C++程序设计（part 2）

OOP

- Why
  - non-OO Solution

```
#include    <stdio.h>
#define     STACK_SIZE   100
struct Stack
{   int   top;
    int   buffer[STACK_SIZE];
};
void main()
{   Stack   st1, st2;
    ~~st1.top = -1;~~   安全隐患
    ~~st2.top = -1;~~
    int  x;
    push(st1,12);
    pop(st1,x);      不符合数据类型定义
    st1.buffer[2] = -1;
    st2.buffer[2] ++;
}
```

```
bool  push(Stack &s, int i)
{   if  (s.top == STACK_SIZE-1)
    {   printf(“Stack is overflow.\n”);
        return false; }
   else
   {    s.top++;  s.buffer[s.top] = i;
        return true;  }
}
```

```
bool pop(Stack &s, int &i)
{   if  (s.top == -1)
    {   printf(“Stack is empty.\n”);
        return false;   }
   else
  {   i = s.buffer[s.top];
      s.top--;
      return true;  }
}
```

OOP

```
bool  push(Stack &s, int i)
{   if  (s.top == STACK_SIZE-1)
    {  printf("Stack is overflow.\n");
       return false; }
   else
   {    s.top++;  s.buffer[s.top] = i;
       return true;  }
}
```

```
bool pop(Stack &s, int &i)
{   if  (s.top == -1)
    {   printf("Stack is empty.\n");
        return false;   }
  else
  {   i = s.buffer[s.top];
      s.top--;
      return true;  }
}
```

- OO Solution

```
#include <iostream.h>
#define   STACK_SIZE 100
struct Stack
{   int   top;
    int   buffer[STACK_SIZE];
};
void main()
{   Stack   st1, st2;
    st1.top = -1;
    st2.top = -1;
    int  x;
    push(&st1, 12);
    pop(&st1, x);
}
```

```
#include <iostream.h>
#define   STACK_SIZE 100
class Stack
{  private:
      int   top;
     int   buffer[STACK_SIZE];
   public:
        Stack()  { top = -1; }
        bool push(int i);
        bool pop(int& i);
};
void main()
{   Stack st1,st2;
    int x;
    st1.push(12);
    st1.pop(x);

    st1.buffer[2] = -1;   X
}
```

```
bool  push(Stack * const this, int i)
{     if  ( this-> top == STACK_SIZE-1)
      {   cout << "Stack is overflow.\n";
          return false;  }
      else
      {  this-> top++;   this->buffer[this->top] = i;
          return true;  }
}
```

```
bool pop(Stack * const this,   int &i)
{    if  (  this-> top == -1)
     {    cout << "Stack is empty.\n";
           return   false;  }
   else
     {    i =  this-> buffer[  this->top];
        this->top--;
          return   true;  }
}
```

```
struct Stack
{   int   top;
    int   buffer[STACK_SIZE];
};
void main()
{   Stack   st1, st2;
    st1.top = -1;  st2.top = -1;
    int  x;      push(st1,12);       pop(st1,x);
}
```

Cfront

Encapsulation

Information Hidding

OOP

- Concepts
  - Program＝Object1 + Object2 +…… + Objectn
  - Object: Data + Operation
  - Message: function call
  - Class
- Classify
  - Object-Oriented
  - Object-Based   Ada
    - Without Inheritance

OOP

需求
架构
构建模式
代码
测试用例
项目组织

- Why
  - 评价标准
    - Efficency of Development
    - Quality
      - External

        Correctness、Efficiency、Robustness、Reliability

        Usability、Reusability
      - Internal

        Readability、Maintainability、Portability

产品在规定的条件下和规定的时间内完成规定功能的能力

ENCAPSULATION

类　成员变量
成员函数

类

成员变量
成员函数

inline

```
class  TDate
{  public:
       void SetDate(int y, int m, int d)
       {    year = y;  month = m; day = d; }
       int  IsLeapYear()
            { return (year%4 == 0 && year%100 != 0)
                                 || (year%400==0); }

  private:
       int  year, month, day;
};       int  year=2000, …
```

a.h

```
class   TDate
{   public:
         void SetDate(int y, int m, int d);
         int IsLeapYear();
    private:
        int  year, month, day;
};
```

ADT

a.cpp

```
void TDate::SetDate(int y,  int m,  int d)
{    year = y;
     month = m;
     day = d;
}
 int TDate:: IsLeapYear()
{ return (year%4 == 0 && year%100 != 0) || (year%400==0); }
```

Value

```
TDate g;

int  main()
{     g.SetDate(2000,1,1);

      TDate t;
      t.SetDate(2015,11,17);

      TDate  *p = new TDate;
      p->SetDate(2015,11,17);
}
```

构造函数

- 对象的初始化
- 描述
  - 与类同名、无返回类型
  - 自动调用，不可直接调用
  - 可重载

  - 默认构造函数 *无参数* Why？
    - 当类中未提供构造函数时，编译系统提供

  - *public*
    - 可定义为*private*
      接管对象创建

构造函数

- 调用
  - 自动调用

```
 class A
{   …
 public:
        A();
        A(int i);
        A(char *p);
};
A   a1=A(1);  ⇔ A a1(1);  ⇔ A a1=1;            //调A(int i)
A   a2=A(); ⇔ A a2;        //调A()，注意：不能写成：A a2();
A   a3=A(“abcd”); ⇔ A a3(“abcd”); ⇔ A a3=“abcd”;  //调A(char *)
A   a[4];            //调用a[0]、a[1]、a[2]、a[3]的A()
A   b[5]={ A(), A(1), A("abcd"), 2, "xyz" };
```

成员初始化表

- 成员初始化表
  - 构造函数的补充
  - 执行
    - 先于构造函数体
    - 按类数据成员申明次序

```
class CString
{     char   *p;
       int   size;
public:                              ?
   CString(int x):size(x),p(new char[size]){}
};
```

减轻Compiler负担

```
class A
{      int    x;
       const  int   y;
       int& z;
  public:
        A(): y(1),z(x), x(0)  {  x = 100; }
};
```

成员初始化表

```
class A
{       int m;
     public:
       A() { m = 0; }
       A(int m1) { m = m1; }
};
```

```
class B
{         int x;
          A a;
      public:
          B(){ x = 0; }
         B(int x1) { x = x1; }
        B(int x1, int m1): a(m1) { x = x1; }
};
```

```
void main()
{       B b1;        //调用B::B()和A::A()
       B b2(1);   //调用B::B(int)和A::A()
      B b3(1,2); //调用B::B(int,int)和A::A(int)
      …
}
```

成员初始化表

- 在构造函数中尽量使用成员初始化表取代赋值动作
  - const 成员/reference 成员/对象成员
  - 效率高
  - 数据成员太多时，不采用本条准则
    - 降低可维护性

析构函数

GC

效率障碍

*存在不能用**GC**的场合*

需要时，程序员自行实现

- 析构函数
  - ~<类名>()
  - 对象消亡时，系统自动调用

释放对象持有的非内存资源

  - *public*
    - 可定义为*private*

Java: finalize()

RAII vs GC

Resource Acquisition Is Initialization

Better Solution:

```
class A
{   public:
        A();
    void destroy() {delete this;}
    private:
        ~A();
};
```

✗***A a;***

```
int main()
{  ✗A aa;
};
```

A *p = new A;

✗***delete p;***

p->destroy();

```
static void free(A *p)
        { delete p; }

A::free(p);
```

强制自主控制对象存储分配

析构函数

```
class String
{            char *str;
  public:
             String()   {  str = NULL; }
             String(char *p)
             { str = new char[strlen(p)+1];
               strcpy(str,p); }

             ~String()  {  delete []str; }

             int length()  { return strlen(str); }
             char get_char(int i)  {  return str[i];
}
```

```
 void set_char(int i, char value)
 {  str[i] = value; }
char &char_at(int i)
{  return str[i]; }
char *get_str()  {   return str; }
char *strcpy(char *p)
{   delete []str;
     str = new char[strlen(p)+1];
   strcpy(str,p); return str;
}
String &strcpy(String &s)
{   delete []str;  str =
newchar[strlen(s.str)+1];
  strcpy(str,s.str);   return *this; }

char *strcat(char *p);
String &strcat(String &s); };
```

拷贝构造函数

- Copy Constructor
  - 创建对象时，用一同类的对象对其初始化
  - 自动调用

```
A  a;
A  b=a;
```

```
f(A a)
{   ....  }


A b;
f(b);
```

```
A  f()
{   A a;  ....
    return a;
}


f();
```

```
public:
    A(const A& a);
```

默认拷贝构造函数

➢逐个成员初始化(member-wise initialization)

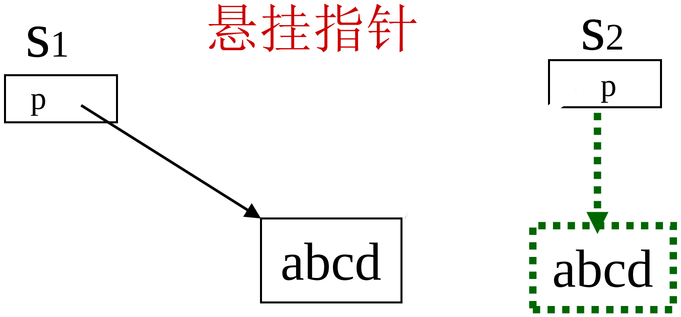
➢对于对象成员，该定义是递归的

何时需要copy constructor?

拷贝构造函数

```
class string
{   char *p;
public:
      string(char *str)
      {  p = new char[strlen(str)+1];
         strcpy(p, str);
      }
     ~string() { delete[] p; }
}

string  s1(“abcd”);
string  s2=s1;
```



```
string::string(const string& s)
{   p = new char[strlen(s.p)+1];
    strcpy(p,  s.p);
}
```

deep copy

拷贝构造函数

```
class A
{       int x,y;
    public:
       A() { x = y = 0; }
       void inc() { x++; y++; }
};
class B
{       int z;
       A a;
   public:
       B() { z = 0; }
       B(const B& b) : a(b.a) { z = b.z; }
       void inc() { z++; a.inc(); }
};
......
B b1;          //b1.z=b1.a.x=b1.a.y =0
b1.inc();      //b1.a.x=b1.a.y=b1.z=1
B b2(b1);     //b2.z=1，b2.a.x=0，b2.a.y=0
```

包含成员对象的类

- **默认拷贝构造函数**

调用成员对象的**拷贝构造函数**

- **自定义拷贝构造函数**

调用成员对象的**默认构造函数**

```
string generate()
{    ......
     return string("test");
}
string S=generate();
```

移动构造函数

move constructor

A(A&&)

移动构造函数

```
string generate()
{    ……
     return string("test");
}

string S=generate();
```

移动构造函数
move constructor
A(A&&)

```
int x=5;

int & y=x;
const int & z=5;
```

```
string::string (String &&s):p(s.p)
{s.p=nullptr; }
```

动态内存

- Types of memory from Operating System
  - Stack – local variables and pass-by-value parameters are allocated here
  - Heap – dynamic memory is allocated here
- C
  - malloc() – memory allocation
  - free() – free memory
- C++
  - new – create space for a new object (allocate)
  - delete – delete this object (free)

动态对象

- 动态对象
  - 在*heap* 中创建
  - *new* / *delete*

为什么要引入new、delete操作符？
constructor/destructor

动态对象

```
class A
{  …
        public:
           A();
           A(int);
 };
```

A *p,*q;

p = new A;

- 在程序的heap中申请一块大小为sizeof(A)的内存
- 调用A的默认构造函数对该空间上的对象初始化
- 返回创建的对象的地址并赋值给p

q = new A(1);

- ……
- 调用A的另一个构造函数 A::A(int)
- ……

delete p;

- 调用p所指向的对象的析构函数
- 释放对象空间

delete q;

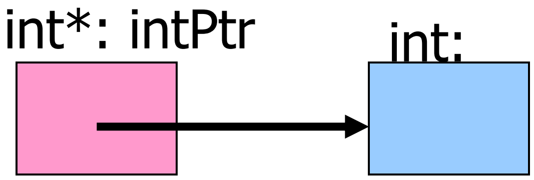
创建对象

- new
  - Works with primitives
  - Works with class-types
- Syntax:
  - `type* ptrName = new type;`
  - `type* ptrName = new type( params );`

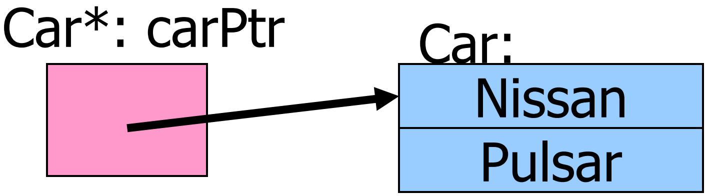
Constructor!

New Examples

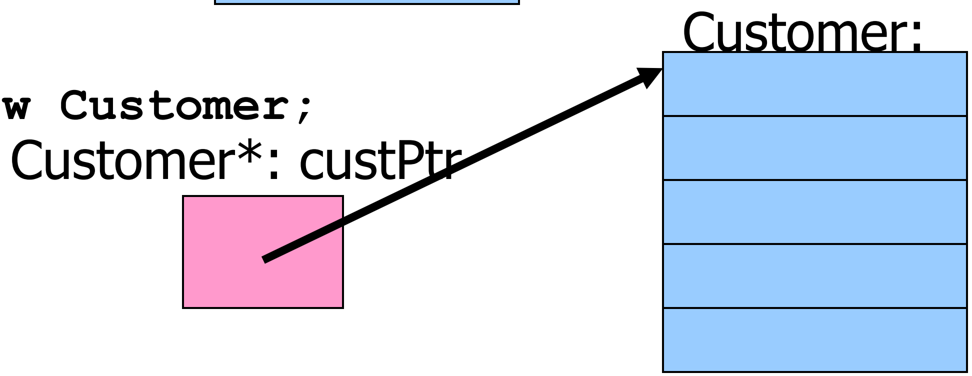
```
int* intPtr = new int;
```



```
Car* carPtr = new Car(“Nissan”, “Pulsar”);
```



```
Customer* custPtr = new Customer;
```



Notice:

These are unnamed objects! The only way we can get to them is through the pointer!

Pointers are the same size no matter how big the data is!

动态对象

- *p = (A *) malloc (sizeof(A))*
  free(p)

  malloc 不调用构造函数
  free 不调用析构函数

- *new* 可重载

对象删除

- delete
  - Called on the pointer to an object
  - Works with primitives & class-types
- Syntax:
  - delete ptrName;
- Example:
  - `delete intPtr;`
  - `intPtr = NULL;`

  - `delete carPtr;`
  - `carPtr = NULL;`

  - `delete custPtr;`
  - `custPtr = NULL;`

Set to NULL so that you can use it later – protect yourself from accidentally using that object!

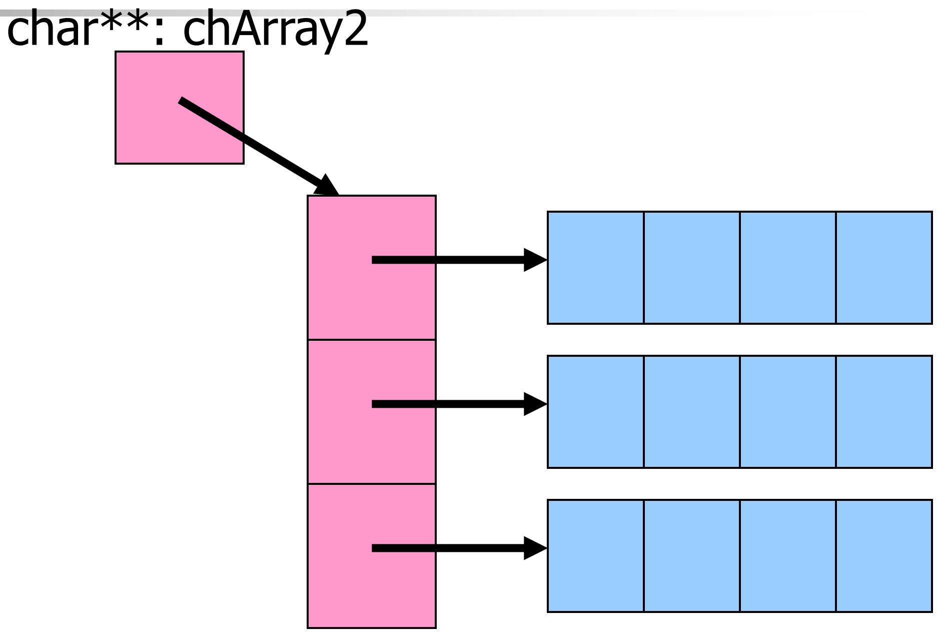
动态对象数组

- 动态对象数组的创建与撤消

```
A *p;
p = new A[100];
delete  []p;
```

- 注意
  - 不能显式初始化，相应的类必须有默认构造函数
  - delete中的[]不能省

动态2D数组



- Algorithm
  - Allocate the number of rows
  - For each row
    - Allocate the columns
- Example

```
const int ROWS = 3;
const int COLUMNS = 4;

char **chArray2;

// allocate the rows
chArray2 = new char* [ ROWS ];

// allocate the (pointer) elements for each row
for (int row = 0; row < ROWS; row++ )
   chArray2[ row ] = new char[ COLUMNS ];
```

动态2D数组

- Delete?
  - Reverse the creation algorithm
    - For each row
      - Delete the columns
    - Delete the rows
- Example

```
for (int row = 0; row < ROWS; row++)
{
   delete [ ] chArray2[ row ];
   chArray2[ row ] = NULL;
}

delete [ ] chArray2;
chArray2 = NULL;
```

Const 成员

- *const* 成员
  - *const* 成员变量

```
class A
{   const int x; }
```

- 初始化放在构造函数的成员初始化表中进行

```
class A
{         const  int x;
      public:
          A(int c): x(c) {  }
}
```

Const 成员

void f(A * const this);
void show(const A* const this);

- *const* 成员函数

```
class A
{    int x,y;
 public:
        A(int x1, int y1);
        void f();
        void show() const ;
};
```

mutable

```
const A a(0,0);

a.f(); X
a.show();   √
```

compiler

```
void A::f()
{  x = 1; y = 1; }

void A::show() const
{  cout <<x << y;}
```

```
class A
{
    int a;
    int & indirect_int;
public:
     A():indirect_int(*new int){ ... }
    ~A() { delete &indirect_int; }
    void f() const { indirect_int++; }
};
```

静态成员

- 静态成员
  - 类刻划了一组具有相同属性的对象
  - 对象是类的实例
  - 问题：同一个类的不同对象如何共享变量？
    - 如果把这些共享变量定义为全局变量，则缺乏数据保护
    - 名污染

静态成员

- 静态成员变量

```
class A
{    int   x,y;
     static int shared;
     .....
};
int A::shared=0;

A a, b;
```

  - 类对象所共享
  - 唯一拷贝
  - 遵循类访问控制

静态成员

- 静态成员函数

```
class A
{    static int shared;
      int x;
 public:
      static void f() { …shared…}
      void q() { …x…shared…}
};
```

  - 只能存取静态成员变量，调用静态成员函数
  - 遵循类访问控制

静态成员

- 静态成员的使用
  - 通过对象使用

    A a; a.f();
  - 通过类使用

    A::f();

- C++支持观点　“类也是对象”
  - Smalltalk

静态成员

```
class A
{       static int obj_count;
        …
  public:
        A()  {   obj_count++; }
        ~A()  {   obj_count--; }
        static int get_num_of_obj()；
         …
};


int  A::obj_count=0;
int  A::get_num_of_obj() { return obj_count; }
```

示例

singleton

```
class  singleton
{   protected:
            singleton(){}
            singleton(const singleton &);
    public:
           static singleton * instance()
           {   return  m_instance == NULL?
                                   m_instance = new singleton: m_instance;
           }
           static void destroy()  { delete m_instance; m_instance = NULL; }
    private:
           static singleton * m_instance;
};
singleton * singleton ::m_instance= NULL;
```

Resource Control

原则：谁创建，谁归还

友元

- 友元
  - 类外部不能访问该类的*private*成员
    - 通过该类的*public*方法
    - 会降低对*private*成员的访问效率，缺乏灵活性
    - 例：矩阵类(Matrix)、向量类(Vector)和全局函数(multiply)，全局函数实现矩阵和向量相乘

友元

```
class Matrix
{     int  *p_data;
      int   lin,col;
  public:
     Matrix(int l, int c)
     {   lin = l;
         col = c;
         p_data = new int[lin*col];
      }
     ~Matrix()
     { delete []p_data; }
```

```
int &element(int i, int j)
{  return *(p_data+i*col+j); }

 void dimension(int &l, int &c)
 {   l = lin;
     c = col;
  }

  void display()
 {   int *p=p_data;
     for (int i=0; i<lin; i++)
     {   for (int j=0; j<col; j++)
         {  cout << *p << ' ';
            p++;
         }
         cout << endl;
     }
};
```

友元

```
class Vector
{   int  *p_data;
     int   num;
  public:
     Vector(int n)
     { num = n;
       p_data = new int[num];
    }
    ~Vector()
    {  delete []p_data;
    }
```

```
int &element(int i)
{  return p_data[i]; }

 void dimension(int &n)
 { n = num; }

 void display()
 {  int *p=p_data;
     for (int i=0; i<num; i++,p++)
         cout << *p << ' ';
     cout << endl;
   }
};
```

友元

```
void   multiply(Matrix &m, Vector &v, Vector &r)
{   int  lin,  col;
     m.dimension(lin,col);
    for (int i=0; i<lin; i++)
   {   r.element(i) = 0;
       for (int j=0; j<col; j++)
           r.element(i) += m.element(i,j)*v.element(j);
    }
}

void main()
{    Matrix m(10,5);
    Vector v(5);
    Vector r(10);
    ......
    multiply(m,v,r);
    m.display();
   v.display();
    r.display();
}
```

友元

- 分类
  - 友元函数
  - 友元类
  - 友元类成员函数

- 作用
  - 提高程序设计灵活性
  - 数据保护和对数据的存取效率之间的一个折中方案

友元

```
void func() ;
class B;
class C
{  …….
     void f();
};
class A
{       …
        friend void func();             //友元函数
        friend class B;                 //友元类
        friend void C::f();             //友元类成员函数
};
```

友元

```
class Matrix
{    ......
     friend void multiply(Matrix &m, Vector &v, Vector &r);
};
class Vector
{    ......
    friend void multiply(Matrix &m, Vector &v, Vector &r);
};
```

- 友元不具有传递性
- 能编译吗？

原则

Law of Demeter

- 避免将data member放在公开接口中

	Get	Set
R	√	
W		√
RW	√	√
NONE		

```
class AccessLevels {
public:
   int getReadOnly const { return readOnly; }
   void setReadWrite(int value) { readWrite = value; }
   int getReadWrite() { return readWrite; }
   void setWriteOnly(int value) { writeOnly = value; }
private:
   int noAccess;
   int readOnly;
   int readWrite;
   int writeOnly;
};
```

- 努力让接口完满 (complete) 且最小化