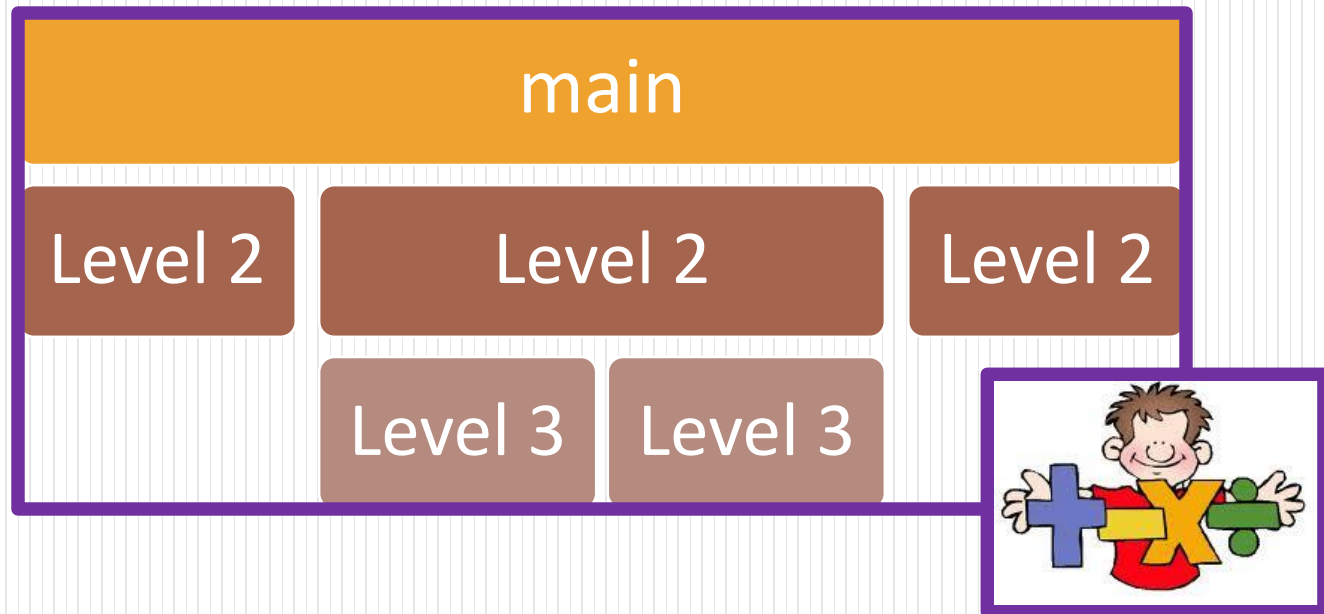


Programming in C



Chapter 7

Programmer-Defined Functions



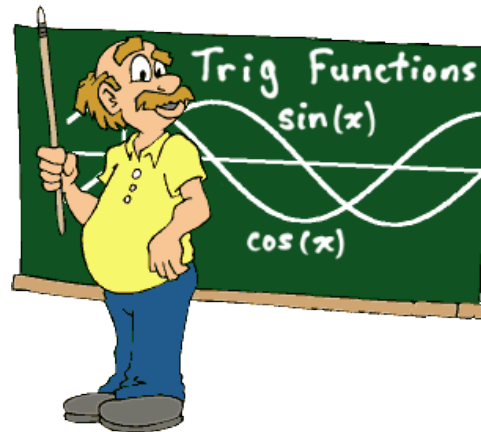
Programmer-Defined Functions

- Modularize with building blocks of programs
 - Divide and Conquer
 - Construct a program from smaller pieces or components
 - Place smaller pieces into functions
 - Pieces are more manageable than one big program
 - Makes other functions smaller
 - Pieces can be independently implemented and tested



Programmer-Defined Functions

- Readability
 - Function name should indicate operations performed
- Reuse
 - Functions may be used multiple times in same program
 - Functions may be used in other programs



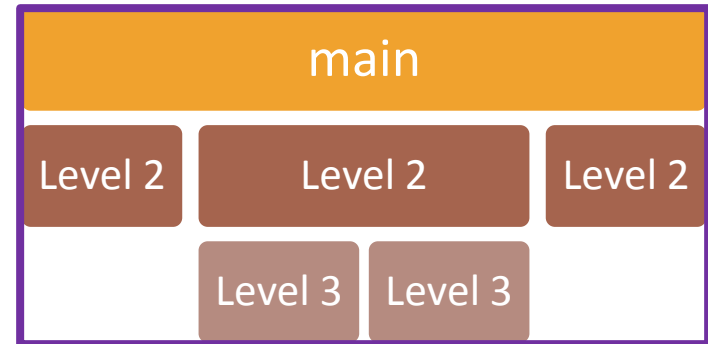
Components of Function Use

- Three steps to implementing functions
 1. Function declaration/prototype
 - If not defined before use
 2. Function definition
 3. Function call
 - Either prototype or definition must come first
- Prototype and/or definitions can go in either
 - Same file as main()
 - Separate file so other programs can also use it
 - #include

1.2.3

Program Function Definition Structure

- main first (preferred)
 - Top down design
 - Some prototypes required
 - Complete prototyping allows function definition in any order
- main is last – lowest level functions first
 - Bottom up design
 - Prototypes not required
- main in the middle
 - Confusing: **Do not do!**



1. Function Declaration/Prototype

- An 'informational' declaration for compiler
- Tells compiler how to interpret calls
- Syntax:

`<return_type> FnName (<formal-parameter-list>);`

- Formal parameter syntax:

`<data_type> Parameter-Name`

- Example:

```
char grade(int score);
```

Function Declaration/Prototype

- Placed before any calls
 - Generally above all functions in global space
 - May be placed in declaration space of calling function
- Example

```
#include <stdio.h>

// Function prototypes
double total_cost(int quantity, double unit_cost);

int main() {
```

Alternative Function Declaration

- Function declaration is 'information' for compiler, so
 - Compiler only needs to know:
 - Return type
 - Function name
 - Parameter list
 - Formal parameter names not needed but help readability
- Example

```
#include <stdio.h>

// Function prototypes
double total_cost(int, double);

int main() {
```


2. Function Definition

- Actual implementation/code for what function does
 - Just like implementing function main()
 - General format – header & basic block:

`<return-type> fn-name (parameter-list)`
`basic block`

← *header*


- Example:

```
double total_cost(int quantity, double unit_cost) {  
    const double TAXRATE = 0.05;  
    double sub_total;  
    sub_total = quantity * unit_cost;  
    return (sub_total + sub_total * TAXRATE);  
}
```

Return Statements

- Syntax: **return return-value-expression**
- Two actions
 - Sets return value
 - Transfers control back to 'calling' function
- **Good programming & course requirement:**
 - One return per function
 - Return is last statement

```
double total_cost(int quantity, double unit_cost) {  
    const double TAXRATE = 0.05;  
    double sub_total;  
    sub_total = quantity * unit_cost;  
    return (sub_total + sub_total * TAXRATE);  
}
```





3. Function Call

- Using function name transfers control to function
 1. Values are passed through parameters
 2. Statements within function are executed
 3. Control continues after the call

- For value-returning functions, either

- Store the value for later use

```
bill = total_cost(number, price);
```

- Use the value returned without storing

```
printf("Cost is %f\n", total_cost(number, price));
```

- Throw away return value

```
total_cost(number, price);
```

Parameters (Arguments)

- Formal parameters/arguments
 - In function declaration
 - In function definition's header
 - 'Placeholders' for data sent in
 - 'Variable name' used to refer to data in definition of function
- Actual parameters/arguments
 - In function call



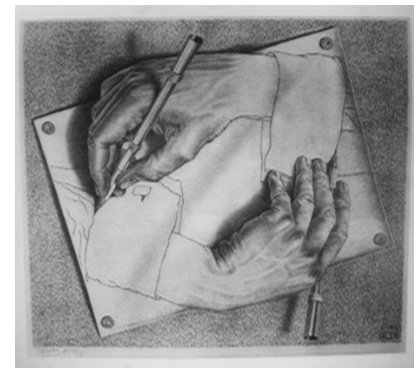
Parameter vs. Argument

- Names used interchangeably
- Technically parameter is 'formal' piece while argument is 'actual' piece



Functions Calling Functions

- We're already doing this!
 - `main()` IS a function calling `printf`!
- Only requirement:
 - Function's declaration or definition must appear first
- Common for functions to call many other functions
 - Function can call itself → Recursion

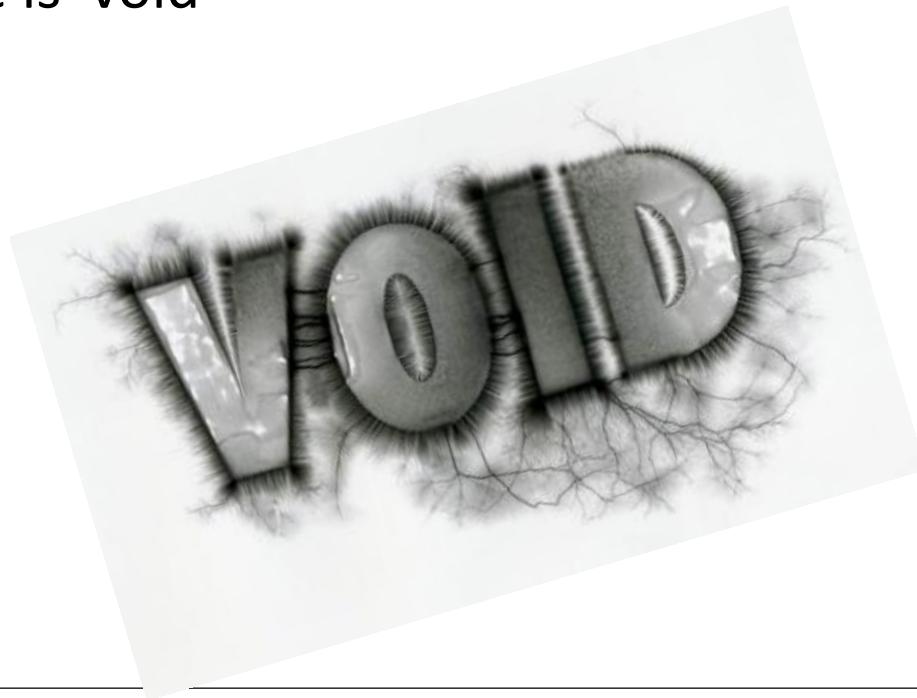


Declaring Void Functions

- Similar to functions returning a value
 - Return type specified as 'void'
- Example prototype:

```
void showResults(double fDegrees, double cDegrees);
```

- Return-type is 'void'



Declaring Void Functions

- Nothing is returned
 - Void functions cannot have return statement with an expression
 - Will return at end of function
 - Non-void functions must have return statement with an expression
- Example definition:

```
void showResults(double fDegrees, double cDegrees) {  
    printf("%.2f degrees fahrenheit equals ", fDegrees);  
    printf("%.2f degrees celsius\n", cDegrees);  
}
```


Calling Void Functions

- From some other function, like main():

```
showResults(degreesF, degreesC);  
showResults(32.5, 0.3);
```

- Cannot be used where a value is required
 - Cannot be assigned to a variable, since no value returned

Function documentation

- Used to aid in program maintenance
- Comments at non-main definition header
 - Purpose of function
 - Parameters
 - Return
 - **Class standard example:**

```
double interest(double balance, double rate);  
// Calculates the interest charge on an account balance  
// Parameters:   balance - non-negative account balance  
//              rate - interest rate percentage  
// Return:      calculated interest charge
```





main(): 'Special'

- Recall: main() IS a function
- 'Special'
 - It is the first function executed
 - Called by operating system or run-time system
 - Can return value to operating system
 - Value can be tested in command scripts
- Tradition holds it should return an int
 - Zero indicates normal ending of program



Scope of Identifier Names

- Region of a program where identifier is visible
 - Begins at definition within block
 - Ends at end of block
- Local variables
 - Name given to variables defined within function block
 - Can have different local variables with same name declared in different functions
 - Cannot have duplicate local names within a function

Scope Rules

- Local variables preferred
 - Maintain individual control over data
 - Need to know basis (Hidden)
 - Functions should declare whatever local data needed to 'do their job'





Global Scope

- Names declared 'outside' function bodies
 - Global to all functions in that file
- Global declarations typical for constants:
 - Declare globally so all functions have scope, can use

```
#include <stdio.h>

    const double TAX_RATE = 0.05;

int main() {
```

Global Constants and Global Variables

- Global variables?
 - Possible, but SELDOM-USED
 - Better alternative is to use parameters
 - Dangerous: no control over usage!
 - **We do not use global variables in this class!**



Block Scope

- Declare data inside nested blocks
 - Has 'block-scope'
 - Note: All function definitions are blocks!

```
if (amount > 5) {  
    int add_in;  
    add_in = prior_amount * .05;  
    amount += add_in;  
}
```


Lifetime



- How long does it last
 - Allocation \Rightarrow Deallocation
- Normally variables are allocated when defined
- Normally variables are deallocated at the end of block

```
double total_cost(int quantity, double unit_cost) {  
    const double TAXRATE = 0.05;    // TAXRATE allocated  
    double sub_total;                // sub_total allocated  
    sub_total = quantity * unit_cost;  
    return (sub_total + sub_total * TAXRATE);  
} // TAXRATE and sub_total deallocated
```

Static & Lifetime

- Variable definition modifier keyword: **static**
- Static variables are only allocated once
- Static variables are not deallocated until program ends

```
int keep_count() {  
    static int count = 0;  
    // count will remain allocated and keep its value  
    count++;  
    return count;  
}
```

Programming in C



Chapter 7

Programmer-Defined Functions

THE END