المحاضرة الثالثة

دوال البنـاء والهدم

Constructor and Destructor Functions

دالة البنـاء:-

هي عبارة عن دالة تنفذ تلقائياً عند تشغيل البرنامج من غير أي استدعاء

أو بصورة أخرى:-

هي عبارة عن دالة يتم استدعاؤها مباشرة عند اشتقاق كائن من صنف معين وتستخدم لإعطاء خصائص الصنف قيم ابتدائية

☑	دالة البناء يجب أن تملك نفس اسم الصنف **Classs**

مثال على دالة البنـاء:-

```
#include<iostream.h>
class myclass
{
int a;
public:
my class( ); // constructor function
void show( );
};
myclass::myclass( )
{
cout<<"constructor function \n";
a=10;
}
void myclass::show( )
{
cout<<a;
}
int main( )
{
myclass ob;
ob.show( );
return 0;
}
```

دالة الهدم:-

هي عبارة عن دالة يتم استدعاؤها مباشرة عند اشتقاق كائن من صنف ولكن عند نهاية البرنامج

مثال توضيحي لدوال البناء والهدم:-

```
#include<iostream.h>
class x
{
public:
x(  );//    دالة بناء
~x(  );//    دالة هدم
};
x::x(  )
{
cout<<"constructor is called \n";
}
x::~x(  )
{
cout<<"destructor is called \n";
getch(  );
}
int main(  )
{
x   x1,x2;
return 0;
}
```

1. يتم استدعاء دالة البناء عند اشتقاق الكائن وبصورة تلقائية 2. دالة الهدم يتم استدعاؤها أيضاً بصورة تلقائية ولكن عند نهاية البرنامج 3. كل كائن يستدعي دالة البناء ودالة الهدم 4. مخرج البرنامج هو : **constructor is called** " " " **destructor is called** " " "	☑

خواص دوال البناء والهدم:-

1. تحمل نفس اسم الصنف ولكن دالة الهدم تسبق بعلامة (~)
2. يتم تعريفهم في مستوى الحماية العام **public**

3. يمكن إنشاء أكثر من دالة بناء
4. يمكن إنشاء دالة هدم واحدة فقط
5. ليس لهما أنواع رجوع

مثال:-

برنامج يقوم بإيجاد العمليات الأساسية (الجمع ،الطرح ، الضرب ، القسمة) لعددين مدخلين من قبل المستخدم باستخدام دالة البناء

```
#include<iostream.h>
class operations
{
float a,b,c;
public:
operations( );
void result( );
};
operations::operations( )
{
cout<<"Mathematical Operations \n";
cout<<" 1- Addition \n";
cout<<" 2- Subtraction \n";
cout<<" 3- Multiplication \n";
cout<<" 4- Division \n";
cout<<" Please Enter Tow Values a and b \n";
cin>>a>>b;
}
void operations::result( )
{
c=a+b;
cout<<a<<"+"<<b<<"="<<c<<"\n";
c=a-b;
cout<<a<<"-"<<b<<"="<<c<<"\n";
c=a*b;
cout<<a<<"*"<<b<<"="<<c<<"\n";
if(b!=0)
{
c=a/b;
cout<<a<<"/"<<b<<"="<<c<<"\n";
}
else
cout<<"the result of division is infinite \n";
}
```

```
int main( )
{
operations  op;
op.result( );
return 0;
}
```

دالة البناء التي تستخدم المعاملات:-

في داخل دالة البناء وذلك من خلال إضافة عدد **parameters**يمكن أن نستخدم أي نوع من المعاملات المعاملات الضرورية في دالة البناء المعلنة

مثال:-

```
#include<iostream.h>
class myclass
{
int a,b;
public:
myclass(int,int);
void show( );
};
myclass::myclass(int x,int y)
{
cout<<"constructor function \n";
a=x;
b=y;
}
void myclass::show( )
{
cout<<a<<"\t"<<b<<"\n";
}
int main( )
{
myclass ob(4,7);
ob.show( );
return 0;
}
```

مؤشر الكائن Pointer of Object

في الأجزاء السابقة تم الوصول إلى أعضاء الصنف باستخدام الأداة(.) وهي طريقة صحيحة

أيضاً تسمح لغة **C++** باستخدام أداة أخرى للوصول إلى أعضاء الصنف وهي أداة السهم (->)

تماماً كما هو الحال في التراكيب ويتم ذلك عندما يكون الكائن عبارة عن مؤشر

مثال:-

```
#include<iostream.h>
class myclass
{
int a;
public:
myclass(int);
int get( );
};
myclass::myclass(int x)
{
a=x;
}
int myclass::get( )
{
return a;
}
int main( )
{
myclass ob(120); // define object
myclass *p;     //define pointer of object
p=&ob;// transfer address of ob to p
cout<<"value from using object :"<<ob.get( )<<"\n";
cout<<"value from using pointer of object :"<<p->get( )<<"\n";
return 0;
}
```

1. استخدمنا الجملة **myclass *p** لبناء مؤشر إلى الكائن 2. بناء مؤشر الكائن بحد ذاته لايبني الكائن وإنما فقط يؤشر إليه 3. للحصول على عنوان موقع الكائن نستخدم العبارة **p=&ob** عند الحديث عن مؤشر الكائن فانه يظهر مفهوم جديد وهو المؤشر الحسابي ونقصد به هو أن مؤشر الكائن يمكن أن يخضع لبعض العمليات الحسابية وهي : 1. (++) أي زيادة المؤشر **Increment** وتعني أن المؤشر يشير إلى الكائن اللاحق 2. (--) أي إنقاص المؤشر **Decremen** وتعني أن المؤشر يشير إلى الكائن السابق	☑

مثال:-

```
#include<iostream.h>
class samp
{
int a,b;
```

```
public:
samp(int n,int m)
{
a=n;
b=m;
}
int get_a( )
{
return a;
}
int get_b( )
{
return b;
}
};
int main( )
{
samp ob[4]={samp(1,2),samp(3,4),samp(5,6),samp(7,8)}; // define object
int i;
samp *p;
p=&ob[i];
for(i=0;i<4;i++)
{
cout<<p->get_a()<<"\t"<<p->get_b()<<"\n";
p++;
}
return 0;
}
```

كما هو ملاحظ في البرنامج فان كل عملية زيادة في **p++** تشير إلى الكائن التالي في متجه الكائنات

وبما أن الحديث في هذا البرنامج يدور حول متجه الكائنات (**objects array**)

فلابد من الأخذ بعين الاعتبار للملاحظات التالية :-

1. إن الكائن هو بحد ذاته متغير كأي متغير آخر وبناءاً عليه فان **C++** تسمح باستخدام متجه الكائنات
2. أسلوب إعلان متجه الكائنات مشابه تماماً لأسلوب إعلان متجه المتغيرات في **C++**

ولتوضيح هذه الملاحظات نورد المثال التالي:-

```
#include<iostream.h>
class samp
{
int a;
public:
void set_a(int n)
{
```

```
a=n;
}
int get_a( )
{
return a;
}
};
int main( )
{
samp ob[4];// objects array
int i;
for(i=0;i<4;i++)
ob[i].set_a(i);
for(i=0;i<4;i++)
cout<<ob[i].get_a( )<<"\n";
return 0;
}
```

هذا البرنامج يبني مصفوفة من أربعة كائنات من النوع **samp** ومن ثم يعين القيم من 0 الي 3 لكل كائن (عنصر) من عناصر مصفوفة الكائنات

☑	إذا كان الصنف يتعامل مع مصفوفة كائنات وفي نفس الوقت مع دوال البناء الخاصة بها فانه يمكن تهيئة عنصر مصفوفة الكائنات **كما في المثال التالي :-**

```
#include<iostream.h>
class samp
{
int a;
public:
samp(int n)
{
a=n;
}
int get_a( )
{
return a;
}
};
int main( )
{
samp ob[4]={-1,-2,-3,-4};
int i;
for(i=0;i<4;i++)
cout<<ob[i].get_a( )<<"\t";
```

```
cout<<"\n";
return 0;
}
```

هذا البرنامج يظهر على الشاشة القيم **-1 -2 -3 -4**

ونلاحظ أن القيم ترسل إلى الكائن**ob** بواسطة دالة البناء **samp(int n)**

☑	هي طريقة مختصرة لتهيئة متجه الكائنات والذي يمكن ان يكتب بطريقة موسعه حسب الجملة التالية **samp ob[4]={-1,-2,-3,-4}**

مساواة الكائنات :-

يمكن مساواة كائن مع كائن آخر إذا كانت الكائنات من نفس النوع

نفرض أن الكائن **A** يساوي الكائن **B** أي (**A=B**) بناءاً على هذه المساواة فانه يتم نسخ جميع معطيات الكائن **B** إلى الكائن **A** على الترتيب

مثال:-

```
#include<iostream.h>
class myclass
{
int a,b;
public:
void set(int i,int j)
{
a=i;
b=j;
}
void show()
{
cout<<a<<"  "<<b<<"\n";
}
};
int main( )
{
myclass ob1,ob2;
ob1.set(10,4);
ob2=ob1;
ob1.show();
ob2.show();
return 0;
}
```

في المثال أعلاه المتغيرات **a** و **b** الخاصة بالكائن **ob1** قيمها 10 و 4 على الترتيب وعندما تتم مساواة **ob2=ob1** فهذا يودي إلى مساواة **ob2.a** مع **ob1.a** وكذلك **ob2.b** مع **ob1.b** ونتيجة تنفيذ البرنامج تظهر على الشاشة كما يلي:-

```
10      4
10      4
```

☑	عند مساواة الكائنات يجب أن تكون من نفس النوع وألا أدى ذلك إلى خطأ

```
#include<iostream.h>
class myclass
{
int a,b;
public:
void set(int i,int j)
{
a=i;
b=j;
}
void show()
{
cout<<a<<"\t"<<b<<"\n";
}
};
class yourclass
{
int a,b;
public:
void set(int i,int j)
{
a=i;
b=j;
}
void show()
{
cout<<a<<"\t"<<b<<"\n";
}
};
int main( )
{
myclass ob1; // define first object
yourclass ob2;// define second object
```

```
ob1.set(10,4);
ob2=ob1;
// error different names of types  my class and yourclass
ob1.show();
ob2.show();
return 0;
}
```

في البرنامج أعلاه سوف يحث خطأ في العبارة

ob2=ob1

وذلك لان الكائنين من نوعين مختلفين ولكن عندما نتعرف على دوال التحويل سوف نجد انه من الممكن أن نقوم بمساواة كائنين من نوعين مختلفين باستخدام طرق تحويل معينة

(سوف نتعرف عليها لاحقاً)

إرسال الكائن إلى دالة Transfer Object to Function

تسمح لغة **C++** بإرسال الكائنات إلى الدوال كما يتم إرسال المعاملات البسيطة

ويمكن توضيح ذلك من خلال المثال التالي:-

```
#include<iostream.h>
class  samp
{
int i;
public:
samp(int n)
{
i=n;
}
int get_i()
{
return i;
}
};
int sqr_i( samp ob)
{
return ob.get_i()* ob.get_i();
}
int main( )
{
samp a(10),b(2);
cout<<sqr_i(a)<<"\n";
cout<<sqr_i(b)<<"\n";
return 0;
}
```

هذا البرنامج يبني الصنف **samp** الذي يحوي المتغير **i** والدالة **sqr_i( )** والناتج المعاد من الدالة يمثل القيمة التربيعية للمتغير **i** الخاص بالكائن **ob**

وبالتالي نتيجة البرنامج هي 100 و 4

كما تسمح لغة **C++** أيضاً بإرسال عناوين الكائنات (استخدام مؤشر الكائن كوسيطة للدالة)

كما في المثال التالي:-

```
#include<iostream.h>
class  samp
{
int i;
public:
samp(int n)
{
i=n;
}
int get_i()
{
return i;
}
};
void sqr_i( samp  *ob)
{
cout<<ob->get_i()*ob->get_i();
cout<<"\n";
}
int main( )
{
samp a(10);
sqr_i(&a);
return 0;
}
```

المخرج هو 100

من خلال المثال السابق لابد من توضيح نقطة هامة جداً وهي أنه عند إرسال الكائن إلى الدالة فأنه داخل الدالة تبنى نسخة من هذا الكائن وبمجرد انتهاء تنفيذ الدالة فأن نسخة الكائن تختفي وبناءاً على ذلك فأنه سوف تظهر فكرتين أساسيتين

1. عند نداء الدالة التي تحوي الكائن كمعامل فأنه لا يمكن نداء دالة البناء الخاصة بهذا المعامل وهذا يعود لان دالة البناء تستخدم لتهيئة بعض مكونات الكائن وعليه لايمكن نداء هذه الدالة لبناء نسخة لكائن قد تم بناءه ولو قمنا بنداء دالة البناء لهذا الكائن فأن ذلك يسبب تغيير محتوى الكائن الموجود

2. بالنسبة لدالة الهدم فأنه يمكن نداء نسخة عنها وذلك لأنها تقوم على تنظيف الذاكرة بعد البرنامج وهذا ما يحدث تماماً في الدالة التي تحوي الكائن كمعامل حيث أنه بمجرد الانتهاء من تنفيذ الدالة فأن نسخة الكائن تنتهي

مثال:-

```
#include<iostream.h>
class  samp
{
int i;
public:
samp(int n)
{
i=n;
cout<<"constructor work \n";
}
~samp( )
{
cout<<"destructor work \n";
return 0;
}
int get_i()
{
return i;
}
};
int sqr_i( samp  ob)
{
return ob.get_i()*ob.get_i();
}
int main( )
{
samp a(10);
cout<<sqr_i(a)<<"\n";
return 0;
}
```

المخرج هو

```
constructor work
destructor work
100
destructor work
```

استخدام new & delete

في لغة C++ نستخدم المؤشر **new** لحجز الذاكرة الديناميكية والمؤشر **delete** لإلغاء الذاكرة المحجوزة (تنظيف الذاكرة)

والصيغة العامة هي:-

```
p_var =new type;
delete   p_var;
```

حيث **type** تحدد نوع الكائن الذي سنقوم بحجز ذاكرة له أما **p** فهي مؤشر هذا النوع

مثال:-

```
#include<iostream.h>
class samp
{
int i,j;
public:
void set_ij(int a,int b)
{
i=a;
j=b;
}
int get_product( )
{
return i*j;
}
};
int main( )
{
samp *p;
p=new samp;
p->set_ij(4,5);
cout<<"result equal :"<<p->get_product( )<<"\n";
delete p;
return 0;
}
```

مثال:-

```
#include<iostream.h>
class crectangle
{
int *height,*width;
public:
```

```
crectangle(int,int);
~crectangle( );
int area( )
{
return  *height**width;
}
};
crectangle::crectangle(int a,int b)
{
height=new int ;
width=new int;
*height=a;
*width=b;
}
crectangle::~crectangle( )
{
delete height ;
delete width ;
}
int main( )
{
int h,w;
cout<<"enter height \n";
cin>>h;
cout<<"enter width \n";
cin>>w;
crectangle rect(h,w);
cout<<"crectangle area = "<<rect.area( )<<"\n";
return 0;
}
```

☑	يمكن استخدام **new** لحجز ذاكرة ديناميكية لكائن منفرد كما يمكن استخدام **new** لحجز ذاكرة لمتجه كائنات حسب الصيغة العامة التالية:- **p_var =new type[size];** وفي هذه الحالة **p-var** تشير إلى أول عنصر في متجه الكائنات ، كما يستخدم **delete** لحذف متجه كائنات من الذاكرة حسب الصيغة العامة التالية:- **delete[] p_var;**

مثال:-

استخدام **new & delete** مع متجه كائنات

```
#include<iostream.h>
class samp
{
int i,j;
public:
void set_ij(int a,int b)
{
i=a;
j=b;
}
int get_product( )
{
return i*j;
}
};
int main( )
{
samp *p;
int i;
p=new samp[10];
for(i=0;i<10;i++)
{
p->set_ij(i,i);
cout<<p->get_product( )<<"\n";
}
delete[ ] p;
return 0;
}
```