بسم الله الرحمن الرحيم

الحمد لله رب العالمين والصلاة والسلام على أشرف المرسلين، بفضل الله وتوفيقه نقدم لكم الدرس الخامس بدورة

أساسيات البرمجة بلغة السي بلس بلس C++

بعنوان

الدوال في لغة السي بلس بلس C++ (ج٢)

أنواع معرفات الدوال والمتغير

تحتوي الدوال فى لغة السي++ على معرف يحدد نوع الدالة Storage Classes وهذا المعرف يكون احد القيم الاتية auto, register, extern, mutable static

هذه المعرفات تحدد المدة التي يكون فيها المتغير او الدالة متاح فى الذاكرة، فبعض المتغيرات تكون لمدة محدودة فى الذاكرة وبعضها يكون متاح دائما اثناء تشغيل البرنامج.

تنقسم المعرفات الي نوعين رئيسين وهما المعرفات متغيرة او أوتوماتيكية والمعرفات ثابتة، auto and register هما معرفان من النوع المتغير حيث يتم تخصيص جزء من الذاكرة فقط عندما يدخل تنفيذ

البرنامج الحيز الذى يتم فيه تعريف المتغيرات باستخدام هذه المعرفات وعندما يغادر البرنامج سيتم ازالة الجزء المخصص لهم فى الذاكرة.

المتغيرات الخاصة

المتغيرات الخاصة هي التي يتم تعريفها داخل الدوال وقد قمنا بالدروس السابقة بتعريف العديد من المتغيرات من هذا النوع فمثلا فى الدرس السابق قمنا بعمل برنامج لحساب الجذر التربيعي وبه المتغيرات الاتية

```
int inputNumber;
double outputNumber;
```

لقد قمنا فقط بتعريف نوع المتغير int او double ولم نقم بتحديد المعرف وذلك لان المتغيرات التي تكون داخل الدوال هي فى الغالب متغيرات خاصة ان لم يتم تحديد غير ذلك ويكون معرفها من النوع المتغير او أوتوماتيكيي، واذا اردنا تحديد ذلك صراحة فى البرنامج يمكننا استخدام النوع auto وسيكون تعريف المتغيرات كما يلي

```
auto int inputNumber;
auto double outputNumber;
```

هذه المتغيرات تكون متاحة فقط داخل الدالة التي تم تعريفهم فيها ولان المتغيرات الخاصة تكون معرفة باستخدام auto ضمنيا فمن النادر جدا استخدام هذا المعرف. وينصح دائما تعريف المتغيرات او الدوال خاصة طالما ليست هناك حاجة لان تكون عامة.

تعريف المتغيرات باستخدام المعرف register

عادة يتم نقل البرنامج من الذاكرة وتخزينه موقتأ فى مسجلات سريعة حتى يتم تنفيذه وبعدها يتم نقل النتائج مرة اخرى الي الذاكرة واذا قمنا باستخدام المعرف register فاننا نخبر المجمع ان يقوم بتخزين هذا المتغير فى احدى مسجلات تحزين البرنامج بدلا من تخزينه فى الذاكرة وتمتاز المسجلات بالسرعة العالية فى الحسابات والقراءة والكتابة ولذاك عادة نستخدم هذا المعرف مع العدادات وبهذا نقلل الوقت الذى يتم فيه النقل من الذاكرة الي المسجلات والعكس.

مثال

```
register int inputNumber;
register double outputNumber;
```

فى بعض الاحيان قد يتجاهل المجمع اننا نريد تخزين هذا المتغير فى احدى المسجلات فى حالة مثلا عدم وجود مسجلات كافيه، ويرجى الانتباه ان register يتسخدم فقط مع المتغيرات الخاصة او معطيات الدوال.

وأغلب المجمعات اليوم تقوم أوتوماتيكيا بتخزين المتغيرات التي ترى انها تستخدم بكثرة فى البرنامج فى المسجلات والمتغيرات التي لا تستخدم بكثرة تقوم بتخزينها في الذاكرة حتى لو قمنا باستخدام المعرف register لانها ترى انه لا حاجة لذلك.

المتغيرات الثابتة

يستخدم دائما المعرفان extern و static لتعريف المتغيرات الثابته وعلى عكس المعرفات المتغيرة فاننا اذا قمنا بتعريف دالة او متغير باستخدام extern و static فانهم يكونوا متاحين دائما فى الذاكرة ويمكن قراءتهم وكتابتهم فى اى وقت طوال مدة تنفيذ البرنامج وليس فقط وقت تنفيذ المكان الذى تم استخدامهم فيه كما فى المعرفات المتغيرة.

ويجب هنا الانتباه الي ان المتغيرات او الدوال التي تم تعريفهم باستخدام extern و static وهما متاحين بالذاكرة فلا يمكن قراءتهم وكتابتهم من خارج الدالة اذا كانوا قد تم تعريفهم كمتغيرات عامة وهذا عندما يتم تعريفهم فى خارج الدالة او فى ملف التعريف الهيدر.

المثال الاتي يتكون من ٣ ملفات

الملف الاول header.h ويحتوي على الاتي

```
#ifndef HEADER_H_
#define HEADER_H_

// declare global variable
extern int number2;

// declare global function
extern void fun2(void);

#endif
```

الملف الثاني other_source.cpp ويحتوي على الاتي

```
#include <iostream>
#include "header.h"

using namespace std;
```

```
// this global function will be called from main
extern void fun2(void)
{
   // write global variable
   number2 = number2 + 30;
   cout << "number3 = " << number2 << endl;
}
```

الملف الثالث main_source.cpp ويحتوي على الاتي

```
#include <iostream>
#include "header.h"

using namespace std;

// static can be removed
static int number1 = 20;

// global variables has be defined
int number2;

// static can be removed
static void fun1(void);

// static function can be called only from this file
static void fun1(void)
{
   cout << "number1 = " << number1 << endl;
}

int main()
{
   // can be also defined as auto
   int number = 10;

   // global variable
   number2 = 30;

   cout << "number = " << number << endl;

   fun1();
   fun2();

   return 0;
}
```

ونرى فى هذا المثال هو وجود ملف هيدر header.h وتحتوي ملفات التعريف او الهيدر على تعريف للدوال والمتغيرات العامة global وقد تحتوي على ماكرو وهي تعريفات صغيرة لبعض الثوابت او الحسابات الصغيرة.

فى الملف header.h يوجد المتغير number2 ومعرفه extern وكذلك الدالة fun2 ومعرفها extern والان اى ملف source يقوم بادراج ملف الهيدر header.h يمكنه من استخدام المتغير number2 او الدالة fun2 كما نرى فى الملف other_source.cpp و main_source.cpp هناك السطر الاتي

```
#include "header.h"
```

ونلاحظ هنا اننا استخدمنا علامات التنصيص وهي دائما تستخدم مع ملفات الهيدر التي قمنا بعملها اما ملفات النظام فستكون بين علامات الاسهم < >

اى دالة او متغير يتم ذكره فى ملف الهيدر لابد ان يكون له تعريف فى مكان بالبرنامج فمثلا المتغير number2 تم تعريفه فى الملف main_source.cpp اما الدالة fun2 فقد تم تعريفها فى الملف other_source.cpp

يحتوي الملف main_source.cpp على المتغير الاتي

```
// static can be removed
static int number1 = 20;
```

وهو من النوع static ويمكننا ايضا تعريفه بدون ذكر static لانه من الطبيعي ان لم يكن قد تم تعريفه فى ملف الهيدر فانه سيكون داخل الملف main_source.cpp فقط اما اذا قد تم تعريفه فى ملف الهيدر مثل number2 وبدون ذكر extern فقد تم اعتباره متغير عام لانه معرف فى ملف الهيدر.

نلاحظ ايضا وجود متغيرات خاصة مثل الذى تم تعريفه فى الدالة main

```
int number = 10;
```

وهذا متغير خاصة داخل الدالة main وله المعرف المتغير auto وحتى ان لم يتم ذكره.

مثال لتوضيح الفرق بين المتغيرات العامة والخاصة

```
#include <iostream>
using std::cout;
using std::endl;

void func1( void );
void func2( void );
void func3( void );

// static variable used only in this file
int number = 1;

int main()
{
   cout << "static number defined outside main is " <<
number << endl;

   // define local variable to main only
   int number = 5;

   cout << "local number defined inside main is " <<
number << endl;

   // define a new scope, this hides the other
definitions
   {
      int number = 7; // hides x in outer scope
      cout << "local number defined inside main inside a
scope is " << number << endl;
   }

   // now we are outside the defined scope.
   cout << "local number defined inside main is " <<
number << endl;

   func1();
   func2();
```

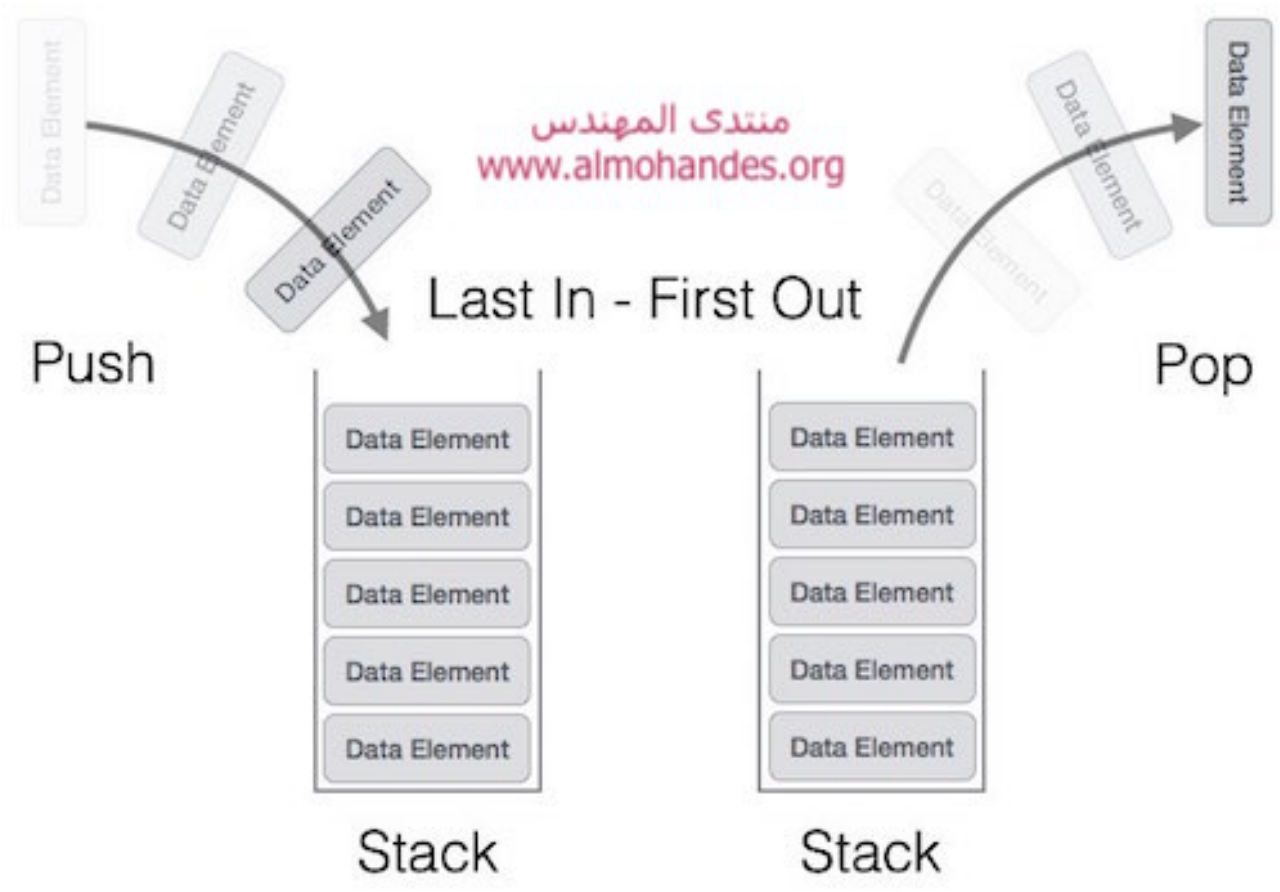
```
   func3();
   func1();
   func2();
   func3();

   cout << "local number defined inside main is " <<
number << endl;
   return 0;
}

void func1( void )
{
   // local variable initialized each time func1 called
   int number = 25;

   cout << "local number is " << number << " on entering
func1" << endl;
   number++;
   cout << "local number is " << number << " on exiting
func1" << endl;
}

void func2( void )
{
   // local variable with static
   // will be not initialized each time func2 called
   static int number = 50;

   cout << "local static number is " << number << " on
entering func2" << endl;
   number++;
   cout << "local static number is " << number << " on
exiting func2" << endl;
}

void func3( void )
{
   cout << "static number is " << number << " on entering
func3" << endl;
   number += 10;
   cout << "static number is " << number << " on exiting
func3" << endl;
}
```

نتيجة التنفيد ستكون كما يلي

```
static number defined outside main is 1
local number defined inside main is 5
local number defined inside main inside a scope is 7
local number defined inside main is 5
local number is 25 on entering func1
local number is 26 on exiting func1
local static number is 50 on entering func2
local static number is 51 on exiting func2
static number is 1 on entering func3
static number is 11 on exiting func3
local number is 25 on entering func1
local number is 26 on exiting func1
local static number is 51 on entering func2
local static number is 52 on exiting func2
static number is 11 on entering func3
static number is 21 on exiting func3
local number defined inside main is 5
```

استعداء الدوال فى لغة السي بلس بلس

للتعرف على كيفية استعداء الدوال فى لغة السي بلس بلس يجب علينا اولا التعرف على ال stack وكيفية عمله فلنفترض ان لدينا عمود من الاطباق مرصوصة فوق بعضها البعض فاذا اردنا اضافة طبق لهذه الكومة فيمكننا اضافته من الاعلي فقط وكذلك عند اضافة عنصر معين لل stack يتم اضافته من الاعلى فقط وتسمى هذه العملية pushing واذا اردنا سحب عنصر معين فيمكن فقط سحب العنصر فى الاعلى وتسمى هذه العملية popping وتسمى هذه العملية بلغة السوفت وير last-in, first-out (LIFO) data structures اى ان اخر من يدخل ال stack هو اول من يخرج منه. ويستخدم جزء معين من stack لتخرين العنوان الذى يتم الرجوع عليه بعد انتهاء تنفيذ الدالة ويسمى function call stack او program execution stack.



والان عند تنفيذ اى دالة فى لغة السي بلس بلس فانها ستستدعي دالة اخرى وهذه الدالة الاخرى قد تستعدي دالة اخرى وهكذا فمثلا فى المثال السابق الدالة main قامت باستعداء مجموعة من الدوال الاخرى فعند دخول الدالة func1 وتنفيذها يجب الرجوع للدالة main وتكملة تنفيذ البرنامج

```
int main()
{
   func1();
   func2();
   func3();
   func1();
   func2();
   func3();
}
```

ولذلك يجب تخزين المكان المطلوب الرجوع اليه pushing بعد الانتهاء من تنفيذ كل دالة فى ال stack ثم يتم تنفيذ الدالة وعند الرجوع

للدالة يتم ازالة هذا العنوان من ال stack وتسمى هذه العملية popping

وال stack له دور اخر مهم وهو تخزين المتغيرات الخاصة بكل دالة

فمثلا اذا كانت الدالة main لدينا متغيرات خاصة x , y كما يلي

```
int main()
{
  // automatic variables
  auto double x, y;

  x = 10;
  y = 20;

  // leave main and go to func1()
  func1();

  // x and y values shall be reserved for later
  int sum = x + y;
}
```

فيجب ان تظل قيمة المتغيرات محفوظة حتى يتم الرجوع من تنفيذ الدالة func1 ومن ثم اتمام عملية الجمع فى النهاية وعند الانتهاء من تنفيذ الدالة main ومغادرتها فيتم ازالة جميع المتغيرات الخاصة من ال stack حيث لم تعد مطلوبة.

الدوال من النوع Inline

من الرائع جدا تقسيم البرامج الي دوال واستعداء بعضها البعض فهذه الطريقة التنظيمية تجعل البرنامج سهل القراءة وكذلك يمكنك استخدام دوال محددة في اكثر من مكان ولكن المشكلة مع كثرة الدوال والمتغيرات التي تمرر لها كذلك القيمة التي يتم ارجاعها قد تسبب في حمل زائد على المعالج ولذلك مع الدوال الصغيرة جدا التي تكون غالبا سطر او سطرين يتم تعريفها من النوع Inline حيث يقوم ال compiler بعمل نسخه منها فى المكان الذى يتم استعداءها فيه

وبذلك يتم تقليل الوقت اللازم لاستعداء الدالة والرجوع منها مرة اخرى والمشكلة هنا اذا كانت الدالة من النوع Inline كبيرة ومستخدمة بكثرة فسيكون هناك الكثير من النسخ وبالتالي زيادة حجم البرنامج.

مثال

```
#include <iostream>
using std::cout;
using std::cin;
using std::endl;

inline static double sum(const double number01, const
double number02)
{
  return number01 + number02;
}

int main()
{
  double number01;
  double number02;

  cout << "Enter the value of the first number: ";
  cin >> number01;

  cout << "Enter the value of the second number: ";
  cin >> number02;

  cout << "The sum value is " << sum(number01, number02)
<< endl;
  return 0;
}
```

وستكون نتيجة تنفيذ البرنامج كما يلي

```
Enter the value of the first number: 2.5
Enter the value of the second number: 3.7
The sum value is 6.2
```

الدالة sum تم تعريفها من النوع inline ونلاحظ هنا اننا عرفنا المتغيرات التي سيتم تمرريها للدالة كثوابت وهذا كنوع من انواع الحماية للبرنامج من الاخطاء فالدالة sum المطلوب منها حساب مجموع

الرقمين بدون تغيير قيمتها فاذا حاولنا تغيير قيمة المتغيرات فسنحصل على الخطأ الاتي

02.png
1644x536 121 KB

تمرير المتغيرات للدوال

يتم تمرير المتغيرات للدوال عن طريق التمرير بقيمة المتغير pass-by-value او عن طريق مرجعه pass-by-reference

فى حالة تمرير بالقيمة فانه يتم عمل نسخة من هذا المتغير وتمريرها للدالة ولذلك فاي تغيرات لهذا المتغير تتم داخل الدالة لا تنعكس على الدالة الاصلية التي تم التمرير منها وهذه الطريقة يمكن استخدامها لتجنب الاثار الجانبية اثناء تمرير المتغيرات

مثال على التمرير بالقيمة

```
#include <iostream>
using std::cout;
using std::endl;

void num(int number)
{
```

```
  number = number + 10;
  cout <<"Number after change in the function is : "<<
number <<endl;
}

int main()
{
  int number = 5;

  // call a function and pass parameter by value
  num(number);

  cout << "Number after return to main function : "<<
number <<endl;
  return 0;
}
```

ونتيجة التنفيذ كما يلي

```
Number after change in the function is : 15
Number after return to main function : 5
```

نلاحظ هنا ان المتغير number قد تمرير للدالة num التي قامت بتغير قيمته ولكن هذا التغيير لم ينعكس داخل الدالة main حيث ظلت قيمته كما هي لم تتغير.

والعيب الاساسي فى طريقة التمرير بالقيمة عندما يكون حجم المتغير كبير كمثلا مجموعة متسلسلة من البيانات او المصفوفات فقد تستغرق عمليه النسخ وقت اطول.

والنوع الثاني هو التمرير بالمرجع حيث يتم تمرير مرجع المتغير وله نفس عنوان المتغير الاصلي وبالتالي ان تغيرات تحدث داخل الدالة سوف تنعكس على الدالة الاصلية ومن اهم مميزات التمرير بالمرجع انه اسرع من التمرير بالقيمة حيث لا يوجد هنا عمل نسخة جديدة من المتغيرات وانما فقط ارسال مرجعها للدالة.

والان لاستخراج مرجع اى متغير نقوم باستخدام الرمز &.

مثال على التمرير بالمرجع

```
#include <iostream>
using std::cout;
using std::endl;

int main()
{
  int number = 5;

  // to get the reference of number.
  int & numberRef = number;

  cout << "number is  : "<< number <<endl;
  cout << "numberRef is : "<< numberRef <<endl;

  // numberRef and number are the same, they save the
same
  // reference, any change on one will be reflected on
the other.
  number = 10;
  cout << "numberRef is : "<< numberRef <<endl;

  return 0;
}
```

ونتيجة التنفيذ ستكون كما يلي

```
number is  : 5
numberRef is : 5
numberRef is : 10
```

نلاحظ هنا اننا عرفنا المرجع للمتغير باستخدام number باستخدام الامر الاتي

```
int & numberRef = number;
```

والان number و numberRef هما متغيران لهم مرجع واحد فأي تغيير على احداهما سوف ينعكس على الاخر كما هو واضح من نتيجة التنفيذ والان بعد ان قمنا بتعريف المرجع سوف سنقوم الان بتمريره للدالة ومحاولة ايجاد الفرق بين التمرير بالمرجع والتمرير بالقيمة

```
#include <iostream>
using std::cout;
using std::endl;
```

```
void num(int & numberRef)
{
  numberRef = numberRef + 10;
  cout <<"The reference of number is : "<< numberRef
<<endl;
}

int main()
{
  int number = 5;

  // call a function and pass parameter by reference
  num(number);

  cout << "Number after return to main function : "<<
number <<endl;

  return 0;
}
```

ونتيجة التنفيذ ستكون كما يلي

```
The reference of number is : 15
Number after return to main function : 15
```

نلاحظ هنا تم تمرير المرجع عن طريق int & number الموجود فى تعريف الدالة فالمتغير number الموجود داخل الدالة num له نفس عنوان المتغير number الموجود داخل الدالة main ولذلك اى تغيير يتم على احدهما سوف ينعكس على الاخر اى انهما كشخص واحد ينظر للمرآه.

المتغيرات الاساسية للدوال

يمكن فى لغة السي بلس بلس استعداء الدوال بدون تمرير جميع المتغيرات ولذلك اذا اردنا استخدام هذه الخاصية فيجب علينا تعريف المتغيرات الاساسية للدوال.

مثال

```
#include <iostream>
using std::cout;
```

```
using std::endl;

// function prototype
int calculateSize(int length = 1, int width = 1, int
height = 1);

int main()
{
  int size;

  // call a function without parameters
  size = calculateSize();
  cout << "size is  : "<< size <<endl;

  size = calculateSize(10);
  cout << "size is  : "<< size <<endl;

  size = calculateSize(10, 10);
  cout << "size is  : "<< size <<endl;

  return 0;
}

// function to calculate the size
int calculateSize(int length, int width, int height)
{
  return length * width * height;
}
```

ونتيجة التنفيذ ستكون كما يلي

```
size is  : 1
size is  : 10
size is  : 100
```

نلاحظ اننا قمنا بتعريف نموذج الدالة مع المتغيرات الاساسية كالاتي

```
int calculateSize(int length = 1, int width = 1, int
height = 1);
```

ففي حالة استعداء الدالة بدون اى تمرير اى متغيرات فسيتم استخدام المتغيرات الاساسية كالاتي

```
size = calculateSize();
```

واذا اردنا تمرير احد المتغيرات فيجب علينا تمرير المتغير على اليسار اولا وتباعا حتى نصل لاخر متغير ففي المثال السابق لا يمكن تمرير العرض width وعدم تمرير الطول length ولكن العكس صحيح وكذلك يمكننا تمرير الطول والعرض وعدم تمرير الارتفاع كما قمنا فى اخر خطوة

```
size = calculateSize(10, 10);
```

تحميل الدوال

يمكن في لغة السي بلس بلس تعريف العديد من الدوال بنفس الاسم مع الاختلاف فى المتغيرات التي تتلقاها الدالة وتسمى هذه الخاصية function overloading

مثال

في المثال الاتي سنقوم بتعريف الدالة sum لحساب المجموع مرة باستخدام متغيرات من النوع int ومرة اخرى بمتغيرات من النوع double كما يلي

```
#include <iostream>
using std::cout;
using std::endl;

// overloaded function
int sum(int number01, int number02);
double sum(double number01, double number02);

int main()
{

  cout << "The sum value is " << sum(4, 5) << endl;
  cout << "The sum value is " << sum(4.5, 5.5) << endl;

  return 0;
}

int sum(int number01, int number02)
{
  return number01 + number02;
```

```
}

double sum(double number01, double number02)
{
  return number01 + number02;
}
```

وتكون نتيجة التنفيذ كما يلي

```
The sum value is 9
The sum value is 10
```

حيث تم استعداء الدالة sum مرة بمتغيرات من النوع int ومرة بمتغيرات من النوع double

قوالب الدوال function templates

استخدام قوالب الدوال هي طريقة لتعريف الدالة مرة واحد اذا اردنا استخدام متغيرات من انواع مختلفة كما هو الحال فى تحميل الدوال نقوم بتعريف الدالة اكثر من مرة اما عند استخدامنا لقوالب الدوال فنقوم بتعريفها مرة واحدة فقط ويمكننا استعدائها بمتغيرات من انواع مختلفة

لتعريف القوالب يتم عن طريق استخدام الكلمة template يتبعها قائمة المتغيرات المطلوبة لهذا القالب

مثال

سنعيد صياغة المثال السابق ولكن مع استخدام قوالب الدوال كما يلي

```
#include <iostream>
using std::cout;
using std::endl;

// function template
template < class T >
T sum( T number01, T number02)
{
  return number01 + number02;
}

int main()
```

```
{
  cout << "The sum value is " << sum(4, 5) << endl;
  cout << "The sum value is " << sum(4.6, 5.5) << endl;

  return 0;
}
```

ونتيجة التنفيذ ستكون كما يلي

```
The sum value is 9
The sum value is 10.1
```

نلاحظ اننا قمنا بتعريف القالب كالاتي

```
template < class T >
```

وهذا القالب من النوع T وهو نوع غير محدد ويتم تحديده علي حسب المتغيرات التي يتم استعداء الدالة بها فمثلا اذا قمنا باستعداء الدالة sum بمتغيرات من النوع int فان نوع القالب T سيكون int ولذلك فلا داعي لاعادة تعريف الدالة اكثر من مرة كما هو الحال فى تحميل الدوال

يمكننا ايضا تعريف القوالب كما يلي

```
template <typename anyName>
```

حيث استخدامنا typename بدلا من class ويمكننا ايضا استبدال T باى اسم نريد

اعادة استخدام الدوال Recursion

وهذه جعل أى دالة تستدعي نفسها، يتم استخدام Recursion في حل بعض المسائل الحسابية المعقدة او التي تتطلب خطوات معينة مماثلة تتم كل مرة مثل المتتابعات الحسابية او كمثال بسيط يمكننا استخدام اعادة استخدام الدوال هو حساب المضروب factorial مثلا حساب المضروب لاي عدد غير سالب n يتم باستخدام المعادلة الاتية

```
n * ( n - 1) * ( n - 2) * ... * 1
```

فمثلا مضروب الرقم ٥ هو حاصل ضرب ٥ و ٤ و ٣ و ٢ و ١

ونلاحظ هنا اننا قمنا بالاتية قمنا بطرح ١ من الرقم خمسة وضرباه في نفس حتي وصلنا للواحد ولذلك ففي كل خطوة نقوم بعملية طرح وضرب حتي نصل للواحد فى النهاية

```
#include <iostream>
using std::cout;
using std::cin;
using std::endl;

// recursive function
int factorial(int number)
{
  // factorial of 1 is 1
  if ( number <= 1 )
    return 1;
  else
    return number * factorial( number - 1 );
}

int main()
{
  int number;
  cout << "Enter number: ";
  cin >> number;

  cout << "The factorial is " << factorial(number) <<
endl;

  return 0;
}
```

ونتيجة التنفيذ ستكون كما يلي

```
Enter number: 5
The factorial is 120
```

نلاحظ فى المثال السابق ان الدالة factorial قامت باستعداء نفسها كل مرة مع طرح ١ من الرقم حتي يصل الرقم الي ١ وعندما يتم انهاء الدالة. ويجب دائما مراعاة انهاء استعداء الدالة عند نقطة معينة والا

سيتم تنفيذ البرنامج بصورة لا نهائية وسيؤدي الي استهلاك كبير للذاكرة حيث يتوقف عن العمل عند استهلاك كل الذاكرة المتاحة.
وإلى اللقاء في الدرس القادم