Functions

Outline

- Passing arguments
  - pass by reference, pass by value
- Declarations and calls
  - definition, allusion, function call
- Examples
- Recursion
- The main function
- Function pointers

Passing arguments

- Because C passes arguments by value, a function can assign values to the formal arguments without affecting the actual arguments

- If you **want** a function to change the value of an object, you must pass a pointer to the object and then make an assignment through the dereferenced pointer.
  - ***remember scanf function !!!***

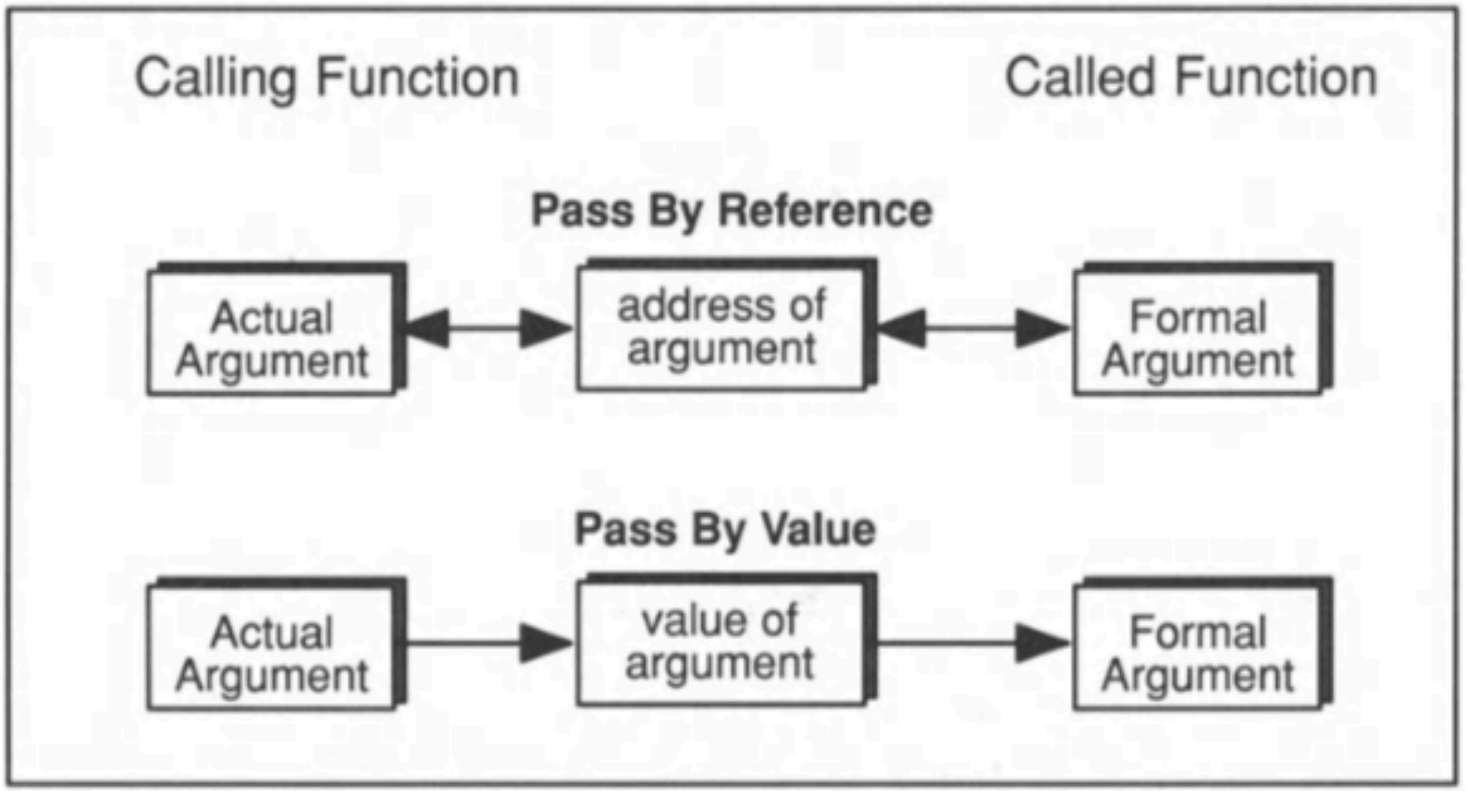


Figure 9-1. Pass By Reference vs. Pass By Value. In Pass By Reference, the actual and formal arguments refer to the same memory area; in Pass By Value, the formal argument is a copy of the actual argument.

Declarations and calls

- Definition
  - Actually defines what the function does, as well as number and type of arguments
- Function Allusion
  - Declares a function that is defined somewhere else
  - Also specifies what kind of value the function returns.
- Function Call
  - Invokes a function, causing program execution to jump to the next invoked function. When the function returns, execution resumes at the point just after the call

Function definition

- A very simple example
  - no arguments
  - no return

```
void simpleFunction1 ( void ) {

        printf("\nThis is
simpleFunction1\n");
}
```

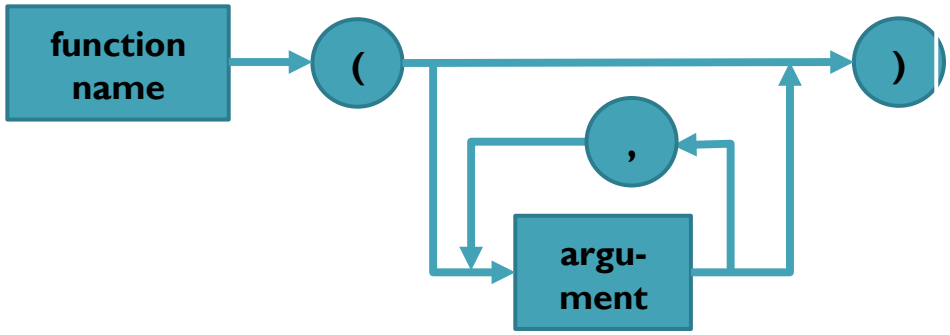
- A relatively complex example
  - a function to calculate factorial n

```
int factorial( int n) {
        int i,f=1;
        for(i=2;i<=n;i++)
                f = f * i;
        return f;
}
```

Function allusion

- void simpleFunction1(void); // prototype of the function
- simpleFunction1(); // alternative to the above allusion
- extern float simpleFunction2(); // no input argument, returns float
- int factorial(int); // takes integer, returns integer
- void sortArray(int *, int); // takes 1 int-pointer, 1 int, returns nothing
- float *mergeSort(float *, int, float *, int, int *);

Function call




```
printf("Hello World\n");
printf("Result is %d\n", factorial(10));
scanf("%s", str);
x = factorial(n) / factorial(m);
printf("%d : %d  : %s : %d\n", i, j, line, rc);
matT = transpose(mat, rows, cols);
```

- Number of the arguments in the function-definition and function-call should be consistent.
- When we call a function, argument types should be consistent and in the same order as they defined.

Order of functions

- In order to use a function you must define it beforehand.
  - In order to use your own function in the ***main() function***, you should define it ***before the main()*** in the same file

- It is also possible to use function allusion (function prototype)
  - You can write the prototype of your function before the ***main() function*** and use it anywhere (main() or any other function of yours)

Function arguments

Passing arrays as function parameter

- Several ways to do it…
- Do NOT forget
  - no boundary checking !
  - remember your motivation to create a function
- Using actual array size
  - void myFunction(int ar[5])
- Using array and a size parameter
  - void myFunction(int ar[], int size)
- Using a pointer and an integer
  - void myFunction(int *ar, int size)

How to return an array from a function

- We don't return an array from functions, rather we return a pointer holding the base address of the array to be returned.
- We must, make sure that the array exists after the function ends!
  - you can **NOT** return local arrays!
- **SOLUTION** : dynamic memory allocation + pointers

EXAMPLE - 1

- Create a sort function for one dimensional arrays
- Use any type of sorting algorithm

EXAMPLE -2

- Write a function that compresses a sparse matrix
- The function should take a matrix as a parameter
- The function should return a new matrix 3 x n or n x 3

RECURSION

- A recursive function is one that calls itself.
  - An example is given on the right side
- It is important to notice that this function will call itself forever.
  - Actually not forever, but till the computer runs out of stack memory
  - It means a runtime error
- Thus, remember to include a stop point in your recursive functions.

```
void recurse ()  {
        static count = 1;
        printf("%d\n", count);
        count++;
        recurse();
}
main()  {
        recurse();
}
```

Recursion

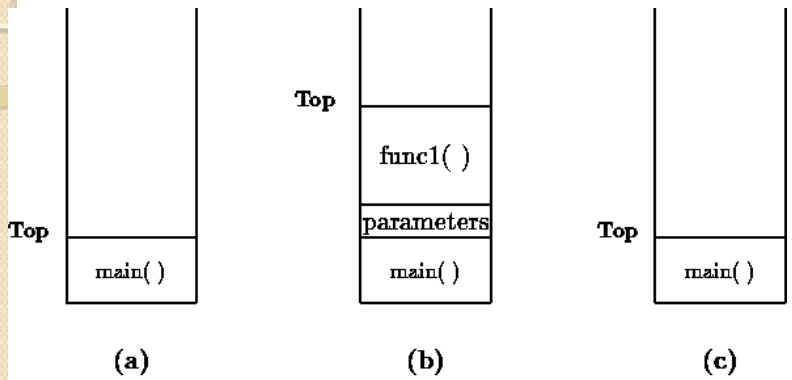


Figure 14.13: Organization of the Stack

- When a program begins executing in the function main(), space is allocated on the stack for all variables declared within main(), **Figure 14.13(a)**
- If main() calls a function, func1(), additional storage is allocated for the variables in func1() at the top of the stack **Figure 14.13(b)**
  - Notice that the parameters passed by main() to func1() are also stored on the stack.
- When func1() returns, storage for its local variables is deallocated, and the top of the stack returns to the 1st position **Figure 14.13(c)**
- As can be seen, the memory allocated in the stack area is used and reused during program execution.
  - *It should be clear that memory allocated in this area will contain garbage values left over from previous usage.*

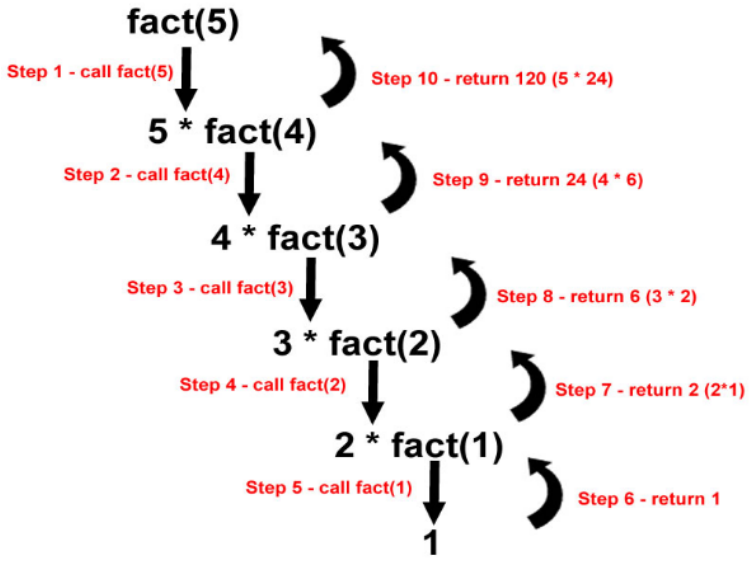
Recursion

- A few examples to solve with recursion
  - Factorial – n!
  - Fibonacci numbers – $F_{n+1} = F_n + F_{n-1}$
  - Binary search
  - Depth-first search

```
int fact( int n )  {
        if( n <= 1 )
                return 1;
        else
                return n*fact(n-1);
}
main()  {
        printf("5! is %d\n", fact(5));
}
```

Recursion

- A few examples to solve with recursion
  - Factorial – n!
  - Fibonacci numbers – $F_{n+1} = F_n + F_{n-1}$
  - Binary search
  - Depth-first search



MAIN() FUNCTION

- All C programs must contain a function called ***main()***, which is always the first function executed in a C program.
- When ***main()*** returns, the program is done.
- The compiler treats the main() function like any other function, except that at runtime the host environment is responsible for providing two arguments
  - **argc** – number of arguments that are presented at the command line
  - **argv** – an array of pointers to the command line arguments

```
main(int argc, char *argv[])  {

        while(--argc > 0 )
                printf("%s\n", *++argv);
        exit(0);
}
```

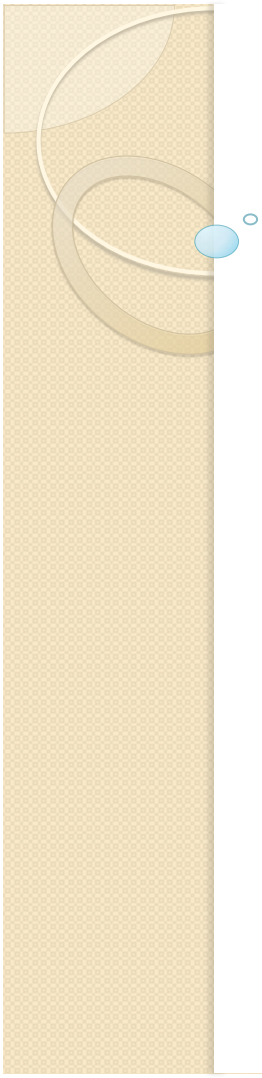
MAIN() FUNCTION

- A better way to handle command line arguments
  - getopt
- The getopt() function parses the command-line arguments. Its arguments *argc* and *argv* are the argument count and array as passed to the main() function on program invocation.
- The variable *optind* is the index of the next element to be processed in *argv*. The system initializes this value to 1. If there are no more option characters, getopt() returns -1.
- The variable *optstring* is a string containing all options characters (e.g. "abc:" in the example)

```
#include <unistd.h>
int getopt(int argc, char * const argv[], const
char *optstring);
extern char *optarg;
extern int optind, opterr, optopt;
```

```
while ((c = getopt (argc, argv, "abc:")) != -1)
  switch (c) {
    case 'a':
      aflag = 1;
      break;
....
    default:
      abort ();
    }
```

http://www.gnu.org/software/libc/manual/html_node/Example-of-Getopt.html#Example-of-Getopt



Function Pointers

FUNCTION POINTERS

- Sometimes we would like to choose different behaviors at different times in the same piece of code or function.
- For instance in a sorting routine, we want to allow the function's caller to choose the order in which the data is sorted
- We can use some functions as arguments to other functions through the function pointers.
- extern int f(); // f by itself is a pointer to a function. But it is illegal to assign a value to f (similar to int ar[5]; => ar is also a pointer, but it cannot be on the left-side of an assignment)
- Definition:
  - int **(*pf)**(); // pf is a pointer to a function returning an int.

Define and assign a value to a function pointer

- Definition:
  - **int (*pf)();** // pf is a pointer to a function returning an int.
  - The () around *pf are necessary for correct grouping. Without them: int *pf(); // this would be a function returning an int pointer
- Assigning value:
  - {
  - extern int f1();
  - int (*pf) (); // pf is a pointer to a function returning an int.
  - pf=f1; // assign the address of f1 to pf
  - pf=f1(); // ILLEGAL, f1 returns an int, but pf is a pointer
  - pf=&f1(); //ILLEGAL, cannot take the address of a function result
  - pf=&f1; // ILLEGAL, &f1 is a pointer to a pointer, but pf is a pointer to an int
  - }

Return type argument

```
extern int if1(), if2(), (*pif)();
extern float ff1(), (*pff)();
extern char cf1(), (*pcf)();

main()
{
  pif = if1; /* Legal -- types match */
  pif = cf1; /* ILLEGAL -- type mismatch */
  pff = if2; /* ILLEGAL -- type mismatch */
  pcf = cf1; /* Legal -- types match */
  if1 = if2; /* ILLEGAL -- Assign to a constant */
}
```

Calling a function using pointers

- Use the same syntax we use to declare the function pointer, include possible arguments. E.g.:
  - {
  - extern int f1();
  - int (*pf) ();
  - int answer;
  - pf=f1;
  - answer=(*pf)(a); // calls f1() with argument a => f1(a)
  - …
  - }

Example (check the whole code that was shown in the class)

- We would like to either add the values of all integers between x and y and return the sum; or want to add square of each integer between x and y.
- We will have one function to cumulatively sum the numbers.
- This function will take a pointer as one of its arguments. So that, user can decide if she wants to use find sum(i) or sum(i^2):

```
/*   Function returns the argument a. */
int self_i(int a)
{
        return a;
}
/*   Function returns the square of a. */
int square_i(int a)
{
        return a * a;
}
/*   Function sums values of *fp applied to
integers from x to y. */
int sum_generic(int (*fp)(), int x, int y)
{
        int i, cumsum = 0;
        for (i = x; i <= y; i++)
                cumsum += (*fp)(i);
        return cumsum;
}
```