Types of Functions

By: Yehia M. Abu Eita

Outlines

- Recursive functions
- Inline functions
- Callback functions
- Reentrant and Non-Reentrant functions
- Synchronous and Asynchronous functions

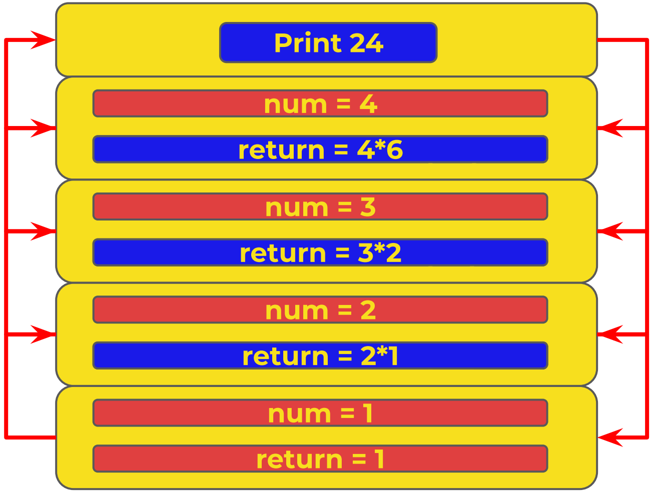
Recursive functions

- A function is said to be recursive if it calls itself directly or indirectly.

```c
#include <stdio.h>
int factorial(int num)
{
        if(num == 1)
        {
                return num;
        }
        else
        {
                return (num * factorial(num-1));
        }
}

int main()
{
        printf("%d", factorial(4));

        return 0;
}
```



Inline functions

- Inline functions are functions that have small definitions and can be substituted at the place where the function call is made.
- There is **no guarantee** that the function will actually be inlined.
- **Compiler** does inlining for performing **optimizations**.
- **Advantages over Macros**:
  - **Since they are functions so type of arguments is checked by the compiler whether they are correct or not.**
  - **They can include multiple lines of code without trailing backslashes.**
  - **Inline functions have their own scope for variables and they can return a value.**
  - **Debugging code is easy in case of Inline functions as compared to macros.**

```
void inline swap(int* x, int* y);
void func_test() __attribute__((always_inline));
```

Callback functions

- The callback function is the function called using a pointer to that function, i.e it isn’t called directly by its name.
- It tells the lower layers modules what to invoke from upper layers when a specific event happens.

Callback functions

```c
// App.c
#include “led.h”
#include “timer.h”
....
int main()
{
     ....
     setOvfCallback(LED_off);
     LED_on();
     TIMER_start(1000);
     while(1)
     {

     }
}
```

```c
// Timer.c
int count = 0;
void static (*ovfCallback)(void);
....
void setOvfCallback(void (*Callback)(void))
{
     ovfCallback = Callback;
}

ISR(TIMER_OVERFLOW)
{
     if (count == 1000)
          ovfCallback();
}
```

Reentrant and Non-Reentrant functions

- The function is called **reentrant** if it can be **interrupted** in the middle of its execution **and be called** safely **again without** any **data corruption**.
- **Conditions that make the function Reentrant**:
  - **It shouldn’t use shared resource (global variable).**
  - **It shouldn’t modify its own code. (state changing in other contexts).**
  - **It shouldn’t call a non-reentrant function.**

```
/* Reentrant */
void swap(int* x, int* y)
{
        int temp;
        temp = *x;
        *x = *y;
        *y = *temp;
}
```

```
int temp;
/* Non-reentrant */
void swap(int* x, int* y)
{
        temp = *x;
        *x = *y;
        *y = temp;
}
```

Synchronous and Asynchronous functions

- A functions/tasks are told to be **synchronous** if **performed one at a time** and the following is waiting the previous functions to finish.
- A functions/tasks are told to be **asynchronous**, when you can **move to another task before the previous one finishes**.

Synchronous and Asynchronous functions

```
/*This shows an example of Synchronous function calls*/
#include <windows.h>
#include <process.h>
#include <stdio.h>

void Func1(void*);
void Func2(void*);

CRITICAL_SECTION Section; //This will act as Mutex

int main()
{
        InitializeCriticalSection(&Section);

        //Synchronous calling
        printf("Synchronous Calling\n");
        Func1(0);
        Func2(0);

        //This is done after all threads have finished processing
        DeleteCriticalSection(&Section);

        printf("Main exit");
        return 0;
}
```

```
void Func1(void *P)
{
  int Count;

  for (Count = 1; Count < 11; Count++)
  {
        EnterCriticalSection(&Section);
        printf("Func1 loop %d\n", Count);
        LeaveCriticalSection(&Section);
        Sleep(1000);
  }
  return;
}
```

```
void Func2(void *P)
{
  int Count;

  for (Count = 10; Count > 0; Count--)
  {
        EnterCriticalSection(&Section);
        printf("Func2 loop %d\n", Count);
        LeaveCriticalSection(&Section);
        Sleep(1000);
  }
  return;
}
```

Synchronous and Asynchronous functions

```
/*This shows an example of Asynchronous function calls*/

#include <windows.h>
#include <process.h>
#include <stdio.h>

void Func1(void*);
void Func2(void*);

CRITICAL_SECTION Section; //This will act as Mutex

int main()
{
        InitializeCriticalSection(&Section);
        //Asynchronous calling
        printf("Asynchronous calling\n");
        HANDLE hThreads[2];

        //Create two threads and start them
        hThreads[0] = (HANDLE)_beginthread(Func1, 0, NULL);
        hThreads[1] = (HANDLE)_beginthread(Func2, 0, NULL);

        //Makes sure that both the threads have finished before going further
        WaitForMultipleObjects(2, hThreads, TRUE, INFINITE);

        //This is done after all threads have finished processing
        DeleteCriticalSection(&Section);

        printf("Main exit");
        return 0;
}
```

```
void Func1(void *P)
{
  int Count;

  for (Count = 1; Count < 11; Count++)
  {
        EnterCriticalSection(&Section);
        printf("Func1 loop %d\n", Count);
        LeaveCriticalSection(&Section);
        Sleep(1000);
  }
  return;
}
```

```
void Func2(void *P)
{
  int Count;

  for (Count = 10; Count > 0; Count--)
  {
        EnterCriticalSection(&Section);
        printf("Func2 loop %d\n", Count);
        LeaveCriticalSection(&Section);
        Sleep(1000);
  }
  return;
}
```

Summary

- You have learned what types of functions you may interact with in embedded systems
- Recursive functions are complex and consumes stack
- Take care of non-reentrant functions
- Callbacks are strong tool used in embedded systems