CSC215
Functions and Program Structure

Dr. Achraf El Allali

Usage

- To avoid repetitive code
- Written once, can be instantiated multiple times
- We have already seen
  - main, printf, scanf, sizeof

Let’s add two numbers

- Mathematical representation of the add function f:

$$f(a, b) = a + b;$$

Let’s add two numbers

```
int f(int a, int b)
{

        c = a + b;


}
```

Let’s add two numbers

```
int f(int a, int b)
{

        c = a + b;
        return c;
}
```

Let’s add two numbers

```
int f(int a, int b)
{
        int c;
        c = a + b;
        return c;
}
```

Adding two numbers

```
int add(int a, int b)
{
        int c;
        c = a + b;
        return c;
}
main ()
{
        int a, b, c;
        a = 5; b= 10;
        c = add(a, b);
        printf(“The sum is %d:”, c);
}
```

Adding two numbers

```
int add(int , int);   /*prototype declaration*/
main ()
{
        int a, b, c;
        a = 5; b= 10;
        c = add(a, b);
        printf(“The sum is %d:”, c);
}
int add(int a, int b){
        int c;
        c = a + b;
        return c;
}
```

Adding two numbers

```
main (){
    int a, b, c, e, f, g;
    a = 5; b = 4; e = 15; f=10;
    c = add(a, b);
    g = add(e, f);
    printf(“The results are %d and %d”, c, g);
}
```

Scope

```
void doubleX(float x)
{
        x *=2;
        printf(“%f”, x);
}
main()
{
        float x = 10;
        doubleX(x);
        printf(“%f”, x);
}
```

Scope

```
void doubleX(float x)
{
        x *=2;
        printf(“%f”, x);  /* The value of x here is 20 */
}
main()
{
        float x = 10;
        doubleX(x);
        printf(“%f”, x); /* The value of x here is 10 */
}
```

Scope

```
float x = 10;        /* global variable */
void doubleX()
{
      x *= 2;
      printf(“%f”, x);
}
main()
{
      doubleX();
      printf(“%f”, x);
}
```

Scope

```
float x = 10;        /* global variable */
void doubleX()
{
      x *= 2;
      printf(“%f”, x);  /* The value of x here is 20 */
}
main()
{
      doubleX();
      printf(“%f”, x); /* The value of x here is 20 */
}
```

Scope

```
float x = 10;        /* global variable */
void doubleX(float x)
{
        x *=2;
        printf(“%f”, x);
}
main()
{
        doubleX(x);
        printf(“%f”, x);
}
```

Scope

```
float x = 10;        /* global variable */
void doubleX(float x)
{
        x *=2;
        printf(“%f”, x);  /* The value of x here is 20 */
}
main()
{
        doubleX(x);
        printf(“%f”, x); /* The value of x here is 10 */
}
```

Scope

```
float x = 10;        /* global variable */
void doubleX()
{
        x *=2;
        printf(“%f”, x);
}
main(){
       float x = 3;
        doubleX();
        printf(“%f”, x);
        }
```

Scope

```
float x = 10;        /* global variable */
void doubleX()
{
        x *=2;
        printf(“%f”, x);  /* The value of x here is 20 */
}
main(){
       float x = 3;
        doubleX();
        printf(“%f”, x); /* The value of x here is 3 */
}
```

Scope

```
float x = 10;        /* global variable */
void doubleX(float x)
{
        x *=2;
        printf("%f", x);
}
main(){
        float x = 3;
        doubleX(x);
        printf("%f", x);
        }
```

Scope

```
float x = 10;        /* global variable */
void doubleX(float x)
{
        x *=2;
        printf(“%f”, x);  /* The value of x here is 6 */
}
main(){
       float x = 3;
        doubleX(x);
        printf(“%f”, x); /* The value of x here is 3 */
}
```

Scope

```
main ()
{
        int x = 5;
        if (x){
                int x = 10;
                x++;
                printf (“%d”, x);
        }
        x++;
        printf (“%d”, x);
}
```

Scope

```
main ()
{
        int x = 5;
        if (x){
                int x = 10;
                x++;
                printf (“%d”, x);        /*11*/
        }
        x++;
        printf (“%d”, x);        /*6*/
}
```

Recursion

- When a function calls itself (directly, or indirectly) it is called a recursive function

```
void change (count)
{
..
..
change(count);
..
}
```

Factorial

- In mathematics,
  - n! = n * (n-1) * ……1
  - 0! = 1

- Example
  - 5! = 5 * 4 * 3 * 2 * 1

Factorial

- In mathematics,
  - n! = n * (n-1) * ……1
  - 0! = 1

- Example
  - 5! = 5 * (4 * 3 * 2 * 1)

Factorial

- In mathematics,
  - n! = n * (n-1) * ……1
  - 0! = 1

- Example
  - 5! = 5 * 4!

Factorial

- Generalization
  - n! = n * (n-1) ! for > 0
  - 0! = 1

Factorial

```
/* The recursive function computing n!*/
int factorial( int n )
{
        if (n ==0)
        return 1; /* Termination condition */
        else
        return n * factorial(n-1); /* Recurse*/
}
```

```
/* The function computes n!*/
int fact( int a)
{
int i, fff;
for (fff= 1, i=1; i<=a; i++) /* a! */
fff*= i;
return fff;
}
```

Arguments to main()

```
main(int argc, char *argv[])
{
        int i;
        for (i = 0; i < argc; i++)
                printf("%s\n", argv[i]);
}
```

Arguments to main()

- Compiling
  - gcc -o main main.c
- Running
  - ./main CSC 215

    Output

    ./main

    CSC

    215

Multiple source files

main.c

```
#include “square.h”
main()
{
      square(3);
}
```

square.h

```
void square(int);
```

square.c

```
#include <stdio.h>
#include “square.h”
void square(int x)
{
 printf(“%d”, x*x)
}
```

Multiple source files

- Compiling

  gcc -o main main.c square.c

- Running

  ./main

Makefile

- Consider the square example

- To compile:

  gcc -o main main.c square.c

- Must use the up arrow, or retype compile line during the test/modify/debug cycle

Simplest Makefile

- Makefile 1

```
squaremake: main.c square.c
    gcc -o main main.c square.c -I.
```

- Save in Makefile or makefile
- type make
- List of file to apply the rule to when typing make
- Inefficient?

Makefile 2

```
CC=gcc
CFLAGS=-I.

squaremake: main.o square.o
    $(CC) -o main main.o square.o -I.
```

- CC is the C compiler to use,
- CFLAGS is the list of flags to pass to the compilation command.
- What if we make change to square.h?

Makefile 3

```
CC=gcc
CFLAGS=-I.
DEPS = square.h

%.o: %.c $(DEPS)
	$(CC) -c -o $@ $< $(CFLAGS)

squaremake: main.o square.o
	$(CC) -o main main.o square.o -I.
```

Makefile 3

- .o file depends upon the .c version of the file and the .h files included in the DEPS macro
- The -c flag says to generate the object file,
- the -o $@ says to put the output of the compilation in the file named on the left side of the :
- the $< is the first item in the dependencies list,
- $@ and $^, which are the left and right sides of the :, respectively

Makefile 4

```
CC=gcc
CFLAGS=-I.
DEPS = square.h
OBJ = main.o square.o

%.o: %.c $(DEPS)
	$(CC) -c -o $@ $< $(CFLAGS)

squaremake: $(OBJ)
	gcc -o $@ $^ $(CFLAGS)
```

- All of the include files should be listed as part of the macro DEPS,
- All of the object files should be listed as part of the macro OBJ.

Makefile 5

```
IDIR =../include
CC=gcc
CFLAGS=-I$(IDIR)

ODIR=obj
LDIR =../lib

LIBS=-lm

_DEPS = square.h
DEPS = $(patsubst %,$(IDIR)/%,$(_DEPS))

_OBJ = main.o square.o
OBJ = $(patsubst %,$(ODIR)/%,$(_OBJ))


$(ODIR)/%.o: %.c $(DEPS)
	$(CC) -c -o $@ $< $(CFLAGS)
```

```
squaremake: $(OBJ)
	gcc -o $@ $^ $(CFLAGS) $(LIBS)

.PHONY: clean

clean:
	rm -f $(ODIR)/*.o *~ core $(INCDIR)/*~
```