Functions

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 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.

Arguments

- The fact that arguments are passed by value has both advantages and disadvantages.

- Since a parameter can be modified without affecting the corresponding argument, we can use parameters as variables within the function, reducing the number of genuine variables needed.

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 requirement that arguments be passed by value makes it difficult to write certain kinds of functions.
- Suppose that we need a function that will decompose a `double` value into an integer part and a fractional part.
- Since a function can’t *return* two numbers, we might try passing a pair of variables to the function and having it modify them:

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


Arguments

- A call of the function:

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

- Unfortunately, `i` and `d` won’t be affected by the assignments to `int_part` and `frac_part`.

- We’ll solve this later in the semester.

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`.

Argument Conversions

- Relying on the default argument promotions is dangerous.
- Example:

```
#include <stdio.h>

int main(void)
{
  double x = 3.0;
  printf("Square: %d\n", square(x));

  return 0;
}

int square(int n)
{
  return n * n;
}
```

- At the time `square` is called, the compiler doesn't know that it expects an argument of type `int`.

Argument Conversions

- Instead, the compiler performs the default argument promotions on `x`, with no effect.
- Since it's expecting an argument of type `int` but has been given a `double` value instead, the effect of calling `square` is undefined.
- The problem can be fixed by casting `square`'s argument to the proper type:

```
printf("Square: %d\n", square((int) x));
```

- A much better solution is to provide a prototype for `square` before calling it.
- In C99, calling `square` without first providing a declaration or definition of the function is an error.

Array Arguments

- 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.
- Instead, we'll have to supply the length – if the function needs it – 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 must supply it as a second argument.

17

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);
```

18

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 telling the function that the array is smaller than it really is.
- Suppose that we've only stored 50 numbers in the `b` array, even though it can hold 100.
- We can sum just the first 50 elements by writing

```
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.

Array Arguments

- 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.
- 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)
{
  …
}
```

35

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()
{
  …
}
```

36

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 */
```

39

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.

40

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.

41