C++语言基础

潘爱民

panaimin@icst.pku.edu.cn

http://www.icst.pku.edu.cn/compcourse

内容简介

- C++语言基础
- Object-Based Programming
- Object-Oriented Programming
- STL

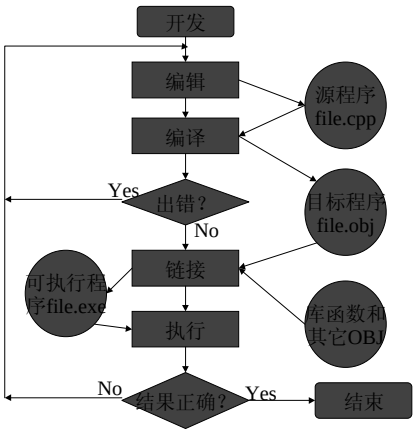
C++之我见

- C++语言在变化，我们的概念也要调整变化
- 最能反映OO思想的语言，掌握C++有助于理解OO
- 区分C和C++，C++不是C语言，不要用看待C语言的方式来看待C++

掌握C++语言的基础知识

- 操作系统的基础知识
  - 文件API、内存管理… infrastructures
- 计算机原理
  - 汇编语言
  - C/C++是产生ASM代码的framework
- 编译和执行过程
  - 编译过程
  - 执行过程
- 许多概念
  - 进程、heap、stack，…

程序开发过程



Build过程：compile time

- Compile + link
- 如何对待错误和警告
  - 错误：一定要排除，从第一个错误找起
  - 警告：或者排除，或者确实理解不会对程序造成危害
- 学会设置compile options和link options
  - IDE中如何设置？ —— 一定要知道！
  - 命令行如何设置？
  - 程序代码中如何设置？

执行过程：runtime

- Load、重定位、初始化、main
- 如何调试
  - 断点
  - assertion
- 程序对环境的依赖性
  - 系统环境
  - CRT
  - 内存影像

C语言例子

```
/* STRCPY.C: This program uses
   strcpy
 * and strcat to build a phrase.
 */

#include <string.h>
#include <stdio.h>

void main( void )
{
   char string[80];
   strcpy( string, "Hello world from " );
   strcat( string, "strcpy " );
   strcat( string, "and " );
   strcat( string, "strcat!" );
   printf( "String = %s\n", string );
}
```

C语言特性

- 结构化
  - 简单的方法可以构造复杂的结构
- 表达式灵活、简练
  - 提高代码质量、程序的可读性
- 提供了一些接近汇编语言的功能及指针
  - 适合于编写系统软件和工具软件；指针非常灵活
- 调试方便
- 移植性好
- 目标代码小、效率高

C++语言内容

- 数据类型
  - 内置(built-in)、标准库、自定义的
- 表达式
- 控制语句
- 函数(模板)
- 异常处理

数据类型

- 常量
  - 10、012、0x12、1L、3.14、1.0E-3、true、'a'、"file name is \"myproc.c\""
- 变量
  - 名字：不能用关键字，往往小写，有意义
  - 风格：下划线(first_filename)、首字母大写(FirstFilename)
  - 定义与声明
- 基本数据类型
  - void、char 、 short 、 int 、 long 、float 、double 、signed 、unsigned

数据类型(续一)

- 指针
  - 指向某个变量的地址
  - 注意：未经初始化的指针不能使用
- 字符串
  - C风格的字符串：指向char的指针
  - 使用string、wstring
- const修饰符
  - 可以修饰变量、指针、函数
- 引用类型
  - 语义：别名

数据类型(续二)

- 结构struct
- 联合union
- 枚举enum
- bool类型
- 数组
  - 一维数组、多维数组
  - 数组和指针之间的关系
  - vector
- typedef
- volatile修饰符

常量

- const定义常量

```
const int size = 100;  // 在size的有效范围内不可变
char *const aptr = mybuf;  // 常量指针
        *aptr = 'a';          // 合法
         aptr = yourbuf;    // 非法
const char *bptr = mybuf;  // 指向常量数据
         *bptr = 'a';          // 非法
         bptr = yourbuf;    // 合法
const char *const p = "abcde";
    char *p = "abcd";          // 合法，—— 例外
```

const也可以用在函数的参数说明中

- const定义成员函数

在类的定义中有

```
int GetData() const;   // 不会改变对象的数据
```

数组

- 一维数组

```
int number[100];
extern int number[];
```

- 多维数组

```
int number[100][10];
extern int number[][10];
```

- 结构数组

```
struct {
    float x, y;
} complex[100];
```

结构struct

- 简单结构

```
struct date
{
    int year, month, day;
    char mon[3];
};
```

- 匿名结构和嵌套结构

```
struct person{
    char name[30];
    char sex;
    int age;
    int weight;
    struct {
     int areacode;
     long number;
    };
} Jim;
Jim.number = 1234567;
```

结构struct(续)

- 结构中的位域

```
struct mybitfields
{
    unsigned a : 4;
    unsigned b : 5;
    unsigned c : 7;
} test;
```

- 访问结构中的成员

.操作符

```
Jim.number = 1234567;
```

联合union

- 联合类型

```
union u_tag
{
    int ival;
    float fval;
    char *pval;
};
```

MFC中典型的例子是VARIANT的定义

- 联合与结构的区别

内存分配、引用方式

枚举类型

❖ 枚举类型

```
enum Days            // 声明Days为枚举类型
{
  saturday,          // 缺省saturday = 0
  sunday = 0,        // 指定sunday = 0
  monday,            // monday = 1
  tuesday,           // tuesday = 2
  wednesday,         // 等
  thursday,
  friday
} today;             // 定义变量today为类型Days
```

❖ 使用例子

```
enum Days yesterday;    // C和C++中都合法
Days tomorrow;          // C++合法，C非法
yesterday = monday;
int i = tuesday;        // 合法; i = 2
yesterday = 0;          // 非法，不允许
yesterday = (Days)0;    // 合法，但结果不确定
```

typedef

❖ 例子

```
typedef char FlagType;
const FlagType x;

typedef int (FAR WINAPI *FARPROC)();
typedef int (WINAPI *PROC)();

PROC proc
.....
int retVal = (*proc)();
```

自定义变量的说明

❖ enum、struct、union

说明变量时不需要使用这些关键字

例如：

// 已经定义了EnumDay、StructPoint、UnionColor

说明：

```
enum EnumDay day;        --------- EnumDay day;
struct StructPoint point;    --------- StructPoint point;
union UnionColor color;    --------- UnionColor color;
```

❖ 无名联合、结构

例如：

```
struct fruit {
    int     type;
    char  *name;
    union {
        DWORD color;
        struct {
            BYTE  red;
            BYTE  green;
            BYTE  blue;
        };
    };
};
```

指针

❖ 指针是一个变量，它包含另一个变量的地址，因此，通过指针可以间接地访问另一个变量

```
int x, *px;      // 声明指针
px = &x;
```

❖ 运算符&和*

&x表示取x的地址

*px表示取px所指的变量

❖ 指针与整数的运算(p为指针)

p++ 表示紧接*p之后的变量的地址

p-- 表示紧靠*p之前的变量的地址

p + n 表示紧接*p之后的第n个变量的地址

p - n 表示紧靠*p之前的第n个变量的地址

p1 - p2 指向同一数组区的两指针之间的变量个数

*sizeof算符

❖ 指针嵌套

int **ppx 表示指向int *的指针

指针(续一)

❖ 指针与结构

```
struct date *pd;
..... 注意对pd一定要赋值
pd->day
pd->month
```

相当于 (*pd).year 说明：括号是必须的，因为. 优先级高

❖ 运算符&和*

&x表示取x的地址

*px表示取px所指的变量

❖ .*和->*

object.*pMember

pobject->*pMember

❖ 指针类型的转换

指针的转换有很大的随意性，但一定要保证安全性

指针(续二)：.*和->*

```
Class A{
public:
int m_n;
char m_ch;
};

int A::*pMember = &(A::m_n);
A object;
A* pobject = &object;
object.*pMember = 1;//实际上是object.m_n = 1;
pobject->*pMember = 2;//实际上是pobject->m_n = 'c';
```

指针(续三)

- 指针例子

1. int x, a[10], *pa;
   pa = &a[0];
   x = *pa;
2. 若有以下说明
   struct { int x, *y } *p;
   则
   ++p->x; // x加一
   p->x++; // x加一(后加)
   (++p)->x; // p先加一，再访问x
   *p->y++; // 先访问y的内容，再加一
   *(p->y)++; // 同上
   *p++->y; // 先访问y的内容，后p加一
   *(p++)->y; // 先访问y的内容，后p加一
3. 函数指针
   定义：void (PASCAL * pFunc)(int, int);
   赋值：pFunc = GetProcAddress(hLib,"Func");
   调用：(*pFunc)(100, 100);

引用——reference

- 引用
  - 其他变量的别名
  - 一定要初始化
- 引用与指针的区别
  - 指针可能未被初始化
- 引用例子
  - int &r = i;

指针变量和引用

- 指针变量

```
#include <stdio.h>
void swap(int *a, int *b)
{
    int tmp = *b;
    *b = *a;
    *a = tmp;
    return;
}
int main()
{
    int i = 7, j = 3;
    swap(&i, &j);
    printf("The Result is :i =%d, "
           " j = \n", i, j);
    return 0;
}
```

- 引用

```
#include <stdio.h>
void swap(int &a, int &b)
{
    int tmp = b;
    b = a;
    a = tmp;
    return;
}
int main()
{
    int i = 7, j = 3;
    swap(i, j);
    printf("The Result is :i =%d, "
           " j = \n", i, j);
    return 0;
}
```

变量类型

- 自动变量和外部变量

自动变量：函数中说明的变量
例如，在某个函数内部：auto int i;
特点：定义在函数中，位于栈中，有效期在函数中
外部变量：所有函数外部定义的变量
例如，在函数中说明：extern int i;
特点：定义在函数外，不在栈中，全程有效

- 静态变量和寄存器变量

静态变量：内部静态变量和外部静态变量
例如：static int i;
特点：变量的连续性，有效期在函数中
分析：有效期和生存期
寄存器变量：
例如：register int i;
register char x;
特点： 只能用于自动变量、个数限制、类型限制(int、char,指针)

变量类型例子

```
#include <math.h>

static int i; /* 外部静态变量 */
extern int Total;
          /* 外部变量，定义在其它文件中*/
int Add(int x, int y)
{
    int z;  /* 自动变量 */
    extern int Total; /* 可以省略 */
    z = x + y;
    Total ++;
    return z;
}

int Sub(int x, int y)
{
    int z;  /* 自动变量 */
    static int SubTime = 0;
    z = x - y;
    SubTime ++;
    return z;
}
```

```
#include <stdio.h>

int Total = 0;
          /* 外部变量，定义 */
int  main()
{
    register int i1, i2;  /* 自动变量 */
    int Result1, Result2;
    i1 = 10;
    i2 = 20;
    Result1 = Add(i1,  i2);
    Result2 = Sub(i1,  i2);

    printf("i1 = %d , i2 = %d\n", i1, i2);
    printf("Result1 = %d , Result2 = %d\n",
           Result1, Result2);
    printf("Total = %d\n",Total);
    return 0;
}
```

new和delete

- 分配内存

在C中：
char *name = (char *)malloc(Length + 1);
使用new：
char *name = new char[Length + 1];

- 释放内存

在C中：
free(name);
使用delete：
delete [] name;

表达式

- 算术运算符
  * / % + - ++ --
- 关系运算符
  ! < <= > >= == != && ||
- 赋值运算符
  =
- 条件运算符
  ?:
- sizeof运算符
  sizeof

表达式(续)

- New、delete
- 按位运算、求反运算
  & | ^ 、 ~(求反)
- 移位运算
  >> <<
- 逗号
- 类型转换
  隐式转换、显式转换
- 复合表达式

控制语句

- if、return、表达式、goto
- for、continue、break、switch
- while、do-while
- try、复合语句
- 逗号既可作分隔符，也可作运算符
- 空操作语句

控制语句例子

- 条件语句
  if (条件表达式) 语句
  if (条件表达式) 语句A
  else 语句B
- 复合语句
  把一组语句用{ }括起来
  if (条件表达式) {
  语句a;
  语句b;
  .....
  }
- 简化条件表达式
  表达式1 ? 表达式2 : 表达式3

- 开关语句

```
switch (表达式)   {
    case 表达式1:
        语句1;
        break;
    case 表达式2:
        语句2;
        break;
    .....
    default:
        语句
        break;
}
```

控制语句例子(续)

- while语句
  - while (表达式) 语句
- for语句
  for(表达式1; 表达式2; 表达式3) 语句
  赋初值 结束条件 控制变量行为
- do-while语句
  do {
  语句;
  } while (表达式);
- break, continue语句
- 逗号用于表达式、空操作语句

函数

- 函数声明、函数原型
  - 函数返回类型、函数参数表
  - 编译时刻，类型检查
  - 例如： bool Func(int x, int y);
- 函数定义(函数体)
- 参数传递
  - pass-by-value
  - pass-by-reference
  - 缺省参数：从右往左
    - 说明：void func(int a, int b, int c=10);
    - 定义：调用：func(5,6); 相当于func(5,6,10);

函数(续一)

- 返回值
  - 也可以返回引用
- 引出函数(MS)
- inline函数
  - 函数说明加前缀inline，例如：

```
inline void func(int a, int b)
{
printf("%\n a = %d, b = %d\n", a, b);
}
```

- extern "C"

函数(续二)

- 命令行参数：int argc, char *argv[]
- extern static
- 调用参数处理：_stdcall _cdecl
- 参数不确定
  - 例如：int printf(char *, ...);
- 函数指针：
  - 定义函数指针 (*fp)
  - 函数名字取地址仍然是函数指针
  - 调用：pf();或者(*pf)();

函数(续三)

- C++标准库
  - 比如：#include <iostream>
  - using namespace std;
- C函数库
  - stdio，stdlib.h，math等
  - #include <cstdio>
- 函数重载
  - 例如：

```
void print(int i);
void print(char *s);
void print(int *p);
```

函数例子

```
/* Function definition */
void work( int number,
           long (*function)(int i) )
{
   int i;    long j;
  for ( i=j = 0; i < number; i++ )
       j += ( *function )( i );
}
```

```
void main()
{
   /* Function prototypes */
   long lift( int ), step( int ), drop( int );
   void work( int number, long
    (*function)(int i));
   int select, count;
   .   .   .
   select = 1;
   switch( select )    {
      case 1: work( count, lift );
              break;
      case 2: work( count, step );
              break;
      case 3: work( count, drop );
      default:
               break;
   }
}
```

说明(声明)

- 说明

1. 变量说明

```
extern int global;
extern int global = 10;   // 定义!
```

2. 函数说明

```
void Func(int x, int y);
*隐含说明
*通常#include文件中包含的是函数的说明
```

3. 结构、联合、枚举类型说明(类class)

```
struct POINT
 {
     int x;
     int y;
 };
```

定义 (definition)

- 定义

1. 变量定义

```
int global;
```

2. 函数定义

```
void Func(int x, int y) {
    .....
    return;
}
```

3. 结构、联合、枚举类型定义(类class)

```
struct POINT
 {
     int x;
     int y;
 };
struct POINT point;
```

注意：C++允许变量在函数的任何地方定义或说明，习惯上，尽量与第一个使用点越近越好

作用域 (scope)

- local scope
  {
  }
- namespace scope
  namespace XXX {
  …
  }
- class scope
- scope operator (::)
- using
- using namespace std;

函数重载

- 相同的函数名，不同的参数，编译器必须能够区别
- 参数表相同，返回值不同，不是重载
- 不能包含两个extern “C” 重载版本
- 重载函数指针

函数模板

- 例子：

```
template <class Type>
Type min(Type a, Type b)
{
  return a<b?a:b;
}

min(10,20);
min(10.0, 20.0);
```

- 模板实例化 ——由参数决定
  - 隐式实例化
  - 显式实例化
  - min<int>(10,20);

函数模板(续)

- 两种编译模式：
  - inclusion模式
  - separation模式(export关键字)
- 模板特化
- 重载函数模板

异常处理(exception handling)

- 错误处理机制、错误代码隔离
- Throw(raise) an exception
  - Throw expression
  - 类似于return 语句
- Try、catch语句(handler)
  - try{}
  - catch(*type*){} 或 catch(*type obj*){}
- 类似于函数调用
  - 但是类型匹配发生在runtime

异常处理(续)

- 嵌套机制，如果最外层没有处理，则由terminate来处理
- Local object正常析构
- rethrow
  - 在catch子句中仍然可以throw
- Catch all handler
  - catch(…)
- 声明函数时指定函数可能的异常
  - int func(int) throw(string);
  - 如果没有指定的异常发生，调用unexpected()

预处理及其他

- 格式
  换行起分隔符的作用，
  续行用“\”
- 注释
  用/*开始，用*/结束
  在C++中，也可以用//
- 宏定义
  符号常数定义
  格式：#define 符号名 常数/字符串
  例如：#define TRUE 1
  带参数的宏

```
#define IS_LEAP_YEAR(y)     y%4==0&&y%100!=0 \
                                ||y%400==0
```

注意：宏只是代换，所以保持变量整体性，如想象IS_LEAP_YEAR(x+1)

```
#define IS_LEAP_YEAR(y)     (y)%4==0&&(y)%100!=0||(y)%400==0
```

预处理及其他(续)

- 条件编译
  用法：

```
#ifdef 标识符    或者    #ifndef 标识符    或者    #if  条件表达式
    语句.....
#else                              (可选)
    语句.....
#endif
```

  例如：

```
#ifdef DEBUG
    针对调试的语句.....
#endif
```

  #undef语句
- 编译指示 #pragma
- 字符串转义
  \t \b \n \r \' \" \a \\ \ooo \xhhh 等

Object-Based Programming

- class实现数据封装
- 对象
- 构造函数与析构函数
- 访问控制、友元
- 嵌套类
- 类中成员初始化
- 虚拟函数
- 类中运算符重载
- 类模板

类的定义

```
class 类名标识符
{
  private :
    成员数据和成员函数说明      // 在类的外部不能直接访问
  protected :
    成员数据和成员函数说明      // 可以在派生类中被访问
  public :
    成员数据和成员函数说明      // 可以在类外被访问
};
```

类的例子

```
class CShape
{
protected:
   COLORREF color;

public:
   CShape() { color = 0;}
   virtual ~CShape() {}

   virtual void Display() = 0;
   virtual void MoveTo(int x, int y) = 0;
};
```

构造函数和析构函数

- 构造函数

构造函数是与类名同名的操作，类可以有多个构造函数
当程序中构造类的对象(实例)时，根据相应的参数，编译器会调用适当的构造函数进行对象初始化工作。
构造函数没有返回值

- 析构函数

析构函数的名字为~加类名
当对象被使用完之后，应该通过delete语句来释放相应的内存，这时析构函数被调用，完成一些结束处理工作。如果对象为自动变量，在栈中建立的，则当变量的生命周期结束时，编译器会自动调用析构函数。

构造函数

- 缺省构造函数
- Copy构造函数

构造函数与析构函数示例

类的定义：

```
class CPoint : public CShape
{
private :
    POINT m_point;
public:
    CPoint();
    CPoint(const CPoint &);
    CPoint(int x, int y);
    ~CPoint();

    void  SetPoint(POINT );
    void Display();
    void MoveTo(int x, int y);
};
```

使用1：

```
CPoint  *point1 = new CPoint;
……
delete point1;
```

使用2：

```
CPoint  *point2 = new CPoint(100,
    100);
……
delete point2;
```

使用3：

```
{  ……
   CPoint  point1;
   CPoint  point2(100,100);
    ……
}  // 自动调用析构函数
```

类的实现

```
CPoint::CPoint(int x, int y)
{
    m_point.x = x; m_ point.y = y;
}
CPoint::~CPoint()
{
}
void CPoint::SetPoint(POINT point)
{
    m_point = point
}
void CPoint::Display()
{
   .......
}
void CPoint::MoveTo(int x, int y)
{
    m_point.x = x; m_ point.y = y;
}
```

类的静态成员

- 所有类的对象共享同一个静态成员，例如：

```
class CPoint {
public:
  int x, y;
  static int nCount;
  …
  static int GetCount();
};
```

  - **定义：**

```
int CPoint::nCount = 0;
int CPoint::GetCount() { return nCount; }
```

  - **通过类名访问：**

```
CPoint::nCount
```

指向类成员的指针

- 例如：

```
class CShape {
public:
   int x, y;
   int width, height;
  …
   int GetHeight();
};

int (*pf)();                    // error!
int (CShape::*pf)();            // ok!
pf  = CShape::GetHeight;
(myShape.*pf)();
```

运算符重载

- 运算符重载增强了C++语言的可扩充性
  - 例如复数可以重载加、减、乘、除等运算
- C++中可以重载的运算符

  + - * / % ^ & | ~ ! , = < >
  += -= *= /= %= ^= &= |= <<
  >> >>= <<= == != <= >= &&
  || ++ -- -> ->* [ ] ( ) new
  delete new[] delete[]

- 注意每个运算符的语义
  - 比如 operator= 返回引用类型: return *this;

运算符重载示例

- 双目运算符

说明：

```
CComplex  operator+(CComplex complex) const;
```

定义：

```
CComplex CComplex::operator+(CComplex complex) const
{
    return CComplex(x + complex.x, y + complex.y);
}
```

使用：

```
Complex z1, z2, z3;
……  // 初始化 z1和z2
z3 = z1 + z2;
```

- 单目运算符

类型转换运算符重载

- Operator type()

```
class CPoint{
    operator char *();
};
```

使用：

隐式转换：用在函数参数

显式转换：

```
char *point = static_cast<char *>(myPoint)
```

类指针

- 指针p，指向一个对象

例如：

```
CComplex  *p;
```

赋值：

```
1. p = new CComplex(1, 1);
2.
   CComplex  z1(10, 10);
   p = &z1;
```

使用：

```
p->x        p->y
z1.x        z1.y
p->InitValue();
z1.InitValue();
```

this指针

- 在类的内部指向类自身的指针
- 在每个成员函数中，this指针提供了使用上的方便
- this指针把类的多个实例对象区分开来

类成员的访问特性

- 三种特性
  - private　　在类的外部不能直接访问
  - protected　　可以在派生类中被访问
  - public　　可以在类外被访问
- 友元
  - 友元类
  - 友元函数

嵌套类

- 说明

```
class BufferIO {
  public :
    enum  IOError { none, Access,
                    General };
    class BufferedInput
    {
     public:
        int read();
        int good();
     private:
        IOError  inputerror;
    };
    class BufferedOutput
    {
       ……
    };
```

- 实现

```
int  BufferIO::BufferedInput::read()
{
   ……
}
```

另：

```
typedef BufferedIO::BufferedInput
 BIO_INPUT;
int BIO_INPUT::read()
{
   ……
}
```

友元

```
// 友元函数
class Complex
{
  public:
     Complex( float re, float im );
  friend Complex operator+(
    Complex first, Complex second );
  private:
     float real, imag;
};

Complex operator+( Complex first,
                   Complex second )
{
  return Complex(
          first.real + second.real,
          first.imag + second.imag );
}
```

```
// 友元类
class MyClass
{
    friend class MyOtherClass;
        // Declare a friend class
    private:
        int topSecret;
};

class MyOtherClass
{
    public:
        void change( MyClass myc );
};

void MyOtherClass::change(
            MyClass myc )
{
    myc.topSecret++;
        // 可以访问MyClass的私有数据
}
```

模板

- 函数模板
  即以数据类型为“参数”的函数

例如：

```
template < class T > Swap(
            T &a, T&b)
{
    T temp;
    temp = a;
    a = b;
    b = temp;
}
```

- 类模板(模板类或类生成器)
  即以数据类型为“参数”的类
  例如：

```
template < class T > class Stack
{
    private :
        T pool[maxNum];
        int sp;
    public:
        Stack();
        void Push(T x);
        T Pop();
        BOOL IsEmpty() const;
        BOOL IsFull() const;
};
```

使用：

```
Stack < int > intStack;
Stack < float > float Stack;
```

模板(续)

- 定义模板
- 实例化
- 模板参数：类型参数、非类型参数
- 两种编译模式：
  - inclusion模式
  - separation模式(export关键字)
- 类模板特化(class template specializations)
- 类模板部分特化(class template partial specializations)

从面向过程转换到面向对象

- 面向过程以功能为组织单元

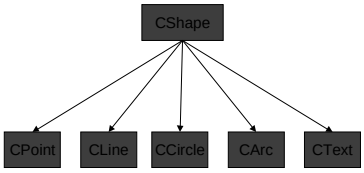
  在C语言中以函数作为功能单元；
  通过数据结构来描述具体的问题；
  数据在功能(函数)之间以参数的形式被传送。

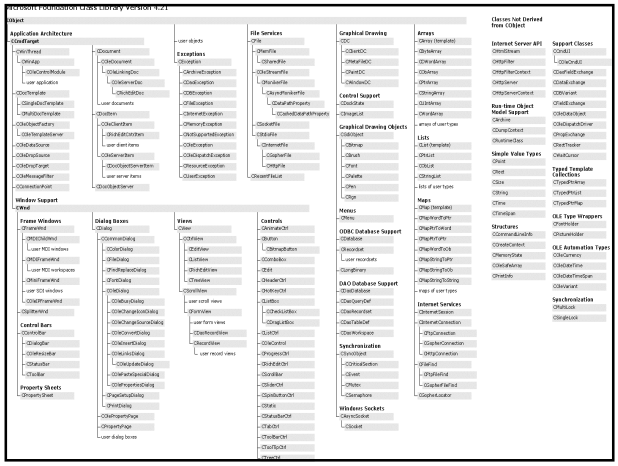
- 面向对象

  用类class封装数据以及与数据相关的操作；
  用类的继承性来实现类的重用性；
  多态性。

Object-Oriented Programming

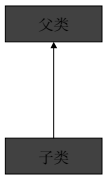
- CShape
- CPoint
- CLine
- CCircle
- CArc
- CText





类的继承，派生类

- 派生类特点
  - 继承父类的属性和操作
  - 修改父类的某些操作
  - 增加新的属性和操作
  - 继承实现了C++语言的可重用性
  - 派生类不能访问父类的私有属性和操作
  - 若一个类有多个父亲，则为多重继承



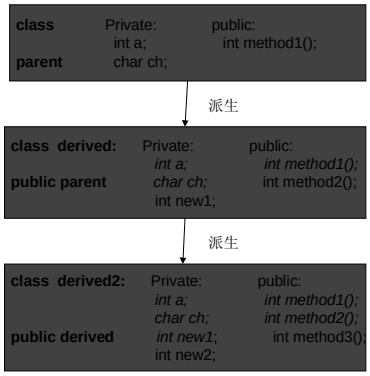
派生类

- 说明

```
class derived_class_name : private
    parent_class_name
{
…
};

class derived_class_name : public
    parent_class_name
{
…
};
```

派生类(续一)



派生类(续二)

- 派生类的构造函数

```
// 利用父类的构造函数进行初始化
derived::derived(int Init_a, char Init_ch, int
    Init_new1)
        : parent(Init_a, Init_ch)
{
   new1 = Init_new1;
}

derived2::derived2(int Init_a, char Init_ch, int
    Init_new1, int Init_new2)
        : derived(Init_a, Init_ch, Init_new1)
{
   new2 = Init_new2;
}
```

- 先执行父类构造函数，再执行派生类的构造函数，顺序：
  - 1 基类
  - 2 内嵌对象
  - 派生类

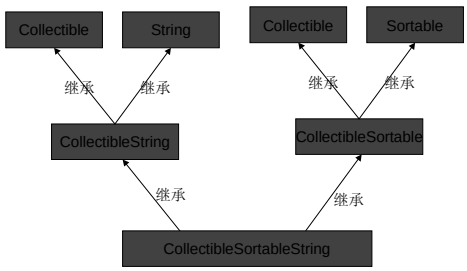
派生类(续三)

- 虚析构函数
  - 如果一个类要成为类层次中的基类，那么析构函数一定要定义成virtual
  - 例如：

```
CShape *obj = new CRect(…);
…
delete obj;
```

  - 如果不定义成virtual，则memory leak

派生类(续四)——多重继承



派生类(续五)

- 在多重继承结构中，相同的基类可能在树结构中出现多次
  - 在内存中有多份实例
  - 如果不希望出现这种情况，必须使用虚拟继承
- 在多重继承情况下，基类中可能包含相同的函数名，如果编译器不能区分该调用特定的函数，则必须用双冒号分辨符(::)显式指定类名

```
MyDerivedClass  *p = new MyDerivedClass;
p->MyBaseClassA::MemFunc();
p->MyBaseClassB::MemFunc();
```

多态性和虚拟函数

- 多态性是面向对象的一个支柱

  允许程序在实施对象的操作时，允许对象按不同的方式完成不同类型对象的操作；不同类型对象有自己实现操作的方法。

- 虚拟函数

  虚拟函数实现了多态性。

  说明：在函数说明之前加上关键字virtual

  在基类的说明中，定义虚拟函数：

```
virtual void Display();
```

  调用：

```
CShape  *pShape = GetCurrentShape();
pShape->Display();
```

纯虚函数

- 纯虚函数

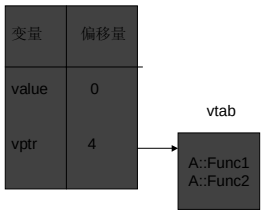
  定义：virtual void Display() = 0;

- 特点
  - 不能生成实例
  - 在派生类中必须提供相应虚函数的实现

对象的内存分布图

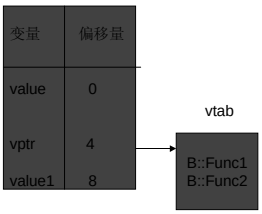
- Vptr指针和vtab表

```
class A {
   private :
      int value;
   public:
      virtual void Func1(void)
      virtual void Func2(void)
};
```



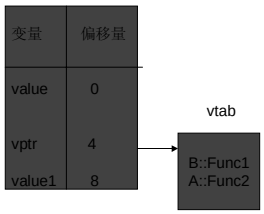
对象的内存分布图(续一)

```
class B : pulic A {
   private :
      int value1;
   public:
      virtual void Func1(void)
      virtual void Func2(void)
};
```



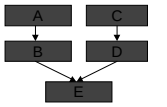
对象的内存分布图(续二)

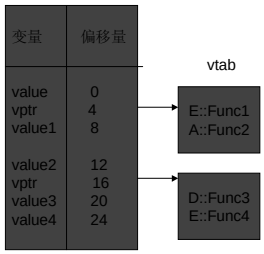
```
class B : pulic A {
   private :
      int value1;
   public:
      virtual void Func1(void)
};
```



对象的内存分布图(续三)

```
class C {
  private :
    int value2;
  public:
    virtual void Func3(void)
    virtual void Func4(void)
};
class D : public C {
  private :
    int value3;
  public:
    virtual void Func3(void)
};
class E : public B, public D {
  private :
    int value4;
  public:
    virtual void Func1(void)
    virtual void Func4(void)
};
```





继承关系

- Public继承
  - “is-a”关系
- private继承
  - “has-a”关系
  - 可以用复合类或者嵌套类来表示
- protected继承
  - 比private继承放宽限制
- virtual继承
  - 单个共享实例
  - 虚基类的构造由the most derived class完成

类型转换

- Static_cast
- Dynamic_cast
  - 向下转换

```
void Draw(CShape *shape)
{
   //shape->DrawRect()
   //shape->DrawCircle()
   if (dynamic_cast<CRect *>(shape)) {…}
}
```

- RTTI：typeid

Static_cast类型转换

```
class B { ... };

class D : public B { ... };

void f(B* pb, D* pd)
{
  D* pd2 = static_cast<D*>(pb);       // not safe, pb may
                                      // point to just B

  B* pb2 = static_cast<B*>(pd);       // safe conversion
  ...
}
```

Static_cast类型转换(续)

```
class B { ... };
class D : public B { ... };

void f(B* pb)
{
  D* pd1 = dynamic_cast<D*>(pb);
  D* pd2 = static_cast<D*>(pb);
}
```

C++标准库

- C标准库
- iostream library
  - cin、cout、cerr
  - ostream& operator <<(ostream& os, const MyClass)
  - istream& operator >>(istream& os, MyClass&)
- STL(Standard Template Library)

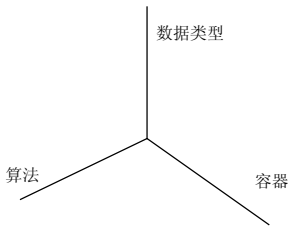
Generic Programming

- 作者：Alexander Stepanov
- 目标：高效、灵活地实现各种算法
- 思想：对算法进行抽象，与数据表示分开
- 以模板技术为基础
  - 函数模板
  - 类模板
- C++的发展使得这种思想得以实现

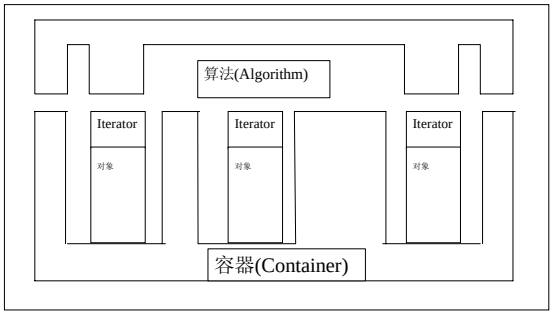
STL基础

- Object-Based Programming
  - 有关class的各种设计技术
- 函数对象(function objects)
  - 重载了函数调用操作符()的类
- 模板
  - 函数模板
  - 类模板
  - 模板特化

STL思想



STL实现

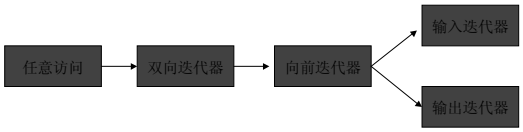


STL容器

- 顺序容器(Sequence Container)
  - vector
  - deque
  - list
- 关联容器(Associative Container)
  - set
  - multiset
  - map
  - multiset

STL迭代器

- 迭代器(Iterator)是指针(pointer)的泛化



STL算法

- 改变顺序的操作
  - reverse、replace
- 不改变顺序的操作
  - for_each、find
- 排序及相关操作
  - sort、rotate
- 常用的数字操作
  - count、sum_up

使用STL

```
#include <functional>
#include <list>
#include <iostream>
#include <algorithm>
using namespace std;
template <typename T>
void print_elements(T elem)
{ cout << elem << " "; }
void (*pfi)(int) = print_elements;
```

```
void main()
{
    int ia[7] = {0,1,2,3,4,5,6};
    list<int> ilist(ia, ia+7);
          // 以陣列做為 list 的初值
    for_each(ilist.begin(), ilist.end(), pfi);
           // 0 1 2 3 4 5 6
    ilist.push_back(7);
    ilist.push_back(0);
     ilist.push_back(7);
    ilist.push_back(9);
    for_each(ilist.begin(), ilist.end(), pfi);
          // 0 1 2 3 4 5 6 7 0 7 9
    ilist.remove_if(bind2nd(modulus<int>(), 2));
          // 去除所有奇數
    for_each(ilist.begin(), ilist.end(), pfi);
          // 0 2 4 6 0
}
```

C++参考书

“Design and Evolution of C++”
“C++ Programming Language”
“C++ Primer”(3/e)
“Inside the C++ Object Model”
“Effective C++”(2/e)
“More Effective C++”
“Exceptional C++”
“C++ Strategies and Tactics”
“Generic Programming and the STL”