### Chapter 4

### Functions



### **OBJECTIVES**

### After studying this chapter you will be able to:

- Structure a program into functions designed to do a specific task.
- Understand the relationship between parameters and arguments
- Understand the concept of local variables.
- Pass data to a function using pass-by-value or pass-by-reference.
- ☐ Use a structure chart to design a program.
- Write function prototype statements.
- Write programs with multiple functions.
- Use mathematical and standard library functions in programs.
- Understand the concept of scope in a C++ program.
- Understand the basic concept of functional cohesion.
- Implement a program using top-down development and stubs.



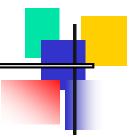
## DESIGNING STRUCTURED PROGRAMS

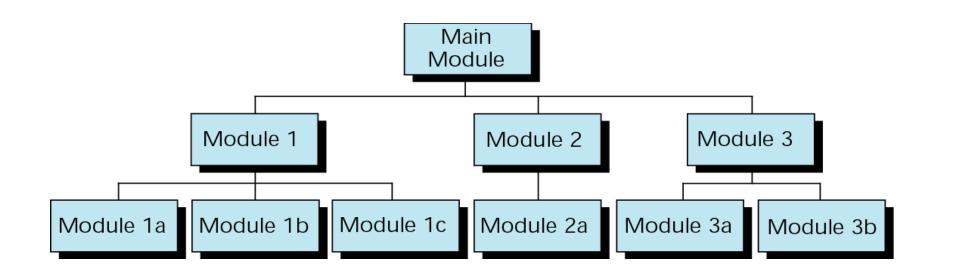


In top-down design, a program is divided into a main module and its related modules. Each module is in turn divided into submodules until the resulting modules are intrinsic; that is, until they are implicitly understood without further division.



### Figure 4-1 Structure chart







In a structure chart, a module can be called by one and only one higher module.



4.2

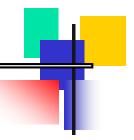
# FUNCTIONS IN C++

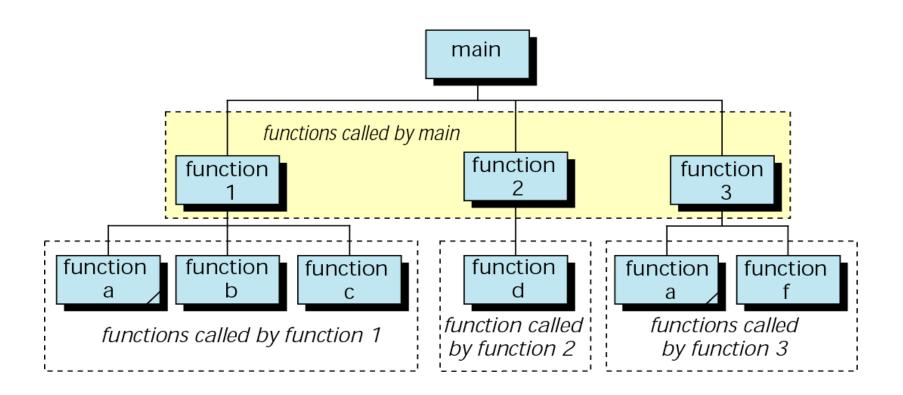


In C++, a program is made of one or more functions, one and only one of which must be named main. The execution of the program always starts with main, but it can call other functions to do some part of the job.



### Figure 4-2 Structure chart for a C++ program







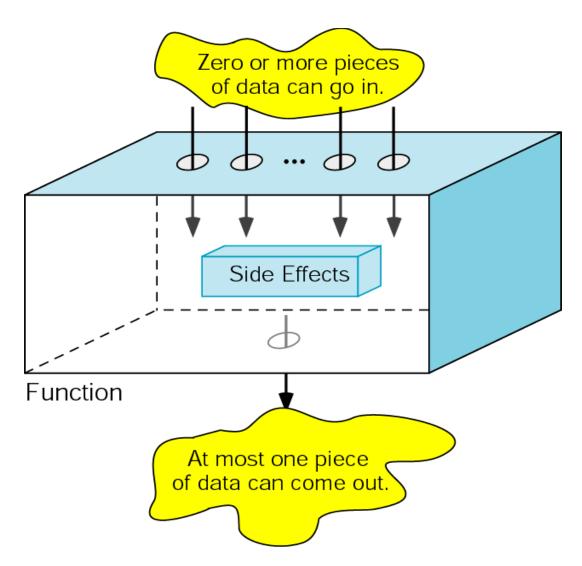
A function in C++ can have a value, a side effect, or both.

- The side effect occurs before the value is returned.
- The function's value is the value of the expression in the return statement.
- A function can be called for its value, its side effect, or both.



### Figure 4-3 Function concept





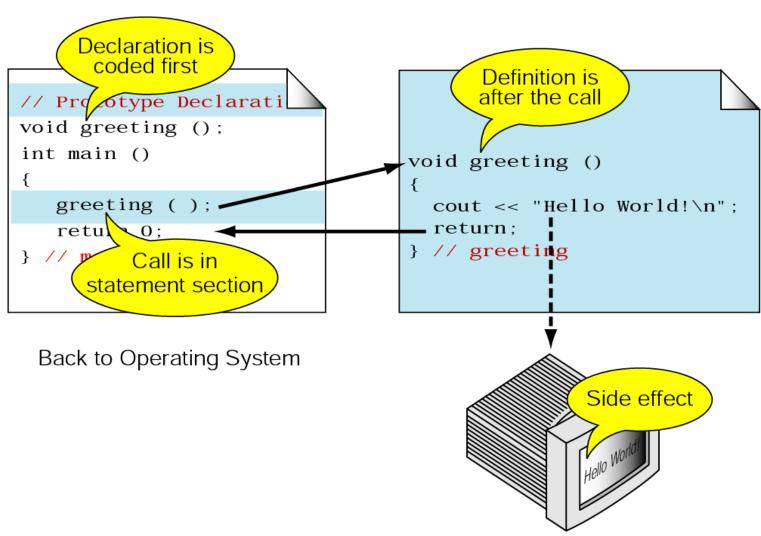


### USER-DEFINED FUNCTIONS



### Figure 4-4 Declaring, calling, and defining functions

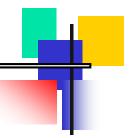




The name of a function is used in three ways: for declaration, in a call, and for definition.



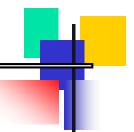
### Figure 4-5 *void* function with parameters



```
// Prototype Declaratio
                                      а
void printOne (int x);
                                      5
int main ()
                  Declaration
   int a = 5;
   printOne (a);
   retui a:
                                Value Copied
         Call
void printOne (int
                                                   Side Effect
   cout << x << end1;
                                      Χ
   return,;
} // print
         Nothing is returned
        to the calling function
```



### Figure 4-6 Calling a function that returns a value



```
// Prototype Declarations
   int sqr (int x);
   int main ()
      int
Call
          a;
      cin >> a;
      int b = sqr(a);
      cout << a << " squared: "
           << b << endl:
      return 0:
      // main
```

```
Returned value
    here
            b
   а
copied
    Χ
```

```
int sqr (int x)
{
    return (x * x);
} // sqr
```

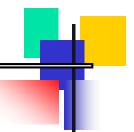


void functions cannot be used in an expression; they must be a separate statement.

Functions that return a value may be used in an expression or as a separate statement.



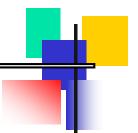
### Figure 4-7 Function definition



```
function
  header
return_type function_name (formal parameter list)
} // function_name
```

function body

### Figure 4-8 Function return statements



```
The function return type should be explicity defined
```

```
int first (...)
{
...
...
return (x + 2);
} // first
```

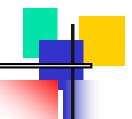
```
void second ( ... )

A return statement should be used even if nothing is returned ... return;
} // second
```

The type of the expression in the return statement must match the return type in the function header.



### Figure 4-9 Function local variables



Two values are received from the calling function

```
double average (int x, int y) {

double sum = x + y;
return (sum / 2);
} // average

One value is returned to the calling function
```



- Formal parameters are variables that are declared in the header of the function definition.
- Actual parameters are the expressions in the calling statement.
- The formal and actual parameters must match exactly in type, order, and number. Their names, however, do not need to be the same.

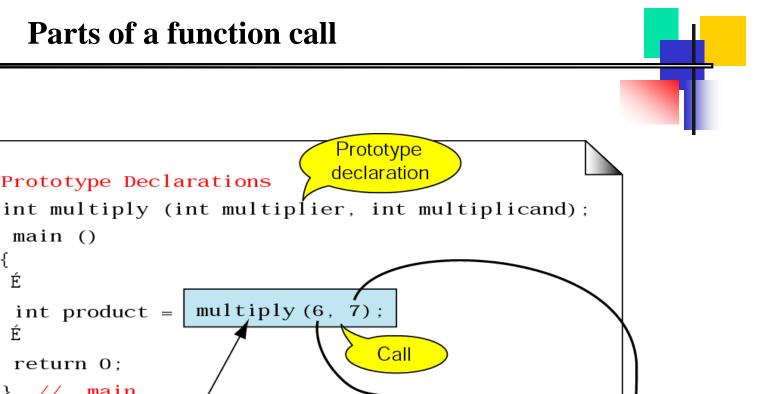


### Figure 4-10 Parts of a function call

int main ()

É

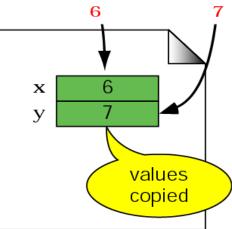
// Prototype Declarations



```
return 0;
                //
                     main
Function
Definition
                              42
        int multiply (in t \times x, int y)
            return | x * y |;
```

multiply

int product = | multiply (6, 7);



Prototype declaration

Call

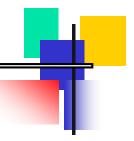
### Figure 4-11 Examples of function calls

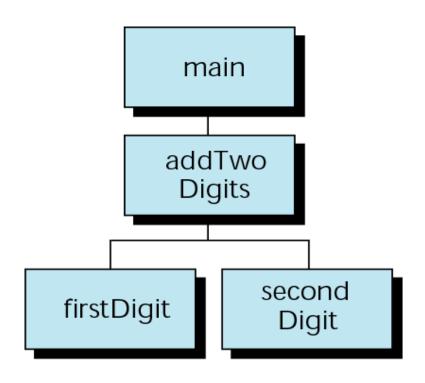


```
multiply (6, 7)
multiply (a, 7)
multiply (6, b)
multiply (a + 6, 7)
multiply (multiply (a, b), 7)
multiply ( ... , ... )
                expression
expression
```



### Figure 4-12 Design for addTwoDigits



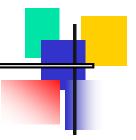


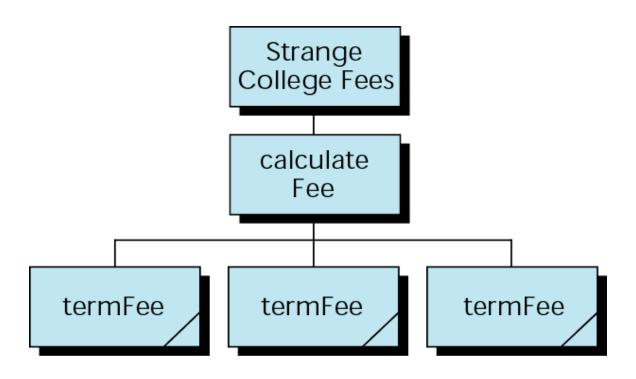


It is the nature of the task to be performed, not the amount of code, that determines if a function should be used.



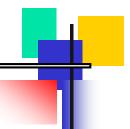
### Figure 4-13 Design for Strange College fees







### Figure 4-14 Pass by value



```
// Prototype Declarations
void fun (int num1);
int main ()
              prints 5
   int
         a
   fun (a)
   cout \ll \dot{a} \ll end1;
   return 0:
                                       One-way
   // main
                                    communication
void fun (int x)
                                     Х
   X = X + 3;
   return;
                                           Only a copy
   // fun
```

### Figure 4-15 Pass by reference



Variables are called

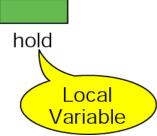
a and b in main ...

a b

num1 num2

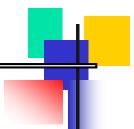
... num1 and num2

in exchange





### Figure 4-16 A bad exchange



```
// Prototype Statements

void exchange (int num1, int num2);
int main ()
{
  int a = 7;
  int b = 5;
  ...
  exchange (a, b);
  cout << a << " " << b << endl;
  ...
  return 0;
} // main</pre>
```

```
void exchange (int num1, int num2)
{
  int hold = num1;
    num1 = num2;
    num2 = hold;
  return;
} // exchange
```

```
Originals
   unchanged
    Values in
a and b are copied
to num1 and num2
          num2
  num1
Copies exchanged
  hold
        Local
        Variable
```

### Figure 4-17 Calculate quotient and remainder

= divnd / divsr;

= divnd % divsr;

```
// Prototype Statements
                                                      b
                                              а
void divide (int
                               divsr,
                 divnd, int
                                                      4
              int& quot, int& rem);
int main ()
  int a;
  int b;
  int c;
  int d;
  divide (a, b, c, d);
  return 0;
                                                             С
                                                                   d
                                                Value
 // main
void divide
             ( int
                     divnd,
                                                           quot
                     divsr,
                                                                  rem
               int
               int&
                    quot,
               int& rem)
                                            divnd
                                                    divsr
```

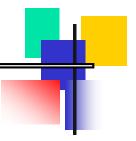


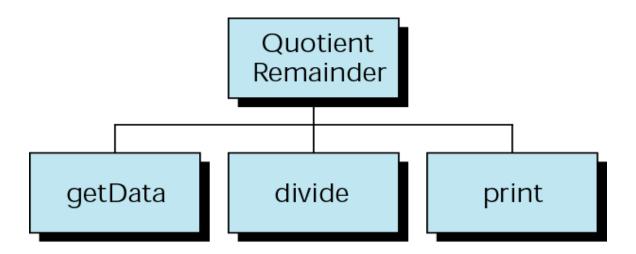
quot

rem

return ; // divide

### Figure 4-18 Quotient and remainder design







# DEFMULT PARMETER ARGUMENTS

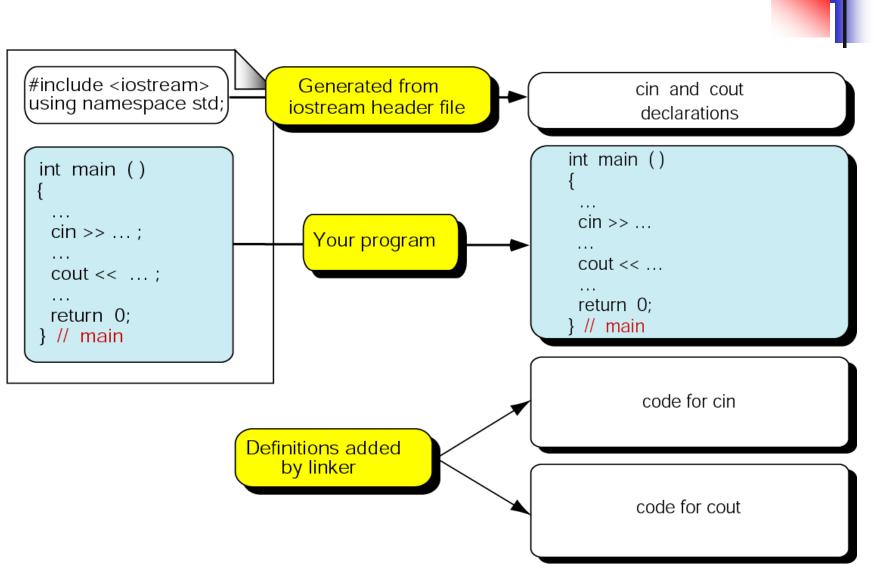


## STANDARD LIBRARY FUNCTIONS

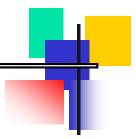


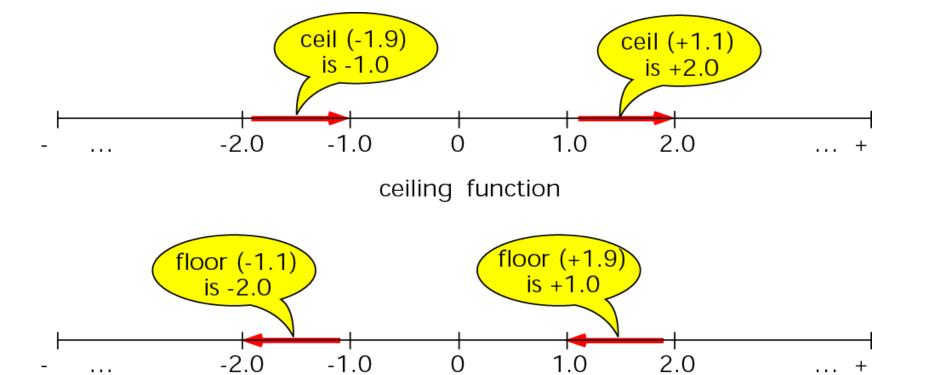
### Figure 4-19 Library functions and the linker





### Figure 4-20 Floor and ceiling functions

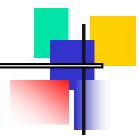


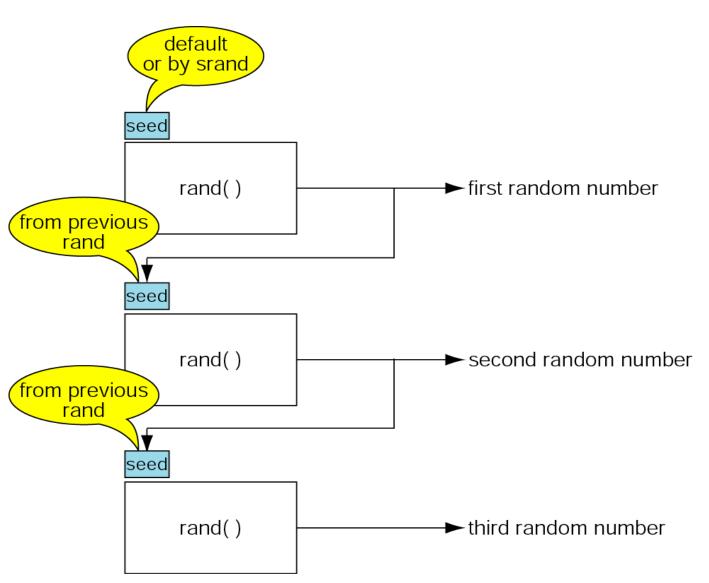


floor function



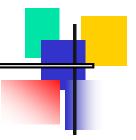
#### Figure 4-21 The random number seed

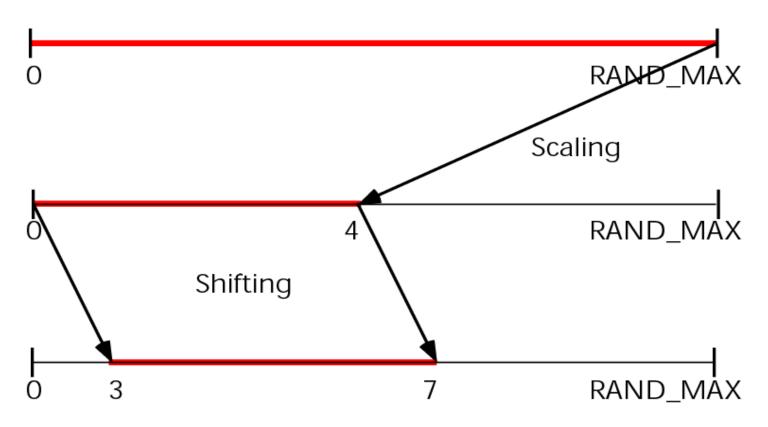






#### Figure 4-22 Random number scaling for 3-7





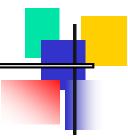


4.6

### SCOPE



#### Figure 4-23 Scope for global and block areas



```
This is a sample to demonstrate scope. The techniques used in
    this sample should never be used in practice.
#include <iostream>
using namespace std;
                                             Global Area
int fun (int a, int b);
  int main ()
   int
                                             Main's Area
         a;
   int
         b:
  float y;
       { // Beginning of nested block
        float a = y / 2;
        float y;
                                            Nested Block
        float z;
                                                   Area
         z = a * b / y;
         // End of nested block
  } // End of Main
int fun (int i, int j)
                                              fun's Area
 int a;
  int y;
 } // fun
```

Variables are in scope from their point of definition until the end of their function or block.



It is poor programming style to reuse identifiers within the same scope.

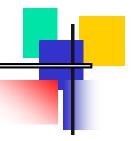


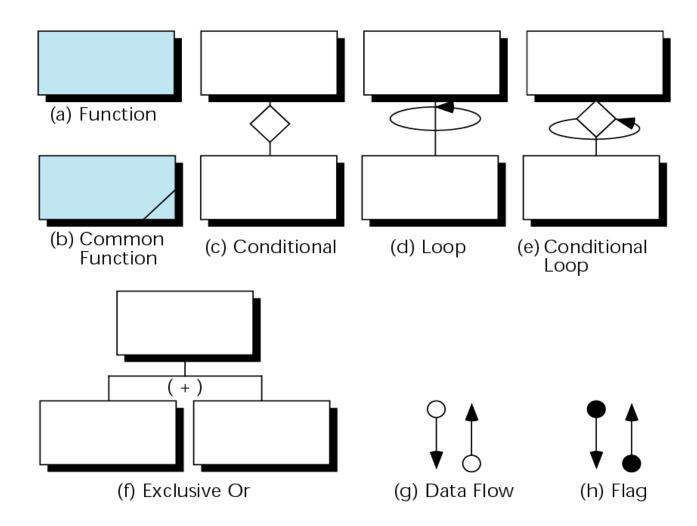
## PROGRAMMING EXMPLE— CALCULATOR PROGRAM

# SOFTWARE ENGINEERING AND PROGRAMMING STYLE



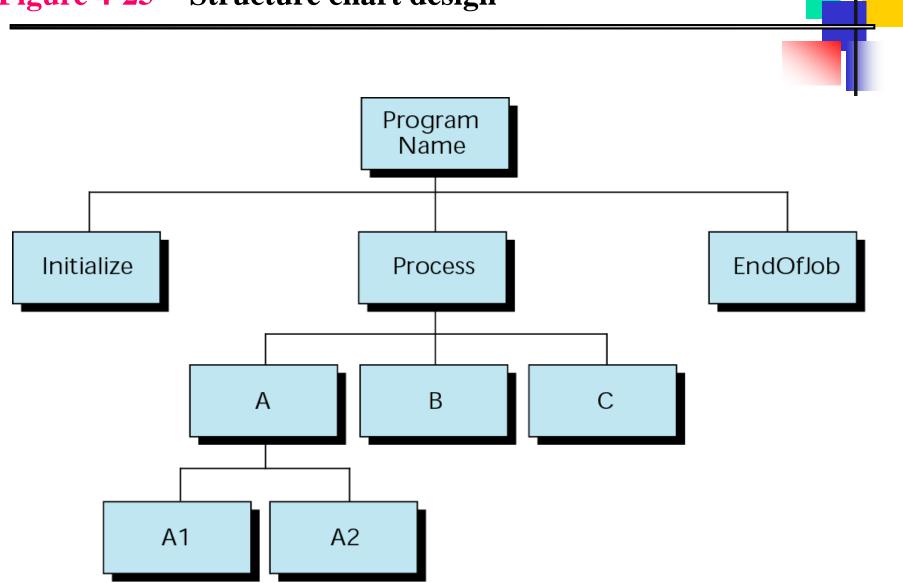
#### Figure 4-24 Structure chart symbols







#### Figure 4-25 Structure chart design

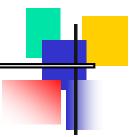


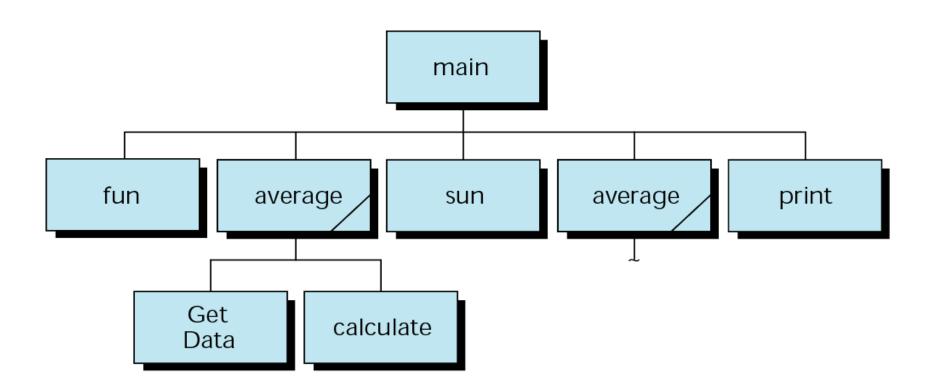


Structure charts show only function flow; they contain no code.



#### Figure 4-26 Common functions in a structure chart





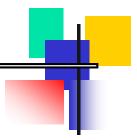


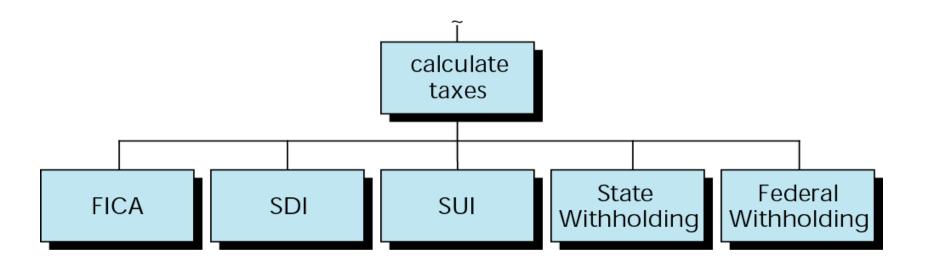
### Rule of Thumb

When a function has more than four parameters, check its cohesion.



#### Figure 4-27 Calculate taxes





#### Figure 4-28 Design for print report

