UNIVERSITY OF KENTUCKY®
College of Engineering

CS 215: Introduction to Program Design, Abstraction and Problem Solving

Chapter 5 Functions and Parameters

Summary

- Functions
- Variable scope
- Call-by-value and call-by-reference

Functions

Lab Assignment 4 will have a few functions you will need to implement. What is a function?

- A sequence of instructions with a name.
- A function is composed of a header, signature or declaration, followed by a body or implementation.

Function we have seen

```
int main()
{
    cout << "Hello, World!" << endl;
    return 0;
}
```

•The header gives the name of the functions and its inputs and outputs.
•The body gives the instructions that make up the function.

Why functions?

Why would we want to use functions?

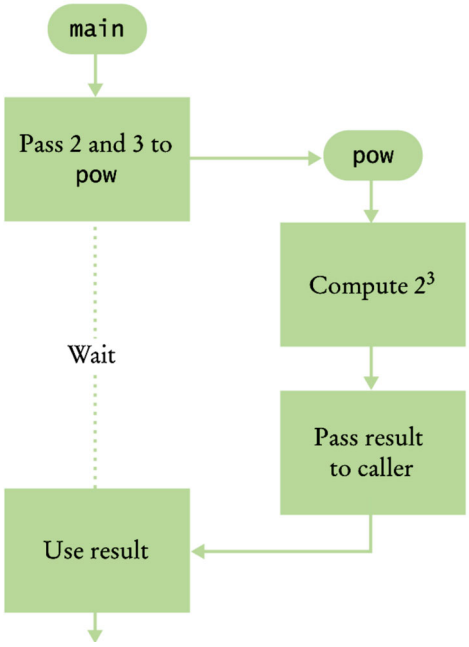
- They make code clearer:
  ```
  if ((year % 4 == 0 && year % 100 != 0) || (year % 400
  == 0))
  if (is_leapYear(year))
  ```
- They allow code re-use.
  - We might need to do the same thing in multiple places.
  - Typing the same code twice is error-prone.
    - If there's a bug, you have to fix it twice.
  - Whenever you write the same thing twice, consider using a function.
  - The third time, you should definitely use a function.

Calling functions

```
double twocubed = pow(2.0, 3.0);
```

- By using the expression `pow(2.0, 3.0)`, main calls the function called `pow`, asking it to compute 2^3.
- The main function is temporarily suspended.
- The body of the `pow` function executes to compute the result.
- `pow` returns its result (8.0) back to main.
- `main` resumes execution, using the returned result as the value of the function call.
  ```
  double twocubed = 8.0;
  ```

Calling functions, illustrated



Parameters

When another function calls `pow`, it provides inputs to it, such as the values 2.0 and 3.0 in the call `pow(2.0, 3.0)`.

- We call these values **parameter values** to avoid confusion with inputs that are provided by a user (`cin >>`).
- The output that the pow function computes is called the **return value** (not output using <<).
- Two types of parameters
  - **Formal parameters**: parameters received by the function definition.
  - **Actual parameters**: arguments provided at the function call.

Defining a function

Suppose we want to write a function to calculate the area of a circle, given its diameter. First, we'll write the signature:

- What will it return? `double`
- Pick a good, descriptive name for the function.
- Base the name on what it does, or what it returns: `calc_area` or `circle_area`.
- Give a (descriptive) name and a type for each parameter. There will be one parameter for each piece of information the function needs to do its job. `double diameter`
- Put it all together: `double circle_area(double diameter)`

Implementation of a function

Now we need to implement our function:
`double circle_area(double diameter)`

- The function body is always a {} block.
- It can use the parameters as variables.
- Other variables have the function body as their scope.
- When you have the answer, use `return` to end the function and send the result back to the caller.

```
double circle_area(double diameter)
{
    const double PI = 3.141592653589;
    double area = PI * pow(diameter/2.0, 2.0);
    return area;
}
```

Function comments

Whenever you write a function, you should comment its behavior.

- Remember, comments are for humans, not compilers.
- No universal standard, but we have one for our CS215.
- At least describe the following:
  - **Purpose**: A short "blurb" on what the function is used for.
  - **Description**: A narrative description of what the function does.
  - **Inputs**: Names, types, and meaning of each parameter, and any restrictions on their values.
  - **Outputs**: The meaning and type of the return value.

void functions

Sometimes the function doesn't need to return anything to the caller.

- For example, if the function's job is to print something to the user.
- Use the keyword **void** for the return type.
  `void string_box(string seq) ...`
  - Call the function:
    `string_box(myseq);`
  - Note: `x = string_box(myseq)` is illegal!

Scope and functions

```
double circle_area(double diameter)
{
  double pi = 3.14159;
  double area = pi * pow(diameter/2.0, 2);
  return area;
}
int main()
{
  double area = circle_area(3.0);
  cout << "The area is " << area << endl;
  return 0;
}
```

- Are the occurrences of `pi` the same variable? Yes!
- Are the occurrences of `area` the same variable? No!

Scope and functions

```
int main()
{
  double area = circle_area(2.0);
  double area = circle_area(5.0);
  // ERROR: cannot define another area variable
  ...
}
```

- The scope of a variable is the part of the program in which it is visible.
- It is not legal to define two variables with the same name in the same scope.
- However, you can define another variable with the same name in a **nested block**.
- A variable in a nested block **shadows** a variable with the same name in an outer block.Potentially confusing situation!

Variable Scope Example

```
int main()
{
  int i = 1;
  cout << i << endl;
  {
    int j = 10;
    cout << i << j << endl;
    i = 2;
    cout << i << endl;
  }
  cout << i << endl;
}
```


Variable Scope Example

```
int main()
{
  int i = 1;
  cout << i << endl;
  {
    int j = 10;
    cout << i << j << endl;
    int i = 2;
    cout << i << endl;
  }
  cout << i << endl;
}
```

Scope and functions

- Each function has its own **scope**.
- The function's parameters, and variables defined in its block, are **local** to that function.
  - Other functions can't access them.
  - Other functions can have variables or parameters with the same names.
  - Local variables come into existence when you call the function...
  - ...and disappear when the function returns.
  - Every time you call the function, you get brand-new variables.
- Note that parameters **are** local variables, and share the same scope:

```
void paint(string color) {
  string color = "blue";
}
```

— Error: variable redefinition!