Functions الدوال

1. Introducing enum
2. Storage Classes
3. Scope Rules
4. Recursion
5. Example Using Recursion: Fibonacci Series
6. Recursion vs. Iteration

العودة من التابع Returning from a function

هنالك طريقتان للعودة من التابع لينهي التابع التنفيذ ويعود إلى المستدعي له

آ- **الطريقة الأولى** : وهي أن ينفذ التابع كل تعليماته حتى يصل إلى نهايته (قوس نهاية جسم التابع }) .

الطريقة الثانية: هي أن معظم returnلإيقاف حيث أن التوابع تعتمد على تعليمة، - التنفيذ التعليمة تنهي التابع وترجع قيمة للبرنامج المستدعي، ويمكن كذلك أن تستخدم لتبسيط عمل البرنامج وزيادة فعاليته بالسماح لأكثر من نقطة خروج فيه.

تبادل المعطيات بين التابع المنادى و التابع المنادي :

يتم تبادل المعطيات بين التابع المنادى و التابع المنادي من خلال المتغيرات ولهذه المتغيرات أنواع عدة :

المتغيرات المحلية variable local

وهي ببساطة المتغيرات التي يعلن عنها داخل التابع وتكون موجودة فقط أثناء تنفيذ التابع، حيث تستخدم في تعابير برمجية تخصص لمتغيرات الشكلية، أو تعاد بواسطتها قيمة إلى التابع المنادي من خلال تعليمة

المتغيرات الشكلية (ariable formal)

وهي المتغيرات التي يتضمنها الإعلان عن التابع بمعنى أنها تكون موجودة في قائمة معاملات التابع .

لمتغيرات الفعليةArguments Variable

وهي المتغيرات التي تستخدم في نداء التابع

استدعاء التابع

call of function

يمكن أن يستدعى التابع بطريقتين:

1-استدعاء التابع بدون تمرير للمتغيرات الشكلية(الوسطاء).

2 -استدعاء التابع مع تمرير للمتغيرات الشكلية، وهنا يمكن أن نميز أشكال عدة:

ا- استدعاء التابع بالقيمة.

ب- استدعاء التابع بالمرجع.

ت- استدعاء التابع بالمصفوفة وحيدة البعد (النسق أو الصف).

ث- استدعاء التابع بالمصفوفة متعددة الأبعاد.

ج- استدعاء التابع بالمؤشر

نموذج الدالة Function Prototype

عندما نحتاج لاستدعاء دالة ما فإنه يحتاج إلى معرفة اسم الدالة وعدد وسيطاتها وأنواعها ونوع قيمة الإعادة

تمرير الوسيطات Passing Parameters

يمكن تصنيف البارامترات بالأساس الى مجموعتين **الفعلية والرسمية**

الوسيطة الفعلية مضمن في (arguments actual:) هي متغير أو تعبير استدعاء دالة الذي يحل محل المعامل الرسمية التي هو جزء من إعلان الدالة.

الوسائط الرسمية في تعريف (: arguments formal) هي البارامترات الموجودة الدالة والتي يمكن أيضا أن تسمى بالبارامترات الوهمية أو محدودة المتغيرات.

هناك طريقتان رئيسيتان لتمرير البارامترات أو الوسائط إلى البرنامج :-

1- **التمرير بالقيمة** (passing by value)

2- **التمرير بالمرجع-** (passing by reference)

تمرير بالقيمة passing by value

عندما يتم تمرير البارامترات حسب القيمة، يتم أخذ القيمة نسخة من المعاملات من دالة الاستدعاء وتمريرها إلى الدالة المطلوبة. لن تتغير المتغيرات الأصلية داخل دالة الاستدعاء ، بغض النظر عن التغييرات التي تجريها الدالة عليها.

تمرير بالمرجع passing by reference

عندما يتم تمرير البارامترات بالمرج يتم نسخ عناوينهم إلى الوسيطات المقابلة في الدالة المدعومة، بدلا من نسخ قيمها. وبالتالي، عادة ما تستخدم المؤشرات في قائمة وسيطات الدوال لتلقي المراجع التي تم تمريرها.

ولإجبار تمرير الوسيطة بالمرجع نضيف الحرف & إلى نوع بيانات الوسيطة في تعريف الدالة وتصريح الدالة

هناك طريقتان لتمرير الوسيطات إلى الدوال في العديد من لغات البرمجة وهما
التمرير بالقيمة pass-by-value
والتمرير بالمرجع . pass-by-reference

عندما يتم تمرير الوسيطة بالقيمة ، يتم انشاء نسخة من قيمة الوسيطة وتمريرها (على مكدس استدعاء الدالة) إلى الدالة التي يتم استدعاؤها.
لا تؤثر التغييرات التي يتم إجراؤها على النسخة على قيمة المتغير الأصلي في المتصل.
هذا يمنع الآثار الجانبية العرضية التي يمكن أن تعيق بشكل كبير تطوير أنظمة برامج صحيحة وموثوقة.
أحد عيوب تمرير القيمة هو أنه إذا تم تمرير عنصر بيانات كبير ، فإن نسخ تلك البيانات يمكن أن يستغرق قدرًا كبيرًا من زمن التنفيذ ومساحة الذاكرة.

التمرير بالمرجع pass-by-reference .
يعد التمرير بالمرجع أمرًا جيدًا لأسباب تتعلق بالأداء ، لأنه يمكن أن يحل مشكلة عبء التمرير لنسخ كميات كبيرة من البيانات

يمكن أن يؤدي التمرير بالمرجع إلى إضعاف الأمان يمكن أن تؤدي الدالة التي تم استدعاؤها إلى إتلاف بيانات المتصل.

استدعاء التابع بالقيمة
Call by value

ستخدم طريقة الاستدعاء بالقيمة لتمرير المتغيرات الفعلية إلى التابع

```
#include<iostream.h>
int sqr(int x)
{
 x=x*x;
 return x;
}
void main()
{
 int t=10;
cout<<t<<" power 2 ="<<sqr(t)<<endl;
}
```

خرج البرنامج 10 power 2 =100

ـاستدعاء التابع بالمرجع
call by reference

يعلن عن المتغير الشكلي بانة مرجع للمتغير الفعلي المتغير والمرجع لهما نفس العنوان في الذاكرة وأي تغيير في أحدهما يغير الأخر وبالتالي سيكون لهما نفس القيمة

ويتم ذلك ببساطة بإلحاق نوع المتغير الشكلي بـ(&)

مثال:
البرنامج التالي يستخدم لإيجاد عاملي الأعداد من 0 حتى 5

```
#include<iostream.h>
int factorial(int n)
      {
       int f=1;
       if(n<0)
            cout<<"no factorial."<<endl;
             while(n>1)
             f*=n--;
      return f;
       }
Int main()
{
       for(int i=0;i<6;i++)
       cout<<i<<"!="<<factorial(i)<<endl;
      Return(0)
}
```

خرج البرنامج:

```
0!=0
1!=1
2!=2
3!=6
4!=24
5!=1201
```

في هذا البرنامج نلاحظ أن المتغير n الذي أعلن عنه في قائمة معاملات التابع factorial ماهو إلا متغير شكلي يستقبل قيمة المتغير الفعلي i الذي يرسل إليه من التابع ( ) main من خلال النداء (i) factorial ، أما المتغير f فما هو إلا متغير محلي أعلن عنه داخل التابع factorial، وأعيدت قيمته إلى البرنامج المنادي من خلال تعليمة return .

استدعاء التابع بالمرجع
Call by referance

من الناحية المفاهيمية ، تحصل الوسيطة الفعلية المشار إليها على اسم جديد في الدالة: كل ما يتم إجراؤه على الوسيطة الوهمية يتم أيضًا على الوسيطة الفعلية

نظرًا لأن المرجع إلى المتغير يتم التعامل معه تمامًا مثل المتغير نفسه ، فإن أي تغييرات يتم إجراؤها على المرجع يتم تمريرها إلى الوسيطة.

بما أن التابع استدعي بالمرجع حيث أعلن عن x بأنه مرجع لـ t . فإن المتغير x ينسخ من المتغير الفعلي t .

وبعد تنفيذ التابع sqr تصبح القيمة الجديدة لـ x مساوية 100 ويأخذ المتغير t هذه القيمة.


```
#include<iostream.h>
void sqr(int &x)
{
        x=x*x;
}
void main()
{
        int t=10;
        cout<< "local t in main befor calling sqr is: ";
        cout<<t<<endl;
       cout<<t<<" power 2 =“<< sqr(t);

       cout<<t<<endl;
       cout<< "local t in main after calling sqr is: ";
        cout<<t<<endl;
}
```

خرج البرنامج:

```
local t in main befor calling sqr is: 10
10 power 2 =100
local t in main after calling sqr is: 100
```

استدعاء التوابع بالأنساق
call with one dimension array

```
#include<iostream.h>
void display(int t[],int n)
{
        int i;
       cout<<"t array is:";
        for(i=0;i<n;i++)
       cout<<t[i]<<" ";
        cout<<endl;
}
void main()
{
       const int n=10;
        int t[n];
        for(int i=0;i<n;i++)
        t[i]=i;
        display(t,n);
}
```

ويستطيع التابع تغيير محتويات عناصر الصف بالاتصال المباشر بأماكن الذاكرة المحددة لهذه العناصر ، وبالتالي وبالرغم من أن اسم الصف تم إرساله كعنوان فإن عناصر هذا الصف يمكن تغيير قيمها كما لو مررت بمرجع

مثال اكتب برنامج يستخدم التوابع للبحث عن عنصر في مصفوفة أعداد صحيحة معرفة في التابع الرئيسي

a [10] = { 18, 25, 36, 44, 12, 60, 75, 89, 10, 50 }

write C++ program, using function, to find (search) X value in array, and return the index of it's location?

```
#include<iostream.h>
int search( int a[ ], int y)
{
        int i= 0;
        while ( a [ i ] != y )
        i++;
        return ( i );
}
void main ( )
{
        int X, f;
        int a [ 10 ] = { 18, 25, 36, 44, 12, 60, 75, 89, 10, 50 };
        cout << "enter value to find it: ";
        cin >> X;
        f= search (a, X);
        cout << "the value " << X << " is found in location "<< f;
}
```

استدعاء التابع بالمصفوفة ثنائية البعد :
call with tow dimension array

عند استدعاء التابع بالمصفوفة ثنائية البعد فإن البعد الأول للمصفوفة لا يتم تحديده في قائمة معاملات التابع في حين يتم تحديد البعد الثاني

مثال : هذا البرنامج يسمح بإدخال وإخراج مصفوفة ثنائية البعد مؤلفة من 3 أسطر وخمسة أعمدة.

```
#include<iostream.h>
void read(int a[][5])
{
   cout<<"enter 15 number,5 per row:"<<endl;
      for(int i=0;i<3;i++)
        {
           cout<<"row "<<i<<" : ";
         for(int j=0;j<5;j++)
            cin>>a[i][j];
         }
}
```

```
void print(int a[][5])
{
   cout<<"a array is:"<<endl;
   for(int i=0;i<3;i++)
   {
         for(int j=0;j<5;j++)
              cout<<" "<<a[i][j];
    cout<<endl;
    }
}
```

```
int  main()
{
  int a[3][5];

    read(a);
    print(a);
Teturn (0);

}
```

خرج البرنامج:

```
enter 15 number,5 per row:
row 0 : 11 22 44 55 66
row 1 : 12 13 14 15 16
row 2 : 14 16 18 19 20
a array is:
11 22 44 55 66
12 13 14 15 16
14 16 18 19 20
```

```
#include <iostream>
using namespace std;
void myFunction(); // prototype
int x = 5, y = 7; // global variables
int main()
{
cout << "x from main:"  << x << endl;
cout << "y from main:"  << y << endl << endl;
myFunction();
cout << "Back from myFunction!" << endl << endl;
cout << "x from main: " << x << endl;
cout << "y from main: " << y << endl;
return 0;
}
 void myFunction()
{
int y = 10;
 cout << "x from myFunction: " << x << endl;
cout << "y from myFunction: " << y << endl << endl;
}
```

المعاملات هي متغيرات محلية Parameters Are Local Variables

الوسيطات التي تم تمريرها إلى الدالة محلية بالنسبة للدالة.
لا تؤثر التغييرات التي تم إجراؤها على الوسائط على القيم في دالة الاستدعاء.
يُعرف هذا بالتمرير بالقيمة ، مما يعني أنه يتم عمل نسخة محلية من كل وسيطة في الدالة . يتم التعامل مع هذه النسخ المحلية مثل أي متغيرات محلية أخرى.

المتغيرات العمومية Global Variables

المتغيرات المعرفة خارج أي دالة لها نطاق عمومي ، وبالتالي فهي متاحة من أي دالة في البرنامج ، بما في ذلك main (). المتغيرات المحلية التي تحمل نفس الاسم مثل المتغيرات العمومية لا تغير المتغيرات العمومية.
المتغير المحلي الذي يحمل نفس اسم المتغيرالعمومي يخفي المتغير العام. إذا كانت الدالة تحتوي على متغير يحمل نفس اسم المتغير العمومي ، فإن الاسم يشير إلى المتغير المحلي - وليس المتغير العام - عند استخدامه داخل الدالة .

Default Parameters القيمة الافتراضية

```
#include <iostream>
using namespace std;
 int AreaCube(int length, int width = 25, int height = 1);
 int main()
 {
 int length = 100;
 int width = 50;
 int height = 2;
 int area;
 area = AreaCube(length, width, height);
 cout << "First area equals: " << area << "\n";
  area = AreaCube(length, width);
 cout << "Second time area equals: " << area << "\n";
  area = AreaCube(length);
cout << "Third time area equals: " << area << "\n";
return 0;
 }
  AreaCube(int length, int width, int height)
 {
 return (length * width * height);
 }
```

```
First area equals: 10000
Second time area equals: 5000
Third time area equals: 2500
```

1. لا تحاول إنشاء قيمة افتراضية للمعامل الأولى في حالة عدم وجود قيمة افتراضية للمرة الثانية. 2. لا تنسى أن معامل التمرير بالقيمة لا يمكن أن تؤثر على المتغيرات في دالة الاتصال.	1. تذكر أن معاملات الدالة تعمل كمتغيرات محلية داخل الدالة. 2. تذكر أن التغييرات في المتغير الشامل في دالة واحدة تغير هذا المتغير لجميع الوظائف.

دوال التحميل الزائد Overloading Functions

يمكّن في لغة C ++ من إنشاء أكثر من وظيفة بنفس الاسم. وهذا ما يسمى وظيفة التحميل الزائد. يجب أن تختلف الدوال في قائمة المعاملات الخاصة بها و بنوع مختلف من المعاملات ، أو عدد مختلف من المعاملات، أو كليهما. هذا مثال:

```
int myFunction (int, int);
int myFunction (long, long);
int myFunction (long);
```

تم تحميل myFunction () بثلاث قوائم معاملات.
يختلف الإصداران الأول والثاني في أنواع المعاملات ، ويختلف الإصدار الثالث في عدد المعاملات.
يمكن أن تكون أنواع الإرجاع هي نفسها أو مختلفة في الدوال المحملة بشكل زائد.

يشير تعدد الأشكال الوظيفي إلى القدرة على "التحميل الزائد" لوظيفة لها أكثر من معنى.
من خلال تغيير رقم أو نوع المعاملات ، يمكنك إعطاء دالتين أو أكثر نفس اسم الدالة ، وسيتم استدعاء الدالة الصحيحة تلقائيًا عن طريق مطابقة المعاملات المستخدمة.

```
int square(int value) {
 return (value * value);
 }

We also want to square floating point numbers:

 float square(float value) {
 return (value * value);
 }
```

دوال التحميل الزائد Overloading Functions

مثال اكتب برنامج يستخدم تابع التحميل الزائد لضرب عددين صحيحين ويعيد قيمة صحيحة وتابع ضرب لضرب عد حقيقي بعدد صحيح ويعيد قيمة حقيقية

```
#include <iostream>
using namespace std;
int mul(int, int);
float mul(float, int);

int mul(int a, int b) { return a * b; }
float mul(double x, int y) { return x * y; }
int main()
{
   int r1 = mul(6, 7);
   float r2 = mul(0.2, 3);
   cout << "r1 is : " << r1 << endl;
   cout << "r2 is : " << r2 << endl;
   return 0;
}
```

Output

```
r1 is : 42
 r2 is : 0.6
```

المحاضرة السادسة توابع العودية

دكندة أبوقاسم

الإعادة الذاتية Recursion

يمكن للتوابع في معظم اللغات C++ أن تستدعي نفسها، ويكون التابع ومنها لغة معيد لنفسه إذا كانت إحدى التعليمات المستخدمة في جسم التابع هي تعليمة استدعاء له.

السماح بعملية المعاودة يتطلب أن يكون التابع قادراً على معاودة نفسه ويجب أن يتحقق ما يلي ليمكن استخدام العودية في التنفيذ.

1 -قيمة بدائية للتابع هي قيمة أو قيم بدائية للتابع تحدد الناتج بدون نداء التابع. anchor An

2 -الخطوة الدورية وهي القيمة التي يأخذها التابع لكي ينتقل من قيمة إلى القيمة step recessive or inductive An المجاورة لها وباتجاه القيمة أو القيم البدائية

المثال يوضح التابع الذي يحسب العاملي لعدد ما صحيح والذي يساوي جداء الأعداد الصحيحة من الواحد وحتى العدد المرغوب حساب العاملية له. القيمة البدائية للعاملية والخطوة على التوالي هي :

a) 0!=1
b) for n>0 , n!= n*(n-1)!
N!=1*2*3 … (n-1)*n

وهنا نجد أن حساب 3! يحتاج لحساب 2! وهذه تحتاج لحساب 1! وهذه تحتاج لحساب 0! وهي المعرفة سلفًا ومن ثم التعويض

```
#include <iostream.h>
#include <iomanip.h>
unsigned long factorial(unsigned long number)
{
        if (number <= 1)
            return 1;
        else

            return number * factorial(number - 1);
}
 void main()
{
      for (int i = 0; i <= 10; i++)
          cout << setw(2) << i << "! = " << factorial(i)
<< endl;
}
```

جَامعة المَنارة
MANARA UNIVERSITY

خرج البرنامج:

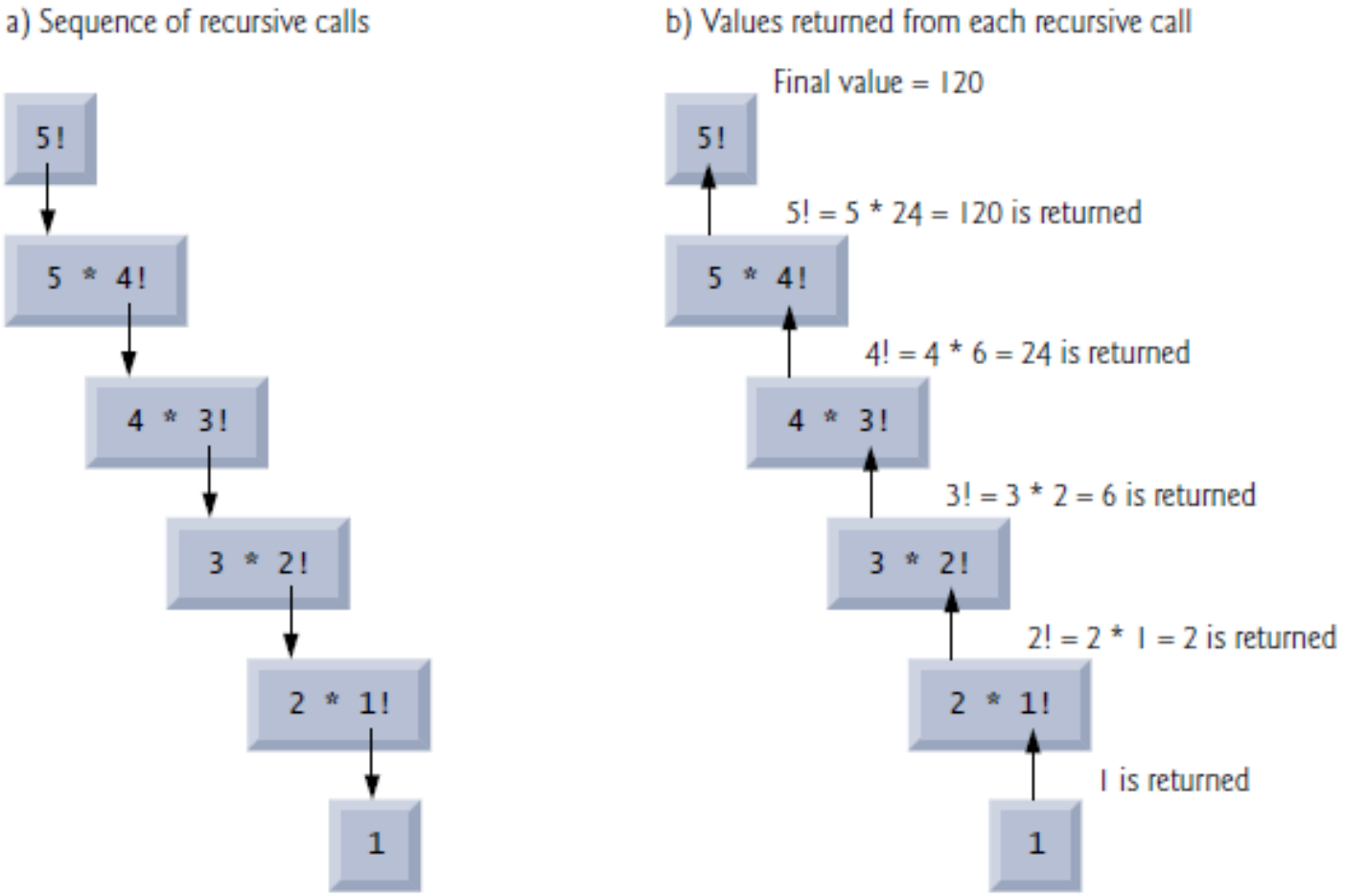
```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
```

في هذا البرنامج عندما يتم استدعاء التابع بمتغير فعلي يساوي الصفر فإنه يرجع قيمة مساوية للواحد، وفي حال استدعي بمعامل لا يساوي الواحد فهو سيرجع الجداء

Recursively Calculating Factorials

The factorial of a nonnegative integer n, written n! (pronounced “n factorial”), is the product

$n \cdot (n - 1) \cdot (n - 2) \cdot \ldots \cdot 1$



عندما يستدعي التابع نفسه ، يتم تخصيص أماكن جديدة في المكدس للمتغيرات والمتحولات المحلية الخاصة به، ويتم تنفيذ التابع على هذه المتغيرات الجديدة من البداية، إن استدعاءات ذاتية للتابع لا تصنع نسخة جديدة منه، فقط نسخة جديدة لمتغيراته الحقيقية. وعند العودة من كل استدعاء ذاتي يتم إزالة المتغيرات القديمة من المكدس ويتابع التابع تنفيذه من النقطة التي استدعي منها داخل التابع

قد تكون التوابع المعاودة لذاتها أبطأ عند التنفيذ من التوابع التي تعتمد على التكرارية وأحيانًا قد تسبب فائض في المكدس (حجم المتغيرات الناتجة عن المعاودة أكبر من حجم المكدس) وهذا يدعى overflow وتعد الفائدة الرئيسية من استخدام stack

عند كتابة التوابع المعاودة function recursive يجب استخدام تعليمة if في مكان ما من التابع لإجبار التابع على العودة بدون تنفيذ استدعاء معاود. وإلا فإنه بمجرد استدعاء التابع لن يعود أبداً.

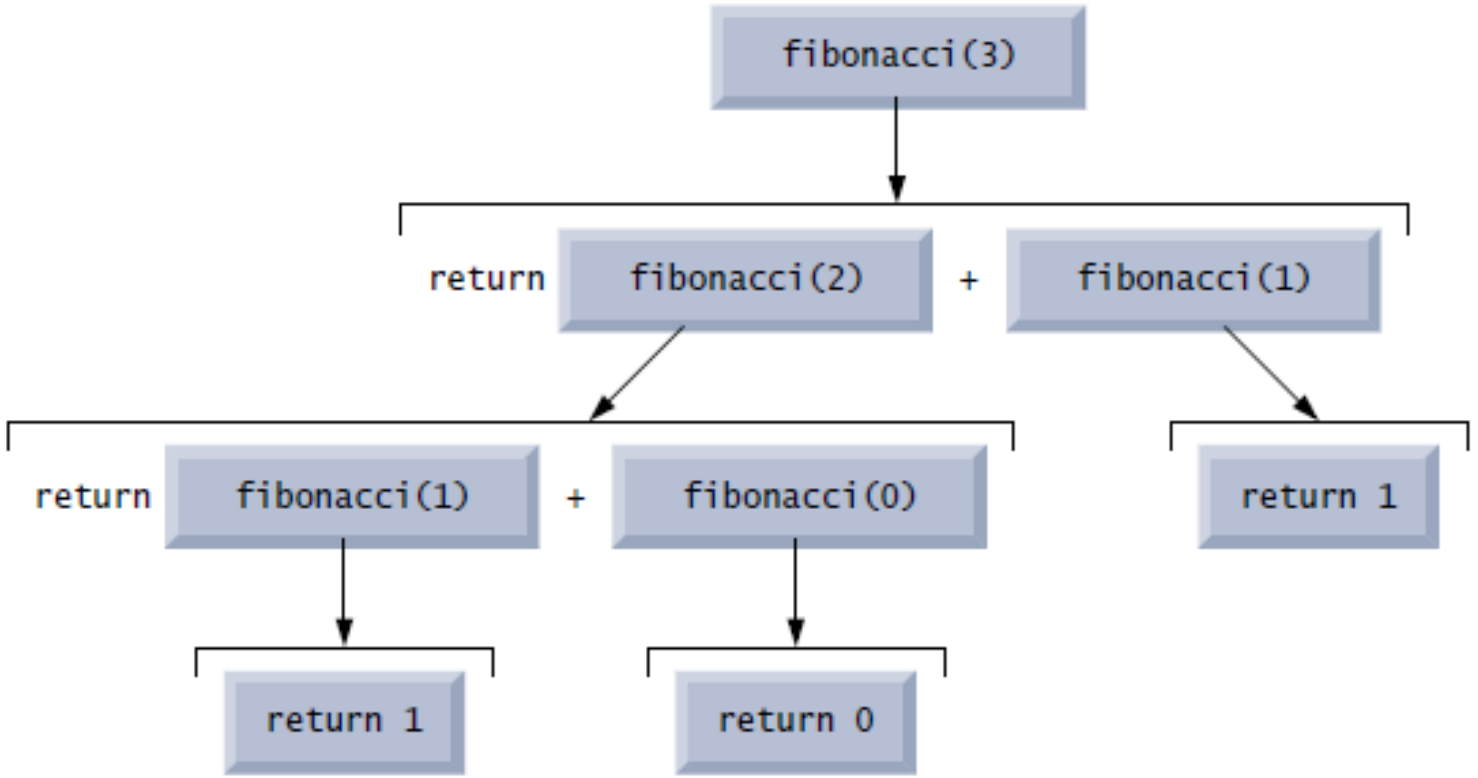
الفيم البدائية للتابع هي
fibonacci(1)=1 , fibonacci(0)=0

والخطوة أن العدد التالي يساوي مجموع العدديين السابقين، وبالتالي نجد:

مثال آخر عن التوابع العودية، سلسلة Fibonacci والتي تبدأ من الأعداد التالية:

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

وتبدأ من القيمتين 1, 0 ويتكون كل عدد من الأعداد التالية من مجموع العدديين السابقين لهما، وأهم تمثيل لهذه السلسلة هي الأشكال الحلزونية، وكذلك تقارب نسبة هي النسبة العدد للعدد السابق له ... والتي نجد 1.61 وهذه النسبة الذهبية ratio golden
لها تطبيقات هامة في الطبيعة ونجدها في إطار الصور والمناظر الطبيعية


fibonacci(3)
return
fibonacci(2)
+
fibonacci(1)
return
fibonacci(1)
+
fibonacci(0)
return 1
return 1
return 0

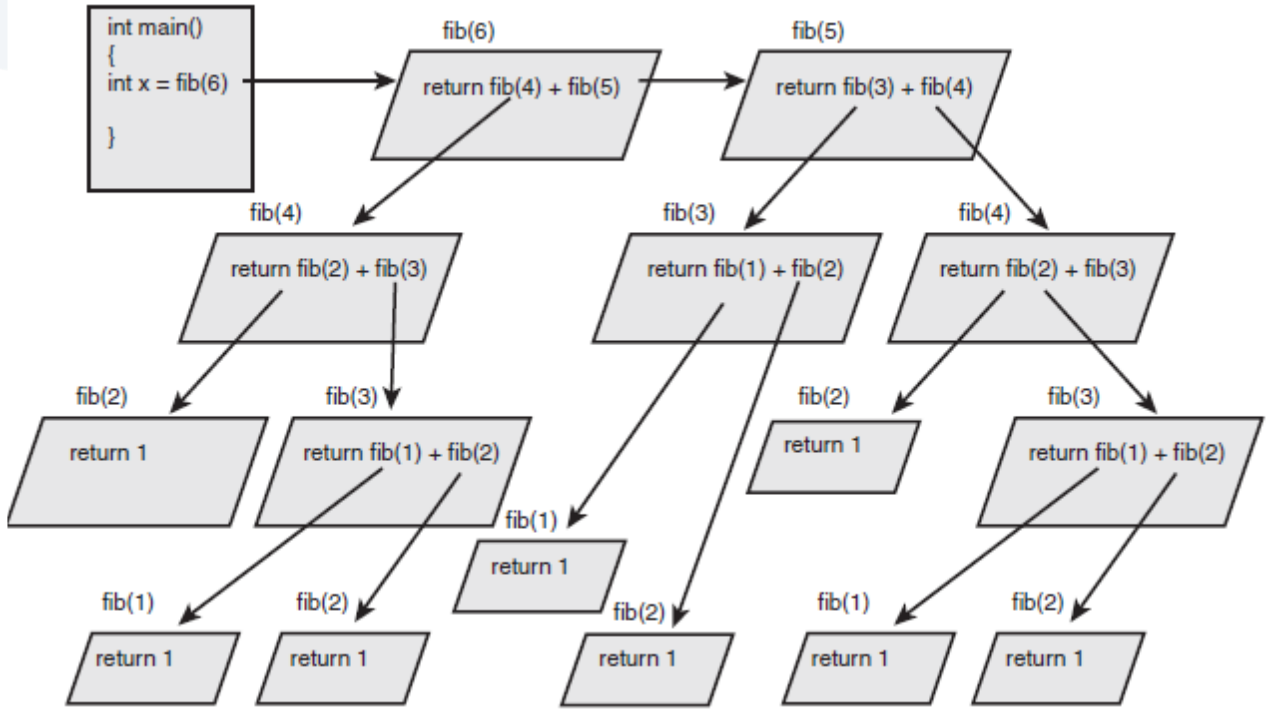
```
#include <iostream.h>
long fibonacci(long n)
{
      if (n == 0 || n == 1)
            return n;
       else
             return fibonacci(n - 1) + fibonacci(n - 2);
}
 void main()
{
long result, number;
 cout << "Enter an integer:";
 while(1)
      {
       cin >> number;
            if(number<0)
            break;
       result = fibonacci(number);
      cout <<"Fibonacci("<< number << ") ="
      << result << endl;
      }
}
```

جَامعة المَنارة
MANARA UNIVERSITY

خرج البرنامج:

```
Enter an integer:0
Fibonacci(0) =0
1
Fibonacci(1) =1
2
Fibonacci(2) =1
3
Fibonacci(3) =2
4
Fibonacci(4) =3
5
Fibonacci(5) =5
6
Fibonacci(6) =8
7
Fibonacci(7) =13
8
-1
Press any key to continue
```

العودية Recursion



يمكن للدالة أن تستدعي نفسها. هذا يسمى العودية ، ويمكن أن تكون العودية مباشرة أو غير مباشرة.

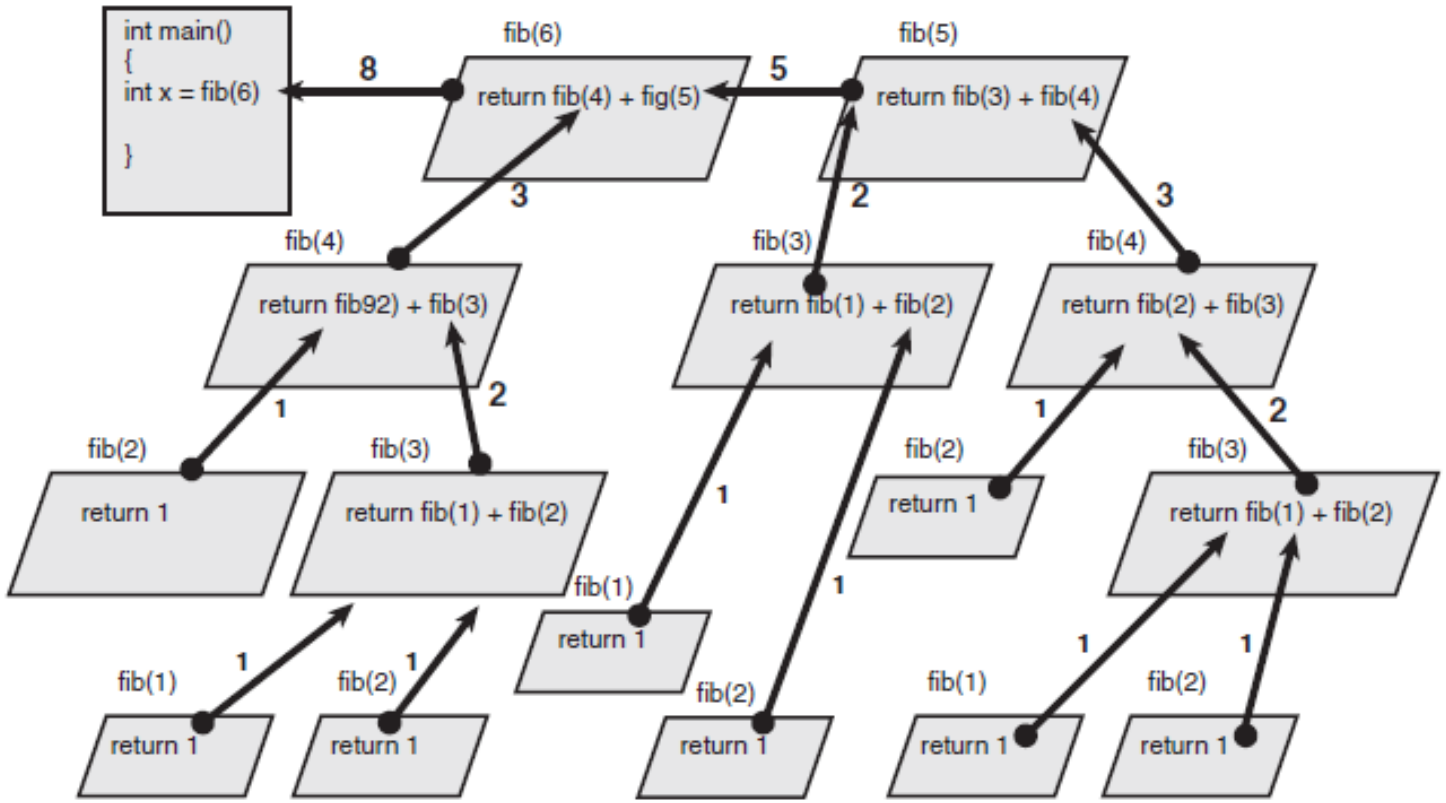
1. تكون مباشرة عندما تستدعي الوظيفة نفسها ؛
2. العودية غير المباشرة عندما تستدعي دالة وظيفة أخرى تستدعي بعد ذلك الوظيفة الأولى

الخوارزمية هي مجموعة من الخطوات التي تتبعها لحل مشكلة ما.
خوارزمية واحدة لسلسلة فيبوناتشي هي التالية

1. اطلب من المستخدم منصبًا في السلسلة:
2. استدعاء دالة fib () بهذا الموضع ، مع تمرير القيمة التي أدخلها المستخدم.

3. The fib() function examines the argument (n).
If n < 3 it returns 1; otherwise, fib() calls itself (recursively) passing in n-2. It then calls itself again passing in n-1, and returns the sum of the first call and the second

The argument n is tested to see whether it is less than 3 on line 25; if so, fib() returns
the value 1. Otherwise, it returns the sum of the values returned by calling fib() on n-2
and n-1.



ملاحظة1 :إن كلا التمرينين السابقين يمكن كتابة برنامجه دون اعتماد العودية في التنفيذ، وفي كلا الحالتين يتم التحقق من صحة شرط التوقف، حيث يتوقف التكرار عندما يصبح شرط استمرار التكرار غير صحيح، وتنتهي العودية عند الوصول إلى الحالة أو الحالات الابتدائية. وفي كلا الحالتين يمكن الوقوع في الحلقة غير المنهية عند عدم تغيير قيمة العداد حتى يختل الشرط أو عدم الوصول إلى الحالة أو الحالات الابتدائية.

ملاحظة2 :يعاني استخدام الطريقة العودية من ضياع الوقت واستخدام الذاكرة الكبير الناتج عن استدعاء التابع لنفسه حتى الوصول إلى القيمة الابتدائية والتعويض بها للوصول إلى الناتج، وبالتالي نجد أن حل المسألة دون استخدام العودية هو افضل ولكن البرامج التي تعتمد العودية في الحل هي أسهل للفهم وأوضح نظراً لمحاكاتها للواقع أكثر من التكرارية.

إذا تم تعريف أي متحول خارج نطاق أي تابع فيكون له مجال للرؤية على مستوى الملف .
ويكون مثل هذا المتحول (معروفًا) من قبل جميع التوابع انطلاقاً من النقطة التي يظهر فيها التصريح المتعلق به حتى نهاية الملف . وعلى اعتبار أن المتحولات العامة ونماذج التوابع وتعريفاتها تظهر خارج نطاق أي تابع من التوابع فيكون لها مجال للرؤية على مستوى الملف.

تعتبر العناوين Labe
هي المعرفات الوحيدة التي لها مجال للرؤية على مستوى التابع، ولا يمكن لها أن تستخدم خارج نطاق جسم هذا التابع

تتمتع المعرفات المصرح عنها ضمن كتلة ما بمجال للرؤية على مستوى هذه الكتلة، ويبدأ مجال الرؤية السابق من النقطة التي وردت فيها التصريحات عن المعروفات وحتى نهاية الكتلة المحددة بالقوس اليميني ({)

إن المتحولات والُمعرفات الوحيدة التي لها مجال للرؤية على مستوى نموذج التابع هي البارامترات التي يرد ذكرها في نموذج التابع

يبين المثال التالي مجال الرؤية المختلفة المتعلقة بالمتحولات العامة وبالمتحولات المحلية والأتوماتيكية وبالمتحولات المحلية نمط static

```
#include <iostream>
 using namespace std;
  void myFunc();
 int main()
 {
 int x = 5;
 cout << "\nIn main x is: " << x;
 myFunc();
 cout << "\nBack in main, x is:"  << x;
 return 0;
 }
 void myFunc()
 {
 int x = 8;
 cout << "\nIn myFunc, local x:"  << x << endl;
  {
cout << "\nIn block in myFunc, x is:"  << x;
 int x = 9;
 cout << "\nVery local x: " << x;
 }
  cout << "\nOut of block, in myFunc, x: " << x << endl;
 }
```

Local Variables Within Blocks

المتغيرات المحلية داخل الكتل يمكنك تحديد المتغيرات في أي مكان داخل الوظيفة ، وليس فقط في الجزء العلوي منها.
نطاق المتغير هو الكتلة التي يتم تعريفه فيها.
وبالتالي ، إذا حددت متغيرًا داخل مجموعة من الأقواس داخل الوظيفة ، هذا المتغير متاح فقط داخل تلك الكتلة.

Output

```
In main x is: 5
In myFunc, local x:8

In block in myFunc, x is:8
Very local x: 9
Out of block, in myFunc, x: 8

Back in main, x is:5
```

```
#include <iostream.h>
void a(void);    // function prototype
void b(void);    // function prototype
void c(void);    // function prototype
int x = 1;      // global variable line 1
main()
{
 int x = 5;     // local variable to main line 2
 cout << "local x in outer scope of main is " << x << endl;
 {              // start new scope line 3
 int x = 7;
 cout << "local x in inner scope of main is " << x << endl;
 }              // end new scope line 4
 cout << "local x in outer scope of main is " << x << endl;
 a();         // a has automatic local x
 b();         // b has static local x
 c();        // c uses global x
 a();        // a reinitializes automatic local x
 b();       // static local x retains its previous value
 c();       // global x also retains its value
 cout << "local x in main is " << x << endl;
 return 0;
}
void a(void)
{
int x = 25;    // initialized each time a is called line 5
cout << endl << "local x in a is " << x << " after entering a"
<< endl;
 ++x;
cout << "local x in a is " << x << " before exiting a" << endl;
}
void b(void)
{
static int x = 50;    // Static initialization only line 6
                      // first time b is called.
cout << endl << "local static x is " << x << " on entering b"
<< endl;
 ++x;
cout << "local static x is " << x << " on exiting b" << endl;
}
void c(void)
{                     // line 7
cout << endl << "global x is " << x << " on entering c" <<
endl;
x *= 10; // line 8
cout << "global x is " << x << " on exiting c" << endl;
}
```