2.5 类型的形式化描述简介

2.5.1 类型的数学模型研究

- 代数模型 – 用代数方法描述抽象数据类型的语法和语义;
- λ演算 – 用λ演算描述类型中的单态函数（包括高阶结构）;
- 全称量词 – 用逻辑方法描述类属类型;
- 存在量词 – 用逻辑方法描述抽象数据类型;
- 记录包含 – 用记录描述类型和子类型的包含关系;
- 项包含 – 用一阶项和格运算对类型之间的偏序进行描述和计算;
- ……

2.5 类型的形式化描述简介

2.5.2 代数模型

- 类型和代数的相似性：
  - 类型：一个值的集和一个作用于值集的操作集。
  - 代数：一个值集和操纵这一值集的操作集。

2.5 类型的形式化描述简介

2.5.2 代数模型

- 与类型有关的基本问题:
  - 当值的表示或操作的实现发生改变时，类型的特性（主要指语法和语义）是否也要发生改变？在什么条件下可以不发生改变？
  - 不同的程序员在不同的背景下可能设计出特性相同的不同类型，如何判定它们是等价的？
  - 不同的人对同样的类型会有不同的理解，如何统一他们的理解？
- 用数学工具来一致地描述特性相同的类型，奠基人是两个博士生：**J.V.Guttag**（可满足语义）和 **J.A.Goguen**（初始代数语义）。

2.5 类型的形式化描述简介

2.5.3 λ演算

- **Church** 已经证明：图灵机、递归函数、λ演算和 **Post** 系统这四种计算模型是等价的，它们是计算机科学理论体系的支柱。
- λ演算是一种关于函数的计算理论，是一种串的重写系统，用一组重写规则对给定的公式反复进行替换，得到的最简形式，就是这个公式的解。因此是函数程序设计的理论基础。

2.5 类型的形式化描述简介

2.5.3 λ演算

- λ演算的基本表达形式（λ表示）： λx. E。x 可以是一个变量，E 一般是受 x 约束的表达式，如:

$$\lambda x . x^2+x+2$$

- 当 x 以某一值取代时，可以计算表达式的值:

$$(\lambda x . x^2+x+2)(3) = 14$$

- 如果 E 不包含变量 x，则 (λx. E) 的值是一常数，就等于 E。
- 对于一个含多变量的表达式，如 E(x, y)，则有:

λx. λy. E(x, y)，或简写成： λx, y . E(x, y)。

2.5 类型的形式化描述简介

2.5.3 λ演算

- λ表达式中的变元可以是一个函数，例如：λf.λx.f(x)，这样，就能够表示高阶的结构，例如程序中的函数跳转表。
- λ表达式中的变元原来是没有类型说明的，这与程序设计程序中的情况有差别，于是发生了扩展：**Cardelli** 和 **Wegner** 在λ表达式中，给变量注明类型，得到类型化的λ演算。

第一次作业（7 天后提交，10天后截止）

试根据ADT的概念，

1. 定义一个类型，来表示某种亮度可调的台灯;

2. 定义一个类型，来表示某种MP3。

表示的手法不限，但须体现类型的主要特征：值集（值的数据结构）、操作集（语法、语义）以及对外的接口。

§3. 面向对象程序设计

3.1 程序设计范型

3.1.1 程序设计范型的概念

- 程序设计范型（*Programming Paradigm*）是人们在程序设计时所采用的基本方式模型。
  - 程序设计范型决定了程序设计时采用的思维方式、使用的工具，同时又有一定的应用范畴。

3.1 程序设计范型

3.1.2 典型的程序设计范型

- **Procedural Programming**（过程程序设计）
  - Decide which *procedures* you want: use the best algorithms you can find.
  - 基于过程抽象。
  - 以对数据进行分步骤地加工和处理作为思维的主干。
  - 代表性程序设计语言：**Fortran**。

3.1 程序设计范型

3.1.2 典型的程序设计范型

- **Modular Programming**（模块化程序设计）
  - Decide which *modules* you want: partition the program so that data is hidden in modules.
  - 基于模块抽象。
  - 以对程序的模块结构进行划分和组织作为思维的主干。
  - 代表性程序设计语言：**Modula-II**。

3.1 程序设计范型

3.1.2 典型的程序设计范型

- **Functional Programming**（函数程序设计）
  - 基于元数学的计算概念（如λ演算）。
  - 利用一组重写规则，通过对给定的计算公式进行反复的替换，求得最简形式（也就是解）。这种范型没有变量存贮的概念，因此在计算步骤之间没有状态依赖关系。
  - 以对给定的问题表示进行分步骤地替换和化简为思维的主干。
  - 代表性程序设计语言：**Lisp**。

3.1.2 典型的程序设计范型

- **Logical Programming**（逻辑程序设计）
  - 基于一阶谓词演算理论。
  - 以从已知条件向结论进行分步骤的推理作为思维的主干。
  - 代表性程序设计语言：**Prolog**。

3.1 程序设计范型

3.1.2 典型的程序设计范型

- **Object-Oriented Programming（OOP，面向对象程序设计）**
  - Decide which *classes* you want: provide a full set of operations for each class; make commonality explicit by using inheritance.
  - 基于数据抽象、继承性和消息传递。
  - 以对实体（包括结构、状态和行为）进行分类、组织和协同作为思维的主干。
  - 代表性程序设计语言：**Smalltalk**。

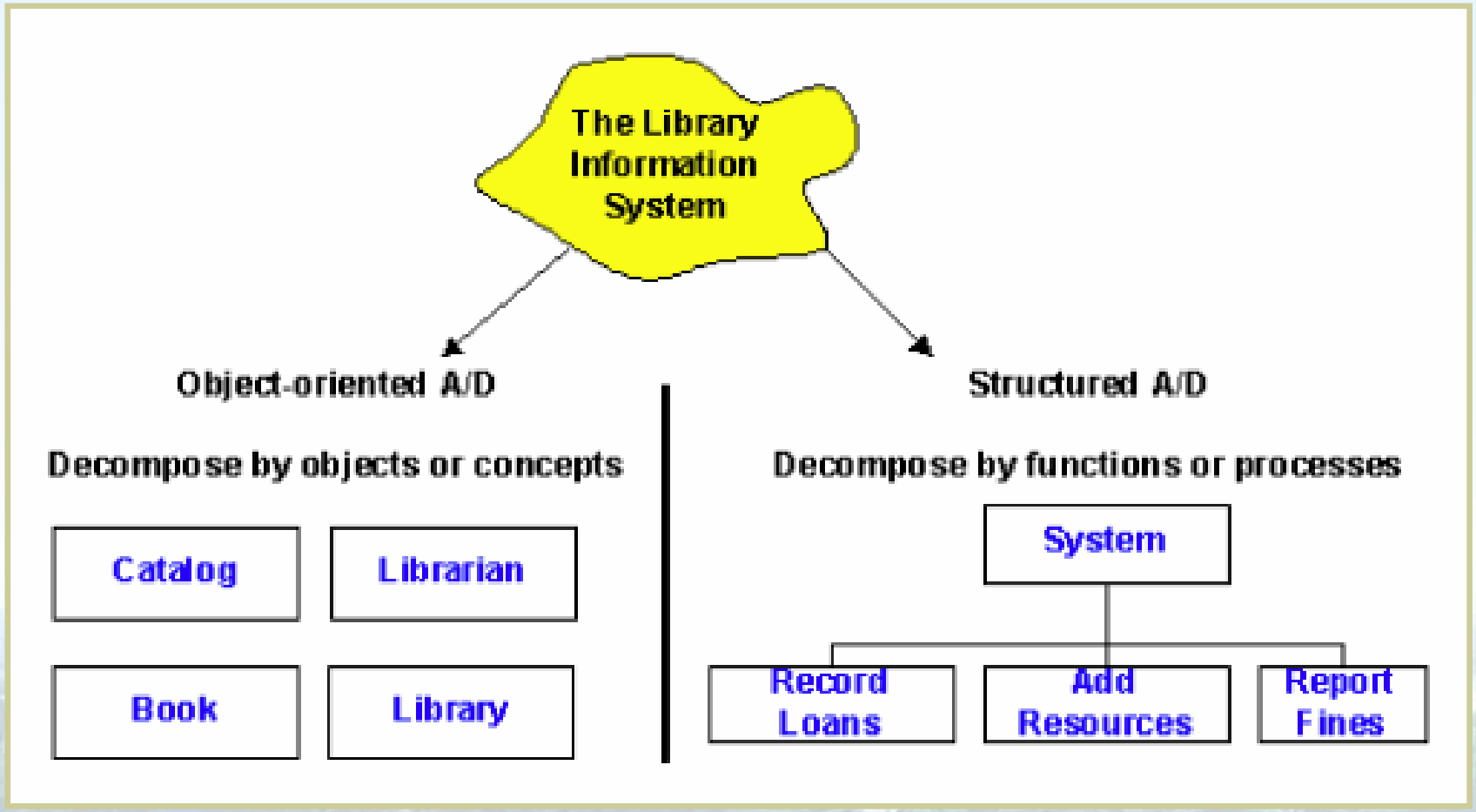
3.1 程序设计范型

3.1.3 面向对象的基本概念体系

- 面向对象技术将计算看成是一个系统的演变过程，系统由对象组成，经历一连串的状态变化以完成计算任务。
- 对象具有状态保持能力和自主计算能力。
- 面向对象设计和实现的重点是多个对象的网状组织结构和协同计算，而不是过程调用的层次结构。

3.1 程序设计范型

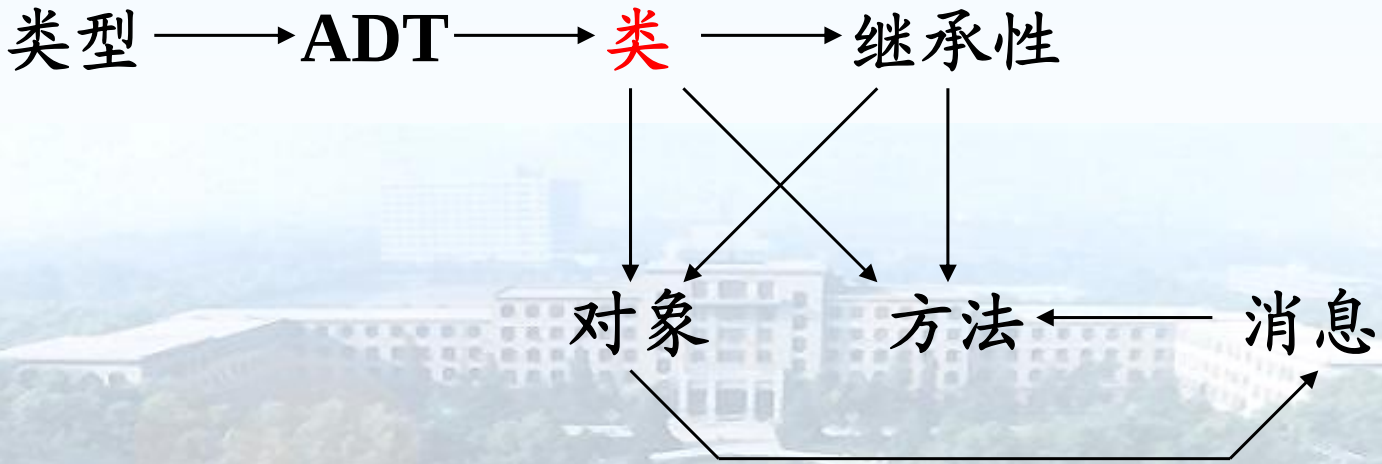
3.1.3 面向对象的基本概念体系



3.1 程序设计范型

3.1.3 面向对象的基本概念体系

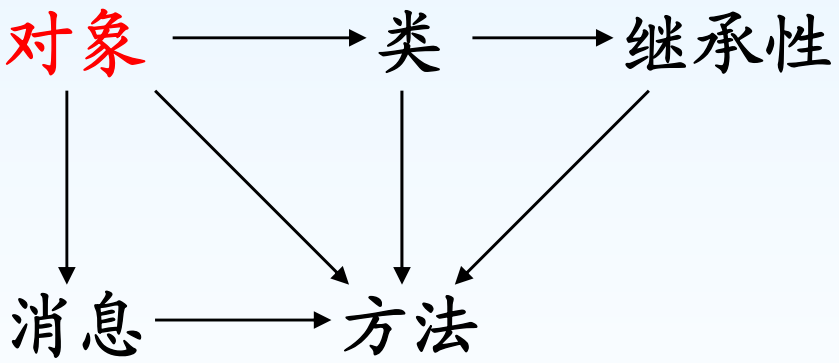
- 面向对象的基本概念：类（*Classes*）、对象（*Objects*）、继承性（*Inheritance*）、方法（*Methods*）、消息（*Messages*）。
- 存在着以类为核心和以对象为核心的两种体系。
- 以类为核心的体系是：



3.1 程序设计范型

3.1.3 面向对象的基本概念体系

- 以对象为核心的体系是:



3.1 程序设计范型

3.1.4 在程序设计时要解决的几个主要问题

- 怎样区分不同种类的对象?
- 怎样表示对象的存贮结构（包括与相关对象的联系）和状态?
- 怎样表示一种对象所能够承担的计算任务（或者叫做这种对象的行为）?
- 怎样表示对象之间的协同方式（或者叫做对象间的通信协议）?
- 怎样表示（和利用）不同种类的对象之间在结构和行为方面的相似性?

3.1.4 在程序设计时要解决的几个主要问题

- 怎样表示对象之间的关联（消息传递路径）？怎样在对象的分类、结构、行为、通信协议等方面的设计上，对组织对象进行协同计算提供支持?
- 怎样在对象的分类、相似性表示等方面的设计上，对程序的重用、扩充和修改提供支持?

3.2 程序设计语言中的OOP机制

3.2.1 类（Classes）

- 对于使用面向对象语言的程序员而言，使用一个自定义类型应当与使用一个基本类型没有本质区别，看到的也应当只是类型名和一组操作的声明（包括操作名、参数、操作涵义、操作使用规则），而看不到操作的具体实现，也看不到该类型所定义的内部数据结构。
- 在面向对象语言中，这样的自定义类型就是类。类是支持继承性和多态的抽象数据类型，具有实例化能力。
- 类的定义、实现和使用通常是分离的，有确定的分离机制。

3.2 程序设计语言中的OOP机制

3.2.1 类

```
struct Date {
  int  d, m, y;
};
```

?

```
void init_date(Date& d, int dd, int mm, int y
void add_year(Date& d, int n);
void add_month(Date& d, int n);
void add_day(Date& d, int n);
```

```
void joking(Date& d)
{
  d.m *= 20;
}
void f()
{
  Date today;
  init(today,23,10,2014);
  // ...
  joking(today);
  // ...
  add_day(today, 1);
}
```

3.2 程序设计语言中的OOP机制

```
// Date.h
struct Date {
  int  d, m, y;

  void init(int dd, int mm, int yy);
  void add_year(int n);
  void add_month(int n);
  void add_day(int n);
};
```

```
#include "Date.h"
Date my_birthday;

void f()
{
  Date today;
  today.init(23,10,2014);
  my_birthday.init(30,12,1988);
  Date tomorrow = today;
  tomorrow.add_day(1);
}
```

```
#include "Date.h"
void Date::init(int dd, int mm, int yy)
{
  d = dd;
  m = mm;
  y = yy;
}
/* ... */
```

3.2 程序设计语言中的OOP机制

```
// Date.h
struct Date {
  int  d, m, y;

  void init(int dd, int mm, int yy);
  void add_year(int n);
  void add_month(int n);
  void add_day(int n);
};
```

```
#include "Date.h"
Date my_birthday;

void f()
{
  Date today;
  today.init(23,10,2014);
  my_birthday.init(30,12,1988);
  Date tomorrow = today;
  tomorrow.add_day(1);
}
```

```
#include "Date.h"
void Date::init(int dd, int mm, int yy)
{
  d = dd;
  m = mm;
  y = yy;
}
/* ... */
```

3.2 程序设计语言中的OOP机制

3.2.1 类

- 定义在一个类中的变量，称为这个类的
  - 数据成员（*Data Members*，**C++**），或
  - 实例变量（*Instance Variables*，**Smalltalk**）。
- 定义在一个类中的函数，称为这个类的
  - 成员函数（*Member Functions*，**C++**），或
  - 方法（*Methods*，**Smalltalk**）。
- 每个数据成员都有确定的类型，因而用它们的值集可以构造出这个类的值集（构造的方法与记录类型类似）。
- 这些成员函数构成了这个类的操作集（前提是没有其他函数有权直接访问这个类的任何数据成员）。

3.2 程序设计语言中的OOP机制

3.2.1 类

- 在面向对象语言中，提供了关于外界对于类的访问控制（*Access Control*）机制。
  - 访问控制程度的不同，也称为“能见度”不同。
- C++语言提供了三种访问控制程度:
  - public – 允许在同一程序的任何地方引用;
  - private – 只允许在本类的成员函数实现体中引用;
  - protected – 允许在本类的成员函数实现体中引用，以及在满足一定条件的子类的成员函数实现体中引用。

3.2 程序设计语言中的OOP机制

3.2.1 类

```
// Date.h
struct Date {
   int  d, m, y;

   void init(int dd, int mm, int yy);
   void add_year(int n);
   void add_month(int n);
   void add_day(int n);
};
```

```
void joking(Date& d)
{
   d.m *= 20;
}
```

3.2 程序设计语言中的OOP机制

3.2.1 类

```
// Date.h
class Date {
private:
   int  d, m, y;
public:
   void init(int dd, int mm, int yy);
   void add_year(int n);
   void add_month(int n);
   void add_day(int n);
};
```

```
void joking(Date& d)
{
   d.m *= 20;
}
```

3.2 程序设计语言

3.2.1 类

```
// Date.h
class Date {
private:
   int  d, m, y;
public:
   void init(int dd, int mm, int yy);
   void add_year(int n);
   void add_month(int n);
   void add_day(int n);
};
```

```
#include "Date.h"
void Date::init(int dd, int mm, int yy)
{
   d = dd;   ✓
   m = mm;   ✓
   y = yy;   ✓
}
/* ... */
```

```
void joking(Date& d)
{
   d.m *= 20;  ✗
}
```

3.2 程序设计语言中的OOP机制

3.2.1 类

- 提供访问控制机制的目的是:
  - 使得这个类的成员函数确实构成了这个类的操作集。
  - 易于进行出错定位（这是调试程序时首先要解决、也是最难解决的一个问题）。
  - 用成员函数的声明组成这个类的接口，将它的数据成员和成员函数的实现封装起来，以减少程序的修改对外部的影响。
  - 有利于掌握一个类型的使用方式（了解数据结构后才能使用类型，实际上是不得已而为之）。

3.2 程序设计语言中的OOP机制

3.2.1 类

```
// Date.h
class Date {
   int  d, m, y;
public:
   void init(int dd, int mm, int yy);
   void add_year(int n);
   void add_month(int n);
   void add_day(int n);
};
```

```
#include "Date.h"

void f()
{
   Date today;
   today.init(23,10,2014);
   Date tomorrow = today;
   tomorrow.add_day(1);
}
```

```
#include "Date.h"

void f()
{
   Date today;
   // today.init(23,10,2014);
   Date tomorrow = today;
   tomorrow.add_day(1);
}
```

3.2 程序设计语言中的OOP机制

3.2.1 类

- 程序设计的悲哀是对不确定的状态进行了确定的操作。
- A better approach is to allow the programmer to declare a function with the explicit purpose of initializing objects.
- Because such a function constructs values of a given type, it is called a constructor.
  - A constructor is recognized by having the same name as the class itself.
  - 其任务是实例生成（包括既定的初始化）。

3.2 程序设计语言中的OOP机制

```
// Date.h
class Date {
   int  d, m, y;
public:

   Date(int dd, int mm, int yy);

   void add_year(int n);
   void add_month(int n);
   void add_day(int n);
};
```

```
#include "Date.h"

void f()
{
   Date today = Date(23,10,2014);
   Date this_day(23,10,2014);
   Date my_birthday;
   Date error_date(26,9);
   /* ...*/
}
```

```
#include "Date.h"
Date::Date(int dd, int mm, int yy)
{
   d = dd;
   m = mm;
   y = yy;
}
/* ... */
```

3.2 程序设计语言中的OOP机制

3.2.1 类

- 一个类中可以根据需要定义多个 Constructors（函数名过载），编译程序根据调用时实参的数目、类型和顺序自动找到与之匹配者。

```
// Date.h
class Date {
  int  d, m, y;
public:
  Date(int dd, int mm, int yy);
  Date(int dd, int mm);      // today's year
  Date(int dd);              // today's month and year
  Date();                    // default Date: today
  Date(const char* p);       // date in string represent
  /* ... */
};
```

```
#include "Date.h"

void f()
{
  Date today(23);
  Date july4("July 4,1983");
  Date guy("5 Nov");
  Date now;
  /* ...*/
}
```

3.2 程序设计语言中的OOP机制

3.2.1 类

- 一个类中可以根据需要定义多个 Constructors（函数名过载），编译程序根据调用时实参的数目、类型和顺序自动找到与之匹配者。

```
// Date.h
class Date {
  int  d, m, y;
public:
  Date(int dd, int mm, int yy);
  Date(int dd, int mm);     // today's year
  Date(int dd);             // today's month and year
  Date();                   // default Date: today
  Date(const char* p);      // date in string represent
  /* ... */
};
```

```
#include "Date.h"

void f()
{
  Date today(23);
  Date july4("July 4,1983");
  Date guy("5 Nov");
  Date now;
  /* ...*/
}
```

3.2 程序设计语言中的OOP机制

3.2.1 类

- 一个类中可以根据需要定义多个 Constructors（函数名过载），编译程序根据调用时实参的数目、类型和顺序自动找到与之匹配者。

```
// Date.h
class Date {
  int  d, m, y;
public:
  Date(int dd, int mm, int yy);
  Date(int dd, int mm);     // today's year
  Date(int dd);             // today's month and year
  Date();                   // default Date: today
  Date(const char* p);      // date in string represent
  /* ... */
};
```

```
#include "Date.h"

void f()
{
  Date today(23);
  Date july4("July 4,1983");
  Date guy("5 Nov");
  Date now;
  /* ...*/
}
```

3.2 程序设计语言中的OOP机制

3.2.1 类

- 一个类中可以根据需要定义多个 Constructors（函数名过载），编译程序根据调用时实参的数目、类型和顺序自动找到与之匹配者。

```
// Date.h
class Date {
  int  d, m, y;
public:
  Date(int dd, int mm, int yy);
  Date(int dd, int mm);     // today's year
  Date(int dd);             // today's month and year
  Date();                   // default Date: today
  Date(const char* p);      // date in string represent
  /* ... */
};
```

```
#include "Date.h"

void f()
{
  Date today(23);
  Date july4("July 4,1983");
  Date guy("5 Nov");
  Date now;
  /* ...*/
}
```

3.2 程序设计语言中的OOP机制

3.2.1 类

- 有时很难估计将来对Constructor形参的组合会有怎样的要求，一种有效的策略是对Constructor 声明有省缺值的形参（*Default Arguments*）。

```
// Date.h
class Date {
   int  d, m, y;
public:
   Date(int dd = 0, int mm = 0, int yy = 0);
   /* ... */
};
```

```
#include "Date.h"

void f()
{
   Date today(23);
   Date someDay(23, 10);
   Date aDay(23, 10, 2014);
   Date now;
   /* ...*/
}
```

3.2 程序设计语言中的OOP机制

3.2.1 类

- 任何一个系统的资源总是有限的，如动态分配的存储空间、文件描述符、信号量等。如果在包括 Constructor 在内的成员函数中动态地获取了上述资源，就必须在实例的生存期结束时释放它们（C++ 编译程序无法自动释放上述资源）。否则，这样的资源可能在程序运行过程中耗尽。
- 因此，需要在一个类中定义一个决定如何释放在其实例生存期中动态申请的资源（善后处理）的成员函数，这就是 **Destructor**。
  - **Destructor** 的命名规定为：~ <类名> ( )。**Destructor** 不允许有形参，因此不允许过载。

3.2 程序设计语言中的OOP机制

3.2.1 类

```
class Name {
  const char* s;
  /* ... */
};
```

```
class LogFile {
  FILE*  fp;  // 文件描述符
public:
  LogFile(char* fileName)
    { fp = fopen(fileName, "w"); }
  ~LogFile() { fclose(fp); }
  printLog(const char*, ...);
};
```

```
class Table {
  Name*  p; // 元素个数为sz的数组，空间是动态申请的
  size_t sz;
public:
  Table(size_t s = 15) { p = new Name[sz = s]; }
  ~Table() { delete [] p; }
  Name* lookup(const char*);
  bool  insert(Name*);
};
```

3.2 程序设计语言中的OOP机制

3.2.1 类

- A default constructor is a constructor that can be called without supplying an argument.
- If a user has declared a default constructor, that one will be used; otherwise, the compiler will try to generate one if needed and if the user hasn't declared other constructors.

3.2 程序设计语言中的OOP机制

3.2.1 类

```
struct Tables{
  int   i;
  int   vi[10];
  Table t1;      // 被自动初始化
  Table vt[10]; // 每个元素被自动初始化
};
/* ... */
Tables tt;
```

```
class Name {
  const char* s;
  /* ... */
};

class Table {
  Name*  p; // 元素个数为sz的数组，空间是动态申请的
  size_t sz;
public:
  Table(size_t s = 15) { p = new Name[sz = s]; }
  ~Table() { delete [] p; }
  Name* lookup(const char*);
  bool  insert(Name*);
};
```

3.2 程序设计语言中的OOP机制

3.2.1 类

- 面向对象语言中的实例化及其机制:
  - 实例化：依据对应的一个类（或一组有继承关系的类），根据作用域控制规则，进行实例生成和实例消除的过程。
  - 实例生成：在用一个类定义一个变量时，或者用这个类的指针类型定义一个指针、且显式执行了 **new** 操作时，编译程序将该声明语句翻译成关于这个类的 **Constructor**（如果定义了的话）的一次调用，按照该类所规定的空间大小，分配一块存贮空间（通常是一块连续的空间），使之与该变量绑定，对这块空间执行在 **Constructor** 中定义的那些操作。

3.2 程序设计语言中的OOP机制

3.2.1 类

- 面向对象语言中的实例化及其机制:
  - 实例消除：当到达了该变量的作用域结束位置，或者显式地对该变量执行了 **delete** 操作，则执行 **Destructor**（如果定义了的话）中规定的操作，释放这个实例占用的存贮空间。

3.2 程序设计语言中的OOP机制

3.2.1 类

```
// Date.h
class Date {
   int  d, m, y;
public:
   /* ... */
};
```

```
#include "Date.h"

void f()
{
   Date today(23,10,2014);
   Date *someDay = new Date(11,11,2000);
   Date aDay(1,12,2001);
   {
      Date now;
      /* ...*/
   }
   delete someDay;
   /* ...*/
}
```

today :

23
10
2014

3.2 程序设计语言中的OOP机制

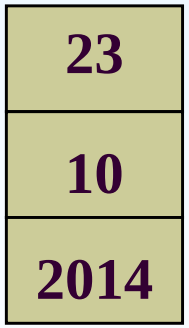
3.2.1 类

```
// Date.h
class Date {
   int  d, m, y;
public:
   /* ... */
};
```

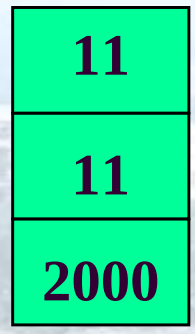
```
#include "Date.h"

void f()
{
   Date today(23,10,2014);
   Date *someDay = new Date(11,11,2000);
   Date aDay(1,12,2001);
   {
      Date now;
      /* ...*/
   }
   delete someDay;
   /* ...*/
}
```

today :



*someDay :



3.2 程序设计语言中的OOP机制

3.2.1 类

```
// Date.h
class Date {
   int  d, m, y;
public:
   /* ... */
};
```

```
#include "Date.h"

void f()
{
   Date today(23,10,2014);
   Date *someDay = new Date(11,11,2000);
   Date aDay(1,12,2001);
   {
      Date now;
      /* ...*/
   }
   delete someDay;
   /* ...*/
}
```

today :

23
10
2014

aDay :

1
12
2001

*someDay :

11
11
2000

3.2 程序设计语言中的OOP机制

3.2.1 类

```
// Date.h
class Date {
   int  d, m, y;
public:
   /* ... */
};
```

```
#include "Date.h"

void f()
{
   Date today(23,10,2014);
   Date *someDay = new Date(11,11,2000);
   Date aDay(1,12,2001);
   {
      Date now;
      /* ...*/
   }
   delete someDay;
   /* ...*/
}
```

today :

23
10
2014

aDay :

1
12
2001

*someDay :

11
11
2000

now :

23
10
2014

3.2 程序设计语言中的OOP机制

3.2.1 类

```
// Date.h
class Date {
   int  d, m, y;
public:
   /* ... */
};
```

```
#include "Date.h"

void f()
{
   Date today(23,10,2014);
   Date *someDay = new Date(11,11,2000);
   Date aDay(1,12,2001);
   {
      Date now;
      /* ...*/
   }
   delete someDay;
   /* ...*/
}
```

today :

23
10
2014

aDay :

1
12
2001

*someDay :

11
11
2000

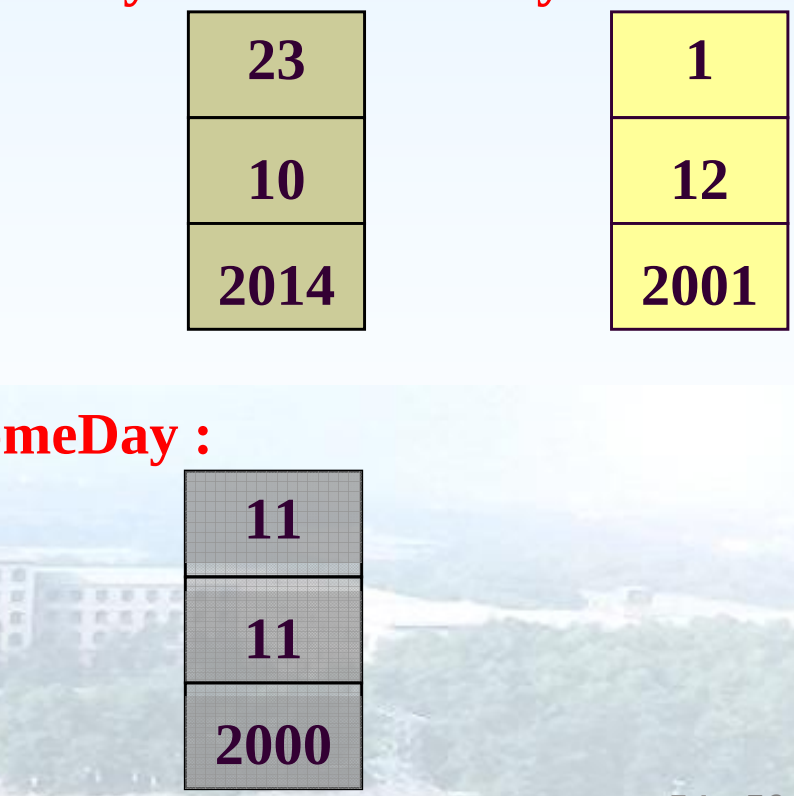
3.2 程序设计语言中的OOP机制

3.2.1 类

```
// Date.h
class Date {
   int  d, m, y;
public:
   /* ... */
};
```

```
#include "Date.h"

void f()
{
   Date today(23,10,2014);
   Date *someDay = new Date(11,11,2000);
   Date aDay(1,12,2001);
   {
      Date now;
      /* ...*/
   }
   delete someDay;
   /* ...*/
}
```



3.2 程序设计语言中的OOP机制

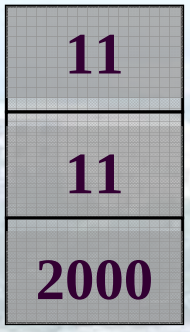
3.2.1 类

```
// Date.h
class Date {
   int  d, m, y;
public:
   /* ... */
};
```

```
#include "Date.h"

void f()
{
   Date today(23,10,2014);
   Date *someDay = new Date(11,11,2000);
   Date aDay(1,12,2001);
   {
      Date now;
      /* ...*/
   }
   delete someDay;
   /* ...*/
}
```

*someDay :



3.2 程序设计语言中的OOP机制

3.2.1 类

- 可以看出，在类中定义的一个数据成员，是这个类的每个实例的一个组成部分，在每个实例中都有对应的、结构相同的存储空间，因而可以使每一个实例保持不同的值，具有不同的状态。
- 类的实例就是对象。
- 用一个类可以产生一组有相同的存储结构、相同的行为规律、相同的接口（通信协议）、不同状态的对象。因此，类是对象的模板。

要点与引伸

- 类与 ADT 属于同一层次的抽象：表示了一组对象的结构、行为和通信协议，忽略了具体对象的值（状态）；
- 类定义了对象，但类本身属于编译时的实体，在运行时只有用类实例化而成的对象，才能产生行为，保持和改变状态。

下一次课的内容

- 面向对象程序设计－程序设计语言中的OOP机制
  - 对象
  - 消息传递和方法
  - 继承性与类层次结构
  - 重置