Functions in C

Fortran 90 has three kinds of units: a *program unit*, *subroutine units* and *function units*.

In C, all units are functions.

For example, the C counterpart to the Fortran 90 program unit is the function named `main`:

```
#include <stdio.h>

main ()
{ /* main */
  float w, x, y, z;
  int i, j, k;

  w = 0.5; x = 5.0; y = 10.0;
  z = x + y * w;
  i = j = k = 5;
  printf("x = %f, y = %f, z = %f\n", x, y, z);
  printf("i = %d, j = %d, k = %d\n", i, j, k);
} /* main */
```

Every C program must have a function named `main`; it's the function where the program begins execution.

C also has a bunch of standard *library functions*, which are functions that come predefined for everyone to use.

Standard Library Functions in C[1]

C has a bunch of standard *library functions* that everyone gets to use for free.

They are analogous to Fortran 90's intrinsic functions, but they're not quite the same.

Why? Because Fortran 90's intrinsic functions are built directly into the language, while C's library functions are not really built into the language as such; you could replace them with your own if you wanted.

Here's some example standard library functions in C:

Function	Return Type	Return Value	`#include` file
`printf`	`int`	number of characters written	`stdio.h`
	Print (output) to standard output (the terminal) in the given format		
`scanf`	`int`	number of items input	`stdio.h`
	Scan (input) from standard input (the keyboard) in the given format		
`isalpha`	`int`	Boolean: is argument a letter?	`ctype.h`
`isdigit`	`int`	Boolean: is argument a digit?	`ctype.h`
`strcpy`	`char [ ]`	string containing copy	`string.h`
	Copy a string into another (empty) string		
`strcmp`	`int`	comparison of two strings	`string.h`
	Lexical comparison of two strings; result is index in which strings differ: negative value if first string less than second, positive if vice versa, zero if equal		
`sqrt`	`float`	square root of argument	`math.h`
`pow`	`float`	1st argument raised to 2nd argument	`math.h`

[1] Brian W. Kernighan & Dennis M. Ritchie, *The C Programming Language*, 2nd ed., Prentice Hall, New Jersey, 1988, pp. 241-258.

Functions in C That Don’t Return Anything

In C, all units are functions. But what if we want a unit that doesn’t return anything?

In that case, we have a special return type: `void`. This type means “no value at all.”

We’ve already seen an example of a `void` library function:

```
exit(-1);
```

Functions with return type `void` in C are analogous to subroutines in Fortran 90.

Functions in C Whose Return Value Is Ignored

We saw on the previous slide that the standard I/O library function `printf` returns an integer representing the number of characters printed.

Yet in every case where we’ve used `printf`, we haven’t considered the return value; we’ve just used the `printf` function as though it were a Fortran 90 `PRINT` statement:

```
printf("x = %f, y = %f, z = %f\n",
       x, y, z);
```

C allows us to ignore the return value of any function; unlike Fortran 90, we don’t have to use the return value in an expression.

If we ignore a function’s return value, it just disappears. Computer geeks refer to this as “sending it to the bit bucket.”

Example: Return Value of `printf`

```
% cat assn3.c
#include <stdio.h>

main ()
{ /* main */
  float w, x, y, z;
  int i, j, k;
  int   numprinted;

  w = 0.5; x = 5.0; y = 10.0;
  z =
    x + y * w;
  i = 12; j = 5; k = i / j;
  numprinted =
    printf("x = %f, y = %f, z = %f\n",
           x, y, z);
  printf("numprinted = %d\n",
         numprinted);
  numprinted =
    printf("i = %d, j = %d, k = %d\n",
    i, j, k);
  printf("numprinted = %d\n",
         numprinted);
} /* main */
% cc -o assn3 assn3.c
% assn3
x = 5.000000, y = 10.000000, z = 10.000000
numprinted = 43
i = 12, j = 5, k = 2
numprinted = 21
```

Functions in C That Have No Arguments

In C, a function doesn't have to have arguments. A function that has no arguments works just like a function that has arguments — except that it has no arguments.

For example, we've seen a function `main` in several programs that takes no arguments:

```
#include <stdio.h>

main ()
{ /* main */
  float w, x, y, z;
  int i, j, k;

  w = 0.5; x = 5.0; y = 10.0;
  z = x + y * w;
  i = j = k = 5;
  printf("x = %f, y = %f, z = %f\n", x, y, z);
  printf("i = %d, j = %d, k = %d\n", i, j, k);
} /* main */
```

Another standard I/O library function that has no arguments is `getchar`, which inputs a single character from the standard input (the keyboard), returning it as an `int`:

```
int c;

c = getchar();
```

Aside: Fortran 90 also supports functions that have no arguments.

Function With No Arguments Example: `getchar`

```
% cat getchartest.c
#include <stdio.h>

main () { /* main */
  char mystring[100];
  int c, i = 0;

  do {
    c = getchar();
    if (c != EOF) {
      mystring[i] = (char)c;
      i++;
    } /* if c != EOF */
  } while (c != EOF);
  mystring[i] = '\0';
  printf("mystring = %s\n",
          mystring);
} /* main */
% cc -o getchartest getchartest.c
% getchartest
 abcde
 ^D
mystring = abcde
```

What does `EOF` refer to?

That's a special code that `getchar` returns, which means that the "end of file" has been reached. It corresponds to typing **Ctrl-D** from the keyboard.

User-Defined Functions in C

Just as in Fortran 90, in C we can define our own functions:

```
% cat cuberoottest.c
#include <stdio.h>
#include <math.h>

#define CUBE_ROOT_POWER  1.0 / 3.0

main () { /* main */
  float cbrt;
  float cube_root(float x);

  printf("cube_root(%3.1f) = %f\n",
    5.0, cube_root(5.0));
} /* main */

float cube_root (float x) {
  return pow(x, CUBE_ROOT_POWER);
} /* cube_root */

% cc -o cuberoottest cuberoottest.c
ld:
Unresolved:
pow
% cc -o cuberoottest cuberoottest.c -lm
% cuberoottest
cube_root(5.0) = 1.709976
```

Notice that in C, unlike in Fortran 90, we don't have a return variable; rather, we return the appropriate value using a `return` statement.

User-Defined Functions in C (continued)

The general form for a user-defined function in C is:

```
returntype funcname ( arg1type arg1, arg2type arg2, ... )
{
   localtype1 localvar1_1, localvar1_2, ... ;
   localtype2 localvar2_1, localvar2_2, ... ;
   ...
   [ function body ]
   return returnvalue;
}
```

So a function definition has:

1. A *header* consisting of:
   (a) a return type;
   (b) a function name;
   (c) a list of arguments, enclosed in parentheses and separated by commas, where each formal argument is preceded by its type;
2. A function block, enclosed in curly braces, consisting of:
   (a) declarations;
   (b) executable statements.

Note: in C, you're allowed to leave off the return type in the function header, in which case the return type is `int` by default.

C Functions Prototypes

Here's our cube root program:

```
% cat cuberoottest.c
#include <stdio.h>
#include <math.h>

#define CUBE_ROOT_POWER  1.0 / 3.0

main () { /* main */
  float cbrt;
  float cube_root(float x);

  printf("cube_root(%3.1f) = %f\n",
    5.0, cube_root(5.0));
} /* main */

float cube_root (float x) {
  return pow(x, CUBE_ROOT_POWER);
} /* cube_root */

% cc -o cuberoottest cuberoottest.c -lm
% cuberoottest
cube_root(5.0) = 1.709976
```

Notice that, in the declaration section of the function `main`, we have a declaration that looks like this:

```
float cube_root(float x);
```

This is called a *function prototype*. It's basically the function header followed by a semicolon.

What does it do?

In C, functions are largely independent of each other — just as in Fortran 90, in which function, subroutine and program units are largely independent of each other.

C function prototypes serve a similar purpose to EXTERNAL declarations of functions in Fortran 90, but they don't just tell the return type; rather, they provide enough information about the function's arguments that it's hard to become confused about what arguments a function expects — and the compiler can alert you if you make a mistake.

C Function With Side Effects

```
% cat userarray.c
#include <stdio.h>

#define MAXNUMELEMENTS  100

main () { /* main */
  float element[MAXNUMELEMENTS];
  int number_of_elements;
  int input_number_of_elements(int maxnumelts);

  number_of_elements =
    input_number_of_elements(MAXNUMELEMENTS);
  printf("The number of elements you want is %d.\n",
    number_of_elements);
} /* main */

#include "inputnumeltsint.c"
% cat inputnumeltsint.c
int input_number_of_elements (int maxelts)
{ /* input_number_of_elements */
  int numelts;

  do {
    printf("How many elements would you like\n");
    printf(
      "  the array to have (between %d and %d)?  ",
      1, maxelts);
    scanf("%d", &numelts);
    if (numelts < 0)
      printf("You can't have negative elements!\n");
    else if (numelts == 0)
      printf("You can't have no elements!\n");
    else if (numelts > maxelts)
      printf("You have too many elements!\n");
  } while ((numelts < 1) || (numelts > maxelts));
  return numelts;
} /* input_number_of_elements */
% cc -o userarray userarray.c
% userarray
How many elements would you like
  the array to have (between 1 and 100)?  -1
You can't have negative elements!
How many elements would you like
  the array to have (between 1 and 100)?  0
You can't have no elements!
How many elements would you like
  the array to have (between 1 and 100)?  101
You have too many elements!
How many elements would you like
  the array to have (between 1 and 100)?  55
The number of elements you want is 55.
```

Changing Argument Values in C Functions

C doesn't support an `INTENT` attribute for arguments, the way that Fortran 90 does.

So, in C, we have complete freedom to change the values of our arguments.

However, just because we change the value of an argument inside a function, that doesn't mean that we'll also change the value of the argument in the function that called it:

```
% cat mybadincrement.c
#include <stdio.h>

main () { /* main */
  int x = 5;
  void myincrement(int var);

  printf("main:  before call, x = %d\n", x);
  myincrement(x);
  printf("main:  after  call, x = %d\n", x);
} /* main */

void myincrement (int var)
{ /* myincrement */
  printf("myincrement:  before inc, var = %d\n", var);
  var = var + 1;
  printf("myincrement:  after  inc, var = %d\n", var);
} /* myincrement */
% cc -o mybadincrement mybadincrement.c
% mybadincrement
main:  before call, x = 5
myincrement:  before inc, var = 5
myincrement:  after  inc, var = 6
main:  after  call, x = 5
```

Why does this happen?

Pass By Value vs. Pass By Reference

In C, when an argument is passed to a function, the program grabs a new location in memory and **copies** the value of the actual argument into this new location, which is then used as the formal argument.

This approach is named *pass by value* or *call by value*.

In Fortran 90, by contrast, arguments are used with an approach named *pass by reference* or *call by reference*.

We can visualize *pass by reference* by imagining Henry's house, which has the address

1802 23rd Ave SE

We can *refer* to Henry's house this way:

Henry's house

But we can also *refer* to Henry's house this way:

Dr. Neeman's house

So, "Henry's house" and "Dr. Neeman's house" are two different names for the same location; they are *aliases*.

In Fortran 90, when we call a procedure, each actual argument and its corresponding formal argument are aliases of the same location in memory.

Fortran 90 Pass By Reference Example

```
% cat henryshouse.f90
PROGRAM henryshouseprog
  IMPLICIT NONE
  INTEGER :: henryshouse
  CALL who(henryshouse)
  PRINT *, henryshouse, " people live in Henry's house."
END PROGRAM henryshouseprog

SUBROUTINE who (drneemanshouse)
  IMPLICIT NONE
  INTEGER :: drneemanshouse
  PRINT *, "How many people live in Dr. Neeman's house?"
  READ *, drneemanshouse
END SUBROUTINE who
% f90 -o henryshouse henryshouse.f90
% henryshouse
 How many people live in Dr. Neeman's house?
 2
           2  people live in Henry's house.
```

C Pass By Value Example: BAD

```
% cat henryshousebad.c
#include <stdio.h>

main () { /* main */
  int henryshouse;
  void who(int house);

  who(henryshouse);
  printf("%d people live in Henry's house.\n",
    henryshouse);
} /* main */

void who (int drneemanshouse)
{ /* who */
  printf("How many people live in Dr Neeman's house?  ");
  scanf("%d", &drneemanshouse);
} /* who */
% cc -o henryshousebad henryshousebad.c
cc: Warning: henryshousebad.c, line 7:
    The scalar variable "henryshouse" is fetched but not
      initialized. (uninit1)
  who(henryshouse);
------^
```

Corrected C Pass By Reference Example

```
% cat henryshouse.c
#include <stdio.h>

main () { /* main */
  int henryshouse;
  void who(int *house);

   /* v */
   /* v */
  who(&henryshouse);
   /* ^ */
   /* ^ */
  printf("%d people live in Henry's house.\n",
    henryshouse);
} /* main */

            /* v */
            /* v */
void who (int *drneemanshouse)
            /* ^ */
            /* ^ */
{ /* who */
  printf("How many people live in Dr Neeman's house?  ");
  scanf("%d", drneemanshouse);
} /* who */
% cc -o henryshouse henryshouse.c
% henryshouse
How many people live in Dr Neeman's house?  2
2 people live in Henry's house.
```

How Does This Work?

In C, the default passing strategy is pass by value.

To pass by reference, we have to piggyback on top of the pass by value strategy.

So, the **value** that we have to pass is the **address** of the actual argument, which we achieve using the *address operator*, the ampersand.

Exercise: Rewriting Tax Input Function

Write a piece of the tax program from Mini Project #$3\frac{3}{4}$ in C, creating a `main` function, as well as an input function to have the user input the wages, taxable interest, unemployment compensation and tax withheld. The `main` function should call only the input function; don't worry about calculations or output.

The body of the program must not have any numeric or logical literal constants; all constants must be defined as macros using appropriate macro names.