Functions

Outline

1. Basic Concepts
   - Understanding what a function is
   - Calling a function and tracing a function call
   - Characteristics of the variables declared inside a function

2. Parameters
   - Defining a function with parameters

3. Returned value
   - Defining a function with a return value
   - The `return` keyword

1.1. What is a Function?

- A *function* is a group of statements that together perform a task.
  - It is also known as a *procedure* or a *subroutine* in other programming languages

- A C program is made up of one or more functions.
  - e.g., `main()`, `printf()`, `scanf()`, `sqrt()`
  - `main()` is the starting point of a program.

Note: In the lecture notes, we use the notation `foobar()` to mean "a function named **`foobar`**".

1.2. A Simple Function

```
1  #include <stdio.h>
2
3  void greet() {
4      printf("Hi! How are you?\n");
5  }
6
7  int main() {
8
9      greet();
10
11     return 0;
12 }
13
```

```
Hi! How are you?
```

1.3. Function Name

```
1  #include <stdio.h>
2
3  void greet() {
4      printf("Hi! How are you?\n");
5  }
6
7  int main() {
8
9      greet();
10
11     return 0;
12 }
13
```

A function has a name.

A function named "greet"

A function named "main"

main() is also the starting point of a C program.

```
Hi! How are you?
```

1.4. Calling/Invoking A Function

```
1  #include <stdio.h>
2
3  void greet() {
4      printf("Hi! How are you?\n");
5  }
6
7  int main() {
8
9      greet();
10
11     return 0;
12 }
13
```

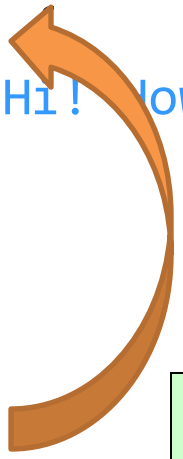
We *call* (or *invoke*) a function by the function's name, followed by a pair of parentheses.

Calling a function ➔ executing the code in that function.

```
Hi! How are you?
```

1.5. Terminology: *Caller* and *Callee*

```
1  #include <stdio.h>
2
3  void greet() {
4      printf("Hi! How are you?\n");
5  }
6
7  int main() {
8
9      greet();
10
11     return 0;
12 }
13
```



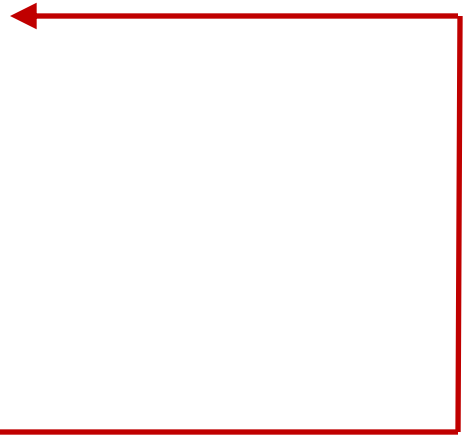
Function call

At line 9, `main()` initiates the ***function call***, and `greet()` is being called. In this situation, we say `main()` is the *caller*, and `greet()` is the *callee*.

```
Hi! How are you?
```

1.6. Control Flow During a Function Call – Part 1

```
1  #include <stdio.h>
2
3  void greet() {
4      printf("Hi! How are you?\n");
5  }
6
7  int main() {
8
9      greet();
10
11     return 0;
12 }
13
```



When `greet()` is called at line 9 during program execution, control is "transferred" to the beginning of `greet()`.

```
Hi! How are you?
```

1.6. Control Flow During a Function Call – Part 2

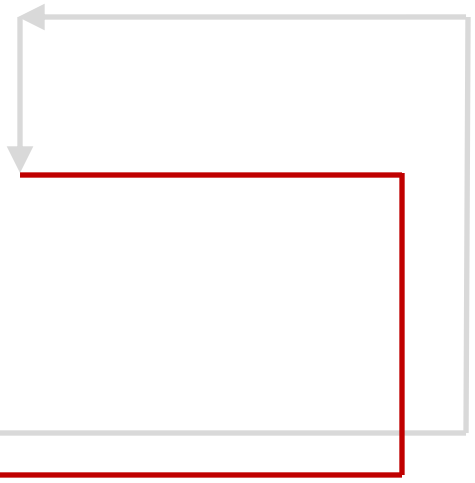
```
1   #include <stdio.h>
2
3   void greet() {
4       printf("Hi! How are you?\n");
5   }
6
7   int main() {
8
9       greet();
10
11      return 0;
12  }
13
```

Statement in `greet()` is executed.

```
Hi! How are you?
```

1.6. Control Flow During a Function Call – Part 3

```
1   #include <stdio.h>
2
3   void greet() {
4       printf("Hi! How are you?\n");
5   }
6
7   int main() {
8
9       greet();
10
11      return 0;
12  }
13
```



When execution is completed in `greet()`, control is returned to the location where the function is called.

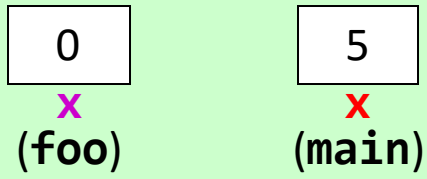
```
Hi! How are you?
```

1.7. Variables declared in a function are local to that function

```
1  void foo() {
2      int x = 0;
3      printf("In foo(): x = %d\n", x);
4  }
5
6
7
8
9  int main() {
10     int x = 5;
11     printf("Before: In main(): x = %d\n", x);
12     foo();
13     printf("After: In main(): x = %d\n", x);
14     return 0;
15 }
```

Every function has its own variables.

x in foo() and x in main() are two different variables.




```
Before: In main(): x = 5
In foo(): x = 0
After: In main(): x = 5
```

1.7. Variables declared in one function are not accessible by another function

```
1   void bar() {
2       int y = 0;
3       printf("In bar(): y = %d\n", y);
4   }
5
6   int main() {
7       bar();
8       printf("%d\n", y); /* Compile-time error */
9       return 0;
10  }
```

Variables declared in a function are *local variables* and are only accessible inside that function.

`y`, being declared in `bar()`, is not accessible in `main()`.

2. Parameters

- Parameters are for passing data from a caller to a callee.

- Proper use of parameters allows programmers to reuse code for different data.
  - e.g., `printf()` can be used to print many kinds of data in different formats.

13

2.1. Formal and Actual Parameters

```
1  #include <stdio.h>
2
3  void foo(int n) {
4      printf("%d\n", n);
5  }
6
7  int main() {
8      int x = 10;
9
10     foo(3);
11     foo(x);
12     foo(x + 3);
13
14     return 0;
15 }
```

Variables for holding the values passed to a function are called *formal parameters*.

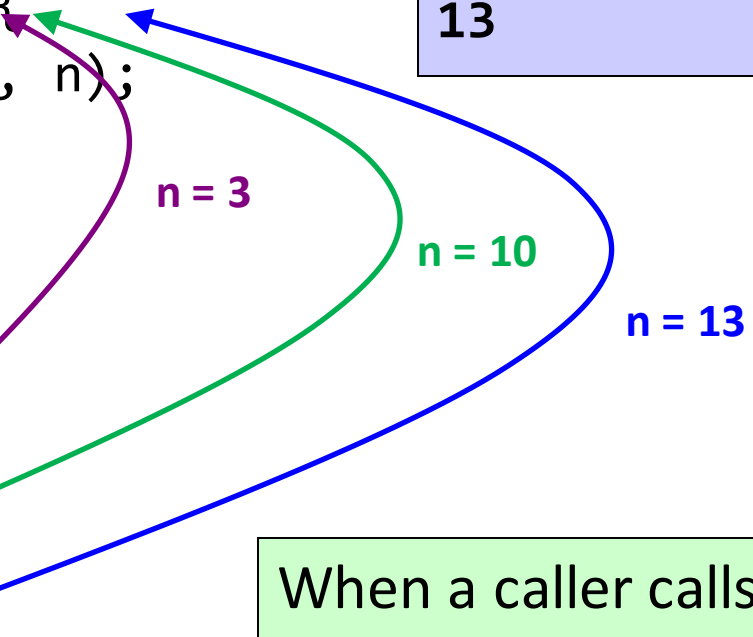
They have local scope in the function.

The expressions specified in the function calls are called *actual parameters* or *arguments*.

2.2. How values are passed to a function

```
1  #include <stdio.h>
2
3  void foo(int n) {
4      printf("%d\n", n);
5  }
6
7  int main() {
8      int x = 10;
9
10     foo(3);
11     foo(x);
12     foo(x + 3);
13
14     return 0;
15 }
```

```
3
10
13
```



When a caller calls a function with parameters, the values of the arguments are copied to the formal parameters.

2.3. Defining a function with parameters (*Syntax*)

```
void function_name( parameter_list ) {

    // declarations and statements
}
```

- `parameter_list`
  - Zero or more parameters separated by comma in the form

    $\texttt{type}_1\ \texttt{param}_1, \texttt{type}_2\ \texttt{param}_2, \ldots, \texttt{type}_N\ \texttt{param}_N$

2.4. Example

- Design and implement a function that can be used to print any character N times.

2.4. Example

```
1  /* A function that prints the character, ch, n times */
2  void printChars(int n, char ch) {
3      int i;
4      for (i = 0; i < n; i++) {
5          printf("%c", ch);
6      }
7      printf("\n");
8  }
9
10 int main() {
11     printChars(17, '#');
12     printf("  Hello World!\n");
13     printChars(17, '*');
14     return 0;
15 }
```

```
#################
  Hello World!
*****************
```

2.4. Example

```
1  /* A function that prints the character, ch, n times */
2  void printChars(int n, char ch) {
3      int i;
4      for (i = 0; i < n; i++) {
5          printf("%c", ch);
6      }
7      printf("\n");
8  }
9
10 int main() {
11     printChars(17, '#');
12     printf("  Hello World!\n");
13     printChars(17, '*');
14     return 0;
15 }
```

%c is the format specifier for printing character.

Character constant is enclosed by a pair of single quotes (').

2.5. Parameters are matched by position

```
1   void foo(int x, int y) {
2       printf("%d %d\n", x, y);
3   }
4
5   int main() {
6       int x = 3, y = 2;
7       foo(x, y);
8       foo(y, x);
9       return 0;
10  }
```

What is the output?

- Arguments and formal parameters are matched by position (not by names and not by types).

2.6. Common Mistakes

```
1  void foo(int x, int y) {  /* Correct */
2    …
3  }
```

```
1  void foo(int x, y) {      /* Incorrect */
2    …
3  }
```

- We need to clearly specify the data type for every parameter even if multiple parameters are of the same type.

3. Return Value

- How to return a value from a function to its caller?

- How does the `return` keyword affect the execution flow inside a function?

3.1. Returning a Value From a Function

A function can return a value to its caller.

```
1   #include <stdio.h>
2
3   int cube(int x) {
4       int y;
5       y = x * x * x;
6       return y;
7   }
8
9   int main() {
10      int result;
11
12      result = cube(3);
13      printf( "Cube of 3 is %d\n", result);
14      return 0;
15  }
```

```
Cube of 3 is 27
```

3.1. Returning a Value From a Function

```
1  #include <stdio.h>
2
3  int cube(int x) {
4      int y;
5      y = x * x * x;
6      return y;
7  }
8
9  int main() {
10     int result;
11
12     result = cube(3);
13     printf( "Cube of 3 is %d\n", result);
14     return 0;
15 }
```

`int` indicates that `cube( )` will return a value of type `int` when the function finishes its execution.

`return` is a keyword that is used to specify the value to be returned to the caller. In this example, the value of `y` is returned.

```
Cube of 3 is 27
```

3.2. Defining a function that returns a value (Syntax)

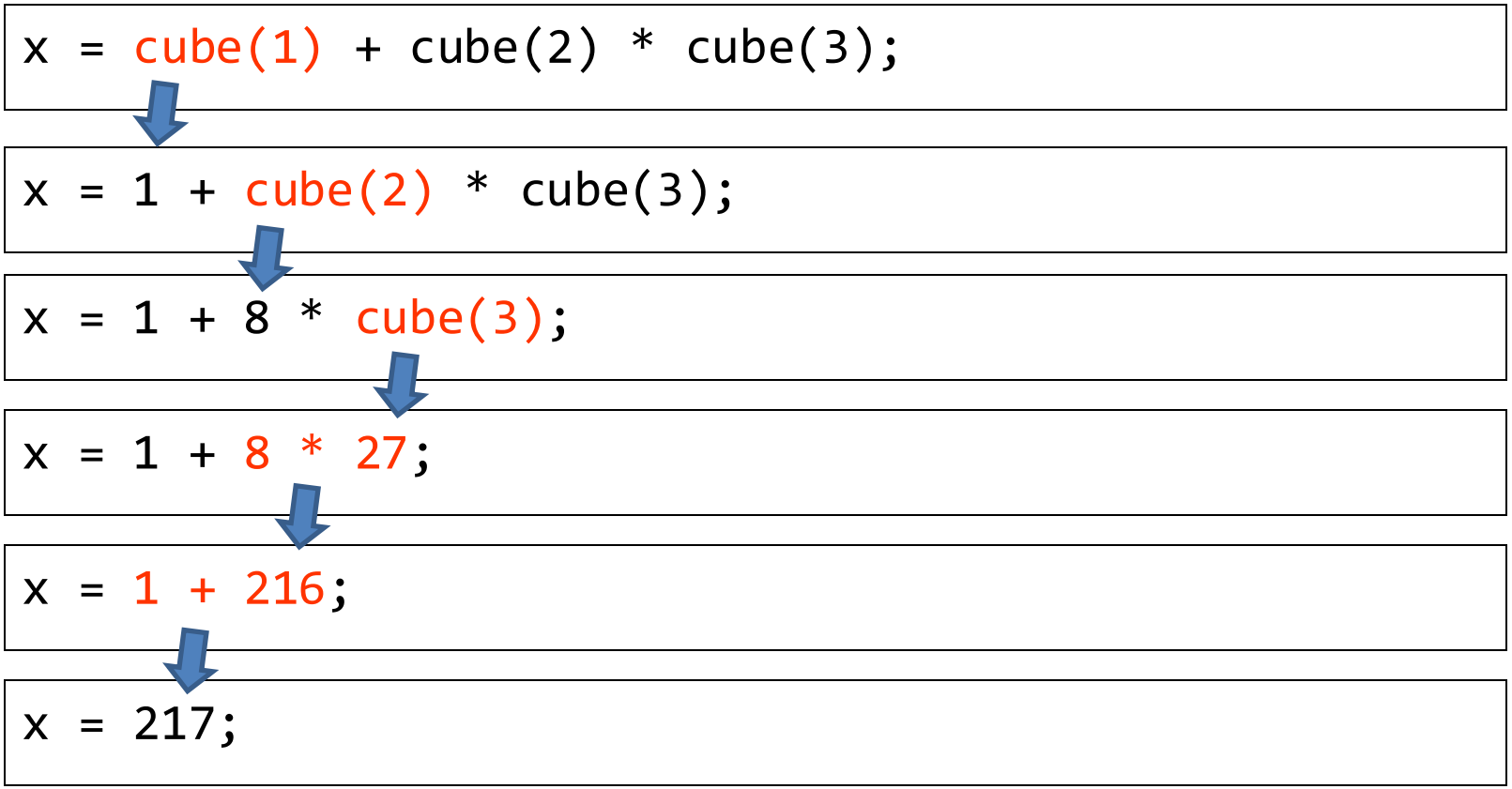
```
return_type function_name( parameter_list ) {
    …
    return expression;
}
```

- `return_type`
  - Data type of data to be returned by the function
  - Use `void` if a function does not return a value

- `return expression`
  - `return` is a keyword.
  - When this statement is executed, the value of `expression` is returned to the caller.

3.3. Evaluating an expression containing function calls

Functions are called first if they are part of an expression.



Note: some compilers call the functions from right to left.

4. Interrupting Control Flow with `return`

A **`return`** statement can also force an execution to leave a function and return to its caller immediately.

```
int min(int x, int y) {

    if (x > y)
        return y;

    return x;
}
```

When "`return y`" is executed, execution immediately stops in `min()` and resumes at its caller.

In this example, if "`x > y`" is true, "`return x`" will not be executed.

4.1. Example

- Implement a function that accepts a month and a year as parameters, and returns the number of days in the given month and year.

4.1. Example (with multiple **return**'s)

```
1   /* Returns # of days in a particular month */
2   int daysPerMonth(int m, int y) {
3      if (m == 1 || m == 3 || m == 5 ||
4          m == 7 || m == 8 || m == 10 || m == 12)
5         return 31;
6
7      if (m == 4 || m == 6 || m == 9 || m == 11)
8         return 30;
9
10     /* if y is a leap year */
11     if (…)
12        return 29;
13
14    return 28;
15  }
```

Only one of the "return" statements will be executed.

4.1. Example (with only one **return**)

```
1   /* Returns # of days in a particular month */
2   int daysPerMonth(int m, int y) {
3       int days;
4       if (m == 1 || m == 3 || m == 5 ||
5           m == 7 || m == 8 || m = 10 || m = 12)
6           days = 31;
7       else
8       if (m == 4 || m == 6 || m == 9 || m = 11)
9           days = 30;
10      else
11      if (…) /* if y is a leap year */
12          days = 29;
13      else
14          days = 28;
15      return days;
16  }
```

A function is easier to debug if there is only one **return** statement because we know exactly where an execution leaves the function.

4.2. Using `return` without a returning value

- When the function return type is `void`, we can use `return` without a return value.

```
void askSomething( int code ) {
    if (code != 7) {
        printf("Who are you?\n");
        return;   /* Leave the function immediately */
    }

    printf("How are you today, James?\n");

    return;   /* This return statement is optional */
}
```

If the return type is **`void`**, placing a return as the last statement is optional (it is implied).

4.3. Additional info about returning a value

- A function can return only one value of a specific data type.

- If a function's return type is not **`void`**, then all paths leaving the function must return a value that matches the return type.

```
double reciprocal(double x) {
    if (x != 0.0)
        return 1.0 / x;
}
```

If x is 0.0, then an undefined value is returned.

Compiler may warn.

5. Function Prototypes

- Also known as *function declarations*
- Why do we need function prototypes?
- How to define function prototypes?

```
1   #include <stdio.h>
2
3
4   int main() {
5       printf( "%d\n", square( 4 ) );
6       return 0;
7   }
8
9   /* Function definition */
10  int square( int y ) {
11      return y * y;
12  }
```

Compile-time warning: undefined '**square**'; assume returning **int**. Why?

- A C compiler performs a 1-pass sequential scan of the source code during compilation. By the time the compiler encounters the identifier "**square**" at line 5, it does not know "**square**" is a function defined at line 10.

```c
#include <stdio.h>

/* Function definition */
int square( int y ) {
    return y * y;
}

int main() {
    printf( "%d\n", square( 4 ) );
    return 0;
}

```

- We could rearrange the functions so that all callees are defined before their callers, but such approach is not always possible.

- A better solution is to declare function prototypes.

```c
#include <stdio.h>

/* Function prototype */
int square( int );

int main() {
    printf( "%d\n", square( 4 ) );
    return 0;
}

/* Function definition */
int square( int y ) {
    return y * y;
}
```

Tells a compiler that:

- **square** is a function name.
- The function takes an argument of type **int**.
- The function returns a value of type **int**.

A function prototype provides a compiler info about a function.

5.1. Function Prototypes: When to use them and why?

```
void foo(void);
void bar(void);

int main() {
  foo();
}

void foo() {
  if (…)
    bar();
}

void bar() {
  if (…)
    foo();
}
```

- When a callee is defined after its caller in the same file
- When a callee and its caller are defined in separate source files
  - Common in large software project

5.2. Function Prototypes (Syntax)

- Function prototype is like a function definition but without the body.

Function *definition*

```
double foo( int x, double y, int A[], int B[][64] ) {
    …
}
```

Function *prototype*

```
double foo( int x, double y, int A[], int B[][64] );
```

or

```
double foo( int, double, char, int [], int [][64]);
```

- Parameter names are optional in function prototypes.
- Function name, return type, and parameter types must match between a function definition and its function prototype.

6. Calling Pre-defined Functions

- C language provides many built-in functions. To use them, you have to know the following info (which can be found in manuals):
  - name, functionality, parameters, return value

- You also need to know which *header file(s)* to include.

  e.g.:
  - To use `printf(…)`, you have to include "stdio.h" with
    ```
    #include <stdio.h>
    ```
  - To use math functions, you have to include "math.h" with
    ```
    #include <math.h>
    ```

Summary

- Understand what "functions", "parameters/arguments", "return value/type" are
- Understand what is happening during a function call
- Know how to define and call a function
- Understand how a return statement can interrupt the flow of execution in a function
- Understand why we need function prototypes and how to declare them

Reading Assignment

- C: How to Program, 8th ed, Deitel and Deitel
- Chapter 5 C Functions
  - Sections 5.1 – 5.5: Function Basics and examples
  - Sections 5.6: Function Prototypes