CSCI 1061U
Programming Workshop 2

Function Basics

Learning Objectives

- Predefined Functions
  - Those that return a value and those that don’t
- Programmer-defined Functions
  - Defining, Declaring, Calling
  - Recursive Functions
- Scope Rules
  - Local variables
  - Global constants and global variables
  - Blocks, nested scopes

Introduction to Functions

- Building Blocks of Programs
- Other terminology in other languages:
  - Procedures, subprograms, methods
  - In C++: functions
- I-P-O
  - Input – Process – Output
  - Basic subparts to any program
  - Use functions for these "pieces"

Predefined Functions

- Libraries full of functions for our use!
- Two types:
  - Those that return a value
  - Those that do not (void)
- Must "#include" appropriate library
  - e.g.,
    - <cmath>, <cstdlib> (Original "C" libraries)
    - <iostream> (for cout, cin)

Using Predefined Functions

- Math functions very plentiful
  - Found in library <cmath.h>
  - Most return a value (the "answer")
- Example: theRoot = sqrt(9.0);
  - Components:
    sqrt = name of library function
    theRoot = variable used to assign "answer" to
    9.0 = argument or "starting input" for function
  - In I-P-O:
    - I = 9.0
    - P = "compute the square root"
    - O = 3, which is returned & assigned to theRoot

The Function Call

- Back to this assignment:

```
theRoot = sqrt(9.0);
```

  - The expression "sqrt(9.0)" is known as a function *call*, or function *invocation*
  - The argument in a function call (9.0) can be a literal, a variable, or an expression
  - The call itself can be part of an expression:
    - bonus = sqrt(sales)/10;
    - A function call is allowed wherever it's legal to use an expression of the function's return type

A Predefined Function That Returns a Value

03-01.cpp UNREGISTERED

switch.cpp | 03-01.cpp

```cpp
//Computes the size of a dog house that can be purchased
//given the user's budget.
#include <iostream>
#include <cmath>
using namespace std;

int main( )
{
    const double COST_PER_SQ_FT = 10.50;
    double budget, area, lengthSide;

    cout << "Enter the amount budgeted for your dog house $";
    cin >> budget;

    area = budget/COST_PER_SQ_FT;
    lengthSide = sqrt(area);

    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
        cout << "For a price of $" << budget << endl
          << "I can build you a luxurious square dog house\n"
          << "that is " << lengthSide
          << " feet on each side.\n";

    return 0;
}
```

Line 25, Column 1 | 1 misspelled word | Spaces: 4 | C++

Sample output

```
Enter the amount budgeted for your doghouse $25.00
For a price of $25.00
I can build you a luxurious square doghouse
that is 1.54 feet on each side.
```

More Predefined Functions

- #include <cstdlib>
  - Library contains functions like:
    - abs() // Returns absolute value of an int
    - labs() // Returns absolute value of a long int
    - *fabs() // Returns absolute value of a float
  - *fabs() is actually in library <cmath>!
    - Can be confusing
    - Remember: libraries were added after C++ was "born," in incremental phases
    - Refer to appendices/manuals for details

More Math Functions

- pow(x, y)
  - Returns x to the power y
    ```
    double result, x = 3.0, y = 2.0;
    result = pow(x, y);
    cout << result;
    ```
    - Here 9.0 is displayed since $3.0^{2.0} = 9.0$
- Notice this function receives two arguments
  - A function can have any number of arguments, of varying data types

Some Predefined Math functions

NAME	DESCRIPTION	TYPE OF ARGUMENTS	TYPE OF VALUE RETURNED	EXAMPLE	VALUE	LIBRARY HEADER
sqrt	Square root	double	double	sqrt(4.0)	2.0	cmath
pow	Powers	double	double	pow(2.0,3.0)	8.0	cmath
abs	Absolute value for int	int	int	abs(-7) abs(7)	7 7	cstdlib
labs	Absolute value for long	long	long	labs(-70000) labs(70000)	70000 70000	cstdlib
fabs	Absolute value for double	double	double	fabs(-7.5) fabs(7.5)	7.5 7.5	cmath
ceil	Ceiling (round up)	double	double	ceil(3.2) ceil(3.9)	4.0 4.0	cmath
floor	Floor (round down)	double	double	floor(3.2) floor(3.9)	3.0 3.0	cmath
exit	End program	int	void	exit(1);	None	cstdlib
rand	Random number	None	int	rand( )	Varies	cstdlib
srand	Set seed for rand	unsigned int	void	srand(42);	None	cstdlib

Predefined Void Functions

- No returned value
- Performs an action, but sends no "answer"
- When called, it's a statement itself
  - `exit(1);` // No return value, so not assigned
    - This call terminates program
    - void functions can still have arguments
- All aspects same as functions that "return a value"
  - They just don't return a value!

Random Number Generator

- Return "randomly chosen" number
- Used for simulations, games
  - `rand()`
    - Takes no arguments
    - Returns value between 0 & RAND_MAX
  - Scaling
    - Squeezes random number into smaller range
      `rand() % 6`
    - Returns random value between 0 & 5
  - Shifting
    `rand() % 6 + 1`
    - Shifts range between 1 & 6 (e.g., die roll)

Random Number Seed

- Pseudorandom numbers
  - Calls to rand() produce given "sequence” of random numbers
- Use "seed" to alter sequence srand(seed_value);
  - void function
  - Receives one argument, the "seed"
  - Can use any seed value, including system time: srand(time(0));
  - time() returns system time as numeric value
  - Library <time> contains time() functions

Examples of Random Function

- Random double between 0.0 & 1.0:
  `(RAND_MAX – rand())/static_cast<double>(RAND_MAX)`
  - Type cast used to force double-precision division
- Random int between 1 & 6:
  `rand() % 6 + 1`
  - "%" is modulus operator (remainder)
- Random int between 10 & 20:
  `rand() % 10 + 10`

Programmer-Defined Functions

- Write your own functions!
- Building blocks of programs
  - Divide & Conquer
  - Readability
  - Re-use
- Your "definition" can go in either:
  - Same file as main()
  - Separate file so others can use it, too

Components of Function Use

- 3 Pieces to using functions:
  - Function Declaration/prototype
    - Information for compiler
    - To properly interpret calls
  - Function Definition
    - Actual implementation/code for what function does
  - Function Call
    - Transfer control to function

Function Declaration

- Also called function prototoype
- An "informational" declaration for compiler
- Tells compiler how to interpret calls
  - Syntax:
    <return_type> FnName(<formal-parameter-list>);
  - Example:
    ```
    double totalCost(int numberParameter,
                     double priceParameter);
    ```
- Placed before any calls
  - In declaration space of main()
  - Or above main() in global space

Function Definition

- Implementation of function
- Just like implementing function main()
- Example:

```
double totalCost(int numberParameter,
                 double priceParameter)
{
    const double TAXRATE = 0.05;
    double subTotal;
    subtotal = priceParameter * numberParameter;
    return (subtotal + subtotal * TAXRATE);
}
```

- Notice proper indenting

Function Definition Placement

- Placed after function main()
  - NOT "inside" function main()!
- Functions are "equals"; no function is ever "part" of another
- Formal parameters in definition
  - "Placeholders" for data sent in
    - "Variable name" used to refer to data in definition
- return statement
  - Sends data back to caller

Function Call

- Just like calling predefined function

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

- Recall: totalCost returns double value
  - Assigned to variable named "bill"
- Arguments here: number, price
  - Recall arguments can be literals, variables, expressions, or combination
  - In function call, arguments often called "actual arguments"
    - Because they contain the "actual data" being sent

User-Defined Function Example

Function declaration; also called function prototype

Function call

Function definition

Function head

Function body

03-05.cpp

```cpp
#include <iostream>
using namespace std;

double totalCost(int numberParameter, double priceParameter);
//Computes the total cost, including 5% sales tax,
//on numberPar items at a cost of pricePar each.

int main( )
{
    double price, bill;
    int number;

    cout << "Enter the number of items purchased: ";
    cin >> number;
    cout << "Enter the price per item $";
    cin >> price;

    bill = totalCost(number, price);

    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << number << " items at "
         << "$" << price << " each.\n"
         << "Final bill, including tax, is $" << bill
         << endl;

    return 0;
}

double totalCost(int numberParameter, double priceParameter)
{
    const double TAXRATE = 0.05; //5% sales tax
    double subtotal;

    subtotal = priceParameter * numberParameter;
    return (subtotal + subtotal*TAXRATE);
}
```

Alternative Function Declaration

- Recall: Function declaration is "information” for compiler
- Compiler only needs to know:
    - Return type
    - Function name
    - Parameter list
- Formal parameter names not needed:
  double totalCost(int, double);
  - Still "should" put in formal parameter names
    - Improves readability

Parameter vs. Argument

- Terms often used interchangeably
- Formal parameters/arguments
  - In function declaration
  - In function definition's header
- Actual parameters/arguments
  - In function call
- Technically parameter is "formal" piece while argument is "actual" piece*
  - *Terms not always used this way

Functions Calling Functions

- We're already doing this!
  - main() IS a function!
- Only requirement:
  - Function's declaration must appear first
- Function's definition typically elsewhere
  - After main()"s definition
  - Or in separate file
- Common for functions to call many other functions
- Function can even call itself → "Recursion"

Boolean Return-Type Functions

- Return-type can be any valid type
  - Given function declaration/prototype:
    bool appropriate(int rate);
  - And function's definition:
    ```
    bool appropriate (int rate)
    {
             return (((rate>=10)&&(rate<20))||(rate==0);
    }
    ```
  - Returns "true" or "false"
  - Function call, from some other function:
    ```
    if (appropriate(entered_rate))
         cout << "Rate is valid\n";
    ```

Declaring Void Functions

- Similar to functions returning a value
- Return type specified as "void"
- Example:
  - Function declaration/prototype:
    ```
    void showResults(double fDegrees,
                     double cDegrees);
    ```
    - Return-type is "void"
    - Nothing is returned

Declaring Void Functions

- Function definition:

```
void showResults(double fDegrees, double cDegrees)
{
     cout.setf(ios::fixed);
     cout.setf(ios::showpoint);
     cout.precision(1);
     cout   << fDegrees
            << " degrees fahrenheit equals \n"
            << cDegrees << " degrees celsius.\n";
}
```

- Notice: no return statement
  - Optional for void functions

Calling Void Functions

- Same as calling predefined void functions
- From some other function, like main():
  - `showResults(degreesF, degreesC);`
  - `showResults(32.5, 0.3);`
- Notice no assignment, since no value returned
- Actual arguments (degreesF, degreesC)
  - Passed to function
  - Function is called to "do it’s job" with the data passed in

More on Return Statements

- Transfers control back to "calling" function
  - For return type other than void, MUST have return statement
  - Typically the LAST statement in function definition
- return statement optional for void functions
  - Closing } would implicitly return control from void function

Preconditions and Postconditions

- Similar to "I-P-O" discussion
- Comment function declaration:

```
void showInterest(double balance, double rate);
//Precondition: balance is nonnegative account balance
//           rate is interest rate as percentage
//Postcondition: amount of interest on given balance,
//           at given rate …
```

- Often called Inputs & Outputs

main(): "Special"

- Recall: main() IS a function
- "Special" in that:
  - One and only one function called main() will exist in a program
- Who calls main()?
  - Operating system
  - Tradition holds it should have return statement
    - Value returned to "caller" → Here: operating system
  - Should return "int" or "void"

Scope Rules

- Local variables
  - Declared inside body of given function
  - Available only within that function
- Can have variables with same names declared in different functions
  - Scope is local: "that function is it’s scope"
- Local variables preferred
  - Maintain individual control over data
  - Need to know basis
  - Functions should declare whatever local data needed to "do their job"

Procedural Abstraction

- Need to know "what" function does, not "how" it does it!
- Think "black box"
  - Device you know how to use, but not it’s method of operation
- Implement functions like black box
  - User of function only needs: declaration
  - Does NOT need function definition
    - Called Information Hiding
    - Hide details of "how" function does it’s job

Global Constants and Global Variables

- Declared "outside" function body
  - Global to all functions in that file
- Declared "inside" function body
  - Local to that function
- Global declarations typical for constants:
  - `const double TAXRATE = 0.05;`
  - Declare globally so all functions have scope
- Global variables?
  - Possible, but SELDOM-USED
  - Dangerous: no control over usage!

Blocks

- Declare data inside compound statement
  - Called a "block"
  - Has "block-scope"
- Note: all function definitions are blocks!
  - This provides local "function-scope"
- Loop blocks:

```
for (int ctr=0;ctr<10;ctr++)
{
     sum+=ctr;
}
```

  - Variable ctr has scope in loop body block only

Nested Scope

- Same name variables declared in multiple blocks
- Very legal; scope is "block-scope"
  - No ambiguity
  - Each name is distinct within its scope

Summary 1

- Two kinds of functions:
    - "Return-a-value" and void functions
- Functions should be "black boxes"
    - Hide "how" details
    - Declare own local data
- Function declarations should self-document
    - Provide pre- & post-conditions in comments
    - Provide all "caller" needs for use

Summary 2

- Local data
  - Declared in function definition
- Global data
  - Declared above function definitions
  - OK for constants, not for variables
- Parameters/Arguments
  - Formal: In function declaration and definition
    - Placeholder for incoming data
  - Actual: In function call
    - Actual data passed to function

CSCI 1061U
Programming Workshop 2

Parameters and Overloading

Learning Objectives

- Parameters
  - Call-by-value
  - Call-by-reference
  - Mixed parameter-lists
- Overloading and Default Arguments
  - Examples, Rules
- Testing and Debugging Functions
  - assert Macro
  - Stubs, Drivers

Parameters

- Two methods of passing arguments as parameters
- Call-by-value
  - "copy" of value is passed
- Call-by-reference
  - "address of" actual argument is passed

Call-by-Value Parameters

- Copy of actual argument passed
- Considered "local variable" inside function
- If modified, only "local copy" changes
  - Function has no access to "actual argument” from caller
- This is the default method
  - Used in all examples thus far

Call-by-Value Example: **Display 4.1** Formal Parameter Used as a Local Variable (1 of 3)

Display 4.1 Formal Parameter Used as a Local Variable

```
//Law office billing program.
#include <iostream>
using namespace std;

const double RATE = 150.00; //Dollars per quarter hour.

double fee(int hoursWorked, int minutesWorked);
//Returns the charges for hoursWorked hours and
//minutesWorked minutes of legal services.

int main( )
{
    int hours, minutes;
    double bill;
```

Call-by-Value Example: **Display 4.1** Formal Parameter Used as a Local Variable (2 of 3)

```
    cout << "Welcome to the law office of\n"
         << "Dewey, Cheatham, and Howe.\n"
         << "The law office with a heart.\n"
         << "Enter the hours and minutes"
         << " of your consultation:\n";
    cin >> hours >> minutes;

    bill = fee(hours, minutes);

    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << "For " << hours << " hours and " << minutes
         << " minutes, your bill is $" << bill << endl;

    return 0;
}
```

The value of `minutes` *is not changed by the call to* `fee`.

(continued)

Call-by-Value Example: **Display 4.1** Formal Parameter Used as a Local Variable (3 of 3)

Display 4.1 Formal Parameter Used as a Local Variable

```
double fee(int hoursWorked, int minutesWorked)
{
    int quarterHours;

    minutesWorked = hoursWorked*60 + minutesWorked;
    quarterHours = minutesWorked/15;
    return (quarterHours*RATE);
}
```

`minutesWorked` *is a local variable initialized to the value of* `minutes`.

SAMPLE DIALOGUE

Welcome to the law office of
Dewey, Cheatham, and Howe.
The law office with a heart.
Enter the hours and minutes of your consultation:
5 46
For 5 hours and 46 minutes, your bill is $3450.00

Call-by-Value Pitfall

- Common Mistake:
  - Declaring parameter "again" inside function:

```
double fee(int hoursWorked, int minutesWorked)
{
  int quarterHours;          // local variable
  int minutesWorked          // NO!
}
```

  - Compiler error results
    - "Redefinition error…"
- Value arguments ARE like "local variables"
  - But function gets them "automatically"

Call-By-Reference Parameters

- Used to provide access to caller’s actual argument
- Caller’s data can be modified by called function!
- Typically used for input function
  - To retrieve data for caller
  - Data is then "given" to caller
- Specified by ampersand, &, after type in formal parameter list

Call-By-Reference Example:
Display 4.1 Call-by-Reference Parameters (1 of 3)

Display 4.2 Call-by-Reference Parameters

```
//Program to demonstrate call-by-reference parameters.
#include <iostream>
using namespace std;

void getNumbers(int& input1, int& input2);
//Reads two integers from the keyboard.

void swapValues(int& variable1, int& variable2);
//Interchanges the values of variable1 and variable2.

void showResults(int output1, int output2);
//Shows the values of variable1 and variable2, in that order.

int main( )
{
    int firstNum, secondNum;

    getNumbers(firstNum, secondNum);
    swapValues(firstNum, secondNum);
    showResults(firstNum, secondNum);
    return 0;
}
```

Call-By-Reference Example:
Display 4.1 Call-by-Reference Parameters (2 of 3)

```
void getNumbers(int& input1, int& input2)
{
    cout << "Enter two integers: ";
    cin >> input1
        >> input2;
}

void swapValues(int& variable1, int& variable2)
{
    int temp;

    temp = variable1;
    variable1 = variable2;
    variable2 = temp;
}

void showResults(int output1, int output2)
{
    cout << "In reverse order the numbers are: "
         << output1 << " " << output2 << endl;
}
```

Call-By-Reference Example:
Display 4.1 Call-by-Reference Parameters (3 of 3)

Display 4.2 Call-by-Reference Parameters

SAMPLE DIALOGUE

```
Enter two integers: 5 6
In reverse order the numbers are: 6 5
```

Call-By-Reference Details

- What's really passed in?
- A "reference" back to caller's actual argument!
  - Refers to memory location of actual argument
  - Called "address", which is a unique number referring to distinct place in memory

Constant Reference Parameters

- Reference arguments inherently "dangerous"
  - Caller's data can be changed
  - Often this is desired, sometimes not
- To "protect" data, & still pass by reference:
  - Use const keyword
    - ```
      void sendConstRef(const int &par1,
                        const int &par2);
      ```
    - Makes arguments "read-only" by function
    - No changes allowed inside function body
      ```

Parameters and Arguments

- Confusing terms, often used interchangeably
- True meanings:
  - Formal parameters
    - In function declaration and function definition
  - Arguments
    - Used to "fill-in" a formal parameter
    - In function call (argument list)
  - Call-by-value & Call-by-reference
    - Simply the "mechanism" used in plug-in process

Mixed Parameter Lists

- Can combine passing mechanisms
- Parameter lists can include pass-by-value and pass-by-reference parameters
- Order of arguments in list is critical:
  `void mixedCall(int & par1, int par2, double & par3);`
  - Function call:
    mixedCall(arg1, arg2, arg3);
    - arg1 must be integer type, is passed by reference
    - arg2 must be integer type, is passed by value
    - arg3 must be double type, is passed by reference

Choosing Formal Parameter Names

- Same rule as naming any identifier:
  - Meaningful names!
- Functions as "self-contained modules"
  - Designed separately from rest of program
  - Assigned to teams of programmers
  - All must "understand" proper function use
  - OK if formal parameter names are same as argument names
- Choose function names with same rules

Overloading

- Same function name
- Different parameter lists
- Two separate function definitions
- Function "signature"
  - Function name & parameter list
  - Must be "unique" for each function definition
- Allows same task performed on different data

Overloading Example: Average

- Function computes average of 2 numbers:

```
double average(double n1, double n2)
{
      return ((n1 + n2) / 2.0);
}
```

- Now compute average of 3 numbers:

```
double average(double n1, double n2, double n3)
{
      return ((n1 + n2) / 2.0);
}
```

- Same name, two functions

Overloaded Average() Cont'd

- Which function gets called?
- Depends on function call itself:
  - `avg = average(5.2, 6.7);`
    - Calls "two-parameter average()"
  - `avg = average(6.5, 8.5, 4.2);`
    - Calls "three-parameter average()"
- Compiler resolves invocation based on signature of function call
  - "Matches" call with appropriate function
  - Each considered separate function

Overloading Pitfall

- Only overload "same-task" functions
  - A mpg() function should always perform same task, in all overloads
  - Otherwise, unpredictable results
- C++ function call resolution:
  - 1st: looks for exact signature
  - 2nd: looks for "compatible" signature

Overloading Resolution

- 1st: Exact Match
  - Looks for exact signature
    - Where no argument conversion required
- 2nd: Compatible Match
  - Looks for "compatible" signature where automatic type conversion is possible:
    - 1st with promotion (e.g., int→double)
      - No loss of data
    - 2nd with demotion (e.g., double→int)
      - Possible loss of data

Overloading Resolution Example

- Given following functions:
  - 1. void f(int n, double m);
    2. void f(double n, int m);
    3. void f(int n, int m);
  - These calls:
    f(98, 99); → Calls #3
    f(5.3, 4); → Calls #2
    f(4.3, 5.2); → Calls ???
- Avoid such confusing overloading

Automatic Type Conversion and Overloading

- Numeric formal parameters typically made "double" type
- Allows for "any" numeric type
  - Any "subordinate" data automatically promoted
    - int → double
    - float → double
    - char → double *More on this later!
- Avoids overloading for different numeric types

Automatic Type Conversion and Overloading Example

- ```
  double mpg(double miles, double gallons)
  {
        return (miles/gallons);
  }
  ```
- Example function calls:
  - `mpgComputed = mpg(5, 20);`
    - Converts 5 & 20 to doubles, then passes
  - `mpgComputed = mpg(5.8, 20.2);`
    - No conversion necessary
  - `mpgComputed = mpg(5, 2.4);`
    - Converts 5 to 5.0, then passes values to function
  ```

Default Arguments

- Allows omitting some arguments
- Specified in function declaration/prototype
  - ```
    void showVolume(int length,
                    int width = 1,
                    int height = 1);
    ```
    - Last 2 arguments are defaulted
  - Possible calls:
    - `showVolume(2, 4, 6); //All arguments supplied`
    - `showVolume(3, 5); //height defaulted to 1`
    - `showVolume(7); //width & height defaulted to 1`
    ```

Default Arguments Example:
Display 4.1 Default Arguments (1 of 2)

Display 4.8 Default Arguments

Default arguments

```cpp

#include <iostream>
using namespace std;

void showVolume(int length, int width = 1, int height = 1);
//Returns the volume of a box.
//If no height is given, the height is assumed to be 1.
//If neither height nor width is given, both are assumed to be 1.

int main( )
{
    showVolume(4, 6, 2);
    showVolume(4, 6);
    showVolume(4);

    return 0;
}

void showVolume(int length, int width, int height)
```

A default argument should not be given a second time.

Default Arguments Example:
Display 4.1 Default Arguments (2 of 2)

```
{
     cout << "Volume of a box with \n"
          << "Length = " << length << ", Width = " << width << endl
          << "and Height = " << height
          << " is " << length*width*height << endl;
}
```

SAMPLE DIALOGUE

```
Volume of a box with
Length = 4, Width = 6
and Height = 2 is 48
Volume of a box with
Length = 4, Width = 6
and Height = 1 is 24
Volume of a box with
Length = 4, Width = 1
and Height = 1 is 4
```

Testing and Debugging Functions

- Many methods:
  - Lots of cout statements
    - In calls and definitions
    - Used to "trace" execution
  - Compiler Debugger
    - Environment-dependent
  - assert Macro
    - Early termination as needed
  - Stubs and drivers
    - Incremental development

The assert Macro

- Assertion: a true or false statement
- Used to document and check correctness
  - Preconditions & Postconditions
    - Typical assert use: confirm their validity
  - Syntax:
    assert(<assert_condition>);
    - No return value
    - Evaluates assert_condition
    - Terminates if false, continues if true
- Predefined in library <cassert>
  - Macros used similarly as functions

An assert Macro Example

- Given Function Declaration:

```
void computeCoin(int coinValue,
                 int& number,
                 int& amountLeft);
//Precondition: 0 < coinValue < 100
//              0 <= amountLeft <100
//Postcondition: number set to max. number of coins
```

- Check precondition:
  - `assert ((0 < currentCoin) && (currentCoin < 100)`
    `&& (0 <= currentAmountLeft) && (currentAmountLeft < 100));`
  - If precondition not satisfied → condition is false → program execution terminates!

An assert Macro Example Cont’d

- Useful in debugging
- Stops execution so problem can be investigated

assert On/Off

- Preprocessor provides means
- ```
  #define NDEBUG
  #include <cassert>
  ```
- Add "#define" line before #include line
  - Turns OFF all assertions throughout program
- Remove "#define" line (or comment out)
  - Turns assertions back on
  ```

Stubs and Drivers

- Separate compilation units
  - Each function designed, coded, tested separately
  - Ensures validity of each unit
  - Divide & Conquer
    - Transforms one big task → smaller, manageable tasks
- But how to test independently?
  - Driver programs

Driver Program Example:
Display 4.9 Driver Program (1 of 3)

Display 4.9 Driver Program

```
//Driver program for the function unitPrice.
#include <iostream>
using namespace std;

double unitPrice(int diameter, double price);
//Returns the price per square inch of a pizza.
//Precondition: The diameter parameter is the diameter of the pizza
//in inches. The price parameter is the price of the pizza.

int main( )
{
    double diameter, price;
    char ans;

    do
    {
        cout << "Enter diameter and price:\n";
        cin >> diameter >> price;
```

Driver Program Example: **Display 4.9** Driver Program (2 of 3)

```
        cout << "unit Price is $"
             << unitPrice(diameter, price) << endl;

        cout << "Test again? (y/n)";
        cin >> ans;
        cout << endl;
    } while (ans == 'y' || ans == 'Y');

    return 0;
}

double unitPrice(int diameter, double price)
{
    const double PI = 3.14159;
    double radius, area;

    radius = diameter/static_cast<double>(2);
    area = PI * radius * radius;
    return (price/area);
}
```

(continued)

Driver Program Example:
Display 4.9 Driver Program (3 of 3)

Display 4.9 Driver Program

SAMPLE DIALOGUE

```
Enter diameter and price:
13 14.75
Unit price is: $0.111126
Test again? (y/n): y

Enter diameter and price:
2 3.15
Unit price is: $1.00268
Test again? (y/n): n
```

Stubs

- Develop incrementally
- Write "big-picture" functions first
  - Low-level functions last
  - "Stub-out" functions until implementation
  - Example:
```
double unitPrice(int diameter, double price)
{
      return (9.99);   // not valid, but noticeably
                       // a "temporary" value
}
```
  - Calls to function will still "work"

Fundamental Testing Rule

- To write "correct" programs
- Minimize errors, "bugs"
- Ensure validity of data
  - Test every function in a program where every other function has already been fully tested and debugged
  - Avoids "error-cascading" & conflicting results

Summary 2

- Formal parameter is placeholder, filled in with actual argument in function call
- Call-by-value parameters are "local copies" in receiving function body
  - Actual argument cannot be modified
- Call-by-reference passes memory address of actual argument
  - Actual argument can be modified
  - Argument MUST be variable, not constant

Summary 2

- Multiple definitions of same function name possible: called overloading
- Default arguments allow function call to "omit" some or all arguments in list
  - If not provided → default values assigned
- assert macro initiates program termination if assertions fail
- Functions should be tested independently
  - As separate compilation units, with drivers

Readings

- Ch. 3