CS2211b

Software Tools and Systems Programming

Week 9b

Functions

Announcements

Assignment 3 due date extended

- March 17th

Group quiz next week

- Could be Tuesday or Thursday

Week of March 19th (Week 11)

- No class (on Tuesday or Thursday) or office hours
- Will still have labs
- Will still have TA office hours
- There will be an assigned reading and an at home activity

Functions

Calling & Using Functions

We have already seen and been using a number of functions:

`scanf`	`getchar`	`time`	`srand`
`printf`	`putchar`	`rand`	

We call functions by using the function name, followed by a list of arguments inside of parentheses (round brackets):

```
function_name(arg1, arg2, …, argN);
```

Calling & Using Functions

Simple Examples:

```
putchar('K');

char c = getchar();

getchar();

printf("%d", getchar());
```

Calling & Using Functions

Simple Examples:

```
putchar('K');
```

Call `putchar` function with argument `'K'`.

Result of the statement (the return of the function) is ignored (nothing is done with it).

```
char c = getchar();
```

```
getchar();
```

```
printf("%d", getchar());
```

Calling & Using Functions

Simple Examples:

```
putchar('K');

char c = getchar();

getchar();

printf("%d", getchar());
```

Call `getchar` function with no arguments (functions are not required to have parameters).

Result of the statement (the return of the function) is stored in the variable c.

Calling & Using Functions

Simple Examples:

```
putchar('K');

char c = getchar();

getchar();

printf("%d", getchar());
```

Also calls getchar but this time the result of the statement is ignored.

The character `getchar` returns is thrown away (not stored anywhere).

Calling & Using Functions

Simple Examples:

```
putchar('K');

char c = getchar();

getchar();
```

In most cases, functions can be called in expressions.

In this case, the value returned from `getchar` is incremented by one and sent to `printf` to be printed as a character.

For example, if the user input 'a', 'b' would be printed.

```
printf("%c", getchar()+1);
```

Calling & Using Functions

Bad Example:

```
getchar;

int x = rand;
```

Need to use parentheses ( ) even if the function takes no parameters.

These will compile but likely not do what you want (pointer to function and not call function).

Parameters vs. Arguments

Arguments are the values/data sent to the function. For example, in the function call `putchar('X');` the value 'X' is the argument.

Parameters are a type of variable that describes the input the function takes. For example, the definition (protype) of `putchar` is:

```
int putchar(int character);
```

`putchar` takes a single integer parameter, the character that will be printed to the screen.

Some Useful Functions

stdlib.h

Function	Description	Example(s)

`void exit(int status)`	Exit the program with the given exit status. Can be used with the EXIT_FAILURE and EXIT_SUCCESS constants from stdlib.h	`exit(0);`
		`exit(1);`
		`exit(EXIT_SUCCESS);`
		`exit(EXIT_FAILURE);`

`int abs(int x)`	Returns the absolute value of x.	`abs(-5);  // 5` `abs(5);  // 5`
`long labs(long x)`	Same as abs but for longs.	`labs(-10l);  // 10`
`int rand(void)`	Returns a pseudo random number between 0 and RAND_MAX (a constant from stdlib.h).	`rand();`
`void srand(unsinged int seed)`	Sets the seed to be used by rand to create random numbers.	`srand(10);` `srand(time(NULL));`

Some Useful Functions

math.h

Function	Description
double pow(double x, double y)	Returns x raised to the power of y (i.e. x^y).
double sqrt(double x)	Returns the square root of x (i.e. $\sqrt{x}$).
double ceil(double x)	Returns the ceiling of x (rounded up). E.g. 5.1 would become 6.
double floor(double x)	Returns the floor of x (rounded down). E.g. 5.9 would become 4.
double fabs(double x)	Absolute value of x but for floating point types. E.g. -5.123 would become 5.123.
double log(double x)	Returns the natural logarithm (base-e logarithm) of x.
double log10(double x)	Returns the common logarithm (base-10 logarithm) of x.
double fmod(double x, double y)	Returns the remainder of x / y. Like the modulus operator (%) but for floating point types.
sin, cos, tan, acos, asin, atan, etc.	math.h contains trigonometry functions for sin, cos, tan, etc.

Some Useful Functions

math.h

Function	Description
double pow(double x, double y)	Returns x raised to the power of y (i.e. x^y).
double sqrt(double x)	Returns the square root of x (i.e. $\sqrt{x}$).
double ceil(doub	
double floor(dou	9
double fabs(doub	s.
double log(double x)	Returns the natural logarithm (base-e logarithm) of x.
double log10(double x)	Returns the common logarithm (base-10 logarithm) of x.
double fmod(double x, double y)	Returns the remainder of x / y. Like the modulus operator (%) but for floating point types.
sin, cos, tan, acos, asin, atan, etc.	math.h contains trigonometry functions for sin, cos, tan, etc.

math.h contains several other functions and even more are added in C99.

See chapter 22.3 of your C textbook for a more complete listing.

Some Useful Functions

Using math.h with gcc

- Most standard C libraries such as stdio.h, stdlib.h and time.h are automatically linked for you by gcc.
- For largely historical reasons, some libraries like math.h are not automatically linked for you and require an extra command line option when compiling.
- **Example:** /cs2211/week9/mathex.c

```
#include <stdio.h>
#include <math.h>

int main() {
        double a = 5;
        printf("%f\n", sqrt(a));
        return 0;
}
```

Some Useful Functions

Using math.h with gcc

- Most standard C libraries such as stdio.h, stdlib.h and time.h are automatically linked for you by gcc.
- For largely historical reasons some libraries like math.h are not automatically linked for you and require an extra command line option when compiling.
- **Example:** /cs2211/week9/

```
#include <stdio.h>
#include <math.h>

int main() {
        double a = 5;
        printf("%f\n", sqrt(a));
        return 0;
}
```

```
[dservos5@cs2211b week9]$ gcc mathex.c
/tmp/cc52sTM9.o: In function `main':
mathex.c:(.text+0x24): undefined
reference to `sqrt'
collect2: error: ld returned 1 exit
status
```

Some Useful Functions

Using math.h with gcc

- Most standard C libraries such as stdio.h, stdlib.h and time.h are automatically linked for you by gcc.
- For largely historical reasons some libraries like math.h are not automatically linked for you and require an extra command line option when compiling.
- **Example:** /cs2211/week9/mathex.c

```
#include <stdio.h>
#include <math.h>

int main() {
        double a = 5;
        printf("%f\n", sqrt(a));
        return 0;
}
```

```
[dservos5@cs2211b week9]$ gcc -lm mathex.c
[dservos5@cs2211b week9]$ a.out
2.236068
```

Some Useful Functions

Using math.h with gcc

- Most standard C libraries such as stdio.h, stdlib.h and time.h are
- **Have to use `-lm` option when compiling code with math.h. This tells gcc to link in the math library.**
- **Example:** /cs2211/week9/mathex.c

```
#include <stdio.h>
#include <math.h>

int main() {
        double a = 5;
        printf("%f\n", sqrt(a));
        return 0;
}
```

```
[dservos5@cs2211b week9]$ gcc -lm mathex.c
[dservos5@cs2211b week9]$ a.out
2.236068
```

Creating Your Own Functions

- We have actually already been creating a function this whole time, the `main` function.
- The `main` function we have been creating with each C program is a function that takes no arguments (this will change later) and returns an integer value (the exit status).
- We can create other functions in a similar way following this general syntax:

```
return_type func_name(param1, …, paramN) {
   declarations
   statements
}
```

Creating Your Own Functions

```
return_type func_name(param1, …, paramN) {
    declarations
    statements
}
```

return_type: The data type that the function will return (e.g. int, float, double, etc). If the function has no return we use the type void.

func_name: The name of the function. We will use this to call the function in our code.

params: Zero or more parameters the function will take. These are data type and parameter name pairs (e.g. int x or float f).

declarations: Variable declaration statements. In C89 these have to come before normal statements.

statements: All other statements the function will contain. This is the code that will be run when we call our function. In C89 these can not contain variable declarations.

Creating Your Own Functions

Recall that the course server does not use C89 by default but GNU89 which is a bit different. In GNU89 (C89 + GNU extension) you can mix variable declarations with statements.

We don't have to worry about this unless we use the -pedantic option (forces stricter compliance with C standards).

If the function has no return we use the type void.

func_name: The name of the function. We will use this to call the function in our code.

params: Zero or more parameters the function will take. These are data type and parameter name pairs (e.g. int x or float f).

declarations: Variable declaration statements. In C89 these have to come before normal statements.

statements: All other statements the function will contain. This is the code that will be run when we call our function. In C89 these can not contain variable declarations.

Creating Your Own Functions

Returning a Value

- Just as we use the `return` statement to return an exit status from our `main` function, we can use the `return` statement to return a value from a function.
- The type of the value returned should match the `return_type` of the function.
- If the `return_type` is `void`, then no return is required.
- Failing to return a value when a `return_type` is nonvoid will result in **undefined behaviour**.
- The `return` statement causes the function to exit at that point. Just like it does in the `main` function.
- **Syntax:**

```
return expression;
```

Creating Your Own Functions

Returning a Value

- Just as we use the `return` statement to return an exit status from our `main` function, we can use the `return` statement to return a value from a function.
- The type of the value returned should match the `return_type` of the function.
- If the `return_type` is `void`, then no return is required.
- Failing to return a value when a `return_type` is nonvoid will result in unde
- The `return` s
  Just like it doe
- **Syntax:**

`expression` is optional if the `return_type` is `void`. In this case, it would simply exit the function without returning a value.

```
return expression;
```

Simple Examples

```c
int max(int x, int y) {
        if(x > y) {
                return x;
        } else {
                return y;
        }
}
```

```c
void count_down(int n) {
        int i;
        for(i = n; i >= 0; i--)
                printf("%d\n", i);
}
```

```c
double avg(double n1, double n2, double n3) {
        return (n1 + n2 + n3) / 3;
}
```

```c
int getRandomNumber() {
        return 4; // chosen by fair dice roll.
                  // guaranteed to be random.
}
```

Simple Examples

```
int max(int x, int y) {
        if(x > y) {
                return x;
        } else {
                return y;
        }
}
```

We are allowed to have multiple returns in one function.

Defines a function named `max` that takes two integer arguments and returns an integer value.

Returns the largest of integer.

Simple Examples

```
void count_down(int n) {
        int i;
        for(i = n; i >= 0; i--)
                printf("%d\n", i);
}
```

No return is needed if return type is void.

Defines a function named count_down with a single integer parameter and does not return any value (void return).

Prints the numbers n to 0.

Simple Examples

Defines a function named avg that takes three double arguments and returns a double value.

Finds the average of three numbers.

```
double avg(double n1, double n2, double n3) {
        return (n1 + n2 + n3) / 3;
}
```

```
int getRandomNumber() {
        return 4; // chosen by fair dice roll.
                  // guaranteed to be random.
}
```

Simple Examples

Defines a function named `getRandomNumber` that takes no arguments and always returns the value 4.

Could also use void keyword to state that function takes no arguments:

```
int getRandomNumber(void) {
  …
}
```

```
int getRandomNumber() {
        return 4; // chosen by fair dice roll.
                  // guaranteed to be random.
}
```

Factorial Example

/cs2211/week9/ex6.c

```c
#include <stdio.h>

long long factorial(int n) {
        long long fact = 1;
        int i;

        for(i = 2; i <= n; i++)
                fact *= i;

        return fact;
}

int main() {
        long long x;
        int y;

        printf("Input number: ");
        scanf("%d", &y);

        x = factorial(y);
        printf("%d! = %lld\n", y, x);

        return 0;
}
```

The factorial of **n** is the product of all positive integers less than or equal to **n**.

That is n! = 1 x 2 x … x **n**

Factorial Example

/cs2211/week9/ex6.c

```c
#include <stdio.h>

long long factorial(int n) {
        long long fact = 1;
        int i;

        for(i = 2; i <= n; i++)
                fact *= i;

        return fact;
}

int main() {
        long long x;
        int y;

        printf("Input number: ");
        scanf("%d", &y);

        x = factorial(y);
        printf("%d! = %lld\n", y, x);

        return 0;
}
```

`factorial` function takes a single integer argument and returns a value of type `long long`.

Factorial Example

/cs2211/week9/ex6.c

```c
#include <stdio.h>

long long factorial(int n) {
        long long fact = 1;
        int i;

        for(i = 2; i <= n; i++)
                fact *= i;

        return fact;
}

int main() {
        long long x;
        int y;

        printf("Input number: ");
        scanf("%d", &y);

        x = factorial(y);
        printf("%d! = %lld\n", y, x);

        return 0;
}
```

Calculates the factorial using a for loop.

Factorial Example

/cs2211/week9/ex6.c

```c
#include <stdio.h>

long long factorial(int n) {
        long long fact = 1;
        int i;

        for(i = 2; i <= n; i++)
                fact *= i;

        return fact;
}

int main() {
        long long x;
        int y;

        printf("Input number: ");
        scanf("%d", &y);

        x = factorial(y);
        printf("%d! = %lld\n", y, x);

        return 0;
}
```

Returns the value of the variable fact.

Factorial Example

/cs2211/week9/ex6.c

```c
#include <stdio.h>

long long factorial(int n) {
        long long fact = 1;
        int i;

        for(i = 2; i <= n; i++)
                fact *= i;

        return fact;
}

int main() {
        long long x;
        int y;

        printf("Input number: ");
        scanf("%d", &y);

        x = factorial(y);
        printf("%d! = %lld\n", y, x);

        return 0;
}
```

Calls the function `factorial` with y as an argument.

The value returned is stored in the variable x.

Factorial Example

/cs2211/week9/ex6.c

```c
#include <stdio.h>

long long factorial(int n) {
        long long fact = 1;
        int i;

        for(i = 2; i <= n; i++)
                fact *= i;

        return fact;
}

int main() {
        long long x;
        int y;

        printf("Input number: ");
        scanf("%d", &y);

        x = factorial(y);
        printf("%d! = %lld\n", y, x);

        return 0;
}
```

Example Input/Output:

```
[dservos5@cs2211b week9]$ ex6
Input number: 8
8! = 40320
```

Nested Function Calls

- Functions are allowed to call other functions.
- **Examples:**

```c
#include <stdio.h>
float c(float z) {
        return z * z;
}

float b(float y) {
        return c(y);
}

float a(float x) {
        return b(x);
}

int main() {
        printf("%d\n", a(2.0));
}
```

What is the output?

Nested Function Calls

- Functions are allowed to call other functions.
- **Examples:**

```c
#include <stdio.h>
float c(float z) {
        return z * z;
}

float b(float y) {
        return c(y);
}

float a(float x) {
        return b(x);
}

int main() {
        printf("%d\n", a(2.0));
}
```

The order of these functions matters, if we switched function b and a around we would get an error.

Nested Function Calls

- Functions are
- **Examples:**

```
error: conflicting types for ‘b’
 float b(float y) {
       ^
note: previous implicit declaration of ‘b’ was here
  return b(x);
         ^
```

```
#include <st
float c(floa
        return z * z;
}

float a(float x) {
        return b(x);
}

float b(float y) {
        return c(y);
}

int main() {
        printf("%d\n", a(2.0));
}
```

Trying to call function b before it is declared.

Function Prototypes

- One solution is to **prototype** the function before declaring it.
- Tells the compiler that you will be declaring this function later on and what parameters and return type to expect.
- It is generally good practice to **prototype** your functions.
- **Syntax:**

```
return_type func_name(param1, …, paramN);
```

Looks similar to a normal function declaration but does not contain the body of the function and is ended by a semicolon.

Function Prototypes

Example:

```c
#include <stdio.h>

float a(float x);
float c(float z);
float b(float y);

int main() {
        printf("%f\n", a(2.0));
}

float c(float z) {
        return z * z;
}

float a(float x) {
        return b(x);
}

float b(float y) {
        return c(y);
}
```

Function Prototypes

Example:

```
#include <stdio.h>

float a(float x);
float c(float z);
float b(float y);

int main() {
        printf("%f\n", a(2.0));
}

float c(float z) {
        return z * z;
}

float a(float x) {
        return b(x);
}

float b(float y) {
        return c(y);
}
```

Prototypes each function.

After the function is prototyped it may be defined in any order.

This code will now compile without error.

Function Prototypes

- If we fail to prototype a function and it has not been declared yet, C89 assumes it has an integer parameter and returns an integer value.
- **Example:**

```c
#include <stdio.h>

int main() {
        printf("%d\n", a(2.0));
        return 0;
}

int a(int x) {
        return x * x;
}
```

Compiles

```c
#include <stdio.h>

int main() {
        printf("%d\n", a(2.0));
        return 0;
}

float a(float x) {
        return x * x;
}
```

Errors

Function Prototypes

- If we fail to prototype a function and it has not been declared yet, C89 assumes it has an integer parameter and returns an
- **Best practice is to prototype or declare function before using it and not rely on this behaviour.**

```c
#include <stdio.h>

int main() {
        printf("%d\n", a(2.0));
        return 0;
}

int a(int x) {
        return x * x;
}
```

Compiles

```c
#include <stdio.h>

int main() {
        printf("%d\n", a(2.0));
        return 0;
}

float a(float x) {
        return x * x;
}
```

Errors

Pass by Value vs. Pass by Reference

- By default, C functions are **pass by value**.
- This means that the value of a variable is sent to the function and not a the variable its self or a reference to it.
- **Example:**

/cs2211/week9/ex7.c

```c
#include <stdio.h>

void f(int x) {
        x = 20;
        printf("Value in function is %d\n", x);
}

int main() {
        int y = 10;
        printf("Value before function is %d\n", y);
        f(y);
        printf("Value after function is %d\n", y);
        return 0;
}
```

Pass by Value vs. Pass by Reference

Output:

```
[dservos5@cs2211b week9]$ ex7
Value before function is 10
Value in function is 20
Value after function is 10
```

/cs2211/week9/ex7.c

```c
#include <stdio.h>

void f(int x) {
        x = 20;
        printf("Value in function is %d\n", x);
}

int main() {
        int y = 10;
        printf("Value before function is %d\n", y);
        f(y);
        printf("Value after function is %d\n", y);
        return 0;
}
```

Pass by Value vs. Pass by Reference

Output:

```
[dservos5@cs2211b week9]$ ex7
Value before function is 10
Value in function is 20
Value after function is 10
```

/cs2211/week9/ex7.c

```
#include <stdio.h>

void f(int x) {
        x = 20;
        printf("Value in
}

int main() {
        int y = 10;
        printf("Value before function is %d\n", y);
        f(y);
        printf("Value after function is %d\n", y);
        return 0;
}
```

This only changed the value of x inside the function.

It had no affect on the value of y in main.

Pass by Value vs. Pass by Reference

- We can **pass by reference** using **pointers**.
- **Pointers** are a special type of variable that holds a memory address rather than a value.
- We will go more in depth into pointers in the coming weeks.
- For now we can use the **&** and * prefix operators to pass by reference.
- **Syntax:**

`&var_name`

Returns the memory address of `var_name`.

Creates a pointer to `var_name`

`*pointer_name`

Dereferences the pointer `pointer_name`.

Returns or sets the value stored at the memory address.

Pass by Value vs. Pass by Reference

Pass by Reference Example 1:

/cs2211/week9/ex8.c

```c
#include <stdio.h>

void sum(int *result, int x, int y) {
        *result = x + y;
}

int main() {
        int result;
        int a = 5, b = 6;

        sum(&result, a, b);

        printf("Sum is %d.\n", result);
        return 0;
}
```

Pass by Value vs. Pass by Reference

Pass by Reference

/cs2211/week9/ex8.c

Function `sum` takes two normal integer arguments `x` and `y`. It also takes `result`, an integer pointer.

```c
#include <stdio.h>

void sum(int *result, int x, int y) {
        *result = x + y;
}

int main() {
        int result;
        int a = 5, b = 6;

        sum(&result, a, b);

        printf("Sum is %d.\n", result);
        return 0;
}
```

Pass by Value vs. Pass by Reference

Pass by Reference Example 1:

/cs2211/week9/ex8.c

```c
#include <stdio.h>

void sum(int *result, int x, int y) {
        *result = x + y;
}

int main() {
        int result
        int a = 5,

        sum(&resul

        printf("Sum is %d.\n", result);
        return 0;
}
```

To read or set the value of result we are required to use the * prefix operator to "deference" the pointer.

If we omit the * we are setting or reading the address rather than the value.

Pass by Value vs. Pass by Reference

Pass by Reference Example 1:

/cs2211/week9/ex8.c

```c
#include <stdio.h>

void sum(int *result, int x, int y) {
        *result = x + y;
}

int main() {
        int result
        int a = 5,

        sum(&result, a, b);

        printf("Sum is %d.\n", result);
        return 0;
}
```

Like with `scanf` we need to use the & prefix operator to send the address rather than the value to the `sum` function.

Output:

```
[dservos5@cs2211b week9]$ ex8
Sum is 11.
```

Value of result is set by the sum function using pass by reference.

```c
#i

void sum(int *result, int x, int y) {
        *result = x + y;
}

int main() {
        int result;
        int a = 5, b = 6;

        sum(&result, a, b);

        printf("Sum is %d.\n", result);
        return 0;
}
```

Pass by Value vs. Pass by Reference

Pass by Reference Example 2:

/cs2211/week9/ex9b.c

```c
#include <stdio.h>

void swap(int *a, int *b) {
    int c = *a;
    *a = *b;
    *b = c;
}

int main() {
    int x = 10, y = 5;

    printf("Before swap:\nx = %d\ty = %d\n", x, y);
    swap(&x, &y);
    printf("After swap:\nx = %d\ty = %d\n", x, y);

    return 0;
}
```

Pass by Value vs. Pass by Reference

Pass by Reference Example 2:

/cs2211/week9/ex9b.c

```c
#include <stdio.h>

void swap(int *a, int *b) {
    int c = *a;
    *a = *b;
    *b = c;
}

int main() {
    int x = 10, y

    printf("Before swap:\nx = %d\ty = %d\n", x, y);
    swap(&x, &y);
    printf("After swap:\nx = %d\ty = %d\n", x, y);

    return 0;
}
```

Declare function swap that has two integer pointer parameters a and b.

a and b are passed by reference.

Pass by Value vs. Pass by Reference

Pass by Reference Example 2:

/cs2211/week9/ex9b.c

```c
#include <stdio.h>

void swap(int *a, int *b) {
    int c = *a;
    *a = *b;
    *b = c;
}

int main() {
    int x = 10, y

    printf("Before
    swap(&x, &y);
    printf("After swap:\nx = %d\ty = %d\n", x, y);

    return 0;
}
```

Value that a points to is stored in c.

Value that a points to is updated to value that b points to.

Value that b points to is updated to value of c.

Pass by Value vs. Pass by Reference

Pass by Reference Example 2:

/cs2211/week9/ex9b.c

```c
#include <stdio.h>

void swap(int *a, int *b) {
    int c = *a;
    *a = *b;
    *b = c;
}

int main() {
    int x = 10, y = 5;

    printf("Before swap:\nx = %d\ty = %d\n", x, y);
    swap(&x, &y);
    printf("After swap:\nx = %d\ty = %d\n", x, y);

    return 0;
}
```

What does this function do when called from main with &x and &y?

Variable Scope

- Scope is the region or area of a program where a variable can be accessed.
- In general, tends to follow blocks defined by curly braces { }.
- If two variables share the same name, the one with the inner most scope is accessed.

Variable Scope

Simple Example:

```
int x;

void bar() {
        int y;

        ...
}

int main() {
        int z;

        ...

        while(...) {
                int w;

                ...
        }
}
```

Variable Scope

Simple Example:

```
int x;                                  x

void bar() {
        int y;

        ...
}

int main() {
        int z;

        ...

        while(...) {
                int w;

                ...
        }
}
```

The variable **x** is accessible anywhere in the code after it has been declared. Could be read or set for example in function `bar` or `main`.

This is considered to be a **global variable**.

Generally bad practice to use **global variables** unless required.

Variable Scope

Simple Example:

```
int x;                                  x

void bar() {
        int y;                          y

        ...
}

int main() {
        int z;

        ...

        while(...) {
                int w;

                ...
        }
}
```

The variable **y** is accessible anywhere in the `bar` function after it has been declared.

Can not be accessed in the `main` function.

This is considered to be a **local variable**.

Variable Scope

Simple Example:

```
int x;                                  x

void bar() {
        int y;                          y

        ...
}

int main() {
        int z;                          z

        ...

        while(...) {
                int w;

                ...
        }
}
```

The variable z is accessible anywhere in the main function after it has been declared. This includes inside loops like the while loop shown here.

Can not be accessed in the bar function.

Also a **local variable**.

Variable Scope

Simple Example:

```
int x;                          x

void bar() {
        int y;                  y

        ...
}

int main() {
        int z;                  z

        ...

        while(...) {
                int w;          w

                ...

        }
}
```

The variable `w` only accessible inside the loop it was declared in.

Can not be accessed in the main function outside of the loop.

Variable Scope

/cs2211/week9/ex10.c

```c
#include <stdio.h>

int x = 1;

void foo(void) {
        printf("A: %d\n", x);
        int x = 2;
        printf("B: %d\n", x);
}

int main() {
        foo();

        printf("C: %d\n", x);
        int x = 3;
        printf("D: %d\n", x);

        while(1) {
                printf("E: %d\n", x);
                int x = 4;
                printf("F: %d\n", x);
                break;
        }

        printf("G: %d\n", x);
        return 0;
}
```

Variable Scope

/cs2211/week9/ex10.c

```c
#include <stdio.h>

int x = 1;

void foo(void) {
        printf("A: %d\n", x);
        int x = 2;
        printf("B: %d\n", x);
}

int main() {
        foo();

        printf("C: %d\n", x);
        int x = 3;
        printf("D: %d\n", x);

        while(1) {
                printf("E: %d\n", x);
                int x = 4;
                printf("F: %d\n", x);
                break;
        }

        printf("G: %d\n", x);
        return 0;
}
```

Output:

```
[dservos5@cs2211b week9]$ ex10
A: 1
B: 2
C: 1
D: 3
E: 3
F: 4
G: 3
```