به نام خدا

جزوه

برنامه نویسی شیء گرا

(Object-Oriented Programming)

تابع:

فرم کلی تعریف یک تابع به صورت زیر است:

(پارامترهای تابع) نام تابع نوع تابع
{

بدنه تابع

}

نوع تابع: مشخص می‌کند مقداری که تابع پس از انجام عملیات بر می‌گرداند از چه نوعی است. اگر تابع هیچ مقداری را بر نگرداند نوع آن void در نظر گرفته می‌شود. اگر نوع تابع نوشته نشود به صورت پیش فرض int در نظر گرفته می‌شود.

هر تابع دارای سه جنبه است:

۱- جنبه اعلان (Prototype) : به کامپیوتر می‌گوید که تابع دارای چه شکلی است که در این حالت فقط نام تابع به همراه پارامترهای تابع بعد از توابع کتابخانه‌ایی و قبل از main برنامه نوشته می‌شود (در قسمت تعریف پارامترهای تابع می‌توان فقط نوع پارامتر را نوشت):

(پارامترهای تابع) نام تابع نوع بازگشتی تابع;

مثال ۱: اعلان تابعی که دو پارامتر int را به عنوان ورودی می‌گیرد و حاصل جمع این دو پارامتر را در نام تابع برمی‌گرداند بصورت زیر می‌باشد:

```
int  sum(int a, int b);
```

یا

```
int  sum(int , int );
```

۲- جنبه تعریف: مجموعه‌ای از دستورات است که عملکرد تابع را مشخص می‌کند. که در این حالت فرم کلی تابع به همراه مجموعه دستورات آن بعد از بدنه اصلی main برنامه بصورت زیر نوشته می‌شود:

(پارامترهای تابع) نام تابع نوع بازگشتی تابع
{

مجموعه دستورات تابع

}

مثال ۲: تعریف تابع مربوط به مثال ۱ بصورت زیر می‌باشد:

```
int  sum(int a, int b)
{
     Return (a+b);
}
```

۳- جنبه فراخوانی: دستوری است که تابع را اجرا می‌کند، فراخوانی تابع با نام آن تابع انجام می‌گیرد. و اگر تابع دارای پارامتر ورودی باشد در هنگام فراخوانی تابع باید به تعداد پارامترهای تابع، مقدار متناظر با نوع آن پارامترها به تابع ارسال شود.

مثال ۳: فراخوانی تابع مربوط به مثال ۲ بصورت زیر می‌باشد:

```
main()
{
     int  m, a, b;
     cin>>a>>b;
     m=sum(a,b);
}
```

توجه: یک تابع هر نوع داده‌ای به جز آرایه را می‌تواند به عنوان خروجی برگرداند.

توجه: همه پارامترهای تابع باید بصورت مستقل تعریف شوند. یعنی بصورت: (نام پارامتر نوع پارامتر)

```
myfunction(int  m, int  n, int  k); // -->> Correct
myfunction(int  m, n, k);  // -->> Incorrect
```

تعریف متغیرهای استاتیک در داخل تابع:

متغیرهایی که داخل یک تابع تعریف می‌شود به عنوان متغیرهای محلی می‌باشند. این متغیرهای محلی، در زمان شروع اجرای تابع ایجاد می‌شوند و هنگام خروج از تابع از بین می‌روند. یعنی متغیرهای محلی نمی‌توانند مقدارشان را در بین فراخوانی توابع حفظ کنند. مگر اینکه متغیر به صورت static تعریف شود. که در این صورت کامپایلر شبیه یک متغیر سراسری با آن رفتار می‌کند.

مثال: f1( ) و f2( ) دو تابع می‌باشند که متغیر i در f2( ) بصورت استاتیک تعریف شده است:

```
int  f1()
{
     int i=100;
     i*=2;
```

```
     return i;
}


int  f2()
{
     static int i=100;
     i*=2;
     return i;
}

main()
{
     for(int i=1; i<5; i++){
     cout<<f1(); //output--> 200, 200, 200, 200, 200
     cout<<f2(); //output--> 200, 400, 800, 1600, 3200
     }
}
```

متغیرهای static در اولین بار فراخوانی تابع، حافظه مربوط به آنها تخصیص داده خواهد شد و تا پایان برنامه، این حافظه باقی می‌ماند. بنابراین یکبار در برنامه مقدار اولیه می‌گیرند.

آرگومان‌های هر تابع بصورت متغیر محلی عمل می‌کنند و تغییرات اعمال شده روی آنها در داخل تابع، هیچ تاثیری روی مقادیر متغیرها در تابع فراخوان کننده ندارد.

پارامترهای تابع با مقادیر اولیه (Default Argument):

برای هر پارامتر تعریف شده در اعلان تابع، هنگام فراخوانی آن تابع باید یک مقدار متناسب با نوع آن پارامتر به تابع فرستاده شود:

```
long myfunction(int);
```

اگر در اعلان تابع یک مقدار اولیه به پارامترها داده شود، آنگاه در هنگام فراخوانی تابع اگر یک مقدار متناسب با آن پارامترها برای تابع ارسال نشود، مقدار اولیه برای پارامتر تابع در نظر گرفته می‌شود.

```
long myfunction(int x=50);
```

یا

```
long myfunction(int =50);
```

این دو اعلان تابع در بالا می‌گویند که تابع myfunction یک مقدار long بر می‌گرداند و یک مقدار int به عنوان پارامتر می‌گیرد و اگر در فراخوانی این تابع مقداری برای پارامتر آن به تابع ارسال نشود مقدار اولیه ۵۰ برای آن در نظر گرفته می‌شود.

توجه: همه پارامترهای یک تابع می‌توانند مقدار اولیه داشته باشند. فقط یک محدودیت وجود دارد اگر یک پارامتر مقدار اولیه نداشته باشد آنگاه پارامترهای قبل از آن نیز نباید مقدار اولیه داشته باشند.

```
Long myfunc(int param1, int param2, int param3=100);
```

زمانی می‌توانیم به param2 مقدار اولیه دهیم که param3 دارای مقدار اولیه باشد.

مثال:

```
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);//area=10000
     area=areacube(length, width); // area=5000
     area=areacube(length); // area=2500
}

int  areacube(int length, int width=25, int height=1)
{
     return (length*width*height);
}
```

Function Overloading:

ایجاد چند تابع با نام یکسان را Overloading یا سربارگذاری گویند این توابع در لیست پارامترها با هم متفاوت هستند که این تفاوت می‌تواند در تعداد پارامترها یا نوع پارامترها و یا هر دو باشد.

مثال: یک تابع به نام myfunction با سه لیست پارامتر متفاوت:

```
int myfunction(int, int);
int myfunction(long, long);
int myfunction(int);
```

در Overloading، نوع بازگشتی تابع می‌تواند یکسان یا متفاوت باشد اما دو تابع با لیست پارامتر یکسان ولی با نوع بازگشتی متفاوت، یک خطای کامپایلری ایجاد می‌کند.

مثال: یک تابع که ورودی خودش را دو برابر می‌کند بدون overloading باید سه تابع با نام مختلف داشته باشیم ولی با overloading سه تابع با نام یکسان ولی پارامترهای متفاوت داریم:

```
int Double (int);
long Double (long);
float Double (float);
```

توجه: یک تابع با پارامترهای با مقادیر اولیه (default argument) ممکن است با یک تابع overload، یکسان به نظر برسد و این سبب ایجاد یک خطای کامپایلری می‌شود. مثلاً اگر در یک برنامه دو تابع با نام یکسان وجود داشته باشد که یکی بدون پارامتر باشد و دیگری دارای چند پارامتر و با مقادیر اولیه برای همه پارامترها باشد. آنگاه موقع فراخوانی تابع بدون پارامتر یک خطای کامپایلری ایجاد می‌شود زیرا کامپایلر نمی‌داند که کدام تابع را باید اجرا کند:

```
int f1();
int f1(int a=10, int 20);
main()
{
     f1(); // یک خطای کامپایلری ایجاد می‌شود
     f1(25);  // در این حالت تابع دومی اجرا می‌شود
}
```

تابع inline:

موقعی که یک تابع تعریف می‌شود کامپایلر یک مجموعه از دستورات را در حافظه ایجاد می‌کند، موقع فراخوانی تابع، برنامه به تابع فوق پرش می‌کند و پس از پایان تابع دوباره به خط بعد از فراخوانی تابع برمی‌گردد و اگر شما مثلاً ۱۰ مرتبه تابع فوق را فراخوانی کنید روند فوق ۱۰ مرتبه تکرار می‌شود و این سبب یک سربار (overhead) در پرش به داخل تابع و خروج از آن می‌شود که این روند وقت گیر است.

اگر قبل از اعلان تابع کلمه کلیدی inline نوشته شود کامپایلر یک تابع واقعی ایجاد نمی‌کند بلکه کد مربوط به تابع inline را به تابع فراخوان کپی می‌کند. اگر چه این سبب افزایش سرعت می‌شود ولی از طرفی سبب افزایش اندازه کد در برنامه فراخوان کننده تابع می‌شود بنابراین فقط زمانی که تابع خیلی کوچک باشد inline مفید می‌باشد.

تابع بازگشتی: یک تابع که خودش را فراخوانی می‌کند تابع بازگشتی نام دارد. موقعی که یک تابع خودش را فراخوانی می‌کند یک کپی جدید از تابع اجرا می‌شود که متغیرهای محلی آن از متغیرهای محلی تابع اولی مستقل می‌باشد و آن‌ها نمی‌توانند بطور مستقیم روی همدیگر تاثیر بگذارند.

مثال: محاسبه تابع فاکتوریل بصورت بازگشتی:

```
int fact (int n)
{
     int answer;
     if (n==1) return 1;
     answer=fact(n-1)*n;
     return answer;
}
```

آرایه‌ها:

یک مجموعه از متغیرها با نام و نوع یکسان می‌باشد عناصر آرایه با استفاده از ایندکس قابل دسترسی هستند همه آرایه‌ها شامل محل‌های پیوسته‌ای از حافظه برای ذخیره شدن می‌باشند. کمترین آدرس مطابق با اولین عنصر و بیشترین آدرس مطابق با آخرین عنصر می‌باشد.

فرم کلی تعریف یک آرایه یک بعدی:

```
type var_name[size];
```

همیشه اولین عنصر آرایه ایندکس صفر دارد:

```
int a[10];  //  a[0]تا a[9]
a[3]=5;   // مقدار دهی به عناصر آرایه
b=a[4];   // دسترسی به عناصر آرایه
```

توجه: اگر آرایه‌ایی با طول n داشته باشیم همیشه ایندکس خانه‌های آرایه بین صفر و n-1 خواهد بود از طرفی در زبان C/C++ هیچ گونه کنترلی روی مرزهای آرایه انجام نمی‌شود و اگر ایندکسی بیشتر از

n-1 در برنامه استفاده شود از طرف کامپایلر خطایی اعلام نمی‌شود اما با این کار به محدوده متغیرهای دیگر تجاوز می‌شود.

نام آرایه بدون ایندکس همیشه به اولین خانه آرایه اشاره می‌کند (آدرس خانه اول را نگهداری می‌کند):

```
int *p;
int sample[10];
p=sample; // آدرس اولین خانه آرایه در متغیر اشاره‌گر قرار می‌گیرد.
```

ارسال آرایه یک بعدی به تابع: نمی‌توان تمام آرایه را به عنوان پارامتر به یک تابع فرستاد در واقع باید یک اشاره‌گر به یک آرایه را (همان نام آرایه بدون ایندکس) به تابع ارسال کرد.

اگر یک تابع یک آرایه یک بعدی را به عنوان پارامتر دریافت کند به سه روش می‌توان پارامتر آن تابع را تعریف کرد:

۱- **به صورت یک اشاره‌گر:**

```
void  myfunction (int *x);
```

۲- **یک آرایه با اندازه مشخص:**

```
void  myfunction (int x[10]);
```

۳- **یک آرایه با اندازه نا مشخص:** طول آرایه برای تابع مهم نیست زیرا تابع یک اشاره‌گر دریافت می‌کند نه یک آرایه با تمام عناصر.

```
void  myfunction (int x[]);
```

هر سه تعریف بالا نتایج یکسانی دارند زیرا همه به کامپایلر می‌گویند که یک اشاره‌گر integer می‌خواهد دریافت شود.

توجه: نمی‌توان بیشتر از اندازه آرایه مقداردهی اولیه کرد:

```
int a[4]={10,20,30,40,50,60} //  Error
```

اگر عناصر کمتر از اندازه آرایه باشند مابقی خانه‌های خالی آرایه صفر می‌شوند:

```
int a[5]={10,20};
```

10	20	0	0	0

رشته‌ها:

در زبان C رشته نوع جدیدی نیست بلکه بصورت آرایه‌ای از کاراکترها تعریف می‌شود. برای تعیین انتهای رشته از کاراکتر خاصی بنام NULL ('\0') استفاده می‌شود بنابراین اگر رشته‌ای بصورت زیر تعریف شود فقط از ۹ کاراکتر می‌تواند استفاده کند.

```
Char str[10];
Str="ali";
```

A	L	I	\0	?	?	?	?	?	?

در C++ همچنین کلاسی بنام CString تعریف شده است که با استفاده از آن می‌توان رشته‌ها را تعریف کرد.

هنگام تعریف رشته‌ها در زبان C می‌توان به آن‌ها مقدار اولیه داد که در این حالت طول آرایه یک واحد بیشتر از تعداد کاراکترهایی است که به آن نسبت داده می‌شود:

```
Char str[]="ali"; // روش اول
Char str[]={'a', 'l', 'i', '\0'}; // روش دوم
```

در روش اول کاراکتر '\0' به طور خودکار در انتهای رشته اضافه می‌شود ولی در روش دوم کاراکتر '\0' باید توسط برنامه نویس در انتهای رشته قرار داده شود.

اشاره‌گرها:

متغیرهایی هستند که آدرس یک محل از حافظه را در خود نگهداری می‌کنند. و دارای دو عملگر & و * می‌باشند.

با استفاده از عملگر & می‌توان به آدرس یک محل از حافظه دسترسی داشت در واقع وقتی عملگر & به دنبال یک متغیر قرار می‌گیرد آدرس آن متغیر در حافظه را برمی‌گرداند.

با استفاده از عملگر * می‌توان به محتویات خانه‌ایی از حافظه که آدرس آن داده شده است دسترسی داشت. یعنی می‌توان به محتویات متغیرهای اشاره‌گر دسترسی پیدا کرد.

تعریف اشاره‌گرها:

```
type_variable  *name_pointer;
int *p;
```

مثال: اگر آدرس متغیر count برابر با ۲۰۰۰ و محتوای آن ۱۰۰ باشد و m یک متغیر اشاره‌گر باشد آنگاه :

```
m = &count;
```
آدرس حافظه‌ایی متغیر count را در m قرار می‌دهد .

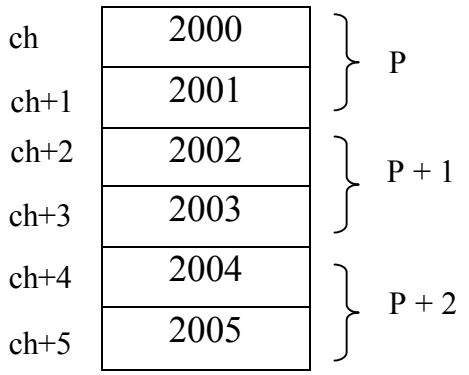
و m=2000 می‌شود.

عملگر * محتوای یک آدرس حافظه را بر می‌گرداند یعنی در دستور زیر q = 100 می‌شود:

```
q = *m;
```

اگر P1 یک اشاره‌گر integer با مقدار ۲۰۰۰ باشد و اندازه هر integer دو بایت باشد آنگاه بعد از اجرای P1++ مقدار P1 برابر با ۲۰۰۲ می‌شود زیرا هر بار که P1 یک واحد اضافه می‌شود آن می-خواهد به integer بعدی اشاره کند. (همه اشاره‌گرها با توجه به طول نوع داده‌ای که به آن اشاره می-کنند به همان میزان افزایش یا کاهش می‌یابند):

```
Char *ch = 2000;
Int *p = 2000;
```




```
int *p = 0 ;
```
یک اشاره‌گر با مقدار صفر، یک اشاره‌گر تهی (null) می‌باشد.

توجه: اگر هنگام تعریف اشاره‌گر به آن مقداری داده شود این مقدار به عنوان آدرس اشاره‌گر در نظر گرفته می‌شود نه محتویات اشاره‌گر. در مثال زیر p به خانه‌ایی از حافظه اشاره می‌کند که آدرس آن ۱۰۰۰ می‌باشد.

```
int *p=1000;
```

اشاره‌گرها ممکن است بصورت آرایه باشند:

```
int *x[10];
x[2]=&var;          تخصیص آدرس یک متغیر integer به نام var به سومین عنصر آرایه اشاره‌گر x:
m=*x[2];            دسترسی به محتوای متغیر var از طریق آرایه‌ایی از اشاره‌گرها:
```

آرایه‌های اشاره‌گر بیشتر برای اشاره به رشته‌ها استفاده می‌شوند.

مثال: تابعی که یک پیغام خطا را با توجه به کد داده شده در پارامتر ورودی نمایش می‌دهد.

```
viod syntax_error (int num ){
     Static char *err[] = {"can not open file \n",
                           "Read error \n",
                           "Write error \n"};
     Printf("%s", err [num]);}
```

برای ارسال پارامتر ورودی به تابع زیر، نام آرایه را بدون ایندکس می‌نویسیم.

```
void func1(int *q[]);
```

q یک اشاره‌گر به یک integer نیست بلکه یک اشاره‌گر به یک آرایه از اشاره‌گرهای integer می‌باشد.

اشاره‌گر به یک اشاره‌گر دیگر:

P یک اشاره‌گر به یک عدد float نیست بلکه اشاره‌گر به یک اشاره‌گر float می‌باشد.

```
float **p;
```

مثال:

```
int x,*p,**q;
   x = 10;
   p = &x;
   q = &p;
```

در مثال بالا محتوای q** برابر ۱۰ می‌باشد.

کلاس‌ها و اشیا در C++:

با تعریف کلاس یک نوع داده جدید ایجاد می‌شود.

فرم کلی تعریف کلاس بصورت زیر می‌باشد:

class *نام کلاس* {

تعریف داده‌ها و توابع اختصاصی

private: // --> *داده‌ها و توابع اختصاصی فقط در داخل کلاس قابل دسترسی هستند.*

تعریف داده‌ها و توابع اختصاصی

public: // --> *داده‌ها و توابع عمومی در سایر کلاس‌های موجود در برنامه قابل استفاده‌اند.*

تعریف داده‌ها و توابع عمومی

protected: // --> *داده‌ها و توابع محافظت شده در وراثت کلاس‌ها مورد استفاده قرار می‌گیرند.*

تعریف داده‌ها و توابع محافظت شده

};

توابعی که در داخل کلاس اعلان شده‌اند باید بصورت کامل بیرون از کلاس بصورت زیر تعریف شوند (البته می‌توان تعریف تابع به همراه اعلان آنرا در داخل کلاس نوشت) :

نوع بازگشتی تابع نام کلاس : : نام تابع (اسامی پارامترها)

{

}

تعریف کلاس اشیای واقعی را ایجاد نمی‌کند. همانطور که نقشه یک ساختمان، ساختمان واقعی را ایجاد نمی‌کند. برای استفاده از یک کلاس باید یک نمونه از آن کلاس ایجاد شود همانطور که برای استفاده از یک مقدار int باید یک متغیر از نوع int تعریف شود (int a;)

ایجاد نمونه ای از یک کلاس را اعلان شئ می‌نامند و بصورت زیر تعریف می‌شود:

نام کلاس نام شئ ;

تمام خصوصیاتی که یک کلاس دارد در اشیایی از آن کلاس نیز وجود دارد یعنی شئ دارای اعضایی است که در کلاس وجود دارند برای دستیابی به اعضای یک شئ از نوع یک کلاس به صورت زیر عمل می‌شود عضو کلاس می‌تواند یک متغیر داده‌ای یا یک تابع باشد:

نام شئ . نام عضو ;

مثال: یک کلاس که طول و عرض مستطیلی را از ورودی گرفته و مساحت آنرا چاپ می‌کند.

```
class rectangle {
     int length, width;
  public:
     void set_l_w(int m, int n);
     int area();
};
void rectangle::set_l_w(int m, int n)
{
     length = m;
     width = n;
}
int rectangle::area()
{
     return (length*width);
}
void main()
{
     int a, b;
     cin>>a>>b;
     rectangle ob1;
     ob1.set_l_w(a, b);
     cout<<"area is:"<<ob1.area();
}
```

نکته: اعضای داده‌ایی کلاس را نمی‌توان مستقیماً در هنگام تعریف متغیر در داخل کلاس، مقداردهی اولیه کرد.

تفاوت public و private :

اعضایی که در داخل کلاس بصورت اختصاصی تعریف می‌شوند فقط در داخل همان کلاس قابل دسترسی هستند و در بیرون از کلاس نمی‌توان به آنها دسترسی داشت. ولی اعضایی که در داخل کلاس بصورت عمومی تعریف می‌شوند در بیرون از کلاس و در سرتاسر برنامه قابل دسترسی هستند.

مثلاً در کلاس زیر a و b اختصاصی تعریف شده‌اند بنابراین فقط از طریق توابع داخل کلاس قابل تغییر هستند و نمی‌توان بیرون از کلاس آنها را مقداردهی کرد یا به آنها دسترسی داشت.

```
class myclass {
     int a;
     int b;
   Public:
     int set_a(int a1, int b1) {a=a1; b=b1;}
      };
Void main()
{
myclass p1;
p1.a=50;//Error  مستقیم به متغیر نمی‌توان دسترسی داشت زیرا بصورت اختصاصی می‌باشد.
P1.set_a(50, 10);   //  باید از طریق یک تابع عمومی به متغیرها دسترسی داشت.
}
```

بهتر است داده‌ها بصورت اختصاصی باشند و از توابع عمومی برای تنظیم کردن و یا گرفتن مقادیر آنها استفاده شود.

سازنده‌ها (Constructors) :

برای مقدار اولیه دادن به اعضای داده‌ای اشیا مستقیماً نمی‌توان در داخل کلاس به متغیرها مقدار اولیه داد بلکه باید از توابع ویژه‌ای به نام توابع سازنده استفاده کرد. تابع سازنده دارای ویژگی‌های زیر است:

۱. همنام با کلاس است.
۲. فاقد نوع بازگشتی است (حتی void هم نیست).
۳. توسط برنامه نویس فراخوانی نمی‌شود بلکه هنگام ایجاد شئ از نوع کلاس، به طور خودکار فراخوانی و اجرا خواهد شد.

مثال:

```
Class  myclass{
     int a, b;
  public:
     myclass(); // constructor
     void show(){cout<<a<<'\t'<<b;}
};
myclass::myclass()
{
```

```
     a = 0;
     b = 0;
 }
Void main()
{
   myclass p1; //  هنگام ایجاد یک شئ از نوع کلاس، تابع سازنده فراخوانی می‌شود.
   p1.show(); //  مقدار صفر برای دو متغیر چاپ می‌شود.
}
```

در کلاس مثال قبل هنگام ایجاد شئ از نوع کلاس myclass تابع سازنده myclass( ) اجرا می‌شود و اعضای داده‌ای a و b برابر صفر می‌شوند.

موقعی که یک شئ از یک کلاس تعریف می‌شود سازنده آن فراخوانی می‌شود و اگر سازنده دارای پارامتر باشد هنگام تعریف شئ از یک کلاس باید مقادیر پارامترهای آن بصورت زیر ارسال شوند:

```
نام کلاس   نام شئ   (مقادیر پارامترهای سازنده);
```

مثال: کلاس myclass دارای یک سازنده می‌باشد که سازنده آن دو پارامتر را از ورودی می‌گیرد. نحوه تعریف یک شئ از نوع کلاس فوق بصورت زیر می‌باشد:

```
Class  myclass{
     int a, b;
   public:
     myclass(int m, int n);  // constructor
     void show(){cout<<a;}
};
myclass::myclass(int m, int n)
{
     a = m;
     b = n;
 }
Void main()
{
 myclass p1(5,10); // هنگام ایجاد یک شئ از نوع کلاس، تابع سازنده فراخوانی می‌شود.
 P1.show();   //  مقدار ۵ چاپ می‌شود.
}
```

مخرب‌ها (Destructors):

بعد از تعریف هر سازنده نیاز به یک مخرب می‌باشد تا حافظه تخصیص داده شده به شئ را آزاد کند. یک مخرب دارای ویژگی‌های زیر است:

۱. همنام با کلاس است. که با علامت ~ دنبال شده است.
۲. نوع بازگشتی ندارد و هیچ پارامتری را به عنوان ورودی نمی‌گیرد.
۳. با پایان یافتن برنامه، مخرب بطور خودکار فراخوانی می‌شود. همچنین توسط برنامه نویس نیز می‌تواند فرخوانی شود.

مثال: کلاس زیر دارای یک مخرب می‌باشد که به محض پایان یافتن عمر شئ (رسیدن به پایان "}" بلوک جاری)، فراخوانی می‌شود.

```
class myclass{
     int a;
     int b;
  public:
     myclass ( int m, int n);
     ~myclass();
};
myclass :: myclass( int m,int n )
{
     a = m;
     b = n;
}
myclass :: ~myclass( )
{
     cout<< "call destructor";
}
```

آرایه‌ای از اشیاء:

در C++ می‌توان آرایه‌ای از اشیاء داشت. ساختار تعریف و استفاده از آرایه‌ای از اشیا شبیه دیگر انواع آرایه می‌باشد.

```
Class myclass {
     int i;
   public:
     void set(int a) {i=a;}
};

void main() {
     myclass ob[5]; //   تعریف آرایه‌ای از اشیا با اندازه ۵ از کلاس
     ob[1].set(5);  //   دسترسی به اعضای شئ دوم از آرایه
```

```
}
```

در برنامه بالا یک آرایه با ۵ عنصر از نوع کلاس myclass تعریف شده است. در واقع یک آرایه از اشیاء از نوع کلاس myclass تعریف کرده‌ایم. برای دسترسی به اعضای کلاس در این حالت باید ایندکس شئ را نیز مشخص کنیم.
برای حالتی که کلاس شامل سازنده می‌باشد و سازنده نیز یک پارامتر می‌گیرد طریقه تعریف آرایه‌ای از اشیاء از نوع کلاس فوق و مقدار دادن به پارامتر سازنده بصورت زیر می‌باشد:

{مقادیر پارامتر سازنده‌ها}=[اندازه آرایه] نام آرایه‌ای از اشیا نام کلاس

```
myclass ob[3]={1,4,7}
```

برای مقدار دادن به پارامترهای یک سازنده در حالتی که سازنده بیش از یک پارامتر می‌گیرد طریقه تعریف آرایه‌ای از اشیاء از نوع کلاس فوق بصورت زیر می‌باشد:

```
myclass ob[3]={myclass(2,3),myclass(1,4),myclass(2,6)}
```

در مثال بالا کلاس myclass یک سازنده دارد که دو پارامتر را به عنوان ورودی می‌گیرد.

تخصیص اشیاء:

اگر دو شئ از یک نوع کلاس باشند آنگاه می‌توان یک شئ را به شئ دیگر تخصیص داد در این حالت داده‌های شئ سمت راست تساوی روی داده‌های شئ سمت چپ تساوی، کپی می‌شوند.

```
class myclass{
     int i;
   public:
     void set_i(int n) {i=n;}
     int get_i(){return i;}
};
void  main()
{
     myclass ob1, ob2;
     ob1.set_i(99);
     ob2=ob1;
```

```
     cout<<ob2.get_i();// مقدار ۹۹ چاپ می‌شود
}
```

عبور اشیاء به تابع:

اشیاء مانند دیگر انواع داده می‌توانند به تابع ارسال شوند اشیاء بصورت ارسال با مقدار به تابع ارسال می‌شوند در واقع یک کپی از شئ به تابع ارسال می‌شود موقعی که یک کپی از شئ ایجاد می‌شود و به تابع ارسال می‌شود تابع سازنده آن اجرا نمی‌شود زیرا موقع ارسال یک شئ به تابع، می‌خواهیم که حالت جاری شئ به تابع ارسال شود و اگر سازنده فراخوانی شود ممکن است اعضای کلاس مقدار اولیه بگیرند و شئ تغییر کند. ولی موقعی که تابع پایان می‌یابد باید تابع مخرب فراخوانی شود زیرا شئ در داخل تابع شبیه یک متغیر محلی عمل می‌کند.

```
class myclass {
     int i;
   public:
     myclass(int n);
     ~myclass();
     void set_i(int n) { i=n; }
     int get_i() { return i; }
};
myclass::myclass(int n)
{
     i = n;
     cout << "Constructing " << i << "\n";
}
myclass::~myclass()
{
     cout << "Destroying " << i << "\n";
}
void f(myclass ob);
int main()
{
     myclass o(1);
     f(o);
     cout << "This is i in main: ";
     cout << o.get_i() << "\n";
     return 0;
}
void f(myclass ob)
{
     ob.set_i(2);
     cout << "This is local i: " << ob.get_i();
```

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

```
Output:

Constructing 1
This is local i: 2
Destroying 2
This is i in main: 1
Destroying 1
```

Const member function :

اگر یک تابع در یک کلاس بصورت ثابت تعریف شود آنگاه آن تابع نمی‌تواند ارزش‌های هیچ عضوی از کلاس را تغییر دهد. برای تعریف یک تابع ثابت در کلاس باید کلمه کلیدی const را بعد از پرانتز و قبل از ";" قرار دهیم.

```
void myfunction() const; //   نحوه تعریف در داخل کلاس
```

Static data member:

موقعی که یک متغیر static داخل کلاس تعریف می‌شود در واقع به کامپایلر گفته می‌شود که فقط یک کپی از آن متغیر بوجود آید و همه اشیاء ایجاد شده از کلاس، از آن متغیر بصورت اشتراکی استفاده کنند. همه متغیرهای static قبل از اینکه اولین شئ ایجاد شود مقدار اولیه صفر دارند موقعی که یک متغیر static داخل کلاس تعریف می‌شود. هیچ حافظه‌ای برای آن تخصیص داده نمی‌شود و باید متغیر بصورت سراسری خارج از کلاس و به همراه نام کلاس بصورت زیر تعریف شود:

نوع متغیر نام کلاس : : نام متغیر;

```
class shared{
     static int a;
     int b;
   public:
     void set(int i,int j){a=i; b=j;}
     void show();
};
int shared::a;
void shared:: show()
```

```
{
     cout<<"this is static a:"<<a;
     cout<<"\n this is non-static b:'<<b;
     cout<<\n";
}

int main()
{
     shared x, y;
     x.set(1,1);  // set a to 1
     x.show();
     y.set(2,2);  // change a to 2
     y.show();
     x.show();
     return 0;
}
```

```
Output:

This is static a: 1
This is non-static b: 1
This is static a: 2
This is non-static b: 2
This is static a: 2
This is non-static b: 1
```

اگر یک متغیر static بصورت public تعریف شود آنگاه قبل از ایجاد یک شئ از کلاس، می‌توان به محتویات متغیر static دسترسی داشت و نیز آن متغیر را مقداردهی کرد.

```
class shared{
   public:
     static int a;
};
int shared::a;
int main()
{
     shared::a=100;
     cout<<"this is initial value of:"<shared::a<<"\n";
     shared x;
     cout<<"this is x.a:"<<x.a;  // display 100
     return 0;
}
```

یکی از کاربردهای استفاده از متغیرهای static در داخل کلاس فراهم کردن کنترل دسترسی به بعضی از منابع اشتراکی است که بوسیله همه اشیاء یک کلاس استفاده می‌شوند برای مثال شما ممکن است چندین شئ از یک کلاس ایجاد کنید و هر شئ نیاز داشته باشد که داخل دیسک بنویسد و از طرفی فقط یک شئ در یک زمان اجازه دارد که داخل دیسک بنویسد در این حالت باید یک متغیر static تعریف کنید که نشان دهد که چه موقع دیسک در حال استفاده است و چه موقع دیسک آزاد می‌باشد هر شئ قبل از نوشتن در داخل دیسک باید این متغیر static را چک کند.

مثال:

```
class c1{
     static int resource;
   public:
     int get_resource();
     void free_resource() {resource=0;}
};
int c1::resource;
int c1::get_resource();
{
   if (resource)
     {cout<<"disk is busy";
     return 0;}
   else
     {resource=1;
     cout<<"object get disk";
     return 1;}
}

int main()
{
c1  ob1,ob2;
if (ob1.get_resource())  cout<<"ob1 get resource\n";
if (!Ob2.get_resource()) cout<<"ob2 denied resource\n";
ob1.free_resource();
if (ob2.get_resource())  cout<<"ob2 can now use
resource\n";
return 0;
}
```

مثال: استفاده از متغیر استاتیک در داخل سازنده و مخرب:

```
class Counter {
```

```
   public:
     static int count;
     Counter() { count++; }
     ~Counter() { count--; }
};
int Counter::count;
void f();
int main(void)
{
     Counter o1;
     cout << "Objects in existence: ";
     cout << Counter::count << "\n";
     Counter o2;
     cout << "Objects in existence: ";
     cout << Counter::count << "\n";
     f();
     cout << "Objects in existence: ";
     cout << Counter::count << "\n";
     return 0;
}
void f()
{
     Counter temp;
     cout << "Objects in existence: ";
     cout << Counter::count << "\n";
     // temp is destroyed when f() returns
}
```

```
Output:

Objects in existence: 1
Objects in existence: 2
Objects in existence: 3
Objects in existence: 2
```

: Static member function

توابع عضو یک کلاس می‌توانند بصورت استاتیک تعریف شوند توابعی که بصورت استاتیک تعریف می‌شوند فقط می‌توانند به دیگر اعضای استاتیک کلاس دسترسی داشته باشند توابع استاتیک می‌توانند قبل از ایجاد شئ از نوع کلاس فراخوانی و اجرا شوند. طریقه فراخوانی تابع استاتیک قبل از ایجاد شئ از نوع کلاس بصورت زیر می‌باشد:

(لیست پارامترها) نام تابع استاتیک : : نام کلاس;

مثال:

```
class static_type {
     static int i;
   public:
     static void init(int x) {i = x;}
     void show( ) {cout << i;}
};
int static_type::i; // define i
int main()
{
// init static data before object creation
     static_type::init(100);
     static_type x;
     x.show( ); // displays 100
     return 0;
}
```

توابع دوست (Friend function) :

یک تابع دوست عضو یک کلاس نیست ولی می‌تواند به داده‌های اختصاصی و محافظت شده یک کلاس دسترسی داشته باشد. برای تعریف یک تابع دوست باید اعلان تابع را در داخل کلاس به همراه کلمه کلیدی friend در ابتدای اعلان تابع بنویسیم.

مثال: تابع sum در مثال زیر عضو کلاس نیست ولی می‌تواند به داده‌های اختصاصی کلاس دسترسی داشته باشد زیرا به عنوان دوست کلاس معرفی شده است:

```
class myclass {
     int a, b;
   public:
     friend int sum(myclass x);
     void set_ab(int i, int j);
};
void myclass::set_ab(int i, int j)
{
     a = i;
     b = j;
}
int sum(myclass x)
{
     return x.a + x.b;
}
```

```
int main()
{
     myclass n;
     n.set_ab(3, 4);
     cout << sum(n);  // display 7
     return 0;
}
```

عیب: توابع دوست کپسوله سازی داده‌ها را نقض می‌کنند. ولی در عوض قدرت C++ را افزایش می-دهند.

مزایا :

۱. توابع دوست برای سربارگذاری کردن نوع‌های ویژه‌ایی از عملگرها مفید می‌باشند.
۲. توابع دوست ایجاد بعضی از توابع I/O را آسان‌تر می‌کنند.
۳. دو یا چند کلاس ممکن است شامل اعضایی باشند که با قسمت‌های دیگر برنامه در ارتباط هستند در این حالت توابع دوست مفید می‌باشند. توضیح این حالت در ادامه به همراه یک مثال آورده شده است.

فرض کنید دو کلاس مختلف موقعی که یک خطا اتفاق بیفتد یک پیغام را روی صفحه نمایش، نشان می‌دهند و از طرفی قسمت‌های دیگر برنامه قبل از اینکه چیزی را روی آن پیغام بنویسند بخواهند بدانند که آیا یک پیغام در حال حاضر روی صفحه نمایش، نشان داده شده است یا خیر. برای چک کردن این وضعیت اگر چه می‌توان توابع عضوی در هر کلاس تعریف کرد و موقعی که یک پیغام فعال می‌باشد آن توابع، یک ارزش را برگردانند ولی این سبب یک سربار اضافی می‌شود یعنی هر بار باید دو تابع هر کلاس فراخوانی شوند. بنابراین در این حالت استفاده از یک تابع که یک دوست برای هر دو کلاس می‌باشد این امکان را می‌دهد که برای چک کردن وضعیت صفحه نمایش فقط یک تابع را فراخوانی و اجرا کنیم.

مثال: استفاده از تابع دوست برای کنترل یک متغیر وضعیت در دو کلاس مختلف:

```
const int IDLE = 0;
const int INUSE = 1;
class C2; // forward declaration
class C1 {
     int status; // IDLE if off, INUSE if on screen
   public:
     void set_status(int state);
     friend int idle(C1 a, C2 b);
```

```
};
class C2 {
     int status; // IDLE if off, INUSE if on screen
   public:
     void set_status(int state);
     friend int idle(C1 a, C2 b);
};
void C1::set_status(int state)
{
     status = state;
}
void C2::set_status(int state)
{
     status = state;
}
int idle(C1 a, C2 b)
{
     if(a.status || b.status) return 0;
     else return 1;
}
int main()
{
     C1 x;
     C2 y;
     x.set_status(IDLE);
     y.set_status(IDLE);
     if(idle(x, y)) cout << "Screen can be used.\n";
     else cout << "In use.\n";
     x.set_status(INUSE);
     if(idle(x, y)) cout << "Screen can be used.\n";
     else cout << "In use.\n";
     return 0;
}
```

:Friend class

یک کلاس می‌تواند به عنوان دوست کلاس دیگری باشد. موقعی که یک کلاس به عنوان دوست کلاس دیگری می‌باشد آنگاه کلاس دوست و همه اعضای اختصاصی تعریف شده داخل آن به کلاس دیگر دسترسی دارند.

```
class TwoValues {
     int a;
     int b;
   public:
```

```
     TwoValues(int i, int j) { a = i; b = j;}
     friend class Min;
};
class Min {
   public:
     int min(TwoValues x);
};
int Min::min(TwoValues x)
{
     return x.a < x.b ? x.a : x.b;
}
int main()
{
     TwoValues ob(10, 20);
     Min m;
     cout << m.min(ob);
     return 0;
}
```

وراثت (Inheritance):

فرم کلی تعریف وراثت در کلاس بصورت زیر می‌باشد:

class *derived-class-name : **access** base-class-name* {
// body of class
};

در ساختار بالا کلمه access نوع دسترسی کلاس مشتق شده از کلاس پایه را تعیین می‌کند که می‌تواند Protected، private و یا public باشد اگر دسترسی بصورت public باشد. آنگاه همه اعضای public در کلاس پایه به عنوان اعضای public برای کلاس مشتق شده در نظر گرفته می‌شوند و همه اعضای protected در کلاس پایه به عنوان اعضای protected در کلاس مشتق شده در نظر گرفته می‌شوند. توابع عضو کلاس مشتق شده به اعضای private کلاس پایه دسترسی ندارند.
بنابراین وضعیت دسترسی به اعضای کلاس پایه داخل کلاس مشتق شده با استفاده از access تعیین می‌شود.

توجه: یک کلاس مشتق شده، توابع دوست کلاس پایه را نمی‌توانند به ارث ببرند.

مثال: در مثال زیر کلاس derived بصورت public از کلاس base مشتق شده است بنابراین می‌توان با استفاده از اشیاء کلاس derived به اعضای عمومی کلاس base دسترسی داشت:

```
class base {
     int i, j;
   public:
     void set(int a, int b) { i=a; j=b; }
     void show() { cout << i << " " << j << "\n"; }
};
class derived : public base {
     int k;
     public:
     derived(int x) { k=x; }
     void showk() { cout << k << "\n"; }
};
int main()
{
     derived ob(3);
     ob.set(1, 2); // access member of base
     ob.show(); // access member of base
     ob.showk(); // uses member of derived class
     return 0;
}
```

موقعی که کلاس پایه بصورت private به ارث برده شده باشد آنگاه همه اعضای public و protected کلاس پایه به عنوان اعضای private کلاس مشتق شده در نظر گرفته می‌شوند.

وراثت و اعضای حفاظت شده (protected) :

موقعی که یک عضو از یک کلاس بصورت protected تعریف شود. آن عضو توسط دیگر عناصر غیر عضو کلاس قابل دسترسی نیست در واقع اعضای protected و private دارای دسترسی یکسان هستند. فقط یک تفاوت بین این دو نوع دسترسی در هنگام ارث‌بری وجود دارد:

اعضای Private یک کلاس پایه توسط دیگر قسمت‌های برنامه شامل هر کلاس مشتق شده‌ای از آن کلاس پایه قابل دسترسی نیستند. ولی اعضای Protected رفتار متفاوتی دارند. اگر کلاس پایه بصورت Public به ارث برده شود آنگاه اعضای حفاظت شده کلاس پایه به عنوان اعضای محافظت شده کلاس مشتق شده در نظـــر گرفته می‌شوند و بنابراین توسط کلاس مشتق شده قابل دسترسی هستند با استفاده از Protected شما می‌توانید اعضای کلاسی ایجاد کنید که برای کلاس شما اختصاصی هستند و همچنین می‌توانند به ارث‌برده شوند و توسط کلاس مشتق شده قابل دسترسی باشند.

اگر یک کلاس بصورت Protected به ارث‌برده شود آنگاه همه اعضای Public و Protected کلاس پایه به عنوان اعضای Protected کلاس مشتق شده در نظر گرفته می‌شوند.

مثال: دسترسی به اعضای محافظت شده کلاس پایه توسط کلاس مشتق شده از آن کلاس پایه:

```
class base {
   protected:
     int i,j; //private to base,but accessible by derived
   public:
     void set(int a, int b) { i=a; j=b; }
     void show() { cout << i << " " << j << "\n"; }
};
class derived : public base {
     int k;
   public:
     void setk() { k=i*j; }
     void showk() { cout << k << "\n"; }
};
int main()
{
     derived ob;
     ob.set(2, 3); // OK, known to derived
     ob.show(); // OK, known to derived
     ob.setk();
     ob.showk();
     return 0;
}
```

اشاره‌گر به اشیاء:

در C++ می‌توان اشاره‌گرهایی از نوع یک کلاس تعریف کرد، برای دسترسی به اعضای یک کلاس که شئ آن از نوع اشاره‌گر می‌باشد باید از عملگر جهتی " <- " استفاده کنید.

```
class cl {
     int i;
   public:
     cl(int j) { i=j; }
     int get_i() { return i; }
};
int main()
{
     cl ob(88), *p;
     p = &ob; // get address of ob
```

```
    cout << p->get_i(); // use -> to call get_i()
    return 0;
}
```

در برنامه بالا اشاره‌گر p از نوع کلا س cl تعریف شده است و سپس به شئ ob از کلاس cl اشاره می‌کند.

یک اشاره‌گر از یک نوع کلاس نمی‌تواند به یک شئ از نوع کلاس دیگری اشاره کند. اما یک استثناء در مورد کلاس‌های مشتق شده وجود دارد یعنی یک اشاره‌گر از نوع کلاس پایه می‌تواند به یک شئ از یک کلاس مشتق شده از آن کلاس پایه، اشاره کند. اما عکس آن درست نیست.

در این حالت اشاره‌گر فقط می‌تواند به اعضایی از شئ مشتق شده دسترسی داشته باشد که از کلاس پایه به ارث برده شده‌اند یعنی اشاره‌گر نمی‌تواند به اعضای کلاس مشتق شده که در کلاس پایه وجود ندارند دسترسی داشته باشند.

مثال:

```
class base {
    int i;
  public:
    void set_i(int num) { i=num; }
    int get_i() { return i; }
};
class derived: public base {
    int j;
  public:
    void set_j(int num) { j=num; }
    int get_j() { return j; }
};
int main()
{
    base *bp;
    derived d;
    bp = &d; // base pointer points to derived object
    // access derived object using base pointer
    bp->set_i(10);
    cout << bp->get_i() << " ";
    /* The following won't work. You can't access
    element of a derived class using a base class
    pointer.
    bp->set_j(88); // error
    cout << bp->get_j(); // error
    */
    return 0;
```

```
}
```

البته باید توجه داشت که با استفاده از تعریف cast می‌توان به همه اعضای کلاس مشتق شده با استفاده از اشاره‌گر کلاس پایه دسترسی داشت که ساختار تعریف آن بصورت زیر می‌باشد:

(اعضای کلاس مشتق شده<- (اشاره‌گر به کلاس پایه (نام کلاس مشتق شده)*

```
cout << ((derived *)bp)->get_j();
```

سازنده و مخرب در وراثت:

در ارث‌بری، هر دو کلاس پایه یا مشتق شده می‌توانند دارای سازنده و یا مخرب باشند و هر وقت یک شئ از نوع کلاس مشتق شده تعریف شود ابتدا سازنده مربوط به کلاس پایه آن، و سپس سازنده مربوط به خود شئ اجرا می‌شود.

مثال:

```
class base {
   public:
     base() { cout << "Constructing base\n"; }
     ~base() { cout << "Destructing base\n"; }
};
class derived: public base {
   public:
     derived() { cout << "Constructing derived\n"; }
     ~derived() { cout << "Destructing derived\n"; }
};
int main()
{
     derived ob;//do nothing but construct and destruct ob
     return 0;
}
```

```
Output:

Constructing base
Constructing derived
Destructing derived
Destructing base
```

عبور پارامتر به سازنده کلاس پایه:

اگر سازنده یک کلاس پایه دارای پارامتر باشد آنگاه هنگام تعریف یک کلاس مشتق شده از آن، باید پارامترهای سازنده کلاس پایه را از طریق سازنده کلاس مشتق شده، به کلاس پایه به فرم زیر ارسال کرد بنابراین کلاس مشتق شده در این حالت باید حتماً دارای سازنده باشد:

```
derived-constructor(arg-list) : base1(arg-list),
                                base2(arg-list), ...
                                baseN(arg-list)
{
// body of derived constructor
}
```

مثال:

```
class base {
   protected:
     int i;
   public:
     base(int x) {i=x; cout << "Constructing base\n"; }
     ~base() { cout << "Destructing base\n"; }
};
class derived: public base {
     int j;
   public:
     // derived uses x; y is passed along to base.
     derived(int x, int y): base(y)
     { j=x; cout << "Constructing derived\n"; }
     ~derived() { cout << "Destructing derived\n"; }
     void show() { cout << i << " " << j << "\n"; }
};
int main()
{
     derived ob(3, 4);
     ob.show(); // displays 4 3
     return 0;
}
```

توابع مجازی (Virtual Function) و چندریختی (Polymorphism) :

یک تابع مجازی یک تابع عضو می‌باشدکه داخل کلاس پایه تعریف شده است و داخل کلاس‌های مشتق شده از کلاس پایه دوباره تعریف می‌شود. برای تعریف تابع مجازی باید کلمه کلیدی virtual قبل از اعلان تابع داخل کلاس پایه نوشته شود.

تابع مجازی چند ریختی یا "یک رابط، چندین روش" را پیاده سازی می‌کند. تابع مجازی داخل کلاس پایه، فرم رابط را تعریف می‌کند و هر تعریف دوباره تابع مجازی داخل کلاس مشتق شده، یک روش ویژه را پیاده سازی می‌کند. آنچه که توابع مجازی را قادر می‌کند که چند ریختی را پشتیبانی کنند رفتار آن‌ها در هنگام دسترسی از طریق اشاره‌گرها می‌باشد یک اشاره‌گر کلاس پایه می‌تواند به هر شئ از کلاس مشتق شده از آن کلاس پایه اشاره کند. و این اشاره‌گر می‌تواند به توابع مجازی به طور مستقیم دسترسی داشته باشد.

مثال:

```
class base {
   public:
     virtual void vfunc() {
     cout << "This is base's vfunc().\n";
     }
};
class derived1 : public base {
   public:
     void vfunc() {
     cout << "This is derived1's vfunc().\n";
     }
};
class derived2 : public base {
   public:
     void vfunc() {
     cout << "This is derived2's vfunc().\n";
     }
};
int main()
{
     base *p, b;
     derived1 d1;
     derived2 d2;
     // point to base
     p = &b;
     p->vfunc(); // access base's vfunc()
     // point to derived1
     p = &d1;
     p->vfunc(); // access derived1's vfunc()
     // point to derived2
     p = &d2;
     p->vfunc(); // access derived2's vfunc()
```

```
    return 0;
}
```

در حالتی که اشاره‌گر p به یک شئ از کلاس مشتق شده اشاره کند آنگاه اگر تابع مجازی داخل کلاس مشتق شده دوباره تعریف شده باشد دستور p -> vfunc( ) تابع مجازی موجود در کلاس مشتق شده را اجرا می‌کند ولی اگر تابع مجازی داخل کلاس مشتق شده تعریف نشده باشد آنگاه p -> vfunc( ) تابع مجازی مربوط به کلاس پایه را اجرا می‌کند. یعنی اولویت اول با تابع مشتق شده است بعد با تابع پایه.

تابع مجازی محض (Pure Virtual Function):

اگر تابع مجازی موجود در کلاس پایه را مساوی صفر قرار دهیم و دستورات مربوط به بدنه آنرا ننویسیم یعنی در کلاس پایه فقط اعلان تابع را به صورت ساختار زیر بنویسیم:

virtual *type func-name(parameter-list*) = 0;

آنگاه تابع فوق را تابع مجازی محض گوییم.

در این حالت هر کلاس مشتق شده‌ای از این کلاس پایه باید تابع مجازی محض را تعریف کند در غیر این صورت کامپایلر خطا می‌دهد.

یک کلاس که شامل یک تابع مجازی محض می‌باشد یک کلاس انتزاعی می‌باشد و نمی‌توان شئ از آن کلاس ایجاد کرد. اما می‌توان یک اشاره‌گر از آن کلاس تعریف کرد. که این خاصیت به کلاس انتزاعی اجازه می‌دهد که چند ریختی را پشتیبانی کند.

مثال: یک مثال که چند ریختی را با استفاده از تابع مجازی محض پیاده سازی می‌کند:

یک کلاس پایه به نام convert که تبدیل از یک سسیتم به سیستم دیگر را انجام می‌دهد. کلاس پایه دو متغیر val1 و val2 دارد که مقدار اولیه و مقدار تبدیل شده را نگهداری می‌کند و دو تابع getconv() و getinit() دارد که مقدار اولیه و مقدار تبدیل شده را بر می‌گرداند این اعضای کلاس convert ثابت هستند و برای همه کلاس‌های مشتق شده از کلاس convert قابل اجرا هستند. همچنین کلاس convert یک تابع به نام compute() دارد که تبدیل را انجام می‌دهد این تابع به صورت یک تابع مجازی محض می‌باشد که در کلاس‌های مشتق شده باید تعریف شود:

```
class convert {
   protected:
```

```
    double val1; // initial value
    double val2; // converted value
  public:
    convert(double i) {
    val1 = i;
    }
    double getconv() { return val2; }
    double getinit() { return val1; }
    virtual void compute() = 0;
};
// Liters to gallons.
class l_to_g : public convert {
  public:
    l_to_g(double i) : convert(i) { }
    void compute() {
    val2 = val1 / 3.7854;
    }
};
// Fahrenheit to Celsius
class f_to_c : public convert {
    public:
    f_to_c(double i) : convert(i) { }
    void compute() {
    val2 = (val1-32) / 1.8;
    }
};
int main()
{
    convert *p; // pointer to base class
    l_to_g lgob(4);
    f_to_c fcob(70);// use virtual function mechanism to convert
    p = &lgob;
    cout << p->getinit() << " liters is ";
    p->compute();
    cout << p->getconv() << " gallons\n"; // l_to_g
    p = &fcob;
    cout << p->getinit() << " in Fahrenheit is ";
    p->compute();
    cout << p->getconv() << " Celsius\n"; // f_to_c
    return 0;
}
```

اشارہ‌گر This (*This Pointer*):

هر شئ از یک کلاس به آدرس خودش با استفاده از یک اشاره‌گر به نام ***this*** دسترسی دارد و این اشاره‌گر ***this*** بوسیله کامپایلر و به عنوان یک آرگومان ضمنی به هر تابع غیر استاتیک عضو شئ ارسال می‌شود در واقع هر تابع غیر استاتیک عضو کلاس، یک پارامتر مخفی با نام اشاره‌گر ***this*** دارد که می‌تواند با استفاده از این اشاره‌گر به اعضای کلاس دسترسی داشته باشد یا شئ خود را با استفاده از دستور ***return*** به خروجی تابع ارسال کند.

مثال: برنامه زیر دارای یک کلاس برای محاسبه توان می‌باشد که هر تابع عضو کلاس از اشاره‌گر ***this*** برای دسترسی به اعضای داده‌ایی استفاده می‌کند.

```
class pwr {
     double b;
     int e;
     double val;
   public:
     pwr(double base, int exp);
     double get_pwr() { return val; }
};
pwr::pwr(double base, int exp)
{
     this->b = base;
     this->e = exp;
     this->val = 1;
     if(exp==0) return;
     for( ; exp>0; exp--)
          this->val = this->val * this->b;
}
int main()
{
     pwr x(4.0, 2);
     cout << x.get_pwr();
     return 0;
}
```

سربارگذاری عملگرها (Operator Overloading)

چند ریختی با استفاده از سربارگذاری عملگرها نیز می‌تواند بدست آید. بسیاری از عملگرها را می‌توان سربارگذاری کرد موقعی که یک عملگر سربارگذاری می‌شود عملکرد قبلی آن بدون تغییر باقی می‌ماند و فقط عملیات ویژه‌ایی در ارتباط با کلاسی که در آن تعریف شده است به آن اضافه می‌شود. مثلاً یک

کلاس که شامل پشته می‌باشد ممکن است عملگر "+" را برای انجام عمل Push دوباره سربارگذاری کند.

سربارگذاری عملگرها با استفاده از تعریف توابع عملگر انجام می‌شود در واقع تابع عملگر، عملکرد عملگر سربارگذاری شده را در ارتباط با یک کلاس تعیین می‌کند یک تابع عملگر می‌تواند عضو یک کلاس باشد و یا به عنوان دوست یک کلاس معرفی شود روش تعریف توابع عضو و غیر عضو (به عنوان تابع دوست) کلاس با همدیگر متفاوت می‌باشد. و هر کدام در ادامه بصورت جداگانه شرح داده می‌شود.

سربارگذاری عملگرها با استفاده از توابع عضو کلاس:

فرم کلی تعریف یک تابع عملگر عضو بصورت زیر می‌باشد:

*ret-type class-name::*operator#*(arg-list)*
{
// operations
}

نوع بازگشتی تابع عملگر معمولاً از نوع کلاسی که در آن تعریف شده است می‌باشد ولی در کل هر نوع داده‌ایی می‌تواند داشته باشد. به جای # باید عملگری نوشته شود که می‌خواهد سربارگذاری شود. مثلاً اگر شما می‌خواهید عملگر "+" را سربارگذاری کنید باید به جای # عملگر "+" نوشته شود. برای سربارگذاری یک عملگر یکانی باید لیست پارامترها *(arg-list)* خالی باشد و برای سربارگذاری یک عملگر دوتایی باید یک پارامتر معمولاً از نوع کلاس به تابع ارسال شود.

مثال: یک کلاس که طول جغرافیایی و عرض جغرافیایی را ذخیره می‌کند و عملگر "+" را در ارتباط با کلاس سربارگذاری می‌کند.

```
class loc {
     int longitude, latitude;
   public:
     loc() {}
     loc(int lg, int lt) {
     longitude = lg;
     latitude = lt;
}
void show() {
     cout << longitude << " ";
```

```
     cout << latitude << "\n";
}
loc operator+(loc op2);
};
// Overload + for loc.
loc loc::operator+(loc op2)
{
     loc temp;
     temp.longitude = op2.longitude + longitude;
     temp.latitude = op2.latitude + latitude;
     return temp;
}
int main()
{
     loc ob1(10, 20), ob2( 5, 30);
     ob1.show(); // displays 10 20
     ob2.show(); // displays 5 30
     ob1 = ob1 + ob2;
     ob1.show(); // displays 15 50
     return 0;
}
```

در مثال بالا تابع عملگر "+" فقط یک پارامتر ورودی دارد در حالی که آن یک عملگر دوتایی را سربارگذاری کرده است در این حالت عملوند سمت چپ عملگر سربارگذاری شده "+" بطور ضمنی با استفاده از اشاره‌گر *this* به تابع ارسال می‌شود. و عملوند سمت راست عملگر با استفاده از پارامتر ورودی تابع، به تابع ارسال می‌شود. دلیل آنکه عملوند سمت چپ بطور ضمنی به تابع ارسال می‌شود این است که در عملگرهای دودویی سربارگذاری شده، شئ سمت چپ، تابع عملگر را فراخوانی می-کند.

مثال: برنامه زیر ۳ عملگر "="، "-" و عملگر یکانی "++" را برای کلاس *loc* سربارگذاری کرده است

```
class loc {
     int longitude, latitude;
   public:
     loc() {} // needed to construct temporaries
     loc(int lg, int lt) {
     longitude = lg;
     latitude = lt;
}
void show() {
     cout << longitude << " ";
     cout << latitude << "\n";
```

```
}
     loc operator+(loc op2);
     loc operator-(loc op2);
     loc operator=(loc op2);
     loc operator++();
};
// Overload + for loc.
loc loc::operator+(loc op2)
{
     loc temp;
     temp.longitude = op2.longitude + longitude;
     temp.latitude = op2.latitude + latitude;
     return temp;
}
// Overload - for loc.
loc loc::operator-(loc op2)
{
     loc temp;
     // notice order of operands
     temp.longitude = longitude - op2.longitude;
     temp.latitude = latitude - op2.latitude;
     return temp;
}
loc loc::operator=(loc op2)
{
     longitude = op2.longitude;
     latitude = op2.latitude;
     return *this; //return object that generated call
}
// Overload prefix ++ for loc.
loc loc::operator++()
{
     longitude++;
     latitude++;
     return *this;
}
int main()
{
     loc ob1(10, 20), ob2( 5, 30), ob3(90, 90);
     ob1.show();
     ob2.show();
     ++ob1;
     ob1.show(); // displays 11 21
     ob2 = ++ob1;
```

```
    ob1.show(); // displays 12 22
    ob2.show(); // displays 12 22
    ob1 = ob2 = ob3; // multiple assignment
    ob1.show(); // displays 90 90
    ob2.show(); // displays 90 90
    ob1=ob1-ob2;
    ob1.show(); // displays 0 0
    ob2.show(); // displays 90 90
    return 0;
}
```

در مثال بالا اگر عملگر "=" سربارگذاری نشده باشد بصورت پیش فرض تمام اعضای داده‌ای یک شئ از یک کلاس در شئ دیگری از همان کلاس کپی می‌شود. ولی با استفاده از سربارگذاری می‌توان تعریف خاصی از این عملگر در کلاس مورد نظر ایجاد کرد. در این حالت هر وقت برای اشیایی از این کلاس از عملگر "=" استفاده شود منظور تعریف سربارگذاری شده آن می‌باشد. توجه شود که تابع عملگر "=" مقدار اشاره‌گر ***this*** را برمی‌گرداند که همان شئ می‌باشد که عملگر را فراخوانی کرده است. در مثال بالا همچنین تابع عملگر "++" بدون پارامتر می‌باشد و تنها عملوند آن از طریق اشاره‌گر ***this*** به تابع ارسال می‌شود. این تابع عملگر پیش افزایشی "++" را پیاده سازی می‌کند. اگر بخواهیم تابع پس افزایشی "++" را پیاده سازی کنیم از آنجایی که نام هر دو تابع عملگر پیش افزایشی و پس افزایشی یکسان می‌باشد برای اینکه دو تابع از هم تمیز داده شوند تابع پس افزایشی یک پارامتر از نوع int دارد که هیچ مقدار صحیحی به این تابع ارسال نمی‌شود که اصطلاحاً به آن پارامتر گنگ گفته می‌شود و ساختار آن بصورت زیر می‌باشد:

```
    loc operator++(int);
```

همان طور که گفته شد همه عملگرها حتی عملگرهای مختصر نویسی مانند "+=" و "-=" در *C++* می‌توانند سربارگذاری شوند مثلاً برای سربارگذاری عملگر "+=" در کلاس ***loc***، باید تابع زیر به کلاس اضافه شود:

```
loc loc::operator+=(loc op2)
{
    longitude = op2.longitude + longitude;
    latitude = op2.latitude + latitude;
    return *this;
}
```

سربارگذاری عملگرها با استفاده از توابع دوست:

می‌توان یک عملگر را با استفاده از توابع دوست یک کلاس سربارگذاری کرد و از آنجایی که تابع دوست، عضو یک کلاس نیست پس نمی‌تواند از اشاره‌گر *this* استفاده کند بنابراین در یک تابع دوست سربارگذاری شده باید عملوندها مشخص شوند یعنی یک تابع دوست که یک عملگر دوتایی را سربارگذاری می‌کند باید دارای دو پارامتر باشد که اولین پارامتر به عنوان عملوند سمت چپ و دومین پارامتر به عنوان عملوند سمت راست در نظر گرفته می‌شود. یک تابع دوست که یک عملگر یکانی را سربارگذاری می‌کند باید دارای یک پارامتر باشد.

مثال: در برنامه زیر عملگر "+" برای کلاس *loc* با استفاده از تابع دوست سربارگذاری شده است.

```
class loc {
     int longitude, latitude;
   public:
    loc() {} // needed to construct temporaries
    loc(int lg, int lt) {
       longitude = lg;
       latitude = lt;
     }
    void show() {
      cout << longitude << " ";
      cout << latitude << "\n";
    }
   friend loc operator+(loc op1, loc op2);//now a friend
};

loc operator+(loc op1, loc op2)
{
     loc temp;
     temp.longitude = op1.longitude + op2.longitude;
     temp.latitude = op1.latitude + op2.latitude;
     return temp;
}
int main()
{
     loc ob1(10, 20), ob2( 5, 30);
     ob1 = ob1 + ob2;
     ob1.show();  // display 15  50
     return 0;
}
```

باید توجه شود که چند محدودیت در استفاده از تابع دوست وجود دارد. با استفاده از تابع دوست نمی‌توان عملگرهای "="، "[ ]"، "( )" و "<-" را سربارگذاری کرد. برای سربارگذاری عملگرهای یکانی "++" و "--" باید از پارامتر مرجع برای تابع دوست استفاده شود.

استفاده از تابع دوست برای سربارگذاری عملگرهای "++" و "--"

همانطور که گفته شد برای سربارگذاری عملگرهای یکانی "++" و "--" با استفاده از تابع دوست باید از پارامتر مرجع استفاده شود.

مثال: برنامه زیر عملگرهای یکانی "++" و "--" را با استفاده از تابع دوست برای کلاس *loc* سربارگذاری می‌کند.

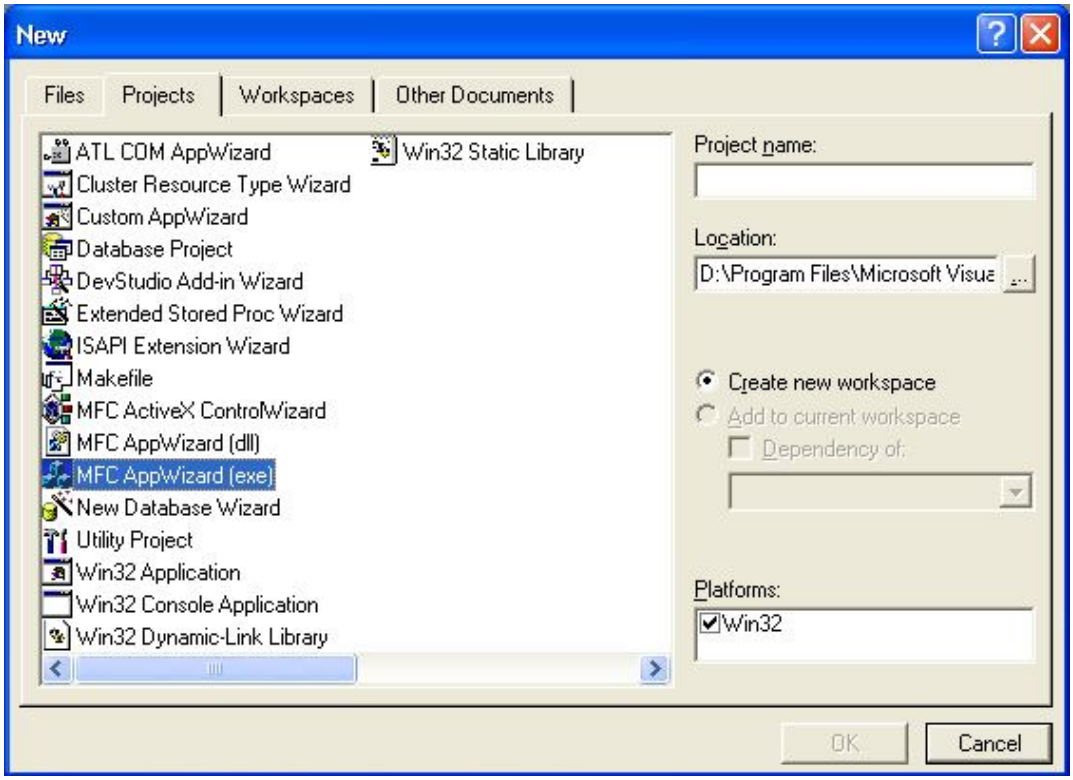
```
class loc {
     int longitude, latitude;
   public:
     loc() {}
     loc(int lg, int lt) {
        longitude = lg;
        latitude = lt;
     }
     void show() {
        cout << longitude << " ";
        cout << latitude << "\n";
     }
     loc operator=(loc op2);
     friend loc operator++(loc &op);
     friend loc operator--(loc &op);
};
// Overload assignment for loc.
loc loc::operator=(loc op2)
{
     longitude = op2.longitude;
     latitude = op2.latitude;
     return *this; // return object that generated call
}
// Now a friend; use a reference parameter.
loc operator++(loc &op)
{
     op.longitude++;
     op.latitude++;
     return op;
```

```
}
// Make op-- a friend; use reference.
loc operator--(loc &op)
{
     op.longitude--;
     op.latitude--;
     return op;
}
int main()
{
     loc ob1(10, 20), ob2;
     ob1.show();
     ++ob1;
     ob1.show(); // displays 11 21
     ob2 = ++ob1;
     ob2.show(); // displays 12 22
     --ob2;
     ob2.show(); // displays 11 21
     return 0;
}
```

آموزش Visual C++ 6

برای ایجاد اولین برنامه ابتدا مراحل زیر را دنبال کنید:

۱- گزینه file/new... را انتخاب کنید تا پنجره ای مانند شکل ۱ ظاهر شود سپس در پنجره Projects گزینه MFC AppWizard (exe) را انتخاب کنید. در بخش Project name نام پروژه و در بخش Location مسیری که پروژه باید ذخیره شود را مشخص کنید.



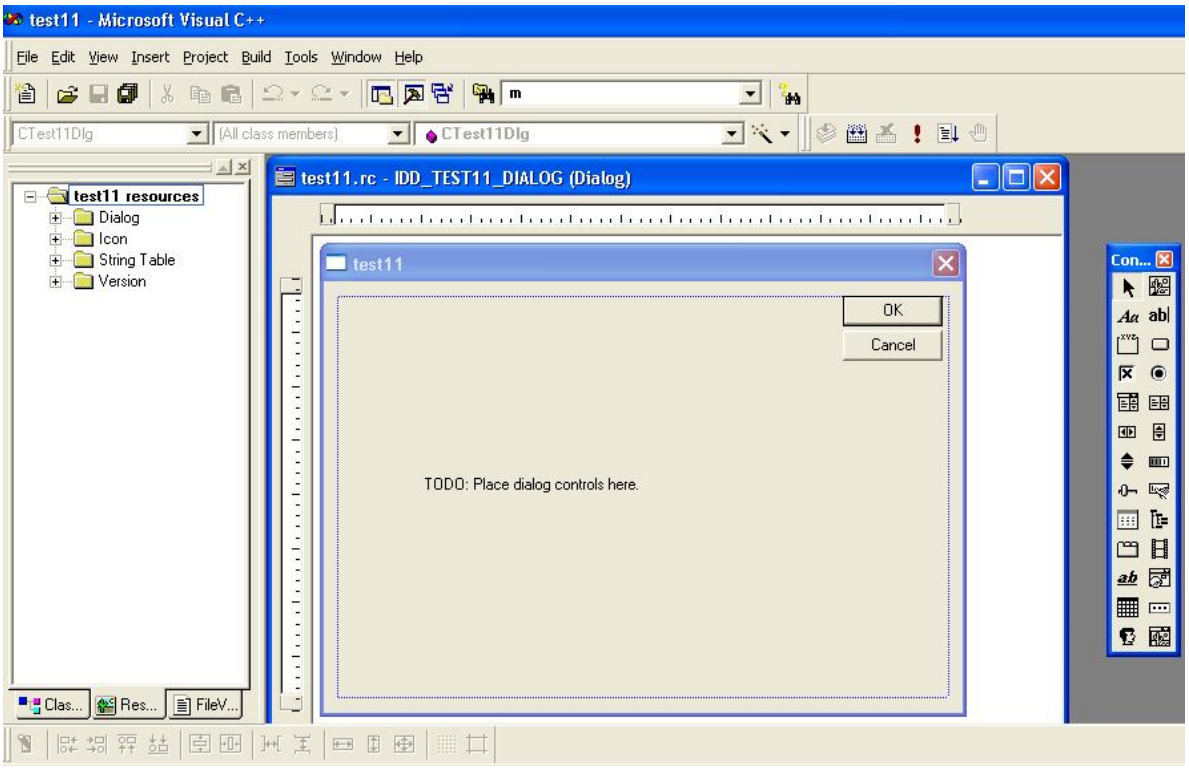
شکل ۱

۲- دکمه OK را کلیک کنید در پنجره بعدی همانند شکل ۲، گزینه Dialog base را انتخاب کنید. که این سبب ایجاد یک معماری مبتنی بر کادر محاوره برای برنامه می‌شود.

۳- تمام گزینه های دیگر را بصورت پیش فرض قبول کنید و دکمه Finish را کلیک کنید. تا شکل ۳ ظاهر شود شکل فوق کادر محاوره ای برنامه را نشان میدهد در واقع محیط کاری برنامه میباشد (از این به بعد در طول این جزوه آموزشی منظور از ایجاد یک پروژه کادر محاوره‌ایی انجام این ۳ مرحله می‌باشد).

۴- بر روی دکمه Ok دکمه سمت راست موس را کلیک و در منویی که ظاهر می‌شود گزینه Properties را انتخاب کنید تا پنجره‌ایی مانند شکل ۴ ظاهر شود.

شکل ۲



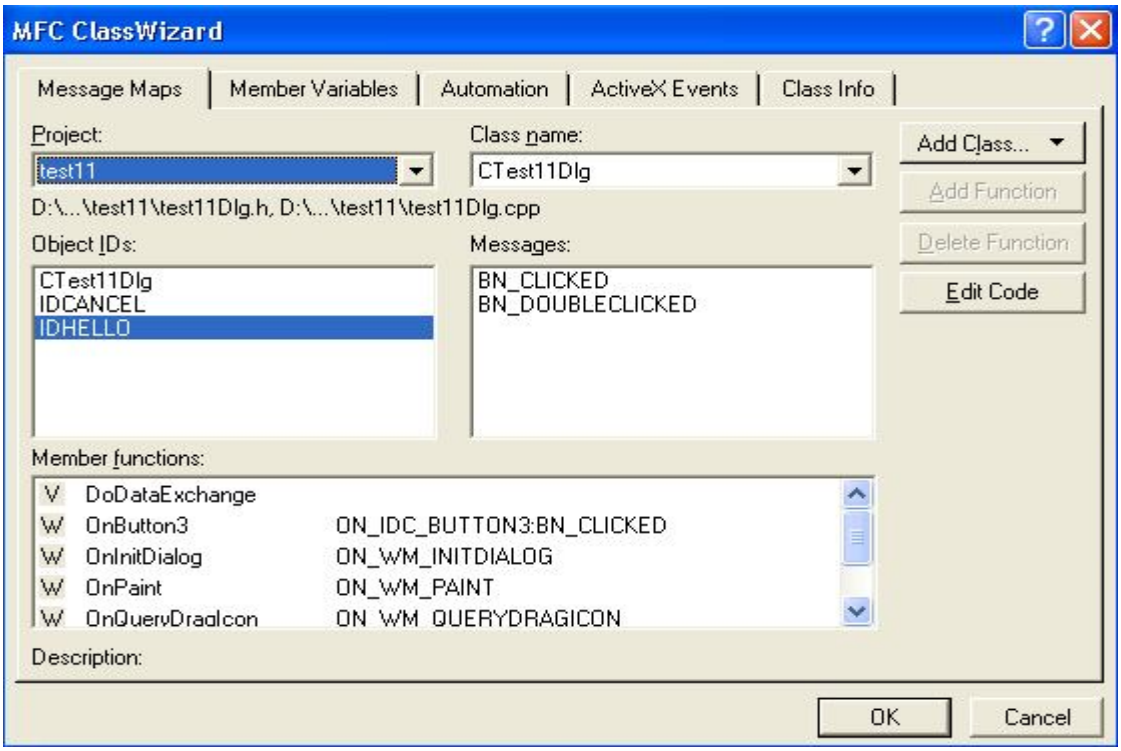
شکل ۳

۵- در شکل ۴، ID ؛ شناسه منحصر به فرد برای هر کنترل می‌باشد که معمولاً با _IDD شروع می‌شود در این حالت مقدار ID را از IDOK به IDHELLO تغییر دهید. فیلد Caption عنوان کنترل را مشخص می‌کند و مقدار آنرا به &Hello تغییر دهید. سپس پنجره Properties را ببندید.

۶- بر روی دکمه Hello دکمه سمت راست موس را کلیک کرده و گزینه ClassWizard را انتخاب کنید تا شکل ۵ ظاهر شود. بخش Object ID شیءهای موجود در کادر محاوره را نشان می‌دهد که بسته به نوع کنترلی که می‌خواهیم برای آن کد بنویسیم شئ مورد نظر را انتخاب می‌کنیم در اینجا IDHELLO را انتخاب می‌کنیم.

۷- بخش Messages رویدادهای مربوط به هر کنترل را نشان می‌دهد که این رویدادها برای هر کنترل متفاوت می‌باشد. در این حالت BN_CLICKED را انتخاب کنید.

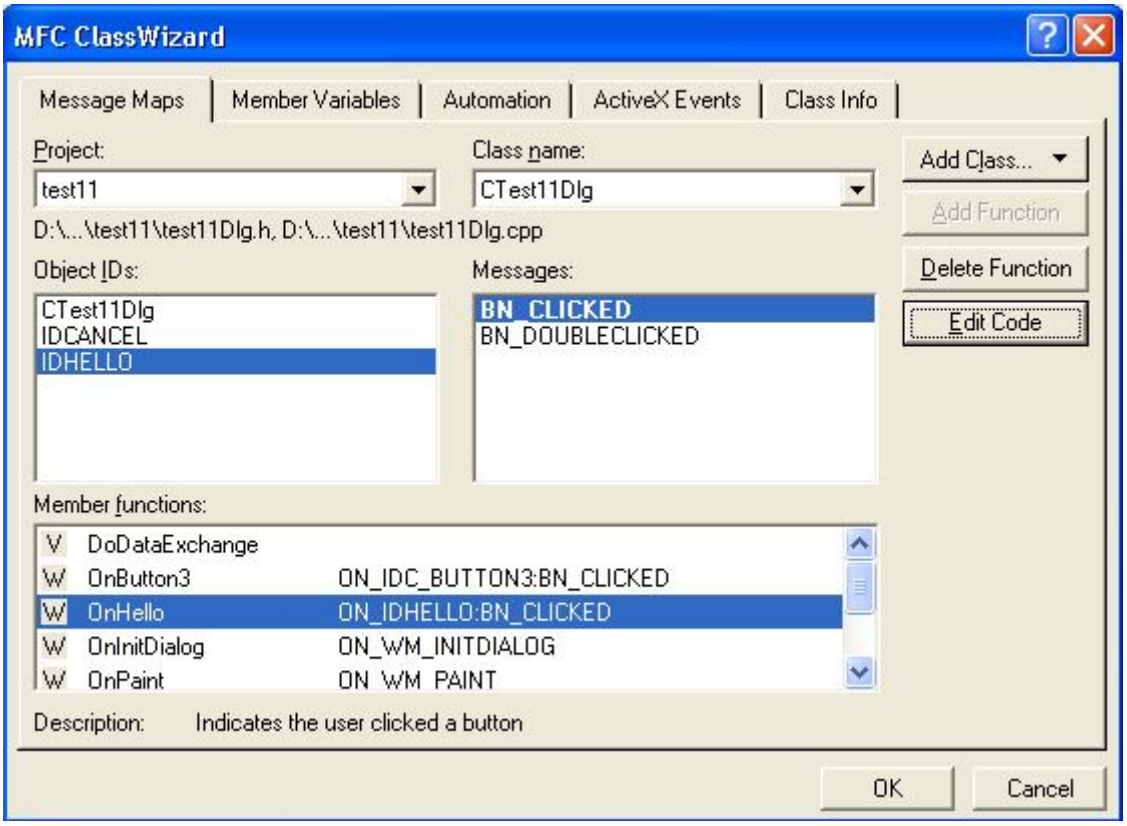
شکل ۴



شکل ۵

۸- اکنون نوبت به اضافه کردن یک تابع برای شئ IDHELLO می‌باشد برای این هدف در سمت راست پنجره دکمه Add Function را انتخاب کنید تا شکل ۶ ظاهر شود. در بخش Member function name باید نام تابع را انتخاب کنید. سپس دکمه Ok را کلیک کنید. حالا نام این تابع در بخش Member Function قرار می‌گیرد (همانند شکل ۷).

شکل ۶



شکل ۷

۹- حال دکمه Edit Code را در سمت راست پنجره کلیک کنید تا پنجره نوشتن کد برای آن تابع باز شود. در این قسمت می‌توان دستورات مربوط به کنترل را نوشت. برای این تابع دستور زیرا را بنویسید:

```
MessageBox("Hello");
```

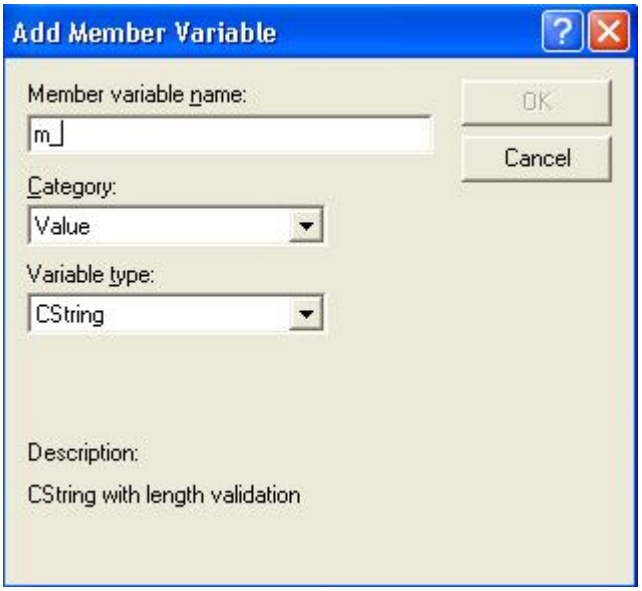
۱۰- پنجره کد را ببندید و در منوی Build گزینه Build (Project name).exe را انتخاب کنید تا برنامه کامپایل شود اگر برنامه اشکالی نداشت در منوی Build گزینه Execute (Project name).exe را انتخاب کنید تا برنامه اجرا شود. اکنون اگر دکمه Hello را کلیک کنید یک پیام با متن "Hello" نشان داده می‌شود.

کنترل کادر ویرایشی Edit Box :

با استفاده از این کنترل می‌توان اطلاعات را از ورودی خواند یا در خروجی نمایش داد. برای انجام عملیات روی کنترل کادر ویرایش ابتدا باید به آن یک متغیر نسبت داد و از این به بعد در طول برنامه از نام این متغیر باید استفاده کرد. برای تعیین نام و نوع متغیر برای یک کنترل کادر ویرایشی باید مراحل زیر را دنبال کنید:

۱- ابتدا از منوی file/new... یک پروژه کادر محاوره‌ایی ایجاد کنید و سپس یک کنترل Edit Box را به کادر محاوره اضافه کنید.

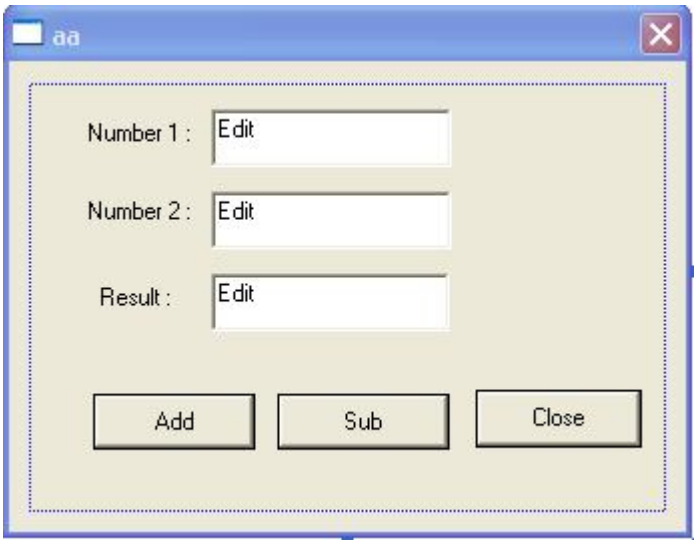
۲- روی کنترل Edit Box کلیک راست کرده، و از منوی ظاهر شده گزینه ClassWizard را انتخاب کنید سپس کادر محاوره MFC ClassWizard باز شده و پنجره Member Variable را انتخاب کنید. در بخش Control ID، کنترل کادر ویرایشی مورد نظر را انتخاب کنید دکمه Add Variables را کلیک کنید تا پنجره Add Member Variable همانند شکل ۸ ظاهر شود. در این پنجره می‌توان نام و نوع متغیر را تعیین کرد. برای کار با متغیرهای داده‌ایی مقدار Category را Value انتخاب کنید. از این به بعد هر وقت در برنامه بخواهیم مقداری را از این کنترل کادر ویرایشی بخوانیم یا مقداری را در آن به عنوان خروجی نمایش دهیم از نام متغیر مربوط به آن استفاده می‌کنیم.



شکل ۸

مثال: برنامه‌ای که دو عدد را از دو کادر ویرایشی گرفته و حاصل جمع و تفریق آن دو عدد را در یک کادر ویرایشی دیگر نمایش دهد.

۱- ابتدا سه کنترل متن ثابت (Label) را به کادر محاوره اضافه کنید و خاصیت کپشن آنها را به Number 1، Number 2 و Result تغییر دهید برای انجام این کار روی هر کدام از کنترل های متن ثابت کلیک راست کرده و گزینه Properties را انتخاب کنید و در پنجره باز شده خاصیت Caption آنها را به Number 1، Number 2 و Result تغییر دهید.



شکل ۹

۲- سه کنترل کادر ویرایشی را به کادر محاوره اضافه کنید (همانند شکل ۹) و سپس متغیرهای num1، num2 و result را به کنترل‌های IDC_EDIT1، IDC_EDIT2 و IDC_EDIT3 تخصیص دهید برای انجام این کار روی فرم کلیک راست کرده و گزینه ClassWizard را انتخاب کنید سپس در کادر محاوره MFC ClassWizard ظاهر شده، صفحه Member Variables را انتخاب کنید و در بخش Control IDs مکان نما را به IDC_EDIT1 انتقال داده، و دکمه Add Variable را کلیک کنید در کادر محاوره Add Member Variable نام متغیر را num1 و نوع آن را int انتخاب کنید مقدار category نیز باید Value انتخاب شود. در پایان دکمه Ok را کلیک کنید. برای دو کادر ویرایشی دیگر همین روند دنبال شود.

۳- یک دکمه به کادر محاوره اضافه کنید و خاصیت ID و Caption آن را به IDADD و Add تغییر دهید.

۴- تابع OnAdd مربوط به دکمه Add را به پروژه اضافه کنید. برای انجام این کار از منوی view گزینه ClassWizard را انتخاب کنید در کادر محاوره MFC ClassWizard ظاهر شده، صفحه Message

Maps را انتخاب کنید. در بخش‌های Object IDs: و Messages: به ترتیب گزینه‌های IDADD و BN_CLICKED را انتخاب کنید. سپس دکمه Add Function را کلیک کنید و در پنجره Add Member Function دکمه Ok را انتخاب کنید.

۵- تابع OnAdd به بخش Member Function اضافه خواهد شد آن را انتخاب و سپس دکمه Edit Code را کلیک کنید و دستورات زیر را در تابع تایپ کنید:

```
UpdateData(true);
result=num1+num2;
UpdateData(false);
```

۶- یک دکمه به کادر محاوره اضافه کنید و خاصیت ID و Caption آن را به IDSUB و Sub تغییر دهید تابع OnSub را همانند تابع OnAdd به پروژه اضافه کنید و دستورات زیر را در ان تایپ کنید:

```
UpdateData(true);
result=num1-num2;
UpdateData(false);
```

۷- پروژه را اجرا کنید و در جلوی Number 1 و Number 2 دو عدد را وارد کنید دکمه Add را کلیک کنید. مجموع آنها در جلوی Result مشاهده خواهد شد.

توجه: تابع ()UpdateData برای تبادل اطلاعات بین کنترل‌ها و متغیرهای عضو کنترل‌ها به کار می‌رود. اگر پارامتر آن True باشد سبب می‌شود داده‌ها از کنترل‌ها به متغیرها منتقل شوند و اگر پارامتر آن false باشد سبب می‌شود داده ها از متغیرها به کنترل‌ها منتقل شوند.

معرفی چند تابع مشترک بین کنترل ها

تابع ( )GetDlgItem

این تابع اشاره‌گری به کنترل مورد نظر بر می‌گرداند و بصورت زیر به کار می‌رود:

```
CWnd* GetDlgItem (int IDC);
```

IDC، خاصیت ID کنترلی است که باید اشاره‌گری به آن کنترل برگردانده شود با استفاده از این تابع می‌توان کنترل‌های کادر محاوره را دستکاری کرد.

تابع ( )ShowWindow

این تابع یک کنترل را مخفی یا آشکار می‌کند و بصورت زیر به کار می‌رود:

```
کنترل -> ShowWindow(SW_HIDE | SW_SHOW);
```

پارامتر SW_HIDE کنترل را مخفی می‌کند و پارامتر SW_SHOW آن را آشکار می‌کند.

```
GetDlgItem (IDC_EDIT) -> ShowWindow (SW_HIDE);
GetDlgItem (IDC_BUTTON) -> ShowWindow (SW_SHOW);
```

دستور اول، کنترل Edit را مخفی می‌کند و دستور دوم کنترل Button را آشکار می‌کند.

تابع EnableWindow( )

این تابع یک کنترل را فعال یا غیر فعال می‌کند و بصورت زیر به کار می‌رود:

```
کنترل->EnableWindow(TRUE | FALSE);
```

تابع SetWindowText( )

این تابع یک رشته را به یک کنترل نسبت می‌دهد و بصورت زیر بکار می‌رود:

```
کنترل->SetWindowText(CString str);
```

به دو طریق می‌توان ار توابع ذکر شده استفاده کرد:

۱- استفاده از اشاره‌گر (تابع GetDlgItem()) که در دستورات قبل معرفی شد و از علامت **->** استفاده می‌شود.

۲- استفاده از متغیر نسبت داده شده به کنترل مورد نظر. یعنی همانند نسبت دادن متغیر به کادر ویرایشی همان روند را دنبال می‌کنیم (شکل ۸) ولی این بار در بخش Category، به جای مقدار Value مقدار Control را انتخاب می‌کنیم و در بخش Variable type نوع کنترل را تعیین می‌کنیم.

```
GetDlgItem (IDC_EDIT1) -> SetWindowText ("test");
str1.SetWindowText ("test");
```

دو دستور بالا یکسان عمل می‌کنند. هر دو یک مقدار رشته‌ای را در کنترل کادر ویرایشی نمایش می‌دهند دستور اول از اشاره‌گر استفاده می‌کند. در دستور دوم str1 متغیری از نوع CEdit می‌باشد.

تابع GetWindowText( )

این تابع رشته موجود در کنترل را برمی‌گرداند و بصورت زیر به کار می‌رود:

```
نام کنترل.GetWindowText (CString str);
```

پارامتر str رشته خوانده شده از کنترل را نگهداری می‌کند.

کنترل کادر لیست ListBox:

با استفاده از این کنترل می‌توان لیستی را ایجاد کرد انتخاب عناصر لیست از طریق صفحه کلید یا کلیک کردن دکمه ماوس بر روی آنها انجام می‌شود کادر لیست می‌تواند یک انتخابی یا چند انتخابی باشد. برای استفاده از ListBox باید متغیری را به آن تخصیص داد. برای انجام این کار روی آن کلیک راست کرده و گزینه ClassWizard را انتخاب کنید و در صفحه Member Variables شماره ID مربوط به ListBox را انتخاب کنید. دکمه Add Variable را انتخاب کنید نام متغیر را تایپ و نوع آن را CListBox انتخاب کنید.

توابعی که در ListBox به کار می‌روند:

تابع ( )AddString : برای اضافه کردن گزینه‌ای به ListBox به کار می‌رود و بصورت زیر استفاده می‌شود:

نام متغیر از نوع ListBox.AddString(“گزینه‌ای که باید اضافه شود”)

```
List1.AddString(“item1”);
List1.AddString(“item2”);
```

این دستورات گزینه‌های item1 و item2 را به کنترل List1 اضافه می‌کنند.

AddString موقعیت جدید کادر لیست را برمی‌گرداند (مکان بعدی که گزینه جدید باید اضافه شود).

تابع ( )InsertString : با این تابع می‌توان گزینه‌ای را در نقطه مشخصی از ListBox اضافه کرد و بصورت زیر استفاده می‌شود:

نام متغیر از نوع ListBox. InsertString(Index, “گزینه ای که باید اضافه شود”);

تابع ( )DeleteString : برای حذف کردن گزینه‌ای از ListBox به کار می‌رود و بصورت زیر استفاده می‌شود:

نام متغیر از نوع ListBox.DeleteString(Index);

تابع ( )ResetContent : این تابع تمام گزینه‌های ListBox را حذف می‌کند و بصورت زیر استفاده می‌شود:

نام متغیر از نوع ListBox.ResetContent ();

تابع ( )GetCount : این تابع تعداد گزینه‌های ListBox را برمی‌گرداند و بصورت زیر استفاده می‌شود:

نام متغیر از نوع ListBox.GetCount();

تابع ( )GetCurSel : این تابع ایندکس گزینه‌ای را برمی‌گرداند که انتخاب شده است و بصورت زیر استفاده می‌شود:

نام متغیر از نوع ListBox.GetCurSel();

اگر هیچ گزینه‌ای انتخاب نشود یا حالت چند انتخابی باشد تابع مقدار LB_ERR را برمی‌گرداند.

```
Int select = List1.GetCurSel();
```

تابع GetText( ) : این تابع اطلاعات گزینه‌ای از ListBox را برمی‌گرداند و بصورت زیر استفاده می‌شود:

نام متغیر از نوع ListBox.GetText(Index, CStr);

Index شماره گزینه‌ای است که اطلاعات آن باید برگردانده شود و CStr مکانی است که اطلاعات بازیابی شده در آن قرار می‌گیرد.

کنترل CheckBox

این کنترل گزینه‌هایی با دو انتخاب تعریف می‌کند برای استفاده از این کنترل باید یک متغیر از نوع Bool را به این کنترل تخصیص داد اگر این کنترل انتخاب شده باشد مقدار این متغیر TRUE خواهد بود در غیر اینصورت مقدار این متغیر FALSE میشود

کنترل Radio Button

این کنترل برای ایجاد مجموعه‌ای از گزینه‌های ناسازگار به کار می‌رود در این مجموعه فقط یک گزینه قابل انتخاب است برای استفاده از این کنترل باید یک متغیر از نوع int در برنامه تعریف کرد و به کنترل نسبت داد و در محیط کد نویسی مربوط به Radio Button از این متغیر استفاده کرد. و براساس مقدار متغیر فوق تصمیم گرفت که آیا این متغیر فعال است یا خیر. یعنی به ازای هر کنترل Radio Button که به کادر محاوره اضافه می‌شود در قسمت کد نویسی مربوط به آن کنترل (با دو بار کلیک کردن روی کنترل وارد محیط کد نویسی مربوط به آن کنترل می‌شوید) مقدار متغیر مربوط به آن را تعیین می‌کنیم سپس در برنامه براساس این مقدار، عمل مورد نظر را انجام می‌دهیم.