Functions

Functions are the building blocks of C programs.

A program that is more than 25 lines long should be comprised of separate functions that are not longer than 25 lines! Advantages of the use of functions include:

- Facilitates of top down design
- Facilitates bottom up implementation and testing
- Increased program readability ... proper function naming can make a program nearly self-documenting.
- Reduction of nesting
- Reuse of code

From the Linux kernel Coding Style Guide

Chapter 5: Functions

Functions should be short and sweet, and do just one thing. *They should fit on one or two screenfuls of text (the ISO/ANSI screen size is 80x24, as we all know), and do one thing and do that well.*

The maximum length of a function is inversely proportional to the complexity and indentation level of that function. So, if you have a conceptually simple function that is just one long (but simple) case-statement, where you have to do lots of small things for a lot of different cases, it's OK to have a longer function.

However, if you have a complex function, and you suspect that a less-than-gifted first-year high-school student might not even understand what the function is all about, you should adhere to the maximum limits all the more closely. Use helper functions with descriptive names (you can ask the compiler to in-line them if you think it's performance-critical, and it will probably do a better job of it than you would have done).

Another measure of the function is the number of local variables. They shouldn't exceed 5-10, or you're doing something wrong. Re-think the function, and split it into smaller pieces. A human brain can generally easily keep track of about 7 different things, anything more and it gets confused. You know you're brilliant, but maybe you'd like to understand what you did 2 weeks from now.

Function definitions

A C function is comprised of 4 components

1 - the type of value returned by the function
2 - the name of the function
3 - parenthesized declaration of function parameters
4 - at least one basic block containing local variable declarations and executable code

```
int main(
int argc,     /* Number of cmd line parms */
char *argv[]) /* Array of ptrs to cmd line parms */
{
--- basic block ---
}
```

How functions facilitate top down design:

```
#include <stdio.h>
char this_word[100];

int main()
{
   int  howmany;
   int  found_hello = 0;

   howmany = fscanf(stdin, "%s", &this_word[0]);

   while (howmany == 1)
   {
       if (isHello(&this_word[0]))
          found_hello = 1;

       howmany = fscanf(stdin, "%s", &this_word[0]);
   }

   if (found_hello)
      fprintf(stdout, "yes\n");
   else
      fprintf(stdout, "no\n");
   return(0);
}
```

We can defer thinking about the grubby details of a hard problem by simply reducing it to a function.

Use of “stubs”

Stubs are dummy implementations of a function. Their use can help us accomplish two important objectives.

- verifying that the main() or other upper level function is working as intended
- verifying that parameters are being passed correctly to the function.

```
#include <stdio.h>
char this_word[100];

int isHello(
char myword[])
{
   fprintf(stderr, "%s\n", myword);
   return(0);
}

int main()
{
   int  howmany;
   int  found_hello = 0;

   howmany = fscanf(stdin, "%s", &this_word[0]);
   while (howmany == 1)
   {
       if (isHello(&this_word[0]))
          found_hello = 1;

       howmany = fscanf(stdin, "%s", &this_word[0]);
   }

   if (found_hello)
      fprintf(stdout, "yes\n");
   else
      fprintf(stdout, "no\n");
   return(0);
}
```

```
acad/cs101/examples/notes ==> p21
asdd hello
asdd
hello
no
```

The value “no” is to be expected because the stub always returns 0. The stub could modified to always return 1.

Bottom up implementation and modular testing.

When building complex programs *it is simply not possible to defer testing until the who program is complete*. It is unfortunately the case that only *the very least capable students even consider trying* (and they *never* succeed.)

The proper approach is to implement simple *"main()"* programs to *unit test* newly written functions.

```
#include <stdio.h>
char this_word[100];

int isHello(
char myword[])
{
   int found = 0;
   fprintf(stderr, "Word to be tested is: %s\n", myword);
   if ((myword[0] == 'h') &&
       (myword[1] == 'e') &&
       (myword[2] == 'l') &&
       (myword[3] == 'l') &&
       (myword[4] == 'o') &&
       (myword[5] ==  0 ))
       found = 1;
}

int main()
{
    fprintf(stderr, "%d \n", isHello("abcd"));
    fprintf(stderr, "%d \n", isHello("hell"));
    fprintf(stderr, "%d \n", isHello("hello"));
    fprintf(stderr, "%d \n", isHello("hello\n"));
}
```

Since *isHello()* returns an *int* it is perfectly legal and in fact common to embed a call to it inside *fprintf*.

Discovering errors in the unit test

```
acad/cs101/examples/notes ==> p22
Word to be tested is: abcd
134514017
Word to be tested is: hell
134513920
Word to be tested is: hello
134513920
Word to be tested is: hello

134513930
acad/cs101/examples/notes ==>
```

The value being "returned" looks suspicious!

The unit test allows us find and fix the error in *isHello()*

```
Word to be tested is: abcd
0
Word to be tested is: hell
0
Word to be tested is: hello
1
Word to be tested is: hello

0
acad/cs101/examples/notes ==>
```

The final phase of development is called system integration. In this phase the real *main()* is mated to the *pre-tested* components.

It is not uncommon to uncover additional programming errors here.

Parameter passing

There are many possible mechanisms by which parameters may be passed.

The standard C language uses *call-by-value* for passing all *scalar* data types.

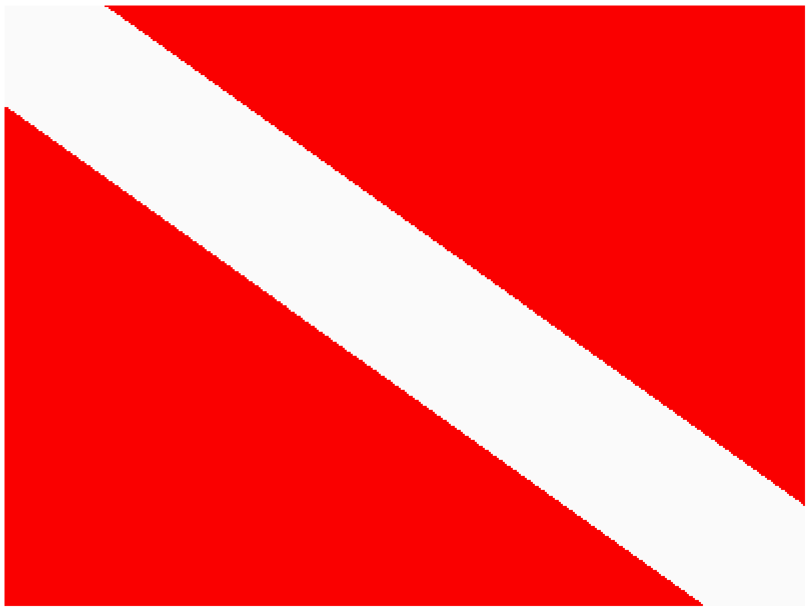
In this approach a *copy* of the parameter is placed on the stack. The called function is free to modify the copy as it wishes, but this will have *no effect* on the value held by the caller.

```
/* p23.c */

/* Parameter passing 1 */

int try_to_mod(
int a)
{
   printf("The address of try's a is %p\n", &a);
   a = 15;
}

main()
{
   int a = 20;

   printf("The address of main's a is %p\n", &a);
   try_to_mod(a);
   printf("a = %d \n", a);
}
```

```
class/215/examples ==> a.out
The address of main's a is 0xbffff884
The address of try's a is 0xbffff870
a = 20
```

Functions and a *flag* problem

Suppose our mission is to produce the following “flag”



A point's position with respect to a line

We might initally observe that it would be a good thing to be able to tell whether or not a pixel lay above or below a *non-vertical* line.

We hopefully know that two points *(x0, y0)* and *(x1, y1)* determine a line.

Furtherthermore the slope of that line is:

$$m = \frac{y1 - y0}{x1 - x0}$$

And the coordinates of any abitrary point (x, y) that lies on the line is:

$$y = y0 + m\,(x - x0)$$

Therefore given a test point (x, y) we compute the y' that corresponds to x and lies on the line

$$y' = y0 + m\,(x - x0)$$

If $y' > y$ then *(x, y)* lies below the line otherwise *(x,y)* lies above the line.

Design of our flag program

There several rational ways to commence building our flag program.

- begin with a top down design and fill in the holes incrementally
- build components from the bottom up and unify them at the end
- some hybrid combination of the above

I chose the third option and started with the *isAbove()* function

The *isAbove* function

```
#include <stdio.h>

int isAbove(
int x,       /* coordinates of test point */
int y,
int x0,      /* coordinates of one point on the line */
int y0,
int x1,      /* coordinates of another point on line */
int y1)
{
   float slope;
   float yval;
   int   out;

   slope = y1 - y0;
   slope = slope / (x1 - x0);

   yval = y0 + slope * (x1 - x0);

   if (yval > y)
      out = 0;
   else
      out = 1;

   return(out);
}

main()
{
   printf("%d\n", isAbove(50, 53, 0, 0, 100, 100));
   printf("%d\n", isAbove(50, 47, 0, 0, 100, 100));
   printf("%d\n", isAbove(50, 52, 0, 100, 100, 0));
   printf("%d\n", isAbove(50, 48, 0, 100, 100, 0));
}
```

```
acad/cs101/examples/notes ==> p23
0    <-- should be above
0
1
1    <--- should be below
```

Finding the error

This error is not especially easy to see and could be *verrrrry harrddd t*o find when embedded in the complete flag program.

```
acad/cs101/examples/notes ==> gdb p23
(gdb) b isAbove
Breakpoint 1 at 0x8048382: file p23.c, line 15.
(gdb) r
Breakpoint 1, isAbove (x=50, y=53, x0=0, y0=0, x1=100, y1=100) at
p23.c:15
15          slope = y1 - y0;
(gdb) n
16          slope = slope / (x1 - x0);
(gdb) n
18          yval = y0 + slope * (x1 - x0);
(gdb) print slope
$1 = 1
(gdb) n
20          if (yval > y)
(gdb) print yval
$2 = 100
(gdb) print x
$3 = 50
(gdb)
```

Oops, the value of y on this line should be 50.

A correct version of *isAbove()*

```
#include <stdio.h>

int isAbove(
int x,       /* coordinates of test point */
int y,
int x0,      /* coordinates of one point on the line */
int y0,
int x1,      /* coordinates of another point on line */
int y1)
{
   float slope;
   float yval;
   int   out;

   slope = y1 - y0;
   slope = slope / (x1 - x0);

   yval = y0 + slope * (x - x0);

   if (yval > y)
      out = 0;
   else
      out = 1;

   return(out);
}

main()
{
   printf("%d\n", isAbove(50, 53, 0, 0, 100, 100));
   printf("%d\n", isAbove(50, 47, 0, 0, 100, 100));
   printf("%d\n", isAbove(50, 52, 0, 100, 100, 0));
   printf("%d\n", isAbove(50, 48, 0, 100, 100, 0));
}
```

```
acad/cs101/examples/notes ==> p23b
1
0
1
0
```

Disabling the *main()* function

When we believe that the function is working correctly. we want to disable the *main()* so that it want conflict with our real *main()* function later on. We could just delete it, but we may need to get it back in case we suspect later that *isAbove()* wasn't as correct as we originally thought.

Here is a way to do that.

```
#if 0
main()
{
   printf("%d\n", isAbove(50, 53, 0, 0, 100, 100));
   printf("%d\n", isAbove(50, 47, 0, 0, 100, 100));
   printf("%d\n", isAbove(50, 52, 0, 100, 100, 0));
   printf("%d\n", isAbove(50, 48, 0, 100, 100, 0));
}
#endif
```

The *#if* directive tells the C preprocessor to include the following block in the compilation only if the expression that follows evaluates to *true*.

If we later discover we need to enable main.

```
#if 1
main()
{
   printf("%d\n", isAbove(50, 53, 0, 0, 100, 100));
   printf("%d\n", isAbove(50, 47, 0, 0, 100, 100));
   printf("%d\n", isAbove(50, 52, 0, 100, 100, 0));
   printf("%d\n", isAbove(50, 48, 0, 100, 100, 0));
}
#endif
```

Controlling the compilation from *gcc*

An even better approach is to use the *#ifdef* facility and pass a symbol through the gcc command.

```
#ifdef DBG_ABOVE
main()
{
   printf("%d\n", isAbove(50, 53, 0, 0, 100, 100));
   printf("%d\n", isAbove(50, 47, 0, 0, 100, 100));
   printf("%d\n", isAbove(50, 52, 0, 100, 100, 0));
   printf("%d\n", isAbove(50, 48, 0, 100, 100, 0));
}
#endif
```

```
gcc -D DBG_ABOVE p23c.c
```

The main function version 1

```
/* dive.c */

#include <stdio.h>
#include <stdlib.h>

int main(
int argc,
char *argv[])
{
   int width = 0;
   int height;
   int row;
   int col;

   if (argc < 2)
   {
      fprintf(stderr, "usage is: dive width-of-flag \n");
      exit(-1);
   }

   sscanf(argv[1], "%d", &width);
   height = width * ASPECT;
   fprintf(stderr, "%d %d\n", width, height);
}
```

```c
/* dive.c */

#include <stdio.h>
#include <stdlib.h>

int make_ppm_hdr(
int w,
int h)
{
   fprintf(stdout, "P6\n");
   fprintf(stdout, "%d %d %d\n", w, h, 255);
}

int main(
int argc,
char *argv[])
{
   int width = 0;
   int height;
   int row;
   int col;

   if (argc < 2)
   {
      fprintf(stderr, "usage is: dive width-of-flag \n");
      exit(-1);
   }

   sscanf(argv[1], "%d", &width);
   height = width * ASPECT;
   fprintf(stderr, "%d %d\n", width, height);

   make_ppm_hdr(width, height);
}
```

The final main program

```
/* dive.c */

#include <stdio.h>
#include <stdlib.h>

int make_ppm_hdr(
int w,
int h)
{
   fprintf(stdout, "P6\n");
   fprintf(stdout, "%d %d %d\n", w, h, 255);
}

int main(
int argc,
char *argv[])
{
   int width = 0;
   int height;
   int row;
   int col;

   if (argc < 2)
   {
      fprintf(stderr, "usage is: dive width-of-flag \n");
      exit(-1);
   }

   sscanf(argv[1], "%d", &width);
   height = width * ASPECT;
   fprintf(stderr, "%d %d\n", width, height);

   make_ppm_hdr(width, height);

   make_line_defs(width, height);

   row = height - 1;
   while (row >= 0)
   {
      write_row(width, height, row);
      row = row - 1;
   }
}
```

The *row* value corresponds to the *y* coordinate of a pixel.

ppm images are necesssarily written *top row down to bottom row* .

Managing line geometry

Here we make use of global variables. Sometimes this is disparaged, but until we understand structures and pointers it helps us avoid passing a huge number of parameters around.

```
#define ASPECT (3.0 / 4.0)
#define OFFSET (1.0 /  8.0)

int ul_x0;
int ul_y0;
int ul_x1;
int ul_y1;

int ll_x0;
int ll_y0;
int ll_x1;
int ll_y1;

int printline(
char tag[],
int  x0,
int  y0,
int  x1,
int  y1)
{
   fprintf(stderr, "%s: (%d, %d) (%d, %d) \n",
                           tag, x0, y0, x1, y1);
}
```

```
int make_line_defs(
int w,
int h)
{
   int pixoff = w * OFFSET;

   ul_x0 = pixoff;
   ul_y0 = h - 1;
   ul_x1 = w - 1;
   ul_y1 = pixoff;

   println("Upper",  ul_x0, ul_y0, ul_x1, ul_y1);

   ll_x0 = 0;
   ll_y0 = h - 1 - pixoff;
   ll_x1 = w - 1 - pixoff;
   ll_y1 = 0;

   println("Lower", ll_x0, ll_y0, ll_x1, ll_y1);
}
```

Building the flag image

```
int write_red_pixel()
{
   fprintf(stdout, "%c%c%c", 250, 0, 0);
}

int write_white_pixel()
{
   fprintf(stdout, "%c%c%c", 250, 250, 250);
}

int make_pixel(
int x,
int y)
{
    if (isAbove(x, y, ul_x0, ul_y0, ul_x1, ul_y1))
       write_red_pixel();
    else if (isAbove(x, y, ll_x0, ll_y0, ll_x1, ll_y1))
       write_white_pixel();
    else
       write_red_pixel();
}

/* write a single row of pixels */

int write_row(
int w,
int h,
int row)
{
   int x;
   int y;

   y = row;
   x = 0;
   while (x < w)
   {
      make_pixel(x, y);
      x = x + 1;
   }
}
```

The order of functions in a C program:

The C language generally requires that all entites be defined in a program before they are referenced.

```
acad/cs101/examples/notes ==> cat p24.c
int main()
{

   x = sum(5, 3);
   int x;
   return(0);
}

int sum(
int a,
int b)
{
   return(a + b);
}

acad/cs101/examples/notes ==> gcc -Wall p25.c
p25.c: In function ‘main’:
p25.c:5: error: ‘x’ undeclared (first use in this function)
p25.c:5: error: (Each undeclared identifier is reported only once
p25.c:5: error: for each function it appears in.)
p25.c:5: warning: implicit declaration of function ‘sum’
p25.c:6: warning: unused variable ‘x’
```

For functions the rule may appear somewhat relaxed.

The compiler may produce a warning instead of an error and proceed to compile the program.

Many people believe this relaxation is a *bad idea.*

Here we fix the declaration of *x*

```
acad/cs101/examples/notes ==>
int main()
{
   int x;

   x = sum(5, 3);
   return(0);
}

int sum(
int a,
int b)
{
   return(a + b);
}
```

The compilation works but produces a *nastygram.*

```
acad/cs101/examples/notes ==> gcc -Wall p24.c
p24.c: In function ‘main’:
p24.c:6: warning: implicit declaration of function ‘sum’
acad/cs101/examples/notes ==>
```

Other times the compiler may generate an error.

The error occurs because the *default return type* for an undeclared function is *int.*

- *At line 7 sum was implicitly assumed to return int.*
- *At line 15 sum was seen to actually return float.*

```
acad/cs101/examples/notes ==> cat p29.c
#include <stdio.h>

int main()
{
   int x;

   x = sum(11.0, 3);
   printf("%d \n", x);
   return(0);
}

float sum(
int a,
int b)
{
   return(a + b);
}

acad/cs101/examples/notes ==> gcc -Wall p29.c
p29.c: In function ‘main’:
p29.c:7: warning: implicit declaration of function ‘sum’
p29.c: At top level:
p29.c:15: error: conflicting types for ‘sum’
p29.c:7: error: previous implicit declaration of ‘sum’ was here
acad/cs101/examples/notes ==>
```

Consequences of ignoring the warning

Even though only a warning may issued by the compiler, you may get a defective result when the function is not defined at the time it is called.

Here the compiler can't know what type of parameters *sum* is expecting. Therefore, it passes it a *float* and *int*.   The *sum* function assumes that its first parameter is an *int* and big trouble ensues.

```
#include <stdio.h>

int main()
{
   int x;

   x = sum(11.0, 3);
   printf("%d \n", x);
   return(0);
}

int sum(
int a,
int b)
{
   return(a + b);
}

acad/cs101/examples/notes ==> !gcc -Wall
acad/cs101/examples/notes ==> gcc -Wall p27.c
p27.c: In function ‘main’:
p27.c:7: warning: implicit declaration of function ‘sum’
acad/cs101/examples/notes ==> a.out
1076232192
acad/cs101/examples/notes ==>
```

Avoiding the warnings and the errors

One way to do this is just to order functions so that the definition always appears before the first invocation of the function.

```
acad/cs101/examples/notes ==> cat p28.c
#include <stdio.h>

int sum(
int a,
int b)
{
   return(a + b);
}

int main()
{
   int x;

   x = sum(11.0, 3);
   printf("%d \n", x);
   return(0);
}

acad/cs101/examples/notes ==> a.out
14
```

Function prototypes

The ordering strategy proposed on the previous page can't solve the problem in two situations:

- The function definition is located in another module (as was the *isAbove()* function)
- Two or more functions are mutually refernential. *a calls b which calls c which calls a.*

A prototype is a function header which is followed by a semicolon instead of a basic block. The prototype tells the compiler:

- the type of value returned by the function
- the type of each parameter

When the compiler has this information it will generate code that will automagically coerce actual arguments to the correct type.

```
#include <stdio.h>

int sum(int a, int b);

int main()
{
   int x;
```

Compiler now generates code to convert the *float* value to *int* before passing it to sum/

```
   x = sum(11.0, 3);
   printf("%d \n", x);
   return(0);
}

int sum(
int a,
int b)
{
   return(a + b);
}
acad/cs101/examples/notes ==> gcc -Wall p30.c
acad/cs101/examples/notes ==> a.out
14
acad/cs101/examples/notes ==>
```

Function prototypes for standard library functions are always available in a *.h* file. Failure to include the proper *.h* file can lead to failure of your program.