Functional Decomposition with \texttt{void} Functions

- A \texttt{void} function does not return a function value, and \textit{it is not called within an expression}.

- Recall: value returning functions return one function value only and \textit{MAY be called within an expression}.

- Any module can be coded as a function, although some modules are so simple they are usually not implemented as functions.

- Functions are used to make a program easier to read.
Functional Decomposition with **void** Functions

- A **void** function looks like the value returning **main** function except for two changes: use **void** instead of **int**, and typically there is no **return** statement

```plaintext
int main ()
{
    statements
    return 0;
}
```

```plaintext
void MyFunction ()
{
    statements
    return; // this statement is optional
}
```

Terms Associated with Functions

- **Function Declaration** – may or may not contain the function body
  - Tells the compiler the **name of the function**, the **return type** (void or some **DataType**) and the **DataType of the parameters** it uses
  - Functions - just like variables - have to be declared before they can be used

- **Function Prototype** - A function declaration without the body of the function. Usually placed above main
  - **DataType** **MyFunction** (**[ParameterList]**); // **ParameterList** is optional
  - **DataType** is **void** or some other **DataType**(i.e. **int**, **float**, etc)
  - **ParameterList** for Function Prototype:
    - **DataType**, **ParameterName**, **DataType**, **ParameterName**, etc.
  - **DataTypes of parameters are required, but the parameter names are optional**
More on Function Terms

• **Function Heading** - First line of the function - contains the name of the function which is the identifier of the function, the return type and parameter declarations - very similar to the function prototype

• **Function definition** - the code that extends from the function heading to the end of the block that is the body of the function. A function declaration that includes the function body

  ```cpp
  DataType MyFunction ( ParameterList ) { function body }
  ```

  • **DataType** is `void` or some other `DataType` (`int`, `float`, etc)
  • **ParameterList** is optional
  • **ParameterList for Function Definition:**
    ```cpp
    DataType & ParameterName , DataType & ParameterName ....
    ```
  • Causes compiler to allocate memory
  • **DataTypes AND names of parameters are required**

More on Function Terms

• **Function Call** -
  • The statement that transfers control to a function - the statement that executes the function.
  • A function call passes arguments to the parameters of the function, and the arguments are passed to the parameters according to position from left to right.
  • Must have the same number of arguments in a function call as there are parameters in the function heading

• **FunctionCall For void Functions:**

  ```cpp
  MyFunction( ArgumentList ); //ArgumentList is optional
  ArgumentList: Expression , Expression ....
  ```

  Expression can be a literal, constant, variable or expression
Example

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

void Area (float radius); // Function prototype

int main ()
{
    float radius;
    cin >> radius;
    while (cin) // As long as the input stream does not enter fail state, perform the loop
    {
        Area (radius); // function call
        cin >> radius; // Keep entering in a radius to check. CTRL-D to exit
    }
    return 0;
}

void Area (float rad) // function definition
{
    cout << "Circle of Radius " << rad << " has area of " << 3.14159*rad*rad << endl;
}

See Ch_08_01.cpp for C++ example of this program
```

More on Function Terms - Example

```cpp
float Average (float first, float second, int number) // function definition
{
    float num1, num2;
    float result; int count;
    result = Average (num1, num2, count); // function call
    function name arguments
}
```

CPE 112 Functions – Part 1 - Slide 7

CPE 112 Functions – Part 1 - Slide 8
More on Function Terms

- **FunctionCall for Value Returning Functions:**
  
  SomeVar = MyFunction(ArgumentList); //ArgumentList is optional

  SomeVar = SomeVar2*MyFunction(ArgumentList);

  Here, SomeVar and SomeVar2 represent a variable that is of the same DataType as the function value returned by MyFunction

 Parameters: Value and Reference

- **Value Parameters**
  
  - Declared without an & (ampersand) at the end of the DataType name

  - Value parameters receive a COPY of the value of the corresponding argument - **memory must be allocated**

  - Arguments passed to value parameters can be constants, literals, variables or arbitrarily complicated expressions - provided the **DataType of the argument matches the DataType of the value parameter** declared in the function prototype
Value Parameters Continued

• **Value Parameters continued:**

  • When a function exits (returns), the contents of its value parameters are destroyed (and so are the contents of the local variables)

  • Value parameters are automatically initialized to the values of the corresponding arguments.

  • Example function prototype with value parameters
    
```
    float average (float, float);
```

  See Ch_08_02.cpp for Examples of Value Parameters

Reference Parameters

• **Reference Parameters**

  • Declared with the & (ampersand) at the end of the DataType name. Can be used with void or value returning functions

  • ** A parameter that receives the location (memory address) of the argument - not its value - memory is not allocated and there is only one copy of the information that is used by both the caller and the called function

  • Arguments passed to reference parameters can only be variables and the DataType of the argument must match the DataType of the reference parameter declared in the function prototype
Reference Parameters (continued)

- **Reference Parameters Continued**
  - When a function exits (returns), the caller of the function will find whatever changes (if any) were made to the argument associated with the reference parameter.
  
- Provide a way to return more than one value from a function.
  
- **Input and output stream variables must be passed by reference**
  
- Example Prototype: float average (float&, float&);

See Ch_08_03.cpp for Examples of Reference Parameters

Memory Usage of Parameters

- **Value Parameters** – Receive a copy of the value of the corresponding argument. Two areas in memory hold values – one for the argument and one for the parameter. Memory declared for the parameter.

- **Reference Parameters** – Receive the memory address of the corresponding argument. One memory location only that is accessed by the argument and the parameter. No memory declared for the parameter.
Example - Parameterless void Function

```cpp
#include <iostream> // function definition
using namespace std;

void PrintTrailer(); // function prototype
void PrintHeader(); // function prototype
int main () { // void functions
    PrintHeader(); // function call
    // rest of program statements
    PrintTrailer(); // function call
    return 0;
}

void PrintHeader () { // function definition
    cout << "***** Start of output *************\n";
    return; // can use a return with no value if we want to in void functions
}

#include <iostream>
using namespace std;

int main() { // void functions
    int width = 5, height = 5;
    MakeBox(width, height);
    MakeBox(width * 3, 10);
    return 0;
}

void MakeBox(int wid, int hgt) { // function definition
    int loop1, loop2;
    for (loop1 = 1; loop1 <= hgt; loop1++) {
        for (loop2 = 1; loop2 <= wid; loop2++)
            cout << "*";
        cout << endl;
    }
}

See Ch_08_04.cpp for C++ example of this program
```

Example - Value Parameter void Function

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

void MakeBox(int /*width*/, int /*height*/); // function prototype
int main() { // void functions
    int width = 5, height = 5;
    MakeBox(width, height);
    MakeBox(width * 3, 10);
    return 0;
}

void MakeBox(int wid, int hgt) { // function definition
    int loop1, loop2;
    for (loop1 = 1; loop1 <= hgt; loop1++) {
        for (loop2 = 1; loop2 <= wid; loop2++)
            cout << "*";
    }
}

See Ch_08_05.cpp for C++ example of this program
```
# Example - Reference Parameter void Function

```cpp
#include <iostream>
#include <fstream> // for file I/O
using namespace std;

void OpenInputFile(ifstream& /*in/out*/, string& /* in/out */); // function prototype

int main()
{
    ifstream inFile;
    string name;
    OpenInputFile(inFile, name); // function call to obtain the name of an input file
    if (!inFile) // if inFile is false, then it was not able to be opened, so an error
        { cout << "error in file name
"; return 1; } // exit main with an error code of 1
    cout << "File: " << name << " is a valid file
";
    return 0;
}

void OpenInputFile(ifstream& inFile, string& name) // function definition
{
    cout << "Enter the name of the file to open:");
    cin >> name;
    inFile.open(name.c_str());
}

See Ch_08_06.cpp for C++ example of this program
```

---

# Example - Value Returning Functions

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

// function prototypes
void InputDim(float& /*in/out*/, float& /* in/out */);
float Diagonal(float wid /* in */, float hgt/* in */);

int main()
{
    float width,height,diagonal,area;
    // function call to obtain the dimensions
    InputDim(width,height);
    area = width*height;
    diagonal = Diagonal(width,height); // value returning with value parameters
    cout << "Rectangle with width: " << width << " and Height: " << height << ", has an area of ";
    cout << area << " and a diagonal of " << diagonal << endl;
    return 0;
}

float Diagonal(float width, float height)
{
    float diag = sqrt(width*width + height*height);
    // show scope of values
    return diag;
}

void InputDim(float& width, float& height)
{
    float diag = sqrt(width*width + height*height);
    // show scope of values
    return diag;
}

See Ch_08_08.cpp for C++ example of this program
```
Local Variables

- **Local Variable** - variables that are accessible within the block or function in which they are declared only. Variables can be declared anywhere as long as it is only done one time and before they are used.
  - Only occupy memory while the area of their declaration is executing - function or block
  - variable declaration in a for loop definition is an example of a local variable declared within a block. i.e. for (int k = 0; k < 10; k++)

- When the block finishes execution or the function returns, the local variables are destroyed

- local variables start with their values undefined, so they must be initialized
  See Ch_08_08.cpp for C++ example of this program

The Return Statement

- Every value returning function must return its function value using a return statement.
  - return value;

- Void functions can have a return statement as well. However there is no function value associated with the return statement.
  - return; // valid only in void functions

- Return statements cause the function to exit immediately and return control to the caller
Header Files

- Header files are included into programs by using the preprocessor directive: `#include <header_file_name>
  
  - `#include <iostream>
  
  - `#include <fstream>`, etc.

- Header files are a series of C++ declarations such as
  
  - Constants
  
  - Classes and objects
  
  - Function prototypes

- Use header files to avoid writing all of the library function prototypes

Designing Functions

- What a function does and how it is invoked define the function interface

- As long as the interface remains the same, a function can be changed without having to change the function call statement

- Function implementation is hidden, and knowledge of the code is not required to use the function - only the interface needs to be known

- Designing a function requires two tasks
  
  - Design of the interface (function heading)
  
  - Design of the implementation (function source code - body)

- Behavior of a function is defined by preconditions and postconditions
  
  - **Preconditions** state what is true upon entry to the function
  
  - **Postconditions** state what is true upon exiting the function
Designing Functions (continued)

• Make note of function Incoming values, Outgoing Values and Incoming/Outgoing values

• Parameter/Argument flow of direction
  • Incoming values - values the function receives from the caller (passed as value parameters)
  • Outgoing values - values the function returns to the caller (passed as reference parameters)
  • Incoming/Outgoing values - values the caller has that the function receives and returns (passed as reference parameters)
    • Modifications to these values by the function are available to the caller of the function

Function Heading

• Example of a function heading showing flow of direction
  void TestFnc( /* in */ int value, /* out */ float& average, /* in/out */ int& sum)

Or another way of showing the function heading
void TestFnc( /* in */ int value, /* out */ float& average, /* in/out */ int& sum)
The assert Library Function

- The assert library function provides a way to test data to ensure that it is valid
- #include <cassert> must be placed in the code with the other #include preprocessor directives
- Using the assert library function:
  
  ```
  assert(logical expression);
  ```
  
  examples of logical expression: a > 0, or num > 0 && num < 20
- if the logical expression IS FALSE, then assert terminates the program and prints out an error message
  ```
  assert(a>0) will terminate when a is <= 0
  assert (num > 0 && num < 20)
  terminates when num <= 0 or num >= 20
  ```
  
  See Ch_08_09.cpp for a C++ example of this feature