FIRMCODES
FIRMWARE DEVELOPERS

FIRMWARE DEVELOPER

Embedded Systems

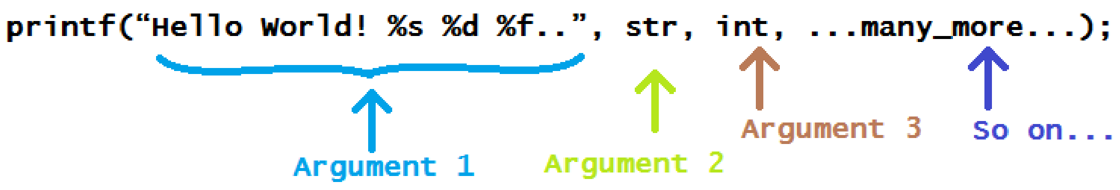
Write your own printf() function in c

Write your own printf() function in c

I was wondering where I can find the C code that's used so that when I write printf("Hello World!") in my C program to know that it has to print that string to standard output(i.e. monitor screen). I looked in <stdio.h>, but there I could only find its prototype int printf(const char *format, …), but not how it looks like internally.

Then i start googling for that but i hardly find a solution for which i am looking for. So after long time spend on net, i found this nice formal C Program.

Before implementation of printf( ) function we have to deal with unusual problem which is variable arguments. As we know that printf can take many arguments besides string, like


printf("Hello World! %s %d %f..", str, int, ...many_more...);
Argument 1
Argument 2
Argument 3
So on...

So we have to use a standard library called **stdarg.h** to handle this variable argument problem. In this implementation context, we don't need learn whole **stdarg.h** library because we use some macro functions of these library which is understandable directly by our C program.

Code to implement printf( )

```c
#include<stdio.h>
#include<stdarg.h>

void Myprintf(char *,...);      //Our printf function
char* convert(unsigned int, int);    //Convert integer number into octal, hex, etc.


int main()
{
 Myprintf(" WWW.FIRMCODES.COM \n %d", 9);

 return 0;
}


void Myprintf(char* format,...)
{
 char *traverse;
 unsigned int i;
 char *s;

 //Module 1: Initializing Myprintf's arguments
 va_list arg;
 va_start(arg, format);

 for(traverse = format; *traverse != '\0'; traverse++)
 {
  while( *traverse != '%' )
  {
   putchar(*traverse);
   traverse++;
  }

  traverse++;

  //Module 2: Fetching and executing arguments
  switch(*traverse)
  {
   case 'c' : i = va_arg(arg,int);  //Fetch char argument
      putchar(i);
      break;

   case 'd' : i = va_arg(arg,int);   //Fetch Decimal/Integer argument
      if(i<0)
      {
       i = -i;
       putchar('-');
      }
      puts(convert(i,10));
      break;

   case 'o': i = va_arg(arg,unsigned int); //Fetch Octal representation
      puts(convert(i,8));
      break;

   case 's': s = va_arg(arg,char *);   //Fetch string
      puts(s);
      break;

   case 'x': i = va_arg(arg,unsigned int); //Fetch Hexadecimal representation
      puts(convert(i,16));
      break;
  }
 }

 //Module 3: Closing argument list to necessary clean-up
 va_end(arg);
}

char *convert(unsigned int num, int base)
{
 static char Representation[]= "0123456789ABCDEF";
 static char buffer[50];
 char *ptr;

 ptr = &buffer[49];
 *ptr = '\0';

 do
 {
  *--ptr = Representation[num%base];
  num /= base;
 }while(num != 0);

 return(ptr);
}
```

Pseudocode

Module 1: Initializing Myprintf's arguments

In this section, we initialize the arguments of Myprintf( ) function by using standard argument library.

```
va_list arg;
```

This line declares a variable, **arg**, which we use to manipulating the argument list containing variable arguments of Myprintf( ). The data type of the variable is va_list, a special type defined by <stdarg.h>.

```
va_start(arg, format);
```

This line initializes **arg variable with** function's last fixed argument i.e. **format**. va_start() uses this to figure out where the variable arguments begin.

Module 2: Fetching and executing arguments

```
i = va_arg(arg,int);
```

va_arg() fetches the next argument from the argument list. The second parameter to va_arg() is the **data type** of the argument we expect.

Note: va_arg( ) function will never receive arguments of type char, short int, or float. va_arg( ) function only accept arguments of type char *, unsigned int, int or double.

Module 3: Closing argument list to necessary clean-up

```
 va_end(arg);
```

Finally, when we're finished processing the all arguments, we call **va_end()**, which performs any necessary cleanup.

Original standard printf( )

Here's the GNU version of printf… you can see it passing in stdout to vfprintf:

```
__printf (const char *format, ...)
{
   va_list arg;
   int done;

   va_start (arg, format);
   done = vfprintf (stdout, format, arg);
   va_end (arg);

   return done;
}
```

Here's a link to vfprintf… all the formatting 'magic' happens here.

This is the same thing as we use varargs(stdarg.h library) to get parameters in a variable argument list. Other than that, they're just traditional C.

Limitation of standard variable argument functions is described in another post named "How can I write a function that takes a variable number of arguments ? " which you can see below in suggested reading.

Source

1. http://www.eskimo.com/
2. http://sourceware.org/
3. http://stackoverflow.com/
4. https://www.google.com/

Suggested Reading

1. C/C++ program to shutdown or turn off computer
2. Function Pointers in C/C++
3. How can I write a function that takes a variable number of arguments?

Donate For Future Update

Donate

Visitors See more

IN 78,235	DE 2,714	BD 1,826
US 23,683	GB 2,437	KR 1,754
PK 3,271	FR 2,167	TR 1,629
CA 2,774	EG 2,144	RU 1,450

Pageviews: 247,606

FLAG counter