Looking at Functions

Overview

In this lesson we will

✓ We will begin the study of functions
✓ We will write functions
  To use data the user can enter into our program
  Return computed results
  Such results can be
    Used as input data in other functions
    Displayed

Anatomy of a Function

Let's now take first look at the other kind of C identifier

Function

Up to this point

Have been looking at some of words in C language
  Tossed a few adjectives and adverbs to spice things up
We now begin to combine
  Words into sentences
  Sentences into paragraphs

C language program simply collection of functions
  Includes main()

Some functions are pre-defined
  We call these library functions
    We find these in the Standard C Libraries
      The printf() function is one example
    We'll examine these shortly

Most functions

User defined
  These will be our first focus
We wrote one function already

\textit{main()}

As we write larger programs
Will need to write more functions
More sophisticated functions
  We’ll also comment later on general rules
  For writing functions
Our main rule will be
  Keep things simple
    We call this the \textit{KISS Principle}
      \textit{Keep It Simple Stupid}

We write such functions
  Decompose program into more manageable pieces
  May combine functions into modules
    Like chapters in a book

If we’re careful with our designs
  Our clients will be able to use our functions
    Accomplish some task
  Will not need to know
    Internals of function

If we’re very clever
  We’ll be able to
    Change and improve functions
      Keep behaviour same
  Clients will never know
  Let’s start to see how this all works

We create a function by \textit{defining it}
Defining a function
Involves providing

*Function header*

*Function Body*

**Function Header**

Specifies

- Function name
- Return type
- Parenthesized parameter list

**Function Name**

Each function has

- Name
- Way to receive data through *arguments*.

Called *parameter list* or *signature*

**Arguments or Parameter List**

Arguments enclosed in parentheses

Appear after the function name

Are separated by commas

Parentheses and the arguments

Called the *argument list*

Number, type and order of arguments

Called *signature* of function

Signature will be very important in later classes

Parameters given values

When we execute the function

Arguments can be used

- Receive data to be used by function
  
  Such arguments are *input arguments*

- Retrieve data from a function
  
  Such arguments are *output arguments*
Return data from a function using a C return statement

When value returned from a function
Using the return statement
Function is said to have a return value

Kind of data returned called the return type.
Can think of function’s return value
Replacing the function call

Sometimes functions have nothing to return
We use them to
Print or display something for example

Such functions have a return value of void

Function Body
Function is
Series of C instructions
Enclosed in curly braces

Such instructions express the function body

This is where the real work gets done

Thus we have…

```c
returnType functionName ( arg0, arg1…argn-1 )
{
    body
}

int multiply(int first, int second)
{
    // this is the function body
}
```
Using a Function

By itself function may be

Interesting
Elegant

Of little use

Unless we can do something with it

Using function called

• Executing
• Evaluating
• Invoking
• Calling

Function is executed

By performing a function call
Function performing the function call
Called the calling function
Function being executed
Called the called function

All functions created equal

Any function can call any other function

For the simple program

Example

#include <header file>
int main(void)
{
    // statements to execute go here

    return 0;
}

Header file or files

Typically library files containing
Prewritten functions that make our job easier
Including the header
Tells compiler to include bits of the files that we need

We see `main()`
- Is a function that has no arguments
  - We write `void` in the argument list
- Returns a zero when it completes

`main()` function called by operating system
- To start your program
- When `main()` completes
  - By executing all statements between the braces

Our program is complete
- Control returns to operating system.

Consider multiply function described above
- We can write the function body such that
  - It is able to multiply the first and second argument
  - Return the product

We might ask
- How do we call the function?
- How do we pass in the numbers to be multiplied?
- How do we use the product that is returned?

Let’s look at the following example

Example
```
functions0.c
#include <stdio.h>
int multiply(int length, int width); // function prototype
int main()
{
  // declare and initialize some variables
  int length = 10;
  int width = 20;
  int area = 0;

```
```c
area = multiply(length, width); // this is the function call
print("%d \n", area);          // displays 200
return 0;
}

int multiply(int first, int second)
{
    int answer;
    answer = first*second;
    return answer;
}
```

To get a function to execute lines of code contained in body

We must call the function
We do so by writing the
Name of the function
Followed by argument list
As line of code in our program
Line
area = multiply(length, width); // this is the function call

Observe
Arguments of multiply()
Not first and second
As written in the function prototype
But are length and width

The names length and width
Belong to the main() function
Names first and second
Belong to the multiply() function

When the multiply() function called
Compiler will make a copy of length
Then name the copy first
Likewise a copy of width will be named second

We say such a process
passes arguments to the function
\textit{multiply()} function will then
Multiply copies and return the product

When C program reaches line
\begin{verbatim}
area = multiply(length, width); // this is the function call
\end{verbatim}
Value needs to be assigned to the \textit{area} variable
However, the value is not present
It must be calculated

System
Step 1
\begin{itemize}
  \item Stops executing in \textit{main()}
  \item Makes copies of \textit{length} and \textit{width}
  \item Names the copies \textit{first} and \textit{second}
  \item Jumps to the opening brace of the \textit{multiply()} function
\end{itemize}

Step 2
\begin{itemize}
  \item Execution now starts going through code in the \textit{multiply()} function
    \begin{itemize}
      \item To calculate \textit{answer}
        \begin{itemize}
          \item Note: this variable belongs to \textit{multiply()}
        \end{itemize}
    \end{itemize}
  \item When the line following line is executed
    \begin{verbatim}
    answer = first * second;
    \end{verbatim}
    \begin{itemize}
      \item Product of \textit{first} (10) and \textit{second} (20) assigned to \textit{answer}
    \end{itemize}
\end{itemize}

Step 3
\begin{itemize}
  \item When the line is executed
    \begin{verbatim}
    return answer;
    \end{verbatim}
    \begin{itemize}
      \item A \textit{copy of answer} (200) must be made
    \end{itemize}
    \begin{itemize}
      \item To return to \textit{main()}
        \begin{itemize}
          \item - 8 of 14 -
        \end{itemize}
    \end{itemize}
\end{itemize}
*return* statement in *multiply()* is the end of the function

After this statement executes

- All of the copies of variables
  - Made for *multiply()* to use are destroyed
- All that’s left is the copy of *answer*

Step 4

Execution resumes in *main()*

*Copy of* *answer* *is now assigned to* *area*

- After the assignment is made
  - Copy of *answer* is destroyed

Now no trace that *multiply()* was ever called

**Function Prototypes**

If we look back at example program

- See line near top
  - `#include <stdio.h>`
  - `int multiply(length, width);`

Line that looks like function header important to program

- Called *function prototype*

In C functions need to be known before can be used

- This is information to the compiler

You might ask why we need this

- After all doesn’t function header provide same information
  - Yep
  - Problems is we often define functions
    - After functions that higher level functions that use them
      - *main()* for example
    - When compiler encounters function
Needs to know how to process it
Needs information contained in header
To keep compiler happy
Sufficient to declare function
If declaration not present
We get compile error

We declare function by listing
Return value (type)
Function name
Parameters
We don’t need function body

What happens if we don’t provide prototype
If we define function before caller
No problem
Compiler has sufficient information
If not
When compiler encounters function
Assumes function has some unknown set of arguments
…and returns an int

Such an assumption can lead to trouble
Compiler cannot do any type checking
To do such type checking
Parameter names not necessary just types
Good coding style strongly recommends including anyway

Very important that every function has prototype
Only function that does not require prototype
main()
Because there is only one such function in the program
It is the same function prototype
In all C programs so the compiler already knows about it.
Nesting Functions

When one function used inside second function
Called nesting

Good question is
How deeply can we nest functions
Accompanying figure show
Functions nested 4 deep

Question has a relative answer

We know compiler
Makes copies of the variables used for arguments
Destroys the copies when the function completes

Seems reasonable that all copies hang around
Until the most deeply nested function call completes

You may view this as airplanes "stacked up" waiting to land
When you run out of the stack memory your program will crash
Amount of stack memory varies with operating systems
Usually adjustable if needed

Generally if you see your program crash
With "no more stack memory"
You might be in some function call loop like this

```c
int main()
{
    functionA();
    return 0;
}

int functionA() int functionB()
{
    
    functionB(), functionA();
}
In above example

\[ \text{functionA()} \] in \text{main()} never returns
Because it calls \text{functionB()}
Which turns around and calls \text{functionA()}
Which calls \text{functionB()}, etc.

**Return Value**

In our study of functions so far
Observed prototype specifies return type
We now ask question
Do all functions have return value

Answer is no

Consider function that
Only displays a message and returns
Such a function might look like:

```c
void displayAnswer(int answer)
{
    printf( "The final answer is: %d \n", answer);
    return;       // this return is optional
                   // but is included for clarity
                   // and completeness
}
```

In this example
An \text{int} variable is received by the function \text{displayAnswer()}

displayAnswer() then
Displays an appropriate message on the screen
Returns to the calling function
\text{main()} in this case

There is no value to return
So the return statement is optional
However C syntax requires
   All functions specify return type
       In the function heading and
       In the function prototype

Can special return type of \textit{void} under such circumstances
   Indicates that no specific type is returned.

Now we use
   \texttt{displayAnswer()} function
       In our example C program:

\textit{Example}
\begin{verbatim}
functions1.c
#include <stdio.h>

int multiply(int length, int width);  // function prototypes
void displayArea(int area);

int main()
{
    // declare and initialize some variables
    int length = 10;
    int width =20;
    int area = 0;

    area = multiply(length, width);  // this is the function call
    displayArea(area);              // displays: The answer is 200
    return 0;
}

int multiply(int first, int second)
{
    int answer;
    answer = first*second;
    return answer;
}

void displayArea(int area)
{
    printf("The answer is %d\n", area);  // displays The answer is 200
    return;
}
\end{verbatim}
Library Functions
In addition to our own functions
Many functions we will use are in libraries

Library is pre-compiled code and
Usually comes with your compiler

To use a library function
We include the appropriate header file
Header file contains the function prototype
This satisfies the compiler

During compile
Function will still be flagged unresolved external
But when the linker searches the library for the function
Will be found and a copy of it placed in executable program

To know which header file to include
Must read the documentation that comes with compiler or
Read about it in a book.

Summary and Review of Objectives
At this time you should have
• A basic understanding of functions
  ✓ Purpose of a function prototype
• Know how to
  ✓ Write a prototype for a function
  ✓ Write functions
  ✓ Enter data into a function
  ✓ Use data the user can enter into our program
  ✓ Return computed results