CSC 104 GENERAL NOTES

Elements of Programming

Process to develop various sets of instruction is known as programming. To develop any instruction there are some elements needed or we can essentially present in all language. So any programming language is made up of 5 basic elements of the instructions as follows:

- **Variables:** variables in programming tell how the data is represented which can range from very simple value to complex one. The value they contain can be change depending on condition. As we know program consist of instructions that tell the computer to do things and data the program use when it is running. Data is constant with the fixed values or variable. They can hold a very simple value like an age of the person to something very complex like a student track record of his performance of whole year.
- **Loops:** we can define loop as a sequence of instructions that are repeated continuously till a certain condition is satisfied. How a loop start understand this first a certain process is done, to get any data and changing it after that applied condition on the loop is checked whether counter reached to prescribed number or not. Basically a loop carry out execution of a group of instruction of commands a certain number of times. There is also a concept of infinite loop which is also termed as endless loop is a piece of code that lack from functional exit and goes to repeat indefinitely.
- **Conditionals:** conditionals specify the execution of the statements depending on the weather condition is satisfied or not. Basically it refers to an action that only fulfilled when the applied condition on instructions satisfied. They are one of the most important components of the programming language because they give freedom to program to act differently every time when it executes that depends on input to the instructions.
- **Input/output:** the element of computer programming allows interaction of the program with the external entities. Example of input/output element are printing something out to the terminal screen, capturing some text that user input on the keyboard and can be include reading and writing files. Let's take a language example to understand the concept of input and output. C++ use streams to perform input and output operation in sequential media in terms of screen, the keyboard or a file. We can define stream as an entity that can insert or extract characters and there is no need to know details about the media associated to the stream or any of its internal specification. We need to know about streams is that they are source or destination of characters and the characters are accepted sequentially.
- **Subroutines and functions:** the element of the programming allows a programmer to use snippet of code into one location which can be used over and over again. The primary purpose of the functions is to take arguments in numbers of values and do some calculation on them after that return a single result. Functions are required where you need to do complicated calculations and the result of that may or may not be used subsequently used in an expression. If we talk about subroutines that return several results. Where calls to subroutines cannot be placed in an expression whether it is in the main program where subroutine is activated by using CALL statement which include the list of inputs and outputs that enclosed in the open and closed parenthesis and they are called the arguments of the subroutines. There are some of the rules follow by both to define name like less than six letters and start with the letters. The name should be different that used for variables and functions.

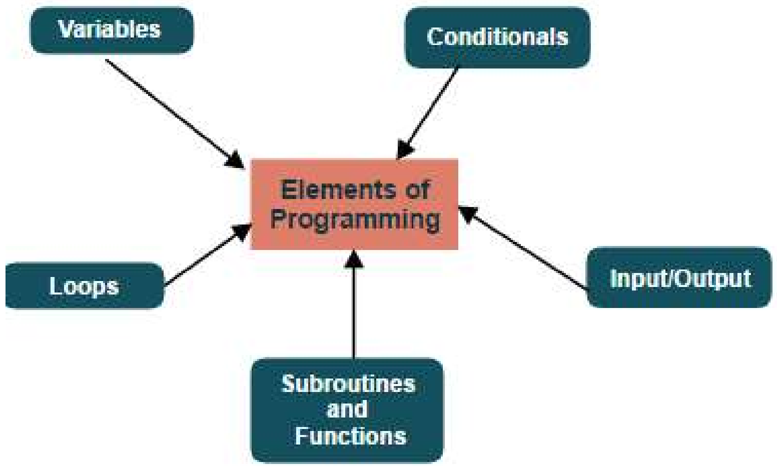
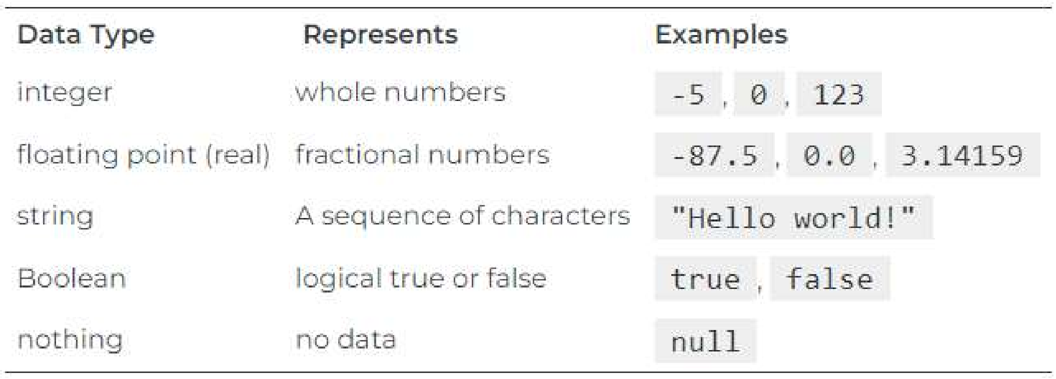


Fig. 1: Five Basic Elements of Programming

Data Types

A data type, in programming, is a classification that specifies which type of value a variable has and what type of mathematical, relational or logical operations can be applied to it without causing an error. A string, for example, is a data type that is used to classify text and an integer is a data type used to classify whole numbers. A major part of understanding how to design and code programs is centered in understanding the types of data that we want to manipulate and how to manipulate that data. Most programming languages support various types of data, including integer, real, character or string, and Boolean.

Data Type	Used for	Example
String	Alphanumeric characters	hello world, Alice, Bob123
Integer	Whole numbers	7, 12, 999
Float (floating point)	Number with a decimal point	3.15, 9.06, 00.13
Character	Encoding text numerically	97 (in ASCII, 97 is a lower case 'a')
Boolean	Representing logical values	TRUE, FALSE



Data Type	Represents	Examples
integer	whole numbers	`-5`, `0`, `123`
floating point (real)	fractional numbers	`-87.5`, `0.0`, `3.14159`
string	A sequence of characters	`"Hello world!"`
Boolean	logical true or false	`true`, `false`
nothing	no data	`null`

The data type defines which operations can safely be performed to create, transform and use the variable in another computation. When a program language requires a variable to only be used in ways that respect its data type, that language is said to be *strongly typed*. This prevents errors, because while it is logical to ask the computer to multiply a float by an integer (1.5 x 5), it is illogical to ask the computer to multiply a float by a string (1.5 x Alice). When a programming language allows a variable of one data type to be used as if it were a value of another data type, the language is said to be *weakly typed*.

Technically, the concept of a *strongly typed* or *weakly typed* programming language is a fallacy. In every programming language, all values of a variable have a static type -- but the type might be one whose values are classified into one or more classes. And while some classes specify how the data type's value will be compiled or interpreted, there are other classes whose values are not marked with their class until run-time. The extent to which a programming language discourages or prevents type error is known as *type safety*.

Program Design

Program design consists of the steps a programmer should take before they start coding a program. These steps when properly documented will make the completed program easier for other programmers to maintain in the future. There are three broad areas of activity:

- Understanding the Program
- Using Design Tools to Create a Model
- Developing Test Data

Understanding the Program

If you are working on a project as one of many programmers, the system analyst may have created a variety of documentation items that will help you understand what the program does. These could include screen layouts, narrative descriptions, documentation showing the processing steps, etc. If you are not on a project and are only creating a simple program, you will likely have only a brief description of the program's purpose. Understanding a program's purpose usually involves understanding its:

- Inputs
- Processing
- Outputs

This IPO approach works well for beginner programmers. It might help to visualize the program running on a computer: You can imagine what the monitor will look like, what the user must enter with the keyboard, and what processing or changes will be made.

Algorithm

An algorithm is a series of specific and finite instructions that produce a result (output), Algorithms are everywhere. For example, in a recipe, directions of a GPS, how to tie a tie, etc. Flowcharts and pseudocode are very useful tools to organize and design algorithms. However, in order to develop a useful algorithm, it is necessary to:

1. Understand the problem
2. Define an input
3. Process the input data
4. Expect output
5. Test and analyze data

Algorithms are the basis of any computer program. Before writing a single line of code it is necessary to design an algorithm that solves the problem. Therefore, a good programmer must be a good problem solver and be knowledgeable of their own inputs.

Using Design Tools to Create a Model

At first, you will not need a hierarchy chart because your first programs will not be complex. But as they grow and become more complex, you will divide your programs into several modules (or functions).

The first modeling tool you usually learn is pseudocode. You will document the logic or algorithm of each function in your program. At first, you will have only one function, and thus your pseudo code will follow closely the IPO approach above.

There are several methods or tools for planning the logic of a program. They include: flowcharting, hierarchy or structure charts, pseudocode, HIPO, Nassi-Schneiderman charts, Warnier-Orr diagrams, etc. Programmers are expected to understand and create flowcharts and pseudocode. These methods of developing a program's model are usually taught in computer courses. Several standards exist for flowcharting and pseudocode and most are very similar. However, most companies have their own documentation standards and styles. Programmers are expected to quickly adapt to any flowcharting or pseudocode standards for the company at which they work. The other methods that are less universal require some training which is generally provided by the employer.

Later in your programming career, you will learn about using application software that helps create an information system and/or programs. This type of software is called Computer-Aided Software Engineering (CASE).

Understanding the logic and planning the algorithm on paper, before you start to code, is a very important concept. Many students develop poor habits and skipping this step is one of them.

Develop Test Data

Test data consists of the programmer providing some input values and predicting the outputs. This can be quite easy for a simple program and the test data can be used to check the model to see if it produces the correct results.

Computer Programming - Arrays

Consider a situation where we need to store five integer numbers. If we use programming's simple variable and data type concepts, then we need five variables of **int** data type and the program will be as follows:

```
#include <stdio.h>

int main() {
   int number1;
   int number2;
   int number3;
   int number4;
   int number5;

   number1 = 10;
   number2 = 20;
   number3 = 30;
   number4 = 40;
   number5 = 50;

   printf( "number1: %d\n", number1);
   printf( "number2: %d\n", number2);
   printf( "number3: %d\n", number3);
   printf( "number4: %d\n", number4);
   printf( "number5: %d\n", number5);
}
```

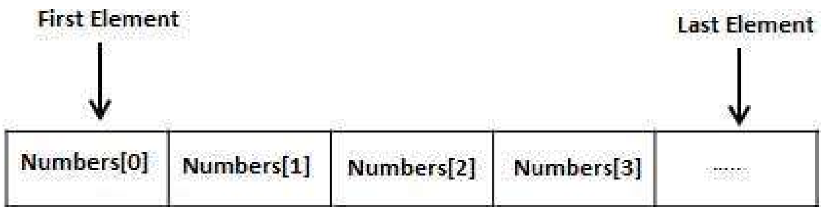
It was simple, because we had to store just five integer numbers. Now let's assume we have to store 5000 integer numbers. Are we going to use 5000 variables?

To handle such situations, almost all the programming languages provide a concept called **array**. An array is a data structure, which can store a fixed-size collection of elements of the same data type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

An array is a collection of items stored at contiguous memory locations. The idea is to store multiple items of the same type together. This makes it easier to calculate the position of each element by simply adding an offset to a base value, i.e., the memory location of the first element of the array (generally denoted by the name of the array). The base value is index 0 and the difference between the two indexes is the offset.

Instead of declaring individual variables, such as number1, number2, ..., number99, you just declare one array variable **number** of integer type and use number1[0], number1[1], and ..., number1[99] to represent individual variables. Here, 0, 1, 2, ...99 are **index** associated with **var** variable and they are being used to represent individual elements available in the array.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



Create Arrays

To create an array variable in C, a programmer specifies the type of the elements and the number of elements to be stored in that array. Given below is a simple syntax to create an array in C programming −

type arrayName [arraySize];

This is called a *single-dimensional* array. The **arraySize** must be an integer constant greater than zero and **type** can be any valid C data type. For example, now to declare a 10-element array called **number** of type **int**, use this statement −

int number[10];

Here, *number* is a variable array, which is sufficient to hold up to 10 integer numbers.

Initializing Arrays

You can initialize an array in C either one by one or using a single statement as follows:

int number[5] = {10, 20, 30, 40, 50};

The number of values between braces { } cannot be larger than the number of elements that we declare for the array between square brackets [ ].

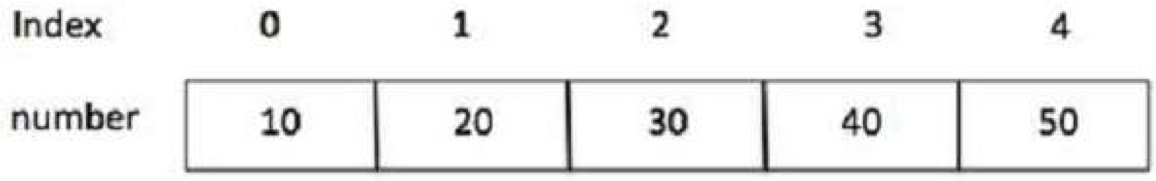
If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write −

int number[] = {10, 20, 30, 40, 50};

You will create exactly the same array as you did in the previous example. Following is an example to assign a single element of the array −

number[4] = 50;

The above statement assigns element number 5th in the array with a value of 50. All arrays have 0 as the index of their first element which is also called the base index and the last index of an array will be the total size of the array minus 1. The following image shows the pictorial representation of the array we discussed above −



Accessing Array Elements

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example −

```
int var = number[9];
```

The above statement will take the 10th element from the array and assign the value to **var** variable. The following example uses all the above-mentioned three concepts viz. creation, assignment, and accessing arrays −

Array’s size

In C language, array has a fixed size meaning once the size is given to it, it cannot be changed i.e. you can’t shrink it neither can you expand it. The reason was that for expanding, if we change the size we can’t be sure (it’s not possible every time) that we get the next memory location to us as free. The shrinking will not work because the array, when declared, gets memory statically allocated, and thus compiler is the only one can destroy it.

Types of indexing in an array:

- 0 (zero-based indexing): The first element of the array is indexed by a subscript of 0.
- 1 (one-based indexing): The first element of the array is indexed by the subscript of 1.
- n (n-based indexing): The base index of an array can be freely chosen. Usually, programming languages allowing n-based indexing also allow negative index values, and other scalar data types like enumerations, or characters may be used as an array index.

Advantages of using arrays:

- Arrays allow random access to elements. This makes accessing elements by position faster.
- Arrays have better cache locality that makes a pretty big difference in performance.
- Arrays represent multiple data items of the same type using a single name.

Disadvantages of using arrays:

You can’t change the size i.e. once you have declared the array you can’t change its size because of static memory allocation. Here Insertion(s) and deletion(s) are difficult as the elements are stored in consecutive memory locations and the shifting operation is costly too.
Now if take an example of implementation of data structure Stack using array there are some obvious flaw.
Let’s take the **POP** operation of the stack. The algorithm would go something like this.

1. Check for the stack underflow
2. Decrement the top by 1

So there what we are doing is that, the pointer to the topmost element is decremented means we are just bounding our view actually that element stays there taking up of the memory space. If you have any primitive data type then it might be ok but the object of an array would take a lot of memory.

Functions and Procedures

Functions and procedures are the basic building blocks of programs. They are small sections of code that are used to perform a particular task, and they are used for two main reasons.

The first reason is that they can be used to avoid repetition of commands within the program. If you have operations that are performed in various different parts of the program, then it makes sense to remove the repeated code and create a separate function or procedure that can be called from those places instead. Not only will this reduce the size of your program, but it will make the code easier to maintain, and it will remove the possibility that some of the code segments are updated, but not others.

This modular approach also facilitates *data abstraction* by centralising data-storage functions.

The second reason for careful use of functions and procedures is to help define a logical structure for your program by breaking it down into a number of smaller modules with obvious purposes. Depending on the programming language you use, you can also compile a library of functions and procedures and import them for use in other programs. The fundamental role of a procedure is to offer a single point of reference for some small goal or task that the developer or programmer can trigger by invoking the procedure itself.

Are Functions and Procedures the Same Thing?

Functions and procedures are very similar - in fact, in some programming languages there are only functions, and procedures are seen as a special case of a function, just as a square is a special type of rectangle. Generally speaking, functions return a value, whereas procedures don't (so a procedure is just a function that doesn't return a value). To confuse matters, though, there are certain subroutines, such as *msgbox()* in VisualBASIC that appear to be both/either (if you specify only the message, msgbox() behaves as a procedure, but if you use the extra options, such as message type, then msgbox() returns a value).

By *returns a value*, we mean that the function creates some sort of results, which is passed back to the calling function. A subroutine called *squared()*, for example, that calculates the square of a number, would be a function, because the result of the calculation would need to be passed back.

Variables, Functions and Variables

Variables declared within functions or procedures are said to be *local* - that is, they can only be used within that function, or other functions called by that function. This is called the *scope* of the variable.

Local variables are destroyed when the function or procedure finishes executing, and their values are lost. Next time you call the function/procedure; the variable is recreated and reset (usually to zero). If you want your local variables to retain their values when the function or procedure is not being executed, you need to declare it to be *static*.

Structured Programming

Structured Programming Approach, as the word suggests, can be defined as a programming approach in which the program is made as a single structure. It means that the code will execute the instruction by instruction one after the other. It doesn't support the possibility of jumping from one instruction to some other with the help of any statement like GOTO, etc. Therefore, the instructions in this approach will be executed in a serial and structured manner. The languages that support Structured programming approach are:

- C
- C++
- Java
- C#

..etc

On the contrary, in the Assembly languages like Microprocessor 8085, etc, the statements do not get executed in a structured manner. It allows jump statements like GOTO. So the program flow might be random.

The structured program mainly consists of three types of elements:

- Selection Statements
- Sequence Statements
- Iteration Statements

The structured program consists of well structured and separated modules. But the entry and exit in a structured program is a single-time event. It means that the program uses single-entry and single-exit elements. Therefore a structured program is well maintained, neat and clean program. This is the reason why the Structured Programming Approach is well accepted in the programming world.

One of the most important concepts of programming is the ability to control a program so that different lines of code are executed or that some lines of code are executed many times. The mechanisms that allow us to control the flow of execution are called **control structures**. Flowcharting is a method of documenting (charting) the flow (or paths) that a program would execute. There are three main categories of control structures:

- **Sequence** – Very boring. Simply do one instruction then the next and the next. Just do them in a given sequence or in the order listed. Most lines of code are this.

- **Selection** – This is where you select or choose between two or more flows. The choice is decided by asking some sort of question. The answer determines the path (or which lines of code) will be executed.

- **Iteration** – Also known as repetition, it allows some code (one to many lines) to be executed (or repeated) several times. The code might not be executed at all (repeat it zero times), executed a fixed number of times or executed indefinitely until some condition has been met. Also known as looping because the flowcharting shows the flow looping back to repeat the task.

A fourth category describes unstructured code.

- **Branching** – An uncontrolled structure that allows the flow of execution to jump to a different part of the program. This category is rarely used in modular structured programming.

All high-level programming languages have control structures. All languages have the first three categories of control structures (sequence, selection, and iteration). Most have the if then else structure (which belongs to the selection category) and the while structure (which belongs to the iteration category). After these two basic structures, there are usually language variations.

The concept of **structured programming** started in the late 1960's with an article by Edsger Dijkstra. He proposed a "go to less" method of planning programming logic that eliminated the need for the branching category of control structures. The topic was debated for about 20 years. But ultimately – "By the end of the 20th century nearly all computer scientists were convinced that it is useful to learn and apply the concepts of structured programming."

Advantages of Structured Programming Approach:

1. Easier to read and understand
2. User Friendly
3. Easier to Maintain
4. Mainly problem based instead of being machine based
5. Development is easier as it requires less effort and time
6. Easier to Debug
7. Machine-Independent, mostly.

Disadvantages of Structured Programming Approach:

1. Since it is Machine-Independent, So it takes time to convert into machine code.
2. The converted machine code is not the same as for assembly language.
3. The program depends upon changeable factors like data-types. Therefore it needs to be updated with the need on the go.
4. Usually the development in this approach takes longer time as it is language-dependent. Whereas in the case of assembly language, the development takes lesser time as it is fixed for the machine.

Structured Programming vs Object-Oriented Programming

Structured Programming	**Object-Oriented Programming**
It is a subset of procedural programming.	It relies on concept of objects that contain data and code.
Programs are divided into small programs or functions.	Programs are divided into objects or entities.

It is all about facilitating creation of programs with readable code and reusable components.	It is all about creating objects that usually contain both functions and data.
Its main aim is to improve and increase quality, clarity, and development time of computer program.	Its main aim is to improve and increase both quality and productivity of system analysis and design.
It simply focuses on functions and processes that usually work on data.	It simply focuses on representing both structure and behavior of information system into tiny or small modules that generally combines data and process both.
It is a method of organizing, managing and coding programs that can give or provide much easier modification and understanding.	It is a method in which set of objects can vary dynamically and can execute just by acting and reading to each other.
In this, methods are written globally and code lines are processed one by one i.e., Run sequentially.	In this, method works dynamically, make calls as per need of code for certain time.
It generally follows “Top-Down Approach”.	It generally follows “Bottom-Up Approach”.
It provides less flexibility and abstraction as compared to object-oriented programming.	It provides more flexibility and abstraction as compared to structured programming.
It is more difficult to modify structured program and reuse code as compared to object-oriented programs.	It is less difficult to modify object-oriented programs and reuse code as compared to structured programs.
It gives more importance of code.	It gives more importance to data.

Elementary structures of structured programs

- **Block:** It is a command or a set of commands that the program executes linearly. The sequence has a single point of entry (first line) and exit (last line).
- **Selection:** It is the branching of the flow of control based on the outcome of a condition. Two sequences are specified: the ‘if’ block when the condition is true and the ‘else’ block when it is false. The ‘else’ block is optional and can be a no-op.
- **Iteration:** It is the repetition of a block as long as it meets a specific condition. The evaluation of the condition happens at the start or the end of the block. When the condition results in false, the loop terminates and moves on to the next block.
- **Nesting:** The above building blocks can be nested because conditions and iterations, when encapsulated, have singular entry-exit points and behave just like any other block.
- **Subroutines:** Since entire programs now have singular entry-exit points, encapsulating them into subroutines allows us to invoke blocks by one identifier.

Module

A module is a software component or part of a program that contains one or more routines. One or more independently developed modules make up a program. An enterprise-level software application may contain several different modules, and each module serves unique and separate business operations.

Modules make a programmer's job easy by allowing the programmer to focus on only one area of the functionality of the software application. Modules are typically incorporated into the program (software) through interfaces.