Functions

Learning Outcomes

After reading this section, you will be able to:

- Design procedures using selection and iteration constructs to solve a programming task
- Connect procedures using pass-by-value semantics to build a complete program
- Trace the execution of a complete program to validate its correctness

Introduction

Procedural programming involves separating source code into self-contained components that can be accessed multiple times from different locations in a complete program. This approach enables separate coding of each component and assembly of various components into a complete program. We call this approach to programming solutions modular design.

This chapter introduces the principles of modular design, describes the syntax for defining a module in the C language, shows how to pass data from one module to another, suggests a walkthrough table structure for programs composed of several modules and includes an example that validates user input.

Modular Design

Modular design identifies the components of a programming project that can be developed separately. Each module consists of a set of logical constructs that are related to one another. A module may refer to other modules. A trivial example is the program described in the chapter on compilers:

```
/* My first program
   hello.c          */

#include <stdio.h>          // information about the printf identifier
```

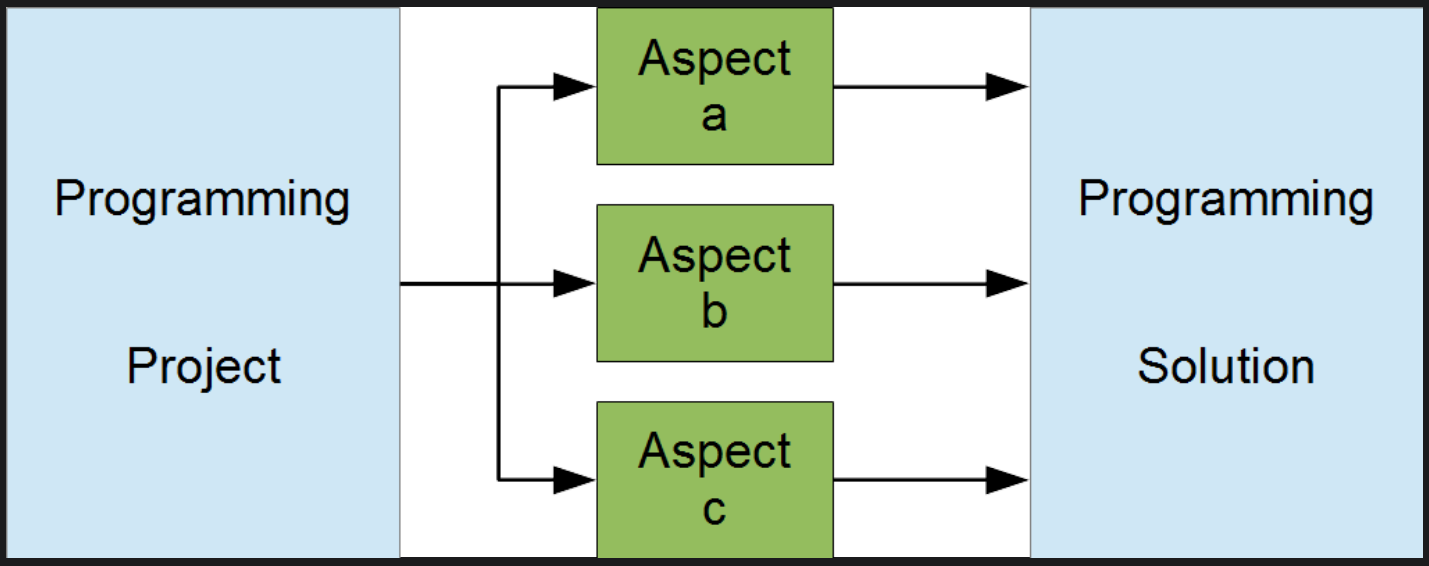
```c
int main(void)              // program startup
{
    printf("This is C"); // output

    return 0;            // return to operating system
}
```

The module named `hello.c` starts executing at statement `int main(void)`, outputs a string literal and returns control to the operating system.

- `main()` transfers control to a module named `printf()`
- `printf()` executes the detailed instructions for outputting the string literal
- `printf()` returns control to `main()`

Design Principles

We can sub-divide a programming project in different ways. We select our modules so that each one focuses on a narrower aspect of the project. Our objective is to define a set of modules that simplifies the complexity of the original problem.



Some general guidelines for defining a module include:

1. the module is easy to upgrade
2. the module contains a readable amount of code
3. the module may be used as part of the solution to some other problem

For a structured design, we stipulate that:

1. each module has one entry point and one exit point
2. each module is highly cohesive
3. each module exhibits low coupling

Cohesion

Cohesion describes the focus: a highly cohesive module performs a single task and only that task.

In creating a cohesive module, we ask whether our tasks belong to that module: a reason to include a task is its relation to the other tasks within the module. A reason to exclude a task is its independence from other tasks within the module.

For example, the following tasks are related:

- receives a date and the number of days to add
- converts the date into a format for adding days
- adds the number of days received
- converts the result to a new date
- returns the new date

The following tasks are unrelated:

- calculates Federal tax on bi-weekly payroll
- calculates the value of π
- outputs an integer in hexadecimal format

We allocate unrelated tasks to separate modules in the program design.

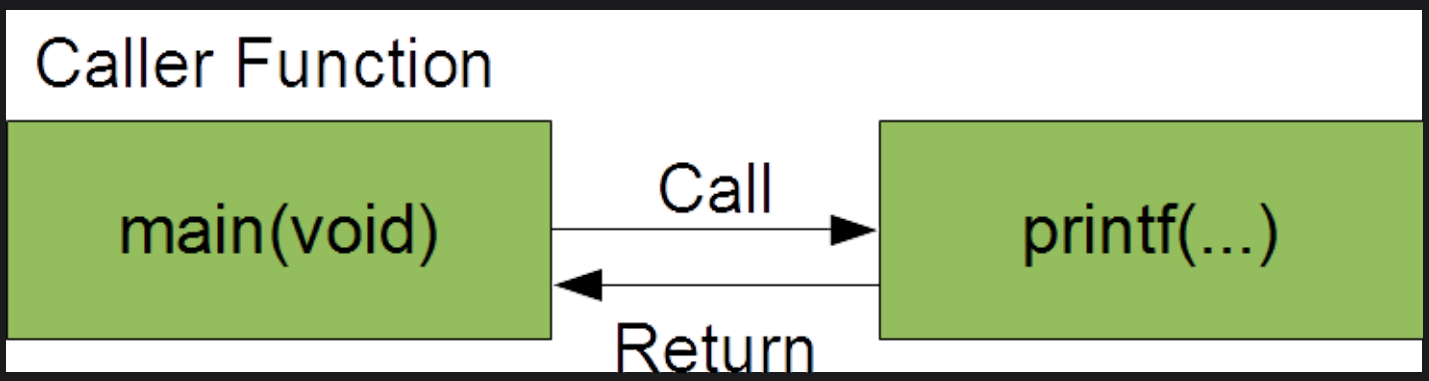
Coupling

Coupling describes the degree of interrelatedness of a module with other modules. The less information that passes between the module and the other modules the better the design. We prefer designs in which each module completely control its own computations and avoids transferring control data to any other module.

Consider a module that receives a flag from another module and performs a calculation based on that flag. Such a module is highly coupled: another module controls its execution. To improve our design, we transfer data to the module and let it create its own flags before completing its task.

Functions

The C language is a procedural programming language. It supports modular design through function syntax. Functions transfer control between one another. When a function transfers control to another function, we say that it ***calls*** the other function. Once the other function completes its task and transfers control to the caller function, we say that that other function ***returns*** control to its ***caller***.



In the example from the introductory chapter on compilers listed above:

1. the `main()` function calls the `printf()` function
2. the `printf()` function outputs the string
3. the `printf()` function returns control to its caller `main()`

Definition

A function consists of a header and a body. The body is the code block that contains the detailed instructions to be performed by the function. The header immediately precedes the body and includes the following ***in order***:

1. the type of the function's return value
2. the function's identifier
3. a parentheses-enclosed list of parameters that receive data from the caller

```
type identifier(type parameter, ..., type parameter)
{
   // function instructions

   return x; // x denotes the value returned by this function
}
```

`type` specifies the type of the return value or the function's parameter, while the `identifier` specifies the name of the function, and `parameter` is a variable that holds data received from the caller function.

For example:

```
/* Raise an integer to an integer
 * power.c
 */

#include <stdio.h>

int power(int base, int exponent)
{
    int i, result;

    result = 1;
    for (i = 0; i < exponent; i++)
        result = result * base;

    return result;
}

int main(void)
{
    int base, exp, answer;

    printf("Enter base : ");
    scanf("%d", &base);
    printf("Enter exponent : ");
    scanf("%d", &exp);

    answer = power(base, exp);
```

```
    printf("%d^%d = %d\n", base, exp, answer);
}
```

Outputs the following:

```
Enter base : 3
Enter exponent : 4
3^4 = 81
```

The first function returns a value of `int` type, while `power` identifies the function, and `base` and `exponent` are the function's parameters; both are of `int` type.

Special Cases

void Functions

A function that does not have to return any value has no return type. We declare its return type as `void` and exclude any expression from the return statement.
For example:

```
void countDown(int n)
{
    while (n > 0)
    {
        printf("%d ", n);
        n--;
    }

    return; // optional
}
```

> **NOTE**
> In such cases, the return statement is **optional** and is usually not included.

No Parameters

A function that does not have to receive any data does not require parameters. We insert the keyword `void` between the parentheses. For example:

```
void alphabet(void)
{
    char letter = 'A';

    do {
            printf("%d ", letter);
            letter++;
    } while (letter != 'Z');
}
```

> **NOTE**
>
> The iteration changes `letter` to the next character in the alphabet, assuming the collating sequence arranged them contiguously.

main

The `main()` function is a function itself. It is the function to which the operating system transfers control after loading the program into RAM.

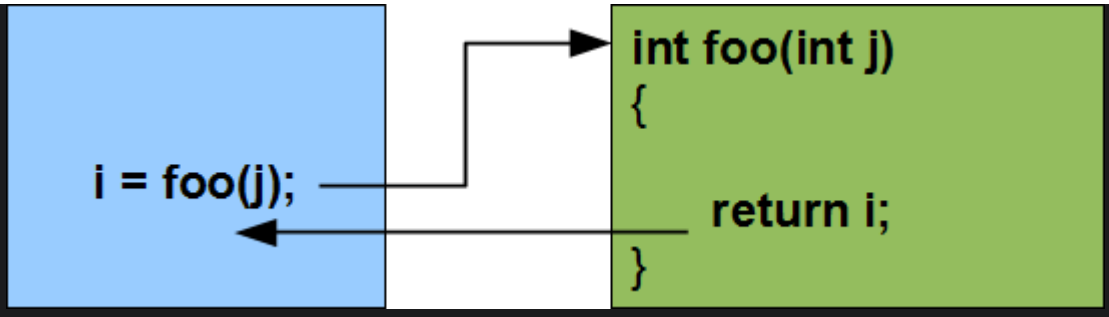
`main()` returns a value of `int` type to the operating system once it has completed execution. A value of 0 indicates success to the operating system.

Function Calls

A function call transfers control from the caller to function being called. Once the function being called has executed its instructions, it returns control to the caller. Execution continues at the point immediately following the call statement. A function call takes the form:

```
identifier(argument, ..., argument)
```

`identifer` specifies the function being called, while `argument` specifies a value being passed to the function being called.



An argument may be a constant, a variable, or an expression (with certain exceptions). The number of arguments in a function call should match the number of parameters in the function header.

Pass By Value

The C language passes data from a caller to a function by value. That is, it passes a copy of the value and not the value itself. The value passed is stored as the inital value is the parameters that corresponds to the argument in the function call.

Each parameter is a variable with its own memory location. We refer to the mechanism of allocating separate memory locations for parameters and using the arguments in the function call to initialize these parameters as ***pass by value***. Pass by value facilitates modular design by localizing consequences. The function being called may change the value of any of its parameters many times, but the values of the corresponding arguments in the caller remain unchanged. In other words, a function cannot change the value of an argument in the call to the function. This language feature ensures the variables in the caller are relatively secure.

Mixing Types

If the type of an argument does not match the type of the corresponding parameter, the compiler coerces (narrows or promotes) the value of the argument into a value of the type of the corresponding parameter. Consider the `power` function listed above and the following call to it:

```
int answer;
answer = power(2.5, 4);
```

The compiler converts the first argument into a value of type `int`

```
int answer;
```

```
answer = power((int)2.5, 4);
```

The C compiler evaluates the cast (coercion) of 2.5 before passing the value of type `int` to `power` and initializing the parameter to the cast result (2).

Walkthroughs

The structure of a walkthrough table for a modular program is a simple extension of the structure of the walkthrough table shown in the chapter entitled Testing and Debugging. The table for a modular program groups the variables under their parent functions.

int			type		
main(void)			--function identifier here--		
type	...	type	type	...	type
variable z	...	variable a	variable z	...	variable a
&z	...	&a	&z	...	&a
initial value	...	initial value	initial value	...	initial value
next value	...	next value	next value	...	next value
next value	...	next value	next value	...	next value
			next value	...	next value
initial value	...	initial value	initial value	...	initial value
next value	...	next value	next value	...	next value
next value	...	next value	next value	...	next value

int			type		
main(void)			--function identifier here--		
			next value	...	next value
			next value	...	next value

Output:

(record output here line by line)

Example

The completed walkthrough table for the power.c program listed above is shown below:

- **& represents the address.**

int			int			
main(void)			power(int base, int exponent)			
int	int	int	int	int	int	int
base	exp	answer	base	exponent	result	i
&100	&104	&108	&10C	&110	&114	&118
3						
3	4					
3	4		3	4		
3	4		3	4	1	

int			int			
main(void)			power(int base, int exponent)			
3	4		3	4	1	0
3	4		3	4	3	0
3	4		3	4	3	1
3	4		3	4	9	1
3	4		3	4	9	2
3	4		3	4	27	2
3	4		3	4	27	3
3	4		3	4	81	3
3	4		3	4	81	4
3	4	81				

> **NOTE**
>
> Each parameter occupies a memory location that is distinct from any other location in the caller. For example, the parameter `base` in `power()` occupies a different memory location than the variable `base` in `main()`.

Validation (optional)

Ensuring that programming assumptions are not breached by the user is part of good program design. Our function to raise an integer base to the power of an exponent is based on the assumption that the exponent is non-negative. Accordingly, we need to validate the user input to ensure that our assumption holds.

Let us introduce a function named `getNonNegInt()` that only accepts non negative integer values from the user:

```c
/* Raise an Integer to the Power of an Integer
 * power.c
 */

#include <stdio.h>

// getNonNegInt returns a non-negative integer
//
// getNonNegInt assumes that the user enters only
// integer values and no trailing characters
//
int getNonNegInt(void)
{
    int value;

    do {
            printf(" Non-negative : ");
            scanf("%d", &value);
            if (value < 0)
                    printf(" * Negative! *\n");
    } while(value <= 0);

    return value;
}

// power returns the value of base raised to
//  the power of exponent (base^exponent)
//
//  power assumes that base and exponent are
//  integer values and exponent is non-negative
//
int power(int base, int exponent)
{
    int result, i;

    result = 1;
    for (i = 0; i < exponent; i++)
            result = result * base;

    return result;
```

```
}

int main(void)
{
    int base, exp, answer;

    printf("Enter base : ");
    scanf("%d", &base);

    printf("Enter exponent\n");
    exp = getNonNegInt();
    answer = power(base, exp);

    printf("%d^%d is %d\n", base, exp, answer);

    return 0;
}
```

Code Output

```
Non-negative : -2
* Negative! * //error message from do-while
Non-negative : 4

Enter base : 3
Enter exponent //exponent becomes "4"

3^4 is 81
```

Walkthrough

The table below lists the values of the local variables in this source file at different stages of execution:

int			int				int
main(void)			power(int base, int exponent)				getNonNegInt(void)
int	int	int	int	int	int	int	int
base	exp	answer	base	exponent	result	i	value
?	?	?					
3	?	?					
3	?	?					-2
3	?	?					4
3	4	?					
3	4	?	3	4	?	?	
3	4	?	3	4	1	?	
3	4	?	3	4	1	0	
3	4	?	3	4	3	0	
3	4	?	3	4	3	1	
3	4	?	3	4	9	1	
3	4	?	3	4	9	2	
3	4	?	3	4	27	2	
3	4	?	3	4	27	3	
3	4	?	3	4	81	3	

int			int				int
main(void)			power(int base, int exponent)				getNonNegInt(void)
3	4	?	3	4	81	4	
3	4	81					

The shaded areas show the stages in their lifetimes at which the variables are visible. The unshaded areas identify the stages at which the variables are out of scope of the function that has control. The values marked ? are undefined.