EDMONTON 99'er COMPUTER USERS' SOCIETY

The Edmonton 99'er Computer Users' Society
p.o. box 11983
Edmonton, Alberta
CANADA T5J 3L1

99'er ON LINE... is the news letter of the Edmonton 99'er Computer User's Society published ten times a year. Unless otherwise stated, all articles may be republished in other news letters provided that source and author are identified. We will credit authors quoted in 99'er ON LINE.

CORRESPONDENCE: Newsletter editor: Bob Pass, 59 Labelle Cr, St. Albert, Alberta, Canada T8N-2G6. (403)458-7658. All other correspondence should be sent to the address at left.

OFFICERS: president Tom Hall, vice-pres Ken Godbeer, treasurer Jim Mulligan, secretary Roxanne Appelt

DISCLAIMER: Information published in this newsletter is created by and for amateurs; therefore, we cannot guarantee the accuracy or use of presented information.

REGULAR MEETINGS... of the Edmonton 99'er Computer User's Society are held on the second Tuesday of each month in room 849 of the General Services building of the University of Alberta from 7:00 till 10:00 PM and are open to all members in good standing. Non-members may attend their first meeting free of charge.

ADVERTIZING... Commercial space is available in this news letter at the following rates: full page $20.00, half page $15.00, 1/4 page $10.00. Discuss your needs with Jim Mulligan at 467-6021, at the next meeting, or send "photo ready" copy to the P/O Box above. Members may advertise their personal computer related items for free but are asked to limit their ads to about 50 words. Mail your ads to the editor's address or hand it to him at the general meeting; newsletter deadline 3'rd Monday of the month.

MEMBERSHIP FEES: Family 12 months, $20.00, 6 months, $15.00. Students 12 months, $15.00, 6 months, $10.00. New member initiation, $20.00.

JANUARY MEETING

HAPPY BIRTHDAY TO US! The Edmonton 99'ers Computer Users' Group is five years old this January and still going strong.

CLUBLINE/99 -- The November issue of the magazine finally arrived and was distributed at the meeting. Jim Beck and Wolly Berezowsky please pick up your issues at the next meeting. We have about 5 extra issues each month which can be purchased for $2.00 a copy on a first come, first served basis. The subscription has been renewed through to February/87. We will conduct a review the product at the next meeting.

Attendance at the meeting was 33 people. It is good to see a good turn out for the meetings. Unfortunately, we forgot to hold the draw for door prizes again! February for sure. Godbeer will be providing about two dozen donuts at each meeting at a cost of ten cents each; please leave money in the coffee fund piggy bank. Thanks Ken and thanks to Lois Meunier and Roxane Appelt too for bringing some more home baking.

Tom Hall mentioned that Miller Graphics will announce on Friday, January 16'th, compatability with IBM. Watch this space for more details as they become available.

Bob Chapman (with able keyboard assistance from John Harbour) put on an excellent demonstration of TI MULTIPLAN. Bob showed how to set up a simple spread sheet and demonstrated the concepts of spread sheeting and how to use some of the powerfull editing features of the package.

Ron Hohmann has relinquished his duties as librarian of external newsletters due to other commitments. Thank you Ron for attending to this chore for the past year and a half. Paul Helwig has assume Ron's duties and has asked for one or two volunteers to help him cross index the news letter library.

There have been some new releases to some of the software packages in our disk library. If you are using any of the following packages then check with our disk library for the latest version:

HORIZON RAM DISK "MENU" ROS 5.5, c99 3.0, FUNLWRITER 3.4, SIDEPRINT V2, PRBASE V2, INCOME TAX CALCULATOR 1986, DISK MANAGER 1000 V3.5

Other additions to the library include programs from the defunct HCM magazine volumes 4.1, 4.5, and 5.1 through 5.6. Or how about "NINJA" by Jim Beck, or the original 99'er programs from Volume 1, Numbers 1 to 6 and Volume 2, Numbers 1 and 2? If you are interested in our history, check out 99'er ONLINE, a disk prepared by Bob Burley featuring every article from our club newsletter up to March, 1983.

Figured out what to do with that Christmas cheque from Aunt Sally? Well here are some GOOD-BUYS: Edmonton Telephones has a limited supply of the Centonics 6LP printer for $199.00. It is fully compatable with the TI machine and features both a parallel and a serial port. Tractor feed is an option available for $35.00. Contact the 'et' Data Group. COSTCO is selling 20 pound clean edge tractor feed paper at $21.00 per box of 2500 sheets. Just the thing to go with your new printer.

Some of you have not yet paid your $20.00 hardware initiation fee. You are really not being fair to your fellow members who are supporting this group. How about it folks?

Another "TI-Cares" package was received and has been placed in our library. It is a set of diagnostic programs called TI-TEST designed to test out all parts of your system and attached TI peripherals. The disk is a "flippie" with one side running in XBasic and the other in MiniMemory. Be sure to sign out the documentation with the disk.

NEXT MEETING

The next meeting will be Tuesday, February 10'th at 7:15 PM. Same place as usual; General Services Building, U of A campus in room 849.

The door prizes promised for the December meeting that did not materialize were there for the January meeting. However, we forgot to hold the draw!!! I guess we got carried away with Bob Chapman's demo of MultiPlan and the general melee that tends to characterize the final stages of our meetings. In any case, the door prizes will be at the February meeting. Would somebody PLEASE remind us to get the job done right this time? And, if you have any suitable items that could be used for a door prize, bring it (them) along. We will be glad to give it away for you!

Planned for the February meeting is a demo of c99 by Paul Helwig; see other articals this month for more info. Also, the video tape of the Ottawa TI-Fair will continue if time permits.

NEWSLETTER DISKETTE AWARDS

This month's award goes to Michal Jaegermann who has written a rather extensive introductory artical on c99, the latest programming language for the TI-99/4A. If you are finding Basic to be just a bit too basic, you like some of the things that can be done with Assembly, and you find Assembly to be something devised by a madman that only the Saints can decipher, then perhaps c99 is the answer for you. It is a structured language like Pascal or Fortran that is compiled into Assembler code so you get all of the speed of the TMS 9900 chip at your finger tips. Best of all, you don't have to learn any Assembly or " be at one" with the archetecture of the machine. All you need to get started is one disk drive, Memory Expansion, the Editor/Assembler module and a copy of c99 available from our library. Pretty soon you will be doing some REAL programming!

At the February meeting, Paul Helwig will demonstrate some c99 programs and offer a brief introduction. Read Michal's artical and contemplate joining SIGc99 (Special Interest Group c99) which we will form if there is enough interest shown by our membership.

My thanks to Michal for an excellent artical. And don't forget to pick up your disk from me at the next meeting.

If you have an item that you would specifically like to be published here and you cannot get access to our newsletter disk system, simply leave a message or file on the board to my attention and I will see that you too get an award disk once your item is published. Thanks to all of you for supporting this newsletter.

FIRST STEPS INTO c99

by Michal Jaegermann

This series (hopefully) of notes is meant for everybody who has a copy of the two club library disks containing the c99 compiler. You have probably heard that this is a very flexible and powerful language, but now you are scratching your head. It is sure not like the old familiar BASIC; how do I start using it? If this is the case - read on. If not, read on anyway. Maybe you will find out that it is a good idea to get yourself a copy. Please remember that c99 is shareware, so if you are going to use it regularly, you are expected to contribute towards the financial well being of the author.

These notes will assume version 2.0 or higher of the compiler. If you still have the one-disk version 1.x, replace it as fast as you can. There is a big qualitative step between this and later implementations. By the time you read these words, Release 3.0, the next update which adds new features, should be available locally. So keep abreast of developments. (Editor's note: release 3.0 is now available and should be in the library shortly.)

Let me say first of all that these notes are not intended to replace the documentation that comes with the compiler; reading it is still obligatory. Also, these notes will not be a substitution for C textbook (check your local public library) which will help you fill out the documentation. But (very big BUT), please keep in mind when you are reading C texts that c99 is not C. It is a version of small-C, i.e. a C-like language, which implements only a small subset of the full language. It has syntax surprisingly close to an older brother but the semantics are often similar to what you will find in the texts (a big smile). But some differences do exist, so not everything you will find in a C textbook will be valid here. And c99 has some functions and libraries which are very TI99 specific. I will try to help a little bit with these problems.

After this long introduction let us see some programs. Probably the most famous one looks like this:

```
main ()
{
        puts ("Hello, world! \n");
}
```

This program, when compiled and executed, will display the greeting "Hello, world!" on the screen. Layout and indentaion of the program is up to you; the format is to make the program readable and is not required by the compiler. The same program could have been typed in as follows:

```
main(){puts("Hello, world! \n");}
```

and compiler will not care, as long as you do not break a line in the middle of a string to be printed. Unless you have very advanced stages of the BASIC disease, and really cannot contain yourself, do not use the second style. This is more important that it looks at the moment. If you are not convinced, then make a copy of any demo program from the c99 disk and reformat it with TI-Writer. You have a good chance that you will end up with valid code that wil execute once compiled. How about reading it?

Now, what does all that mumbo-jumbo mean? Simplifying a little bit - everything in a C program is either a statement, a declaration or a function. A statement just tells a computer to do something. And it has to be terminated with semicolon (;). Exercise: find a statement in the program above. A missing semicolon is a popular error in C programs. If the compiler comes back with a funny error message, check if you ended your statements properly. There is a good chance that the compiler is confused; it is attempting a compilation of a previous line against the programmers' intentions.

For all intents and purposes, every single statement can be usually replaced by a whole group of statements, provided you enclose them in a pair of braces like this { .... }. These braces define the limits of a program module and is treated as a single entity, much like a subroutine is in BASIC. Since our program above consists only of a single statement, then the braces can be skipped and the compiled code will be exactly the same. But it is a good idea to keep these extra embelishments. One reason is for readability. Another reason is that you will be able to use this module in a library and add it to another program in the future. Or perhaps you wish to expand this program by adding a second statement:

```
puts ("from my old, faithful TI99/4a. \n");
```

Now braces are necessary.

Now functions. Lets try to write a c99 function which will take a number from 0 to 9 and then print on the screen a corresponding digit. We should start with a good name. How about "putdigit". Names can be long and expressive. The only problem with c99 is that later the assembler will not accept more than the first six characters. The functions "putdigit" and "putdigit2" would end up with the same name. Names like "put2digit" will take care of this, but sometimes you have to move with a little bit of grace. Just make sure that the first six characters of function names are unique. Ok..., so putdigit has to accept, as an argument, one number. Therefore its header will look like this:

```
putdigit (number)
int number;
{
        /* the rest of "putdigit" goes here */
}
```

Here we told our compiler that the function will need one argument which is declared to be an integer ("int number;" is a declaration). Don't forget the semicolon. This is only information for the compiler so that it will deal with an object as an integer type. In c99 we may have only integers, characters and pointers to them. I'll leave pointers alone, for a while. Text between /* and */ is a comment. Comments may span many lines and nearly anything goes inside. Nested comments are, alas, not allowed in c99.

Back to our conversion program. We will use the fact that the ASCII values for digits are consecutive numbers starting with ASCII for zero. So the value for two is the value for zero plus two. We may write something as follows:

```
putdigit (number)
int number;
{
        int work;

        work = number + '0';
        putchar (work);
}
```

An expression "int work;" (semicolon - watch out) declares a local variable, which our function will use for internal purposes. Local means that the variable is invisible to all other parts of the program external to the function. And if another function uses a local variable with the same name, there will not be the slightest of correlations between the two variables. This is very different from BASIC but if you have used CALL type subroutnes in XBASIC, you will be familiar with this logic. This feature of C makes possible handling functions in a "black box" manner and the creation of libraries. Your own special purpose libraries in particular. Once you have defined an interface, then what is going on inside of a function is irrelevant, as long as the results are the same. All programs which use the function are not concerned with implementation details. For example, you may replace the body of putdigit with a single statement " putchar (number + '0');" without changing its meaning. The symbol '0' is just a replacement for a longer code which the computer uses to display a symbol for zero. You may put there, manually, a numeric value of the code, but you have to know which one it is. The computer has a big enough memory so we'l let it worry about it! This applies to any printable character. Non-printables can be referred to by their code values, but more popular ones have their own names - like, for example, \n, which stands for a "newline". Check your documentation.

"putchar" is the name of a predefined display function, which comes together with the compiler. Every C compiler in the world comes with a set of more or less standard functions which perform some basic tasks. The reason for that is that the language itself does not contain, for example, any I/O at all. Also other important tasks are left to library functions. It follows, that if you do not like the behavior of a function, you may reimplement this function in your own way. It also follows from that, that the power of a particular compiler depends very much on supplied libraries. This is one of the most important differences between different releases of c99.

I am talking all the time about functions, but, you can tell, usually a function takes some arguments and returns some value. We can see arguments, but what about results? That is correct. Every C function takes some number of arguments - starting from zero - and always returns something. You may not care what is returned, it may be a garbage, but something is returned always. How about "putdigit"? We may make it return information if an attempt to display a digit was succesful or not. Always a popular choice. Let see.

```
putdigit (number)
int number;
{
        if ((number < 0) | (number > 9)
                return (0);
        else
        {
                putchar ('0' + number);
                return (-1);
        }
}
```

Hope that this code is self explanatory. A vertical bar is a symbol for a "bitwise or", which behaves like OR in X-BASIC. The symbol for a "bitwise and" is &. Logical operators || and && are missing in c99. Therefore it is a good idea to use 0 - all bits off - and (-1) - all bits on - for boolean values.

Lets try some further modifications in order to define a function which will print a hex digit.

```
puthex (number)
int number;

        if ((number < 0) | (number > 15)
                return (0);

        /* if argument out of range, then function just                returned, so "else" clause is not
necessary */
```

```
        if (number < 10)
                putchar ('0' + number);
        else
                putchar ('A' + number - 10);

        /* everything ok...  brag about it */

        return (-1);
}
```

Here you have an example in a little bit different style. Once again, please remember that the indentation is for ease of reading only, so if you would like to execute more then one statement after the "if" clause, then braces would be necessary. The compiler will not second guess your intentions.

As an exercise, you could write a version of "putdigit" which will accept as arguments a number and a base and will either print the corresponding digit in a given base and return negative one, or it will return 0, if printing will be impossible.

Once we have a function, how about using it in a program. Every C program, by convention, has a function or module called "main" which calls up all the other guys required to perform the real job. Lets try something very simple

```
/* Great experimental program in c99.  It prints something.  */

#define LIMIT 20

main ()
{
        int i;

        for (i = 0; i < LIMIT; i++)
        {
                if (putdigit (i) == 0)
                {
                        puts ("~~THAT~~ was not a valid input!\n");
                        break;
                }
                else
                        putchar (' ');
        } /* end of "for" loop */
        exit (0);
} /* end of main */


putdigit (number)
/* Prints on screen a decimal digit, if possible.  Returns 0 if
   failed and -1 otherwise                                      */

int number;
{
        if ((number < 0) | (number > 9)
                return (0);
        else
        {
                putchar ('0' + number);
                return (-1);
        }
}
```

Comments: #define is equivalent to the LET statement in BASIC or the EQU directive in assembler. It lets you define a meaningful name to a constant. Also, if a constant was used in twenty different places in your program, and you decide to change it, it is enough to edit one #define directive, instead of hunting for all occurences of the number. For typographical reasons this is not evident, but "#define" has to start at the left margin.

"for (i = 0; i < LIMIT; i++)" is a familiar iterative (FOR-NEXT) loop, but with an important twist. The three statements inside tell what to do at the start, when to terminate, and what to do at every iteration. "i++" is equivalent to "i = i + 1", and just means to increment the loop index by one. You may use any form. The difference with this from other languages is that C allows for these three statements virtually anything. You may cram a whole program there, but do not overdo it. You may spent long hours deciphering your own smart ideas! On the other hand a construct "for (;;)" is a very popular C idiom and it means "loop forever". (Exercise: does it make any sense?)

"break", when encountered, leaves a loop immediately (only one at a time, if you have nested loops). For "exit (n)" function check your documentation.

IMPORTANT! The comparison operator "==" is different from an an assignement "=". If you wrote "0 = putdigit (i)" then you are trying to assign a return value of putdigit to a constant 0 and the compiler should barf on you. But if you tried to compare values of two variables using "=", then you will create a bug because you will have made the value of one of the variables equal to the other instead of comparing them.

Note that the argument name in a call to putdigit is a different one than in it's definition. That is ok just as it is in XBASIC subroutine CALLs. The compiler only has to know that it is going to pass one integer to a function. Names have to be consistent only inside one function - like main or putdigit. This feature greatly enhances the portability of functions via libraries.

After all this work,it is time to compile our Opus Magna. Please refer to your documentation for details. Let us only note here that the c99 compiler does not produce object or directly executable code; it produces 9900 source code which has to be assembled with an assembler such as that contained in your Editor/Assembler package, which is the mst popular one. If you wonder why the compiler does not create a runable object file, which would be possible in principle, there are a number of different reasons. Not all of them of the same validity.

One is a tradition. The first C compiler was written in an environement which stressed the philosophy of creating sets of specialized tools, which performed single tasks well and passed its efforts further down a line. Most C compilers have followed suit. Another reason, not independent, of course, is that a compiler written in that manner does not have to contain an assembler: why duplicate an existing assembler that works perfectly well? It needs a source code generator, that is true, but it does not have to know anything about invalid or multiple labels, unresolved references and all that stuff which is necessary for linking programs. We do not have a separate linker, and this is a lot of code. So it helps to keep the compiler small. Still another reason is that our arrangement makes possible an intervention into assembler input - either manually, by a programmmer, or by filter programs. An example of such is a code optimizer, which you will find on your c99 disk. Lastly, if there are some capabilities which you need badly and are difficult or impossible to get in a C function, one may write library functions in assembler, add them in an appriopriate spot and assemble the whole ball of wax together. This ensures great flexibility, though it requires some knowledge of assembler programming and c99 interfaces. But not everything at once.

So you compiled your program, removed all typos, assembled, found in the documentation how to make it run, and it runs and even is doing what was expected. Great! Try something else of your own invention and stay tuned for future instalments.