

Fundamentals of Programming

Introduction

Before attempting to learn any specific programming language, some fundamental concepts common to all languages should be understood. The specific examples in this document may be taken from Perl, Visual Basic, or KiXtart, but will be typical of most programming languages.

Before undertaking any programming project, it is a good idea to think about the process as a whole. Try to break the process into major parts, think about what information you will need to process, how it will get into the program, and what will be done to it as an end-result. Programmers often add notes to their initial document using words or phrases that resemble the programming language. This is known as “pseudo-code” and will help them identify the logic process.

Another useful tool is “flow-charting”. This is a diagram of the entire process, using simple descriptions of major functions and decision points. Additional flowcharts can be used to evaluate and describe the major functions in more detail.

Learning this document & diagram process early is important, as it makes it easier to tackle more involved projects later.

Besides the overall approach that you establish to your project, there are a number of concepts that are common to all languages. It is important to have a firm understanding of them before learning a specific language.

Practice

Write a procedure and draw a flowchart that describes the process for building your favorite sandwich. Use the example Peanut Butter & Jelly process for reference. Some pseudo-code has been used.

1. Gather ingredients (bread, peanut butter, jelly, plate, knife)
Define Slice[2], PB, Jelly, Plate, Knife, NoCrust, Divide, Sandwich variable names
2. Place bread on plate, heel to heel
Mount(Slice[1],Left,Plate); Mount(Slice[2],Right,Plate)
3. Spread slice #1 with Peanut Butter
fnSpread(Slice[1],PB) # covers slice evenly with ingredient
4. Spread slice #2 with Jelly
fnSpread(Slice[2],Jelly) # covers slice evenly with ingredient
5. Flip Slice #1 onto Slice #2
Sandwich = fnCombine(Slice[1],Slice[2])
6. If defined, trim crust and discard
If NoCrust <> 0 Trim(Sandwich)
7. If defined, cut assembled sandwich in half at 45-degree angle
If Divide <> 0 Cut(Sandwich, 50%, 45)
8. Enjoy!
Satisfaction = Eat(Sandwich)

Variables

Every programming language uses some form of variables. These are symbolic, meaningful names that refer to some piece of information within the program. Generally, variable names can be any combination of letters or numbers, although certain languages may impose restrictions on their names, such as:

- Must begin with a letter
- Must be less than 14 characters in length
- Must not include special characters (such as :, *, #, %)

These restrictions, while specific to certain languages, make sense in general and should probably be followed as “rules of common sense”.

Types of Variables

Variables can hold different kinds of information – text, numbers, and Boolean (true/false). The language keeps track of the type of information stored in a variable so it knows how to manipulate the information. For example, if the variables “X” and “Y” contained the text “Foot” and “ball”, the statement

$$\text{Result} = X + Y$$

would store “Football” in the new variable called Result. The “+” would be interpreted as “concatenation” instead of “addition”. If the variables instead contained numeric values, the variable Result would contain the sum of those two values.

Text variables are referred to as “strings”, because they contain strings of characters.

Some languages have a generic variable (called a “variant”) that can hold anything. These might need to be “cast” (forced to a particular type) when they are referenced in order to be properly evaluated.

Numeric variables can be further divided into the type of number they contain. Integer and Real (floating decimal) are two common types. Integers are used for counters and indexes, and can often represent large numbers in a small amount of storage. *Real* variables provide a high degree of accuracy, but consume more storage and result in higher overhead.

Another class of variable is an *array*. This can be thought of as a matrix of variables, usually all of the same type. An example of an array might be the “OSInfo” array, in which one element identifies the OS Type, another identifies the version, and so on, until all unique values of a computer’s operating system are represented in a single named element.

Assignment

Variables are “assigned” values by placing them on the left side of an equal sign (“=”). The right side is the source, and can be a fixed number or string, the value returned from a function, or even another variable. A common (but strange looking) assignment occurs when the same variable appears on both sides of the assignment. The right side is always evaluated first, so “X = X + 1” adds 1 to the current value of “X”, then stores the new result in “X”, effectively incrementing the value of “X” by 1. The assignment of a value

may define the variable type. For example, 'X = 3' assigns the *value* of three, while 'X = "3"' (note that the number is in quotes) assigns the *character string* "3" to the variable.

Variables and Scope

Scope refers to the "visibility" of a variable. As programs get more complex, they tend to be made of several individual functions. The main part of the program can invoke the functions to perform certain tasks. Functions are often used when the same task must be performed at several points within the program.

Variables can be defined in such a way that they can be referenced in the main program and in all functions. These are known as *global* variables. They can be defined in the main program, evaluated and modified in any function, and the new value will be available to the main program and in all other functions. These are useful, but can also be dangerous, since careless use of global variables can make for difficult troubleshooting.

The opposite of a global variable is one defined with *local* scope. When defined in this manner, the variable "X" in the main program could contain "6", while "X" defined in a function might contain "yes". Neither can view (or affect) the other because their scope is local to their definition.

Often, variables must be *declared* before use. In some languages, you define the name and type of data it will hold, as in "INT X, Y". This defines a local variable that will hold integer numbers. Other languages may use the generic *variant* class until the first value is actually assigned to the variable.

Conversion

Sometimes a variable will not contain the data in the format you need. As described earlier, the definition 'X = "3"' results in a *string* variable type. If we then performed 'X = X + 4', the result would be "34" and not 7, because the variable was treated as a string, forcing the "+" to be interpreted as a string concatenation instead of mathematical addition.

Several functions exist that force the conversion of a value to a specific type. While each is language specific, the most common syntax is illustrated here:

- Cint(var) Converts *var* to integer number
- Cdbl(var) Converts *var* to double-precision real number
- Cstr(var) Converts *var* to string

You will need to consult your specific language reference manual to learn which conversion functions exist. What is important to understand now is that variables are "typed" by the language, and conversion may be needed before you can perform operations on the variable!

Identifiers

Some languages allow any word (not already reserved by the language itself) to be used as a variable. Others identify variables by preceding the variable name with an identifier character, such as a "\$". Some use a balancing act, using the plain name for assignment and the identified name when referencing the contents of the variable. Consult your language reference manual for specific rules.

Input & Output

Programs would not have much value if they did not accept input and deliver output. All programs have both input and output. (*consider the DOS dir command – what is it’s “input”?*)

Most of us think of input as something the user types in response to program queries. While this is very common, other forms of input can be files, other programs, and the file system itself. Output is usually the console screen, but can be a file, printer, or other attached device. Console input and output are so common, however, that an entire programming concept known as *stdio* (**STanDard Input & Output**) is dedicated to it. The console keyboard is sometimes referred to as the *stdin* device, while the console screen is known as *stdout*. These references can be *redirected* from the command line (or in other scripts) to get input from and save output to a file by using the redirection operators “<” and “>”. These are not programming objects themselves, but knowing that they exist (and understanding how they work) allow you to take advantage of their capabilities when writing your own programs. Another important *stdio* concept is the “standard error” device, called *stderr*. Error messages are written to the console screen through the *stderr* device so that normal program output may be redirected to an output file or another program, but error messages will still appear on the screen.

Getting User Input

All languages have the ability to read the *stdin* device. Remember that unless the operator has redirected the input to come from a file or other program, the input must be typed at the console.

Languages have a myriad of commands for reading input, and while the command syntax may be different, the basic concepts remain the same.

- **Numbers** All input commands will return a “number” if the input is restricted to digits, commas, and a period. Most input commands will work this way.
- **Text Strings** Typical commands are *input*, *gets*, *lineinput*, *read*. All continue to read input until a *line terminator* character is typed. This is usually the *enter* or *return* key on a keyboard.
- **Single Characters** Commands might include *getc*, *get*, *inkey* or *readc*. These commands are useful when asking simple Y/N questions or accepting menu choices. The command returns the first character typed on the keyboard.

Note that many computer systems *buffer* the input, allowing the user to type information even when a program is not requesting input. The input commands actually read from the buffer and not the keyboard itself. Some languages provide commands to flush (clear) the buffer, forcing the user to enter a response only when the program is ready.

Displaying Output

Most output functions will send their data to the *stdout* device, usually the console screen. Some languages that operate in a GUI environment have enhanced capabilities to generate output in pop-up windows (dialog boxes) in addition to the text console.

Again, the actual commands are language specific, but might be *print*, *write*, or *output*. Sometimes, commands exist to help format the output, such as a *cls* command to clear the screen, *at* or *pos* commands to place the cursor on the screen, or even commands to control the size or color of the text.

- Messages In nearly all languages, simple text messages must be enclosed in quotes. *print "this is my first output"* is an example of how quotes delimit the message that is to be displayed.
- Variables Sending the contents of a variable to your output destination can sometimes be confusing, especially when the language does not require an Identifier character to precede the variable name. A simple rule that works in all languages is to enclose any text in quotes, and use string concatenation to output the variable data. For example:
print "The value of X is " + X
- Dialog Boxes This capability will depend on the language, and whether the language is expected to operate in a GUI environment. Perl, for example, does not have this capability, but does have special extensions that support it. Most other modern languages have some form of dialog-box support built-in.

A typical dialog box will have a *title*, *text message*, and – optionally – one or more *response buttons*. The response buttons are often Yes/No, OK/Cancel, or similar combinations. A value is usually returned from this function indicating which button (if any) was selected.

Conditionals

Conditionals are commands that can make decisions and change how the program operates. All conditional statements perform a test, and depending on whether the result of the test is TRUE or FALSE, an appropriate action will occur.

The test part of the conditional usually compares two values. Most common comparisons are equality (=), inequality (<>), less-than (<), or greater-than (>). Some languages offer additional comparisons, and a few use slightly different syntax (like “= =” for equality, or “!=” for inequality). No matter what the syntax is, the comparison results in a true or false evaluation, and the corresponding actions are performed.

If – Else

The most common conditional is the *if* statement. The syntax is similar in nearly all languages. It starts with the *If*, followed by some type of test. Some languages require that the test end with the word *then*, although this requirement is fading from use. Following the *If/Test* line is the *True Execution Block*. This is a block of one or more lines of program code that will be executed if the test is true.

If a different set of commands should be executed if the test is false, then an *Else* statement will be used, and the lines that follow become the *False Execution Block*.

Some languages require that you define the beginning and end of these execution blocks, often using braces (“{” and “}”). Other languages require that the end of the *If* be *terminated* with some sort of *End If* statement. These languages assume that anything following the test is the *True Execution Block*, which is terminated by either the *Else* or *End If* statement. If *Else* is found, the *False Execution Block* is defined between the *Else* and *End If* commands.

Immediate If

Because the *if* statement is used so often to define a variable with one or another value, the Immediate *If* (usually *iff*) has found its way into many languages. It can assign a value to a variable based on the result of a test. It usually follows the form

X = iff(Var=10,50,0)

Thus, if *var* is equal to 10, *X* will be set to 50, otherwise *X* will be set to 10. This eliminates the true and false execution blocks and all of the extra typing associated with them. It also makes reading the code somewhat simpler.

Compound Conditionals with AND or OR

Sometimes, a simple test will just not be enough. Logical operators *and* and *or* permit more complex tests to be defined. For example, the statement

If X = 10 or X = 20 And Y = 5

will be considered TRUE if *X* were equal to 10. It would also be TRUE if *X* were equal to 20 *and* *Y* were equal to 5. Note that when *X* = 10, the first part of the test is TRUE, the remaining part following the OR is not evaluated, as the statement is already TRUE!

Check your program language manual for additional logical operators that may be supported.

Loops

Where conditionals can control which part of a program gets executed, *loops* control *how many times* a block of code will be executed. There are different types of loops available in most languages, to accommodate different situations. Most modern languages support at least a variant of the loop formats discussed here.

Iterative Loops – For & For Each

An *iterative loop* performs a loop a finite number of times. These are often used to process a well-defined number of elements, such as each of the arguments passed to a program or function, or the number of elements in an array.

The most common iterative loop is the *for / next* loop. The usual syntax is something like
For X = 0 to 100 step 10
statements...
Next

In this example, the loop counter “X” is initialized to zero. The statements in the loop are executed, and are permitted to use (and often modify!) the value of X. When the *next* statement is reached, the value of X is compared to the upper limit, defined here to be 100. If it is not equal to or greater than the upper limit, the *step value* is added to X and the loop is repeated. The loop exits when the loop counter equals or exceeds the upper limit. In this example, the value of X counts by tens, from 0 to 100.

A conditional may be placed within the loop statements to alter the loop counter. Often, a loop should be executed a particular number of times, but if an exception occurs, the loop can be exited by setting the counter equal to the upper limit. The loop will terminate when the last statement is executed.

A common alternative to the *for / next* loop is the *for each / next* loop. Many newer languages support this functionality, which allows each element in an array to be evaluated without knowing how many elements exist. Assume that a function returns an array containing the total of each sale made. There is no way to pre-determine how many sales will be made, so a fixed *for/next* loop isn’t appropriate. The code below will evaluate each item in the SALES array, assigning the next element value to the Sale variable:

For Each Sale in SALES
TOTAL = TOTAL + Sale
Next

This example builds a running total of the sales by adding the Sale value to the TOTAL variable each time the loop is executed. The loop terminates automatically when the last array element is referenced.

Conditional Loops – Do and While

Conditional loops process while (or until) a particular condition is TRUE. A *do* loop performs its test at the end of the code block. This causes the statements in the loop to always be executed at least once. A *while* loop performs its test at the top of the loop. If the test is immediately false, the statements in the loop will be bypassed, having never been run. Both forms have value, but in greatly differing situations.

The syntax of conditional loops varies widely between languages – consult your manual for exact syntax.

Branching & Functions

Branching allows control of the program to be diverted to another block of code. This diversion can be permanent, as in the case of a *goto*, or temporary, as implemented by *gosub* or *functions*.

- Goto

Immediately transfers control to a new location within the program. The new location is defined by a label, or – in older languages – a line number. The use of *goto* is discouraged, as it makes the program logic difficult to follow. It is still useful, when diagnosing problems, as it can quickly allow processing to temporarily bypass a section of code that may be in error.

- GoSub

Immediately transfers control to a defined subroutine. This is a block of code that performs specific processing, then returns control to the statement that follows the *gosub*. A subroutine generally starts with a label, contains one or more program statements, then terminates with a *return* statement. Subroutines are often used when a group of statements would otherwise be repeated at several locations within the program. A subroutine is part of the main program, and has the same variable scope. This allows variables to be referenced and manipulated directly. Unfortunately, variables defined and used within the subroutine are also visible to the main program, and can affect the operation of the program if careful consideration is not given to the variable names used in subroutines!

- Functions

Functions are similar to subroutines in concept, but are totally separate from the main program. Variables are passed to and returned from functions, which provides isolation between the function and the main program. Only variables explicitly declared as Global are shared between the main program and the function.

Functions also have an additional advantage – they can be stored in an external file and used in many programs, eliminating the need to re-write the statements for commonly used tasks!

Most languages support functions, and most allow any number of values to be passed to the function. The number of values that can be returned by a function varies, but is usually limited to one. Consider returning an array if you need to return more values. A function can usually return an *exit code* in addition to a value. The exit code indicates that the function was successful, or that it encountered an error. Generally, an exit code of zero indicates success, while any other value would indicate some specific error had occurred.

Advanced Concepts

As programming languages evolved, special commands were added to simplify the process of manipulating data. Depending on the programming language you have chosen, there may be commands for reading system files (registry, logs, and configuration files) or communicating with other programs (Windows COM).

The statements illustrated here are fairly common to modern languages in one form or another.

String Manipulation

- Left Left(sVar, 5)
Returns the 5 left-most characters from sVar
- Right Right(sVar,5)
Returns the 5 right-most characters from sVar
- SubStr SubStr(sVar,5,2)
Returns 2 characters from sVar, starting at the 5th position
- InStr InStr(sVar, "happy")
Returns the character position in sVar where the string "happy" was found.
Returns zero or an error if it wasn't found.
- Trim / LTrim / RTrim Trim(sVar)
trim – returns sVar with leading and trailing spaces removed
ltrim – returns sVar with leading spaces removed
rtrim – returns sVar with trailing spaces removed

Multiple Conditional

- Case Statement
Sometimes, you want to perform different actions depending on the value of a variable. You could do this with many If statements, but a *case* statement is much easier to use, and better to troubleshoot. There are many formats to the case statement, but the general logic is the same in all languages. In general, it looks like this:

BeginCase

Case X > 10

statements executed when X is more than 10...

Case X > 1

statements executed when X is more than 1 (but 10 or less!)...

Case X = 1

statements executed when X is exactly 1...

EndCase

The first case evaluates the value of X, and performs the statements if it is greater than 10. The next statement block is executed if X > 1, and the third block

executes only if X equals 1. Most languages require the beginning and end of the case block to be delimited in some fashion as illustrated with the *BeginCase* and *EndCase* statements. Also, once a case statement evaluates to true, all remaining tests are skipped. If this was not done, the example above would perform the “>10” and “>1” statements when X was 10 or greater!

Special Definition of Variables

- Chr() X = Chr(9)
Assigns a tab character to the variable X. This function converts a numeric value to its corresponding ASCII character. In ASCII, the character code “9” represents a tab. Any ASCII code can be assigned or output using this method.
- Asc() X = Asc(“A”)
Assigns the value “65” to the variable X. Converts a text character to its representative ASCII code.

Manipulation of Data Files

- Open
The *open* command initiates a connection with a data file of some kind. Generally, you specify the file name and location (the *filespec*), a numeric identifier that you will refer to this connection by (the *filehandle*), and the method of accessing the file (the *access mode*), which can generally be Read or Write, although most languages support a special write mode called Append, and some support Random, which allows simultaneous reading and writing of specially formatted files. The command returns a value indicating success or failure, which can be evaluated prior to continuing file operations.
- Read / Write
Commands to read from or write to the file that was *opened*.
Var = read(filehandle) Reads a line from the defined file handle
write(filehandle, Var) Writes the specified variable to the open file handle
Note that many languages do not automatically append a line-terminator character sequence when writing data files. If you want a line-delimited file, you must make sure the data you write includes the LineFeed and Carriage Return character sequences (or those appropriate for your file system).
- Close
Terminates the connection with the file, flushing all input or output buffers. This insures that any data written is actually on the disk.
close(filehandle) Closes the file associated with *filehandle*

Special Variables & “Built-Ins”

Many programs have built-in “constants” that are defined each time the program runs. The term *constant* may not actually be accurate, as the value of these “constants” may change based on external events. For example, one “constant” might be *time*. The program considers it a constant because it is not allowed to modify its value, even though the value changes every second! Other examples of such “constants” might be *date*, *hostname*, *process*, or *version*.

Some languages do not utilize constants at all, and others have dozens! Refer to your programming language manual to see what is provided.

KiXtart-Specific Statements & Functions

Using Variables

KiXtart defaults to a *variant* type of variable.

Variables are defined and referenced using a “\$” to identify variable names. Variable names are permitted in text strings, although this is generally discouraged.

Variables can be declared as local via a DIM statement, or global using the GLOBAL statement. Multiple variables can be declared on a single line. Variables not explicitly declared default to GLOBAL.

```
Dim $X, $Y, $Z
Dim $aThings[10]      ; an array of 10 things (0-9)
Global $InFile
```

Variable names must begin with a “\$”, and should be limited to letters and numbers. No distinction is made between upper and lower case letters.

Variables can be strings, integers, or double-precision floating-point numbers. Strings can contain up to 32,000 characters.

Expressions

- + Numeric Sum or String Concatenation
- Numeric subtraction
- * Multiplication
- / Division
- mod Divides two numbers and returns the remainder
- & Performs a bitwise AND of two numbers
- | Performs a bitwise OR of two numbers

- < Less-than
- > Greater-than
- = Equal
- <> Not equal
- <= Less-than or Equal
- >= Greater-than or Equal
- == Case sensitive string comparison
- Not Logical NOT operator - Negates the test or value
- And Logical AND operator
- Or Logical OR operator

Input & Output

There are two commands that perform user input, and one command for output. Additionally, there is a function to display a dialog box and obtain a response based on the button that the user selects.

Gets	Gets \$Input	Accepts a string and stores it in the specified variable.
Get	Get \$Char	Accepts one character, stores it in the specified variable
CLS	CLS	Clears the screen
?	? "message"	"?" causes output to move to a new line. Any text in quotes is output to the console. Variables inside a quoted string, or references without an assignment are displayed.

MessageBox A function to display a message box. Displays a title, message text, and various configurations of buttons.

Here is an example of the commands shown above:

```
Cls
"Welcome!" ?
"Please enter your name: "
Gets $Name
? "Hello "
$Name
?
; "36" is the messagebox code for Yes/No and a "?" icon
$M = MessageBox("Title", "Message", 36)
```

Conditionals

This is the general syntax of the *if* and *iif* statements. The Else and FalseStatementBlock parts of the If statement are optional.

```
If test
  TrueStatementBlock. . .
Else
  FalseStatementBlock. . .
EndIf
```

```
$X = IIf(test, TrueValue, FalseValue)
```

Test should be a comparison, usually between a variable and a constant, or two variables.

```
$X = 5
If $X = 3
  "X is 3" ?
Else
  "X is NOT 3!" ?
EndIf
```

Iterative Loops

KiXtart supports For / Next and For Each / Next forms of iterative loops.

```
$X = 5
For $I = 1 to 10 Step 2
    "X=" $X " I=" $I ?
Next

$array = 1,4,6,2,3
For Each $Element in $Array
    "Element="
    $Element
    ?
Next
```

Conditional Loops

KixTart provides both Do / Until and While / Loop statements. The *do* loop will always have its statements executed at least once, as the conditional test occurs at the end of the loop. The *while* loop performs its test at the beginning, which can prevent the loop statements from executing at all!

```
Do
    $X = Rnd(9)           ; get a random # between 0 and 9
    'X='                 ; print it
    $X
    ?
Until $X = 5           ; try again?

; This example starts with a random number. If, by chance
; that number is zero on the first try, the loop will
; not be executed, and processing will continue after
; the LOOP statement.
$X = Rnd(9)           ; get a random # between 0 and 9
While $X <> 0
    'X='                 ; print it
    $X
    ?
    $X = Rnd(9)         ; get another random # between 0 and 9
Loop
```