

# C Programming II

Alexander B. Pacheco LTS Research Computing June 3, 2015

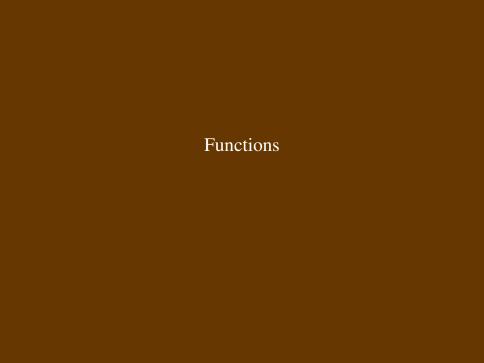
### Outline

Functions

2 Arrays

3 Pointers

4 Input/Output



#### **Functions**

- A function is a group of statements that together perform a task.
- Every C program has at least one function, which is main()
- Functions receive either a fixed or variable amount of arguments.
- Functions can only return one value, or return no value (void).
- In C, arguments are **passed by value** to functions
- How to return value? Pointers
- Functions are defined using the following syntax:

```
return_type function_name( parameter list )
{
  body of the function
}
```

- A function declaration tells the compiler about a function's name, return type, and parameters.
- A function **definition** provides the actual body of the function.

#### **Function Definition**

- **Return Type:** Function's return type is the data type of the value the function returns. When there is no return value, return void.
- Function Name: This is the actual name of the function.
- **Parameter:** The parameter list refers to the type, order, and number of the parameters of a function. A function may contain no parameters.
- Function Body: The function body contains a collection of statements that define the function behavior.

```
/* function returning the max between two numbers */
int max(int i, int j)
{
    /* local variable declaration */
    int result;

if (i > j)
    result = i;
    else
        result = j;
    return result;
}
```

# Example of using a Function

```
#include <stdio.h>
/* function declaration */
int max(int i, int j);
int main() {
  /* local variable definition */
  int i = 100, j = 200, maxval;
 /* calling a function to get max value */
 maxval = max(a, b):
  printf( "Max value is : %d\n", maxval );
  return 0:
/* function returning the max between two numbers */
int max(int i, int j)
 /* local variable declaration */
  int result:
 if (i > j)
    result = i;
  else
    result = j;
  return result;
```

### Scope Rules: Local & Global Variables I

- A scope is a region of the program where a defined variable can have its existence and beyond that variable can not be accessed.
- Local Variables: declared inside a function or block.
   can be used only by statements that are inside that function or block of code.
   Local variables are not known to functions outside their own.
- Global Variables: defined outside of a function, usually on top of the program.
   will hold their value throughout the lifetime of your program and,
   they can be accessed inside any of the functions defined for the program.
- A program can have same name for local and global variables but value of local variable inside a function will take preference.

# Scope Rules: Local & Global Variables II

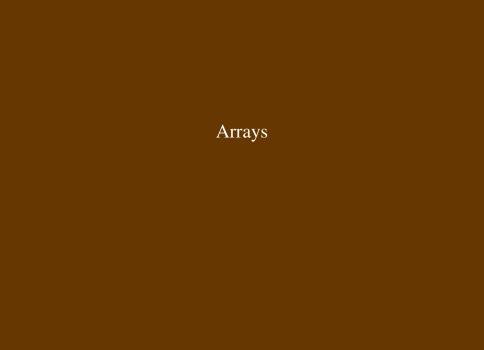
```
#include <stdio.h>
/* global variable declaration */
int a = 20;
int main ()
 /* local variable declaration in main function */
 int. a = 10:
 int b = 20;
 int c = 0;
 printf ("value of a in main() = %d\n", a);
 c = sum(a, b);
 printf ("value of c in main() = %d\n", c);
  return 0;
/* function to add two integers */
int sum(int a, int b)
 printf ("value of a in sum() = %d\n", a);
 printf ("value of b in sum() = %d\n", b);
  return a + b;
   value of a in main() = 10
   value of a in sum() = 10
    value of b in sum() = 20
    value of c in main() = 30
```

### Initializing Local & Global Variables

- Local Variables are not initialized by the system, the programmer must initialize
  it.
- Global variables are automatically initialized by the system depending on the data type

Data Type	Initial Default Value
int	0
char	'\0'
float	0
double	0
pointer	NULL

• It is a good programming practice to initialize variables properly otherwise, your program may produce unexpected results because uninitialized variables will take some garbage value already available at its memory location.



### Arrays

- Arrays are special variables which can hold more than one value using the same name with an index.
- Declaring Arrays: type arrayName [ arraySize ];

```
/* simply define the arrays */
double balance[10];
float atom[1000];
int index[5];
```

• C array starts its index from 0

[0]	[1]	[2]	[3]	[4]
10	15	14	3	7

index[2] (3rd element of the array) has a value 14

• Initialize arrays with values

```
/* initialize the array with values*/
double atmass[4] = {12.0, 1.0, 1.0, 16.0};
double atmass[] = {12.0, 1.0, 1.0, 16.0};
atmass[0] = 12.0
```

Access array values via index

```
/* access the array values*/
int current_index = index[i];
double current_value=value[current_cell_index];
```

# Array Example

```
#include <stdio.h>
int main ()
{
    int n[ 10 ]; /* n is an array of 10 integers */
    int i, j;

/* initialize elements of array n to 0 */
    for ( i = 0; i < 10; i++)
    {
        n[ i ] = i + 100; /* set element at location i to i + 100 */
    }

/* output each array element's value */
    for (j = 0; j < 10; j++)
    {
        printf("Element[%d] = %d\n", j, n[j] );
    }

return 0;</pre>
```

### Accessing C arrays

- C arrays are a sequence of elements with contiguous addresses.
- There is no bounds checking in C.
- Be careful when accessing your arrays
- Compiler will not give you error, you will have \*undefined\* runtime behavior:

```
#include <stdio.h>
int main() {
  int index[5]={5, 4, 6, 3, 1};
  int a=3;
  /* undefined behavior */
  printf("%d\n",index[5]);
```

# Multidimensional Arrays

• General form of multidimensional array

```
type name[size1][size2]...[sizeN];
```

• Declaring 2D and 3D arrays:

```
float array2d[4][5];
double array3d[2][3][4];
```

Initializing multidimensional arrays

	col0	col1	col2	col3
row0	a[0][0]=0	a[0][1]=1	a[0][2]=2	a[0][3]=3
row1	a[1][0]=4	a[1][1]=5	a[1][2]=6	a[1][3]=7
row2	a[2][0]=8	a[2][1]=9	a[2][2]=10	a[2][3]=11

• C arrays are **row major** order i.e. in memory, the C array appears as

a[0][0]	a[0][1]	a[0][2]	a[0][3]	a[1][0]	a[1][1]	 a[1][3]	a[2][0]	 a[2][3]

### Example: Arrays

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
int main () {
 /* Program to calculate the sum, min and max of an integer array */
  int i. sum. min. max. n=11 :
  int a[] = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\};
  sum = max = 0.0 : min = 10.0 :
  /* Initialize array */
  for (i = 0 : i < n : i++) {
   sum += a[i] :
   if (a[i] > max ) max = a[i];
   if (a[i] < min ) min = a[i];</pre>
  printf("The max value is: %d\n", max);
  printf("The min value is: %d\n", min);
  printf("The sum value is: %d\n", sum);
  return 0:
/* define string */
char str[7]={'H', 'E', 'L', 'L, 'O', '!', '\0'};
```

# Strings in C I

 Strings in C are a special type of array: array of characters terminated by a null character '\0'.

```
/* define string */
char str[7]={'H', 'E', 'L', 'L', 'O', '!', '\0'};
char strl="HELLO!";
```

• Memory presentation of above defined string in C/C++:

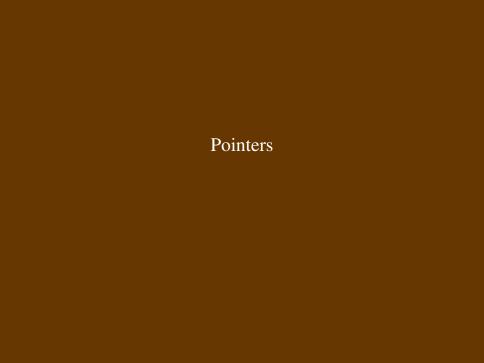
str[]	[0]	[1]	[2]	[3]	[4]	[5]	[6]
	'H'	'E'	'L'	'L'	'O'	'!'	'\0'

• C uses built-in functions to manipulate strings:

```
/* C sample string functions */
strcpy(sl, s2); /* Copies string s2 into string s1.*/
strcat(s1, s2); /* Concatenates string s2 onto the end of string s1. */
strlen(s1); /* Returns the length of string s1. */
strcmp(s1, s2); /* Returns 0 if s1 and s2 are the same; less than 0 if
s1/s2; greater than 0 if s1/s21. */
```

# Strings in C II

```
#include <stdio.h>
#include <string.h>
int main ()
  char str1[12] = "Hello";
  char str2[12] = "World";
  char str3[12];
  int len;
  strcpy(str3, str1);
  printf("strcpy( str3, str1) : %s\n", str3 );
  strcat( strl, str2);
  printf("strcat( strl, str2): %s\n", strl );
  len = strlen(strl):
  printf("strlen(strl) : %d\n", len );
  return 0:
```



#### **Pointers**

- Pointers are a very important part of the C programming language.
- They are used in many ways, such as:
  - Array operations (e.g., while parsing strings)
  - Dynamic memory allocation
  - Sending function arguments by reference
  - Generic access to several similar variables
  - Malloc data structures of all kinds, especially trees and linked lists
  - Efficient, by-reference "copies" of arrays and structures, especially as function parameters
- Necessary to understand memory and address · · · and the C programming language.

#### What is a Pointer

- A pointer is essentially a **variable** whose value is the address of another variable.
- Since it is a variable, it must be declared before use.
- Pointer "points" to a specific part of the memory.
- How to define pointers?

```
/* type: pointer's base type
var-name: name of the pointer variable.
asterisk *:designate a variable as a pointer */
type *pointer_var_name;
```

#### Examples

```
int *i_ptr; /* pointer to an integer */
double *d_ptr; /* pointer to a double */
float *f_ptr; /* pointer to a float */
char *ch_ptr; /* pointer to a character */
int **p_ptr; /* pointer to an integer pointer */
```

#### Pointer Rules

• There are two prefix unary operators to work with pointers.

```
& /*"address of" operator */
* /*"dereferencing" operator */
```

- Use ampersand "&" in front of a variable to access it's address, this can be stored in a pointer variable.
- Use asterisk "\*" in front of a pointer you will access the value at the memory address pointed to (**dereference** the pointer).
- Example

#### int a = 8; int \*p; /\* point p to a \*/ p = &a; /\* dereference pointer p \*/ \*p = 10;

#### Part of symbol table

var_name	var_address	var_value
a	bff5a400	8
p	bff5a3f6	bff5a400

# Pointer to variables and dereference pointers

```
/* pointer_rules.c */
#include <stdio.h>
int main() {
  int a = 6, b = 10;
  int *p;
  printf("\nInitial values:\n\tthe value of a is %d, value of b is %d\n", a, b);
  printf("the address of a is: %p, address of b is: %p\n", &a, &b);
  p = &a; /* point p to a */
  printf("\nafter \"p = &a\":\n");
  printf("\tthe value of p is %p, value at that address is %d\n", p, *p);
  p = &b; /* point p to b */
  printf("\nafter \"p = &b\":\n");
  printf("\tthe value of p is %p, value at that address is %d\n", p, *p);
  /* dereference pointer p */
  *p = 6, p = &a, *p = 10;
  printf("\nafter dereferencing the pointer:\n");
  printf("\tthe value of a is %d, value of b is %d\n", a, b);
  return 0;
```

### Never dereference an uninitialized pointer!

- In order to dereference the pointer, pointer must have a valid value (address).
- What is the problem for the following code?

```
int *ptr;
*ptr = 3;
```

- Again, you will have \*\*undefined behavior\*\* at runtime, you are operating on unknown memory space.
- Typically error: "Segmentation fault", possible illegal memory operation
- Always initialize your variables before use!

var	_name
	ptr

var_address	var_value
0x22aac0	0xXXXX
0xXXXX	3

#### **NULL Pointer**

- Memory address 0 has special significance, if a pointer contains the null (zero) value, it is assumed to point to nothing, defined as NULL in C.
- Set the pointer to NULL if you do not have exact address to assign to your pointer.
- A pointer that is assigned NULL is called a null pointer.

```
/* set the pointer to NULL 0 */
int *ptr = NULL;
```

• Before using a pointer, ensure that it is not equal to NULL:

```
if (ptr != NULL) {
   /* make use of pointer1 */
   /* ... */
}
```

#### Pointers and Functions I

- In C, arguments are passed by value to functions: changes of the parameters in functions do \*\*not\*\* change the parameters in the calling functions.
- Take a look at the below example, what are the values of a and b after we called swap(a, b);

```
/* this is the main calling function */
int main() {
   int a = 2;
   int b = 3;
   printf("Before: a = %d and b = %d\n", a, b );
   swap( a, b );
   printf("After: a = %d and b = %d\n", a, b );
}
/* this is function, pass by value */
void swap(int pl, int p2) {
   int t;
   t = p2, p2 = pl, pl = t;
   printf("Swap: a (pl) = %d and b(p2) = %d\n", pl, p2 );
}
```

#### Pointers and Functions II

- The values of a and b do not change after calling swap(a,b)
- Pass by value means the called function's parameter will be a copy of the caller's passed argument. The value of the caller and called functions will be the same, but the identity (the variable) is different - caller and called function each has its own copy of parameters

```
/* this is function, pass by reference */
void swap_by_reference(int *pl, int *p2) {
  int t;

  t = *p2, *p2 = *p1, *p1 = t;
  printf("Swap: a (p1) = %d and b(p2) = %d\n", *p1, *p2);
}

/* call by-address function */
swap_by_reference( %a, %b);
```

- The most frequent use of pointers in C is for walking efficiently along arrays.
- Remember, array name is the first element address of the array (it is a constant)

### Pointers and Functions III

Recall 2D array structure: combination of 1D arrays

```
int a[2][2]={{1,2},{3,4}};
```

- The 2D array contains 2 1D arrays: array a[0] and array a[1]
- a[0] is the address of a[0][0], i.e:
  - a[0] ⇔ &a[0][0]
  - a[1] ⇔ &a[1][0]
- Array a is then actually an address array composed of a[0], a[1], i.e. a ⇔ &a[0]

# Walk through array with pointer

```
#include <stdio.h>
const int MAX = 3:
int main () {
  int a i[] = \{10, 20, 30\};
 double a f[] = \{0.5, 1.5, 2.5\};
  int i;
  int *i_ptr;
 double *f_ptr;
 /* let us have array address in pointer */
  i_ptr = a_i;
  f ptr = a f;
 /* use the ++ operator to move to next location */
  for (i=0; i<MAX; i++,i_ptr++,f_ptr++ ) {</pre>
    printf("adr a_i[%d] = %8p\t", i, i_ptr );
   printf("adr a_f[%d] = %8p\n", i, f_ptr );
   printf("val a_i[%d] = %8d\t", i, *i_ptr );
    printf("val a_f[%d] = %8.2f\n", i, *f_ptr);
 return 0:
```

### Dynamic memory allocation using pointers

- For situations that the size of an array is unknown, we must use pointers to dynamically manage storage space.
- C provides several functions for memory allocation and management.
- Include <stdlib.h> header file to use these functions.
- Function prototype:

```
/* This function allocates a block of num bytes of memory and
    return
a pointer to the beginning of the block. */
void *malloc(int num);
/* This function release a block of memory block specified by
address. */
void free(void *address);
```

# Example of 1D dynamic array

```
#include <stdio.h>
#include <stdlib.h>
int main (void) {
  int n;
 int* i_array; /* define the integer pointer */
 int j;
 printf("Input the number of elements in the array:\n");
 scanf("%d", &n);
  i_array = (int*)malloc(n*sizeof(int));
  for (j=0; j<n; j++) {</pre>
   i_array[j]=j; /* use the pointer to walk along the array */
   printf("%d ",i_arrav[i]);
  printf("\n");
  free((void*)i array); /* free memory after use*/
  return 0:
```

