Beginning c99 - Part 1

Getting Started

By Vern Jensen

Let me start by introducing myself. I'm a long time TI owner and programmer, and many of you know me as the author of the Extended Basic games Maze Mania and The Castle, as well as the c99 game Virus Attack, each of which has been reviewed here in the Micro-Reviews column. I originally bought my TI back in 1987 as a "Game Machine", since I had heard that some games could be purchased for as low as $3 each. I then discovered TI-Basic, and have been a programming addict ever since.

Since I enjoy playing games, I naturally wanted to make games as well. I was thrilled when I discovered that with Extended Basic you can add sprites and other advanced features to your program. However, when creating Maze Mania and The Castle, I quickly ran into the limits of Extended Basic, the main one being speed. I spent much effort making The Castle run as fast as possible, and even added some assembly routines for special effects, but the game was still much slower than I wanted it to be. Also, since it ran as slow as it did, I couldn't add many features that I wanted to add to the game, such as enemies, moving platforms, and so on.

Naturally, I wanted a way to create games that avoided these problems; games that ran fast and could handle the features I wanted to implement. However, Assembly Language was not the answer, as least not for me. I had already tried learning it, and it was simply too complicated. I wanted something that was easy to use, but powerful. Could something like this possibly exist? And then I discovered it - c99, which is truly easy to use and also very powerful. However, I still had a problem: I didn't know C, and all the books that taught C were made for the IBM or Macintosh.

Since I had no way of learning how to use c99, I gave up on it for a while. Then after a while our family purchased a color Macintosh, and I also picked up a C compiler for it, which enabled me to learn how to use C on the Mac. Once I had learned C, it was easy to learn how to use c99, and I started programming my first game using c99, Virus Attack. If you don't believe that c99 is powerful, or you don't believe that it is fast, I suggest that you take a look at Virus Attack. It will change your mind if anything will! (Virus Attack is a shareware game similar to Dr. Mario that can be ordered from me for $15.)

Now that I know how to use c99, I'd like to share my knowledge with you, so that you too can become a c99 programmer without having to go through what I went through, and without having to buy a Macintosh. To make this process easier, I

have assembled a five-disk DSSD “c99 Starter Kit” that gives you everything you need to use c99 all in one easy to use package. The disks are very organized, each disk having its own category: 1) The c99 compiler, set up to load from Funnelweb, 2) Documentation, 3) Libraries, 4) Example programs, and 5) Some great c99 Libraries made by Bruce Harrison. Even if you already have the c99 compiler, I would strongly recommend that you order this disk set, since it is well organized, and contains the new c99 compiler version 5.0, which has been rewritten to go hand-in-hand with FunnelWeb, and also compiles faster than older versions. Also, this series will assume that you have these disks, since I can then give you precise step-by-step instructions on how to use the c99 compiler, giving you exact information on where you can find each file and so on. You can order this disk set from me by sending $5 (to cover S&H) to the address at the end of this article.

In order to use c99, it is recommended that you have at least two DSSD disk drives, and having a RAM-Disk or hard drive would help, although they certainly aren’t required. A printer is also highly recommended, as it can be very useful to print out your source code to help you track down bugs. Although it is possible to run c99 from only a single SSSD disk drive, I wouldn’t recommend it, since you would have to constantly swap disks to go back and forth between the editor, compiler, and assembler, as well as having separate disks for your source and compiled code. Also, my five disk “c99 Starter Kit” won’t work on a SSSD drive, so you’d have to order a copy of c99 from someone else, such as Tex-Comp. I myself have two DSSD drives, and get along just fine, although a RAM-Disk or hard drive would be nice, since it would greatly speed up the compiling and assembling process.

The rest of this article is written assuming you have ordered and received the “c99 Starter Kit” mentioned above. I suggest that you order this kit, and continue reading this article once it arrives. I will also assume that you have two DSSD disk drives. If you don’t, then it will be up to you to determine when you need to swap disks. This issue you will learn, step by step, how to type in a simple c99 program, compile it, and run it. Next issue we will start to compare c99 with Extended Basic and talk about the differences between c99 code and Extended Basic code, so you can easily start to learn how to write your own programs in C.

To get started, insert the c99 disk called “c99 Libraries” in drive 1, and a blank disk in drive 2. Then use a Disk Manager to copy the files CSUP, GRF1, and GRF1;H from drive 1 to drive 2. Now remove the c99 Libraries disk from drive 1 and insert in its place the disk named “c99 Compiler”. Leave your other disk in drive 2. Disk 1 is your “work disk”, and contains the editor you will use to write your source code, the c99 compiler which compiles it into assembly language, the assembler which translates the assembly language code into machine code, and the E/A loader, which lets you run your completed program. All of these programs are easily accessible from FunnelWeb. Disk 2 is where you will store your source code and the libraries needed by your program. Unlike Extended

Basic which has everything you need all ready to go, in c99 you have to load "libraries" that contain the functions you want your program to be able to use. The CSUP library you copied a moment ago from the Libraries disk is needed by all c99 programs, and the GRF1 library contains many functions for displaying graphics and sprites that are very similar to the familiar Extended Basic functions that perform the same task. We'll talk about libraries more in later issues.

Now that you have the libraries you need, boot up FunnelWeb (disk drive 1) using either the Extended Basic or the Editor Assembler cartridge. (To boot FunnelWeb using Editor Assembler, use option 5, "Run Program File", and type "DSK1.FW".) If you loaded from Extended Basic, press 2 ("Edit/Assm") from the main menu to get to FunnelWeb's Editor/Assembler menu. If you loaded from the Editor Assembler cartridge, this menu will automatically appear. You can toggle between Funnelweb's Editor Assembler menu and the TI Writer menu by pressing the space bar. You know you are looking at the Editor Assembler menu if the second menu option says "Assembler".

Once you get to FunnelWeb's Editor/Assembler menu, press option 1, "Program Ed" to load the program editor. This editor is similar to TI Writer, but it doesn't save information about the layout of the document, as this data could confuse the C compiler. When the editor has finished loading, type E and press enter to start editing the document. Then type in the source code for the c99 program listed after this article. Don't worry about typing in the wrong number of spaces between words, since c99 is very flexible and doesn't care how many spaces you leave between words. This gives you the freedom to adopt any coding style you wish. Also take note that there are both upper-case characters and lower-case characters, unlike Assembly Language and Extended Basic.

In case you're not familiar with the Program Editor, here is a list of some commands you can use:
FCTN-0: Toggle line numbers on/off
FCTN-1: Delete character
FCTN-2: Insert character(s)
FCTN-3: Erase line
FCTN-4: Move down a page
FCTN-5: Move right half a page
FCTN-6: Move up a page
FCTN-7: Tab
FCTN-8: Insert line
FCTN-9: Return to menu at top of screen
FCTN-E,S,D,X: Move cursor

Once you're ready to save the file, press FCTN-9 to return to the menu, then type SF (SaveFile) and press enter. When prompted for the filename, type "DSK2.TEST;C" and press enter. Once the file is saved, press FCTN-9 again to return to the menu, type Q, and press enter to quit. When asked if you really want to quit, type E (Exit), press enter, and you should be returned to the Editor Assembler menu. Now it's time to compile your program! Select option 4, "C-

Compiler", to load the c99 Compiler. Once it loads, you will be prompted for a response for several options. Press enter twice to select the default of "No" for the first two options, but type Y for the last one ("Assume Long Jump?"). There is no need to press enter after typing Y. These first three options will be explained in a future article, but for right now we just want to compile the program.

When prompted for the Input Filename, the name of the last file you worked with, "DSK2.TEST;C", should come up. This is what we want for the input file, so just press enter. Next you will be prompted for an output filename. Modify the text to read "DSK2.TEST;S". The "C" in the first filename means that the file is C source code, and the "S" in the second means the file is assembly language source code. After you finish typing in the output file name, press enter, and the file will be compiled. Since it is a very small program, it will compile very quickly. If there are any errors (hopefully there aren't), the C compiler will report them, in which case you should write them down and then open the file back up with the editor and correct any typos you have. I should also mention that the compiler only "sees" the first six letters of each variable and function name, and will only report the six letter version when reporting an error. So for instance if an error were reported on a line that has the variable "keyCode", the compiler will only display "keyCod". Don't let this confuse you. When the compiler is finished, type N (No) when prompted if you want to rerun it.

Once back at the Editor/Assembler menu, select option 2, "Assembler", to assemble the program. Once it loads, all of the fields should be filled out for you. The source file name should already read "DSK2.TEST;S" and the object file name should read as "DSK2.TEST;O". You don't need to worry about the "List device name" field; just leave it blank. Press enter three times to get down to the options field, and press FCTN-1 once to delete the "R", which should leave you with only a C. The R stands for Registers, which c99 code doesn't use. Removing the R should make it assemble slightly faster. The C stands for Compact, which will make your code smaller, so we will leave that option on. Press enter again, and then FCTN-6 to proceed. Or, if you made a mistake, you can press FCTN-8 to change the fields. When the assembler is finished, press enter to return to the Editor Assembler menu. (It should take about as long to assemble as it does to compile.)

Now it's time to run the program! Type 3, "Loaders", to get to the E/A Loaders menu. Then type 4 for "Load and Run". You should then see "DSK2.TEST;O" on the screen. Press enter, and the object code for your program will be loaded. Now you need to load the c99 support library and the graphics library. Press FCTN-3, type "DSK2.CSUP" and press enter, then press FCTN-3 again, type "DSK2.GRF1", and press enter. Next, clear the text by pressing FCTN-3 once more and then enter, leaving the line blank. Now you should be presented with a screen containing all the names you can use to try to start the program. Move to the name "START" by using the arrow keys (FCTN and E-S-D-X) and press FCTN-6 (Proceed). If you accidentally hit enter instead, just hit it a few more times to get back to the first screen. If everything goes well, you should see the screen turn purple and then a blast of characters being displayed on the screen.

(Pretty fast, isn't it?) Congratulations, you've just run your first c99 program! (Don't worry, the next program will be a little more interesting.) When you're done watching it, press FCTN-plus to reset your computer.

Next time, you'll start to learn how to make your own programs in C. I will show you a more interesting c99 program and an Extended Basic program that does the same thing, and discuss what is different about writing the C code. This will teach you the differences between C and Basic, which will enable you to quickly understand how to write C code. I'll also explain what some of those compiler options do, tell you an easier way to load your programs, and describe how to create c99 programs that can be loaded from E/A Option-5.

If you have any questions or comments, I'd love to hear them. Just send them to the address below, or you can email me at Vern_Jensen@lamg.com.

Vern L. Jensen
910 Linda Vista Ave
Pasadena, CA 91103

```
#include "DSK2.GRF1;H"


main()
{
  int chr, keyCode, status;

  Grf1();
  Screen(14);

  chr=33;

  do
  {
    HChar(1,1,chr,768);
    chr++;

    if (chr >= 126)
      chr=33;

    keyCode = Key(0,&status);

  } while (status == 0);
}
```

Beginning c99 - Part 2

Comparing c99 with Extended Basic

By Vern Jensen

Last issue you learned how to type in a c99 program and compile it. This issue we'll take a look at the differences between c99 and Extended Basic by comparing a c99 program and Extended Basic program that do the same thing. You'll also learn an easier way to load a c99 program, as well as how to create E/A Option 5 Program files.

One of the first things you should do is to print some of the documentation that comes on the "c99 Documentation" disk. You should print all four of the C99-DOC files, which tell how to use the c99 compiler, and GRF1DOC, which describes each of the routines in the GRF1 library (a library for displaying graphics, and sprites). Some other files you might also want to print out are CLOADDOC, RANDOMDOC, SNDSAYDOC, and SOUNDOC. Use FunnelWeb's Formatter (option 2 on the TI Writer menu) to print the documentation. In addition, the file "C99-README" tells what's new in version 5.0 of the c99 compiler. If you previously used version 4.0, you may want to take a look at this. And if you want to know what any of the other files on the documentation disk are for, simply take a look at the "-README" file. There are also "-README" files on the "c99 Libraries" and "c99 Programs" disks, so if you ever wonder what a particular file is for, just take a look at the "-README" file on that disk.

Something I should clarify is that the $5 I charged for the c99 disk set was to cover S&H only; it doesn't pay for the compiler itself. If you use the c99 compiler, you should send Clint Pulley the $40 shareware fee. This is a very low price when you consider how much work he must have put into this excellent program. Clint Pulley's address is listed in the first C99-DOC file.

To get started, type in the program at the end of this article called "TEST2;C", and save it onto Disk 2 (the one you've previously saved c99 programs onto). When you are done, press FCTN-9 to return to FunnelWeb's menu, and type Q. However, this time, instead of hitting E to exit, type P to Purge. Then type in the file called "TEST2;L" (also listed at the end of this article), and save it to Disk 2. Then Purge the contents of the editor again, and type in the file called "TEST2;P", and save it to Disk 2 as well. We will use the TEST2;L file to load the program later, and we'll use the TEST2;P file to create an E/A Option 5 Program file.

Compiler Options

Once you're finished typing in and saving those files, exit the editor and start up the c99 compiler, and compile the TEST2;C program, using the same options as we did last installment. (No for the first two options, Yes for the third.) The first option, "Include C Source?", if turned on, will include the C source code as comments in the assembly language output file. This can be useful for

debugging, since if you get an assembly error for a particular line, you can load the assembly code and see which C line is responsible for the problem. This option can also be used just for fun, to see how many assembly language lines are produced by a single line of C code. However, since this option makes the output file slightly larger, you should normally leave this option off. (Pressing enter will accept the default response of No.)

The second option, “Inline Push Code?", will make the assembled program slightly faster, but the program will also be larger as a result. Since the size of the program is generally more important than a little bit of extra speed (at least for large programs), you’ll usually want to leave this option off as well.

The third option, “Assume Long Jump?", is a new option that appeared in version 5.0 of the compiler. It was an option that used to be built into the c99 optimizer, but Winfried Winkler, who updated the c99 compiler to version 5.0, decided to put this option in the compiler itself, since it is much easier for the compiler to do it than for the optimizer to do it. You should respond to this option with a “No" only when you are finished your program and are ready to optimize it, since, although it can make your program smaller and faster, it also requires a bit more work on your part before the program can be run. This option is completely documented in the file “J2BDOC" on the documentation disk, so I won’t go into it here.

The C-LOADER

Now start up the Assembler and assemble the TEST2;C program. You should be able to simply hit enter four times and then press FCTN-6, since FunnelWeb fills in all the fields for you. (Pretty neat, huh?) Once it is done assembling, instead of hitting 3 once back at the FunnelWeb menu to get to the Loaders menu, hit 5 for C-LOADER. The C-LOADER is a great program by Tom Bentley that greatly simplifies the process of loading a c99 program. Instead of having to remember all of the files that need to be loaded (such as CSUP, GRF1, and TEST2;O), we can simply type in the name of a file that contains the names of all the files that need loading, and C-LOADER will do all the work for us. So once the C-LOADER has loaded, type in “DSK2.TEST2;L" and press enter, and sit back and watch it go to work! When it is done, press Enter to exit C-LOADER, then hit FCTN-3 to erase the text, and press enter to get to the “Program Name" screen. Select “Start" and press FCTN-6, and the program should start. Use the arrow keys to move the sprite around the screen, and the period key to change the sprite’s color. Since two different keyboard scan codes are used to scan for the arrow keys and the period key, you can change the color of the sprite while moving it around.

When you’re done playing around with the program, hit Q to quit. You will then be asked if you want to rerun the program . Type Y if you do, or N if you don’t. This prompt to rerun the program is something that is built into c99; it can be useful if you want to stop a program and start it again from the beginning without having to reload it, which can be great for testing purposes. There are ways to disable this feature, however, if you should so desire. This is covered in the c99 manual, and we’ll probably cover it in a future article as well.

If, after quitting the c99 program, your computer locks up, it is because the c99 program was loaded “on top" of FunnelWeb in memory, and therefore when the TI tried to return to the FunnelWeb program, it locked up. This won’t happen

if you load the c99 program from the Editor/Assembler cartridge, because the E/A program is kept on the cartridge, not in RAM, and therefore can't be overwritten. This isn't terribly important because most programs don't have a "Quit" option anyway; they simply force you to reset your computer when you want to stop the program.

Creating an Option 5 Program File

I did promise that we'd cover how to create an Option 5 Program file, so let's go ahead an do that now. However, before it can be created, you need to copy three files from the c99 Libraries disk over to your "Work disk" (the one in drive 2). The files that need to be copied are C99PFF, C99PFI, and SAVE. In order to create an E/A Program file, you must load all the files you normally load when you want to run a program, but must "sandwich" them between the C99PFF and C99PFI files, and then load SAVE after everything else.

Go back into FunnelWeb and start up the C-LOADER once more (option 5 on FunnelWeb's Editor/Assembler menu). This time, however, type in "DSK2.TEST2;P" and press enter. This not only loads the program and all of the libraries it needs, but it also loads the files needed in order to create an Option 5 Program file. Once it is finished, press enter, then FCTN-3, and then enter again to get to the Program Name screen. But this time do not select START! Instead, you want to find the program name SAVE, which most likely won't be on the first screen of names. To get to it, press enter a few times until you see SAVE, and then use FCTN-D (the right arrow) to select it, then hit FCTN-6. This will take you to the screen for saving an Option 5 Program file. Simply type in the name you want the program file to be saved as (such as "DSK2.TEST2") and press enter. That's it! You can then run this program as much as you like, by simply using the E/A Option 5 "Run Program File".

A Look At The Source Code

Probably one of the first things you noticed when you typed in your first c99 program is that there are no line numbers! If you're an assembly language programmer, this won't seem very strange, but to an XB programmer, this can be quite a shock. (Note that the line numbers displayed by the editor are for your reference only; they aren't saved as part of the file.) You may wonder, "But how can I write a program without any GOTO statements?". Don't worry; c99 has many flow control statements that Extended Basic doesn't, and they more than compensate for the lack of a GOTO statement. Actually, c99 does have a GOTO statement, but it's generally considered bad programming practice to use it, since it can make your code hard to read. And once you get used to C's flow control statements, you'll never want to go back to using GOTO again.

Just as Extended Basic programs start execution at the lowest line number, c99 programs start execution in the main() function. In c99, everything happens inside "functions". These are very similar to the Extended Basic's subroutines, except that in C, the entire program is contained in functions, with "main()" being the function that is called automatically when the program first starts. (For this reason all C programs must have a main() function.) Functions can also accept parameters, just like XB subroutines. We'll cover functions in a future c99 article.

Another thing you'll notice about C code is the use of open braces ("{") and close braces ("}"). These are used to group several statements together, or to define a function. For instance, following main(), there is an open brace, and

then at the end of main() there is a close brace. Everything in between these two braces make up the main() function. Braces are not only used for functions, but for many other things as well, such as if statements. Here's an example of an if statement that doesn't use braces:

```
if (numLives < 1)
  dead = true;
else
  dead = false;
```

But what if we wanted execute more than one statement if numLives is smaller than 1? Then we could write the code like this:

```
if (numLives < 1)
{
  dead = true;
  score = score - 50;
  DoGameOver();
}
else
  dead = false;
```

By using braces, we were able to group three statements together, so that "dead" would be set to true, "score" would be decremented by 50, and the function DoGameOver() would be called if the variable numLives were smaller than 1. Just as FOR and NEXT statements in Extended Basic can be nexted, so can braces be nested. The only rule is that every open brace must be matched with a close brace.

Another thing that XB programmers will notice about C code is that each statement ends with a semicolon, such as the statement "x = 1;". This is necessary because the C compiler totally ignores all spaces, so you could put more than one statement on a single line if you want. Because of this, there must be some way for the compiler to know where the end of a statement is, and the semicolon is used for this. As an example, the code above could have been written like this:

```
if (numLives<1) { dead=true; score=score-50; DoGameOver(); } else dead =
false;
```

This is exactly the same code as far as the C compiler is concerned. However, this is obviously much harder to read than the previous example. Therefore, you are free to write your code in whatever style you prefer; however, whatever style you adopt, you should be consistent, and should choose a style that makes the code easy to read. As an example of another method, some people prefer to write their code like this, putting the open brace at the end of the first line:

```
if (numLives < 1) {
  dead = true;
}
```

However, I prefer the other approach, since it makes it easy to visually line up the open and close braces.

Another main difference between Basic code and C code is that in C code, you have to tell the C compiler what variables you are going to use before you use them. Basic does this for you by scanning through your code before your

program is run and making a list of all the variables. In today's example program, the following lines at the beginning of the main() function tell the C compiler what variables we will be using:

```
int keyCode, status, myColor;
int rowVel, colVel;
```

The word "int" at the beginning of those statements tells the compiler that those variables are integers. There are two types of variables provided by c99: char and int. The char variables can contain any value from 0 to 255, and integers can contain any value from -32768 to 32767. Char variables take up half as much memory as an int, although the size of a variable doesn't matter much when you only have a few variables, like today's program. It is important to realize that these variables can only contains whole numbers; trying to assign a char or int a value such as 1.5 won't work. (There are ways around this however, such as using fixed-point numbers. We may cover this in a future article if there is interest.)

Variables must be declared either at the beginning of a function, or outside of a function. If they are declared inside a function, they are called "local" variables, because they are "local" to that function; they can't be "seen" by any other function. However, if a variable is declared outside of a function, such as at the beginning of the program (before main()), it is called a "global" variable, because all of the functions can access it. As I mentioned last issue, the c99 compiler only recognizes the first six letters of any name, be it a variable name, function name, or a definition name. You can make your variables longer than six letters in order to make them more readable, but you must remember than only the first six letters are used by the compiler. So if you make two variable called "myValue1" and "myValue2", you will run into problems, because the compiler will turn them both into "myValu". Variable and function names can be any mix of letters and numbers, but they must start with a letter. In normal C, capitalization does matter, so "myguy" is different from "myGuy", but in c99, your code is compiled into assembly language, where everything is in upper-case, so when writing c99 code, don't rely on capitalization to keep your variable names, function names, and definitions apart, since in c99 "myguy" and "myGuy" are both compiled into "MYGUY".

I'll explain two other things today and then call it quits. The first is that in C, to compare two values to see if they are equal, you must use two equal signs, like this:

```
if (x == 1)
    DoMyStuff();
```

This is because if you used only one equals sign, the value would be assigned to x instead of being compared with it, even though this is being done inside an if statement. In C, "x = 1" always means "assign 1 to x", no matter what the context is.

The second thing I'll explain is that whenever you see a character sandwiched between two apostrophes, that character will actually be converted by the C compiler into its ASCII equivalent, which is very similar so Extended Basic's ASC() function. As an example, 'A' would be converted by the C compiler to 65. The great thing about this is that this conversion is done during compilation, before the program is run, so using a statement such as "if (x ==

'A')" is just as fast as using "if (x == 65)", but is often much easier to read and use, since you don't have to memorize the ASCII value for A. This is just one of the many nice little "perks" of the C language.

If there is any other part of the C code in today's example that you don't understand, just take a look at the section of the Extended Basic code also at the end of this article that does the same thing. Today's example program was written in both languages in order to give you something familiar to compare with something that is unfamiliar. Also take a look at the GRF1DOC; it explains some of the stuff you'll see in the c99 code (such as Grf1() and SpMct(1)) that isn't in the Extended Basic code. And finally, feel free to experiment; the c99 compiler won't bite! If you want to try something out, go ahead! Experimenting will do much help you learn.

In Conclusion

Today I've covered many aspects of the C language, and hopefully removed some of the mystery of it. I will continue to compare C code with XB code for the next few articles in order to get you up and running with C as quickly as possible, but there is no way I can cover everything about the language in just a few articles. Therefore, I strongly suggest that anyone interested in learning C purchase the book The C Programming Language by Brian Kerninghan and Dennis Ritchie. The great thing about this book is that it is short and to the point; you won't have to read 500 pages just to figure out how to assign a value to a variable. It also makes a great reference book. There will be some things in the book that won't work with the c99 compiler, but much of what the book has to say does apply to c99. In any case, I've found the book to be an indispensable reference, and I highly recommend it.

Contacting Me

Have any questions about c99, suggestions for future articles, or comments about past articles? Or maybe you want to take a look at my game Virus Attack, which was written completely in c99? Don't hesitate to write. You can contact me via email at Vern_Jensen@lamg.com, or via snail mail at Vern L. Jensen, 910 Linda Vista Ave., Pasadena, CA 91103.

TEST2;C

```
#include "DSK2.GRF1;H"

main()
{
    int keyCode, status, myColor;
    int rowVel, colVel;

    Grf1();
    Screen(8);
    myColor = 5;

    Display(10,7,"Use the arrow keys");
    Display(11,7,"to move the sprite.");
    Display(13,3,"Press '.' to change color.");
    Display(24,8,"-Type Q to Quit-");

    ChrDef(64,"FFFFFFFFFFFFFFFF");
    SpMag(2);
    Sprite(0,64,myColor,80,112);
    SpMct(1);
```

```
   do
   {
      keyCode = Key(2,&status);
      if (keyCode == 13)
      {
         myColor++;
         if (myColor == 8)
            myColor++;
         else if (myColor > 16)
            myColor = 2;

         SpColr(0,myColor);
      }

      keyCode = Key(0,&status);
      if (keyCode >= 'a' && keyCode <= 'z')
         keyCode = keyCode - 32;

      rowVel = 0;
      colVel = 0;

      if (keyCode == 'E')
         rowVel = -16;
      else if (keyCode == 'X')
         rowVel = 16;
      else if (keyCode == 'S')
         colVel = -16;
      else if (keyCode == 'D')
         colVel = 16;

      SpMotn(0,rowVel,colVel);
   } while (keyCode != 'Q');
}


Display(theRow, theCol, string)
char theRow, theCol, *string;
{
   Locate(theRow,theCol);
   PutS(string);
}
```

TEST2;L

```
DSK2.TEST2;O
DSK2.CSUP
DSK2.GRF1
```

TEST2;P

```
DSK2.C99PFI
DSK2.TEST2;O
DSK2.CSUP
DSK2.GRF1
DSK2.C99PFF
DSK2.SAVE
```

TEST2;XB

```
100 CALL CLEAR
110 CALL SCREEN(8)
120 MYCOLOR=5
130 DISPLAY AT(10,7):"Use the arrow keys"
140 DISPLAY AT(11,7):"to move the sprite."
150 DISPLAY AT(13,3):"Press '.' to change color."
160 DISPLAY AT(24,8):"-Type Q to Quit-"
170 CALL CHAR(64,"FFFFFFFFFFFFFFFF")
```

```
180 CALL MAGNIFY(2)
190 CALL SPRITE(#1,64,MYCOLOR,80,112)
200 !
210 CALL KEY(2,KEYCODE,STATUS)
220 IF KEYCODE=13 THEN 230 ELSE 260
230 MYCOLOR=MYCOLOR+1
240 IF MYCOLOR=8 THEN MYCOLOR=MYCOLOR+1 ELSE IF MYCOLOR>16 THEN MYCOLOR=2
250 CALL COLOR(#1,MYCOLOR)
260 CALL KEY(0,KEYCODE,STATUS)
270 IF KEYCODE>=ASC("a")AND KEYCODE<=ASC("z") THEN KEYCODE=KEYCODE-32
280 ROWVEL=0 :: COLVEL=0
290 IF KEYCODE=ASC("E")THEN ROWVEL=-16 ELSE IF KEYCODE=ASC("X")THEN ROWVEL=16
300 IF KEYCODE=ASC("S")THEN COLVEL=-16 ELSE IF KEYCODE=ASC("D")THEN COLVEL=16
310 CALL MOTION(#1,ROWVEL,COLVEL)
320 IF KEYCODE<>ASC("Q")THEN 210
```

Beginning c99 - Part 3

The If-Else Statement

By Vern Jensen

In the last two articles in this series, I've covered a lot of things very briefly, in order to get you up and running with the c99 compiler as quickly as possible. Now that you know how to type in and compile a program, and are familiar with some of the similarities between c99 and Extended Basic, we'll start taking things a little slower, examining each part of the language in detail.

But before we get started, I'd like to make available to you a new disk that I recently acquired. When I made my c99 Starter Kit available, a few people sent along a disk containing programs they had written along with their $5 payment for the c99 disk set. I got two games that way, as well as a special bonus: a disk containing several TI-Basic programs that had been converted to c99. It turns out that these programs, for the most part, had been converted to c99 with a special compiler the author had written that compiles TI-Basic programs into c99 code. The compiler is not finished, but the author sent me a copy anyway, and I'm very impressed with it. He's currently working on improving the compiler, which includes things such as adding XB support.

Even though I can't distribute the compiler (since it isn't finished), I can distribute the disk containing the TI-Basic programs that have been converted to c99 code. The programs range from a simple demo (just a few lines) to a full-blown mine sweeper type game. I think this disk could be quite helpful to those new to c99, since it provides several examples of c99 programs that actually compile, as well as Extended Basic source code that you can print out and compare with the c99 code. And those who already know c99 still might find the disk interesting. (I know I did.) In any case, the disk can be ordered from me for $3. (I'd normally charge less for just one disk, but I've decided to order 50 floppy disk mailers for this purpose, since the manilla envelopes I used to ship the c99 Disk Kit wouldn't be sturdy enough to ship a single floppy disk, and charging $3 will help me to cover that expense.)

Using the c99 Compiler With a RAMDisk

Recently someone asked me if I happened to know what files need to be copied from the Funnelweb disk included with my c99 Starters Kit to a RAMDisk in order to use the c99 compiler with the Funnelweb program they already had set up on their RAMDisk. The only files you should need to copy are CC and CD (the c99 compiler), and CL (the C-Loader). Configure your copy of Funnelweb to load CC as a TI-Writer Program File, and CL as a Load/Run file. That's it!

Getting Started

One of the first things you have to learn about any language is how to make decisions based upon certain conditions. This is accomplished with if-else statements. The syntax of an if-else statement in c99 is

```
if (expression)
   statement1;
else
   statement2;
```

If `expression` is true, `statement1` is executed; if it is false, `statement2` is executed. Here's an example of a very basic if-else statement:

```
if (value > 99)
   value = 99;
else
   value = 1;
```

This statement sets `value` to 99 if it was above 99, and sets `value` to 1 if `value` was not above 99. The same code in Extended Basic would look like this:

```
IF VALUE>99 THEN VALUE=99 ELSE VALUE=1
```

The main difference between the XB code and c99 code is that in c99, the expression (`value > 99`) is contained within parentheses, and each statement ends with a semicolon. If statements do not need to have an "else"; that part is optional, just like in Extended Basic. In addition, you can add "else if"s to a statement, also just like you can in Extended Basic:

```
if (value > 99)
   value = 99;
else if (value < 0)
   value = 0;
else if (value == a)
   value = 0;
else
   a = 1;
```

You can add as many "else if" statements to the chain as you want, ending it with an "else" statement at your option. By now you may be wondering if it is possible to execute more than one statement if a particular condition is true or false. The answer is yes. As we've mentioned before very briefly, you can "group" statements together with braces. Here's an example:

```
if (a > b)
{
   a = b;
   b = 0;
   z = 1;
}
else if (b > a)
{
```

```
   b = a;
   a = 0;
   z = 2;
}
```

By using open and close braces, we were able to group three statements together, so the first group will be executed if a is larger than b, or the second group if b is larger than a. You can group together as few or as many statements as you like. In fact, you can even put zero statements within the braces, and the program will still compile just fine. This can be useful if you're going to put something there later, but want to work on other things first.

Expressions

One interesting characteristic of the if statement is that it simply tests the numeric value of the expression. An expression such as "a > b" will have a value of 1 (true) if a is larger than b, or a value of 0 (false) if a is not larger than b. Interestingly enough, you can assign the result of an expression to a variable, so a statement like "n = a>b;" will set n to 1 if a is larger than b, or 0 if it is not.

This means that you could put variables or even numbers in an if statement, instead of an expression, since an if statement simply checks to see whether the value contained between the parentheses is true (non-zero). Here's an example of an if statement that uses a variable, instead of an expression:

```
if (n)
   a = n;
```

This will set `a` to `n` if `n` is non-zero. I'm not sure if it is well known, but the same thing can also be done in Extended Basic:

```
IF N THEN A=N;
```

However, I don't think Extended Basic lets you assign the result of an expression to a variable, such as "n = a>b". In c99 you have the flexibility to do this. You might wonder how this could be useful, but there are cases every now and then. One example would be if you wanted to make a complex (and time-consuming) comparison, and save the results of that evaluation in a variable for use in more than one subsequent if statement.

Operators

So far I have used only a few types of operators in my if statements. By now you may be wondering if c99 can do everything that Extended Basic can. Rest assured; the C language has a whole suite of operators that can be used in your flow control statements. Here is a list of them:

==	(Equal to)
>	(Greater than)
<	(Less than)
>=	(Greater than or equal to)

<= (Less than or equal to)

! (Not - turns a true value false and vice versa)

!= (Not equal to; same as <> in XB)

|| (Or - FCTN-A on the keyboard)

&& (And - shift-7 on the keyboard)

Most of those are pretty self-explanatory. The last two, the OR and AND operators, are similar to the OR and AND statements of Extended Basic. For instance, the following statement will decrement k by 32 if k is larger than 96 and smaller than 123:

```
if (k>96 && k<123)
   k = k-32;
```

This would be the same as the following Extended Basic code:

IF K>96 AND K<123 THEN K=K-32

I used this statement in Virus Attack after reading the keyboard to convert lower case ASCII values to upper case ASCII values, so the keys would report the same values regardless of whether the Alpha Lock key was down.

Make sure to use two symbols when using AND and OR operators! The statement "if (a>1 && b>1)" is quite different from "if (a>1 & b>1)". I won't get into the & and | operators right now. They are bitwise operators, and will probably be covered in a future article. (Bitwise operators are used to perform operations on a number in binary format, such as shifting all the bits in a number left one digit.)

Take note that the "equal to" operator ("==") is also made up of two characters. Using only one, such as in the statement "if (c = 5)", would assign 5 to c and then execute the statement since c would be non-zero, instead of comparing c with 5 and executing the statement if the two are the same. You see, in C, the code "c = 5" always means "assign 5 to c", regardless of whether that code is used in an if statement or not, just like the code "c == 5" always means "compare c with 5", regardless of whether it is used in an if statement or not. Using two equal signs instead of one when comparing values is probably one of the hardest habits to break when starting to write c99 code.

The "!", or logical negation operator, turns a non-zero operand into 0, and a zero operand into 1. So a statement such as

```
if (!value)
   statement;
```

would execute the statement if value is *not* true (that is, if value is equal to 0). The ! operator can also be used with expressions, such as

```
if ( !(a > b && c == 1) )
   statement;
```

This first evaluates the expression contained within the parentheses, and if this expression is *not* true, then the statement will be executed, since the ! operator turns a false value into a true value. However, the most common use of the !

operator is simply to determine if a variable is zero. You may wonder why you wouldn't simply use "if (value == 0)" instead. The answer is that by using the ! operator, you can sometimes more clearly indicate the action of the if statement.

For instance, you might have a variable that keeps track of whether sounds are to be played during a game, called something like `playSounds`. You would assign 1 (true) to the variable if sounds are to be played during the game, or 0 (false) if they are not. Then during the game you can easily write two kinds of if statements:

```
1) if (playSounds)
   statement;
2) if (!playSounds)
   statement;
```

The first statement could be used to actually play the sounds if that option is turned on. The second statement could be used if you wanted to perform some other action when the sounds are not turned on. This statement is read as "If not playSounds, do statement." You can see how using the ! operator in this case would be a bit clearer than writing "if (playSounds == 0)".

Order of Evaluation

If statements are evaluated from left to right, and evaluation is terminated as soon as the outcome of the expression is known. For instance, in the statement

```
if (a > 99 || a < 0)
   statement;
```

`a` is first compared with 99, and if `a` is larger than 99, then the statement is executed immediately, without `a` being compared with 0. This means that you can optimize your if statements by placing the most likely case at the beginning of the statement, where it will be evaluated first.

You can use parentheses to group expressions together. For instance, the following code executes the statement if `playSounds` is true (non-zero) and `volume` is either 1 or 2:

```
if (playSounds && (volume == 1 || volume == 2))
  statement;
```

If the parentheses weren't used, the statement would be executed if `playSounds` was true and `volume` was equal to 1, or if volume was equal to 2. This means that if volume was equal to 2, the statement would be executed, regardless of whether `playSounds` was true or not. Putting in the parentheses avoids this problem. Another way of writing the example above would be

```
if (playSounds)
   if (volume == 1 || volume == 2)
      statement;
```

Nested If-Else Statements

If statements can be nested. When they are, it can sometimes be confusing which else statements match which if statements. The rule here is that each else statement matches the closest if statement that does not already have an else statement associated with it. For example, in the code below, the first else statement is associated with the inner if, and the next else statement is associated with the outer if, which I have shown by indentation:

```
if (value == 1)
    if (a == 65)
        b = 1;
    else
        b = 2;
else
    b = 3;
```

However, you must remember that the C compiler ignores indentation. A statement such as the one below would not do what you expect, if you want the else statement to be executed when the first if statement is false:

```
if (value == 1)
    if (a == 65)
        b = 1;
else
    b = 3;
```

In order to achieve the desired association, braces may be used, as in the following example:

```
if (value == 1)
{
    if (a == 65)
        b = 1;
}
else
    b = 3;
```

Now the else statement will be associated with the first if statement. As a general rule, it's a good idea to use braces with your nested if statements whenever necessary to maintain clarity, even if the braces aren't needed to make the code operate as desired.

Next installment we'll take a look at the other flow control statements, such as the for, while, do while, and switch statements. These will be easy to learn now that we've covered the if statement. Then in future articles we'll take a look at functions, pointers, arrays, definitions, and others fundamental parts of the C

language. Then we'll use all these things to write a complete game in c99. After that, I'll offer some tips on debugging c99 programs, programming practices that will help you avoid bugs in the first place, as well as any special interest topics I can come up with.

Contact Information

If you have any questions or comments, feel free to email me at Vern_Jensen@lamg.com, or write to me at Vern L. Jensen, 910 Linda Vista Ave., Pasadena, CA 91103.

Beginning c99 - Part 4

Flow Control Statements

By Vern Jensen

One of the nice things about C is that it has many flow control statements that Extended Basic doesn't have. These statements give you much more flexibility than XB's statements, and often help to make the code much clearer. Last issue we took a look at the `if-else` flow control statement. Now that you have that under your belt, we can take a look at the rest of them: the `for, while, do-while,` and `switch` statements. We'll also cover the `break`, `continue`, and `goto` statements.

The While and Do While Statements

One of the most basic flow control statements in c99 is the `while` statement. The format of a `while` statement is

```
while (expression)
     statement;
```

First `expression` is evaluated, and if it is true (non-zero), the `statement` is executed. The `expression` is then re-evaluated, and this cycle continues until the `expression` becomes false (zero). Here's a very basic example of a `while` statement:

```
while (status != 1)
     keyCode = Key(0,&status);
```

This continually calls the `Key` function (part of the GRF1 library) until the `status` returned by the function is 1, indicating that a new key was pressed. However, one must remember that `status` is evaluated before the call to `Key` is made in the first place, which means that if `status` was already equal to 1 before this code was encountered, this code would be skipped, since the evaluation would fail before `Key` could even be called. There is a way around this problem, however, and it's by using the `do-while` statement. This statement allows you to execute the `statement` before doing the evaluation. The format of a `do-while` statement is

```
do
     statement;
while (expression);
```

So our code above could be written like this:

```
do
     keyCode = Key(0,&status);
while (status != 1);
```

This would correctly make the call to `Key` before `status` is evaluated, so that it won't matter what value is contained in the `status` variable before this code is executed. Because the statement is executed before the evaluation is

made, the statement part of a `do-while` loop will always be executed at least once, even if the expression is false. It is important to note the semicolon that is placed at the very end of the `do-while` statement; you'll get compile errors if you leave it out.

And as you'd expect, you can group multiple statements together for use in the various flow control statements by using braces. For instance, we could write

```
do
{
     keyCode = Key(0,&status);
     HChar(1,1,keyCode,768);
} while (status != 1);
```

The semicolon at the very end of the `do-while` statement is still necessary even when you use braces to group multiple statements together. And in case you're wondering, let me also say that the `expression` in a `do-while` statement (or just about any other flow control statement, for that matter) is just the same as the `expression` in an `if` statement. That is, anything we talked about last article dealing with the `expression` part of an `if` statement applies here as well. So another example of the while statement would be this:

```
gameOver = 0;
while (!gameOver)
     PlayGame();
```

This code repeatedly calls the function `PlayGame` until the `gameOver` variable is set to true (1).

The For Statement

Now we come to a flow control statement you're most likely already familiar with: the `for` statement. The format of the `for` statement is

```
for (expr1; expr2; expr3)
     statement;
```

which is functionally equivalent to

```
expr1;
while (expr2)
{
     statement;
     expr3;
}
```

An example of a simple `for` statement would be

```
for (chr=65; chr <= 128; chr++)
     HChar(1,1,chr,768);
```

This would fill the screen with all the ASCII characters from 65 to 128. I should mention that the "++" is an "increment operator"; the statement "chr++" is the same as "chr = chr+1", but is more efficient. The equivalent of the example above in Extended Basic would be

```
FOR CHR=65 TO 128 :: CALL HCHAR(1,1,CHR,768) :: NEXT CHR
```

In practice, the first expression in a `for` statement is an initializer (such as "`n=1`"), the second expression is an evaluation (such as "`n < 100`"), and the third expression increments the variable used in the loop by a value (such as

"`n = n+5`"). However, these expressions don't have to be this way. Each expression can be any kind of statement you want to put there, although you must remember that the first one is only executed once, the second one should be some sort of evaluation, and the third is executed after each pass of the loop. You also have the option of leaving out any or all of the expressions if you so desire; for instance, here's a `for` loop that omits the first expression (although you must leave the semicolon there):

```
for (; n < 100; n=n+5)
     statement;
```

This would be equivalent to

```
while (n < 100)
{
     statement;
     n=n+5;
}
```

You can also add more than one statement to each expression by separating the statements with commas. For instance:

```
for (a=low,b=high; a < b; a++,b--);
```

In this example we have two statements in the first expression, and two statements in the third expression, both separated by commas. We also omit the statement that usually follows the `for` loop, since it isn't necessary in this example. (Notice the semicolon at the end of the `for` statement signifying its end.) This `for` statement assigns the variables `low` and `high` to `a` and `b` and then increments a and decrements b until the two values meet or pass each other.

Whether it is better to use the `for` or `while` statement for a particular situation largely depends on which better suits your needs. For example, if you don't need the first expression in the `for` statement, it would probably be better to use the `while` statement instead. But when you need all three expressions, the `for` will most likely be the best choice, since it places all elements of the loop in a common location making it easier to read.

The Switch Statement

Next we come to the switch statement, a multi-way decision that compares an expression with any number of integer constants. (A constant is a value that doesn't change, such as a number; a variable is not a constant, since its value can change. An integer constant is a constant value that must follow the same rules applied to integer variables; that is, the constant must be a whole number between -32768 and 32767.) The format of the `switch` statement is

```
switch (expression)
{
     case constant:
          statements
     case constant:
          statements
     default:
          statements
}
```

The `expression` is the same as what we've already discussed for the `if` statement; it can be a variable or a relational expression such as "`a > b`". The result of the expression (`true` or `false`) will be compared with each case. There can be as many or as few cases as you want, and the default case is optional. Execution starts with the case that matches the expression, or with the default case (if there is one) if no matching cases can be found. Execution continues until the `break` statement or the end of the `switch` statement is encountered.

Below is an example of the `switch` statement that compares a variable with eight constants. In this case, the constants are "character constants", such as 'A', which, as I've mentioned in the past, are translated by the compiler into the corresponding ASCII value when the program is compiled ('A' is translated into 65). Character constants are still valid integer constants, so they can be used in the `switch` statement.

```
switch (myKey)
{
     case 'E': case 'e':
          MoveUp();
          break;
     case 'S': case 's':
          MoveLeft();
          break;
     case 'D': case 'd':
          MoveRight();
          break;
     case 'X': case 'x':
          MoveDown();
          break;
}
```

We opted not to use a `default` case for this example, since it isn't needed. We use two cases for each section of code, which is perfectly legitimate; you don't need to have code after each case. (Remember that execution starts at the code following the matching case and continues until the break statement is encountered.) Cases are simply "labels", which means that when a matching case is found, the code beneath the case is executed without delay, regardless of how many cases, or labels, are between the matching case and the actual code.

Each of the cases, including the `default` case, can be in any order; you could put the `default` case at the top if you wanted to, or in the middle. However, it's generally common practice to put it at the end. (Just like the `else` part of the `if-else` statement is always at the end.)

It is important to remember to put in `break` statements after each section of code when you want it to stop. If you don't, the execution will "fall through" to the code intended for the next case. Consider the example above; if no break statements were used, and myKey were equal to 'E', the execution would start at that case and continue through all the code, calling each of the four functions, since a `break` statement would never stop it. This means you don't have to use braces to group together more than one

statement, but do have to remember to put the `break` statement at the end of each group.

Every now and then you may run across a situation where you find it desirable to write code that "falls through" to the code intended for the next case as well, but usually you should avoid this practice, since you can easily introduce bugs into your code when adding new cases, forgetting that your other cases will fall through into it. If you ever do write code that falls through, make sure to comment it well, so that if you need to modify it at a later date, you aren't as likely to introduce problems. The example above where I use more than one label for each statement isn't considered "falling through", since there are no statements between the extra cases.

As you may have guessed, most anything you could write with a `switch` statement could also be written using the `if-else` statement. Each have their advantages and limitations. With `if` statements, you can compare an expression or variable with anything; you aren't limited to constant integers, and `if` statements are often more compact. However, if you find that you need to compare a single variable or expression with a number of constants, you may find that the switch statement is more desirable. In case you're curious, this is what the example above would look like if written using the `if-else` statement:

```
if (myKey == 'E' || myKey == 'e')
     MoveUp();
else if (myKey == 'S' || myKey == 's')
     MoveLeft();
else if (myKey == 'D' || myKey == 'd')
     MoveRight();
else if (myKey == 'X' || myKey == 'x')
     MoveDown();
```

This may appear to be more compact than the `switch` statement, but I'll bet it's less efficient. My C manual doesn't say, but my guess is that since the expression in a `switch` statement is compared with constant values that are known at compile time, the compiler can take advantage of the situation and write code that is "smarter", and finds the matching case faster, without comparing each case one-by-one. If this is true, then the `switch` statement would be somewhat similar to XB's ON GOTO statement, where the code

```
ON X GOTO 100,200,300
```

would be more efficient than

```
IF X=1 THEN GOTO 100 ELSE IF X=2 THEN GOTO 200 ELSE IF
X=3 THEN GOTO 300
```

The Break and Continue Statements

The `break` and `continue` statements don't provide new ways of doing loops or making decisions, but rather, these statements are to be used with the flow control statements we've already covered, such as the `while` or `for` loops.

The `break` statement lets you immediately exit a `for`, `while`, or `do` loop, and can also be used to exit a `switch` statement, as you've already seen. This allows you to exit a loop at any time, without having to reach the test at the top or bottom of the loop. Here's an example:

```
char n, myArray[100], foundNum = 0;

for (n=0; n < 100; n++)
{
     if (myArray[n] == 150)
     {
          foundNum = 1;
          break;
     }
}
```

This example cycles through all elements of an array searching for the number 150. If the number is found, the variable `foundNum` is set to true, and the `break` statement is used to exit the `for` loop, since we don't need to cycle through the rest of the array now that we know the result of our search. Here's the same thing in XB:

```
100 DIM MYARRAY(100) :: FOUNDNUM=0
110 FOR N=0 TO 99 :: IF MYARRAY(N)=150 THEN FOUNDNUM=1 ::
GOTO 130
120 NEXT N
130 ! PUT REST OF PROGRAM HERE
```

I should mention that the `break` statement only exits the innermost enclosing loop; if you have two nested loops and use the `break` statement inside the innermost loop, that will cause an exit from that loop only, but the outer loop will continue, such as in the example below, which cycles through a two-dimensional array, again searching for the value 150. This time, when the number is found, the number 1 (true) is stored in `myArray[a][0]`; otherwise, this element is set to 0 (false). Keep in mind that this is just a code "snippet"; I haven't included the part that actually fills the array with values in the first place, since that part isn't necessary for you to understand the point I'm making.

```
char a, b, myArray[10][100];

for (a=0; a < 10; a++)
{
     myArray[a][0] = 0;
     for (b=1; b < 100; b++)
     {
          if (myArray[a][b] == 150)
          {
               myArray[a][0] = 1;
               break;
          }
     }
}
```

This may be a little confusing, so let me also give you an XB version:

```
90 DIM MYARRAY(9,99)
100 FOR A=0 TO 9
110 MYARRAY(A,0)=0
120 FOR B=0 TO 99
130 IF MYARRAY(A,B)=150 THEN MYARRAY(A,0)=1 :: GOTO 150
140 NEXT B
```

```
150 NEXT A
```
Of course, there are times when you'll want to be able to exit all enclosing loops; not just the innermost loop. This can be accomplished with c99's goto statement, which is covered later on in this article. I should also mention that even though the `break` statement in the example above is "enclosed" in an `if` statement, it will still exit properly from the enclosing `for` loop. This is because the `break` statement only affects the `for`, `while`, `do`, and `switch` statements, but has no effect on `if` statements; it just acts as if the `if` statement isn't even there, and exits the `for` loop as desired.

The `continue` statement is similar to the `break` statement, but instead of causing an exit from the innermost enclosing loop, it causes the next iteration of that loop. Therefore the `continue` statement can be used in the `for`, `while`, and `do` loops, but not in the `switch` statement. (Although if you use it in a `switch` statement that is enclosed in a loop, it will cause the next iteration of the loop.) Here's an example of the `continue` statement where it is used to skip the 50th element of an array, but process all other elements:

```
for (a=0; a < 100; a++)
{
     if (a==50)
          continue;

     for (b=1; b < 100; b++)
     {
          if (myArray[a][b] > 99)
               myArray[a][b] = 99;
     }

     myArray[a][0] = a;
}
```

Of course, this could've been written like this instead, using an `if` instead of a `continue`:

```
for (a=0; a < 100; a++)
{
     if (a != 50)
     {
          for (b=1; b < 100; b++)
          {
               if (myArray[a][b] > 99)
                    myArray[a][b] = 99;
          }

          myArray[a][0] = a;
     }
}
```

However, we then have to indent everything a level and add another set of braces, which could be undesirable in some situations, especially if the code being indented is quite long and complex. Therefore, the `continue` statement can be used at times during loops to keep them cleaner than they would otherwise be, and is often simply more convenient than adding another

`if`. It can also be used anywhere in the loop; you don't have to put it at the beginning of the loop like I do in the example above. This means the `continue` statement can be used to start the next iteration of the loop in places where it would be difficult to do the same thing using other methods.

The Goto Statement

The goto statement is used to branch to a label that is in the same function as the goto statement. You should avoid using the goto statement, since it often makes the code harder to read. (Basic programmers are all too familiar with this fact.) However, there are times when the statement can help. One example would be if you want to jump out of a nested loop. Here's an example:

```
for (a=1; a<100; a++)
     for (b=1; b<100; b++)
          if (myArray[a][b] == 150)
               goto exit;
exit:
     DoOtherStuff();
```

This example may have been a little clearer if I had used braces, but I decided not to, since there was only one statement for each "group", so this is a perfectly legitimate way to write the code. In any case, this code cycles through all elements of myArray, searching for the value 150. If it is found, the statement "goto exit;" transfers control to the label "exit:". Labels are familiar to those of you who are assembly programmers. For you XB programmers, a label is similar to a line number: going to a label starts executing the statements immediately following the label.

Label names follow the same rules applied to variable names: they must start with a letter, only the first six characters are used, and you can't rely on capitalization to distinguish between two labels of the same name. Keep in mind that your label names shouldn't conflict with any other names, such as function or variable names. Put a colon after the label name, so the C compiler knows that it is a label.

Any code that is written using the goto statement can be written without it, although sometimes at the expense of extra comparisons. For instance, the example above could have been written as

```
exitLoop = 0;
for (a=1; a<100 && !exitLoop; a++)
     for (b=1; b<100 && !exitLoop; b++)
          if (myArray[a][b] == 150)
               exitLoop = 1;
DoOtherStuff();
```

Obviously, the goto method would be more efficient in this case, since it is then not necessary to check the `exitLoop` variable each pass through the loop. However, there are few times when the goto statement provides a more efficient way of doing things than other methods. As a rule of thumb, avoid the goto whenever possible, using it only in situations where it is more convenient or helps the code run faster, and doesn't make the code much harder to read.

More on the If Statement

Last installment, when discussing how in c99 you can assign the result of a relational expression to a variable (such as "result = a<b;"), I expressed doubt about whether one could do the same thing in Extended Basic. Bruce Harrison, our local Assembly Language guru (who also knows a thing or two about Extended Basic) was quick to write in and point out that Extended Basic can do the same thing.

There are only two differences. The first is that you must enclose the relational expression in parentheses. The second is that in XB, the "true" value is -1, rather than 1. Here are some examples Bruce sent that you can try in XB's Command mode:

A=1 :: N=(A=1) :: PRINT N <ENTER> (prints -1)

Now hit FCTN-8 and modify this command to:

A=1 :: N=-(A=1) :: PRINT N <ENTER> (prints 1)

Hit FCTN-8 again and change it to this:

A=5 :: N=(A=1) :: PRINT N <ENTER> (prints 0)

You can also use the operators AND and OR within the expression, but more parentheses are needed. [e.g. `N=-((A=1) AND (B=0))`] This also works using the other relational operators, such as >, <, >=, <=, etc.

Contact Information

If you have any questions or comments, feel free to email me at Jensen@lafn.org, or write to me at Vern L. Jensen, 910 Linda Vista Ave., Pasadena, CA 91103. The email address I gave out in the first two installments will also still work; I'm just giving out the Jensen@lafn.org address instead because I "lost" a few messages at the other address. The U.S. Postal Service isn't the only place that loses mail!

Beginning c99 - Part 5

Time for a little fun!

By Vern Jensen

After going through some pretty boring (but important) stuff the last two issues, today we're going to do something fun. We're going to make a "screen saver" that simulates flying through space. In the process, you'll learn a number of things, such as how to add comments to your source code, how to make definitions, and how to use functions and arrays.. Who says work can't be fun?

To get started, start up the Editor and type in the program in today's sidebar. Save it as DSK2.SPACE;C, then compile, assemble, and run it. After seeing what the program does, continue reading below.

Comments!

Probably the first thing you'll notice about today's example is that it has lots of comments. In C, you can specify the beginning of a comment with the characters "/*" and the end of the comment with the characters "*/" (without quotes, of course). And although all the comments in today's example are only one line long, you are not limited to a single line. (Remember that the compiler ignores spaces.) The really great thing about this is that you can easily comment out half your program if you want, simply by adding the "/*" and "*/" characters to the beginning and end of the code you want to comment out.

For instance, as I was working on this program, I didn't like the way the star's initial position was calculated in NewStar(), so I commented out all the if statements and wrote new code. Later, I realized my new method wasn't that great, and it was easy to uncomment the old code and start using it again. If I had previously deleted it rather than commenting it out, I would have had to type it in from scratch.

There is just one problem with commenting out whole sections of code like this, and that is that comments can not be nested. For instance, if I were to try to comment out the lines below as demonstrated, it wouldn't work properly, because the compiler would see the "end of comment" character at the end of the "/* Move left */" comment, and assume the first comment was done:

```
/*
if (myKey == 'S')  /* Move left */
      hDelta = -5;
*/
```

Because the compiler ignores everything from the first "start comment" characters it sees to the first "end comment" characters it sees, the code above would be turned into this:

```
      hDelta = -5;
*/
```

Obviously this is not what we wanted, so you have to be careful when commenting out entire sections of code to make sure that the code doesn't already have other comments within it. The only reason I was able to comment out all the if statements in the NewStar() function while working on the program was because I hadn't added the other comments to the code yet.

Definitions

At the beginning of the program, following the #include statements, you'll notice a series of #define statements, which allow you to assign a number to a name. When the program is compiled, the names will be replaced with their corresponding numbers. This means that using #defined names in your program is just as fast as using the actual numbers themselves. For instance, the statement

```
if (a == 69)
```

is just as fast as

```
#define kUpKey 69
...
if (a == kUpKey)
```

since the second example is changed into the first during compilation. Definitions provide a way of keeping your code much neater, since you can put all your definitions at the beginning of a program, allowing you to easily change those values in the future if necessary, without having to hunt down all the places they are used in the program. For instance, in today's program, the definition kNumStars is used twice in the program. By using this definition instead of the actual number in the program, I can easily change the number of stars by changing the definition, rather than having to hunt down all the places the number 15 is used in the program.

In case you're wondering why the definitions start with the letter k, it's just to keep them from conflicting with variable and function names. (A habit I picked up when programming on the Mac.) And, just like variable and function names, the c99 compiler only recognizes the first 6 characters of a definition name. However, since definitions are replaced with their corresponding value *before* assembling, definitions *are* case sensitive - so if you #define kFColor 5 at the beginning of the program, and then use the name kfcolor later in the program, the compiler won't recognize it, and won't replace it with the number 5. This will result in an "Undefined Symbol" error when you try to assemble the program, because the assembler won't recognize the name kfcolor either, which should have been replaced with 5 by the compiler. To demonstrate this is the following program, which is a modified version of the test program we used in the first article in this series:

```
/************/
/* MYTEST;C */
/************/
#include “DSK2.GRF1;H”

#define CHR 65
#define myChr     66

main()
{
  int chr, keyCode, status;

  Grf1();
  Screen(14);

  chr=33;

  do
  {
    HChar(1,1,chr,768);
    HChar(12,1,CHR,32);
    HChar(13,1,mychr,32);
    chr++;

    if (chr >= 126)
      chr=33;

    keyCode = Key(0,&status);

  } while (status == 0);
}
```

You will notice two definitions at the beginning of the program. The first defines CHR as 65. However, this does not conflict with our variable of the same name, because the variable is in lower case. This means that if you want, you could put your definitions in upper-case in order to keep them from conflicting with variable names, rather of starting the definition name with k.

The second definition defines myChr as 66. Later in the program, mychr (lowercase) is used. If you try to compile and assemble the program the way it is written, you'll get an "Undefined Symbol" assembly error. I'd recommend doing this just for the experience, in case

this actually happens to you when you're writing your own program. When you get an assembly error, load the file being assembled (in this case, MYTEST;S) in the Editor and take a look at the line number containing the error. If the file is too big to load in the Editor, you can still look at the line using the LINEHUNTER program, option 6 in Funnelweb's menu. Once you find the line, you'll see that the name MYCHR is causing the problem, and you can then search MYTEST;C for that name until you figure out which line contains the error.

It's things like this that gave me extra incentive to write this series. The c99 manual makes absolutely no mention that definitions are case sensitive, while variable and function names are not, which caused me a lot of frustration while working on Virus Attack. By sharing these and other tips with you, hopefully you can avoid the problems I encountered, and be able to spend your time writing code rather than hunting down bugs.

Arrays

In c99, you can have one or two dimensional arrays. Arrays can be of type char or int, and the first element of an array is 0, just like in XB (unless you call OPTION BASE 1 first). Defining an array is simple - just define a variable name as usual, placing the number of desired elements between brackets at the end of the variable name:

```
int myArray[10];
```

It is important to realize that this statement only creates 10 elements, meaning that since the first element is 0, the last element is 9. Thus, trying to access element 10 of this array will cause problems. To create a two-dimensional array, add another set of brackets, like so:

```
int myArray[10][10];
```

Note that this is not the "add a comma" method used by XB, so this statement would be wrong:

```
int myArray[10,10];
```

Accessing an array is straightforward; the following example shows how to set all elements of the array defined above to 0:

```
for (a=0; a<10; a++)
      for (b=0; b<10; b++)
            myArray[a][b] = 0;
```

In the SPACE;C example, our star array keeps track of 3 variables for each star - the x, y, and z coordinates. If you look after the array definition, you'll notice these statements:

```
#define kX 0
#define kY 1
#define kZ 2
```

These statements allow us to easily access each of the three variables for each star without having to remember which array element corresponds to the desired variable. For instance, to access the y variable of star 5, we can use the statement

```
      y = star[5][kY];
```

rather than having to remember this:

```
      y = star[5][1];
```

The first statement is much clearer, since at a glance you can tell that the y variable of star 5 is being read, while the second statement is unclear. This also makes debugging much easier, since at a glance you can tell if you accidentally accessed the wrong element of an array.

Functions

In today's program, you'll notice that very little actually happens in the main() function. Instead, it calls other functions that do most of the work. Functions are very similar to XB's subroutines. In XB, all you have to do to call a subroutine is precede the subroutine name with the word CALL. In c99, it's just as easy - all you have to do is name the function and put parentheses after the name. (Even if it has no parameters.) In fact, the main() function does nothing but call other functions - first functions to prepare for the animation, and then the MainLoop() function to run it.

Creating your own function is just about as easy as calling one. In fact, whether you know it or not, you've created a function each time you've typed in a c99 program, because main() is a function. The only difference between main() and the other functions in a program is that main() is automatically called when your program is first started.

Take just a minute to look at the SetUp() and MainLoop() functions. They're about the same as the main() function, as they have no parameters. I could have put the code for these functions inside the main() function, but chose to split it up into separate functions to make the code even easier to read. As you can see, all that needs to be done to create a function is to name it, follow it with parentheses containing the names of your parameters (if any), and then braces containing the code that makes up the function. Thus, to make an "empty" function that does nothing, you would write this:

```
Empty()
{
}
```

Adding parameters is easy as well. For an example of a function that has parameters, take a look at the GetRnd() function at the end of the program. Here you see that to add parameters, all one must do is place the names of the parameters between the parentheses, separated by commas. You also must tell the compiler what type of variables the parameters are (char or int) with a separate statement (or statements) following the function name. It is possible to have parameters of different types; for instance, one int and one char. This would be done like so:

```
MyFunc(key,number)
int key;
char number;
{
}
```

One important point to remember is that functions, unlike XB's subroutines, are given copies of the values passed to them as parameters. This means that if you pass variables as parameters to a function and the function changes its copy of those variables, the originals will remain unchanged. The same thing is possible in XB by placing parentheses around your variable names when passing them to a subroutine (see page 181 in your XB manual).

This may bring you to wonder how functions can return a value to the caller. There are two solutions: using pointers, and the return statement. We will cover pointers next issue, but the return statement is simple enough we can cover it now.

Normally, all the statements in a function are executed, and then control returns to the line after the calling statement. However, if the return statement is encountered, the function is exited early, just like SUBEXIT causes an early exit from an XB subroutine. However, c99's return statement has an advantage - it can optionally return a value to the caller. For instance, the following function squares a number, and returns the result:

```
Square(myNum)
int myNum;
{
      return myNum * myNum;
}
```

We would call this function like this:

```
n = Square(n);
```

This would call the Square() function, passing the variable n as the parameter. The function would receive a copy of this variable, multiply it by itself, and return the result, which would then be assigned to the variable n. When calling a function that returns a value, you can optionally discard the value if you don't need it. For instance, we could call the function above with the statement "Square(n);", although that would be pretty pointless, since the whole purpose of the function is to return a number.

As I mentioned before, a function can not change a variable that is passed to it by modifying its own copy of the variable. To illustrate this, let us consider a different version of the Square function:

```
Square(myNum)
int myNum;
{
      myNum = myNum * myNum;
}
```

When this function is called, it receives a copy of the value passed to it and squares it, assigning the result to myNum. However, this is completely pointless, since changing myNum doesn't affect the original variable that was passed to the function.

In addition to returning the value of a variable, a function can also return a fixed number. For instance, this function returns 1 (true) if the requested character can be found on the screen, or 0 (false) if it can not:

```
FindChar(myChar)
char myChar;
{
      int r, c;

      for (r=1; r <= 24; r++)
            for (c=1; c <= 32; c++)
                  if (GChar(r,c) == myChar)
                        return 1;   /* Found a match */

      return 0;   /* Loop ended - no match found */
}
```

This example may be slightly confusing because of the call to GChar(), which is a function in the GRF1 library that returns the value of the character at the specified row and column of the screen. In this case, rather than assigning the returned value to a variable, we place the call to the GChar() function itself in the if statement, which compares the value returned by the GChar() function with myChar. This is a technique that may look quite foreign to XB programmers, but is used quite often by C programmers. Again, the fact that you can actually put a function call inside an if statement is yet another example of how flexible C is.

Something you should keep in mind when writing code that calls functions is that a single C statement for calling a function has to be translated into many assembly language statements when the program is compiled. The more parameters a function has, the more assembly language statements there will be. Because of this, writing code with many function calls can add considerably to the size of a program. In addition, you should avoid calling functions as much as possible when in a time-critical loop where speed is important.

That's why the MainLoop() function in today's example does all that is needed for the animation in one function - I didn't break that code up into several smaller functions. (Except for creating a new star, which only happens once a star reaches the edge of the screen.) However, I did take the liberty to make a separate SetUp() function, rather than placing that code in the main() function, since it makes the code clearer, and it isn't in a time-critical loop. Also, SetUp() doesn't have any parameters and is only called once, so doing this doesn't add much to the length of the program.

Global and Local Variables

The C language has so much in common with Extended Basic that sometimes I wonder if parts of Extended Basic were modeled after C. The next lesson is one of those instances. As you may know, variables that are used in a subroutine in XB are "local" to that subroutine, meaning they can't be "seen" by code outside that subroutine. In fact, you can have variables in your subroutine that have the same name as variables used in the main part of your program, and they won't conflict.

It's just the same in C. Variables that are declared within a function are called "local" variables, since they are local to that function only, and can't be used by any other function. Variables that are declared outside of a function are called "global" variables, because any function can use them.

In today's program, we have many local variables and only two global variables. The global variables are created before the main function with this statement:

```
int   n, star[kNumStars][3];
```

This means that any function can access the n variable or the star array. However, variables that are declared within a function can only be used by that function. For instance, the variables x, y, and z which are declared in the MainLoop() function can only be used by that function. Local variables must be declared at the very beginning of a function (before any statements), and global variables should be declared at the beginning of the program (before any functions).

In standard C, the position of your global variables can affect their scope, so placing the declaration of the global variables n and star[][] after the MainLoop() function but before the NewStar() function would make those variables accessible only to the NewStar() function

and those after it; they would be "hidden" from all the functions in front of the definition. I doubt c99 supports this, and there generally isn't a need for it anyway, so you should just stick to declaring your global variables at the beginning of the program unless you have tested the other method and know that it works.

Both global and local variables have a random value at the start of the program or function. However, you can assign an initial value to global variables. For instance, you could put this statement at the beginning of your program:

```
int   numLives=3, level=1;
```

This would start the player off with 3 lives on level 1. However, you must remember to reset these values if the player starts the game over again. You can not assign a value to a local variable when declaring it as you can to a global variable. The reason for this is that global variables are permanent, so they can be given a starting value, but local variables are temporary.

Local variables are called "auto" variables because they are created when a function is entered and disposed when the function is done. This means that you can not assign a value to a local variable and expect it to retain that value when the function exits and is then called again. For this reason, you should use global variables when you need a variable to retain its value even after a function exits, or if you need to share a variable between several functions. However, it's generally best to use local variables for "temporary" functions, such as cycling through a loop. Consider the following program:

```
int n;
main()
{
      Grf1();
      for (n=2; n<=16; n++)
            TextColor(n);
}

TextColor(color)
char color;
{
      for (n=4; n<16; n++)
            Color(n,color,1);
}
```

Here, the code in main() attempts to cycle through all colors, calling a separate function to change the text ascii characters to that color. However, since the same global variable is used in both loops, the first loop won't function properly, since the value of n will change when TextColor() is called. This is one good reason why loops and other things which can use local variables should use them. And although the problem above is pretty obvious, in a real program, it often isn't, so using local variables when possible can help you avoid a nightmare of problems.

I mentioned above that local variables in one function can't be seen by other functions. While this is true, there is either a bug, or undocumented limitation in the c99 compiler that appears if one function calls another that uses a variable of the same name. In normal C, this shouldn't be a problem, but in c99 it is. In this case, the variable of the same name seems to be "shared" by the two functions, so changing its value in one will change its value in the other. Consider again our program above, this time modified to use local variables:

```
main()
{
      int n;
      Grf1();
      for (n=2; n<=16; n++)
            TextColor(n);
}

TextColor(color)
char color;
{
      int n;
      for (n=4; n<16; n++)
```

```
            Color(n,color,1);
}
```

While this would work in normal C just fine, it doesn't in c99. The solution is to use a different name in the second function, such as c instead of n, so the two variables don't conflict. In addition to keeping local variable names from conflicting with each other, you also need to keep your global variable names from conflicting with local variable names.

One technique commonly used nowadays it to place the letter g at the beginning of global variable names. However, on the TI, this uses up one of your six letters, making it a less desirable technique. A technique I prefer is to make sure my local variable names don't conflict with the global names, which I often do by putting the letters "my" at the beginning of the local variable names, such as "myCol" or "myRow", which would keep them from conflicting with global variables with the names "col" and "row".

About the Actual Program Itself

I've spent most of this article talking about basics of c99, as I should. However, a few of you may be wondering how the program works. Most of it is pretty self-explanatory. The stars are given a destination somewhere outside the boundaries of the monitor, and a depth; the greater this depth, the closer the stars appear to the center of the screen. Each frame, the depth is decremented, bringing each star closer to its destination. When the depth is equal to 1, the star has reached its destination outside of the screen.

What makes the code in MainLoop() a little harder to understand is the fact that I needed to be able to decrease the depth of each star (the kZ element of the star array) by .1 per frame, which is impossible in c99 because the variables can only contain whole numbers. The solution to overcoming this was to multiply the x, y, and z variables by 10 when the star is first created, and then to divide them by 10 when actually calculating the star's position on the screen. I could then decrement the depth variable by 1 each frame, which would effectively lower the actual depth by .1.

As many of you know, the screen has 256 visible pixels horizontally and 192 visible pixels vertically. This means that 128 is the middle pixel horizontally, and 96 is the middle pixel vertically. Therefore, when you see numbers like 960 or 1280 being used in the program, keep in mind this is just 96 or 128 being multiplied by 10.

One other interesting thing about this program is the following statement in the MainLoop() function:

```
star[n][kZ] = z = z - kSpeed;
```

You're probably wondering why it has two equal signs. Well, the fact of the matter is that c99 is very flexible, so you can put assignments anywhere you'd like, in as many places as you'd like. The key to remember is that statements such as these (unlike if statements) are processed from right to left, meaning that first kSpeed is subtracted from z, then this result is assigned to z, and then z is assigned to star[n][kZ]. Generally code like this is harder to read than it would be if broken up into separate statements, although it does have its place. The most common example would be when setting a number of variables to the same value:

```
a = b = c = d = 5;
```

One final thing I might mention about this program is that this time I called it SPACE/C rather than SPACE;C. To tell you the truth, using a slash is the standard naming convention for c99 filenames; a semicolon is the standard for assembly language programs. However, I found it easier to hit the semicolon key when typing a filename quickly, which is why I've used it up until now. Naturally, you can use whatever you like best, although you should be consistent.

In Conclusion

Once again, we have a pretty long column. Hopefully it's been helpful to you on your journey from Basic to c99, and should help keep you busy until the next issue. You now know enough to start making your own programs, which I suggest doing, as experimenting with the language is the best way to learn. Have fun!

Beginning c99 - Part 6

Pointers - What are they?

By Vern Jensen

Last issue we mentioned that there are two ways of returning values from a function: the first is by using the return statement, and the second is by using pointers. But what is a pointer? It is simply a variable that contains the address of another variable's location in memory; thus the name "pointer", since it "points to" another variable. Pointers must be declared differently than other variables. To declare a variable as a pointer, you must put an asterisk in front of the variable when declaring it:

```
char *myPtr;
```

The asterisk makes only the variable it is in front of a pointer, so in this example, the variable intPtr becomes a pointer, while the other variables do not:

```
int key, *intPtr, status;
```

When declaring a pointer, you determine what type of variable the pointer can point to, whether char or int. In the examples above, myPtr can point to other char variables, and intPtr can point to int variables. To assign the address of a variable to a pointer, you use the & operator, which gives the address of a variable:

```
intPtr = &status;
```

This will assign the address of the status variable to intPtr. Keep in mind that this is quite different than assigning the contents of the status variable to intPtr. If we wanted to do that, we would simply use "intPtr = status". Once we have the address of a variable stored in a pointer, we can access that variable through the pointer by using the dereferencing operator "*":

```
myNum = *intPtr;  /* "myNum = status" */
```

This will assign to myNum whatever intPtr pointers to. In this case, it points to the variable status, so the contents of status will be assigned to myNum. Things can go in the other direction as well. For instance, this assigns the contents of myNum to whatever variable intPtr points to:

```
*intPtr = myNum;  /* "status = myNum" */
```

Since intPtr points to status, *intPtr can appear anywhere status could, so statements such as these are possible:

```
*intPtr = *intPtr * 5; /* Multiply status by 5 */
(*intPtr)++;           /* Increment status */
```

The parentheses are necessary in the last example because without them, intPtr would be incremented instead of what it points to, because operators like ++ and * associate right to left. And as you might expect, you can assign the contents of one pointer to another, so if otherP is also an int pointer, this would assign the contents of intPtr to otherP, making otherP point to status as well:

```
otherP = intPtr;
```

Functions and Pointers

"So what's all the fuss about?", you may wonder. "Sure, pointers are cool, but how do they help me?" Well for starters, you can pass the address of a variable to a function. This allows the function to modify the original copy of that variable. Below is an example of a function that accepts pointers to two variables and swaps the contents of the original variables. Notice that the function parameters are declared as pointers:

```
void  Swap(a, b)
int *a, *b;
{
      int temp;

      temp = *a; /* Save contents of a */
      *a = *b;   /* assign contents of b to a */
      *b = temp; / *assign old a value to b */
}
```

To call this function, you would use

```
int myA, myB;
...
Swap(&myA, &myB);
```

and the contents of myA and myB would be swapped. Another way to call this would be like so:

```
int   *aPtr, *bPtr;

aPtr = &myA;
bPtr = &myB;
Swap(aPtr, bPtr);
```

In both cases, the addresses of myA and myB are passed to the function. The function can then dereference the pointers to access the variables they are pointing to, allowing the function to change values it otherwise would be unable to change. But how about a "real life" example? Here's something you'll likely use every time you write a program:

```
int char, status;

char = Key(0, &status);
```

Here the address of the status variable is passed to the Key function (part of the GRF1 library) so the function can place the current status in that variable. In addition, this function also makes use of the ability to return a value with the return statement, and returns the key's character code that way. Another example of pointers at work is the Joyst function, which returns x, y, and status values:

```
int s, x, y;

s = Joyst(0, &x, &y);
```

Pointers and Arrays

Pointers and arrays are closely related. In fact, any operation that can be performed by array subscripting can also be done with pointers. We have seen that you can assign the address of a variable to a pointer. You can also assign the address of an array, or any element of the array, to a pointer. Here's an example:

```
char a, *myPtr, array[50];

myPtr = &array[0]; /* myPtr points to array element 0 */
a = *myPtr;        /* a = array[0] */
*myPtr = 7;        /* array[0] = 7 */
```

First we assign the address of the first element of the array to the pointer, then we dereference the pointer by using the "*" (dereferencing) operator, so that variable a is given the contents of whatever myPtr points to. Finally, we use the dereferencing operator again to assign 7 to whatever myPtr points to. If we didn't use the dereferencing operator, myPtr itself would have 7 assigned to it, instead of array[0]. If you then tried to use myPtr as a pointer, you would run into problems, since you would be accessing whatever is kept at memory location 7, which certainly isn't the address of one of your variables.

There is another way to get the address of an array, and that is by simply using the array's name without the "&" operator and without subscripting. For instance, this would set myPtr to point to array element 0, just as the code above does:

```
myPtr = array;
```

This is because the name of an array contains the address of the first element of that array. When you add a subscript, such as "array[5]", the C compiler uses the address contained in the array name to access the desired element of the array. Subscripting is not limited to arrays, however. You can also subscript pointers:

```
a = myPtr[5];
```

This tells the compiler to access the fifth object after what myPtr points to, which in this case is array[5]. So by assigning the address of array to myPtr, we can use myPtr just as if it is the array. This can be very useful when calling functions, since you can pass the address of your array to the function, and then that function can access any element of the array. Keep in mind that while you can get the address of an array using the array's name, you can't change it. So this wouldn't work:

```
array = myPtr;
```

However, you can change the value in myPtr. When you add or subtract a value from a pointer, it changes what the pointer points to. For instance, adding 1 to a pointer makes it point to the very next object in memory. If the pointer is a Char, the compiler is smart enough to move it up one byte in memory (since a Char takes up one byte). If the pointer is an Int, then adding one will move it up two bytes in memory (an Int takes up two bytes). This allows you to increment a pointer that points to an array, and it will point to the next element, regardless of whether the array is of type Char or Int.

Here's an example that sets a to the value contained in array[5]. First we add 5 to myPtr, then we dereference the new address:

```
a = *(myPtr+5);  /* Same as a = array[5] */
```

Or if you want to change the pointer itself, you could do this:

```
myPtr = myPtr + 5;  /* myPtr now points to element 5 */
```

Now accessing myPtr as an array will mean that myPtr[0] is the same as array[5], myPtr[1] is the same as array[6], etc. This trick could be useful when passing the address of an array to a function if you wanted the function to think a particular element of the array was actually the beginning of the array. (You could pass something like "&array[5]" as the parameter to your function.)

Enough!

Enough confusing nonsense for right now. Here's an example function that will help clear things up. It scans an array for the requested ascii value, and returns the element of the array in which the value was found. For the sake of simplicity, it is assumed that the value is contained in the array, so the array's boundaries will not be exceeded:

```
ScanArray(myChar, myArray)
char myChar, *myArray;
{
  int n;

  for (n=0; myArray[n] != myChar; n++)
    ;

  return n;
}
```

Here we have an example of an "empty" loop. The for loop executes no statements, because all the necessary statements are provided in the loop itself. (Just try to do that in Extended Basic!) The function receives the address of the array in its pointer (myArray), and uses the pointer as if it were an array. It starts with element 0 and increments the current element until myChar is found, at which point the loop ends and the current element is returned to the caller. The function above could also be written without any array subscripting, although it might be a little more confusing:

```
ScanArray(myChar, myArray)
char myChar, *myArray;
{
  int n;

  for (n=0; *myArray != myChar; n++, myArray++)
    ;

  return n;
}
```

The key to remember is that the statement "*myArray" gives the value that myArray points to, and the statement "myArray++" increments the myArray pointer so that it points to the next element. Normally you wouldn't write your code like this, since it is a little harder to understand, but this type of code does have its place. To call the ScanArray function, you might use something like this:

```
element = ScanArray('E', "HELLO");
```

The expression 'E' should be familiar - it is converted by the compiler into the corresponding ascii value (69). However, the "HELLO" text in quotes is new. This is called a string constant, meaning it's a string of char type data that can't be changed. When you use a string constant as a parameter to a function, the address of the string is passed to the function. The function will then see an array filled with these values:

```
[72][69][76][76][79][0]
```

As you can see, a string constant is represented in memory with ascii values for each character in the string, terminated by a 0 to indicate the end of the string. Each character in a string constant is only one byte (the size of a char), meaning that if you assign the address of a string constant to a pointer, the pointer must be a char pointer. You may be wondering by now if C has the equivalent of XB's strings. After all, while string constants are nice, you can't change them. The answer is that you use arrays to store the contents of your strings, and if your array is a char array, you can use a string constant to initialize it, creating an array which contains your string, but can be modified:

```
char myString[] = "Pretty neat, huh?";
```

Notice that I didn't put a number between the brackets of the array. That's because if you leave it out, the size of the array is calculated based on the size of the string you provide - it will have the same number of elements as the number of characters in the string plus one, to account

for the "end of string" character, ascii value 0. If you do specify the number of elements in the array when initializing it with a string constant, it must be larger than the number of characters in your string. Extra elements not in your string will be filled with 0's. It is also possible to initialize an array with numbers:

```
int prime[] = {1,3,5,7,11,13,17};
```

Keep in mind that since a string constant is not used in this second example, the array will not be terminated with a zero - that's something you'll have to add if you want it. Array initializers like the ones described above will not work if the array is a local variable, since local variables come and go as the function is entered and exited. For the examples above to work, they'd have to be global arrays (meaning they are defined at the beginning of your program, not in a function). And unfortunately, you can't assign a string constant to an array that has already been created. For example, the following code will not work:

```
char myArray[5];
myArray[0] = "Hello";   /* Won't work. */
```

You can, however, assign the address of a string constant to a char pointer:

```
char *myPtr;
myPtr = "Hello";
```

In this example, myPtr will point to the address of the first character in the string, allowing you to access that string like a char array. However, you must remember that the string is still a constant - you can read those characters, but you can't change them. On the other hand, you can change the pointer so that it points to something else, although you would then lose the address of the "Hello" data.

A Silly Program

To conclude our article is a program that demonstrates the topics we've just covered. The program is pretty pointless - it displays text on the screen both horizontally and vertically, but it does cover a number of important points. Most importantly, it gives you an example of how to use what we've covered in a real program. I say this is important because I ran into quite a few compile errors when I tried to make it. If I had this much trouble making it, I can't imagine how much trouble a beginner would have trying to use what we've just covered if an example program wasn't provided!

This issue we're going to do something a little different - I'm going to walk through the source code, explaining each line.

```
char helloS[] = {72,69,76,76,79,0};
char introS[] = "This is an example of an array containing text.";
char arrayS[] = "And this is another array";
char *constP = "This is a pointer toa string constant";
```

These lines demonstrate three different ways of saving text. The first uses numbers to initialize the array. This is a pretty stupid example, because a much easier way would be to use "helloS[] = "HELLO". However, I wanted to show how this is done, since there will be times when you'll want to initialize an array with numbers. The next two lines initialize two arrays with text, and the last line creates a char pointer that points to the first character of a string constant. Remember that this is quite different from an array, since you can't change the contents of the string constant.

```
main()
{
  int set, status;

  Grf1();
  Screen(2);
```

```
    /* White on black */
  for (set=0; set<16; set++)
    Color(set,16,1);
```

Next we initialize the GRF1 library, set the screen to black, and the text to white.

```
  Display(1,14,&helloS[0]);
  Display(2,5,introS);
```

Here we use two different methods of passing the address of the first element of an array to a function. Remember that "introS" is the same as "&introS[0]".

```
  VDisplay(5,8,"This is a string    constant");
  VDisplay(5,13,arrayS);
  VDisplay(5,18,constP);
```

After displaying our horizontal text, we call VDisplay to display our vertical text. The first call places a string constant directly into the function call. The address of the string constant will be passed to the function. (Note: no & operator is necessary when using string constants.) The next two functions display the contents of an array, and the contents of the string constant that constP points to. I know it may seem silly to use all these different methods in one program, but I just wanted to show each technique.

```
  do
    Key(0,&status);
  while (status != 1);
}
```

After putting the text on the screen, we wait for a new key to be pressed, allowing the user to read everything on the screen before the program quits.

```
Display(myRow, myCol, string)
int myRow, myCol;
char *string;
{
  int n, chr;
  n = 0;
  chr = string[n];

  while (chr != 0)
  {

    HChar(myRow, myCol, chr, 1);
    myCol++;
    n++;
    chr = string[n];

      /* Bump to next row? */
    if (myCol > 28)
    {
      myCol = 3;
      myRow++;
    }
  }
}
```

Next we come to our Display function, which displays text horizontally across the screen, much like Extended Basic's Display At. We could have just called Locate(myRow,myCol) and then PutS(string), but I wanted to demonstrate how pointers are used, so I wrote a function that does everything from scratch. (For more information about the Locate and PutS functions, see the c99 manual. These functions are included as part of the CSUP library, and since CSUP must be included by all c99 programs, these functions can be used by any program.)

One of my first mistakes when making this function was to use char variables for myRow and myCol. When the program didn't work correctly, I looked up the documentation for HChar and found that it expects int variables for the row and column. This means it was reading two bytes of data, and I was only providing one. After fixing this, the text appeared in the correct place on the screen, but was all scrambled. I then discovered that HChar expects an int for the character it is drawing as well. I fixed this by defining the chr variable as an int, and copying each value from the char array into chr before passing it to HChar. This is done by the line "chr = string[n]", and then chr is passed to HCHar, rather than string[n]. It is perfectly legal to assign a char to an int or an int to a char, although if you do the latter, you may get unexpected results if your int is larger than 255.

```
VDisplay(startRow, myCol, string)
int startRow, myCol;
char *string;
{
  int myRow, chr;

  myRow = startRow;
  chr = *string;

  while (chr != 0)
  {
    HChar(myRow, myCol, *string, 1);
    myRow++;
    string++;
    chr = *string;

    if (myRow > 24)
    {
      myRow = startRow;
      myCol++;
    }
  }
}
```

Finally we come to the VDisplay function. To keep you from getting bored, we did this differently than the other function. Rather than assigning string[0] to chr, we assign *string to chr, which does exactly the same thing. (Remember that *string access whatever string points to, which in this case is the first element of an array.) Then instead of incrementing n, we increment string, so it points to the next character in the array. This method may not be as easy to understand, but it's more efficient, since the statement "chr = *string" results in fewer assembly lines than "chr = string[n]". (The latter is actually converted into "chr = *(string+n)" before it is assembled.)

I hope this isn't too confusing. If you get stuck, just take a break and then read this article again, carefully going over anything you don't understand, and hopefully it will all fall into place. The concept of pointers may be hard to understand at first, but they are quite easy to use and very helpful once you know how to use them.

TEXTFUN;C

```
/*************************/
/* TEXTFUN;C. SEP/OCT '97 */
/* ISSUE OF MICROPENDIUM  */
/* BY VERN JENSEN         */
/*************************/

#include "DSK2.GRF1;H"

char helloS[] = {72,69,76,76,79,0};
char introS[] = "This is an example of an array containing text.";
```

```
char arrayS[] = "And this is another array";
char *constP = "This is a pointer toa string constant";

main()
{
  int set, status;

  Grf1();
  Screen(2);

    /* White on black */
  for (set=0; set<16; set++)
    Color(set,16,1);

  Display(1,14,&helloS[0]);
  Display(2,5,introS);

  VDisplay(5,8,"This is a string    constant");
  VDisplay(5,13,arrayS);
  VDisplay(5,18,constP);

  do
    Key(0,&status);
  while (status != 1);
}


Display(myRow, myCol, string)
int myRow, myCol;
char *string;
{
  int n, chr;
  n = 0;
  chr = string[n];

  while (chr != 0)
  {

    HChar(myRow, myCol, chr, 1);
    myCol++;
    n++;
    chr = string[n];

      /* Bump to next row? */
    if (myCol > 28)
    {
      myCol = 3;
      myRow++;
    }
  }
}


VDisplay(startRow, myCol, string)
int startRow, myCol;
char *string;
{
  int myRow, chr;

  myRow = startRow;
  chr = *string;

  while (chr != 0)
  {
    HChar(myRow, myCol, *string, 1);
```

```
    myRow++;
    string++;
    chr = *string;

    if (myRow > 24)
    {
      myRow = startRow;
      myCol++;
    }
  }
}
```

Beginning c99 - Part 7

All About the #include Statement

By Vern Jensen

For a while now I've been using something in each program that I haven't explained very much. You see it at the beginning of each program with the line:

```
#include “DSK2.GRF1;H"
```

The #include statement simply inserts the named file into your source code in the exact spot that the #include statement is encountered. This is done when the program is compiled. You can use the #include statement in many ways. When working on Virus Attack, it allowed me to break my source code up into several files, since the program was too long to fit into the Editor as a single file. The first file had several #include statements at the end that included all the other source files for Virus Attack. And when the program was compiled, the compiler saw them all as one single file.

However, a more common use of the #include statement is to allow the user to access the functions in a library, such as the GRF1 library. A library is simply a set of functions written in either c99 or assembly that have been compiled and assembled. The resulting object code can then be loaded along with your program at run time, allowing your program to use those functions. However, how does the compiler know that these functions you are using in your code are going to be loaded later? The answer is in the extern statement.

When you want access to a function that is part of a library that you are going to load at run time, you must name that function with the extern statement before using it. For instance, if the only functions we were going to use in the GRF1 library were the Grf1(), Screen(), ChrDef(), and HChar() functions, we could put this statement at the beginning of our program:

```
extern Grf1(), Screen(), ChrDef(), HChar();
```

However, we would only be able to use those functions, since the C compiler has no way of knowing about a function that is part of a library unless we specifically tell it about the function, since the library is loaded after the program is compiled. Now would be a good time to boot up Funnelweb, fire up the editor, and take a look at the file GRF1;H. (It should be on your work disk, normally in drive 2.) If you do, you'll notice that GRF1;H is simply a bunch of extern statements, naming every single function in the GRF1 library. So #including GRF1;H in your program is an easy way to make all the functions in GRF1 available to your program, although you still have to remember to load the GRF1 library later.

The "H" at the end of GRF1;H stands for "Header". The C standard is that files containing extern statements should be called headers. This makes it clear that it's not a program, and not a library when cataloging your disk. The original c99 4.0 release had the file named as GRF1RF, the RF probably standing for "References" (extern statements are converted into assembler REF statements). I changed this to make it conform more to the C standard.

In addition to sharing functions between object modules, you can also share variables. For instance, if a certain library contains a global variable called myVar that you'd like access to, you can add this statement to your program:

```
extern myVar;
```

Note, however, that for you to have access to this variable, the library must specifically make it available to other object modules with the entry statement, described below. Notice also that there are no parentheses after myVar. If you use parentheses, the c99 compiler assumes it is a function name, and if you don't, it assumes it is a variable name.

If you want to create your own library, you can do so quite easily. Just as you use the extern statement to make external functions and variables accessible to your program, you use the entry statement to make local functions and variables accessible to other object modules. However, it is important to realize that parentheses are not used in entry statements. So to create a library containing a single function, we could write something like this:

```
/* POWER/C */

entry Power;

Power(theNum, power)
int theNum, power;
{
  int result;

  result = 1;
  while (power > 0)
  {
    result = result * theNum;
    power--;
  }

  return result;
}
```

Since function parameters are just copies of the values passed to the function, we can modify the parameters without fear of changing any values passed by the caller. In this case, we change the power parameter, decrementing it until it reaches 0, and multiplying the result by theNum each iteration through the loop.

Go ahead and type in the code above, saving it as POWER/C. Then compile and assemble it, saving the compiled result as POWER/S, and the assembled object code on disk 2 as POWER/O. You've just created your first library! Now type in the test program below, saving it as POWTEST/C.

```
/* POWTEST/C - FROM THE NOV/DEC */
/* ISSUE OF MICROPENDIUM        */

  /* This lets us use these external functions */
extern Power(), PutNum(), DisStr();

main()
{
  int n, row;

  row = 5;
  for (n=0; n <= 14; n++)
  {
    DisStr(row,2,"2 to the power of");
    PutNum(row,20,n,0);
    DisStr(row,23,"is");
    PutNum(row,26,Power(2,n),0);
    row++;
  }
}
```

After saving the program above, purge the contents of the editor and type in this CLOADER file, saving it as POWTEST/L.

```
DSK2.CSUP
DSK2.POWTEST/O
DSK2.POWER/O
DSK2.PUTNUM/O
```

This will load CSUP as usual as well as POWTEST/O. However, we also load our library, POWER/O, which was compiled and assembled earlier. The last file, PUTNUM/O, is one of Bruce Harrison's excellent c99 utilities. You'll need to copy it from his disk (included in the c99 starter kit) onto your "work disk", which should be in drive 2. It provides the functions PutNum and DisStr, for displaying numbers and strings on the screen. Run the program PRINTINST on Harrison's disk to print the instructions for all the c99 utilities on it.

Now compile and assemble POWTEST/C, saving the results as POWTEST/S and POWTEST/O. Then run CLOADER, entering DSK2.POWTEST/L at the prompt. If everything goes as planned, you should see a screen full of the results of multiple calls to the Power function. Congratulations! You've just created your first library and used it in an actual program.

Most libraries are written in assembly language, since it's much faster and more compact if you write it in assembly than if you write it in C. However, there is nothing to stop you from writing libraries in C, as we have just done. I wrote several utilities for Virus Attack in C and compiled them into a library, although before the program was finished I ended up writing to Bruce Harrison, asking him to make faster versions of my routines. Without his help, Virus Attack wouldn't be the same as it is today. This is often how C programs are written, even on modern computers; you write the bulk of the code in C, and the time-critical portions in assembly. This allows you to produce a fast application, but without having to write everything in assembly.

Besides using the #include statement to include header files, you can also use it to include source code directly, as I've already mentioned. So instead of writing POWER/C as shown above, we could eliminate the entry statement, and include POWER/C itself in our program, rather than using the extern statement and loading POWER/O at run time. This produces a more compact result, since everything is put in a single object file, instead of two. This can be important when you write large programs, since eventually you'll run into the limits of the 99/4a's memory, and will want to do everything possible to keep your program small enough to load.

There is, however, a drawback to including source code directly into your program instead of loading it later as an object file at run time. The drawback is that the source code must be compiled every single time you compile your program, resulting in an additional delay. How much of a delay it adds depends on how long the file is, and whether it's written in C or assembly. (Assembly goes much faster, since it doesn't need to be compiled.) In fact, the RANDOM;C file on the c99 Libraries disk uses this exact method; you include RANDOM;C directly into your program, rather than loading a library at run-time. It doesn't add much of a delay to the compilation process, since the routines in RANDOM;C are written in assembly. (Load the file and see for yourself.)

By now you probably know more about the #include statement and libraries than you ever wanted to know. So I'll leave you to ponder what we've just covered, while I consider what to write about next issue.

Contact Information

We're now on our 7th article, and I'm curious to know how many of the 20+ people who ordered the c99 Starter Kit are still on board. Do you find the articles helpful? Confusing? I'm especially interested in hearing from those who had never programmed in c99 before I started this series. Do you now find yourself able to code simple programs, or did you get lost a couple of articles back? Knowing where you are at will help me as I write articles in the future.

My new email address is Jensen@loop.com. And as always, you can write to me at Vern L. Jensen, 910 Linda Vista Ave., Pasadena, CA 91103.

Beginning c99 - Part 8

Dealing With Memory Requirements

By Vern Jensen

As you become more experienced with c99 and start writing larger programs, you will discover that it is important to keep your code as compact as possible. This is because both the code for your program and all of your variables must load into the TI's memory at run time. The longer your code, the less room you have for variables, and vice-versa. So writing compact code would be especially important for applications that need a lot of data space, such as a word processor.

However, it is important for all applications, since if you don't watch out, your program will eventually become too large to load. This happened when I was developing Virus Attack. I added so much to the game that it wouldn't fit into the TI's memory until Bruce Harrison wrote a special routine that would load it into both high and low memory.

While we have the luxury of writing in a simple, fast language, it does have its drawbacks. One is that a single c99 statement may be compiled into may assembly language statements, making it difficult to write compact code. However, by keeping this in mind, there are some tricks you can do to make your program shorter. For one, putting repetitive statements in loops can help immensely.

Consider the task of setting up the graphics for a game. In Extended Basic, you would typically make repeated CALL CHAR statements. However, in c99, function calls are probably one of the most "expensive" statements in terms of the amount of assembly code that is needed for each c99 statement. Instead of making repeated ChrDef calls, if we place the data for each group of characters in an array, we can loop through the array and only have to write a single ChrDef call, which results in fewer assembly statements than there would be if we made several separate ChrDef calls.

However, we now run into another problem; to my knowledge, c99 provides no way to initialize character arrays with strings longer than one line, thus limiting your array data to 80 characters. However, with a little trickery, we can get around this limitation. We can create one big array by placing several small arrays in a row, one right after the other. Then we can access the first one as if it were as big as the sum total off all the arrays combined. When we get outside the bounds of the first array, we drop into the second one.

One thing to keep in mind though is that if you initialize your arrays with character string constants, as I do below, then each array will be appended with a 0 to mark the end of the string. This means than an array with 64 characters will actually take up 65 bytes.

In addition, if you take a look at the assembled output of these lines after they are compiled, you will notice each array is ended with the assembly language EVEN statement. I must confess my complete assembly language ignorance here. However, my guess is that the EVEN statement aligns the next block of data so that it starts on an even memory address. And since 65 is not an even number, the EVEN statement will place the next block of data 1 byte beyond the end of the previous block of data. This means that if you have a 64 character array like the one below, and you want to start reading from the next array, you should read from element 66. In other words, cData[66] below is equivalent to cData2[0]. The program below actually compiles. Try it out if you want.

```
#include "DSK2.GRF1;H"

char chrNum[] = {65,69,73,77,0};

char cData[] =
  "0000000000000000111111111111111122222222222222223333333333333333";
char cData2[] =
  "4444444444444444555555555555555566666666666666667777777777777777";
char cData3[] =
  "8888888888888888999999999999999AAAAAAAAAAAAAAABBBBBBBBBBBBBBBB";
```

```
char cData4[] =
  "CCCCCCCCCCCCCCCCDDDDDDDDDDDDDDDDEEEEEEEEEEEEEEEEFFFFFFFFFFFFFFFF";

main()
{
  int n;
  Grf1();

  for (n=0; n <= 15; n++)
    VChar(5,n+10,65+n,15);

  InitChar();

  while (1);
}

InitChar()
{
  int c;
  c = 0;

  while (chrNum[c] != 0)
  {
    ChrDef(chrNum[c],&cData[c*66]);
    c++;
  }
}
```

When InitChar is called, the code loops through all elements of the chrNum array until it hits 0. The chrNum array is used to specify the character set for each block of data. Next, the address of the appropriate section of the cData array is passed to the ChrDef function. When the element number exceeds the bounds of cData, it spills into cData2, cData3, and cData4.

Normally you don't have to use tricks like this in standard C, since standard C provides a way to break long strings up between several lines, but since we don't have this capability in c99, we must make do with what we have. Of course, if you don't have room for any Char data at all in your program, you could move it to a separate file, then have your program open that file and reads its contents when starting up. However, this technique generally isn't too popular, since it adds an additional delay to the program's loading time, even if it saves a little memory. I imagine most users should find the technique described above more than adequate for keeping your InitChar function as compact as possible.

If you've never done it before, you should compile a c99 program sometime with the "Include C Source" option turned on. Then open up the resulting assembly language output file with the editor and take a look at the number of assembly language statements necessary for each c99 statement. (The program above would be a perfect program to do this with.) Knowing how much each c99 statement "costs" in terms of program length will help you when writing your program, since you will know how to write code in the most efficient way possible. Typically, simple variable assignments, if statements, and arithmetic operations require fewer assembly language statements than the "bigger" c99 statements, such as function calls or complicated if statements.

Bits and Bytes

Writing more compact code isn't the only way to make a program take up less memory. If you need to keep track of lots of "on/off" choices, such as the preferences settings a user has selected, you can change the individual bits of a number, where each bit corresponds to one of your choices, rather than devoting an entire variable to each choice.

For example, a while back I was considering working on a c99 game where the player would walk around in a maze of rooms, each room filling the screen. The rooms would contain various objects that the player could pick up. When the player leaves a room and comes back, the objects he picked up should not reappear.

I wanted the game to contain around 50 or so rooms, and I wanted each room to be able to contain at least 10 objects. If I were to create a char array with one element for each object, each element specifying whether the corresponding object had been picked up, my array would take up 500

bytes! This would obviously leave me less room for actual program code, since the data for the objects alone takes up a good chunk of memory.

A better way to do it would be to dedicate a single int variable to each room, and set the individual bits of that variable to indicate whether the corresponding object has been picked up. A 0 bit would mean it hasn't been picked up yet, while a 1 bit would indicate that it has. And since an int has 16 bits, we could have 16 objects per room, while only requiring a total of 100 bytes (2 bytes per room * 50 rooms).

When each room is loaded from disk, my program would scan through the room and store the position of each object in a two-dimensional array with 16 elements in the first dimension (one element for each object), and 2 elements in the second dimension. The second dimension of the array would be used to store each object's row and column on the screen. So the code for scanning a level might look something like this:

```
char objArray[16][2];

ScanLevel()
{
  int row, col, char, objNum;
  objNum = 0;

  for (row=1; row <= 24; row++)
  {
    for (col=2; col <= 30; col++)
    {
      char = GChar(row,col);
      if (char == kObj1 || char == kObj2 ||
          char == kObj3 || char == kObj4)
      {
        objArray[objNum][0] = row;
        objArray[objNum}[1] = col;
        objNum++;
      }
    }
  }
}
```

Once an object is picked up, this array would be scanned to find the array element with the matching row and column. This array element is the object's "ID". This method ensures that each object on the screen gets its own unique ID, and these IDs will be the same each time the level is loaded, since the room is always scanned in the same order. Once we know an object's ID, all we have to do to mark that object as being removed is to set the corresponding bit in the variable that keeps track of objects for the current level. That code might look something like this:

```
  /* Below is an array of 50 integers, one for each room, */
  /* for keeping track of which objects have been picked up. */
  /* You'd want to set each element to 0 before starting the game. */
int ObjData[50];

StoreObject(room, row, col)
int room, row, col;
{
  char objID;
  objID = 0;

    /* Scan objArray to find the ID for this object */
  for (objID=0; objID <= 15; objID++)
  {
    if (objArray[objID][0] == row &&
        objArray[objID][1] == col)
      break;
  }
```

```
    /* Mark this object as having been removed. */
  SetBit(&ObjData[room], objID, 1);
}
```

The SetBit function is covered later in this article. You simply pass it the address of an integer, the position of the bit you wish to change (0-15), and whether you want the bit changed to a 1 or a 0.

To keep objects from reappearing when the player reenters a room, we must first build the array that keeps track of the position of all objects in the room (the objArray), then we must look at each bit in the variable that keeps track of what objects in the room have been picked up. Naturally, each object that hasn't been picked up shouldn't be there, so we erase it with a call to the HChar statement:

```
EraseObjects()
{
  int row, col, char, objID, r, c;
  objID = 0;

    /* Scan entire room for objects */
  for (row=1; row <= 24; row++)
  {
    for (col=2; col <= 30; col++)
    {
      char = GChar(row,col);

        /* Did we find an object? */
      if (char == kObj1 || char == kObj2 ||
          char == kObj3 || char == kObj4)
      {
          /* Determine the object ID for this object */
        for (objID=0; objID <= 15; objID++)
        {
          if (objArray[objID][0] == row &&
              objArray[objID][1] == col)
            break;
        }

          /* Has this object been picked up? */
        if (GetBit(ObjData[room],objID) == 1)
        {
            /* If so, erase it! */
          r = objArray[objID][0];
          c = objArray[objID][1];
          HChar(r,c,32,1);
        }
      }
    }
  }
}
```

Once again, the GetBit() function is covered later in this article. It simply returns the value of a single bit from an integer. You pass the integer to the function, and the position of the bit you wish to read (0-15). Don't spend a lot of time trying to understand the example functions above. If they don't make sense, it doesn't matter - the important part is what is coming below. The code above is provided simply to demonstrate one situation where you might want to set the individual bits of a number, rather than dedicating an entire variable to each on/off choice. I should also warn you that the functions above are not tested. They should work, although it's possible I may have made a mistake when writing the code. (Nobody's perfect.)

A Little Background

As you probably know, variables are stored in the computer as a series of bits, which are simply "on/off" switches with a value of either 0 or 1. A Char variable takes up a Byte, which is a term meaning 8 bits. An int takes up 2 bytes, so it contains 16 bits. In Base-2, or Binary format, the right-

most bit of a number is the one's place, the bit to the left of that is the two's place, the next is worth 4, the next 8, then 16, 32, 64, 128, 256, and so on. So this is what the number 11 would look like in Binary format:

```
00001011
```

Look at which bits are on and which are off. The bits in the 1, 2, and 8 places are turned on. So 1+2+8 = 11. Here's what a Char variable would look like with all its bits turned on:

```
11111111
```

That would be 1+2+4+8+16+32+64+128, or 255, the maximum value of a Char variable. Here's another example, with the binary equivalent of 76 (64+8+4):

```
01001100
```

Binary Operators

The C language provides several operators for changing a number at the bit level. The simplest of these is the shift operator, which we will look at first. There are actually two shift operators, "<<" and ">>", which shift the bits of a number left or right a specified number of bits, filling vacated bits with 0. Here's an example that shifts the bits stored in x left two positions. The result is then copied back into x:

```
x = x << 2;
```

If x had been 00101110 (46), it would become 10111000 (184). Or if x had been 01001010 (74), it would become 00101000 (40). Notice that we lose the 1 bit on the far left because it is shifted right out of the number. Shifting a number left 2 places is the same as multiplying the number by 4. (4x46 = 184. And 4x74 = 296, although as that number exceeds the bounds of a char variable, the result is 296-256, or 40.) Shifting a number left by only 1 is the same as multiplying that number by 2.

Right shifting a number is similar to left shifting. Here's an example that shifts the bits in x 4 places to the right:

```
x = x >> 4;
```

That would turn 11110000 into 00001111, and 10101011 into 00001010. Next we come to the "one's complement" operator ~ (FCTN-W on the keyboard), which converts each 0 bit into a 1 bit and vice versa. So if n below is 00100111, it would be turned into 11011000:

```
n = ~n;
```

You can also use this operator in combination with other statements. Here's an example that shifts the bits in b right c positions, then inverts the bits, storing the result in a. It should be noted that b and c are not changed by this operation; a is the only variable that is changed.

```
a = ~(b>>c);
```

Next we come to the bitwise AND operator "&", not to be confused with the logical AND operator commonly used in IF statements ("&&"). The bitwise AND operator compares two numbers, setting each destination bit to 1 only if the corresponding bit of both numbers is also 1. For example, the statement "n = a&b" would set n to the result of the following operation, where a and b are the top two numbers:

```
10111001 (a)
00011101 (b)
--------
00011001 (n)
```

Here you can clearly see how each bit in the result is turned on only if the corresponding bit in both a and b is on. The bitwise AND operator is usually used to mask off a certain set of bits. For instance, the statement "n = a&7" sets n to the first three right-most bits of a. (Remember that 7 is equal to 00000111 in binary.) That is, if a was 11010110, then n becomes 00000110.

You can use the & and >> operators together to determine whether a specific bit of a number is on or off. Here's a function that returns whether the specified bit is on or off:

```
GetBit(theNum, theBit)
int theNum;
char theBit;
{
  if (theBit == 0)
    return myNum & 1;
  else
    return (theNum >> theBit) & 1;
}
```

To call this function, simply pass an integer and the position of the bit you would like to read. Keep in mind that since integers can be both positive and negative, the left-most bit is used to specify whether the number is negative or positive. Passing a value of 0 as the theBit parameter will read the right-most bit (the 1's place), a value of 1 will read the bit to the left of that, and so on. Since there are 16 bits in an int, you can pass any value between 0 and 15 as the theBit parameter.

Now lets take a look at how the functions works. Lets say you call the function with an integer of 99 and a theBit parameter of 5. (Remember that since the first bit position is 0, a value of 5 will read the 6th bit.) A value of 99 is the following in Binary (64+32+2+1):

```
0000000001100011
```

As you can see, the 6th bit from the right is a 1, so the function should return true. But first, let's take a look at how the function works. First the bits in theNum are shifted right by the value in the theBit parameter, resulting in this number:

```
0000000000000011
```

Next, this number is used with the AND operation to get the right-most bit:

```
0000000000000011
0000000000000001
----------------
0000000000000001
```

The resulting number is returned to the user, which in this case is a 1. If the theBit parameter had been a 4 instead of a 5, a 0 would have been returned. You may have noticed that a special case is added for when the theBit parameter is a 0. This is because for some reason, the shift operators don't work correctly on the TI when used with a value of 0. Normally, shifting a number right or left 0 bit positions should do absolutely nothing, but attempting to do this on the TI results in code that doesn't work right. By adding the special case to the GetBit function, we avoid this problem.

The OR Operators

By now you may be wondering how one would write a function to set a particular bit. This leads us to the OR operators. The first version, the bitwise OR operator |, turns the destination bit on when the corresponding bit in either or both of the source numbers is on. Here's an example:

```
char a, b, c;

a = 150;
b = 227;

c = a | b;
```

Again, the bitwise OR operator "|" is not to be confused with the logical OR operator "||" that you use in IF statements. In the operation above, the bits in C are turned on when the same bits in either a or b are on. Here's an illustration:

```
10010110 (a)
```

```
11100011 (b)
--------
11110111 (c)
```

The other OR operator ^, called the exclusive OR operator, turns the destination bit on when the same bit in either a or b is on, but not when both bits are on:

c = a ^ b;

```
10010110 (a)
11100011 (b)
--------
01110101 (c)
```

Below is a function which uses the bitwise OR operator to turn a single bit on or off, leaving the other bits unchanged:

```
SetBit(theNum, theBit, onOff)
int *theNum, onOff;
char theBit;
{
  if (onOff)
  {
    if (theBit == 0)
      *theNum = *theNum | 1;
    else
      *theNum = *theNum | (1 << theBit);
  }
  else
  {
    if (theBit == 0)
      *theNum = *theNum & ~1;
    else
      *theNum = *theNum & ~(1 << theBit);
  }
}
```

To call this function, pass the address of an integer you wish to change, the position of the bit to be modified, and a 1 or a 0. If the bit is to be set to 1, the left shift operator is used to move a 1 into the correct bit position, then the OR operator is used to add that bit to the number. If the bit is to be set to 0, the one's compliment operator is used to get a mask that has all bits turned on except for the bit to be set to 0, and this mask is used with the AND statement to turn that bit in the number off.

Again, remember that when you call this function, the theBit parameter can be any value from 0-15, where 0 indicates the first bit on the right, 1 indicates the second bit from the right, and so on. And just like earlier, we must add a special case for when the theBit parameter is 0, to avoid shifting a number 0 positions.

A Test Program

Together, the left and right shift operators, the AND and OR operators, and the one's complement operator provide everything needed to operate on a number at the bit level. To conclude this article is a program which lets you toggle the individual bits of a number between 1 and 0, and then shows the result in both binary and decimal format.

Before getting started, you'll need to copy the ACCNUM/O file from Bruce Harrison's c99 Utils disk (which comes with my c99 Starter Kit) to your work disk in drive 2. You should also copy the PUTNUM/O file to your work disk if you didn't last issue.

In addition, you'll need to type in the BPHONK assembly code below, saving it in its own file called BPHONK on your work disk. As you may have guessed, it contains two useful routines for generating the standard beep and honk sounds. The BPHONK file was originally included with c99 release 4, but somehow didn't make it into my starter kit. To use the routines just add this line to your program:

```
#include "DSK2.BPHONK"
```

Then you can invoke a beep or a honk simply by calling the function with the appropriate name, such as Beep(), or Honk(). Since the code is compiled along with your program, there is no need to load a separate library at run time.

BPHONK

```
#asm
 REF C$GPLL
BEEP BL @C$GPLL
 DATA >34
 B *13
HONK BL @C$GPLL
 DATA >36
 B *13
#endasm
```

BITTEST/L

```
DSK2.CSUP
DSK2.GRF1
DSK2.ACCNUM/O
DSK2.PUTNUM/O
DSK2.BITTEST/O
```

BITTEST/C

```
/* BITTEST/C - FROM THE JAN/FEB */
/* 1998 ISSUE OF MICROPENDIUM   */

#include "DSK2.BPHONK"
#include "DSK2.GRF1;H"
extern PutNum(), AccNum(), DisStr();

int theBit, oldBit;
int theNum = 0;

main()
{
  int s;

  Grf1();
  Screen(8);

  DisStr(3,6,"B I T   T E S T E R");
  DisStr(8,2,"Modify which bit? (0-15):");
  DisStr(13,2,"Decimal");
  DisStr(17,2,"Binary");

  while (1) /* Endless loop */
  {
      /* Display the number in both */
      /* decimal and binary format. */
    PutNum(13,10,theNum,0);
    PutBinary(17,10,theNum);

    Beep(); /* Part of BPHonk */
    do
    {
        /* Get the bit to be changed */
      theBit = AccNum(8,28,1);

        /* Wait for enter key to be released.    */
```

```
        /* (AccNum should do this, but doesn't.) */
      do
      {
        Key(0,&s);
      } while (s);

        /* Honk if invalid number */
      if (theBit < 0 || theBit > 15)
        Honk();
    } while (theBit < 0 || theBit > 15);

      /* Toggle bit between 0 and 1 */
    oldBit = GetBit(theNum,theBit);
    SetBit(&theNum,theBit,!oldBit);
  }
}


PutBinary(myRow, myCol, myNum)
char myRow, myCol;
int  myNum;
{
  char n;
  int myBit;

    /* Draw each of the 16 bits */
  for (n=0; n <= 15; n++)
  {
    myBit = GetBit(myNum, 15-n);
    HChar(myRow, myCol+n, '0' + myBit, 1);
  }
}


GetBit(myNum, myBit)
int  myNum;
char myBit;
{
  if (myBit == 0)
    return myNum & 1;
  else
    return (myNum >> myBit) & 1;
}


SetBit(myNum, myBit, onOff)
int *myNum, onOff;
char myBit;
{
  if (onOff)
  {
    if (myBit == 0)
      *myNum = *myNum | 1;
    else
      *myNum = *myNum | (1 << myBit);
  }
  else
  {
    if (myBit = 0)
      *myNum = *myNum & ~1;
    else
      *myNum = *myNum & ~(1 << myBit);
  }
}
```

Beginning c99 - Part 9

Wrapping Things Up

By Vern Jensen

This will be the last of my c99 articles. I know this may come as a surprise to some of you, but I never intended for the series to go on for a long time. As you will notice from the title, this series was intended mainly to get you started with c99. Having done that, I hope you now know enough to continue on your own.

So today we will cover a few odds and ends, and then wrap things up with a complete working program - a basic "snake eats apple" type game. But first, I have a correction to make. Last issue, I stated that there was no way to extend a line beyond the Editor's 80-character limit. This means character initializer statements, such as the ones used in last issue's first example program, were quite limited.

However, thanks to experimentation by Phil Van Nordstrand, a partial solution was found. He discovered that you actually can break up a single statement between multiple lines by using the backslash (\) character at the end of each line. So the statements from last issue's program might have been written like this:

```
char cData[] =
"00000000000000001111111111111111222222222222222233333333333333333\
44444444444444445555555555555555666666666666666677777777777777777\
88888888888888889999999999999999AAAAAAAAAAAAAAAABBBBBBBBBBBBBBBB\
CCCCCCCCCCCCCCCCDDDDDDDDDDDDDDDDEEEEEEEEEEEEEEEEFFFFFFFFFFFFFFFF";
```

Unfortunately, when I tried compiling this, the compiler gave me a "Line too long" error. Apparently, this technique can be used only when the total combined length of the line is not too long. (I'm not sure what the limit is.) So the backslash character may be helpful in some situations where you need just a bit more space, although it won't work for really long lines.

More On Pointers

I realize that my article on pointers a while back maybe have been a bit confusing, so I've decided to do a little review, this time with pictures to help you visualize what I'm talking about. As you may recall, pointers are variables that store the address of other variables in memory. To create a pointer, simply declare a variable as usual, adding an asterisk in front of the variable name:

```
char *charPtr;
int *intPtr;
```

The char and int specifiers only tell the compiler what type of variable these pointers will be pointing to. (i.e. charPtr will only point to char variables, while intPtr points to integers.) Both pointers are actually the same size, taking up 16-bits (the size of an Int). That is because pointers need all 16 bits to be able to specify all the various memory locations in RAM that their destination variable could be sitting in. So don't let the char and int specifiers confuse you - when applied to pointers, they only specify what type of variable the pointer will point to; it doesn't specify how much memory the pointer itself takes up.

As you may recall, assigning the address of a variable to a pointer is done with the & (address of) operator. So this would make our pointers above point to the variables below:

```
char myChar;
```

```
int myInt;

charPtr = &myChar; /* Store address of myChar in charPtr */
intPtr = &myInt;  /* Store address of myInt in intPtr */
```

If you were to take a look at the values contained in charPtr and intPtr, you would most likely find a 5-digit number (such as 31298) that specifies the physical location of the myChar and myInt variables in memory. Naturally, you don't want these numbers - you want to be able to access the variables these pointers are pointing to. To do that, you use the dereferencing operator (*):

```
*charPtr = 5;   /* myChar = 5 */
*intPtr = -500; /* myInt = -500 */
```

Whenever you use the dereferencing operator with a pointer, you gain access to whatever the pointer points to. So the statements above actually store 5 in myChar, and -500 in myInt. You can read pointers in the same way:

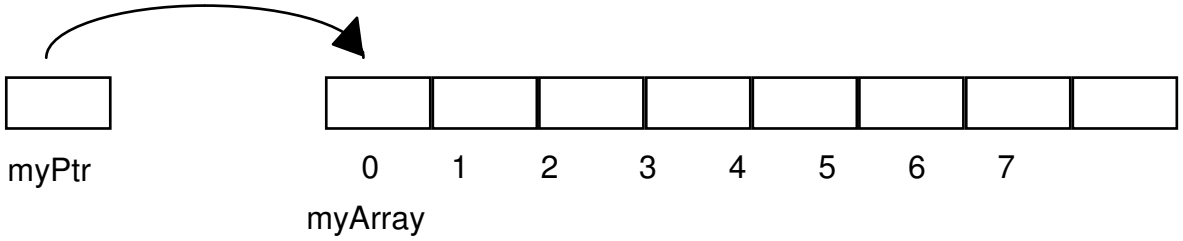
```
char a;
int b;

a = *charPtr; /* Assign contents of myChar (5) to a. */
b = *intPtr; /* Assign contents of myInt (-500) to b */
```

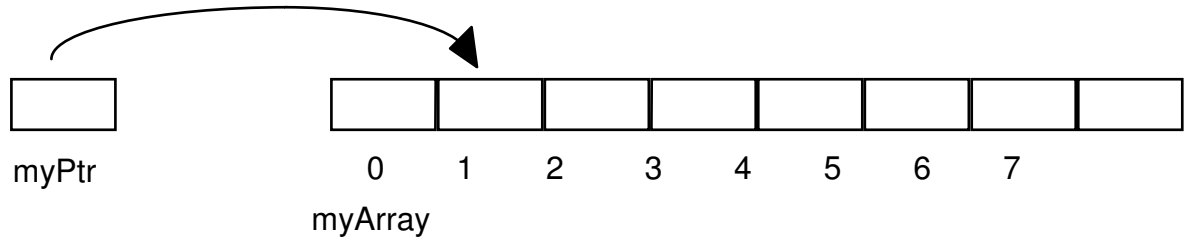
So far, so good, right? But things get interesting once you add arrays to the dimension. First of all, what is an array? It is simply a bunch of char or int variables laid out consecutively in memory. So if you have a pointer pointing to the first element of an array, and you increment the pointer, its value will now point to the next element of the array. Below is some code, as well as a visual example of what happens.

```
int myArray[8], *myPtr;

myPtr = myArray[0]; /* Point to the first element of the array */
```



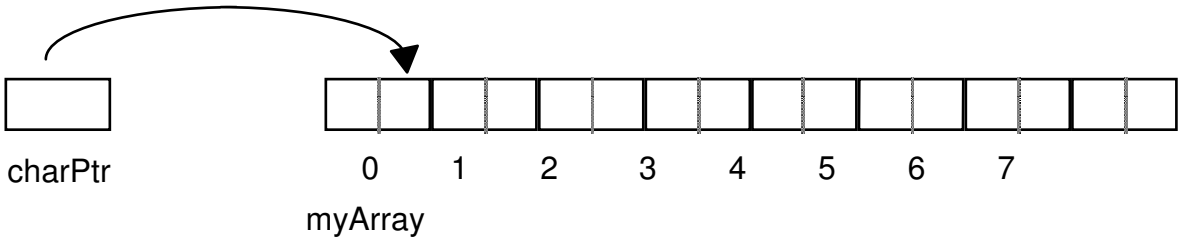
Now when you use *myPtr, you're actually reading the first element of the array. When you increment the pointer (myPtr++;), the pointer's address is changed, making it point to the next element of the array.



Keep in mind that incrementing the pointer itself is much different from incrementing what it points to:

```
myPtr++;          /* Increment pointer */
(*myPtr)++; /* Increment what it points to. */
```

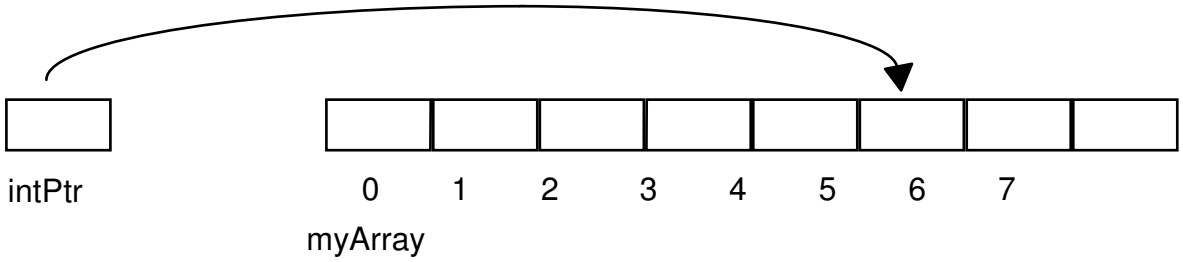
When you increment a pointer as we do in the example above, the pointer is moved 1 byte forward in memory if it is a char pointer, or 2 bytes forward in memory if it is an int pointer. This is why the array example above works correctly. When we increment myPtr, it is moved 2 bytes forward in memory (2 bytes is the size of an int), thus making it point to the very next element in the array. If our pointer were a char pointer instead of an int pointer, incrementing it would move it forward only 1 byte in memory, which obviously wouldn't work correctly if pointing to an int variable:



The dotted lines above show the separation between the two bytes that make up each int element. As the diagram shows, if a char pointer were used to point to this array, incrementing it would result in a pointer that points to the second byte of the first element. So now you know what the difference is between char and int pointers, and why an int pointer should always point to an int variable (or array), and a char pointer should always point to a char variable or array.

The rule described above holds true for adding or subtracting any value from a pointer. Here's an example:

```
intPtr = myArray[0];
intPtr = intPtr + 5;
```



What's the Fuss About?

Naturally, this may seem like overkill. Who needs pointers, anyway? You get along in Basic without them perfectly well. But in C, things are different. For one thing, you need to pass the address of a variable to a function if you want that function to be able to change the variable, as demonstrated by functions such as the Key function in GRF1:

```
int s, k;
s = Key(0, &k);
```

In addition, you can pass the address of an array to the function, and then use the function's pointer as if it were the original array, thus allowing you to access all of its elements, as well as make changes to the array if necessary.

I hope this clears up any confusion any of you may have had about pointers. The concept is really quite simple once you get used to it, and enables you to do many things that would otherwise be impossible.

Our Final Project - A Game

I thought it would be fitting to conclude this series with a "real" application; one that actually does something, which in this case is a game. I actually started this game a long time ago, and only recently got around to finishing it, so not everything in the code is necessarily the best way to do things. For example, my global variables don't start with "g", making them much more likely to conflict with local variables. However, I'm sure you will find that most of the code is pretty clean.

To run this program, you will need to copy the files SOUND and SOUND;H from the c99 Libraries disk to your work disk. There are actually two sound libraries, the other one called SND;O. Since I used SND;O in Virus Attack, I decided to test out the other library for this project.

Debugging Your Code

Debugging is hands-down the hardest part of c99 development. I ran into many problems while working on Virus Attack, many of which were easy to fix, some of which were harder, and some which were nearly impossible. One or two bugs I fixed by changing my code to something similar that does the same thing, but using a slightly different method. I did this in exasperation after not being able to figure out what was causing the problem. To this day I still don't know what the problem was; I was just glad enough to have it gone when I changed my code slightly.

When you create any program that's longer that a simple test application, you are going to run into bugs. And lots of them. My suggestion is this: get a piece of paper and write down all the things you see wrong with your program when you run it. Include ideas you may have for fixing those problems. Then quit and load up the source code and work on fixing each problem. If you can't figure something out, remember to keep an open mind! The problem may be completely different from what you had suspected; something totally "unrelated" to the bug.

And finally, don't be afraid to add tests to your code. For instance, if you're not sure whether a piece of code is being executed, you may add a call to HChar within that code to place a certain character at a specific location on the screen. Then when you run the program, if you see that character, you know the code is being executed.

Finally, print your source code! If often helps to be able to go through your program, line-by-line, seeing exactly what is happening in a certain spot, while running the program on your TI.

As some bug-hunting examples, while I was working on the Snake game for this issue, I ran into two particularly nasty bugs. One was that whenever the snake moved up, the image for the snake moving left was drawn, rather than the image for the snake moving up. The code that handles this looks like this:

```
if (vDelta == -1)
  HChar(headRow, headCol, cHead, 1);
else if (hDelta == 1)
  HChar(headRow, headCol, cHead+1, 1);
else if (vDelta == 1)
  HChar(headRow, headCol, cHead+2, 1);
else
  HChar(headRow, headCol, cHead+3, 1);
```

Since I didn't know whether the first HChar statement was indeed being called and just not operating as expected, or if the first if statement wasn't working right, I modified the first HChar statement to draw an apple instead of the snake's head. Since I still got the same results after compiling and running, I knew the first if statement wasn't working right. But why?

It turns out the problem wasn't in this section of code. Is was where I declared the variables! I had stupidly declared the vDelta and hDelta variables as chars instead of ints. But chars can't have negative values! So that's why the if statement wasn't working correctly. But here's the funny part. The code below for moving the snake's position did work correctly, even though the variables were chars:

```
headRow = headRow + vDelta;
headCol = headCol + hDelta;
```

Why this worked even when the snake moved up or left is beyond me. But the point is to never assume you know what the problem is, and especially don't assume you know what the problem isn't!

The other major problem I ran into wasn't mine: it was due to the sound library I was using. It turns out the SOUND library on the c99 Libraries disk apparently has a bug in the Sound2() function that plays 2 sounds at once. For some reason it plays them much longer than it should. The solution? Either use Sound1() or Sound3() (which I haven't tested), or use the SND;O library instead. (Which I used in Virus Attack, and works beautifully.)

GetTime

One of the coolest functions in this code was actually written by Bruce Harrison. Towards the end of the program, you will see a function called GetTime. This is a simple function that returns the number of 60ths of a second that have passed since the previous call to GetTime. This code is invaluable for keeping your program running at a constant speed.

First, call GetTime to clear the counter to 0, then call it later in your code to find out how much time has passed since the first call. So pseudo-code for the main loop of your program would look something like this:

#define kDelay 5

```
RunGame()
{
  char ticks;

  GetTime(); /* Clear timer, ignoring value returned */

  PlayGame();

    /* Make sure minimum amount of time has passed */
  ticks = 0;
  do
  {
    ticks = ticks + GetTime();
  } while (ticks < kDelay);
}
```

The code above makes sure the PlayGame routine is not called any more frequently than every five 60ths of a second. If the PlayGame routine goes faster than this, the do loop at the bottom of RunGame() waits until the required amount of time passes. You can easily change the amount of delay by changing the kDelay definition.

The snake game actually modifies the amount of delay on the fly, making the game go faster and faster each time the snake eats an apple. There are many uses for a routine like this. One is for simply delaying a specified amount of time. I implemented this in the Wait() routine, also at the bottom of the program. This is called whenever the snake eats an Apple, to give the player a moment to catch their breath.

Improving the Code

There are many things that could be done to improve the game, such as adding a pause key, implementing FCTN-9 to return you to the title screen, adding a score, and more level layouts. In addition, the level and lives numbers don't display properly above 9. That's because I "cheated" - I called HChar(row,col,'0'+number,1) to display the number. This could easily be fixed by using Bruce Harrison's excellent PutNum routine.

I'm sure you could also come up with other ways to tweak the game. So feel free to make any changes you like, and most of all, have fun with it! I hope you have enjoyed this series, and have found it useful in beginning your journey into c99 programming.