Return Values

Today

- Parameter passing
- Return values
- Function prototype

Implementing functions

Example: Calculate the area of a circle

1) Pick a good descriptive name for the function

2) Give a type and name for each parameter

    There will be one parameter for each piece of information the function needs to do its job

3) Specify the type of the return value:

    double areaOfCircle(double radius);

4) Then write the body of the function, as statements enclosed in curly braces { … }

Implementing functions

Example: Calculate the area of a circle

Note: Useful comments at the top: description, parameters, return, algorithm

```
/*
     Computes the area of a circle
     @param radius -- the radius of the circle
     @return the area of the circle
*/
double areaOfCircle(double radius)
{
     const double PI = 3.14;
     double area = PI * radius * radius;
     return area;
}
```

Implementing functions

- How do you know your function works as intended??
  - You should always test the function
  - Write a main() function to do this
  - Let’s test a couple different radii values for our areaOfCircle function and see if it outputs the correct areas

```
int main()
{
   double result1 = areaOfCircle(2);
   double result2 = areaOfCircle(10);
   cout << “A circle with a radius of 2 has area of ” << result1 <<
   endl;
   cout << “A circle with a radius of 10 has area of ” << result2
   << endl;
   return 0;
}
```

Parameter passing

Parameter Passing

- When a function is called, a *parameter variable* is created for each value passed in.
- Each parameter variable is *initialized* with the corresponding parameter value from the call.

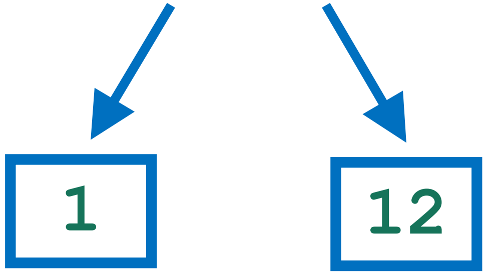
```
int hours = readValueBetween(1, 12);
…
int readValueBetween(int low, int high);
```

Parameter Passing

- When a function is called, a *parameter variable* is created for each value passed in.
- Each parameter variable is *initialized* with the corresponding parameter value from the call.

```
int hours = readValueBetween(1, 12);

. . .
```



```
int readValueBetween(int low   , int high)
```

Parameter Passing

- Example: A call to our areaOfCircle function:

```
      double result1 = areaOfCircle(2);
```

- Here is the function definition:

```
double areaOfCircle(double radius)
{
      const double PI = 3.14;
      double area = PI * radius * radius;
      return area;
}
```

- Let’s keep track of the variables and their parameters:

```
      result1, radius, area
```

Parameter Passing – the play-by-play

- **First,** the function call: `double result1 = areaOfCircle(2);`

→ `result1 = ____      radius = ____`

Parameter Passing – the play-by-play

- **First,** the function call: `double result1 = areaOfCircle(2);`

→ `result1 = ____     radius = ____`

- **Second,** initializing function parameter variable: `double result1 = areaOfCircle(2);`

→ `result1 = ____     radius= 2`

Parameter Passing – the play-by-play

- **Third,** execute `areaOfCircle` function:

```
double area = PI * radius * radius;
return area;
```

→ `result1 = ____`   `radius = 2`   `area = 12.56`

Parameter Passing – the play-by-play

- **Third,** execute `areaOfCircle` function:

```
    double area = PI * radius * radius;
    return area;
```

→ `result1 = ____     radius = 2   area = 12.56`

- **Finally,** after the function call: `double result1 = areaOfCircle(2);`

→ `result1 = 12.56`

Parameter Passing

- In the calling function (`main`), the variable **`result1`** is declared.
- When the **`areaOfCircle`** function is called, the parameter variable **`radius`** is created & initialized with the value that was passed in the function call.
- After the return statement, the local variables `radius` and `area` disappear from memory.
- The calculated volume is stored in the variable, `result1`

Return values

Return Values

The `return` statement ends the function execution. This behavior can be used to handle unusual cases.

What should we do if the side length is negative?

We choose to return a zero and not do any calculation:

```
double areaOfCircle(double radius)
{
     if (radius < 0)
        return 1;
    const double PI = 3.14;
    double area = PI * radius * radius ;
    return area;
}
```

- Nothing is executed after a return statement !!!
- Execution returns to main()

Return Values: Shortcut

The **`return`** statement can return the value of any expression.

Instead of saving the return value in a variable and returning the variable, it is often possible to eliminate the variable and return a more complex expression:

```
double areaOfCircle(double radius)
{
   return 3.14 * radius * radius;
}
```

Common Error – Missing Return Value

Your function always needs to return something.

The code below: what is returned if the call passes in a negative value?

You need to ensure all paths of execution include a `return` statement.

```
double areaOfCircle(double radius)
{
  if (radius >= 0)
  {
     return 3.14 * radius * radius;
  }
}
```

Functions without return values

- Consider the task of writing/printing a string with the following format around it
- Any string could be used
- For example, the string “Hello” would produce:

```
-------
!Hello!
-------
```

Functions without return values – the *void* type

Definition: This kind of function is called a void function

- `void` is a type, just like `int` or `double`
- Use a return type of void to indicate that a function does not return a value
- `void` functions are used to simply perform a sequence of instructions, but not return any particular values to the caller
- Example: `void boxString(string str)`

Calling void functions

- A void function has no return value, so we cannot call it with assignment like this:

```
	result = boxString(“Hello”);
```

// Error: boxString does not return a result

- Instead, we call it like this, without assignment:

```
	boxString(“Hello”);
```

Function Prototype

Function Declarations (Prototype Statements)

- It is a compile-time error to call a function that the compiler does not know
  - just like using an undefined variable.

- So define all functions before they are first used
  - But sometimes that is not possible, such as when 2 functions call each other

Function Declarations (Prototype Statements)

- Therefore, some programmers prefer to include a definition, aka "prototype" for each function at the top of the program, and write the complete function after `main(){}`

- A prototype is just the function header line followed by a semicolon:

```
 double areaOfCircle(double radius);
```

- The variable names are optional, so you could also write it as:

```
    double areaOfCircle(double);
```

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

// Declaration of areaOfCircle
double areaOfCircle(double radius);

int main()
{
   double result1 = areaOfCircle(2); // areaofCircle function call
   double result2 = areaOfCircle(10);
   cout << "A circle with a radius of 2 has an area of "<< result1 << endl;
   cout << "A circle with a radius of 10 has an area of "<< result2 << endl;
   return 0;
}

// Definition of areaOfCircle
double areaOfCircle(double radius)
{
   double area = 3.14 * radius * radius;
   return area;
}
```

Function Declaration (prototype)

Common error: No function declared before encountering function call in `main()`

```
int main()
{
   double area = areaOfCircle(2.0);
}


double areaOfCircle(double radius)
{
   double area = 3.14 * radius * radius;
   return area;
}
```

Steps to Implementing a Function

1. Describe what the function should do.
    - EG: Compute the volume of a pyramid whose base is a square.
2. Determine the function's "inputs".
    - EG: height, base side length
3. Determine the types of the parameters and return value.
    - EG: `double pyramidVolume(double height, double base_length)`
4. Write pseudocode for obtaining the desired result.

```
volume = 1/3 x height x base length x base length
```

5. Implement the function body.

```
{
    double base_area = base_length * base_length;
    return height * base_area / 3;
}
```

6. Test your function
    - Write a `main()` to call it multiple times, including boundary cases

Good Design – Keep Functions Short

- There is a certain cost for writing a function:
  - You need to **design, code, and test** the function.
  - The function needs to be **documented**.
  - You need to spend some effort to make the function **reusable** rather than tied to a specific context.

- So it’s tempting to write long functions to minimize their number and the overhead
- BUT as a rule of thumb, a function that is too long to fit on a single screen should be broken up.

Long functions are hard to understand and to debug

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

/**   Computes the volume of a pyramid whose base is a square.
   @param height the height of the pyramid
   @param base_length length of one side of the pyramid’s base
   @return the volume of the pyramid
*/
double pyramidVolume(double height, double base_length)
{
   double base_area = base_length * base_length;
   return height * base_area / 3;
}

int main()
{
   cout << "Volume: " << pyramidVolume(9, 10) << endl;
   cout << "Expected: 300";
   cout << "Volume: " << pyramidVolume(0, 10) << endl;
   cout << "Expected: 0";
   return 0;
}
```