Functions

- So far, we have seen programs in which all code resides within a `main` function
- Complex programs consist of sub-programs that perform particular tasks
- In C++, these subprograms are called `functions`
- A function accepts inputs, called `arguments`, and computes a `return value` that is the function's output
Designing a Function

- Decompose application into subtasks
- How do subtasks interact?
  - If subtask A delegates work to subtask B, what data does A need to furnish B?
  - What data does A require from B to complete its task?
- These questions determine a function's declaration
- Implementing each subtask completes function's definition

Calling a Function

- The input values passed to a function are the actual parameters (arguments)
- Inside the function, these values are the formal parameters (parameters)
- A function's definition includes its return type, name, formal parameters with types, and statements (the body)
- Its declaration is all this except the body
- If a function is called before it is defined, then it must be declared first
Procedures vs. Functions

- Many programming languages have *procedures*: subroutines that perform actions but do not return a value.
- They also have distinct subprograms called *functions*, which compute output values from input values.
- In C/C++, *all* subprograms are functions.
- Can specify a return type of `void`, so that no value is returned, and the function is essentially a procedure.

Overloading Functions

- Often functions perform similar tasks on inputs of different types.
- Shouldn't require distinct functions.
- Functions can be *overloaded* with different definitions for different inputs.
- When function is called, compiler determines which definition to use.
- Cannot overload with definitions that have same parameter types but different return types.
The `inline` Keyword

- Many functions have a simple definition, such as
  ```cpp
define printint(int s)
  { cout << s; }
```
- When compiling, the overhead of a function call can be costly, but using a function is still best for modularity
- Preceding declaration with the `inline` keyword asks the compiler to replace calls to `printint` with its code

Pass by Value or Reference?

- Example: in `void f(string x);` the argument `x` is *passed by value*
- In `f(y);` the value of `y` is copied to `x`
- Any change in `x` does not affect `y`
- In `void g(string& str);` the argument `s` is *passed by reference*
- In `g(s);` the *address* of `s` is passed and assigned to `str`, so `str` is a *reference* to `s`
- A change in `str` is reflected in `s`
The **const** Qualifier

- Calling `void f(string& s);` by `f("Hello");` is not valid
- "Hello" is an array of `char`, which supports literals, so allowing this implies a literal's value could be changed!
- Use `void f(const string& s);` to indicate `s` is not changed by `f()`, or create a `string` object and pass that
- `const` helps the compiler optimize code by reducing possible memory writes

The **return** Statement

- If a function is not of type `void`, then it should return a value, using a `return` statement
- Form: `return expr;` where `expr` is the value to be returned
- A `return` statement causes an immediate exit from the function!
- When function exits, local, non-static variables are deallocated, so never `return` a reference to a local object!
**static Variables**

- Normally, variables local to a function are destroyed when the function exits
- If a local variable is declared `static`, it persists throughout execution
- Value is retained, and available when function is called again
- Beware: initialization only happens once, when the function is first called!
- Global static variables can only be used in the file in which they are declared

---

**Default Arguments**

- Suppose we declare, for example, 
  \[
  \text{int } f(\text{int } \text{arg1}, \text{int } \text{optArg} = 0); 
  \]
- If we call \( f \) as follows: \( x = f(3) \); then inside \( f() \), the value of \( \text{optArg} \) is 0
- We call \( \text{optArg} \) a *default argument*
- An argument cannot be a default argument unless all subsequent arguments are also default
- Using defaults reduces overloading
Separate Header Files

- So far, we have declared and defined functions in the same file.
- What if a function $f()$ defined in a file $f.cpp$ needs to call a function $g()$ defined in another file $g.cpp$?
- Solution: declare $g()$ in a header file, for example, $g.h$, and in $f.cpp$, use `#include "g.h"`
- Header files typically have a `.h` extension (sometimes `.hpp`).

Naming New Types

- C++ provides many ways of creating new types from existing ones: pointers to types, arrays of types, functions with return and argument types, etc.
- Types are complicated to describe.
- The `typedef` keyword allows simple names to be given to these types.
- Form (in most cases):
  ```cpp
typedef type-expression type-name;
```
  where `type-name` is the new name.
Uses of `typedef`

- Opaque type names: to hide details about a type from a user, for example:
  ```c
  typedef unsigned int size_t;
  ```
- Descriptive: to convey interpretation of data:
  ```c
  typedef int years;
  ```
- Simpler: to condense complex type expressions such as:
  ```c
  typedef int (*func_type)(int);
  ```
  where `func_type` is a pointer to a function that accepts, and returns, an `int`.

Next Time

All about streams and files:
- Console, file and string streams
- Stream manipulators
- Object persistence
- Project 2 distribution!