CMPE-013/L

Loop Structures (cont'd)

Maxwell James Dunne
for
do-while
for Loop

Syntax

\textbf{for} (\textit{expression}_1; \textit{expression}_2; \textit{expression}_3) \textbf{statement}

- \textit{expression}_1 initializes a loop count variable once at start of loop (e.g. \texttt{i = 0})
- \textit{expression}_2 is the test condition – the loop will continue while this is true (e.g. \texttt{i <= 10})
- \textit{expression}_3 is executed at the end of each iteration – usually to modify the loop count variable (e.g. \texttt{i++})
for Loop

Flow Diagram

Syntax

```
for (expression_1; expression_2; expression_3)
statement
```

Initialize loop variable

```
i = 0
```

Modify loop variable

```
i++
```

Test loop variable for exit condition

```
i < n
```

TRUE

statement

FALSE

END
for Loop

Example (Code Fragment)

```c
int i;

for (i = 0; i < 5; i++) {
    printf("Loop iteration \%d\n", i);
}
```

Expected Output:

```
Loop iteration 0
Loop iteration 1
Loop iteration 2
Loop iteration 3
Loop iteration 4
```
for Loop

- Any or all of the three expressions may be left blank (semi-colons must remain)
- If \( \text{expression}_1 \) or \( \text{expression}_3 \) are missing, their actions simply disappear
- If \( \text{expression}_2 \) is missing, it is assumed to always be true

**Note**

Infinite Loops
A for loop without any expressions will execute indefinitely (can leave loop via break statement)

```c
for (; ; ) {
    ...
}
```
for Loop

Example (Code Fragment)

FILE *f = fopen("myfile.txt", "r");
char c;
for (c = getc(f); c != EOF; c = getc(f)) {
    printf("Char: '8c' \n", c);
}

Maxwell James Dunne
do-while Loop

Syntax

\[
\text{do statement while (expression);}\
\]

- \textit{statement} is executed and then \textit{expression} is evaluated to determine whether or not to execute \textit{statement} again.

- \textit{statement} will always execute at least once, even if the expression is false when the loop starts.
do-while Loop

Flow Diagram

Syntax

do statement while (expression);

START

statement

expression?

TRUE

FALSE

END
if (test)
  foo();
while (!test) {
  foo();
}

while (test) {
  foo();
}
do-while Loop

Trivial example

Example (Code Fragment)

```c
int i = 0;  // Loop counter initialized outside of loop

do {
    printf("Loop iteration %d\n", i++);
} while (i < 5);  // Condition checked at end of loop iterations

// Expected Output:
Loop iteration 1
Loop iteration 2
Loop iteration 3
Loop iteration 4
Loop iteration 5
```
do-while Loop

Useful example

Example (Code Fragment)

```c
int numInputs;
float input1, input2;

do {
    printf("Enter two numbers:\n");
    numInputs = scanf("%f %f", &input1, &input2);
    fflush(stdin);
} while (numInputs != 2);
```
**break Statement**

**Syntax**

```
break;
```

- Causes immediate termination of a loop even if the exit condition hasn't been met
- Also exits from a `switch` statement
**break Statement**
Flow Diagram Within a `while` Loop

**Syntax**

```plaintext
break;
```
**break Statement**

**Example**

```c
int i = 0;

while (i < 10) {
    i++;
    if (i == 5) {
        break;
    }
    printf("Loop iteration %d\n", i);
}
```

Exit from the loop when $i = 5$. Iteration 6-9 will not be executed.

**Expected Output:**

```
Loop iteration 1
Loop iteration 2
Loop iteration 3
Loop iteration 4
```
While( )

While( )

break
continue Statement

Syntax

```
continue;
```

- Causes program to finish current iteration and begin the next loop
**continue Statement**

Flow Diagram Within a **while** Loop

**Syntax**

```
continue;
```
continue Statement

Example

Example (Code Fragment)

```c
int i = 0;

while (i < 6) {
    i++;
    if (i == 2) {
        continue;
    }
    printf("Loop iteration %d\n", i);
}
```

Expected Output:

```
Loop iteration 1
Loop iteration 3
Loop iteration 4
Loop iteration 5
```

Skip remaining iteration when i = 2.
Iteration 2 will not be completed.
Office Hours

Wednesday 3:30-5PM
CMPE-013/L

Introduction to “C” Programming

Maxwell James Dunne
// declare function

MatrixMath.h

clear Mat(float Mat[3][3])
isPass
Unit testing
Unit testing

- Testing portions of code in isolation
- Normally testing is per function
- Requires input and expected output to be known a priori

\[ 3 + 4 = 7 \]
\[ M \text{Equal}() \]

\[
\begin{align*}
M_1 &= M_1^T \\
M_2 &= M_1 F \\
M_2 + \frac{FP - \Delta T}{2} &= M_2 
\end{align*}
\]

\[ +M_1 -M_2 \]

\[ \text{DET()} = 0 \]

\[ M \text{Equal()} \]

\[ \text{return TRUE;} \]
\[ T = T \cdot f = f \]
Unit testing

Rationale

• Find problems early
  – Before integration

• Simplify testing by only testing small, segmented portions of code

• Test functionality that may not be exposed otherwise

• Find documentation errors

CMPE 118

99/100
6 hours

Maxwell James Dunne
Unit testing
Preparing

• The most important question:

"How am I going to test this?"

• Break code into clean functions with:
  – Clear input
  – Clear output
  – No/minimal side effects
Unit testing
Testing architecture

Known input
f() → Actual output

Expected output

Output matches

Print failure
no → yes
Print success

CMPE-013/L: “C” Programming
Maxwell James Dunne
Unit testing
Testing architecture

Example

// Declare test constants
testInput ← some input
testExpOutput ← precalculated output

// Calculate result
testActOutput ← function result

// Output test results
if testActOutput equals testExpOutput
  output "Test passed"
else
  output "Test failed!"
Unit testing
Trivial example

ExampleLib.c

```c
int AddFive(int x)
{
    return x + 5;
}
```

main.c

```c
#include "ExampleLib.h"

int main(void)
{
    // Declare test constants
    int test1Input = 0;
    int test1ExpOutput = 5;

    // Calculate result
    int test1ActOutput;
    test1ActOutput = AddFive(test1Input);

    // Output test results
    if (test1ActOutput == test1ExpOutput) {
        printf("Test1 passed. \n");
    } else {
        printf("Test1 failed! \n");
    }
}
```
Unit testing
Writing tests

- Write multiple tests
  - At least 1 for every group of inputs
  - Each edge case should have their own test

- Each test should check **one** part of the total functionality
  - One function or logical block of code at a time

Try to break the code you're testing!
Unit testing
Testing framework

- Track how many tests passed/failed
  - Per function
- Track how many functions passed/failed
  - With all tests must pass for the function to pass
- Each test cleanly separated from other tests
  - Both in code and in logic
- Output results
  - Per function/per test results
Unit testing example
Parameter passing

Pass by value

Pass by reference
Parameter Passing

By Value

- Parameters passed to a function are generally passed by value.
- Values passed to a function are copied into the local parameter variables.
- The original variable that is passed to a function cannot be modified by the function since the function has a duplicate of the variable, not the original.
Parameter Passing

By Value

Example

```c
int a, b, c;

int Foo(int x, int y)
{
    x = x + (y++);
    return x;
}

int main(void)
{
    a = 5;
    b = 10;
    c = Foo(a, b);
}
```

The **value** of `a` is **copied** into `x`.
The **value** of `b` is **copied** into `y`.
The function does not change the value of `a` or `b`. 
Parameter Passing
By Value

Example function

```c
int Foo(int x, int y)
{
    int z = x + (y++);
    return z;
}
```

Example main

```c
int main(void)
{
    int a = 6, b = 19;
    Foo(a, b);
    while (1);
}
```
Parameter Passing
By Reference

- Parameters can be passed to a function by reference
- Entails passing around memory address
- The original variable that is passed to a function can be modified by the function since the function knows where the data "lives" in memory
Parameter Passing

ADD \((m_1, m_2, \text{result})\) By Reference

Example function

```c
int Foo(int x[3])
{
    int z = x[2];
    x[1] = 0;
    return z;
}
```

Example main

```c
int main(void)
{
    int a[3] = \{6, 19, -1\};
    Foo(a);
    while (1);
}
```
Matrix

\( \text{Scanf}(\text{"\%d"}, \&c) \)
Scope
Scope
Variables Declared Within a Function

- Variables declared within a **code block** are **local** to that block.

**Example**

```c
int x, y, z;

int Foo(int n)
{
    int a;
    ...
    a += n;
}
```

The `n` refers to the function parameter `n`.

The `a` refers to the `a` declared locally within the function body.
Scope
Variables Declared Within a Function

• Variables declared within a block are not accessible outside that block

Example

```c
int x;
int Foo(int n)
{
    int a;
    return (a += n);
}
int main(void)
{
    x = Foo(5);
    x = a;  // This will generate an error. a may not be accessed outside of the scope where it was declared.
}
```
Scope

Variables Declared Within a Function

- Variables declared within a block are not accessible outside that block

Example

```c
int x;
int main(void)
{
    {
        int a = 6;
    }

    x = Foo(5);
    x = a;  // This will generate an error. `a` may not be accessed outside of the scope where it was declared.
}
```
Scope
And the stack

Example function

```c
int Foo(int x, int y)
{
    int z = x + (y++);
    return z;
}
```

Example main

```c
int main(void)
{
    int a = 6, b = 19;
    Foo(a, b);
    while (1);
}
```
Scope
And the stack

Example function

```c
int Foo(int x, int y)
{
    int z = x + (y++);
    return z;
}
```

Example main

```c
int main(void)
{
    int a = 6, b = 19;
    Foo(a, b);
    while (1);
}
```
Scope
Global versus Local Variables

Example

```c
int x = 5;  // x can be seen by everybody

int Foo(int y)
{
    int z = 1;
    return (x + y + z);  // foo's local parameter is y, foo's local variable is z, foo cannot see main's a, foo can see x
}

int main(void)
{
    int a = 2;  // main's local variable is a
    x = foo(a);  // main cannot see foo's y or z, main can see x
    a = foo(x);
}
```
Scope
Parameters

"Overloading" variable names:

Declared Locally and Globally

```c
int n;

int Foo(int n)
{
    y += n;  // local n hides global n
    ...
}
```

Declared Globally Only

```c
int n;

int Foo(int x)
{
    y += n;  // local n hides global n
    ...
}
```

A locally defined identifier takes precedence
Scope

Parameters

Example

```c
int n;

int Foo(int n)
{
    y += n;
}

int Bar(int n)
{
    z *= n;
}
```

- Different functions may use the same parameter names
- The function will only use its own parameter by that name
#define x 2

void Test(void)
{
  #define x 5
  printf("%d\n", x);
}

void main(void)
{
  printf("%d\n", x);
  Test();
}
Storage Class Specifiers
Scope and Lifetime of Variables

- Scope and lifetime of a variable depends on its storage class:
  - Automatic Variables
  - Static Variables
  - External Variables
  - Register Variables

- Scope refers to where in a program a variable may be accessed
  Lifetime refers to how long a variable will exist or retain its value

Maxwell James Dunne
Storage Class Specifiers

Automatic Variables

**Auto**

- Local variables declared inside a function
  - Created when function called
  - Destroyed when exiting from function

**auto** keyword *usually* not required – local variables are automatically `auto`*

- Typically created on the stack

```c
int Foo(int x, int y)
{
    int a, b;
    ...  
```

*Except when the compiler provides an option to make parameters and locals static by default.*
**Storage Class Specifiers**

`auto` Keyword with Variables

```c
int Foo(auto int x, auto int y) {
  ...
}
```

- **auto** is almost never used
- Many books claim it has no use at all
- Some compilers still use **auto** to explicitly specify that a variable should be allocated on the stack when a different method of parameter passing is used by default
Storage Class Specifiers

Static Variables

- Given a permanent address in memory
- Exist for the entire life of the program
  - Created when program starts
  - Destroyed when program ends
- Global variables are always static (cannot be made automatic using `auto`)

```c
int x;  // Global variable is always static

int main(void)
{
    ...
}
```
Storage Class Specifiers

*static* Keyword with Variables

- A variable declared as *static* inside a function retains its value between function calls (not destroyed when exiting function)
- Function parameters cannot be *static* with some compilers (XC32)

```c
int Foo(int x)
{
    static int a = 0;
    ...
    a += x;
    return a;
}
```

a will remember its value from the last time the function was called. If given an initial value, it is only initialized when first created – not during each function call.
Storage Class Specifiers

External Variables

- Variables that are *defined* outside the scope where they are used
- Still need to be *declared* within the scope where they are used
- `extern` keyword used to tell compiler that a variable defined elsewhere will be used within the current scope

**External Variable Declaration Syntax:**

```
extern type identifier;
```

**External Variable Declaration Example:**

```
extern int x;
```
Storage Class Specifiers

External Variables

• A variable \textit{declared} as \texttt{extern} within a function is analogous to a function prototype – the variable may be \textit{defined} outside the function after it is used.

Example

\begin{verbatim}
int Foo(int x)
{
    extern int a;
    ...
    return a;
}

int a;
\end{verbatim}
Storage Class Specifiers

External Variables

- A variable *declared* as `extern` outside of any function is used to indicate that the variable is *defined* in another source file – memory only allocated when it's *defined*.

```c
Main.c

extern int x;

int main(void)
{
    x = 5;
    ...
}

SomeFileInProject.c

int x;

int Foo(void)
{
    ...
}
```
Storage Class Specifiers

Register Variables

- `register` variables are placed in a processor's "hardware registers" for higher speed access than with external RAM
  - Common with loop counters
- Not as important when RAM is integrated into processor package (microcontrollers, ...)
- May be done with PIC®/dsPIC®, but it is architecture/compiler specific...
Storage Class Specifiers
Scope of Functions

• Scope of a function depends on its storage class:
  – Static Functions
  – External Functions
• Scope of a function is either local to the file where it is defined (static) or globally available to any file in a project (external)
Storage Class Specifiers

External Functions

- Functions by default have global scope within a project
- `extern` keyword not required, but function prototype is required in calling file

```c
Main.c

int foo(void);

int main(void)
{
    ...
    x = foo();
}

SomeFileInProject.c

int foo(void)
{
    ...
}
```
Storage Class Specifiers

Static Functions

- If a function is declared as `static`, it will only be available within the file where it was declared (makes it a local function)

```c
Main.c
int foo(void);
int main(void)
{
    ...  \xmark
    x = foo(); \xmark
}
```

```c
SomeFileInProject.c
static int foo(void)
{
    ...
}
```
Storage Class Specifiers

Static Functions

- If a variable is declared as `static`, it will only be available within the file where it was declared.

```c
Main.c
extern int myVar;

int main(void)
{
    ...
    myVar = 6;
}
```

```c
SomeFileInProject.c
static int myVar = 0;
```