HOW TO PROGRAM A COMPUTER

Using the oo2c Oberon-2 Compiler

by Donald Daniel

2001

Revised Mar 2015

Updated for version 2 of oo2c

www.waltzballs.org

CONTENTS

- Introduction
- Installing linux in addition to windows
- Installing the Compiler
- Syntax
- Structure
- Control
- Data Structures
- Input/Output
- Separate Compilation
- Debugging
- Interactive Programs
- Graphics Output
- Audio Output
- Mathematical Programming
- Programming Hints
- How to Convert Pascal to Oberon

If the lines of text are too long you can fix the problem with these instructions.

up one level

INTRODUCTION

For a quick example of what programming is all about, jump ahead to the example of the quarter mile running track.

On television, the author of the book "Finding the next Steve Jobs" said that he "would rather hire a self taught programmer than a PhD in computer science". So this self taught course may be your big chance.

All of the software used here can be downloaded free from the internet and costs absolutely nothing. The only background you need to do everything presented here is the education of a typical 18 year old. Only a small portion, trigonometric functions and complex numbers, requires that much education. So many instructions are presented here that you will make mistakes and have to start over again. That is normal. If you are determined and persistent you will succeed. If you cannot stand to do anything where you will make mistakes, you should consider janitorial work instead. If you cannot run a mile that means your body is out of shape. If this course seems too difficult in the beginning, that means your mind is out of shape. Your mind can get out of shape just like your body can. By the time you master this material your mind will be in much better shape.

A computer memory is a single long list of "binary words". Each word has an address that specifies where it is in the list. If each word consists of 32 binary bits, the computer is called a 32 bit computer. If each word is 64 binary bits, it is a 64 bit computer. You will need to know whether your computer is a 32 bit or a 64 bit computer so you can select the right version of software to install on your computer. What we have just described is the way software uses memory, which we can call the logical arrangement of memory. Sometimes the physical arrangement of the memory is different than the logical arrangement of the memeory. The logical arrangement of the memeory is what determines the right software to install. For instance one 64 bit computer had words arranged physically as 256 bit words. The central processing unit, also called the CPU or the microprocessor, will be described as 32 bit or 64 bit according to the logical arrangement of memory.

Oberon 2 is worth learning because of its clarity, simplicity and ease of use compared with the dominant language of today, C. C started out as a full fledged production language, but a very tricky one to use. Pascal was developed about the same time as C but as a simple teaching language. Pascal was developed by Niklaus Wirth through successive languages called Modula, Modula-2, Oberon and Oberon-2 to become a production language. Thus Oberon-2 in a very weak sense is almost Pascal-5, though different enough from the original Pascal that Pascal-5 would not quite be appropriate as a name for it. In saying this it should be noted that there is a commerical product called Pascal-5 that is entirely different from Oberon-2. Oberon-2 is clear and safe, C is obscure and tricky. No doubt C's trickiness has convinced many a beginner that he did not have what it takes to be a programmer, and has cost many a project cost overruns because of the difficulty of writing bug free C programs. Like the C, Oberon has separate compilation, and a separate library. Like the newest version of C, C++, it also has objects. This introductory course does not cover objects. Objects are covered in the advanced book "Object-Oriented Programming in Oberon-2" by Hanspeter Mossenbock published by Springer-Verlag. Oberon-2 is much simpler than other contending Pascal based languages such as Ada and Modula3.

The fact that Oberon-2 is a production language is easily supported. Oberon-2 is an extension of the earlier language Oberon. An operating system and compiler were written in Oberon. The source code is given in "Project Oberon, The Design of an Operating System and Compiler" by Niklaus Wirth and Jurg Gutknecht, 1992. The use of the operating system is documented in "The Oberon System, User Guide and Programming Manual" by Martin Reiser 1991. Some facilities needed for a production language that are part of the Oberon-2 language definition are omitted from the oo2c implementation. Namely, some machine dependent parts of the module SYSTEM. This omission is no doubt because oo2c is implemented for automatic portable installation on a wide variety of systems. These facilities could be added by a sophisiticated user who needed them for his particular machine. An example would be type conversions which violate type compatiblity rules.

Programmers who would like to use Oberon, but are required to produce source code in C, should be aware of a an Oberon compiler that produces readable high level C code, namely Ofront, available at http://www.software-templ.com/shareware.html.

This article is based on the oo2c implementation of Oberon-2. See http://sourceforge.net/projects/ooc/. It runs on Intel chips, AMD chips, and other chips that support the operating systems that it runs under. It is available in both 32 bit and 64 bit versions. It runs under unix or linux, and presumably under MacOS and other unix based operating systems, but I have only tried it on linux. Instructions are given in a later section to add linux to the windows that is already on your hard drive. If you wonder whether linux is a good operating system, note that Microsoft, which sells Windows, runs their Bing search engine on linux, not on Windows or Mac OS. A much higher percentage of linux users do programming than the percentage of Windows users who do programming. Linux is an environment that appeals to programmers much more than Windows. If you wanted to write a program to sell to Windows users, you could get your program running in linux using the free oberon compiler recommended here and then use "ofront", described above, to translate it to C before compiling it on Windows with a windows C compiler. The linux environment is much better. That is why Microsoft used linux for their search engine. Decisions were made in the design of windows that makes it easier for a non-programmer to use. Decisions were made in the design of linux that makes it easier for a programmer to use. Windows and most of the software that is used on windows costs money. Linux and most of the software used on linux is free. If you have a windows computer, a bit later a description will be given of how to install linux for free on your windows computer. It should work under MacIntosh OS X since MacOS is also a clone of unix like linux is, but I have no experience installing it under MacOS. I have only used it on linux. The oo2c compiler is supplied as source code, and is compiled on your system using your C compiler during installation. The installation is mostly automatic, requiring little on your part. The oo2c implementation produces unreadable low level C code, then calls the C compiler to produce native code which executes about as fast as any other native code. For other implementations of oberon see www.modulaware.com/zel/oberon/osci.htm . For comparisons with C see http://www.modulaware.com/zel/oberon/fromctoo.htm . Another important site is www.oberon.ethz.ch.

This article that you are now reading must be supplemented by a definition of the Oberon-2 language. Instruction is available free online. A textbook for the original Oberon language is at www.inf.ethz.ch/personal/wirth/ProgInOberon.pdf. Another text is at http://www.excelsior-usa.com/doc/xds/o2rep.html . These texts only describe the language, not the library. The language must be supplemented by an Oberon library to be useable. The oo2c library manual "OOCRef" is freely downloadable from the sourceforge link in a previous paragraph. These detailed references may be too lengthy for the impatient. This article is brief, but complete enough to convey the basic nature of the language and the library. It shows working examples to solve the most common problems the beginner will face writing programs for text, graphics and mathematical applications. Care has been taken to make each example as brief and clear as possible. All of the source code presented in this article is in the public domain, and may be used without attribution, except for "cxarith.Mod" which is issued under the GNU public license.

The author wishes to acknowledge help from Michael Van Acken of the oo2c project when the author was learning some of the tricks disclosed herein.

In 2007 Wirth, in keeping with his philosophy of software engineering, proposed an improved version of oberon with minor changes called oberon07. Unfortunately, the only compiler available is for the ARM chip at projectoberon.com. Perhaps another version will someday become available. Certainly any new compiler written after this point in time should be in accordance with the new standard. Anyone who feels inclined to write a new compiler should consider Wirth's book "Compiler Construction". The oo2c compiler uses the additional compiler technology described in "Optimizing Compilers for Structured Programming Languages" by Marc Michael Brandis, ETH Zurich dissertation number th11024.ps. It can be downloaded free from http://www.inf.ethz.ch/de/forschung.html Perhaps someone should consider writing an Oberon compiler to the Java virtual machine.

INSTALLING LINUX IN ADDITION TO WINDOWS

This course has been developed on Linux. Linux can be obtained for free, so it does not cost anything to try it. People who would like to try linux, but do not have Linux, should know that it is possible to have both Windows and Linux on the same computer with a single hard drive. It used to be hard to do this kind of installation, but now it is easy. Here is how I did it. Try it at your own risk. If you make a mistake, just start over and try again. Backup your personal files onto a USB stick or blank CD, because they may not be safe during the installation procedure.

I use a "composition book", which is a notebook 9.75 inches by 7.5 inches with pages blank except for horizontal lines to write on. Something like this should be available in the school supplies section of your local grocery store. During this procedure write down everything that you do as you do it. If something goes wrong in this complicated procedure you will know where you need to do better the next time you try it. After the installation of linux, document every additional piece of software that you install. This will serve as a logbook for your computer. Leave the first 10 pages of the book blank for a table of contents that you will add later. Put a page number in the corner of each page as you fill the book up. It is a good idea to write the date the first time you write in the book each day.

Go to www.whatismyscreenresolution.com and write down your screen resolution. This number will be useful at the end to check whether you did the installation correctly.

Debian linux can be installed over the net starting with a "netinst" CD that you download and burn yourself. Or, you can get a set of CD's from a vendor listed on the Debian website and install it entirely from CD's. What follows is the installation procedure using the netinst CD. This works best if you have a high speed ethernet connection. Two alternatives for installation are presented, so you should read the whole thing before you start lest you get the two alternatives mixed up. If you have a new netbook computer, the ethernet chip may not be compatible with linux, in which case you need special instructions that you can find if you click here. If you have an older, tiny netbook, your ethernet chip will work but your disk space will be so small that you will need special instructions found at the end of this section.

The linux installation sofware is in an iso file. Windows does not have the capability to write an iso file to a CD in the correct manner. This capability can be added. An iso file could be written like any other file, or it can be written in a special way that iso files are intended to be written, which is what you want. If it is written like an ordinary file, you will see the iso file on the CD. If it is written correctly you will see the files that were contained in the iso file. Go to http://isorecorder.alexfeinman.com/ and be very careful to click on the version for your particular version of windows. When asked to run or save click run. Pop up windows will ask if you really want to do this, you do. When installation is finished reboot to make sure it takes effect.

Your Windows files may be scattered all over the disk. You need to tidy up your disk and move all Windows files to a neat group at the beginning of the disk, so that the last half of the disk will be free to install linux. You must "defragment" your disk. To do that, when in Windows XP, click on "my computer", right click on C drive, click on "properties", "tools", "defragment now". Defragmentation may take a while.

Go to debian.org, download the free 180 Mb "netinst" iso file. If your PC runs 32 bit windows, you will need the "i386"version. If your PC runs 64 bit windows, you will need the "AMD64" version even if your PC has an Intel chip, not an AMD chip. Click on the appropriate link in the "small CDs" section at www.debian.org/distrib/netinst. Download the iso file and make note of where it is downloaded on your computer so you can find it. Insert a blank CD into your machine to write the iso file onto. The following detailed proceedure was performed on Windows 7, but will probably be similar on other versions of Windows. Go to where the iso file is downloaded and right click on the iso file. In the popup window click on "open

with" then "ISO recorder" then "next". After the iso file is burned to the CD, the CD may eject. Push the button to put the CD back in, then shutdown and restart the computer. You should see the Debian install screen when the computer restarts. There is no risk to your system to go this far, as nothing is written to your disk until the installation process gets to the "partitioning" phase and you deliberately start the writing process.

A detailed example of installing windows, linux and oberon is given at this link. But you might want to finish reading this section first as an introduction.

The easiest way to install linux on an existing windows system is to reduce the size of the windows partition during the linux installation and install linux in the new free space. The first time you reboot after this, windows will insist on doing a file system check and reboot. This will require two reboots of your windows system to get windows working normally again. But suppose you are afraid that if you mess up, you might ruin windows, not get linux installed, and not have a working computer? This is not likely, but you should know how to install windows from scratch. The next paragraph describes the alternative of installing windows from scratch instead of simply reducing the size of the windows partition. You may feel safer knowing how to do this, so here it is.

The alternative is to do a fresh install of windows followed by a linux installation. If you want to do a fresh windows installation, find your windows CD and the associated product key number, which you must have to install windows. You will also need any device driver CD's that were supplied when you bought your computer. Put in your windows CD and shut down the computer with the windows CD in the drive. Turn on the computer with the windows CD in the drive. When it says "press any key", do so quickly, or it will not do the installation, it will just boot back into windows. You will be prompted with questions to install windows. You can accept defaults except when it asks if you want to install windows on the whole disk. You do not. It will tell you the size of the hard disk. Enter a number about half the size of the disk. You will have to shutdown and reboot a time or two to complete the windows installation. After the basic installation insert the driver CD's to bring your system up to its full functionality. If you do not have a driver CD for some part of your system, you can probably find a free download on the net. Save it in "my documents", then double click on it to install the driver. Remember to activate your windows system after you finish installing it. This is the end of the windows installation, if you decided to do this instead of just defragmenting windows and using the linux installation to reduce the size of the windows partition.

When you are ready for the linux installation put in the linux net install CD and shut down the computer. Windows should be fully up and running and you should do a normal shutdown. If Windows is only partly powered up when the power is turned off, the file system may be left in a condition where the following procedure will not work. The computer must be connected to the internet, as most of the software is not on the CD, and will be downloaded from the internet. You may not need driver CD's for linux, because most of the necessary drivers will be downloaded automatically. If you have a new model of graphics card, it will work with the default driver in linux, but for best results you may have to go to the manufacturers website to find a free linux driver to download for it. Turn on the computer with the linux CD in the drive. Select "install". During the installation process you will be asked questions. Use the up/down arrow keys to move the cursor vertically, the tab key to move horizontally, the enter key to accept a choice and the spacebar to select a choice. You will need to enter your country, your name, and make up your user name root name and two passwords. Root user is also sometimes called superuser. You can accept defaults to all the other questions except where to put linux. You do not want to put linux on the whole drive, put it on an empty partition. If you have done a fresh install of windows there is already an empty partition.

If the original windows installation still occupies the whole disk, you will need to reduce the windows partition. There may be two or three

partitions already on the disk. Windows will be in an "ntfs" partition. If there is a small one and a large one, Windows will be in the large one. When you get to the partitioning part of the linux installation, select "manual" partitioning, use the down arrow key to highlight the windows "ntfs" partition, hit "enter", select "resize partition", enter "50%" for the new size. Then "write previous changes to disk and continue"--yes. This will reduce the size of the ntfs partition and create a free partition of equal size. If you get an error message and the installer cannot resize the partition, you may have not done a normal shutdown of Windows before the installation.

Once you have a free partition after the ntfs partition, select "guided partitioning", then "use the largest continuous free space", and "all files in one partition", "write to disk". When you get to task selection, use the spacebar to select "Debian desktop" and "standard system". If you have a printer connected to your computer "print server" may also be checked, select that too. Select "continue". On my DSL internet connection it took one hour and ten minutes to download and install the software. Yours might take twice this long or half this long depending on the speed of your connection. At the end it asks if you want to install "grub", the answer is "yes".

Now both windows and linux are on the computer. When you start up the computer you have a few seconds to choose between linux, linux single user mode, or windows to boot up. If you do not choose, linux will boot up. In /etc/default/grub if GRUB_DEFAULT=0 then linux boots first, if GRUB_DEFAULT=2 then windows boots first. If you change this, run update-grub afterwards.

If you reduced the size of the windows partition without re-installing windows, the first time you boot into windows it will want to do a file system check, which you should let it do. It should only do this one time. If it asks for a file system check every time you boot into windows, something is wrong. To fix it, put the windows CD in the drive and boot into the CD. DO NOT select "install", instead select "repair the system". This should fix the problem.

In linux you will see "activities" in the upper left corner of the screen. If you click on that you will see a column of icons to start popular programs. The most important icon to notice is a circular lifesaver icon that gives access to information to help you configure your system. For computer programming you will need to add a terminal icon to that list, and you will need to learn how to use the linux terminal. To do both these things click here on how to use the linux terminal.

If you have a printer connected, you need to configure the software to talk to it. Click on "activities" "applications", scroll down to "system settings", click on this then click on "printers". Click on "unlock", enter root password. Click on "add new printer". After the software has time to find your printer, the name of your printer will appear highlited in blue. Click on "add". Click on "print test page". Click on "default" to make this the default printer. Click on "x" to fininsh.

To set up your email, click on "activities", click on the envelope icon named "evolution". The way evolution works has been changed in this new version of debian linux, "wheezy". Click "continue" and answer the questions until finished. The first time you try to get your email, you will be promted for your email password, as you would expect. But you will also be promted for a keyring password. It worked best for me when I used a new password, not either of the two already used. To test your email send a test message to your own address. Then click "send/recieve" button to recieve the message you sent yourself.

In a terminal window use the "df" command and write down how much of the linux partition is used by the basic linux installation you have just completed. Now that you are finsished with the linux installation, you should go to debian.org, click on "debian packages", "view package lists", "view packages in the stable distribution", and look through all the free software available from debian. If you are curious what additional software I added, my list is: g++, make, libtool, libgc-dev, dcraw, mmv, xfig, xfig-doc, tidy, gv, zip, vorbis-tools, cdrskin, geomview, texlive-latex-base,

remind, sox, ftp-ssl, xorriso, wodim, cdparanoia and webalizer. This long list only increased the software on my disk by 13%. The "df" command shows that my disk is still only 2% full.

To add new software click on "activities", "applications", scroll down to "synaptic". Alternatively, open a terminal, enter "su" and your root password, and enter the command "synaptic".

If you have an old computer or a tiny netbook with small disk space there will not be enough disk space for both Windows and linux, and you will have to install linux alone to run oo2c. You will need a separate plug-in USB CD drive to install linux on a tiny netbook. For instance, on a computer with only 4Gb of disk space, a full linux installation with oo2c will occupy 95% of the available disk space as shown by the command "df". Therefore you will need to do a skimpy linux install to save space. Do not select "gnome desktop" when you get to that step. Continue the rest of the installation as described above. When the installation is complete, you will have only a black screen with white text. To fix this, as superuser "apt-get install gnome-core gdm synaptic". When this is finished, shut down as superuser by "shutdown -h now". Upon reboot you will have a blue screen with icons. Open a terminal window, as superuser enter "synaptic", and search for and get epiphany-browser, unzip, vim and ghostview, which will appear in the list simply as gv. Now you will have a browser. After installing oo2c as described below, df will show only 51% of disk space used, but you will not have wifi or openoffice, just ethernet internet capability. Other useful programs that do not take up much space that you might want to install are xfig, xfig-doc, imagemagick, mmv and htmldoc, which will bring the total up to 53%.

INSTALLING THE COMPILER

If you are new to linux, read the article how to use linux terminal.

To see if your linux system has the software already installed that is needed to support oo2c, use the "whereis" command to see that you have the programs "g++", "gc", "make" and "libtool". Thus "whereis gc" might typically show "gc: /usr/bin/gc", which shows that you do indeed have gc somewhere on your computer. If it just shows "gc:", then you do not have it. These can be installed by their names except for "gc". When you go to install gc, you will have to ask for "libgc-dev", but after installation, to find it with "whereis" you must ask for simply "gc".

Make a directory called "prog". "cd prog" to get in the directory prog. In the directory "prog", "mkdir sym obj bin src" to make four new directories under prog. Go to the sourceforge website listed above and download the oo2c software, oo2c_32-2.1.11.tar.bz2. This is the 32 bit version, indicated by 32 in the name. If you have a 64 bit computer you will need the 64 bit version. You will have to do a lot of extra clicking on the sourceforge site to find the 64 bit version, but it is there. Click on "browse all files", then "oo2c", then "2.1.11" if that is the latest version, then you will see a choice of either the 64 bit or 32 bit versions. It will probably initially download into ~/Downloads directory, but you should move it into the directory "prog". "tar xvfj oo*" to unzip the software. You will now, in addition to the tar file, have a new directory oo2c_32-2.1.11 . You now have both files and directories in your prog directory. To see the difference, do "ls -aF". Now "cd oo*" to get into the directory oo2c_32-2.1.11 . If you are running MacOS you will need to read additional instructions found in this directory in the file README.MACOS. In this directory do "./configure ", and some text will quickly scroll by. When this is done, you will need to use the "su" command and enter the password to become root or superuser, and the prompt symbol will change from "$" to "#". As superuser do "make install". Lots of text will scroll by over a period of minutes, and your computer fan may speed up. Next, still as superuser, do "ldconfig", then "exit" so you are no longer superuser. In addition to any software already there, oo2c files will be installed in /usr/local/bin and /usr/local/lib.

In May 2013 Debian converted to the new version, "wheezy". It has a bug in the libgc-dev package that prevents the "make install" command

mentioned in the previous paragraph from being successful. When the text stops scrolling by you will see in the last few lines the error message "GC_PTR undeclared". This problem may be fixed at Debian by the time you read this, but if not, here is how you can easily fix the problem. In the directory "/usr/include/gc" the file "gc.h" is defective. You need to move it to "gchorig" to get it out of the way, and replace it with a correct gc.h that you can download by cliking here. You do not want to open the file, you want to save it. It will save in your "Downloads" directory that you can see if you "ls" in a terminal window in your home directory. You will need to "su" and enter your superuser password to make changes in the /usr/include/gc directory. In that directory "mv gc.h gchorig" to rename the original file, and "mv /home/yourname/Downloads/gc.h ." to replace it with the one you just downloaded. The final period "." before the end of the command means move the file to where I am right now, in /usr/include/gc. "yourname" is whatever name you call your home directory. Now start over and do configure, make install and ldconfig.

Do not delete the directory oo2c_32-2.1.11, because it documents the changes from version 1 to version 2. Inside that directory important documentation is found in lib/oocdoc/html and in doc/from-v1-to-v2. The version 1 manual , OOCref-000229.tar.gz, should also be downloaded and unpacked with "tar xvzf OO*". Enter "man oo2c" to see the man page for oo2c.

The directories "src" and "bin" are new additions with version 2 of oo2c; version 1 did not need them. With version 2, if you are in your directory "prog", or whatever you choose to call it, you will put your program source text files in "src". To use the vi editor to edit the program "tiny.Mod" you would enter "vi src/tiny.Mod". After compilation, to execute the program you would enter "bin/tiny".

SYNTAX

From this point on, many small programs will be presented as examples. In the beginning you will not understand why each program is written the way it is and not some other way. But you should at least be able to read each program to get some understanding of why it does what it does, and does not do something else. I will usually provide some explanation, but my explanations are only hints; the important thing is to read the program. Each program is to some extent self explanatory, and you must read each program carefully to understand as much as you can about how it works. Later, when you want to write your own program, you will have some notion of how to do it, and will know what details you need to look up in the more detailed and tedious descriptions of the language that are linked to in the introduction.

If you have never read programs before, it will be distressing and distasteful to read them. In the beginning you may be able to make sense of only half of what you read. But you will never be a programmer if you never become comfortable reading programs, and you will never become comfortable reading programs unless you force yourself to read some programs. You should read in detail every example in this article except "getitem", "graf2", "obelisk", "beep" and "cxarith", which you only need to read if you are going to do something similar. The rest of the examples illustrate things that every programmer should know.

You will need to learn the vi editor to copy programs from this article and to write your own programs. To learn the vi editor click here.

The best way to get the programs and files from this article is to save the html source file to disk complete with embedded html commands. If you save the html source of this web page to your disk, you can use the vi editor on your Linux system to copy the programs out of this document so that you do not have to copy them by hand. In vi, in the edit mode, not the input mode, put the cursor at the beginning of a program and type ":nu". This will show the line number in this document. Do the same for the end of the program. Suppose the name of the program is "tiny", the beginning was 201 and the end was 208. Then type ":201,208 w! tiny.Mod". You will then have a copy in a file "tiny.Mod" that you can compile and run. Unfortunately, some symbols, namely "<", ">" and "&", are not the same in html as in plain text. They are all represented in the plain text as

code words beginning with "&". These symbols appear correct when viewed with your browser, but not in the original file. Therefore, it will be necessary to compare a few programs with the way they appear in your browser to see what changes are necessary so that they will compile correctly. The lines of program text where this is a problem are marked with comments in the programs. If you use the vi editor to search for the ampersand character, "&" you will find all the lines that need to be changed.

An Oberon program consists of the word MODULE, then the name you choose to call the program followed by declarations, if any, of special things you wish to use in the program, then BEGIN followed by the program instructions, then END followed by the name of the program and a period. All words that are a permanent part of Oberon-2 must be capitalized; traditionally user defined words are in lower case to distinguish them from Oberon-2 words. This capitalization would be a nuisance if you had to do it as you write the program. The programming hints section of this document shows how to automate capitalization after you have written the program in lower case. The words in the oo2c library have the first letter of every word capitalized. Thus, while LONGREAL is an Oberon word, and LongReal is a library word, you could still use longreal as a user defined word if you wanted to.

The simplest program which does nothing is:

```
MODULE nothing;
BEGIN
END nothing.
```

This or any other program is broken into lines as needed for readability; it would work the same if it were all on one line. Assume this program is in a file named nothing.Mod. The file name must end in ".Mod". When you are in the directory "prog" and the program is in the directory "prog/src" it is compiled to check for errors by:

oo2c --error-style char-pos nothing | ooef | more

This is a lot to type. To reduce the typing, use the vi editor to create a script named "ooc" as follows:

```
oo2c --error-style char-pos $1 | ooef | more
```

Then "chmod +x ooc" to make the script exececutable. Then type "./ooc nothing" and achieve the same result with less typing. To test this, put the letter "a" between BEGIN and END, and you will get an error message. Do NOT make the mistake of typing "./ooc nothing.Mod", leave off the "Mod" when compiling. If you become superuser and move the ooc script to /usr/local/bin, then the leading "./" is no longer necessary and you can just enter "ooc nothing".

If no errors, it could also be compiled to produce an executable program by:

oo2c -M nothing

It is then executed by:

bin/nothing

and nothing happens because the program does nothing. Note that we execute the program by "bin/nothing" because we are in the directory above "bin". If we were in the directory "bin" we would execute the program by "./nothing".

It is useful to have the script "ooc" shown above and an additional script "oocm" which shows errors and produces an executable:

```
oo2c -M --error-style char-pos $1 | ooef | more
```

The next example in this section will be to compute the number of meters and the number of yards around the inside lane of a quarter mile oval running track. We use the facts that one meter is 100 centimeters, one inch is exactly 2.54 centimeters, one foot is 12 inches, one yard is 3 feet, and one mile is 5280 feet. A nautical mile is different than a statute mile; here we are talking about a statute mile. We start with one centimeter, and build up our units of measure from there.

```
MODULE mile;
IMPORT Out;
VAR mi,qmi,cm,in,ft,yd,m:REAL;
BEGIN
cm:=1.0;
m:=100*cm;
in:=2.54*cm;
ft:=12*in;
yd:=3*ft;
mi:=5280*ft;
qmi:=mi/4.0;
Out.String('a quarter mile is:');Out.Ln;
Out.RealEng(qmi/m,20,6);Out.String(' meters');Out.Ln;
Out.RealEng(qmi/yd,20,6);Out.String(' yards');Out.Ln;
END mile.

a quarter mile is:
             402.336 meters
             440.000 yards
```

Now what have we done in this program? We decided what variables we needed to do the work, and we declared them before BEGIN. We arbitrarily chose names for the variables that made it easy to remember what they represented. Then we described the work to be done with a series of statements, ending with clear output statements. Then, not shown here, but as described in previous paragraphs, we run the program through the oo2c compiler to create the executable version of the program, then execute it to get the results. In this example the work used numbers. In some other example it might use characters, bits, or a user defined type such as complex numbers, personnel records, coordinates on a diagram, lines to be drawn or whatever types you can devise using the built-in types as building blocks.

It should be clear from the example of the quarter mile track that you cannot program the computer to do anything unless you know how to do the thing yourself. Often, you will not know everything about how to do the thing you want to do when you start, and will learn and figure things out as you write the program. Some people relish the challenge. One said "what good is a project if you don't learn something new?" To help you

figure things out you can search the internet, there are free courses online and books in libraries. For more information click here. Or perhaps an employer will explain how to do something that he wants you to write a program to accomplish.

What kind of programs can you write? Anything a computer can do. Do simple calculations like the previous example. Input a person's measurements and print out patterns for custom clothing. Simulate the sounds of musical instruments. More different things than any one person can imagine.

The declaration headings in a program are optional, and not needed unless they are actually going to be used to declare something. A very simple program which uses some declarations is:

```
MODULE tiny;
IMPORT Out;
CONST a=3;
VAR b,c:LONGINT;
BEGIN
b:=2; c:=a+b;
Out.LongInt(c,4);Out.Ln;
END tiny.

   5
```

The declarations before BEGIN are information needed to execute the statements after BEGIN, which is where execution of the program starts. Each statement ends with a semicolon ";". Any place where one statement can go, any number of statements can be put there, one after the other. The names of variables and constants in this program are a, b, and c. These user defined names must have a letter, not a number, as their first character, but they can be many characters long.

In this program the "=" symbol when used by itself represents a definition, a=3. Thus "a" is a constant and can never change its value. The mathematical constant pi=3.14159 would be a good candidate for this kind of treatment. If you get confused in writing a large program and try to change the value of a constant, the compiler will warn you with a compilation error.

The combination symbol ":=" means a variable is assigned a new value. As an example, if your bank account is "b" and you write a check for ten dollars thus reducing your bank account by ten dollars, you would represent this by the statement in English "b becomes b minus ten" or in Oberon, "b:=b-10.00;". The right side of this expression means "copy the contents of the memory location called b and subtract 10.00 from the copy"; the left side means "put the result in the memory location called b". In this case you would define your bank account to be a REAL variable, not LONGINT, so that it could contain decimal fractions of a dollar.

The "5" after the program is the output of the program when it is executed. Since there is no I/O (input/output) in the Oberon language, the external oo2c library module "Out" must be imported to give simple output capability. For fancier I/O capability, there are other modules described in the oo2c manual. These modules are part of the oo2c library, and not part of Oberon itself. You are free to write your own I/O modules if you are not happy with the ones supplied.

"a" was declared as a constant just to illustrate that constants can be declared. The variables "b" and "c" were declared as LONGINT rather than

INTEGER because most computers have 32 bit words which match the size of LONGINT. While any type supported on a machine can be used on a machine, types that match the machine word can be expected to compute faster than other types.

The number "4" in the Out.LongInt statement allowed four spaces for the number to be printed out in. Out.Ln moved output to the next line.

Computers represent numbers internally as binary numbers. Decimal numbers use the digits 0,1,2,3,4,5,6,7,8,9. Binary numbers only use the digits 0,1. Even though computer hardware uses binary numbers, computer programs are usually have input and output expressed in decimal numbers. The software translates between decimal numbers for humans and binary numbers for the computer. The decimal number 111 means ten to the second power, 100, plus ten to the first power, 10, plus ten to the zeroth power, 1, or one hundred and eleven. The binary number 111 means two to the second power, 4, plus two to the first power, 2, plus two to the zeroth power, 1, or seven. The binary number 111 is expressed as three binary bits, and is the largest possible 3 bit number. The largest 7 bit binary number is 2 to the 7th power minus one, or decimal 127, and the largest 15 bit binary number is decimal 32767. An 8 bit binary number is usually a 7 bit number plus a sign bit, in "two's complement" form. A 16 bit binary number is usually a 15 bit number plus a sign bit.

An integer number has no fractional part, like decimal 37. A real number can have a fractional part like decimal 37.64332. If a real number happens to have no fractional part, like 37.000, it is still represented in the software as a real number. An integer number is represented differently from a real number. An integer is stored as a signed number. A real is stored as a signed number and a signed exponent. A real is stored as binary but in decimal an example would be 2.775 times an exponent term of ten to the minus seventh power. This could be written as 2.775E-7. A real 32 bit number would typically be 24 bits representing a 23 bit signed number, and 8 bits representing a 7 bit signed exponent. The number 2 raised to the power 127 is decimal 1.7... times ten to the 38th power. The exponent can be from 10 to the plus 38th to ten to the minus 38th. The binary number 1.11111... is nearly decimal 2.0. The binary number 1.0000... is decimal 1.0. The 23 bits are the fractional part of the number. The "1" to the left of the decimal point is implied, and not actually stored. This scheme can represent any number from 10 to the plus 38 to 10 to the minus 38th. It cannot represent the number 0, which is handled differently as a special case, where the implied "1" is not implied. The maximum value of a 32 bit real number is 3.40282347E+38. If you want to see a website that will convert any decimal number to its floating point binary equivalent see http://babbage.cs.qc.cuny.edu/IEEE-754/ .

For the sake of completeness, we should mention ultra-reliable memory, called error correcting code memory or ECC memory. This is available only on 64 bit computers. Presently it is only available on a class of computers called workstations. It is not standard on workstations, but is available at extra cost. ECC memory uses 72 bit words in hardware that look like 64 bit words in software. Instead of being stored as a 64 bit word, the data is stored as a 72 bit code that represents the 64 bit word. If only one of any of the 72 bits fails, hardware logic automatically corrects the error and the software sees only a correct 64 bit word. The error correcting logic hardware is not in the memory, it is between the 72 bit memory and the 64 bit central processing unit or CPU. Memory bits are so tiny there is a remote possibility that a memory bit could fail temporarily because of a charged cosmic ray particle. Or it could fail permanently because of a hardware failure.

Oberon has integer types SHORTINT, 8 bits, INTEGER, 16 bits, LONGINT, 32 bits, and, on 64 bit machines only, and only with the oo2c implementation, HUGEINT, 64 bits. On both 32 bit and 64 bit machines it has real types REAL, 32 bits, and LONGREAL, 64 bits. It has type conversion functions SHORT and LONG to convert to the next larger or smaller version of integer or real. Real division is expressed by the symbol "/", integer division by "DIV". Add, subtract and multiply are represented by "+", "-" and "*" for both real and integer types. While raising a number to a REAL power is not directly provided in the language, it can be done with a function provided in the RealMath library module. REAL can have the fractional part removed to make the next smaller integer by the function ENTIER. The function ABS gives the absolute value of any real or

integer type. Integer "i" can be converted to real "r" by "r:=i". SET variables are a group of bits equal in number either to the size of a machine word or to the size of word written or read by the hardware to the hard disk. In oo2c the SET size is 32 bits for both 64 bit and for 32 bit computers, presumably because the same the same hardware interface to the hard drive is typically used in both cases. A CHAR variable takes on the value of a character, such as "a", "2", "&", etc. CHAR variables are represented in binary by the 8 bit ascii character code. The functions ORD and CHR convert between ascii characters and integers. A BOOLEAN variable takes on the values TRUE or FALSE.

Complicated statements can be simplified by the use of parentheses. Thus "x:=(a*b)/(c+d);" means that the computations in each pair of parentheses "(...)" will be done before the "/" outside of the parentheses. Parentheses can be nested "(...(...))" to handle even more complexity, in which case the inner parentheses is computed before the outer.

We end this section with one final example of a simple calculation. I used to run on a one mile track, and knew how fast I could run a mile. Now where I live the most convenient place to run is 1.44 miles, but I would still like to know how many minutes per mile I run. At the end of my run, my stopwatch tells how many minutes and seconds it took to run 1.44 miles. The ENTIER function was described in a previous paragraph. There are 60 seconds in a minute. The following program tells how many minutes and seconds it took me to run one mile:

```
MODULE run;
IMPORT In,Out;
CONST dist=1.44;
VAR minin,secin,sectot,secpmi,minpmi:REAL;
minout,secout:LONGINT;
BEGIN
Out.String('enter minin, secin');Out.Ln;
In.Real(minin);In.Real(secin);
sectot:=60*minin+secin;
secpmi:=sectot/dist;
minpmi:=secpmi/60;
minout:=ENTIER(minpmi);
secout:=ENTIER(secpmi-minout*60);
Out.String('run time per mile');Out.LongInt(minout,3);
Out.String(' minutes ');Out.LongInt(secout,3);
Out.String(' seconds');Out.Ln;
END run.

enter minin, secin
14 45
run time per mile 10 minutes  14 seconds
```

The line "14 45" was entered from the keyboard in response to the line before it that asked for data that the program needed to do its job.

STRUCTURE

The human mind can only be comfortable with small programs. Large programs must be built up from many small programs to be

comprehensible. This section explains how these small programs communicate with each other to form a large program. An important feature of modern computer languages is an emphasis on hierarchical structure of the program as achieved by internal procedures and functions. These procedures and functions are almost small programs within the program. The motivation behind intensive use of procedures is readability, understandability and modifiability of the program. The perfectly valid Oberon statement "REPEAT stirring UNTIL thick();" sounds like plain English. It is valid if "stirring" is the name we gave to a procedure and "thick" is the name we gave to a boolean function. A major part of writing a good program is figuring out the best way to subdivide the program and the best names to call procedures, functions and variables.

The variables that a procedure uses to communicate with the rest of the program are called its parameters. There are different kinds of parameters, namely global, value and variable parameters. In Wirth's terminology, a "variable" is a memory location or machine address where a "value" or data can be stored. If you pass a value parameter to a procedure you are giving it some data to work with in one of its own internal memory locations. If you pass a variable parameter to a procedure you are giving it the address or memory location of a variable outside the procedure.

The simplest case is the global parameter:

```
MODULE proc1;
IMPORT Out;
VAR a:LONGINT;

PROCEDURE add1; BEGIN a:=a+1;END add1;

BEGIN a:=1; add1; Out.LongInt(a,4);Out.Ln;END proc1.

   2
```

In the program proc1 there is no declaration of the variable "a" in the procedure so "a" is a global parameter. The "a" in the procedure is the same "a" as the "a" outside the procedure. This is the same as the GOSUB..RETURN of the primitive programming language BASIC except that now names, rather than line numbers, can be used to refer to the procedures for enhanced readability.

Next we consider variable parameters:

```
MODULE proc2;
IMPORT Out; VAR x,y:LONGINT;

PROCEDURE add1(VAR a:LONGINT);
BEGIN a:=a+1; END add1;

BEGIN x:=10;y:=20;add1(x);add1(y);
Out.LongInt(x,3);Out.LongInt(y,3);Out.Ln;END proc2.

 11 21
```

The appearance of the variable "a" in the parameter list of the procedure constitutes a declaration, as opposed to a use, of the variable "a". Since "a" is declared in the procedure it is not a global parameter. Notice that the procedure add1 was called twice in the main program and given different variables to work on each time. The statement a:=a+1 actually went to the address "x" and did x:=x+1, then went to "y" and did y:=y+1. There is no address "a". This is the same as the parameters of a FORTRAN subroutine in early versions of the programming language FORTRAN.

```
MODULE proc3;
IMPORT Out; VAR x,y:LONGINT;

PROCEDURE add1(a:LONGINT; VAR b:LONGINT);
BEGIN b:=a+1; END add1;

BEGIN x:=1;add1(x,y);
Out.LongInt(x,4);Out.LongInt(y,4);Out.Ln;END proc3.

   1   2
```

In the above example "a" is a value parameter because it is not preceded by VAR in the parameter list. The call add1(x,y) accomplishes the implied assignment a:=x; then b:=a+1 in the procedure accomplishes y:=a+1. There actually is an address "a" in the procedure because "a" is a value parameter. CONST items can be passed into a procedure as value parameters, but not as VAR parameters.

Variables used in a procedure may be declared outside the procedure, in the procedure parameter list, or in the declaration section of the procedure. Parameters in the calling statement may use the same name as the procedure does for the parameters or different names. What follows is the simplest example that covers all these possibilities. You should painstakingly thread your way through the program to see how each number in the output was calculated. The notes that follow the program should help. Put your mind in low gear, this is the heart of the course. Note that anything enclosed within the pair (* *) is a comment for human readers, and is ignored by the compiler.

```
MODULE proc4;
IMPORT Out; VAR a,b,c,d,e,f:LONGINT;

PROCEDURE change(a,x:LONGINT;VAR c,y:LONGINT);
(*The stuff in parentheses on the line above is the
   parameter list.  *)
VAR f:LONGINT;
BEGIN
f:=60; a:=1+a;x:=1+x;c:=1+c;y:=1+y;e:=1+e;f:=f+1;
Out.String('in procedure:');
Out.LongInt(a,3);Out.LongInt(x,3);Out.LongInt(c,3);
Out.LongInt(y,3);Out.LongInt(e,3);Out.LongInt(f,3);
Out.Ln;
END change;
```

```
BEGIN(*this is where the execution of the program
starts*)
a:=10;b:=20;c:=30;d:=40;e:=50;f:=70;
Out.String('in program-1:');Out.LongInt(a,3);
Out.LongInt(b,3);Out.LongInt(c,3);Out.LongInt(d,3);
Out.LongInt(e,3);Out.LongInt(f,3);Out.Ln;
change(a,b,c,d); (*this is the procedure call*)
Out.String('in program-2:');Out.LongInt(a,3);
Out.LongInt(b,3);Out.LongInt(c,3);Out.LongInt(d,3);
Out.LongInt(e,3);Out.LongInt(f,3);Out.Ln;
END proc4.

in program-1: 10 20 30 40 50 70
in procedure: 11 21 31 41 51 61
in program-2: 10 20 31 41 51 70
```

Notes:

1. The procedure named change is only called once in this example, but a procedure can be called as often as you like in different parts of a program. Different variables could be used each different place in the program where the procedure is called. Thus, a,b,c,d could be replaced by r,s,t,u somewhere else, just so long as the data type of the variables used in the call agrees with those in the procedure declaration.

2. The order of the variables in the procedure call and in the procedure declaration determines which variables in the call will be associated with which variables in the declaration. The names of the variables in the call have nothing whatever to do with the names of the variables in the procedure declaration.

3. The parameters (a,x) in the declaration are called "value parameters". They are not preceded by VAR. Value parameters are names of variables in the procedure. The calling statement transfers data to these variables just as if the two assignment statements a(*in procedure*):=a(*in program*); x:=b; had been executed. The calling statement can have an expression such as x+y in the position corresponding to a value parameter. An expression, once evaluated, has a value, and that is what is needed in a value parameter. In the example, after the data has been transferred, the statements a:=a+1 and x:=x+1 in the procedure are performed on the variables in the procedure, not on the ones in the calling statement.

4. The parameters (c,y) in the declaration are called "variable parameters". They follow the word VAR. The variable parameters (c,y), are dummy names for variables in the calling statement that will be operated on by the procedure. Thus the statement y:=y+1 in the procedure actually performs the operation d:=d+1 on the variable d in the calling statement. There is no actual variable y in the procedure, y is just what the procedure calls d. An expression such as x+y cannot be used in the calling statement in the position corresponding to a variable parameter in the parameter list.

5. The variable f is declared in the procedure and therefore has nothing whatever to do with the variable f in the main program. The variable e is used, but not declared in the procedure. Since it is not declared in the procedure, it is called a "global variable". It is the e in the main program.

6. The procedure has two value parameters and two variable parameters to illustrate that it makes no difference whether the variable name used in the calling statement is the same as or different from the corresponding variable in the declaration. Variables declared in the parameter list or in the body of the procedure are called "local variables" because it doesn't matter if their names happen to be the same as other variables outside the procedure.

7. If variable parameters of more than one data type are used in a parameter list, a separate VAR word is needed for each data type. Thus (VAR x,y:REAL;VAR i,j:INTEGER;VAR k,l:BOOLEAN).

Procedures can have procedures inside them. This is illustrated by the following example:

```
MODULE proc5;
IMPORT Out;
VAR a:LONGINT;

PROCEDURE add1;

PROCEDURE add6;
BEGIN(*add6*) a:=a+6;END add6;

BEGIN(*add1*) a:=a+1;add6;END add1;

BEGIN(*proc5*) a:=1;add1;Out.LongInt(a,4);Out.Ln;
END proc5.

   8
```

In the example proc5 the procedure add6 is part of the declaration section in procedure add1; it is nested inside add1. The procedure add6 can be called from add1, but it could not be called from proc5.

A function procedure is similar to an ordinary procedure except that it may take on a value. It can only take on a single value, not a set of values. It can take on any type of single value. Here we use LONGINT, but it could also be REAL, CHAR, BOOLEAN, etc. Since a function can take on a value it can be used in an expression:

```
MODULE fcn1;
IMPORT Out;
VAR x,y:LONGINT;

PROCEDURE add1(a:LONGINT):LONGINT;
BEGIN RETURN a+1;END add1;

BEGIN x:=0;y:=1+add1(x);Out.LongInt(y,4);
Out.Ln;END fcn1.
```

```
   2
```

In the main program add1 is a name to which a value will be assigned when add1 is called; add1 is given its value by the RETURN statement inside the function procedure add1. Here the function procedure had a parameter list in parentheses. Even if there is no parameter list, a function procedure, unlike other procedures, requires the parentheses.

If the memory location representing a variable resides in a procedure, as with value parameters or variables declared in the procedure, the variable cannot be depended on to retain its value between the time the procedure finishes executing and the next time the procedure is called. I could not construct an example which proved this with the oo2c compiler. Even if the oo2c compiler prevents this problem, other Oberon-2 compilers cannot be counted on to do likewise. Therefore do not rely on a procedure to "remember" the values of such variables between calls.

A somewhat esoteric example illustrates procedure types and procedure variables:

```
MODULE pv1;
IMPORT Out;
TYPE
pt=PROCEDURE(z:LONGINT):LONGINT;
VAR
p1,p2:pt;

PROCEDURE square(x:LONGINT):LONGINT;
BEGIN RETURN x*x;END square;

PROCEDURE cube(x:LONGINT):LONGINT;
BEGIN RETURN x*x*x;END cube;

PROCEDURE print(i:LONGINT;p:pt);
BEGIN
Out.LongInt(p(i),5);Out.Ln;
END print;

BEGIN
p1:=square;p2:=cube;
print(3,p1);print(3,cube);
END pv1.
    9
   27
```

This is our first example of a user defined type, defined in the TYPE section. The built-in types, such as REAL, INTEGER, BOOLEAN, are pre-defined and do not need a TYPE section. Here we have defined a procedure type which we have named "pt".

Procedure variables are needed when different procedures must be passed as a parameter to another procedure and called within the other procedure. It is most often used with function procedures which take on a value. If each function only needed to be called once, it would not be

necessary to pass the function as a parameter, just the value resulting from calling each function once could be passed. If, however, each function must be called many times within another procedure, it is necessary to pass the function as a parameter. Applications include procedures which plot graphs of different functions, and procedures which numerically integrate different functions. The above example did not need to call each function more than once, but it serves to illustrate the technique. In one case a variable whose value was the name of a procedure was used. In the other case the name of a procedure was used directly.

Occasionally you will want to use a procedure before it is declared. The word "before" needs to be clarified. The program execution starts at the last "begin" in the program. The compiler compiles the program starting at the top and going to the bottom. Thus a the compiler could find a procedure being used before it is declared, even though it is declared before the last begin where execution starts. To let the compiler know that the procedure will be declared after it is used, use a "forward" declaration, which is only the heading of the procedure, with the word PROCEDURE followed by "^" with no space. This is illustrated by the program fwd.

```
MODULE fwd;
IMPORT Out;
VAR x:LONGINT;

PROCEDURE^ add1(a:LONGINT;VAR b:LONGINT);

PROCEDURE useit(c:LONGINT);
VAR d:LONGINT;
BEGIN
add1(c,d);
Out.LongInt(d,4);Out.Ln;
END useit;

PROCEDURE add1(a:LONGINT;VAR b:LONGINT);
BEGIN
b:=a+1;
END add1;

BEGIN
x:=1;useit(x);
END fwd.
```

```
   2
```

We will end this section with a program that uses some of the ideas introduced in this section, and computes something interesting. We use the simplified actuator disk formula for the thrust of an airplane propeller that can be found in textbooks. We calculate the thrust of one propellor of the type found on a real airplane of the 1940's and 1950's. This program is included only to show a simple handy application of programming. You do not need to look up the formulas used to appreciate the interesting results. When we run the program we input the data that the propellor is 17

feet in diameter, the engine is running at a full power of 3500 horsepower, the speed of the plane is zero because it has not started its takeoff roll, and the altitude of the airport is 3200 feet above sea level. The result that the program gives is that the thrust of the propellor is 15190.93 pounds. This would be 7.5 tons of thrust just from fanning thin air. Neat huh? The speed of the air blown out the back of the propellor is 86.41 miles per hour.

```
MODULE prop2;
(*actuator disk McCormick p.73-76*)
IMPORT rm:=RealMath,In,Out;
CONST pi=3.14159;
VAR ro,a,d,rad,p,t,v0,w,q,r,s1,s2,x1,x2,x3,alt,sigma:REAL;
str:ARRAY 40 OF CHAR;

PROCEDURE pwr(x,y:REAL):REAL;
BEGIN RETURN rm.exp(y*rm.ln(x));END pwr;

PROCEDURE dens(alt:REAL;VAR sigma:REAL);
(*computes sigma as fraction of sealevel dens*)
BEGIN
IF alt<35332.0 THEN
sigma:=pwr((1.0-6.879E-6*alt),4.2358)
ELSE sigma:=0.3058*rm.exp(-4.778E-5*(alt-35332.0));END;END dens;

BEGIN
Out.String('enter d,p,v,alt');Out.Ln;
In.Real(d);In.Real(p);In.Real(v0);In.Real(alt);In.Line(str);
Out.String('dia(ft)=');Out.RealFix(d,4,1);
Out.String(' hp=');Out.RealFix(p,6,1);
Out.String('  plane v(mph)=');Out.RealFix(v0,6,1);
Out.String(' alt=');Out.RealFix(alt,4,1);Out.Ln;
d:=d*0.3048;p:=p*745.7;v0:=v0*0.447;
(*convert to meter kilogram second units*)
dens(alt,sigma);
Out.String('density=');Out.Real(sigma,0,0);Out.Ln;
ro:=1.18*sigma;
rad:=d/2.0;a:=pi*rad*rad;
(*cubic solution Abramowitz and Stegun p17*)
q:=-(1/9)*v0*v0;
r:=(1/27)*v0*v0*v0+0.25*p/(ro*a);
x1:=q*q*q; x2:=r*r;
Out.String('q^3+r^2=');Out.Real(x1+x2,0,0);Out.Ln;
x3:=rm.sqrt(x1+x2);
s1:=pwr((r+x3),1/3);
IF (r-x3)<0.001 THEN s2:=0.0 ELSE
```

```
s2:=pwr((r-x3),1/3);END;
w:=s1+s2-2*v0/3;
t:=ro*a*(v0+w)*2*w;
t:=t/4.448;(*newtons to pounds*)
Out.String('thrust(lbs)=');
Out.RealFix(t,8,2);
Out.String('  induced velocity at disk w(mph)=');
Out.RealFix(((w)/0.447),8,2);Out.Ln;
Out.String('ideal efficiency=');
Out.RealFix((1/(1+w/v0)),5,2);
Out.Ln;
END prop2.

bin/prop2
enter d,p,v,alt
17 3500 0 3200
dia(ft)=17.0 hp=3500.0  plane v(mph)=   0.0 alt=3200.0
density=9.10024703E-1
q^3+r^2=8.30308416E+8
thrust(lbs)=15190.93  induced velocity at disk w(mph)=   86.41
ideal efficiency= 0.00
```

Note that the symbol "^" was used to indicate an exponent in the printed output of the program, but that would not have been legal for computation in the program.

CONTROL

The next example shows illustrates the FOR statement which is used where a fixed number of iterations are needed.

```
MODULE for1;
IMPORT Out;
VAR i:LONGINT;
BEGIN
FOR i:=1 TO 9 DO Out.LongInt(i,2);END(*FOR*);
Out.String(' end');Out.Ln;
END for1.

 1 2 3 4 5 6 7 8 9 end
```

Between DO and END(*FOR*) we put only one statement, but any number of statements could go there. Other repetitive statements are "REPEAT...UNTIL...;", "WHILE...DO...END", and "LOOP...EXIT...END". These repetitive statments end when a BOOLEAN value becomes TRUE. This can

happen with a boolean variable that is set to TRUE or with an expression that evaluates to a boolean value, such as "i=5" or "x>3.2". ">" means "greater than", "<" means "less than" and "#" means "not equal to".

It is possible to write a program with a repetitive statement that goes on forever, and never ends. If this was a mistake and not your intention, you can halt the program by holding down the "ctrl" key and tapping the "c" key. This is called hitting "control-c" or "ctrl-c". An example of a bad program with a never ending loop follows. You should run it and stop it with control-c.

```
MODULE inf;
BEGIN
LOOP END;
END inf.
```

An example of a good program using REPEAT follows:

```
MODULE repeat1;
IMPORT Out;
CONST a=10;
VAR b:LONGINT;

PROCEDURE thick():BOOLEAN;
BEGIN
IF b>a THEN RETURN TRUE ELSE RETURN FALSE; END(*if*);
END thick;

PROCEDURE stirring;
BEGIN
b:=b+1; END stirring;

BEGIN
b:=0;
REPEAT stirring UNTIL thick();
Out.String('b=');Out.LongInt(b,3);Out.Ln;
END repeat1.

b= 11
```

The next example shows WHILE:

```
MODULE while1;
IMPORT Out;
VAR r:REAL;
BEGIN
r:=1.0;
```

```
WHILE r < 1.5 DO r:=1.1*r;END;
Out.String('r=');Out.Real(r,5,0);Out.Ln;
END while1.

r=1.61051011
```

The next example shows LOOP:

```
MODULE loop1;
IMPORT Out;
VAR i:LONGINT;
BEGIN
i:=0;
LOOP i:=i+1;
IF i>7 THEN EXIT END(*IF*);
i:=i*2;END(*LOOP*);
Out.String('i=');Out.LongInt(i,2);Out.Ln;
END loop1.

i=15
```

The next example declares meaningful names as numerical constants so they can be used to clarify the meaning of a CASE statement:

```
MODULE case1;
IMPORT Out;
CONST mon=1;tue=2;wed=3;thurs=4;fri=5;sat=6;sun=7;
VAR day:LONGINT;
BEGIN
FOR day:=mon TO sun DO
CASE day OF
fri:Out.String('tgif');Out.Ln;
|tue:Out.String('what a grind');Out.Ln;
|wed:Out.String('meetings all day');Out.Ln;
|mon:Out.String('not again!');Out.Ln;
|sat,sun:Out.String('zzzz...');Out.Ln;
|thurs:Out.String('big weekend ahead');Out.Ln;
END(*CASE*);END(*FOR*);END case1.

not again!
what a grind
meetings all day
big weekend ahead
tgif
```

```
zzzz...
zzzz...
```

In the above example note that the order in which the items are printed out corresponds to the order the days are executed in the FOR statement, which in turn was defined by the numerical values given to the days of the week. The order in which the items are printed out does not correspond to the order of their listing in the CASE statement. Note also that the extra work needed to provide names, not just numbers, for the case selectors has made the program more readable.

The IF statement is a simple basic control statement:

```
MODULE if1;
IMPORT Out;
VAR x,y:BOOLEAN;
BEGIN
x:=TRUE;y:=FALSE;
IF x=TRUE THEN
Out.String('x=TRUE');Out.Ln;END;
IF y=TRUE THEN
Out.String('y=TRUE');Out.Ln;END;
END if1.

x=TRUE
```

The IF THEN ELSE statement may be used in the form of nested ELSIFs to provide something similar to the CASE statement where the selectors are logical conditions rather than values of a variable. The following example illustrates this:

```
MODULE elsif1;
IMPORT Out;
VAR i:LONGINT;
BEGIN
FOR i:=1 TO 9 DO
IF i=1 THEN Out.String('C1');Out.LongInt(i,2);
Out.Ln;
(*the following two lines must be modified to look the way
they look when seen in your browser*)
ELSIF i>8 THEN Out.String('C2');Out.LongInt(i,2);
Out.Ln;
ELSIF i>7 THEN Out.String('C3');Out.LongInt(i,2);
Out.Ln;
ELSE Out.String('C4');Out.LongInt(i,2);Out.Ln;
END(*IF*);END(*FOR*);END elsif1.

C1 1
```

```
C4 2
C4 3
C4 4
C4 5
C4 6
C4 7
C3 8
C2 9
```

Our final example in this section illustrates the MOD and DIV operators, and the ABS function. The DIV and MOD operators work only on integer types. The ABS function works on both integer types and real types.

```
MODULE mod3;
IMPORT Out;
VAR i:LONGINT;
BEGIN
FOR i:=-9 TO 9 DO
Out.LongInt(i,2);END(*for*);Out.Ln;
FOR i:=-9 TO 9 DO
Out.LongInt((i MOD 3),2);END(*for*);Out.Ln;
FOR i:=-9 TO 9 DO
Out.LongInt((i DIV 3),2);END(*for*);Out.Ln;
FOR i:=-9 TO 9 DO
Out.LongInt(ABS(i),2);END(*for*);Out.Ln;
END mod3.
```

```
-9-8-7-6-5-4-3-2-1 0 1 2 3 4 5 6 7 8 9
 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0
-3-3-3-2-2-2-1-1-1 0 0 0 1 1 1 2 2 2 3
 9 8 7 6 5 4 3 2 1 0 1 2 3 4 5 6 7 8 9
```

DATA STRUCTURES

The following is a short example using an array:

```
MODULE array1;
IMPORT Out;
TYPE ara=ARRAY 5 OF LONGINT;
VAR i:LONGINT;a:ara;
BEGIN
a[0]:=6;a[1]:=7;a[2]:=8;a[3]:=9;
(*watch for uninitialized variables*)
FOR i:=0 TO 4  DO Out.LongInt(a[i],3);Out.Ln;END;
```

```
END array1.

  6
  7
  8
  9
  0
```

Notice that we failed to give a[4] a value, but we used it anyway. In this case the compiler was apparently set up to initialize all variables to zero. Most compilers cannot be counted on to do this. If not, a random number could have been printed out for a[4]. Using array elements that have not been initialized is perhaps the most common cause of errors in computer programs. Even in the case of an ordinary variable that is not part of an array, if it is declared, but then used without being intitialized, the oo2c compiler will flag it with an "undefined variable" warning.

Note especially that an array of size 5 has elements numbered from 0 to 4, not from 1 to 5. Thus, if you plan to access array elements from 1 to "n", you should declare the array to be of size "n+1".

Arrays of more than one dimension are also possible. An example of a large two dimensional array of real would be declared as "ara=ARRAY 100,100 OF REAL" and an element accessed as "a[57,34]".

Next we give an example of records. Records are a user defined type. They are defined in the TYPE section and used elsewhere in the program. A record is a way of combining items of different built-in and/or user defined data types into a single variable.

A simple example of a record definition:

```
CONST male=1;female=2;engineer=1;technician=2;
laborer=3; admin=4;management=5;
TYPE
employeerec=RECORD
            name:ARRAY 40 OF CHAR;
            salary:REAL;
            occupation,sex:LONGINT;
          END;
VAR employee:ARRAY 5000 OF employeerec;
```

To test the occupation of employee 323 we would say:

IF employee[323].occupation=engineer THEN (whatever)...

For an example of use of records see MODULE cxarith in the "mathematical programming" section of this document.

Records can be very complicated. They can be used to make some programs much simpler and more compact than they would otherwise be. They can also contain pointers, allowing the construction of linked lists, binary trees etc. There is no provision for file I/O of record types. Any such files would not be portable between different Oberon implementations. Instead, each field of the record type should be converted to ASCII text for

file I/O, then portability can be assured.

A final example is taken from a working program to show the complexity that is sometimes desired in data structures. The final record "fltrrec", combines reals, integers, booleans, and arrays of records. You will not understand the motivation behind it, but you should be able to see how the pieces fit together. The reason for the asterisks "*" will be explained later in the section on separate compilation.

```
IMPORT cx:=cxarith;
CONST maxpoles=20;
TYPE
polerec*=RECORD sigma*, omega*, gain*:REAL;
xk*,yk*,ykm1*,onemeps*,rt1meps*,eps*,gaincx:cx.complex;
first*:BOOLEAN; END;
zerorec*=RECORD sigma*,omega*,gain*:REAL;xk*,yk*,xkm1*,
rt1meps*,norm*,gaincx:cx.complex;first*:BOOLEAN;END;
para*=ARRAY maxpoles+1 OF polerec;
zara*=ARRAY maxpoles+1 OF zerorec;
fltrrec*=RECORD fc*,bw*,f1*,db*,gain*:REAL;n*,nz*,nzo*:LONGINT;
        firstb*:BOOLEAN;p*:para;z*:zara END;
```

Notice that "fltrrec" uses "para" and "zara". The type "para" uses "polerec" and "zara" uses "zerorec". The variable in "fltrrec" that is of type "zara" is the variable "z". A procedure in the program that uses fltrrec would have a variable in the parameter list declared as "fr:fltrrec". The variable "fr" is declared as being of type "fltrrec". A statement from the program that sets a variable "sigma" to zero is:

```
fr.z[i+fr.nz].sigma:=0.0;
```

There are two different variables named "sigma". You should be able to see that the one selected here is from "zerorec", not from "polerec". The "[i+fr.nz]" array index is unusual, "[i]" would be more typical, but this example serves to show what is possible.

If we had two variables of type "fltrrec", like fr1 and fr2, we could copy all of the data from fr1 to fr2 with "fr2:=fr1;".

Records can be used to form objects, for object oriented programming, a topic beyond the scope of this introductory course.

INPUT/OUTPUT

The input/output functions rely heavily on the oo2c library. The library is not nearly as self explanatory as the language. Therefore, it will not be obvious why everything is done the way it is, you just have to accept it and use it as an example if you ever need to do something similar yourself. The library manual provides additional explanation, and the source code for the library provides even more.

When in doubt, it is best to write short test programs to get input/output statements right.

Test a word read from the keyboard:

```
MODULE testwd;
IMPORT In,Out;
TYPE str=ARRAY 40 OF CHAR;
VAR str1:str;
BEGIN
Out.String('enter abc');Out.Ln;
In.Line(str1);
IF str1='abc' THEN Out.String('right word');Out.Ln;
ELSE Out.String('wrong word'); Out.Ln; END;
END testwd.

bin/testwd
enter abc
xyz
wrong word

bin/testwd
enter abc
abc
right word
```

You should should modify the above "testwd" program to eliminate "Out.Ln" after "enter abc". The program will not work. This is a common mistake of beginners.

Read real numbers from the keyboard:

```
MODULE readreal;
IMPORT In,Out;
VAR x,y:REAL;
BEGIN
Out.String('enter x,y, real numbers');Out.Ln;
In.Real(x);In.Real(y);
Out.Real(x,5,6);Out.Ln;
Out.Real(y,5,6);Out.Ln;
END readreal.

bin/readreal
enter x,y, real numbers
8.5 -3.2
8.50000
-3.20000
```

We can run the program again to show how large and small numbers are printed in floating point format:

```
bin/readreal
enter x,y, real numbers
1000 0.001
1.00000E+3
1.00000E-3
```

There are other ways to write out real numbers. You should have downloaded the "OOCref" document. In the table of contents of that document go to the "Standard I/O" section to see ways to write real numbers. You should replace Out.Real with Out.RealEng and Out.RealFix to see other formats to write out the numbers.

Read a word, such as a filename, from the keyboard:

```
MODULE getname;
IMPORT In,Out;
TYPE str=ARRAY 40 OF CHAR;
VAR filename:str;
BEGIN
Out.String('enter filename');Out.Ln;
In.Line(filename);
Out.String('fileid=');
Out.String(filename);Out.Ln;
END getname.

bin/getname
enter filename
myfile.txt
fileid=myfile.txt
```

At your keyboard enter "man ascii" to see decimal equivalents of ascii characters. The next program serves to demonstrate the functions CHR and ORD. A character is read, converted to the ascii decimal equivalent, then the ascii decimal equivalent is converted back into the character. The character is printed out between two "xxx" so that invisible characters can be seen in the example. Note that after "In.Char" an "In.Line" is needed to read to the end of the line. This is so the next character read will be the first character on the next line, not the newline character created when we press "enter". The first example with an apparently blank line has the space character as the first character of the line. The space character can be seen as a space between the two "xxx" terms. The second blank line example has the newline character as the first character of the line. This was accomplished by hitting "enter" twice in a row. The newline character is made apparent by the fact that the two "xxx" terms are on successive lines.

```
MODULE char;
IMPORT In,Out;
VAR i:LONGINT;ch:CHAR;str:ARRAY 20 OF CHAR;
BEGIN
REPEAT
```

```
In.Char(ch);In.Line(str);
i:=ORD(ch);
Out.LongInt(i,3);Out.Ln;
Out.String('xxx');
Out.Char(CHR(i));
Out.String('xxx');
Out.Ln;
UNTIL ch='7';
END char.

q
113
xxxqxxx
#
 35
xxx#xxx

 32
xxx xxx


 10
xxx
xxx
7
 55
xxx7xxx
```

The output of the preceding programs would go to the screen.

The following pair of programs illustrate reading and writing to a disk with text files. In the following examples the filenames are compiled into the program. However, the variable filename with no quotes in the example "getname" above could be substituted for 'demo1.txt' with quotes in the following example, if it were desired for the user to enter filenames.

These programs deal with mixed text and numbers. Note that the text must be enclosed in quotes to be read properly

```
MODULE ritfil1;
IMPORT Msg, Files, TextRider;
VAR resv:Msg.Msg;outvar:Files.File;
outfile:TextRider.Writer;
a,b:LONGINT;c,d:REAL;
BEGIN
a:=5;b:=6;c:=2.3;d:=-2.1;
```

```
outvar:=Files.New('demo1.txt',{Files.write},resv);
outfile:=TextRider.ConnectWriter(outvar);
outfile.WriteLInt(a,3); outfile.WriteString('" some text"');
outfile.WriteRealFix(c,5,1); outfile.WriteLn;
outfile.WriteLInt(b,3); outfile.WriteString('" more text"');
outfile.WriteRealFix(d,5,1); outfile.WriteLn;
outvar.Close;END ritfil1.
```

After running ritfil1 we can look at the file demo1 with the editor and we will find our text in it. This is the way it will appear in the editor:

```
  5" some text"  2.3
  6" more text" -2.1
```

We can also use the lpr command to transfer the file to the printer. Finally we can read the file with the following program which displays it on the screen. Note that str2 is only used to read to the end of line so the next line can be read.

```
MODULE redfil1;
IMPORT Out, Msg, Files, TextRider;
VAR resv:Msg.Msg;invar:Files.File;
infile:TextRider.Reader;
str1,str2:ARRAY 40 OF CHAR;
x:LONGINT;y:REAL;
BEGIN
invar:=Files.Old('demo1.txt',{Files.read},resv);
infile:=TextRider.ConnectReader(invar);
LOOP infile.ReadLInt(x);
IF infile.res#TextRider.done THEN EXIT END;
infile.ReadString(str1); infile.ReadReal(y);
infile.ReadLine(str2);
Out.LongInt(x,3); Out.String(str1);
Out.RealFix(y,5,1); Out.Ln;END;
invar.Close; END redfil1.

  5 some text  2.3
  6 more text -2.1
```

Text files represent numbers as decimal digits, each digit being represented in turn by an 8 bit ASCII character. Such a file of numbers can be read by a person with a text editor. It is more efficient in storage and speed to represent numbers in binary form. Binary files cannot be read by a person with a text editor. The following pair of programs write and read a file of real (floating point) numbers in binary form.

```
MODULE ritfil2;
IMPORT Msg, Files, BinaryRider;
VAR resv:Msg.Msg;outvar:Files.File;
```

```
outfile:BinaryRider.Writer;
i:LONGINT;r:REAL;
BEGIN
outvar:=Files.New('demo2',{Files.write},resv);
outfile:=BinaryRider.ConnectWriter(outvar);
FOR i:=1 TO 5 DO
r:=i; outfile.WriteReal(r); END;
outvar.Close; END ritfil2.
```

```
MODULE redfil2;
IMPORT Out, Msg, Files, BinaryRider;
VAR resv:Msg.Msg;invar:Files.File;
infile:BinaryRider.Reader;r:REAL;
BEGIN
invar:=Files.Old('demo2',{Files.read},resv);
infile:=BinaryRider.ConnectReader(invar);
LOOP infile.ReadReal(r);
IF infile.res#BinaryRider.done THEN EXIT END;
Out.RealFix(r,12,4);Out.Ln;END;
invar.Close; END redfil2.
```

```
      1.0000
      2.0000
      3.0000
      4.0000
      5.0000
```

In the Out.RealFix statement the 12,4 part means the number is right justified in 12 spaces and 4 digits to the right of the decimal point will be shown. There are other ways to write real numbers; consult the oo2c reference manual.

Next, we write a file of integers, read the file, and use set variables to print the results out in binary.

```
MODULE ritfil3;
IMPORT Msg, Files, BinaryRider;
VAR resv:Msg.Msg;outvar:Files.File;
outfile:BinaryRider.Writer;
i:LONGINT;int:INTEGER;
BEGIN
outvar:=Files.New('demo3',{Files.write},resv);
outfile:=BinaryRider.ConnectWriter(outvar);
FOR i:=1 TO 11 DO
int:=SHORT(i); outfile.WriteInt(int); END;
outvar.Close; END ritfil3.
```

```
MODULE redfil3;
IMPORT Out, Msg, Files, BinaryRider;
VAR resv:Msg.Msg;invar:Files.File;
infile:BinaryRider.Reader;s:SET;
i:INTEGER;
BEGIN
invar:=Files.Old('demo3',{Files.read},resv);
infile:=BinaryRider.ConnectReader(invar);
LOOP infile.ReadSet(s);
IF infile.res#BinaryRider.done THEN EXIT END(*if*);
FOR i:=0 TO MAX(SET) DO
IF (MAX(SET)-i) IN s THEN Out.Int(1,1);
ELSE Out.Int(0,1);END(*if*);END(*for*);
Out.Ln; END(*loop*);
invar.Close; END redfil3.
```

```
00000000000000100000000000000001
00000000000001000000000000000011
00000000000001100000000000000101
00000000000010000000000000000111
00000000000010100000000000001001
```

The file was written as 16 bit INTEGER type from 1 to 11. The command "ls -l" shows 22 bytes in the file "demo3". There are 2 bytes per 16 bit word. 22 divided by 2 is 11, so there are 11 of the 16 bit words in the file. But the program redfil3 reads the file as 32 bit machine words. There are two 16 bit integers in each 32 bit word. Reading a set results in reading a machine word worth of bits from the file. There is no provision for reading other numbers of bits when reading sets. Since the last 16 bit word by itself was not a complete 32 bits, it was not read. Had the file been written with 32 bit LONGINT type it would have been easier to read the binary numbers.

Since there is no type checking when reading files this way, the bit pattern of any type of file can be displayed, whether integer, real, text or executable code.

We can use sets to read the 16 bit numbers out of this file demo3 and write them out in a new file demo4 as 32 bit numbers as shown in the program below. If set s:={}, that means all bits are set to zero. s:=s+{7} means that bit 7 of set s is made to equal 1, not zero. With suitable modification this technique can extract any word size from an old file and write the same data, possibly truncated, to any different word size in a new file, even though the actual physical reading and writing takes place in machine words. This might be needed for burning ROM chips for hardware controllers.

```
MODULE rrfil4;
IMPORT  Msg, Files, BinaryRider;
VAR resv1,resv2:Msg.Msg;invar,outvar:Files.File;
infile:BinaryRider.Reader;
outfile:BinaryRider.Writer;
```

```
s,s2:SET; i,j:INTEGER;
BEGIN
s2:={};
invar:=Files.Old('demo3',{Files.read},resv1);
infile:=BinaryRider.ConnectReader(invar);
outvar:=Files.New('demo4',{Files.write},resv2);
outfile:=BinaryRider.ConnectWriter(outvar);
LOOP infile.ReadSet(s);
IF infile.res#BinaryRider.done THEN EXIT END(*if*);
FOR i:=0 TO MAX(SET) DO
j:=i MOD 16;
IF i IN s THEN s2 := s2 + {j};END;
IF j=15 THEN outfile.WriteSet(s2);s2:={};END;
END(*for*); END(*loop*);
invar.Close; outvar.Close;END rrfil4.
```

If redfil3 is recompiled to read "demo4", it will output:

```
00000000000000000000000000000001
00000000000000000000000000000010
00000000000000000000000000000011
00000000000000000000000000000100
00000000000000000000000000000101
00000000000000000000000000000110
00000000000000000000000000000111
00000000000000000000000000001000
00000000000000000000000000001001
00000000000000000000000000001010
```

Suppose you want to read a file of unsigned 8 bit integers. Unsigned integers are all positive, and could be from 0 to 255. The Oberon type SHORTINT refers to 8 bit signed twos complement integers with a maximum value of 127. You would read the file as SHORTINT, then using the function LONG assign the value to a 16 bit INTEGER variable. Then do the appropriate conversion to get the negative numbers to the correct positive values.

In our earlier example "redfil1" we knew the format of the data in the file before we read it. In that case, we used built in functions in the OOC library to read each type. But what if we want to read a file of ascii data whose format we do not know? More complex text files could contain lines having a mix of numbers, words and characters in an order that might differ unpredictably from line to line. The following very general program reads lines in a text file one at a time, finds "items" separated by one or more spaces on a line. In a real application you would test each item to see what kind it was, and interpret and respond to it appropriately. It is not likely that you will face a problem like this, but it serves as an example of low level I/O programming. Here, we merely print out each item to show that we have properly found each item. Warning, there are two html ampersands in this program that will have to be manually changed to simple ampersands before the program will compile properly.

```
MODULE getitem;
IMPORT Out,Files,TextRider;
TYPE str=ARRAY 40 OF CHAR;
VAR f:Files.File;r:TextRider.Reader;
res:Files.Result;
str1,item:str;pos:LONGINT;

PROCEDURE anotheritem(str1:str;VAR i1:LONGINT;
          VAR str2:str):BOOLEAN;
VAR i2:LONGINT;start,finish,space,eol:BOOLEAN;
(*i1 must be initialized to zero by calling program
before first call to this procedure*)
BEGIN
start:=FALSE;finish:=FALSE;space:=FALSE;
eol:=FALSE;i2:=0;
LOOP
IF (str1[i1]=00X) THEN eol:=TRUE;END;
(*the line below must be modified to look the way it looks
when seen in your browser*)
IF (eol & ~start) THEN EXIT END;
IF (str1[i1]=' ') THEN space:=TRUE;
ELSE space:=FALSE;END;
(*the line below must be modified to look the way it looks
when seen in your browser*)
IF ((space OR eol) & start) THEN finish:=TRUE;
EXIT;END;
IF ~space THEN start:=TRUE;str2[i2]:=str1[i1];
INC(i2);END;INC(i1);END(*loop*);
str2[i2]:=00X;
RETURN finish;END anotheritem;

BEGIN
f:=Files.Old("temp.txt",{Files.read},res);
r:=TextRider.ConnectReader(f);
LOOP r.ReadLine(str1);
IF r.res#Files.done THEN EXIT END;
Out.String(str1);Out.Ln;
pos:=0;
WHILE anotheritem(str1,pos,item) DO
Out.String(item);Out.Ln;
END(*while*); END(*loop*); f.Close;
END getitem.
```

The file "temp.txt":

```
product: hammer $20.00
5.83 lbs  discontinued
ref: memo 11-23-41
%  A23.4 %
```

Output of the program:

```
product: hammer $20.00
product:
hammer
$20.00
5.83 lbs  discontinued
5.83
lbs
discontinued
ref: memo 11-23-41
ref:
memo
11-23-41
%  A23.4 %
%
A23.4
%
```

Some file handling programs use standard input and standard output. A simple example is a program to add carriage returns to Linux files. If you type "man 7 ascii" you can see that the symbol for linefeed is "\n", and for carriage return is "\r". Create a tiny test file "test1":

```
xx
yy

zz
```

Then do "hexdump -c test1". You can see that the end of each line is "\n". But Windows needs both "\r" and "\n". The following program addcr.Mod will do this:

```
MODULE addcr;
IMPORT In,Out;
VAR str:ARRAY 256 OF CHAR;
BEGIN
In.Line(str);
WHILE In.Done() DO
```

```
Out.String(str);Out.Char(CHR(13));Out.Ln;
In.Line(str);
END;
END addcr.
```

The program uses standard input and output:

```
bin/addcr < test1 > test2
```

The hexdump of test2 will show that "test2" is compatible with Windows. If as superuser you move addcr to /usr/local/bin, you will be able to use it as a built in system command.

Another simple filter program will convert ordinary text files with a plain font to postscript files with a fancy font. The program is very primitive and will only handle one page of output at a time. If you have the "zip" package installed on your computer, find a font in the "font.zip" format, download it and unzip it and hopefully it will be in the "font.ttf" format or some other format that ghostscript can use. Do "gs -h" to find where ghostscript looks for fonts, and put a fancy font you downloaded from the internet where ghostscript will find it. In this example the fancy font is "OldeEnglish.ttf". The fonts have adjustable size, the size specified is "font=24", a large font size. You can view the postscript file with "gv", if it installed on your computer, or you can print it out. In Debian linux click on "applications" "system tools" "file browser" and you can see how it will appear on a sheet of paper before you print it out. You can convert the postscript to pdf with "ps2pdf". The advantage of pdf over postscript is that the font is actually in the pdf file, not elsewhere on the computer. If you email a pdf file or post it on the web other people will see the fancy font. Another way to post a fancy font on the web is as a png file referenced by an html file, which is the way the example is shown here. The program is shown here followed by input and output:

```
MODULE tx2ps;
(*ascii text to postscript with fancy font.
use: bin/tx2ps < file.txt > file.ps *)
IMPORT In,Out;
CONST xorg=72; yorg=720; font=24;
VAR str:ARRAY 256 OF CHAR;i,j:LONGINT;
BEGIN
i:=0;
Out.String("%!PS");Out.Ln;
Out.LongInt(xorg,4);Out.LongInt(yorg,6);
Out.String(" moveto");Out.Ln;
Out.String("/OldeEnglish.ttf");
Out.LongInt(font,4);Out.String(" selectfont");Out.Ln;
In.Line(str);
WHILE In.Done() DO
Out.Char("(");
Out.String(str);Out.String(") show");Out.Ln;
i:=i+1;j:=yorg-i*font;
Out.LongInt(xorg,4);
```

```
Out.LongInt(j,6);Out.String(" moveto");Out.Ln;
In.Line(str);
END;
Out.String("showpage");Out.Ln;
END tx2ps.

This is Fancy Text
```

You will sometimes need for your program to issue commands to the linux operating system. The "ProcessManagment.system" command will do this. You can enter the command in your program between double quotes "xyz" as an argument of the above command, but you may need the program to compute the command as a string variable, which will require a more complex procedure. The "Strings.Append", "IntStr.IntToStr" and "Object.NewLatin1" commands are useful for this. If at the keyboard you enter the command "echo cat324" the result will be that "cat324" will be printed on the screen. We now give a trivial example of computing a string variable to issue the command "echo cat324", which will result in "cat324" being printed on the screen.

```
MODULE sys;
IMPORT IntStr,Strings,Object,OS:ProcessManagement;
TYPE str=ARRAY 40 OF CHAR;
VAR str1,str2:str;
str3:STRING;i,j:LONGINT;
BEGIN
str1:="echo ";
str2:="cat";
Strings.Append(str2,str1);
i:=324;
IntStr.IntToStr(i,str2);
Strings.Append(str2,str1);
str3:=Object.NewLatin1(str1);
j:=ProcessManagement.system(str3);
END sys.

cat324
```

This technique will allow you to program complicated tasks with Oberon that would otherwise have to be programmed with bash. Oberon is much easier to program than bash.

SEPARATE COMPILATION

Sometimes it is convenient to compile separate pieces of a program separately, and not have everything in one large file. The following example

has files "mod1.Mod" and "mod2.Mod". Mod1 and mod2 are compiled separately. To compile mod1, type "oo2c mod1". To compile mod2 type "oo2c -M mod2". Only the main module needs to be compiled with the -M option. To execute both, type "bin/mod2". To show errors the scripts defined earlier in the "syntax" section are useful: "./ooc mod1" then "./oocm mod2". Separate compilation can involve several layers of modules. Whenever a module is changed, it must be recompiled and every module below it in the chain down to the executable module must be recompiled also.

Mod2 calls the routine "halveara" in mod1 and uses it. Halveara is declared in mod1 with an asterisk, "*", which makes it visible to any external program which uses mod1. Anything made visible with a minus sign, "-" is read only, and cannot be modified by the external program. Anything not explicitly made visible is invisible to the outside.

```
MODULE mod1;
TYPE ara=ARRAY OF REAL;
VAR half:REAL;

PROCEDURE halveara*(n:LONGINT;VAR ar:ara);
VAR i:LONGINT;
BEGIN
FOR i:=0 TO n DO ar[i]:=ar[i]*half;END;
END halveara;

BEGIN half:=0.5; END mod1.

MODULE mod2;
IMPORT m1:=mod1, Out;
CONST m=3;
TYPE ara2=ARRAY m+1 OF REAL;
VAR br:ara2;i:LONGINT;
BEGIN
FOR i:= 0 TO m DO br[i]:=i;END;
FOR i:= 0 TO m DO Out.RealFix(br[i],10,2);
Out.Ln;END;
m1.halveara(m,br);Out.Ln;
FOR i:= 0 TO m DO Out.RealFix(br[i],10,2);
Out.Ln;END;
END mod2.

bin/mod2
      0.00
      1.00
      2.00
      3.00

      0.00
```

```
      0.50
      1.00
      1.50
```

Notice in mod2 that mod1 was imported with a name change: "m1:=mod1". This is desirable when the name of the external module is excessively long. It could have been imported without a name change simply as "mod1".

Notice that the part at the bottom of Mod1 happens automatically when mod1 is imported. If it did not the value of "half" would be a random number or zero when halvara was called.

While mod1 could have declared an array, it did not. It declared an array type without any declaring the size of the array. In the procedure halveara the array was declared as a VAR parameter. The memory locations corresponding to VAR parameters are not in the procedure that declares the VAR parameter, but in the procedure that calls the procedure that has the VAR parameter. Mod1 has no array, only instructions for operating on an array. Mod2 is the only one of the two modules that actually has an array in it. This way mod1 is general purpose, and could be used by some other mod3 that had an array of completely different size from the one in mod2. This kind of array parameter without dimension is called an "open array parameter", and is useful within a module as well as between modules as shown here.

An Oberon browser is included with oo2c. This allows you to see interface information from a separately compiled module without the necessity of reading the detailed code of the module. Thus if we type "oob mod1" we get:

```
MODULE mod1;

TYPE
  [ara] = ARRAY OF REAL;


PROCEDURE halveara (n: LONGINT; VAR ar: ara);

END mod1.
```

Separate compilation can provide the environment needed by many of the examples in the book by Erik Nikitin. He assumes a special operating system that allows keyboard commands to accomplish what requires separate compilation under linux. For instance his first example "OfeHello" on page 7 of his book can be made to work by the following case of separate complilation:

```
MODULE nikitin;
IMPORT OfeHello;
BEGIN
OfeHello.Do;
END nikitin.
```

DEBUGGING

C programmers depend on special programs known as debuggers to get their programs running. Oberon programmers should have less need for debuggers, because of the clarity of the language. However, they will still need basic debugging techniques. The following example illustrates the basic technique:

```
MODULE debug;
IMPORT Out;
VAR ar:ARRAY 3 OF LONGINT;
i:LONGINT;
BEGIN
FOR i:=1 TO 3 DO
(*the following line must be modified to look
the way it does in your browser*)
IF i>2 THEN Out.String('out of range i=');
Out.LongInt(i,4);Out.Ln; HALT(1);END;
ar[i]:=i;
Out.LongInt(ar[i],4);Out.Ln;
END;
END debug.
```

Without the IF statement the program "bombs off" with a run time error when it is executed. The two line IF statement was inserted to write out intermediate results and terminate the program just before it got to the fatal error. In a large program, many such statements might have to be tried at different locations to track down the problem.

Sometimes the compiler issues misleading error statements. If the symbols comma (,), semicolon (;), or colon (:) are interchanged or missing the compiler may complain about missing END words. Single character errors are difficult to see. You should deliberately sabotage a working program with several different single character errors, one at a time, to get a feel for what error messages might mean.

An important mistake to avoid is using the ".Mod" suffix when compiling. If you have a program test.Mod, "./oocm test" will compile it, but "./oocm test.Mod" will not, and will not give you any warning. You may think your corrections have been ignored by the compiler, but your corrections were never compiled because you added the Mod suffix by mistake when you compiled your corrections.

INTERACTIVE PROGRAMS

The following example responds to user word commands to execute procedures "firstproc" or "secondproc" or terminate the program. The commands are "first", "second" or "q". This technique is handy for making user friendly programs:

```
MODULE intrct;
IMPORT In,Out;
CONST er=0;q=1;first=2;second=3;last=4;
TYPE str=ARRAY 10 OF CHAR;
VAR cmdara:ARRAY last+1 OF str;
command:LONGINT;
```

```
PROCEDURE firstproc;
BEGIN Out.String('executed first procedure');Out.Ln;
END firstproc;

PROCEDURE secondproc;
BEGIN Out.String('executed second procedure');
Out.Ln;
END secondproc;

PROCEDURE initvar;
BEGIN
cmdara[er]:='er';
cmdara[q]:='q';
cmdara[first]:='first';
cmdara[second]:='second';
cmdara[last]:='last';
END initvar;

PROCEDURE determine(VAR cmdvar:LONGINT);
CONST bell=7;
VAR i:LONGINT;str1:str;
BEGIN
Out.String('enter command');Out.Ln;
cmdvar:=er;
In.Line(str1);
FOR i:=q TO last DO
IF (cmdara[i]=str1)THEN cmdvar:=i;END;END;
IF cmdvar=er THEN
(*the line below must be modified to look the way it looks
when seen in your browser*)
Out.Char(CHR(bell));Out.String('<--<< error');
Out.Ln;END;
END determine;

BEGIN
initvar;
LOOP
determine(command);
IF command=q THEN EXIT;END;
CASE command OF
er:|
first:firstproc|
```

```
second:secondproc END;
END;END intrct.
```

The first case of "er:|" is required so that if a mistake is made the program will not quit with an error message. Instead, the user will be warned that he has made an error and can try again.

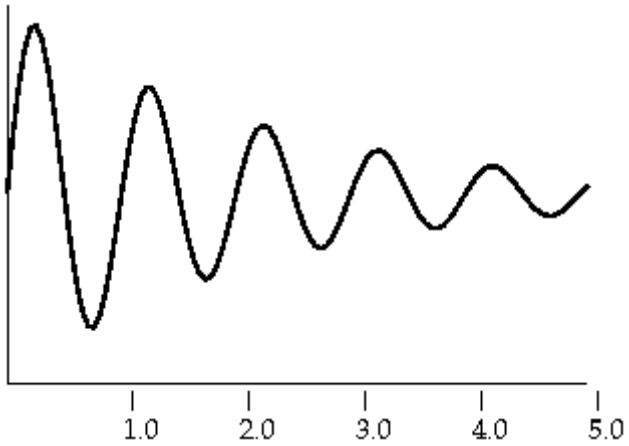
GRAPHICS OUTPUT

A very important application of computing is to plot graphs, or create other graphical output. The book "PostScript by Example", by Henry McGilton and Mary Campione, Addison-Wesley 1992, shows clearly and simply how to use postscript to produce graphics as does the book "Mathematical Illustrations" by Bill Casselman. The latter book is available in hardcopy from Cambridge University Press or free online at www.math.ubc.ca/~cass/graphics/manual/. The following program shows the basics of graphics programming. You must have a window system such as X11, Gnome or KDE running for this program to work. It requires that you have Ghostscript installed on your system. The program produces a plot of a damped sine wave. Then type "quit" to clear it from the screen, and it will print out on your printer.

```
MODULE graf1;
IMPORT In,Out,Msg,Files,TextRider,
OS:ProcessManagement,rm:=RealMath;
CONST move=1;draw=2;
TYPE str=ARRAY 80 OF CHAR;
VAR str1:str;outvar:Files.File;resw:Msg.Msg;
outfile:TextRider.Writer;

PROCEDURE initfile;
BEGIN
outvar:=Files.New('temp1',{Files.write},resw);
outfile:=TextRider.ConnectWriter(outvar);
outfile.WriteString('%!PS');outfile.WriteLn;
END initfile;

PROCEDURE moveto(x,y:REAL;md:LONGINT);
BEGIN
outfile.WriteRealFix(x,8,2);
outfile.WriteRealFix(y,8,2);
IF md=move THEN
outfile.WriteString(' moveto');
ELSE
outfile.WriteString(' lineto'); END;
outfile.WriteLn;
END moveto;

PROCEDURE width(w:REAL);
```

```
BEGIN
outfile.WriteRealFix(w,8,2);
outfile.WriteString(' setlinewidth'); outfile.WriteLn;
outfile.WriteString('stroke');outfile.WriteLn;
END width;

PROCEDURE drawaxis;
BEGIN
moveto(157,584,move);moveto(157,396,draw);
moveto(446,396,draw);width(1);
END drawaxis;

PROCEDURE drawcurve;
CONST xorig=157;pi2=6.28318;ampl=94;cycle=58;
yorig=396;
VAR i:LONGINT;x,y:REAL;
BEGIN
moveto(xorig,yorig+ampl,move);
FOR i:=1 TO 100 DO
x:=(5*cycle/100)*i+xorig;
y:=rm.exp((x-xorig)*(-0.46/cycle))
*ampl*rm.sin((pi2/cycle)*(x-xorig))+ampl+yorig;
moveto(x,y,draw);END;
width(2);
END drawcurve;

PROCEDURE initfont;
BEGIN
outfile.WriteString('/Palatino-Roman 14 selectfont');
outfile.WriteLn;
END initfont;

PROCEDURE labelaxis;
CONST xorig=157;cycle=58;
yorig=396; charsize=14;
VAR i:LONGINT;ir:REAL;
BEGIN
FOR i:=1 TO 5 DO
moveto(xorig+i*cycle,yorig-charsize,move);
outfile.WriteString('(|) show');outfile.WriteLn;
moveto(xorig+i*cycle,yorig-2*charsize,move);
outfile.WriteString('(');
ir:=i;
```

```
outfile.WriteRealFix(ir,8,1);
outfile.WriteString(') show');
outfile.WriteLn;
END;END labelaxis;

PROCEDURE endfile;
BEGIN
outfile.WriteString('showpage'); outfile.WriteLn;
outvar.Close; END endfile;

PROCEDURE viewplot;
VAR i:LONGINT;
BEGIN
i:=ProcessManagement.system("gs -sDEVICE=x11 temp1");
END viewplot;

PROCEDURE printplot;
VAR i:LONGINT;
BEGIN
i:=ProcessManagement.system("lpr temp1");
(*lpr is assumed to send file through
Ghostscript to printer*)
Out.String('sent to printer');Out.Ln;
END printplot;

BEGIN
initfile;drawaxis;drawcurve;initfont;labelaxis;
endfile;
Out.String
('hit return when ready to view plot');Out.Ln;
Out.String
('enter quit when finished viewing plot');Out.Ln;
In.Line(str1);
viewplot;printplot;
END graf1.
```



The program writes out the file "temp1" to disk, and then invokes the system commands "gs" and "lpr" to send "temp1" to the screen, then to the printer. System commands are really just programs that are located in places where they can be run just by typing their names, without the user knowing where they are located. In this program the ProcessManagemet statements invoke external programs just as you would typing at the keyboard. Thus the "gs" and "lpr" programs are used by this program the same way you would use them.

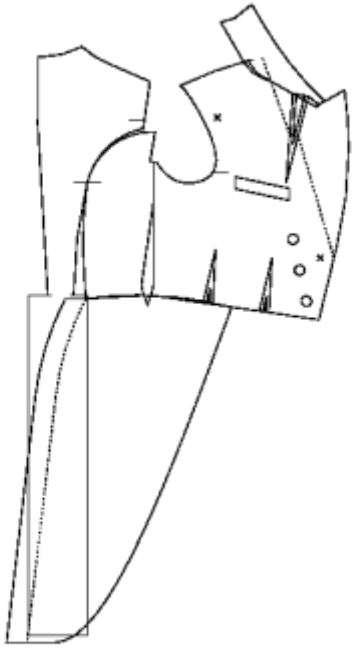
On my printer the x,y coordinates of the lower left corner of the page are 14,18 in PostScript units. The upper right corner is 591,774, and the middle of the page is 302,396. This is for letter paper, 8.5 inches by 11 inches. Your printer may be set up a little bit different from this. There are 72 postscript units per inch, and 2.54 centimeters per inch. You can use real numbers to specify locations to fractions of a postscript unit.

The above program shows the use of the postscript commands "x y moveto" and "x y lineto". For drawing arcs use "xc yc r sa ea arc". xc and yc are the coordinates of the center of the arc, r is the radius of the arc, and sa and ea are the start and end angles of the arc in degrees. Another useful command is the bezier curve for connecting two straight lines with a curve with the "curveto" command. The format is "x1 y1 x2 y2 x3 y3 curveto". The first straight starts where you are and ends at x1 y1. The second straight line starts at x2 y2 and goes to x3 y3. The curve starts at the beginning of the first straight line and ends at the end of the second straight line. It is tangent to each straight line at either end of the curve. The straight lines are not drawn, only the curve is drawn.

If you want to take oversize graphics like clothing patterns or architectural designs to a blueprint shop or graphics shop to be printed you will want to convert them from postscript to pdf format. For letter sized drawings the linux command "ps2pdf" will work. For large size drawings a different way is needed. You may have to install free missing commands on your computer. The above program produces the file "temp1" in postscript format. "pstoedit -f fig temp1 temp1.fig" will convert it to fig format. "fig2dev -L pdf -b 20 temp1.fig temp1.pdf" will convert it to pdf. You can see a part of your large figure on the screen with the linux command "gv" and scroll around the figure, if your computer has enough memeory to handle the size of graphics you have plotted. Even if your computer does not have enough memeory to display large graphics it will probably have no trouble creating the large pdf files. You can now transfer "temp1.pdf" to a USB stick and take it to a copy shop or graphics services shop that has a very large printer and have it printed. This procedure produces a pdf file that spans the smallest rectangle that can hold your image. Large scale printers typically do not print the quarter inch (6mm) nearest the edge of the pdf file. So you will have to draw a thin rectangle beyond the edges

of your graphics for all of your graphics to be printed.

An example of the output of a program that took a man's measurments and drew the pattern for an old fashioned tailcoat follows. It is shown here in a small size. It would be printed in a large size by a large printer at a graphics shop. The program that produced it is too large to include here. The program was written following the instructions in a tailoring book that told how to draw the pattern by hand. It is drawn entirely with straight lines, arcs and bezier curves. No computed curves like the one in "graf1" above were used. If drawn by hand a string compass, ruler and drafting triangle would be used, but to draw it in a computer program geometry and trigonometry was required. The advantage of having a computer program to draw it is that the program works for any measurements:



If you only need to print a small part of your large pdf file, you do not need to go to a graphics shop. If you have the imagemagick package installed, use the display command to crop out the small part you want. You must be very careful to select a region to crop that is neither as wide nor as tall as the size of paper you are going to print on, or the scale will be reduced. "display large.pdf". Click left mouse button, select "transform", "crop", drag the mouse to select what you want, "crop", "file", "save", enter the name "small.pdf", "save", "letter", "select", get out of the program and "gv small.pdf" to see it, or "lpr small.pdf" to print it.

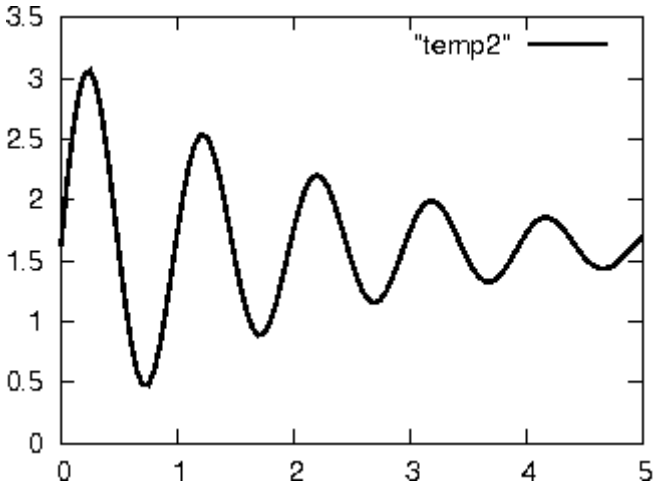
Sometimes you want your text output to be greek letters or math symbols instead of ordinary text. The procedure "initfont" selects "Palatino-Roman", which is ordinary text. Create a similar procedure "initsfont" which selects "Symbol", which is greek letters and math symbols. Each call to either procedure switches to that kind of text. You can have both kinds of text mixed in the same word, if you want to. The size of text selected here is "14". If you want slightly larger text, select "16". When using the Symbol font, you will specify "s" and greek sigma will appear; "w" will result in omega, etc. Some symbols in the Symbol font are beyond the range of the keyboard, and must be specified by an octal number. For example, if you search the web for the symbol font, you can find that the decimal number for the first character of the square root sign is 214. If you install the wcalc command, "wcalc -o 214" gives the octal equivalent 0326. In your postscript file "(\326) show" will produce the character. The top of the square root sign can be completed by the underscore character from the keyboard. If you are going to include something complicated like an equation in your graphics, it would be too cumbersome to debug it in your program. Instead, write the postscript for the equation by hand using the vi editor, and view it each step of the way with the "gv" command. You will need the postscript commands "save", "scale", "restore" and "rmoveto". Once you have the postscript for the equation debugged, it will be easy to add commands to generate it in your program.

One application for graphics is publishing, especially on the internet. You can use the word processing system called Latex that is supplied with Linux systems to create fancy text. You can integrate plots like the one created by the program above in your text. You can use the Linux command "ps2pdf" to convert your PostScript document to a pdf document suitable for posting on the internet. An example of this, showing the graph produced by the above program is the PDF Example at this website. Another alternative is to convert it to a png file. You will need the packages "gv" and "netpbm" installed on your system to do this. Using the command "gv temp1" you can move the cursor to find the lower left and upper right bounds of the figure. Then using the vi editor add the following line as the second line in temp1: "%%BoundingBox: 156 368 464 584". Add the last line "%%EOF". These two changes change the file from ordinary postscript to encapsulated postscript, which is required for some programs. Now 464-156=308, the x-extent, and 584-368=216, the y-extent. Now "pstopnm -xborder=0 -yborder=0 -xmax=308 -ymax=216 -stdout temp1 > temp1.pnm". Then "pnmtopng temp1.pnm > temp1.png". This is discussed at length at http://mintaka.sdsu.edu/GF/bibliog/latex/PSconv.html. A faster way to determine the bounding box is to use the command ps2epsi, which produces a version of the postscript file with the epsi suffix, line 8 of which is the closest possible bounding box. Another way to get png from encapsulated postscript is to use pstopnm with no arguments. pstopnm temp1 will create a ppm version "temp1001.ppm". "display temp1001.ppm" will display it. It will most likely be flipped in orientation. "pnmflip -cw *ppm" > temp1.ppm" will flip it back. But it is probably too large. "pnmscale 0.5 temp1.ppm > temp1.pnm" will shrink it. Then "pnmtopng temp1.pnm > temp1.png" will produce the png version.

Note that the above example used the Oberon programming language to write a file in the Postscript graphics language to be displayed by the Ghostscript graphics viewer. Similarly, the Oberon programming language could write files in any other graphics language for display by the apropriate viewer. Other viewers for other graphics languages are available in linux. In Debian linux, the artistic graphics programs are listed under graphics, and the graphics programs for plotting mathematical functions are listed under mathematics in the list of debian packages. What follows is an example of the same program shown above re-written to use the gnuplot graphics program. We did not have to do as much detailed work in this example.

```
MODULE graf2;
IMPORT In,Out,Msg,Files,TextRider,
OS:ProcessManagement,rm:=RealMath;
TYPE str=ARRAY 80 OF CHAR;
VAR str1:str;outvar:Files.File;resw:Msg.Msg;
outfile:TextRider.Writer;

PROCEDURE initfile;
BEGIN
outvar:=Files.New('temp2',{Files.write},resw);
outfile:=TextRider.ConnectWriter(outvar);
END initfile;

PROCEDURE drawcurve;
CONST xorig=0.0;pi2=6.28318;ampl=1.62;cycle=1.0;
yorig=0.0;
VAR i:LONGINT;x,y:REAL;
BEGIN
FOR i:=0 TO 100 DO
x:=(5*cycle/100)*i+xorig;
y:=rm.exp((x-xorig)*(-0.46/cycle))
*ampl*rm.sin((pi2/cycle)*(x-xorig))+ampl+yorig;
outfile.WriteRealEng(x,10,4);outfile.WriteRealEng(y,10,4);
outfile.WriteLn;
END;
outvar.Close;
END drawcurve;

PROCEDURE cmdfile;
BEGIN
outvar:=Files.New('cmd',{Files.write},resw);
outfile:=TextRider.ConnectWriter(outvar);
outfile.WriteString('set yrange [0.0:3.5]');
outfile.WriteLn;
outfile.WriteString('set style line 1 lw 2 lt -1');
outfile.WriteLn;
outfile.WriteString('set terminal postscript portrait size 5.15,3.57');
outfile.WriteLn;
outfile.WriteString('set output "temp2.ps"');
outfile.WriteLn;
outfile.WriteString('plot "temp2" with lines ls 1');
outfile.WriteLn;
outvar.Close;
```

```
END cmdfile;

PROCEDURE viewplot;
VAR i:LONGINT;
BEGIN
cmdfile;
i:=ProcessManagement.system("gnuplot cmd");
i:=ProcessManagement.system("gs -sDEVICE=x11 temp2.ps");
END viewplot;

PROCEDURE printplot;
VAR i:LONGINT;
BEGIN
i:=ProcessManagement.system("lpr temp2.ps");
END printplot;

BEGIN
initfile;drawcurve;
Out.String
('hit return when ready to view plot');Out.Ln;
Out.String
('enter quit when finished viewing plot');Out.Ln;
In.Line(str1);
viewplot;printplot;
END graf2.
```



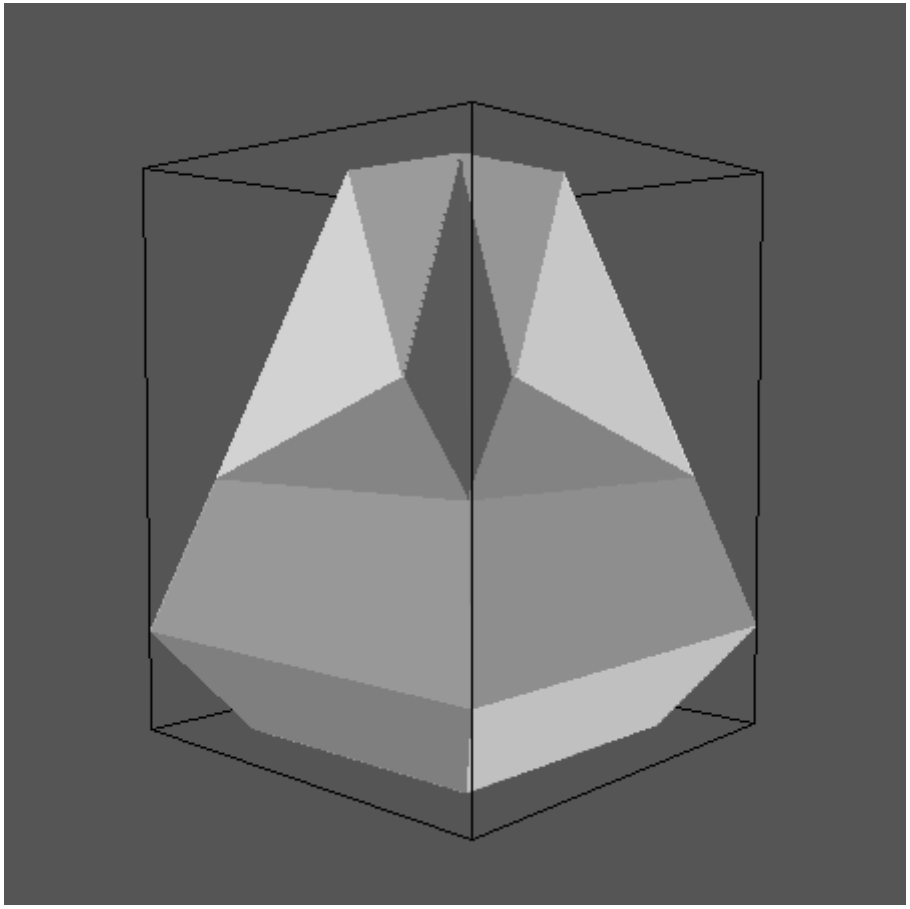
We have now seen two ways to plot the same graph. The first way is more general, and could have been used to draw a building or the artwork for a printed circuit board. The second way is more convenient to illustrate numerical data. The gnuplot program can draw curves, surfaces, or various kinds of charts.

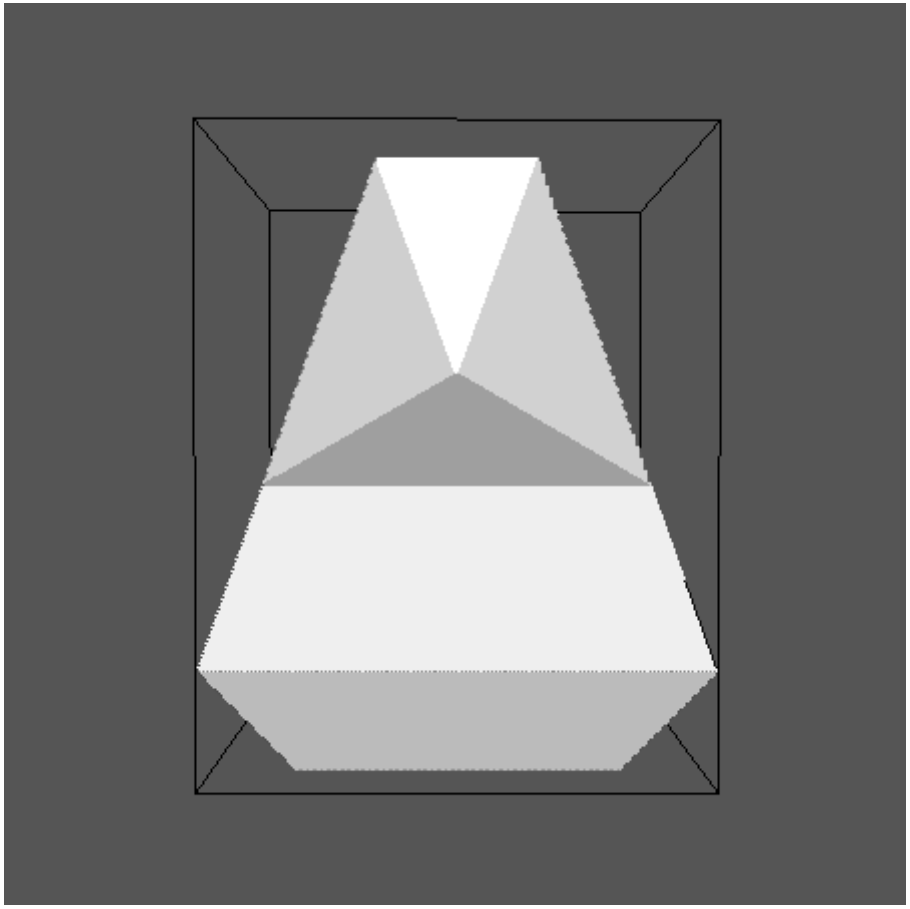
Next we give an example of graphics programming to create a geometrical object, in this case a modern obelisk like would be used to decorate a campus or a park. It would be made of metal plates welded together, preferably heliarc welded titanium to prevent corrosion. We use the linux graphics program "geomview". Our program will write a file in the format that geomview can read, and call geomview to display the object. We can change the input data and change the proportions of the object. Our object is built in pieces that are added one at a time to finish the final object. Theoretically the object could have been created in one piece, but errors would be too difficult to correct that way. Geomview allows you to rotate the object any way you want to view it from any angle, but you must use the left and the middle mouse buttons for different movements. As per the instructions the vertices of each face are enumerated in the program in a counterclockwise direction as seen from the exterior of the object. To see the structure without the fins, comment out the call to fins at the end of the program, recompile ignoring the compiler warning, and run.

```
MODULE obelisk;
IMPORT  Msg, Files, TextRider,
OS:ProcessManagement;
VAR h1,h2,h3,h4,w0,w1,w2,w3,w4: REAL;
i,vert:LONGINT;
resv:Msg.Msg;outvar:Files.File;
outfile:TextRider.Writer;
infile:TextRider.Reader;invar:Files.File;

PROCEDURE square(h,w:REAL;norm:BOOLEAN);
VAR w2:REAL;sq:ARRAY 4,3 OF REAL;
i,j:LONGINT;
BEGIN
w2:=w/2;
IF norm THEN
sq[0,0]:=w2;sq[0,1]:=w2;sq[0,2]:=h;
sq[1,0]:=-w2;sq[1,1]:=w2;sq[1,2]:=h;
sq[2,0]:=-w2;sq[2,1]:=-w2;sq[2,2]:=h;
sq[3,0]:=w2;sq[3,1]:=-w2;sq[3,2]:=h;
ELSE
sq[0,0]:=w2;sq[0,1]:=0;sq[0,2]:=h;
sq[1,0]:=0;sq[1,1]:=w2;sq[1,2]:=h;
sq[2,0]:=-w2;sq[2,1]:=0;sq[2,2]:=h;
sq[3,0]:=0;sq[3,1]:=-w2;sq[3,2]:=h;END;
FOR i:=0 TO 3 DO
FOR j:=0 TO 2 DO
outfile.WriteRealFix(sq[i,j],6,2);END;
outfile.WriteString(' # ');outfile.WriteLInt(vert,3);
```

```
outfile.WriteLn;INC(vert);END;
END square;

PROCEDURE block(hl,wl,hu,wu:REAL);
BEGIN
vert:=0;
outfile.WriteString('{ OFF');outfile.WriteLn;
outfile.WriteString('8 6 0');outfile.WriteLn;
square(hl,wl,TRUE);
square(hu,wu,TRUE);
outfile.WriteString('4 3 2 1 0');outfile.WriteLn;(*bottom*)
outfile.WriteString('4 0 1 5 4');outfile.WriteLn;(*sides*)
outfile.WriteString('4 1 2 6 5');outfile.WriteLn;
outfile.WriteString('4 2 3 7 6');outfile.WriteLn;
outfile.WriteString('4 3 0 4 7');outfile.WriteLn;
outfile.WriteString('4 4 5 6 7');outfile.WriteLn;(*top*)
outfile.WriteString(' }');outfile.WriteLn;
END block;

PROCEDURE fins;
BEGIN
vert:=0;
outfile.WriteString('{ OFF');outfile.WriteLn;
outfile.WriteString('12 8 0');outfile.WriteLn;
square(h2,w2,TRUE);
square(h3,w3,FALSE);
square(h4,w4,TRUE);
outfile.WriteString('3 0 8 4');outfile.WriteLn;
outfile.WriteString('3 0 5 8');outfile.WriteLn;
outfile.WriteString('3 1 9 5');outfile.WriteLn;
outfile.WriteString('3 1 6 9');outfile.WriteLn;
outfile.WriteString('3 2 10 6');outfile.WriteLn;
outfile.WriteString('3 2 7 10');outfile.WriteLn;
outfile.WriteString('3 3 11 7');outfile.WriteLn;
outfile.WriteString('3 3 4 11');outfile.WriteLn;
outfile.WriteString(' }');outfile.WriteLn;
END fins;

BEGIN
invar:=Files.Old('obin',{Files.read},resv);
infile:=TextRider.ConnectReader(invar);
infile.ReadReal(h1); infile.ReadReal(h2); infile.ReadReal(h3);
infile.ReadReal(h4); infile.ReadReal(w0); infile.ReadReal(w1);
```

```
infile.ReadReal(w4); invar.Close;
w2:=(w1-w4)*((h4-h2)/(h4-h1))+w4; w3:=w4;
outvar:=Files.New('obelisk.txt',{Files.write},resv);
outfile:=TextRider.ConnectWriter(outvar);
outfile.WriteString('{ LIST ');outfile.WriteLn;
outfile.WriteString('# h1,h2,h3,h4=');
outfile.WriteRealFix(h1,7,3); outfile.WriteRealFix(h2,7,3);
outfile.WriteRealFix(h3,7,3); outfile.WriteRealFix(h4,7,3);
outfile.WriteLn;
outfile.WriteString('# w0,w1,w4=');
outfile.WriteRealFix(w0,7,3); outfile.WriteRealFix(w1,7,3);
outfile.WriteRealFix(w4,7,3); outfile.WriteLn;
outfile.WriteString('# w2,w3=');
outfile.WriteRealFix(w2,7,3); outfile.WriteRealFix(w3,7,3);
outfile.WriteLn;
outfile.WriteString('# base ');outfile.WriteLn;
block(0,w0,h1,w1);
outfile.WriteString('# side');outfile.WriteLn;
block(h1,w1,h2,w2);
outfile.WriteString('# bevel');outfile.WriteLn;
block(h2,w2,h3,w3);
outfile.WriteString('# shaft');outfile.WriteLn;
block(h3,w3,h4,w4);
outfile.WriteString('# fins');outfile.WriteLn;
fins;
outfile.WriteString(' }');outfile.WriteLn;
outvar.Close;
i:=ProcessManagement.system("geomview obelisk.txt");
END obelisk.

the input data file is:

2 5 7 11 5.717 8.55 3
```





The above example illustrates the advantage of the combination of a custom program with a drawing program over a drawing program alone. Certain practical and aesthetic rules can be enforced while changing the proportions for the most satisfactory result. In this example, all horizontal cross-sections of the inner structure are square. The shaft is of constant cross-section. The corner of the sides and the outer edge of the fin are always on the same straight line. The inner vertices of adjacent fins meet at the base of the shaft. Any or all of the input data parameters can be changed without violating any of these rules.

The default behavior of geomview is spectacular, but not easily useable. To tame it and make it more useable, create the following file named ".geomview". The leading period in the name is essential. Put the file in your home directory:

```
(progn
(ui-motion inertia off)
```

```
(ui-motion constrain on)
(ui-motion own-coordinates on)
)
```

The leading period in the name ".geomview" means that when you do "ls" you will not see it. If you do "ls -aF" in your home directory you will see it and other configuration files.

The picture was produced for this web page in the following manner. In geomview, click "file", "save", "commands" "ppm software snapshot". Then enter "mypicture.ppm" in the selection window, then click "ok". Then get out of geomview and use the netpbm command "pnmtopng mypicture.ppm > mypicture.png". Then if you have imagemagick installed, "display mypicture.png" will display the result.

If you want to program games, you will want to use OpenGL. Install the program glxgears and run it to see that OpenGL is working properly on your system. The program glxinfo will give the status of your OpenGL software. The GLUT package is helpful for developing OpenGL programs. There are books available on OpenGL programming.

AUDIO OUTPUT

We give here a trivial example of audio programming. We create a short beep file at an audio frequency of 1 kHz at the sample rate of audio CD's, 44.1 kHz. We use 16 bit integer arithmetic just as is used on audio CD's. The largest number in 16 bit signed numbers is 32768. To make the beep 20dB quieter than the maximum, we want a number about one tenth of the maximum. The maximum value of a sine wave is one. Therefore we multiply our sine wave by 3000. We have only one channel, not two as CD's have. We then use the program oggenc to change the file to a compressed ogg vorbis format. The open ogg vorbis audio format is to the proprietary MP3 audio format what the open png graphics format is to the proprietary GIF graphics format. We then use the program ogg123 to play our ogg vorbis file on the computer's speaker.

```
MODULE beep;
IMPORT Msg,Files,BinaryRider,rm:=RealMath,
OS:ProcessManagement;
CONST pi2=6.28318;rate=44.1E3;lf=20.0;seconds=2.0;
VAR t,a,b,freq,xkm1,ykm1,est,gf:REAL;i,e:LONGINT;
ai:INTEGER;resv:Msg.Msg;outvar:Files.File;
outfile:BinaryRider.Writer;first:BOOLEAN;

PROCEDURE highpass(t,fhp,xk:REAL;VAR yk:REAL);
BEGIN
IF first THEN est:=rm.exp(-pi2*fhp*t);
gf:=(1+est)/(1-est);first:=FALSE; END(*IF*);
yk:=(xk-xkm1)/2;xkm1:=xk;
xk:=yk;yk:=est*ykm1+(1-est)*xk;ykm1:=yk;
yk:=gf*yk;
END highpass;

BEGIN
```

```
t:=1.0/rate;xkm1:=0;ykm1:=0;first:=TRUE;
outvar:=Files.New('beepfile',{Files.write},resv);
outfile:=BinaryRider.ConnectWriter(outvar);
e:=ENTIER(rate*seconds);freq:=1000;
FOR i:=1 TO e DO
a:=3000*rm.sin(pi2*freq*i/rate);
highpass(t,lf,a,b);
ai:=SHORT(ENTIER(b));
IF i>ENTIER(rate*10/lf)THEN
outfile.WriteInt(ai);END;END;
outvar.Close;
i:=ProcessManagement.system("oggenc -r -C 1 beepfile");
i:=ProcessManagement.system("ogg123 beepfile.ogg");
END beep.
```

The procedure "highpass" is a highpass digital filter that is 3dB down at "lf", in this case 20Hz. It is used to make sure that the output is equally positive and negative so that it averages to zero and has no "dicrect current" (DC) component. Output is inhibited until the output of the highpass filter has had a chance to settle. The filter is not needed in this example, since the sinewave generated has no DC component. When devising more complicated waveforms it is hard to avoid a DC component, which the filter will eliminate.

You can download and play the above "beepfile.ogg". If you have windows and wish for free to add the capablity to play vorbis audio files to your computer, go to www.vorbis.com. Then click on "windows", then "DirectShow Filters", then "opencodecs..." then run the installation. After installation reboot. If you have linux install the program ogg123, probably in the "vorbis tools" package, from your linux distribution, then reboot. To download the "beepfile.ogg" created by the program "beep" above click here.

If you want to put the beepfile in /usr/local/bin and use it from a script in /usr/local/bin, absolute, not relative filenames must be used. Thus, the line in your script that would invoke the beepfile would be "ogg123 /usr/local/bin/beepfile.ogg".

The file "beepfile" is in raw format, not ogg format. If you change the name to "beepfile.raw", then the following commands will convert it to two channel audio and write it to a blank CD in a format that will play on an audio CD player:

```
sox -r 44.1k -e signed -b 16 -c 1 beepfile.raw -c 2 beepfile.wav

wodim -v -pad -eject dev=/dev/sr0 -dao -swab *.wav
```

If you want to go further in audio programming, the book "the physics of musical instruments" by N.H.Fletcher and T.D.Rossing will be helpful. Also the articles on digital filtering in this website, click here.

MATHEMATICAL PROGRAMMING

If you will not need to know how to do mathematical programming then skip to the next section.

Oberon does not have the built in mathematical functions of FORTRAN. The language does not even have elementary transcendental functions. However, any implementation will probably provide at least these, certainly oo2c does, in the library module "RealMath". You can read about the available functions in the OOCref manual in the "mathematical functions" section. You can add your own math functions without much difficulty. The book "Handbook of Mathematical Functions" by Abramowitz and Stegun contains formulas for all the math functions you could ever want. It also includes numerical tables of functions so that you can check that your math functions are producing the right answers. Novice programmers give too much importance to built in math functions. More important is how the language facilitates writing complex programs in a comprehensible manner.

The following module provides basic complex arithmetic.

```
MODULE cxarith;
(*written 1980 donald daniel.  Originally written in pascal and
translated to Oberon-2 in 2000. This program is free software:
you can redistribute it and/or modify it under the terms of the
GNU General Public License as published by the Free Software
Foundation, version 3 of the License.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General
Public License along with this program.  If not, see
http://www.gnu.org/licenses/.  *)
IMPORT rm:=RealMath;
TYPE complex* = RECORD r*,x*:REAL END;
VAR one-,zero-,jone-,mone-:complex;
PROCEDURE neg*(z1:complex;VAR z2:complex);
BEGIN z2.r:=-z1.r;z2.x:=-z1.x END neg;
PROCEDURE conj*(z1:complex;VAR z2:complex);
BEGIN z2.r:=z1.r; z2.x:=-z1.x END conj;
PROCEDURE add*(z1,z2:complex;VAR z3:complex);
 BEGIN z3.r:=z1.r+z2.r;z3.x:=z1.x+z2.x END add;
PROCEDURE sub*(z1,z2:complex;VAR z3:complex);
 BEGIN z3.r:=z1.r-z2.r;z3.x:=z1.x-z2.x END sub;
PROCEDURE mult*(z1,z2:complex;VAR z3:complex);
 BEGIN z3.r := z1.r*z2.r-z1.x*z2.x;
 z3.x := z1.r*z2.x+z2.r*z1.x END mult;
PROCEDURE rmult*(r:REAL;z2:complex;VAR z3:complex);
BEGIN z3.r:=r*z2.r;z3.x:=r*z2.x;END rmult;
PROCEDURE div*(z1,z2:complex;VAR z3:complex);
```

```
(*z3:=z1/z2*)
VAR h:REAL;
BEGIN
IF (ABS(z2.r)>ABS(z2.x)) THEN
h:=z2.x/z2.r;z2.r:=h*z2.x+z2.r;
z3.r:=(z1.r+h*z1.x)/z2.r;
z3.x:=(z1.x-h*z1.r)/z2.r
ELSE
h:=z2.r/z2.x;z2.x:=h*z2.r+z2.x;
z3.r:=(h*z1.r+z1.x)/z2.x;
z3.x:=(h*z1.x-z1.r)/z2.x END END div;
PROCEDURE cxr*(r:REAL;VAR z:complex);
BEGIN z.r:=r;z.x:=0;END cxr;
PROCEDURE cxj*(r:REAL;VAR z:complex);
BEGIN z.r:=0;z.x:=r;END cxj;
PROCEDURE cpx*(r,x:REAL;VAR z:complex);
BEGIN z.r:=r; z.x:=x  END cpx;
PROCEDURE abs*(z1:complex):REAL;
VAR h:REAL;
 BEGIN
 z1.r:=ABS(z1.r);z1.x:=ABS(z1.x);
 IF z1.x>z1.r THEN h:=z1.r;z1.r:=z1.x;z1.x:=h END;
 IF z1.x=0.0 THEN RETURN z1.r ELSE
 RETURN z1.r*rm.sqrt(1.0+(z1.x/z1.r)*(z1.x/z1.r)) END END abs;
PROCEDURE expj*(x:REAL;VAR z:complex);
(* e^jx *)
 BEGIN z.r:=rm.cos(x); z.x:=rm.sin(x) END expj;
PROCEDURE exp*(z1:complex;VAR z2:complex);
VAR x:REAL;
BEGIN
x:=rm.exp(z1.r);
z2.r:=x*rm.cos(z1.x);z2.x:=x*rm.sin(z1.x);END exp;
PROCEDURE sqrt*(z1:complex;VAR z2:complex);
VAR h:REAL;
BEGIN
h:=rm.sqrt((ABS(z1.r)+abs(z1))/2.0);
IF z1.x#0.0 THEN z1.x:=z1.x/(2.0*h)END;
IF z1.r>=0.0 THEN z1.r:=h
ELSIF z1.x>=0.0 THEN z1.r:=z1.x;z1.x:=h
ELSE z1.r:=-z1.x;z1.x:=-h END;
(*the else part of the following if statement
adopts the convention that zero to pi, rather
thanzero to plus or minus half pi, is the
```

```
principal root*)
IF z1.x>=0.0 THEN z2.r:=z1.r;z2.x:=z1.x
ELSE z2.r:=-z1.r;z2.x:=-z1.x;END;
END sqrt;

BEGIN
one.r:=1.0;one.x:=0.0;
zero.r:=0.0;zero.x:=0.0;
jone.r:=0.0;jone.x:=1.0;
mone.r:=-1.0;mone.x:=0.0 END cxarith.
```

Note that any other module which imports cxarith could shorten the name, as "IMPORT cx:=cxarith". This way, complex multiply would be called as "cx.mult(x,y,z);".

PROGRAMMNIG HINTS

Some people, like myself, may find all the capitilization required in Oberon to be inconvenient. There is an easy solution. Write the program in lower case. Create a separate file called "caps". The version of caps that you see below looks correct in your browser, but the embedded html code is not correct. For a useable version click on caps.txt. On linux systems change the name from caps.txt to caps. Note that there is a minor insurmountable problem: "char" becomes "Char" in the library, and "CHAR" in the language.

```
%s/\<import\>/IMPORT/g
%s/\<module\>/MODULE/g
%s/\<in\>/In/g
%s/\<out\>/Out/g
%s/\<ln\>/Ln/g
%s/\<line\>/Line/g
%s/\<string\>/String/g
%s/\<procedure\>/PROCEDURE/g
%s/\<const\>/CONST/g
%s/\<type\>/TYPE/g
%s/\<longint\>/LONGINT/g
%s/\<entier\>/ENTIER/g
%s/\<div\>/DIV/g
%s/\<real\>/REAL/g
%s/\<begin\>/BEGIN/g
%s/\<end\>/END/g
%s/\<var\>/VAR/g
%s/\<for\>/FOR/g
%s/\<do\>/DO/g
%s/\<if\>/IF/g
%s/\<then\>/THEN/g
```

```
%s/\<else\>/ELSE/g
%s/\<case\>/CASE/g
%s/\<of\>/OF/g
%s/\<to\>/TO/g
%s/\<boolean\>/BOOLEAN/g
%s/\<true\>/TRUE/g
%s/\<false\>/FALSE/g
%s/\<or\>/OR/g
%s/\<array\>/ARRAY/g
%s/\<char\>/CHAR/g
%s/\<chr\>/CHR/g
%s/\<repeat\>/REPEAT/g
%s/\<loop\>/LOOP/g
%s/\<halt\>/HALT/g
%s/\<exit\>/EXIT/g
%s/\<until\>/UNTIL/g
%s/\<while\>/WHILE/g
%s/\<abs\>/ABS/g
```

Use the vi editor to edit your Oberon program. When you type "vi" you will get either of two versions of the vi editor, depending on how your system is set up. The versions are nvi and vim. You need vim for this, so type "vim" rather than vi, if you are not sure which is the default on your system. Then in the vim editor, when in the edit mode, not the input mode, type ":so caps". You may need to hit the space bar several times before the process is finished. When done, all the proper key words will be capitalized automatically.

Managers and professional experts who don't write much code make a big fuss over the format of programs. This is nonsense. Usually it is best to use one line per statement. But sometimes more than one statement on a line enhances readability in a program for the same reason that it does in English prose. Skipping lines between procedures enhances readability for the same reason skipping lines between paragraphs does in English. Fancy indenting schemes help a lot only if you write procedures that are much longer than you should be writing. The real keys to readability are taking the time to concentrate hard on the names you call things and most importantly thinking and rethinking how you subdivide the problem into procedures. Write comments explaining the purpose of each procedure if the name of the procedure does not make it obvious.

If the same block of code is repeated in different places, extract it into a procedure that is called from different places. Elimination of repeated code means that there is less code to read to understand the program, and less code to change if changes must be made.

The programmer should be aware of how to use flowcharts, dataflow diagrams and other conceptual tools, but he should not be forced to use them where he does not feel they would do him any good. This would be analogous to requiring a mechanic to use a particular wrench for everything he does in fixing a car. The author wrote a 44 page program where dataflow diagrams and structure charts seemed to help only at the highest level. Flowcharts would have been worse than useless for all but one page of the program, and that page would never have been figured out without flowcharting. Informal essays describing each major section yet to be programmed were very useful to get the ball rolling, but proved to be less than 50% accurate after the section was completed, and needed to be rewritten to properly document the program. I do not have any experience in a team all working on the same program. In such a situation more use of formal procedures is probably appropriate.

Another area where irrational dogmatism has come raining down on the poor programmer is global variables versus parameter lists. Some authorities virtually rule out the use of global variables, claiming parameter lists should be used to pass every variable to every procedure or function. This is nonsense. Such a rule greatly discourages subdividing the program into small procedures to enhance readability and modifiability. If the program is about say, taxpayers, and a taxpayer record is used throughout the program, then it should be global to the whole program and should seldom appear in a parameter list. If a procedure will be called several times with different arguments, these should be passed through a parameter list. If the argument is the same every time, putting it in a parameter list may or may not significantly improve the readability of the program, and should be at the discretion of the programmer.

LARGE MODULES

There are considerations for large modules that don't arise in small ones.

First, there may be a page or more of global type and variable declarations. One frequently needs to refer to these in writing and debugging the module, and it can be difficult to find one if they are all lumped together. The module can usually be considered to have major sections, or at least major topics, even if these topics are not lumped form sections. The TYPE and VAR declarations should each be subdivided into paragraphs labeled by comments according to section or topic, to make it easy to find what you are looking for and easy to understand the meaning of the module.

In a large program you cannot wait until the whole thing is finished to see if you are getting right answers to intermediate steps. It will save much debugging time to write otherwise unnecessary procedures whose sole purpose is to write out intermediate data in an understandable form to check the program as it is being written, well before it is finished.

Large programs are the primary justification for learning objects, which are provided in Oberon-2 but are not covered in this article.

HOW TO CONVERT PASCAL TO OBERON

Converting old Pascal programs to Oberon-2 is not hard. The following hints will be helpful:

```
array[100,100] --> ARRAY 100,100

if...then...; --> IF...THEN...END;

if...then begin...end else begin...end;
--> IF...THEN...ELSE...END;

else if --> ELSIF

<> --> #

integer --> LONGINT
```

```
while...do begin...end --> WHILE...DO...END

for...to...do begin...end;
--> FOR...TO...DO...END;

case...of...:...;...:...;end;
-->CASE...OF...:...|...:...END;

function func(...):...;
begin...func:=...;end;
--> PROCEDURE func(...):...;
BEGIN...RETURN...;END func;

round -->ENTIER
```

The file "caps" in the previous section may be modified to help with conversion of Pascal to Oberon.

up one level

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.