Functions

A *function* is a sequence of statements that have been grouped together and given a name.

Each function is essentially a small program, with its own declarations and statements.

Some 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.
- The memory cost of the program can be reduced if the memory needed by each a function is released when the function terminates.

The Caesar Cipher example discussed earlier provides an illustration of the use of functions in the design and implementation of a small C program.

©2005-2020 WD McQuain

Function Definitions

General form of a *function definition:*

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

```
double average(double a, double b) {

   return (a + b)/2.0;
}
```

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 and later, omitting the return type is illegal.

©2005-2020 WD McQuain

Function Definitions

Variables declared in the body of a function can't be examined or modified by other functions.

In C99 and later, variable declarations and statements can be mixed, as long as each variable is declared prior to the first statement that uses the variable.

Functions that do not return a value are declared with a return type of **`void`**.

The returned valued from a call to a non-**`void`** function may be ignored.

Many **`void`** functions can be improved by using a return type of **`bool`**.

©2005-2020 WD McQuain

Function Declarations

C doesn't require that the definition of a function precede its calls:

```
#include <stdio.h>

int main(void) {
  double x, y;

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

  return 0;
}

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

However, in that case, there should be a *declaration* of the function before the call.

Not doing so is always a bad idea… and may lead to errors…

In the absence of a function declaration, the compiler will infer one from the call…

 ©2005-2020 WD McQuain

Function Declarations

The declaration that the compiler infers may clash with the definition:

```
#include <stdio.h>

int main(void) {
  double x, y;

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

  return 0;
}

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

The C compiler infers a declaration, based on the call:

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

The C compiler always assumes return type is `int`.

©2005-2020 WD McQuain

Function Declarations

The compiler then complains bitterly when it a call without a declaration:

```
#772 wmcquain: T06> gcc -o fndecl -Wall -W fndecl.c

fndecl.c: In function ‘main’:
fndecl.c:8:47: warning: implicit declaration of function ‘average’ [-
Wimplicit-function-declaration]
    printf("Average of %g and %g: %g\n", x, y, average(x, y));
. . .
```

And complains again when it the return type it guesses mismatches a format specifier:

```
#772 wmcquain: T06> gcc -o fndecl -Wall -W fndecl.c

. . .                                                      ^~~~~~~
fndecl.c:8:35: warning: format ‘%g’ expects argument of type ‘double’, but
argument 4 has type ‘int’ [-Wformat=]
    printf("Average of %g and %g: %g\n", x, y, average(x, y));
. . .
```


©2005-2020 WD McQuain

Function Declarations

And complains one final time when the inferred declaration doesn't match the function definition:

```
#772 wmcquain: T06> gcc -o fndecl -Wall -W fndecl.c

. . .
fndecl.c: At top level:
fndecl.c:13:8: error: conflicting types for ‘average’
 double average(double a, double b) {
        ^~~~~~~
fndecl.c:8:47: note: previous implicit declaration of ‘average’ was here
    printf("Average of %g and %g: %g\n", x, y, average(x, y));
                                               ^~~~~~~
```

Moral: never ignore an *implicit function declaration* warning.

©2005-2020 WD McQuain

Function Declarations

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

The general form of a function declaration:

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

The declaration of a function must be consistent with the function's definition.

A function declaration looks just like the first line of the function definition:

```
double average(double a, double b);
```

Note: the parameter names can be omitted, but not the parameter types.

©2005-2020 WD McQuain

Function Declarations

The declaration of a function is usually placed at file level (or in a header file):

```
#include <stdio.h>

double average(double a, double b);

int main(void) {
  double x, y;

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

  return 0;
}

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

This version of the code will compile without issues.

©2005-2020 WD McQuain

Compiler Checks

The compiler will check whether a call to a function matches the declaration of that function:

```
// declaration:
double average(double a, double b);
```

Which will compile? Will there be errors? Warnings?

```
double x = 2.437, y = -3.194;
double z = average(x, y);
```

all types match; no problem

```
double x = 2.437, y = -3.194;
double z = average(x);
```

number of parameters in call does not agree with declaration; compilation error

```
int    x = 2, y = -3;
double z = average(x, y);
```

int parameters in call converted to doubles for fn

```
double x = 2.437, y = -3.194;
int    z = average(x, y);
```

return value converted from int to double

```
double x = 2.437, y = -3.194;
average(x, y);
```

parameter types match; return value is discarded

©2005-2020 WD McQuain

Formal vs Actual Parameters

Formal parameters are the names used in the function definition.

Actual parameters are the names used in the function call.

```
#include <stdio.h>

double average(double a, double b);

int main(void) {
  double x, y;

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

  return 0;
}

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


©2005-2020 WD McQuain

Formal Parameters are Pass-by-Value

Formal parameters have automatic storage duration and block scope, just like local variables.

Formal parameters are automatically initialized with a copy of the value of the corresponding actual parameter.

There is no connection between the actual and formal parameters other than that they store the same value at the time of the function call.

```
int Exp  = 10;
int Base =  4;

int BaseToExp = Power(Base, Exp);

printf("%d ^ %d = %d\n",
       Base,              // still 4
       Exp,               // still 10
       BaseToExp);
```

```
int Power(int X, int N) {

   int Result = 1;

   while (N-- > 0) {
      Result = Result * X;
   }

   return Result;
}
```

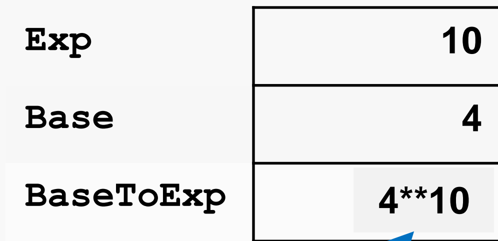

©2005-2020 WD McQuain

Formal Parameters are Pass-by-Value


```
int main() {
   . . .
   int Exp  = 10;
   int Base =  4;

   int BaseToExp = Power(Base, Exp);

   printf("%d ^ %d = %d\n",
          Base,              // still 4
          Exp,               // still 10
          BaseToExp);
   . . .
}
```

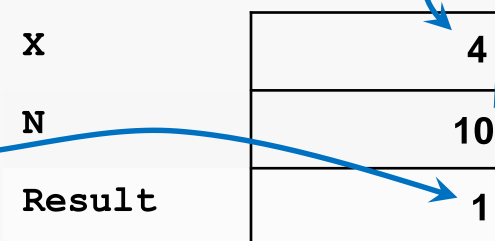


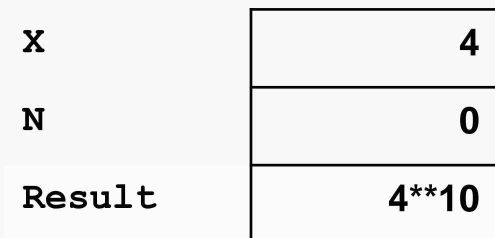

```
int Power(int X, int N) {

   int Result = 1;

   while (N-- > 0) {
      Result = Result * X;
   }

   return Result;
}
```





©2005-2020 WD McQuain

Code Organization for User-defined Type


```
// Type definition typically goes
// in a header file (Rational.h).
struct _Rational {
  int top;
  int bottom;
};
typedef struct _Rational Rational;

// Declarations of "public" functions go in header file.
Rational Rational_Create(int top, int bottom);
Rational Rational_Add(Rational left, Rational right);
. . .
```

A "public" function is one that we intend to call from code that is in other `.c` files.

Any interesting C program will consist of multiple `.h` and `.c` files.

©2005-2020 WD McQuain

Code Organization for User-defined Type

```
// Helper functions that are only intended to be called from
// within a single .c file are typically declared and defined
// within that .c file.
//
// Declaring the function as "static" makes it callable
// only from within this file.

static Rational Rational_Normalize(Rational original);

Rational Rational_Create(int top, int bottom) {
   . . .
}
. . .

static Rational Rational_Normalize(Rational original) {
    . . .
}
. . .
```


©2005-2020 WD McQuain

Function Header Comments

Each function should be given a header comment that supplies the caller with enough information to understand what the function does, what pre-conditions are necessary for a successful call, what post-conditions are guaranteed (if appropriate), what value is returned (if appropriate).

For development purposes, it's also useful to specify callers and callees.

```
/**
 *   Computes the sum of left and right.
 *   Pre:
 *        Left and Right are proper.
 *   Returns:
 *        A proper Rational object X equal to left + right.
 *   Called by:
 *        Client code.
 *   Calls:
 *        Rational_Normalize() on result.
 */
Rational Rational_Add(Rational left, Rational right);
```

It's good practice to put the header comment on both the function declaration and definition.

©2005-2020 WD McQuain

Shell Side to C

C programs can receive command-line arguments from the shell:

```
#775 wmcquain: T06> hexer Virginia Polytechnic Institute

    Virginia:   56 69 72 67 69 6E 69 61
 Polytechnic:   50 6F 6C 79 74 65 63 68 6E 69 63
   Institute:   49 6E 73 74 69 74 75 74 65
. . .
```

The shell initializes an integer variable and an array of C-style strings:

```
int argc:   4
```

```
                  +----------------+
char* argv[]:     | "hexer"        |
                  +----------------+
                  | "Virginia"     |
                  +----------------+
                  | "Polytechnic"  |
                  +----------------+
                  | "Institute"    |
                  +----------------+
                  | NULL           |
                  +----------------+
```


©2005-2020 WD McQuain

main() Interface

These arguments are passed as parameters to `main()`:

```
int argc:   4
```

```
char* argv[]:   +----------------+
                | "hexer"        |
                +----------------+
                | "Virginia"     |
                +----------------+
                | "Polytechnic"  |
                +----------------+
                | "Institute"    |
                +----------------+
                | NULL           |
                +----------------+
```

```
// hexer.c
. . .
int main(int argc, char* argv[]) {
   . . .
```

So, the C program can now check the number of command-line "tokens" and process them as needed.

 ©2005-2020 WD McQuain

C Code


```
// hexer.c
#include <stdio.h>

int main(int argc, char* argv[]) {

   int argno = 1;    // start with argument 1

   while ( argv[argno] != NULL ) {

      char* currArg = argv[argno];  // slap handle on current one

      // echo current argument
      printf("%10s:  ", argv[argno]);

      // print ASCII codes of characters, in hex format:
      int pos = 0;
      while ( currArg[pos] != '\0' ) {
         printf(" %X", (unsigned char) currArg[pos]);
         pos++;
      }
      printf("\n");

      argno++;    // step to next argument (if any)
   }
   return 0;
}
```


©2005-2020 WD McQuain

The Stack

The execution of a C program is organized by use of a collection of *stack frames* (or *activation records*) stored in a stack structure.

Each time a function is called, a stack frame is created and pushed onto the stack.

The stack frame provides memory for storing:

- values of parameters passed into the function
- values of local variables declared within the function (unless they're **`static`**)
- the return address (of the instruction to be executed when the function terminates)

We will examine the details of the stack later.

©2005-2020 WD McQuain