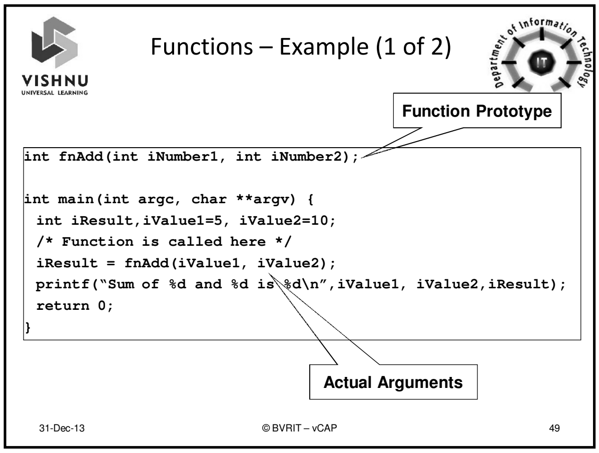
```
void fnDisplay() ;

int main(int argc, char **argv)
{
       fnDisplay();
       return 0;
}

void fnDisplay()
{
       printf(“Hello World”);
}
```

How Functions Work?



Formal and Actual Parameters

- The variables declared in the function header are called as **formal parameters**
- The variables or constants that are passed in the function call are called as **actual parameters**
- The formal parameter names and actual parameters names can be the same or different

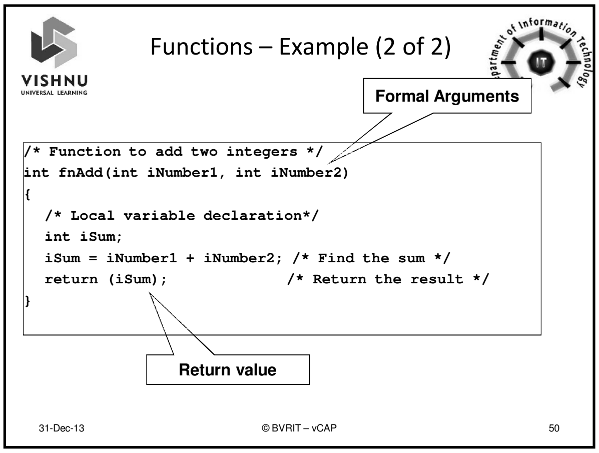


Functions – Example (1 of 2)

Function Prototype

```
int fnAdd(int iNumber1, int iNumber2);

int main(int argc, char **argv) {
 int iResult,iValue1=5, iValue2=10;
 /* Function is called here */
 iResult = fnAdd(iValue1, iValue2);
 printf(“Sum of %d and %d is %d\n”,iValue1, iValue2,iResult);
 return 0;
}
```

Actual Arguments

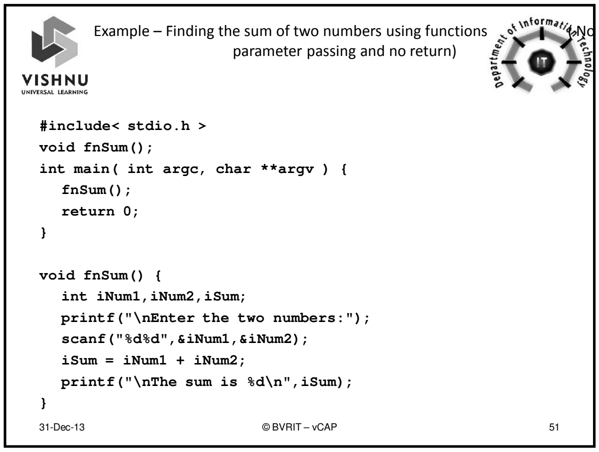


Functions – Example (2 of 2)

Formal Arguments

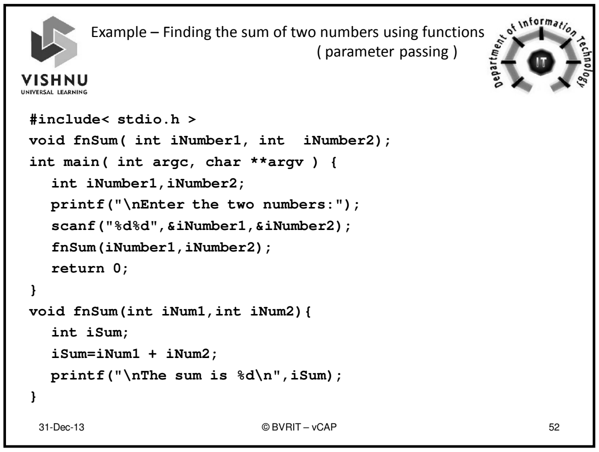
```
/* Function to add two integers */
int fnAdd(int iNumber1, int iNumber2)
{
  /* Local variable declaration*/
  int iSum;
  iSum = iNumber1 + iNumber2; /* Find the sum */
  return (iSum);              /* Return the result */
}
```

Return value



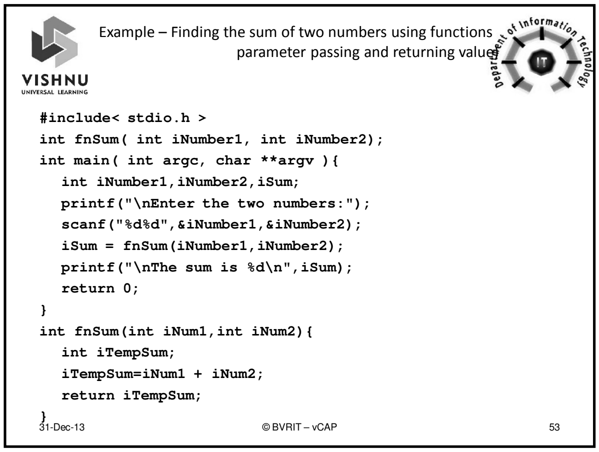
Example – Finding the sum of two numbers using functions (No parameter passing and no return)

```
#include< stdio.h >
void fnSum();
int main( int argc, char **argv ) {
   fnSum();
   return 0;
}

void fnSum() {
   int iNum1,iNum2,iSum;
   printf("\nEnter the two numbers:");
   scanf("%d%d",&iNum1,&iNum2);
   iSum = iNum1 + iNum2;
   printf("\nThe sum is %d\n",iSum);
}
```



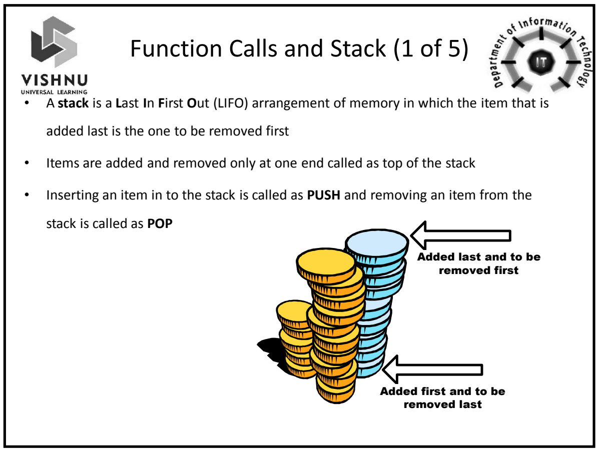
Example – Finding the sum of two numbers using functions (parameter passing)

```
#include< stdio.h >
void fnSum( int iNumber1, int  iNumber2);
int main( int argc, char **argv ) {
   int iNumber1,iNumber2;
   printf("\nEnter the two numbers:");
   scanf("%d%d",&iNumber1,&iNumber2);
   fnSum(iNumber1,iNumber2);
   return 0;
}
void fnSum(int iNum1,int iNum2){
   int iSum;
   iSum=iNum1 + iNum2;
   printf("\nThe sum is %d\n",iSum);
}
```



Example – Finding the sum of two numbers using functions parameter passing and returning value)

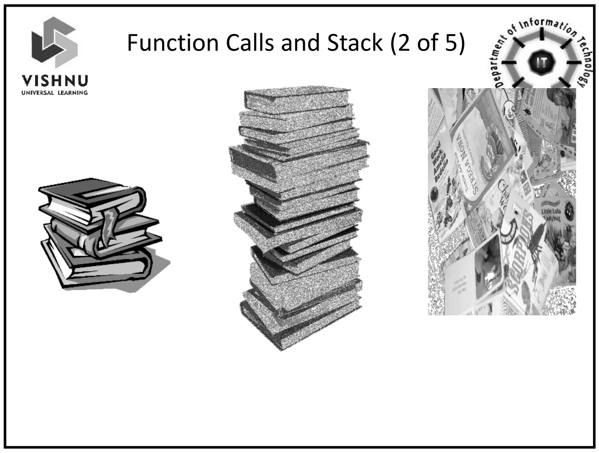
```
#include< stdio.h >
int fnSum( int iNumber1, int iNumber2);
int main( int argc, char **argv ){
   int iNumber1,iNumber2,iSum;
   printf("\nEnter the two numbers:");
   scanf("%d%d",&iNumber1,&iNumber2);
   iSum = fnSum(iNumber1,iNumber2);
   printf("\nThe sum is %d\n",iSum);
   return 0;
}
int fnSum(int iNum1,int iNum2){
   int iTempSum;
   iTempSum=iNum1 + iNum2;
   return iTempSum;
}
```



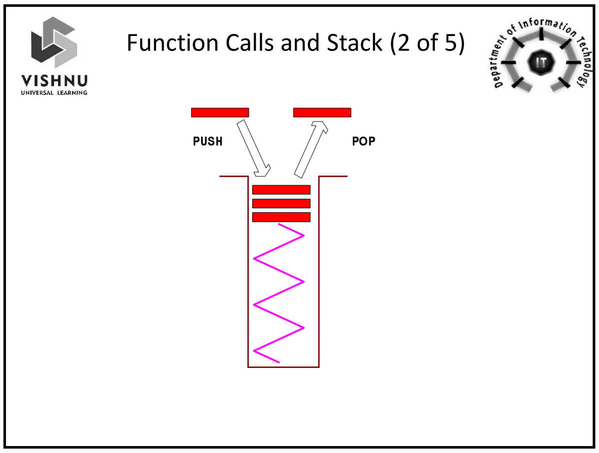
Function Calls and Stack (1 of 5)

- A **stack** is a **L**ast **I**n **F**irst **O**ut (LIFO) arrangement of memory in which the item that is added last is the one to be removed first
- Items are added and removed only at one end called as top of the stack
- Inserting an item in to the stack is called as **PUSH** and removing an item from the stack is called as **POP**

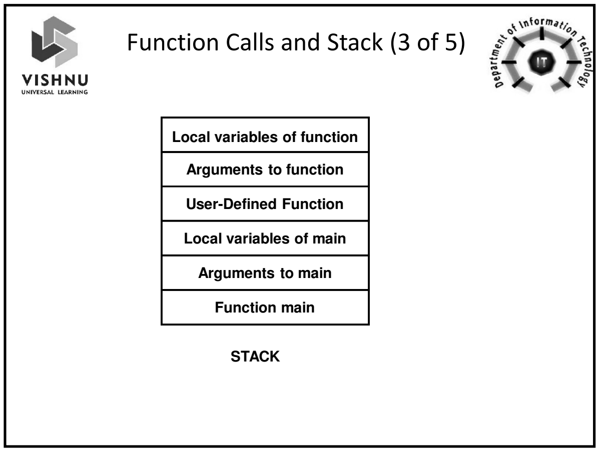
Added last and to be removed first
Added first and to be removed last



Function Calls and Stack (2 of 5)



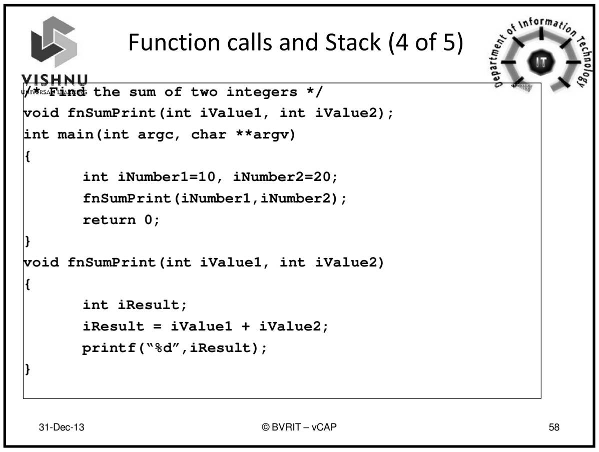
Function Calls and Stack (2 of 5)



Function Calls and Stack (3 of 5)

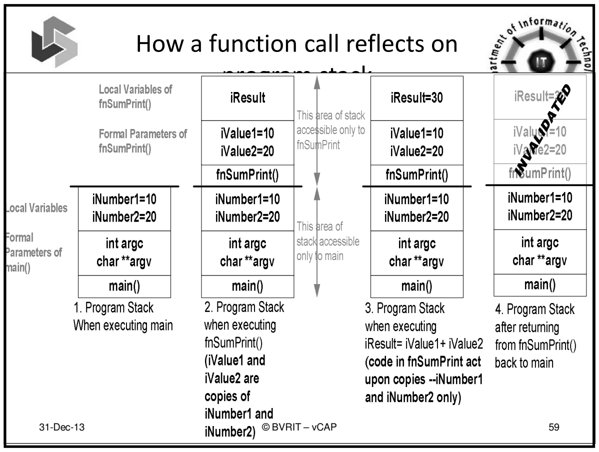
Local variables of function
Arguments to function
User-Defined Function
Local variables of main
Arguments to main
Function main

STACK



Function calls and Stack (4 of 5)

```
/* Find the sum of two integers */
void fnSumPrint(int iValue1, int iValue2);
int main(int argc, char **argv)
{
       int iNumber1=10, iNumber2=20;
       fnSumPrint(iNumber1,iNumber2);
       return 0;
}
void fnSumPrint(int iValue1, int iValue2)
{
       int iResult;
       iResult = iValue1 + iValue2;
       printf("%d",iResult);
}
```



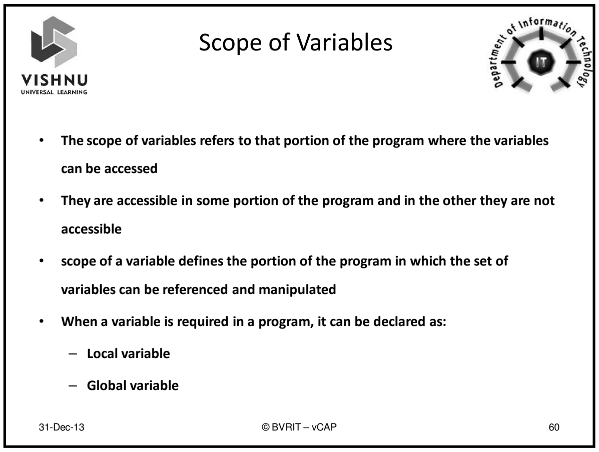
How a function call reflects on program stack

1. Program Stack When executing main

2. Program Stack when executing fnSumPrint() **(iValue1 and iValue2 are copies of iNumber1 and iNumber2)**

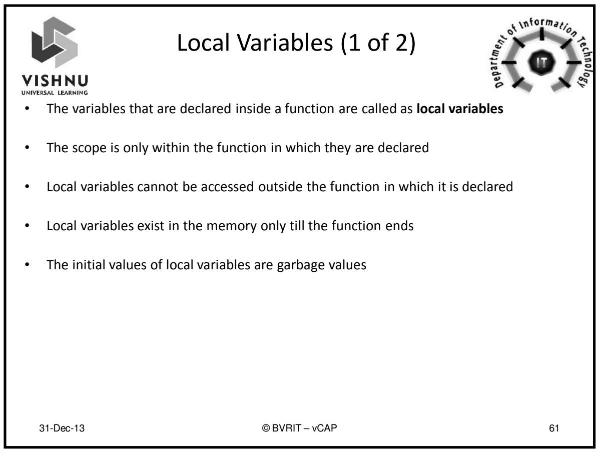
3. Program Stack when executing iResult= iValue1+ iValue2 **(code in fnSumPrint act upon copies --iNumber1 and iNumber2 only)**

4. Program Stack after returning from fnSumPrint() back to main



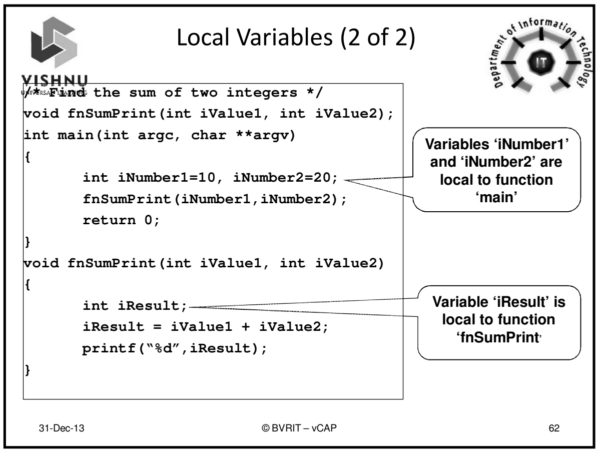
Scope of Variables

- The scope of variables refers to that portion of the program where the variables can be accessed
- They are accessible in some portion of the program and in the other they are not accessible
- scope of a variable defines the portion of the program in which the set of variables can be referenced and manipulated
- When a variable is required in a program, it can be declared as:
  - Local variable
  - Global variable



Local Variables (1 of 2)

- The variables that are declared inside a function are called as **local variables**
- The scope is only within the function in which they are declared
- Local variables cannot be accessed outside the function in which it is declared
- Local variables exist in the memory only till the function ends
- The initial values of local variables are garbage values

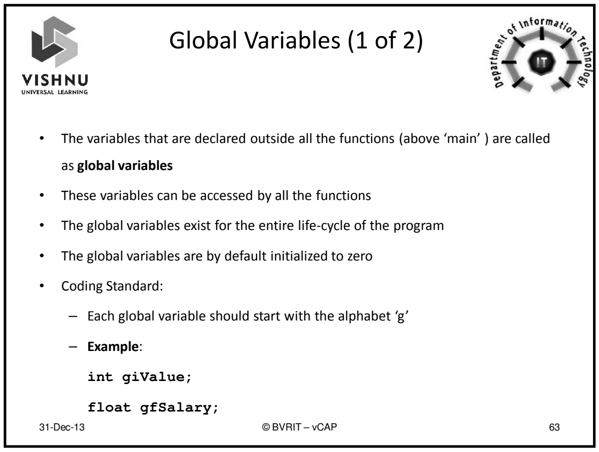


Local Variables (2 of 2)

```
/* Find the sum of two integers */
void fnSumPrint(int iValue1, int iValue2);
int main(int argc, char **argv)
{
       int iNumber1=10, iNumber2=20;
       fnSumPrint(iNumber1,iNumber2);
       return 0;
}
void fnSumPrint(int iValue1, int iValue2)
{
       int iResult;
       iResult = iValue1 + iValue2;
       printf(“%d”,iResult);
}
```

Variables ‘iNumber1’ and ‘iNumber2’ are local to function ‘main’

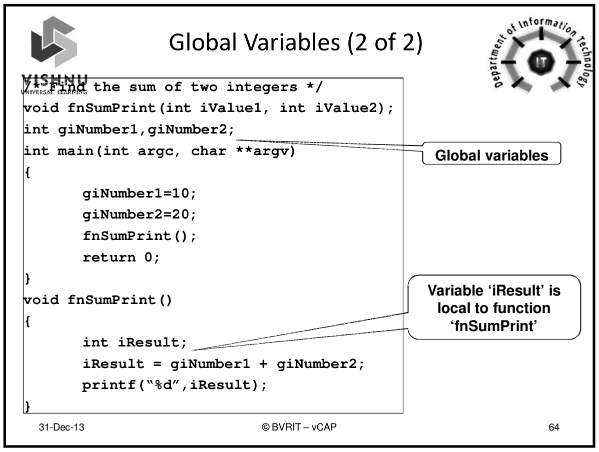
Variable ‘iResult’ is local to function ‘fnSumPrint’



Global Variables (1 of 2)

- The variables that are declared outside all the functions (above ‘main’) are called as **global variables**
- These variables can be accessed by all the functions
- The global variables exist for the entire life-cycle of the program
- The global variables are by default initialized to zero
- Coding Standard:
  - Each global variable should start with the alphabet ‘g’
  - **Example**:

```
int giValue;
float gfSalary;
```

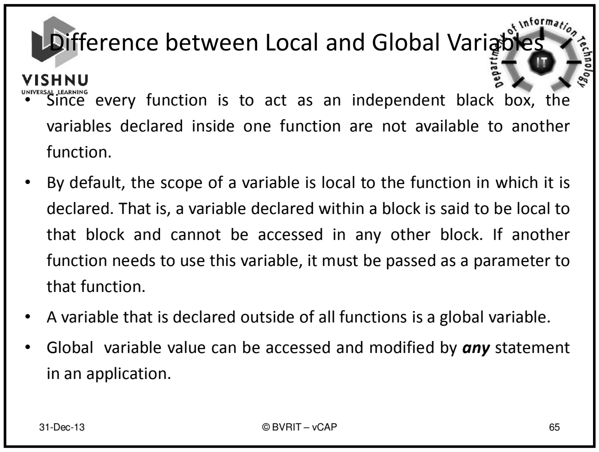


Global Variables (2 of 2)

```
/* Find the sum of two integers */
void fnSumPrint(int iValue1, int iValue2);
int giNumber1,giNumber2;
int main(int argc, char **argv)
{
       giNumber1=10;
       giNumber2=20;
       fnSumPrint();
       return 0;
}
void fnSumPrint()
{
       int iResult;
       iResult = giNumber1 + giNumber2;
       printf(“%d”,iResult);
}
```

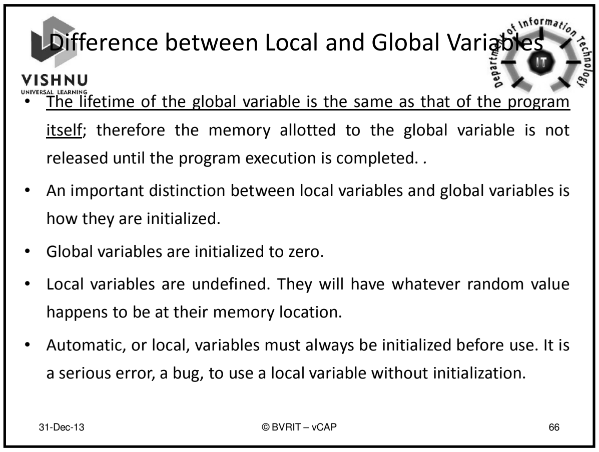
Global variables

Variable ‘iResult’ is local to function ‘fnSumPrint’



Difference between Local and Global Variables

- Since every function is to act as an independent black box, the variables declared inside one function are not available to another function.
- By default, the scope of a variable is local to the function in which it is declared. That is, a variable declared within a block is said to be local to that block and cannot be accessed in any other block. If another function needs to use this variable, it must be passed as a parameter to that function.
- A variable that is declared outside of all functions is a global variable.
- Global variable value can be accessed and modified by ***any*** statement in an application.

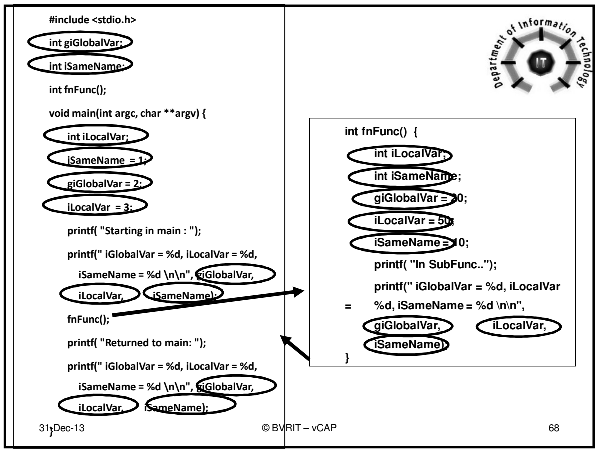


Difference between Local and Global Variables

- The lifetime of the global variable is the same as that of the program itself; therefore the memory allotted to the global variable is not released until the program execution is completed. .
- An important distinction between local variables and global variables is how they are initialized.
- Global variables are initialized to zero.
- Local variables are undefined. They will have whatever random value happens to be at their memory location.
- Automatic, or local, variables must always be initialized before use. It is a serious error, a bug, to use a local variable without initialization.

Difference between Local and Global Variables

- When inside a function, a local variable has the same name as a global variable (iSameName, for example), the local variable gets precedence to the global variable.
- **Storing variables - Stack and Heap**
- When functions are in execution, memory is allocated from the stack for variables that are referenced in a function. This storage is released as soon as the function completes the execution.
- The variables declared inside a function (i.e. all the local variables) are allocated on the stack, as part of the function's stack frame.
- This stack frame is wiped out once the function exits. All the local variables go away when the stack frame is wiped out.
- Global variables, that are visible to every single function in the program, are stored on the heap memory. Since they are accessible to every program the lifetime of global variables is the lifetime of the program.



```
#include <stdio.h>
int giGlobalVar;
int iSameName;
int fnFunc();
void main(int argc, char **argv) {
    int iLocalVar;
    iSameName  = 1;
    giGlobalVar = 2;
    iLocalVar  = 3;
    printf( "Starting in main : ");
    printf(" iGlobalVar = %d, iLocalVar = %d,
      iSameName = %d \n\n", giGlobalVar,
      iLocalVar, iSameName);
    fnFunc();
    printf( "Returned to main: ");
    printf(" iGlobalVar = %d, iLocalVar = %d,
      iSameName = %d \n\n", giGlobalVar,
      iLocalVar, iSameName);
}
```

```
int fnFunc()  {
        int iLocalVar;
        int iSameName;
        giGlobalVar = 20;
        iLocalVar = 50;
        iSameName = 10;
        printf( "In SubFunc..");
        printf(" iGlobalVar = %d, iLocalVar
=       %d, iSameName = %d \n\n",
        giGlobalVar,        iLocalVar,
        iSameName);
}
```

```
#include <stdio.h>
int giGlobalVar;
int iSameName;
int fnFunc();
void main(int argc, char **argv) {
    int iLocalVar;
    iSameName  = 1;
    giGlobalVar = 2;
    iLocalVar  = 3;
    printf( "Starting in main : ");
    printf(" iGlobalVar = %d, iLocalVar = %d,
      iSameName = %d \n\n", giGlobalVar,
      iLocalVar, iSameName);
    fnFunc();
    printf( "Returned to main: ");
    printf(" iGlobalVar = %d, iLocalVar = %d,
      iSameName = %d \n\n", giGlobalVar,
      iLocalVar, iSameName);
}
```

```
int fnFunc()  {
        int iLocalVar;
        int iSameName;
        giGlobalVar = 20;
        iLocalVar = 50;
        iSameName = 10;
        printf( "In SubFunc..");
        printf(" iGlobalVar = %d, iLocalVar
=       %d, iSameName = %d \n\n",
        giGlobalVar,
        iLocalVar,iSameName);
}
```

Disadvantages of Global Variables

- **Lifetime of global variables is throughout the program**
  - **Hence usage of global variables leads to wastage of memory**
- **Scope of the global variable is throughout the program**
  - **Hence more than one function can modify the value of the global variable. This makes debugging difficult.**

What is the output of the following code snippet?

```
int giGlobal ;

int main(int argc, char **argv) {
        int iLocal;
        printf(" Value of Local = %d \n,
           Value of Global = %d", iLocal,
 giGlobal);
        return 0;
}
```

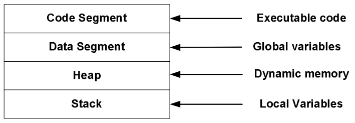
The output is:

Value of Local = <some garbage value>

Value of Global = 0

Program stack and heap

- During execution of a program, the storage of program and data is as follows:
  - The executable code is stored into the code /Text segment
  - The global variables are stored into data segment
  - The heap memory is used for dynamic memory allocation The local variables are stored into the stack
- Heap: A section of memory within the user job area that provides a capability for dynamic allocation (Not discussed in this course)



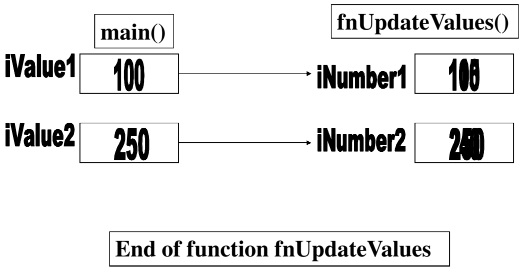
Parameter Passing Techniques

- When a function is called and if the function accepts some parameters, then there are two ways by which the function can receive parameters
  - Pass by value
  - Pass by reference

Pass by Value

- When parameters are passed from the called function to a calling function, the value of the actual argument is copied onto the formal argument
- Since the actual parameters and formal parameters are stored in different memory locations, the changes in formal parameters do not alter the values of actual parameters

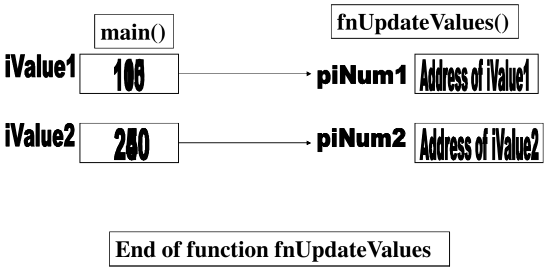
Pass by Value



Pass by Reference

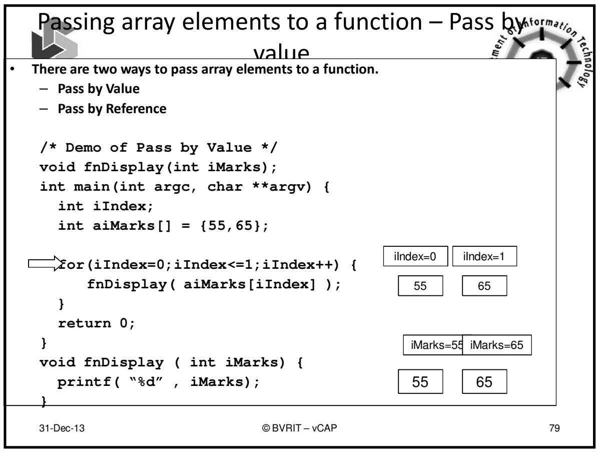
- Addresses of actual parameters are passed
- The function should receive the addresses of the actual parameters through pointers
- The actual parameters and formal parameters are referencing the same memory location, so the changes that are made become permanent

Pass by Reference (4 of 5)



Difference between pass by value and pass by reference

Pass by value	Pass by reference
Consumes more memory space because formal parameter also occupies memory space.	Consumes less memory space. Because irrespective of the actual arguments data type, each pointer occupies only 4 bytes.
Takes more time for execution, because the values are copied	Takes less time because no values are copied



Passing array elements to a function – Pass by value

- **There are two ways to pass array elements to a function.**
  - **Pass by Value**
  - **Pass by Reference**

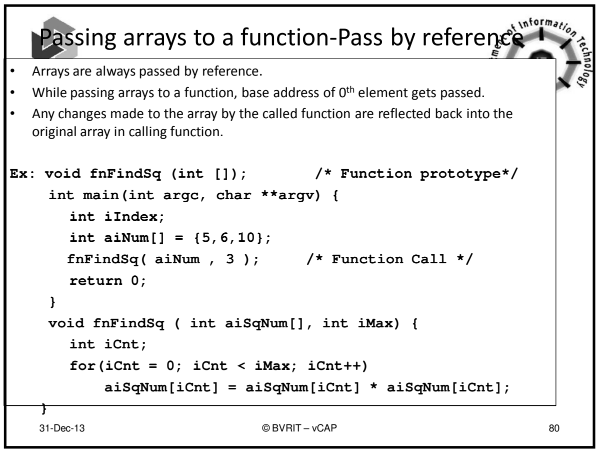
```
/* Demo of Pass by Value */
void fnDisplay(int iMarks);
int main(int argc, char **argv) {
  int iIndex;
  int aiMarks[] = {55,65};

  for(iIndex=0;iIndex<=1;iIndex++) {
      fnDisplay( aiMarks[iIndex] );
  }
  return 0;
}
void fnDisplay ( int iMarks) {
  printf( "%d" , iMarks);
}
```

iIndex=0 iIndex=1

55 65

iMarks=55 iMarks=65

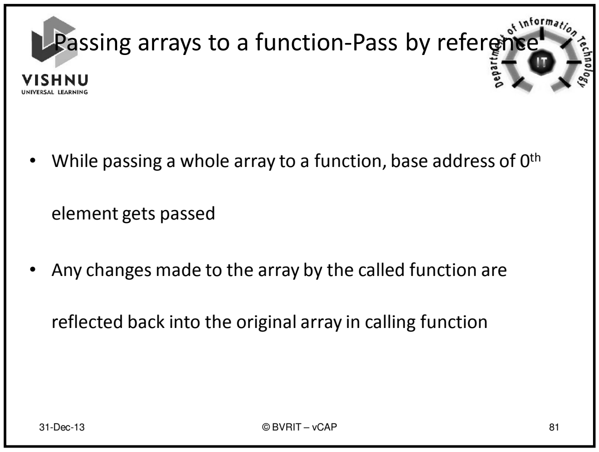
55 65



Passing arrays to a function-Pass by reference

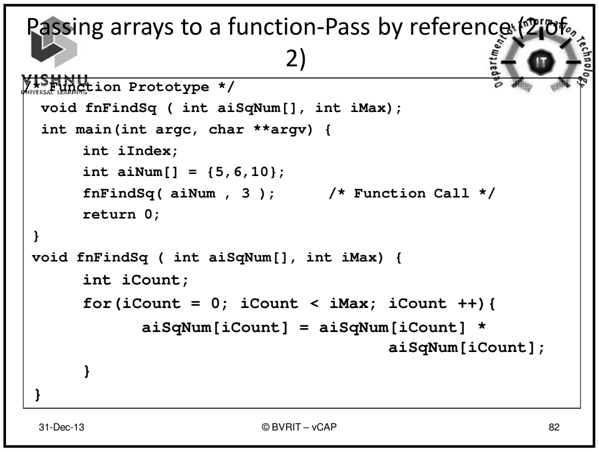
- Arrays are always passed by reference.
- While passing arrays to a function, base address of 0th element gets passed.
- Any changes made to the array by the called function are reflected back into the original array in calling function.

```
Ex: void fnFindSq (int []);        /* Function prototype*/
     int main(int argc, char **argv) {
        int iIndex;
        int aiNum[] = {5,6,10};
       fnFindSq( aiNum , 3 );      /* Function Call */
        return 0;
     }
     void fnFindSq ( int aiSqNum[], int iMax) {
        int iCnt;
        for(iCnt = 0; iCnt < iMax; iCnt++)
            aiSqNum[iCnt] = aiSqNum[iCnt] * aiSqNum[iCnt];
    }
```



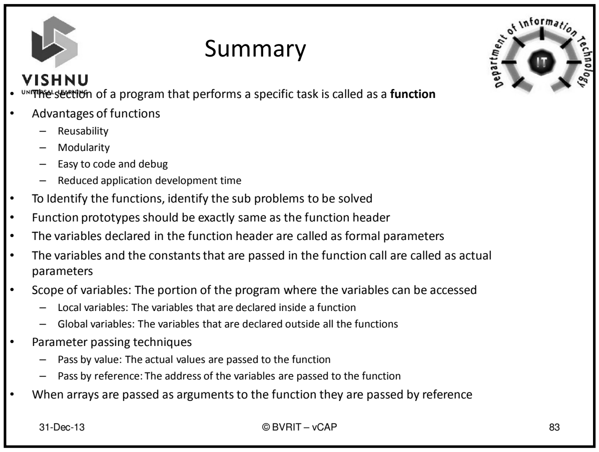
Passing arrays to a function-Pass by reference

- While passing a whole array to a function, base address of 0th element gets passed
- Any changes made to the array by the called function are reflected back into the original array in calling function



Passing arrays to a function-Pass by reference (2 of 2)

```
/* Function Prototype */
 void fnFindSq ( int aiSqNum[], int iMax);
 int main(int argc, char **argv) {
      int iIndex;
      int aiNum[] = {5,6,10};
      fnFindSq( aiNum , 3 );      /* Function Call */
      return 0;
}
void fnFindSq ( int aiSqNum[], int iMax) {
      int iCount;
      for(iCount = 0; iCount < iMax; iCount ++){
             aiSqNum[iCount] = aiSqNum[iCount] *
                                      aiSqNum[iCount];
      }
}
```



Summary

- The section of a program that performs a specific task is called as a **function**
- Advantages of functions
  - Reusability
  - Modularity
  - Easy to code and debug
  - Reduced application development time
- To Identify the functions, identify the sub problems to be solved
- Function prototypes should be exactly same as the function header
- The variables declared in the function header are called as formal parameters
- The variables and the constants that are passed in the function call are called as actual parameters
- Scope of variables: The portion of the program where the variables can be accessed
  - Local variables: The variables that are declared inside a function
  - Global variables: The variables that are declared outside all the functions
- Parameter passing techniques
  - Pass by value: The actual values are passed to the function
  - Pass by reference: The address of the variables are passed to the function
- When arrays are passed as arguments to the function they are passed by reference

Vishnu Career Advancement Program

C Programming Students Manual

TABLE OF CONTENTS

1. INTRODUCTION

The goal of this standards document is to promote error free source code that is readable, usable, maintainable, and portable. This guide defines a particular style, offers some justification for it, and presents examples where appropriate.

This guide is designed to serve as a reference for experienced library developers, and to acquaint new developers with the standard.

Each project may be segregated into functional phases, depending on customer requirements and development sequencing.

2. C LANGUAGE

2.1 ANSI C

All code must be composed of valid ANSI C statements with no reliance on particular language constructs which might cause platform/compiler dependence.

3. NAMING CONVENTIONS

3.1 Program files

A 8.3 character file-name format can be used to name all program files. Program files include:

Source files
Header files

3.1.1 Naming source files

The initial 8 characters for source files can be made up as follows:
<application name><module name>

Application names should contain a **maximum** of 4 small letters (e.g. xadm).

Module names can be arrived at keeping in mind the following points :

The module name could clearly identify the functional area which the module addresses. E.g.: xadmio.c, mdmSave.c, mdmClone.c, mdmu.c.

The module name could identify a particular user interface/messaging object which the module addresses. E.g.: mdmIcon.c, udmmBar.c, udmmFont.c, mdmFldId.c.

The module name could just describe if it deals with user interface or backend. E.g.: udmmBknd.c

The main module (containing function main() or entry point to the application) should be <application_name>.c. E.g.. xadm.c, mdm.c.

The naming conventions for each of these kinds of applications/libraries are listed in the following tables.

Table 3 - Source file naming convention for applications/libraries

Name	Description
xadm.c	Main module
xadmio.c	Module containing functions performing I/O using API.
xadmp.c	Module containing page routines supported by the application.
xadmu.c	Module containing utility routines (typically to assist I/O functions defined in xadmio.c - processing of data after I/O)
udmmBknd.c	Module containing code for interfacing with the application infrastructure.

Note: The link between function names and the source file names should be maintained, so that given a function name it is easy to determine which source file contains its definition.

E.g.: xadm which is an API-based application could possibly have a file named xadmio.c (all the input/output routines) . So all the function names could start as xadmio_readline( ).

3.1.2 Header files

The initial 8 characters for header files should be made up as follows:
<application name><extension>

Application names should contain a **maximum** of 4 small letters (e.g. xadm)

There should always be a header file called <application_name>.h. All the other header files can be included via this header file.

Extensions will be chosen to clearly indicate what the header file contains.

The header file naming conventions for pure API-based applications/libraries are given in the table below:

Table 2 : Header file naming convention for API applications/libraries (short filenames)

Name	Description
xadm_d	Definition file containing structure definitions and typedefs for the structures
xadm_c	Constants and macros
xadm_f	Function Prototypes
xadm_p	Portability File
An optional '_p' can be appended to '_d', '_c' and '_f' extensions to indicate 'portability' related header files.	

3.2 Functions

- Function name can begin with module name followed by a description (e.g. xadm_save_all, mdm_init_jazz_engine). The general logic to be applied while naming functions is

```
<application + module_name>_<operation>_<object>
```

- Only functions which are not called explicitly anywhere can begin without a module name. Typically notify ,issue functions ,event handlers etc. which are assigned to function pointers or are called intrinsically come under this category. (e.g. set_domain_background_color_issue).

- Function Declaration—External to File

Functions called from *outside* of a file must be defined by prototypes in an include file (for that file). This implies that prototypes should never occur in C source files (`.c` files); instead, the `.c` file should `#include` the appropriate include file. For example, if the file `cashflow.c` defines the functions `GtoFree CFL and Gto NewCFL,` the include file `cashflow.h` should contain:

```
TcashFlowList  *GtoNewCFL
                       (TDate  *dates,   /* (I) Dates */
                           double *amounts,  /*(I) Amounts */
                           int       numItems);    /*(I)Length */
                           void GtoFreeCFL (TCashFlowList *);
                              /* Destructor */
```

- Code Reuse

  Any time there is a need for more than a couple lines of code in more than one place, the code must be placed in one function or macro which is then called from multiple places.

- Function Size

  In general, functions must not be longer than a page or two. Nesting of `for, while, do,` and `if` statements should not be more than four levels deep.

- Function Order Within a File

  Within a file, higher level functions (those which call other functions) must come first.

3.3 Variables

- All variables are to be named in Hungarian Notation using alpha-numeric characters only. The data type is prefixed to the variable name based on the following table :

Table 1 : C variable naming convention

Prefix	Data type	Example
1. i	1. int (signed and unsigned)	1. iIndex
2. c	2. char (signed and unsigned)	2. cOperator
3. f	3. function	3. fButtonNotify
4. d	4. double	4. dAskPrice
5. s	5. structure or typedef structure	5. sTradeGroup, sEnv
6. p	6. pointer	6. pHndl
7. pts	7. pointer to 'type defined' structure	7. ptsTradeGroup
8. pc	8. pointer to character array	8. pcCharacterArray
9. pd	9. pointer to double	9. pdBidPrice
10. pi	10. pointer to integer	10. piIndexToArray
11. pv	11. pointer to void	11. pvVoid
12. a	12. function arguments whose value will be returned to its caller.	12. aHndl
13. ac	13. array of char or address of char	13. acOperator
14. ai	14. array of integers or address of integer	14. aiErrorCode
15. ad	15. array of double or address of double	15. adAskPrice
16. ap	16. array of pointers or address of pointer	16. apNameList

For register variables, add 'Reg' after the prefix (eg. iRegLoopCount)

4. DATA STRUCTURES

4.1 Define Structures as Typedefs

All structures must be defined as a `typedef.` For example:

```
typedef struct
{
     int       fNumItems;
     TDate     *fArray;
}    TDateList;
```

4.2 Structure Tags

All structures must have a tag which names the structure preceded by a single underscore. In other words, the previous example should really look like this:

```
typedef struct _TDateList     /* Tag here */
{
     int       fNumItems;
     TDate     *fArray;
}  TDateList;
```

5. PROGRAMMING CONVENTIONS

5.1 Source files

The source file structure should generally adhere to the following layout :

Comment block for module description (see section : 6.1)

All source files should be surrounded by

```
#ifndef  <source_file_name>_C_INCLUDED   /* eg. MDM_C_INCLUDED */
#define <source_file_name>_C_INCLUDED
:
:
#endif                                   /* At the end of file */
```

#include header files

Macros (#defines) block to define all macros specific to this source module

Static Globals block. The order is C data types, application data types followed by user defined data types

Static function prototypes block

Functions definitions.

5.2 Header Files

All header files should be surrounded by

```
#ifndef  <header_file_name>_H_INCLUDED   /* eg. OS_H_INCLUDED */
#define <header_file_name>_H_INCLUDED
:
:   (contents of header file)
#endif
```

5.3 Variables

The following conventions should be followed while naming and locating the C variables :

Variables should be declared individually, one per line.

Correct	`int  iIndex;`
	`int  iSeconds;`

Incorrect	`int  iIndex,`
	`iSeconds;`

Variables should be named as defined in section 3.3

The format for defining pointers is :

<type><space>*<one or more spaces><pointer variable>;

E.g.:

```
int *    pCode;
char *   pcBuf;
```

Static variables to be defined in the source files only

Global variables should be always be defined as

```
       EXTERN  struct tsNCharcb sOpenRoutineName;
```

where EXTERN is defined as

```
#ifdef    <application_name>_C_INCLUDED
#define   EXTERN
#elseif
#define   EXTERN    extern
#endif
```

Global variables should not be initialized during declaration

Global variables should be initialized separately in a initialization routine

Initialize only one variable per statement.

Correct	iIndex = 0;
	iSeconds = 0;

Incorrect	iIndex = iSeconds = 0;

Separate the “tokens” in the intended manner
e.g. write y = x / *p; rather than y=x/*p;

(*p is the value pointed to by p, in the second case everything beyond x is treated as comment and the intent is lost)

Do not assume automatic initialization of Global variables

Avoid using static variables inside functions unless it is absolutely necessary

Register variables should be used only for counters for large loops. Preferably let the compiler handle register optimization

Explicitly modify variables which occur more than once in one statement; not as part of the statement itself

Correct	iXXX = piYYY[iIndex] + piZZZ[iIndex];
	iIndex++;

Incorrect	iXXX = piYYY[iIndex] + piZZZ[**iIndex++**];

5.4 Functions

The following conventions should be followed while writing functions

Prototypes of static functions should be included in the source files only

Arguments should be listed one per line in a function’s declaration and in its prototype

Example:

```
  int read_emp_all (
                        char acEmpNo[],
                        char  acEmpName[],
                        float  iEmpSalary
                        )
```

When a call to a function spans more than one line, each argument should be placed on its own line

Upon success, a function should return an int whose value is set to OK, otherwise it should return an int whose value is set to NOT_OK

The last argument to a function should be an int * for which dereferencing is valid only when the function returns NOT_OK

Return arguments should be enclosed in parenthesis

Functions should be written in pairs - one to **do** something an the other to **undo** it.

5.5 Braces and Indentation

Left braces should appear five spaces indented from the beginning of the previous line

A right brace should appear in the same column as its matching left brace

Other statements should appear on the same line as a brace except at function level where the left brace appears on the first column and the statements appear five spaces indented from the left brace

When multiple arguments of a function call are written one per line, all the arguments should appear on the same column as the first argument. For pointer data types, the * is placed immediately after the data type with a single space between them. The variable names should be aligned to the same column.

```
int xadm_add_to_socket_list( tsDialogInfo * ptsDialogInfo,
  tsNCharcb *            ptsSocketName,
  tsNCharcb *            ptsSocketAddr,
  int *                  aiCode)
```

5.6 Other Issues

The following issues should be observed carefully to write portable and understandable code

Do not assume the sizes of various data types. Always use the **sizeof** operator. An integer on a 16-bit operating system may be 2 bytes while on a 32-bit operating system, it may be 4 bytes.

Use parentheses judiciously to make the code more readable
for e.g.

`*sStatus.piErrorCode` is less readable than
`*(sStatus.piErrorCode)`

If a statement appears over-parenthesized, break it up into multiple statements

`goto` statements should not be used

Do not use "break" to come out of loops; use flags instead

Always handle `default` in `switch` statements. Every `case` statement block should have a `break` statement.

Correct	`switch(iItemType) {`
	`     case TYPE_A :`
	`          ...`
	`          break;`
	`     case TYPE_B :`
	`          ...`
	`          break;`
	`     default :`
	`          ...`
	`          break;`
	`     }`

Incorrect	`switch (iItemType) {`
	`     case TYPE_A :`
	`          ...`
	`          break;`
	`     case TYPE_B :`
	`          ...`
	`     }`

Avoid magic numbers. Always use `#define` or `const` to represent such numbers

Correct	`#define   MAX_CLASS_SIZE 36`
	`...`
	`if (iClassSize < MAX_CLASS_SIZE)`
	`     ...`

Incorrect	`if (iClassSize < 36)`
	`     ...`

For frequently used strings, use a `const char *`. This is preferable to using `#define` macro to declare constant strings.

Correct	`const char     *pPrompt = “Press any key to continue”;`
	`...`
	`printf(pPrompt);`
	`...`
	`printf(pPrompt);`
	`...`
	`printf(pPrompt);`

Incorrect	`...`
	`printf(“Press any key to continue”);`
	`...`
	`printf(“Press any key to continue”);`
	`...`
	`printf(“Press any key to continue”);`

6. DOCUMENTATION

Documentation is to be provided for the following purposes :

6.1 Source header and modification history

All source and header files will contain a section providing information about the source or the header file. The format is given below

```
/* File        : <filename>
 *
 * Description : <description>
 *
 * Author      : <author> (Infosys Tech. Ltd., Bangalore )
 *
 * Started On  : 6 June 1996
 *
 * Modification History :
 *
 *  Date       Name          Change/Description
 * ----------------------------------------------------------
 * DDMMMYYYY xxxxxxx yyyyyyy yyyyy yyyyyyyyy yyyyy yyy yy yyy
 *
 */
```

The modification history should record any significant changes to the program logic.

6.2 Procedure headers

All function are preceded by a comment block which will be of the format given below

```
/******************* 80 characters wide **************************
* Function     : <Function Name>
*
* Description  : <Overview of the function>
*
```

```
*
*
* Input Parameters :
*
*
*
*
*
* Returns      :
*
*
*
* Globals      :
*
*
*
* Static funcs : aaaaa()
*
* Extern funcs : bbbbb()
*
*
*

*******************************************************************/
```

6.3 In-line and block comments

In-line comments are discouraged. Provide in-line comments only if they are a must

Other comments should begin with the same indentation as the succeeding source code and end on the 80th column

Blank lines occur before and after the comment blocks.

Avoid commenting individual statements. Instead comment a group of statements explaining the logic

Avoid trivial comments like `/* increment counter */`

C Programming Assignment

1. Write a program to find whether the number entered by the user is prime number or not. Extend this program to list all the prime numbers between two given numbers.

2. Do the following for the user-entered number of students. Find the average marks for a student of his marks in 3 subjects. Print whether he passed or failed. A student will fail if his average is less than 50. Use for loop

3. Do the following for an unknown number of students. (User will explicitly indicate when to terminate). Find the average marks for a student of his marks in 3 subjects. Print whether he passed or failed. A student will fail if his average is less than 50. Use While loop.

4. Write a program, that accepts a integer from the user and print the integer with reverse digits. For eg: rev(1234) = 4321.

5. Find the sum of the digits of a given number.

6. Given three numbers, determine whether they can form the sides of triangle.

7. Write a program which allow to perform any of the following operations on two 3*3 arrays

a) Add Arrays.
b) Multiply Arrays.
c) Subtract Arrays.

Assessment Question – 1

1. Write a program that takes in three arguments, a start temperature (in Celsius), an end temperature (in Celsius) and a step size. Print out a table that goes from the start temperature to the end temperature, in steps of the step size; you do not actually need to print the final end temperature if the step size does not exactly match. You should perform input validation: do not accept start temperatures less than a lower limit (which your code should specify as a constant) or higher than an upper limit (which your code should also specify). You should not allow a step size greater than the difference in temperatures. (This exercise was based on a problem from C Programming Language).

   Sample run:

```
Please give in a lower limit, limit >= 0: 10
Please give in a higher limit, 10 > limit <= 50000: 20
Please give in a step, 0 < step <= 10: 4

Celsius         Fahrenheit
-------         ----------
10.000000       50.000000
14.000000       57.200000
```

```
18.0      64.400000
```

2. Here's a simple help free challenge to get you started: write a program that takes a file as an argument and counts the total number of lines. Lines are defined as ending with a newline character.

   Program usage should be “count filename.txt”
   And
   The output should be the line count.

3. In this challenge, given the name of a file, print out the size of the file, in bytes. If no file is given, provide a help string to the user that indicates how to use the program. You might need help with taking parameters via the command line or file I/O in C++ (if you want to solve this problem in C, you might be interested in this article on C file I/O).

4. Here is another mathematical problem, where the trick is as much to discover the algorithm as it is to write the code: write a program to display all possible permutations of a given input string--if the string contains duplicate characters, you may have multiple repeated results. Input should be of the form

Assessment Question – 1

```
permute string
```

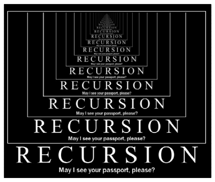
and output should be a word per line.

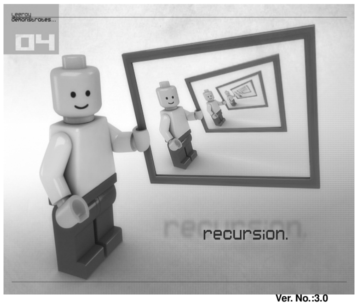
Here is a sample for the input *cat*

```
cat
cta
act
atc
tac
tca
```

C
Language

C Programming – Level 2 and 3





ER/CORP/CRS/LA87 Ver. No.:3.0

Session Plan

- **Recursive Functions**
- **Testing**
- **Debugging**
- **Code Review**
- **Some Exercises on control structures**
- **PF Project Discussion**

Recursive Functions (1 of 7)

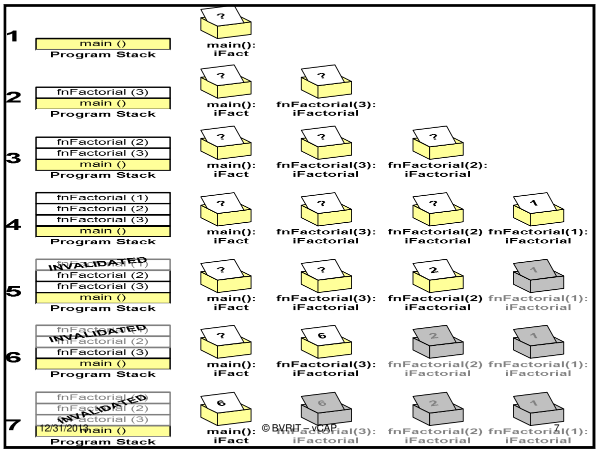
- When a function calls itself it is called as **Recursion**
- Many mathematical, searching and sorting algorithms, can be simply expressed in terms of a recursive definition
- A recursive definition has two parts:
  - **Base condition** : When a function will terminate
  - **Recursive condition** :The invocation of a recursive call to the function
- When the problem is solved through recursion the source code looks elegant

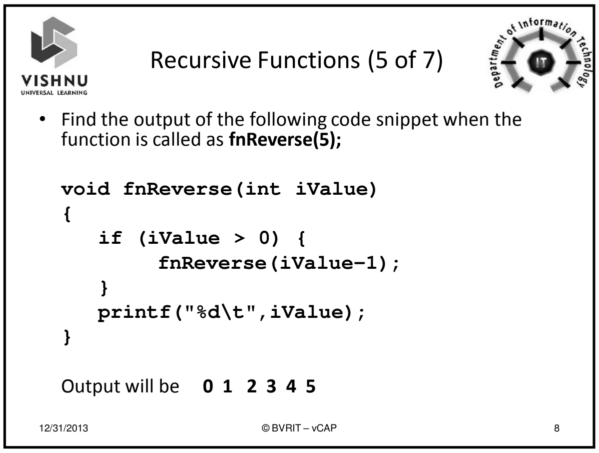
Recursive Functions (2 of 7)

```
/* Finding the factorial of an integer using a
   recursive function */

int fnFact(int iNumber); /* Function Prototype */

int main(int argc, char **argv) {
      int iFactorial;
      iFactorial=fnFact(4);
      printf("The factorial is %d\n",iFactorial);
      return 0;
}
```

Recursive Functions (3 of 7)

```
int fnFact(int iNumber)
 {
      int iFact;
      if (iNumber <= 1) {
              return 1;
      }
      else {
              iFact = iNumber * fnFact(iNumber - 1);
      }
      return iFact;
 }
```





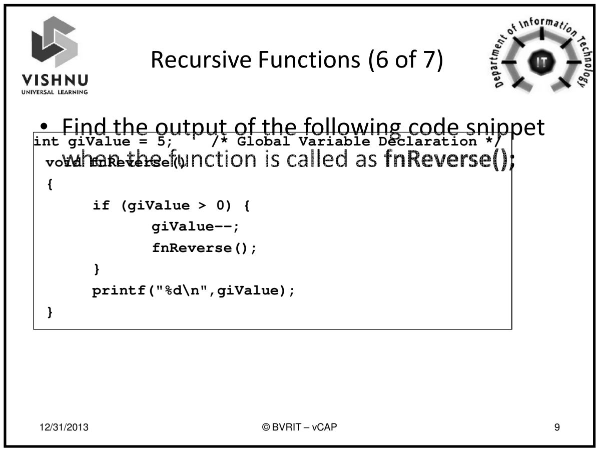
VISHNU UNIVERSAL LEARNING

Recursive Functions (5 of 7)

- Find the output of the following code snippet when the function is called as **fnReverse(5);**

```
void fnReverse(int iValue)
{
   if (iValue > 0) {
        fnReverse(iValue-1);
   }
   printf("%d\t",iValue);
}
```

Output will be **0 1 2 3 4 5**

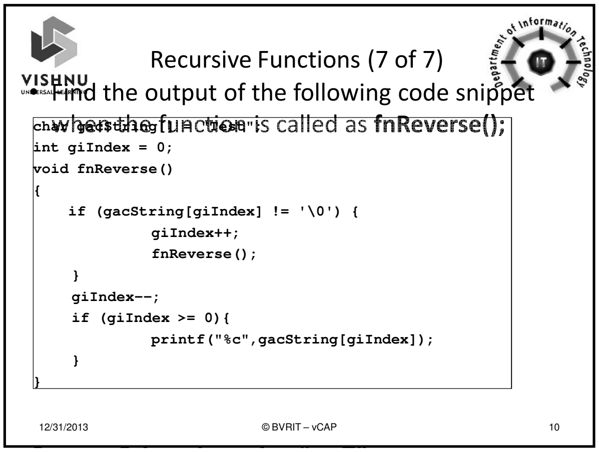


VISHNU UNIVERSAL LEARNING

Recursive Functions (6 of 7)

- Find the output of the following code snippet when the function is called as **fnReverse();**

```
int giValue = 5;    /* Global Variable Declaration */
 void fnReverse()
 {
      if (giValue > 0) {
             giValue--;
             fnReverse();
      }
      printf("%d\n",giValue);
 }
```

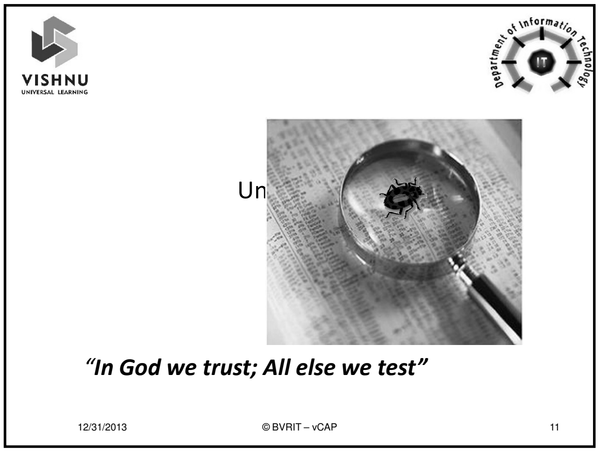


VISHNU UNIVERSAL LEARNING

Recursive Functions (7 of 7)

- Find the output of the following code snippet when the function is called as **fnReverse();**

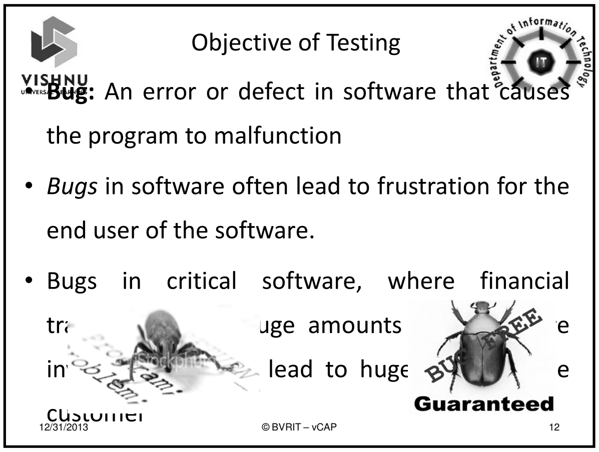
```
char gacString[] = "Test";
int giIndex = 0;
void fnReverse()
{
    if (gacString[giIndex] != '\0') {
               giIndex++;
               fnReverse();
     }
     giIndex--;
     if (giIndex >= 0){
               printf("%c",gacString[giIndex]);
     }
}
```



VISHNU UNIVERSAL LEARNING

Un

"In God we trust; All else we test"

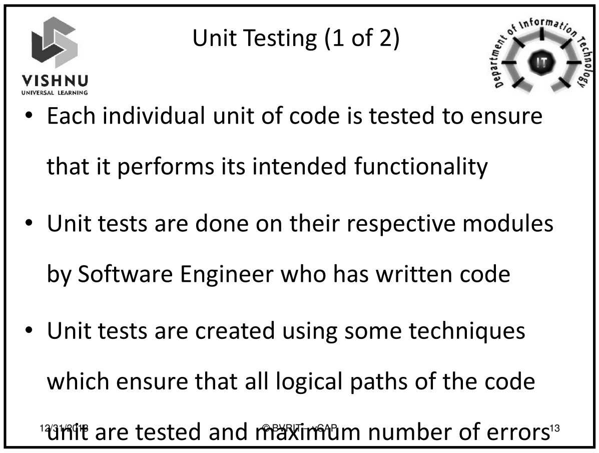


VISHNU UNIVERSAL LEARNING

Objective of Testing

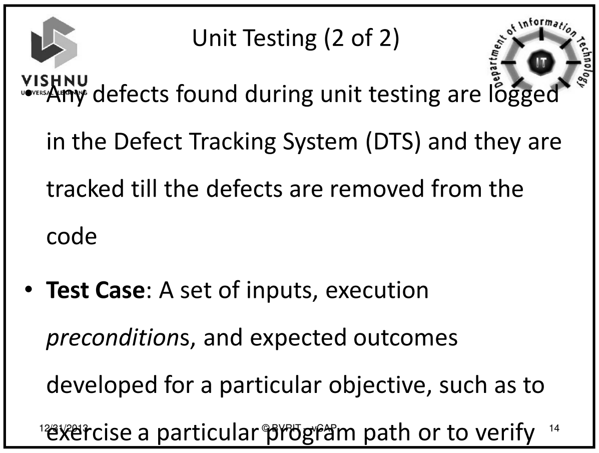
- **Bug:** An error or defect in software that causes the program to malfunction
- *Bugs* in software often lead to frustration for the end user of the software.
- Bugs in critical software, where financial tra uge amounts e in lead to huge e customer

BUG FREE
Guaranteed



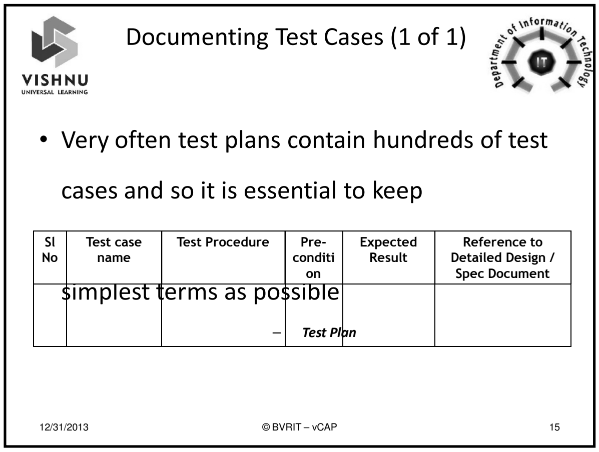
Unit Testing (1 of 2)

- Each individual unit of code is tested to ensure that it performs its intended functionality
- Unit tests are done on their respective modules by Software Engineer who has written code
- Unit tests are created using some techniques which ensure that all logical paths of the code unit are tested and maximum number of errors



Unit Testing (2 of 2)

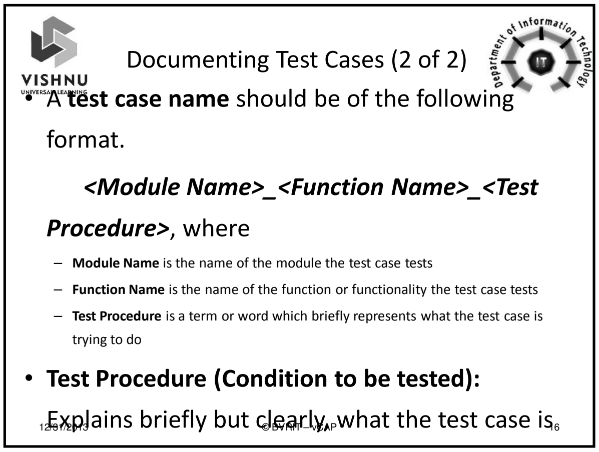
- Any defects found during unit testing are logged in the Defect Tracking System (DTS) and they are tracked till the defects are removed from the code
- **Test Case**: A set of inputs, execution *precondition*s, and expected outcomes developed for a particular objective, such as to exercise a particular program path or to verify



Documenting Test Cases (1 of 1)

- Very often test plans contain hundreds of test cases and so it is essential to keep simplest terms as possible
  - *Test Plan*

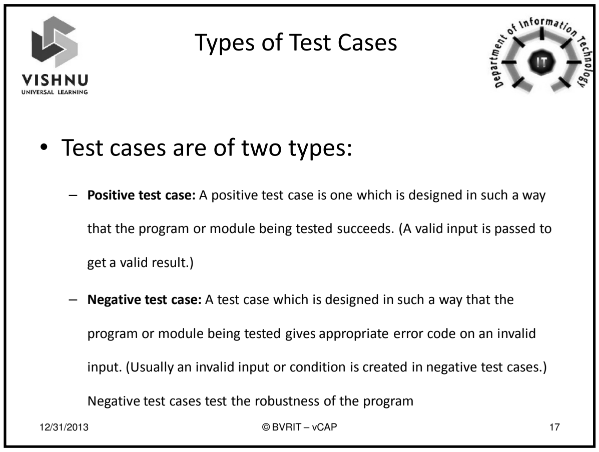
Sl No	Test case name	Test Procedure	Pre-condition	Expected Result	Reference to Detailed Design / Spec Document



Documenting Test Cases (2 of 2)

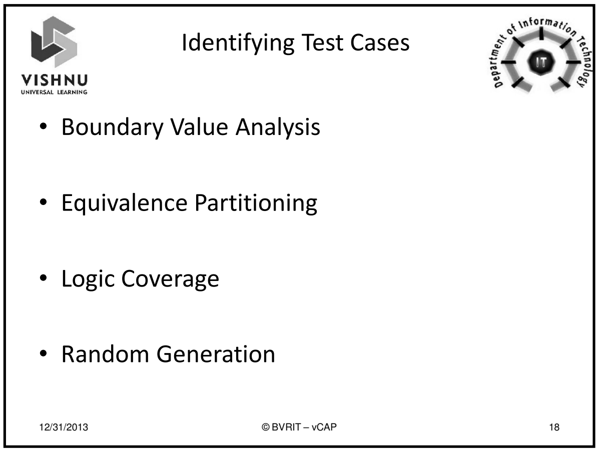
- A **test case name** should be of the following format.

  <Module Name>_<Function Name>_<Test Procedure>, where
  - **Module Name** is the name of the module the test case tests
  - **Function Name** is the name of the function or functionality the test case tests
  - **Test Procedure** is a term or word which briefly represents what the test case is trying to do
- **Test Procedure (Condition to be tested):** Explains briefly but clearly, what the test case is



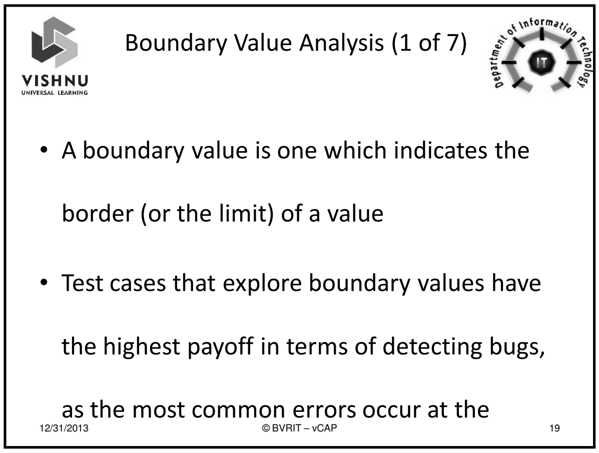
Types of Test Cases

- Test cases are of two types:
  - **Positive test case:** A positive test case is one which is designed in such a way that the program or module being tested succeeds. (A valid input is passed to get a valid result.)
  - **Negative test case:** A test case which is designed in such a way that the program or module being tested gives appropriate error code on an invalid input. (Usually an invalid input or condition is created in negative test cases.) Negative test cases test the robustness of the program



Identifying Test Cases

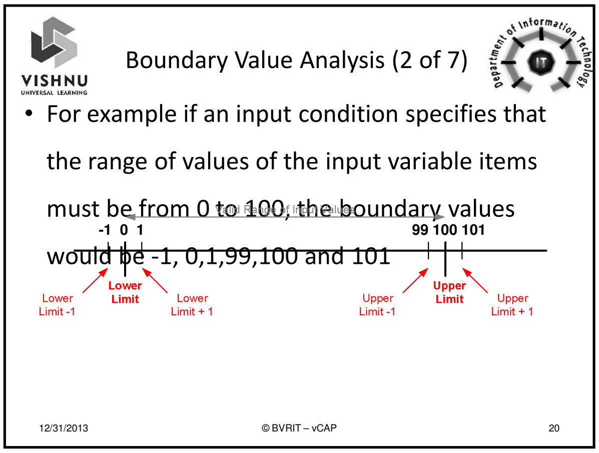
- Boundary Value Analysis
- Equivalence Partitioning
- Logic Coverage
- Random Generation



Boundary Value Analysis (1 of 7)

- A boundary value is one which indicates the border (or the limit) of a value
- Test cases that explore boundary values have the highest payoff in terms of detecting bugs, as the most common errors occur at the

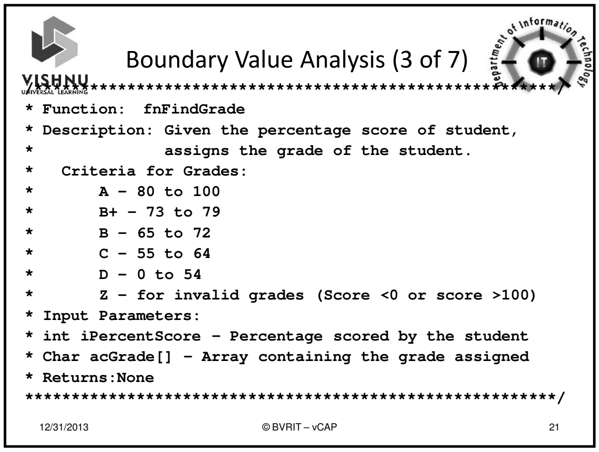
12/31/2013 © BVRIT – vCAP 19



Boundary Value Analysis (2 of 7)

- For example if an input condition specifies that the range of values of the input variable items must be from 0 to 100, the boundary values would be -1, 0,1,99,100 and 101

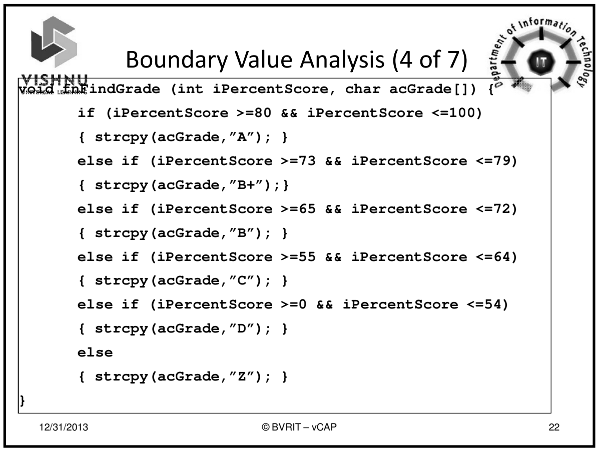
12/31/2013 © BVRIT – vCAP 20



Boundary Value Analysis (3 of 7)

```
/*************************************************************/
* Function:  fnFindGrade
* Description: Given the percentage score of student,
*              assigns the grade of the student.
*   Criteria for Grades:
*       A - 80 to 100
*       B+ - 73 to 79
*       B - 65 to 72
*       C - 55 to 64
*       D - 0 to 54
*       Z - for invalid grades (Score <0 or score >100)
* Input Parameters:
* int iPercentScore - Percentage scored by the student
* Char acGrade[] - Array containing the grade assigned
* Returns:None
*************************************************************/
```

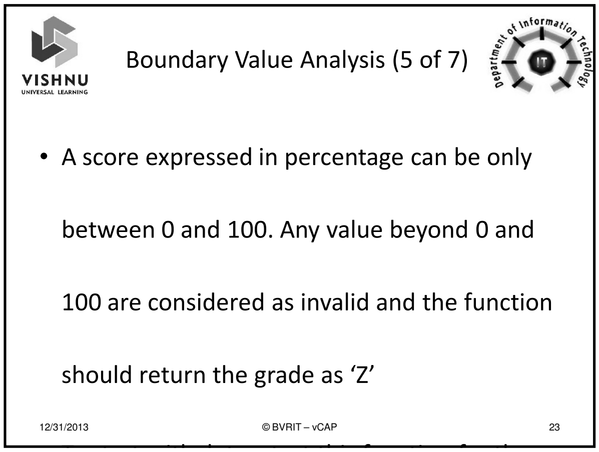
12/31/2013 © BVRIT – vCAP 21



Boundary Value Analysis (4 of 7)

```
void fnFindGrade (int iPercentScore, char acGrade[]) {
      if (iPercentScore >=80 && iPercentScore <=100)
      { strcpy(acGrade,"A"); }
      else if (iPercentScore >=73 && iPercentScore <=79)
      { strcpy(acGrade,"B+");}
      else if (iPercentScore >=65 && iPercentScore <=72)
      { strcpy(acGrade,"B"); }
      else if (iPercentScore >=55 && iPercentScore <=64)
      { strcpy(acGrade,"C"); }
      else if (iPercentScore >=0 && iPercentScore <=54)
      { strcpy(acGrade,"D"); }
      else
      { strcpy(acGrade,"Z"); }
}
```

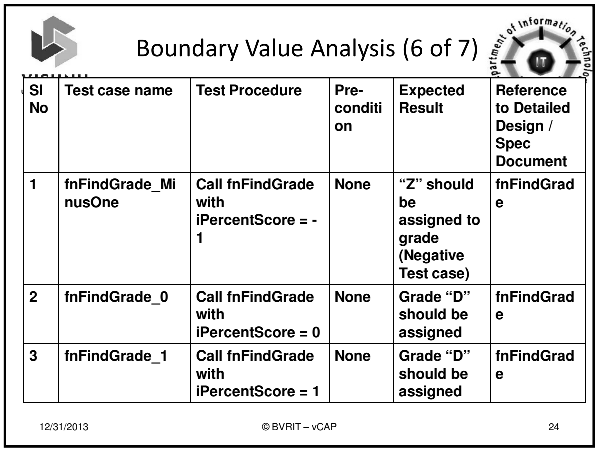
12/31/2013 © BVRIT – vCAP 22



Boundary Value Analysis (5 of 7)

- A score expressed in percentage can be only between 0 and 100. Any value beyond 0 and 100 are considered as invalid and the function should return the grade as 'Z'

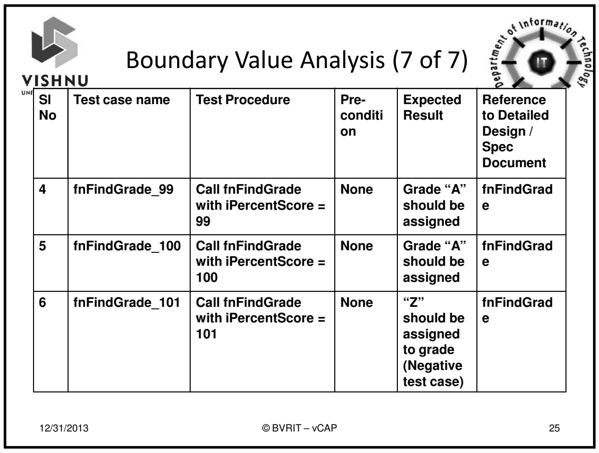
12/31/2013 © BVRIT – vCAP 23



Boundary Value Analysis (6 of 7)

Sl No	Test case name	Test Procedure	Pre-condition	Expected Result	Reference to Detailed Design / Spec Document
1	fnFindGrade_MinusOne	Call fnFindGrade with iPercentScore = -1	None	"Z" should be assigned to grade (Negative Test case)	fnFindGrade
2	fnFindGrade_0	Call fnFindGrade with iPercentScore = 0	None	Grade "D" should be assigned	fnFindGrade
3	fnFindGrade_1	Call fnFindGrade with iPercentScore = 1	None	Grade "D" should be assigned	fnFindGrade

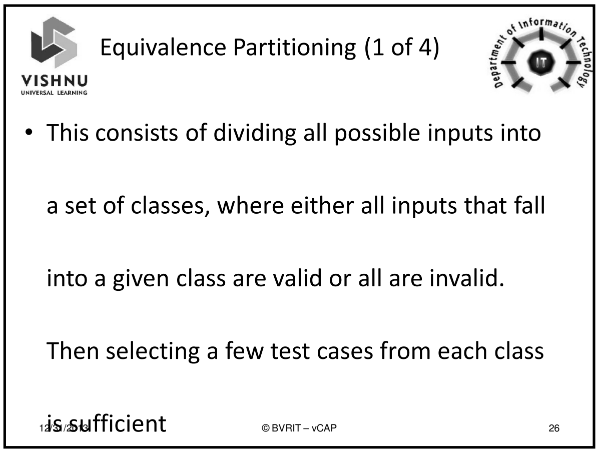
12/31/2013 © BVRIT – vCAP 24



Boundary Value Analysis (7 of 7)

Sl No	Test case name	Test Procedure	Pre-condition	Expected Result	Reference to Detailed Design / Spec Document
4	fnFindGrade_99	Call fnFindGrade with iPercentScore = 99	None	Grade "A" should be assigned	fnFindGrade
5	fnFindGrade_100	Call fnFindGrade with iPercentScore = 100	None	Grade "A" should be assigned	fnFindGrade
6	fnFindGrade_101	Call fnFindGrade with iPercentScore = 101	None	"Z" should be assigned to grade (Negative test case)	fnFindGrade

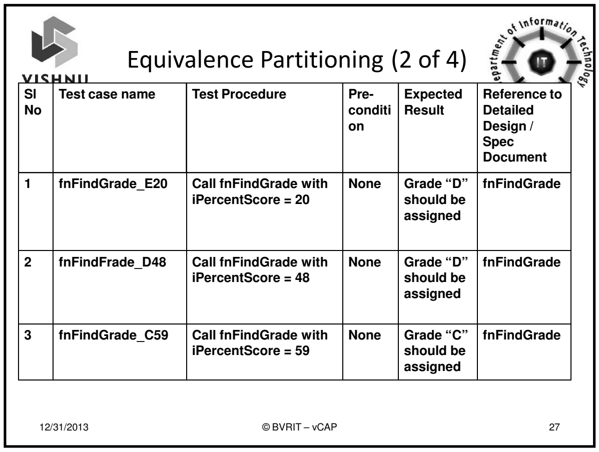
12/31/2013 © BVRIT – vCAP 25



Equivalence Partitioning (1 of 4)

- This consists of dividing all possible inputs into a set of classes, where either all inputs that fall into a given class are valid or all are invalid. Then selecting a few test cases from each class is sufficient

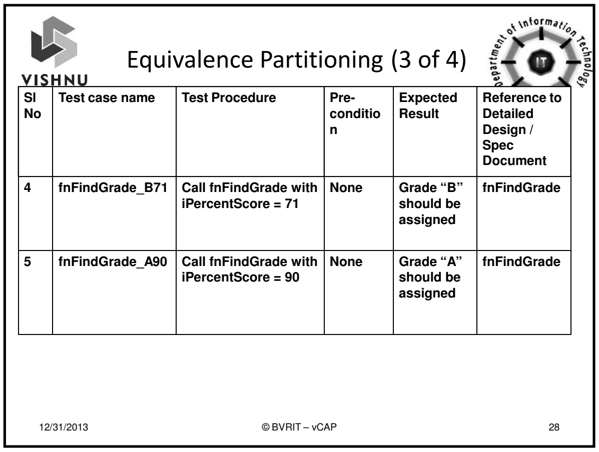
12/31/2013 © BVRIT – vCAP 26



Equivalence Partitioning (2 of 4)

Sl No	Test case name	Test Procedure	Pre-condition	Expected Result	Reference to Detailed Design / Spec Document
1	fnFindGrade_E20	Call fnFindGrade with iPercentScore = 20	None	Grade "D" should be assigned	fnFindGrade
2	fnFindFrade_D48	Call fnFindGrade with iPercentScore = 48	None	Grade "D" should be assigned	fnFindGrade
3	fnFindGrade_C59	Call fnFindGrade with iPercentScore = 59	None	Grade "C" should be assigned	fnFindGrade

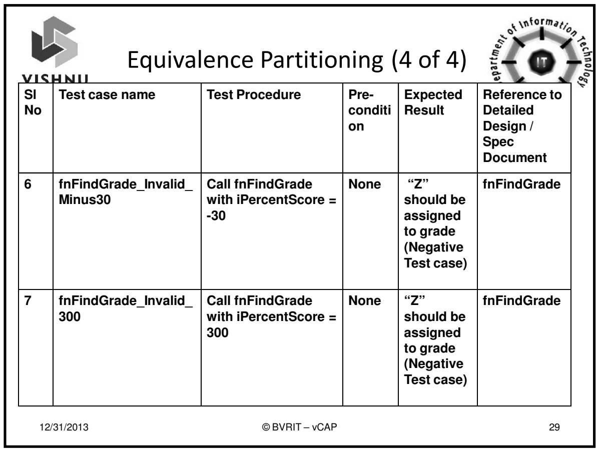
12/31/2013 © BVRIT – vCAP 27



Equivalence Partitioning (3 of 4)

Sl No	Test case name	Test Procedure	Pre-condition	Expected Result	Reference to Detailed Design / Spec Document
4	fnFindGrade_B71	Call fnFindGrade with iPercentScore = 71	None	Grade "B" should be assigned	fnFindGrade
5	fnFindGrade_A90	Call fnFindGrade with iPercentScore = 90	None	Grade "A" should be assigned	fnFindGrade

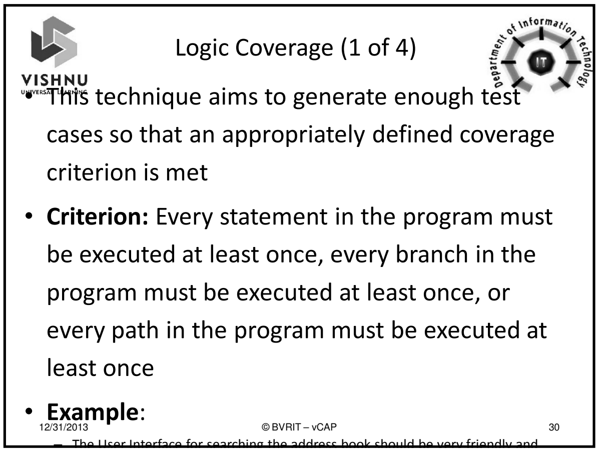
12/31/2013 © BVRIT – vCAP 28



Equivalence Partitioning (4 of 4)

Sl No	Test case name	Test Procedure	Pre-condition	Expected Result	Reference to Detailed Design / Spec Document
6	fnFindGrade_Invalid_Minus30	Call fnFindGrade with iPercentScore = -30	None	"Z" should be assigned to grade (Negative Test case)	fnFindGrade
7	fnFindGrade_Invalid_300	Call fnFindGrade with iPercentScore = 300	None	"Z" should be assigned to grade (Negative Test case)	fnFindGrade

12/31/2013 © BVRIT – vCAP 29

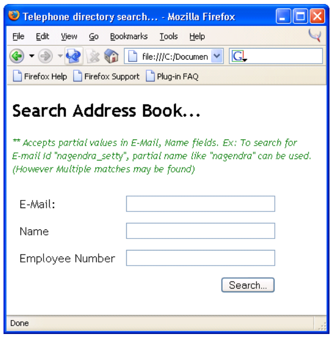


Logic Coverage (1 of 4)

- This technique aims to generate enough test cases so that an appropriately defined coverage criterion is met
- **Criterion:** Every statement in the program must be executed at least once, every branch in the program must be executed at least once, or every path in the program must be executed at least once
- **Example**:
  - The User Interface for searching the address book should be very friendly and

12/31/2013 © BVRIT – vCAP 30

Logic Coverage (2 of 4)


Telephone directory search... - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
file:///C:/Documen
Firefox Help
Firefox Support
Plug-in FAQ
Search...
Done

Search Address Book...

** Accepts partial values in E-Mail, Name fields. Ex: To search for E-mail Id "nagendra_setty", partial name like "nagendra" can be used. (However Multiple matches may be found)

E-Mail:

Name

Employee Number

Sl No	Test case name	Test Procedure	Pre-condition	Expected Result	Reference to Detailed Design
1	addrbook_all_blank	All the fields are kept blank and click on 'Search…'	None	Address book must display an Error message and prompt user to enter at least one field. (Negative Test case)	Address book Module
2	addrbook_empno_ok	Type in an employee numbe r (Ex: 7342) and then click on 'Search…'	None	Address book must fetch one (only one) entry of the person with that employee number	Address book Module
3	addrbook_empno_fail	Type in an invalid employee number and then click 'Search…'	None	Address book must fetch zero records and display that record is not found. (Negative Test case)	
4	addrbook_email_full	Type in a full e - mail id (Ex: nagendra_setty) and then click on 'Search…'	None	Address book must fetch one (only one) entry of the person corresponding to the e -mail Id.	Address book Module

5	addrbook_email_partial	Type in a partial but valid e-mail id (Ex:nagen) and then click on 'Search…'	None	Address book should fetch one or more record s where e -mail id begin s with the same letters .	Address book Module
6	addrbook_email_fail	Type in an invalid name (Say jhsgjss) and click on 'Search…'	None	Address book must fetch zero records and display that record is not found. (Negative Test case)	Address book Module
7	addrbook_name_full	Type in a full name (Ex: Nagendra R Setty) and then click on 'Search…'	None	Address book must fetch one (only one) entry of the person with that name.	Address book Module
8	addrbook_name_partial	Type in a Partial name (Ex: Nagend) and then click on 'Search…'	None	Address book shou ld fetch one or more records where name begins with the same letters.	Address book Module

Random Generation

- Data is generated randomly either using a tool or manually. This is the simplest method but not the most efficient

Implementing Test Cases (1 of 2)

- Unit Tests can be executed either manually or can be automated
- Usually, testing of User Interfaces (screens) is done manually
- Testing a function or piece of code can be

Implementing Test Cases (2 of 2)

Sl No	Test case name	Test Procedure	Pre-condition	Expected Result	Reference to Detailed Design
6	fnFindGrade_101	Call fnFindGrade with iPercentScore = 101	None	'Z' should be assigned to grade (Negative test case)	fnFindGrade

- Within the test function, when the test case does not result in the expected output, it is always a good practice to print all relevant

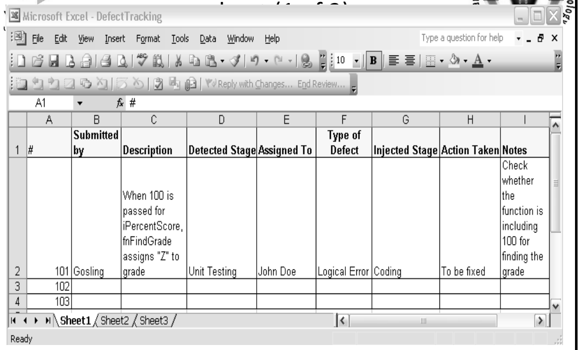
Recording / Logging a Defect

- Any defect found in code or document must be recorded
- Recording of defects will ensure that the

Defect Tracking System

- Most software companies have a dedicated system, for logging and tracking defects
- Most defect tracking systems can also do detailed analysis of the defects to help a project take corrective action in due course of

A sample defect tracking Excel sheet (1 of 2)


Microsoft Excel - DefectTracking
File Edit View Insert Format Tools Data Window Help
Type a question for help
10
B
Reply with Changes... End Review...
A1
fx #
Sheet1 Sheet2 Sheet3
Ready

	A	B	C	D	E	F	G	H	I
1	#	Submitted by	Description	Detected Stage	Assigned To	Type of Defect	Injected Stage	Action Taken	Notes
2	101	Gosling	When 100 is passed for iPercentScore, fnFindGrade assigns "Z" to grade	Unit Testing	John Doe	Logical Error	Coding	To be fixed	Check whether the function is including 100 for finding the grade
3	102								
4	103								

A sample defect tracking Excel sheet (2 of 2)

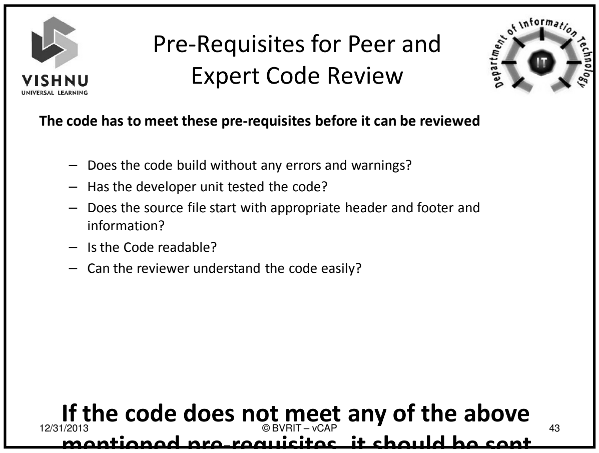
- A defect log captures some of the following information:
  - **Defect Number or Id:** The number or Id which identifies the defect
  - **Submitted by:** Person who found the defect
  - **Description:** A detailed description of the defect
  - **Detected Stage:** The stage at which defect was detected
    - **Example**: Unit Testing, Code Review etc.
  - **Assigned to:** The developer who has to remove this defect
  - **Type of defect:** Type of defect tells about the nature of the defect
    - **Example**: Coding Standards related, Logical Error
  - **Injected Stage:** The stage of software life cycle where this defect might have been introduced
    - High Level Design, Detailed Design, Coding etc
  - **Action Taken:** To be fixed or fixed

Code Review

- A process where several people offer constructive criticism of a Software Engineer's code with a view to simplify it, to make it more efficient and to eliminate errors
- Locates or identifies potential bugs and failure

Code Review

- Types:
  - **Self Code Review:** The person who wrote code reviews his/her own code using the code review checklist. Defects are fixed as they are found
  - **Peer Code Review:** The team member reviews the code written by another team member using the code review checklist
  - **Expert Code Review:** Another person, who is an expert, reviews the code using the code review checklist. Defects are logged into a Defect Tracking System, and tracked to closure. The person who wrote the code has to remove the defects from the code (not the Reviewer)

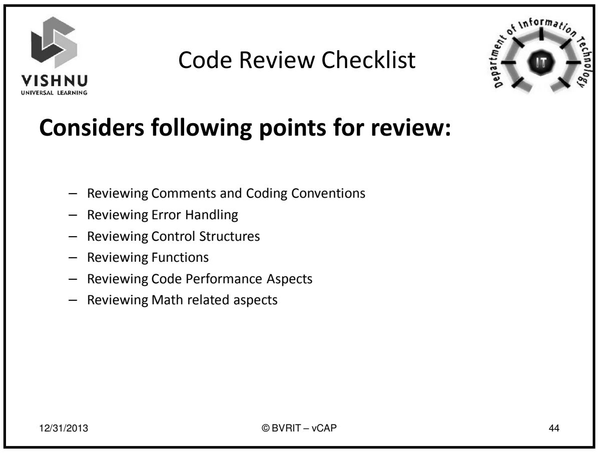


Pre-Requisites for Peer and Expert Code Review

The code has to meet these pre-requisites before it can be reviewed

- Does the code build without any errors and warnings?
- Has the developer unit tested the code?
- Does the source file start with appropriate header and footer and information?
- Is the Code readable?
- Can the reviewer understand the code easily?

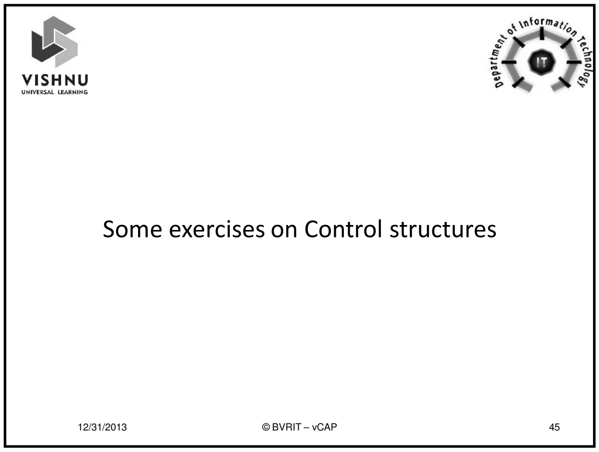
If the code does not meet any of the above mentioned pre-requisites, it should be sent



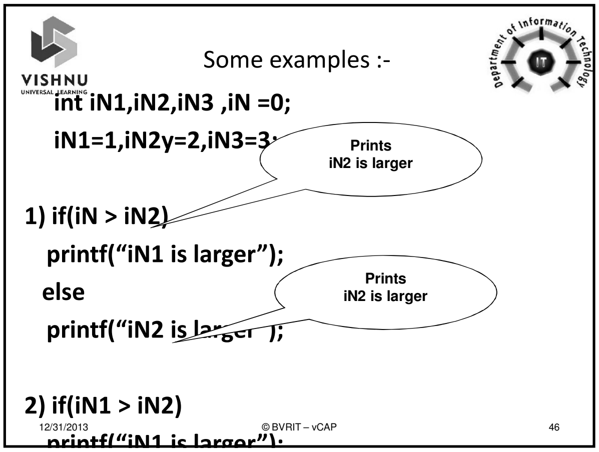
Code Review Checklist

Considers following points for review:

- Reviewing Comments and Coding Conventions
- Reviewing Error Handling
- Reviewing Control Structures
- Reviewing Functions
- Reviewing Code Performance Aspects
- Reviewing Math related aspects



Some exercises on Control structures

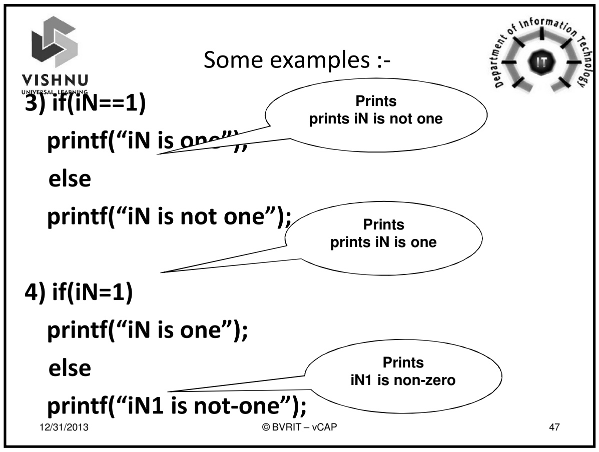


Some examples :-

```
 int iN1,iN2,iN3 ,iN =0;
 iN1=1,iN2y=2,iN3=3;

1) if(iN > iN2)
   printf(“iN1 is larger”);
  else
   printf(“iN2 is larger”);

2) if(iN1 > iN2)
   printf(“iN1 is larger”);
```

Prints
iN2 is larger

Prints
iN2 is larger



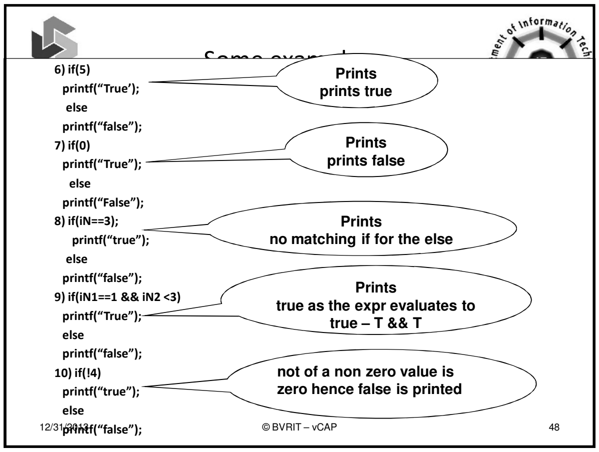
Some examples :-

```
3) if(iN==1)
   printf(“iN is one”);
   else
   printf(“iN is not one”);

4) if(iN=1)
   printf(“iN is one”);
   else
   printf(“iN1 is not-one”);
```

Prints
prints iN is not one

Prints
prints iN is one

Prints
iN1 is non-zero



```
6) if(5)
  printf(“True’);
   else
  printf(“false”);
7) if(0)
  printf(“True”);
    else
  printf(“False”);
8) if(iN==3);
     printf(“true”);
   else
  printf(“false”);
9) if(iN1==1 && iN2 <3)
  printf(“True”);
  else
  printf(“false”);
10) if(!4)
  printf(“true”);
  else
  printf(“false”);
```

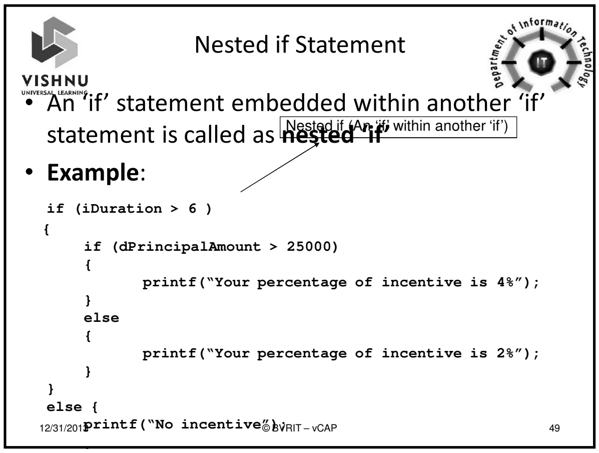
Prints
prints true

Prints
prints false

Prints
no matching if for the else

Prints
true as the expr evaluates to true – T && T

not of a non zero value is zero hence false is printed



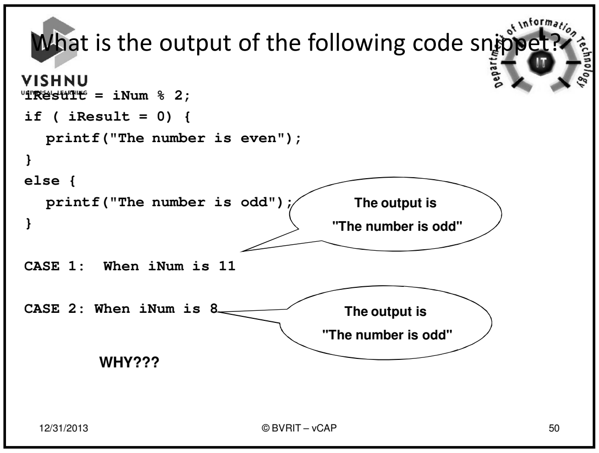
Nested if Statement

- An ‘if’ statement embedded within another ‘if’ statement is called as **nested ‘if’**
- **Example**:

Nested if (An ‘if’ within another ‘if’)

```
if (iDuration > 6 )
{
      if (dPrincipalAmount > 25000)
      {
               printf(“Your percentage of incentive is 4%”);
      }
      else
      {
               printf(“Your percentage of incentive is 2%”);
      }
}
else {
      printf(“No incentive”);
```

12/31/2013 © BVRIT – vCAP 49



What is the output of the following code snippet?

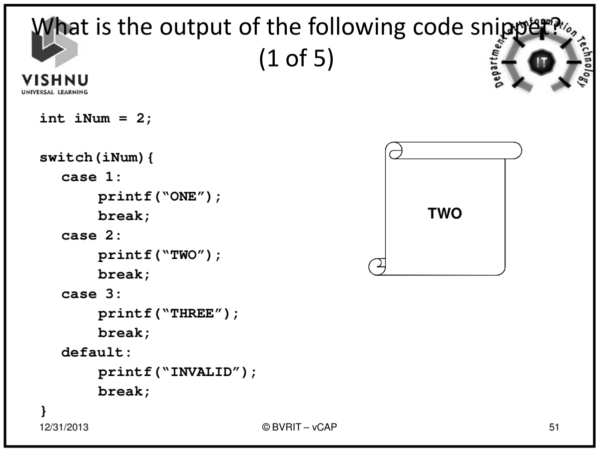
```
iResult = iNum % 2;
if ( iResult = 0) {
   printf("The number is even");
}
else {
   printf("The number is odd");
}

CASE 1:  When iNum is 11

CASE 2: When iNum is 8
```

The output is "The number is odd"

The output is "The number is odd"

WHY???

12/31/2013 © BVRIT – vCAP 50

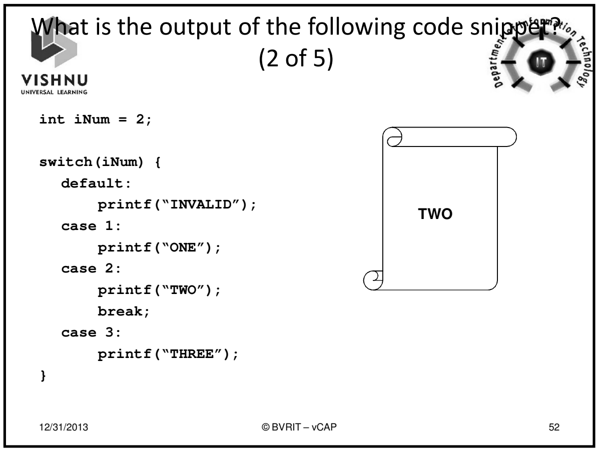


What is the output of the following code snippet? (1 of 5)

```
int iNum = 2;

switch(iNum){
   case 1:
        printf(“ONE”);
        break;
   case 2:
        printf(“TWO”);
        break;
   case 3:
        printf(“THREE”);
        break;
   default:
        printf(“INVALID”);
        break;
}
```

TWO

12/31/2013 © BVRIT – vCAP 51

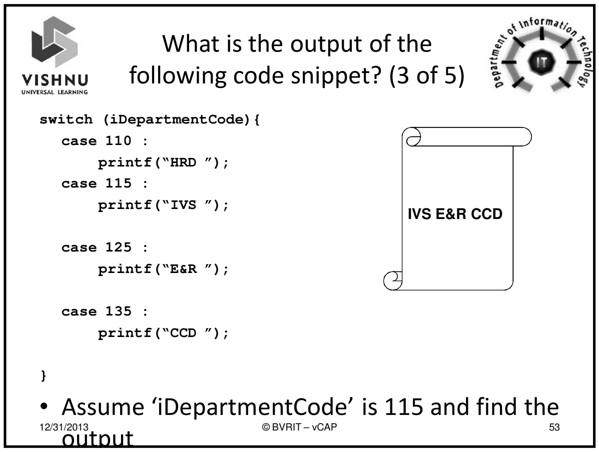


What is the output of the following code snippet? (2 of 5)

```
int iNum = 2;

switch(iNum) {
   default:
        printf(“INVALID”);
   case 1:
        printf(“ONE”);
   case 2:
        printf(“TWO”);
        break;
   case 3:
        printf(“THREE”);
}
```

TWO

12/31/2013 © BVRIT – vCAP 52



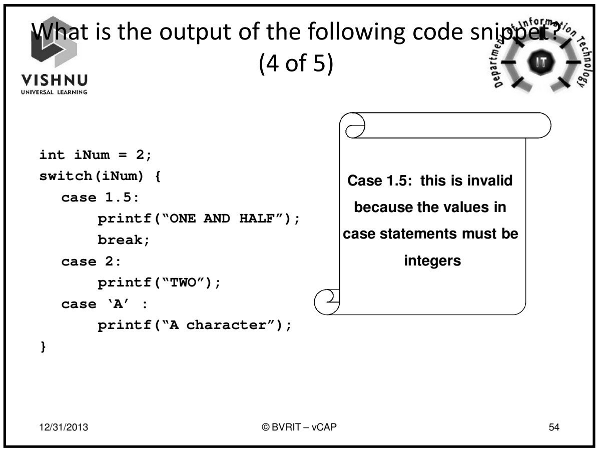
What is the output of the following code snippet? (3 of 5)

```
switch (iDepartmentCode){
   case 110 :
        printf(“HRD ”);
   case 115 :
        printf(“IVS ”);

   case 125 :
        printf(“E&R ”);

   case 135 :
        printf(“CCD ”);

}
```

IVS E&R CCD

- Assume ‘iDepartmentCode’ is 115 and find the output

12/31/2013 © BVRIT – vCAP 53

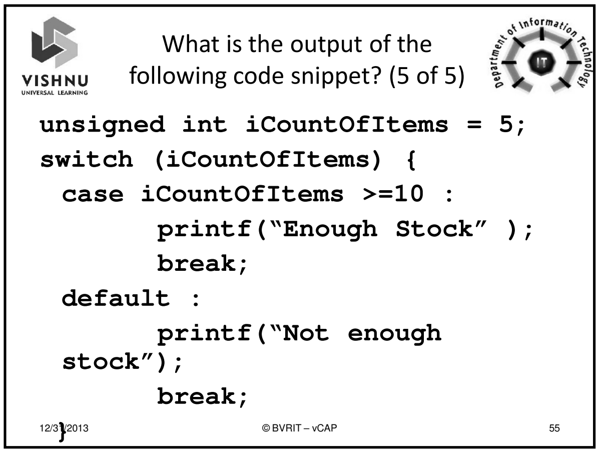


What is the output of the following code snippet? (4 of 5)

```
int iNum = 2;
switch(iNum) {
   case 1.5:
        printf(“ONE AND HALF”);
        break;
   case 2:
        printf(“TWO”);
   case ‘A’ :
        printf(“A character”);
}
```

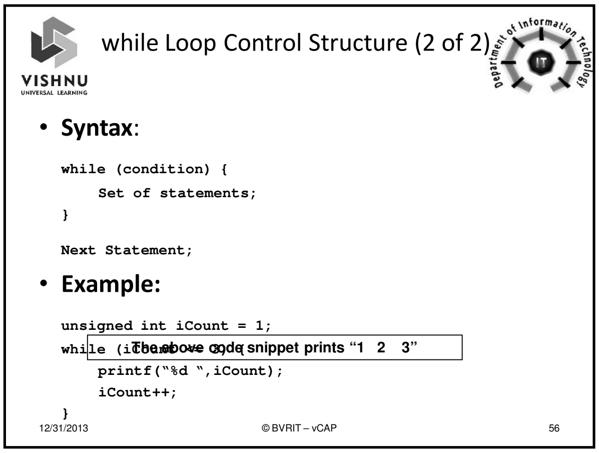
Case 1.5: this is invalid because the values in case statements must be integers

12/31/2013 © BVRIT – vCAP 54



What is the output of the following code snippet? (5 of 5)

```
unsigned int iCountOfItems = 5;
switch (iCountOfItems) {
  case iCountOfItems >=10 :
         printf("Enough Stock" );
         break;
  default :
         printf("Not enough
  stock");
         break;
}
```



while Loop Control Structure (2 of 2)

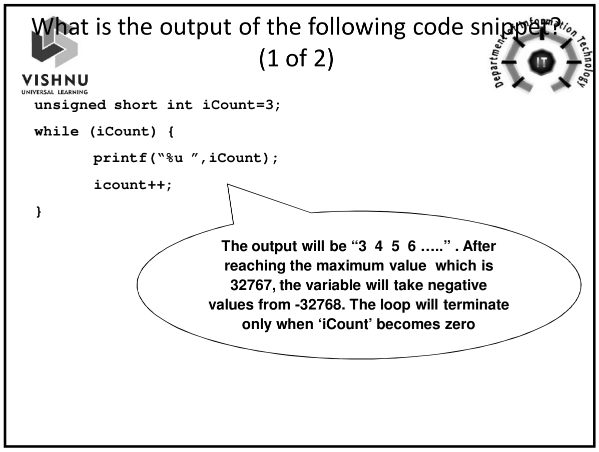
- **Syntax**:

```
while (condition) {
    Set of statements;
}

Next Statement;
```

- **Example:**

```
unsigned int iCount = 1;
while (iCount <= 3) {
    printf("%d ",iCount);
    iCount++;
}
```

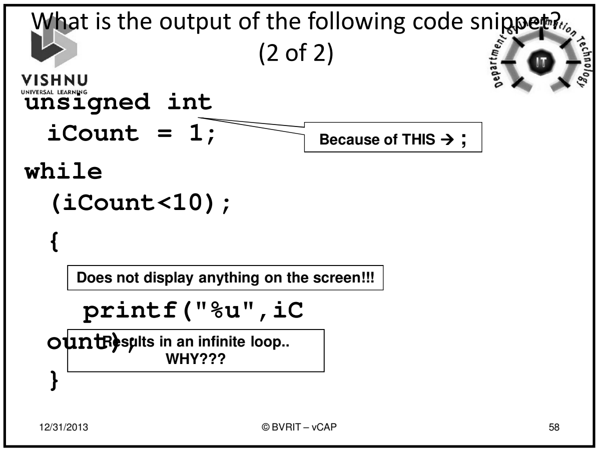
The above code snippet prints "1 2 3"



What is the output of the following code snippet? (1 of 2)

```
unsigned short int iCount=3;
while (iCount) {
       printf("%u ",iCount);
       icount++;
}
```

The output will be "3 4 5 6 ….." . After reaching the maximum value which is 32767, the variable will take negative values from -32768. The loop will terminate only when 'iCount' becomes zero



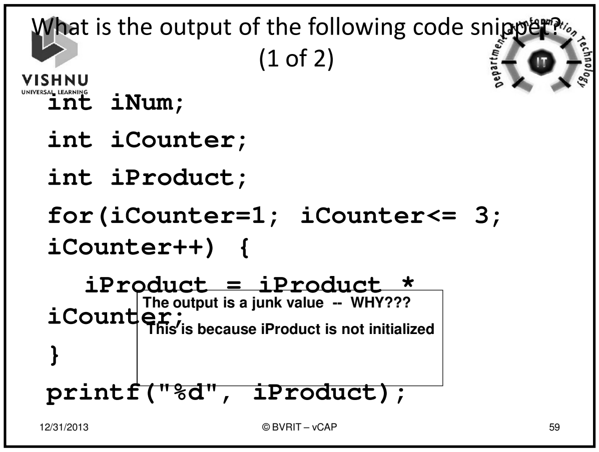
What is the output of the following code snippet? (2 of 2)

```
unsigned int
  iCount = 1;
while
  (iCount<10);
  {
    printf("%u",iC
  ount);
  }
```

Because of THIS → ;

Does not display anything on the screen!!!

Results in an infinite loop.. WHY???

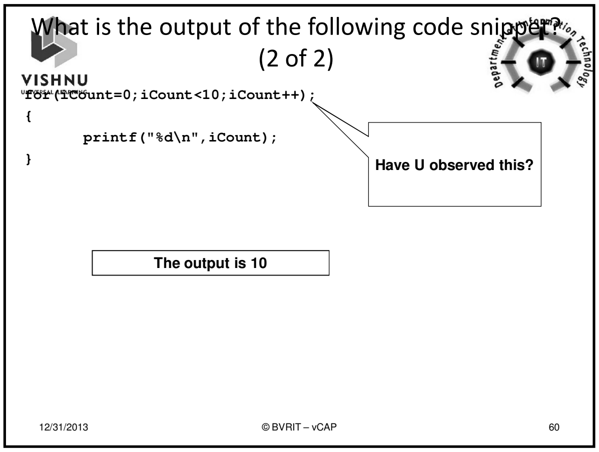


What is the output of the following code snippet? (1 of 2)

```
int iNum;
int iCounter;
int iProduct;
for(iCounter=1; iCounter<= 3;
iCounter++) {
   iProduct = iProduct *
iCounter;
}
printf("%d", iProduct);
```

The output is a junk value -- WHY???

This is because iProduct is not initialized

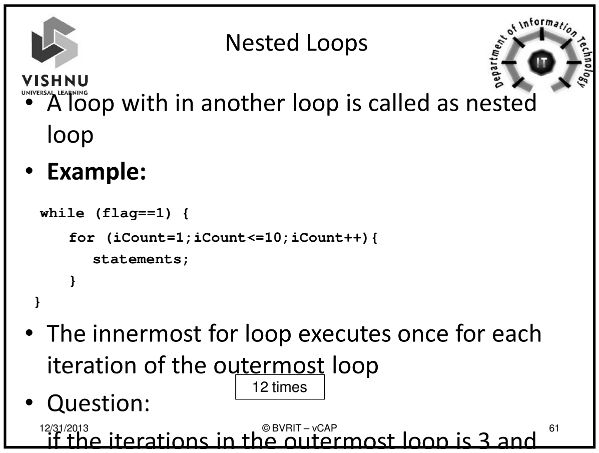


What is the output of the following code snippet? (2 of 2)

```
for(iCount=0;iCount<10;iCount++);
{
      printf("%d\n",iCount);
}
```

Have U observed this?

The output is 10



Nested Loops

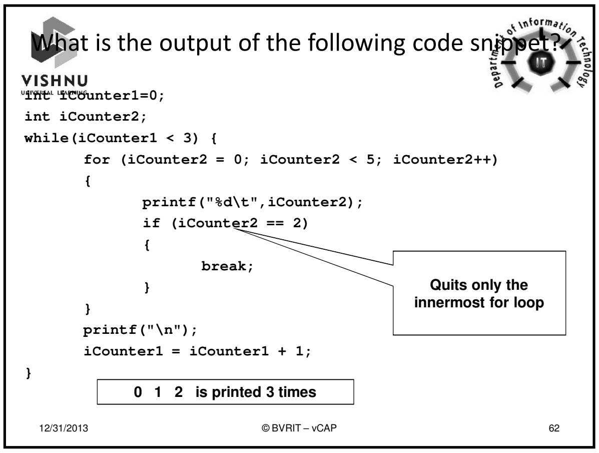
- A loop with in another loop is called as nested loop
- **Example:**

```
while (flag==1) {
   for (iCount=1;iCount<=10;iCount++){
     statements;
   }
}
```

- The innermost for loop executes once for each iteration of the outermost loop
- Question:

12 times

if the iterations in the outermost loop is 3 and

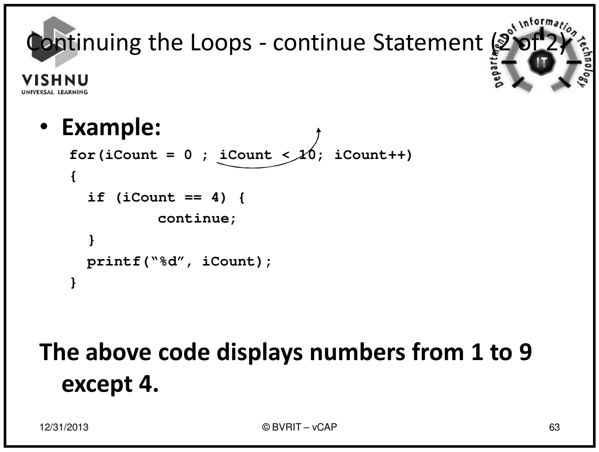


What is the output of the following code snippet?

```
int iCounter1=0;
int iCounter2;
while(iCounter1 < 3) {
      for (iCounter2 = 0; iCounter2 < 5; iCounter2++)
      {
            printf("%d\t",iCounter2);
            if (iCounter2 == 2)
            {
                  break;
            }
      }
      printf("\n");
      iCounter1 = iCounter1 + 1;
}
```

Quits only the innermost for loop

0 1 2 is printed 3 times

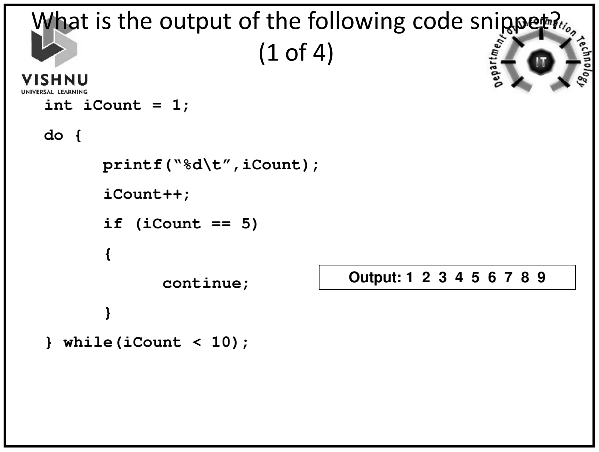


Continuing the Loops - continue Statement (2 of 2)

- **Example:**

```
for(iCount = 0 ; iCount < 10; iCount++)
{
  if (iCount == 4) {
        continue;
  }
  printf("%d", iCount);
}
```

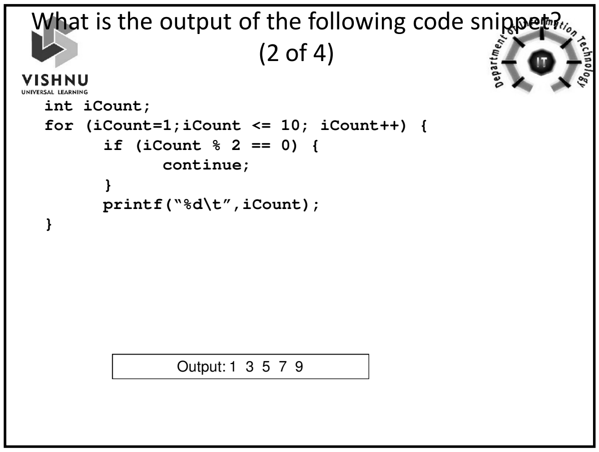
The above code displays numbers from 1 to 9 except 4.



What is the output of the following code snippet? (1 of 4)

```
int iCount = 1;
do {
      printf("%d\t",iCount);
      iCount++;
      if (iCount == 5)
      {
            continue;
      }
} while(iCount < 10);
```

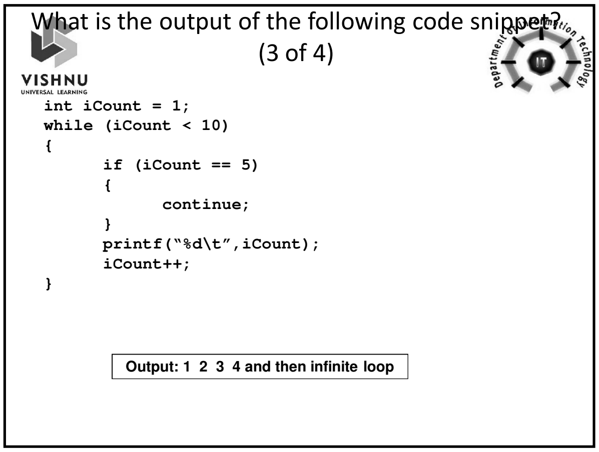
Output: 1 2 3 4 5 6 7 8 9



What is the output of the following code snippet? (2 of 4)

```
int iCount;
for (iCount=1;iCount <= 10; iCount++) {
      if (iCount % 2 == 0) {
            continue;
      }
      printf("%d\t",iCount);
}
```

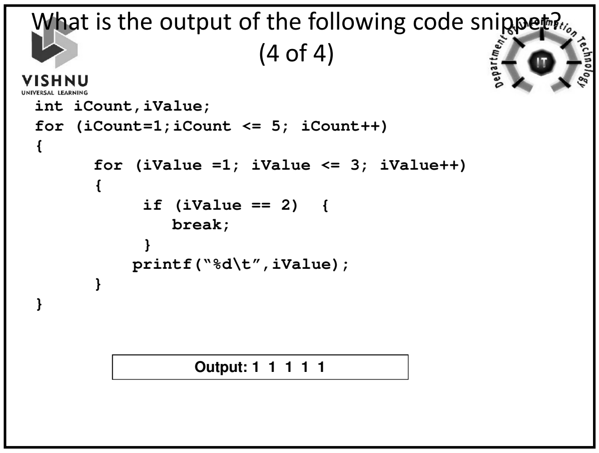
Output: 1 3 5 7 9



What is the output of the following code snippet? (3 of 4)

```
int iCount = 1;
while (iCount < 10)
{
      if (iCount == 5)
      {
            continue;
      }
      printf("%d\t",iCount);
      iCount++;
}
```

Output: 1 2 3 4 and then infinite loop



What is the output of the following code snippet? (4 of 4)

```
int iCount,iValue;
for (iCount=1;iCount <= 5; iCount++)
{
      for (iValue =1; iValue <= 3; iValue++)
      {
           if (iValue == 2)  {
              break;
           }
          printf(“%d\t”,iValue);
      }
}
```

Output: 1 1 1 1 1

C Programming Assignment

For the following assignments, write down the prototypes for the functions used before writing the functions

1. Write a program to find nearest smaller prime number for a given Integer; use a function to decide whether a number is prime or not.

2. Write a program that takes a positive integer as input and outputs the Fibonacci sequence up to that number.

3. Write a program which to print the multiplication table from 1 to m for n where m, n is the values entered by the user.

4. Write a program that will accept a string and character to search. The program will call a function, which will search for the occurrence position of the character in the string and return its position. Function should return –1 if the character is not found in the input string.

5. Write a function, which prints a given number in words.

6. Write an program which will set the array element a[i] to 1 if i is prime, and to 0 if i is not prime. Assume the array size to be 10000.

7. Write a program to count the number of vowels in a given string.

8. Write a program to obtain the transpose of a 4*4 array. The transpose is obtained by exchanging the elements of each row with the elements of the corresponding column.

Assessment Question – 2

1. Write a program that takes an integer and displays the English name of that value. You should support both positive and negative numbers, in the range supported by a 32-bit integer (approximately -2 billion to 2 billion).

   Examples:

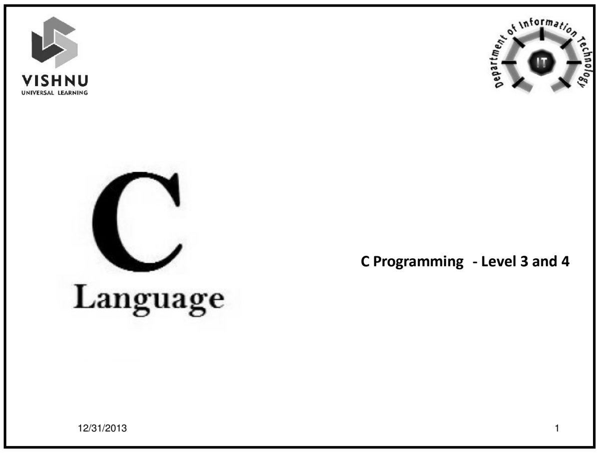
```
10 -> ten
121 -> one hundred twenty one
1032 -> one thousand thirty two
11043 -> eleven thousand forty three
1200000 -> one million two hundred thousand
```

2. Write a program that determines the number of trailing zeros at the end of X! (X factorial), where X is an arbitrary number. For instance, 5! is 120, so it has one trailing zero. (How can you handle extremely values, such as 100!?) The input format should be that the program asks the user to enter a number, minus the !.

3. Write a program that takes two arguments at the command line, both strings. The program checks to see whether or not the second string is a substring of the first (without using the substr -- or any other library -- function). One caveat: any * in the second string can match zero or more characters in the first string, so if the input were abcd and the substring were a*c, then it would count as a substring. Also, include functionality to allow an asterisk to be taken literally if preceded by a \, and a \ is taken literally except when preceding an asterisk.

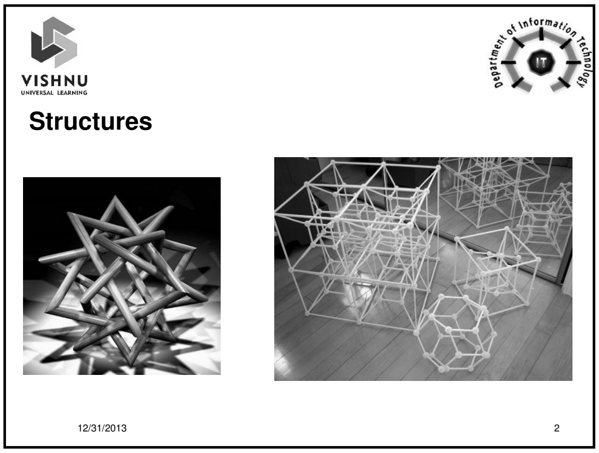
4. Write a program that accepts a base ten (non-fractional) number at the command line and outputs the binary representation of that number. Sample input is

```
dectobin 120
```

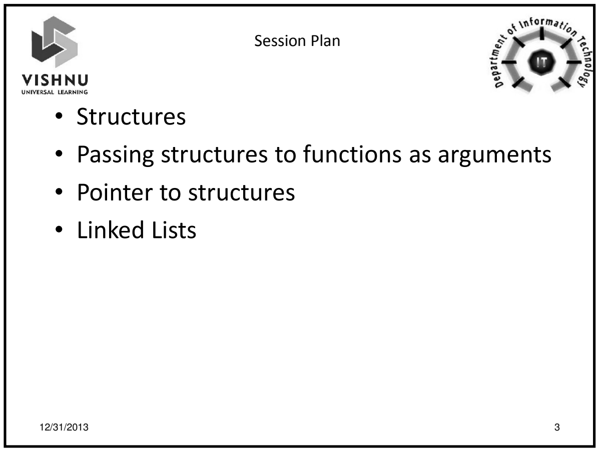


C Language

C Programming - Level 3 and 4

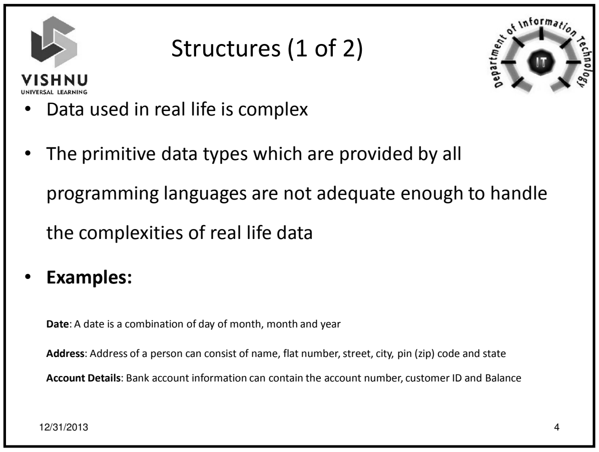


Structures



Session Plan

- Structures
- Passing structures to functions as arguments
- Pointer to structures
- Linked Lists



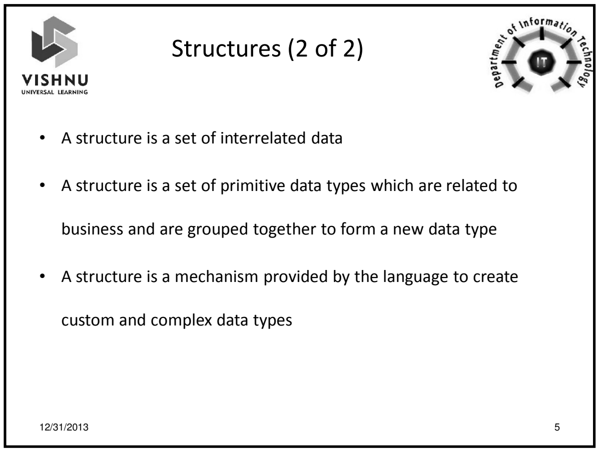
Structures (1 of 2)

- Data used in real life is complex
- The primitive data types which are provided by all programming languages are not adequate enough to handle the complexities of real life data
- **Examples:**

Date: A date is a combination of day of month, month and year

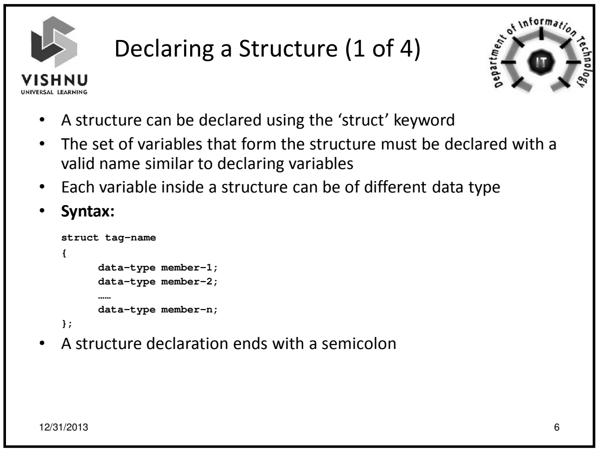
Address: Address of a person can consist of name, flat number, street, city, pin (zip) code and state

Account Details: Bank account information can contain the account number, customer ID and Balance



Structures (2 of 2)

- A structure is a set of interrelated data
- A structure is a set of primitive data types which are related to business and are grouped together to form a new data type
- A structure is a mechanism provided by the language to create custom and complex data types

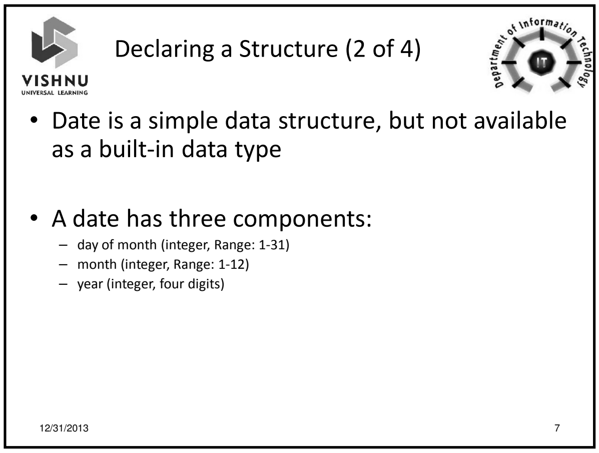


Declaring a Structure (1 of 4)

- A structure can be declared using the ‘struct’ keyword
- The set of variables that form the structure must be declared with a valid name similar to declaring variables
- Each variable inside a structure can be of different data type
- **Syntax:**

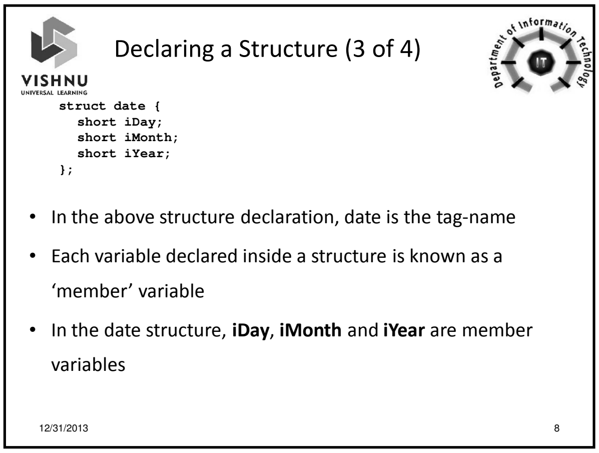
```
struct tag-name
{
      data-type member-1;
      data-type member-2;
      ……
      data-type member-n;
};
```

- A structure declaration ends with a semicolon



Declaring a Structure (2 of 4)

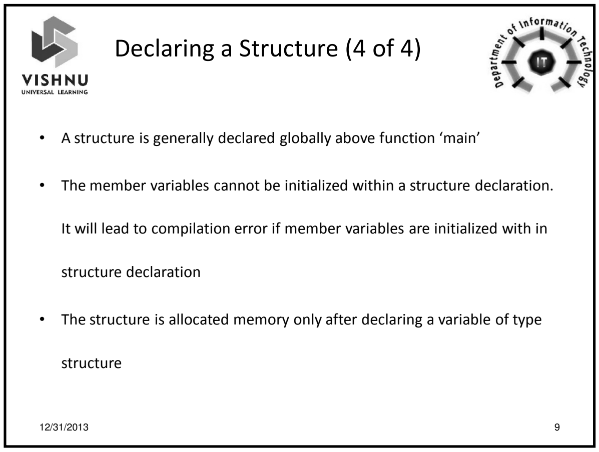
- Date is a simple data structure, but not available as a built-in data type
- A date has three components:
  - day of month (integer, Range: 1-31)
  - month (integer, Range: 1-12)
  - year (integer, four digits)



Declaring a Structure (3 of 4)

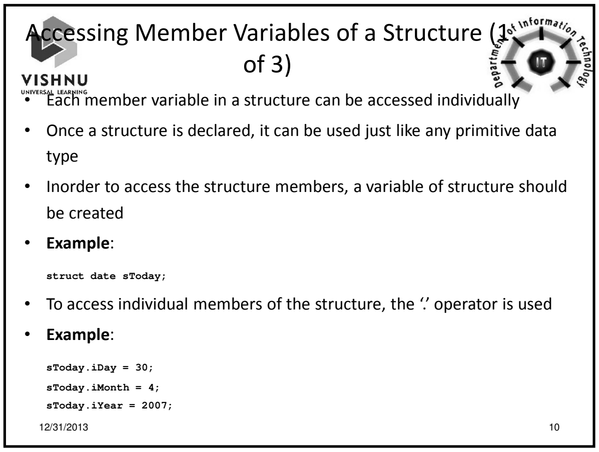
```
struct date {
  short iDay;
  short iMonth;
  short iYear;
};
```

- In the above structure declaration, date is the tag-name
- Each variable declared inside a structure is known as a ‘member’ variable
- In the date structure, **iDay**, **iMonth** and **iYear** are member variables



Declaring a Structure (4 of 4)

- A structure is generally declared globally above function ‘main’
- The member variables cannot be initialized within a structure declaration. It will lead to compilation error if member variables are initialized with in structure declaration
- The structure is allocated memory only after declaring a variable of type structure



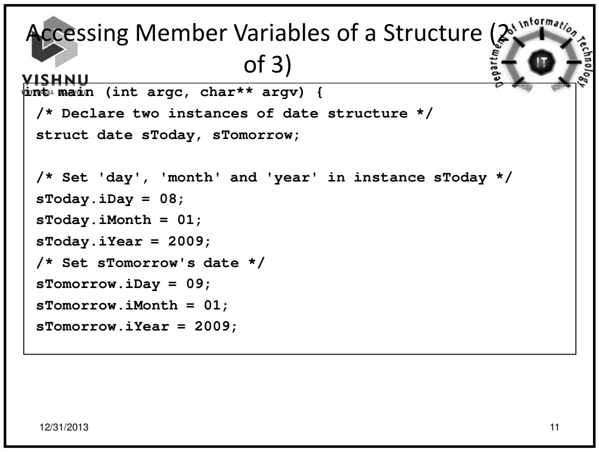
Accessing Member Variables of a Structure (1 of 3)

- Each member variable in a structure can be accessed individually
- Once a structure is declared, it can be used just like any primitive data type
- Inorder to access the structure members, a variable of structure should be created
- **Example**:

```
struct date sToday;
```

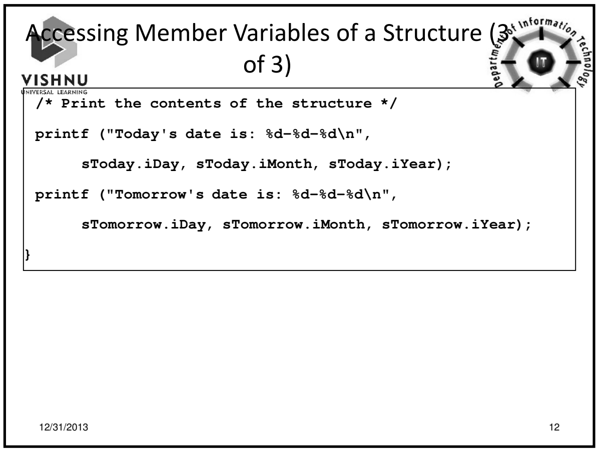
- To access individual members of the structure, the ‘.’ operator is used
- **Example**:

```
sToday.iDay = 30;
sToday.iMonth = 4;
sToday.iYear = 2007;
```



Accessing Member Variables of a Structure (2 of 3)

```
int main (int argc, char** argv) {
 /* Declare two instances of date structure */
 struct date sToday, sTomorrow;

 /* Set 'day', 'month' and 'year' in instance sToday */
 sToday.iDay = 08;
 sToday.iMonth = 01;
 sToday.iYear = 2009;
 /* Set sTomorrow's date */
 sTomorrow.iDay = 09;
 sTomorrow.iMonth = 01;
 sTomorrow.iYear = 2009;
```



Accessing Member Variables of a Structure (3 of 3)

```
 /* Print the contents of the structure */
 printf ("Today's date is: %d-%d-%d\n",
      sToday.iDay, sToday.iMonth, sToday.iYear);
 printf ("Tomorrow's date is: %d-%d-%d\n",
      sTomorrow.iDay, sTomorrow.iMonth, sTomorrow.iYear);
}
```

typedef Keyword (1 of 2)

- One type of data can be renamed with a different name using the ‘typedef’ keyword (typedef is a short form of ‘define type’)
- A struct date had to be instantiated by using:

```
/* Create an instance of date structure */
```

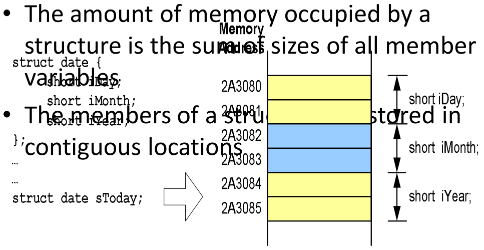
typedef Keyword (2 of 2)

- **Example:**

```
/* Declare the structure date */
struct _date {
      short iDay;
      short iMonth;
      short iYear;
};
/* Define the structure `_date' as a new data type
   `date' */
typedef struct _date date;

/* Create an instance of date structure */
date sToday;
```

Structures in Memory

- A structure instance occupies memory space
- The amount of memory occupied by a structure is the sum of sizes of all member variables
- The members of a stru stored in contiguous locations



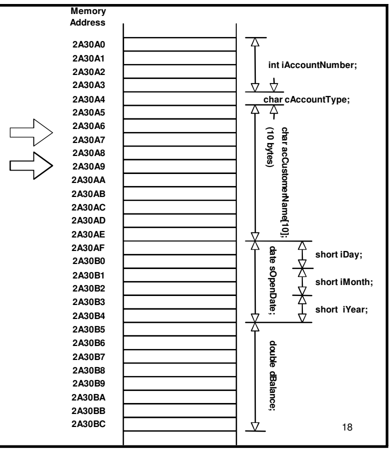
Structure within a Structure (1 of 3)

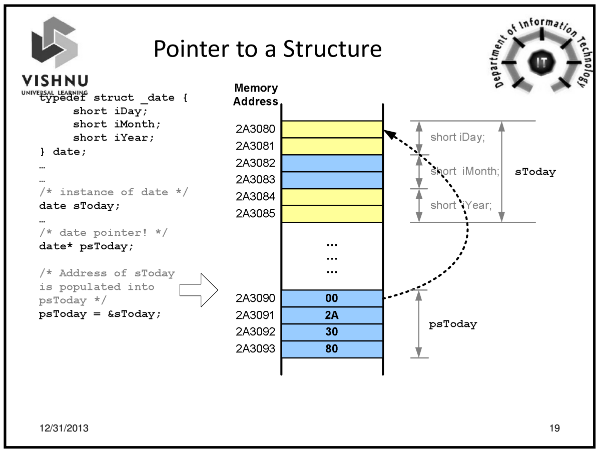
```
typedef struct _accountdetails {
      int iAccountNumber;
      char cAccountType;
      char acCustomerName[10];
      date sOpenDate;
      double dBalance;
} accountdetails;

/* Declare an instance of accountdetails */
accountdetails sAccount;
```

Structure within a Structure (2 of 3)

```
sAccount.iAccountNumber = 702984;
 sAccount.cAccountType = 'S';
 sAccount.dBalance = 5000.0;
sAccount.acCustomerName="George"

 /* Populating the date sturucture within the
    accountdetails structure */
 sAccount.sOpenDate.iDay = 1;
 sAccount.sOpenDate.iMonth = 6;
 sAccount.sOpenDate.iYear = 2005;
```

```
typedef struct _accountdetails {
          int iAccountNumber;
          char cAccountType;
          char acCustomerName[10];
          date sOpenDate;
          double dBalance;
          } accountdetails;

          accountdetails sAccount;
```

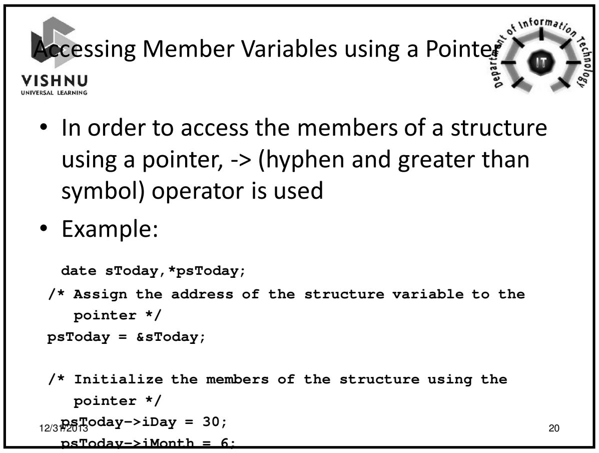




Pointer to a Structure

```
typedef struct _date {
      short iDay;
      short iMonth;
      short iYear;
} date;
…
…
/* instance of date */
date sToday;
…
/* date pointer! */
date* psToday;

/* Address of sToday
is populated into
psToday */
psToday = &sToday;
```

12/31/2013 19

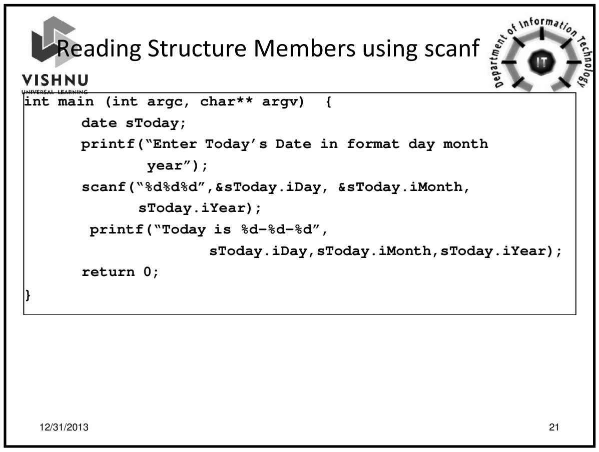


Accessing Member Variables using a Pointer

- In order to access the members of a structure using a pointer, -> (hyphen and greater than symbol) operator is used
- Example:

```
  date sToday,*psToday;
 /* Assign the address of the structure variable to the
    pointer */
 psToday = &sToday;

 /* Initialize the members of the structure using the
    pointer */
  psToday->iDay = 30;
  psToday->iMonth = 6;
```

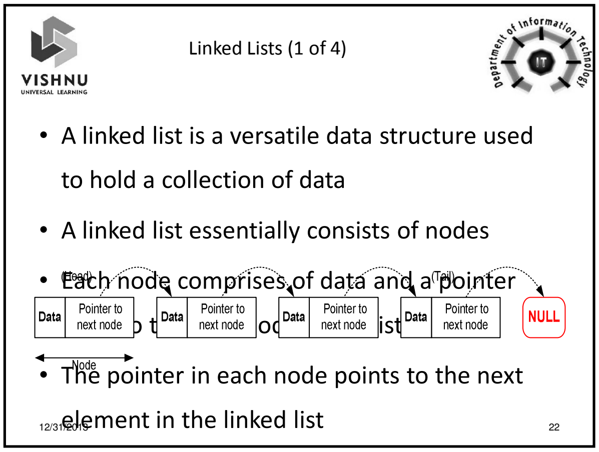
12/31/2013 20



Reading Structure Members using scanf

```
int main (int argc, char** argv)  {
       date sToday;
       printf("Enter Today's Date in format day month
               year");
       scanf("%d%d%d",&sToday.iDay, &sToday.iMonth,
              sToday.iYear);
        printf("Today is %d-%d-%d",
                      sToday.iDay,sToday.iMonth,sToday.iYear);
       return 0;
}
```

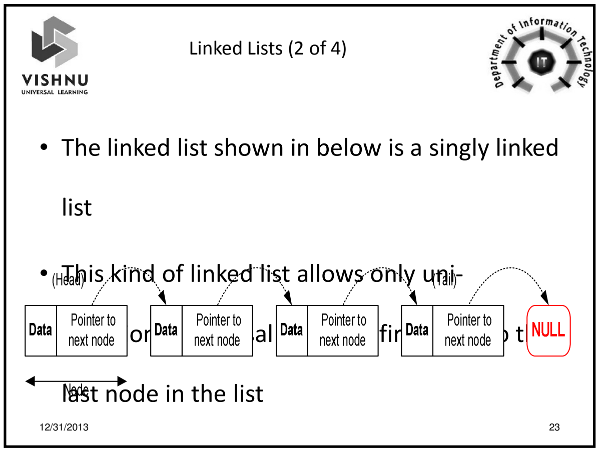
12/31/2013 21



Linked Lists (1 of 4)

- A linked list is a versatile data structure used to hold a collection of data
- A linked list essentially consists of nodes
- Each node comprises of data and a pointer o t oc ist
- The pointer in each node points to the next element in the linked list

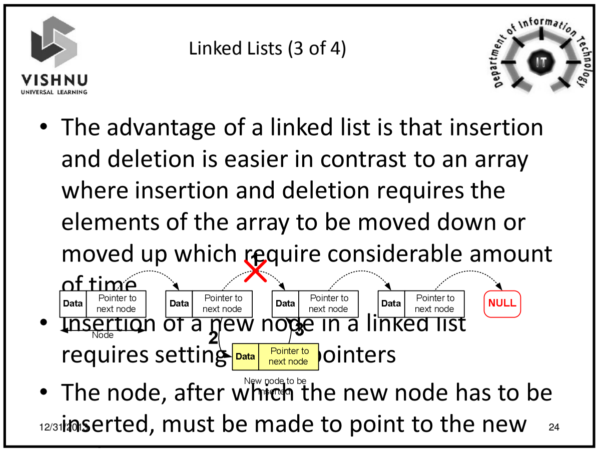
12/31/2013 22



Linked Lists (2 of 4)

- The linked list shown in below is a singly linked list
- This kind of linked list allows only uni- or al fir o t last node in the list

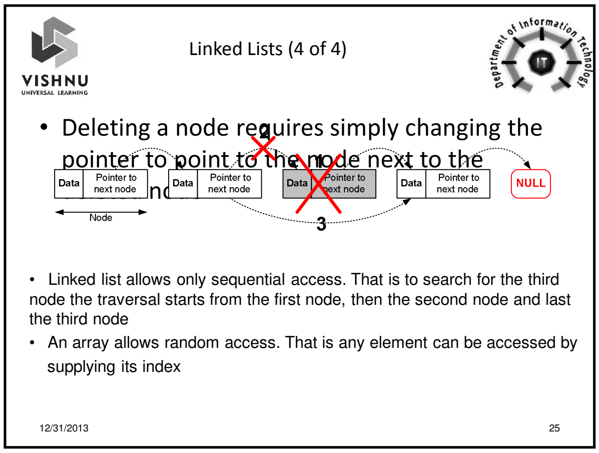
12/31/2013 23



Linked Lists (3 of 4)

- The advantage of a linked list is that insertion and deletion is easier in contrast to an array where insertion and deletion requires the elements of the array to be moved down or moved up which require considerable amount of time
- Insertion of a new node in a linked list requires setting pointers
- The node, after which the new node has to be inserted, must be made to point to the new

12/31/2013 24



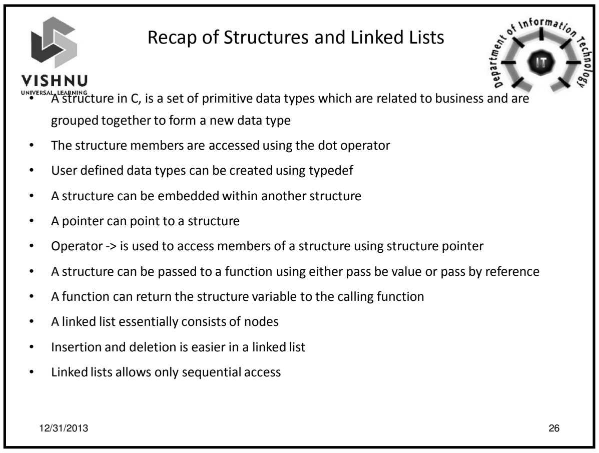
VISHNU UNIVERSAL LEARNING

Linked Lists (4 of 4)

- Deleting a node requires simply changing the pointer to point to the node next to the [illegible]

- Linked list allows only sequential access. That is to search for the third node the traversal starts from the first node, then the second node and last the third node
- An array allows random access. That is any element can be accessed by supplying its index

12/31/2013 25



VISHNU UNIVERSAL LEARNING

Recap of Structures and Linked Lists

- A structure in C, is a set of primitive data types which are related to business and are grouped together to form a new data type
- The structure members are accessed using the dot operator
- User defined data types can be created using typedef
- A structure can be embedded within another structure
- A pointer can point to a structure
- Operator -> is used to access members of a structure using structure pointer
- A structure can be passed to a function using either pass be value or pass by reference
- A function can return the structure variable to the calling function
- A linked list essentially consists of nodes
- Insertion and deletion is easier in a linked list
- Linked lists allows only sequential access

12/31/2013 26

C Programming Assignment

Note: In all the below problems, use and define as many as functions as possible.

1. Write a function, which checks whether one string is a sub-string of another string.

2. Write a program that accepts a sentence and returns the sentence. With all the extra spaces trimmed off. (In a sentence, words need to be separated by only one space; if any two words are separated by more than one space, remove extra spaces).

3. Write a program, which checks for duplicate string in an array of strings.

4. Write functions to insert and delete a string from an array of strings. Write a program that displays a menu to the user.

a) Insert String

b) Delete Strings

c) Exit

Depending on the user choice the program will call functions that will insert / delete a string from an array of strings.

5. Write a program to print whether the number entered is a prime/odd use functions.

6. Write a program that accepts input of a number of seconds , validates it and outputs the equivalent number of hours ,minutes and seconds.

C Programming Assignment

7. Write a program that can either add or multiply two fractions. The two fractions and the operation to be performed are taken as input and the result is displayed as output.

8. Write a recursive function to compute the factorial to a given number. Use the function to write program which will generate a table of factorials of numbers ranging from 1 to m where m is number entered by the user.

9. Write a program to implement student structure with following fields (Name, Roll no, Age) Eg: (Ramu,15,21).

Assessment Question – 3

1. Write the cleanest possible function you can think of to print a singly linked list in reverse. The format for the node should be a struct containing an integer value, val, and a next pointer to the following node.

2. Write a Program to reverse the complete linked list. The format for the node should be a struct containing an integer value, val, and a next pointer to the following node.

3. Write a program that, when run, will print out its source code. This source code, in turn, should compile and print out itself. (Fun fact: a program that prints itself is called a quine.)

4. Given an array of integers, the goal is to efficiently find the subarray that has the greatest value when all of its elements are summed together. Note that because some elements of the array may be negative, the problem is not solved by simply picking the start and end elements of the array to be the subarrray, and summing the entire array.

   For example, given the array

```
{1, 2, -5, 4, -3, 2}
```