Coding

The coding is the process of transforming the design of a system into a computer language format. This coding phase of software development is concerned with software translating design specification into the source code. It is necessary to write source code & internal documentation so that conformance of the code to its specification can be easily verified.

Coding is done by the coder or programmers who are independent people than the designer. The goal is not to reduce the effort and cost of the coding phase, but to cut to the cost of a later stage. The cost of testing and maintenance can be significantly reduced with efficient coding.

Goals of Coding

1. **To translate the design of system into a computer language format:** The coding is the process of transforming the design of a system into a computer language format, which can be executed by a computer and that perform tasks as specified by the design of operation during the design phase.
2. **To reduce the cost of later phases:** The cost of testing and maintenance can be significantly reduced with efficient coding.
3. **Making the program more readable:** Program should be easy to read and understand. It increases code understanding having readability and understandability as a clear objective of the coding activity can itself help in producing more maintainable software.

For implementing our design into code, we require a high-level functional language. A programming language should have the following characteristics:

Characteristics of Programming Language

Following are the characteristics of Programming Language:

Characteristics of Programming Language

Readability: A good high-level language will allow programs to be written in some methods that resemble a quite-English description of the underlying functions. The coding may be done in an essentially self-documenting way.

Portability: High-level languages, being virtually machine-independent, should be easy to develop portable software.

Generality: Most high-level languages allow the writing of a vast collection of programs, thus relieving the programmer of the need to develop into an expert in many diverse languages.

Brevity: Language should have the ability to implement the algorithm with less amount of code. Programs mean in high-level languages are often significantly shorter than their low-level equivalents.

Error checking: A programmer is likely to make many errors in the development of a computer program. Many high-level languages invoke a lot of bugs checking both at compile-time and run-time.

Cost: The ultimate cost of a programming language is a task of many of its characteristics.

Quick translation: It should permit quick translation.

Efficiency: It should authorize the creation of an efficient object code.

Modularity: It is desirable that programs can be developed in the language as several separately compiled modules, with the appropriate structure for ensuring self-consistency among these modules.

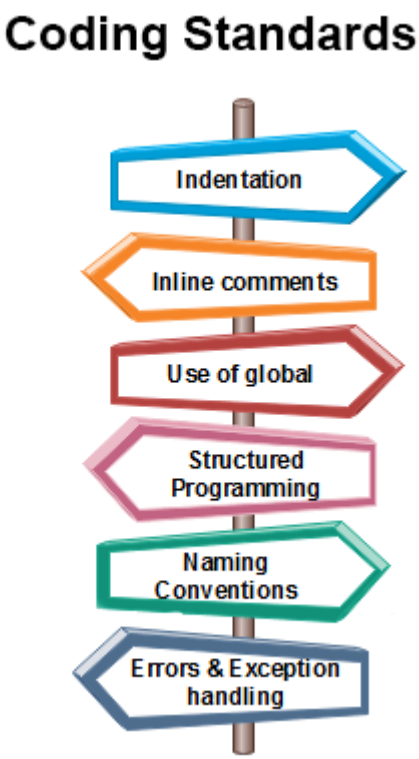
Widely available: Language should be widely available, and it should be feasible to provide translators for all the major machines and all the primary operating systems.

A coding standard lists several rules to be followed during coding, such as the way variables are to be named, the way the code is to be laid out, error return conventions, etc.

Coding Standards

General coding standards refers to how the developer writes code, so here we will discuss some essential standards regardless of the programming language being used.

The following are some representative coding standards:



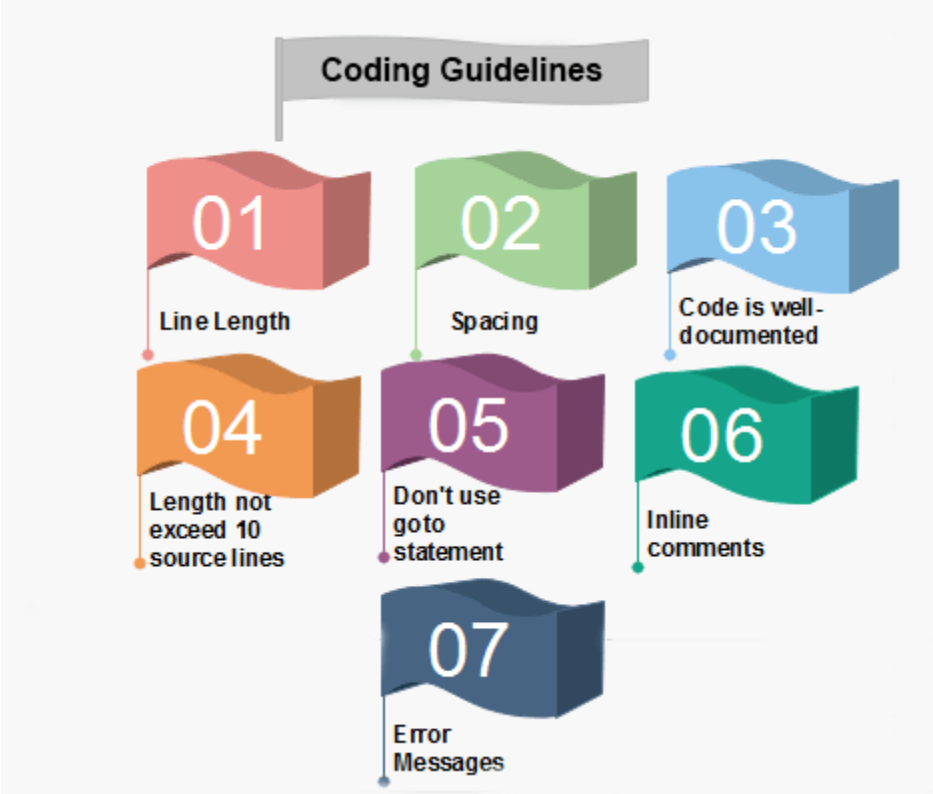
1. **Indentation:** Proper and consistent indentation is essential in producing easy to read and maintainable programs.
   Indentation should be used to:
   - Emphasize the body of a control structure such as a loop or a select statement.

    - Emphasize the body of a conditional statement
    - Emphasize a new scope block
2. **Inline comments:** Inline comments analyze the functioning of the subroutine, or key aspects of the algorithm shall be frequently used.
3. **Rules for limiting the use of global:** These rules file what types of data can be declared global and what cannot.
4. **Structured Programming:** Structured (or Modular) Programming methods shall be used. "GOTO" statements shall not be used as they lead to "spaghetti" code, which is hard to read and maintain, except as outlined line in the FORTRAN Standards and Guidelines.
5. **Naming conventions for global variables, local variables, and constant identifiers:** A possible naming convention can be that global variable names always begin with a capital letter, local variable names are made of small letters, and constant names are always capital letters.
6. **Error return conventions and exception handling system:** Different functions in a program report the way error conditions are handled should be standard within an organization. For example, different tasks while encountering an error condition should either return a 0 or 1 consistently.

Coding Guidelines

General coding guidelines provide the programmer with a set of the best methods which can be used to make programs more comfortable to read and maintain. Most of the examples use the C language syntax, but the guidelines can be tested to all languages.

The following are some representative coding guidelines recommended by many software development organizations.



1. Line Length: It is considered a good practice to keep the length of source code lines at or below 80 characters. Lines longer than this may not be visible properly on some terminals and tools. Some printers will truncate lines longer than 80 columns.

2. Spacing: The appropriate use of spaces within a line of code can improve readability.

Example:

Bad:
```
cost=price+(price*sales_tax)
fprintf(stdout ,"The total cost is %5.2f\n",cost);
```

Better:
```
cost = price + ( price * sales_tax )
fprintf (stdout,"The total cost is %5.2f\n",cost);
```

3. The code should be well-documented: As a rule of thumb, there must be at least one comment line on the average for every three-source line.

4. The length of any function should not exceed 10 source lines: A very lengthy function is generally very difficult to understand as it possibly carries out many various functions. For the same reason, lengthy functions are possible to have a disproportionately larger number of bugs.

5. Do not use goto statements: Use of goto statements makes a program unstructured and very tough to understand.

6. Inline Comments: Inline comments promote readability.

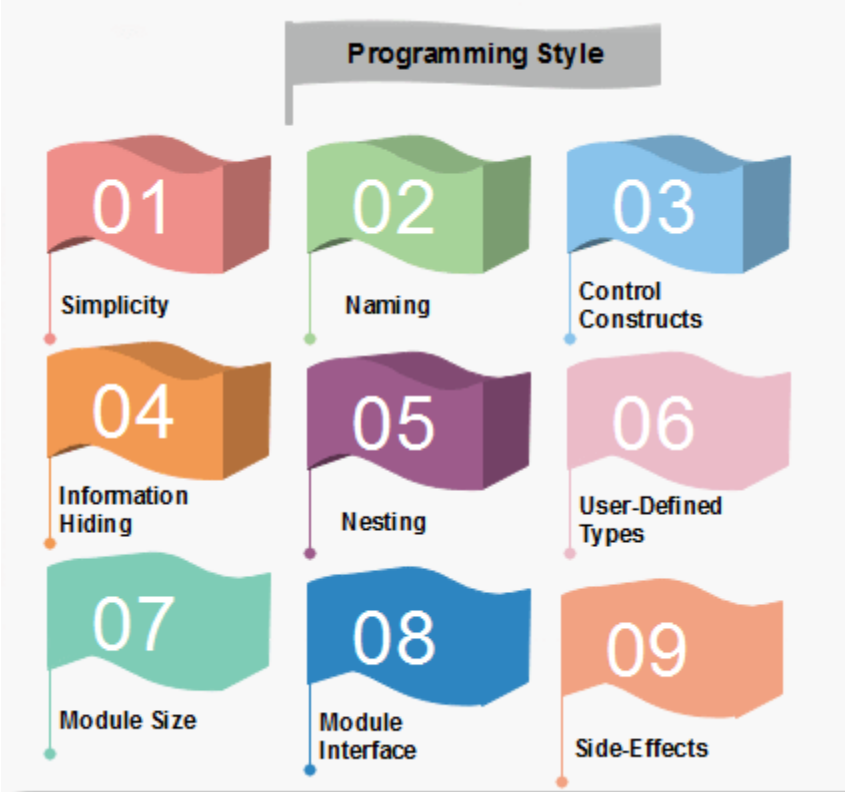
7. Error Messages: Error handling is an essential aspect of computer programming. This does not only include adding the necessary logic to test for and handle errors but also involves making error messages meaningful.

Programming Style

Programming style refers to the technique used in writing the source code for a computer program. Most programming styles are designed to help programmers quickly read and understands the program as well as avoid making errors. (Older programming styles also focused on conserving screen space.) A good coding style can overcome the many deficiencies of a first programming language, while poor style can defeat the intent of an excellent language.

The goal of good programming style is to provide understandable, straightforward, elegant code. The programming style used in a various program may be derived from the coding standards or code conventions of a company or other computing organization, as well as the preferences of the actual programmer.

Some general rules or guidelines in respect of programming style:



1. Clarity and simplicity of Expression: The programs should be designed in such a manner so that the objectives of the program is clear.

2. Naming: In a program, you are required to name the module, processes, and variable, and so on. Care should be taken that the naming style should not be cryptic and non-representative.

For Example: a = 3.14 * r * r
area of circle = 3.14 * radius * radius;

3. Control Constructs: It is desirable that as much as a possible single entry and single exit constructs used.

4. Information hiding: The information secure in the data structures should be hidden from the rest of the system where possible. Information hiding can decrease the coupling between modules and make the system more maintainable.

5. Nesting: Deep nesting of loops and conditions greatly harm the static and dynamic behavior of a program. It also becomes difficult to understand the program logic, so it is desirable to avoid deep nesting.

6. User-defined types: Make heavy use of user-defined data types like enum, class, structure, and union. These data types make your program code easy to write and easy to understand.

7. Module size: The module size should be uniform. The size of the module should not be too big or too small. If the module size is too large, it is not generally functionally cohesive. If the module size is too small, it leads to unnecessary overheads.

8. Module Interface: A module with a complex interface should be carefully examined.

9. Side-effects: When a module is invoked, it sometimes has a side effect of modifying the program state. Such side-effect should be avoided where as possible.

Structured Programming

In structured programming, we sub-divide the whole program into small modules so that the program becomes easy to understand. The purpose of structured programming is to linearize control flow through a computer program so that the execution sequence follows the sequence in which the code is written. The dynamic structure of the program than resemble the static structure of the program. This enhances the readability, testability, and modifiability of the program. This linear flow of control can be managed by restricting the set of allowed applications construct to a single entry, single exit formats.

Why we use Structured Programming?

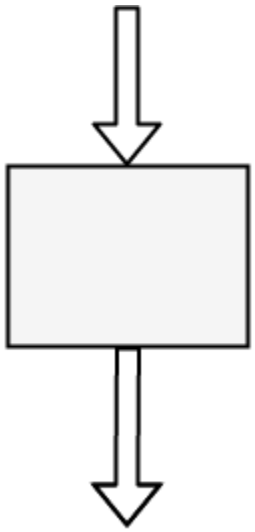
We use structured programming because it allows the programmer to understand the program easily. If a program consists of thousands of instructions and an error occurs then it is complicated to find that error in the whole program, but in structured programming, we can easily detect the error and then go to that location and correct it. This saves a lot of time.

These are the following rules in structured programming:

Structured Rule One: Code Block

If the entry conditions are correct, but the exit conditions are wrong, the error must be in the block. This is not true if the execution is allowed to jump into a block. The error might be anywhere in the program. Debugging under these circumstances is much harder.

Rule 1 of Structured Programming: A code block is structured, as shown in the figure. In flow-charting condition, a box with a single entry point and single exit point are structured. Structured programming is a method of making it evident that the program is correct.

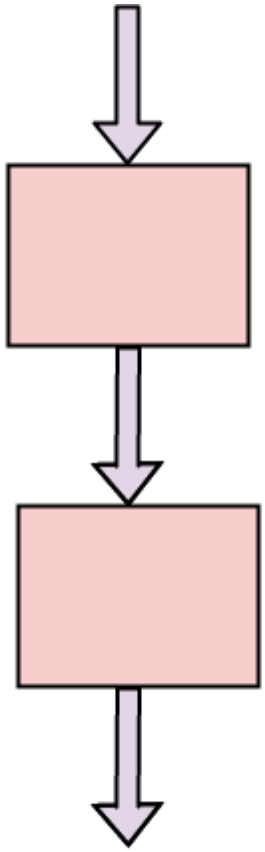


Rule1: Code block is structured

Structure Rule Two: Sequence

A sequence of blocks is correct if the exit conditions of each block match the entry conditions of the following block. Execution enters each block at the block's entry point and leaves through the block's exit point. The whole series can be regarded as a single block, with an entry point and an exit point.

Rule 2 of Structured Programming: Two or more code blocks in the sequence are structured, as shown in the figure.



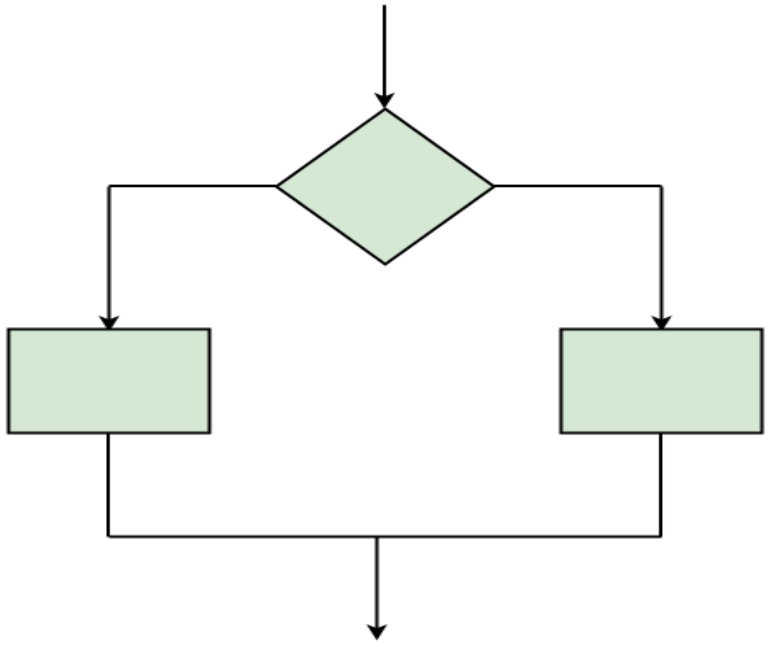
Rule2: A sequence of code blocks is structured

Structured Rule Three: Alternation

If-then-else is frequently called alternation (because there are alternative options). In structured programming, each choice is a code block. If alternation is organized as in the flowchart at right, then there is one entry point (at the top) and one exit point (at the bottom). The structure should be coded so that if the entry conditions are fulfilled, then the exit conditions are satisfied (just like a code block).

Rule 3 of Structured Programming: The alternation of two code blocks is structured, as shown in the figure.

An example of an entry condition for an alternation method is: register $8 includes a signed integer. The exit condition may be: register $8 includes the absolute value of the signed number. The branch structure is used to fulfill the exit condition.

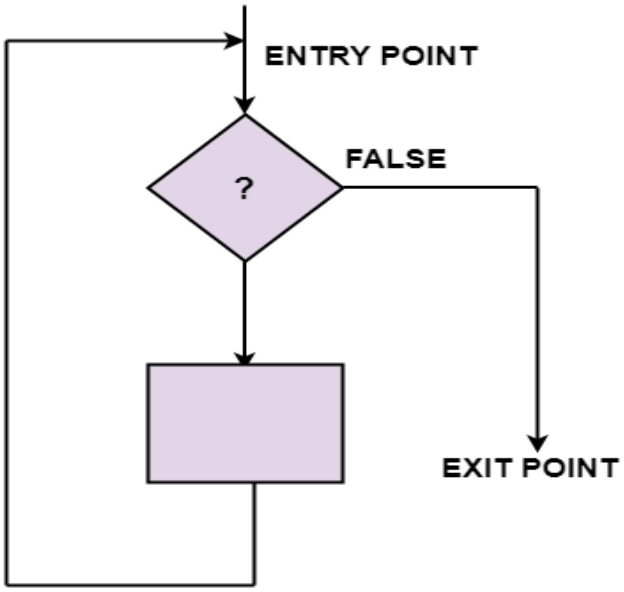


Rule 3: An alternation of code blocks is structured

Structured Rule 4: Iteration

Iteration (while-loop) is organized as at right. It also has one entry point and one exit point. The entry point has conditions that must be satisfied, and the exit point has requirements that will be fulfilled. There are no jumps into the form from external points of the code.

Rule 4 of Structured Programming: The iteration of a code block is structured, as shown in the figure.

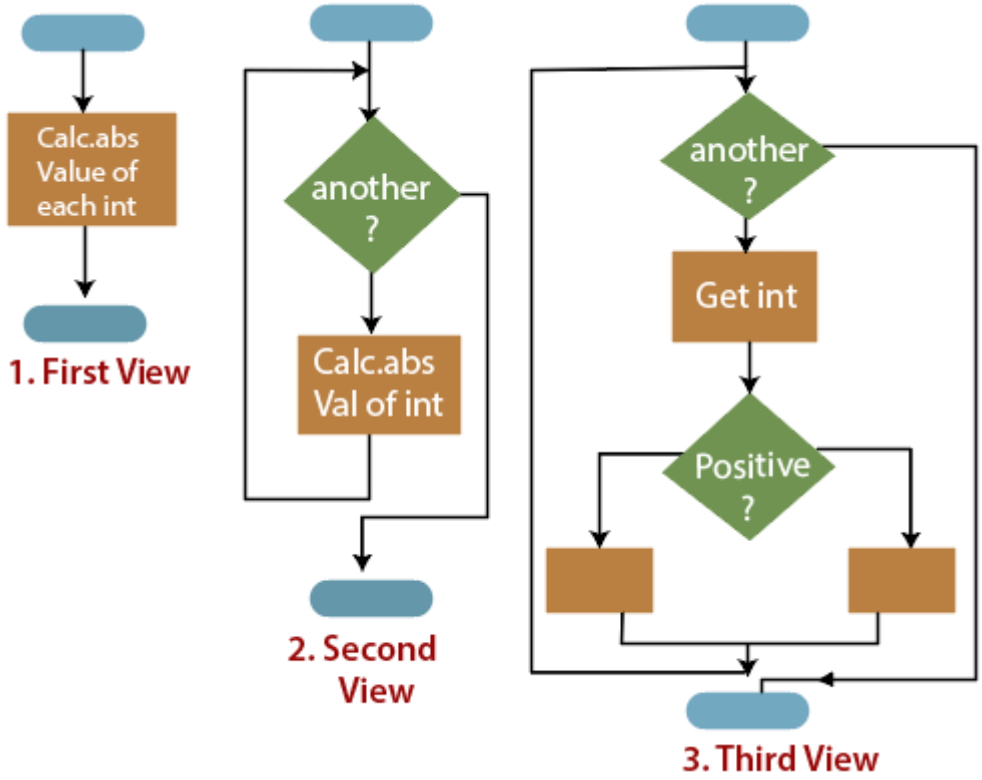


Rule 4: Iteration of code blocks is structured

Structured Rule 5: Nested Structures

In flowcharting conditions, any code block can be spread into any of the structures. If there is a portion of the flowchart that has a single entry point and a single exit point, it can be summarized as a single code block.

Rule 5 of Structured Programming: A structure (of any size) that has a single entry point and a single exit point is equivalent to a code block. For example, we are designing a program to go through a list of signed integers calculating the absolute value of each one. We may **(1)** first regard the program as one block, then **(2)** sketch in the iteration required, and finally **(3)** put in the details of the loop body, as shown in the figure.



The other control structures are the case, do-until, do-while, and for are not needed. However, they are sometimes convenient and are usually regarded as part of structured programming. In assembly language, they add little convenience.