Functions

CS A150 - C++ Programming 1

FUNCTIONS

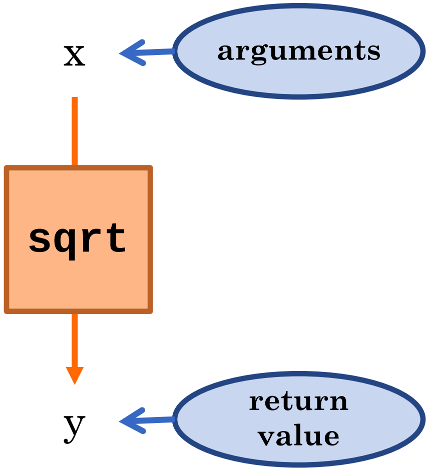
- **Functions** are fundamental building blocks of a programming language
- Other terminology in other languages:
  - ***Procedures***, ***subprograms***, ***methods***
  - In C++: **functions**
- Easily re-used
- Easy to debug
- The idea:
  - Break up a complex task into smaller, simpler tasks

FUNCTIONS (CONT.)

- You have already used some functions:

```
sqrt( )
max( )
```

- Don't need to know how either does the job
- A function is a ***black box***
  - You pass in values (**arguments**)
  - Function returns an answer or performs an action (or both)
  - **Input → Process → Output**



Functions (cont.)

- ***Caller*** calls function
  - For example, if the call to the function is in **`main()`**, then the **`main()`** function becomes the ***caller***
  - You pass an ***argument*** (some value) to **`sqrt`**
    - Can be *any* expression

```
sqrt ( b·b + 4·a c )
```

  - The ***caller*** is suspended
  - **`sqrt()`** becomes active and computes the answer
  - Returns answer to caller
  - ***Caller*** continues

Functions that Return a Value

- Libraries are full of functions for our use
- Two types:
  - Those that return a value
  - Those that do not (**`void`**)
- Must "**`#include`**" appropriate header
  - e.g.,
    - **`<cmath>, <cstdlib>`** (Original "C" headers)
    - **`<iostream>`** (for **`cout`**, **`cin`**)

Functions that Return a Value (cont.)

- **Math** functions
  - Found in library **`<cmath.h>`**
  - Most return a value (the "answer")
- Example:

```
        double theRoot = sqrt(9.0);
```

```
theRoot  = variable of type double to store "answer”
sqrt     = name of library function
9.0      = argument or "starting input" for function
```

Functions that Return a Value (cont.)

- **`pow(x, y)`**
  - Returns x to the power y

```
double result;
double x = 3.0,
       y = 2.0;
result = pow(x, y);
cout << result;
```

- Notice this function receives ***two*** arguments
  - A function can have any number of arguments of varying data types.

The Function Call

- Back to this assignment:

```
double theRoot = sqrt(9.0);
```

- The expression "`sqrt(9.0)`" is known as a ***function call***, or ***function invocation***
- The ***argument*** in a function call (`9.0`) can be a ***literal***, a ***variable***, or an ***expression***
- The call itself can be part of an expression:

```
double bonus = sqrt(sales/10);
```

Book Example

- Example 1: A predefined function that returns a value.

PREDEFINED VOID FUNCTIONS

- A **void function**
  - Does **not** return a value
  - Performs an action, but sends no "answer“ back
  - When called, it is a statement itself

    `exit(1);` // No return value

    - This call terminates the program
  - Can still have **arguments**

PROGRAMMER-DEFINED FUNCTIONS

- You can write your own function
- Building blocks of programs
  - **"Divide and conquer"**
  - **Readability**
  - **Re-use**
- Your function can go in either:
  - Same file as **`main()`**
  - Separate files so other can use it too

Components of Function

- Functions have three components:
  - Function **declaration/prototype**
    - Information for compiler
    - To properly interpret calls
  - Function **definition**
    - Actual implementation/code for what function does
  - Function **call**
    - Transfer control to function

Function Declaration

- Also called function **prototype**
- An "informational" declaration for the compiler
- Tells compiler how to interpret calls
  - Syntax:

```
returnType functionName(parameters);
```

  - Example:

```
double totalCost (int numberOfItems, double price);
```

  - You can *omit* parameter names:

```
double totalCost (int,  double );
```

- Placed *above* **`main()`** in global space

Function Definition

- Implementation of function
- Just like implementing function **`main()`**
- Example:

```
 double totalCost(int numberOfItems, double price)
{
     double subTotal;
     subtotal = numberOfItems * price;
     return (subtotal);
}
```

- Placed *after* function **`main()`**
- **`return`** statement
  - Sends data ***back*** to caller

Function Definition (cont.)

- Avoid creating variables when not needed:

```
double totalCost(int numberOfItems, double price)
{
        double subTotal;
        subtotal = numberOfItems · price;
        return (subtotal);
}
```

- Can be simplified using one statement only:

```
double totalCost(int numberOfItems, double price)
{
        return (numberOfItems · price);
}
```

  - **No** need to create a variable

FUNCTION CALL

- Just like calling predefined function

```
double bill = totalCost(numberOfItems, price);
```

- Recall: **`totalCost`** returns double value
  - Assigned to variable named "**`bill`**"
- Arguments: **`number`**, **`price`**
  - Recall **arguments** can be literals, variables, expressions, or a combination of any of the above

Book Example

- Example 2: A programmer-defined function that returns a value.

Functions Calling Functions

- We are already doing this!
  - **`main()`** IS a function!
- *Only* requirement:
  - **Function's declaration** must appear ***first***
- Function's **definition** typically elsewhere
  - After **`main()`**'s definition
  - Or in separate file
- Common for functions to call many other functions
- Function can even call itself → "**Recursion**"

Boolean Return-Type Functions

- **Return-type** can be any valid type

```
bool negative(int x)
{
     if (x < 0)        // avoid long form!!!
         return true;
     else
        return false;
}
```

- You should simplify it:

```
bool negative(int x)
{
     return (x < 0);
}
```

Declaring Void Functions

- Similar to functions returning a value
- Return type specified as "`void`"
- Example:
  - **Function declaration/prototype**:

```
void showResults( double fDegrees, double cDegrees );
```

- Return-type is "`void`"
- Nothing is returned

Declaring Void Functions (cont.)

- Function **definition**:

```
void showResults ( double fDegrees, double cDegrees )
{
     cout.setf ( ios::fixed );
     cout.setf ( ios::showpoint );
     cout.precision (1);

     cout    << fDegrees
             << " degrees fahrenheit equals \n"
             << cDegrees << " degrees celsius.\n";
}
```

- **Note: no** return statement.

Calling Void Functions

- Same as calling predefined **void** functions
- From some other function, like main() :

```
showResults ( degreesF, degreesC );
showResults ( 32.5, 0.3 );
```

- Notice ***no*** assignment, since ***no*** value returned
- Actual arguments (degreesF, degreesC)
  - Passed to function
  - Function is called to "do its job" with the data passed in

Book Example

- Example 3: Use of return in a **`void`** programmer-defined function.

Comments

- Comment can be of the type *precondition/postcondition*:

```
void showInterest (double balance, double rate);
//Precondition: Balance is nonnegative account balance
//            rate is interest rate as percentage.
//Postcondition: Amount of interest on given balance,
//            at given rate …
```

- We will use a different set of comments

```
void showInterest (double balance, double rate);
//showInterest – What the function does.
//@param double – The balance...
//@param double – The rate...
```

- You can find details on how to write comments on the **syllabus**.

main(): A "Special“ Function

- Recall: main() IS a function
- "Special" in that:
  - One and only one function called **main()** will exist in a program.
- Who calls **main()**?
  - **Operating system**
  - Tradition holds it should have **return** statement
    - Value returned to "caller" → Here: **operating system**
  - Should return "**int**"

Scope Rules

- **Local variables**
  - Declared *inside* body of given function
  - Available *only* within that function
- Can have variables with ***same*** names declared in ***different*** functions
  - **Scope** is local → the function is the scope
- Local variables are **preferred**
  - Maintain individual control over data
  - Functions should declare whatever local data is needed to "do the job"

Global Constants and Variables

- **Global declarations** typical for **constants**:

```
const double TAX_RATE = 0.05;
```

  - Declare globally so ***all*** functions have scope (that is, all functions have access to it)

- Global variables?
  - Possible, but SELDOM USED
  - Dangerous: **no** control over usage!

Book Example

- Example 4: A Global Named Constant

PROCEDURAL ABSTRACTION

- Need to know "**what**" function does, **not** "**how**" it does it!
- Think "black box"
  - Device you know how to use, but not its method of operation
- Implement functions like black box
  - User of function (other programmer)
    - Only needs the function declaration
    - Does NOT need function definition
  - Called **Information Hiding**
  - Hide details of "how" function does its job

Blocks

- Declare data inside compound statement
  - Called a "block"
  - Has "block-scope"
- **Note:** all function definitions are blocks!
  - This provides local "function-scope"
- Loop blocks:

```
for (int i= 0; i < 10; ++i)
{
        sum += i;
}
```

  - Variable **i** has scope in loop body block ***only***

Nested Scope

- ***Same*** name variables declared in multiple blocks
- Very legal; scope is "block-scope"
  - No ambiguity
  - Each name is *distinct* within its scope

```
int someFunction (int someValue)
{
    for (int idx = 0; idx < 10; ++idx)
        // do something…
    for (int idx = 2; idx < 20; ++idx)  //same var: int idx
        // do something else…
}
```

To Sum It Up…

- Two kinds of functions:
  - "**Return-a-value**" and **void** functions
- Functions should be "black boxes"
  - Hide "**how**" details
  - Declare own **local** data
- Function declarations should **self-document**
  - Provide all "caller" needs for use
  - We will use comments as shown in the **syllabus**

To Sum It Up… (cont.)

- **Local** data
  - Declared in function definition
- **Global** data
  - Declared above function definitions
  - OK for constants, **not** for variables
- **Parameters** / **Arguments**
  - In function **declaration** and **definition**
    - Placeholder for incoming data
  - In function **call**
    - Actual data passed to function

Random Numbers

- Return "randomly chosen" number
- Used for simulations and games
  - `rand()`
    - Takes no arguments
    - Returns value **between 0 & RAND_MAX** (typically 32767, or 2147483647)
    - Found in `cstdlib` header
  - Scaling → `rand() % 6`
    - Squeezes random number into smaller range
    - Returns random value between 0 and 5
  - Shifting → `rand() % 6 + 1`
    - Shifts range between 1 and 6 (e.g., die roll)

Random Number Seed

- **Pseudorandom** numbers
  - Calls to `rand()` produce given "sequence" of random numbers
- Use "seed" to alter sequence

```
srand(seed_value);
```

  - `void` function
  - Receives one argument, the "seed"
  - Can use any seed value, including system time:

```
srand(time(0));
```

  - `time()` returns system time as numeric value
  - Library `<ctime>` contains `time()` functions

Random Examples

- Random *double* between 0.0 and 1.0:

```
(RAND_MAX – rand()) / static_cast<double>(RAND_MAX)
```

  - Type cast used to force double-precision division

- Random *int* between 1 and 6:

```
rand() % 6 + 1
```

  - "%" is *modulus operator* (remainder)

- Random *int* between 10 and 20:

```
rand() % 11 + 10
```

Example

- Examples:
  - Random
  - Random Seed

QUESTIONS?

(Functions)