面向对象程序设计

类和对象

2020 年 春

耿楠

计算机科学系
信息工程学院

西北农林科技大学
NORTHWEST A&F UNIVERSITY
中国 · 杨凌

将不同数据类型组合成一个整体

语法

```
struct 结构类型名
{
    数据类型 1 成员变量 1;
    数据类型 2 成员变量 2;
    …
    数据类型 n 成员变量 n;
};
```

定义

```
struct StuNode{
    int ID;
    char name[20];
    char gender[12];
    int age;
};
```

结构体的实际大小

有时 ≠ 结构体内部成员变量所占内存之和

字节数

```
sizeof(StuNode) = 40;

2*sizeof(int)
+sizeof(char)*(20+12)
= 40;
```

定义

```
struct StuNode{
    int ID;
    char name[20];
    char gender[12];
    int age;
};
```

结构体的实际大小

有时 ≠ 结构体内部成员变量所占内存之和

字节数

```
sizeof(StuNode) = 44;

2*sizeof(int)
+sizeof(char)*(20+13)
= 41;
```

定义

```
struct StuNode{
    int ID;
    char name[20];
    char gender[13];
    int age;
};
```

字节数

```
sizeof(StuNode) = 44;

2*sizeof(int)
+sizeof(char)*(20+15)
= 43;
```

定义

```
struct StuNode{
    int ID;
    char name[20];
    char gender[15];
    int age;
};
```

结构体的实际大小

SumSize = 结构体内部成员变量所占内存之和
L = 结构体内基本类型长度的最大值

```
SumSize % L == 0 ? SumSize : (L * (SumSize / L) % + L)
```

字节数

```
sizeof(StuNode) = 44;

2*sizeof(int)
+sizeof(char)*(20+16)
= 44;
```

定义

```
struct StuNode{
    int ID;
    char name[20];
    char gender[16];
    int age;
};
```

设置内存**对齐方式**：#pragam pack(value)

```
#pragma pack(1)
```

结构体的使用

语法

```
struct 结构类型名 结构变量名 = {
  成员 1 初始值,
  成员 2 初始值,
  …,
  成员 n 初始值
};
```

定义

```
struct StuNode{
  int ID;
  char name[20];
  char gender[16];
  int age;
};
```

声明

```
struct StuNode myNode0;
struct StuNode myNode = {101,"Tom","male",35};
```

▶ 结构体中可以有函数

语法

```
struct 结构类型名 {
  数据类型 1 成员变量 1;
  …;
  数据类型 n 成员变量 n;
  函数返回类型 函数名 (参数列表);
};
```

```
// 例 03-01: ex03-01.cpp
// 结构体定义及其中函数的实现

struct StuNode
{
    int ID;
    char name[20];
    char gender[16];
    int age;
    void Set(int id, char *n, char *g, int a);
};

// 赋值函数，为结构体中的数据赋值
void StuNode::Set(int id, char *n, char *g, int a)
{
    ID = id;
    strcpy(name, n);
    strcpy(gender, g);
    age = a;
}
```

- 结构体中可以有函数

```cpp
// 例 03-02: ex03-02.cpp
// 结构体的定义

struct StuNode
{
   int ID;
   char name[20];
   char gender[16];
   int age;

   // 结构体中的赋值函数
   void Set(int id, char *n, char *g, int a)
   {
      ID = id;
      strcpy(name, n);
      strcpy(gender, g);
      age = a;
   }
};
```

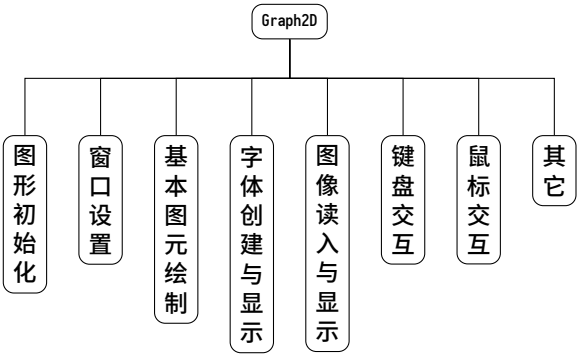
▶ 新结构体的使用

```
struct 结构类型名 {
  数据类型 1 成员变量 1;
  数据类型 2 成员变量 2;
  …;
  数据类型 n 成员变量 n;
  函数返回类型 函数名 (参数列表);
};

[struct] 结构类型名 变量名;
```

C++ 中可以省略

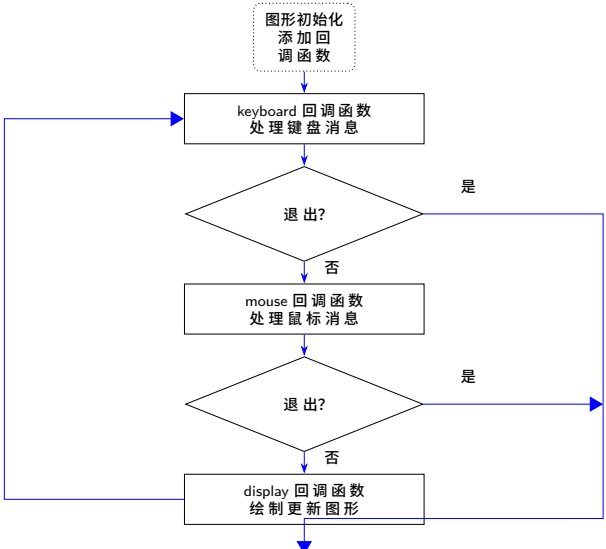
```
// 例 03-03: ex03-03.cpp

struct StuNode
{
   int ID;
   char name[20];
   char gender[16];
   int age;

   // 赋值函数，以同类型的对象为参数
   void Set(StuNode inNode)
   {
      ID = inNode.ID;
      strcpy(name, inNode.name);
      strcpy(gender, inNode.gender);
      age = inNode.age;
   }
};
int main()
{
   StuNode myNode0;
}
```

▶ Graph2D 功能结构[1]

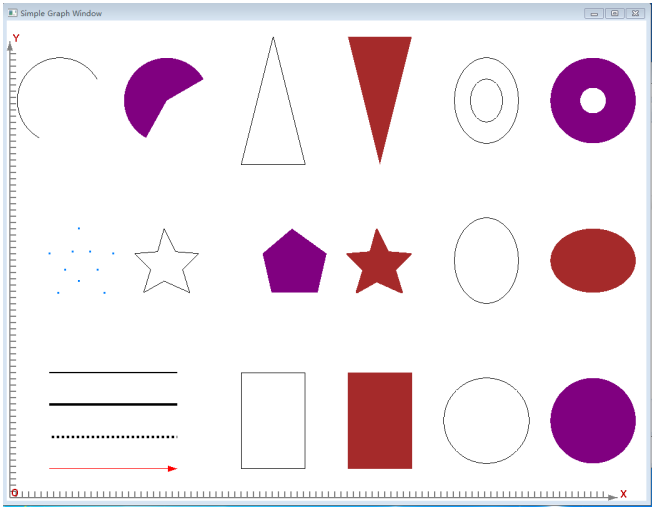


[1]详情请参考软件包中的Graph2D Specification V1.0.pdf文件

Graph2D 运行机制

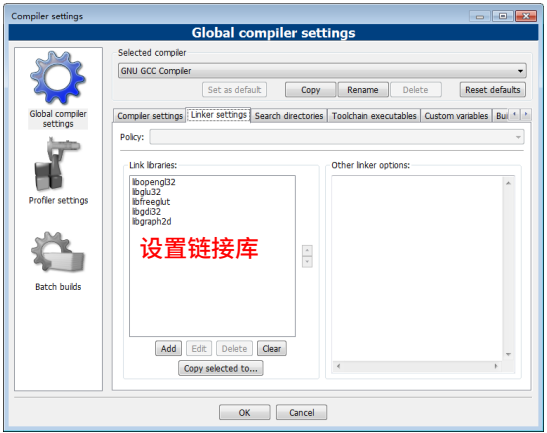


- Graph2D 坐标系统（二维笛卡尔右手坐标系）

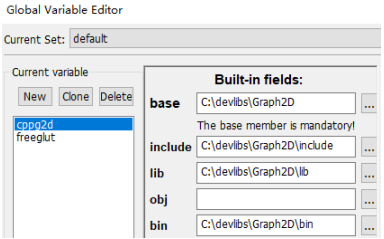


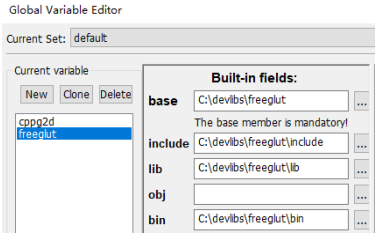
- 图形库的安装
  - Graph2D 库包含三个文件
    - graph2d.h
    - libgraph2d.a
    - graph2d.dll
  - 安装
    - 手动安装：参见 Readme.txt 文件
    - 自动安装：运行 InstallGraph2D.exe

- 图形库在 Code::blocks 中的配置
  - 在菜单 Settings > Compiler > Link settings 中添加 libgdi32、libopengl32、libglu32、libfreeglut 和 libgraph2d



- Graph2D 图形库 **Code::Blocks** 开发向导[2]
  - 将 **devLibs** 文件夹及其中所有文件保持目录结构 拷贝到任意位置 (如 C 盘根目录)
  - 将 **Graph2D** 文件夹和 **config.script** 复制到 Code::Blocks 安装目录的 **wizard** 文件夹[3]
  - 在菜单 Settings 》 Global variables... 中添加 cppg2d和freeglut环境变量[4]



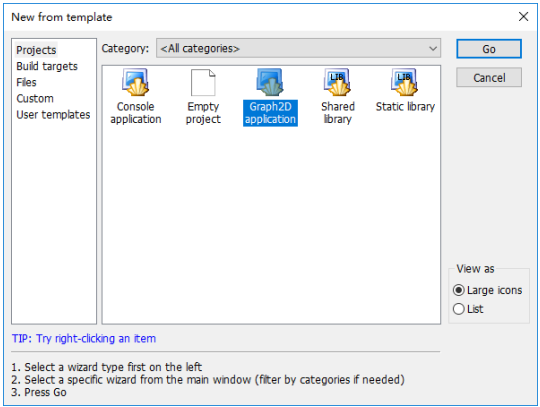


[2]详见：https://gitee.com/registor/Graph2D

[3]如：C:\Program Files (x86)\CodeBlocks\share\CodeBlocks\templates\wizard

[4]注意要与 devLibs 在自己硬盘上的路径一致

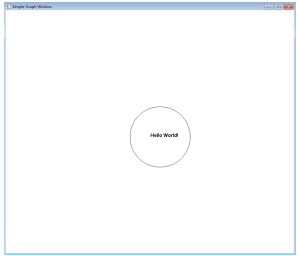
- Graph2D 图形库 Code::Blocks 开发向导[5]
  - 在向导中选择 Graph2D application



[5]详见：https://gitee.com/registor/Graph2D

- Graph2D 骆驼式命名法
  - 第一个单词小写
  - 后面的单词首字母大写
  - 示例：showCoordinate

```
// 例 03-35: ex03-35.cpp
// C++ 基本绘图函数的演示

#include <Graph2D.h>

using namespace graph;

void display()
{
    circle(512, 384, 100);
    putText(480, 384, "Hello World!");
}

int main(int argc, char *argv[])
{
    initGraph(display);
    return 0;
}
```

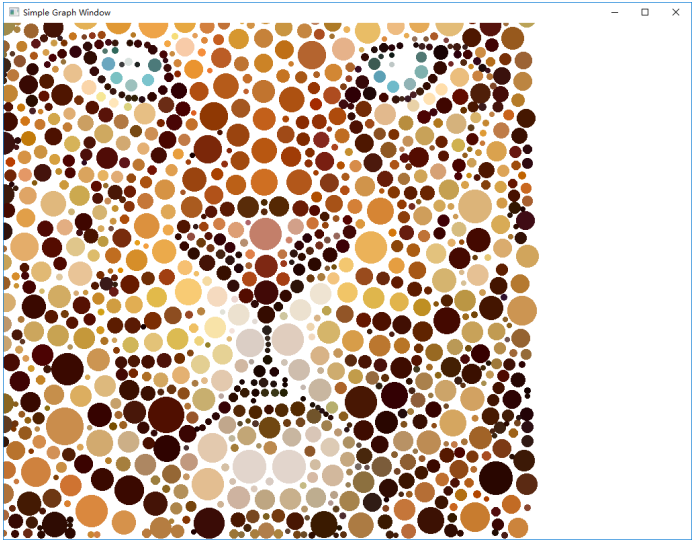


Graph2D 简单图形库的使用

```cpp
// 例 03-03-01: ex03-03-01.cpp

#include <Graph2D.h>
using namespace graph;
struct myRect
{
    double x;
    double y;
    double width;
    double height;

    void Draw()
    {
        setColor(RED);
        fillRectangle(x, y,
           x + width, y + height);
    }

    void Move()
    {
        x += 10;
        y += 10;
    }
};
```

```cpp
// 例 03-03-02: ex03-03-02.cpp

myRect r = {10.0, 10.0,
            200.0, 100.0};

void display()
{
    r.Draw();
}

void keyboard(unsigned char key)
{
    r.Move();
}

int main(int argc, char *argv[])
{
    initGraph(display, keyboard);

    return 0;
}
```

Graph2D 简单图形库的使用



- 面向过程程序设计
  - 程序模块由函数（过程）组成
  - 数据与函数分离，通过参数传递给函数
- 面向对象程序设计
  - 程序模块由类组成
  - 函数与数据是一个整体（封装）
  - 访问限制
  - 继承与多态性

- 一个包含函数的结构体

```
class 类名
{
public: //存取权限控制符
    公有数据成员或公有函数成员的定义;
protected:
    保护数据成员或保护函数成员的定义;
private:
    私有数据成员或私有函数成员的定义;
};
```

类定义结束符

▶ 实例

```
// 例 03-13: ex03-13.cpp
// 定义类 Stash，对其中成员（数据或函数）
// 根据需要赋予不同的权限

class Stash
{
private:
     int size; // Number of a spaces
     int next; // Next empty space
     int *a;// Dynamically allocated arrays
     void inflate(int increase);
public:
     void initialize();
     void cleanup();
     int add(int element);
     int fetch(int index);
     int count();
};
```

▶ 成员变量（数据成员）

```
// 例 03-14: ex03-14.cpp
// 数据成员为结构体类型的类

struct Record
{
    char name[20];
    int score;
};
class Team
{
private:
    int num;    //基本类型
    Record *p; //构造类型
};
```

```
// 例 03-15: ex03-15.cpp
// 关于类中数据成员定义为类类型的示例

class Team;    //已定义的类
class Grade
{
    Team  a;  // 已定义的类类型
    Grade *p; // 正在定义的类类型定义指针成员
    Grade &r; // 正在定义的类类型定义引用成员
    Grade b;  // 错误！使用了未定义完整的类
};
```

数据成员描述了类对象所包含的数据类型，数据成员的类型可以是 C++基本数据类型，也可以是构造数据类型。

- 成员函数
  - 对类中的数据成员实施的操作
  - 消息传递

```
// 例 03-16: ex03-16.cpp
// 类 Stash 成员函数 add 的实现

// 在 a 的尾部添加元素 element，若 a 中已无空间，则调用
// inflate 函数增加 increment 个单位
// 函数返回 a 中最后一个元素的 index

int Stash::add(int element)
{
    if(next >= size)
        inflate(increment);
    a[next] = element;
    next++;
    return (next - 1);
}
```

- 成员函数
  - 既可以放在类外定义，也可放在类中定义（内联函数）

```
// 例 03-17: ex03-17.cpp
// 类中成员（数据或函数）默认权限为 private
// 成员函数也可以在类定义中实现

class Stash
{
    int size; // Number of a spaces
    int next; // Next empty space
    int *a;// Dynamically allocated arrays
    void inflate(int increase);
public:
    int add(int element)
    {
        if(next >= size)
            inflate(increment);

        a[next] = element;
        next++;
        return(next - 1);
    }
};
```

- 成员函数
  - 成员函数中可以使用该类定义变量

```
// 例 03-18: ex03-18.cpp
// 类类型的变量作为函数参数，返回值的演示

class Clock
{
private:
    int H, M, S;
public:
    Clock AddTime(Clock C2)    // 形参为 Clock 类型的变量
    {
        Clock T;               // 函数体中定义了 Clock 类型的变量
        ...
        return T;              // 返回类型为 Clock 类型
    }
};
```

- 与新结构体的区别与联系
  - The only difference between a class and a struct is that by default all members are public in a struct and private in a class.

- 类是包含函数的自定义数据类型，它不占内存，是一个抽象的“虚”体
- 使用已定义的类建立对象与用数据类型定义变量一样
- 对象建立后，对象占据内存，变成了一个“实”体
- 类与对象的关系就像数据类型与变量的关系一样

对象的建立

```
类名 对象名;
```

对象的使用

```
对象名.属性;
对象名.成员函数名（参数);
```

```
// 例 03-19: ex03-19.cpp
// 创建 Stash 的实例，调用相关函数进行处理

int main()
{
    Stash s;

    s.initialize();

    for(int i = 0; i < 100; i++)
        s.add(i);

    for(int j = 0; j < s.count(); j++)
        cout << "No." << j << " = "
             << s.fetch(j) << endl;

    s.cleanup();
}
```

- 成员的存取控制
- 存取控制属性
  - 公有类型 (**public**): 适用于完全公开的数据
  - 私有类型 (**private**): 适用于不公开的数据
  - 保护类型 (**protected**): 适用于半公开的数据

存取属性	意义	可存取对象
public	公开 (公有)	该类及基所有对象
protected	保护	该类及其子类成员
private	私有	该类成员

成员的存取控制

```
// 例 03-20: ex03-20.cpp
// 类中成员（数据或函数）protect 权限的示例

class Stash
{
    int size; // Number of a spaces
    int next; // Next empty space
    int *a;// Dynamically allocated arrays
protected:
    void inflate(int increase);
public:
    void initialize();
    void cleanup();
    int add(int element);
    int fetch(int index);
    int count();
};
```

```
Stash s;
int len = s.size;  ✘
s.inflate(100);    ✘
s.initialize();    ✔
```

- 句柄类
  - 公有接口
  - 私有指针指向实际数据

```
#ifndef HANDLE_H_INCLUDED
#define HANDLE_H_INCLUDED

class Handle{
public:
   void initialize();
   void cleanup();
   int read();
   void change(int);
private:
   // 前向引用说明
   struct Inner;
   // 指向数据成员
   Inner * pointer;
};

#endif // HANDLE_H_INCLUDED
```

```
#include "handle.h"

struct Handle::Inner
{
   int i;
};

void Handle::initialize()
{
   pointer = new Inner;
   pointer->i = 0;
}

void Handle::cleanup()
{
   delete pointer;
}

int Handle::read()
{
   return pointer->i;
}

void Handle::change(int x)
{
   pointer->i = x;
}
```

▶ 初始化和清理工作

```
// 例 03-21: ex03-21.cpp
// 用类定义栈类型

class ch_stack
{
private:
    char *s; //栈的内容保存在 s 中
    int tp; //栈顶指示器，栈空为－ 1
    int size;
public:
    void init();
    void release();
    void push(char c);
    char pop();
    ...
};
```

```
ch_stack s;
s.init();        初始化
…
s.release();     清理
```

若无构造和析构函数，需显式调用函数

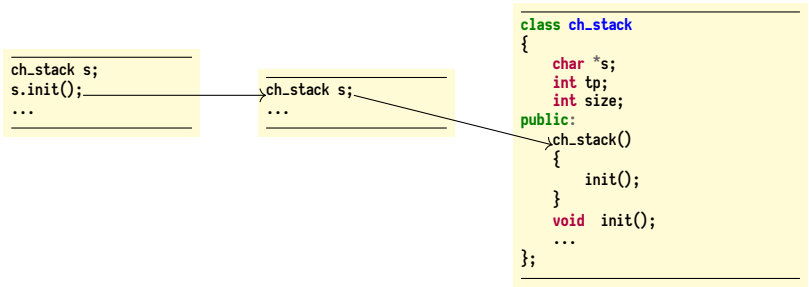
```
ch_stack s;
s.init();
…
s.release();
```

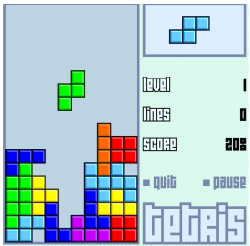
```
void ch_stack::init()
{
    s = (char *)malloc(STACK_INIT_SIZE);
    tp = EMPTY;
    size = STACK_INIT_SIZE;
}
```

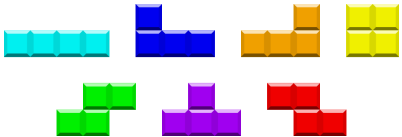
```
void ch_stack::release()
{
    if(size != 0)
    {
        free(s);
    }
}
```

- 构造函数
  - 系统自动调用的初始化函数
  - 函数名与类名相同
  - 无返回值
  - 公有函数
- 委托构造
  - 委托其他构造函数完成初始化
- 析构函数
  - 析构函数没有返回值
  - 析构函数没有任何参数，不能被重载
  - 析构函数在对象消失时由系统自动调用
- 拷贝构造函数
  - 在建立新对象时将已存在对象的数据成员值拷贝给新对象
  - 拷贝构造函数的形参是本类的对象的引用
- 浅拷贝和深拷贝
  - 在默认拷贝构造函数中，直接将原对象的数据成员值依次拷贝给新对象中对应的数据成员
  - 对于需要动态分配内存的场合，浅拷贝会出错

- 构造函数
  - 系统自动调用的初始化函数
  - 函数名与类名相同
  - 无返回值
  - 公有函数







类

类

```
CTetromino
{
  属性：颜色，四个方块的布局，大小，位置，速度，···
  行为：平移，旋转，加速，碰撞检测，显示，···
};
```

```
CTetromino I, J, L, O, S, T, Z;
```

对象

```cpp
// 例 03-23: ex03-23.cpp
#include "Cie2DGraph.h"
using namespace Cie2DGraph;
class CTetromino
{
    unsigned long color;
    int vxInit, vyInit;
    int vx[4], vy[4];
    int size;
    float speed;
public:
    CTetromino();
    CTetromino(char inType,
               unsigned long inColor,
               int inSize = 20,
               float inSpeed = 1);
    void Translate(int xOffset, int yOffset);
    void Draw();
    ...
};
```

构造函数可以重载

带默认形参的构造函数

```cpp
// 例 03-24: ex03-24.cpp
// 类 CTetromino 默认构造函数的实现

CTetromino::CTetromino()
{
    size = 20;
    speed = 1;
    color = 0xFF0000;
    vxInit = 600 / (2 * size) - 2;
    vyInit = 800 / size - 1;
    vx[0] = 0; vy[0] = 0;
    vx[1] = 1; vy[1] = 0;
    vx[2] = 2; vy[2] = 0;
    vx[3] = 3; vy[3] = 0;
}
```

```cpp
// 例 03-25: ex03-25.cpp
// 类 CTetromino，有参数构造函数实现

CTetromino::CTetromino(char inType,
           unsigned long inColor,
           int inSize, float inSpeed)
{
   size = inSize;
   speed = inSpeed;
   color = inColor;
   vxInit = 600 / (2 * size) - 2;
   vyInit = 800 / size - 1;
   switch (inType)
   {
   case 'I': case 'i':
      vx[0] = 0; vy[0] = 0;
      vx[1] = 1; vy[1] = 0;
      vx[2] = 2; vy[2] = 0;
      vx[3] = 3; vy[3] = 0;
      break;
      .
      .
      .
   case 'T': case 't':
      vx[0] = 0; vy[0] = 0;
      vx[1] = 1; vy[1] = 0;
      vx[2] = 2; vy[2] = 0;
      vx[3] = 1; vy[3] = 1;
      break;
   default:
      vx[0] = 0; vy[0] = 0;
      vx[1] = 1; vy[1] = 0;
      vx[2] = 2; vy[2] = 0;
      vx[3] = 3; vy[3] = 0;
      break;
   }
}
```

- 初始化列表实现
  - 初始化列表唯一
  - 保证参数一致性

```cpp
// 例 03-25-01-delegating : ex03-25-01.cpp
#include <iostream>

using namespace std;

class X
{
public:
   X(int _a, int _b, int _c):a(_a), b(_b), c(_c)
   {cout << "X(int, int, int)" << endl;}
   X(int _a, int _b):X(_a, _b, 0)
   {cout << "X(int, int)" << endl;}
   X(int _a):X(_a, 0, 0)
   {cout << "X(int)" << endl;}
   X():X(1, 1)
   {c = 1; cout << "X()" << endl;}
public:
   int a, b, c;
};

int main(){
   cout << "1: " << endl;
   X one(1, 2, 3);
   cout << "2: " << endl;
   X two(1, 2);
   cout << "3: " << endl;
   X three(1);
   cout << "4: " << endl;
   X four;
}
```

- 对象消失时的清理工作（如释放内存单元）
- 定义

```
~类名 ();
```

```
// 例 03-26: ex03-26.cpp
// 演示类中析构函数的定义及实现

class ch_stack
{
    char *s;
    int tp;
    int size;
public:
    ~ch_stack()
    {
        release();
    }
    void  release();
    ...
};
```

- 析构函数没有返回值
- 析构函数没有任何参数，不能被重载
- 析构函数在对象消失时由系统自动调用

```
// 例 03-27: ex03-27.cpp
int main
{
    ch_stack s;
    s.init();
    ...
    s.release();
    return 0;
}
```

```
// 例 03-28: ex03-28.cpp
int main
{
    ch_stack s;
    ...
    return 0;
}
```

- 在新建对象时将已有对象的数据成员值拷贝给新对象
- 拷贝构造函数的形参是本类的对象的引用

```
类名 (类名& 对象名)
{
  .
  .
  .
};
```

```
// 例 03-29: ex03-29.cpp
// 类默认构造函数的定义

class ch_stack
{
    char *s;
    int tp;
    int size;
public:
    ch_stack();
     ch_stack(ch_stack &sObj);
    ...
};
```

- 拷贝构造函数调用时机
  - 用类的一个对象去初始化该类的另一个对象时
  - 调用函数中，将对象作为实参传递给形参时
  - 如函数返回值是类的对象，函数执行完成后将返回值返回时
- 在默认的拷贝构造函数中，是直接将原对象的数据成员值依次拷贝给新对象中对应的数据成员
- 在对于需要动态分配内存的场合，默认的拷贝构造函数会出错

浅拷贝

```
// 例 03-30: ex03-30.cpp
// 类拷贝构造函数的实现（浅拷贝）

ch_stack::ch_stack(ch_stack & sObj)
{
    size = sObj.size;
    tp = sObj.tp;
    s = sObj.s;
}
```

```
// 例 03-31: ex03-31.cpp
// 类拷贝构造函数的效果演示

int main()
{
    ch_stack s;

    for(int i = 0; i < 7; i++)
        s.push('a' +i);

    // 自动调用拷贝构造函数
    ch_stack sCopy(s);

    while(!s.empty()) s.pop();

    for(int i = 0; i < 7; i++)
        s.push('1' +i);

    while(!sCopy.empty)
        cout << sCopy.pop();
    cout << endl;
}
```

深拷贝

```
// 例 03-32: ex03-32.cpp
// 类拷贝构造函数的实现（深拷贝）

ch_stack::ch_stack(ch_stack & sObj)
{
    size = sObj.size;
    tp = sObj.tp;
    s = NULL;
    if (size != 0)
    {
        s = (char *)malloc(size);
        for (int i = 0; i <= tp; i++)
            s[i] = sObj.s[i];
    }
}
```

```
// 例 03-31: ex03-31.cpp
// 类拷贝构造函数的效果演示

int main()
{
    ch_stack s;

    for(int i = 0; i < 7; i++)
        s.push('a' +i);

    // 自动调用拷贝构造函数
    ch_stack sCopy(s);

    while(!s.empty()) s.pop();

    for(int i = 0; i < 7; i++)
        s.push('1' +i);

    while(!sCopy.empty)
        cout << sCopy.pop();
    cout << endl;
}
```

- 相同点
  - 系统自动建立默认构造函数和析构函数
  - 系统自动调用
  - 无返回值
  - 必须是公有函数
- 相异点
  - 功能不同
  - 函数名不同
  - 构造函数可被重载，也可带 (默认) 形参，析构函数不可以

- 对象指针指向对象存放的地址
- 定义与使用

```
类名 * 对象指针名;
对象指针名 -> 数据成员;
对象指针名 -> 数据函数;
```

- 优点
  - 地址传递
  - 使用对象指针效率高

- 对象引用与被引用对象共享地址空间
- 定义与使用

```
类名 &对象指针名;
对象指针名->数据成员;
对象指针名->数据函数;
```

- 示例

```
CTetromino tetrObj('I', 0xFF0000, 30, 0.5);

CTetromino &tetrRef=tetrObj;
```

- 以与对象为元素的数组
- 定义与初始化

```
类名 对象数组名 [n];
初始化: 数组名 [n] = {类名 (初始值1, 初始值2, ···),
                      类名 (初始值1, 初始值2, ···),
                                    ⋮
                      类名 (初始值1, 初始值2, ···)
                     };
```

- 示例

```
CTetromino tetrObj[2] = {CTetromino('I', 0xFF0000, 30, 0.5),
                         CTetromino('T', 0x00FF00, 30, 0.5)
                        };
```

- 动态建立的对象
- 定义与初始化

```
类名 对象指针名 = new 类名 (初始值1, 初始值2,···);
delete 对象指针名;

类名 对象指针名 = new 类名 [n];
delete [] 对象指针名;
```

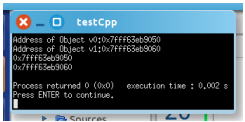
- 示例

```
CTetromino *tetrObj;
tetrObj = new CTetromino('I', 0xFF0000, 30, 0.5);
          .
          .
          .
CTetromino *tetrObj;
tetrObj = new CTetromino
          .
          .
          .
tetrObj->Draw();
tetrObj[0].Draw();
```

- 系统预定义指针，指向当前对象 (即当前对象的地址)

```cpp
// 例 03-33-01: ex03-33-01.cpp
// 构造函数重载，理解 this 指针

class Point2D
{
    float x, y;
public:
    Point2D()
    {
        x = y = 0;
    }
    Point2D(int x, int y)
    {
        this->x = x;
        this->y = y;
    }

    void Output()
    {
        cout << this << endl;
    }
};
```

```cpp
// 例 03-33-02: ex03-33-02.cpp
//  this 指针的值及其意义

int main()
{
    Point2D v0, v1;

    cout << "Address of Object v0:"
         << &v0 << endl;
    cout << "Address of Object v1:"
         << &v1 << endl;

    v0.Output();
    v1.Output();

    return 0;
}
```



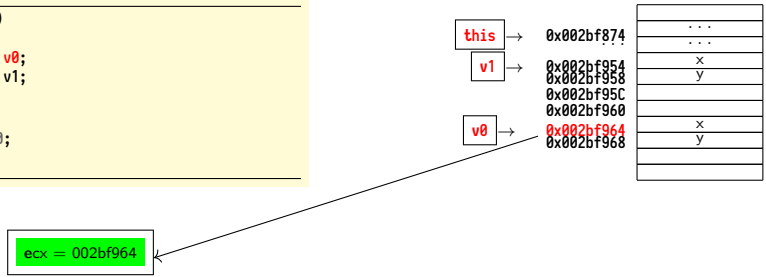
- 利用 this 指针明确成员函数当前操作的数据成员是属于哪个对象

```
// 例 03-33-01: ex03-33-01.cpp
// 构造函数重载，理解 this 指针

class Point2D
{
    float x, y;
public:
    Point2D()
    {
        x = y = 0;
    }
    Point2D(int x, int y)
    {
        this->x = x;
        this->y = y;
    }

    void Output()
    {
        cout << this << endl;
    }
};
```

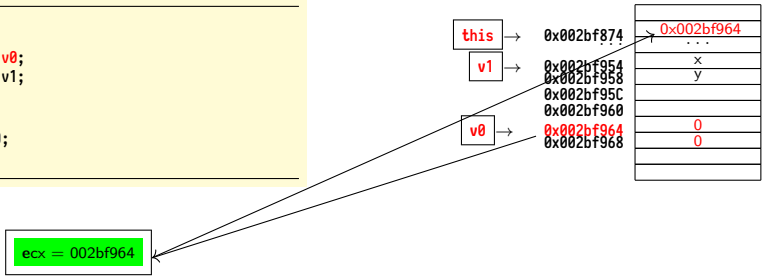
```
// 例 03-34: ex03-34.cpp

int main()
{
    Point2D v0, v1;
    ...
    return 0;
}
```

```
CPoint2D v0, v1;
 0x011B14DE  lea     ecx,[v0]
0x011B14E1  call    CPoint2D::CPoint2D
 0x011B14E6  lea     ecx,[v1]
0x011B14E9  call    CPoint2D::CPoint2D
```

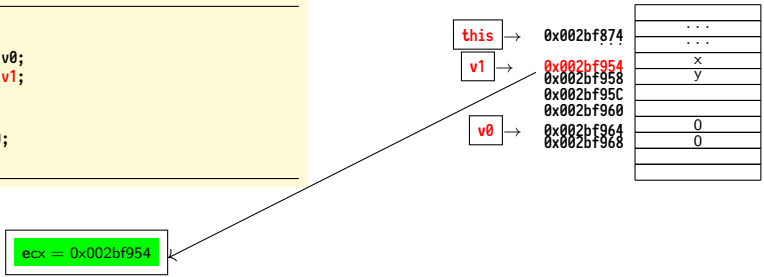
```
int main()
{
  Point2D v0;
  Point2D v1;
  .
  .
  .
  return 0;
}
```



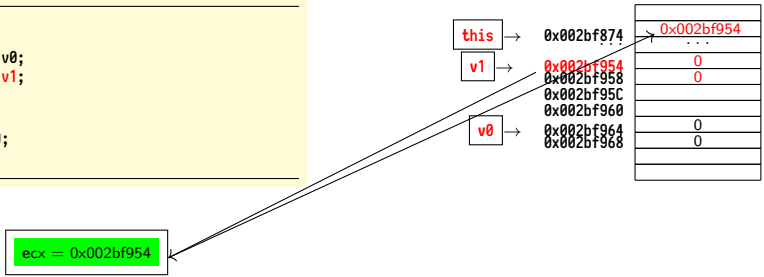
```
int main()
{
  Point2D v0;
  Point2D v1;
  .
  .
  .
  return 0;
}
```



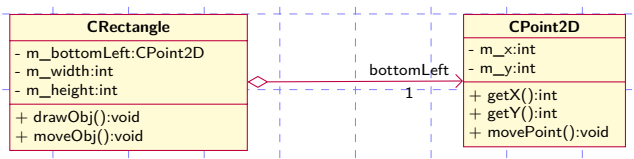
```
int main()
{
  Point2D v0;
  Point2D v1;
  .
  .
  .
  return 0;
}
```



```
int main()
{
  Point2D v0;
  Point2D v1;
  .
  .
  .
  return 0;
}
```



- 组合类：类中含有其它类的对象作为成员
- 组合对象
- 组合类的定义
  - 先定义成员类，再定义组合类



CPoint2D 类

```
// 例 03-36: ex03-36-point.h
// CPoint2D 类的定义
#ifndef CPOINT2D_H
#define CPOINT2D_H
class CPoint2D
{
public:
   CPoint2D();
   CPoint2D(int x, int y);
   CPoint2D(CPoint2D &pt);
   int getX(){
      return m_x;
   }
   int getY(){
      return m_y;
   }
   void movePoint();
private:
   int m_x;
   int m_y;
};
#endif // CPOINT2D_H
```

```
// 例 03-36: ex03-36-point.cpp
// 重载构造函数及不同构造函数的实现
#include <iostream>
#include "point.h"
using namespace std;

// 默认构造函数
CPoint2D::CPoint2D(){
   m_x = 0;
   m_y = 0;
}

// 带参构造函数
CPoint2D::CPoint2D(int x, int y):
   m_x(x), m_y(y){
}

// 拷贝构造函数
CPoint2D::CPoint2D(CPoint2D &pt){
   m_x = pt.m_x;
   m_y = pt.m_y;
}
void CPoint2D::movePoint(){
   m_x += 10;
   m_y += 10;
}
```

▶ CRectangle 类

```
// 例 03-36: ex03-36.cpp
// CRectangle 类的定义

#ifndef CRECTANGLE_H
#define CRECTANGLE_H
#include "point.h"
class CRectangle{
public:
   CRectangle();
   CRectangle(int x, int y, int width, int height);
   CRectangle(CPoint2D &pt, int width, int height);
   void drawObj();
   void moveObj();
private:
   CPoint2D m_bottomLeft;
   int m_width;
   int m_height;
};

#endif // CRECTANGLE_H
```

CRectangle 类

```
// 例 03-36: ex03-36.cpp
// CRectangle 类的实现

#include <Graph2D.h>
#include "rectangle.h"
using namespace graph;
// 默认构造函数
CRectangle::CRectangle():m_bottomLeft(0, 0)
{
   m_width = 0;
   m_height = 0;
}
// 带参构造函数
CRectangle::CRectangle(int x, int y, int width, int height):
   m_bottomLeft(x, y), m_width(width), m_height(height)
{
}
// 带参构造函数
CRectangle::CRectangle(CPoint2D &pt, int width, int height):
   m_bottomLeft(pt), m_width(width), m_height(height)
{
}
void CRectangle::drawObj()
{
   setColor(RED);
   fillRectangle(m_bottomLeft.getX(), m_bottomLeft.getY(),
            m_bottomLeft.getX() + m_width,
            m_bottomLeft.getY() + m_height);
}
void CRectangle::moveObj()
{
   m_bottomLeft.movePoint();
}
```

初始化列表

▶ 使用类对象

```
// 例 03-36: ex03-36.cpp
// C++ 基本绘图函数的演示

#include <Graph2D.h>
#include "rectangle.h"
using namespace graph;
CRectangle rect(100, 100, 200, 100);
void display()
{
   rect.drawObj();
}
void keyboard(unsigned char key)
{
   rect.moveObj();
}
int main(int argc, char *argv[])
{
   initGraph(display, keyboard);
   return 0;
}
```

使用 graph 名子空间

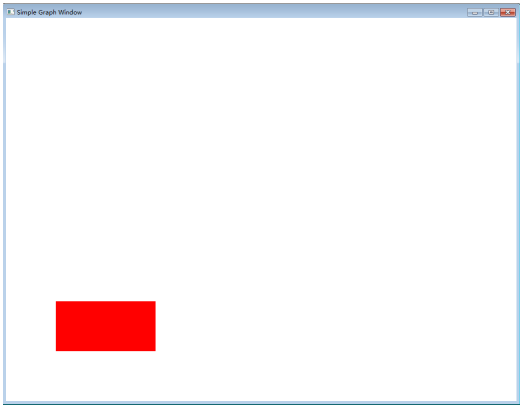
定义 rect 对象，并初始化

调用成员函数绘图

调用成员函数移动

函数指针 (回调函数)

- 使用类对象



- 非成员函数 (或外部函数) 访问私有成员 (保护成员)
  - 设置访问控制属性为**public**
  - 破坏了类的封装性和隐藏性
- 友元
  - 友元函数
  - 友元类
- 友元不是类的成员，但能够访问类中被隐蔽的信息

友元函数

```
// 例 ex03-38: ex03-38.cpp
class  A
{
private:
    int  i ;
    void MemberFun(int) ;
    friend void FriendFun(A * , int);
} ;

void FriendFun(A * ptr , int x)
{
    ptr -> i = x ;
}
void A::MemberFun( int x )
{
    i = x ;
}
```

友元函数的声明

友元函数的定义

友元函数

```
// 例 ex03-39: ex03-39.cpp
class  A
{
private:
    int  i ;
    void MemberFun(int) ;
    friend void FriendFun(A * , int) ;
} ;

friend void  A::FriendFun(A * ptr , int x)
{
    ptr->i = x ;
}
void A::MemberFun( int x )
{
    i = x ;
}
```

friend必须出现在类中，定义时不能使用域操作

友元函数

```
// 例 03-40: ex03-40.cpp
// 类友元函数的定义及实现

class  A
{
private:
    int  i ;
    void MemberFun(int);
    friend void FriendFun(A * , int);
};

void FriendFun(A * ptr , int x)
{
     i = x;
}
void A::MemberFun(int x)
{
    i = x;
}
```

friend不能直接访问成员变量
必须通过参数传递访问私有成员

▶ 求两点间的距离

```
// 例 03-41: ex03-41.cpp
// 类友元函数的定义

//point2d.h
#ifndef POINT2D_INCLUDED
#define POINT2D_INCLUDED
class CPoint2D
{
private:
    float x, y;
public:
    CPoint2D()
    {
        x = y = 0;
    }
    CPoint2D(float x, float y)
    {
        this->x = x;
        this->y = y;
    }
    friend float Distance(CPoint2D p1,
                          CPoint2D p2);
};

#endif // POINT2D_INCLUDED
```

```
// 例 ex03-41-main: ex03-41-main.cpp
#include <iostream>
#include <cmath>
#include "point2d.h"
using namespace std;
float Distance(CPoint2D p1,
               CPoint2D p2){
    float dx = p1.x - p2.x;
    float dy = p1.y - p2.y;
    return sqrt(dx * dx + dy * dy );
}
int main(){
    CPoint2D v0(1, 1), v1(4, 5);
    cout <<  Distance (v0, v1) << endl;
    return 0;
}
```

不用通过对象调用友元函数

- 友元函数不能直接访问成员变量，但可以通过参数传递操作对象中的所有成员

- 友元函数在类中声明，但在类外定义时勿需域操作符

- 友元函数调用时勿需通过对象

▶ 前向引用声明

```
// 例 03-42: ex03-42.cpp

class B; // 前向声明
class A
{
public:
    void funA(B b);
};

class B
{
public:
    void funB(A b);
};
```

```
// 例 03-43: ex03-43.cpp
// 类的前向声明，参数为引用型的函数示例

class Matrix;   // 类的前向声明
class Vector{
    float e[4];
    //...
    friend Vector Multi(const Matrix&, const Vector&);
};
class Matrix{
    Vector v[4];
    //...
    friend Vector Multi(const Matrix&, const Vector&);
};
Vector Multi(const Matrix &m, const Vector &v){
    Vector r;
    for(int i = 0; i < 4; i++){ //r[i]=m[i]*v;
        r.e[i] = 0;
        for(int j = 0; j < 4; j++)
            r.e[i] += m.v[i].e[j] * v.e[j];
    }
    return  r;
}
```

友元类

- 如果 A 类声明为 B 类的友元，则将 A 类称为 B 类的友元类
- 若 A 类为 B 类的友元类，则 A 类的所有成员函数都是 B 类的友员函数
- 定义

```
class B
{
  .
  .
  .
  friend class A;
}
```

▶ 公有数据成员

```
// 例 03-44: ex03-44.cpp
// CPoint2D 的定义

class CPoint2D
{
public:
    float x, y;
    CPoint2D()
    {
        x = y = 0;
    }
    CPoint2D(float x, float y)
    {
        this->x = x;
        this->y = y;
    }
    void Translate(float x, float y);
    void Scale(float r);
    void Rotate(float angle);
};
```

```
// 例 03-44: ex03-44.cpp
// CRectangle 的定义

class CRectangle
{
    unsigned long color;
     CPoint2D wPos, lv1, lv2, lv3, lv4;
public:
    CRectangle();
    CRectangle( unsigned long color,
                CPoint2D w,
                float width,
                float height);
    void Translate(float xOffset,
                   float yOffset);
    void Rotate(float angle);
    void Scale(float r);
    void Draw();
};
```

▶ 友元类

```
// 例 03-45: ex03-45.cpp

class CPoint2D
{
     float x, y;
public:
    CPoint2D()
    {
        x = y = 0;
    }
    CPoint2D(float x, float y)
    {
        this->x = x;
        this->y = y;
    }
    void Translate(float x, float y);
    void Scale(float r);
    void Rotate(float angle);
    friend class CRectangle;
};
```

```
// 例 03-44: ex03-44.cpp
// CRectangle 的定义

class CRectangle
{
    unsigned long color;
     CPoint2D wPos, lv1, lv2, lv3, lv4;
public:
    CRectangle();
    CRectangle( unsigned long color,
                CPoint2D w,
                float width,
                float height);
    void Translate(float xOffset,
                   float yOffset);
    void Rotate(float angle);
    void Scale(float r);
    void Draw();
};
```

- 利用友元类访问类的私有成员

```
// 例 ex03-46: ex03-46.cpp
void CRectangle::Draw()
{
    SetPenColorHex(color);
    DrawFillBox(wPos.x + lv1.x, wPos.y + lv1.y,
                wPos.x + lv2.x, wPos.y + lv2.y,
                wPos.x + lv3.x, wPos.y + lv3.y,
                wPos.x + lv4.x, wPos.y + lv4.y);
}
```

访问私有成员

友元类注意事项

- 友员关系是非传递的
  - Y 类是 X 类的友员，Z 类是 Y 类的友员，但 Z 类不一定是 X 类的友员
- 友员关系是单向的
  - 若 Y 类是 X 类的友员，则 Y 类的成员函数可以访问 X 类的私有和保护成员，反之则不然
- 友员提高了数据的共享性，但在一定程度上削弱了数据的隐藏性

▶ 常对象

```
// 例 03-47: ex03-47.cpp
// CPoint2D 的实现

class CPoint2D
{
    float x, y;
public:
    CPoint2D()
    {
        x = y = 0;
    }
    CPoint2D(float x, float y)
    {
        this->x = x;
        this->y = y;
    }
    void Translate(float x, float y)
    {
        this->x = this->x + x;
        this->y = this->y + y;
    }
    //…
};
```

```
// 例 03-47: ex03-47.cpp

int main()
{
    const CPoint2D p1(1, 2);
    p1.Translate(1, 1);✘

    CPoint2D p2;
    p1 = p2;✘
}
```

- 常数据成员能通过初始化列表获得初值

```
// 例 03-48: ex03-48.cpp

class A
{
private:
    const int& r;          //常引用数据成员
    const int a;           //常数据成员
    static const  int b;   //静态常数据成员
public:
    A(int i): a(i), r(a)
    {
        cout << "constructor!" << endl;
    };
    void display()
    {
        cout << a << "," << b << "," << r << endl;
    }
};
```

- 使用const关键字修饰的用于访问类的常对象的函数
- 定义

```
返回类型 成员函数名（参数表）const;
```

```
// 例 03-49: ex03-49.cpp
// CPoint2D 的定义

class CPoint2D{
   float x, y;
public:
   CPoint2D(){
      x = y = 0;
   }
   CPoint2D(float x, float y){
      this->x = x; this->y = y;
   }
   float GetX() const {
      return x;
   }
   float GetY() const;
   void Set(float x, float y){
      this->x = x; this->y = y;
   }
};
```

```
// 例 03-49: ex03-49.cpp

float CPoint2D::GetY() const
{
    return y;
}
int main()
{
    const CPoint2D p(1, 2);
    float x = p.GetX();
    float y = p.GetY();
}
```

使用

```
// 例 ex03-50-constfun.cpp
class CPoint2D{
    float x, y;
public:
    CPoint2D(){
        x = y = 0;
    }
    CPoint2D(float x, float y){
        this->x = x; this->y = y;
    }
    float GetX() const {
        return x;
    }
    float GetY() const;
    void Set(float x, float y) const {
        this->x = x; this->y = y;✘
    }
};
```

常成员函数不能更新对象的数据成员，也不能调用对象的非常成员函数

```
// 例 03-49: ex03-49.cpp

float CPoint2D::GetY() const
{
    return y;
}
int main()
{
    const CPoint2D p(1, 2);
    float x = p.GetX();
    float y = p.GetY();
}
```

- 使用

```
// 例 ex03-51-constfun.cpp
class CPoint2D{
    float x, y;
public:
    CPoint2D(){
        x = y = 0;
    }
    CPoint2D(float x, float y){
        this->x = x; this->y = y;
    }
    float GetX() const{
        return x;
    }
    float GetY() const{
    void Set(float x, float y) {
        this->x = x; this->y = y;
    }
};
```

常对象可以调用常成员函数

```
// 例 ex03-51-main.cpp
float CPoint2D::GetY() const
{
    return y;
}
int main()
{
    const CPoint2D p(1, 2);
    float x = p.GetX();
    float y = p.GetY();
}
```

- **mutable易变量**
  - 任意可变量
  - 不受const常对象限制

```
// 例 03-51-mutable : ex03-51-01.cpp
class Data
{
   // mutable 成员
   mutable bool cache_valid;
   mutable string cache;
   void compute_cache_value() const;
   // ...
public:
   // ...
   string string_rep() const;
};

// 常函数中可以修改数据成员
string Data::string_rep()const{
   if(!cache_valid){
      compute_cache_value();
      cache_valid = true;
   }
   return cache;
}
```

- 不同对象之间数据成员和函数的共享
- 在内存中只有一个对应的存储区域
- 定义
  - `static` 数据类型静态成员名;
  - `static` 返回类型函数名 (){};
- 静态成员的初始化
  - 数据类型类名::静态成员名 = 初始值

▶ 例子：统计点的个数

```
// 例 03-37: ex03-37-class.cpp
class CPoint2D
{
    int x, y;
     static int num;
public:
    CPoint2D(){
        x = y = 0;
         num++;
    }
    CPoint2D(int x, int y){
        this->x = x;
        this->y = y;
         num++;
    }
    ~CPoint2D(){
        num--;
    }
     static int GetCounter(){
         return num;
     }
};
```

静态成员变量的初始化

```
// 例 03-37: ex03-37-main.cpp
int CPoint2D::num = 0;

int main()
{
    CPoint2D v0, v1;
    int num1, num2;
    num1 =  CPoint2D::GetCounter();
    num2 =  v0.GetCounter();
}
```

- 静态成员变量的初始化
  - 类名::静态成员变量名
- 在建立对象之前通过类就可操作静态成员
- 静态成员函数中没有this指针
  - 不能直接访问类中的非静态成员变量
  - 不能调用非静态成员函数
  - 不能声明为const或volatile

- 单件模式
  - 保证一个类仅有一个实例
  - 利用**private**访问控制与**static**成员实现

```cpp
// 例 03-37-01-singleton : ex03-37-01.cpp
#include <iostream>

using namespace std;

class Singleton
{
private:
   int num;
   static Singleton* sp;
   Singleton(int _num){num = _num;}
   Singleton(const Singleton &){}
public:
   static Singleton * getInstance(int _num);
   void handle()
   {
      if(num > 0)
      {num -= 1;}
      else
      {cout << "num is zero!" << endl;}
   }
};
```

```cpp
// 例 03-37-01-singleton : ex03-37-01.cpp
Singleton* Singleton::sp = nullptr;

Singleton* Singleton::getInstance(int _num){
   if(sp == nullptr) {sp = new Singleton(_num);}
   return sp;
}

int main(){
   Singleton* sp = Singleton::getInstance(1);
   sp->handle();
   Singleton* st = Singleton::getInstance(10);
   st->handle();
}
```

语法

声明

```
成员类型 类名::*指向数据成员的指针;
```

使用

```
对象.*指向数据成员的指针;
对象指针->*指向数据成员的指针;
```

示例

```cpp
// 例 03-38-01 : ex03-38-01-01.cpp
#include <iostream>

using namespace std;

class Data
{
public:
   int a, b, c;
   void printf() const
   {
      cout << "a = " << a
           << ", b = " << b
           << ", c = " << c << endl;
   }
};
```

```cpp
// 例 03-38-01 : ex03-38-01-02.cpp
int main()
{
   Data d, *dp = &d;

   int Data::*pmInt = &Data::a;
   // pmInt 指向 Data 的 int 成员
   dp->*pmInt = 12;
   pmInt = &Data::b;
   d.*pmInt = 24;
   pmInt = &Data::c;
   dp->*pmInt = 36;
   dp->printf();

   return 0;
}
```

语法

声明

```
返回类型 (类名::*指向成员函数的指针)(形参表);
```

使用

```
指向成员函数的指针 = &类名::函数成员名称;
```

示例

```cpp
// 例 03-38-02 : ex03-38-02-01.cpp
#include <iostream>
using namespace std;
class FuncTable
{
   void f(int) const {cout << "FuncTable::f()\n";}
   void g(int) const {cout << "FuncTable::g()\n";}
   void h(int) const {cout << "FuncTable::h()\n";}
   void i(int) const {cout << "FuncTable::i()\n";}
   static const int cnt = 4;
   void (FuncTable::*fptr[cnt])(int) const; // 函数表
public:
   FuncTable(){  // 初始化函数表
      fptr[0] = &FuncTable::f;
      fptr[1] = &FuncTable::g;
      fptr[2] = &FuncTable::h;
      fptr[3] = &FuncTable::i;
   }
```

```cpp
// 例 03-38-02 : ex03-38-02-02.cpp
   void select(int i, int j){
      if(i < 0  i >= cnt)
         return;
      (this->*fptr[i])(j);
   }
   int count() {return cnt;}
};

int main()
{
   FuncTable ft;
   for(int i = 0; i < ft.count(); i++){
      ft.select(i, 20);
   }

   return 0;
}
```

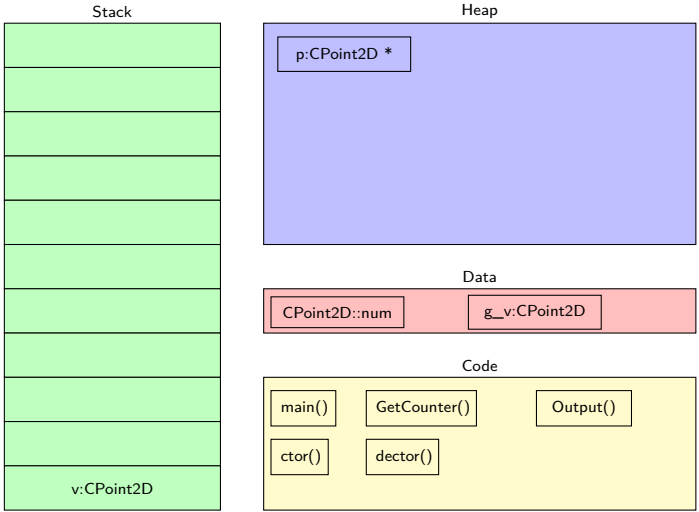
- 类只是一个类型，除静态数据成员外，在没有实例化成对象前不占任何内存
- 对象的存储
  - 数据段：全局对象、静态对象
  - 代码段：成员函数、静态函数等
  - 栈：局部对象、参数传递时的临时对象
  - 堆：动态内存分配

测试代码

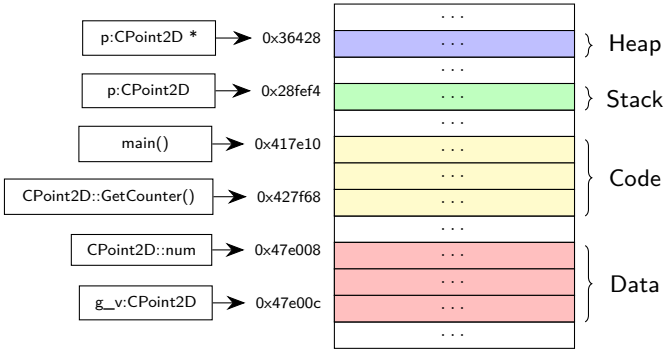
```
// 例 03-52: ex03-52.cpp
// static 成员的示例

#include <iostream>
using namespace std;
class CPoint2D{
   int x, y;
   // 静态数据成员
   static int num;
public:
   CPoint2D(){
      x = y = 0;
      num++;
   }
   CPoint2D(int x, int y){
      this->x = x;
      this->y = y;
      num++;
   }
   ~CPoint2D(){
      num--;
   }
   // 静态函数成员
   static int GetCounter(){
      return num;
   }
   void Output();
};
```

```
// 例 03-52: ex03-52.cpp
// 不同作用域对象的对比，类静态成员的地址查看

void CPoint2D::Output(){
    cout << "静态变量地址:" << &num << endl;
    cout << "静态函数地址:" << GetCounter << endl;
 }
int CPoint2D::num = 0;
CPoint2D g_v(1, 1);
int main(){
    CPoint2D v;
    CPoint2D *p;
    p = new CPoint2D;
    p->GetCounter();
    cout << "全局对象地址:" << &g_v << endl;
    cout << "局部对象地址:" << &v << endl;
    cout << "动态对象地址:" << p << endl;
    p->Output();
    delete p;
    return 0;
}
```

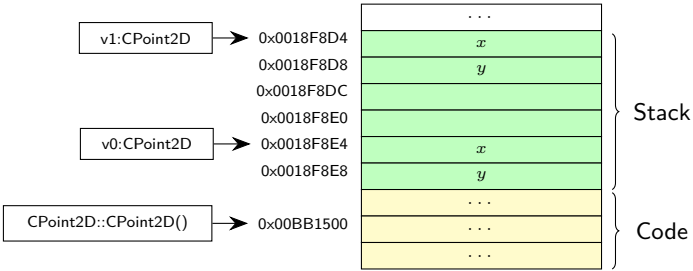
内存分配示意图



内存地址示意图



- 类的不同对象的数据成员拥用各自独立的内存空间
- 类的成员函数为该类的所有对象共享



- 全局对象数据成员占有的内存空间在程序结束时释放
- 局部对象与实参对象数据成员的内存空间在函数调用结束时释放
- 动态对象数据成员的内存空间要使用**delete**语句释放
- 对象的成员函数的内存空间在该类的所有对象生命周期结束时自动释放

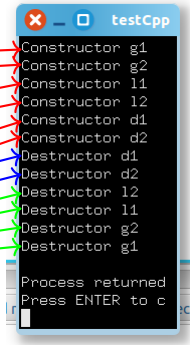
▶ 对象构造函数与析构函数调用顺序

```
// 例 03-53: ex03-53.cpp
// 构造函数，析构函数

#include <string>
#include <iostream>
using namespace std;
class base{
    string mark;
public:
    base(string str){
        mark = str;
        cout << "Constructor "
             << mark << endl;
    }
    ~base(){
        cout << "Destructor "
             << mark << endl;
    }
};
```

```
// 例 ex03-53-ctor-main.cpp
base g1("g1");
base g2("g2");
int main()
{
    base l1("l1");
    base l2("l2");
    base *d1, *d2;
    d1 = new base("d1");
    d2 = new base ("d2");
    delete d1;
    delete d2;
    return 0;
}
```

- 对象构造函数与析构函数调用顺序

```
// 例 ex03-53-ctor-main.cpp
base g1("g1");
base g2("g2");
int main()
{
    base l1("l1");
    base l2("l2");
    base *d1, *d2;
    d1 = new base("d1");
    d2 = new base ("d2");
    delete d1;
    delete d2;
    return 0;
}
```

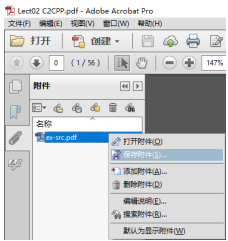


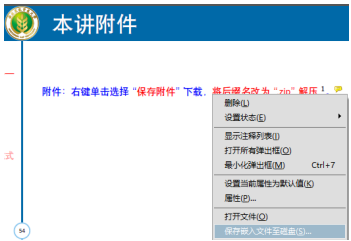
- 主要内容
  - 初始化方式
    - 设计构造函数
  - 销毁操作
    - 析构函数
  - 操作
    - 成员函数
  - 属性
    - 数据结构
    - 类型转换
    - 动态分配
  - 辅助函数
- 不适合用类表示的问题
  - 无所不能的类
  - 只有数据没有行为了的类
  - 只有行为没有数据的类
- UML 图

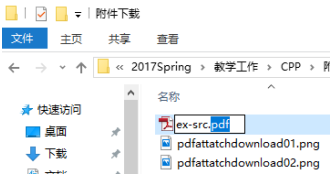
- 类是对逻辑上相关的函数与数据的封装，它是对问题的抽象描述
- 类中有数据成员与成员函数，成员的访问控制属性有 **private**、**protected**和 **public**
- 类是“型”，不占内存，使用类建立对象后，对象占有内存空间
- 建立对象时调用构造函数初始化对象的数据成员，一个类提供默认的构造函数与默认的拷贝构造函数 (浅拷贝)
- 对象使用完后系统自动调用析构函数
- 建立对象指针、对象引用均没有建立对象，不调用构造函数
- 通过对象指针使用对象成员使用 “->”, 通过对象引用使用对象成员使用 “.”
- 对象数组的定义、赋值、引用与普通数组一样。对象数组初始化需要通过构造函数完成
- 建立动态对象使用语句 “**new**”，建立动态对象数组使用语句 “**new** []”

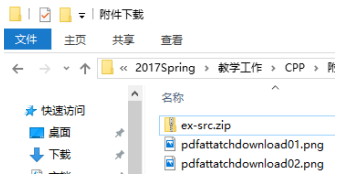
- “this” 指针是一个系统预定义的指向当前对象的指针
- 组合类是含有类对象的类，组合类对象称为组合对象。定义组合对象时调用构造函数的顺序为类中成员对象定义的顺序
- 静态数据成员是类的数据成员，独立于类存在。静态成员函数属于整个类，是该类的所有对象共享的成员函数
- 友元函数不是类的成员，但它可以访问类的任何成员。一个类的友元类可以访问该类的任何成员
- 常对象与常成员

附件：右键单击该链接，选择“保存附件”下载，将后缀名改为“.zip”解压[6] [7]。









[6] 请退出全屏模式后点击该链接。

[7] 以 Adobe Acrobat Reader 为例。

本讲结束，谢谢!
欢迎多提宝贵意见和建议

西北农林科技大学
NORTHWEST A&F UNIVERSITY
中国 · 杨凌