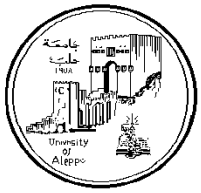
جامعة حلب
كلية الهندسة الكهربائية والالكترونية
قسم هندسة التحكم والأتمتة
مخبر التحكم



مقرر المتحكمات المصغرة

الجلسة التعريفية

السنة الرابعة ميكاترونيك

2023/2022

مدرس المقرر: _____

الغاية من الجلسة

1- التعرف على الدوال
2- الفرق بين الـ Declaration والـ Definition
3- ملفات c. وملفات الـ h.
4- المؤشرات (pointers)
5- Struct
6- Enum
7- استخدام typedef لتعريف نوع جديد من المتحولات
8- التعامل على مستوى البت BitWise
9- C preprocessor

1- التعرف على الدوال (التوابع) functions

الدالة (Function) عبارة عن مجموعة أوامر مجمعة في مكان واحد و تتنفذ عندما نقوم باستدعائها.
الدوال الجاهزة في C يقال لها **Built-in Functions**
الدوال التي يقوم المبرمج بتعريفها يقال لها **User-defined Functions**

- **بناء الدوال في C :**

عند تعريف أي دالة في C عليك إتباع الشكل التالي:

```
returnType functionName(Parameter)
{
  // Function Body
}
```

returnType: يحدد النوع الذي سترجعه الدالة عندما تنتهي أو إذا كانت لن ترجع أي قيمة.
functionName : يمثل الإسم الذي نعطيه للدالة ، و الذي من خلاله يمكننا استدعاءها.
Parameter : المقصود بها البارامترات (وضع البارامترات إختياري).
Function Body : تعني جسم الدالة، و المقصود بها الأوامر التي نضعها في الدالة.
نوع الإرجاع (returnType) في الدالة يمكن أن يكون أي نوع من أنواع البيانات الموجودة فيC (int - double - bool – string) إلخ.(..
في حال كانت الدالة لا ترجع أي قيمة، يجب وضع الكلمة void مكان الكلمة returnType.
مثال:

```
int getSum(int a, int b)
{
  return a + b;
}
int main()
{
  // result هنا قمنا بتخزين ناتج العددين 3 و 5 الذي سترجعه الدالة get_sum() في المتغير
  int result = getSum(3, 7);
 }
```

- **إعطاء قيمة إفتراضية للباراميترات في C**

تتيح لك لغة C وضع قيم إفتراضية للباراميترات مما يجعلك عند استدعاء الدالة مخيّر على تمرير قيم مكان الباراميترات بدل أن تكون مجبراً على ذلك، وتدعى القيمة الإفتراضية التي نضعها للباراميتر .Default Argument

```
int getSum(int a=1, int b=2)
{
  return a + b;
}
```

إذا كانت الدالة تملك أكثر من باراميتر و تريد وضع قيمة إفتراضية لأحد الباراميترات التي تمكلها فقط فيجب وضع الباراميترات التي تملك قيم إفتراضية في الآخر.

- **أين يجب تعريف الدوال في لغة C:**

مترجم لغة C يقرأ الكود سطراً سطراً مع تنفيذ الأوامر الموضوعة في كل سطر بشكل مباشر عندما يتم تشغيل البرنامج، لهذا السبب يجب دائماً أن تكون الدالة التي تريد استدعاءها معرّفة مسبقاً كي لا يظهر لك مشكلة عند تشغيل البرنامج.

في المثال التالي قمنا بوضع الدالة getSum() بعد الدالة التي قمنا باستدعائها منها (main())، المشكلة التي ستحدث عند التشغيل هنا سببها أن المترجم سيكون لا يعرف ما هي getSum() حيث أنه تم استدعاءها قبل أن يقوم المترجم قد سبق و قرأها:

```
int main()
{
  // x هنا قمنا بتخزين ناتج العددين 3 و 5 الذي سترجعه الدالة get_sum() في المتغير
  int result = getSum(3, 7);
}
int getSum(int a, int b)
{
  return a + b;
}
```

وسيظهر الخطأ : 'get_sum()' was not declared in this scope|
لحل مشكلة عدم التعرف على الدالة التي حدثت في المثال السابق عندنا خيارين:

- إبقاء الدالة 'get_sum() مكانها و ذكر تعريفها (Function Declartion) في أول الملف فقط، و هذه الطريقة تعتبر الأكثر تفضيلاً.
- وضع الدالة 'get_sum() فوق الدالة main() حتى يقوم المترجم بقرائها و التعرف عليها و تصبح قادر على استدعاءها في الدالة main() الموجودة بعدها.

```
#include <main.h>
```

هنا قمنا بوضع declaration للدالة get_sum() كي يتعرف عليها المترجم ويصبح بإمكاننا استخدامها فيما بعد//

```
int getSum(int a=1, int b=2);
int main()
{
  // getSum () هنا قمنا باستدعاء الدالة
  int result = getSum(3, 7);
}
```

هنا قمنا بتعريف(Definition) جسم الدالةgetSum أو بمعنى آخر تعريف ما سيحدث عندما يتم استدعاءها //

```
int getSum(int a, int b)
{
  return a + b;
}
```

2- الفرق بين الـ Declaration والـ Definition:

التعريف(Definition)	التصريح (Declaration)
يتم تعريف دالة أو متحول مرة واحدة فقط ضمن الكود	يمكن التصريح عن دالة أو متحول عدة مرات ضمن الكود
يتم حجز ذاكرة	لا يتم حجز ذاكرة
هنا يتم تحديد الوظيفة التي ستقوم بها هذه الدالة: **int f(int a)** **{** **return a;** **}**	يتم التصريح عن دالة بهذا الشكل: **int f(int);** وهنا نخبر المترجم بوجود دالة اسمها f تستقبل بارامتر واحد من نوع int وتعيد بارامتر من نوع int

مثال:

```
#include "main.h"
extern int variable1; <= variable declaration
extern int Function1(int a); <= function declaration
int main(void){}
```

حيث تدل كلمة extern أن هذا المتحول أو التابع موجود في مكان ما من المشروع.

3- ملفات c. وملفات h.:

تتضمن الملفات ذات الامتداد c. التعليمات المتعلقة بلغة الـ c والتي سيتم عمل compile(ترجمة) لها.

بينما تتضمن عادة الملفات ذات الامتداد h. التصريحات الخاصة بالمتحولات والتوابع المستخدمة و لايتم عمل compile(ترجمة) لهذا النوع من الملفات لذا وجب الانتباه إلى عدم إضافة أي تعليمات خاصة بلغة c ضمنها ، أيضاً لا يمكن إسناد قيم للمتحولات ضمنها وإنما فقط يمكن إضافة قيم ابتدائية لها أثناء التصريح عنها.

أما الغاية الحقيقية من تقسيم الكود إلى ملفات c. وملفات h. هو أننا أحياناً نحتاج أن نصرح عن عدة متحولات وعدة توابع ضمن عدة ملفات وقد يكون عددها كبير لذا كان من الأفضل فصل التصريح عن المتحولات والتوابع بملف منفصل بامتداد h. ، ومن ثم يصبح بإمكاننا تضمين هذا الملف عدة مرات ضمن عدة ملفات ، لكن وجب الانتباه إلى أن تضمين الملفات من نوع h. يعني نسخ محتوياتها إلى هذا الملف، لذا نجد عادةً في بداية كل ملف من نوع h. :

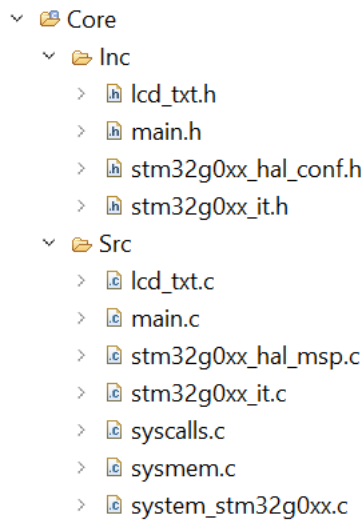
```
*Functions.h ×
1
2 #ifndef INC_FUNCTIONS_H_      <- أي في حال لم يتم تعريف الماكرو INC_FUNCTION_H
3 #define INC_FUNCTIONS_H_      <- قم بتعريف الماكرو INC_FUNCTION_H
4
5 //Function1(int a) declaration
6 int Function1(int a);          <- ضمن الماكرو نقوم بالتصريح عن التوابع والمتحولات اللازمة
7                                  ( وبذلك نضمن عدم تكرار تضمين ملف من نوع .h )
8 #endif /* INC FUNCTIONS H  */
```

ثم يتم تضمين ملف: functions.h ضمن ملف الـ main.c بالشكل التالي:

```
*Functions.h  *main.c ×
1 #include "main.h"
2 #include "Variables.h"
3 #include "Functions.h"
```

ملاحظة:

يتم عادة إنشاء مجلدين في كل تطبيق الأول باسم src ويتضمن جميع الملفات ذات الامتداد .c ، ومجلد باسم Inc ويتضمن جميع الملفات ذات الامتداد .h. كما في الشكل التالي:



4- المؤشرات Pointers:

المؤشر يقوم بحجز مساحة في الذاكرة لتخزين عنوان الشيء الذي يؤشر إليه، لتعريف مؤشر جديد نستخدم الرمز * مع الإشارة إلى أن نوع المؤشر يجب أن يكون نفس نوع الشيء الذي سيشير له في الذاكرة فإذا أردنا تعريف مؤشر نوعه int واسمه x عندنا ثلاث خيارات كالتالي:

```
// الأسلوب الأول و الذي يعتبر الأكثر استخداماً //
int* x;

// الأسلوب الثاني //
int *x;

// الأسلوب الثالث //
int * x;
```

الآن، لتعريف مؤشر و جعله يشير لقيمة شيء موجود في الذاكرة، يجب أن نقوم بتمرير عنوان هذا الشيء كقيمة للمؤشر و عندها سيصبح المؤشر قادر على الوصول لقيمته و عنوانه كما سنرى في المثال التالي:

```
//Getting the address of the variable
    ptr = &var8Bit;

    //Accessing var8Bit by its address
*ptr = 100
```

فأصبحت هنا قيمة المؤشر هي عنوان المتحول var8Bit ، أما القيمة التي يشير إليها المؤشر فهي قيمة المتحول var8Bit

- **تمرير البارامترات للتوابع كقيم Passing by value:**

بفرض لدينا التابع التالي:

```
uint8_t Add_5_PassByValue(uint8_t value)
{
    value += 5;
    return value;
    }
```

وهو عبارة عن تابع يقوم بإضافة الرقم 5 للمتحول الذي يتم تمريره، ولاستدعاء هذا التابع:

```
var32Bit = Add_5_PassByValue(x);
```

فهنا نلاحظ أننا قمنا بتمرير البارامتر للتابع كقيمة مباشرة x، حيث تم أخذ نسخة من المتحول x الذي تم تمريره ثم تمت عملية الإضافة عليه دون أن يتأثر المتحول الأصلي x

- **تمرير عنوان للتابع Passing by reference:**

بفرض لدينا التابع التالي:

```
uint8_t Add_5_PassByReference(uint8_t* pData)
{
```

```
	*pData += 5;
	return *pData;
}
```

وهو عبارة عن تابع يقوم بإضافة الرقم 5 للمتحول الذي يتم تمريره، ولاستدعاء هذا التابع:

```
var32Bit = Add_5_PassByReference(&x);
```

فهنا نلاحظ أننا قمنا بتمرير عنوان المتحول x للتابع ، وفي هذه الحالة سيتم إضافة القيمة 5 للمتحول x نفسه

- **تمرير مصفوفة للتابع Passing an array:**

بفرض لدينا التابع التالي:

```
uint16_t CalculateSumOfValues(uint8_t* pData, uint8_t length)
{
	uint16_t value = 0;
	for(uint8_t i = 0;i<length;i++)
	{
		value += pData[i];
		//OR: value += *(pData + i);
	}

	return value;
}
```

وهو عبارة عن تابع يقوم بجمع عناصر المصفوفة، ولاستدعاء هذا التابع:

```
sum = CalculateSumOfValues(arrayOfNumbers, 10);
```

فهنا قمنا بتمرير المصفوفة من خلال اسمها arrayOfNumbers وتمرير حجمها، مع ملاحظة أن اسم المصفوفة يمكن اعتباره مؤشر يعبر عن عنوان العنصر الأول في المصفوفة

يمكن أيضاً استدعاء هذا التابع من خلال تمرير عنوان العنصر الأول في المصفوفة بشكل مباشر

```
sum = CalculateSumOfValues(&numbers[0], 10);
```

- **تمرير متحول الخرج للتابع كعنوان والتأكد من صحة إدخال البارامترات وصحة تنفيذ التابع:**

هذا الحالة نستخدمها بكثرة في الحياة العملية كما تستخدمها جميع توابع مكتبة HAL ، ولفهم بنيتها سنفرض أنه لدينا التابع التالي:

```
uint8_t CalculateSumOfValues(uint8_t* pData, uint8_t length, uint16_t* sum)
{
    if(sum == NULL || pData == NULL)
        return 1;

    uint16_t value = 0;
    for(uint8_t i = 0;i<length;i++)
    {
        value += pData[i];
    }
    *sum = value;
    return 0;
}
```

مصفوفة — حجم المصفوفة — متحول الخرج

validation التحقق من صحة البارامترات المدخلة او مايعرف بالـ
وفي حال لم يتم إدخال البارامترات بشكل صحيح يتم ارجاع القيمة 1 لتنبيه المستخدم
على الادخال الخاطئ للبارامترات وفي هذه الحالة يتم الخروج من التابع دون اكمال تنفيذه

هنا تم الاسناد بشكل مباشر لمتحول الخرج دون الحاجة لوضعه ضمن الـ Return باعتباره من نوع pointer

هنا تم ارجاع القيمة 0 والتي تخبر المستخدم أنه تم تنفيذ التابع بشكل صحيح

ولاستدعاء هذا التابع:

```
status = CalculateSumOfValues(NULL,10,&var16Bit);
```

ففي حال كان status=1 هذا يعني أن هناك خطأ ما بإدخال البارامترات، أما إذا كان status=0 فهذا يعني أنه تم تنفيذ التابع بشكل صحيح

5- **Struct**:

الكلمة تستخدم لتعريف نوع جديد و هذا النوع يمكنه أن يحتوي على مجموعة من القيم من أي نوع كانت بشكل مرتب و سهل التعامل معها.

كمثال بسيط, إذا كنا ننوي إرسال معلومات مجموعة من المنتجات و كل منتج يملك المعلومات التالية:

إسم المنتج, تاريخ إنتاجه, سعره و مكوناته.هنا سيكون خيار ممتاز أن ننشئ نوع جديد يمثل المنتج, أي نوع فيه المعلومات الأساسية التي لا بد أن يمتلكها أي منتج، و عندها أي منتج جديد نريد تعريفه, نجعله نسخة منه.

أي نوع جديد تعرّفه بواسطة الكلمة Struct يقال له Structure ، وأي نسخة تنشئها من النوع الجديد الذي قمت بتعريفه يقال لها كائن object

لتعريف struct جديد نتبع الأسلوب التالي:

```
struct struct_name {
   member_definition;
   member_definition;
   member_definition;
 } object_names;
```

struct_name: مكانها نضع الإسم الذي سنعطيه للنوع الجديد.

member_definition; : هنا يمكنك تمرير إسم و نوع أي شيء تنوي جعل النوع الجديد يملكه.

object_names: إذا أردت إنشاء كائن (نسخة) من النوع الجديد مباشرةً عند تعريفه, فأي إسم تضعه هنا سيتم إعتباره كائن منه.

في المثال التالي, قمنا بتعريف نوع جديد إسمه Book يمثل المعلومات التي يمكن أن يتضمنها أي كتاب كعنوانه, إسم المؤلف, سعره و عدد صفحاته.

```
struct Book {
   string title;
   string author;
   double price;
   int numberOfPages;
};
```

لإنشاء كائن من struct هناك عدة طرق يمكن اتباعها لإنشاء كائنات من struct:

الطريقة الأولى:

```
struct Book {
   string title;
   string author;
   double price;
   int numberOfPages;
};
// هنا قمنا بإنشاء كائن من Book إسمه book
struct Book book;
```

الطريقة الثانية:

```
struct Book {
   string title;
   string author;
   double price;
   int numberOfPages;
} book;
```

الطريقة الثالثة:

```
struct Book {
   string title;
   string author;
```

```
    double price;
    int numberOfPages;
};
// هنا بإنشاء ثلاث كائنات من Book, الأول إسمه book1, الثاني إسمه book2 و الثالث إسمه book3
struct Book book1;
struct Book book2;
struct Book book3;
```

الطريقة الرابعة:

```
struct Book {
    string title;
    string author;
    double price;
    int numberOfPages;
};
// هنا بإنشاء ثلاث كائنات من Book, الأول إسمه book1, الثاني إسمه book2 و الثالث إسمه book3
struct Book book1, book2, book3;
```

- **الوصول للأشياء الموجودة داخل كائن من struct:**

للوصول لقيم المتغيرات الموجودة فيه, نستخدم العامل . أي النقطة العادية، مثال:

```
struct Book {
    string title;
    string author;
    double price;
    int numberOfPages;
};
int main()
{
    // هنا قمنا بتعريف كائنين من Book, الأول إسمه book1 و الثاني إسمه book2
    struct Book book1;
    struct Book book2;
    // هنا قمنا بإعطاء قيم لمتغيرات الكائن book1
    book1.title = "C++ for beginners";
    book1.author = "Mhamad Harmush";
    book1.price = 9.99;
    book1.numberOfPages = 420;
    // هنا قمنا بإعطاء قيم لمتغيرات الكائن book2
    book2.title = "Network 1";
    book2.author = "Nadine Masri";
```

```
book2.price = 22.49;
book2.numberOfPages = 310;
```

6- Enum:

النوع enum يستخدم لتعريف قائمة من القيم أو عدة مجموعة قيم ثابتة بشكل منطقي، في حال أردت تعريف مجموعة قيم مترابطة يستحيل أن تتبدل فالخيار الأمثل هو تعريف هذه القيم في الأساس بداخل enum .

أمثلة حول بعض المعلومات الثابتة في الحياة و التي أيضاً تعتبر ثابتة في المنطق هي:

- فصول السنة (الخريف, الشتاء, الربيع, الصيف)
- الإتجاهات (الشمال - الجنوب - الشرق – الغرب)
- أيام الأسبوع (الإثنين - الثلاثاء - الأربعاء إلخ..)
- أشهر السنة(كانون الثاني - شباط - آذار إلخ..)
- الجنس (ذكر – أنثى)

عند وضع متغيرات بداخله يمكنك إعطاؤهم قيم بنفسك أو ترك المترجم يعطيهم قيم إفتراضية بنفسه.
إفتراضياً المتغيرات التي تضعها يتم إعطاؤها قيم بالترتيب إبتداءاً من صفر, فمثلاً أولاً أول متغير تكون قيمته 0 والثاني تكون قيمته 1 وهكذا..
مثال:

```
enum Status
{
        Temp_Status_Ok = 0,
        Temp_Status_Error
};
```

ولاستخدام هذا الـ enum في تابع معين:

```
enum Status CheckValuesUsingEnumStatus(uint8_t value)
{
        enum Status status;
        if(value >= MIN_VALUE && value <= MAX_VALUE)
                status = Temp_Status_Ok;
        else
                status = Temp_Status_Error;

        return status;
}
```

7- لتعريف نوع جديد من المتحولات باستخدام typedef:

مثلاً إذا أردنا تعريف نوع جديد من المتحولات وتكون بنيته من نوع enum نكتب التالي:

```
typedef enum
```

```
{
        Status_Ok = 0,
        Status_Error
}Status_t;
```

حيث هنا قمنا بتعريف نوع جديد من المتحولات باستخدام typedef وهذا النوع يعتمد في بنيته على النوع enum واسم هذا النوع الجديد Status_t، وفي هذه الحالة اذا أردنا استخدام هذا النوع الجديد في تابع معين نكتب:

```
Status_t CheckValuesUsingStatus_t(uint8_t value)
{
        Status_t status;
        if(value >= MIN_VALUE && value <= MAX_VALUE)
                status = Status_Ok;
        else
                status = Status_Error;

        return status;
}
```

حيث نلاحظ هنا أننا أصبحنا نستخدم الكلمة Status_t كما نستخدم أي نوع من أنواع البيانات المعرفة مسبقاً ضمن اللغة كالـ int، float، string وغيرها

8- التعامل على مستوى البت BitWise :

في كثير من الأحيان نحتاج لتفعيل بت معين من أحد المسجلات أو تصفير هذا البت أو عكس حالته أو فحص قيمته ، وغيرها من العمليات المنطقية وهي ماتسمى بالـ Bitwise:

Operator	Description
&	bitwise AND
|	bitwise OR
^	bitwise exclusive OR
<<	shift left
>>	shift right
~	one's complement

- **لتفعيل بت معين:**

```
bits = bits | (1 << 7) ;   /* sets bit 7 */
```

$b_7b_6b_5b_4b_3b_2b_1b_0$ | 10000000 → $1b_6b_5b_4b_3b_2b_1b_0$

وغالباً ما تكتب بالشكل التالي:

```
bits |= (1 << 7) ; /* sets bit 7 */
```

- **لتصفير بت معين:**

```
bits &= ~(1 << 7) ; /* clears bit 7 */
```

```
(1 << 7)    ➡  10000000
~(1 << 7)   ➡  01111111
```

وغالباً ما تتم كتابتها بالشكل التالي:

```
bits &= ~(1 << 7) ; /* clears bit 7 */
```

- **لعكس حالة بت معين:**

```
bits ^= (1 << 7) ; /* sets bit 7/*
```

- **لفحص حالة بت:**

```
if ((bits & 64) != 0)     /* check to see if bit 6 is set */
```

$b_7b_6b_5b_4b_3b_2b_1b_0$ & 01000000 ➡ $0b_6000000$

ويمكن أيضاً كتابتها بالشكل:

```
if (bits & (1 << 6)) /* check to see if bit 6 is set */
```

9- C preprocessor:

هي مرحلة تسبق عملية الترجمة compile إذا تتم معالجة مسبقة للكود قبل تمريره إلى المترجم compilerوكمثال بسيط عندما يكون هناك أمر توجيهي include فإنه يتم البحث عن المكتبة أو الملف المحدد ويتم تضمينه في الملف المصدري. إذ أن مهام المعالجة المسبقة preprocessor لا تتعلق بتنفيذ الكود وترجمته وإنما في بعض الإجرائيات الغير تنفيذية وإنما التوجيهية directive كما سنرى لاحقاً كالبحث عن الكلمات الرمزية المعرفة واستبدالها بالتعريفات الخاصة بها.

بعض استخدامات C preprocessor

سنذكر بعض التطبيقات مع شرح مبسط، وسيتم لاحقاً الحديث بشكل مفصل عن كل تطبيق منها.

- **include** : أمر لتضمين ملف أو مكتبة معينة.
- **Define** : أمر لتعريف دلالات لكلمات كأن نعرف أنه كلما وردت كلمة Pi في الكود فهو يعني 3.14 وكذلك لإعطاء معرّف(رمز) لكتلة أسطر برمجية مع إمكانية وجود وسطاء arguments وهذا ما يسمى **macros** function-like .

- **pragma** : أمر لتحديد بعض الأوامر للمترجم compiler
- **الترجمة الشـرطية conditional compilation** : مجموعة أوامر تسـتخدم في الترجمة حسـب شروط معينة، مثال: نخبر المترجم أنه لو كنت في نظام تشغيل ويندوز قم بتضمين المكتبة الفلانية أما لو كنت في نظام تشغيل لينكس فقم بتضمين مكتبة أخرى.

C preprocessor syntax

أي سطر برمجي يبدأ برمز المربع # hash فإن ما يليه هو أمر سيوجه إلى ال preprocessor .

معنى #include

أكثر الاستخدامات شيوعاً للC preprocessor وهو أمر توجيهي من أجل تضمين مكتبة (وهي عبارة عن مجموعة تعريفات) أو حتى ملفات مصدرية أخرى

مثال:

بفرض الكود المصدري للملف main.c يقوم بتضمين الملفات التالية:

```
#include "main.h"
#include <stdio.h>
#include "../ECUAL/LCD16x2/LCD16x2.h"
```

نلاحظ أن التضمين الثاني كان بين قوسين <> والأول والثالث بين إشارتين " " ،حيث أن الاستخدام الثاني يكون عندما نريد أن يتم البحث عن الملف المحدد ضمن المسارات المتاحة في إعدادات المترجم. والاستخدام الأول والثالث يكون عندما نريد البحث عن الملف المحدد ضمن المجلد المصدري نفسه أو ضمن مسار نقوم نحن بتحديده.

معنى #define

تتسمى مجموعة الأسطر البرمجية المعرفة عبر define بـmacros .

يمكننا من خلال الC preprocessor إنجاز ما يشـبه التوابع ولذلك تسـمى function-like مع اختلاف جوهري بين التوابع و function-like macros، شـكل الشـبَه الوحيد هو إمكانية إنجاز macros يمكن أن تأخذ وسطاء، أما من الناحية التنفيذية فليس هناك أي وجه شبه:

- function : كتلة من الكود البرمجي يمكن أن نسـتدعيها بأي مكان من البرنامج الأسـاسـي لتقوم بالتنفيذ واستخدام الوسطاء في حال وجودها مع إمكانية إرجاع قيمة بعد التنفيذ.
- Function-like macros : مجموعة من الأسطر البرمجية المختصـرة برمز، و عند إيراد هذا الرمز في الكود فهو بمثابة إيراد هذه الأسطر البرمجية كما هي، بخلاف مبدأ التوابع المعتمد على الاستدعاء مع وجود الكود دون تكرار .

لابد التنويه إلى الجانب السلبي لاستخدام الC preprocessor وهو حجم البرنامج النهائي، إذ أن استخدام الـ C preprocessor لإنجاز ما يشـبه التوابع function-like macros يؤدي إلى تضـاعف حجم البرنامج، فكما قلنا في الجزء الأول اسـتخدام الC preprocessor لا يتعدى في النهاية عن اختصـار أسـطر برمجية

بكلمات مفتاحية ولكنها من ناحية الأداء لا تختلف لو كتبنا الأســـطر البرمجية بدل الكلمات المفتاحية، بعكس التوابع التي مهما كررنا استدعاءها فإن الكود واحد لا يتكرر.
مثال: ليكن لدينا الكود التالي:

```
#include "main.h"
#define portA_on  GPIOA->ODR = 0Xffff
#define portA_off  GPIOA->ODR = 0X0000
#define leds  GPIOA
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
int main(void)
{
   HAL_Init();
   SystemClock_Config();
   MX_GPIO_Init();
  while (1)
 {
          portA_on;
              HAL_Delay(250);
         //***************
              portA_off;
              HAL_Delay(250);
}}
```

ففي الكود السابق سيقوم الــ preprocessor بالبحث عن أماكن ورود PORT_on و PORT_off و leds ويقوم باســـتبدالها بما هو مذكور في تعريفها. أي أن الكود ســـيصـــبح كالتالي قبل دخوله في عملية الترجمة الفعلية:

```
  while (1)
 {
             GPIOA->ODR = 0Xffff
              HAL_Delay(250);
         //***************
             GPIOA->ODR = 0X0000
              HAL_Delay(250);
}}
```

ملاحظة مهمة :إن ما يرد بعد الأمر التوجيهي#define لا يتم معالجته أو تنقيح أخطاؤه.

ما فائدة استخدام الـ C preprocessor حتى الآن؟
لنفترض أن الكود الذي كتبناه سيتم تنفيذه على نفس المتحكم ولكن بوابة A غير متاحة للاستخدام ،فكل ما علينا فعله هو تعديل الكود كالتالي:

```
#define portB_on  GPIOB->ODR = 0Xffff
#define portB_off  GPIOB->ODR = 0X0000
```

وبهذا جعلنا الكود قابل لإعادة الاستعمال بتعديل واحد فقط، طبعاً لا يمكن الشعور بأهمية ذلك إلا في الحالات الأكثر واقعية وتعقيداً.

بعض الفوائد الأخرى

- قابلية إعادة استخدام الكود حتى لو تم تبديل عائلة المتحكم الذي نعمل عليه أو بعض التوصيلات.
- التخلص من التعامل مع العمليات المعقدة مثل عمليات تعديل بيتات معينة في السجلات.

function-like macros syntax:

لكتابة macros يأخذ بارامتر يجب أن نتبع اسم المايكرو (الرمز) بأقواس متوسطة تحوي أسماء البارامترات مباشرة ، دون تحديد نوعها كما في التوابع، إذ لا يهم الـــ function-like macros نوع البارامترات إذ لا تتعامل معها كبارامترات حقيقيين وإنما كقيم مجردة سيتم تمريرها إلى تعريف الـ macros.

مثال عملي:

على استخدام الـ C preprocessor لتسهيل التعامل مع بتات السجلات أو ما يسمى Bitwise Operations :

```
#include "main.h"
/* ——— Macros For Building BitFields ———— */
#define BIT_SET(ADDRESS,BIT) (ADDRESS |= (1<<BIT))
#define BIT_CLEAR(ADDRESS,BIT) (ADDRESS &= ~(1<<BIT))
#define BIT_FLIP(ADDRESS,BIT) (ADDRESS ^= (1<<BIT))
#define BIT_GET(ADDRESS,BIT) (ADDRESS & (1<<BIT))

void SystemClock_Config(void);
static void MX_GPIO_Init(void);
int main(void)
{
    HAL_Init();
    SystemClock_Config();
```

```
    MX_GPIO_Init();
    int8_t Number1=0b00000110;
    int8_t Number2=0b00000000;
 while (1)
  {
 /* ————————- Example Usage ——————————-*/
       if(BIT_GET(Number1,1) == 0)
          BIT_SET(Number2,5);
       else
          BIT_CLEAR(Number2,5);
 /* ———————————————————— */
 }}
```

وكنتيجة لتنفيذ الكود السابق ستصبح قيمة المتحول Number2=0b00100000

سنأخذ أحد الـ function-like macros كمثال:

```
#define BIT_SET(ADDRESS,BIT) (ADDRESS |= (1<<BIT))
```

اسم الـ macro هو BIT_SET والبارامترات هم ADDRESS,BIT وعند الاستخدام نمرر البارامترات التي نريد، مثال:

```
BIT_SET(Number2,5);
```

وهذا مكافئ تماماً للتالي:

```
Number2 |= (1<<5)
```

الترجمة الشرطية conditional compilation

تستخدم الأوامر if, ifdef, ifndef, else, elif ,endif في الترجمة الشرطية.

- **#if … #endif**

إن if و endif تحويان كود برمجي ينفذ عندما يكون الوسيط بعد if له قيمة غير صفرية، مثال:

```
#define DEBUG 1
#if DEBUG
#include <debug.h>
#endif
```

وليس بالضـــرورة أن يكون الكود بين if و endif هو من أوامر C preprocessor إذا يمكن أن نكتب أي كود نريد. وبالتالي يمكن اسـتخدام هذه الخاصـية أيضـاً في جعل الكود قابل لإعادة الاسـتخدام أو التخصـيص عبر المطور.

- **#elif … #else**

تستخدم else لتوجيه المترجم لكود آخر في حال عدم تحقق شرط if

```
#define DEBUG 1
#if DEBUG
#include <debug.h>
#endif
```

أما elif فهي تختلف عن else بوجود شرط تقوم بتفحصه، مثال:

```
#define STATE DEBUG
#if STATE == DEBUG
#include <debug.h>
#elif STATE == TESTING
#include <testing.h>
#elif STATE == RELEASE
#include <machine.h>
#endif
```

- **#ifdef … #ifndef**

يستخدم ifdef من أجل تضمين كود في حال التأكد من أن الوسيط معرف، أما ifndef فهي من أجل تضمين كود في حال أن الوسيط غير معرف. يجب الانتباه أننا هنا لا نتحدث عن وسيط له قيمة غير صفرية وإنما هل هو معرف أم لا.

```
#define DEBUG
#ifdef DEBUG
#include <debug.h>
#endif
#ifndef DEBUG
```

```
#include <machine.h>
#endif
```

يمكن استخدام الC preprocessor من أجل تمرير رسائل معينة لإظهارها أثناء الترجمة في حال حدوث شرط معين، مثال:

```
#if RUBY_VERSION == 190
# error 1.9.0 not supported
#endif
```

ففي الكود السابق يقوم بتفحص رمز سابق (معرّف مسبقاً لدى المترجم) وفي حال لم يكن بقيمة معينة فإنه يظهر رسالة إلى المستخدم.