Techie's Guide to C Part 16

This month we'll look at a complex function for getting input. More than simply a prompt and a prayer, this get string function asks for characters in style.

Steve Rimmer

There are a number of functions which are common to many of the sorts of programs which get written in C. Trolling for DOS files, something we considered a few months ago, is certainly one of them. The simple task of asking a user for in input string is another.

If you've perused the documentation for your C compiler, you'll probably know that the standard C library provides a function for getting a string, either from the console... in this case the keyboard... or from a file. This function is, predictably, a bit crude. There are a number of ways to improve on it, of which the one we'll look at this month is probably the most elegant.

In most cases, the user interface of a program occupies well over three quarters of the programming effort involved and of the resultant code. Getting string input elegantly, then, is something worth doing.

Stringing it Out

The simple get string function provided with C is called *gets*. Its operation is pretty uninvolved... you pass it a pointer to a string and it waits for input. Everything typed by the user of your program will wind up in the string until the first time a carriage return comes down the pipe.

If you wanted to, you could easily write a function to do this.

```
gets(s)
char*s;
{
while((*p++=putchar(getch())) !=
13);
    *p=0;
    }
```

Note that the function *putchar*... another denizen of the C library... prints the character passed to it to the screen and then returns the character, allowing us to devise this rather elegant structure.

There are a number of problems with this function. To begin with, it allows your users to enter anything they like into your string, whether or not it's what you had in mind. They can enter more data than you've allowed for in allocating the string passed to *gets*, in which case they'll probably manage to trash the stack of your program and do something nasty. Finally... and perhaps more important from the point of view of being elegant and civilized... this function provides no editing facilities.

If you've used DOS for a while and have installed one of the extended command line editors... DOSEDIT or something similar... you'll probably appreciate how primitive a simple function like this can be in real life. If someone types a long string into it and discovers that there's a typo at the beginning of the string, there's no way to back up and fix the problem. We can improve on this slightly by adding a backspace facility to it.

```
gets(s)
char*s;
{
int c,i=0;

do {
c=getch();
if(c==8 && i 0) {
--i;
putchar(8);
putchar(32);
putchar(8);
} else p[i++]=putchar(c);
} while(c != 13);
p[i]=0;
}
```

There are a few things you'll want to know in order to make sense of this. The ASCII value for the backspace key is eight, and if you print character eight to the screen the cursor will backspace by one position. If you print eight, followed by thirty-two... a space... followed by eight again, the cursor will back up, erase the most recent character and then back up again.

This function does allow you to backspace and fix your errors, but it means retyping a whole line if you discover that there's an error at the beginning.

The Better Gets

A *gets* function which allowed for editing of inputted lines, something like what BASIC and DOSEDIT implement, would be a great improvement over this. Unfortunately, such a function is a bit of a pig to write. Being able to insert text into the middle of a string, manage extended editing keys and so on are all a bit demanding of your code.

You might want to consider the program accompanying this article to get a feel for the magnitude of the task. This is, I think, the ultimate get string function. It allows for full line editing... having typed something you can cursor back through the line to insert, delete or change characters. It also allows your program specify a default string to be edited, trashed or accepted by your users.

In order to avoid confusion with the stock *gets* function, we'll call this one *get_string*. It's called as

```
get_string(p,dflt,n);
```

where *p* is the buffer where your gotten string will go, *dflt* is the string you'd like to appear when the function is first called and *n* is the maximum number of characters the gotten string can contain.

In many cases you won't want a default string, in which case you can just pass an empty string for this argument, that is, two double quote marks with nothing between them.

The basic structure of this rather immense function is essentially the same as that of the simple *gets* function, although it checks for a lot more characters. One of the important things about this function is that it uses *GetKey*, rather than *getch*, to get keyboard characters. This is done because *getch* can only return stock ASCII characters, while many of the editing functions, such as the cursor mover keys, are handled by some of the extended keys of the PC keyboard. This is handled a bit oddly... if you call *getch* and it returns zero, there's the scan code of an extended key waiting in the keyboard buffer, in which case you should call *getch* a second time to retrieve it. This is essentially what *GetKey* does.

There are a few other ancillary functions involved in this code, such as the ones which change the size of the cursor. These use INT 10H BIOS calls, something we haven't really discussed as yet. You can look up what they do in a PC hardware manual if you like, you you can just trust 'em for the time being.

One of the interesting things about this enormous string getting function is that it does all its editing without actually positioning the cursor. It moves around solely by using nondestructive backspaces to move left and overprinting the existing string data to move right. This means that it doesn't care... or even actually know... where it's located on the screen. There's a considerable amount of juggling involved in making it do its stuff.

A thorough explanation of each of the cases involved in this function is beyond the scope of this article, and is probably unnecessary. If you've been following this series for the past few months, you can probably ''read'' C well enough to be able to walk your way through this code pretty easily. There's nothing in the least bit mysterious about how it works or what it does.

The complete string get function follows.

```
#define BS 0x08
#define CR 0x0d
#define ESC 0x1b
#define BLNK '_'

GetKey()
{
 int c;

 c=getch();
 if(!(c & 0x00ff)) c=getch() << 8;
 return(c);
}

hidecursor()
{
 union REGS r;

 r.x.ax=0x0f00;
 int86(0x10,&r,&r);

 r.x.ax=0x0200;
 r.x.dx=0x1a00;
 int86(0x10,&r,&r);
}

getst(size,deflt,buffer) /* get a string
*/
 int size;
 char *deflt,*buffer;
 {
  char *p;
  int i,l,c,cursor=0,insert=0;

  *buffer=0;
  if((p=malloc(size+1)) != NULL) {
   small_cursor();
   for(c=0;c<size;++c) putch(BLNK);
   for(c=0;c<size;++c) putch(BS);

   do {
    l=strlen(buffer);
    if(*(deflt)==0) c=GetKey();
    else c=*deflt++;
    switch(c) {
     case DEL:
      if(cursor<l) {
       memcpy(p,buffer,cursor);
          memcpy(p+cursor,buffer+cur-
sor+1,(l-cursor)+1);
       strcpy(buffer,p);
            i=printf("%s%c",buffer+cur-
sor,BLNK);
```

continued on page 34

continued from page 9

```
   while(i) {
    putch(BS);
    --i;
   }
  }
  break;
 case INS:
  if(insert) {
   insert=0;
   small_cursor();
  }
  else {
   insert=1;
   big_cursor();
  }
  break;
 case HOME:
  while(cursor) {
   putch(BS);
   --cursor;
  }
  break;
 case END:
  while(cursor<l) {
   putch(*(buffer+cursor));
   ++cursor;
  }
  break;
 case CURSOR_RIGHT:
  if(cursor<l) {
   putch(*(buffer+cursor));
   ++cursor;
  }
  break;
 case CURSOR_LEFT:
  if(cursor) {
   putch(BS);
   --cursor;
  }
  break;
 case BS:
  if(cursor==l) {
   if(l) {
    --l;
    --cursor;
    *(buffer+l)=0;
    putch(BS);
    putch(BLNK);
    putch(BS);
   }
  }
  else if(cursor<l && cursor>0) {
   --cursor;
   memcpy(p,buffer,cursor);
      memcpy(p+cursor,buffer+cur-
sor+1,(l-cursor)+1);
   strcpy(buffer,p);
      i=printf("%c%s%c",BS,buff-
er+cursor,BLNK)-1;
   while(i) {
    putch(BS);
    --i;
    }
   }
   break;
  case ESC:
   while(cursor<l) {
    putch(*(buffer+cursor));
    ++cursor;
   }
   while(l--) {
    putch(BS);
    putch(BLNK);
    putch(BS);
   }
   cursor=0;
   *buffer=0;
   break;
  default:
   if(c>=0x20 && c<=0x7f) {
    if(cursor==l && l<size) {
     *(buffer+l++)=c;
     *(buffer+l)=0;
     putch(c);
     ++cursor;
    }
    else if(cursor<l) {
     if(!insert) {
      *(buffer+cursor++)=c;
      putch(c);
     }
     else if(l<size) {
      memcpy(p,buffer,cursor);
      *(p+cursor)=c;
      memcpy(p+cursor+1,
       buffer+cursor,(l-cursor)+1);
      strcpy(buffer,p);
      i=printf("%s",buffer+cursor)-1;
      while(i--) putch(BS);
      ++cursor;
     }
    }
   }
   break;
  }
 } while(c!=CR);
 free(p);
 small_cursor();
 return(strlen(buffer));
 } else return(-1);
}

big_cursor()   /* make the cursor big
*/
{
 union REGS r;

 r.h.ah=15;
 int86(0x10,&r,&r);

 r.h.ah=1;
 r.h.cl=7;
 r.h.ch=3;
 int86(0x10,&r,&r);
}

small_cursor()   /* make the cursor
small */
{
 union REGS r;

 r.h.ah=15;
 int86(0x10,&r,&r);
 r.h.ah=1;
 r.h.cl=6;
 r.h.ch=5;
 int86(0x10,&r,&r);
}
```