# IN3200/IN4200: C Programming Tutorial

A drastically simplified version of `https://www.tutorialspoint.com/cprogramming/`

# C programs

- A C program is made up of
  - Preprocessor commands
  - Variables
  - Statements and expressions
  - Functions
  - Comments
- A C program can be as simple as having only 3 lines, or as comprehensive as being composed of millions of lines
- A C program can be stored in one file with name extension `.c` (or spread over many `.h` and `.c` files)
- Use of libraries—groups of already-coded functions and declarations—actually happens all the time

# Hello-World example

```
#include <stdio.h>

int main() {
   /* my first program in C */
   printf("Hello, World! \n");

   return 0;
}
```

# Hello World example explained

- `#include <stdio.h>` is a preprocessor command, which tells a C compiler to include the header file `stdio.h`
- `int main()` is the main function where the program execution begins
- `/*...*/` is a comment
- `printf(...)` is a standard library function available in C
- `return 0;` terminates the `main()` function and returns the value 0

## Demo of compilation and execution

# Identifiers

- A C identifier is a name used to identify a variable, function, or any other user-defined item
- Examples of acceptable identifiers

```
mohd        zara     abc    move_name   a_123
myname50    _temp    j      a23b9       retVal
```

- C is a case-sensitive programming language

# Keywords – reserved words

| | | | |
|---|---|---|---|
| auto | else | long | switch |
| break | enum | register | typedef |
| case | extern | return | union |
| char | float | short | unsigned |
| const | for | signed | void |
| continue | goto | sizeof | volatile |
| default | if | static | while |
| do | int | struct | _Packed |
| double | | | |

# C data types

- **Basic Types**
  They are arithmetic types and are further classified into: (a) integer types and (b) floating-point types
- **Derived types**
  They include (a) Pointer types, (b) Array types, (c) Structure types, (d) Union types and (e) Function types

# Integer types

| Type | Storage size | Value range |
|---|---|---|
| `char` | 1 byte | -128 to 127 or 0 to 255 |
| `unsigned char` | 1 byte | 0 to 255 |
| `signed char` | 1 byte | -128 to 127 |
| `int` | 2 or 4 bytes | -32,768 to 32,767 or -2,147,483,648 to 2,147,483,647 |
| `unsigned int` | 2 or 4 bytes | 0 to 65,535 or 0 to 4,294,967,295 |
| `short` | 2 bytes | -32,768 to 32,767 |
| `unsigned short` | 2 bytes | 0 to 65,535 |
| `long` | 4 bytes | -2,147,483,648 to 2,147,483,647 |
| `unsigned long` | 4 bytes | 0 to 4,294,967,295 |

# The `sizeof` operator

To get the exact size of a type or a variable on a particular platform, you can use the `sizeof` operator. The expression `sizeof(type)` yields the storage size of the object or type in bytes.

```
#include <stdio.h>

int main() {
   printf("Storage size for int : %d \n", sizeof(int));

   return 0;
}
```

# Floating-point types

| Type | Storage size | Value range | Precision |
|---|---|---|---|
| `float` | 4 bytes | 1.2E-38 to 3.4E+38 | 6 decimal places |
| `double` | 8 bytes | 2.2E-308 to 1.8E+308 | 15 decimal places |
| `long double` | 16 bytes | 3.4E-4932 to 1.2E+4932 | 18 decimal places |

Note: The actual values can be machine-dependent!

# Header file `float.h`

The header file `float.h` defines macros that allow you to use these values and other details about the binary representation of real numbers in your programs

```
#include <stdio.h>
#include <float.h>

int main() {
   printf("Storage size for float : %lu \n", sizeof(float));
   printf("Minimum float positive value: %E\n", FLT_MIN );
   printf("Maximum float positive value: %E\n", FLT_MAX );
   printf("Precision value: %d\n", FLT_DIG );

   return 0;
}
```

# C variables

A variable is a name given to a storage area that a C program can manipulate. Each variable in C has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

The name of a variable can be composed of letters, digits, and the underscore character. It must begin with either a letter or an underscore. Upper and lowercase letters are distinct because C is case-sensitive.

```
int    i, j, k;
char   c, ch;
float  f, salary;
double d;
```

# C operators

An operator is a symbol that tells the compiler to perform specific mathematical or logical functions. C language is rich in built-in operators:

- Arithmetic operators `+` `-` `*` `/` `%` `++` `-`
- Relational operators `==` `!=` `>` `<` `>=` `<=`
- Logical operators `&&` `||` `!`
- Bitwise operators
- Assignment operators

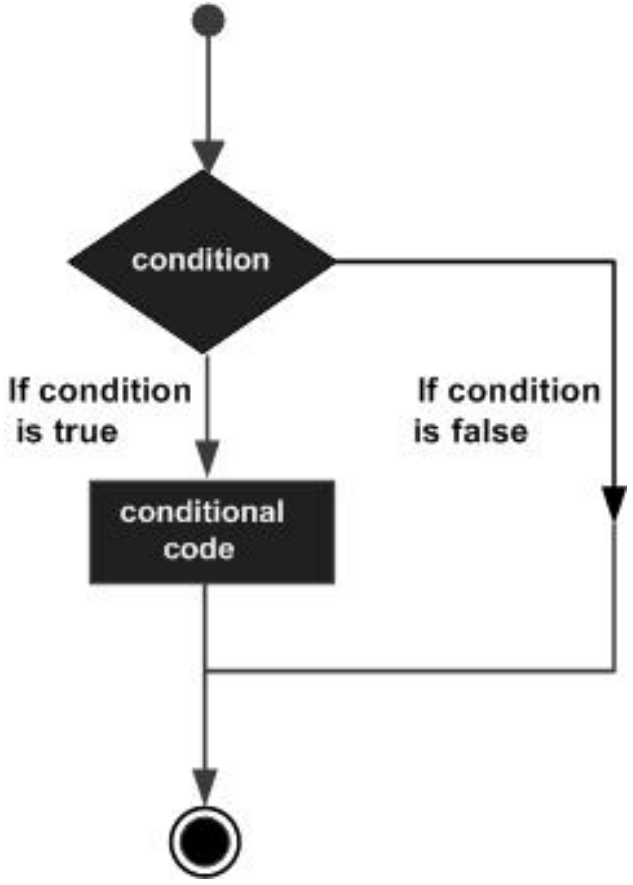
# Bitwise operators

A bitwise operator works on bits and performs bit-by-bit operation

| **p** | **q** | **p&q** | **p\|q** | **p^q** |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 |

# Decision making

Decision making structures require that the programmer specifies one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.



# Loops

To execute a statement or a group of statements multiple times:

- `for`
- `while`
- `do ... while`

# Functions

A function is a group of statements that together perform a task. Every C program has at least one function, which is `main()`, and you can define additional functions.

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.

The general form of a function definition in C programming language:

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

# Function arguments

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the **formal parameters** of the function.

Formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

While calling a function, the formal parameters get the values of the `actual parameters`.

# One example

```
#include<stdio.h>
void func_1(int);

int main()
{
    int x = 10;

    printf("Before function call\n");
    printf("x = %d\n", x);

    func_1(x);

    printf("After function call\n");
    printf("x = %d\n", x);

    return 0;
}

void func_1(int a)
{
    a += 1;
    a++;
    printf("\na = %d\n\n", a);
}
```

# Scope rules

A scope in any programming is a region of the program where a defined variable can have its existence and beyond that variable it cannot be accessed. There are three places where variables can be declared in C programming language:

- Inside a function or a block which is called **local** variables.
- Outside of all functions which is called **global** variables.
- In the definition of function parameters which are called **formal** parameters.

# Global variables

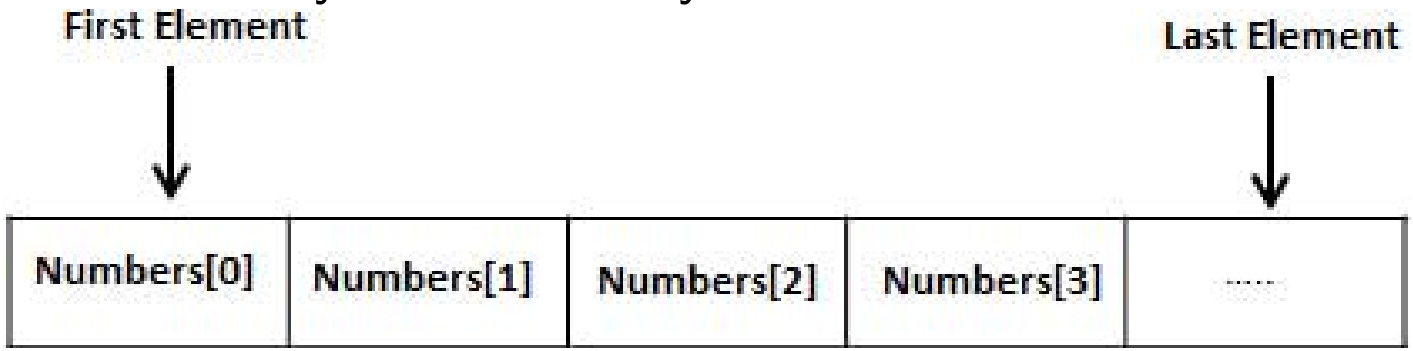
Global variables hold their values throughout the lifetime of your program and they can be accessed inside any of the functions defined for the program.

```
#include <stdio.h>

/* global variable declaration */
int g;

int main () {

  /* local variable declaration */
  int a, b;

  /* actual initialization */
  a = 10;
  b = 20;
  g = a + b;

  printf ("value of a = %d, b = %d and g = %d\n", a, b, g);

  return 0;
}
```

# Arrays

An array is a kind of data structure that can store a sequential collection of elements of the same type.

Instead of declaring individual variables, such as `Number0`, `Number1`, ..., and `Number99`, you can declare one array variable, named such as `Numbers`, and use `Numbers[0]`, `Numbers[1]`, ..., and `Numbers[99]` to represent individual variables. A specific element in an array is accessed by an index.



# Fix-sized arrays

A programmer can specify the type of the elements and the number of elements required by an array

```
type arrayName [ arraySize ];
```

`arraySize` must be an integer constant greater than zero.

# Address in memory

Every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator, which denotes an address in memory.

```
#include <stdio.h>

int main () {

   int  var1;
   char var2[10];

   printf("Address of var1 variable: %x\n", &var1  );
   printf("Address of var2 variable: %x\n", &var2  );

   return 0;
}
```

# Pointers

A pointer is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before using it to store any variable address.

```
int    *ip;    /* pointer to an integer */
double *dp;    /* pointer to a double */
float  *fp;    /* pointer to a float */
char   *ch     /* pointer to a character */
```

# How to use pointers?

- Define a pointer variable
- Assign the address of a variable to a pointer
- Access the value at the address stored in the pointer variable (via operator *)

```
#include <stdio.h>

int main () {
   int  var = 20;   /* actual variable declaration */
   int  *ip;        /* pointer variable declaration */

   ip = &var;  /* store address of var in pointer variable*/

   printf("Address of var variable: %x\n", &var  );

   /* address stored in pointer variable */
   printf("Address stored in ip variable: %x\n", ip );

   /* access the value using the pointer */
   printf("Value of *ip variable: %d\n", *ip );

   return 0;
}
```

# More pointer concepts

- **Pointer arithmetic**: Four arithmetic operators can be used on pointers: ++, –, +, -
- **Array of pointers**: You can define arrays to hold a number of pointers.
- **Pointer to pointer**: C allows you to have pointer on a pointer and so on.
- **Passing pointers to functions in C**: Passing an argument by address allows the passed argument to be changed.
- **Return pointer from functions in C**: C allows a function to return a pointer to the local variable, static variable, and dynamically allocated memory as well. (**Be very careful with such usage!!!!!!**)

# An example of function returning a pointer

```
#include <stdio.h>

int *getMax(int *m, int *n) {
  /* if the value pointed by pointer m is greater than n
   * then, return the address stored in the pointer variable m */
  if (*m > *n) {
    return m;
  }
  else {
    return n;
  }
}

int main(void) {
  // integer variables
  int x = 100;
  int y = 200;

  // pointer variable
  int *max = NULL;

  /* get the variable address that holds the greater value
   * for this we are passing the address of x and y
   * to the function getMax() */
  max = getMax(&x, &y);

  // print the greater value
  printf("Max value: %d\n", *max);

  return 0;
}
```

# C structures

**structure** is a user defined data type available in C that allows to combine data items of different kinds.

To define a structure, you must use the `struct` statement:

```
struct Books {
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
} book;
```

# Dynamic memory management

The C programming language provides several functions for memory allocation and management. These functions can be found in the `<stdlib.h>` header file.

- `void *calloc(int num, int size);` – allocates an array of num elements each of which size in bytes will be size.
- `void free(void *address);` – releases a block of memory block specified by address.
- `void *malloc(int num);` – allocates an array of num bytes and leave them uninitialized.
- `void *realloc(void *address, int newsize);` – re-allocates memory extending it upto newsize.

# Example of dynamic memory allocation

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int n, i, *ptr, sum = 0;

    printf("Enter number of elements: ");
    scanf("%d", &n);

    ptr = (int*) malloc(n * sizeof(int));
    if(ptr == NULL) {
        printf("Error! memory not allocated."); exit(-1);
    }

    printf("Enter elements: ");
    for (i = 0; i < n; ++i) {
        scanf("%d", &(ptr[i]));
        sum += ptr[i];
    }

    printf("Sum = %d\n", sum);
    free(ptr);
    return 0;
}
```

# Command-line arguments

Input arguments to the `main` function:

```
#include <stdio.h>

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

   if( argc == 2 ) {
      printf("The argument supplied is %s\n", argv[1]);
   }
   else if( argc > 2 ) {
      printf("Too many arguments supplied.\n");
   }
   else {
      printf("One argument expected.\n");
   }
}
```

# Simple I/O

A file represents a sequence of bytes, regardless of it being a text file or a binary file. C programming language provides both high-level functions and low-level function calls to handle files.

- Opening a file

```
FILE *fopen(const char *filename, const char *mode);
```

- Closing a file

```
int fclose(FILE *fp);
```

- Writing to a file (many different functions available)
- Reading to a file (many different functions available)
- Binary I/O functions

```
size_t fread(void *ptr, size_t size_of_elements,
             size_t number_of_elements, FILE *a_file);
size_t fwrite(const void *ptr, size_t size_of_elements,
             size_t number_of_elements, FILE *a_file);
```