

Functions and Modular Programming

Outline

- ❖ Functions:
 - Need, Definition
 - Defining functions
 - Calling functions
 - Prototypes
- ❖ Scopes
 - Scope and visibility
 - Storage classes
- ❖ Recursive functions
- ❖ Multiple source files
- ❖ Makefiles

Introduction

- ❑ Design your solution so that it keeps the flow of control as simple as possible
 - top-down design:
decompose the problem into smaller problems each can be solved easily
- ❑ Some problems are complicated
 - break them down into smaller problems
 - conquer each sub problem independently
- ❑ Your programs will consist of a collection of user-defined functions
 - each function solves one of the small problems
 - you call (invoke) each function as needed

What is a Function?

- ❑ Function: a group of statements that together perform a task
 - divide up your code into separate functions such that each performs a specific task
 - every C program has at least one function, which is **main()**
 - most programs define additional functions
- ❑ Why
 - to avoid repetitive code : “reusability” written once, can be called infinitely
 - to organize the program : making it easy to code, understand, debug and collaborate
 - to hide details : “what is done” vs “how it is done”
 - to share with others
- ❑ Defining functions:
 - Predefined (library functions): We have already seen
 - main, printf, scanf, getchar, gets
 - User-defined

Defining Functions

❑ Syntax:

```
<return_type> <function_name>(<parameter_list>){  
    <function_body>  
}
```

- Return_type: data type of the result
 - Use `void` if the function returns nothing
 - if no type is specified and `void` is not used: it defaults to `int`
- Function_name: any valid identifier
- Parameter_list:
 - declared variables: `<param_type> <param_name>`
 - comma separated
- Function_body:
 - declaration statements
 - other processing statements
 - `return` statement, if not `void`

Example

❑ In many application, finding the greatest common factor is an important step

❑ GCF function:

- takes two input integers
- finds the greatest integer that divide both of them
- returns the result to the calling context
- Euclidean algorithm:
 - if $a > b \rightarrow \text{gcf}(a, b) = \text{gcf}(b, a \bmod b)$
 - if $b > a$, swap a and b
 - Repeat until b is 0

❑ In c:

```
int gcf(int a, int b){
    /* if a < b swap them, to be discussed later*/
    while (b) {
        int temp = b ;
        b = a % b ;
        a = temp ;
    }
    return a;
}
```

Calling Functions

- ❑ Syntax:
`<function name>(<argument list>)`
- ❑ A function is invoked (called) by writing:
 - its name, and
 - an appropriate list of arguments within parentheses
 - arguments must match the parameters in the function definition in:
1- count , 2- type and 3- order
- ❑ Arguments are passed by value
 - each argument is evaluated, and
 - its value is copied to the corresponding parameter in the called function
- ❑ What if you need to pass the variable by reference?
 - you cannot
 - But you can pass its address by reference

Calling Functions

❑ Example:

/* Does not work as expected*/

```
void swap(int a, int b) {  
    int temp = a;  
    a = b;  
    b = temp;  
}  
  
int main(){  
    int a = 3, b = 5;  
    swap(a, b);  
    printf("a=%d, b=%d\n", a, b);  
    return 0;  
}
```

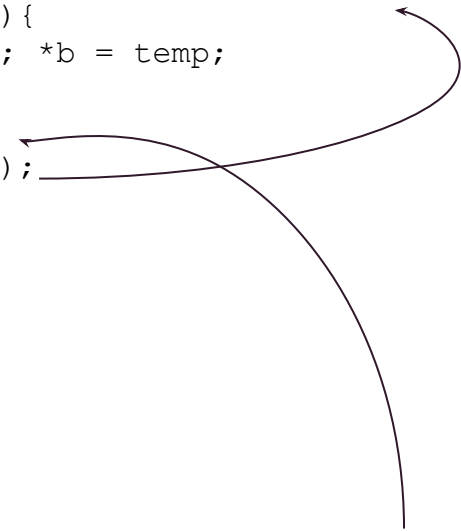
/* Works as expected*/

```
void swap(int *a, int *b){  
    int temp = *a;  
    *a = *b;  
    *b = temp;  
}  
  
int main(){  
    int a = 3, b = 5;  
    swap(&a, &b);  
    printf("a=%d, b=%d\n", a, b);  
    return 0;  
}
```


Calling Functions

- ❑ A function can be called from any function, not necessarily from `main`
- ❑ Example:

```
void swap(int *a, int *b){
    int temp = *a; *a = *b; *b = temp;
}
int gcf(int a, int b){
    if (b > a) swap(&a, &b);
    while (b) {
        int temp = b ;
        b = a % b ;
        a = temp ;
    }
    return a;
}
int main(){
    int a = 3, b = 5;
    printf("GCF of %d and %d is %d\n", a, b, gcf(a, b) );
    return 0;
}
```



The diagram consists of two curved arrows. One arrow originates from the `swap(&a, &b);` line within the `gcf` function and points to the `swap` function definition above it. The second arrow originates from the `gcf(a, b)` argument within the `printf` call in the `main` function and points to the `gcf` function definition above it.

Function Prototypes

- ❑ If function definition comes textually after use in program:
 - The compiler complains: warning: implicit declaration of function
- ❑ Declare the function before use: Prototype
`<return_type> <function_name> (<parameters_list>);`
- ❑ Parameter_list does not have to name the parameters
- ❑ Function definition can be placed anywhere in the program after the prototypes.
- ❑ If a function definition is placed in front of main(), there is no need to include its function prototype.

Function Prototypes: Example

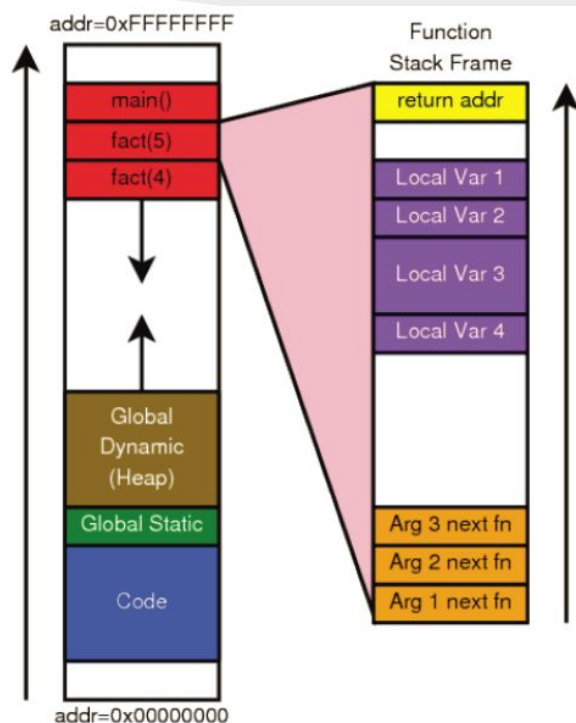
```
#include <stdio.h>
int gcf(int, int);
void swap(int*, int*);
int main(){
    int a = 33, b = 5;
    printf("GCF of %d and %d is %d\n", a, b, gcf(a, b) );
    return 0;
}
int gcf(int a, int b){
    if (b > a) swap(&a, &b);
    while (b) {
        int temp = b ;
        b = a % b ;
        a = temp ;
    }
    return a;
}
void swap(int *a, int *b){
    int temp = *a; *a = *b; *b = temp;
}
```

Function Stub

- ❑ A stub is a dummy implementation of a function with an empty body
 - A placeholder while building (other parts of) a program
 - so that it compiles correctly
 - Fill in one-stub at a time
 - Compile and test if possible

Memory Model

- ❑ Program code
 - Read only
 - May contain string literals
- ❑ Stack (automatic storage):
 - Function variables:
 - Local variables
 - Arguments for next function call
 - Return location
 - Destroyed when function ends
- ❑ Heap:
 - Dynamically allocated space
- ❑ Data segment:
 - Global variables
 - Static variables



Scopes

- ❑ Scope: the parts of the program where an identifier is valid
- ❑ A global variable:
 - A.K.A. external variable
 - defined outside of the local environment (outside of functions)
 - available anywhere within the file
- ❑ A local variable:
 - A.K.A. internal and automatic variable
 - defined within the local environment inside { }
 - local to that block, whether the block is a block within a function or the function itself
 - parameters in a function header are local to that function
 - it can mean different things in different contexts
 - if two variables share the same name but are in different blocks or functions
 - the variable declared in the current environment will be the one used in a reference

Scopes: Examples

Ex1:

```
#include <stdio.h>

void doubleX(float x){
    x *= 2;
    printf("%f\n", x);
}

int main(){
    float x = 3;
    doubleX(x);
    printf("%f\n", x);
    return 0;
}
```

```
6.000000
3.000000
```

Ex2:

```
#include <stdio.h>

float x = 10;

void doubleX(){
    x *= 2;
    printf("%f\n", x);
}

int main(){
    float x = 3;
    doubleX();
    printf("%f\n", x);
    return 0;
}
```

```
20.000000
3.000000
```

Ex3:

```
#include <stdio.h>

float x = 10;

void doubleX(float x){
    x *= 2;
    printf("%f\n", x);
}

void printX(){
    printf("%f\n", x);
}

int main(){
    float x = 3;
    doubleX(x);
    printf("%f\n", x);
    printX();
    return 0;
}
```

```
6.000000
3.000000
10.000000
```

Ex4:

```
#include <stdio.h>

int main(){
    int x = 5;
    if (x){
        int x = 10;
        x++;
        printf("%d\n", x);
    }
    x++;
    printf("%d\n", x);
    return 0;
}
```

```
11
6
```

Storage Classes

- ❑ Storage Classes: a modifier precedes the variable to define its scope and lifetime
- ❑ **auto**: the default for local variables
- ❑ **register**: advice to the compiler to store a local variable in a register
 - the advice is not necessarily taken by the compiler
- ❑ **static**: tells the compiler that the storage of that variable remains in existence
 - Local variables with static modifier remains in memory so that they can be accessed later
 - Global variables with static modifier are limited to the file where they are declared
- ❑ **extern**: points the identifier to a previously defined variable
- ❑ Initialization:
 - in the absence of explicit initialization:
 - static and external variables are set to 0
 - automatic and register variables contain undefined values (garbage)

Storage Classes: Examples

Ex1:

```
#include <stdio.h>

int main(){
    float x = xx;    ✗

    return 0;
}

float xx;

void foo(){
    float x = xx;    ✓
}

/*
    main doesn't know
    about xx
*/
```

Ex1 correction:

```
#include <stdio.h>

int main(){
    extern float xx;    ✓
    float x = xx;        ✓

    return 0;
}

float xx;

void foo(){
    float x = xx;        ✓
}

/*
    declare xx in main
    as extern to point to
    the external xx, this
    will not create new xx
*/
```

Ex2:

```
/*file1.c    */

#include <stdio.h>
int sp = 0;
double val[1000];

int main(){
    return 0;
}

/*file2.c    */

#include <stdio.h>

void foo(){
    printf("%d", sp);    ✗
}

int bar(){
    return (int)val[0];    ✗
}
```

Ex2 correction:

```
/*file1.c    */

#include <stdio.h>
int sp = 0;
double val[1000];

int main(){
    return 0;
}

/*file2.c    */

#include <stdio.h>
extern int sp;    ✓
extern double val[];    ✓

void foo(){
    printf("%d", sp);    ✓
}

int bar(){
    return (int)val[0];    ✓
}
```

Recursive Functions

- ❑ Recursive function: a function that calls itself (directly, or indirectly)

- ❑ Example:

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

- ❑ The algorithm needs to be written in a recursive style

- a step or more uses the algorithm on a smaller problem size

- ❑ It must contain a base case that is not recursive

- ❑ Each function call has its own stack frame

- consumes resources

Recursive Functions: Examples

❑ Multiply $x \times y$:

```
int multiply(int x, int y){  
    if (y == 1) return x;  
    return x + multiply(x, y-1);  
}
```

❑ Power x^y :

```
int power(int x, int y){  
    if (y == 0) return 1;  
    return x * multiply(x, y-1);  
}
```

❑ Factorial $x!$:

```
int fac(int x){  
    if (x == 1) return 1;  
    return x * fac(x-1);  
}
```

❑ Fibonacci:

```
int fib(int x) {  
    if (x == 0) return 0;  
    if (x == 1) return 1;  
    return fib(x-1) + fib(x-2);  
}
```

❑ Palindrome:

```
int isPal(char* s, int a, int b) {  
    if (b >= a) return 1;  
    if (s[a] == s[b])  
        return isPal(s, a+1, b-1);  
    return 0;  
}
```

Optional Parameters

- ❑ C permits functions to have optional parameters
- ❑ Syntax: `<returntype> <name>(<paramslist>, ...)`
 - ... indicates that further parameters can be passed, must be listed only after the required parameters
 - since you specify the parameters as ..., you do not know their names!
- ❑ How to use these additional parameters when they are passed?
 - `stdarg.h` file contains the definition of `va_list` (variable argument list)
 - declare a variable of type `va_list`
 - use the macro `va_start` which initializes your variable to the first of the optional params
 - use the function `va_arg` which returns the next argument

Optional Parameters

❑ Example:

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

int sum(int, ...);

int main(){
    printf("Sum of 15 and 56 = %d\n",    sum(2, 15, 56) );
    return 0;
}

int sum(int num_args, ...){
    int val = 0;
    va_list ap;
    int i;
    va_start(ap, num_args);
    for(i = 0; i < num_args; i++)
        val += va_arg(ap, int);
    va_end(ap);
    return val;
}
```

Multiple Source Files

- ❑ A typical C program: lot of small C programs, rather than a few large ones
 - each .c file contains closely related functions (usually a small number of functions)
 - header files to tie them together
 - Makefiles tells the compiler how to build them

- ❑ Example:

- a calc program defines:
 - a stack structure and its:
 - pop and push functions
 - getch and ungetch to read one symbol at a time
 - getop function to parse numbers and operators
 - main function
- main calls: getop, pop, and push
- getop calls: getch and ungetch
- can be organized in 4 separate files:
- Where to place prototypes and external declarations?
- How to compile the program?

```
/* stack.c */
#include <stdio.h>
int sp = 0;
double val[1000];
void push(double x) {
    ...
}
double pop() {
    ...
}
```

```
/* getch.c */
#include <stdio.h>

ch getch() {
    ...
}

void ungetch(char c) {
}
```

```
/* main.c */
#include <stdio.h>

int main() {
}
```

```
/* getop.c */
#include <stdio.h>

int getop(char[] s) {
}
```

Multiple Source Files: Header File

- ❑ Prototypes can be placed in a single file, called a header file
 - as well as all other shared definitions and declarations
 - typically contains definitions and declarations
 - but not executable code

- ❑ Example: calc program
 - add a header file calc.h contains:
 - prototypes and
 - common declarations
 - and **include** it where needed!

```
/*    getch.c    */
#include <stdio.h>
#include "calc.h"
ch getch(){
    ...
}
void ungetch(char c){
}
```

```
/*    calc.H    */
void push(double);
double pop();
ch getch();
void ungetch(charc);
int getop(char[]);
```

```
/*    main.c    */
#include <stdio.h>
#include "calc.h"
int main(){
}
```

```
/*    stack.c    */
#include <stdio.h>
#include "calc.h"
int sp = 0;
double val[1000];
void push(double x){
    ...
}
double pop(){
    ...
}
```

```
/*    getop.c    */
#include <stdio.h>
#include "calc.h"
int getop(char[] s){
}
```

File Inclusion

- ❑ Syntax:
 - **#include** <filename>
 - search for the file filename in paths according to the compiler defined rules
 - replaced by the content if the file filename
 - **#include** "filename"
 - search for the file filename in source program directory or according to the compiler rules
 - replaced by the content if the file filename
- ❑ When an included file is changed
 - all files depending on it must be recompiled
- ❑ Multiple inclusion of a file: problem
- ❑ Circular inclusion: problem

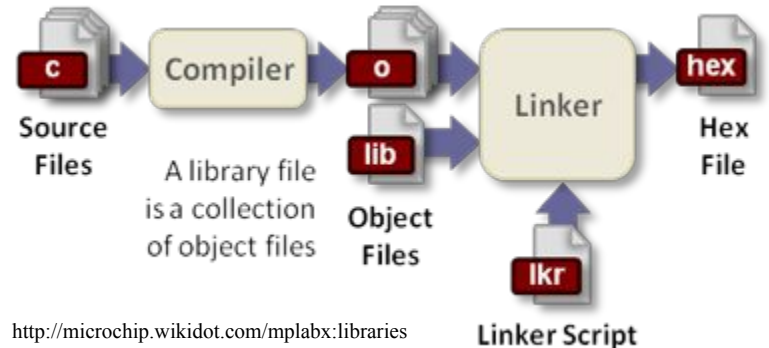
Conditional Inclusion

- ❑ Control preprocessing with conditional statements
- ❑ Syntax:
 - **#if**
 - evaluates a constant integer expression
 - if the expression is non-zero, all following lines until an **#endif** or **#elif** or **#else** are included
 - **#else** , **#elif**
 - provide alternative paths
 - **#endif**
 - marks the end of the conditional block
- ❑ Can be used to avoid repetitive and circular inclusions:
 - included file:

```
#if !defined(HDR)
#define HDR
/* contents of hdr.h go here */
#endif
```

Compiling Multiple Sources

- ❑ The compiler 1st stage is the preprocessor
 - deals with the # directives: define, include, conditional ...
- ❑ The compiler 2nd stage is translate .c files to .o files
 - each .c file will be translated to a single .o file
 - to invoke this stage only, use gcc option -c
- ❑ The compiler then links .o files together
 - along with library files



Makefile

- ❑ To compile multiple source files:

```
gcc -Wall -ansi -o <output> <file1.c> <file2.c> ...
```

- ❑ Or use makefiles:

- Special format file used to build and manage the program automatically
- contains a collection of rules and commands

- ❑ Syntax:

```
<target> [<more targets>] : [<dependent files>]  
<tab> <commands>
```

- ❑ Example:

```
calc: main.c stack.c getch.c getop.c calc.h  
    gcc -Wall -ansi -o calc main.c stack.c getch.c getop.c
```

- How to use: on the command line type: `make calc` ↩

Makefile

❑ Conventional macros:

- `CC` : Program for compiling C programs; default is `'cc'`
- `CFLAGS`: Extra flags to give to the C compiler.

❑ Example:

```
CC=gcc
```

```
CFLAGS= -Wall -ansi
```

```
calc: main.c stack.c getch.c getop.c calc.h
```

```
    ${CC} ${CFLAGS} -o calc main.c stack.c getch.c getop.c
```

❑ Usage:

```
make
```

```
or
```

```
make calc
```

Makefile

❑ At object level:

```
CC=gcc
```

```
CFLAGS= -Wall -ansi
```

```
calc: main.o stack.o getch.o getop.o calc.h
```

```
    ${CC} ${CFLAGS} -o calc main.o stack.o getch.o getop.o
```

```
main.o: main.c calc.h
```

```
    ${CC} ${CFLAGS} -c main.c
```

```
stack.o: stack.c calc.h
```

```
    ${CC} ${CFLAGS} -c stack.c
```

```
getch.o: getch.c calc.h
```

```
    ${CC} ${CFLAGS} -c getch.c
```

```
getop.o: getop.c calc.h
```

```
    ${CC} ${CFLAGS} -c getop.c
```

❑ Can invoke any target by:

make <target>

- ❑ If the dependency object file has not changed since last compile, it will be linked as is. Otherwise, it is recompiled

Makefile

❑ With useful extra targets:

```
CC=gcc
CFLAGS= -Wall -ansi

calc: main.o stack.o getch.o getop.o
    ${CC} ${CFLAGS} -o calc main.o stack.o getch.o getop.o

main.o: main.c calc.h
    ${CC} ${CFLAGS} -c main.c

stack.o: stack.c calc.h
    ${CC} ${CFLAGS} -c stack.c

getch.o: getch.c calc.h
    ${CC} ${CFLAGS} -c getch.c

getop.o: getop.c calc.h
    ${CC} ${CFLAGS} -c getop.c

clean:
    rm *.o calc
```