Objectives

In this chapter, you will learn about:

- Top-Down Design
- Predefined Functions
- Programmer-Defined Functions
- Procedural Abstraction
- Local Variables
- Overloading Function Names
Top-Down Design

- **Top-Down Design Technique same as**
  - Divide and Conquer (refer to chapter 1)
  - Stepwise Refinement

- **Basic Idea**
  - **Problem Decomposition (divide and conquer)**
    - Decompose Program Task into Tractable subtasks
    - Solve each subtask independently using “sub-algorithms”
  - **Algorithm Decomposition (stepwise refinement)**
    - Decompose the algorithm into Tractable subtasks
    - Decompose each subtask into smaller subtasks
    - Write code for each smaller subtasks
Top-Down Design

Benefits

- Software is easy to write by a team of Software Developers
- Software is easy to map to requirements
- Software is easy to test
- Software is easy to Maintain
Predefined Functions

Using Predefined Functions

- C++ has libraries of predefined functions
- Examples:
  - Square root function
  - Powers function
  - Absolute function
  - …..
Using Predefined Functions
Square Root function

- The Square Root Function
  - Name: `sqrt`
  - Purpose: Compute ("return") the square root of a number
  - "Function Call":

```
the_number = sqrt(9.0);
```

Stores the returned value 3.0
(Type: double)

Argument (Type: double)
Function Call
Square Root function

- **sqrt(9.0)**
  - Invokes the sqrt function in the C++ library

Example – Function argument is a variable

```cpp
double the_number, sales = 9.0,
    the_number = sqrt(sales);
```

Example – Function call within an expression

```cpp
double the_number, sales = 9.0,
    the_number = 4.0*sqrt(sales)/10.0;
```
Function Call
Syntax

*Function Name*(Argument_List)

*Argument List* is a comma-separated list of arguments:

Argument_1, Argument_2, ..., Argument_last

Example:

```
cout << "2.5 to the power 3.0 is" << pow(2.5, 3.0)
```

Invokes the library function pow( ).

Which Library?
Predefined Function Libraries

- Include the library name in your program before you make the function call
- The function `sqrt()` is contained in a math library called `cmath`
- To include `cmath` library in your program:
  - `#include<cmath>`
  - Additionally, your compiler (or processor) may also require `using` directive:
  - `using namespace std;`
# Some Predefined Function Specifications

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Argument Types</th>
<th>Type of Value Returned</th>
<th>Library (Header File)</th>
<th>Example</th>
<th>Value Returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>sqrt</td>
<td>square root</td>
<td>double</td>
<td>double</td>
<td>cmath</td>
<td>sqrt(4.0)</td>
<td>2.0</td>
</tr>
<tr>
<td>pow</td>
<td>powers</td>
<td>double</td>
<td>double</td>
<td>cmath</td>
<td>pow(2.0, 3.0)</td>
<td>8.0</td>
</tr>
<tr>
<td>abs</td>
<td>absolute value for int</td>
<td>int</td>
<td>int</td>
<td>cstdlib</td>
<td>abs(-7)</td>
<td>7</td>
</tr>
<tr>
<td>fabs</td>
<td>absolute value for double</td>
<td>double</td>
<td>double</td>
<td>cstdlib</td>
<td>fabs(-7.5)</td>
<td>7.5</td>
</tr>
</tbody>
</table>

See Appendix 4 for more pre-defined functions
**Type Casting**

Consider:

```cpp
int total_candy = 9, number_of_people = 2;
double candy_per_person;
candy_per_person = total_candy/number_of_people;
```

Output:
```
candy_per_person
4.0
```

**Type Casting:** A Technique to convert a value from one type to another

To correct the problem, use:

```cpp
candy_per_person = static_cast<double>(total_candy)/number_of_people;
```

or

```cpp
candy_per_person = total_candy/ static_cast<double>(number_of_people);
```

**What is wrong with this?**
```
candy_per_person = static_cast<double>(total_candy/number_of_people);
```
Predefined Functions

Summary

- Determine the value of each of the following expressions:
  - `double credit; credit = 10/4;`
  - `pow(2,3)`
  - `pow(2.0, 3.0)`
  - `fabs(-3.5)`
  - `ceil(5.8)`
  - `floor(5.1)`

- Convert the following to C++ arithmetic expressions:
  - $\sqrt{x + y}$
  - $\frac{-b + \sqrt{b^2 - 4ac}}{2a}$
  - $|x - y|$
You can define your own function either in the same file containing the main program or in a separate file.

Programmer Defined Function has two parts:

- Function Declaration

```c
Type_returned  Function_Name(Parameter_List);  //Comment describing what function does
```
- Shows how to call the function
- Should appear in the source code before the function can be called

- Function Definition
Example:

double total_cost(int number_par, double price_par);
// Compute total cost, including 5% sales tax on
// number_par items at cost of price_par each

- The return type
- The name of the function
- How many arguments are needed
- The types of the arguments
- The formal parameter names
  - Placeholders for the actual arguments used when the function is called
  - Names can be any valid identifier
Function Definition

```
Type_returned Function_Name(Parameter_List)
{
    //code to make the function work
}
```

- Shows Function heading
- Shows Function body
- Can appear before or after the function is called
Programmer Defined Functions

Function Definition

Example

```c
double total_cost (int number_par, double price_par) {
    const double TAX_RATE = 0.05; //5% tax
    double subtotal;
    subtotal = price_par * number_par;
    return (subtotal + subtotal * TAX_RATE);
}
```

- Shows Function declaration info
- Function body describes the instructions
Programmer Defined Functions

Return Statement

- Returns the value computed by the function
- Syntax: \[\text{return expression;}\]
  - Where \textit{expression} could be:
    - a value
    - Variable containing the calculated value
    - Arithmetic, logical .. Expression
- Function does not return a value until the return statement is executed
- Ends execution of statements in function body

Examples:

\[\text{return 0;}\]
\[\text{return ‘A’;}\]
\[\text{return “KSU”;}\]
\[\text{return subtotal + subtotal * TAX_RATE;}\]
\[\text{return (subtotal + subtotal * TAX RATE);}\]


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

char mystery (int first_par, int second_par);

int main( )
{
    cout << mystery(10,9) << “ow\n”;  
    return 0;
}

char mystery (int first_par, int second_par)
{
    if (first_par >= second_par)
        return “W”;
    else
        return “H”;
}
```

Function Declaration

Function Call

Function Definition
Programmer Defined Functions

Function call

- Specifies the **name** of the desired function
- Lists applicable **arguments** within parenthesis
- Arguments must be separated by **comma**
- Example:

```plaintext
bill = total_cost(number, price);
```
```cpp
#include <iostream>
using namespace std;

double total_cost(int number, double price_par);
//Computes total cost, including 5% sales tax.

int main() {
    double price, bill;
    int number;
    cout << "Enter the number of items purchased: ";
    cin >> number;
    cout << "Enter price per item: ";
    cin >> price;
    bill = total_cost(number, price);
    cout << number << " items\nFinal bill, incl tax is $" << bill << endl;
    return 0;
}

double total_cost(int number_par, double price_par) {
    const double TAX_RATE = 0.05;
    double subtotal;
    subtotal = price_par * number_par;
    return (subtotal + subtotal * TAX_RATE);
}
```
#include <iostream>
using namespace std;

double total_cost(int number, double price_par);
//Computes total cost, including 5% sales tax..
int main ( )
{
    double price, bill;
    int number;
    cout << "Enter the number of items purchased: ";
    cin >> number;
    cout << "Enter price per item: ";
    cin >> price;
    bill = total_cost (2, 10);
    cout << number," items\n" << "Final bill, incl tax is $" << bill << endl;
    return 0;
}

double total_cost (int number_par, double price_par)
{
    const double TAX_RATE = 0.05;
    double subtotal;
    subtotal = price_par*number_par;
    return (subtotal + subtotal*TAX_RATE);
}
The first argument (in the function call) is “plugged” into the first formal parameter (in the function definition), the second argument in the second formal parameter, the third in the third formal parameter and so on.

The formal parameter is a “local variable”, and is used only in the function body.

The value plugged into the formal parameter is used to initialize the corresponding local variable.

What happens if you list the arguments are in the wrong order?
When a function Call lists arguments in **Wrong Order:**

- If the type of argument does not match formal parameter type, some compilers will generate warning messages.
- If argument type matches formal parameter type, the compiler will not detect the error message, and you function will, for the most part, return the wrong value.
List formal parameter names ("BAU"):  
- double total_cost(int number_par, double price_par);

Omit formal parameter names, but list the types:  
- double total_cost(int, double);

Let’s stay with BAU for our function declarations
Programmer Defined Functions

Function Definition Syntax

**Function Declaration**

```c
Type_returned Function_name (Parameter List);
```

*Insert Function Declaration Comments here*

**Function Definition**

```c
Type_returned Function_name (Parameter List) {
  Declaration_1
  Declaration_1
  Declaration_Last
  Executable_Statement_1
  Executable_Statement_2
  Executable_Statement_Last
}
```

Must include one or more return statements
Precede the function call with either
- The function declaration
or
- The function definition itself

If the definition appears before the function call, then
- A function declaration is not required
Placing of Function Definitions

Example

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

double total_cost (int number_par, double price_par)
{
    const double TAX_RATE = 0.05;
    double subtotal;
    subtotal = price_par*number_par;
    return (subtotal + subtotal*TAX_RATE);
}

//Computes total cost, including 5% sales tax..
int main ( )
{
    double price, bill;
    int number;
    cout << “Enter the number of items purchased: “;
    cin >> number;
    cout << “Enter price per item: “;
    cin >> price;
    bill = total_cost (number, price);
    cout << number <<", items\n<<“Final bill, incl tax is $” << bill <<endl;
    return 0;
}
```
Procedural Abstraction
Programming

- **Procedure**
  - General term for “routines”, “sub-routines”, “functions”

- **Abstraction**
  - Implies taking away the behavior of the procedure without knowing the details (e.g., algorithm) of the Procedure
Procedural Abstraction
Functions

Basic Idea

- How to write and use functions without knowing the details (algorithms) of the functions

What user needs to know:

- A driver simply checks the speedometer to know how fast he/she is driving

No need to know about the axle, transmission system, wheels etc.
1. The Function Declaration Comment should describe:
   - All conditions required of arguments to the function
   - The returned value

2. Any Variables used in the function (i.e., between the start and end braces), besides the formal parameters, should be declared in the function body

Write functions so the declaration and comment are all a programmer needs to use the function
Functions
Local Variables

- Variables declared in a function:
  - Are local to that function, they cannot be used from outside the function
  - Their scope is defined within the start and end function braces

- Variables declared in the main part of a program:
  - Are local to the main part of the program, they cannot be used from outside the main part
  - Have the main part as their scope
Global Constants -- constants that are available to

- main( ) and
- More than one functions

To specify a Global Constant, you should declare the constant:

- Outside the main function body
- Outside any function body
- Before any function that uses it

Example:

```c
const double PI = 3.14159;
double volume (double);
int main( )
{
    ........
}

PI is available to the main function and to function volume
```
Global Variables - variables that are available to more than one function

- When multiple functions need to use a common variable!

To specify a Global Variable, you should declare the variable:

- Outside the main function body
- Outside any function body
- Before any function that uses
Functions

Global Variables - Example

double sales;
double volume (double);
int main()
{
    ........
}

sales is available to the main function
and to function volume

Global Variables make programs more difficult to understand and maintain
Formal Parameters
Local Variables

- Formal parameters specified in function definitions can be viewed as local variables for the given function
  - No need to re-declare them in the function
  - They are different from other variables with identical names

- The arguments in a function call are used to initialize the formal parameters
Functions
Namespaces Revisited

- Place statement: `using namespace std;`
  - Inside the function starting braces
  - Benefits:
    - Allows the use of different namespaces in different functions
    - Makes the “using” directive local to the function

Refer to display 4.14
Functions

Overloading

- Overloading a Function:
  - To specify more than one declaration and definition for a unique function name
  - Convenient if:
    - The same function is needed for different number of arguments and/or argument types
Functions
Overloading Example

```cpp
double ave(double n1, double n2)
{
    return ((n1 + n2) / 2);
}

double ave(double n1, double n2, double n3)
{
    return ((n1 + n2 + n3) / 3);
}
```

Compiler checks the number and types of arguments in the `function call` to decide which function to use.

`cout << ave(10, 20, 30);`

uses the second definition
Overloading Rules

- Overloaded functions should:
  - Have different numbers of formal parameters AND / OR
  - Have at least one different type of parameter
  - return a value of the same type
Multiple Function Defines Pitfall

What happens if you define the same function name twice with:

- Same number of parameters
- Different parameter types?

```c
double mpg(double miles, double gallons)
{
    return (miles / gallons);
}

int mpg(int missed, int goals)
// computes the number of shots
{
    return (missed + goals);
}
```

Determine output:
`total = mpg(10, 2)`

Is this Overloading?
Multiple Function Defines Pitfall

- The *compiler will select the function, whose parameter types match the argument types

This is not this Overloading?

*ANSI