L.IN.OLEUM TUTORIAL

by Ponche

SECTION 0 - INDEX

SECTION 1 - OVERVIEW

1.1) Why learn L.in.oleum?

Ok. You have found this programming language and you don't know if it's a good idea to spend some of your time into learning it. I'm going to tell you why it's a good (very good indeed) idea.

First of all, the *speed* of the obtained applications is very high. As you'll see in section 1.2, L.in.oleum (refered to as Lino from now on) is *similar* to assembler. We can say that assembler is the second most direct way of telling the computer what you want to do. The truth is that Lino is not as fast as assembler, but its speed is almost identical.

You can ask yourself... if assembler is faster, why should i learn Lino instead? The answer is so simple: learning Lino is much *easier*. And that's not the only reason. Lino is also *universal*. What does this mean? It means that if you write a program today in a specific platform (like Windows), you can be sure that the same program will also work in any other platform at any other time (like Linux). Maybe you don't see the advantages of this point, but *portability* is one of the more important aspects of modern software. Also, from a general point of view, Lino lets you access many parts like the (input/output devices) of the machine in a fast, easy, and *transparent* way. Parts like the display, memory, audio devices, printers, keyboard, pointing devices (mouse), networks, and such. Not enough reasons? Then take notice that this software package is *100% free*.

1.2) Differences between Lino and assembler

As i said, Lino is very similar to assembly language, basically because both are low-level languages. This means that when using Lino or assembler, you communicate with the computer in a very direct way, contrary to high-level languages (like C, Pascal or Java). Here is a list of the main differences between assembler and Lino:

Features	Assembler	L.I.N.OLEUM
Speed	very fast	Just a little less
Portability	none - platform specific	Universal
I/O devices access	yes but difficult	Yes, and so easy
Learning difficulty	hard	read the next section

1.3) Is it *really* difficult to learn Lino programming?

I've been visiting the *anywherebb.com community* for a relatively long time. I've noticed that most people trying to learn Lino, just stop the process because they say it is too difficult. Most of these people have no/little programming experience. If your programming experience is low, then learning Lino is not exactly an easy task. But its also not impossible. I've written this tutorial for this group of people. If you have some experience with other programming languages (like high-level ones) then it's just a matter of *"changing the chip"*. This is because Lino is a low-level language. It's like trying to learn Object-Oriented Programming (OOP) when you have only experience with classic procedimental languages (like C or Pascal). On the other hand, if you know how to program in assembler, then you can be sure that learning Lino is as easy as pie. This tutorial will help you, no matter in which of this categories you consider yourself in. Anyway, i recommend that you to give Lino a try.

1.4) What can you achieve with Lino?

With this I also mean what can't you do, or better yet, what is it that Lino is designed for. Lino allows you to use the display, keyboard, mouse, printers, networks, file systems, memory, CPU registers (which is great), game devices (like joysticks), and some more advanced things. This is, you can access practically the whole machine. Notice that Lino is a *general-purpose language*, so it's designed to create all kind of applications. System programming, like the coding of device drivers and such platform-specific software is possible (since some features like machine code fragments were allowed in *v1.13.11b*) but this makes this specific software not portable, logically.

So you can write programs that mess with graphics, digital audio, 3d software rendering, multimedia data management, and all you can imagine. Of course this can be applied to any field like utilities, business, games, etc...

1.5) Contact the author

That's me! If you want to make any suggestions about this tutorial, or if you have any questions about Lino programming, you can contact me by writing an e-mail to *ponche@programmer.net*. Instead of this though, I recommend you check out the anywherebb Linoleum forums (*www.anywherebb.com*), where you can ask anything you want, and see alot of nice stuff, like the member's personal projects, or the anywherebb specific projects (whose creator is *Alex.tg*, also the manager of this great webpage), or games like *Noctis V*. You can also download the latest L.in.oleum version. There are a lot of things to do in anywherebb, and you must know that it's the **best** internet community in the whole world. Please check it out! *A final note*: i'm spanish, so we have non-English versions of this tutorial as well... Oh! And check my current project (EcoLife) progress, coded in Lino too! :P

1.6) GNU Free Documentation License

Version 1.2, November 2002

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- **A.** Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

- **B.** List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

- **C.** State on the Title page the name of the publisher of the Modified Version, as the publisher.

- **D.** Preserve all the copyright notices of the Document.

- **E.** Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

- **F.** Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

- **G.** Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

- **H.** Include an unaltered copy of this License.

- **I.** Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

- **J.** Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

- **K.** For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

- **L.** Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

- **M.** Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

- **N.** Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

- **O.** Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

SECTION 2 - BASICS

Usually, programming tutorials start by introducing a first program and explaining it from a global point of view. I think this is a good method and is what i'm going to do in this section. First, let's see what we are going to learn in this chapter. If you finish this lesson, you'll have knowledge about:

- how a Lino program looks and what the basic "*skeleton*" you will use in most applications looks like.

- what *CPU registers are* and how to give them a basic utility in the programs, using them to make basic calculations.

- how to *compile* the programs you write in order to create the final executable file that the user runs.

2.1) Our first program

First, you must know what a *program* is; basically it's a sequence of orders given to the computer. The computer takes the program and executes each of these orders *secuentially*. So you tell the computer what to do step by step and the computer does *exactly* that. Technically, these orders are called *instructions*. When you create a program, what you write is the *source code* of the program. Then you compile this code and you obtain the executable file that the computer understands. The computer can't execute the source code directly; you must compile it first. Now that you know what a program is, let's see what a Lino program looks like, by looking at an easy example (this is what we call the source code, mentioned in the parahraph above):

```
"directors"
      unit = 32;

"programme"
      (here comes the real code)
      a = 5;
      b = 3;
      a + b;
      show registers;
```

Ok, slowly... In the above source code you can see some things. The only thing you must know about the first two lines

```
"directors"
      unit = 32;
```

is that they must be included in all your programs (except when we are writing a library, but for now just write these two lines in every program). In later sections i'll explain its meaning, so don't worry. Now look at the next line

```
"programme"
```

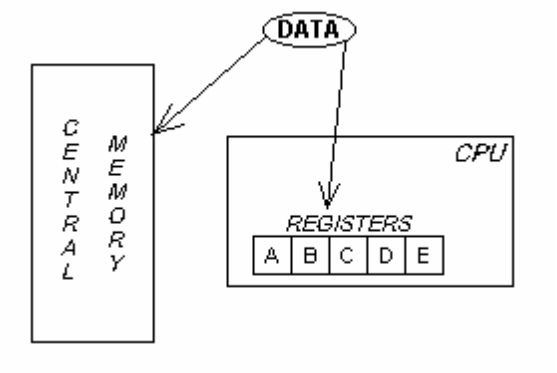
this marks the start of the "*real program*". After this comes the sequence of instructions. So when you write "programme" you are telling Lino that the rest of the lines are the instructions of the program. However, a program in Lino is not just the instructions. Though for now we are going to concentrate only on the "programme" part of the source code. After "programme" you see:

```
      (here comes the real code)
```

Look at the parenthesis surrounding this sentence. This kind of line is what we call *comments*. It's so simple: everything you write between parenthesis is ignored by the compiler. Why is it there then? Just for reference and explainitory purposes. Imagine that you are writing a program and a year later you want to modify it. Without comments you'll only see a lot of messy code and it won't be immediatly clear what that part of the source code does. If you write comments it makes it easier to understand and easy to make modifications if necissary. Also someone else might need to read/modify the code you wrote: If you make sensible but *not abussive* use of comments, you will make his, her, or your own life much easier.

2.2) Using the CPU registers

Right. Now i must explain what a *register* is. The main and most important part of a computer is the CPU (*Central Processing Unit*). Let's say that it is the *brain* of the machine. The CPU can do many things and one of them is *calculations*, like addition, substraction, and such. But you should know that in order to make one of these operations you also need some *data*. Well, you must store this data some place, and you also need some place to store the result of these operations. There are basically two places where you can store the data: one is the *central memory,* and we'll talk more about it in other section; the other is the CPU registers. Using Lino you have access to five registers, whose names are A, B, C, D, and E.



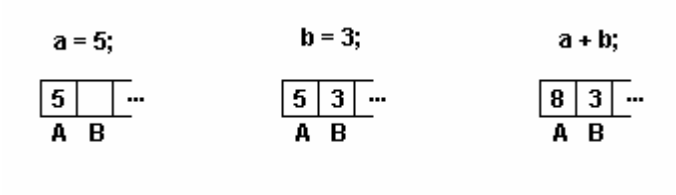
In each of these registers you can store a single data value. Generally you will store a numeric value (a number), but we'll see that they can be used in other tasks.

Now you can use your program to tell the CPU to store some data in the registers and to make some calculations with this data. This is exactly what our program does in the next three lines of code:

```
a = 5;
b = 3;
a + b;
```

The first instruction is an example of how to store some data in a register (register A). This is like saying "I want to store the number 5 in register A". The second one does something similar, but the register we use in this case is B and the value we store is 3. The last one is a little more difficult to understand, but easy still. With it, we tell the CPU to make an addition (look at the plus sign), between the values of register A and register B. These values are logically 5 and 3, because we have just stored them in the registers. So, the result of the operation is 8. Difficult to understand? I think not... Earlier, I said that we needed to also store the result of the operations. The question is: where is the result stored? The answer is so simple: on the register at the left side of the plus sign. In our case, this is register A. So, after this last instruction,

register A will contain value of 8, instead of value of 5. Look at this scheme if you need more help...



Maybe you have noticed that these instructions end with a *semicolon* (;). It is just there to indicate where a instruction finishes and where the next one starts. You *must* put a semicolon after each instruction. That's all.

There is just one more instruction in our source code, and it is

```
show registers;
```

This is a self-explainitory instruction. It does two things: opens a little window showing the values stored in the CPU registers, and then it finishes the execution of the program. So it is exactly what we need, isn't it? There are some other ways to finish the execution of a program, and we'll see how later.

2.3) Compiling and running our program

This is very nice, but... maybe you want to see it in "motion". As i explained before, after writing the source code, you must compile it because the computer doesn't understand it directly. Let's do that, then.

I assume that you have unzipped the "lino.zip" file. If you haven't "installed" it, then you must do it now. Go to the lino docs folder (usually, C:\LINOLEUM\DOCS) and there double-click on the "install.reg" icon. After this, open a text editor (like Notepad), and write our program. When you have finished writting it, save it, for example, in the lino folder (C:\LINOLEUM). Then go to this folder, and right-click on its icon. Select the option "Compile as LINOLEUM Source". A black window will appear, and some stuff can be readed on it. We only care of the last lines, where you can see if there were errors, warnings or compatibility warnings. If it indicates that there were some of this, it means you didn't write the source code correctly (if so, check it, and compile again). If there are not any errors/warnings, then you have succesfully compiled the program. Look at the folder where the source code was and check for a file called "PROGRAM.EXE". It is the final executable file, so just double-click on its icon, and the program will run.

2.4) Default program skeleton

The program we have written is just a very simple example. Before going into the next section, we should know the default *skeleton* of a Lino application. This skeleton is used in most programs. The names between quotes are what we call *periods.* There are *seven* different periods. You can deduce from this, that "directors" and "programme" (we have covered thus far), are also periods. Here comes a list of the seven periods of a Lino program, and an overall description of each one of them. In later sections we will only use those periods that are really needed for each case. For

now, the only thing you should memorize is its exact order. They must appear in this order in every program.

"libraries"

- In this period we will declare what libraries we are going to use in our program. A library is basically a file that lets you do some specific things.

"stockfile"

- In this period we will declare some *support* files (like graphics, sounds, and such) that will be used in our programs.

"directors"

- Here you'll write some sentences that allow you to configure some basic program parameters and input/output devices.

"constants"

- In the constants period you will declare the constants that your program will use (you will learn what a constant is in a later section).

"variables"

- As in the constants period, here you must declare the variables you will use in your program (read the next section to know what a variable is).

"workspace"

- Similar to the "variables" period but with some differences.

"programme"

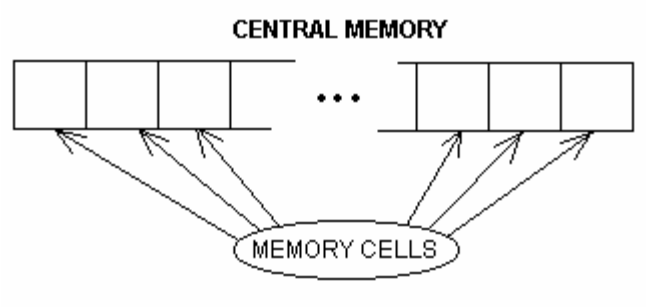
- You should know what this period is needed for... if not read this section again.

SECTION 3 - USING MEMORY VARIABLES

In the previous section we have seen how to store data in the registers, and how to do a simple calculation with them. You also know that there are five registers (A, B, C, D and E). However if you think about it, there are not very many of them. In real life, having only five places to store data is not viable.

3.1) Central memory structure

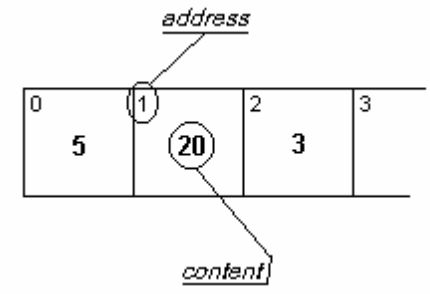
Evidently, we need more space. The solution to this problem is using the *central memory.* And what is it? Central memory (refered to as *memory* from now on) is a set of memory *cells*. A memory cell is, almost, like a single register. This means that you can store a single value on each of these memory cells. And there is alot of these cells. All these cells are placed one after another in the memory.



Right. So now we have alot more places where we can store data. When managing data with registers we accessed it through the register name in our program. Remember these instructions:

```
a = 5;
b = 3;
```

How do you supposed we are going to access all these memory cells? Imagine that there are millions of them! The solution is easy: assign an identifing number to each of them. Let's assign number 0 (zero) to the first memory cell of the memory, the number 1 to the next, and so. Ok, so at this point we have a lot of memory cells we can access by giving its identifing number and each one stores a single value. Technically, the identifing number of a cell is called the *address* of the cell, and the value stored in the cell is called the *content* of the cell.



There's no more to know about the structure of the memory. Remember, we have a set of *cells*, accessed by its *address*, and each one stores a single value (*content*).

3.2) Using memory: variables

Now we know what memory is and how it's organized, but... how can we use all this in a Lino program? Before being able to access the memory cells, we must tell Lino how many of them we will use in our program by declaring them in the "variables" period. And also, instead of accessing them throught their address, we are going to do it through its name. Imagine you want to use a memory cell in your program where you'll store the age of one person. What we are going to achieve is to access this cell throught a symbolic name, like "age of peter". Look at the following program:

```
"directors"
      unit = 32;

"variables"
      age of peter = 25; (this is the variable declaration)

"programme"
      a = [age of peter];
      a - 10;
      [age of peter] = a;
      end;
```

There are lot of new things in this program. The first one you will notice is that we have included the "variables" period along with the "directors" and "programme" periods. Look at the order of the periods. In the "variables" period we *declare* the memory cells we are going to use in the program, and from this point they turn into what we call *variables.* So a variable is a memory cell but with a name. In our case, the only one variable declared is *age of peter. A*lso notice that after the name of the variable there is a value assignment (= 25). What does this mean? Can you access the variable outside the "programme" period? Not exactly. That value is the *initial value* we assign to the variable, this is, to the content of the memory cell assigned to that variable. Read again and slower if you need to... So, that variable will have a value of 25 at the start of the program. Now into the "programme" period. There are four instructions in this program. The first one is

```
      a = [age of peter];
```

clearly an assignment (look at the equal sign). But what kind of assignment? I think it's clear that the destination is register A, right? And what will we store in A? The value 25. Take notice that in the "programme" period we have accessed the contents of *age of peter* by writting it between *square brackets,* like this [age of peter] memorize the following:

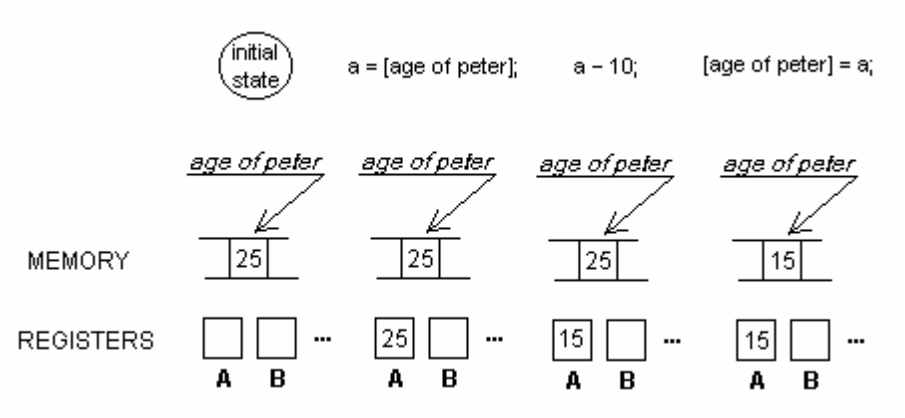
- when, in the "programme" period, you write the name of a variable between square brackets, you are refering to the *content* of the variable.

```
      a - 10;
```

This one is easier. It is a substraction. It just substracts 10 from register A.

```
      [age of peter] = a;
```

This is the opposite of the first one. We store in the variable *age of peter* the value contained in register A. So, what is the state of the memory and registers at this point? Look at the scheme and you'll find the answer



The only thing that remains uncommented is the very last instruction of the program

```
end;
```

this instruction is very similar to "show registers" but with the difference that it doesn't show the registers in a window. It just finish the execution of the program.

3.3) A more complex example

```
"directors"
      unit = 32;

"variables"
      first exam = 6;
      second exam = 8;
      third exam = 3;
      average = undefined;

"programme"
      [average] = [first exam];
      [average] + [second exam];
      [average] + [third exam];
      [average] / 3;
      end;
```

Let's see what this program does. As you can see we are declaring four variables this time. There are three exam results (*first exam, second exam* and *third exam*) and an extra variable called *average* where we will store the average of the three results. The exam results are initialized as usual, but... what is the initial value in *average*? Well, you can use *undefined* if you don't want to assign an initial value to a variable. This is because *average* will only contain a valid value after we have done the average calculation. So its initial value is just that, undefined.

Now look at the program code. The first instruction is an assignment. It assigns the contents of the *first exam* variable to the *average* variable. The second and third instructions do something similar. They add the other exam results to the *average* variable. The other instruction just divides (/) the contents of *average* by 3. So, in retrospect, what we have done is:

$$\text{average} = \frac{\text{first exam} + \text{second exam} + \text{third exam}}{3}$$

3.4) When should you use registers?

We know that both CPU registers and memory cells can be used to store single data values for our operations. Their behaviour and application is almost the same, except that you access a register throught its name (A, B, C, etc) and memory cells must be accessed throught its address (or its symbolic name if its a variable). Ask yourself this question: If i have only five registers, and a-lot-of memory cells, why should I use the registers, then? The answer is: because registers are much faster than memory when using them to make calculations. *Much faster.* For small applications this is not very important, but when you'll code time-critical applications you should use registers whenever you can. Registers speed up things a lot, believe me. But the fact of having only five of them, implies that you must make intelligent use of them too. That's all. Don't worry, programming is not difficult. The problem is doing it well. I'm going to tell you a secret... the key to becoming a good programmer is this: *experience*. Now, let's review our Lino knowledge:

- A Lino program is divided into various sections called *periods.* These periods must be placed in a strict order.

- We know what the *registers* are and what the *memory* is, and how to have access to both. Memory *cells* have two components: *address* and *content*. Also we can access these cells through a symbolic name, and then we call them *variables*. Variables are declared in the "variables" period. We can assign an *initial value* to these variables or leave them *undefined*.

- We can use memory and registers to make *assignments* (=) and basic arithmetic operations like *additions* (+), *substractions* (-) and *divisions* (/).

3.5) This time your on your own

It's your turn. This is going to be your first attempt into coding a program alone. Remember all you have learned, and refer to it if you need it. The solution to the problem is provided below the example, but before checking it, try it yourself, ok?

Ok, the problem is this: you want to code a program that calculates the distance taken by a particle with an initial velocity of 5m/s (v0), that accelerates at a rate of 10m/s2 (a) in a period of 4s (t). If you remember something about physics, you should know that the formula needed is:

$$r = v_0 t + \frac{1}{2} a t^2$$

r being the distance. That's it. A few comments:

- you need a memory variable for each physical variable.

- you have to know how to do a *multiplication* in Lino. It's so easy. It's like doing addition but subsituting the plus sign with an asterisk (*).

- you don't need a separate operation for doing the power of two. Just think a **little**.

- you can use registers if you want to make the calculations, or use the variables instead (or both). The only thing i ask for is that the final result (r) must be stored in a variable called *distance*. Ready?

Warning: Don't read the solution until you have given this a try... This is just one of the possible solutions.

```
"directors"
      unit = 32;

"variables"
      initial velocity = 5;
      aceleration = 10;
      time = 4;
      distance = undefined;

"programme"
      a = [initial velocity]; ( a = v0 )
      a * [time]; ( a = v0*t ... first part of the formula)
      b = [time]; ( b = t )
      b * b; ( b = t^2 ... t squared )
      b * [aceleration]; ( b = a * t^2 )
      b / 2; ( b = (a * t^2) / 2 ... second part of the formula)
      a + b; ( a = v0*t + (a * t^2) / 2 ... done! )
      [distance] = a;
      end;
```

SECTION 4 - DATA TYPES

Our basic knowlegde of Lino is increasing. At this point, we can code programs that do basic arithmetic computations. Thus, we have something similar to a powerful calculator. But we have only been using *integer* numbers. Integer numbers are those without a *decimal* part (like 6, 1001 or 0). Also we haven't used *negative* numbers. If we want to make our "calculator" a little more complex, we need to add capability of managing negative (like -4, -100) and decimal values (like 2.2, 3.1416 or -600.045). This is because most applications would need to make use of more than just positive or not integer values. For example, imagine you want to store a temperature like -12.5ºC in a variable.

4.1) Negative and floating-point numbers

Let's see how we can declare negative integer values for the initial content of our variables. It's very easy. Imagine we have a variable declaration in the "variables" period declaring a *temperature* variable with an initial value of 12. We should write this:

```
"variables"
      temperature = 12;
```

now, if you want to make this value a negative value, you just have to add the word "*minus*" before the number:

```
"variables"
      temperature = minus 12;
```

Right? Now to declare a decimal value, whose technical name is a *floating-point* number. We want to assign to our *temperature* an initial value of 12.24:

```
"variables"
      temperature = 12.24f;
```

it's easy, Just add an *f* after the number. And what if you want to declare the same value, but negative? Then mix both things:

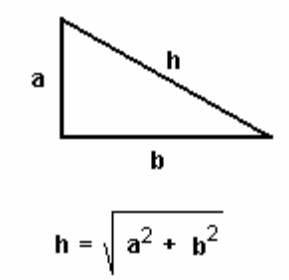
```
"variables"
      temperature = minus 12.24f;
```

That's right. Negative integer values operate exactly the same as positive ones. So you can use the known arithmetic instructions (+, -, * and /) no matter if you are managing negative or positive integer values. But be aware that you can't apply this operations to floating-point values, because floating-point operations are not executed in the same way as integer ones. We'll use another four different instructions to operate with floating-point numbers. These operations are:

INSTRUCTION	*EXAMPLE*
floating-point addition (++)	`a ++ 6.4f;`
floating-point substraction (--)	`d -- [temperature];`
floating-point multiplication (**)	`[speed] ** [time];`
floating-point division (//)	`[number] // minus 3.14159f;`

As you can see, the corresponding floating-point instruction is written like the integer one but by duplicating the symbol. Also looking at the examples, that you'll notice the use of floating-point values in the instructions (like with integers) marking them with the *f*, and negative ones with the *minus* tag.

So you must remember that stored data can be integer values or floating-point values (there are more data types, which will be explained later). You must remember also, that you should only operate data of the same type (integer with integer, floating-point with floating-point), using the corresponding instructions. It's a frequent mistake to use integer instructions (+, -, *, /) with floating-point data. Using integer instructions with variables along with registers containing floating-point data will produce *unpredictable* results, and viceversa. I think an example program will clarify all this better than words. Let's apply this theory (Pitagora's theorem) in this example program:




```
"directors"
      unit = 32;

"variables"
      side a = 2.0f;
      side b = 6.5f;
      hyp = undefined;

"programme"
      a = [side a];
      a ** a;
      b = [side b];
      b ** b;
      a ++ b;
      /~ a;
      [hyp] = a;
      end;
```

There is a new instruction in this program (/~). It calculates the *square root* of the following operand (register A in this case). Take into account that you can *only* use this instruction over floating-point data (notice that A contains floating-point data when the square root is calculated). If you don't know how to write the ~ symbol, press the ALT key, and keeping it pressed, type 126 in the right side of your keyboard, then release the ALT key, it may also be located just prior to the 1 key on the top row on some keyboards.

4.2) Integer/Floating-point conversion

From this example, you may ask yourself: what happens if i have an integer value stored, and i want to calculate its square root? The /~ instruction can only be used over floating-point data, as i've just said. There's no integer square root instruction in Lino. How are we going to achieve this then? We are going to do the following:

1) *convert* the integer value into a floating-point value
2) calculate the square root of the obtained floating-point value

Now we only need to know how to make that conversion. There is another instruction in Lino that allows you to do this. The following code fragment does exactly those two steps (suppose that register A contains an integer value):

```
b ,= a;
/~ b;
```

As you can see, the conversion instruction is the first one (,=). It is like saying "store in B the conversion to floating-point of A". From this moment B contains a floating-

point value, so in the next instruction its ok to calculate the square root. What if you want to do the reverse operation? This is, a conversion of a floating-point value to an integer value, then you need the =, instruction:

```
a =, b;
```

Maybe you can't see the possible uses of these two instructions, but in fact, they are used frequently in Lino programming.

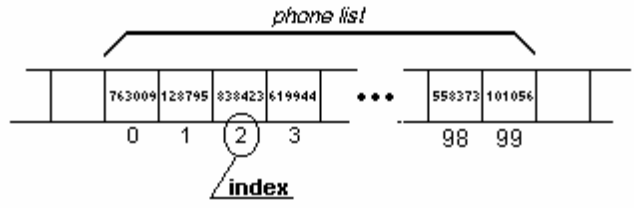
4.3) Complex data: vectors

Great. Using both integer and floating-point data we can now code more complex applications, but there's more. The following example will show you that there are some cases where using one register or one variable to store *single* data values is not appropiate. Imagine you want to create a phone number list with a hundred entries. You should declare one different variable for each of these entries, shouldn't you? So the "variables" period would be something like this:

```
"variables"
      phone entry 0  = 763009;
      phone entry 1  = 128795;
      phone entry 2  = 838423;
      phone entry 3  = 619944;
      (...)
      phone entry 98 = 558373;
      phone entry 99 = 101056;
```

That is, you will have 100 different variables, one for each phone number. So writting this in the source code will take 100 lines of code, right? And what happens if you want 10000 entries instead of 100? Evidently, this method it's not viable. Also, think of this: Imagine you have in register A a number ranging from 0 to 99, obtained by letting the user type it throught the keyboard. This number is the entry of the phone list that the user wants to access. Can you imagine any method to access the correct entry with these variables? Probably not (there is a way, but it's also not viable). We need a solution. We need *vectors.*

So... what is exactly a vector? We can say that a vector is a *set* of memory cells, that can be accessed throught the *same* variable name, but using a certain number that indicates the appropiate entry of the list (this number is called *index*). This scheme should clarify the vector concept:



Observe that the first entry of the vector has the index of 0 (zero). Another important concept is the vector *length,* thats just the number of cells of the vector. In the above example, the *phone list* vector has a length of 100, with their indexes ranging from 0 to 99. So, for a vector with a length of 100, their indexes ranges from 0 to 100-1. Understood? It's not too difficult! Read again if you need it, please.

Now you have the vector concept fixed in your mind, right? How do we apply all this stuff in Lino? First, let's learn how to declare a vector (in the "variables" period), and then we'll learn how to access it, or better yet, it's individual cells.

4.4) Declaring a vector

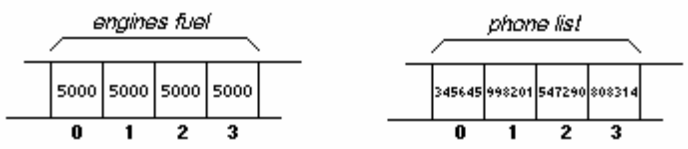
To declare a vector in the "variables" period you can use two different methods. The first one is indicating the length of the vector and the same initial value for every individual cell. Imagine, for example, that we want to declare a vector that indicates the quantity of fuel contained in four (*length*) engines of an aircraft, this quantity being 5000 galons (*initial value*) for each engine at the start of the flight. You should write something like this:

```
"variables"
      vector engines fuel = 5000 *** 4;
```

We indicated that *engines fuel* is a vector by adding the word *vector* before the variable name. Now look at the other side of the equal sign: first you write the initial value of every memory cell, then the *** symbol, and after this you write the length of the vector. Easy. What about the other method of declaring a vector? In this second method, what you do is to declare every initial value of every cell of the vector. The length is implicity declared, depending on how many values you write. The example of the phone list is very appropriate here:

```
"variables"
      vector phone list = 345645; 998201; 547290; 808314;
```

We indicated that *phone list* is a vector the same way as we did with *engines fuel*. Then we wrote the initial values of every cell of the vector, separated by semicolons. In this example we've also written four phone numbers, so the *implied* length of the vector is 4.



4.5) Accessing a vector

So we have declared our vectors in the "variables" period and now we must use them in the program. Let's take the *phone list* example. It's a vector of length 4, this is, formed of 4 memory cells whose indexes are 0, 1, 2 and 3. Imagine that you want to modify one of the phone number entries because a friend of yours has changed his phone number. Let's suppose that the entry of the *phone list* corresponding to that friend is at index number 2. This is what we need to write (in the "programme" period, of course):

```
[2 plus vector phone list] = 448275;
```

Notice that here you must use the *square brackets* as when your accessing a "normal" variable. You need to tell Lino what cell of the vector is going to be accessed by writting the *index* number of the cell, followed by the *plus* tag, and the vector you are accessing. As you can see, you must write the *vector* word before the vector name. *Be careful*, when accessing a vector, of not using an index that is outside the vector limits, because it's possible to do it, but it's not recommended unless you know exactly what you are doing. So, for now, keep the index number inside the range of valid indexes of the vector (in the *phone list* example this range is 0 to 3).

Now we know how to declare vectors, and how to access them through a *fixed* index. What do I mean by fixed? Remember the example about the phone list at section 4.3. What happens if we have an index that must be accessed in a *register*? The only way we can access a vector right now is throught an *immediate* index value, right? So we need some way to access a vector throught an arbitrary index value stored in a register.

4.6) Indirect addressing mode applied to vectors

Ok, we have our *phone list* vector (length of 4), and the user wants to modify an entry of the list. The user types that index throught the keyboard, and, by some magical way, this index is stored in register A. For example, let's suppose that the updated phone number is stored in register D (typed by the user too). The following instruction does exactly what we want to do:

```
[a plus vector phone list] = d;
```

I think it makes alot of sense... Just write the register containing the index instead of an immediate value. If, for example, register A contains value 1, then this instruction is (almost) the same as writting

```
[1 plus vector phone list] = d;
```

with one difference: register A *can* change during the execution of the program, but the inmediate value 1 *cannot*. So this is a more dynamic way of accessing a vector. This way of accessing a vector is called *indirect addressing mode.* Which is just one of its uses, but for simplicity the others will be learned later.

4.7) Vectors example and use of constants

This is a good time for an example program to digest all our knowledge about vectors. Imagine a teacher has a list of his students' grades, and he wants to add a half point to each of them because of good behaviour. He asks us to code a program that does this. So let's do it, then...

```
"directors"
      unit = 32;

"variables"
      vector grades = 4.5f; 5f; 7.5f; 3.5f; 6f;

"programme"
      [0 plus vector grades] ++ 0.5f;
      [1 plus vector grades] ++ 0.5f;
      [2 plus vector grades] ++ 0.5f;
      [3 plus vector grades] ++ 0.5f;
      [4 plus vector grades] ++ 0.5f;
      end;
```

As you can see from this example, vector cells can contain floating-point data apart from integer values (which is logical). You should be able to understand this program perfectly. If not, well, press the PageUp key... Now notice that for each grade we did a floating-point addition, adding an inmediate value 0.5f to each one. If we need to modify the program because the teacher gets crazy and decides to add 2 points to the students' grades, we have to change every instruction to reflect this update. There are only five students in this classroom, but imagine doing this in a big class of 150 students... you would need to change *every* single value, that would be desperaging. The solution to this problem is the use of *constants*:

```
"directors"
      unit = 32;

"constants"
      grade addition = 2f;

"variables"
      vector grades = 4.5f; 5f; 7.5f; 3.5f; 6f;

"programme"
      [0 plus vector grades] ++ grade addition;
      [1 plus vector grades] ++ grade addition;
      [2 plus vector grades] ++ grade addition;
      [3 plus vector grades] ++ grade addition;
      [4 plus vector grades] ++ grade addition;
      end;
```

Now we have included the "constants" period to define constants for our program. A constant is just a name assigned to an immediate value. After you have defined a constant, you can use the constant name instead of writing the value directly. Advantages? Notice that if you wanted to change the grade addition of the students, now you only need to change the value in the constants period, instead of updating each individual add instruction. Remember the 150 students classroom and you'll see this advantage clearly. Also, the use of constants gives your programs more *clarification* when reading the code (like comments), and as we said, this is very important. But you *must* know that a constant is *not* a variable: you can't modify its value during execution (that's why they are called constants, their value remains constant all the time). But constants also don't take any memory cells from the memory. Here is another version of the grades program but using registers in indirect addressing mode:

```
"directors"
      unit = 32;

"constants"
      grade addition = 2f;

"variables"
      vector grades = 4.5f; 5f; 7.5f; 3.5f; 6f;

"programme"
      a = 0;
      [a plus vector grades] ++ grade addition;
      a +;
      [a plus vector grades] ++ grade addition;
```

```
a +;
[a plus vector grades] ++ grade addition;
a +;
[a plus vector grades] ++ grade addition;
a +;
[a plus vector grades] ++ grade addition;
end;
```

Register A contains the index of the vector cell we are going to access. Initially, we assign it to 0 (zero), in order to access the vector cell with index 0 (the first one). After each access to the vector, you can use this instruction:

```
a +;
```

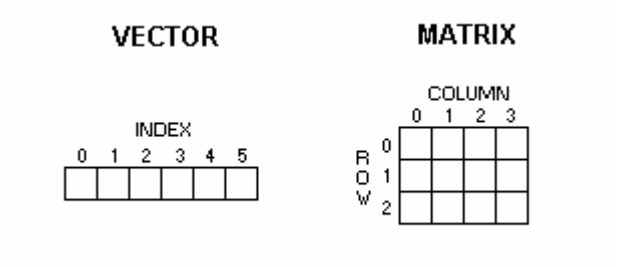
this instruction is called an *increment*, and its effect is exactly the same as doing:

```
a + 1;
```

but you should use the increment instead of the addition just because it's just plain faster. So after every access to the vector, register A is incremented by 1, causing the next instruction we access to go directly to the next student grade.

4.8) Matrixs

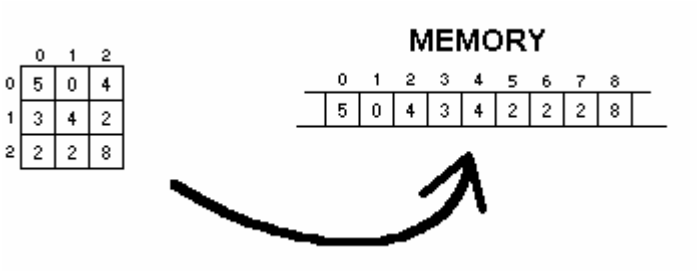
A vector is a set of memory cells, as we know. It is said of vectors to have *one dimension*, because their elements are accessed through one index. *Matrixs* are like vectors but with *two or more dimensions*. For this tutorial, we'll only take care of *2-dimensional* matrixs (so when I say matrix i'm refering to a 2-dimensional matrix).



The vector of the scheme has a length of 6. And what about the matrix? Because it has two dimensions, we need to indicate its horizontal length (*width*), and its vertical length (*height*). So this matrix has a width of 4 and a height of 3. This is the same as saying that it is a 4x3 matrix. And how do we declare a matrix in Lino. This way:

```
"variables"
      matrix my matrix = 5; 4; 6;
                         9; 0; 0;
                         1; 2; 2;
```

As you can see it's a 3x3 matrix. We declared a matrix with the *matrix* word instead the *vector* word. But how is a matrix represented in memory? Like a vector. This scheme will explain this concept better than a thousand words:



So, if a matrix is represented like a vector, then we can access its elements through a single index. Look at the above scheme. Now look at the element containing a 3. It is the element at position (0, 1), isn't it? Now look at the memory representation of the matrix, and see that its index inside the vector is 3. We need some formula to associate a row and a column with the position in the vector representation. Try this:

vector index = (row * matrix width) + column

so, the example above is the same as:

vector index = (1 * 3) + 0 = 3

Let's see how to access a matrix in Lino through this formula (using indirect addressing mode):

```
"directors"
      unit = 32;

"constants"
      width = 3;
      height = 2;

"variables"
      matrix my matrix = 3; 5; 4;
                         0; 1; 9;

"programme"
      (change the (2,1) element)
      a = 1; (row)
      a * width; (row * width)
      a + 2; (row * width + column)
      [a plus matrix my matrix] = 5;
      end;
```

Let's review what we have learned in this section:

- registers and memory cells can contain *integer* and *floating-point* values, and this values can be *positive*, *negative* or *zero*. Integer and floating-point values *must* be operated using their corresponding arithmetic instructions. Data can be *converted* from one format to another, when needed. Some instructions like the *square root* (/~) can only be used over floating-point data.

- sometimes data needs to be grouped in *vectors*. Vectors are sets of memory cells refered to by the same name but with an index indicating its position inside the vector. We have learned how to *declare* vectors and how to *access* them in the "programme" period (with the index being and *immediate value*, or by using the *indirect addressing mode*). *Matrixs* are two-dimensional vectors.

- we have also learned how to define *constants* in our programs and the advantages of their use.

Note: More data types exist (like the *strings*) but they will be explained in later sections.

SECTION 5 - EXECUTION CONTROL

Until this moment, all our programs have been executed secuentially, from the first instruction to the last one, one after the another, and unconditionally. Usually, if you want to code an application, you shouldn't be restricted by this limitation.

5.1) Labels

Before going deeper within this section, we must learn what a *label is*. A label is like a bookmark in a book. You can use labels to mark certain points in your program. Labels are names (like constants or variables) written between *quotes*. Like this one:

```
"this is a label"
```

Labels are **not* instructions, so they are not executed in any way, and so they have not *direct* effect over the program results. But, of course, they are not useless.

5.2) Unconditional jumps

So, how can we break the secuential nature of our programs? Easy... just "*jumping*". No! No! Not that way, please sit down again... A *jump* is an instruction. You know that in all our programs coded until now, that after the execution of one instruction, the instruction that is executed after, is very next one placed after it in our code, right? Well with a jump, you can break this sequence. And what instruction will be executed next, then? Well, along with the jump instruction you indicate a *label name*, that will be the destination of the jump. Look at this code fragment:

```
a = 0;
-> jump now;      (this is the jump instruction)
a = 2;            (this instruction is never executed)
"jump now"        (this is the destination of the jump)
a +;
```

What is the effect of this code? First, value 0 (zero) is stored in register A. Then comes the jump instruction. To do a jump, you write the jump symbol (->) followed by the name of the label that will be the destination of the jump. And what instruction is executed after? Easy. Instead of executing

```
a = 2;
```

the very next line it jumps to the instruction after the "*jump now*" label, which is:

```
a +;
```

Understood? So, that code fragment has the same effect than this one:

```
a = 0;
a +;
```

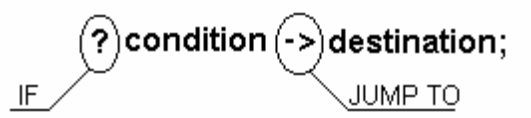
and the final value contained in register A is 1.
These kinds of jumps are called *unconditional jumps*, because no matter what happens, if the program finds one of these jumps, they are always done. You must also keep in mind, that the destination of a jump doesn't need to be *after* the jump instruction. The following code shows this idea:

```
a = 0;
"jump now"
a +;
-> jump now;
```

Here, register A is assigned to a value 0, at first. Then it's incremented by 1 (notice that the label is not executed, as we said). After this, the program jumps to the "*jump now*" label, so the next instruction executed is the increment (again). And after the second increment, we jump again. And this process is repeated again and again. So register A is incremented again and again...
This is what we call a *loop*. And this kind of loop is called an *infinite loop*, because it never ends (unless you switch off the computer, of course). As you can imagine, this kind of loop is useless (and must be avoided too). But there are other kind of loops that are not that useless.

5.3) Conditional jumps

There are other kinds of jumps that have the property of being done only under a certain condition (*conditional jumps*). The structure of a conditional jump instruction goes like this:



This is like an unconditional jump, because it contains the jump symbol (->) followed by the label name that is the destination of the jump. However before this, you must write the *if* symbol (?) and the condition needed to do the jump. So you can word one of these instructions this way "*if condition is true then jump to destination*". If the condition is not true, then the jump is not done, and the next instruction executed is the one in the next line of code. And what does such a condition looks like? Let's see a code example:

```
a = 10;
? a = 5 -> my label;
a +;
```

```
"my label"
(...)
```

Here the condition is: if register A *is equal* to 5 then jump to "my label"
Notice that the condition is *not* an
assignment in this case, just a *comparation* of values, right? In this case, the condition is *not true* (or false), because register A contains value 10 (which is not 5, evidently); this means that the jump is not done, and the next instruction executed is:

```
a +;
```

so after "*my label*", register A contains the value 11. Let's see another similar example:

```
a = 10;
? a < 20 -> my label;
a +;
"my label"
(...)
```

In this case the condition is different. The instruction can be read as "*if* register A *is lower than* 20 then *jump* to "*my label*". So in this case, the condition is true, because 10 is lower than 20, and the jump is done. This means that after "*my label*", register A contains the value 10 and not 11. Difficult? Not really... You can use variables instead of registers to make comparisons, and you can also compare registers with registers or registers with variables and so on. These instructions are valid, too:

```
? a = b -> my label; (comparing two registers)
? [my variable] < 100 -> other label; (you can use variables too)
? c = [apples] -> jump here; (register and variable)
? [apples] < e -> jump now; (variable and register)
(the next one uses a vector entry)
? [1 plus vector phone list] = 345455 -> it is my phone;
(...you can combine registers, variables, vectors, and inmediate values,
 but inmediate values must always be placed at the right side)
```

There are more conditions used in Lino, and some of them are these:

condition	*meaning*
REG or VAR = REG, VAR or INM	equal
REG or VAR != REG, VAR or INM	not equal
REG or VAR > REG, VAR or INM	greater than
REG or VAR < REG, VAR or INM	lesser than
REG or VAR >= REG, VAR or INM	greater or equal than
REG or VAR <= REG, VAR or INM	lesser or equal than

REG - REGISTER VAR - VARIABLE INM - INMEDIATE VALUE

Another thing. If you want to compare floating-point values instead of integer values, you must write the *if* symbol doubled (??):

```
?? a != 3.14159f -> jump now;
?? [my var] <= 560.122f -> another jump;
```

Let's see a practical example of how to use conditional jumps. Do you remember the students' grades example in the section 4.7? Sure. Imagine that now the teacher only wants to add the *grade addition* to those students that didn't pass the exam (those whose grade *is lesser than* 5). So we can do the following:

```
"directors"
      unit = 32;

"constants"
      grade addition = 0.5f;

"variables"
      vector grades = 4.5f; 6f; 7f; 2.5f; 9.5f; (5 students)

"programme"
      (using indirect addressing mode)
      a = 0;
      ?? [a plus vector grades] >= 5f -> check student 1;
      [a plus vector grades] ++ grade addition;
      "check student 1"
      a +;
      ?? [a plus vector grades] >= 5f -> check student 2;
      [a plus vector grades] ++ grade addition;
      "check student 2"
      a +;
      ?? [a plus vector grades] >= 5f -> check student 3;
      [a plus vector grades] ++ grade addition;
      "check student 3"
      a +;
      ?? [a plus vector grades] >= 5f -> check student 4;
      [a plus vector grades] ++ grade addition;
      "check student 4"
      a +;
      ?? [a plus vector grades] >= 5f -> all grades checked;
      [a plus vector grades] ++ grade addition;
      "all grades checked"
      end;
```

Notice that we use the ?? symbol for floating-point comparations. For each student we compare their grade with value 5f; if the grade is *greater or equal* than 5f (the student has passed the exam) and thus the program avoids the addition and checks the next student. If not, the grade is updated. Check the code slowly and try to understand it.

Now imagine this situation: there are 150 students in the classroom instead of five. Are you going to write this program the same way but with 150 students?? If the answer is yes, you are completely crazy. I'll show you a better, more elegant way of achieving this:

```
"directors"
```

```
      unit = 32;

"constants"
      grade addition = 0.5f;

"variables"
      vector grades = 4.5f; 5f; 8f; (...150 students in total)

"programme"
      a = 0;
      "update next grade"
      ?? [a plus vector grades] >= 5f -> do not update this one;
      [a plus vector grades] ++ grade addition;
      "do not update this one"
      a +;
      ? a < 150 -> update next grade;
      end;
```

Wow! Little code. Notice that: register A is initialized to zero (so it points to the first student). Then we update the grade if it is lesser than 5f, right? If not, it is not updated. Then we increment register A so it points to the next student grade. And then comes the *magic* instruction. We check if the student pointed to is lesser than the 150th, if so we continue with the process. If at this point register A contains value 150, it means that we have checked all the students. Why? Because the vector indexes go from zero to 149. Think about this, and sooner or later you'll understand it. What we've just done is a *loop,* and a *finite* one, instead of an *infinite* one. Finite loops are not useless as you can see...

5.4) Loop until zero

There is another way to do a finite loop, and it's the *loop until zero*. It's an special instruction, and you can use it this way:

```
a = 10; (number of iterations)
"here starts the loop"
(...some code here...)
a ^ here starts the loop; (loop until zero)
```

First, store the number of times (number of *iterations*) you want to repeat the loop in a register or variable (register A in the example), then put a label indicating the start of the loop. Then write the code block that must be executed in the loop, and then at the bottom put the loop until zero instruction; in this instruction you must write the register or variable that stores the number of iterations followed by the ^ sign, and then the label where the loop must start again. This instruction does exactly the following:

1) the value of the register (or variable) is *decremented* by 1
2) the value of the register (or variable) is *compared with zero*
3) if it *doesn't equal zero*, then it *jumps* to the label indicated

If you think about it a little, you will see that the loop is done exactly the number of times that you stored in the register (or variable). The grades example can also be written this way...

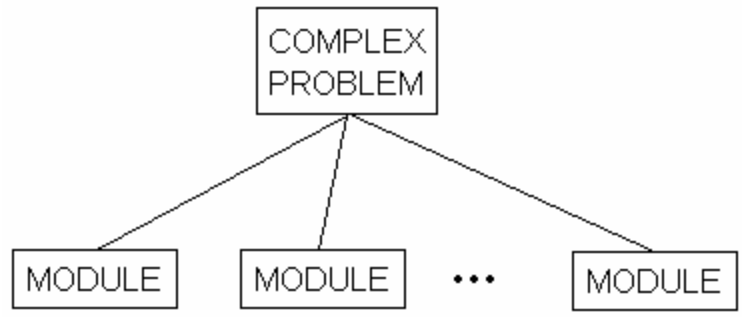
```
"directors"
      unit = 32;

"constants"
      grade addition = 0.5f;

"variables"
      vector grades = 4.5f; 5f; 8f; (...150 students in total)

"programme"
      a = 150; (number of iterations)
      b = 0; (now b points to the student grade)
      "update next grade"
      ?? [b plus vector grades] >= 5f -> do not update this one;
      [b plus vector grades] ++ grade addition;
      "do not update this one"
      b +;
      a ^ update next grade;
      end;
```

SECTION 6 - SUBROUTINES

In the last section, we coded programs that can do complex tasks. So from now on our programs can be more and more complex. If you want to code some decent applications in the future, the *size* of the code will be heavily increased. And when the size of the code grows it is harder to keep it readable.

6.1) Divide and you will conquer

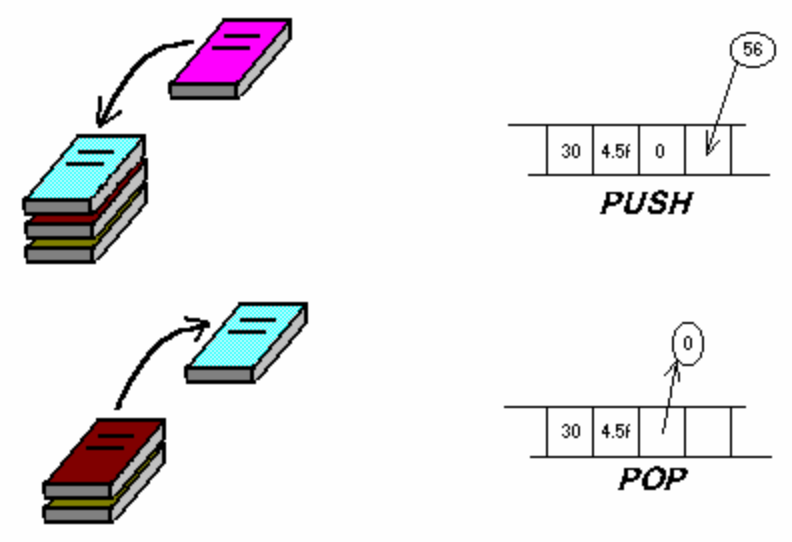
One solution is to take the global problem (that is supposed to be complex), and divide it in several smaller and simpler parts (called *modules*), and join them together.



So each module is used for a single (and relatively simple) task. There is also a special module called the *main module*, it's job is to direct the application of the rest of the modules. In Lino you can do modular programming through what we call *subroutines*. But before learning how to create subroutines we must learn...

6.2) The stack

Imagine you have a lot of books. Now imagine you put one of these books in a box. Then you put another one on top of the previous book. And so... Now, if you want to take a book out, you must take the book out that is at the top of the stack first, right? So to put a book *into* the box, you must put it on top of the previous book, and if you want to take out a book *from* the box, you must take the one at the top. Now convert the books into data values, and the box into a portion of the central memory we call the *stack*. You can *push* a data value *into* the stack over the previous data value *pushed*. Also you can *pop* a data value *from* the stack, but it must be the one at the top.



So the only two operations you can do over the stack are *push* data and *pop* data. Althought it seems useless, the stack is required for a lot of tasks. Let's learn how to use the stack in a Lino program:

```
a = 10;
--> a; (push 10)
a = 20;
<-- a; (pop 10)
```

What happens here? First register A is assigned to value 10; then we push (-->) its contents into the stack (imagine the scheme). Then register A is assigned to value 20; at last we pop (<--) the value at the top of the stack (10) into register A. So register A will contain value 10 at the end of this code fragment. So simple!
An important note: you *must* make a pop for each push instruction in a program. Remember this. If not your programs will report *run-time* errors.

6.3) Subroutines in Lino

Ok, now we are ready to divide our Lino programs into modules. These modules are called *subroutines*. Subroutines are like little programs. They take some *input* data, make some calculations with that data, and return some *output* data, generally the result of those calculations. Look at this program:

```
"directors"
      unit = 32;
```

```
"variables"
      vector peter grades = 5.5f; 4.5f; 7f;
      vector johny grades = 7f; 8.5f; 6.5f;
      peter average = undefined;
      johny average = undefined;

"programme"
      a = [0 plus vector peter grades];
      b = [1 plus vector peter grades];
      c = [2 plus vector peter grades];
      a ++ b;
      a ++ c;
      a // 3f; (grades average)
      [peter average] = a;
      a = [0 plus vector johny grades];
      b = [1 plus vector johny grades];
      c = [2 plus vector johny grades];
      a ++ b;
      a ++ c;
      a // 3f; (grades average)
      [johny average] = a;
      end;
```

As you can see, in this program we have two vectors containing the grades of three exams, one for Peter and one for Johny, and we calculate the average grade of each one. Observe that the way of calculating the average is the same in both, except for the input data, and the place where we store the results. We are going to create a subroutine to calculate the average of three grades, no matter which student it is, we want to know their average. We are going to assume that the subroutine gets the input data from A, B and C (the three noted) and the average (output) will be stored in register A, for example. The subroutine will look like this:

```
"calculate average"
(a, b and c contains the three grades)
a ++ b;
a ++ c;
a // 3f;
(a now contains the average)
leave;
```

We are going to refer to the subroutine by its name, and we declare its name with a label "*calculate average*". This is because we are going to do something like a jump in order to use the subroutine. Notice that i've written the *leave* tag, this marks the end of a subroutine. And how can we use this subroutine to calculate the two specific averages of Peter and Johny? Well, something like this will work:

```
"programme"
      a = [0 plus vector peter grades];
      b = [1 plus vector peter grades];
      c = [2 plus vector peter grades];
      => calculate average;
      [peter average] = a;
      a = [0 plus vector johny grades];
      b = [1 plus vector johny grades];
```

```
c = [2 plus vector johny grades];
=> calculate average;
[johny average] = a;
end;
```

In this code, we loaded A, B and C with the input data of Peter, and then we *called* the subroutine (=>). To use a subroutine you must call it. When you do this call, the code of the subroutine is executed, and after that, the next instruction executed is the one following the call instruction. Now after the subroutine call we have the average in register A, so we store it on a variable (*peter average*), for a possible use in the future, and because we are going to do the same with Johny's grades, so register A will be overwritten with other input data. Notice that we don't have to rewrite all the code corresponding to the average calculation because we used the subroutine call instead. The complete program is:

```
"directors"
      unit = 32;

"variables"
      vector peter grades = 5.5f; 4.5f; 7f;
      vector johny grades = 7f; 8.5f; 6.5f;
      peter average = undefined;
      johny average = undefined;

"programme"
      a = [0 plus vector peter grades];
      b = [1 plus vector peter grades];
      c = [2 plus vector peter grades];
      => calculate average;
      [peter average] = a;
      a = [0 plus vector johny grades];
      b = [1 plus vector johny grades];
      c = [2 plus vector johny grades];
      => calculate average;
      [johny average] = a;
      end;

      "calculate average"
      (a, b and c contains the three grades)
      a ++ b;
      a ++ c;
      a // 3f;
      (a now contains the average)
      leave;
```

Notice how we have divided the program into smaller, more readable modules. One is the main module (the code between the "programme" period name and the *end* tag), and the other is the subroutine (the code from the *calculate average* label and the *leave* tag). And more importantly: we have *reused* our code.

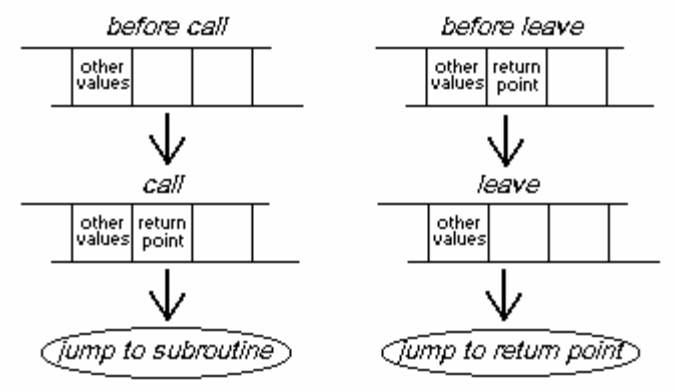
6.4) What happens when you call a subroutine?

You must know how this process is really done. When the program finds a call instruction (=>) it does the following:

1) push into the stack the location where to return from the subroutine (this is, the instruction that is just below the call instruction in the code)
2) jumps to the label indicated in the call instruction

This is because when we call a subroutine, the program must know where to return after the execution of the subroutine code. And what happens when the program finds the *leave* tag inside the subroutine code?

1) pop from the stack the location where to return (previously pushed with the call instruction)
2) jumps to that location



As you would imagine, it is important to pop every value pushed *inside* a subroutine, because when the *leave* tag is executed, the top of the stack *must* contain the *return point*.

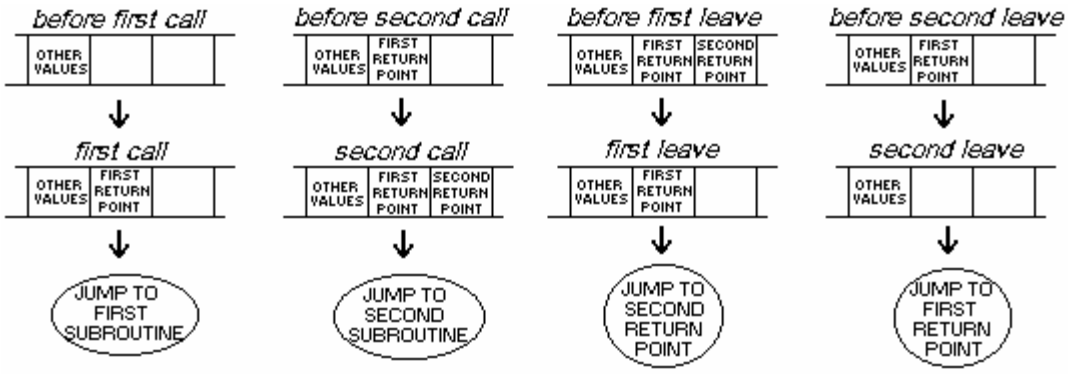
6.5) Calling a subroutine from other subroutine

So when programming, we divided the program into modules, and these modules are called from the main module. However suppose one of the modules is a complex task.. we can also apply modularity over single modules. As an example: we might want to do something similar to the previous example, but adding the *grade addition* to the average grade. What we are going to do is to create two subroutines, one named "*calculate updated grade*" (that will return the average updated by the *grade addition*), and another named "*calculate average*" (which is exactly the same one used in the previous example). From the main module we will call "*calculate updated grade*", obtaining the final grade directly. Which from within this subroutine we've called "*calculate average*" and added the *grade addition* to it:

```
"directors"
      unit = 32;

"constants"
      grade addition = 0.5f;

"variables"
      vector peter grades = 5.5f; 4.5f; 7f;
      vector johny grades = 7f; 8.5f; 6.5f;
      peter average = undefined;
      johny average = undefined;

"programme"
```

```
a = [0 plus vector peter grades];
b = [1 plus vector peter grades];
c = [2 plus vector peter grades];
=> calculate updated grade;
[peter average] = a;
a = [0 plus vector johny grades];
b = [1 plus vector johny grades];
c = [2 plus vector johny grades];
=> calculate updated grade;
[johny average] = a;
end;

"calculate updated grade"
=> calculate average;
a ++ grade addition;
leave;

"calculate average"
a ++ b;
a ++ c;
a // 3f;
leave;
```

Now we have called a subroutine from within another one. Look at the scheme and you'll discover what happens on the stack when we do this



SECTION 7 - THE WORKSPACE

In section 3 we learned the structure of the memory, and how to use variables in order to store data. Let's delve deeper into these concepts.

7.1) The truth about variables and vectors

We know that a variable is a memory cell storing a single data value, accessed through a symbolic name. But in central memory the memory cell (assigned to that variable name) is accessed through its address. Until this point we accessed variable content this way

```
[my variable] = 10;
```

so when you put a variable name between square brackets, you are refering to the variable *contents*. Now you must know that you can refer to the variable *address*, too. Remember this:

- when you write the variable name *between square brackets*, you are refering to the *content* of the variable (the content of its memory cell).

- when you write the variable name *without the square brackets*, you are refering to the *address* of the variable (the address of its memory cell).

It's important for you to memorize this, and have this concept clear in your mind. When you declare a variable in the "variables" period, its address is determined by Lino, and not by you. Imagine you declare a variable called *my variable*, with an initial value of 10, and Lino assigns it to the address 1056 inside the memory, then:

```
[my variable]
```

means 10 to Lino (the variable content), and:

```
my variable
```

means 1056 to Lino (the variable address). Understood? If not read it again and don't continue with this section until you have understood completely.

And what about vectors? Something similar. If you declare a vector in the "variables" period, Lino assigns it a starting address, this is, the address of its first memory cell. Imagine you declare a vector whose name is *my vector*, and Lino assigns the address of its first cell to 38490, then:

```
[vector my vector]
```

means the content of the first cell of the vector to Lino, and:

```
vector my vector
```

means 38490 to Lino (the address of its first memory cell). Logically, the address of its second cell is 38491, and so... This is the reason why when you want to access the content of the second cell of the vector you write:

```
[1 plus vector my vector]
```

this is, the contents of the cell with address 1 + 38490 (38491).

7.2) Indirect addressing mode in general

So, to summarise, if you write an address between square brackets, then you are refering to the content of the memory cell with that address. Remember indirect addressing mode applied to vectors? Well, it can be used to access memory in general, not only vectors. The truth is that you use indirect addressing mode by storing the

address of the memory cell you want to access into a register, and then writting that register between square brackets. So it's perfectly possible to do the following:

```
a = my variable; (a = address of my variable)
[a] = 10; (the same as writing [my variable] = 10;)
```

And you can add a number to the register with the *plus* tag to point to another cell:

```
a = my variable;
[a plus 1] = 10; (the same as writing [my variable plus 1] = 10;)
```

It's all a matter of *converting* names into addresses. If you think a little, the indirect addressing mode applied to vectors makes sense too:

```
[a plus vector my vector] = 10;
```

7.3) Using the workspace

We've being using the "variables" period to declare variables and vectors. We declared our variables and gave them an initial value. You should know that every variable or vector you declare in this period *increases the size of the application file*. So if you declare a really big vector with a length of 100000, then the size of the file will be incresed heavily. There is another way of defining variables and vectors without increasing the size of the file, but their initial value will be *always zero*. This is done using the "workspace" period. Declaring variables and vectors in this period is similar to doing it in the "variables" period, but a little different. Instead of writing the initial value of the variable or vector (which will be alway zero), you declare how many memory cells you want to use through the variable/vector name:

```
"workspace"
      a variable = 1; (reserve one memory cell)
      another variable = 1; (reserve one memory cell)
      big vector = 10000; (reserve 10000 memory cells)
```

Every name declared in the "workspace" period is treated as vector, including those that reserve only one memory cell. But with one difference. You don't need to write the *vector* word with the names declared in this period. Let's see an example program:

```
"directors"
      unit = 32;

"variables"
      vector my vector = 5f; 0f; 1f; (a 3d vector)
      scalar factor = 10f;

"workspace"
      result vector = 3; (a 3d vector too, but uninitialized)

"programme"
      a = vector my vector;
      b = result vector; (don't use the vector word with this)
      [b plus 0] = [a plus 0]; (x component)
      [b plus 0] ** [scalar factor];
      [b plus 1] = [a plus 1]; (y component)
      [b plus 1] ** [scalar factor];
      [b plus 2] = [a plus 2]; (z component)
      [b plus 2] ** [scalar factor];
      end;
```

As you can see vectors defined in the "workspace" period, don't need to be marked with the *vector* word.

SECTION 8 - THE DISPLAY

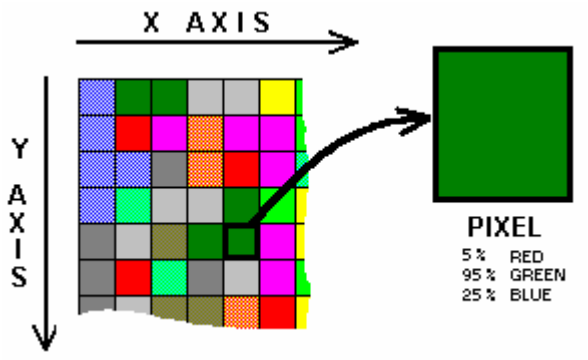
In this section we are going to learn how to use with Lino with one of the more important output devices of a computer: the *display*. But before that, let's see what this thing called the isokernel is.

8.1) The isokernel

The *isokernel* is like a module, included in *every* Lino program. It allows you to do several things. You tell the isokernel what you want to do through some special variables (the *isokernel variables*), and then calling the *isokernel subroutine*. As i've just said there are a lot of different tasks you can achieve through the isokernel, but in this section we are going to concentrate on a certain group of them. To be specific we are going to use the display variables group of the isokernel. But... what is the display in Lino? How do we access it? And how is it organizated?

8.2) Video memory and primary display

You probably know that the physical screen (what you are looking at right now) is divided in little individual elements called *pixels* (coming from *picture elements*). Pixels are those little light points that form the image on the screen. The screen is then a *matrix* (remember matrixs?) of pixels, with a certain number of pixels forming the X axis (*width*), and a certain number of pixels forming the Y axis (*height*). So, if the screen has a width of 800 pixels and a height of 600 pixels, then there are 800*600 = 480000 pixels on the screen. These two parameters can be modified. Also, each pixel on the screen can be shown in a different *color*. The color of a pixel is represented by a number, divided into three components. These components are *red*, *green* and *blue*. By mixing certain quantities of these three colors, you can obtain any color you want.



Basically, that's all. By asigning certain colors to each pixel of the screen you obtain the desired image output. The most important thing i want you to remember is that the screen is a *matrix*. So if you want to modify the color of a pixel then you must know its row and column, and then access the matrix (you know how to access a certain element of a matrix, don't you? It is explained in section 4.8. Right. Now, the physical screen matrix is stored in a special memory called the *VRAM* (*Video RAM*) or *video memory*. So, you can think that if we change this matrix directly, the problem is solved. Well, you are right, but instead of that we are going to do the following:

1) declare another matrix in central memory of the same *width* and *height* of the *video memory*. We'll call this matrix the *primary display.*
2) modify the pixels we want to change in the *primary display.*
3) copy the *display memory* into the *video memory.*

Seems strambotic, but in practice is an easy process. Let's see how to do it in Lino. First, know that in Lino, the whole screen can be used by your program, or just a window inside the screen (something similar to Windows' windows :P). At the moment we are going to use the window method. But, at all sizes, you won't notice much of a difference when accessing the screen. Now, we must decide the size of our display (this is, the width and height of the display). Let's take, for example, a 300x300 display. So our primary display matrix will have 300x300 = 90000 memory cells. Now, we must declare the primary display matrix (as i've just said). In order to keep the executable file size to a minimal, we'll declare the primary display in the "workspace" period:

```
"workspace"
      primary display = 300 multiplied 300; (300x300)
```

The *multiplied* tag multiplies the two values, so its the same as writing the 90000 value directly, but it's clearer to the reader (and to the programmer this way). Now that it's declared, let's learn how to do the other two steps of the list in the "programme" period by drawing a single pixel in the screen, this is where we will use the isokernel for the first time.

8.3) Drawing a pixel

To use the display in Lino, we need to mess with the display variables group of the isokernel. There are several variables in this group, but we aren't going to use them all. Just these ones:

```
display width
display height
display origin
display command
```

With *display width* and *display height* we'll indicate to the isokernel the width and height of the *primary display* matrix. Simple. With *display origin* we'll tell the isokernel the address of the first memory cell of the *primary display* matrix (you should know that by writing the matrix name without square brackets you are refering to this address). We'll use *display command* to tell the isokernel that we want to copy the primary display into the video memory (at the appropiate moment, of course), that is, the third step of the list mentioned before. Here is the program:

```
"directors"
```

```
        unit = 32;

    "workspace"
        primary display = 300 mtp 300;

    "programme"
        [display width] = 300;
        [display height] = 300;
        [display origin] = primary display;

        a = 100; (row)
        a * 300; (row * width)
        a + 150; (row * width + column)
        [a plus primary display] = FFFFFFh; (white color)

        "show the display"
        [display command] = retrace;
        => [isokernel];

        [console command] = get console input; (*)
        => [isokernel];                        (*)
        ? failed -> show the display;          (*)
        end;
```

Notice how it's assigned the width, height and origin of the display. After that we modify the pixel at (150,100) to be white (in a few lines i'll explain that strange FFFFFFh). Then we use the *display command* variable to tell the isokernel to *retrace* the display (this is, to copy the primary display into the video memory), and this order is executed by calling the *isokernel subroutine* (=> [isokernel];). For now, the only thing you must know about the lines marked by (*) is that they wait for any key to be pressed to finish the program. You can write this program into a text file, compile it, and execute it to see the results. And why is the pixel we drew white? It's something related to *hexadecimal* numbers, but i'm going to make it a little simpler for you (this is a tutorial, isn't it??).

As i said, a color is formed by three components (red, green and blue). Well, one method to declare these three components for a color is using hexadecimal notation. For declaring the intensity of one of this components you use two hexadecimal digits, which are these: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.
Instead of using decimal notation, which numbers are

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, ...

we use this progression in hexadecimal notation:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 1A, 1B, 1C, 1D, 1E, 1F, 20, 21, ...

And later...

...,9B, 9C, 9D, 9E, 9F, A0, A1, A2, A3, A4, ...(being 95, 96,97,98,99,100, 101,102,103,104 in decimal)

And so. Then, with two hexadecimal digits you can write numbers ranging from 0 to FF. Now, we use six hexadecimal digits to define the three components of the color. Depending on the intensity of each component you obtain one color or another.

Some basic colors and their hexadecimal representation are:

Red = *FF0000*
Green = *00FF00*
Blue = *0000FF*
Cyan = *00FFFF*
Yellow = *FFFF00*
Gray = *909090*
White = *FFFFFF*

If you want to obtain other colors, you should experiment...One last thing. To write a hexadecimal number in Lino you need to add the *h* letter at the end of the number:

```
[a plus primary display] = FFFFFFh; (white color)
```

And that's all.

SECTION 9 - LIBRARIES

In section 6 we learned how to do modular programming to get clearer programs and reutilizate our code. In this section you'll learn how to create subroutines that you will use in any program you want without rewriting it.

9.1) What is a library?

A *library* is a Lino file with some subroutines that you can *include* in any of your programs. If you include a library into a program, then you can call any of the subroutines contained in that library. The advantages of this is that you can write a usefull subroutine and save it like a library. Later, you can write a program using this subroutine without having to rewrite the subroutine. Also, if you use libraries coded by other programmers, you don't need to know the code of the subroutine, just the input and output interface, and its name in order to call it. First, let's see how to include libraries in our Lino programs.

9.2) Including libraries into programs

If you want to use a created library in your program, you must indicate its file name in the "*libraries*" period, *without the TXT extension*. In the following example we include a default library included in the L.in.oleum package called *random.txt*, that contains subroutines for generating random numbers, and it's placed in the *gen* folder:

```
"libraries"
      gen/random;
```

If you want to know more about this, please refer to the Lino Manual (in the "libraries" period section).