Functions

Dr Deepayan Bhowmik Dr Andrea Bracciali

In this lecture …

- Functions

- Definitions
- Calls
- Declarations

- Parameter passing (including arrays).

- Termination

Introduction

- A function is a series of statements that have been grouped together and given a name.
- Each function is essentially a small program, with its own declarations and statements.
- Advantages of functions:
  - A program can be divided into small pieces that are easier to understand and modify.
  - We can avoid duplicating code that's used more than once.
  - A function that was originally part of one program can be reused in other programs.

Introduction

- A function is a series of statements that have been grouped together and given a name.
- Each function is essentially a small program, with its own declarations and statements.

```
double average(double a, double b)
{
  return (a + b) / 2;
}
```

Program: Computing Averages

- The `average.c` program reads three numbers and uses the `average` function to compute their averages, one pair at a time:

```
Enter three numbers: 3.5 9.6 10.2
Average of 3.5 and 9.6: 6.55
Average of 9.6 and 10.2: 9.9
Average of 3.5 and 10.2: 6.85
```

average.c

```
/* Computes pairwise averages of three numbers */

#include <stdio.h>

double average(double a, double b)
{
  return (a + b) / 2;
}

int main(void)
{
  double x, y, z;

  printf("Enter three numbers: ");
  scanf("%lf%lf%lf", &x, &y, &z);
  printf("Average of %g and %g: %g\n", x, y, average(x, y));
  printf("Average of %g and %g: %g\n", y, z, average(y, z));
  printf("Average of %g and %g: %g\n", x, z, average(x, z));

  return 0;
}
```

Program: Printing a Countdown

- If a function has no return value, its return type is **`void`**:

```
void print_count(int n)
{
  printf("T minus %d and counting\n", n);
}
```

- `void` is a type with no values.

- A call of `print_count` appears in a statement by itself, the `countdown.c` program calls `print_count` 10 times inside a loop.

countdown.c

```
/* Prints a countdown */

#include <stdio.h>

void print_count(int n)
{
  printf("T minus %d and counting\n", n);
}

int main(void)
{
  int i;

  for (i = 10; i > 0; --i)
    print_count(i);

  return 0;
}
```

Program: Printing a Pun

- When a function has no parameters, the keyword **`void`** is placed in parentheses after the function's name:

```
void print_pun(void)
{
  printf("To C, or not to C: that is the question.\n");
}
```

- The `pun2.c` program calls the `print_pun` function: function's name, followed by parentheses (needed!):

```
print_pun();
```

pun2.c

```
/* Prints a bad pun */

#include <stdio.h>

void print_pun(void)
{
  printf("To C, or not to C: that is the question.\n");
}

int main(void)
{
  print_pun();
  return 0;
}
```

Function Definitions

- General form of a function definition:

return-type function-name (*parameters*)
{
    declarations
    statements
}

Function Definitions

- The return type of a function is the type of value that the function returns.
  - Functions may **not return arrays**.
  - If the return type is **`void`** the function doesn't return a value.
  - If the return type is omitted in C89, the function is presumed to return a value of type `int`.
  - In C99, omitting the return type is illegal.

(remember that C89 and C99 are C standards – revise if needed!)

Function Definitions

- As a matter of style, some programmers put the return type *above* the function name:

```
double
average(double a, double b)
{
  return (a + b) / 2;
}
```

- Putting the return type on a separate line is especially useful if the return type is lengthy, like `unsigned long int`.

Function Definitions

- After the function name comes a list of formal parameters.

return-type function-name (*parameters*)
{
  declarations
  statements
}

```
void print_count(int n)
```

- Each parameter is preceded by a specification of its type; parameters are separated by commas.
- If the function has no parameters, the word `void` should appear between the parentheses.

Function Definitions

- Variables declared in the body of a function are local variables, and can't be seen or modified by other functions.

Important:

Linked to the concept of scoping rules, or visibility rules for variables, and environment.

- In C89, variable declarations must come first, before all statements in the body of a function.
- In C99, variable declarations and statements can be mixed, as long as each variable is declared prior to the first statement that uses the variable.

Function Definitions

- The body of a function whose return type is `void` (a "`void` function") can be empty:

```
void print_pun(void)
{
}
```

- Leaving the body empty may make sense as a temporary step during program development.

Function Calls

- The value returned by a non-`void` function can be used, assigned or discarded if it's not needed:

```
printf("%f", average(x, y));/* uses return value */
x = average(x, y);          /* assigns return value */
average(x, y);              /* discards return value */
```

The last call is an example of an expression statement: a statement that evaluates an expression but then discards the result.

Function Calls

- Ignoring the return value of `average` is an odd thing to do, but for some functions it makes sense, e.g.

```
num_chars = printf("Hi, Mom!\n");
```

- It returns the number of characters that it prints, `num_chars` will have the value 9.

- We'll normally discard `printf`'s return value:

```
printf("Hi, Mom!\n"); /* discards return value */
```

Function Calls

- To make it clear that we're deliberately discarding the return value of a function, C allows us to put `(void)` before the call:

```
(void) printf("Hi, Mom!\n");
```

Function Calls

Important:

procedural vs functional languages and side-effects:

a “pure function” does not have a state, it computes a results from the parameters and no other variable is permanently altered by the execution of the function.

in procedural languages, functions may alter “global” variables, a side-effect! This might be a not-so-safe programming style, since side-effects are not generally understandable from the declaration of the function! (more when studying programming languages theory).

Function Declarations

- Suppose we put the definition of `average` *after* `main`.

```
#include <stdio.h>

int main(void)
{
  double x, y, z;

  printf("Enter three numbers: ");
  scanf("%lf%lf%lf", &x, &y, &z);
  printf("Average of %g and %g: %g\n", x, y, average(x, y));
  printf("Average of %g and %g: %g\n", y, z, average(y, z));
  printf("Average of %g and %g: %g\n", x, z, average(x, z));

  return 0;
}

double average(double a, double b)
{
  return (a + b) / 2;
}
```

Function Declarations

- When the compiler encounters the first call of `average` in `main`, it has no information about the function.
- Instead of producing an error message, the compiler assumes that `average` returns an `int` value.
- We say that the compiler has created an implicit declaration of the function.
- When it encounters the definition of `average` later, the compiler notices the function's return type is `double`, not `int`, and so we get **an error message**.

Function Declarations

- C solution : declare each function before calling it.

return-type function-name (*parameters*) ;

- A function declaration provides the compiler with a brief glimpse at a function whose full definition will appear later.

- Here;s the `average.c` program with a declaration of `average` added.

Function Declarations

```
#include <stdio.h>

double average(double a, double b);    /* DECLARATION */

int main(void)
{
  double x, y, z;

  printf("Enter three numbers: ");
  scanf("%lf%lf%lf", &x, &y, &z);
  printf("Average of %g and %g: %g\n", x, y, average(x, y));
  printf("Average of %g and %g: %g\n", y, z, average(y, z));
  printf("Average of %g and %g: %g\n", x, z, average(x, z));

  return 0;
}

double average(double a, double b)     /* DEFINITION */
{
  return (a + b) / 2;
}
```

Function Declarations

- Function declarations of the kind we're discussing are known as function prototypes.
- C also has an older style of function declaration in which the parentheses are left empty.

- A function prototype doesn't have to specify the names of the function's parameters, as long as their types are present:

```
double average(double, double);
```

- It's usually best not to omit parameter names.

Function Declarations

- C99 has adopted the rule that either a declaration or a definition of a function must be present prior to any call of the function.
- Calling a function for which the compiler has not yet seen a declaration or definition is an error.

Arguments

- In C, arguments are passed by value: when a function is called, each argument (m and n) is evaluated and its value assigned to the corresponding parameter (x and y).

```
int sum (int x, int y); /* Declaration */
k = sum(m, n);          /* Call */
```

- Since the parameter, which is a local variable in the function, contains a copy of the argument's value, any changes made to the parameter during the execution of the function don't affect the argument (even if x and y were to be changed in sum, m and n would not be affected).

Arguments

- Consider the following function, which raises a number `x` to a power `n`:

```
int power(int x, int n)
{
  int i, result = 1;

  for (i = 1; i <= n; i++)
    result = result * x;

  return result;
}
```

Arguments

- Since `n` is a *copy* of the original exponent, the function can safely modify it, removing the need for `i`:

```
int power(int x, int n)
{
  int result = 1;

  while (n-- > 0)
    result = result * x;

  return result;
}
```

Arguments

- C's call-by-value may makes it difficult to write certain kinds of functions, e.g.

decompose a `double` number into an integer and a fractional part

A function can't *return* two numbers. Can we pass two variables to the function to hold the results, like `int_part` and `frac_part` below?

```
void decompose(double x, long int_part, double frac_part)
{
  int_part = (long) x;
  frac_part = x - int_part;
}
```

Arguments

- Unfortunately that will not work. Indeed, `i` and `d` in the call below, won't be affected by any assignments to `int_part` and `frac_part`

```
decompose(3.14159, i, d);
```

- This problem can be solved by using pointers, as in the discussed case of `scanf()` (more to come!).

Array Arguments

Array arguments need more details.

- When a function parameter is a one-dimensional array, the length of the array can be left unspecified:

```
int f(int a[])  /* no length specified */
{
  …
}
```

- C doesn't provide any easy way for a function to determine the length of an array passed to it. The length − if the function needs it − will be passed as an additional argument.

Array Arguments

- Example:

```
int sum_array(int a[], int n)
{
  int i, sum = 0;

  for (i = 0; i < n; i++)
    sum += a[i];

  return sum;
}
```

- Since `sum_array` needs to know the length of `a`, we supply it as a second argument. The length of `a` may well be a constant of the program.

Array Arguments

- The prototype for `sum_array` has the following appearance:

```
int sum_array(int a[], int n);
```

- As usual, we can omit the parameter names if we wish:

```
int sum_array(int [], int);
```

Array Arguments

- When `sum_array` is called, the first argument will be the name of an array, and the second will be its length:

```
const int LEN = 100;

int main(void)
{
  int b[LEN], total;
  …
  total = sum_array(b, LEN);
  …
}
```

- Notice that we don't put brackets after an array name when passing it to a function:

```
total = sum_array(b[], LEN);    /*** WRONG ***/
```

Array Arguments

- A function has no way to check that we've passed it the correct array length.

- We can exploit this fact by fixing a smaller boundary for the array, if useful.
- For instance, suppose we only have 50 numbers in the `b` array, even though it can hold 100, and we want to add only those 50:

```
total = sum_array(b, 50);
```

Array Arguments

- Be careful not to tell a function that an array argument is *larger* than it really is:

```
total = sum_array(b, 150);    /*** WRONG ***/
```

`sum_array` will go past the end of the array, causing undefined behavior.

Important:

This is a typical C error: the program will read beyond the memory space of the array with unpredictable results – typically a run-time error. C assumes you know what you are doing, be careful!

Array Arguments

Important:

A function is allowed to change the elements of an array parameter, and the change is reflected in the corresponding argument.

- A function that modifies an array by storing zero into each of its elements:

```
void store_zeros(int a[], int n)
{
  int i;

  for (i = 0; i < n; i++)
    a[i] = 0;
}
```

Array Arguments

- A call of `store_zeros`:

```
store_zeros(b, 100);
```

- The ability to modify the elements of an array argument may seem to contradict the fact that C passes arguments by value.

- We'll learn later why there's actually no contradiction.

Array Arguments

- If a parameter is a multidimensional array, only the length of the first dimension may be omitted.
- If we revise `sum_array` so that `a` is a two-dimensional array, we must specify the number of columns in `a`:

```
const int LEN = 10;

int sum_two_dimensional_array(int a[][LEN], int n)
{
  int i, j, sum = 0;

  for (i = 0; i < n; i++)
    for (j = 0; j < LEN; j++)
      sum += a[i][j];

  return sum;
}
```

Array Arguments

- Not being able to pass multidimensional arrays with an arbitrary number of columns can be a nuisance.
- We can often work around this difficulty by using arrays of pointers (later).

- C99's variable-length array parameters provide an even better solution.

Variable-Length Array Parameters (C99)

- C99 allows the use of variable-length arrays as parameters.
- Consider the `sum_array` function:

```
int sum_array(int a[], int n)
{
  …
}
```

As it stands now, there's no direct link between `n` and the length of the array `a`.

- Although the function body treats `n` as a's length, the actual length of the array could be larger or smaller than `n`.

Variable-Length Array Parameters (C99)

- Using a variable-length array parameter, we can explicitly state that `a`'s length is `n`:

```
int sum_array(int n, int a[n])
{
  …
}
```

- The value of the first parameter (`n`) specifies the length of the second parameter (`a`).
- Note that the order of the parameters has been switched; order is important when variable-length array parameters are used.

Variable-Length Array Parameters (C99)

- There are several ways to write the prototype for the new version of `sum_array`.
- One possibility is to make it look exactly like the function definition:

```
int sum_array(int n, int a[n]);  /* Version 1 */
```

- Another possibility is to replace the array length by an asterisk (*):

```
int sum_array(int n, int a[*]);  /* Version 2a */
```

Variable-Length Array Parameters (C99)

- The reason for using the `*` notation is that parameter names are optional in function declarations.

- If the name of the first parameter is omitted, it wouldn't be possible to specify that the length of the array is `n`, but the `*` provides a clue that the length of the array is related to parameters that come earlier in the list:

```
int sum_array(int, int [*]);    /* Version 2b */
```

Variable-Length Array Parameters (C99)

- It's also legal to leave the brackets empty, as we normally do when declaring an array parameter:

```
int sum_array(int n, int a[]);   /* Version 3a */
int sum_array(int, int []);      /* Version 3b */
```

- Leaving the brackets empty isn't a good choice, because it doesn't expose the relationship between `n` and `a`.

Variable-Length Array Parameters (C99)

- In general, the length of a variable-length array parameter can be any expression.
- A function that concatenates two arrays `a` and `b`, storing the result into a third array named `c`:

```
int concatenate(int m, int n, int a[m], int b[n],
                int c[m+n])
{
  …
}
```

- The expression used to specify the length of `c` involves two other parameters, but in general it could refer to variables outside the function or even call other functions.

Variable-Length Array Parameters (C99)

- Variable-length array parameters with a single dimension have limited usefulness.
- They make a function declaration or definition more descriptive by stating the desired length of an array argument.
- However, no additional error-checking is performed; it's still possible for an array argument to be too long or too short.

Variable-Length Array Parameters (C99)

- Variable-length array parameters are most useful for multidimensional arrays.
- By using a variable-length array parameter, we can generalize the `sum_two_dimensional_array` function to any number of columns:

```
int sum_two_dimensional_array(int n, int m, int a[n][m])
{
  int i, j, sum = 0;

  for (i = 0; i < n; i++)
    for (j = 0; j < m; j++)
      sum += a[i][j];

  return sum;
}
```

Variable-Length Array Parameters (C99)

- Prototypes for this function include:

```
int sum_two_dimensional_array(int n, int m, int a[n][m]);
int sum_two_dimensional_array(int n, int m, int a[*][*]);
int sum_two_dimensional_array(int n, int m, int a[][m]);
int sum_two_dimensional_array(int n, int m, int a[][*]);
```

Using `static` in Array Parameter Declarations (C99)

- C99 allows the use of the keyword `static` in the declaration of array parameters.
- The following example uses `static` to indicate that the length of `a` is guaranteed to be at least 3:

```
int sum_array(int a[static 3], int n)
{
  …
}
```

Program Termination

- Normally, the return type of `main` is `int`:

```
int main(void)
{
  …
}
```

- Older C programs often omit `main`'s return type, taking advantage of the fact that it traditionally defaults to `int`:

```
main()
{
  …
}
```

Program Termination

- Omitting the return type of a function isn't legal in C99, so it's best to avoid this practice.
- Omitting the word void in main's parameter list remains legal, but − as a matter of style − it's best to include it.

Program Termination

- The value returned by `main` is a status code that can be tested when the program terminates.
- `main` should return 0 if the program terminates normally.
- To indicate abnormal termination, `main` should return a value other than 0.
- It's good practice to make sure that every C program returns a status code.

The `exit` Function

- Executing a `return` statement in `main` is one way to terminate a program.
- Another is calling the `exit` function, which belongs to `<stdlib.h>`.
- The argument passed to `exit` has the same meaning as `main`'s return value: both indicate the program's status at termination.
- To indicate normal termination, we'd pass 0:

```
exit(0);    /* normal termination */
```

The `exit` Function

- Since 0 is a bit cryptic, C allows us to pass `EXIT_SUCCESS` instead (the effect is the same):

```
exit(EXIT_SUCCESS);
```

- Passing `EXIT_FAILURE` indicates abnormal termination:

```
exit(EXIT_FAILURE);
```

- `EXIT_SUCCESS` and `EXIT_FAILURE` are macros defined in `<stdlib.h>`.
- The values of `EXIT_SUCCESS` and `EXIT_FAILURE` are implementation-defined; typical values are 0 and 1, respectively.

The `exit` Function

- The statement

  `return` *expression*`;`

  in `main` is equivalent to

  `exit(`*expression*`);`
- The difference between `return` and `exit` is that `exit` causes program termination regardless of which function calls it.
- The `return` statement causes program termination only when it appears in the `main` function.