Programming in C III

Content

- Strings
- Pointers
- Functions

Strings

- Character arrays
  - Strings are 1D arrays of characters
  - Strings must be terminated by the null character '\0' which is (naturally) called the end-of-string character.
  - Strings must be declared before they are used like any other variables in C.
  - Unlike other 1D arrays the number of elements set for a string set during declaration is only an upper limit.
  - Initializing a string can be done in three ways: 1) at declaration, 2) by reading in a value for the string, and 3) by using the **strcpy** function.

Strings

- String initialization
  - **By reading in a value**
    - Example:

```
char name[34];
scanf("%s",name);
```


Strings

- String initialization
  - **By strcpy function**
    - To use the *strcpy function* be sure to include the **string.h** header file
    - Example:

```
#include <string.h>
main ()
{
char job[50];
strcpy(job,"Professor");
printf("You are a %s \n",job);
}
```


Strings

- String I/O special functions
  - gets(string_name);
  - puts(string_name);
  - Example:

```
char a_string[100];
printf("Please enter a sentence\n");
gets(a_string);
puts(a_string);
```

  - ***gets*** function reads in a string from the keyboard
  - ***puts*** function displays a string on the monitor

Strings

- More string functions

Function	Operation
strcat	Appends to a string
strchr	Finds first occurrence of a given character
strcmp	Compares two strings
strcmpi	Compares two, strings, non-case sensitive
strcpy	Copies one string to another
strlen	Finds length of a string
strncat	Appends ***n*** characters of string
strncmp	Compares ***n*** characters of two strings
strncpy	Copies ***n*** characters of one string to another
strnset	Sets ***n*** characters of string to a given character
strrchr	Finds last occurrence of given character in string
strspn	Finds first substring from given character set in string

Pointers

- A pointer is a variable that stores the memory address of another variable as its value.
- A pointer variable points to a data type (like int) of the same type, and is created with the * operator.
- Pointers contain **Memory Addresses**, Not **Data Values**

Pointers

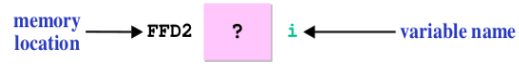
- Pointers enable us to
  - effectively represent sophisticated data structures
  - change values of actual arguments passed to functions (“call-by-reference”)
  - work with memory which has been dynamically allocated
  - more concisely and efficiently deal with arrays

Pointers

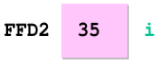
- Declaring a simple variable, like

```
int i;
```

- a memory location with a certain address is set aside for any values that will be placed in `i`. We thus have the following picture:



- After the statement `i=35;` the location corresponding to `i` will be filled



Pointers

- You can find out the memory address of a variable by simply using the address operator &.
- **Example:** `&v`
- The above expression should be read as “**address of v**”, and it returns the memory address of the variable **v**.
- **Example:**

```
#include <stdio.h>
main() {
   float v;
   v=5.17;
printf("The value of v is %f\n",v);
printf("The address of v is %X\n",&v); }
```


Pointers

- Like all other C variables, pointers must be declared before they are used.
- **Example:** Pointer declaration

```
int *p;
float *y;
```

- The prefix ***** defines the variable to a pointer.
- In the above example, **p** is the type “**pointer to integer**” and **y** is the type “**pointer to float**”.

Pointers

- Once a pointer has been declared, it can be assigned an address. This is usually done with the address operator.
- **Example:** Pointer declaration

```
int *p;
int x;
p=&x;
```

- After this assignment, we say that **p** is "**referring to** " the variable **x** or " **pointing to** " the variable **x**. The pointer **p** contains the memory address of the variable **x**.

Pointers

- Consider the following example which returns the contents of the address stored in a pointer variable.
- **Example:** Using a Pointer

```
#include <stdio.h>
main() {
int a=1,b=78,*p;
p=&a;
b=*p; /* equivalent to b=a */
printf("The value of b is %d\n",b); }
```

- The ***p** expression is read as "**contents of p**". What is returned is the value stored at the memory address p.

Functions

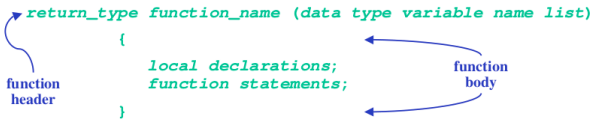
- A function in C is a **small "sub-program"** that performs a particular task
- It is a block of code which only runs when it is called
- Why functions?
  - **Don't have to repeat the same block of code many times in your code. Make that code block a function and call it when needed.**
  - **Function portability: useful functions can be used in a number of programs.**
  - **Supports the top-down technique for devising a program algorithm. Make an outline and hierarchy of the steps needed to solve your problem and create a function for each step.**
  - **Easy to debug. Get one function working well then move on to the others.**
  - **Easy to modify and expand. Just add more functions to extend program capability**
  - **For a large programming project, you will code only a small fraction of the program.**
  - **Make program self-documenting and readable.**

Functions

- To use functions, the programmer must do three things
  - Define the function
  - Declare the function
  - Use the function in the main code.

Functions

- Function definitions have the following syntax:



  - **return_type** in the function header tells the type of the value returned by the function (default is int)
  - **data type variable name list** tells what arguments the function needs when it is called (and what their types are)
  - **local declarations** in the function body are local constants and variables the function needs for its calculations.

Functions

- Example: a function that calculates n!

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

Functions

- Some functions will not actually return a value or need any arguments. For these functions the keyword void is used.
- Example:

```
void write_header(void) {
printf("Last Modified: ");
printf("12/04/95\n");
}
```

- The **1st void** keyword indicates that no value will be returned
- The **2nd void** keyword indicates that no arguments are needed for the function.
- This makes sense because all this function does is print out a header statement.

Functions

- A function returns a value to the calling program with the use of the keyword **return**, followed by a data variable or constant value.
- Example:

```
return 3;
return n;
return ++a;
return (a*b);
```

- The data type of the return expression must match that of the declared **return_type** for the function.

Functions

- Using a function is known as **calling** or **invoking** a function
- **Example 1:** Using our previous factorial program

```
ans = factorial(9);
```

- **Example 2:** To invoke our write_header function

```
write_header();
```

- When your program encounters a function invocation, control passes to the function.
- When the function is completed, control passes back to the main program.

Functions

- On implementing functions in C, two variables may be used; **Local variables** and **Global variables**
- **Local variables**
  - They exist and their names have meaning only while the function is being executed.
  - They are unknown to other functions.
  - When the function is exited, the values of automatic variables are not retained.
  - If a local variable has the same name as a global variable, only the local variable is changed while in the function. Once the function is exited, the global variable has the same value as when the function started.
- **Global variables**
  - Is declared at the beginning of a program outside all functions
  - Can be accessed and changed by any function in the program

Functions

- **Example :** Function in a program

```
#include <stdio.h>
int factorial (int n)
{
int i,product=1;
for (i=2; i<=n; ++i)
product *= i;
return product;}
int main() {
int x,ans;
printf ("Enter a number: ");
scanf("%d",&x);
ans = factorial(x);
printf ("The factorial of %d is %d\n",x,ans);
}
```


Functions

- Function prototypes are used to declare a function so that it can be used in a program before the function is actually defined.
- **Example 1:** Consider the previous program using a function prototype

```
#include <stdio.h>
int factorial (int n);       // A function prototype
int main() {
int x,ans;
printf ("Enter a number: ");
scanf("%d",&x);
ans = factorial(x);
printf ("The factorial of %d is %d\n",x,ans);
}
int factorial (int n)
{
int i,product=1;
for (i=2; i<=n; ++i)
product *= i;
return product;}
```

Functions

- Recursion
  - Recursion is the process in which a function repeatedly calls itself to perform calculations.
  - **Example:** Consider the previous factorial function using recursion

```
int factorial(int n) {
int result;
    if (n<=1)
    result=1;
    else
    result = n * factorial(n-1);
return result;
}
```


Functions

- Function call
  - Functions can be invoked in two ways: **Call by Value** or **Call by Reference**.
  - **Actual parameters** are the values passed to a function during a function call, whereas **formal parameters** are the variables declared in the function definition that receive these values
  - Also, they may be referred to as **arguments** and **parameters**. **Arguments** meaning actual parameters and **parameters** meaning formal parameters.

Functions

- Difference between the Call by Value and Call by Reference

Call By Value	Call By Reference
While calling a function, we pass the values of variables to it. Such functions are known as "Call By Values".	While calling a function, instead of passing the values of variables, we pass the address of variables(location of variables) to the function known as "Call By References.
In this method, the value of each variable in the calling function is copied into corresponding dummy variables of the called function.	In this method, the address of actual variables in the calling function is copied into the dummy variables of the called function.
With this method, the changes made to the dummy variables in the called function have no effect on the values of actual variables in the calling function.	With this method, using addresses we would have access to the actual variables and hence we would be able to manipulate them.
In call-by-values, we cannot alter the values of actual variables through function calls.	In call by reference, we can alter the values of variables through function calls.
Values of variables are passed by the Simple technique.	Pointer variables are necessary to define to store the address values of variables.

Functions

- **Example:** Call by Value

```
#include <stdio.h>
void swapx(int x, int y);  // Function Prototype
 int main()
{
    int a = 10, b = 20;
     // Pass by Values
    swapx(a, b); // Actual Parameters
    printf("In the Caller:\na = %d b = %d\n", a, b);
     return 0; }
// Swap functions that swaps two values
void swapx(int x, int y) // Formal Parameters
{
    int t;
    t = x;
    x = y;
    y = t;
   printf("Inside Function:\nx = %d y = %d\n", x, y); }
```


Functions

- In call by value method of parameter passing, the values of actual parameters are copied to the function's formal parameters.
  - There are two copies of parameters stored in different memory locations.
  - One is the original copy and the other is the function copy.
  - Any changes made inside functions are not reflected in the actual parameters of the caller.

Functions

- **Example:** Call by Reference

```
#include <stdio.h>
void swapx(int*, int*);  // Function Prototype
 int main()
{
    int a = 10, b = 20;
     // Pass Reference
    swapx(&a, &b); // Actual Parameters
    printf("In the Caller:\na = %d b = %d\n", a, b);
     return 0; }
// Function to swap two variables by references
void swapx(int *x, int *y) // Formal Parameters
{
    int t;
    t = *x;
    *x = *y;
    *y = t;
   printf("Inside Function:\nx = %d y = %d\n", *x, *y); }
```

Functions

- In call by reference method of parameter passing, the address of the actual parameters is passed to the function as the formal parameters.
  - Both the actual and formal parameters refer to the same locations.
  - Any changes made inside the function are actually reflected in the actual parameters of the caller.