C Programming

Functions
Part 2

Functions

Changing(?) Parameters

Changing Parameters

You can change the values of **parameters**

Those changes do not affect the **arguments**

Example

Function call:

- loadData(count, 10);

Function definition:

```
void loadData(int counter, long max)
{
    counter++;  // this doesn't affect the value of count
    // do other code here
}
```

Call-by-value

Called call-by-value

a copy of the value passed as an argument is given to the parameter

Just like changing a copy of a piece of paper doesn't change the original, changing the parameter doesn't change the value used in the argument.

Really Important Thing To Realize!

The arguments passed to a function will not likely have the same name as the parameters they go into.

- But they can have, if that's the way it works out.

Functions

Returning Values

Returning Information From A Function

Use return statements

with values

or expressions

datatype provided in function definition

e.g. int loadData(int count, long max)

- loadData() must return an int value to whatever calls it.
- The int value can be a variable, number, arithmetic expression, or even another function call.

Receiving Returned Values

Receive returned values by assigning to variables

e.g. myValue = loadData(50, 100);

- loadData() would be executed and return a value when done. That value would go into the myValue variable.

Examples of Returning a Value

e.g. return 40;

e.g. return counter;

e.g. return counter + 40;

e.g. return counter * blah();

Example of Returning a Value

```
int loadData(short count, long max)
{
int howMany = 0;
    while( dataAvailable() == TRUE )
    {
        loadOneItem();
        howMany++;
    }
    return howMany;
}
```

howMany gets sent back to the calling function.

e.g. loadedCount = loadData(30, 40);

- The loadedCount variable would contain the value that howMany contained.

Ignoring Return Values

And you can ignore them too

Important Distinction!

Even though you can ignore return values, you can't ignore arguments, since they're required.

- If you do, you'll have a compile error or a bug.

How Many?

The compiler allows as many return statements as you want

You can, however, only return one item (which may have different values depending on which return statement is executed).

e.g.

```
int enoughThere(int count)
{
   if( count > 10 )
        return 1;       // is legal
   else
        return 0;
}
```

Course Requirement

When you return
from a function,
you must return
from only one place

e.g.

```
int enoughThere(int count)
{
int status = 0;
   if( count > 10 )
       status = 1;
   return status;
}
```

If you're not returning a value, then you just automatically return from the bottom anyway

Example: No return At All

```
void isWorldSeriesYear(int value)
{
   if( (value == 1992) || (value == 1993) )
        printf("World Series!\n");
   else
        printf("No world series :-(\n");
}
```

Functions

Prototypes

Protecting Yourself

A compile error is **always** preferable to a bug.

- You always know about compile errors and can fix them.
- You don't know about bugs until you detect your program not working.

You can get protection against missing arguments by using prototypes.

Prototypes

Restate the function header, with a semicolon at the end of it

Located before any function definitions

The compiler uses prototypes to check:

- the number of arguments
- the data types of arguments
- the compatibility of the return type with whatever it is returned into

Example of Using a Prototype

```
int loadData(int count, long max);
/* located before any function definitions */

int main(void)
{
    // put code here
}

int loadData(int count, long max)
{
    // put code here
}
```

Compiler checks argument data types

Compiler checks number of arguments

Compiler checks return data types

Compiler does NOT check names of parameters or arguments

You often get
an error
or
a warning

Visual Studio
is surprisingly bad
at what it checks for

Easiest and Safest Way to Make a Prototype

When you make your function definition, copy and paste the function header in the area in which you'll put your prototypes.

Then put a semicolon at the end of the prototype.

Gotchas With Prototypes

Don't think that you're safe simply because your function is defined before any calls of it (seen on page 206).

Empty Parameter Lists

If you don't have any parameters going into a function, don't just leave it blank.

Use *void* instead.

e.g. int beepSpeaker(void);

That way, your prototypes will be correct.

Empty Argument Lists

If you call a function that takes no parameters, you must still put the empty brackets in.

- e.g. loadData();

This is different from some other languages.

Empty Parameter Lists in Prototypes

If you don't put void in, then the result isn't what you expect.

e.g. int beepSpeaker(); // this is bad

- The compiler interprets this as "I don't care about the parameter list."
- This is **different** from C++.
- You are expected to use C, not C++.

e.g. int beepSpeaker(void); // this is good

Returning Nothing

- The return type can be *void*.

- e.g. void beepSpeaker(void);

The Last Word On Prototypes

Use

Prototypes!

Or Lose 15 marks

Functions

Organizing

Back to Function Definitions

The order the functions appear in matters to **programmers**

Recommended Ordering

Put main() first, then your functions.

Group your functions together as they interrelate

Example of Recommended Layout

```
#include <stdio.h>

// prototypes
void loadData(void);
void processData(void);
void saveData(void);


int main(void)
{
    loadData();
    processData();
    saveData();
}
```

```
void loadData(void)
{
    // put code here
}

void processData(void)
{
    // put code here
}

void saveData(void)
{
    // put code here
}
```

Functions

Naming

Function Naming

The same rules and conventions apply to function naming as applied to variable naming.

SET Code Standard: start functions with lowercase letters.

- Reason: Uppercase letters are typically used for C++ class methods.

Meaningfulness of Function Names

Kent Beck:

"Name your functions based on the intent of the caller. Describe the goal they would seek to fulfil, not the mechanism or algorithm used to fulfil it."

(Paraphrased from http://xunitpatterns.com/Intent%20Revealing%20Name.html)

Functions

Comments

Function Comments

With the exception of main(), all functions you write must have a function comment.

Contents:

- function name
- description
- parameter descriptions
- return values

Example

```
/*
 * Function: displayCosts()
 * Description: This function will display the costs
   associated with the manufacturing process, along
   with desired margins.  It only displays information
   where the cost is non-zero.
 * Parameters: int howMany: how many items will we
   process
 * Returns: int: 0 if there were no items displayed,  -1 if
   there was an error, 1 otherwise
 */
```

Functions

getNum()

getNum()

In order to make input easier for you in assignments, I've created a function called getNum() that you should use to get a number.

It is detailed on the end of the assignment #1b and #2 requirements.

It uses many things we haven't covered yet ... don't worry about that.

Copy from the assignment and paste it at the end of your C source file, with reformatting.

You do need to put that code in your source code file! Otherwise, your program won't work!

Functions

Summary

Functions: In Summary

- Create a function when you want to do something specific. Have it follow your design.
- You need exactly one main() function.
- You define functions by creating them.
- You call functions to use them within other functions.

- You send information to a function through a parameter list (a.k.a. arguments).
- Changing parameter values does not change the values of the arguments that were used in the call.
- You send information back from a function to the caller through a return value.
- Use prototypes for protection.
- All functions need a function comment.