# Functions

Lecture 6 COP 3014 Spring 2022

February 21, 2022

◆□▶ ◆□▶ ◆臣▶ ◆臣▶ 臣 のへぐ

#### Functions

A function is a reusable portion of a program, sometimes called a *procedure* or *subroutine*.

- Like a mini-program (or subprogram) in its own right
- Can take in special inputs (arguments)
- Can produce an answer value (return value)
- Similar to the idea of a function in mathematics

With functions, there are 2 major points of view

- Builder of the function responsible for creating the declaration and the definition of the function (i.e. how it works)
- Caller somebody (i.e. some portion of code) that uses the function to perform a task

# Why write and use functions?

#### Divide-and-conquer

- Can breaking up programs and algorithms into smaller, more manageable pieces
- This makes for easier writing, testing, and debugging
- Also easier to break up the work for team development
- Reusability
  - Functions can be called to do their tasks anywhere in a program, as many times as needed
  - Avoids repetition of code in a program
  - Functions can be placed into libraries to be used by more than one "program"

# Using Functions

- The user of a function is the **caller**.
- Use a function by making calls to the function with real data, and getting back real answers.
- Consider a typical function from mathematics:
  f(x) = 2x + 5
- In mathematics, the symbol 'x' is a placeholder, and when you run the function for a value, you "plug in" the value in place of x. Consider the following equation, which we then simplify:

у	=	f(10)		//	must	evaluate f(10)
у	=	2 * 10 + 5		//	plug	in 10 for x
у	=	20 + 5				
у	=	25	//	so	f(10)	results in 25

In programming, we would say that the call f(10) returns the value 25.

# Using Functions

 C++ functions work in largely the same way. General format of a C++ function call:

```
functionName(argumentList)
```

- The argumentList is a comma-separated list of arguments (data being sent into the method).
- Use the call anywhere that the returned answer would make sense.
- In keeping with the "declare before use" policy, a function call can be made ONLY if a declaration (or definition) of the function has been seen by the compiler first.
  - This can be done by placing a declaration above the call
  - This is handled in libraries by including the *header* file for the library with a #include directive

# Using Functions

There is a pre-defined math function "sqrt", which takes one input value (of type double) and returns its square root. Sample calls:

double x = 9.0, y = 16.0, z;

z = sqrt(36.0); //returns 6.0 (stored in z)

z = sqrt(x); //returns 3.0 (stored in z)

z = sqrt(x + y); //returns 5.0(stored in z)

cout<<sqrt(100.0);// prints the returned 10.0
cout<<sqrt(49);</pre>

//due to automatic type conversion rules we
// can send an int where a double is
// expected. This call returns 7.0

# Predefined Functions

- There are many predefined functions available for use in various libraries.
  - ► These typically include standard libraries from both C and C++
  - These may also include system-specific and compiler-specific libraries depending on your compiler
  - Typically, C libraries will have names that are prefixed with the letter 'c'. (cmath, cstdlib, cstring)
- To make such functions available to a program, the library must be included with the #include directive at the top of your file. Examples:

```
\texttt{\#include} \ < \texttt{iostream} >
```

```
\texttt{#include} \ < \texttt{cmath} >
```

```
#include <cstdlib>
```

- // common I/O routines
- // common math functions
- // common general C  $\,$
- // functions

# **Building Functions**

- The builder of a function (a programmer) is responsible for the declaration (also known as prototype) and the definition.
- ► A function declaration, or prototype, specifies three things:
  - the function name usual naming rules for user-created identifiers
  - the return type the type of the value that the function will return (i.e. the answer sent back)
  - the parameter list a comma separated list of parameters that the function expects to receive (as arguments)
    - every parameter slot must list a type (this is the type of data to be sent in when the function is called)
    - parameter names can be listed (but optional on a declaration)
    - parameters are listed in the order they are expected
- Declaration Format:

return-type function-name( parameter-list );

#### Examples:

// GOOD function prototypes

int Sum(int x, int y, int z);

double Average (double a, double b, double c);

bool InOrder(int x, int y, int z);

int DoTask(double a, char letter, int num);

```
double Average (double, double, double);
// Note: no parameter names here
// okay on a declaration
```

```
// BAD prototypes (i.e. illegal)
```

double Average(double x, y, z);
 // Each parameter must list a type

PrintData(int x); // missing return type

int Calculate(int) // missing semicolon

int double Task(int x);
 // only one return type allowed!

# Defining a Function

- a function definition repeats the declaration as a header (without the semi-colon), and then adds to it a function body enclosed in a block
  - The function body is actual code that is implemented when the function is called.
  - In a definition, the parameter list must include the parameter names, since they will be used in the function body. These are the formal parameters.
- Definition Format:

```
return-type function-name( parameter-list )
{
```

function-body (declarations and statements)

To send the return value out, use the keyword return, followed by an expression that matches the expected return type

return expression;

## Definition Examples:

```
int Sum(int x, int y, int z)
// add the three parameters and return the sum
ł
     int answer;
     answer = x + y + z;
     return answer;
}
double Average (double a, double b, double c)
// add the parameters, divide by 3, return the result
ł
     return (a + b + c) / 3.0:
}
```

More than one return statement may appear in a function definition, but the first one to execute will force immediate exit from the function.

```
bool InOrder(int x, int y, int z)
/* answers yes/no to the question "are these
parameters in order, smallest to largest?"
Returns true for yes, false for no. */
{
    if (x <= y && y <= z)
        return true;
    else
        return false;
}</pre>
```

# Scope of Identifiers

- The scope of an identifier (i.e. variable) is the portion of the code where it is valid and usable
- A global variable is declared outside of any blocks, usually at the top of a file, and is usable anywhere in the file from its point of declaration.
  - "When in doubt, make it global" == BAD PROGRAMMING PRACTICE
  - Best to avoid global variables (except for constants, enumerations. Sometimes)
  - Function names usually global. (prototypes placed at the top of a file, outside any blocks)

# Scope of Identifiers

- A variable declared within a block (i.e. a compound statement) of normal executable code has scope only within that block.
  - Includes function bodies
  - Includes other blocks nested inside functions (like loops, if-statements, etc)
  - Does not include some special uses of block notation to be seen later (like the declaration of a class – which will have a separate scope issue)
- Variables declared in the formal parameter list of a function definition have scope only within that function.
  - These are considered local variables to the function. Variables declared completely inside the function body (i.e. the block) are also local variables

# void functions and empty parameter lists

- Parameter lists
  - Mathematical functions must have 1 or more parameters
  - C++ functions can have 0 or more parameters
  - To define a function with no parameters, leave the parintheses empty
  - Same goes for the call. (But parintheses must be present, to identify it as a function call)
- Return Types
  - A mathematical function must return exactly 1 answer
  - ► A C++ function can return **0 or 1** return value
  - To declare a function that returns no answer, use void as the return type
  - A void function can still use the keyword return inside, but not with an expression (only by itself). One might do this to force early exit from a function.
  - To CALL a void function, call it by itself do NOT put it in the middle of any other statement or expression

### Functions and the compiler

- The reason for the declare-before-use rule is that the compiler has to check all function CALLS to make sure they match the expectations.
  - ▶ the "expectations" are all listed in a function declaration
  - function name must match
  - arguments passed in a call must match expected types and order
  - returned value must not be used illegally
- Decisions about parameters and returns are based on type-checking.
  - legal automatic type conversions apply when passing arguments into a function, and when checking what is returned against the expected return type