اشاره گر this

- توابع عضو هر شـــی به یک اشاره گر خاص به نام this دســـتـــرسی دارند که به خـــود شی اشاره می کند.
- معنای وجود این متغیر در کلاس یعنی هر شی آدرس خود در خافظه را می داند و یا تعبیر دیگر اینکه هر شـــی خـــودش را می بیند و خودش را می شناسد.

this مثالی اشاره گر

```
class where
 {
   private:
     char charArr[10];
   public:
void reveal()
{
     cout<< "\nMy object's address is:"<<this;
}
};
```

مثالی اشاره گر this

```
void main()
{
    where w1,w2;
    w1.reveal();
    w2.reveal();
    cout <<"w1 Address is "<<&w1;
    cout <<"w2 Address is "<<&w2;
}
```

کاربردهای دیگر اشاره گر this

- یکی از پرکاربردترین کاربردها زمانی است که می خواهیم خروجی یک تابع عضو خود شی و یا آدرس خود شی باشد.
- کاربرد دیگر زمانی است که می خواهیم خود شی را به عنوان آرگومان یک تابع ارسال نماییم
- فرض کنید تابعی داریم که یک شی را گرفته و اطلاعات آن را نمایش دهد. ما می خواهیم همین تابع نمایش دادن بصورت یک تابع عضو کلاس نیز باشد. برای استفاده از تابع موجود می توانیم از اشاره گر this استفاده نماییم.

فصل سوم

وراثت (Inheritance)

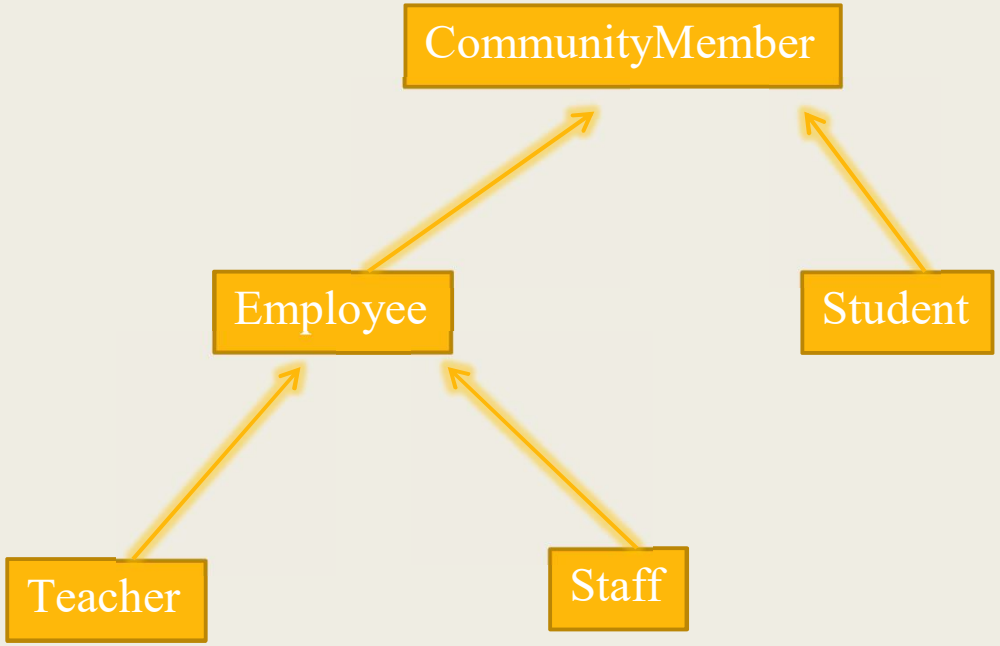
وراثت – مقدمه

- **تعریف:**
  - **فرایند ایجاد کلاس های جدید(مشتق) از کلاس موجود(پایه) است.**
- **ویژگی ها:**
  - **کلاس مشتق شده تمام توانایی های کلاس پـــایه را به ارث می برد اما می تواند اعضای دیگری نیز داشته باشد.**
  - **کلاس پایه بدون تغییر باقی می ماند.**
  - **صرفه جویی در وقت و افزایش اطمینان برنامه**
  - **قابلیت استفاده مجدد در توزیع آسان کتابخانه های کلاس**

کلاس مشتق – کلاس پایه

- **مثال:**

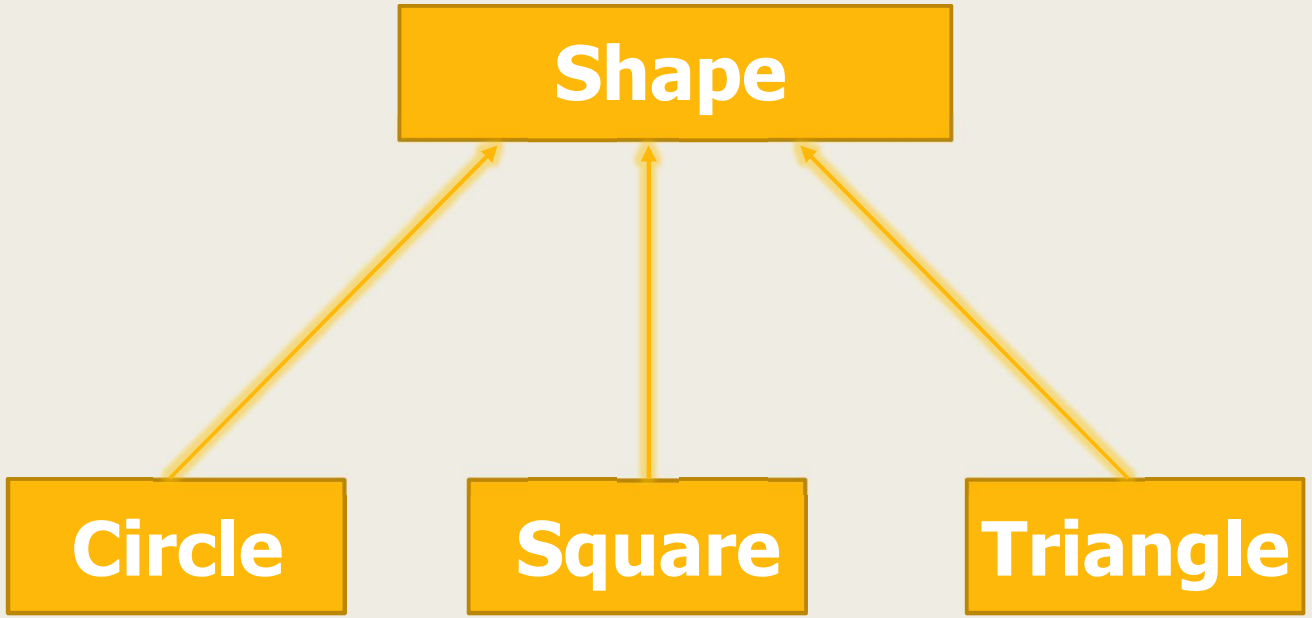
این یک نمونه از وراثت در جامعه دانشگاه است که کلاس های دانشجو و کارمندان از کلاس اعضای جامعه مشتق شده اند و کلاس های استاد و کارمندان اداری از کلاس کارمندان مشتق شده اند.



کلاس مشتق – کلاس پایه

- **مثال:**

این نمونه ای از وراثت در کلاس اشکال است که در آن کلاس های دایره و مثلث و مستطیل از این کلاس مشتق شده اند.



نحوه تعریف کلاس مشتق از کلاس پایه

```
class DerivedClass: public BaseClass
{
        //class members
};
```

نام کلاس مشتق نوع ارث بری نام کلاس پایه

- **مثال:**

```
class Triangle: public Shape
{
        //class members
};
```

نحوه تعریف کلاس مشتق از کلاس پایه– مثال

```
class counter
{
   protected:
      unsigned int count;
   public:
      counter():count(0){};
      counter(int c):count(c){};
      unsigned int get_count() const{return count;}
      counter operator++(){return
counter(++count);}
};
```

نحوه تعریف کلاس مشتق از کلاس پایه– مثال

```
class countDn:public counter
{
public:
      counter operator--(){return counter(--
count);}
};
```

- کد بالا نشان می دهد که این کلاس جدید با دسترسی عمومی از کلاس اول مشتق شده است.
- توابع عضو کلاس پایه را می توان به وسیله اشیاء کلاس مشتق مورد دسترسی قــرار داد که البته این به قابلیت دسترسی کلاس مشتق باز می گردد.

ویژگی های کلاس مشتق

- همانطور که اشاره شد هنگامی که یک کلاس از یک کلاس پایه مشتق می شود باعث پرهیز از دوباره کاری می گردد زیرا تمامی رفتارها و ویژگی های موجود در کلاس پایه به کلاس مشتق شده انتقال می یابد
- توجه داشته باشید که وقتی شما کلاس جدیدی را از یک کلاس پایه مشتق می کنید همانند گذشته هیچ گونه شی خاصی ساخته نمی شود و تنها کلاس تعریف می گردد
- کلاس مشتق شده می تواند رفتارها و ویژگی های دیگری نیز علاوه بر آنچه که از والد خود به ارث برده است داشته باشد.
- مثلاً در مثال شمارنده کلاس مشتق شده یک تابع با نام operator-- اضافه تر از آنچه که از کلاس والد به ارث برده، دارد

انواع ارث بری

- در مثال های قبل نوع ارث بری مشخص شده از نوع public می باشد.
- انواع ارث بری که می توان برای ساخت کلاس مشتق استفاده کرد عبارت اند از:
  - public
  - private
  - protected

انواع ارث بری

- public: در این نوع ارث بری تمامی متغیرها و توابع عضو public و protected کلاس پایه به همان شکل در کلاس مشتق شده انتقال می یابند
- protected: در این نوع ارث بری تمامی متغیرها و توابع عضو public و protected کلاس پایه به شکل protected به کلاس مشتق شده انتقال می یابند
- private: در این نوع ارث بری تمامی متغیرها و توابع عضو public و protected کلاس پایه به شکل private به کلاس مشتق شده انتقال می یابند

انواع ارث بری

- قابلیت دسترسی به متغیرهای ارث برده شده از کلاس پایه در کلاس مشتق با توجه به نوع ارث بری

Access specify	Accessible from Own Class	Accessible from Derived Class	Accessible from Object Outside Class
Public	yes	yes	yes
Protected	yes	yes	no
Private	yes	no	no

انواع ارث بری

- تا به حال تنها از حالت ارث بری عمومی استفاده کردیم حال اگر از حالت ارث بری خصوصی استفاده کنیم اشیاء کلاس مشتق دیگر نمی توانند به توابع عضو عمومی کلاس پایه دسترسی پیدا کنند مثال:

```
class baseClass
{
   public:
         void setNumber( int x, int y);
         void show();
   private :
         int num1;
         int num2;
};
```

انواع ارث بری

```
void baseClass::setNumber(int x, int y)
{
       num1 = x;
       num2 = y;
}
//*****************
void baseClass::show()
{
       cout << "num1 = " << num1 << ", num2 = " << num2
<< endl;
}
class derivedClass : private baseClass
{
    private :
              int derivedNum;
    public :
              derivedClass(int number);
              void showDerivedNum();
};
```

انواع ارث بری

```
derivedClass::derivedClass(int number)
{
	derivedNum = number;
}
//********************
void derivedClass::showDerivedNum()
{
	cout << "derivedNum = " << derivedNum << endl;
}
int main()
{
	derivedClass derivedObject(100);
	cout << "Accsee to baseClass members : \n";
	derivedObject.setNumber(200, 300); //access member
of baseClass
	derivedObject.show();		//access member of
baseClass
	cout << "\nUses the derivedClass member : \n";
	derivedObject.showDerivedNum();	//uses member of
derivedClass
	cin.get();
}
```

انواع ارث بری – متغیرهای عضو private

- همانگونه که از متغیرهای private عضو یک کلاس انتظار داریم، این متغیرها باید از دسترس تمامی کلاس های دیگر مخفی بماند
- در ارث بری متغیر عضو private به کلاس مشتق شده انتقال می یابد ولی کلاس مشتق شده نمی تواند بطور مستقیم با متغیر کار کند و باید از طریق دیگر توابع عمومی که از کلاس پایه گرفته مقدار عضو را تغییر دهد.

انواع ارث بری – متغیرهای عضو private

```
class baseClass
{
   public:
          void setNumber( int x, int y);
          void show();
   protected :
          int num1;
          int num2;
};
void baseClass::setNumber(int x, int y)
{
       num1 = x;
       num2 = y;
}
//*****************
void baseClass::show()
{
       cout << "num1 = " << num1 << ", num2 = " << num2
<< endl;
}
```

انواع ارث بری – متغیرهای عضو private

```
class derivedClass : public baseClass
{
    private :
              int derivedNum;
    public :
              void setDeriveNum();
              void showDerivedNum();
};


void derivedClass::setDeriveNum()
{
       derivedNum = num1 * num2; //dervideClass access
baseClass' num1, num2
}
//********************
void derivedClass::showDerivedNum()
{
       cout << "derivedNum = " << derivedNum << endl;
}
```

انواع ارث بری – متغیرهای عضو private

```
int main()
{
	derivedClass derivedObject;
	cout << "Accsee to baseClass members : \n";
	derivedObject.setNumber(200, 300); //access member
of baseClass
	derivedObject.show();	//access member of
baseClass
	cout << "\nUses the derivedClass member : \n";
	derivedObject.setDeriveNum();	//use member
of derivedClass
	derivedObject.showDerivedNum();	//uses member
of derivedClass
	cin.get();
}
```

سازنده ها و مخرب ها در کلاس های مشتق

- چون کلاس مشتق، اعضای کلاس پایه را به ارث می برد، وقتی شیئی از کلاس مشتق ایجاد می شود، سازنده کلاس پایه باید فراخوانی شود تا اعضای کلاس پایه ای را که در شیء مشتق وجود دارند، مقداردهی اولیه نماید.
- دو سوال؟؟؟؟
  - سازنده و مخرب کلاس پایه و مشتق چه زمانی اجرا می شوند؟
  - چگونه می توان پارامتری را به تابع سازنده کلاس پایه ارسال کرد؟

سازنده ها و مخرب ها در کلاس های مشتق

```
class baseClass
{
  public :
        baseClass();
        ~baseClass();
};
baseClass::baseClass()
{
       cout << "Constructing baseClass object." << endl;
}
//******************
baseClass::~baseClass()
{
       cout << "Destructing baseClass object." << endl;
       cin.get();
}
```

سازنده ها و مخرب ها در کلاس های مشتق

```
class derivedClass : public baseClass
{
  public:
     derivedClass();
     ~derivedClass();
};
derivedClass::derivedClass()
{
       cout << "Constructing derivedClass object ." <<
endl;
}
//************************
derivedClass::~derivedClass()
{
       cout << "Destructing derivedClass object ." <<
endl;
}
```

سازنده ها و مخرب ها در کلاس های مشتق

```
class derivedClass : public baseClass
{
  public:
     derivedClass();
     ~derivedClass();
};
derivedClass::derivedClass()
{
       cout << "Constructing derivedClass object ." <<
endl;
}
//************************
derivedClass::~derivedClass()
{
       cout << "Destructing derivedClass object ." << endl;
}
//************************************************************
******
int main()
{
       //creat a derivedClass object
       derivedClass derivedObject;
}
```

ارسال پارامتر به سازنده های کلاس پایه

- اگر سازنده کلاس پایه آرگومانی را به عنوان ورودی دریافت کند (کلاس پایه سازنده پیش فرض نداشته باشد)
- حتماً باید این ورودی در سازنده کلاس مشتق دریافت شده و سازنده کلاس پایه فراخوانی شود
- نحوه فراخوانی سازنده کلاس پایه به صورت زیر، از (:) استفاده شود.

```
Derived::Derived(int g):Base(g)
{
        ....
}
```

سازنده ها و مخرب ها در کلاس های مشتق

```
class derivedClass : baseClass
{
   public:
          //a is used by derivedClass and b sent to
baseClass
         derivedClass(int a, int b) : baseClass(b)
         {
                number2 = a;
                cout << "Constructing derivedClass " <<
endl;
         }
         //******************
         ~derivedClass();
         void showData();
   private :
          int number2;
};
derivedClass::~derivedClass(void)
{
   cout << "\nDestructing derivedClass .  " << endl;
}
```

سازنده ها و مخرب ها در کلاس های مشتق

```
void derivedClass::showData()
{
	cout << endl << "in baseClass number1 = " <<
number1 << endl
		<< "in derivedClass number2 = " << number2
<< endl;
}

int main()
{
	//100 used in derivedClass and 200 sent to
baseClass
	derivedClass derivedObject(100, 200);
	derivedObject.showData();
}
```

تبدیل اشاره گر کلاس پایه به اشاره گر کلاس مشتق

- با شیئی از کلاس مشتق می توان به صورت شیئی از کلاس پایه رفتار کرد.
- این کار امکاناتی را به وجود می آورد؛ مثلا علی رغم این که اشـــیایی از کلاس های مختلف که از یک کلاس پایه مشتق شده اند، می توانند با هم متفاوت باشند، آن ها را می توان در یک لیست پیوندی قرار داد.

تبدیل اشاره گر کلاس پایه به اشاره گر کلاس مشتق

```
class point
{
        friend ostream &operator<<(ostream &, const point
&);
  public:
     point(int = 0, int = 0);   //default constructor
     void setPoint(int, int);  //set coordinates
     int getx() const;  //get x coordinate
     int gety() const;    //get y coordinate
  protected:                          //accessible by
derived class
     int x, y;
};
int point:: getx() const  //get x coordinate
{
       return x;
}
int point::gety() const    //get y coordinate
{
       return y;
}
```

تبدیل اشاره گر کلاس پایه به اشاره گر کلاس مشتق

```
point::point(int a, int b)
{
       setPoint (a, b);
}
void point::setPoint(int a, int b)
{
   x = a;
   y = b;
}
//outpot point with overloaded <<
ostream &operator<<(ostream &output, const point &p)
{
  output << '[' << p.x << ", " << p.y << ']';
 // output <<" In point  << " << endl;
  return output;
}
```

تبدیل اشاره گر کلاس پایه به اشاره گر کلاس مشتق

```
class circle : public point
{
     friend ostream &operator<<( ostream &, const circle
&);
   public:
      circle(double r = 0.0, int x = 0, int y = 0);
      void setRadius(double);   //set radius
      double getRadius() const; //return radius
      double area() const;      //calculate area
   protected:
      double radius;

};


circle::circle(double r, int a, int b) : point(a, b)
//call base class constructor
{
  setRadius(r);
}
```

تبدیل اشاره گر کلاس پایه به اشاره گر کلاس مشتق

```
void circle::setRadius(double r)
{
   radius = (r >= 0 ? r : 0);
}
//****get radius of circle
double circle::getRadius() const
{
   return radius;
}
//calculate area of circle
double circle::area() const
{
   return 3.14159 * radius * radius;
}
//**output a circle in the form:
//center = [x, y]; radius = #.##
ostream &operator<<( ostream &output, const circle &c)
{
   output << "Center = " << static_cast < point >( c )
          << "; Radius = " << c.radius;
   return output;
}
```

تبدیل اشاره گر کلاس پایه به اشاره گر کلاس مشتق

```
int main()
{
  point *pointPtr = 0, p(30, 50);
  circle *circlePtr = 0, c(2.7, 120, 89);

  cout << "point p: " << p << endl
          << "\circle c: " << c << endl;
  //treat a circle as a point(see only the base class part)

  pointPtr = &c; //assign address of circle to pointPtr
  cout << "\ncircle c (via *pointPtr): "
         << (*pointPtr) << '\n';

  //treat a circle as a circle(with some casting)
  pointPtr = &c;
  //cast base-class pointer to derived-class pointer
  circlePtr = static_cast<circle *>(pointPtr);
  cout << "\ncircle c(via *circleptr) :\n" << *circlePtr
       << "\nArea of c(via circlePtr): "
       << circlePtr -> area() << endl;

  //DANGEROUS: treat a point as a circle
  pointPtr = &p;
  //cast base-class pointer to derived-class pointer
  circlePtr = static_cast<circle *>(pointPtr);
}
```

هم پوشانی توابع عضو

- ویژگی منحصر به فرد دیگری که وراثت امکان آن را به ما می دهد هم پوشانی توابع عضو کلاس های پایه و مشتق است به این معنی که در هر دو کلاس یک تابع با یک اسم مشترک داریم برای مثال در مثال اشکال هم کلاس پایه و همه ی کلاس های مشتق آن، تابع ترسیم را دارند و از آنجایی که اشیاء کلاس مشتق قابلیت صدا زدن توابع عضو کلاس پایه را دارند این سوال پیش می آید که کدام تابع فراخوانی می شود؟

```
class Base
{
public:
    void Fun();
    Base();
    virtual ~Base();
};
```

```
class Derived : public Base
{
public:
    void Fun();
    Derived();
    virtual ~Derived();
};
```

هم پوشانی توابع عضو

- **که اگر مثل قدیم توابع را تعریف کنیم تابع کلاس مشتق فراخوانده می شود (مثال بالا). اگر الگوی دو تابع یکسان نباشد، به جای اینکه تابع جایگزین شود، تعریف مجدد (Override) می شود. اما اگر در کلاس مشتق در هنگام تعریف تابع از عملگر تعیین حوزه (::) استفاده کنیم تابع ازکلاس پایه فراخوانده می شود!**

```
class Base
{
public:
    void Fun();
    Base();
    virtual ~Base();
};
```

```
class Derived : public Base
{
public:
    void Fun();
    Derived();
    virtual ~Derived();
};
```

هم پوشانی توابع عضو

```
class Employee
{
    public:
              Employee(string first, string last);
              Employee();
              void setName();
              void showName();
              void setSalary();
    private :
          string firstName;
          string lastName;
};
Employee::Employee() //default constructor
{
}
Employee::Employee(string first, string last)
{
       firstName = first;
       lastName = last;
}
```

هم پوشانی توابع عضو

```
void Employee::setName ()
{
       cout << "Enter first name, last name : " ;
       cin >> firstName >> lastName;
       cin.ignore();
}
//**********************
void Employee::showName ()
{
       cout << "first name is : " << firstName
              << " , last name is : " << lastName <<
endl;
}
```

هم پوشانی توابع عضو

```
class hourlyEmployee : public Employee
{
    public:
           hourlyEmployee(string first, string last, int
h, int hp );
              hourlyEmployee();
              void showName();
              void setName();
    private :
              int hours;
              int hourPay;
};
hourlyEmployee::hourlyEmployee()  //default constructor
{
}
```

هم پوشانی توابع عضو

```
hourlyEmployee::hourlyEmployee(string first, string last,
int h, int hp)
       :Employee(first, last) //call base class constructor
{
    hours = h;
       hourPay = hp;
}

void hourlyEmployee::setName()
{
   Employee::setName();
   cout << "hours and hourPay : " ;
   cin >> hours >> hourPay;
   cin.ignore ();
}

void hourlyEmployee::showName()
{
       Employee::showName();
       cout << "Wage = " << hours * hourPay << endl;
}
```

هم پوشانی توابع عضو

```
int main()
{
   Employee emp1("Ali ", "Ahmadi");
   Employee emp2;

   hourlyEmployee hourEmp1("Reza" , "Razavi", 50, 20000);
   hourlyEmployee hourEmp2;
   cout << "Employee :";
   emp2.setName();
   cout << "hourly employee :";
   hourEmp2.setName();

   cout << "\nTwo employees are : " << endl;
   emp1.showName();
   emp2.showName();

   cout << "\nTwo hourly employees are : " << endl;
   hourEmp1.showName();
   hourEmp2.showName();
}
```

هم پوشانی توابع عضو

- در مثال بالا تابع مربوطه با عملگر (::) از کلاس پایه صدا زده شد که البته اگر ورودی داشته باشد به همان روال عادی ورودی ها را نیز در آن می نویسیم.
- البته این بحث این جا به پایان نمی رسد و در فصل بعد به آن بازمی گردیم و پیرامون خاصیت اصلی آن در فصل توابع مجازی بحث می کنیم.