



# Introduction to Microprocessors

---

## Introduction to C Programming



# Introduction to C Programming

---

- A C program consists of functions and variables.
  - A function contains statements that specify the operations to be performed.
    - Types of statements:
      - Declaration
      - Assignment
      - Function call
      - Control
      - Null
    - A variable stores a value to be used during the computation.
  - The **main ()** function is required in every C program.



# Data types

---

- Data types
  - C has five basic data types: *void*, *char*, *int*, *float*, and *double*.
  - The *void* type represents nothing and is mainly used with function.
  - A variable of type *char* can hold a single byte of data.
  - A variable of type *int* is an integer that is the natural size for a particular machine.
  - The type *float* refers to a 32-bit single-precision floating-point number.
  - The type *double* refers to a 64-bit double-precision floating-point number.
  - Qualifiers *short* and *long* can be applied to integers. For ICC12 compiler, *short* is 16 bits and *long* is 32 bits.
  - Qualifiers *signed* and *unsigned* may be applied to data types *char* and *integer*.



# Declarations

---

- Declarations
  - A declaration specifies a *type*, and contains a list of one or more variables of that type.
- Examples of declarations
  - `int i, j, k;`
  - `char cx, cy;`
  - `int m = 0;`
  - `char echo = 'y'; /* the ASCII code of letter y is assigned to variable echo. */`



# Constants

---

- Constants
  - Types of constants: integers, characters, floating-point numbers, and strings.
  - A character constant is written as one character within single quotes, such as 'x'.
  - A character constant is represented by the ASCII code of the character.
  - A string constant is a sequence of zero or more characters surrounded by double quotes, as "MC9S12DP256 is made by Motorola" or "", which represented an empty string. Each individual character in the string is represented by its ASCII code.
  - An integer constant like 1234 is an *int*. A long constant is written with as 44332211.



# Arithmetic Operators

---

- + add and unary plus
- - subtract and unary minus
- \* multiply
- / divide -- truncate quotient to integer when both operands are integers.
- 
- % modulus (or remainder) -- cannot be applied to float or double
- ++ increment (by 1)
- -- decrement (by 1)



## Bitwise Operators

---

- C provides six operators for bit manipulations; they may only be applied to integral operands.
- `&` AND
- `|` OR
- `^` XOR
- `~` NOT
- `>>` right shift
- `<<` left shift
- `&` is often used to clear one or more bits of an integral variable to 0.
- `PTH = PTH & 0xBD` → clears bit 6 and bit 1 of PTH to 0
- `|` is often used to set one or more bits to 1.
- `PTB = PTB | 0x40;` /\* sets bit 6 to 1
- `^` can be used to toggle a bit.
- `abc = abc ^ 0xF0;` /\* toggles upper four bits



# Bitwise Operators

---

- `>>` can be used to shift the involved operand to the right for the specified number of places.
- `xyz = xyz >> 3` → shift right 3 places
- `<<` can be used to shift the involved operand to the left for the specified number of places.
- `xyz = xyz << 4` → shift left 4 places
- The assignment operator `=` is often combined with the operator. For example,
- `PTH &= 0xBD;`
- `PTB |= 0x40;`
- `xyz >>= 3;`
- `xyz <<= 4;`



# Relational and Logical operators

---

- Relational operators are used in expressions to compare the values of two operands.
  - The value of the expression is 1 when the result of comparison is true. Otherwise, the value of the expression is 0.
- Relational and logical operators:

==	equal to
!=	not equal to
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
&&	and
	or
!	not (one's complement)



# Control Flow

---

- Control flow
  - The control-flow statements specify the order in which computations are performed.
  - Semicolon is a statement terminator.
  - Braces { and } are used to group declarations and statements together into a *compound statement*, or *block*.



## If-Else

---

- if (*expression*)
- *statement1*
- else                   -- The **else** part is optional.
- *statement2*
- **Example,**
- if (a != 0)
- r = b;
- else
- r = c;
- A more concise way for this statement is
- r = (a != 0)? b : c;



# Multiway Conditional Statement

---

- if (expression1)
- statement1
- else if (expression2)
- statement2
- else if (expression3)
- statement3
- ...
- else
- statementn
- Example:
- if (abc > 0) return 5;
- else if (abc = 0) return 0;
- else return -5;



# Switch Statement

---

- `switch (expression) {`
- `case const_expr1:`  
      `statement1;`  
      `break;`
- `case const_expr2:`  
      `statement2;`  
      `break;`
- `...`
- `Default:`  
      `statementn;`  
      `break;`
- `}`



# Switch Statement

---

- `switch (i) {`
- `case 1: printf("*");`  
      `break;`
- `case 2: printf("**");`  
      `break;`
- `case 3: printf("***");`  
      `break;`
- `case 4: printf("****");`  
      `break;`
- `default:`  
      `break;`



## For loop

---

- `for (expr1; expr2; expr3)`
- `statement;`
- where, *expr1* and *expr2* are assignments or function calls and *expr3* is a relational expression.
- Example
- `sum = 0;`
- `for (i = 1; i < 10; i++)`
- `sum += i * i;`
- `for (i = 1; i < 20; i++)`
- `if (i % 2) printf(“%d “, i);`



# While Statement

---

- while (*expression*)
  - statement
  - Example
  - `int_cnt = 5;`
  - `while (int_cnt); /* do nothing while the variable int_cnt  $\neq$  0 */`
- Do-While Statement
  - do
    - statement
  - while (*expression*)
  - Example
  - `Int digit = 9;`
  - do
    - `printf(“%d “, digit--);`
  - while (`digit >= 1`);



# Functions and Program Structures

---

- Every C program consists of one or more functions
- Definition of a function cannot be embedded within another function
- A function can be called from several different places within a program
- A function will process information passed to it from the calling portion of the program, and return a single value
- Information is passed to the function via special identifiers called arguments and returned via return statement
- Some functions, however, accept information but do not return anything



# Functions and Program Structures

---

- Syntax of a function definition:  
return\_type function\_name (declarations of arguments)  
{  
    declarations and statements  
}

Example

```
char lower2upper (char cx)
{
    if (cx > 'a' && cx <= 'z') return (cx - ('a' - 'A'));
    else return cx;
}
```



# Functions and Program Structures

---

- A function cannot be called before it has been defined
- The dilemma is solved by using the function prototype statement
- Syntax for the function prototype statement:
- Return\_type function\_name (declaration of arguments)
- Ex:
- `Char test_prime(int a);`
- Declaring the function prototype before main solves the problem



# Pointers and Addresses

---

- Ability to work with memory addresses is an important feature of the C language
- Feature allows programmer the freedom to perform operations similar to assembly language
- Array elements can be more efficiently reached through pointers than by sub-scripting
- Addresses that can be stored and changed are called pointers
- Pointer is just a variable that contains an address
- Pointers can be used to reach objects in memory



# Pointers and Addresses

---

- Pointer is a variable that holds the address of the variable
- Pointers can be used to pass information back and forth between a function and its reference point
- Pointers provide a way to return multiple data items from a function via function arguments
- Pointers also permit references to other functions to be specified as arguments to a given function
- Pointers are also closely associated with arrays and therefore provide an alternative way to access individual array elements



# Pointers and Addresses

---

- Pointer Declaration:
- Datatype \*variable(s)
- Ex:
- Int \*ip
- Ip is a pointer which can hold the starting address of an integer variable
- Unsigned short \*pt3
- Declares pt3 as a pointer to the unsigned 16 bit integer
- Long \*pt4
- Declares pt4 as a pointer to the 32 bit data



# Pointers and Addresses

---

- Pointers are used to get start address of a variable
- Ampersand (&) before variable name generates its starting address
- `Int x=5,y;`
- `&x` → provides the address of integer x
- `&y` → provides the address of integer y
- Address of the variable cannot be stored in other variables
- Address of variables can be stored only in pointer variables
- Pointer variables can only hold addresses of other variables



# Pointers and Addresses

---

- Ex:
- `int x, *ip`
- `x` is an integer variable
- `ip` is a pointer variable that can hold the start address of an integer variable
- `ip = &x`
- `ip` contains the start address of integer variable `x`
- Once address is stored in the pointer variable, it points to that particular memory address



# Pointers and Addresses

---

- Ex: To access the contents of variables through its address
- Main()
- {
- `Int x = 5,y,*ip;`
- `Ip = &x;`
- `Y = *ip`
- }
- Above example access the contents of memory location pointed by pointer variable ip and stores in variable y



# Pointers and Variables

---

- \* before pointer variable is called as indirection operator
- Indirection operator can be placed only before a pointer variable
- Indirection operator before a pointer variable indicates the contents of the variable pointed by the pointer variable
- $ip \rightarrow$  contents of  $ip \rightarrow$  Starting address of a variable
- $\&ip \rightarrow$  starting address of  $ip$
- $*ip \rightarrow$  contents of the variable pointed by  $ip$
- Ex:
- ```
Main() {
```
- ```
  Int x,y,z,*ip1,*ip2,*ip3;
```
- ```
  ip1 = &x; ip2 = &y; ip3 = &z;
```
- ```
  *ip1 = 6;  $\rightarrow$  x = 6
```
- ```
}
```



# Pointers and Variables

---

- Indirection operator can be used on both sides of the assignment operator
- Ex:
- Assume the example in the previous slide
- `*ip1 = 6;`
- `*ip2 = 4;`
- `*ip3 = *ip1 * (*ip2);`
- Contents of the variable pointed by ip1 is multiplied with contents of the variable pointed by ip2 and the result is assigned to the variable pointed by ip3



# Pointers and Addresses

---

- Passing by reference:
- Passing addresses of the variables from the calling function to the called function is called as passing by reference
- Ex:
- `Void swap(int *, int *);`
- `Main(){`
- `Int x= 5,y = 10;`
- `Swap(&x,&y);`
- `}`
- `Void swap(*ip1, *ip2)`
- `{ int temp;`
- `Temp = *ip1; *ip1 = *ip2; *ip2 = temp;}`



# Pointers and Arrays

---

- Any operation that can be achieved through array sub-scripting can also be done with pointers
- Pointer version will in general be faster but somewhat harder to understand
- Ex:
- `Int ax[20];` → defines an integer array of 20 elements
- `Int *ip` → declares ip as a pointer to integer variable
- `ip = &ax[0]` or `ip = ax`
- Makes ip contain the address of `ax[0]`
- In general name of the array itself is its starting address
- `ip+1` points to `ax[1]` and `ip+i` points to `ax[i]`



## Pointers and Arrays

---

- An array can be used as an argument to a function thus permitting the entire array to be passed to the function
- To pass an array to a function the array name should appear by itself without brackets or subscripts, as an actual argument within the function call
- When declaring a one-dimensional array as a formal argument, the array name is written with a pair of empty square brackets
- Size of the array is not specified within the formal argument
- If the array is two-dimensional, then there should be two pairs of empty brackets following the array name



# Pointers and Arrays

---

- Ex:
- `Int average(int n, int arr[ ]);`
- `Main() {`
- `Int n, avg;`
- `Int arr[50];`
- `Avg = average(n,arr);`
- `}`
- `Int average(int k, int *brr)`
- `{`
- `.....`
- `}`



# Structures

---

- A structure is a group of related variables that can be accessed through a common name
- Each item within a structure has its own data type, which can be different from other data items
- Ex:
- `Struct struct_name { //Name is Optional//`
- `type1 member1;`
- `type2 member2;`
- `-----`
- `-----`
- `};`



# Structures

---

- The `struct_name` is optional and if it exists, defines a structure tag
- A struct declaration defines a type
- The right brace that terminates the list of the members may be followed by a list of variables
- Ex:
- `Struct catalog_tag{`
- `Char author[40];`
- `Char title[40];`
- `}card;`
- Variable `card` is of type `catalog_tag`



# Structures

---

- A structure definition that is not followed by a list of variables reserves no storage
- If the declaration is tagged, the tag can be used later in declarations of the instances of the structure
- Ex:
- Struct point {
- Int x;
- Int y;
- };
- Struct point pt; → defines a variable pt of type point
- Sq\_distance = pt.x\*pt.x + pt.y\*pt.y;