Subject:
Year. Month. Date. ( )

```
# include <iostream.h>
class Time
{
public:          // می توانند ورودی داشته باشند
Time ( );        // هم اسم رده و نوع برگشتی ندارند
void set time (int, int, int);
void print military ( );
void print standard ( );
private:
int hour;
int minute;
int second;
};
```

default در class ، private است

در هر رده تابع عضو سازنده داشته باشیم که شی تعریف باشد

(سازنده - نابودکننده) ها هیچ کدام نوع برگشتی ندارند

نحوه تعریف نابودکننده ~ اسم رده ، به هیچ نوع ورودی ندارند

این توابع به صورت اتوماتیک سازنده ،

فراخوانی می شوند

سازنده بطور اتوماتیک وقتی شی از آن نوع تعریف می شود

فراخوانی می شود

اگر داخل یک بلوک یک شی local داشته باشیم قبل از پایان

بلوک (منظور جایی است که از آن بلوک خارج می شویم)

نابودکننده فراخوانی می شود

سازنده سه مقدار دهی اولیه به صورت default (صفر می گیرد)

```
Time ~~t1 (6, 20, 30)~~;

Time :: Time ( )
{
hour = minute = second = 0;
}
```

Time :: → نشان می دهد عضوی از رده Time است

P4PCO

struct مثل Type جدید تعریف می‌کنیم که
یک type جدید است

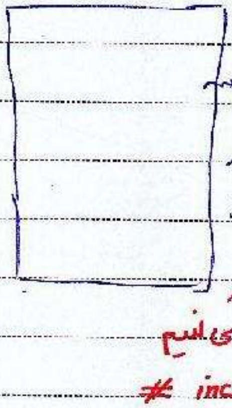
class در C++ مثل struct است در C

برای مقدار دادن اولیه به object ها باید تابع سازنده استفاده می‌شود
که به صورت Automatic صدا زده می‌شود

اگر در class یک اشاره‌گر داشته باشیم حافظه در زمان تعریف به آن تخصیص داده شده و این
کار سازنده انجام می‌دهد
نابودکننده: برای این است که وقتی حافظه‌ای به object اختصاص داده شده بود نابود کنند اول حافظه
آزاد می‌شود و بعد شیء نابود می‌شود

وقتی برنامه‌ها کمی بزرگتر شود کدهای نوشته شده را جدا می‌کنند
معمولاً Type های که توسط کاربر تعریف می‌شود در file دیگری (header file) تعریف می‌کنیم و
پسوند آن .h است علاوه بر تعاریف prototype توابع نیز در این جا نوشته می‌شود

prog.h



اسم توابع و Type های که تعریف شده در اختیار می‌گذارد بدون جزئیات

می‌توان از یک فایل cpp یک فایل obj
ساخت و همراه (prog.h) (header file)
در اختیار کاربر گذاشت بدون اینکه جزئیات برنامه و
source برنامه را در اختیار داشته باشد

header file که خودمان تعریف می‌کنیم به این صورت include می‌کنیم
include "prog.h"
header file های برنامه را به این صورت
#include<iostream.h>

Subject:

Year. Month. Date. ( )

```
// time1.h
#ifndef TIME1_H
#define TIME1_H
class Time
{
   public:
   Time();
   void SetTime(int, int, int);
   void printMilitary();
   void printStandard();
  private:
   int hour;
   int minute;
   int second;
};
#endif
```

یک header file است.

وقتی با # شروع می شود دستور به پیش پردازنده است

اگر این const define نشده آن را define کن (تعریف const)

اگر قبلاً define شده بود نشده تعریف نکن

interface

ifdef ≡ # endif

اگر if نکرده بودیم بین دستور endif آن را ببندیم

PAPCO

Subject:
Year. Month. Date. ( )

```
// time1.cpp
#include <iostream.h>
#include "time1.h"
```

نشان می‌دهد که این توابع جزئی از class هستند

```
Time::Time() { hour = minute = second = 0; }
```

تابع سازنده نوع برگشتی ندارد.

```
void Time::setTime(int h, int m, int s)
{
  hour = ((h >= 0 && h < 24) ? h : 0);
  minute = ((m >= 0 && m < 60) ? m : 0);
  second = ((s >= 0 && s < 60) ? s : 0);
}
```

سازنده پیش‌فرض همه مقادیر صفر ندارد، اگر داشته باشد مقادیر پیش‌فرض می‌گیرد

```
Time t(12, 1, 1)
```

تعریف سازنده

```
Time::Time(int h, int m, int s)
    {hour = h; minute = m; second = s;}
```

PAPCO

Subject:
Year. Month. Date. ( )

Data ها را private تعریف می‌کنیم

```
class Time
{
  public
  Time(int =0, int=0, int=0);
  void  setTime(int,int,int);
  void  setHour(int);
  void  setMinute(int);
  void  setSecond(int);
  int  getHour();
  int  getminute();
  int  getSecond();
  void  printmilitary();
  void  printstandard();
  private
  int hour, minute, second;
};

Time::Time(int h, int m, int s)
{  setTime(h,m,s); }

void  Time::SetTime(int h, int m, int s)
{
  SetHour(h);
  SetMinute(m);
  SetSecond(s);
}
```

— برای همان چیزی که در بالا فراخوانی شده
فراخوانی میشود

PAPCO

Subject:
Year. Month. Date. ( )

تمرین ۱۵ و ۴

```
void SetHour (int h)
{ hour=(h >= 0 && h<=23) ? h : 0; }
```

hour هر زمانی که نتایج برای آن نامعتبر شده handle نشد

این مقدار دقیق این فانکشن

```
void incminute (Time &t, const int count)
{
    for (int i=0; i<count; i++) {
        t.SetMinute((t.getminute()+1) % 60);
        if (t.getminute() == 0)
            t.setHour(t.getHour()+1);
    }
}
```

مقدار minute اگر بیشتر

بطور مستقیم به member که به صورت private تعریف شده نمی‌توان دسترسی داشت

```
Time::
int & getHourRef()
{ return hour
}
```

متغیر hour با کلاس class داریم

Refrence یعنی همین را بر می‌گرداند

چون که با آدرس بر می‌گرداند

```
int main()
{ Time t;
  t.getHourRef() = 5;
  t.Hour
```

اگر یکی از کلاس class یا دو متغیر از یک struct را داشته باشیم می‌توانیم مساوی قرار دهیم

به این صورت است که تمام عضوها یکی یکی copy می‌شود

```
Time t1, t2;
t1 = t2;
```

فصل 7 :

می‌توان object را هم مانند متغیرها به صورت const تعریف کرد

const Time t(12,0,0); ← این سه متغیر به تابع سازنده پاس می‌شود

~~t.setHour(7,20,26);~~

① نمی‌شود توابع عضو را برای یک شیء ثابت فراخوانی کرد مگر اینکه خود توابع نیز const باشند

2. توابع ثابت نمی‌توانند شیء را تغییر دهند مثلاً اگر minute را کم و زیاد کنند
آن هیچ تغییری نمی‌دهد

```
class Time
{
public :
    Time (int = 0, int = 0, int = 0);
    void SetHour(int);
    void Set minute(int);
    void SetSecond(int);
    int getHour() const;
};
```

← تابع ثابت به این صورت تعریف می‌شود

در هنگام پیاده‌سازی نیز باید کلمه const را بیاوریم

```
int Time::getHour() const
{ return hour; }

int main()
{ Time t1(6,45,0);
  const t2(12,0,0);
  t1.SetHour(7);
```

شیء غیرثابت - تابع غیرثابت

Subject.

Year. Month. Date. ( )

```
t1.getHour(1);
X t2.setHour(8);
t2.getHour(1);
return 0;
}
```

error : compiler کلید

Subject:

Year. Month. Date. ( )

طریقه‌ی مقداردهی refrence / const/static data member

بقیه Data member ها را می‌توانیم در سازنده مقداردهی کنیم ولی این

سه Data member را فقط به این طریق می‌توانیم مقداردهی کنیم

```
class Test
{
 private:
   Test (int, int)
   const int x;
   int y;
};
```

یکی از Data member ها مشترک است

Data member ها را با این طریق سازنده مقداردهی می‌کنیم

```
Test::Test (int a, int b) :X(a)
{
   x=a;
   y=b;
```

به این صورت به const Data member مقداردهی کنیم — تعیین کننده مقدار اولیه عضو (member initializer)

چون const است نمی‌توانیم به این صورت بنویسیم (x=a)

ترکیب: زمانی که از اشیای چند کلاس در یک کلاس استفاده کنیم

```
class Date
{
 ===
};
class Employee
{ private:
   Date d1;
   int x;
  public:
   employee(int, int, int, int)
};
```

عضو های این کلاس می‌توانند اشیای از یک کلاس دیگر باشند

به تعداد value ها باید پارامتر تعریف کنیم

اگر یک class از یک class دیگر استفاده کرده بودیم در تابع سازنده باید به تعداد value های خودش و value های class که در داخل آن تعریف شده پارامتر بنویسیم

```
employee::employee (int a, int b, int c, int d) : d1(a, b, c)
{
   x = d;
}
```

سه پارامتر اول را به سازنده d1 پاس می‌دهیم

Subject:

Year. Month. Date. ( )

توابع دوست

توابعی که عضوی از رده نیستند ولی می‌توانند به صورت مستقیم به Data member های رده دسترسی داشته باشند

خارج از رده

```
void Inc(Test &x) →
{
    x.Data++;
}
```

این x یک متغیر از کلاس Test است

این تابع عضوی از رده نیست پس به Data private دسترسی ندارد

این خط خطا است چون به Data دسترسی ندارد

در حالی که توابع دوست تعریف شده می‌توانند به داده‌های private در داخل class دسترسی داشته باشند

```
class Test
    friend void Inc(Test &); →
{ public:
      Test() { Data=0 }
  private
      int Data;
}

int main()
{ Test t;
  t.Data = 5;
  Inc(t)
  return 0;
}
```

(the line `t.Data = 5;` is crossed out)

برای دوست کردن تابع با کلاس، آن را به صورت دوست تعریف می‌کنیم

یک رده را می‌توان به عنوان یک دوست برای یک رده دیگر تعریف کرد. در این صورت تمام توابع عضو رده دوست به Data member های آن رده می‌توانند دسترسی داشته باشند.

هر رده‌ای دوست خودش است.

PAPCO

Subject:
Year. Month. Date. ( )

اشاره‌گر this

هر تابع عضوی یک اشاره‌گر this به طور ضمنی دریافت می‌کند

data++ = this → data++

وقتی تابع در محدوده فضای شیء و با x ... this آدرس x را در اختیار دارد

→ this را ما نمی‌توانیم تعریف کنیم چون خود compiler به طور ضمنی این را دریافت می‌کند

کاربرد: چند تابع را می‌خواهیم پشت هم call کنیم، به هم بچسبانیم (concat)

```
class Test
{ public:
      Test( ) {x = 0}
Test & print( );
   void BAC( ).
   private:

      int x;

 };
Test & Test::print( )
        { cout << x ;  & return *this; }
int main( )

{

   Test t;

  (t.print( )).print( );
```

به همین دلیل می‌خواهیم دو تا یا چند تابع را به هم بچسبانیم (این concat کردن توابع است)

```
}
```

cout

```
cout << "max = " << max << endl;
```

cout یک refrence

Subject.
Year. Month. Date. ( )

گرانبار کردن عملگرها (operator overloading)

```
class complex
{

};
```

$c = (a,b) = a + bi$

$c_1 + c_2 = (a_1,b_1) + (a_2,b_2) = (a_1 + a_2 , b_1 + b_2)$ ← جمع دو عدد مختلط

برای اینکه بتوانیم دو عدد مختلط را باهم جمع کنیم باید operator + را برای آن تعریف کنیم

این operator ها احتیاجی به overloading ندارند ← $c_1 = c_2$; & c_1

عملگرهایی که در سیستم هست را می‌توانیم overload کنیم (مثلاً #)

تعداد عملوندهای یک operator را نمی‌توانیم عوض کنیم

operator ها هم مثل function های دیگر به صورت عضو یا دوست تعریف می‌کنیم

عملگرها → دودویی (دو تا عملوند می‌گیرند) / یکانی (یک عملوند می‌گیرند)

+ ☐ می‌توانیم از دو عملوند استفاده کنیم که این عملوند اول ... تعریف می‌کنیم

می‌توانیم از ... ورودی

cout << c1 (دوست)

... ردها (دوست)

وقتی که عملگر دودویی سمت چپ آن از نوع شیء نیست باید به صورت دوست تعریف کنیم

$c_1 + c_2$

+(c1,c2) دوست } c1.+(c2) عضو

PAPCO

Subject:

Year. Month. Date. ( )

```
#include<iostream.h>
#include<iomanip.h>

class phoneNumber {
    friend ostream &operator<<(ostream &, const phoneNumber &);
    friend istream &operator>>(istream &, phoneNumber &);
  private:
    char areacode[4];
    char exchange[4];
    char line[5];
};

ostream &operator<<(ostream &out, const phoneNumber &p)
{
    out<<"("<<p.areacode<<")"
       <<" "<<p.exchange<<"-"<<p.line;
    return out;
}

istream &operator>>(istream &in, phoneNumber &p)
{
    in.ignore(2);
    in>>setw(4)>>p.areacode;
    in.ignore(2);
    in>>setw(4)>>p.exchange;
    in.ignor( )
    in>>setw(5)>>p.line;
    return in;
}
```

exchange: 456 - 7890 (line)

(300) areacode

PqPCO

Subject:

Year. Month. Date. ( )

```
int main ( )
{
  phoneNumber p;
  cin >> p;
  cout << p;
  return 0;
}
```

برای اشیایی از رده phonenumber تعیین شده است

== سه برای آرایه به دو تا که از نوع array باشند با هم نسبت می‌دهد

== سه آرایه a1,a2 را از خانه اول شروع می‌کند مقایسه می‌کند وقتی تا آخر آرایه رفت true برمی

درغیر اینصورت به دو تا خانه نامساوی برخورد کرد false برمی‌گرداند

PAPCO

Subject:
Year. Month. Date. ( )

عملگر درج در سینک و درج در خروجی باید به صورت friend تعریف بشود، این friend بودن برای اینکه دسترسی به class را داشته باشند حتما نباید زیر public یا زیر private باشند.

Array list = 10; / سازنده پیش فرض

Array (const Array &); کپی را مقدار این array را در دیگری کپی کند

= به صورت عضو تعریف می‌شود که پارامتر بگیرد، پارامتر سمت چپ عضوی از این class است.

تبدیل انواع

انواع موجود ← نوع تعریف شده توسط کاربر: سازنده تبدیل

```
string s1("ali");
```

نوع تعریف شده توسط کاربر ← انواع موجود: عملگر تبدیل

```
char * ptr = (char *) s1;
```

سازنده تبدیل نوع بازگشتی ندارد.

تمرین‌های ۸: ۱۵ و ۱۹

Subject:

Year. Month. Date. ( )

وراثت:

(بدون نوشتن ویژگی‌های اضافه می‌کنیم)

اگر یک سری رده قبلاً تعریف شده و می‌توانیم از این رده‌ها استفاده کنیم یعنی تغییری در آنها بدهیم رده‌ی جدیدی تعریف کنیم

رده‌ای که از آن ارث می‌بریم ← پایه Base

" ارث می‌برد ← مشتق drived

کلاس پدر
کلاس ← وراثت
کلاس فرزند

انواع وراثت:
- عمومی public
- خصوصی private
- محافظت شده protect

```
class Test
{
  public:
  private:
  protected:
};
```

=> سه نوع داده داریم

وراثت عمومی: public

اعضای عمومی پایه ← اعضای عمومی مشتق

اعضای محافظت شده ← اعضای محافظت شده مشتق

" خصوصی پایه ← پنهان هستند ← معنای این نوع این است که دیگر از داده‌ها کپی نمی‌کند

عضو پنهان یک حالت خصوصی است و فقط از طریق توابع عضو پایه می‌توان به آن دسترسی داشت

وراثت خصوصی: private

اعضای عمومی و محافظت شده ← خصوصی

خصوصی‌ها ← پنهان اند

وراثت محافظت شده protected

اعضای عمومی و محافظت شده ← محافظت شده

خصوصی‌ها ← پنهان اند

(دوستی‌ها میراث نمی‌رسند (دوست پدر لزوماً دوست پسر نیست))

PAPCO

Subject:

Year. Month. Date. ( )

more بسته به مرحله خروجی باید معلوم باشد.

تعداد state ها 2^n نباشد می‌توانیم تعدادی don't care داشته باشیم (مدارمان کوچکتر می‌شه)

اگر m تا state داشته باشیم تعداد FF ، $\lceil \log_2^n \rceil$ است (مثلاً اگر ۹ تا state داشته باشیم

باید 4 تا FF داشته باشیم که باید به سمت بزرگتر گرد کنیم)

(اگر ۹ تا state داشته باشیم برای FF ، 7 state اضافی داریم) $n \geq \lceil \log_2^n \rceil$

state Assignment ; Example 1

Subject:
Year. Month. Date. ( )

```
class point
{
   friend iostream &operator << (ostream &, const point &);
 public:
      point (int = 0, int = 0)
      void set point (int, int)
      int get x() const { return x; }
      int get y() const { return y; }
 protected:
      int x, y;
};

point :: point (int a, int b)
{   Set point (a, b); }

void point::setpoint (int a, int b)
{ x = a;    y = b; }
```

درج در خروجی x و y به صورت [x,y] چاپ می‌کند چون عضوی از این رده نبود

```
ostream & operator << (ostream & out, const point & p)
{
   out << "[" << p.x << "," << p.y << "]" ;
   return out;
}
```

PAPCO

Subject.
Year. Month. Date. ( )

class circle : public point {
private
protected

friend ostream & operator << (ostream &, const circle &);
public:
circle (double r=0.0, int x=0, int y=0);
void setRadius (double);
double getRadius ( ) const;
double area ( ) const;
protected:
double Radius;
};

Circle :: circle (double r, int a, int b) : point (a,b)
{ Radius = r; }

ostream & operator << (ostream & out, const circle &c)
{
out << static_cast<point> (c) <<
"Radius = " << c.Radius;
}

Subject:
Year. Month. Date. ( )

برای مثال

```
int main ()
{
    circle c1 (10, 2, 5);
}
```

x, y به point پاس داده میشوند

main اصلی برنامه

```
int main ()
{
    point * pointptr=0 , p(30,50);
    circle * circleptr=0 , c(2.7, 120, 89);
    cout<< "point p: " << p << "\n circle c: " << c << "\n";
    pointptr= &c;
    cout<< "circle c (via pointer): " << * pointptr
```

pointer از نوع point — متغیر از نوع point

عملگر درج در خروجی point — عملگر درج در خروجی circle

آدرس یک شیء مشتق را که از کلاس پایه است می‌توان به اشاره‌گر پایه تخصیص داد

← محتوای این همان c می‌شود

چون این از نوع point است پس عملگر قبلی از عملگر درج در خروجی point می‌شود چون که رجوع می‌شود براساس type است که pointptr دارد و بعد از compile شدن برنامه معلوم می‌شود.

~~cout~~

نمی‌توانیم pointptr را مستقیماً در circle گذاشت

```
    circleptr= static_cast<circle *> (pointptr);      circleptr = pointptr
    cout<< "circle c (via circleptr): " << circleptr;
```

چون که اینجا خروجی جواب می‌دهد چون circleptr که از نوع circle است و عملگر درج در خروجی عملگر درج در خروجی circle است

آدرس پایه را نمی‌توانیم مستقیم در مشتق بگذاریم

```
    pointptr= &p
    circleptr= static_cast<circle *> (pointptr)
```

به جای p اشاره می‌کند

<< circle
point

P4PCO

Subject:

Year. Month. Date. ( )

p را point می‌خواهیم از طریق اشاره‌گر circle چاپ کنیم برای circle است (که وجود ندارد ما چاپ می‌کند که نمی‌خواهیم)

```
cout << "point p (via circleptr : " << circleptr;
```

یعنی چون نقطه هیچ شعاعی ندارد و این دستور را می‌دهیم circleptr به یک point اشاره [illegible] circleptr می‌کنیم [illegible] می‌کند ندارد و اگر آن را بخواهیم چاپ کنیم به اشتباه [illegible] چاپ می‌کند [illegible] point نیست و مقدار آن مشخص نیست

هیچ وقت دیدن یک شیء از پایه را از رده مشتق نداریم مگر اینکه بدانیم چه کار می‌کنیم و به مقدار [illegible] باید [illegible] پیدا کنیم که مشخص نیست

ساعتی
مثال یک بوده employee داریم که از آن رده کارمند ساعتی را مشتق می‌کنیم
چیزی که در رده مشتق هست [illegible] روی چیزی که در پایه است روی [illegible]

```
class Employee
{
  public:
    Employee(const char *, const char *);
    void print() const;
    ~Employee();    // [illegible] first name و last name [illegible] آزاد کند

  private:
    char * first name;
    char * last name;
};
```

Subject:
Year. Month. Date. ( )

```
void Employee :: print ( ) const
{
  cout << first name << " " << last name ; }

class Hourly workers : public Employee
{
 public:
    Hourly workers ( const char *, const char * , double , double ) ;
    double getpay ( ) const ;
    void print ( ) const ;
 private:
    double wage ;
    double hours ;
 } ;
```

تمام توابع کارمند برای کارگرهای Employee ساعتی هم هست

تمام چیزهایی را که Employee دارد به ارث می‌بره

تابع print را overwrite می‌کنیم دوباره می‌نویسیم

print را تعریف می‌کنیم هر جا نیاز داشتیم اجرا می‌شه

```
void Hourly worker :: print ( ) const
{
 Employee :: print ( ) ;
 cout << "wage = " << wage <<
    ", hours = " << hours ; }
```

firstname , lastname , wage , hour را باید چاپ کنه

اگر فقط print بنویسیم تابع recursive فرض می‌شه (تابعی که خودش را دوباره فراخوانی می‌کنه)

برای چاپ first n و last n از تابع print رده پایه استفاده می‌کنیم

اگر در رده پایه و مشتق تابع هم‌نام باشد در رده مشتق اون تابع overwrite می‌شود

PAPCO

Subject:

Year. Month. Date. ( )

در هنگام تعریف شئ از نوع کلاس مشتق ابتدا سازنده کلاس پایه و سپس سازنده کلاس مشتق فراخوانی میشود. نابودکننده برعکس است.

```
class point
{
    public:
        point (int =0, int =0);
        ~point ();
    protected:
        int x,y;
};

point :: point (int a, int b)
{
    x=a; y=b;
    cout<<"point constructore ["<< x<<","<< y <<"]"<<endl;
}

point :: ~point ()
{
    cout<<"point destructor ["<<x<<","<<y<<"]"<<endl;
}

class circle: public point
{ public:
    circle (double r=0, int =0, int=0);
    ~circle ();
  private:
    double radius
};
```

P4PCO

Subject.
Year. Month. Date. ( )

```
circle :: circle (double r, int a, int b) : point (a,b)
 {  radius = r;
    cout << " circle constructor " << radius << " [ " << x
    << " , " << y << " ] " << endl;
}

circle :: ~ circle ( )
{   cout << "circle destructor " << radius << " [ " << x << " , " << y << " ] " << endl;

int main( )
{
   { point b(11,22);
   }
     circle circle1 ( 4.5, 72, 29 );
     circle circle2 (10, 5, 5 );
     return 0;
  }
```

Notes beside the code:

- `{ point b(11,22);` — اینجا یک کلاس ساخته که شیء از نوع point است)
- `}` — اینکه بیرون می‌آییم نابودکننده فراخوانی میشه
- `circle circle1 ( 4.5, 72, 29 );` — سازنده circle فراخوانی میشه قبلش constructor ، point فراخوانی میشه یعنی در خروجی دستور پیغام می‌بینیم اول مربوط به point و بعد مربوط به circle است)
- `circle circle2 (10, 5, 5 );` — مثل بالایی دستور پیغام چاپ میشه
- `return 0;` — دستورات main تموم شد بعد باید نابود بشه یعنی اول نابودکننده پایینی (circle2) فراخوانی میشه و بعد نابودکننده (circle1) بعد هم پایه مشتق هم اول نابودکننده مشتق فراخوانی میشه

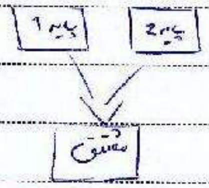
اول ~~سازنده~~ نابودکننده مشتق بعد پایه ، سازنده: اول پایه بعد مشتق (برعکس ترتیب تعریف شدن در برنامه نابودکننده فراخوانی میشد)

PAPCO

Subject:
Year. Month. Date. ( )

وراثت چندگانه

رده‌ای بخواهیم بیش از یک رده ارث ببرد




```
class Base1
{ ...
};
class Base2
{ ...
};
class Drived: public Base1, public Base2;
{
...
};
```

نوع ارث بری‌ها می‌تواند متفاوت باشد

```
Drived :: Drived (int a, int b, int c) : Base1(a), Base2(b)
```

هر قسمتی می‌توان سازنده‌های Base پاس داد

تمرین: ۱۴-۱