Unit testing

Testing architecture

Example

```c
// 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

- 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 = 10;
    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.

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);
}
```
### Scope

#### Variables Declared Within a Function

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

```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)
{
    { 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**

Global versus Local Variables

**Example**

```c
int x = 5;

int Foo(int y)
{
    int z = 1;
    return (x + y + z);
}

int main(void)
{
    int a = 2;
    x = foo(a);
    a = foo(x);
}
```

- `x` can be seen by everybody.
- `foo`'s local parameter is `y`.
- `foo`'s local variable is `z`.
- `foo` cannot see `main`'s `a`.
- `foo` can see `x`.
- `main`'s local variable is `a`.
- `main` cannot see `foo`'s `y` or `z`.
- `main` can see `x`.

---

**Scope**

Parameters

- "Overloading" variable names:

  - `n` declared locally and globally:
    ```c
    int n;
    int Foo(int n)
    {
        y += n;
    }
    ```
    - `local n` hides `global n`.
  - `n` declared globally only:
    ```c
    int n;
    int Foo(int x)
    {
        y += n;
    }
    ```
    - A locally defined identifier takes precedence.
**Scope**

**Parameters**

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

```c
int n;
int Foo(int n)
{
    y += n;
}
int Bar(int n)
{
    z *= n;
}

Example
```

**Scope**

**Preprocessor and scoping**

```c
#define x 2
void Test(void)
{
    #define x 5
    printf("%d\n", x);
}
void main(void)
{
    printf("%d\n", x);
    Test();
    #define x 5
}

Result:
```

```c
5 2
5 2
```
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

Storage Class Specifiers
Automatic Variables

- 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

- 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

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

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;
}
```

**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:**

```c
extern type identifier;
```

**External Variable Declaration Example:**

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

**External Variables**

- A variable *declared* as `extern` within a function is analogous to a function prototype – the variable may be *defined* outside the function after it is used.

#### Example

```c
int Foo(int x)
{
    extern int a;
    ...
    return a;
}
```

- 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*.

#### Main.c

```c
extern int x;

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

#### SomeFileInProject.c

```c
int Foo(void)
{
    ...
}
```
Scope: visibility

Permanence: "life time"

"Information Hidden"

static int x; module level var

.c only