Notes #8

Functions

Chapter 6

CMPT 125/128

Topics

1) How can we break up a program into smaller sections?
2) How can we pass information to and from functions?
3) Where can we put functions in our code?

Functions

Functions

- Functions:
  - Each function should perform...
  - Also called methods, or procedures.
  - Ex:
    - hold the window open, display the menu, ...
  - Allows for the divide and conquer approach:
    - Divide: split the big problem down into multiple smaller problems.
    - Conquer:

Function definition

```
// A simple C++ program.
#include <iostream>
using namespace std;

void displayMsg() {
    cout << "Hello world";
}

int main () {
    displayMsg();
    return 0;
}
```

The type of value/information the function returns.

displayMsg.

List of variables to hold values passed into the function.

Statements to carry out the task of the function.

Must have () on call or get msg:

```
warning C4551: function call
missing argument list
```


Function definition

- A function (like a variable) must be
  - For the moment, put the definition of a function earlier (above) in the file than any calls to the function; otherwise will not compile.
- Function Return Type:
  - a specific type (such as int or double); or
  -

Review

- What is the difference between defining a function and calling a function?
- Write a function to display "I code therefore I am."

Getting data in and out of a function.

Function Parameters

Function call (use):

Arguments

```
displaySum(  2,    'a');
```

Function definition:

Parameter List

```
void displaySum(int x, char y) {
     cout << "Sum: " << ( x + y);
}
```

Parameters

- **Arguments:**
- **Parameter List:**
  - Inside the (...) of the function header.
  - May be empty if no parameters required.
- **Parameters:**
  - These are variables inside the method.

Returning a value

- The return statements does 2 things:
  - Causes the current function to exit, returning control to the calling function.
  –

```
/*
     Return the number of points the user scored based
     on the number of zombies killed.
     Returns 0 if number killed is less than 0.
*/
int calcScore(int numZombies) {
     if (numZombies < 0) {
          return 0;
     }
     return numZombies * POINTS_PER_ZOMBIE;
}
```


Review

- Write a function:
  - named add()
  - which accepts 2 float parameters; and
  - returns the sum of the two parameters as a float

Local vs Global Scope

Local variables

- Local variables:
  - Restricted scope (visibility) to within the function.
  - Restricted lifetime to when function is executing.
  - (These Includes function parameters.)
- What's that mean?
  - Cannot use a local variable outside the function.
  - Local variables are...

    Next time through, a new one is created.

Global variables

- Global variables are
  - Accessible anywhere between its definition and the end of the .cpp file.
  - Lifetime is the same as the program.
- Guidelines:
  - Good for constants:
    const int DAYS_PER_WEEK = 7;
  - Often problematic for variables: can be very

    • Use local variables as much as possible.

Scope and variable names

- Scope is
  - Global scope:
  - Local scope to a function: Inside a function.
  - Blocks: Any block {...}, such as for a while loop.
- You *could* reuse a variable name in different nested scopes, but is very confusing!
  - Try and give variables in nested scope unique names.

Static local variables

- Static local variables:
  - scope: accessible inside the local function only;
  - lifetime: ...
    Lifetime same as the program.
    - Only initialized once!
- Example:

```
int getNewNum() {
    static int nextNum = 0;          // Done once!
    return ++nextNum;
}
```

First value returned is...

Scope, Lifetime, and Puppies

Lifetime	Scope: Local	Scope: Global
Temporary		
Persistent		

Pass-by-reference vs
pass-by-value

Pass by value

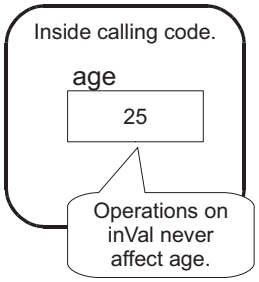
- Normally, only the value of an argument is passed into the function's parameter.

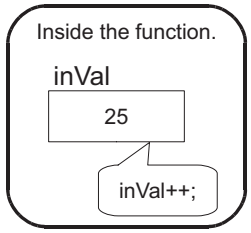
```
void growOlder(int inVal) {
    inVal++;
}

int main () {
    int age = 25;
    growOlder(age);
    cout << "Age is:   " << age << endl;
}
```

- Changing the parameter's value in a function...

Explaining pass by value

- Pass by value:
  function's parameter is set to a copy of argument.
  - Changing the copy does not affect the original.

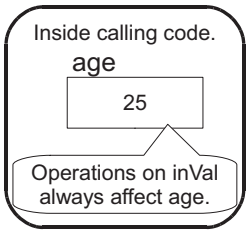


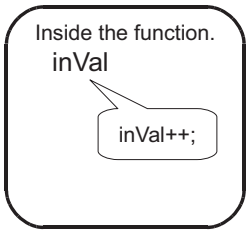


But what if you want to change the value of age when you change inVal?

Explaining pass by reference

- Reference:
  - one variable refers to the actual memory used by the another variable...
- Pass by reference:
  function's parameter refers to the actual argument.
  - Changing the parameter's value...





Pass by reference

- To pass-by-reference, put an & between the parameter's type and name in the parameter list.
  - This makes the function's parameter an alias for the calling argument.

```
void growOlder(int &inVal) {
    inVal++;
}

int main () {
    int age = 25;
    growOlder(age);
    cout << "Age is:   " << age << endl;
}
```

say: "inVal is a reference to an int."

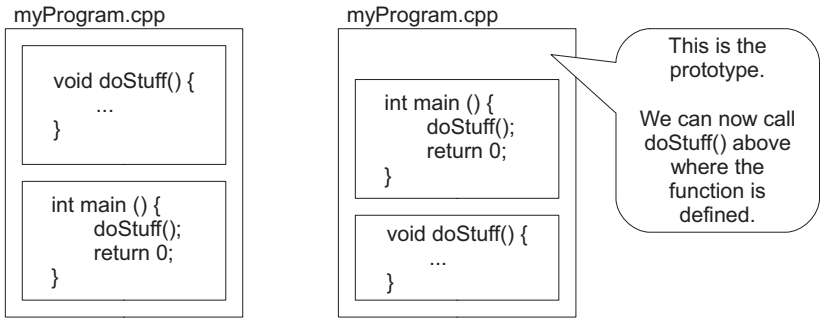
Uses for pass-by-reference

- Useful for passing back multiple values:
  ```
  // Return true if successfully read first and last names.
  // Otherwise, return false.
  bool readName(string &first, string &last);
  ```
- Cautions on Use:
  - Use pass-by-value as much as possible!
  - Use a return value to pass back a single value.
  - Arguments for pass-by-reference...
    - Ex:
      ```
      string a, b;
      readName(a, b);                  // Good
      readName("Hello", "World");      // Compile Error.
      ```


Prototypes

Prototypes

- Must know some things about a function to call it.
  - Function prototypes eliminates the need to put



Needed information to call

- To call a function we need to know:
  -
  - number, type, and order of parameters,
  - return type of function.
- Function prototype idea:
  - Rather than defining the whole function at the top, tell the compile at the top of the .cpp file

Using prototypes

- Function prototype is similar to a function definition except:
  - (place a ';' instead of {...})
  - Do not *need* names for parameters.

```
// Prototype
void printSum(int x, int y);      // or: void printSum(int, int);

int main () {
    printSum(1,2);
    return 0;
}

// Display the sum of the two values.
void printSum(int x, int y) {
    cout << x << " + " << y << " = " << (x + y) << endl;
}
```


Prototype and references

- When using pass by reference, function prototype and definition must have the &.

```
void growOlder(int &inVal);

int main () {
    int age = 25;
    growOlder(age);
    cout << "Age is:   " << age << endl;
}

void growOlder(int &inVal) {
    inVal++;
}
```

Equivalent prototypes:
void growOlder(int &inVal);
void growOlder(int& inVal);
void growOlder(int &);
void growOlder(int&);

Summary

- Function definition: type, name, parameter list, body.
- Function call must use (): int age = getAge();
- Use return to pass back a value.
- Local variables exist only inside the function.
  - Static local variables maintain value between calls.
- Global variables often bad; global constants good!
- Pass-by-...
  - Pass-by-value: pass in just a copy.
  - Pass-by-reference: working on actual argument.
- Use prototypes to define function below a call to it.