4 Procedural Abstraction

- The Black Box Analogy
  - A **black box** refers to something that we know how to use, but the method of operation is unknown.
  - A person using a program does not need to knowhow it is coded.
  - A person using a program needs to know what the program does, not how it does it.
- Functions and the Black Box Analogy
  - A programmer who uses a function needs to know what the function does, not how it does it.
  - A programmer needs to know what will be produced if the proper arguments are put into the box.

4.1 Information Hiding

- Designing functions as black boxes is an example of information hiding.
  - The function can be used without knowing how it is coded.
  - The function body can be "hidden from view".

4.2 Function Implementations and The Black Box

- Designing with the black box in mind allows us,
  - To change or improve a function definition without forcing programmers using the function to change what they have done.
  - To know how to use a function simply by reading the function declaration and its comment.

4.3 Procedural Abstraction and C++

- **Procedural Abstraction** is writing and using functions as if they were **black boxes**
  - Procedure is a general term meaning a "function like" set of instructions.
  - Abstraction implies that when you use a function as a black box, you abstract away the details of the code in the function body.

4.4 Procedural Abstraction and Functions

- Write functions so the declaration and comment is all a programmer needs to use the function
  - Function comment should tell all conditions required of arguments to the function.
  - Function comment should describe the returned value.
  - Variables used in the function, other than the formal parameters, should be declared in the function body.

4.5 Formal Parameter Names

- Functions are designed as self-contained modules
- Different programmers may write each function.
- Programmers choose meaningful names for formal parameters,
  - Formal parameter names may or may not match variable names used in the main part of the program.
  - It does not matter if formal parameter names match other variable names in the program.
  - Remember that only the value of the argument is plugged into the formal parameter.

4.6 Case Study: Buying Pizza

- What size pizza is the best buy?
  - Which size gives the lowest cost per square inch?
  - Pizza sizes given in diameter.
  - Quantity of pizza is based on the area which is proportional to the square of the radius.

4.6.1 Buying Pizza Problem Definition

- Input:
  - Diameter of two sizes of pizza.
  - Cost of the same two sizes of pizza.
- Output:
  - Cost per square inch for each size of pizza.
  - Which size is the best buy,
    - Based on lowest price per square inch,
    - If cost per square inch is the same, the smaller size will be the better buy.

4.6.2 Buying Pizza Problem Analysis

- Subtask 1
  - Get the input data for each size of pizza
- Subtask 2
  - Compute price per inch for smaller pizza
- Subtask 3
  - Compute price per inch for larger pizza
- Subtask 4
  - Determine which size is the better buy
- Subtask 5
  - Output the results

4.6.3 Buying Pizza Function Analysis

- Subtask 2 and subtask 3 should be implemented as a single function because
  - Subtask 2 and subtask 3 are identical tasks
    - The calculation for subtask 3 is the same as the calculation for subtask 2 with different arguments
  - Subtask 2 and subtask 3 each return a single value
- Choose an appropriate name for the function
  - We'll use **unitprice**

Buying Pizza **unitprice** Declaration

```
double unitprice(int diameter, double price);
// Returns the price per square inch of a pizza
// The formal parameter named diameter is the
// diameter of the pizza in inches.  The formal
// parameter named price is the price of the
// pizza.
```

4.6.4 Buying PizzaAlgorithm Design

- Subtask 1
  - Ask for the input values and store them in variables
    diameter_small diameter_large
    price_small price_large
- Subtask 4
  - Compare cost per square inch of the two pizzas using the less than operator
- Subtask 5
  - Standard output of the results

4.6.5 Buying Pizza unitprice Algorithm

- Subtasks 2 and 3 are implemented as calls to function **unitprice**
- **unitprice** algorithm
  - Compute the radius of the pizza
  - Computer the area of the pizza using πr^2
  - Return the value of (price / area)

4.6.6 Buying Pizza unitprice Pseudocode

- Pseudocode
  - Mixture of C++ and english
  - Allows us to make the algorithm more precise without worrying about the details of C++ syntax
- **unitprice** pseudocode
  - radius = one half of diameter;
    area = π * radius * radius
    return (price / area)

4.6.7 Buying PizzaThe Calls of unitprice

- Main part of the program implements calls of **unitprice** as

```
double unit_price_small, unit_price_large;
: // subtask 1
:
unit_price_small = unitprice(diameter_small, price_small); // subtask 2
unit_price_large = unitprice(diameter_large, price_large); // subtask 3
```

4.6.8 Buying Pizza First try at unitprice

```
double unitprice (int diameter, double price)
{
        const double PI = 3.14159;
        double radius, area;
        radius = diameter / 2;
        area = PI * radius * radius;
        return (price / area);
}
```

Oops! Radius should include the fractional part

4.6.9 Buying Pizza Second try at unitprice

```
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);
}
```

Now radius will include fractional parts

```
        radius = diameter / 2.0 ; // This would also work
```

4.6.10 Program Testing

- Programs that compile and run can still produce errors
- Testing increases confidence that the program works correctly
  - Run the program with data that has known output
    - You may have determined this output with pencil and paper or a calculator
  - Run the program on several different sets of data
    - Your first set of data may produce correct results in spite of a logical error in the code
      - Remember the integer division problem? If there is no fractional remainder, integer division will give apparently correct results

4.7 Use Pseudocode

- Pseudocode is a mixture of English and the programming language in use
- Pseudocode simplifies algorithm design by allowing you to ignore the specific syntax of the programming language as you work out the details of the algorithm
  - If the step is obvious, use C++
  - If the step is difficult to express in C++, use English