MPL Training
Lesson 11
Black Panther

Time to Write

-=-=-=-=-=-=-=-=-

I know we talked a bit about both the Write and WriteLn functions in an earlier lesson, but let’s dive a little deeper into what can be accomplished with these two functions.

```
Write(Text)
WriteLn(Text)
```

Now, if you remember, we can hard code some text to print to the screen for the user by using:

```
WriteLn(‘This is hard coded text’)
```

We can also write the contents of a string variable to the screen:

```
Var Text : String=’This is hard coded text’

WriteLn(Text)
```

If we have a variable of a different type, let’s say an integer, it will need to be typecast to a string before we can print it to the screen.

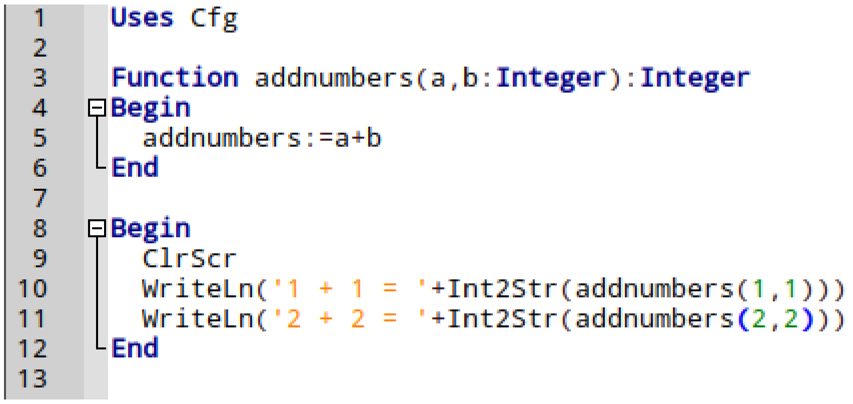
```
Var Count : Integer=2020

WriteLn(Int2Str(Count))
```

Now, if we want to combine different ‘text’ to print to the screen, we can combine them.

```
Var
  Text : String=’The year’
  Text1 : String=’really sucked’
  Year  : Byte=2020

WriteLn(Text+’ ‘+Int2Str(Year)+’ ‘+Text1+’!’)
```

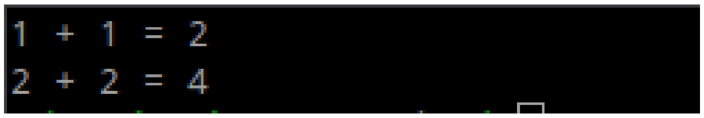
Output: The year 2020 really sucked!

It is also possible to call another function or procedure in your program inside of a Write/WriteLn function statement.



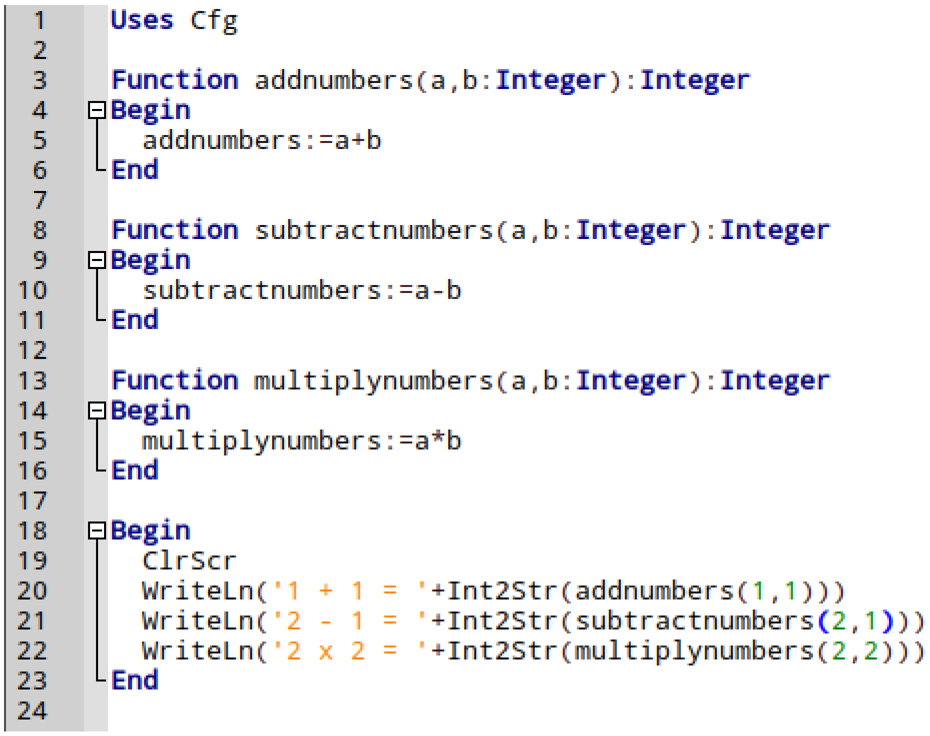
```
Uses Cfg

Function addnumbers(a,b:Integer):Integer
Begin
  addnumbers:=a+b
End

Begin
  ClrScr
  WriteLn('1 + 1 = '+Int2Str(addnumbers(1,1)))
  WriteLn('2 + 2 = '+Int2Str(addnumbers(2,2)))
End
```

When we run this MPL, we get the following output:



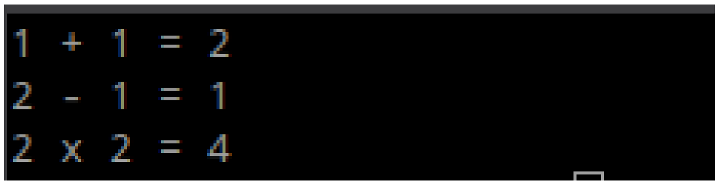
```
1 + 1 = 2
2 + 2 = 4
```

Or, we could get really complicated… ;)



```
Uses Cfg

Function addnumbers(a,b:Integer):Integer
Begin
  addnumbers:=a+b
End

Function subtractnumbers(a,b:Integer):Integer
Begin
  subtractnumbers:=a-b
End

Function multiplynumbers(a,b:Integer):Integer
Begin
  multiplynumbers:=a*b
End

Begin
  ClrScr
  WriteLn('1 + 1 = '+Int2Str(addnumbers(1,1)))
  WriteLn('2 - 1 = '+Int2Str(subtractnumbers(2,1)))
  WriteLn('2 x 2 = '+Int2Str(multiplynumbers(2,2)))
End
```

Output:



```
1 + 1 = 2
2 - 1 = 1
2 x 2 = 4
```

Now, let’s say you have a bunch of information to write to the screen. There are multiple ways of doing it while still keeping your code readable.

If you have code like:

```
  WriteLn('|15  '+PadRt(StripMCI(Call[count].Handle),25,' ')+''+PadRt(Call[count].SecName,30,' ')
+PadRt('|08'+StripMCI(Call[count].City),25,' ')+PadCT('|12'+Int2Str(TermSizeX)
+'x'+Int2Str(TermSizeY),24,' ')+'|04 '+PadRt(Int2Str(+Call[count].Node),2,' ')+'|06'+'
'+DateSTR(Call[count].LastCall,1)+'|07'+'     '+TimeStr(Call[count].LastCall,False)+'|CR');
```

Notice how word-wrap made a mess of it? Well, we could type it in like:

```
  WriteLn('|15  '+PadRt(StripMCI(Call[count].Handle),25,' ')+''+PadRt(Call[count].SecName,30,' ')
+PadRt('|08'+StripMCI(Call[count].City),25,' ')+PadCT('|12'+Int2Str(TermSizeX)
+'x'+Int2Str(TermSizeY),24,' ')+'|04 '+PadRt(Int2Str(+Call[count].Node),2,' ')+'|06'
+'    '+DateSTR(Call[count].LastCall,1)+'|07'+'     '+TimeStr(Call[count].LastCall,False)
+'|CR');
```

That would be perfectly valid, and the complier would have to problems with it.

A better way to write it, so it’s still readable, would be like:

```
Write('|15  '+PadRt(StripMCI(Call[count].Handle),25,' ')+''+PadRt(Call[count].SecName,30,' '))
Write(PadRt('|08'+StripMCI(Call[count].City),25,' ')+PadCT('|12'+Int2Str(TermSizeX))
Write('x'+Int2Str(TermSizeY),24,' ')+'|04 '+PadRt(Int2Str(+Call[count].Node),2,' ')+'|06')
WriteLn('    '+DateSTR(Call[count].LastCall,1)+'|07'+'     '+TimeStr(Call[count].LastCall,False)+'|CR')
```

Did you see what I did there? I split the line up at the ‘+’ signs. Used Write statements so it didn’t insert a carriage return at the end, until the last one. This way, you can still read all of the information without having to scroll left/right in your IDE.

Another thing you can do with using both Write and WriteLn, is use some of the Mystic functions within them. For example, if you wanted to make sure there were not MCI codes in the record you were trying to print to the screen. You could do something like I did in the above example:

```
WriteLn(PadRt(‘|08’+StripMCI(Call[count].City),25,’ ‘)
```

The StripMCI, will remove all of the <pipe> commands from the string. If someone were to enter their location as ‘|11bbb|01.|11castlerockbbs|01.|11com’, it could cause issues while trying to space the field on the screen. So, once you remove them and have ‘bbs.castlerockbbs.com’ it is just text and can be spaced properly. (Yes, there is someone who did that on CRBBS, and messed up my display of the last caller screen. I don’t want to mention Ktulu or anything…) ;)

I think this would be a good time to talk briefly about some other functions that are very useful when trying to format output to the scree. I refer to these as helper functions, as they are there to help you, and the write functions.

```
PadCt(S:String, N:Byte, C:Char): String
```

This function will center the text using the parameters you give it. The ‘S’ string, is the text you’d like to have centered on the screen. The ‘N’, is how wide you want the field. For example, if you want a string to be centered on an 80 column screen, you would pass 80. The ‘C’ is the character that you want the compiler to use to fill in the space in the beginning and end. Normally, you would just use a space, which would be enclosed within single quotes. ‘ ‘

```
Var Str : String

Str := PadCt(‘Hello World’, 80, ‘ ‘)
WriteLn(Str)
```

```
PadLt(S:String, N:Byte, C:Char): String
PadRt(S:String, N:Byte, C:Char): String
```

Both of these are set up the same way as the PadCt function. The PadLt, will pad the left side of the string, and the PadRt will pad the right side of the screen.

```
POS(Sub: String, S: String): Byte
```

This function is very useful if you are trying to find a substring, within a string. Let’s say you are trying to do a search function within an MPL, that will search for a BBS name. If you do a standard ‘If userinput = BBSName Then’ will need to be an EXACT match in order to be true. However, if a user is searching for Castle Rock BBS, and they just type in ‘Castle’, this won’t find the match.

If we use this function:

```
Var
  UserInput : String

POS(UserInput, BBSList)
```

The POS function will return the location within the string where the match was found. If there is a match, this return will ALWAYS be greater than 0. So, we could do a check, such as:

```
If POS(UserInput, BBSList) > 0 Then
```

This will only be true if a match was found. The user could try searching for ‘cas’, and this function would return 1, as the match occurred in the first position of the string. If they searched for ‘rock’, the POS function would return 7. Either way, the return Byte is greater than 0, so we can process the match.

```
Replace(Str1, Str2, Str3 : String): String
```

If you have text within a string that you want to change a specific word, it can be done with the Replace function. The first parameter, Str1, is the string you would like to search. Str2 is the substring you would like to replace, and Str3 is what you would like to replace it with.

```
Var
  Str  : String='Hello Hello Hello'

Str:= Replace(Str, 'Hello', 'World')
```

At this point, the string would contain ‘World World World’.

StrComma(LI: LongInt): String

While this function isn’t used a lot, it is very handy. If you have a large number, that you would like to make more human-readable, you would enter the thousand separator, in the US it would be a comma. A good example is in the game I’ve been working on. In MPL, I could use this function whenever I showed the user the amount of gold they had on hand.

```
StrComma(User.Gold):String
```

This would change the number:
1537682548362
to a more readable:
1,537,682,548,362

Just as a side-note, if you want to do this is C, it takes three functions and almost 100 lines of code, just to insert commas into a number like this…