# Pawn

**embedded scripting language**

# A Gentle Introduction to Programming

**February 2006**

*ITB CompuPhase*

"CompuPhase" is a registered trademark of ITB CompuPhase and "PAWN" is a trademark of ITB CompuPhase.

"Java" is a trademark of Sun Microsystems, Inc.

"Linux" is a registered trademark of Linus Torvalds.

"Microsoft" and "Microsoft Windows" are registered trademarks of Microsoft Corporation.

"Unicode" is a trademark of Unicode, Inc.

Copyright © 2005–2006, ITB CompuPhase; Eerste Industriestraat 19–21, 1401VL Bussum, The Netherlands (Pays Bas); telephone: (+31)-(0)35 6939 261
e-mail: info@compuphase.com, WWW: http://www.compuphase.com

The information in this manual and the associated software are provided "as is". There are no guarantees, explicit or implied, that the software and the manual are accurate.

Requests for corrections and additions to the manual and the software can be directed to ITB CompuPhase at the above address.

Typeset with TEX in the "Computer Modern" and "Palatino" typefaces at a base size of 11 points.

# A gentle introduction to programming

The world of programming computers or other devices may be intimidating when you first step into it. You are greeted with a new jargon, a vast set of technologies that are identified with unfamiliar mnemonical names, and a pile of details that all seem interrelated. The learning curve appears steep. You do not know where to start. On top of this, every specialist that you ask will give you different advice —sometimes strongly opinionated.

The three hurdles that a beginning programmer typically encounters are:
◇ mastering the analytical way of problem solving;
◇ learning a "programming language" needed to express the solution in;
◇ and getting fluent with the software tool set for constructing programs.

These three items are indeed interrelated: the tools that you have determine the way that you approach and fix a problem —if you hold a hammer, every problem is solved with a nail. It is difficult to go into one of these subjects and postpone the introduction of the others. Yet, we must start somewhere.

The approach that I have chosen here is to first cover the entire programming process very succinctly, and then to start over again adding a few details. The first part is quite incomplete and it probably raises more questions than it answers; at the same time, you may find that the second part repeats some of the first part. Fortunately, this introduction is quite thin.

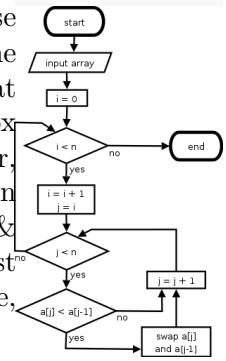
## A brief tour

Whatever the language or tool, programming is a craft that requires a particular approach to problem solving —or even a particular way of thinking about activities and information. It centres around *analysis*: "what needs to be done under which conditions and criterions", *reductionism*: subdividing a large task into a hierarchy of smaller tasks, and *synthesis*: bringing it all together.

Before starting to build a program, one must understand the problem at hand. Once you have a good idea of what the program must do, you have to think about a solution in small discrete and deterministic steps. Each step may have inputs and outputs and you can describe the function of the step in terms of the inputs and outputs alone —this is what the term *discrete* refers to. As soon as you cannot explain a step without referring to other "steps" in the program, you have either a hidden input or an implied assumption, and you would do well to reconsider the

analysis. The term *deterministic* is meant to say that the functioning of a step or a sequence of steps may not depend on (human) interpretation or judgement.

Many beginning programmers find it helpful to write down these steps in a flow chart (or some other schematic). During the analysis of the problem and its solution, you will often find that some steps are rather big —too big to be annotated in a little box in a flow chart. Such larger steps must be subdivided further, perhaps in another flow chart. A flow chart, such as the one on the right, shows start & termination points, processing, input & output functions, decisions and loops. Flow charts are the oldest schematics for software. Many others exist today; for example, see the many charts in the UML standard.




What most people perceive as "programming", the act of writing *code* in a programming language, only starts after the above analysis is done. Many programming languages exist, and what they all have in common is that they have a strict and rigid "grammar" (called *syntax*) and the ability to invent new "words" from a very small core vocabulary. The biggest difference between programming languages and *natural* languages, however, is that programming languages were devised to allow communication with a *machine*. A machine, or another programmable device, does *exactly* as instructed —no more, no less. A machine does not assume anything about its environment; it neither anticipates any future instructions, nor remembers what it did a fraction of a second ago. The "small, discrete, deterministic steps" that the preceding paragraphs mentioned must, hence, also be *precise* and *comprehensive*.

Flow charts, or other kinds of charts, can still be made with pencil on paper, but building a program that a computer can run requires special tools —tools with names like *compiler*, *linker/locator*, *editor* and *debugger*. Most programmers are also "power users" on their computers and the tools that they use are frequently less polished than your favorite office suite. On the other hand, the tools made for programmers often focus on letting you work efficiently, rather than wasting your time with animations, silly sounds and other attempts to look *cool*. Briefly: be not deceived by the spartan interface of programmer's tools.

Once we have gone through it all —when the analysis of the requirements and the decomposition into small steps lie behind us and after having keyed in the program code, we can (finally) run the program and watch its output. Often, the program does not behave as intended right away. It may not even run in its very

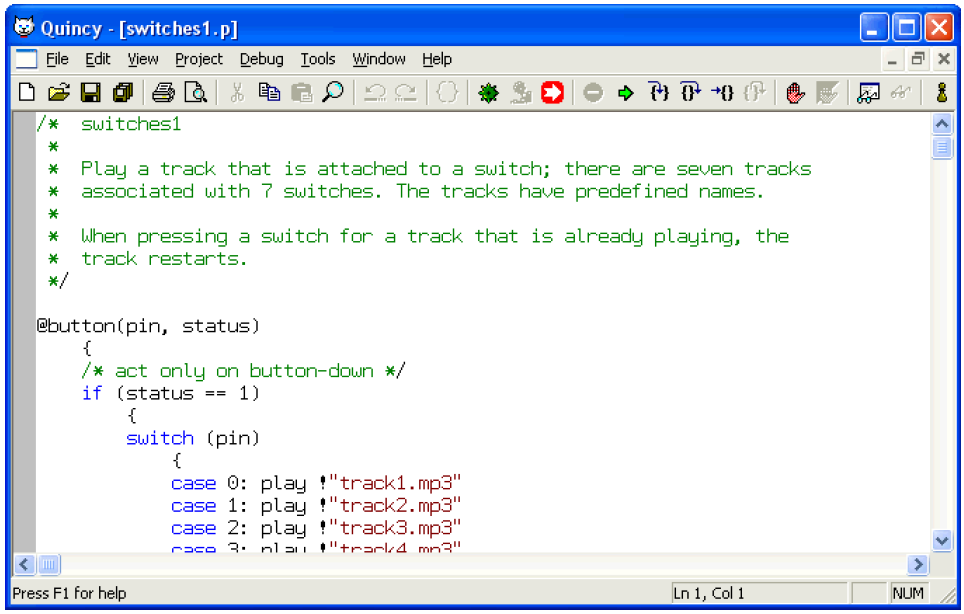



FIGURE 1: *A programmer's tool: the Quincy IDE for* PAWN

first version. This may be due to simple typing errors, which are easily fixed; but it may also be due to the programmer not properly understanding the problem. Naturally, we now have to find and fix the errors in the program, which is a cycle very similar to the one for the initial development:

- ◇ understand the program's behaviour and what the program *should* be doing instead,
- ◇ find a solution/fix in small discrete steps,
- ◇ write these steps in a programming language,
- ◇ and review and test the program.

Basically, there is not much difference between the processes of the initial development of a program and of the correction/maintenance of an existing program. I have more to say about finding and fixing errors further on, but here is a good place to affirm that testing and *debugging* (the process of correcting "bugs" in a program) is an integral part of the development cycle.

## A programming language . . .

The programming language is the most notable aspect of everything related to programming. Many courses that teach programming focus on one or more programming languages, and this introduction is no exception. For purpose of illustrating how a programming language looks, let us plunge ahead and walk through a simple program (in the PAWN language): one that adds two hard-coded numbers together.

Listing: **Adding two numbers**

```
main()
    {
    new a, b, total

    a = 24
    b = 32
    total = a + b

    printf "%d", total
    }
```

The first line says "`main()`". This is the definition of the entry point for the program. The program starts here, and it will run all the instructions that are enclosed between the curly brackets —these are typically called *braces*. From this point on, the computer will run through the program downwards, one instruction at a time. You will later learn that there are specific instructions in a programming language that changes the *flow* of control, but the basic flow is sequentially downwards.

The first instruction following the opening brace ("`{`"), is a declaration of three *variables*. A variable may be viewed as a container: most variables contain a single value, some contain a collection of values or other information. The program can put something in a variable, and later retrieve it back from it. The contents of a variable do not change by themselves; the only way a variable can receive new information is because a program stores something different into the variable. Noteworthy is that a variable takes space in the memory (RAM) of the computer and that it therefore must be *created*.

The next three statements in the program are "assignments". An assignment stores a value (or something else) in a variable. For the mathematically inclined, the use of the "=" symbol may be confusing at first sight, because this symbol *does not* denote equality. Rather, it says that what is on the right hand of the = must be stored to the variable that is on the left hand. The right hand may have a complete arithmetic expression, such as in the instruction that adds "`a`"

FIGURE 2: *A marble running through a maze, as a metaphor for the computer running through a program. You might want to object that such a maze is very linear and very simple, but ①with some ingenuity you can build complex behaviour in a maze —see for example the marble "flip-flop" at the right, and ② you are right on the mark: a maze is really simple, but in essence a computer is nothing more than a set of electronic switches. Only because there are so many of these switches, the computer can pretend to be an intelligent device. Computer programming is, hence, remarkably similar to building complex marble mazes.*

and "`b`" together and stores it into "`total`"; the left hand of the = symbol must always be a single variable.

There is one more statement before the closing brace: "`printf "%d", total`". On this cryptic line, the first word, "`printf`" indicates the activity: sending output to a terminal or to a (computer) display; the last word, "`total`", is the variable whose value we want to show, and in the middle there is a code that controls *how* the value is shown. The word "`printf`" is a system function, it is documented in the programmer's reference, along with all control codes.* The code "`%d`" means: show a value in decimal base and without a fractional part.

* The "f" in `printf` stands for "formatted". While printing text to a display or console window, the function is able to format values on the fly.

That completes the program. When the program "runs", it executes all statements between the braces, and terminates when reaching the closing brace. Run it again, and it will go through exactly the same steps, without ever tiring and without ever realizing that this is already the second (or third, fourth,...) time that it runs.

The variables in this program have the names "`a`", "`b`" and "`total`". You may choose the names of your variables, but there are naming rules that you must adhere to. For instance, you may only use letters, digits and the "_" character in a variable name, and the first character may not be a digit. Moreover, although upper case letters and lower case letters are both valid, they will indicate *different* variables. For example, you can have two distinct variables in your program that are called "`baba`" and "`Baba`" respectively. Another way of putting this is to say that the PAWN programming language is *case sensitive*. Not all programming languages are case sensitive.

The purpose of an assignment is to save a value or some piece of "data" for later use. The data can be anything: account numbers, names, running totals, or whatever elements you need the program to remember of a while. A program has no other memory than what is in its variables. When a program runs step-by-step through the statements, it does not remember anything about a preceding statement(s). If the result of the current statement is important for another statement further down in the program, that result must be stored in a variable. On the other hand, if your program does *not* use the contents of the variable at any later moment, the assignment is redundant: your program would have worked exactly the same as *without* the assignment —only a little faster.

It is perfectly normal to change the value of a variable while running through a program. For example, when you want to count the number of seconds that your program is "up and running", you would use a variable that starts at zero and that is incremented by one on every tick of the clock —such as in the expression: `runtime = runtime + 1`. In algebra, this would be a *falsehood*: a value cannot be equal to itself *plus one*; but in a programming context it simply means that the *new* value of the variable will become its *current* value plus one.

The order of the three assignment statements in the example program is important. What the program does (after the declaration of the variables), is to put a value into "`a`", then another one in "`b`", and finally a calculated result in "`total`" —where it uses the values previously stored in "`a`" and "`b`". If you were to step over to your neighbour and tell him that you spent your money on a hat costing €24.00 plus a book, and, oh yeah, the book was €32.00, your neighbour will not

have any problem to figure out our total spendings. But if you say in a program something like the lines below, you are bound to get the wrong answer:

```
a = 24
total = a + b
b = 32
```

The first assignment, "`a = 24`" is okay. When the computer sees the second assignment, it does not yet know what value "`b`" should have. The computer will, however, focus blindly on that single instruction: "`total = a + b`"; it will not look ahead, nor remember what happened in the recent past. And as a result, the computer will just *assume* some value for "`b`". It is likely that "`b`" is assumed to be zero in the above sequence, so the second assignment sets "`total`" to $24 + 0$, which is 24. The last assignment is okay as far as the computer is concerned, but it happens to be useless at this spot.

## . . . the tools to do it with . . .

Programmer's do not use Microsoft Office —not for programming, at least. Programmers write their code in an editor that saves the output in a plain text file (originally this was called an *ASCII* file). The little desktop applet "NotePad" that comes with Windows is a plain text editor, but no programmer will actually use NotePad on a day-to-day basis. NotePad is a very bare-bones editor. Many programming languages require that the source code (that you type) be converted to a compact binary form that is ready for execution. This process is called "compiling". The tool that does this is the *compiler*. When the compiler analyses the source code, it may already find problems with the syntax (remember: programming languages have a rigid syntax), and produce a list of warnings and errors. You then have to go through the list and correct any mistakes.* None of this is difficult, but for a beginning programmer, it is new stuff that has to be learned.

I have talked about the tools as different programs, and originally they are separate programs. However, the programs are mostly integrated in an *IDE*: an "Integrated Development Environment". You type your source code inside the IDE, which looks superficially like a word processor (e.g. Microsoft Word), you compile it by clicking on a toolbar button, and any errors or warning messages

* Tip: start from the top. Sometimes a simple error in the source may confuse the compiler and cause it to interpret your source code completely differently. Fix the first few errors and then re-compile.

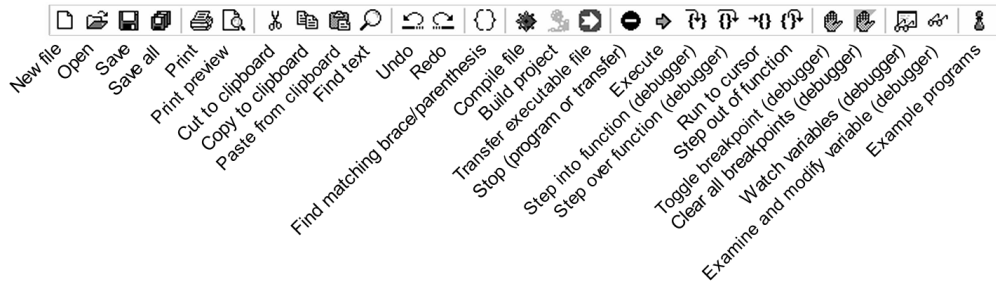



FIGURE 3: *The toolbar for the Quincy IDE*

get presented to you as a kind of search window: double-clicking on a line in that window moves the text cursor right to the offending line in the editor. When working within an IDE, you can largely remain unaware of the processing of compilers and the like, and focus on the program itself. The existence of IDEs is another important reason why programmers shy away from NotePad.

The IDE for PAWN is called "Quincy", and after you installed PAWN, you can launch it. Figure 1 shows a screen shot of the application and an annotated toolbar is in figure 2. After its first launch, the main window will probably be empty, so your first step is to create a new file using File / New or the toolbar button at the far left. For the file type, choose a "Pawn script". This gives you an edit window in which you can type everything you want —but the goal is, of course, that you type in a valid program. After you complete your program, you build it (Project / Compile; there is an equivalent button on the toolbar too). If the build succeeds, you run it (Project / Execute). If the build does not succeed, you correct the errors in the program and then attempt to build it again.

The above description is the basic work flow, but there are exceptions, extensions and customizations. When you build a program that must run *on a different apparatus* than the PC that you compile it on, you may have to *transfer* it, rather than running it (Debug / Transfer to remote host) Typically, the apparatus will then execute it automatically. If you want to *debug* the program, rather than simply run it, you have to turn on "debugging information" *before* you build the program. Other options that you may want to set are the paths ("folders" or directories) where the output file(s) are stored, and the level of reporting. See the dialog under Tools / Options... for details.

Many people learn by example, and therefore a noteworthy little button on the toolbar is the one at the far right: the "example programs". This opens a separate

window with a list of example programs with short descriptions. You can also obtain this list through the menu: Help / Example Programs....

## . . . and how it all relates

Whatever the programming language and the tools that a programmer uses, the essence of "programming" is the decomposing of a complete task into fully described logical steps. This requires a holistic "bird's eye" view and deep knowledge of the abundance of low-level details at the same time. Oversights in this process and misunderstandings about how the pieces fit together are the main causes of software errors, or "bugs". While it is sometimes claimed that if you switch to some (novel) programming language of methodology, bugs will be something of the past, these claims are as realistic as a hammer that is hard on nails and soft on thumbs.*

As was already stated earlier, finding and repairing bugs is an integral part of the craft of programming. While the idea that "programmers cannot test their code" is popular in some circles, testing programs and finding the root cause of bugs requires the same mental skills as creating the program in the first place. Programmers are therefore actually quite good at testing code, and in addition they have their programmer's tools to make it efficient. Every respectable programmer's kit comes with a tool that is called a *debugger* and whose purpose it is to give a detailed and dynamic insight to the program. The time that you invest in getting to know the debugger for your language, will pay off in a matter of weeks (or perhaps even days).

When creating a program, there is an almost natural tendency to work *bottom-up*. You start by picking some part of the product that you can build and test independently, and from there the project grows. The risk of this strategy is that any conceptual error may surface only at the stage of testing. To counter this effect, many programming *methodologies* are *top-down*: start with the global requirements and go progressively into more detail. The risk of this alternate strategy is that the analysis is based on assumptions and guesswork, because "hard data" can only come from measuring functional code. There is no silver

* Some researchers claim that software should be *proven to be correct* by mathematical reasoning before deployment, but mathematical "proofs" have a long tradition of containing flaws as well —the original, widely reviewed and publicized, but *flawed*, proof of Fermat's last theorem by Andrew Wiles in 1993 is illustrative in this respect.

bullet, but most designers agree that an iterative analyse-build-review cycle works best.

## In closing

This "gentle introduction" has been a whirlwind tour through the world of programming. There is much more to say, on all of the topics that this introduction has only touched upon. If you manage to type in and build the little program that I discussed above, that is a big step —yet, the program is not going to impress your friends. Every journey needs to start with a first step and the best practical advise that I can give you is make that first step. Make small steps at first and experiment with new techniques or syntaxes, but also verify everything you think you know.

PAWN's strength lies in the scripting of devices or environments with high demands on performance and restricted resources. This combination is present in (tiny) electronic devices as well as in high end computer games, which need every byte of RAM and every cycle of the processor for their graphic, network and game-play engines. These environments are extremely diverse. The example program in this introduction prints the result of a summation to the display... *assuming that there is a display.* If you are developing for a programmable audio device/MP3 player (for example), there might very well not be a display, and the program is quite useless. The diversity of platforms on which PAWN runs make it hard to write a general-purpose language tutorial —"impossible" might be the more appropriate word, in fact. Yet, the "Language Guide" contains a tutorial, and I recommend that you read it, along with any reference of the device or (game) environment/application that you wish to program.

Best of luck.