c programming language

Principles of Programming Languages (Concordia University)

Procedural programming (C).

.Decomposes problems into subroutines,
.A subrutine is a named block of code that should perform one task and can be called when needed.

Imperative programming:

- C uses statements like if, else, while, for, and switch to directly control the flow of execution.
- The programmer explicitly specifies the steps the program should take at each point
- The program's state changes as the program executes.
  C provides manual memory management which is inherently imperative.

Basic Structure of a Multi-file Program.

Header files (.h): These files contain function declarations, macros constants, and type definitions.
Source files (.c): These files contain the actual implementation of the functions.
Main file: The main program logic resides here, including the main() function, and it can call functions defined in other .c files.

Steps for multi-file compilation.

example Project Structure:

```
project/
├── main.c              # contains the main() function
├── math_functions.c    # contains math function implementations
└── math_functions.h    # contains the declarations of math functions.
```

example math_functions (header file):

```
#ifndef MATH_FUNCTIONS_H // Include guard to avoid double inclusion
#define MATH_FUNCTIONS_H
int add(int a, int b);      // declaration of the add function
int subtract(int a, int b); // declaration of the subtract function
#endif
```

.The header file contains the declaration of functions (add and subtract) which are implemented elsewhere.
.The # ifndef and # define directives are used to prevent the header from being included multiple times in a project (include guard).

Library header-files:

. Header file provides the interface to a module.
. The contents of a well defined module should treat the source file as a black box
. The Header filename end in .h
. The filname conventionally matches its corresponding .c file Point.h / Point.c
. Header files are surronded in macro "include guards" (#ifndef and # endif)

Things to not include in a library header file:
- Static function prototypes, constants, and macros
- int Global; (global variables need to be in the .c file itself).

Things to include in a library header file:
- include guard (making sure to not include the same header file twice to the same project.
- other include files
- new data types.
- public function prototypes, constants, and macros.
- extern int Global.

example math_function.c (Source file)

```
#include "math_functions.h" // include the corresponding header file
int add(int a, int b){
    return a+b;
}
int subtract(int a, int b){
    return a-b;
}
```

.The source file contains the actual implementation of the functions declared in math_functions.h
.The Main Program (main.c) includes the header file so it knows the signature of add() and subtract() and can use them.

Compiling Multi-file programs.

Single step_compilation

- You can compile all the .c files in one command using:

```
gcc main.c math_functions.c -o program
```

1. This compiles both main.c and math_functions.c
2. Then, links the resulting object files into a single executable called program

Separate Compilation and linking:

- You can also compile each file separately into object files .o, then link them together
  1. Compile each file into an object file (.o):

```
gcc -c main.c            # Generates main.o
gcc -c math_functions.c  # Generates math_functions.o
```

  2. Link the object files to create the final executable:

```
gcc main.o math_functions.o -o program
```

- In this process, the -c flag tells gcc to compile the files into object files .o, but not link them. The final gcc command links the object files into an executable.

How the compilation works.

Processing: Each .c is processed, which includes expanding macros and including the contents of the header files.

Compilation: Each .c file is compiled into an object file .o

linking: The object files are linked together to form the final executable. Any unresolved function calls (like calling add() in main.c are linked to the actual function definitions (in math_functions.o).

Advantages of Multi file compilation.

Modularity: You can split large programs into manageable parts.

Reusability: Function in a .c file can be reused in other projects simply by including the .h file and compiling the .c file.

Reduced Compilation Time: If you only modify one .c file, you don't need to recompile the entire program; just recompile the modified file and link again.

Compilation warnings

- The -Wall flag is a compiler option that stands for "Warn all"
- -Wall enables most of the commonly useful warning messages to help you identify potential issues in your code.
- Identifies potential bugs
- Improves code quality
- Enforces good practices.

```
gcc -Wall main.c math_functions.c -o program.
```

Basic data types.

Integer types: char (1 byte), short (2 bytes), int (4 bytes), long (4 to 8 bytes), long long (8 bytes)

Floating point types: float (4 bytes), double (8 bytes), long double (8 or 16 bytes)

- The size of these data can vary across different platforms
- The appropriate data type must be chosen based on the size, and memory requirements

Bitwise operators

Bitwise and (&): compares each bit of two numbers. If both bits are 1 the result is 1 otherwise it is 0.

example:

```
int a = 6    // 00000110 in binary
int b = 3    // 00000011 in binary
int result = a & b;  // 00000010 (decimal 2)
```

Bitwise OR (|): compares each bit of two numbers. If at least one of the bits is 1, the result is 1 otherwise, it is 0.

example:

```
int a = 6    // 00000110 in binary
int b = 3    // 00000011 in binary
int result = a | b;  // 00000111 (decimal 7)
```

Bitwise XOR (^): compares each bit of two numbers, if the bits are different the result is 1; if they are the same, the result is 0.

example:
```
int a = 6    // 00000110 in binary
int b = 3    // 00000011 in binary
int result = a ^ b; // 00000101 (decimal 5)
```

Bitwise NOT (~): flips all the bits (turns 1 into 0 and vis versa) it's called a complement.

example:
```
int a = 6    // 00000110 in binary
int result = ~a // 11111001 (-7 in two's complement)
```

Left shift: shifts the bits of a number to the left by a specified number of positions.
This is equivalent to multiplying the number by 2 for each shift position.

example:
```
int a = 6  // 00000110
int result = a<< 1; // 00001100 (decimal 12)
```

Right shift: shifts the bits of a number to the right by a specified number of positions.
This is equivalent to multiplying the number by 2 for each shift position

example:
```
int a = 6  // 00000110
int result = a >>1; // 00001100 (decimal 12)
```

Assigning value of bitwise to itself

x = x ^ a → x ^= a
x = x & a → x |= a
x = x | a → x &= a

Struct Structure

- A struct (structure) is a user-defined data type that allows you to group variables of different types under a single name.
- It's particularly useful when you need to manage multiple pieces of data as one entity.

example
```
struct StructName {
    data_type member1;
    data_type member 2;
};
```

- Each member can be of any data type, and they can even be arrays, pointers, or other structures.
- They look like java classes except they don't have methods.

Initializing a struct

struct StructName strmem1 = { member1, member2};

Accessing struct Members.

- Use the dot operator (.) to access individual members of a struct variable.
- If you have a pointer to a structure, you can access the members using the arrow operator (→)
- Setting up a struct does not allocate memory space to them, it defines the format of the new type.

example
```
struct Student{
    char name[50];
    int age;
    float GPA;
};
int main(){
    struct Student student1 = {"Alice", 30, 3.7};
    struct Student *ptr = &student1; // pointer to the student structure.
    // accessing members using the arrow operator:
    printf("Student Name: %s\n", ptr-> name); // or student1.name
    printf("Student Age: %d\n", ptr-> age); // or student1.age
    printf("Student GPA: %f\n", ptr-> GPA); // or student1.GPA
    return 0;
}
```

functions.

A function is a block of code designed to perform a particular task.
Every C program has at least one function which is main()

The basic Syntaxc for a function:

```
return_type  function_name (parameter_list) {
    // code to be executed
}
```

example :

```
# include <stdio.h>
void greet () {
    printf ("Hello World!");
}
int main () {
    greetC); // call the function
    return 0;
} // Output: Hello World!
```

example 2 : Function with parameters and return Type.

```
# include < stdio.h>
int  add (int a , int b) {
    return a+b;
}
int  main () {
   int result = add (3,3); // call function with arguments
   printf ("The sum is: %d \n", result);
   return 0;
} // Output:
```

function declaration and definition.

Declaration: Declares the function's signature to the compiler It includes the the function's name, return type and parameters.

Definition: Contains the actual body of the function , specifying what it does.

You can declare a function before main and define it later:

example :

```
# include <stdio.h>
int  multiply (int a , int b); // function declaration (prototype)
int  main () {
   int result = multiply (4,5),
   printf("Result: %d \n", result);
   return 0;
}
// Function definition
int multiply(int a , int b){
      return a* b;
}
```

function Scope and Liftime

Local variables : Variables declared inside a function are only accessible within the function (local scope) and are destroyed when the function exits.

Global variables: Declared outside all functions and can be accessed by any function in the program. They persist throughout the program execution.

function parameters:

Pass by value : The default in C. When you pass an argument, the function works with a copy of the argument , so changes made to the parameter inside the function don't affect the original argument.

Pass by reference : Achieved using pointers, allowing the function to modify the original variable.

example :

```
# include <stdio.h>
 void byValue (int a) {
      a = 100;
 }
 Void  byReference (int  * a){
     * a = 100,
 }
 int  main () {
      int x = 10;
      by value (x);
      printf(" After by Value : %d \n",  x ); // output : 10
      by Reference (x);
      printf("After beReference : %d \n", x ); // output : 100
      return 0;
 }
```

Static

It modifies the scope and liftime of functions and variables depending on how and when it is used.

Static Variables

Static Local variables (within a function):

- When a variable is declared as static inside a function it maintains it's value between function calls. Normally, Local variables inside functions are destroyed after the function returns.
- Static local variables are initialized only once, and their value is preserved for the liftime of the program.

example:

```c
#include <stdio.h>
void counter(){
    static int count = 0; // Static variable initialized once.
    count++;
    printf("Count is: %d \n", count);
}
int main(){
    counter(); // output: Count is: 1
    counter(); // output: Count is: 2
    counter(); // output: Count is: 3
    return 0;
}
```

Static local variables.

- Static local variables are initialized only once (at program startup or the first call)
- They retain their value between function calls.
- Their scope is limited to the function in which they are declared, but lifetime is the entire duration of the program.

Static global variables

A global variable made static is limited to the file in which it is defined. They have internal linkage so it can not be accessed outside it's source file.

example:

```c
#include <stdio.h>
static int globalCount = 0; // Static global variable.
void increment(){
    globalCount++;
    printf("Global count: %d \n", globalCount);
}
int main(){
    increment(); // Output: Global count: 1
    increment(); // output: Global count: 2
    return 0;
}
```

- static global variables can only be accessed within the file where they are declared.
- They cannot be used in other files, even if declared with extern.

Static functions

.It has internal linkage. It can only be called from within the same file.

example: External file access:

```c
// another_file.c
#include <stdio.h>
// This will fail because greet() is static in the other file
extern void greet(); // Linker error: undefined reference to greet
int main(){
    greet(); // Error: greet is not visible here.
    return 0;
}
```

```c
// file.c
#include <studio.h>
static void greet(){
    printf("Hello");
}
```


Common Use Cases for static

1. Encapsulation: You may use static to hide global variables or functions within a single file, to avoid conflicts and ensure that other files cannot accidentally use them.
2. Persistence of Data: Use static variables inside functions when you need to keep track of data across function calls.

Memory Management: Advanced Details

Pointers:

A pointer is a variable that stores the address of another variable.
Pointer declaration: `int *p;`
Dereferencing: Accessing the value at the memory location via `*p`
Pointer arithmetic: You can add or subtract integers to a pointer to move to different memory locations (useful when dealing with arrays).

example:

```c
#include <stdio.h>
int main(){
    int arr[5] = {1,2,3,4,5}; //pointer to the first element of the array
    int *ptr = arr;
    for (int i=0; i<5; i++){
        printf("Element %d: %d \n", i, *(ptr+i)); //pointer arithmetic
    }
    return 0;
}
```

Pointer to function

A function pointer allows you to point to a function and call it dynamically

example:

```c
#include <stdio.h>
int add(int a, int b){
    return a+b;
}
int subtract(int a, int b){
    return a-b;
}
int main(){
    int (*operation)(int,int); // Declare a point to a function that takes two ints and returns an int
    operation = &add;
    printf("Addition: %d \n", operation(5,3)); // call the add function via pointer
    operation = &subtract
    printf("Subtraction: %d \n", operation(5,3)); // call the substract function via pointer.
    return 0;
}
```

Pointers in detail:

Pointer: A variable-like reference that holds a memory address to another variable
* : indirection operator (value at address)
& : memory address of variable.

example:
```
int age = 21;
printf("Address of age in the memory: %p", &age); // 000F1236
printf("Value of age: %d", age); // 21

int *pAge = &age;
printf("Address of age by pointer: %p", pAge); // 000F1236
printf("Value at stored address: %d", *pAge); // 21
```

- It is good practice to assign NULL if declaring a pointer and not yet assigning a value.

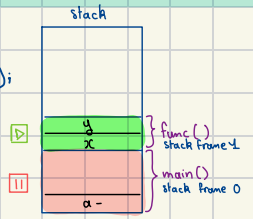
Aathomatic Memory Management (Stack frames)

- Stack is a region of memory that stores local variables, function parameters and control flow information (like return addresses).
- It operates in a Last in, first out manner
- Each time a function is called, a new stack frame is created on the stack.
- When the function returns (finishes executing), the stack frame is destroyed and memory is deallocated automatically.

Content of a Stack frame:

1. Function parameters: Any arguments passed to the function.
2. Local variables: Variables declared inside the function.
3. Return address: The address of the instruction that called the function (so the program knows where to return after the function ends).
4. Saved registers: registers that the function may need to restore after it finishes.

example:
```
# include <stdio.h>
void func(int x){
    int y = 10;
    printf("x = %d, y = %d \n", x, y);
}
int main(){
    int a = 5;
    func(a);
    return 0;
}
```



Dynamic Memory Management (Heap)

- Heap is a region of memory used for dynamic allocation
- Memory is allocated and deallocated manually by the programmer.
- Where we use malloc(), calloc(), realloc() and free().
- It is a free form of memory area where you can allocate and release memory blocks as needed.

Allocating memory on the heap.

We first need to include <stdlib.h> library, as it contains the declaration of the following functions.

malloc(size): Allocates the size bytes memory and returns a void pointer to to the allocated memory.

calloc(n, size): Allocates memory of an array of n elements, each size bytes, and initializes them to zero

realloc(ptr, size): Resizes the previously allocated memory block to the new size, and initializes the new elements with garbage values (deffault).

free(ptr): Frees the memory pointed to by ptr that was previously allocated by malloc() or calloc()

example:
```
# include <stdio.h>
# include <stdlib.h>
int main(){
    int n;
    printf("Enter size of array \n");
    scanf("%d", &n); // n = 2.
    int *A = (int*) malloc(n * sizeof(int)); // dynamically allocated array.
    if(A == null){ // check if memory allocation succeded.
        printf("Memory allocation failed \n");
        return 1;}
    for(int i=0; i<n; i++){
        A[i] = i+1; // assign a value to each element of the array
    }
    for(int i=0; i<n; i++){
        printf("%d ", A[i]); // output: 1 2
    }
    int *B = (int*) realloc(A, 2*n*sizeof(int)); // dianamically reallocate the array
    for(int i=0; i<2*n; i++){
        printf("%d ", B[i]); // output: 1 2 285687 285689
    }
    free(B); // free memory space heap (the space is now deallocated)
    return 0;
}
```

- To assign a value to an element of an array using pointers.

A[0] = 1; or *A = 1;

A[1] = 3; or *(A+1) = 3;

address of the second element of the array.

Dynamic Memory allocation:

Malloc: Allocates memory and returns a pointer to the beginning of the block.
- It doesn't initialize the memory
- You must cast the returned pointer to the appropriate type

Calloc: Similar to malloc, but it also initializes the alocated memory to zero
Syntax: calloc(num_elements, size of each element).

Realloc: Resizes previously allocated memory blocks.
- It eather expands or shrinks the allocated memory block and may move it to a new location if necessary.

Static management

- Creating a global variable that lasts throughout the whole program.