Advanced-C

By P. Sailesh Kumar

Basic usage of Struct

```
Struct Sensor
{
  int id;
  float val;
};
main()
{
  Struct Sensor temp = {50, 66.6};
  Pf("The Sensor id is %d & Sensor reading is %.f\n",
                    temp.id, temp.val);
```

Same thing we can implement with typedef

```
typedef Struct Sensor
{
  int id;
  float val;
} Sensor;
main()
{
  Sensor temp = {25, 36.44}
  Pf
```

Array of Structs

Suppose 3 Sensor

```
typedef Struct Sensor
{
  char name[10];
  int id;
  float val;
} Sensor;

main()
{
  Sensor temp[3] =
  {{"lm35", 25, 66.2},
   {"lm65", 35, 88.2},
   {"lm85", 45, 99.2}};
```

```
for(int i = 0; i < 3; i++)
{
    printf("%s %d %.f", temp[i].name,
                        temp[i].id,
                        temp[i].val);
}
```

Nesting of Structures ⟹ Structure inside a Structure

```
typedef struct date
{ int id, int month,
  int year} Date
```

```
typedef struct sensor
{
  int id;
  float val;
  Date timestamp;
} Sensor;
```

Declare a variable named timestamp whose type is Date

```
main()
{
  Sensor packet = {20, 44.5, {1, 9, 2025}};

  printf("Sensor id is %d & Sensor val is %f at today's date
          is %d/%d/%d", packet.id, packet.val,
                        packet.timestamp.day,
                        packet.timestamp.month,
                        packet.timestamp.year);
```

X

Structure pointer

```
typedef Struct
{
  char name[50];
    int age;
   float marks
};Student

main()
{
  Student s1 = {"Sailesh", "21", 88.8};

    Student *p = &s1;

  Pf("%s %d %f", p→name, p→age, p→marks);

        p→marks = 90.2

  Pf("After modifying my updated marks
                    are %d", p→marks)
};
```

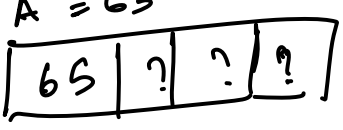
Union Example

```
Union Data
{
  int num;
  char ch;
};
main()
{
  Union data D;
    d.num = 100;
      Pf("%d", d.num);

    d.ch = 'A';
    Pf("%c", ch);

  Now again try to overwrite

  Pf("New num is %d", d.num);
```

---

100 le byte

A = 65



So d.num will give me 65 (first byte)

Structure Inside a Union:

* A MC receive Sensor data from different Sensors
* At a given time only one Sensor data is captured (Eg Either Temp & Accelerometer)
* So if we use union we can have a common memory for different Sensor

Advantage Memory optimised

Eg

```
struct TempSensor
{
  int tmp;
};

struct AccelSensor
{
  int x, y, z;
};

Union SensorData
{
  struct TempSensor temp;
  struct AccelSensor accel;
};

main()
{
  Union SensorData Sd;
  Sd.temp.tmp = 25;
  Pf("%d", Sd.temp.tmp);
  Sd.accel.x = 1;
  Sd.accel.y = 2;
  Sd.accel.z = 3;
  Pf("%d", Sd.accel.x);
```

Only Unions : one sensor at a time, one value at a time

```
Union SensorData
{
  int temp;
  float pressure;
  int humidity;
};

main()
{
  Union SensorData sd;
     sd.temp = 25;
    pf
   sd.pressure = 100.5;
     pf
    sd.humidity = 60;
       pf
}
```

temp sensor, only attribute temp

pressure sensor only attribute pressure

humidity sensor only attribute humidity

Nesting of unions:

I have STM32 & connected multiple sensors to it. It usually reads like temperature & humidity.

If sensor malfunction we have to report/see the error code.

So either we have to report sensor data / error code

```
Union inner
{
  int temp;
  int humidity;
};

Union outer
{
  int error code;
  Union inner Sensor data;
};
```

```
main()
{
  Union outer data;
  data.error code = 2;
  data.Sensordata.temp = 25;
  data.Sensor data.humidity = 80;
}
```

Union Inside a Struct

⇓ Used one at a time

⇓ Some variables needed everytime

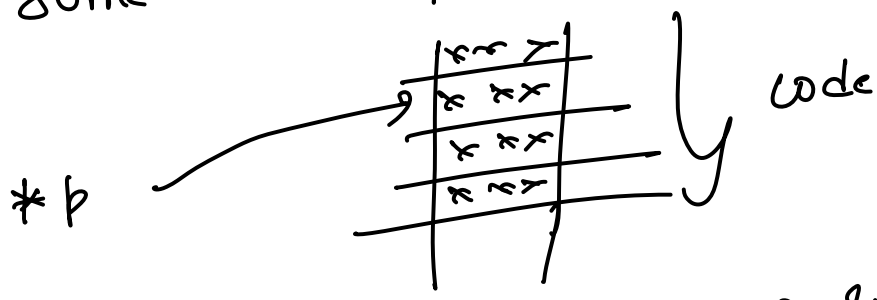
```
Struct Sensor_packet
{
  int id;
  int time_stamp;
  union
  {
    int tmp;
    int humidity;
    float pressure;
  } data;
};
```

} Alway Req

```
main()
{
  Struct Sensor_packet pt;
  pt.id = 10;
  pt.time stamp = 225;
  pt.data.tmp = 25;

  pt.id = 20;
  pt.time.stamp = 226;
  pt.data.pressure = 600;

  pt.id = 30;
  pt.ts = 227;
  pt.data.humidity = 55;
```

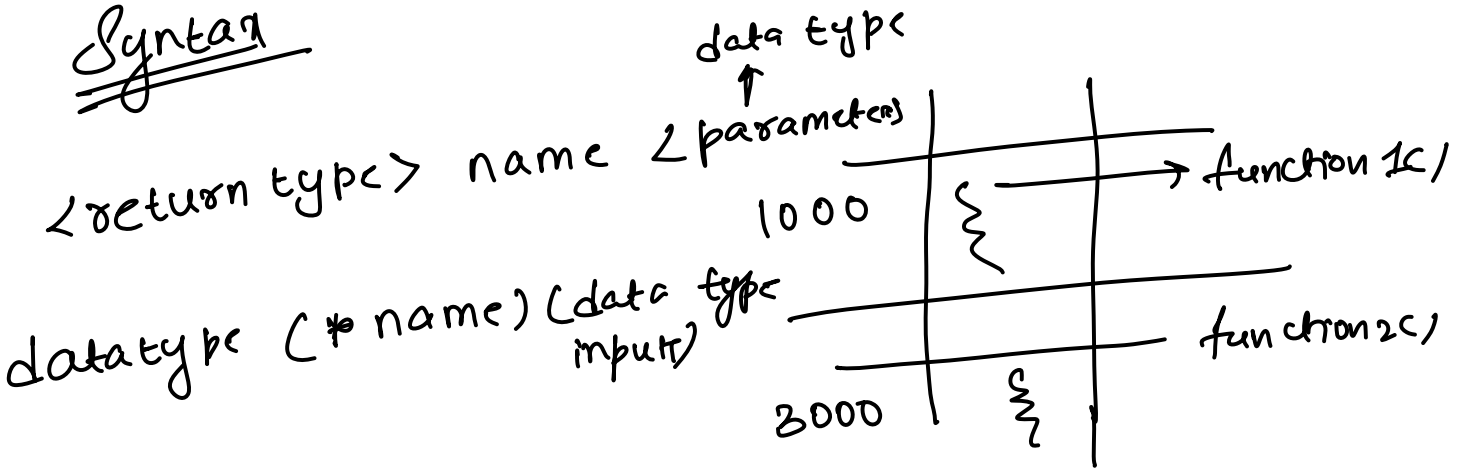
Function pointer

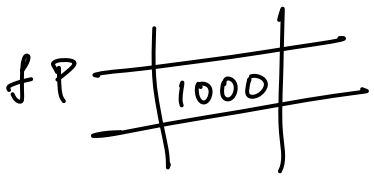
It is a pointer that points to memory region where some code is placed



Function pointer point to memory & content is assumed to be some code (instructions.

Syntax






```
void (*fp) ();
   fp = func1;   fp = & func-1
   fp();   or   (*fp) ();
   fp = func2;
   fp();
```

```
typedef void (*func_ptr)(int);
```

This becomes a new-type.

Write above main (not inside main)

```
func_ptr fp;
```

Array of function pointers

```
int (*fp[4])(int, int);

fp[0] = &add;
fp[1] = &sub;
fp[2] = &mul;
fp[3] = &div;

int x = 10;
int y = 5;

for(int i=0; i<4; i++)
{
    printf("%d", fp[i](x,y));
}
```

Advanced I/O

Character functions

getchar() → Read one char from keyboard

putchar() → write one char at a time

The reason we are not using printf & scanf is because they are formatted I/O

They must parse the %c, %d & sometimes deal with locale/float conversions

So it becomes very slow

getchar & putchar combination

```
int c;
ptf("enter char");
while ((c = getchar()) != '\n')
{
    putchar(c);
}
```

Files:

To store data mostly in permanent memory (Disks)

(Read/write operations)

Data logging.

can be used with combinations of getchar & putchar

A file is represented by the file pointer

```
FILE *fp;
```

File modes
- r → read
- w → write
- a → append

getc : Read one char from the file

(we can read from any file)

Eg open a file. Read content print to monitor, close the file

```
FILE *fp;
int c;
① fp = fopen("<name>", "<mode>");
  if(fp == NULL)
  {
    printf("Error");
    return 1;
  }
② while((c = getc(fp)) != EOF)   → Read content of file
  {
    putchar(c);   → print to o/p monitor
  }
③ fclose(fp);
```

---

call back functions

A function whose address is given to another function, so that other function can call it back whenever required.

Let's Understand this behaviour w.r.t with & without call back

No call back

```
void B()
{ printf("Hi from B");
}
void A()
{
   printf("Hi from A");
   B();
}
main()
{
  A();
}
```

In this code

A is always calling B

Suppose code is updated and if you want to A to call C, you have to modify the A's code to call C.

This is not flexible

So before moving to call back function's Let's understand function pointers (Because in callback function, we pass address of function as Arguments

```
void func1()
{ printf("Hi from func1");
}
void (*fp)();
fp = func1;
fp();
fp = func2;
fp();
```

```
void func2()
{ printf("Hi from func2");
}
```

O/P: Hi from func1
Hi from func2

call back function

```
typedef void (*callbackfunc)();

void B()
{
  Pf("hi from func B");
}
void A (callbackfunc callback)
{
  Pf("currently in func A");
  callback(); call back the function passed
              from main.
}
main()
{
  A(B); pass Address of function B to A
}
```

O/P:

I am in func A

hi from func B

Working

A(B); we are passing address of function B to A.

Note: we are not calling B here directly.

that would be A(B());

A(B); passing the function pointer (Address of B)

1) A receives parameter callback

2) callback points to B
[function pointer holding the address of B]

3) A will print I am in Func A

4) Now callback is pointing to B
when we call this, it is equivalent to calling B();
callback();

5) After B finishes, control goes to A

6) A finishes and control goes to main.

```c
#include <stdio.h>

typedef void (*CallbackFunc)();   // typedef for callback function

// -------- Application Functions --------
void TurnOnLED()
{
    printf("LED turned ON\n");
}

void SendUART()
{
    printf("UART message sent\n");
}

// -------- Driver Function (Button Handler) --------
void ButtonHandler(CallbackFunc callback)
{
    printf("Button pressed\n");
    callback();   // Call the user's function
}

// -------- main() --------
int main()
{
    // Case 1: Button should turn ON LED
    ButtonHandler(TurnOnLED);

    // Case 2: Button should send UART message
    ButtonHandler(SendUART);

    return 0;
}

```

Embedded Example for call back functions

```c
#include <stdio.h>

// Define GPIO struct with typedef
typedef struct {
    unsigned int PIN0 : 1;
    unsigned int PIN1 : 1;
    unsigned int PIN2 : 1;
    unsigned int PIN3 : 1;
    unsigned int RESERVED : 28;
} GPIO;

int main() {
    GPIO gpio = {0};  // All pins LOW initially

    gpio.PIN0 = 1;
    gpio.PIN2 = 1;

    printf("PIN0=%d, PIN1=%d, PIN2=%d, PIN3=%d\n",
           gpio.PIN0, gpio.PIN1, gpio.PIN2, gpio.PIN3);

    return 0;
}
```

All pins set to low

Set pin0 & pin 2 high.

```c
#include <stdio.h>

// UART Control Register (8-bit)
typedef struct {
    unsigned int TX_EN     : 1;
    unsigned int RX_EN     : 1;
    unsigned int PARITY_EN : 1;
    unsigned int BAUD      : 3;
    unsigned int RESERVED  : 2;
} UART_CTRL;

int main() {
    UART_CTRL uart = {0};

    uart.TX_EN = 1;
    uart.RX_EN = 1;
    uart.BAUD  = 3;

    printf("TX=%d RX=%d PARITY=%d BAUD=%d\n",
           uart.TX_EN, uart.RX_EN, uart.PARITY_EN, uart.BAUD);

    return 0;
}
```

Enable Tx & Rx

Set Baud rate

3 bits $\Rightarrow 2^3$ options

$\Rightarrow$ 8 options

```c
#include <stdio.h>

// Timer Control Register (16-bit)
typedef struct {
    unsigned int ENABLE    : 1;
    unsigned int INTERRUPT : 1;
    unsigned int MODE      : 2;
    unsigned int PRESCALER : 4;
    unsigned int RESERVED  : 8;
} TIMER_CTRL;

int main() {
    TIMER_CTRL timer = {0};

    timer.ENABLE    = 1;
    timer.INTERRUPT = 1;
    timer.MODE      = 2;
    timer.PRESCALER = 8;

    printf("ENABLE=%d, INT=%d, MODE=%d, PRESCALER=%d\n",
           timer.ENABLE, timer.INTERRUPT, timer.MODE,
timer.PRESCALER);

    return 0;
}
```

Enable Timer

Enable Interrupt

```c
#include <stdint.h>

// Define GPIO control register with typedef
typedef struct {
    unsigned int PIN0 : 1;
    unsigned int PIN1 : 1;
    unsigned int PIN2 : 1;
    unsigned int PIN3 : 1;
    unsigned int RESERVED : 28;
} GPIO_CTRL;

// Map struct directly to hardware register
#define GPIO_CTRL_REG   (*(volatile GPIO_CTRL *)0x40000000)

int main() {
    GPIO_CTRL_REG.PIN0 = 1;  // Set GPIO Pin 0 HIGH
    GPIO_CTRL_REG.PIN1 = 0;  // Set GPIO Pin 1 LOW
    GPIO_CTRL_REG.PIN2 = 1;  // Set GPIO Pin 2 HIGH

    return 0;
}
```

Point to base address of HW register

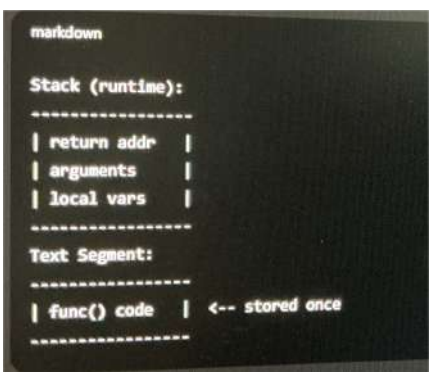
Inside H/W register Set the GPIO PIN0 & PIN-2

Inline functions

Use them whenever we want to call small functions.

Generally in normal functions

1) Push args into stack
2) Push return address to stack
3) Jump to function's memory address
4) Execute function code.
5) Pop return address & args from stack.
6) Return to the caller.



Inline function Example

2 Inline Function (No Stack Overhead)

```c
inline void func() {
    // do something
}

int main() {
    func();  // inlined
}
```

Steps at runtime:

1. Compiler copies the function body into `main()` at the call site.
2. No stack push/pop for arguments or return address (unless function has local variables).
3. CPU just executes instructions in-place.

Memory Implications:

- Text segment: instructions of the inline function copied at each call site
- Stack: no return address or arguments pushed for the call (only locals if any)

Diagram:

```
Stack (runtime):
----------------
| local vars if any |
----------------
Text Segment:
----------------
| main() code       |
| func() code copied|  <-- inlined here
| func() code copied|  <-- another call site
----------------
```

Pre-processor directives

* Basically it's an set of instructions to the compiler's pre-processor
* They run before the actual compilation starts
1) #define PI 3.14, text substitution happens

```
#include <stdio.h>

#define DEBUG   // try commenting this line

int main()
{
#ifdef DEBUG
    printf("Debug mode is ON\n");
#endif
    printf("Program running...\n");
    return 0;
}
```

if debug was defined we would get o/p as Debug mode ON

if debug was not defined we won't get debug mode ON msg

```c
#include <stdio.h>

#ifndef RELEASE
#define RELEASE 0
#endif

int main() {
    printf("Release mode = %d\n", RELEASE);
    return 0;
}
```

If release macro was not defined, go and define it.
and make it zero (0)

```c
#include <stdio.h>

// Uncomment to enable debug mode
#define DEBUG

// Define RELEASE only if it is not already defined
#ifndef RELEASE
#define RELEASE 0
#endif

int main() {

    // Check if DEBUG is defined
#ifdef DEBUG
    printf("Debug mode is ON\n");
#endif

    // Check if DEBUG is not defined
#ifndef DEBUG
    printf("Debug mode is OFF\n");
#endif

    // Check if RELEASE is defined
#ifdef RELEASE
    printf("Release mode is defined, value = %d\n", RELEASE);
#endif

    // Check if RELEASE is not defined
#ifndef RELEASE
    printf("Release mode is NOT defined\n");
#endif
```

debug define

Release 0

O/p: Debug ON

Releas is defined & value is zero

```
#define RELEASE 0
#define DEBUG   1

#if DEBUG == 1
    printf("Debug mode is ON\n");
#elif RELEASE == 1
    printf("Release mode is ON\n");
#else
    printf("Both Debug and Release are OFF\n");
#endif
```

O/P: Debug is ON

```
#include <stdio.h>

#define DEBUG 0
#define RELEASE 1

int main() {
#if DEBUG
    printf("Debug mode is ON\n");
#elif RELEASE
    printf("Release mode is ON\n");
#else
    printf("Both Debug and Release are OFF\n");
#endif
    return 0;
}
```

O/P:

Release mode is ON