# Functions

Instructor: Cong Pu

cong.pu@ttu.edu

# 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.

# Program: Computing Averages

- A function named `average` that computes the average of two `double` values:

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

- The word `double` at the beginning is the ***return type*** of `average`.
- The identifiers `a` and `b` (the function's ***parameters***) represent the numbers that will be supplied when `average` is called.

# Program: Computing Averages

- Every function has an executable part, called the ***body,*** which is enclosed in braces.
- The body of `average` consists of a single `return` statement.
- Executing this statement causes the function to "return" to the place from which it was called; the value of `(a + b) / 2` will be the value returned by the function.

# Program: Computing Averages

- A function call consists of a function name followed by a list of ***arguments.***
  - `average(x, y)` is a call of the `average` function.
- Arguments are used to supply information to a function.
  - The call `average(x, y)` causes the values of `x` and `y` to be copied into the parameters `a` and `b`.
- An argument doesn't have to be a variable; any expression of a compatible type will do.
  - `average(5.1, 8.9)` and `average(x/2, y/3)` are legal.

# Program: Computing Averages

- We'll put the call of `average` in the place where we need to use the return value.
- A statement that prints the average of `x` and `y`:

```
printf("Average: %g\n", average(x, y));
```

  The return value of `average` isn't saved; the program prints it and then discards it.
- If we had needed the return value later in the program, we could have captured it in a variable:

```
avg = average(x, y);
```

# 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.
- Rules governing the return type:
  - Functions may not return arrays.
  - Specifying that the return type is `void` indicates that 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.

# Function Definitions

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

- The body of a function may include both declarations and statements.
- An alternative version of the `average` function:

```
double average(double a, double b)
{
  double sum;          /* declaration */

  sum = a + b;         /* statement */
  return sum / 2;      /* statement */
}
```

# Function Calls

- A function call consists of a function name followed by a list of arguments, enclosed in parentheses:

```
average(x, y)
```

If the parentheses are missing, the function won't be called:

```
average;    /*** "WRONG" ***/
```

This statement is legal but has no effect.

# Function Calls

- A call of a `void` function is always followed by a semicolon to turn it into a statement:

- A call of a non-`void` function produces a value that can be stored in a variable, tested, printed, or used in some other way:

```
avg = average(x, y);
if (average(x, y) > 0)
  printf("Average is positive\n");
printf("The average is %g\n", average(x, y));
```

# Function Declarations

- Declare each function before calling it.
- A ***function declaration*** provides the compiler with a brief glimpse at a function whose full definition will appear later.
- General form of a function declaration:

  *return-type* *function-name* ( *parameters* ) ;
- The declaration of a function must be consistent with the function's definition.
- 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.***
- 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.

# Arguments

- The difference between parameter and argument
- In C, arguments are ***passed by value*:** when a function is called, each argument is evaluated and its value assigned to the corresponding parameter.
- Since the parameter 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.

# Argument Conversions

- C allows function calls in which the types of the arguments don't match the types of the parameters.
- The rules governing how the arguments are converted depend on whether or not the compiler has seen a prototype for the function (or the function's full definition) prior to the call.

# Argument Conversions

- ***The compiler has encountered a prototype prior to the call.***
- The value of each argument is implicitly converted to the type of the corresponding parameter as if by assignment.
- Example: If an `int` argument is passed to a function that was expecting a `double`, the argument is converted to `double` automatically.

# Argument Conversions

- ***The compiler has not encountered a prototype prior to the call.***
- The compiler performs the ***default argument promotions:***
  - `float` arguments are converted to `double`.
  - The integral promotions are performed, causing `char` and `short` arguments to be converted to `int`.

# The `return` Statement

- A non-`void` function must use the `return` statement to specify what value it will return.
- The `return` statement has the form

  `return` *expression*`;`
- The expression is often just a constant or variable:

```
return 0;
return status;
```

- More complex expressions are possible:

```
return n >= 0 ? n : 0;
```

# The `return` Statement

- If the type of the expression in a `return` statement doesn't match the function's return type, the expression will be implicitly converted to the return type.
  - If a function returns an `int`, but the `return` statement contains a `double` expression, the value of the expression is converted to `int`.

# The `return` Statement

- `return` statements may appear in functions whose return type is `void`, provided that no expression is given:

```
return;  /* return in a void function */
```

- Example:

```
void print_int(int i)
{
  if (i < 0)
    return;
  printf("%d", i);
}
```

# 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

- 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.