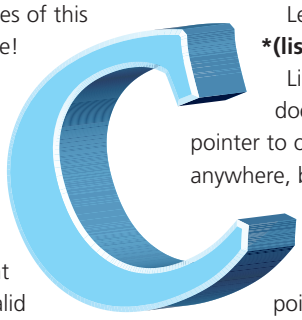
C: Part 4
LANGUAGE OF THE 'C'

Of all the elements in 'C' programming, pointers are known to cause the most problems. They needn't. In this article, I'll show you why.

Electric Counterpoint

So what can pointers do, and why are they so useful? Well, pointers (as the name suggests) point to a location in memory that holds information we are interested in. In keeping with the themes of this series, that information is a temperature! Used appropriately they can help keep the code clean and save processing time, since we pass pointers to data into functions, instead of the data itself.



It might be obvious, but I shall state it for the record - pointers have to point to something. This must be our own valid data (in our own data segment, see last months Memory access boxout) or we will core dump. If the pointer has no valid data to point to it should be set to zero. This is termed a NULL pointer.

Pointers, like structures, arrays before them, and variables before them, can not be changed once created! An integer pointer will always point to integers, but it can point to a number of different integers during its lifetime. Although there is nothing to stop you pointing it at floating point numbers, you can only read that data from memory as integers - which produces strange looking numbers!

Let's start with some simple examples: ***(listing 1)**

Line 5 declares a pointer to an integer. It doesn't declare the integer itself, just a pointer to one. It could be referencing anything, from anywhere, because (like all local variables) 'C' does not automatically initialise them to sensible values. This is called a 'dangling' pointer. If we try to use pNum1 without pointing it to something we are very likely to core dump (in fact, you would be incredibly lucky not to core dump!). Line 10 (explained later) shows how we point it to something.

Line 6 contains two pointer declarations. Note that each variable needs the '*' prefix, if it is to become a pointer. Omitting this is a common newbie mistake, although the compiler is good enough to point this out to us if we try to use a normal int as an int pointer! Wherever a variable type (like int) is valid - a return type or function parameter - a pointer to a type is also valid.

We see the initialisation of a NULL pointer in line 7. This is the safest way to declare a pointer as we can check its validity with:

```
if (pAListOfLongs)
   {
   /* this pointer is valid */
   }
```

However, should we forget to check the NULL pointer, we can guarantee a core dump when its contents are studied!

Line 10 is more interesting. This points 'pNum1' to somewhere valid - in this case, the address of the integer variable declared at line 8, iValue. The ampersand can be used in front of any variable name type (regardless of its type) to produce the address in

Listing 1

```
1  #include <stdio.h>
2
3  int main(int argc, char *argv[])
4  {
5  int *pMyList;
6  int *pNum1, *pNum2;
7  long *pAListOfLongs = NULL;
8  int iValue = 4;
9
10 pNum1 = &iValue;
11 printf("Value at the pNum1 ptr is %d\n", *pNum1);
12 *pNum1 = 3;
13 printf("Value at the pNum1 ptr is %d\n", *pNum1);
14 printf("iValue is now %d\n", iValue);
15 return 0;
16 }
```

memory of that variable. And because the variable has the number four, our pointer also points to the number four.

To read the information from the pointer we have to dereference it. This is done by prefixing the pointer's name with an '*', as in lines 11 & 13. We can also use this syntax to write information back into that memory location, as with line 12:

```
12 *pNum1 = 3;
```

You will notice that because pNum1 and iValue both reference the same memory location (and not just the same value) changing the data of either one, affects the other; which is why line 14 will report iValue to being 3 despite the fact we never change it explicitly.

Anticoh Arrow

In addition to dereferencing basic variable types (like int, long and float), we can do exactly the same thing with structures. You probably wouldn't be surprised by the code?

```
float  Convert(float fValue, struct sConversion
*pConv)
{
   return (fValue * *pConv.fMultiplier) +
*pConv.fAddition;
}
```

We dereference 'pConv' with '*', making the type of '*pConv' a structure; and since structures use '.' to retrieve individual elements, we write '*pConv.fMultiplier'. It's not really that strange. (You must remember to put a space between the two '*'s, otherwise 'C' will try and understand the symbol '**' - which it can't - and complain heartily!) At this point, the bright kid at the back of the class raises his hand, "Isn't there a neater way, sir?" Well, yes. There is.

Because this type of operation is very common, we have a special symbol that combines the '*' and the '.' into one. For the lazy typists amongst you, I'm afraid it still has two characters, but it does look like a spaceship - making it infinitely cooler! That symbol is '->', and is called the 'pointer to' operator. It can be used directly in place of '*' and '.', turning the above function into:

```
float  Convert(float fValue, struct sConversion
*pConv)
{
   return (fValue * pConv->fMultiplier) +
pConv->fAddition;
}
```

As well as passing data in, it is possible to pass a structure out of a function using a pointer. It's much better than returning all the data on the stack (which requires copying memory) and allows the function to write its results directly into the structure.

Your Latest Trick

Pointers can be kept simple by making sure the left and right hand sides of each expression have matching types. So, if the variable on the left hand side is a 'float *', make sure the right hand side is one too (either another 'float *', or the address of a float variable). Be careful though, once you have a

Incrementation

When an expression like, *pTempList++ is evaluated there is the question of 'when exactly does the pTempList variable get incremented'?

The actual answer varies for every compiler you might use! All the 'C' standard expects is that the incrementation will occur at some point before the next sequence point (the ';'). But it does not specify when. Usually, this isn't a problem. But if your code changes the same variable twice before the next sequence point - it is!

```
iResult = iValue++ * iValue++;
/* Very bad code ! */
```

Only the compiler know if this is treated as:

```
Get iValue1
Increment iValue
Get iValue2
Increment iValue
Add them
Multiply
Assign to iResult
(sequence point here)
```

or

```
Get iValue1
Get iValue2
Add them
Multiply
Increment iValue
Increment iValue
Assign to iResult
(sequence point here)
```

Only the compiler knows what it's going to do - and you are not the compiler, so you shouldn't be writing the code like that!

The same ambiguity is true for the expression:

```
iValue = iValue++;
/* also, very bad code! */
```

' iValue++' evaluates to whatever value ' iValue' is (post-increment, remember), this is remembered (usually in a register, or on the stack), and used later when it comes to performing the assignment ' iValue =?'. Using the above notation:

```
Get iValueRHS
Store iValue in iStackTemp
Increment iValue
Assign iStackTemp to iValueLHS
(sequence point here)
```

or

```
Get iValueRHS
Store iValue in iStackTemp
Assign iStackTemp to iValueLHS
Increment iValue
 (sequence point here)
```

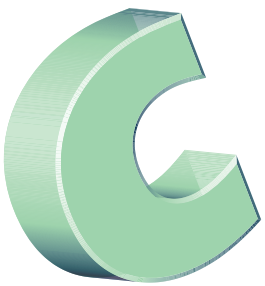
The first version leaves iValue unchanged, whilst the second version would increment it. Naturally, this is very ungood, and is not encouraged!

```
(*pTempList)++
```

This creation will cause the compiler to dereference pTempList, and increment the value - at that memory location. This saves copying data from memory, incrementing, and copying it back.

Listing 2

```
1  #include <stdio.h>
2  #include <string.h>
3
4  int main(int argc, char *argv[])
5  {
6  char szFullname[] = "Sandra Bullock";
7  char * pSurname;
8
9      pSurname = strchr(szFullname, ' ');
10     if (pSurname)    /* person may not have a surname! */
11        {
12     pSurname++; /* skip the space */
13     strcpy(pSurname, "Goodwin");
14     }
15 printf("Full name is now %s\n", szFullname);
16
17 return 0;
18 }
```



pointer, don't take the address of it!

```
pNum1 = &iValue;     /* Good! */
pNum2 = pNum1;       /* Good! */
pNum2 = &pNum1;      /* Bad! pNum1 is ➥
                     already a pointer! */
pNum2 = pAListOfLongs;  /* Bad! We have a int➥
                        * on the left, and ➥
                        long * on the right */
```

Angels vs Animals (or Pointers vs Arrays)

Before we move on, there's one more assignment we should cover. Arrays. Like other variables, they exist in memory, so we should be able to assign a pointer to point to them. The question, is how? The answer, is simple!

```
float fTemperateEachHour[24];
float *pTempList;
pTempList = &fTemperateEachHour[0];
pTempList = fTemperateEachHour;
```

Both examples produce the same result, but the second uses a synonym (or syntactic sugar!) within the language. The array name is (by design) the location in memory where the array data was created; making it similar to a pointer. And, because the array name is treated as a single variable, it can be passed into functions like we saw (but didn't explain) last month with strings. However, the differences between an array and pointer are many (see the comp.lang.c FAQ), but primarily, a pointer can point to any chunk of memory (valid or not) but does not allocate that memory. The array, on the other hand, will allocate some memory, but can only point to it - nowhere else.

Working In A Coalmine

Now we have a pointer, we need to do something with it. We know how to handle data in the memory it points to, and we know how to initialise it, but so far we haven't worked out how to modify the pointer. Since a pointer is like any other variable we can re-assign it, or use one of the arithmetic operators (like '+' or '—'). This is both legal and useful. However, multiplication of pointers doesn't make any sense (nor does division). So stick to the sensible ones; '+', '-', '++', '-', '+=' and '-='.

Assuming we have an array of hourly temperatures in 'fTemperateEachHour', and 'pTempList' points to the first one, we can reference the second temperature easily because the array is always sequential. There are several ways of doing this:

```
t = pTempList[1]
```

We act like it's an array. Valid, but is not good practise because it implies an array - which it isn't.

```
t = *(pTempList+1)
```

Directly equivalent to the method above. Here we treat the pointer like a variable, adding '1' to it. This will move on one element. (If we point to floats, one element is the size of one floating point number, so we never need to know the size of each element in bytes). The data is then dereferenced from memory with the usual '*' notation.

```
++pTempList;
t = *pTempList;
```

After then first instruction pTempList points to the second array element, which we can dereference with a single '*'.

```
t = *(++pTempList);
```

Exactly the same as above. We've just joined both lines into one, using brackets to ensure the correct evaluation order.

The most usual way to read a block of data using pointers is with a variation of the last example.

```
for(i=0;i<24;i++)
    printf("Temp at index %d is %f\n",➥
    i, *pTempList++);
```

(I hope we are all comfortable dropping the braces when there's only one statement, since this is more usual)

So what happens in '*pTempList++'? Well, it is the same as *(pTempList++), as opposed to (*pTempList)++ (see BOXOUT: (*pTempList)++). The post-increment part (pTempList++) evaluates to the

Most of these examples use floating point numbers because we'd like decimal places in our temperatures. Unfortunately, floating point numbers are inaccurate since it is not possible to represent every value exactly with them. This means you will see some strange numbers in places, for example 9/5 is 1.8. But 32+1.8 is not 33.8, but 33.799999 because the processor can not represent 33.8. For this reason it is (very) rare to compare floating point numbers directly with '==', rather it is better to say 'if two numbers are within 0.0001 of each other, they are the same'. This small value is called the epsilon.

value of 'pTempList' before the variable is incremented. This value is then dereferenced by the '*' operator. It occurs in this order because of precedence. But that is a more complex topic, so will be dealt with in later issues. Also see BOXOUT: Incrementation.

After completion, pTempList has been incremented beyond the end of valid data. To run the loop again, we must rewind the pointer with:

```
pTempList = &fTemperateEachHour[0];
```

You have now learnt enough to re-write all the string functions we saw last month, and understand all the others! The strcpy function, for example, might be:

```
void MyStringCopy(char *pStringDest, ➥
char *pStringSrc)
{
   while(*pStringSrc)  /* until we reach ➥
   the NULL terminator */
      *pStringDest++ = *pStringSrc++;
   *pStringDest = '\0'; /* add a new NULL ➥
   terminator */
}
```

Sonata Number 2 for Flute and Strings in C# Minor

In part three, I briefly remarked about writing the BASIC instruction LEFT$ with one simple line of 'C'. It would have been possible to produce versions of MID$ and RIGHT$ fairly easily. It would have also been a waste of time! Why? Strings are rarely manipulated in that manner because we have more powerful methods at our disposal, notably pointers. Your gateway to this fountain of proverbial knowledge is:

1. A string is a pointer to NULL terminated data.
2. A pointer to a character within a string is still a string, because of 1.

Well, that's it explained in zen-style language. What about in the 'C' language?!

```
char szFullname[] = "Sandra Bullock";
char *pSurname = &szFullname[7];
```

Our new string (pSurname) can be treated like any another (since if szFullname ends in NULL, so must pSurname), meaning we can use all the normal string functions, like strcpy, on it.

```
strcpy(pSurname, "Goodwin");       /* for when
Sandra decides to marry me! */
printf(szFullname);
```

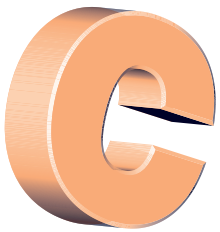
We must be very careful when building strings in this manner. If our original string doesn't have enough space for the new surname, we'll overrun the szFullname buffer, causing the overrun problems mentioned last month. However, it doesn't matter if we underrun. Most people will declaring strings with 128 or 256 characters in them, knowing that any string manipulation will not be so large as to overrun. It is very dangerous

Listing 3

```
1 #include <stdio.h>
2
3 void ConvertToFahrenheit(int iNumElements, float *pTemperatures)
4 {
5 float fFahrenheit, fCelsius;
6 int i;
7
8  for(i=0;i<iNumElements;i++)
9     {
10    fCelsius = *pTemperatures;
11    fFahrenheit = fCelsius*(9.0f/5.0f) + 32.0f;
12    *pTemperatures++ = fFahrenheit;
13    }
14}
15
16 int main(int argc, char *argv[])
17 {
18 float fTempList[] = {
19 20, 18, 17, 16, 15, 17, 18, 19, 20, 21, 22, 23,
20 24, 25, 26, 27, 28, 29, 27, 26, 25, 24, 23, 22,
21 };
22 int iNumHours = sizeof(fTempList)/sizeof(fTempList[0]);
23 int i;
24
25 ConvertToFahrenheit(iNumHours, &fTempList[0]);
26 for(i=0;i<iNumHours;i++)
27    printf("Element %d is %f\n", i, fTempList[i]);
28
29 return 0;
30 }
```

Listing 4

```
1 #include <stdio.h>
2
3 void ToFahrenheit(float fTemp)
4 {
5  printf("%f c => %f f\n", fTemp, fTemp*(9.0f/5.0f)+32.0f);
6 }
7
8 void ToKelvin(float fTemp)
9 {
10 printf("%f c => %f K\n", fTemp, fTemp+273.15f);
11 }
12
13 void ConvertTemperatureRange(int iWhichFn)
14 {
15 void (*pFunction)(float);
16 float fTemp;
17
18    switch(iWhichFn )
19    {
20    case 0:
21       pFunction = ToFahrenheit;
22       break;
23    case 1:
24       pFunction = ToKelvin;
25       break;
26    }
27
28    for(fTemp=-100;fTemp<=100;fTemp+=10)
29        (*pFunction)(fTemp);
30 }
31
32 int main(int argc, char *argv[])
33 {
34    ConvertTemperatureRange(0);
35    ConvertTemperatureRange(1);
36 return 0;
37 }
```



assumption, and only acceptable in casual, non-critical, code.

One of the functions mentioned last month is strchr. This function searches a string for a particular character (a single, literal, character, that is) and returns a pointer to the first occurrence. If no character could be found, it returns a NULL pointer. We could use this function to generalise our above code, thus see listing 2.

Pass The Dutchie

Pointers are often used as parameters to functions. This lets us modularise our data and algorithms, since we can say 'do your processing, with our data'. And, as mentioned before, by supplying pointers to data, we don't need to waste time making a special copy of the data see listing 3.

We have seen all of this code before, in some form or another. We have seen dereferenced pointers (lines 10 & 12), the incrementing pointer (12), the automatic array size (18-21), and the size of operator (22).

You should be able to say why we don't increment the pointer in line 10. You should also be able to say how we could enhance this example so we don't loose the original 'temperature in Celsius' data.

Finally a word of warning. If you intend to pass a pointer back from a function (and there are several reasons why you'd want to) then make sure the data it points to will still exist when the function exits. Local variables will not! They are created on the stack, and will leave your calling routine with a dangling pointer, looking at an invalid area of memory.

Take A Bow

When you call a function, the processors program counter is set to the memory location of the function. Simple. But why should the compiler be limited to such operations? What if we (as the programmer) knew the address in memory of a function - what stops us from using it to call that function? In 'C' - nothing!

We can declare such a variable by taking the function prototype,

```
void PrintMessage(void);
```

and replacing the function name like so:

```
void (*pFunction)(void);
```

The declaration enforces the only limitation with pointers to functions. That is, the variable can only point to functions with exactly the same prototype. To ease readability, this could also be typedefed as in listing 4.

When a function name is used as part of an expression (as in lines 21 and 24), a function pointer is generated, and can be stored in a variable (as it is here), or passed to another function for processing. A call to that function is made by dereferencing it, (line 29). This is how a number of fractal generators are written: there is a common 'for all pixels in image' loop which calls (with a pointer to function) the program code for a particular fractal. This prevents duplicating the 'for all pixels in image' code, and allows new modules to be added easily to the program. Providing a new fractal type requires, perhaps, four lines of code!

Next month?IO: how to handle files, the keyboard and the screen.