Add instructor notes here.



# To understand the following concepts Characteristics of a good program Readable Maintainable Modular Guidelines for writing good code Coupling and Cohesion Robust program Difference between correctness and robustness Refactoring

# 2.1 Characteristics of a good program Characteristics of a Good Program



Characteristics of a good program are:

- Readable
  - Naming Conventions
  - · Layout techniques
  - Comments
- Maintainable
- · Remove Hardcoded constants
- Modular
- Coupling
- Cohesion
- Robust

If any program has following features then it is called as good program

1. Readable

If the names of variables are meaningful and the code is easy to understand

2. Maintainable

If the program is easy to understand and If it is easy to modify then the program is called as maintainable

3. Modular

A small unit of code for a single purpose

4. Coupling

Coupling or Dependency is the degree to which each program module relies on each other.

5. Cohesion

A cohesion is a measure of how the activities within a single module are related to one another.

Robustness

A program is said to be robust when it is fault tolerant. You should test your program extensively with positive and negative test conditions to prove its robustness

# 2.1.1 Readable Naming Conventions



Use Meaningful names

- Use Verb-noun format (be specific):
- Avoid generic names

Good variable names ensures readability.

Use naming conventions to distinguish Scope of data.

Use capitalized names for distinguishing constants among other variables.

Names should be readable, memorable and appropriate

A good name tends to express the "what" than the "how"

#### Naming Conventions

- Meaningful Names:
  - The name fully and accurately describes what the variable represents
  - The name should refer to the real-world problem rather than to the programming-language solution
  - Use Verb-noun format (be specific):

For example: Read-Employee-Record, Calculate-Deductions, Print-

#### Pay-slip

Avoid generic names:

For example: Process-inputs, Handle-calculations

- Good variable names are a key element of program readability.
- Use naming conventions to distinguish Scope of data like local/global.

For an Example, MAX\_USERS\_G variable scope is global

 Use capitalized names for distinguishing among type names, named constants, enumerated types, and variables.

```
2.1.1 Readable
Naming Conventions (Contd...)

Example of Poor Variable Names:

X = X - XX;

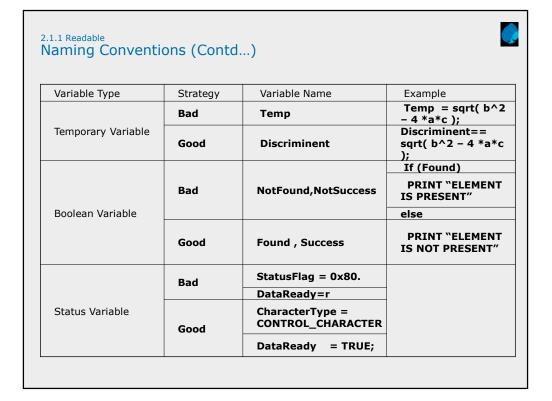
marypoppins = (superman + starship) / god;

X = X + Interest(X1, X);

Example of Good Variable Names:

Balance = Balance - LastPayment;

Balance = Balance + Interest (CustomerID, Balance);
```



Naming Status Variable: Think of a better name than "flag" for status variables.

```
if ( DataReady ) .....
```

if (CharacterType & PRINTABLE CHAR) ...

if ( ReportType == AnnualRpt ) ...

DataReady = TRUE;

CharacterType = CONTROL\_CHARACTER;

ReportType = AnnualRpt;

RecalcNeeded = FALSE;

CharacterType = CONTROL\_CHARACTER is more meaningful than

StatusFlag = 0x80.

**Naming Temporary Variable:** Use meaningful, descriptive names for Temporary variables. Don't use Temp, x or some other vague name.

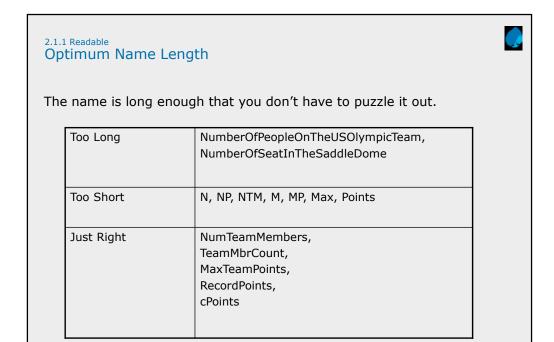
Bad Temporary variable Name

Temp =  $sqrt(b^2 - 4*a*c)$ ;

A Better approach is

Discriminant =  $sqrt(b^2 - 4*a*c)$ ;

Never use a numeric suffix to differentiate variables



**Naming Boolean Variables:** Use names that imply true or false like done, error, found, success as boolean variables

Avoid using negative names like NotFound, NotDone and NotSuccessful "If not notFound" is difficult to read Better way is to use "if found" instead

Give meaningful names for Loop index.

```
i = 0

ACCEPT Emp, Basic

G = B * 1.8 + 1700

PF = 0.12*B

T = ((G*12 - 150000)*0.3 + 19000)/12

N = G - PF - T - 200

PRINT Emp, B, G, PF, T, N

i=i +1

IF i==10 THEN Stop

Continue
```

In this example variable 'i' is used to count how many pay slips have been generated.

'Rec\_Count' can be used instead of 'i'

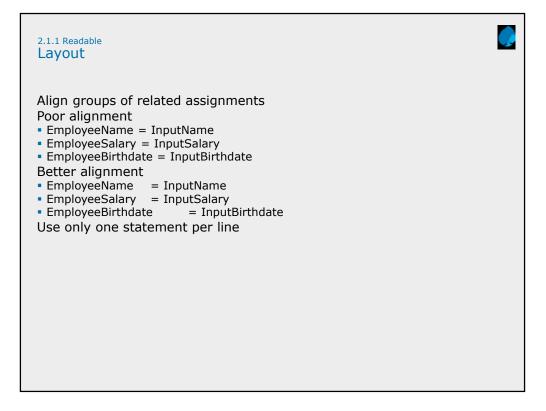
#### 2.1.1 Readable Layout Techniques

Ţ

Provide White space Grouping statements Use Blank lines wherever required Do uniform alignment of elements Use indentation to show the logical structure of a program.

#### Layout Techniques:

- Use white space to enhance readability.
  - · Use vertical and horizontal whitespace generously.
  - Indentation and spacing should reflect the block structure of the code.
  - · A long string of conditional operators should be split onto separate lines
- Grouping: ensure related statements are grouped together.
- Blank lines: separate unrelated statements from each other. Use blank lines to divide groups of related statements into paragraphs, to separate routines from one another, and to highlight comments.
- Alignment: align elements that belong together.
- Indentation: use indentation to show the logical structure of a program



#### Layout:

- Use "=" in a uniform positions in a group of statements.
- Use only one statement per line
- Within a block, Provide tabspace to start with a statement.

```
2.1.1 Readable
Layout (Contd...)

Example of bad layout: unformatted complicated expression

IF(('0' <= InChar and InChar <= '9') or ('a' <= InChar and InChar <= 'Z'))
THEN

Example of a good layout: readable complicated expression

IF (('0' <= InChar and InChar <= '9') or
('a' <= InChar and InChar <= 'Y') or
('a' <= InChar and InChar <= Y'Z') or
('A' <= InChar and InChar <= Y'Z') THEN
```

#### Layout Techniques:

- Statement belongs to if/loop should be started after a tab space
- A long string of conditional operators should be split onto separate lines. For example the below if statement

#### Better version of using For LOOP: Consider for loop have 3 sections

```
FOR (curr = *varp, trail = varp;
    curr != NULL;
    trail = &(curr->next), curr = curr->next)
```

# 2.1.1 Readable Self-documenting Code

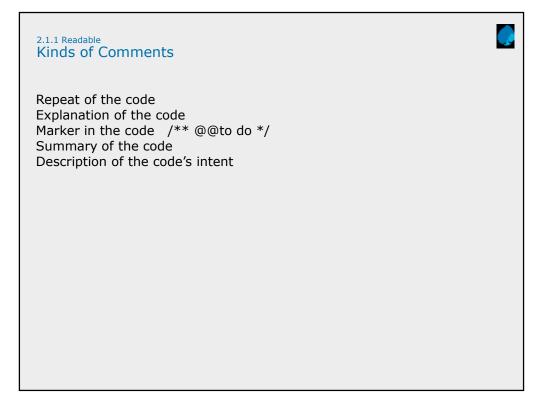


Main contributor to code-level documentation isn't only  $% \left( 1\right) =\left( 1\right) \left( 1\right) =\left( 1\right) \left( 1\right) \left($ 

- Good program structure
- Use of straight forward and easily understandable approaches
- Good variable names
- Good routine names
- Remove hard coded constants
- Clear layouts
- Minimization of control-flow and data-structure complexity

#### Self documenting code:

- Good Program Structure: Follow program structure like modularity
- Use of straight forward and easily understandable approaches.
- Good variable names : Meaningful variable names
- Good routine names : Meaningful routine names
- Remove hard coded constants: Use of named constants instead of literals
- Clear Layout : Follow all the layout techniques



#### Kinds of comments

Repeat of the code:

A repetitious comment restates what the code does in different words. It merely gives the reader of the code more to read without providing additional information

Explanatory Comments:

Explanatory comments are typically used to explain complicated, tricky or sensitive pieces of code. If the code is so complicated that it needs to be explained, its nearly always better to improve the code than it is to add comments. Make the code itself cleared and then use summary comments

- Marker in the code /\*\* @@todo\*/:
  todo is a form of comment used to convey that the code is yet to be completed by
  replacing todo comment with the functionality logic. For an example, /\*\*@@todo\*/
- Summary of the code: Use comment to provide description of the program like header block comment. For an example, //Program Description:
- Description of the code's intent: Use this comment, to describe the layout used

# 2.1.1Readable Commenting Techniques



#### Endline comments:

- Avoid Endline comments on single lines or multiple lines.
- Use Endline comments to annotate data declarations and to mark ends of blocks.

#### Paragraph comments:

- Focus paragraph comments on the why rather than the how.
- Use comments to prepare the reader for what is to follow.
- Avoid abbreviations, comments should be unambiguous.

#### **Commenting Techniques**

Commenting is amenable to several different techniques depending on the level to which the comments apply: program, file, routine, paragraph, or individual line

- Endline Comments
  - Endline comments are comments that appear at the ends of lines of code
  - Use endline comments to annotate data declarations like

#### Declare index as integer and store 0. //upper index of an array

- · Use endline comments to mark ends of blocks
- Paragraph Comments
  - Most comments in a well documented program are one sentence or two sentence comments that describe paragraphs of code
  - Use to describe the purpose of the block of code
  - For an example,

\* Search for an employee

\* \*

Additional notes for instructor

## 2.1.2 Maintainable Maintainable



If the program is easy to understand and if it is easy to modify then the program is called as maintainable.

Selection of proper data management technique helps to make code more simpler and maintainable.

Achieve maintainability by eliminating hard coded constants from the code

If the program is easy to understand and if it is easy to modify then the program is called as maintainable. Selection of proper data management technique helps to make code more simpler and maintainable. Achieve maintainability by eliminating hard coded constants from the code.

Additional notes for instructor

```
Program to find the circumference of a circle.

BEGIN

ACCEPT radius
circumference = 2 * 3.14159 * radius
PRINT "Circumference of a circle : ", circumference
END

Better Version

BEGIN

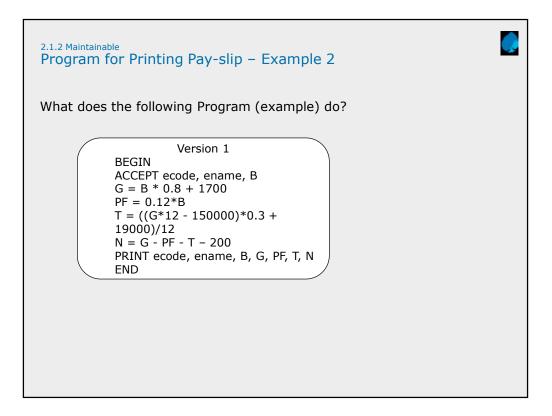
DECLARE CONSTANT PI AS INTEGER AND STORE
3.14159

ACCEPT radius
circumference = 2 * PI* radius
PRINT "Circumference of a circle : ", circumference
END
```

#### Example:

The given program is used to find the circumference of a circle based on radius. In the given code, hardcoded constant exists which is eliminated by using hard coded constant.

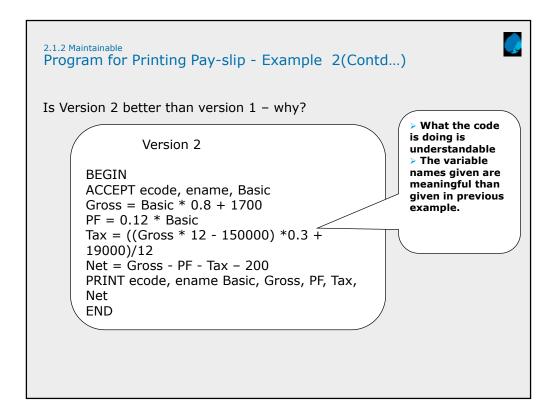
- Show this code to participants and give them time to find issues in the code
- Subsequently, explain the issues given in the notes pages



What are problems in the code above:

Variable names are not meaningful. Understanding meaning of variables G, B, T, N etc is difficult. Hence the code is not easily understandable We don't understand what the given code is doing?

- Show this code to participants and give them time to find improvements in the code
- Explain improvements
   Given in the notes pages



See the above example what it is doing?

It reads employee code, employee name and basic salary from keyboard.

Calculates Gross pay, Provident Fund (PF), Tax and Net pay.

Prints the pay slip

It is understandable because the variable names given are meaningful than given in previous example.

It shows that if you give meaningful variable names then it helps you to understand program better.

None

# 2.1.2 Maintainable Program for Printing Pay-slip - Issues



What are the issues in understanding the program calculating the gross pay?

- Poor readability
- · Comments are not added in the code
- Poor variable names
- Maintainability
- Hard-coded constants
- · The program is not modular

Programming standards that are to be followed are given below:

Use meaningful variable names. It helps to easily understand the program.
 Avoid using hard-coded constants in the program. Instead, declare them as constants at the beginning of the program, and use these constant names through the program. If value of constant changes later, the value can be modified for the declared constant at one place. The effect will be visible everywhere this constant name is used.

For example: tax rate

- · Include comments at the header and module level
- Don't use literal values in the program instead create constant variable to store fixed literal value for making program to be more maintainable.
- · Ensure that you have created more modular program.

None

# 2.1.2 Maintainable Program for Printing Pay-slip - Solution



What solutions do you recommend for these issues?

- Use Header block for comments.
- Use meaningful variable names.
- Eliminate hard coded constants from the code.
- Avoid use of obscure code

```
For an example:

HRA = 0.5 * Basic/*** avoid obscure code G = B * 0.8 +

1700 ***/

OPA = 0.3 * Basic /*** Offshore project allowance ***/

Conveyance = 1700
```

Use comments to describe program flow or complex sections of code.

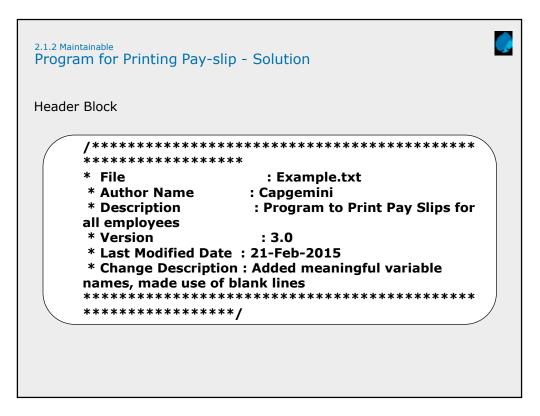
If we use above solution to update the code then the code will become more readable, maintainable.

By reading the code, G = B \* 0.8 + 1700, we do not understand what is meant by 1700, nor what is meant by 1.8 in the gross calculation.

However, the given code explains that Conveyance is 1700. HRA is 50% of Basic. Offshore Project allowance is 30% of Basic. So do not try to club these equations. Otherwise, maintenance of the code will be difficult, and understandability of code will reduce, as well.

Hence avoid such obscure code. Keep it simplified.

Explain the what can be added the header block



See the given header block of comments

It includes:

File : Give the name of the file.

Author name : Provide the author name who involved in the development of program

• Description : Detailed description of the program

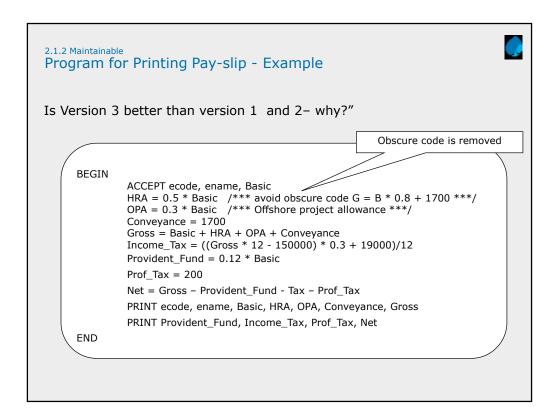
Version : Version of the program

· Last modified date : Date on which the program is last modified

• Change Description: History of changes happened in the program

- Show this code to participants and give them time to find changes in the code
- Explain what other improvements are required

Given in the notes pages and on the next slide



Why version 3 is better than version 1 and 2

- 1. Hard coded constants are given proper names. E.g HRA, OPA, Conveyance.
- 2. The given code is easy to maintain because if conveyance amount, percentage for HRA or OPA changes then we can easily understand where to make the change in the code, which was difficult in version 1 and 2
- 3. Code is readable as naming conventions are followed and layout is applied

Following improvements are required in the code:

a) If your code includes any complicated calculations It is necessary to document it in the code. In the above example the calculation of income tax includes many operations. What is meaning of it need to be documented in the code.

Steps involved in income tax calculations

1. Calculate annual salary: Gross \* 12

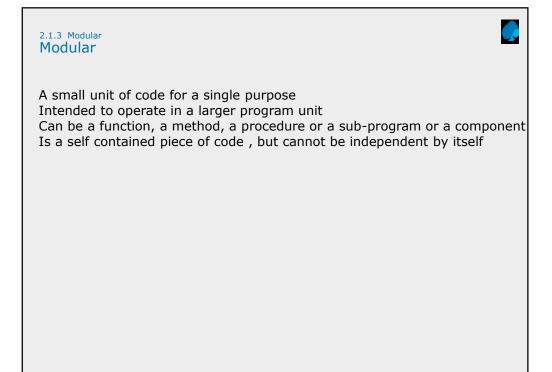
2. Calculate taxable amount: 1,50,000 will be subtracted from annual salary
 3. Calculate annual tax: Annual Tax = 30% of taxable amount + 19000

4. Calculate monthly tax Monthly tax = Annual tax/12

b) The given code is not modular. It doesn't have any modular structure. If the code is huge. It is performing various functions then it is better to create separate module for each function which increases reusability of the program. If any of these modules can be reused in some other application Programmer's efforts and time will be saved.

A good example is login module. Almost every application requires login screen which authenticate users. Such modules can be written once and used in multiple applications.

Additional notes for instructor



Modular: A small unit of code for a single purpose intended to operate in a larger program unit. It can be a function, a method, a procedure or a sub-program or a component. Module is a self contained piece of code, but cannot be independent by itself

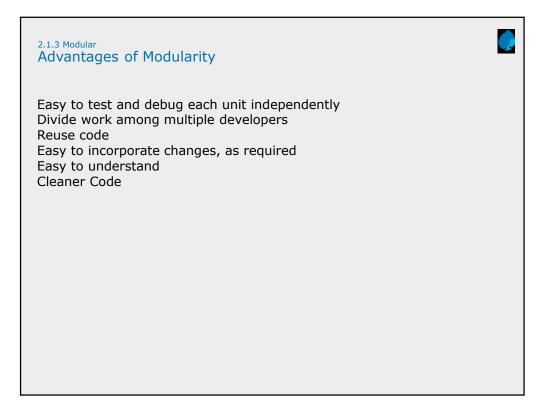
Additional notes for instructor

Reduce complexity
Better documentation
Avoid duplication of code
Avoid dependencies
Improve performance

#### Reasons for creating a module

- Reduce complexity: By using the abstracting power of modules, complex code can be made to appear simpler and easy to understand
- Better documentation: By putting a set code into a well defined module, makes the code self –explanatory
- Avoid duplication of code
- Avoid dependencies: Sections of code that depend on each other makes changes difficult to incorporate
- · Improve performance: Easy to test and debug units of code, than a long one

Additional notes for instructor



#### Advantages of Modularity

- Easy to test and debug each unit independently so that defects will be identified at the earlier stage.
- Divide work among multiple developers so that application development time will be reduced.
- Reuse code: Once a module is written, the same module can be reused in another application.
- Easy to incorporate changes, as required so that the code will be more maintainable.
- Easy to understand as the code is written in a single unit called module.
- Cleaner Code

Additional notes for instructor

#### 2.1.3 Modular Characteristics of well defined modules

Ţ

They always return same set of results for same set of inputs They perform a single well defined functionality High cohesion Low coupling Modular structure Meaningful names

#### Characteristics of well defined modules

- They always return same set of results for same set of inputs
- They perform a single well defined functionality
- High Cohesion do one thing, and do it well
- Low Coupling reduce dependencies between modules

Ideally when there is "high cohesion" and "low coupling", one can change the implementation of a routine without impacting the call interface.

For example: A sort routine can change its algorithm from "Bubble" to "Quick sort" – without causing the calling code to break.

#### Meaningful names

Use Verb-noun format (be specific):

For example: Read-Employee-Record, Calculate-Deductions, Print-Pay-slip Avoid generic names:

For example: Process-inputs, Handle-calculations

Additional notes for instructor

2.1.3 Modular
Best practices to follow when creating modules



Few of the best practices to follow when creating modules

- Informative module name
- Module logic should be specific
- Test each module immediately once it is created
- Parameter Passing should be accurate.
- Ensure that there is no "Type mismatch" for any parameter.
- Ensure that there is no "NOPS" module definition

#### Best Practices to follow when creating modules

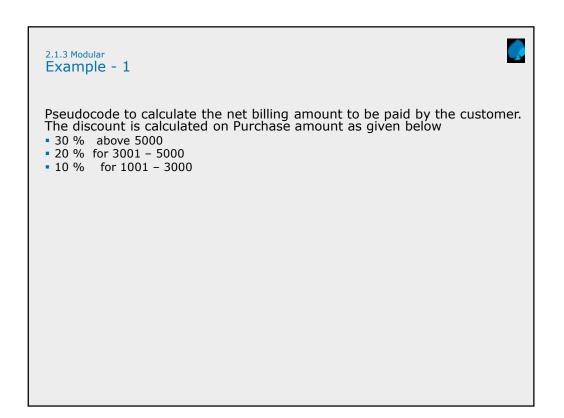
- Informative Module Name: Give the module an informative name like getProductPrice, calculateSalary, printDetails.
- Module logic should be specific: Identify a clear purpose for the module before you start writing
- · Test each module as it is created by performing unit testing
- Parameter passing should be accurate: Ensure that the number of parameters, and the sequence is correct for the module call.
- Ensure that there is no "Type mismatch" for any parameter.
- Ensure that there is no "NOPS" (No Operation) Module definition. i.e. Empty module definition shouldn't be there.

#### calculateTotal(Integer price, Integer quantity)

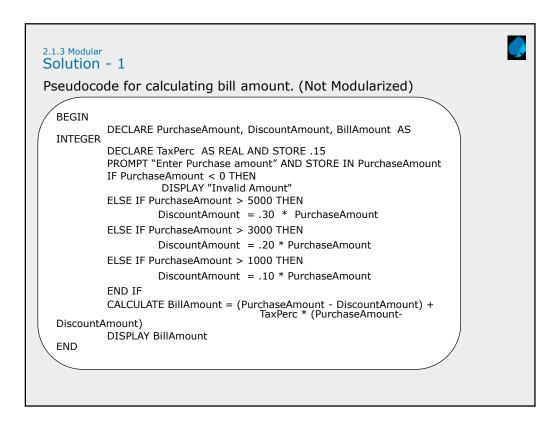
Refer the valid and invalid statements to invoke a module

- calculateTotal(3,5); //Valid
- calculateTotal(4,3,4); //Invalid
- calculateTotal('Test',3); //Invalid

Additional notes for instructor



Additional notes for instructor



The above pseudocode for calculating bill amount is not modularized

Additional notes for instructor

```
2.1.3 Modular
Solution - 2

Modularized Pseudocode for calculating bill amount.(Contd..)

SUB CalculateDiscount(PurchaseAmount)

DECLARE DiscountAmt AS INTEGER AND STORE 0

IF PurchaseAmount > 5000 THEN

DiscountAmt = .30 * PurchaseAmount

ELSE IF PurchaseAmount > 3000 THEN

DiscountAmount = .20 * PurchaseAmount

ELSE IF PurchaseAmount > 1000 THEN

DiscountAmt = .10 * PurchaseAmount

END IF

RETURN DiscountAmt

END SUB
```

The above pseudocode for calculating bill amount is modularized

Additional notes for instructor

2.1.3 Modular
Code Considerations During Modularization

During modularization of the code we need to decide:

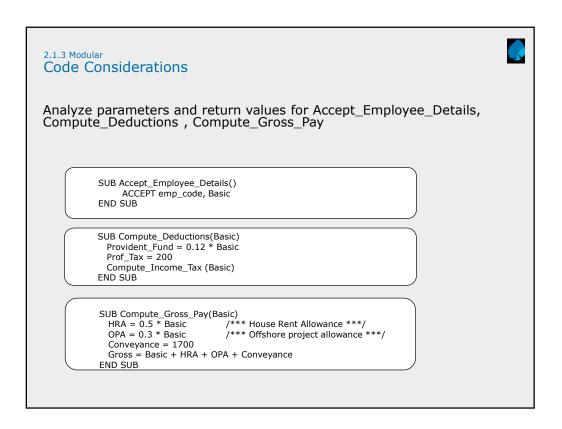
Input Parameters:

For each lower level routine, what input parameters should be passed?

Output Parameters:

Should the output be in the form of a "parameter" or a "return value"?

Discuss about the input and output parameters in the given module.



Additional notes for instructor

Guidelines to follow while using arguments

Identify input and output parameters
Only include the parameters which are used by the module
If parameters are related to each other, then pass record as an argument instead of multiple parameters

#### Guidelines to follow while using arguments

- Identify input and output parameters: After analyzing the requirement, identify
  input and output parameters before writing module code. For an Example, if you
  want to create a module for implementing a logic related to retrieving product
  details based on product id, then input parameter is producted and output
  parameter is variable of type record which contains product details.
- Only include the parameters which are used by the module, never pass unused parameters.
- If parameters are related to each other, then pass record as an argument instead
  of multiple parameters which strive for high cohesion and low coupling. For an
  example, refer the below module code snippet to add an employee details
  SUB addEmployee(empld, name, salary)

**END SUB** 

Instead use the below module signature

SUB addEmployee(Employee emp) //Consider Employee is a record END SUB

Additional notes for instructor

2.1.3 Modular Best practice to follow for return values



Have a single exit point from each module Return null values instead of zero length arrays

Use well defined exceptions and error codes

Based on the number of values to be returned, use specific return type like array or records.

Consider the side effects of return values while integrating all the modules together.

#### Best Practice to follow for return values

Use only one return statement in each module as shown below:

SUB checkDigit(number) DELCARÉ result ÀS BOOLEAN IF(num>0) THEN result=true **ELSE** 

result=false

**END IF** 

RETURN result

#### **END SUB**

- Return null values instead of zero length arrays.
- Return exceptions/error code if an invalid input is accepted.

SUB getProductPrice(productId)

DECLARE errorcode AS INTEGER AND STORE 0

IF(elementfound(productId)) THEN RETURN productPrice

**ELSE** 

errorcode = -1

RETURN errorcode;

**END IF** 

#### **END SUB**

Use array as return type if more than one value has to be returned of same type or use record if more than one value has to be returned of different type.

# 2.2 Guidelines for writing good code Guidelines for writing good code



Program for people, not the machine
Analyze the case study well
Design first then code
Develop in small steps
Keep your code simple
Understand the standards
Document your code
Paragraph your code
Paragraph and punctuate multi-line statements
Use white space
Specify the order of operations. (Use parenthesis)

#### Guidelines for writing good code

- While writing program, keep in your mind that program will be used by people, so make program to be user friendly.
- Before starts with development, analyze the case study well to incorporate all the requirements.
- Do the high level (like database design) and low level designing(pseudocode) of an application before working in coding phase.
- Create program in an incremental approach, so that after implementation of each logic it can be easily tested for finding defects.(As finding defects in earlier stage, decreases cost and save development time).
- Make your code to more simple by avoiding the usage of complex data structure/constructs
- Understand the standards:
  - All the coding standards as well as processed should be understandable and apply the same in your code.
  - · Believe in them
  - Make them part of your quality assurance process
  - · Adopt the standards that make the most sense for you
- Document your code using comments for making it to be more readable
- Paragraph your code by applying modularity to make code reusable.
- Use whitespace as a layout technique to make code more readable
- Specify the order of operations using parenthesis for prioritizing

Explain the meaning of coupling with examples

#### Example for Coupling:

Make Tea in Pot (min)
Pour Boiling Water in Pot
Add Tea powder in Pot
Wait min Minutes

**Well Coupled** - all parameters are needed

# Make Tea in Pot (min, temp)

Pour Boiling Water in Pot Add Tea powder in Pot Wait min Minutes

Not Well Coupled - temp variable is not used

## 2.3 Coupling and cohesion Coupling



Coupling or Dependency is the degree to which each program module relies on each other.

Tightly coupled systems disadvantages:

- A change in one module forces a ripple-effect of changes in other modules.
- Assembly of modules might require more effort and/or time due to the increased inter-module dependency.
- A particular module might be harder to reuse and/or test because dependent modules must be included

If a module does only one thing, we need to pass less parameters. If a module does too many things, we need to pass more parameters. Passing more parameters means high coupling.

We should strive for low coupling.

Additional notes for instructor

#### 2.3 Coupling and cohesion Coupling



Loosely coupled systems advantages:

- A change in one module usually does not force a ripple-effect of changes in other modules.
- Assembly of modules might require less effort and/or time due to the decreased inter-module dependency.
- A particular module might be easier to reuse and/or test because dependent modules do not need to be included

Low Coupling – reduce dependencies between modules

- Note that changes in one routine should not normally impact other routines, as long as the interface is the same.
- Remember that, in case too many things are done in one routine, a lot of data needs to be shared. This increases the dependencies, and also the chances of defects
- Consider "Smaller interface" (low coupling) versus "long list of parameters" (high)
- Consider data sharing through "parameters" (low) versus "global data" or "global files" (high)
- Note that passing "flags" that control the processing implies High coupling.

Additional notes for instructor

# 2.3 Coupling and Cohesion Cohesion



A cohesion is a measure of how the activities within a single module are related to one another.

Principle of Cohesion:

A module should do one thing and do it well

If a module follows the given principle, then it is high cohesion.

Writing function with high cohesion is always good practice. Ideally, we should strive for **High Cohesion** and **Low Coupling**.

# For example:

1. Sin (x); Cos (x); Tan (x); (High Cohesion, Low Coupling)

#### Note:

A good program requires high cohesion and low coupling.

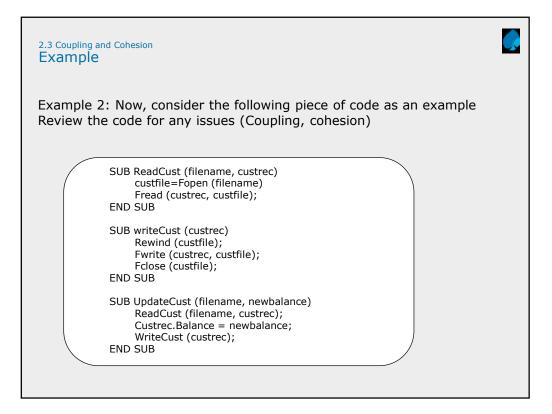
2.Trig (Type, x) where type is Sin, Cos, or Tan; (Less Cohesion, High Coupling)

#### Note:

Since in function trig we are trying to add all three functions together, we require to pass one extra parameter i.e. Type. This parameter indicates whether to calculate Sin, Cos, or Tan.

- a. More number of parameters are required to pass, so it is High Coupling.
- b. b. The function is not performing just a single task, so it is Low Cohesion.

Additional notes for instructor



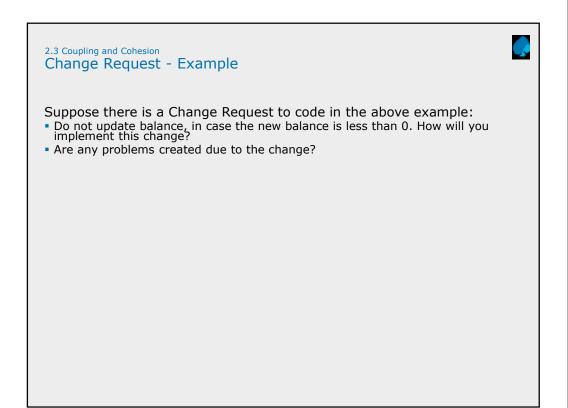
Consider Fopen, Fwrite, Rewind, Fclose are predefined functions.

Fopen: To open a file

Fwrite: To write data to a file Fclose: To close the opened file

Rewind : Moving cursor back to the first position of a file.

Explain why the program will crash because of change request. It is discussed on further slides.



Additional notes for instructor

```
2.3 Coupling and Cohesion
Change Request - Example

Sometimes the change is made as follows

SUB UpdateCust (filename, newbalance)
ReadCust (filename, custrec);
IF(newbalance >= 0) THEN
Custrec.Balance = newbalance;
WriteCust (custrec);
END IF
END SUB

This means file will not be closed whenever "newbalance < 0". This is because file gets closed in writecust, and writecust will not be called.
After a few hours the program will crash saying "too many open files".
```

Once participants found the issues explain the problems in the code because of not following principle of coupling and cohesion.

Cohesion principle:

# 2.3 Coupling and cohesion Drawbacks in the given code



ReadCust is doing more than just reading. It is also opening the file. WriteCust is doing more than just writing. It is also closing the file.

• This violates the principle of Cohesion:

What is the other drawback in the given code with respect to performance overheads?

 Every time we need to write a record, we are opening and closing the file. This will slow down the program!

#### **Cohesion:**

- It means, the function should do one thing only, and the code should not be mixed up.
- As a result, the code becomes more readable and more maintainable.

Once participants found the issues explain the problems in the code because of not following principle of coupling and cohesion.

Cohesion principle:

2.3 Coupling and cohesion

How can we avoid this?



Use a STATIC variable to represent the STATE of the file.
Use global variable for accessing the STATE of the file in an application(Not recommended)

Use higher-level calling routine.

Encapsulate functions to perform I/O operations in a separate module.

Steps to be taken care for avoiding drawbacks on the implementation of cohesion and coupling:

- Use a STATIC variable(cust-file-already-open) to represent the STATE of the file as OPEN or not
- We cannot access the STATIC variable cust-file-already-open outside this routine
- We will be able to access it outside the routine if we declare it as GLOBAL.
   However, that is not a good practice.
- It is better to open and close the cust-file in the higher-level calling routine. The routine should call UpdateCust only to write the record.
- The calls to fread / fwrite / fopen / fclose are encapsulated or wrapped in the modules ReadCust / WriteCust / OpenCust / CloseCust respectively.

Once participants found the issues explain the problems in the code because of not following principle of coupling and cohesion.

Cohesion principle:

```
SUB ReadCust (filename, custrec)
Fread (custrec, sizeof (custrec), 1, custfile);
END SUB
SUB WriteCust (custrec)
Fwrite (custrec, sizeof (custrec), 1, custfile);
END SUB
SUB OpenCust (filename, mode, Cfile)
Cfile = Fopen (filename, mode);
END SUB
SUB CloseCust (Cfile)
Fclose (Cfile);
END SUB
```

The calls to fread / fwrite / fopen / fclose are encapsulated or wrapped in the modules ReadCust / WriteCust / OpenCust / CloseCust respectively.

Explain why the program will crash because of change request. It is discussed on further slides.

```
2.3 Coupling and Cohesion
Revised Code (Contd..)

SUB UpdateCust (filename, newbalance)

STATIC BOOLEAN cust-file-already-open = FALSE;

IF (newbalance < 0) THEN

{return;}

ELSE IF (NOT cust-file-already-open) THEN

OpenCust (filename, "r+", Cfile);

END IF

ReadCust (Cfile, custrec);

Custrec.Balance = newbalance;

WriteCust (custrec);

END SUB

/**** CloseCust (Cfile);

Now we need to close the file only once at the end ***/
```

Used STATIC variable(cust-file-already-open) to represent the STATE of the file as OPEN or not

Additional notes for instructor

# 2.3 Coupling and Cohesion Advantages



Wrapper modules help isolate system specific code in one place rather than all over the application.

- They are easier to change during migration to different versions of Compiler or OS.
- They are extremely useful when porting code:
  - · To different platforms, or
- To different database management systems
- For example: In the code given in Example 2, "Read\_cust" is acting as "wrapper module". It is hiding the details about how to read the file.

None

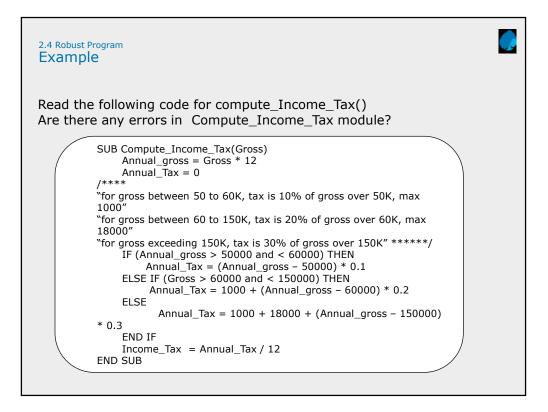
# 2.4 Robust Program What is Robust program?



Robust program anticipates common and uncommon problems To ensure software is well defended, one should write robust program Robust program ensures that software handles invalid inputs reasonably preventing abnormal termination

The program should terminate gracefully and provide appropriate debugging information for the programmer

Robust program ensures that software handles invalid inputs reasonably preventing abnormal termination. Robust program anticipates common and uncommon problems. To ensure software is well defended, one should write robust program. The program should terminate gracefully and provide appropriate debugging information for the programmer



In the above program we are not considering a case where annual\_gross is less than 40000. As a result, in case the statements at the bottom of the program are executed, we will get wrong result.

Note: When we use a nested IF, be careful while writing the last ELSE. This is because, the program will be executed for all unhandled conditions, as well.

The above program has one more mistake. We have used Gross instead of Annual\_Gross. This error will not be detected by the compiler because we are using Gross as a variable form during calculation of monthly gross. This will result in wrong output.

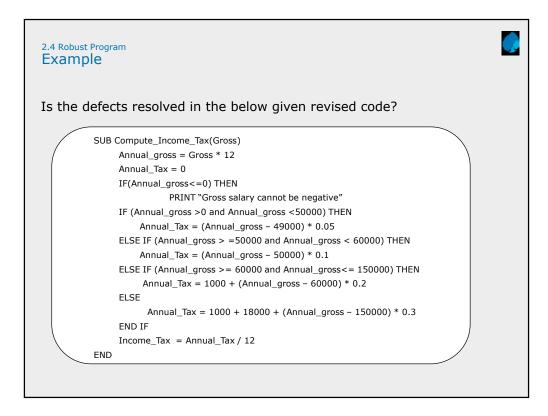
#### 2.4 Robust Program Defects Introduced in the Program



Do you see any new defects introduced in the program?

- What tax will be calculated in the Compute\_Income\_Tax module for an income of 40000?
- Nested IF-THEN-ELSE should take care of all possible conditions. Is the nested clause doing so?
- Be careful about the last ELSE it may end up doing more than it should
- Program has mistakes in variable name references: Example: Gross instead of Annual\_Gross

For resolving all the defects mentioned in the slide, rework on the code.



In the above program we have considered a case where annual\_gross is less than 40000. We have renamed, Gross as Annual\_Gross. The statements in ELSE block will be executed only if Annual\_Gross value is greater than 150000.

Annual Gross salary is calculation is taken care based on the below data

```
/****
```

"for gross less than 50K, tax is 5% of gross over 49K"

"for gross between 50 to 60K, tax is 10% of gross over 50K, max 1000"

"for gross between 60 to 150K, tax is 20% of gross over 60K, max 18000"

"for gross exceeding 150K, tax is 30% of gross over 150K" \*\*\*\*\*\*/

None

# 2.4 Robust Program Make a Program Robust



Will the program work (or fail gracefully) with unexpected input?

• Check if it can display error message as "Input cannot be negative".

Do not assume everything will be alright.

Check for unexpected inputs or conditions.

Remember GIGO – Garbage In, Garbage Out.

Provide meaningful error messages

#### Is the program robust?

- We test the program with different expected values, and it works fine as per our expectation.
- · However, what if somebody provides an unexpected input?
  - · will the program give proper error message and stop gracefully, or
  - · will the program fail and give some garbage output
- Hence to make the program robust, add appropriate messages to handle unexpected input as mentioned in the slide.
- How can we make the program robust?

Check if it can display error message as "Input cannot be negative" Check if it can accept extreme values for inputs.

For example: Basic = 1 crore

Check if it can handle Output / Printer related problems.

· Check whether:

"Can the Tax calculated be negative?" or

"Can the Net Pay calculated be negative?"

# 2.4 Robust Program Difference between correctness and robustness



Correctness means building code which never returns inaccurate result

- Safety critical applications tend to focus on correctness, failure to achieve a result being regarded as better than inaccurate result.
- For an Example, Software which controls a Bank machine should focus on correctness because it is better to return no value than an inaccurate value when an error could mean dispensing or recording wrong amounts of money

Robustness favor's the return of any result even inaccurate one

- Consumer applications typically favor robustness as any result is better than software crashing
- For an Example, Web browsers should focus on robustness as they often have to handle invalid input.

First determine whether you want the program to primarily offer robustness or correctness

- Examples of Robustness
  - Consumer applications typically favor robustness as any result is better than software crashing
  - You may want a word processing program to sometimes display unwanted characters rather than to shut down when it detects them
  - Web browsers should focus on robustness as they often have to handle invalid input.
- · Examples of Correctness
  - Safety critical applications tend to focus on correctness, failure to achieve a result being regarded as better than inaccurate result.
  - E,g software which controls radiation equipment for patients is best shut down if it receives bad input for a radiation dosage.
  - Software which controls a Bank machine should focus on correctness because it is better to return no value than an inaccurate value when an error could mean dispensing or recording wrong amounts of money

Additional notes for instructor

# 2.5 Refactoring Definition of Refactoring



Code refactoring is the process of rearranging the source code without modifying its functional behavior in order to improve some of the non-functional attributes of the software

Refactoring is more essential as it addresses the problems like

- Maintainability
- Extensibility

Some of the refactoring task/techniques which can be used to refactor the code are:

- Removal of Dead and duplicated code
- Method/Field/Component name Refactoring
- Architecture Driven Refactoring
- Method Slicing/Extraction

Refactoring is the process of clarifying and simplifying the design of existing code, without changing its behavior. Refactoring requires care because it can introduce bugs to the code

Refactoring is more essential as it addresses the problems like

- Maintainability Code is not maintainable
- Extensibility Very difficult to add new feature or upgrade an existing feature

Refactoring task/techniques are:

- Removal of Dead and duplicated code Remove the unused variables/code, unreachable statements and duplicated code.
- Method/Field/Component name Refactoring Name of a method/field/component might be confusing, provide a meaningful name for the same and ensure the changes are reflected in other references wherever the variable/field/component is used.
- Architecture Driven Refactoring Few functionalities in an application can be
  potentially reused in other applications. In order to reuse such a functionality,
  separate the code in a separate component by adhering to the architecture
  design.
- Method Slicing/Extraction Longer method needs to be broken up into smaller ones to enhance readability and maintainability. Also need to achieve cohesion by implementing only one logic in a method with more suitable name.

Additional notes for instructor

# 2.5 Refactoring Benefits of Refactoring

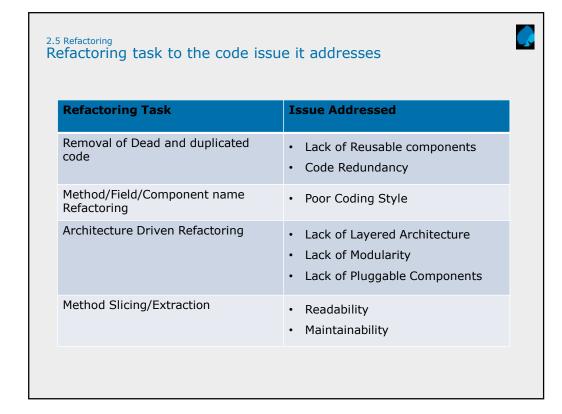


Improves Readability and modularity
More Maintainable
Improves extensibility
Improves internal structure of an application
Makes code more flexible and reusable
Improves design of a software
Retains with the same behavior even though internal structure changed
If the code works, then use refactoring as valid activity

# Benefits of Refactoring:

- Improves code readability and modularity by improving code structure and design
- Reduced complexity to improve the maintainability of the source code
- · Improves extensibility: Easier to add new features
- Refactoring involves improving internal structure of a software without altering its external behavior
- Goal of refactoring is to make the software more flexible and reusable
- It involves changing the design of the software after it has been coded to improve the design of a software. Refactoring is mainly used to improve badly designed software.
- Ensure software's external behavior remains the same after refactoring
- Refactoring is a valid activity only when the code is already in working state

Discuss about the code issues which will be resolved by implementing refactoring task.



Each refactoring task is addressed to the code issue it addresses.

Explain the code to participants and show that following lines are repeating Amt = Base \* TX\_RATE; Calc = 2 \* Basis (Amt) + Extra (Amt) \* 1.05;

```
2.5 Refactoring
Example
Example 1: Can we refactor the following code?
                  IF (State == TEXAS) THEN
                       Rate = TX_RATE;
                       Amt = Base * TX_RATE;
                       Calc = 2 * Basis (Amt) + Extra (Amt) * 1.05;
                  ELSE IF ((State == OHIO) OR (State == MAINE)) THEN
                       Rate = (STATE == OHIO)? OH RATE: ME-RATE;
                       Amt = Base * Rate;
                       Calc = 2 * Basis (Amt) + Extra (Amt) * 1.05;
                       IF (State == OHIO) THEN
                                        Points = 2;
                       END IF
                  ELSE
                       Rate = 1;
                       Amt = Base;
                       Calc = 2 * Basis (Amt) + Extra (Amt) * 1.05;
```

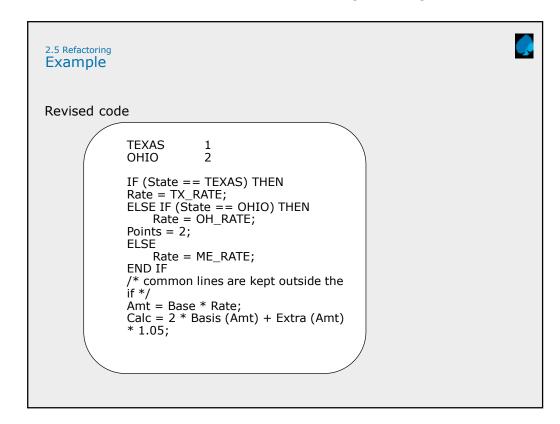
The above code is used to calculate tax based on the state. Tax rate varies for each state but similarity exists in the tax calculation.

Rate, Amt and Calc variables are assigned values in each if-else condition.

Use Refactoring for performing the below task:

- Avoid Duplicate code in each if else condition
- Common code should be kept outside if condition

Additional notes for instructor

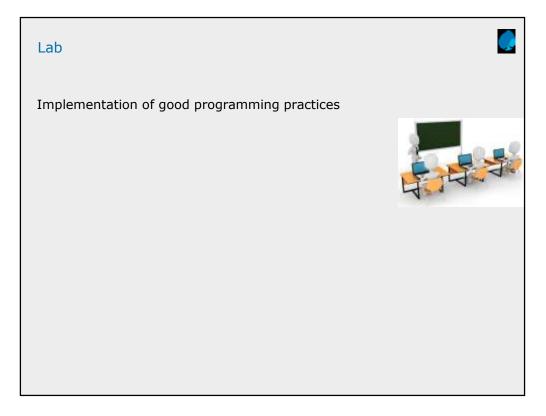


**Note:** The code is more efficient and readable as compared to the previous code.

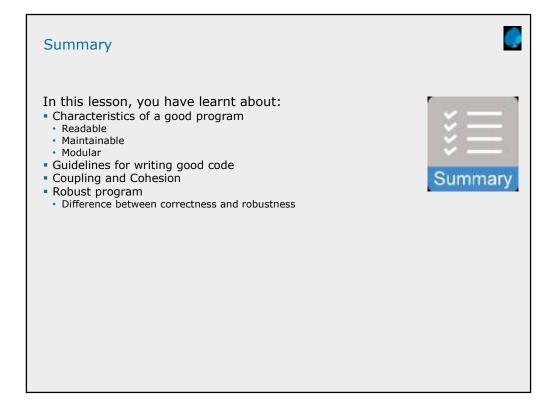
#### Are any further improvements possible?

If the program is used for three states, it is likely to be used for other states, as well, in the future.

Instead of using a nested IF ELSE statement, using a table or an array for tax rates of all states will make it easy to add more states later on.



Write pseudocode for all assignments in lab 2 to apply the characteristics of good programming practices like modularity approach.



Question 1: A,B,C Question 2: B

# **Review Question**



Question 1: Why should you make code self documenting?

- A. To help the review process
- B. To improve the quality of the code
- C. To make it easier to debug
- D. To meet international standards

Question 2: How can you minimize hard coding?

- A. By changing named constants in several places in your code
- B. By using enumerated types instead of named constants
- C. By using named constants to replace Booleans when you require more answers than true or false



Question 3: A,C,D Question 4: A,B,C

# **Review Question**



Question 3: What are the objectives of good layout?

- A. An accurate logical structure
- B. To improve compatibility with compiler
- C. To improve readability
- D. To withstand modifications

Question 4: What are the guidelines for good layout?

- A. Use braces as boundaries to demarcate block of code
- B. Use indentation to show logical structure of code
- C. Use line breaks for statements over 80 characters
- D. Use white space sparingly



Question 5 : A,C Question 6: C,D

# **Review Question**



Question 5 : You are writing a module to update the visitor counter on a web page ,what steps can you take to make this module as strong as possible ?

- A. Call the module pageCountUpdate()
- B. Call the module webPage6()
- C. Test the module as soon as it is complete
- D. Wait until all modules have been added to the program before testing it



- A. Bank machine software
- B. Radiation machine software
- C. Web browser software
- D. Word processing software

