

C++ 程序设计

王洪波

网络技术研究院

网络与交换技术国家重点实验室

宽带网研究中心

课程目标

- 本课程是一门以**实践**为目的的程序设计语言教学课程。
- 通过本课程的学习实践，学生应能够掌握利用C++语言进行面向对象程序设计的一般方法，并具有使用C++语言进行程序设计和解决实际问题的能力

教学方式

- 教学地点：教学实验综合楼N110
- 课堂教学(9月4、6、8、12日下午)
 - ❑ C++语言的语义、语法
 - ❑ 使用C++语言进行程序设计的基本技巧
 - ❑ 解决编程实践中存在的问题
- 编程实践
 - ❑ 提高同学的实践能力
 - ❑ 练习与编程作业

主要内容

➤ 基本概念

➤ C++语言基础

➤ 面向对象的程序设计初步与C++的类

➤ C++的高级特性

➤ C++程序设计综合训练

基础

➤ 先修课程

- ❑ 高级语言程序设计：C/Pascal
- ❑ 其它（非必须）：汇编语言、编译原理、...

➤ 参考书目

❑ 入门级别：

- ☒ C++语言程序设计（第3版），郑莉等编著，清华大学出版社
- ☒ Essential C++中文版，Stanley B.Lippman著，华中科技大学出版社

❑ 大全类：

- ☒ C++程序设计语言，Bjarne Stroustrup著，机械工业出版社
- ☒ C++ Primer中文版，Stanley B.Lippman等著，人民邮电出版社

❑ 高级进阶类：

- ☒ Effective C++, More Effective C++, Thinking in C++, ...

考核方式

➤ 主要考核同学应用编程实践的完成情况和能力

☐ 必须完成的内容

☒ 课后上机作业

☒ 综合编程练习

☐ 每个同学必须**独立**完成

➤ 课件资料、实验安排及通知等：

☐ 百度网盘：<https://pan.baidu.com/s/1pLyV36r>

☐ 口令：4fgu

➤ 答疑、提交作业方式：

☐ 邮箱：bupthomework@163.com

程序设计的基础

程序设计语言的发展

➤ 硬件语言

- ❑ 机器语言、汇编语言

➤ 非结构化语言

- ❑ FORTRAN
- ❑ BASIC

➤ 结构化的语言

- ❑ PASCAL
- ❑ C
- ❑ Ada

➤ 面向对象的语言

- ❑ Smalltalk
- ❑ C++
- ❑ Java

➤ 组件编程

- ❑ .net
- ❑ C#/C++/Java

结构化程序设计

➤ 结构化程序设计方法

□ 自顶向下，逐步求精，模块化

☒ 先考虑总体，后考虑细节；先考虑全局目标，后考虑局部目标

☒ 对复杂问题，分解成若干简单问题，确立子目标，逐步细节化

□ 限制直接跳转（goto）的使用



结构化程序设计语言

➤ 非结构化程序设计语言

- ❑ ASM
- ❑ Fortran
- ❑ BASIC



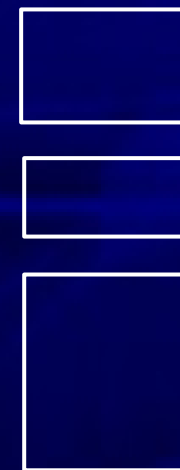
程序结构

➤ 结构化程序设计语言

- ❑ Algol
- ❑ Pascal
- ❑ C
- ❑ Ada



Pascal 结构



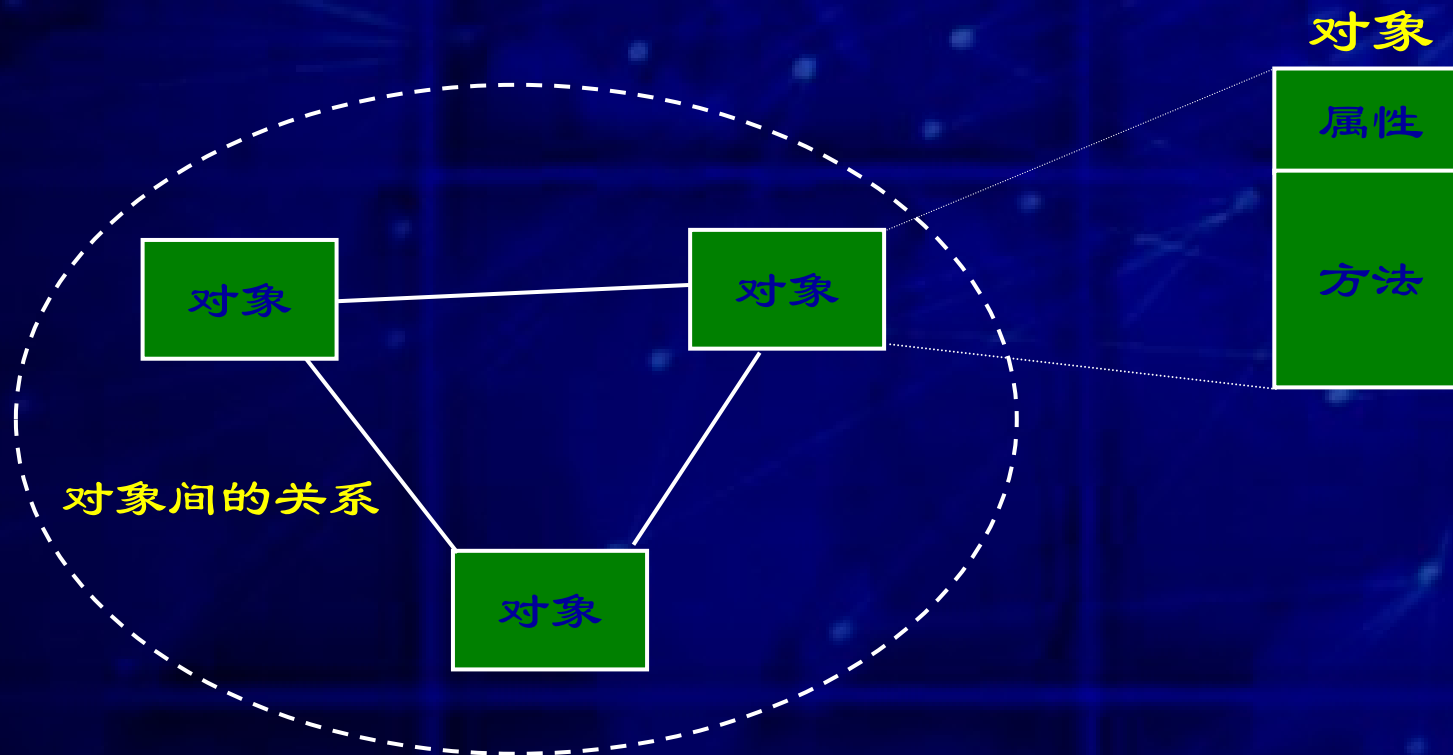
C 结构

面向对象的程序设计

➤ 面向对象的程序设计（Object Oriented Programming）

- ❑ 使用对象模型来描述和解决自然世界中的问题
- ❑ 能够大幅度的提高软件项目的成功率，提高软件的可移植性、重用（re-use）性和可靠性，减少日后的维护费用。
- ❑ 特征
 - ⊗ 对象模型，对象概念在从建模到构建程序的各个方面广泛使用
 - ⊗ 抽象化，对象的属性进行抽象
 - ⊗ 封装性，对对象的操作被封装在特定的作用范围
 - ⊗ 多态性，派生对象的操作（方法）可以存在不同实现
 - ⊗ 继承性，方法和属性可以在类间被继承和传递

面向对象的程序设计



程序 = 对象 + 对象间的相互作用

属性：描述对象性质的数据集合

方法：处理对象之间相互作用的操作方式

面向对象的程序设计语言

➤ Simula-67

➤ SmallTalk

➤ C++

➤ Java

➤ C#

C++ 与 C 的关系

➤ C++ 源于 C 语言

□ C 语言是在 B 语言的基础上发展起来的

- ⊗ 1960: ALGOL 60
- ⊗ 1963: 剑桥大学推出了CPL (Combined Programming Language) 语言, 后来经简化为BCPL语言
- ⊗ 1970: 贝尔实验室的 K.Thompson 以BCPL语言为基础, 设计了一种新语言, 取其第一字母 B, 称为 B 语言
- ⊗ 1972: 贝尔实验室的Dennis M.Ritchie为克服B语言的诸多不足, 在B语言的基础上重新设计了一种语言, 取其第二字母C, 称为 C 语言

□ C语言的特点是极高的代码效率, 但:

- ⊗ 不支持面向对象, 不支持类与封装机制
- ⊗ 难以支持代码重用

□ 1980年, 贝尔实验室的 Bjarne Stroustrup 对 C 语言进行了扩充, 提出了“带类的C”, 多次修改后起名为 C++, 以后又经过不断的改进

- ⊗ C++改进了C的不足之处, 支持面向对象的程序设计, 在改进的同时保持了C的简洁性和高效性。

C++ 与 C 的关系

➤ C++语言是在C语言的基础上进行了扩充和改进而得到的

- ❑ 它继承了 C 语言的全部内容
- ❑ 并在 C 语言的基础之上增加了面向对象编程的内容
- ❑ C++既支持面向过程的程序设计，又支持新型的面向对象程序设计

➤ C++保持了与C语言的兼容

- ❑ 大部分的 C 代码的程序不经修改，或很少修改就可被 C++ 使用
- ❑ 用 C 语言编写的许多库函数和应用软件也都可以用于 C++

➤ C++不是一个纯粹的面向对象程序设计语言

- ❑ C 语言是面向过程的语言，C++与C兼容，支持面向过程的程序设计
- ❑ 由于面向过程程序设计和面向对象程序设计是两种不同风格的程序设计技术，对于习惯于面向过程程序设计的程序员在学习使用 C++ 时可能存在一定的障碍

C++ 对 C 的扩展

➤ 增强的安全性

❑ 改进类型系统，强制的类型检查，有利于减少程序错误

```
#include <stdio.h>
main()
{
    int a=1;
    int b=2;
    printf("a+b=%d\n", func(a,b));
}
int func(int a, int b)
{
    return a+b;
}

a+b=3
```

```
#include <stdio.h>
main()
{
    float a=1;
    int b=2;
    printf("a+b=%d\n", func(a,b));
}
int func(int a, int b)
{
    return a+b;
}

a+b=1072693248
```


C++ 对 C 的扩展

➤ 增强的安全性

❑ 改进类型系统，强制的类型检查，有利于减少程序错误

```
#include <stdio.h>
main()
{
    int a=1;
    int b=2;
    printf("a+b=%d\n", func(a,b));
}
int func(int a, int b)
{
    return a+b;
}

a+b=3
```

```
#include <stdio.h>
int func(int a, int b); // 函数原型的说明
main()
{
    float a=1;
    int b=2;
    printf("a+b=%d\n", func(a,b));
}
int func(int a, int b)
{
    return a+b;
}

a+b=3
```

```
#include <stdio.h>
main()
{
    float a=1;
    int b=2;
    printf("a+b=%d\n", func(a,b));
}
int func(int a, int b)
{
    return a+b;
}

a+b=3
```

C++ 对 C 的扩展

- 增加了一些在新的运算符，使得C++应用起来更加方便
 - ❑ new, delete 用于内存管理，用户不需直接使用库函数
 - ❑ 增加了引用 &，使得引用函数参数带来了很大方便。
- 函数重载，设置缺省参数，提高编程灵活性，减少了代码冗余
- 引进了内联函数的概念，提高了程序的效率
- 变量在需要时进行说明，方便了程序编写
 - ❑ C语言先对变量的说明语句，再是执行语句

```
void func(int a, int b) { ... }  
void func(float a, float b){ ... }  
void foo(int k, int m=1);  
  
int i,j;  
float g,h;  
  
main()  
{  
    func(i,j);  
    func(g,h);  
    foo(100);  
}
```

```
void func(int a, int b)  
{  
    int i;  
    float g;  
  
    ...  
    i = a+b;  
    g = a-b;  
    ...  
}
```

C++ 对 C 的扩展

- 增加了一些在新的运算符，使得C++应用起来更加方便
 - ❑ new, delete 用于内存管理，用户不需直接使用库函数
 - ❑ 增加了引用 &，使得引用函数参数带来了很大方便。
- 函数重载，设置缺省参数，提高编程灵活性，减少了代码冗余
- 引进了内联函数的概念，提高了程序的效率
- 变量在需要时进行说明，方便了程序编写
 - ❑ C语言先对变量的说明语句，再是执行语句

```
void func(int a, int b) { ... }  
void func(float a, float b){ ... }  
void foo(int k, int m=1);  
  
int i,j;  
float g,h;  
  
main()  
{  
    func(i,j);  
    func(g,h);  
    foo(100);  
}
```

```
void func(int a, int b)  
{  
    ...                // 执行语句  
  
    int i;  
    i = a+b;  
    ...                // 其他执行语句  
    float g;  
    g = a-b;  
    ...  
    for (int j=0;j<100;j++)  
    { ...  
    }  
}
```

C++ 对 C 的扩展

➤ 与面向对象相关的功能：

❑ 类与对象：抽象、封装、继承与派生、多态性

➤ 异常处理

➤ 模板编程，提供标准模板库 (STL)

❑ 容器、流式 IO、字符串、...

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

main()
{
    char buf_1[1024];
    char buf_2[1024];

    printf("Please input word 1:");
    sscanf(buf_1,"%s");
    printf("Please input word 2:");
    sscanf(buf_2,"%s");
    strcat(buf_1,buf_2);
    printf("%s\n",buf_1);
}
```

```
#include <iostream>
#include <string>
using namespace std;

main()
{
    string word_1, word_2;
    cout << " Please input word 1:" ;
    cin >>word_1;
    cout << " Please input word 2:" ;
    cin >>word_2;
    cout << word_1+word_2 << endl;
}
```

C++ 对 C 的扩展

➤ 与面向对象相关的功能：

❑ 类与对象：抽象、封装、继承与派生、多态性

➤ 异常处理

➤ 模板编程，提供标准模板库 (STL)

❑ 容器、流式 IO、字符串、...

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

main()
{
    char buf_1[1024];
    char buf_2[1024];

    printf("Please input string 1:");
    scanf(buf_1,"%s");
    printf("Please input string 2:");
    scanf(buf_2,"%s");
    strcat(buf_1,buf_2);
    printf("%s\n",buf_1);
}
```

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

main()
{
    ofstream outfile ( "filename_1" );
    outfile << " This is a test ! " << endl;

    ifstream infile ( "filename_2" );
    string word;
    while ( infile >> word )
    {
        cout << word << endl;
    }
}
```

```
word 1:" ;
word 2:" ;
<< endl;
```


C++ 对 C 的扩展

➤ 编程方法

- ❑ C++ 对 C 的兼容是建立在发展和完善的基础上的
- ❑ C++ 支持面向对象的程序设计
- ❑ C 是面向过程的程序设计语言

➤ 程序结构

- ❑ C++ 程序（面向对象）的结构采用“对象+消息”模式
- ❑ C 程序结构采用“数据+算法”模式

➤ 适用性

- ❑ 底层系统程序、嵌入式程序设计、大规模高层应用设计、通用程序设计、数值科学计算 ...

程序设计

代码的版式与风格

➤ 关系到代码的可读性

- ❑ 继而影响到代码的调试、管理、移植等多方面的性能
- ❑ 因此，在编写代码时，应养成良好的习惯
- ❑ 良好的代码风格最主要是为自己和他人提供可读性
 - ✉ 这对于大型软件开发是非常重要的
- ❑ 没有固定的模式，关键是使代码流程清晰，易于识别

代码的版式与风格

➤ 一般规则

- ❑ 文件开始写出完整的文件版本说明
- ❑ 必需的注释
 - ✉ 例如，函数的用途、输入/输出的说明
- ❑ 适当的空行和空格
- ❑ 与模块化一直的对齐和缩进
- ❑ 长行的拆分
- ❑ 良好的命名规则

代码的版式与风格

➤ 一般规则：文件开始写出完整的文件版本说明

```
/* -----  
    HY-1 Communication Subsystem : event log subroutines (common file)  
    -----  
    Programmer : Gong Xiangyang  
    version    : 1.00  
    Date       : 2006-02-25  
    *****  
    Error and event handling and logging  
    -----*/  
  
#include <sys/types.h>  
#include <sys/stat.h>  
#include <fcntl.h>  
#include <unistd.h>  
#include <string.h>  
#include <stdio.h>  
#include <sys/time.h>
```

代码的版式与风格

➤ 一般规则：必需的注释

✉ 例如，数据结构的功能和含义；函数的用途、输入/输出的说明

```
/* ++++++
data structures for logging
-----*/

#define EVH_FILELOG      0x01      // log to file
#define EVH_CONSOLE     0x02      // log to console

typedef struct    __EventInfoRecord__    // Event information record
{
    int          iEvLogFlag;           // event handling flags
    int          iEvLogLevel;          // event level
    int          iParNr;               // parameter number
    int          iEventID;             // event identifier
}
    EvInfoRecord;

/*****
Global data structure
*****/
char      sMsgBuf[4096];              /* message buffer */
static FILE *    pLogFile = NULL;    // log file pointer
```

代码的版式与风格

➤ 必需的注释

➤ 适当的空格、空行，适当的对齐和缩进

```
/*+++++
function: event handling, normal operation is to log this event
input:    iEventID :      Event ID
          sEvDesc:      Event decription
output:    none
+++++ */
void LogEvent(int iEventID, char * sEvDesc)
{
    EvInfoRecord * pEvInfo;

    if (!iEventID)
    {
        OutputEventLog(0, sEvDesc, EVH_FILELOG | EVH_CONSOLE | EVH_PROMPT);
        pEvInfo = GetEvInfoTabEntry(iEventID);

        if (iEventID != pEvInfo->iEventID)
            return;
    }
    else
    {
        if (pEvInfo->iEvLogFlag & EVH_SENDMSG)
            SendCtrlMessage(iEventID, sEvDesc);
    }

    OutputEventLog(iEventID, sEvDesc, pEvInfo->iEvLogFlag);
}
```

代码的版式与风格

➤ 一般规则：长行的拆分

```
/*+++++
function:  return current date and time in a string
format is:
                2001-05-20 22:30:58
+++++ */
char *    GetCurrentTimed(void)
{
    static char        sBuf[20];
    struct timeval      stCurTime;
    struct tm*         pTM;

    gettimeofday(&stCurTime,NULL);
    pTM = localtime(&(stCurTime.tv_sec));

    sprintf(sBuf,"%04d-%02d-%02d %02d:%02d:%02d",
            pTM->tm_year+1900,pTM->tm_mon+1, pTM->tm_mday,
            pTM->tm_hour,pTM->tm_min,pTM->tm_sec);

    return    sBuf;
}
```

代码的版式与风格

➤ 标识符的命名规则

- ❑ 标识符可以“望文生义”，易于理解
- ❑ 应当符合“min-length max-information”原则
- ❑ 命名规则尽量与所采用的操作系统或开发工具的风格保持一致

➤ 常用法则：Microsoft 的“匈牙利”命名法

- ❑ 以一个有意义的词组来描述标识符，各单词首字母大写
- ❑ 标识符上增加适当前缀，以增进对程序的理解

```
#define MAX_VALUE 65536
```

```
DWORD g_dwSum;
```

```
class CConfig
```

```
{
```

```
private:
```

```
    CConfig * m_pConfig;
```

```
public:
```

```
    int ReadKey(const char * pSecName, const char * pKeyName);
```

```
    int WriteKey(const char * pSecName, const char * pKeyName, int iValue);
```

```
};
```

C++ 语言基础

北京 电大学

网络与交换技术国家 点实 室

宽带网研究中心

内容

- 基本数据类型
- 表达式
- 基本控制结构
- 函数

C++字符集

➤ 大小写的英文字母

A~Z a~z

➤ 数字字符

0~9

➤ 符号

!	#	%	^	&	*	:		
_	+	=	-	~	<	>	/	\
`	"	;	.	,	()	[]	{}	

➤ 其他符号

<空格符> <换行符> <TAB>

词法记号

- 关键字 C++ 定义的单词
- 标识符 程序员声明的单词 它命名程序正文中的一些实体
- 文字 在程序中直接使用符号表示的数据
- 操作符 用于实现各种 算的符号: $+$ $-$ $*$ $/$...
- 分 符 用于分 各个词法记号或程序正文
- 空白符
 - ❑ 空格
 - ❑ 制表符
 - ❑ 换行符
 - ❑ 注

关键字

► 关键字是C++保留的 作为专用定义符的单词 在程序中不允许另作它用

auto	break	case	char	class
const	continue	default	do	default
delete	double	else	enum	explicit
extern	false	float	for	friend
Goto	if	inline	int	long
Mutable	new	operator	private	protected
public	register	return	short	signed
sizeof	static	static_cast	struct	switch
this	ture	typedef	union	unsigned
Virtual	void	while		

标识符

➤ 标识符是对实体定义的一种定义符

- ❑ 由字母或下划线开头、后跟字母或数字或下划线组成的字符序列
- ❑ 具有特定的有效度
 - ⊠ ANSIC 标准规定 31 个字符
- ❑ 用来标识用户定义的常名、变名、函数名、文件名、数组名、和数据类型名和程序等。
- ❑ 大写字母和小写字母代表不同的标识符

文字常量

➤ 数字常

1	100	200U	10000L
0x100	3.14	1.25e+10	

➤ 字符常

‘a’ ‘b’

➤ 字符串

□ 字符串是由双引号括起来的字符串常
“China”、 “C++ Program”

分隔符与空白符

➤ 分 符

- 用于分 各个词法记号或程序正文
{ } () 和空白符

➤ 空白符

- 空格、换行、制表符

□ 注

⊗ 是用来帮助 读、理解及维护程序

⊗ 注 分被忽略 不产生目标代码

⊗ C++语言提供两种注 方式

一种是与C兼容的多行注 用 /* 和 */ 分界

另一种是单行注 以 “//” 开头 表明本行中 “//” 符号后的内容
是注

C++ 简单实例

```
//2_1.cpp
#include <iostream>
using namespace std;

int main()
{
    cout<<"Hello!\n";
    cout<<"Welcome to c++!\n";
}
```

行结果

```
Hello!
Welcome to c++
```

C++ 简单实例

```
#include <iostream>
using namespace std;

int main(void)
{
    const int PRICE = 30;           // 符号常量

    int num, total;                 // 变量
    num = 10;
    total = num * PRICE;
    cout << "Total:" << total << endl; // 字符串常量

    float v, r, h;
    r = 2.5;                        // 文字常量
    h = 3.2;
    v = 3.14159 * r * r * h;

    cout << v << endl;
}
```


数据类型

➤ 内 数据类型

□ char int short long __int64 float double bool 指 *)

□ 在整型类型前可以用 unsigned 来修

□ 内存映像 32bit Intel 为例

提倡用 sizeof () 来确定数据类型或变 的大小

1 byte

char
bool

2 byte

short

4 byte

int
long
float
pointer (*)

8 byte

double
__int64
long long)

枚举类型—enum

➤ 只要将 要的值一一列举出来 便构成了一个枚举类型。

➤ 枚举类型的声明形式如下

```
enum 枚举类型名 {变 值列表};
```

➤ 例如

```
enum weekday {sun,mon,tue,wed,thu,fri,sat};
```

➤ 枚举元素具有缺省整型值 它们依次为 0,1,2,.....。

□ 也可以在声明时另行指定枚举元素的值 如

```
enum weekday {sun=7,mon=1,tue,wed,thu,fri,sat};
```

数据类型

➤ 用户定义数据类型/抽象数据类型 UDT/ADT

□ 使用 struct / class 定义

```
struct Sample_Struct
{
    int    iCount;
    char * pName;
} sample;

class CComputer
{
private:
    char * m_pComputerName;
    ...
public:
    void PowerOn();
    void Startup();
    ...
};
```

联合

➤ 声明形式

```
union 联合名
{
    数据类型 成员名 1
    数据类型 成员名 2

    数据类型 成员名 n
};
```

➤ 联合的成员共享相同的地址空

➤ 联合体类型变 说明的语法形式

联合名 联合变 名

➤ 引用形式

联合变 .成员名

联合指 ->成员名

typedef 语句

➤ 为一个已有的数据类型另外命名

➤ 语法形式

typedef 已有类型名 新类型名表;

➤ 例如

```
typedef double area, volume;
```

```
typedef int natural;
```

```
natural i1, i2;
```

```
area a;
```

```
volume v;
```

变量初始化

例

```
int      a = 3;
```

```
double   f = 3.56;
```

```
char     c = 'a';
```

```
int      i(5);
```

变量的存储类型

➤ auto 变量的缺省类型

- ❑ 临时存储变量 具有生命周期 生命期结束后存储空间 可
以被其他自动变量 覆盖使用

➤ register

- ❑ 建议使用 寄存器

➤ extern

- ❑ 声明该变量的定义不一定在本源文件中

➤ static

- ❑ 静态 在程序的整个生命周期 执行期 有效

```
extern int e;  
void func(int a, int b)  
{  
    int i;  
    auto int j;  
    static int x;  
    register int y;  
}
```

```
void func(int a, int b)  
{  
    int i;  
    ...  
    {  
        int i = 1;  
        i++;  
        ...  
    }  
}
```


类型转换

- 不同类型数据 行混合 算时 C++编译器会自动 行类型转换。
- 为了 免不同的数据类型在 算中出现类型 应尽 使用同种类型数据。
- 可以 用强制类型转换

强制类型转换

➤ 语法形式

类型说明符(表达式)

或

(类型说明符)表达式

➤ 强制类型转换的作用是将表达式的结果类型转换为类型说明符所指定的类型

```
float c;
```

```
int a , b;
```

```
c = float(a) / float(b);
```

```
c = (float)a / (float)b;
```

运算符

Operator	Name or Meaning	Associativity
::	Scope resolution	None
.	Member selection (object)	Left to right
->	Member selection (pointer)	Left to right
[]	Array subscript	Left to right
()	Function call	Left to right
()	member initialization	Left to right
++	Postfix increment	Left to right
--	Postfix decrement	Left to right
typeid()	type name	Left to right
const_cast	Type cast (conversion)	Left to right
dynamic_cast	Type cast (conversion)	Left to right
reinterpret_cast	Type cast (conversion)	Left to right
static_cast	Type cast (conversion)	Left to right

`CComputer::PowerOn()`

`Comp.Mouse`

`pComp->Mouse`

`int array[10]`

`printf("%d\n", a);`

`CComputer Comp(10);`

`a++`

`a--`

`typeid(Comp)`

运算符

<code>sizeof</code>	Size of object or type	Right to left
<code>++</code>	Prefix increment	Right to left
<code>--</code>	Prefix decrement	Right to left
<code>~</code>	One's complement	Right to left
<code>!</code>	Logical not	Right to left
<code>-</code>	Unary minus	Right to left
<code>+</code>	Unary plus	Right to left
<code>&</code>	Address-of	Right to left
<code>*</code>	Indirection	Right to left
<code>new</code>	Create object	Right to left
<code>delete</code>	Destroy object	Right to left
<code>()</code>	Cast	Right to left
<code>.*</code>	Pointer-to-member (objects)	Left to right
<code>->*</code>	Pointer-to-member (pointers)	Left to right

`sizeof (CComputer)`

`++a`

`--a`

`~a`

`!a`

`-a`

`+a`

`&Comp`

`*pComp`

`pComp = new Computer;`

`delete pComp;`

`(unsigned) a`

`Comp.*Startup()`

`pComp->*Startup()`

运算符

*	Multiplication	Left to right
/	Division	Left to right
%	Modulus	Left to right
+	Addition	Left to right
-	Subtraction	Left to right
<<	Left shift	Left to right
>>	Right shift	Left to right
<	Less than	Left to right
>	Greater than	Left to right
<=	Less than or equal to	Left to right
>=	Greater than or equal to	Left to right
==	Equality	Left to right
!=	Inequality	Left to right
&	Bitwise AND	Left to right
^	Bitwise exclusive OR	Left to right
	Bitwise inclusive OR	Left to right

&&	Logical AND	Left to right
	Logical OR	Left to right
<i>e1?e2:e3</i>	Conditional	Right to left
=	Assignment	Right to left
*=	Multiplication assignment	Right to left
/=	Division assignment	Right to left
%=	Modulus assignment	Right to left
+=	Addition assignment	Right to left
-=	Subtraction assignment	Right to left
<<=	Left-shift assignment	Right to left
>>=	Right-shift assignment	Right to left
&=	Bitwise AND assignment	Right to left
=	Bitwise inclusive OR assignment	Right to left
^=	Bitwise exclusive OR assignment	Right to left
throw <i>ex</i>	throw expression	Right to left
,	Comma	Left to right

表达式

➤ 算术表达式

□ 基本算术 算符 + - * / (若整数相 结果取整)

➤ 赋值表达式: = += -= *= /= %= <<= >>= &= ^= |=

➤ 逗号表达式: 表达式1 表达式2

➤ 关系表达式: < <= > >= == !=

➤ 逻辑表达式: ! && ||

➤ 条件表达式: 表达式1 表达式2 表达式3

sizeof 运算符

➤ 语法形式

sizeof (类型名)

sizeof (变 名)

sizeof (表达式)

➤ 结果值

“类型名”所指定的类型或“表达式”的结果类型所占的字节数。

➤ 例

sizeof (short)

sizeof (x)

位运算

➤ 按位与 (&)

➤ 按位或 (|)

➤ 按位异或 ^ ,即半加

➤ 按位取反 ~

➤ 左移 算 <<

➤ 右移 算 >>

C++ 语句

➤ 声明语句

➤ 表达式语句

➤ 择语句

➤ 循环语句

➤ 跳转语句

➤ 复合语句

➤ 标号语句

语句之 要使用分 符

```
int a = -1;

if (a < 0) a = 0;

for (int i=0; i<10; i++)
{
    a ++;
    a += i;
}

if (a < 0) a = 0;
```


C++ 语句

➤ 声明语句 声明或定义变 、函数等

```
int a;  
extern void func(int x);
```

➤ 表达式语句

```
a = 100;
```

➤ 将多个语句用大括号包围 便构成一个复合语句

```
{  
    sum = sum + i;  
    i ++;  
}
```

简单的输入、输出

➤ 向标准输出设备 显示器 输出

例

```
int x;  
cout << "x=" << x;    // 表达式语句
```

➤ 从标准输入设备 盘 输入

例

```
int x;  
cin >> x;                // 表达式语句
```

其中 cin、cout 是 C++ 流式 IO 库中定义的
所以使用时一般会有如下语句

```
#include <iostream>  
using namespace std;
```

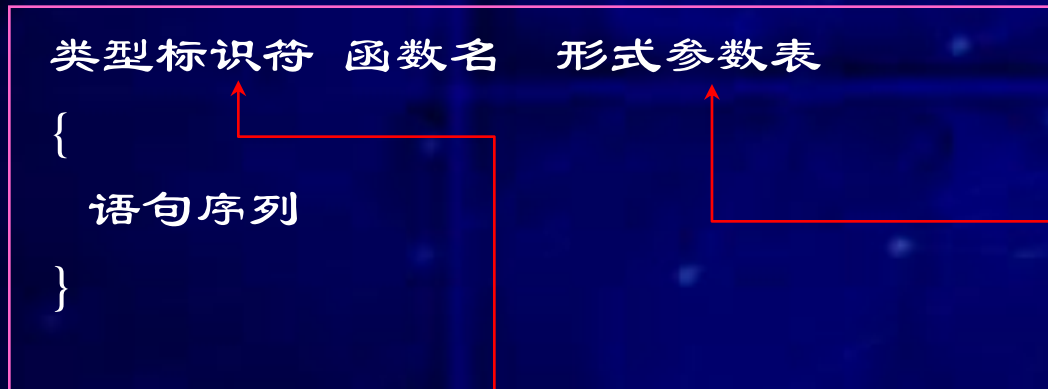
算法的基本控制结构

- **序结构**
- **分支结构** if, switch
- **循环结构** while do-while, for,
break 和 continue

函数

函数的声明

➤ 函数声明的语法形式



是被初始化的内部变量，寿命和可见性仅限于函数内部

若无返回值，应写void

函数的声明

➤ 形式参数表

$\langle \text{type}_1 \rangle \text{ name}_1, \langle \text{type}_2 \rangle \text{ name}_2, \dots, \langle \text{type}_n \rangle \text{ name}_n$

➤ 函数的 返回值

□ 由 `return` 语句给出 例如

`return 0`

□ 无 回值的函数 `void`类型 不必写 `return`语句

函数的调用

➤ 调用前必 先定义该函数 或声明函数原型

□ 函数原型说明

类型标识符 被调用函数名 含类型说明的形参表 ;

例如

```
void func(int a, int b);  
float foo(float a);
```

➤ 调用形式

函数名 实参列表

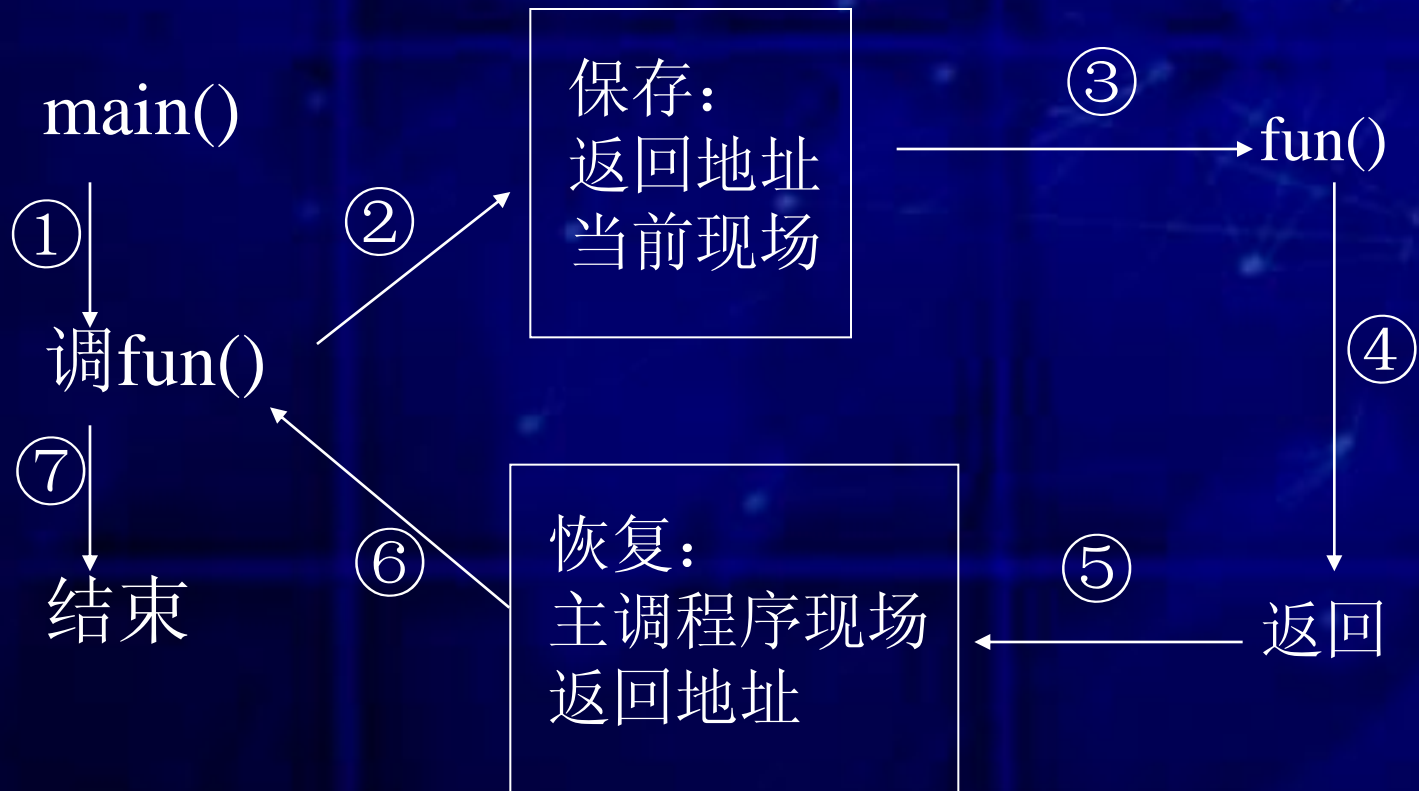
➤ 嵌套调用

□ 函数可以嵌套调用 但不允许嵌套定义。

➤ 递归调用

□ 函数直接或 接调用自身。

函数调用的执行过程



嵌套调用

函数的声明与使用

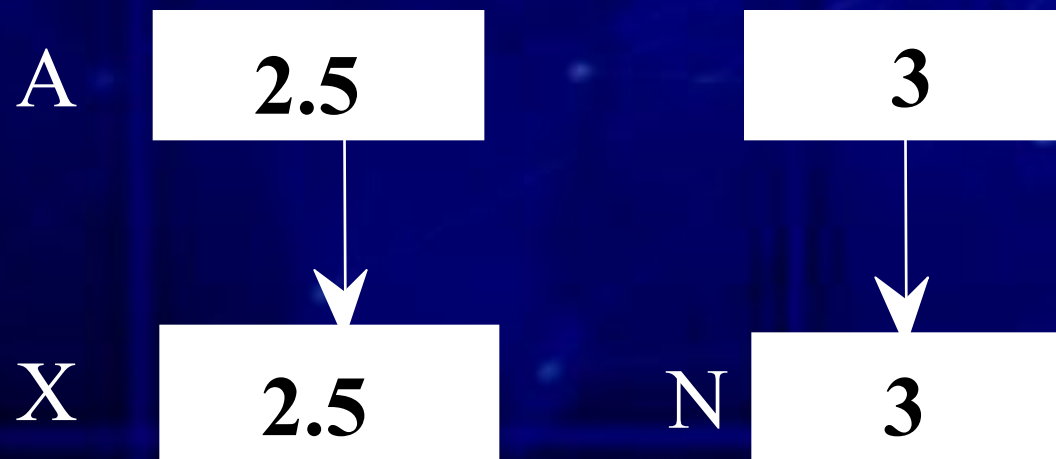


函数的参数传递机制

- 在函数被调用时才分配形参的存储单元。
- 实参可以是常量、变量或表达式。
- 实参类型必须与形参相符。
- 传参时是传参数值，即单向传参。

函数的参数传递机制

调用者: $D = \text{power}(A, 3)$



被调函数:

`double power(double X, int N)`

例：参数传递

```
#include<iostream>
using namespace std;

void Swap(int a, int b);

int main()
{
    int a(5), b(10);

    cout << "(a,b)=" << a << "," << b << endl;
    Swap(a,b);
    cout << "(a,b)=" << a << "," << b << endl;
    return 0;
}

void Swap(int a, int b)
{
    int t;
    t=a;
    a=b;
    b=t;
}
```

输出结果

(a,b)=5,10

(a,b)=5,10

例：参数传递（使用指针）

```
#include<iostream>
using namespace std;

void Swap(int *a, int *b);

int main()
{
    int a(5), b(10);

    cout << "(a,b)=" << a << "," << b << endl;
    Swap(&a, &b);
    cout << "(a,b)=" << a << "," << b << endl;
    return 0;
}

void Swap(int *a, int *b)
{
    int t;
    t=*a;
    *a=*b;
    *b=t;
}
```

输出结果

(a,b)=5,10

(a,b)=10,5

数据类型：引用

➤ 引用(&)是标识符的别名,例如:

```
int  i, j;  
int& ri=i;           //建立int型的引用ri,初始化为变  i的引用  
                        //  
  
j   = 10;  
ri  = j;             //相当于 i=j;  
  
  
double  f;  
double &rf = f;  
rf = 100.5;         //相当于 f=100.5;
```

- 声明一个引用时 必 同时对它 行初始化 使它指向一个已存在的变 对象 。
- 一旦一个引用被初始化后 就不能改为指向其它对象。
- 引用可以作为函数的参数 行传 也可以作为 回值

```
void swap(int& a, int& b) {...}
```

例：参数传递（使用引用）

```
#include <iostream>
using namespace std;

void Swap(int& a, int& b);

int main()
{
    int a(5), b(10);

    cout << "(a,b)=" << a << "," << b << endl;
    Swap(a,b);
    cout << "(a,b)=" << a << "," << b << endl;
    return 0;
}

void Swap(int& a, int& b)
{
    int t;
    t=a;
    a=b;
    b=t;
}
```

输出结果

(a,b)=5,10

(a,b)=10,5

内联函数

➤ 声明时使用关键字 inline。

```
inline void Swap(int& a, int& b)
{
    int t;
    t=a; a=b; b=t;
}
int main()
{
    int a(5), b(10);
    Swap(a,b);
    return 0;
}
```

➤ 编译时在调用处用函数体 行替换,节省了参数传、控制转移等开。

□ 内联函数的声明必 出现在内联函数第一次被调用之前

默认参数

➤ 函数在声明时可以 先给出 认的形参值 调用时如给出实参 则 用实参值 否则 用 先给出的 认形参值。

➤ 例如

```
int add(int x=5, int y=6)
{
    return x+y;
}

int main()
{
    add(10,20);      // 10+20
    add(10);          // 10+6
    add();            // 5+6
    return 0;
}
```

默认参数

- 认形参值必 从右向左 序声明 并且在 认形参值的右 不能有 认形参值的参数。因为调用时实参取代形参是从左向右的 序。

- 例

```
int add(int x,    int y=5, int z=6); //正确
int add(int x=1, int y=5, int z);    // 误
int add(int x=1, int y,    int z=6); // 误
```

默认参数

- 若调用出现在函数体实现之后 认参数可在函数实现时给出
- 认参数值也可以函数原型中给出 但 认参数只能定义一次

```
int add(int x, int y);  
void func()  
{  
    add(10);  
}  
int add(int x=5, int y=6)  
{  
    return x+y;  
}
```

```
int add(int x=5, int y=6);  
int main()  
{  
    add(10);           // 10+6  
    add();             // 5+6  
    return 0;  
}  
int add(int x, int y = 6)  
{ return x+y;  
}
```

```
int add(int x, int y);  
void func()  
{  
    add(10,20);  
    add(10);  
}  
int add(int x=5, int y=6);  
int main()  
{  
    add(10);           // 10+6  
    add();             // 5+6  
    return 0;  
}  
int add(int x, int y)  
{  
    return x+y;  
}
```

默认参数

- 在相同的作用域内 认形参值的说明应保持唯一 但如果在不同的作用域内 允许说明不同的 认形参。

➤ 例

```
int add(int x=1,int y=2);

int main()
{
    int add(int x=3,int y=4);
    add();           //使用局    认形参值    实现3+4
    retrun 0
}

void fun(void)
{
    ...
    add();           //使用全局    认形参值    实现1+2
}
```

函数重载 (function overloading)

- C++允许功能相同的函数在相同的作用域内以相同函数名声明从而形成 重载
- 实 意义和功能相同或相 方便使用 便于记忆
- 编译器自动根据参数类型、个数等调用相应的函数

```
int    add(int x, int y);  
float  add(float x, float y);
```

} 形参类型不同

```
int    add(int x, int y);  
int    add(int x, int y, int z);
```

} 形参个数不同

```
int    x, y, z;  
float  e, f;
```

```
add(x, y);
```

```
add(x, y, z);
```

```
add(e, f);
```

函数重载

- ❑ 重载函数的形参必须不同：个数不同或类型不同。
- ❑ 编译程序将根据实参和形参的类型及个数的最佳匹配来选择调用哪一个函数。

```
int add(int x,int y);
```

```
int add(int a,int b);
```

编译器不以形参名来区分重载

```
int add(int x,int y);
```

```
void add(int x,int y);
```

编译器不以返回值来区分重载

- ❑ 不要将不同功能的函数声明为重载函数，以免出现调用结果的误解、混淆

类与对象

北京邮电大学

网络与交换技术国家重点实验室

宽带网研究中心

主要内容

➤ 面向对象的程序设计

➤ C++的类与对象

- 类的定义

- 成员

- 构造函数

- 析构函数

面向对象的程序设计

面向过程与面向对象

➤ 面向对象程序设计 (OOP)

- ❑ Object Oriented Programming
- ❑ 目前一种主流程序设计方法

➤ C++是混合编程语言

- ❑ C++保持与 C 相兼容
- ❑ 既支持面向对象程序设计方法，也支持面向过程程序设计方法，是一种混合编程语言。

面向过程程序设计方法

➤ 机制

- 首先定义所要实现的功能
- 然后为这些功能设计所必要的步骤或过程
 - ⊗ 解决问题的重点是如何实现过程的细节
 - ⊗ 数据与操作这些数据的过程分离
 - ⊗ 围绕功能（过程或操作）实现来设计程序
- 采用自顶向下（瀑布式的流程），功能分解法
 - ⊗ 采用逐步求精的方法来组织程序的结构
- 程序组成形式 —— 主模块 + 子模块
 - ⊗ 它们之间以数据作为连接纽带
 - ⊗ 程序 = 算法 + 数据结构
- 数据处于次要的地位，而过程是关心的重点

面向过程程序设计

➤ 缺点：

- ❑ 数据与操作这些数据的过程分离，一旦问题改变（数据结构发生变化），就需要重新修改问题的解决方法（操作数据的过程）
- ❑ 软件维护成本高
- ❑ 不利于代码重用（re-use）
 - ⊗ 以函数（功能、过程）的方式实现代码重用，效率低
 - ⊗ 理想的方式：问题的解决方案能够重用
- ❑ 不适于中大型、巨型软件程序设计

例：数据与操作这些数据的过程分离

```
#include<iostream>
using namespace std;

int global;

void f()
{
    global=5;
}

void g()
{
    cout<<global<<endl;
}

int main()
{
    f();
    g();
    return 0;
}
```

**需要处理的数据
很多的时候怎么办？**

面向对象程序设计

➤ 面向对象概念

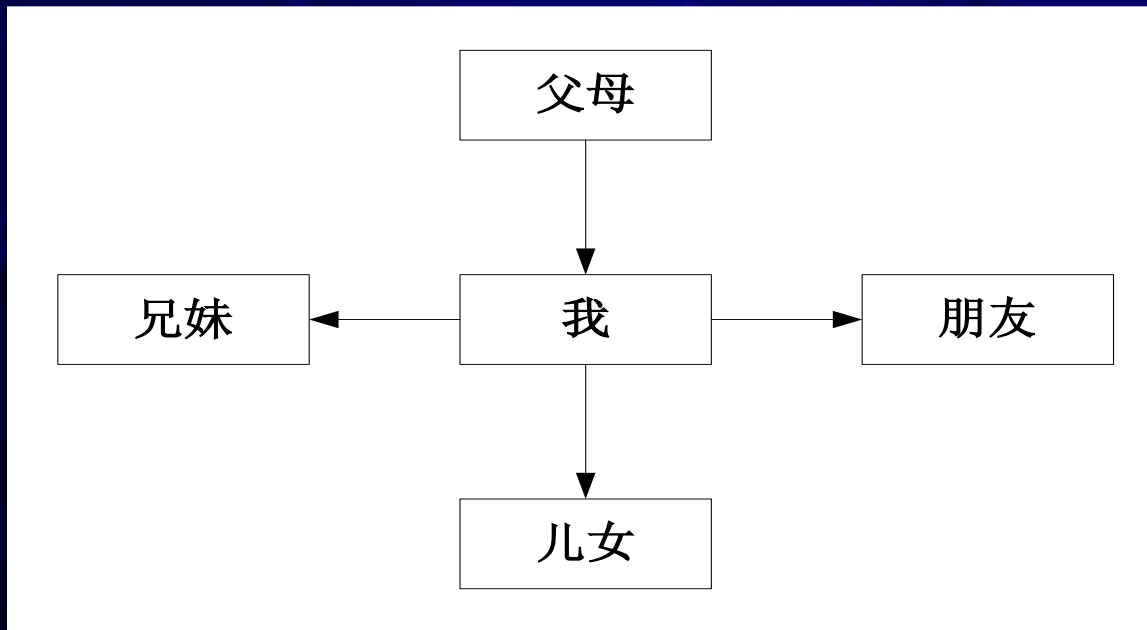
- ❑ 是一种解决问题的方法或观点
- ❑ 认为自然界是由一组彼此相关联并相互作用（通信）的实体所组成，称为对象（Object）
 - ☒ 对象化的表示更接近于对世界的自然描述

➤ 程序员使用面向对象的观点来分析问题

- ❑ 将所要解决的问题转化为程序中的对象——任何问题在程序中都映射为对象
- ❑ 找出问题的属性（数据描述）与操作方法（通过函数来体现）
- ❑ 然后用计算机语言来描述问题，最后在计算机中加以处理

以“人”为例：

- 每一个“人”的个体都是“人类”的一个实例（对象）
- 每一个“人”有自己的属性
 - ❑ 姓名、年龄、职业、...
- 个体之间存在着各种关系
 - ❑ 个体之间能够相互作用（操作）：交谈、合作、...



面向对象程序设计

➤ 程序设计

- ❑ 程序员在解决问题时应具有高度的概括、抽象能力
- ❑ 根据自然实体（待解决的问题）的属性进行分类和抽象
- ❑ 准确描述实体类的属性和行为
- ❑ 对问题进行分析和抽象，使用程序设计语言中的类和类之间的关系来描述待解决的问题及其相关性
- ❑ 对类进行具体化，产生出对应的问题对象，以消息传递的机制来组织对象之间的相互作用和操作

➤ 程序组成形式：对象 + 消息

- ❑ 对象与对象之间通过消息作为连接相互驱动
- ❑ 对象的行为（操作）体现为对消息的处理方式
- ❑ 对象（问题）之间的关系是编程关心的重点，而对象功能实现细节则处于次要的地位，并且通常被封装

面向对象程序设计

➤ 面向对象程序设计的优越性

- ❑ 提高软件质量：实现数据与方法的封装，通过方法来操作改变数据，提高了数据访问的安全性
- ❑ 易于软件维护
- ❑ 支持软件重用，大大提高软件生产的效率
- ❑ 实现可重用的软件组件，实现软件设计的产业化
 - ⊗ 由于程序是类的集合从而可以根据问题的相关性来组装程序
 - ⊗ 而面向过程程序设计则是函数的集合，零散不便于代码重用

例：数据与行为的封装

```
#include<iostream>
using namespace std;

class Application
{
    public:
        void f();
        void g();
    private:
        int global;
};

void Application::f()
{
    global=5;
}
```

```
void Application::g()
{
    cout<<global<<endl;
}

int main()
{
    Application  MyApp;

    MyApp.f();

    MyApp.g();

    return 0;
}
```

总结：面向过程与面向对象

➤ 面向过程的程序设计

- Program = Algorithms + Data Structures

- 问题求解的方法：向过程传递数据

➤ 面向对象的程序设计

- 问题求解的方法：向对象发送消息

➤ 面向过程的程序设计语言与面向对象的程序设计语言

- Modula-2、Ada

 - ☒ 支持数据隐藏和数据封装

- C++、Java

 - ☒ 支持数据隐藏和数据封装

 - ☒ 支持继承和多态、支持重用

OOP 基本手段——抽象

- 对具体问题（对象）进行分类概括，提取出这一类对象的共同性质并且加以描述的过程。
- 编程的要求：
 - ❑ 先注意问题的本质及描述，其次是实现过程或细节。它直接决定程序的优劣——类的定义及组成元素；
 - ❑ 所涉及到的主要内容：
 - ⊗ 数据抽象（属性）——描述某类对象的属性或状态（对象相互区别的物理量）；
 - ⊗ 行为抽象（方法）——描述某类对象的共有的行为特征或具有的功能。
 - ❑ 抽象的实现：在 C++ 中通过类 class 来声明

OOP 基本手段——抽象

➤ 例：对现实中钟表类的抽象

□ 属性抽象：所有钟表都具有以下属性

```
int Hour;  
int Minute;  
int Second;
```

□ 行为（操作）抽象：钟表对外（其他对象）体现的行为

```
SetTime(int Hour, int Minute, int Second);  
ShowTime();
```

□ 注：钟表内部的如何保证准确的时间是钟表内部实现的细节，使用钟表功能的外部对象并不关心这些细节
在 OOP 中，这些细节也是次要的

OOP 基本手段——抽象

➤ 例：对人的抽象得到“人类”

□ 属性抽象

```
char *name;  
int age;  
int id;  
...
```

□ 行为的抽象：

```
Eat();  
Speak();  
Work();  
...
```

OOP 基本手段——封装

➤ 将抽象出的属性和行为结合，作为一个整体来对待

- ❑ 使用的工具：在 C++ 中使用类 class 来进行封装
- ❑ 属性表示为类中的数据成员
- ❑ 行为表示为类中的成员函数
- ❑ 封装机制将这二类成员组合在一起，形成问题的类，类实例化就可以产生类的实体——对象

➤ 封装的目的：

- ❑ 增强安全性和简化编程，使用者不必(甚至是不能)了解具体的实现细节，而只需要通过类外部接口，以特定的访问权限，来使用类的成员
- ❑ 高度模块化，易于生产软件组件（例如控件等）

➤ 实现封装：定义 class

数据封装和数据隐藏

➤ 数据封装和数据隐藏

❑ Encapsulation

- ✉ 将基本数据和能够对这些基本数据进行的操作（方法）结合起来
- ✉ 对这些数据的操作只能通过指定的方法（method）来进行
- ✉ 这些操作方法也称为接口（Interface）

❑ Data hiding

- ✉ 将基本数据的内部结构对外隐藏起来，使内部数据结构对外具有不可访问性

❑ 数据封装和数据隐藏的优点：可维护性

- ✉ 将数据及对数据的处理封装起来，对外隐藏实际的数据结构及对数据处理的实现，对外提供一致的接口
- ✉ 当数据及操作的内部实现发生改变时，只要接口保持一致，将不会影响软件的其他部分

数据封装和数据隐藏

➤ C++ 的类是一种用于数据封装和数据隐藏的工具

□ 类 (class) 定义用户的抽象数据结构

- ⊗ 基本数据和抽象数据结构 (对象), 即类的成员变量
- ⊗ 对这些数据的操作方法 (类的成员函数)
- ⊗ 所有的成员函数中, 暴露给外部、可被外部使用者调用的方式 (public 方法) 是类的接口 (interface) 方法

□ 对象 (Object) 是类的实例

- ⊗ 一个对象是一个具有内存映像、符合类结构的实体
- ⊗ 使用者对对象实体的操作, 通过调用该对象的接口方法来实现
- ⊗ 调用对象的接口方法, 可以看作是向该对象发送了一条消息; 接口方法的实现 (对内部数据进行操作), 是对象对该消息进行处理的过程

OOP 基本手段——封装

► 实例：

```
class Clock
{
    public:                                // 外部接口
        void SetTime(int NewHour, int NewMin, int NewSec);
        void ShowTime(void);

    private:                               // 内部属性
        int Hour;
        int Minute;
        int Second;
        void Run(void);                    // 内部行为
};
```

成员变量

成员函数

内部行为

封装的特征

➤ 有一定的边界

- ❑ 所有的内部变化都限制在此边界内；

➤ 有外部接口（类中的 public 成员）

- ❑ 此对象利用它与其他对象发生关联，进行操作
- ❑ 向对象发送消息：调用对象的 public 成员函数

➤ 有特定的数据保护或访问权限

- ❑ 例如：类中的 private 成员，在对象外部不允许访问或修改
- ❑ 保护内部细节，内部实现的变化不会影响外部对对象的访问

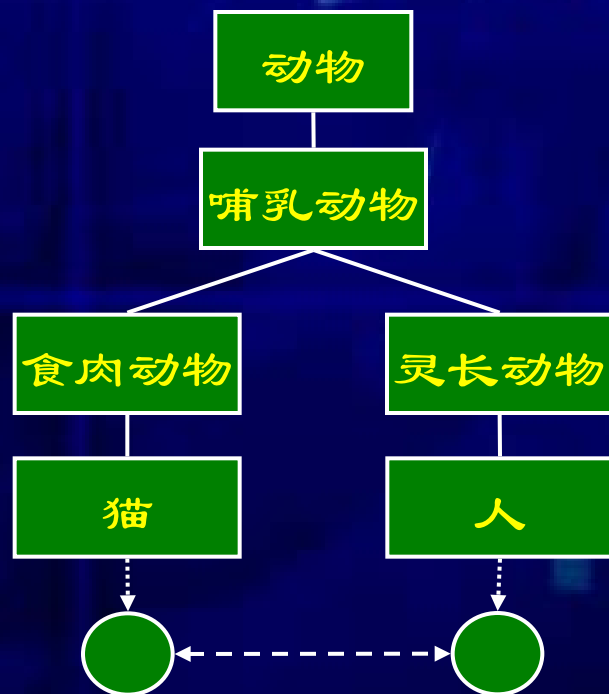
OOP 基本手段——继承与派生

➤ 在 C++ 中，支持分类层次的一种机制

- 允许程序员定义一个新的类，新类在原有类属性和行为的基础上，定义更具体、更详细的属性和行为

➤ 现实意义：

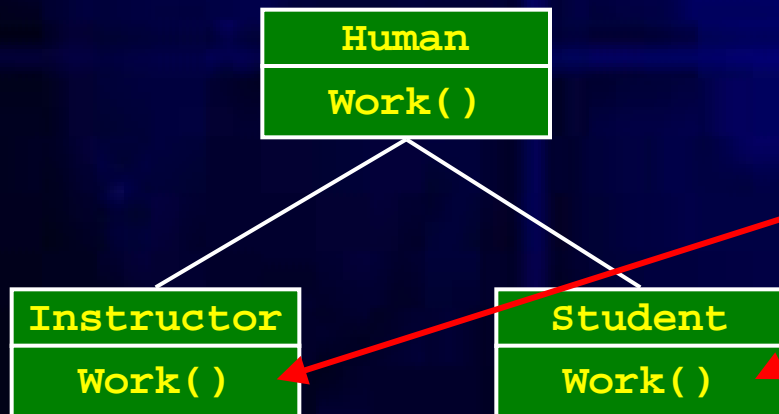
- 现实世界中对象并不是孤立的，而是相互关联的（横向）
- 而每一个对象所属的类在纵向上体现为从属或继承关系，这为 OOP 的继承 提供现实基础
- 通过继承可以实现对现有软件的重用、扩展
- 继承通过类的派生来实现



OOP 基本手段——多态性

➤ 在类的派生过程中，允许不同的派生类对同一个操作具有不同的行为实现

- ❑ 多态的体现：不同的派生类中，对同一成员函数采用不同的实现方式，在调用时总是能够保证正确的方法被调用
- ❑ 目的：行为与标识统一
- ❑ 实现：虚函数与重写（override）函数



```
Instructor A;  
Student B;  
Human *p;  
  
p = &A;  
p->Work();  
p = &B;  
p->Work();
```

C++ 类与对象

C++ 类

- 类是具有相同属性和行为的一组对象的集合，它为属于该类的全部对象提供了统一的抽象描述，其内部包括属性和行为两个主要部分。
- 利用类可以实现数据的封装、隐藏、继承与派生。
- 利用类易于编写大型复杂程序，其模块化程度比C中采用函数更高。

类的声明形式

类是一种用户自定义类型，与 struct 类似

声明形式：

```
class 类名称
{
    public:
        公有成员（外部接口）
    private:
        私有成员
    protected:
        保护型成员
};
```


类成员的访问类型

➤ public

- 声明后面的成员变量或成员函数能够被外部的函数或对象直接访问，用于定义类的外部接口

➤ private

- 声明后面的成员变量或成员函数只允许在本类对象的成员函数中被访问，不允许外部访问
- 用于定义类的内部数据和代码实现
- 省略情况下，若前面没有 public/private/protected 关键字声明，class 中的成员都是 private 类型

➤ protected

- 声明后面的成员变量或成员函数只允许在本类或其派生类对象的成员函数中被访问，不允许外部访问

类定义的例子

```
class Clock
{
    public:
        void SetTime(int NewHour, int NewMin,int NewSec);
        void ShowTime();
    private:
        int Hour, Minute, Second;
};

void Clock::SetTime(int NewHour, int NewMin, int NewSec)
{
    Hour    = NewHour;
    Minute  = NewMin;
    Second  = NewSec;
}

void Clock::ShowTime()
{
    cout << Hour << ":" << Minute << ":" << Second;
}
```

类成员的定义

➤ 数据成员（属性、成员变量）

- ❑ 数据与一般的变量声明相同，但需要放在类的声明体中

➤ 成员函数

- ❑ 在类说明中给出原型，在类说明以外给出函数的实现
- ❑ 在定义成员函数时，需要在函数名前使用类名加以限定
`Clock::SetTime`
- ❑ 也可直接在类说明中给出函数体，形成内联（inline）成员函数
- ❑ 允许声明重载函数和带默认形参值的函数

内联成员函数

➤ 为了提高运行时的效率，对于较简单的成员函数可以声明为内联形式。

- ❑ 将函数体放在类的声明中
- ❑ 使用inline关键字

```
class Point
{
    public:
        void Init(int initX,int initY)
        {
            X = initX;
            Y = initY;
        };
        int GetX() {return X;};
        int GetY() {return Y;};
    private:
        int X,Y;
};
```

内联成员函数

```
class Point
{
public:
    void Init(int initX,int initY);
    int GetX();
    int GetY();
private:
    int X,Y;
};
inline void Point::Init(int initX,int initY)
{
    X=initX;
    Y=initY;
}
inline int Point::GetX()
{
    return X;
}
inline int Point::GetY()
{
    return Y;
}
```

class 与 struct

➤ 在 C++ 中，class 与 struct 作用相同，差别仅在于：

- ❑ 不加访问类型限制（public/private/protected）时，class 成员的缺省的访问限制都是 private
- ❑ 不加访问类型限制（public/private/protected）时，struct 成员的缺省的访问限制都是 public

C++对象

➤ 类的对象是该类的某一特定实例，是类型为该类的一个变量

➤ 声明形式：

类名 对象名；

➤ 例：

Clock myClock;

对象成员的访问方式

➤ 对对象成员的访问，与访问结构成员类似

□ 使用“对象名.成员名”方式访问

MyClock.ShowTime(); // 访问成员函数

MyClock.Hour = 12; // 访问成员变量，注意访问限制

□ 注意：

⊗ 从类的外部访问该类的对象时，需要受到访问类型（public/private/protected）的限制

⊗ 在同一个对象的成员函数中访问本对象的成员，可以省略“对象名.”，直接使用成员名

类成员访问

```
class Clock
{
    public:
        void SetTime(int NewHour, int NewMin,int NewSec);
        void SetTime(Clock OtherClock);
        void ShowTime();
    private:
        int Hour, Minute, Second;
};

...
void Clock::SetTime(Clock OtherClock)
{
    Hour    = OtherClock.Hour;
    Minute  = OtherClock.Minute;
    Second  = OtherClock.Second;
}

main()
{
    Clock Clock1, Clock2;
    Clock1.Hour = 12;
    Clock1.SetTime(12,0,0);
    Clock2.SetTime(Clock2);
}
```



C++类中的自引用

➤ 在类的成员函数的上下文中，存在一个指向被调用的对象的指针（用关键字 `this` 来指出）

- ❑ 调用同一类的不同对象的成员函数时，其上下文的 `this` 指针不同，分别指向不同对象
- ❑ 尽管这些对象采用相同的成员函数代码，但实际操作是不同的对象成员的数据成员
- ❑ `this` 是类成员函数的一个
隐含参数
(除了静态成员函数外)

```
void CCounter::Increase()  
{  
    iCounterValue ++;  
    // 等同于  
    // this->iCounterValue ++;  
}  
...
```

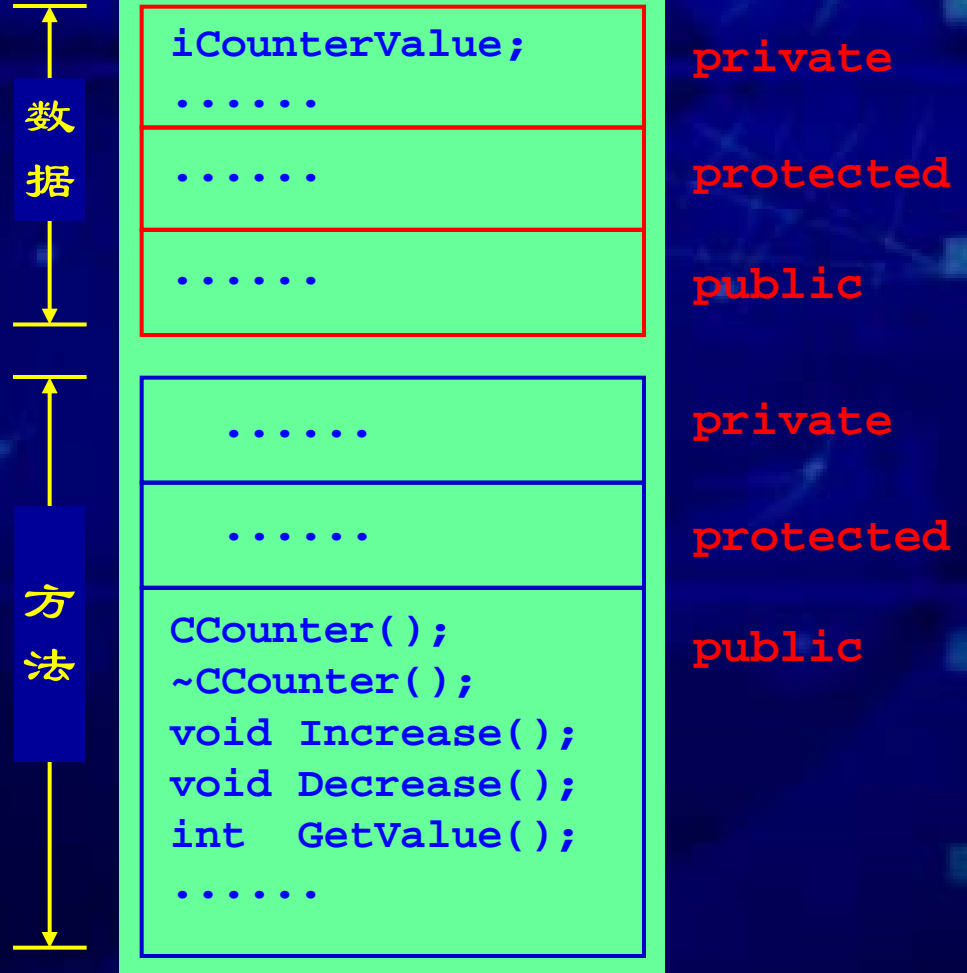
C++ 类的概念结构

➤ C++ 类定义抽象数据结构

```
class CCounter
{
private:
    int  iCounterValue;
    ...
protected:
    ...
public:
    CCounter();
    ~Ccounter();

    void Increase();
    void Decrease();
    int  GetValue();
    ...
};
```

```
CCounter counter;
```



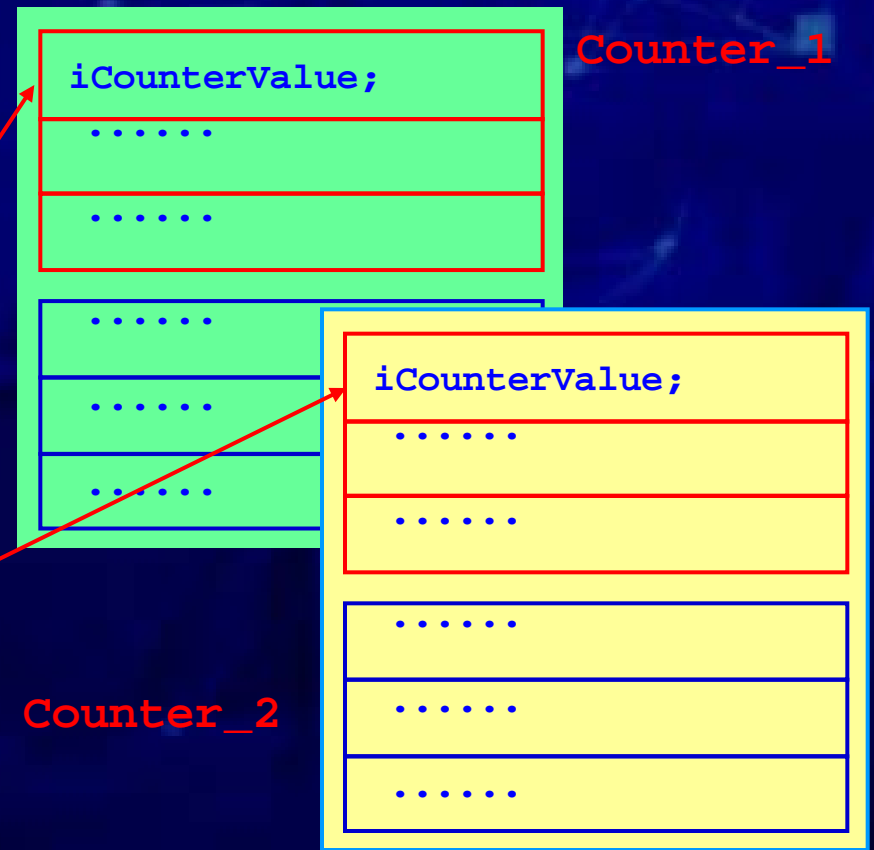
C++类的概念结构

➤ 属于一个类的所有实例（对象）具有相同的结构

□ 当调用一个对象的方法（向该对象发送消息）时，其操作（访问）的目标是本对象中的数据成员，而不对其他对象的数据成员进行操作

☒ 这种方式可以防止程序中对非相关数据的意外修改和访问，提高程序的安全性和健壮性

```
void CCounter::Increase()  
{  
    iCounterValue ++;  
}  
...  
main()  
{  
    CCounter counter_1;  
    CCounter counter_2;  
    ...  
    counter_1.Increase();  
    counter_2.Increase();  
    ...  
}
```



动态存储分配

C++中引入的新算符

new/delete



动态申请内存操作符 new

➤ 语法:

new 类型名T (初值列表)

➤ 功能

- ❑ 在程序执行期间，动态申请一块用于存放类型T的对象的内存空间，并依初值列表赋以初值
- ❑ 动态内存空间是由操作系统管理的，可供所有程序共享申请的内存空间，称为堆 (heap)
- ❑ new 使程序员不必调用库函数，如 malloc 等

➤ 结果值

- ❑ 成功：T类型的指针，指向新分配的内存
- ❑ 失败：0 (NULL)

释放内存操作符 delete

➤ 语法:

delete 指针表达式

➤ 功能:

- ❑ 释放指针表达式所指向的动态内存空间，归还给操作系统
- ❑ new 使程序员不必要调用库函数，如 free 等

例：申请和释放内存

```
int *p;  
p = new int;           // 申请内存存放一个 int 类型, 地址返回到 p 中  
delete p;              // 释放 p 指向的内存空间 (一个 int 类型)  
p = new int(2);        // 申请内存, 初始化为 2  
delete p;  
  
p = new int[100];      // 申请一个 int 类型的数组, 长度为 100, 返回数组首  
    地址  
delete [] p;           // 释放数组
```


类的构造函数

➤ `int a=0;` 或 `int a;`

```
int a;  
int function()  
{  
    int b;  
    a=b;        // a=??  
}
```

➤ 如何为自定义的类对象实现像内建数据类型的初始化功能而且可以避免忘记赋初值发生错误？

➤ 构造函数的作用

- ❑ 在类的对象被创建的时候，对被创建的对象的数据成员的值进行特定的初始化，或者说将对象初始化为一个特定的状态
- ❑ 在对象创建时由系统自动调用类的构造函数创建对象
- ❑ 如果未声明构造函数，则系统自动产生出一个类的默认构造函数
- ❑ 构造函数也允许为内联函数、重载函数、带默认形参值的函数
- ❑ 构造函数没有返回值

✉ 也不是返回 `void`

构造函数举例

```
class Clock
{
public:
    Clock (int Hour, int Min, int Sec);           // 构造函数, 与类名相同
    ...
private:
    int Hour, Minute, Second;
};

Clock::Clock(int Hour, int Min, int Sec)
{
    this->Hour = Hour;
    Minute    = Min;
    Second    = Sec;
}

int main()
{
    Clock  MyClock(0,0,0);                       // 调用构造函数, 初始化 MyClock
    MyClock.ShowTime();
    return 0;
}
```

构造函数的重载

```
class Clock
{
public:
    Clock();
    Clock(int Hour, int Min, int Sec);
    ...
};

Clock::Clock()
{
    Hour = Minute = Second = 0;
}

Clock::Clock(int Hour, int Min, int Sec)
{
    ...
}

int main()
{
    Clock MyClock(0,0,0);
    Clock MyClock2;
    return 0;
}
```

// 无参数的构造函数
// 构造函数

The diagram consists of red lines and arrows. A vertical line on the right side of the code block has two horizontal arrows pointing left. The top arrow points to the `Clock()` constructor declaration in the `public` section of the `Clock` class. The bottom arrow points to the `Clock(int Hour, int Min, int Sec)` constructor declaration. From the bottom of this vertical line, a horizontal line extends to the left, then turns down to become a vertical line. This vertical line has two horizontal arrows pointing left. The top arrow points to the `Clock MyClock(0,0,0);` line in the `main` function. The bottom arrow points to the `Clock MyClock2;` line in the `main` function.

拷贝构造函数

- `int a=0; int b=a;` 对自定义的类对象如何实现第二条语句？
- 拷贝构造函数是一种特殊的构造函数，其形参为本类的对象引用。
- 当用类的一个对象去初始化（创建）该类的另一个对象时系统自动调用拷贝构造函数实现拷贝赋值。

```
class 类名
{
    public :
        类名 (类名 &对象名) ;    //拷贝构造函数
    ...
};
```

拷贝构造函数

```
#include <iostream>
using namespace std;
class Point
{
    public:
        Point(int xx=0,int yy=0)
            {X=xx; Y=yy;};
        Point(Point& p);
        int GetX() {return X;};
        int GetY() {return Y;};
    private:
        int X,Y;
};

Point::Point (Point& p)
{
    X=p.X;
    Y=p.Y;
    cout << "Copy Constructor"
         << endl;
}
```

```
void func1(Point p)
{
    cout << p.GetX() <<endl;
}

Point func2()
{
    Point A(1,2);
    return A; //调用拷贝构造函数
}

int main()
{
    Point A(1,2);
    Point B(A); //拷贝构造函数被调用
                //或 Point B=A;

    func1(A);    // 调用拷贝构造函数

    Point C = func2();
}
```

拷贝构造函数的调用

- 当用类的一个对象去初始化（创建）该类的另一个对象时，拷贝构造函数被调用，初始化新对象
- 若函数的参数是类对象，调用函数时，系统自动调用拷贝构造函数，将实参赋对象的值复制一份，作为函数调用的形参
- 当函数的返回值是类对象时，用 return 语句返回对象时，系统自动调用拷贝构造函数，复制一份对象的拷贝，作为返回值
- 缺省拷贝构造函数：
 - ❑ 如果程序员没有为类声明拷贝初始化构造函数，则编译器自己生成一个拷贝构造函数
 - ❑ 这个构造函数执行的功能是：用作为初始值的对象的每个数据成员的值，初始化将要建立的对象的对数据成员
 - ❑ 在某些情况下，缺省拷贝构造函数可能是不正确的，例如存在指针的情况

析构函数

- 在C语言编程时，是不是经常有动态申请内存后、忘记释放，打开文件、忘记关闭的情况？尤其是第三方使用时？如何避免？
- 完成对象被删除前的一些清理工作
 - ❑ 关闭对象在生存期间打开的文件
 - ❑ 释放动态申请的资源：内存、...
- 在对象的生存期结束的时刻系统自动调用它，然后系统才能销毁此对象所占用的空间。
- 如果程序中未声明析构函数，编译器将自动产生一个默认的析构函数。
- 形式：~类名（）；

构造函数和析构函数举例

```
#include <iostream>
using namespace std;
#define BUF_SIZE 512
class DataBuf
{
private:
    char * Buf;
public:
    DataBuf();
    DataBuf(DataBuf & DF);
    ~ DataBuf();

    //... 其它函数
    void ReadBuf();
    ...
};
```

```
DataBuf :: DataBuf( )
{
    // 构造函数
    Buf = new char [BUF_SIZE];
}
DataBuf :: DataBuf(DataBuf & DF)
{
    // 拷贝构造函数
    Buf = new char [BUF_SIZE];
}
DataBuf :: ~ DataBuf()
{
    // 析构函数
    delete [] Buf;
}
```


类的组合

- 类中的成员是另一个类的对象。
- 用于在已有的抽象的基础上实现更复杂的抽象。

组合类例子

```
class Point
{
    private:
        float x,y;    //点的坐标
    public:
        //构造函数
        Point();
        Point(float h,float v);

        float GetX(void);
        float GetY(void);
        void Draw(void); //画点
};

//...函数的实现
```

```
class Line
{
    private:
        Point p1,p2; //两个端点
    public:
        //构造函数
        Line(Point a,Point b);

        Void Draw(void); //画线段
};

//...函数的实现略
```

组合类的构造函数

➤ 原则：不仅要负责对本类中的基本类型成员数据赋初值，也要对对象成员初始化。

➤ 声明形式：

类名::类名(参数表)

: 成员(初始化参数), 成员(初始化参数),

{

本类初始化

}

需要使用参数进行初始化的成员
(包括对象成员) 在列表中列出;
没有列出的对象成员, 则调用其
默认的构造函数

组合类构造函数

```
class Point
{
    private:
        float x,y;    //点的坐标
    public:
        //构造函数
        Point();
        Point(Point& p);
        Point(float h,float v);

        float GetX(void);
        float GetY(void);
        void Draw(void); //画点
};
```

//...函数的实现

```
class Line
{
    private:
        Point p1,p2; //两个端点
        int width; //线宽
    public:
        //构造函数
        Line();
        Line(int x1, int y1);
        Line(int x1, int y1,
              int x2, int y2
              int w);
        Line(Point a,Point b,int w);
};
```

组合类构造函数

```
Line::Line()
```

```
{  
    ...  
}
```

对 p1、p2 调用默认构造函数 Point()

```
Line::Line(int x1, int y1) : p1(x1,y1)
```

```
{  
    ...  
}
```

对 p1 调用构造函数 Point(h, v),
对 p2 调用默认构造函数 Point()

```
Line::Line(int x1, int y1, int x2, int y2, int w)
```

```
    :p1(x1,y1), p2(x2,y2), width(w)
```

```
{  
    ...  
}
```

用 w 初始化成员 width

对 p1、p2 调用构造函数 Point(h, v)

```
Line::Line(Point a,Point b,int w):p1(a), p2(b), width(w)
```

```
{  
    ...  
}
```

对 p1、p2 调用构造函数 Point(p)

组合类的构造函数调用

- 构造函数调用顺序：先调用内嵌对象的构造函数（按内嵌时的声明顺序，先声明者先构造）。然后执行本类的构造函数。（析构函数的调用顺序相反）
- 若调用默认构造函数（即无形参的），则内嵌对象的初始化也将调用相应的默认构造函数
 - 这时，若成员对象类没有定义默认构造函数，就会出错

总结：构造函数与析构函数

➤ 构造函数

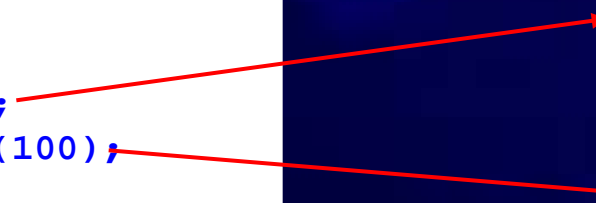
- ❑ 在类的对象被创建时，将会自动调用类的构造函数对类的对象进行初始化
- ❑ 构造函数没有返回值
- ❑ 构造函数可以重载
- ❑ 构造函数可以包含参数

```
main()
{
    ...
    CCounter counter_1;
    CCounter counter_2(100);
    ...
}
```

```
class CCounter
{
private:
    int iCounterValue;
    ...
public:
    CCounter();
    CCounter(int iStart);
    ...
};

CCounter::CCounter()
{
    iCounterValue = 0;
}

CCounter::CCounter(int iStart)
{
    iCounterValue = iStart;
}
```



总结：构造函数与析构函数

➤ 构造函数中数据成员的初始化

❑ 情况1：成员为基本数据类型时

❑ 情况2：成员为对象时

➤ 使用初始化列表

```
class CDbCounter
{
private:
    int iMaxCounterValue;
    CCounter counter_1;
    CCourter counter_2;
    ...
public:
    CDbCounter();
    CDbCounter(int iStart_1,
                int iStart_2,
                int iMaxCntVal);
    ...
};
```

```
CDblCounter::CDblCounter()
{
    iMaxCounterValue = 1000;
    ...
}

CDblCounter::CDblCounter(int iStart_1,
                          int iStart_2,int iMaxCntVal)
: counter_1(iStart_1),
  counter_2(iStart_2),
  iMaxCounterValue(iMaxCntVal)
{
    ...
    // iMaxCounterValue = iMaxCntVal;
    ...
}
```


构造函数与析构函数

➤ 构造函数什么时候被调用？

❑ 全局对象变量

在开始执行 main() 函数代码前，
全局对象变量的构造函数被调用

❑ 局部（自动）对象变量

在定义局部对象变量时，对象的
构造函数被调用

❑ 动态创建的对象

对象申请动态空间成功后被调用

❑ 对象的成员

✉ 在容器对象被创建时，其构
造函数被调用时被执行

对象数组（包括动态的）通常不
能使用带有参数的构造函数

```
class CCounter
{
    ...
};


CCounter g_counter;

main()
{
    CCounter counter_1(200);
    CCounter counter_2;
    CCounter Counter_List[16];

    CCounter * pCounter;
    pCounter = new Ccounter(100);

    Ccounter * pCounterArray;
    pCounterArray = new Ccounter[16];

    ...
}
```



构造函数与析构函数

➤ 析构函数

□ 在对象被销毁时被调用，用于销毁前的准备工作

✉ 如释放对象占用的资源（内存空间、设备...）

```
#define MAX_STR_LEN    1024
class CMyString
{
private:
    int    iStrLen;
    char * pStrBuf;
    ...
public:
    CMyString();
    CMyString(char * pStr);
    ~CMyString();

    CopyString(char * pSource);
};
```

```
CMyString::CMyString():iStrLen(0)
{
    pStrLen = new char[MAX_STR_LEN];
    pStrLen[0] = 0;
}
CMyString::CMyString(char * pStr)
{
    pStrLen = new char[MAX_STR_LEN];
    CopyString(pStr);
}
CMyString::CopyString(char * pSource)
{
    ::strcpy(pStrBuf, pSource);
    iStrLen = ::strlen(pSource);
}
CMyString::~~CMyString()
{
    delete [] pStrLen;
}
```

构造函数与析构函数

➤ 析构函数什么时候被调用？

❑ 全局对象变量

在main() 函数退出前，全局对象变量的析构函数被调用

❑ 局部（自动）对象变量

在局部对象变量的生命期结束之前，对象的析构函数被调用

❑ 动态创建的对象

在释放对象的动态空间之前，其析构函数被调用

❑ 成员对象

☒ 容器对象被销毁时，成员对象的析构函数后被调用

```
class CCounter
{
    ...
};

CCounter g_counter;

main()
{
    ...
}
```

g_counter 的析构
函数被调用

构造函数与析构函数

➤ 析构函数什么时候被调用？

❑ 全局对象变量

在main() 函数退出前，全局对象变量的析构函数被调用

❑ 局部（自动）对象变量

在局部对象变量的生命期结束之前，对象的析构函数被调用

❑ 动态创建的对象

在释放对象的动态空间之前，其析构函数被调用

❑ 成员对象

✉ 容器对象被销毁时，成员对象的析构函数后被调用

析构函数被调用

```
void func()
{
    CCounter counter;
    ...
}

main()
{
    if (...)
    {
        CCounter counter_1(200);
        ...
    }

    CCounter * pCounter;
    pCounter = new Ccounter(100);
    Ccounter * pCounterArray;
    pCounterArray = new Ccounter[16];

    delete pCounter;
    delete [] pCounterArray;
}
```

Counter 的析构函数被调用

Counter_1的析构函数被调用

前向引用声明

- 类应该先声明，后使用
- 如果需要在某个类的声明之前，引用该类，则应进行前向引用声明。
- 前向引用声明只为程序引入一个标识符，但具体声明在其它地方。

前向引用声明举例

```
class B;    //前向引用声明

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

class B
{   public:
    void g(A a);
};
```

前向引用声明注意事项

- 使用前向引用声明虽然可以解决一些问题，但它并不是万能的。需要注意的是，尽管使用了前向引用声明，但是在提供一个完整的类声明之前，不能定义该类的对象，也不能在内联成员函数中使用该类的对象。请看下面的程序段：

```
class Fred;           //前向引用声明
class Barney
{
    Fred x;           //错误：类Fred的声明尚不完善
};
class Fred
{
    Barney y;
};
```

前向引用声明注意事项

```
class Fred;    //前向引用声明
```

```
class Barney
{
    public:
        void method()
        {
            x->youDo();    //错误：Fred类的对象在定义之前被使用
        };
    private:
        Fred* x;    //正确，经过前向引用声明，可以声明Fred类的对象指针
};
```

```
class Fred
{
    public:
        void youDo();
    private:
        Barney* y;
};
```

使用前向引用声明时，只能使用被声明的符号，而不能涉及该类的任何实现细节，因为该类还没有定义

流类库与输入/输出

北京邮电大学

网络与交换技术国家重点实验室

宽带网研究中心

I/O流的概念

- 当程序与外界环境进行信息交换时，存在着两个对象，一个是程序中的流对象，另一个是文件对象
- 流是一种抽象，它负责在数据的生产者和数据的消费者之间建立联系，并管理数据的流动
- 程序建立一个流对象，并指定这个流对象与某个文件对象建立连接，程序操作流对象，流对象通过文件系统对所连接的文件对象产生作用
- 读操作在流数据抽象中被称为（从流中）提取，写操作被称为（向流中）插入

输出流

➤ 重要的输出流类型 (C++标准库中预定义)

- ❑ ostream

- ❑ ofstream

ostream

➤ 预先定义了如下的输出流对象:

- ❑ cout 标准输出
- ❑ cerr 标准错误输出，没有缓冲，发送给它的内容立即被输出。
- ❑ clog 类似于cerr，但是有缓冲，缓冲区满时被输出。

插入运算符 (<<)

- 插入 (<<) 运算符对所有标准C++数据类型预先进行了操作符的重载
- 用于将数据转换成输出字节串的形式， 传送到一个输出流对象

控制输出格式

➤ 控制输出宽度

- ❑ 为了调整输出，可以通过在流中放入setw操纵符或调用width成员函数为每个项指定输出宽度。

➤ 例： 使用 width 控制输出宽度

```
#include <iostream>
using namespace std;
int main()
{
    double values[] = {1.23, 35.36, 653.7, 4358.24};
    for (int i=0;i<4;i++)
    {
        cout.width(10);
        cout << values[i] <<' \n' ;
    }
}
```

输出结果：

1.23
35.36
653.7
4358.24

例：使用*填充

```
#include <iostream>
using namespace std;
int main()
{
    double values[]={1.23,35.36,653.7,4358.24};
    for(int i=0; i<4; i++)
    {
        cout.width(10);
        cout.fill('*');
        cout<<values[i]<<'\n';
    }
}
```

输出结果：

*****1.23

*****35.36

*****653.7

***4358.24

例：使用setw指定宽度

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    double values[]={1.23, 35.36, 653.7, 4358.24};
    char *names={"Zoot", "Jimmy", "Al", "Stan"};
    for (int i=0; i<4; i++)
    {
        cout << setw(6) << names[i];
        cout << setw(10) << values[i] << endl;
    }
}
```

输出结果：

Zoot	1.23
Jimmy	35.36
Al	653.7
Stan	4358.24

ofstream

- ofstream 类支持磁盘文件输出
- 如果在构造函数中指定一个文件名，当构造这个文件时该文件是自动打开的

```
❑ ofstream myFile("filename", iosmode);
```

- 可以在调用默认构造函数之后使用open成员函数打开文件

```
ofstream myFile; //声明一个静态输出文件流对象  
myFile.open("filename", iosmode);  
//打开文件，使流对象与文件建立联系
```

ofstream 成员函数

➤ open函数

把流与一个特定的磁盘文件关联起来。
需要指定打开模式。

➤ put函数

把一个字符写到输出流中。

➤ write函数

把内存中的一块内容写到一个输出文件流中

➤ seekp和tellp函数

操作文件流的内部指针

➤ close函数

关闭与一个输出文件流关联的磁盘文件

➤ 错误处理函数

在写到一个流时进行错误处理

➤ << 插入操作符

例：文件输出

```
#include <fstream>
using namespace std;
struct Date
{
    int mo,da,yr;
};
int main()
{
    Date dt = {6,10,92};
    ofstream tfile("date.dat",ios::binary);
    tfile.write((char *) &dt,sizeof dt);
    tfile.close();

    ofstream txtfile("Data.txt");
    txtfile << dt.mo << dt.da << dt.yr << endl;
    txtfile.close();
}
```

输入流

➤ 重要的输入流类型：

- ❑ istream类最适合用于顺序文本模式输入。

cin 是预定义的标准输入对象

- ❑ ifstream类支持磁盘文件输入

➤ 提取运算符 (>>)

- ❑ 对于所有标准C++数据类型预定义了重载

- ❑ 是从一个输入流对象获取数据最容易的方法

ifstream

- 如果在构造函数中指定一个文件名，在构造该对象时该文件便自动打开。

```
ifstream myFile("filename",iosmode);
```

- 在调用缺省构造函数之后使用open函数来打开文件。

```
ifstream myFile; //建立一个文件流对象  
myFile.open("filename",iosmode);  
//打开文件"filename"
```

- 成员函数

```
open () get () getline () read () seekg () tellg ()  
close ()
```

例：文件输入

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{ char ch;
  ifstream tfile("payroll", ios::binary);
  if(tfile)
  { tfile.seekg(8);
    tfile.get(ch);
    while(tfile.good())
    {cout<<ch;  tfile.get(ch);
      }
  } else
    { cout<<"ERROR: Cannot open file 'payroll'."<<endl; }
  tfile.close();
}
```

C++ 程序设计与结构

北京邮电大学

网络与交换技术国家重点实验室

宽带网研究中心

主要内容

- 作用域与可见性
- 对象的生存期
- 数据与函数
- 静态成员
- 友元
- 常量类型

标识符的作用域

➤ 最小作用域优先原则（与 C 语言一样）

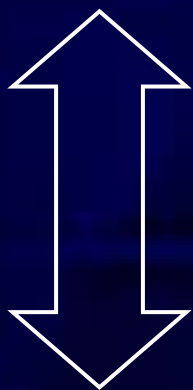
- 在使用一个标识符时，若不限定，编译程序将其理解为在有效作用域内同名标识符中，作用域最小的一个

函数原型的作用域

函数原型中的参数，其作用域始于
"(", 结束于")"。

➤ 例如，设有下列原型声明：

```
double Area(double radius);
```



radius 的作用域仅在于此，不能用于程序正文其它地方，因而可有可无。

```
double Area(double);
```

程序块作用域

- 在块中声明的标识符，其作用域自声明处起，限于块中（与C语言相同），例如：

```
void fun(int a)
{
    int b(a);
    cin>>b;

    if (b>0)
    {
        int b;
        int c;
        b = 1
        .....
    }
}
```

b c 的作用域

b 的作用域

类作用域

➤ 在 C++ 中引入，适用于 class 和 struct 定义类

➤ 类作用域作用于特定的类的成员

□ 成员变量和成员函数

➤ 类 X 的成员 M 具有类作用域：

□ 在 X 的成员函数中可直接访问

✉ 如果函数中声明了与 M 同名的局部标识符，那么需要加限定符

□ 对外部，在对象 X 的作用域内

✉ 通过表达式访问 public 成员

x.M x.X::M

ptr->M ptr->X::M

✉ 成员的作用域与对象变量 x 的作用域相同

```
void f2();  
class X  
{  
    public:  
        int M;  
        void f1();  
        void f2();  
    public:  
        void abc(void)  
        {  
            M = 1;  
            f1();  
            {  
                int M;  
                M = 2;  
                X::M = 3; //或 this->M  
                f2();      // X::f2()  
                ::f2();    // 全局 f2()  
            }  
        }  
};
```

文件作用域

➤ 文件作用域

- ❑ 不在前述各个作用域中出现的标识符的声明
- ❑ 标识符的作用域开始于声明点，结束于文件尾

```
#include <iostream>
```

```
void func()
```

```
{
```

```
    i++;
```

```
}
```

```
int i = 0;
```

```
void func2()
```

```
{
```

```
    i++;
```

```
}
```

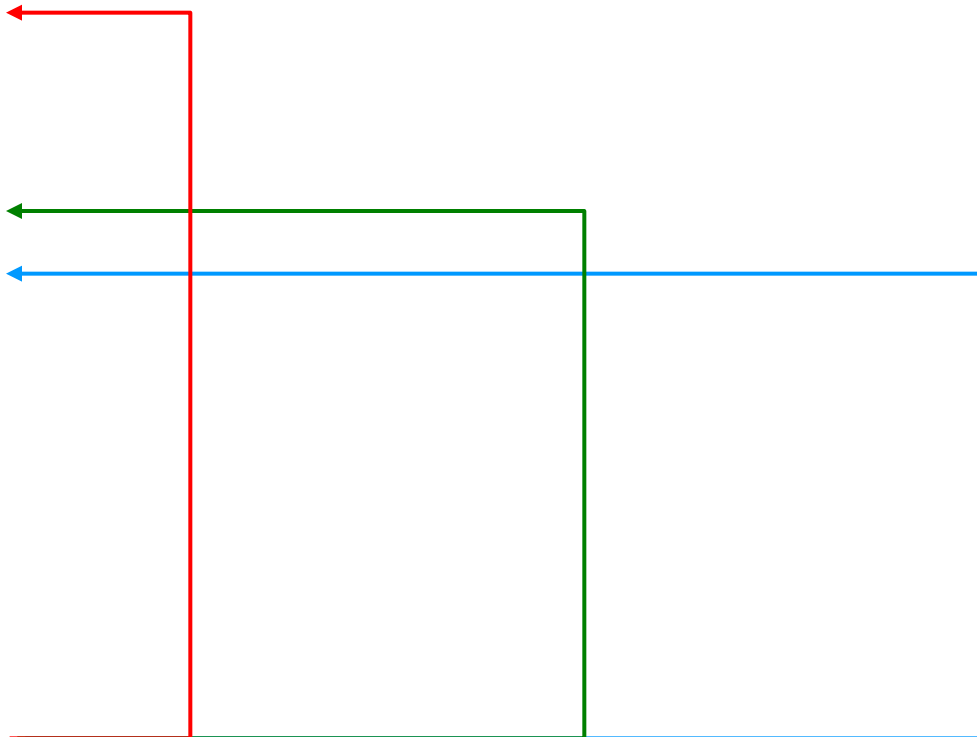
```
int main()
```

```
{
```

```
    i++;
```

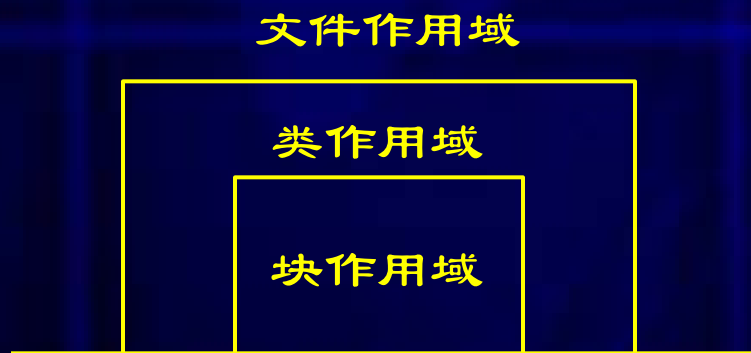
```
    ...
```

```
}
```



可见性

- 可见性是从对标识符的引用的角度来谈的概念
- 可见性表示从内层作用域向外层作用域“看”时能看见什么。
- 如果标识在某处可见，则就可以在该处引用此标识符。



可见性

- 标识符应声明在先，引用在后。
- 如果某个标识符在外层中声明，且在内层中没有同一标识符的声明，则该标识符在内层可见。
- 对于两个嵌套的作用域，如果在内层作用域内声明了与外层作用域中同名的标识符，则外层作用域的标识符在内层不可见。

同一作用域中的同名标识符

- 在同一作用域内的对象名、函数名、枚举常量名会隐藏同名的类名或枚举类型名。
- 重载的函数可以有相同的函数名。

作用域

```
#include<iostream>
using namespace std;
int i=1;      //文件作用域

int main()
{
    int i=5;      //块作用域, 隐藏了全局变量i
    cout << "i=" << i; //输出5

    {
        int i=7;      //块作用域, 隐藏了外层的i
        cout <<"i=" << i << endl; //输出7
    }

    cout <<"i=" << ::i << endl; //输出1,全局变量
    ...
}
```

对象的生存期

- 与基本类型的变量的生存期是一样的
 - ❑ 基本类型（如 int 等）的变量在 C++ 中也认为是一种对象
- 对象从创建到撤销的这段时间就是它的生存期
- 在对象生存期内，可以对对象进行操作(若其可见)
 - ❑ 如修改、更新成员变量（若允许）
 - ❑ 调用成员函数
- 没有操作时，对象将一直保持它的状态值

静态生存期

➤ 这种生存期与程序的运行期相同

□ 尽管与程序生命周期相同，但并非在程序的任何地方都是可见的

➤ 在文件作用域中声明的对象（全局变量）具有这种生存期

➤ 在函数内部声明静态生存期对象，要冠以关键字 `static`

➤ 对象的生存期结束时（程序结束时），将调用该对象的析构函数（若存在）

静态生存期

```
#include<iostream>
using namespace std;

int i;                                //文件作用域，具有静态生存期

void foo(void)
{
    static int j=0;  //块作用域，具有静态生存期
    j++;
}
int main()
{
    i++;
    foo();
    return 0;
}
```

动态生存期

► 动态生存期的对象

- ❑ 在块作用域中声明的，用auto修饰、或者无修饰的对象
- ❑ 也称局部生存期对象
- ❑ 生存期开始于程序执行到声明点时，结束于标识符的作用域结束处
- ❑ 动态对象的生存期结束时，将调用该对象的析构函数（若存在）

例：对象的生存期与可见性

```
#include <iostream>
using namespace std;

int i=1;                // i 为全局变量，具有静态生存期
int main()
{
    static int a=0;      // 静态局部变量，有全局寿命，局部可见。
    int b=-10;           // b为局部变量，具有动态生存期。

    i=i+10;
    void other(void);
    other();
    ...
}

void other(void)
{
    static int b, a=2;    // a,b为静态变量，全局寿命，局部可见，第一次进入函数时初始化
    int c=10;            // c 为局部变量，具有动态生存期，每次进入函数时都初始化。

    b = (a+=2);
    ...
}
```

例: 对象的生存期与可见性


```
#include<iostream>
using namespace std;

class Clock                //时钟类声明
{
    ...                    // Clock类 的定义
};

Clock globClock;          //声明对象globClock, 具有静态生存期, 文件作用域

void func()
{
    Clock myClock(globClock);    //创建具有块作用域的对象 myClock
    myClock.ShowTime();          //引用具有块作用域的对象
}

int main()
{
    globClock.SetTime(8,30,30);  //引用全局作用域的对象:
    func();
}
```



MyClock 在生存期结束时被销毁

数据与函数的关系

➤ 若需要处理的数据存储在局部对象中，通过参数传递实现共享

□ 在函数（过程）之间的传递参数，返回结果

➤ 若数据存储在全局对象中

□ 各函数（过程）对全局对象进行处理，结果也保存在全局对象中

➤ 将数据和使用数据的函数封装在类中

面向
过程

面向
对象

使用全局对象

```
#include<iostream>
using namespace std;

int global;

void f()
{
    global=5;
}

void g()
{
    cout<<global<<endl;
}

int main()
{
    f();
    g();
    return 0;
}
```

**需要处理的数据
很多的时候怎么办？**

数据与行为的封装

```
#include<iostream>
using namespace std;

class Application
{
public:
    void f();
    void g();
private:
    int global;
};

void Application::f()
{
    global=5;
}
```

```
void Application::g()
{
    cout<<global<<endl;
}

int main()
{
    Application  MyApp;

    MyApp.f();

    MyApp.g();

    return 0;
}
```

类的静态成员

➤ 静态数据成员

- ❑ 用关键字static声明
- ❑ 该类的所有对象维护该成员的一个拷贝
- ❑ 必须在类外定义和初始化，用(::)来指明所属的类。
- ❑ 不需要实例化对象就可以访问类的静态数据成员

➤ 静态成员函数

- ❑ 静态成员函数只能引用属于该类的静态数据成员或静态成员函数。
- ❑ 类外代码可以使用类名和作用域操作符来调用静态成员函数
- ❑ 不需要实例化对象就可以调用类的静态成员函数

例：静态成员

```
#include <iostream>
using namespace std;

class Point
{
private:
    int X,Y;
    static int countP;
public:
    Point(int xx=0, int yy=0)
    {
        X=xx; Y=yy; countP++;
    };
    Point(Point &p)
    {
        X=p.X; Y=p.Y; countP++;
    };
    int GetX() {return X;};
    int GetY() {return Y;};
    static int GetC()
    {
        return countP;
    };
};
```

```
int Point::countP = 0;

int main()
{
    cout << Point::GetC() << endl;

    Point A(4,5);
    cout << A.GetC() << endl;

    Point B(A);
    cout << B.GetC() << endl;

    /* 如果 countP 是 public 的,
       则下列语句也是正确的:

       cout << Point::countP << endl;
    */
}
```

例：静态成员

```
class A
{
    public:
        static void f();
        static void g();
        void h();
    private:
        static int s;
        int x;
};

void A::f()
{
    cout << s;
    g();

    cout << x;    // 错误，在静态函数中的只能访问静态成员变量
    h();          // 错误，在静态函数中的只能调用静态成员函数

    A a;          // a 是一个实例化的对象，
    a.h();         // 可以访问 a 的非静态成员
    ...
}
```

友元

- 友元是C++提供的一种破坏数据封装和隐藏的机制
 - ❑ 目的是为了更方便编程、提高代码效率，增加灵活性
 - ❑ 使程序员可以在封装性和快速性方面做合理折衷
 - ❑ C++ 允许将一个类或函数声明为另一个类的友元，这样在这个类或函数中就能够直接访问第二个类的隐藏信息（即 private/protected 类型的成员变量和成员函数）
- 可以使用友元函数和友元类。
- 为了确保数据的完整性，及数据封装与隐藏的原则，建议尽量不使用或少使用友元。

友元函数

➤ 友元函数

- ❑ 在类声明中由关键字 friend 修饰说明的非成员函数
- ❑ 友元函数可通过对象名访问 private 和 protected 成员

```
#include <iostream>
#include <cmath>
using namespace std;

class Point
{
    friend double
    Distance(Point &a, Point &b);

public:    //外部接口
    ...
private: //私有数据成员
    int X,Y;
};

//成员函数的实现
...
```

```
double
Distance(Point &a, Point &b)
{
    double dx = a.X-b.X;
    double dy = a.Y-b.Y;
    return sqrt(dx*dx + dy*dy);
}

int main()
{
    Point p1(3.0, 5.0);
    Point p2(4.0, 6.0);
    double d = Distance(p1, p2);

    cout<< d <<endl;
    return 0;
}
```

友元类

➤ 友元类

- ❑ 所有成员函数都能访问对方类的 `private/protected` 成员
- ❑ 声明语法：在类中将友元类的类名用 `friend` 修饰

➤ 友元关系是单向的

- ❑ 如果声明B类是A类的友元，B类的成员函数就可以访问A类的私有和保护数据，但A类的成员函数却不能访问B类的私有、保护数据。

例：友元类

```
class A;
class B
{
    public:
        void Set(A& a, int i);
        void Display(A& a);
    private:
        ...
};
```

```
class A
{
    friend class B;
    private:
        void Display();
    private:
        int x;
};
```

```
void B::Set(A& a, int i)
{
    a.x=i;
}
```

```
void B::Display(A& a)
{
    a.Display();
}
```

```
class A;
class B
{
    public:
        void Set(A& a, int i);
        void Display(A& a);
    private:
        ...
};
```

```
class A
{
    friend void B::Set(A& a,int i);
    private:
        void Display();
    private:
        int x;
};
```

```
void B::Set(A& a, int i)
{
    a.x=i;
}
```

```
void B::Display(A& a)
{
    a.Display();
}
```

将 B 的一个成员
函数设为友元



常量类型

➤ 常类型的变量（对象）必须进行初始化，在程序中不能被更新，常量不能作为左值

➤ 常量引用：被引用的对象不能被更新

`const` 类型说明符 &引用名

➤ 常量对象：必须进行初始化，不能被更新

类名 `const` 对象名

➤ 常量数组：数组元素不能被更新

类型说明符 `const` 数组名[大小]...

➤ 常量指针

常量类型

```
const int x = 100;
x++;           // 错误
int array[x];  // C++中正确, C中错误
int &rx = x;    // 错误
const int &crx = x;

int a;
int &ra = a;
a = 1;
ra = 2;
cout << a << endl; // output: 2
const int &cra = a;
cra = 3; // 错误
```

```
class A
{
public:
    A(int i, int j) {x=i; y=j;}
    Set(int i, int j) {x=i; y=j;}
public:
    int x, y;
};

main()
{
    A a(3,4);
    a.Set(5,6);

    A const b(7,8);
    b.x = 9;           // 错误
    b.Set(9,10);       // 错误
    ...
}
```

常量类型作为参数

```
class A
{
    public:
        A(int i,int j)    {x=i; y=j;}
        Set(int i, int j) {x=i; y=j;}
    public:
        int x,y;
};
```

// 常量引用做形参，要求在函数中不能更新引用的对象，有助于提高安全性，防止无意的修改
// 编程中应提倡

```
void display(const A& a)
{
    cout<<a.x<<" "<<a.y<<endl;
}
```

```
main()
{
    A a(3,4);
    display(a);
    ...
}
```

const 修饰符

```
int const i=100; } 等价  
const int i=100;
```

```
int const &r = i; } 等价  
const int &r = i;
```

```
class A  
{  
    ...  
};  
const A a; } 等价  
A const a;
```

```
int a;  
int const *p = &a; } 等价  
const int *p = &a;
```

const 修饰指针

```
int a, b;  
const int *p1 = &a;           // 指针指向的内容不能修改  
int * const p2 = &a;          // 指针指向的地址不能修改  
const int * const p3 = &a;    // 指针指向的地址和内容都不能修改  
*p1 = 10;   X  
p1 = &b;  
*p2 = 10;  
p2 = &b;   X  
*p3 = 10;   X  
p3 = &b;   X  
  
char str[] = "abcdef";  
const char *s1 = str;  
char * const s2 = str;  
const char * const s3 = str;
```

用 const 修饰的对象成员

➤ 常成员函数

- ❑ 使用const关键字说明的函数

- ❑ 常成员函数不能更新对象的数据成员

 - ✉ 常成员函数中，不能调用非常量成员函数

- ❑ 常成员函数说明格式：

 - 类型说明符 函数名（参数表） const;

 - 这里，const是函数类型的一个组成部分，因此在实现部分也要带 const 关键字

- ❑ const 关键字可以被用于参与对重载函数的区分

 - ✉ 相同的函数名可以带或不带 const，用于区分两个不同函数

➤ 使用常量对象时，只能调用其常成员函数

➤ 常数据成员

- ❑ 使用 const 说明的数据成员。

例：常成员函数

```
#include<iostream>
using namespace std;
class R
{
public:
    R(int r1, int r2)
    {
        R1=r1;R2=r2;
    }
    void print();
    void print() const;
private:
    int R1,R2;
};
```

```
void R::print()
{    // 可以修改成员变量
    cout<<R1<<":"<<R2<<endl;
}
void R::print() const
{    // 不可以修改成员
    cout<<R1<<";"<<R2<<endl;
}
int main()
{
    R a(5,4);
    a.print();    //调用print()
    const R b(20,52);
    b.print();    //调用print() const
}
```


例：常数据成员

```
#include<iostream>
using namespace std;
class A
{
    public:
        A(int i);
        void print();
    private:
        const int a;
};
A::A(int i):a(i)
{
    ...
}
```

```
void A::print()
{
    cout<<a<<endl;
}
int main()
{
    A a1(100);
    A a2(0);
    a1.print();    // output: 100
    a2.print();    // output: 0
}
```

在构造函数中，对常数据成员初始化，此后常量数据成员的值不能被修改

编译预处理命令

➤ #include 包含指令

- ❑ 将一个源文件嵌入到当前源文件中该点处。

- ❑ #include<文件名>

 - ☒ 按标准方式搜索，文件位于C++系统目录的include子目录下

- ❑ #include"文件名"

 - ☒ 首先在当前目录中搜索，若没有，再按标准方式搜索。

➤ #define 宏定义指令

- ❑ 定义符号常量，很多情况下已被const定义语句取代。

- ❑ 定义带参数宏，已被内联函数取代。

➤ #undef

- ❑ 删除由#define定义的宏，使之不再起作用。

条件编译指令 #if 和 #endif

#if 常量表达式1 //当“常量表达式”非零时编译

程序正文

#elif 常量表达式2

程序正文

#else

程序正文

#endif

条件编译指令

`#ifdef` 标识符 或者 `#ifndef` 标识符

程序段1

`#else`

程序段2

`#endif`

如果“标识符”已经用`#defined`定义过（或者未定义），
则编译程序段1，否则编译程序段2

对象数组、指针

北京邮电大学

网络与交换技术国家重点实验室

宽带网研究中心

数组的概念

➤ 数组是具有一定顺序关系的若干相同类型变量的集合体，组成数组的变量称为该数组的元素。

➤ 数组属于构造类型。

➤ 一维数组的声明

类型说明符 数组名 [常量表达式] ;

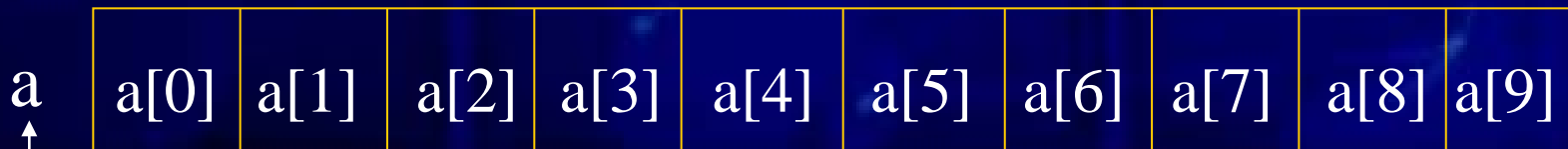
➤ 引用

❑ 必须先声明，后使用。

❑ 只能逐个引用数组元素，而不能一次引用整个数组
数组名 [整型表达式]

一维数组的存储顺序

- 数组元素在内存中顺次存放，它们的地址是连续的。
- 例如：具有10个元素的数组 a，在内存中的存放次序如下：



数组名字是数组首元素的内存地址。

数组名是一个常量指针，不能被赋值。

一维数组的初始化

可以在编译阶段使数组得到初值：

❑ 在声明数组时对数组元素赋以初值。

例如：`static int a[10]={0,1,2,3,4,5,6,7,8,9};`

❑ 可以只给一部分元素赋初值。

例如：`static int a[10]={0,1,2,3,4};`

❑ 在对全部数组元素赋初值时，可以不指定数组长度。

例如：`static int a[]={1,2,3,4,5}`

❑ 注意：不存在与数组初始化相对应的数组赋值

```
char v[3];
```

```
v={'c','a','d'}// error
```


多维数组的声明及引用

数据类型 标识符[常量表达式1][常量表达式2] ...;

例:

```
int a[5][3];
```

表示a为整型二维数组，用于存放 5 行 3 列的整型
数据表格

二维数组的声明及引用

➤ 二维数组的声明

类型说明符 数组名[常量表达式][常量表达式];

例如：float a[3][4]; 由 3 个一维数组构成

可以理解为：

a	[a[0]	——	a ₀₀	a ₀₁	a ₀₂	a ₀₃
		a[1]	——	a ₁₀	a ₁₁	a ₁₂	a ₁₃
		a[2]	——	a ₂₀	a ₂₁	a ₂₂	a ₂₃

➤ 存储顺序

按行存放，上例中数组 a 的存储顺序为：

a₀₀ a₀₁ a₀₂ a₀₃ a₁₀ a₁₁ a₁₂ a₁₃ a₂₀ a₂₁ a₂₂ a₂₃

➤ 引用

例如：b[1][2]=a[2][3]

二维数组的初始化

- 将所有数据写在一个{}内，按顺序赋值

```
int a[3][4]={1,2,3,4,5,6,7,8,9,10,11,12};
```

- 分行给二维数组赋初值

```
int a[3][4]={ {1,2,3,4},{5,6,7,8},{9,10,11,12}};
```

- 可以对部分元素赋初值

```
int a[3][4]={ {1},{0,6},{0,0,11}};
```

数组作为函数参数

- 数组元素作实参，与单个变量做参数一样。
- 数组名作参数，实际上是指针做参数
 - 形、实参数都是数组名，类型要一样，传送的是数组首地址
 - 形参是指针，实参是数组名，类型相同，传送的也是数组首地址
 - 对形参数组的改变会直接影响到实参数组

对象数组

➤ 声明：

类名 数组名[元素个数];

➤ 访问方法：

通过下标访问

数组名[下标].成员名

对象数组初始化与删除

- 数组中每一个元素对象被创建时，系统都会调用类构造函数初始化该对象。

- 通过初始化列表赋值。

Point A[2]={Point(1,2), Point(3,4)}; // 调用 2 次构造函数

- 如果没有为数组元素指定显式初始值，数组元素便使用默认值初始化（调用默认构造函数）

Point A[2]; // 这时 Point 类必须定义了默认构造函数

- 当数组中每一个对象被删除时，系统都要调用一次析构函数

指针变量

➤ 数据存贮在内存空间中，对数据的访问方式：

□ 通过变量名访问（直接访问）

✉ 数据存贮的地址由编译程序分配，固定的

✉ 变量标识符与其地址对应

□ 通过地址访问（间接访问）

✉ 指针变量，用于记录要访问的数据的存贮地址

✉ 访问的数据存贮地址由程序指定，可变

➤ 地址运算符：&表达式

□ 取表达式的内存地址，表达式必须是一个左值

&var 取变量 var 的地址

&(Object.ab) 取成员变量的地址

&(a[8]) 取数组元素的地址

指针变量

概念

指针：内存地址，用于间接访问内存单元

指针变量：用于存放地址的变量

指针变量是有类型的

声明

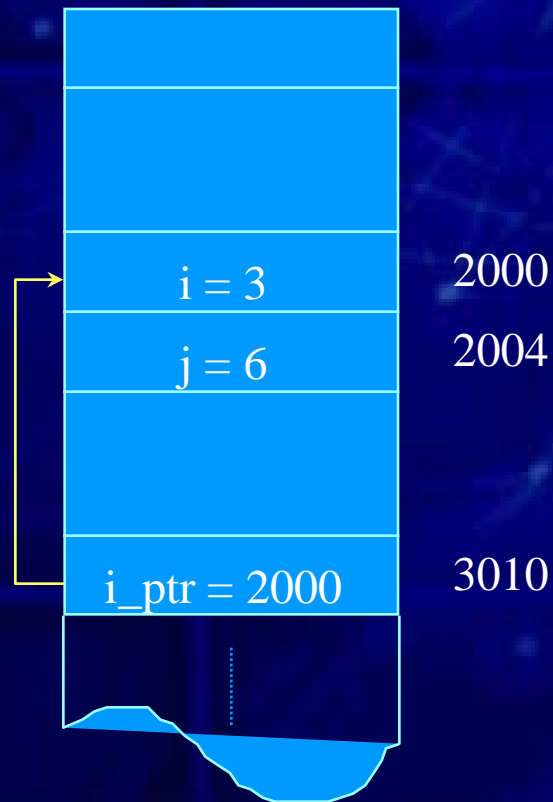
```
例：int i;  
     int *i_ptr = &i;
```

指向整型变量的指针

引用

```
i = 3;  
*i_ptr = 3;
```

内存用户数据区



void 类型指针

```
void *pv;        // 用于记录一个地址，类型未知（不关心）

int i;

pv = &i;         // 记录 i 的地址

*pv = 0;         // error

pv = pv + 1;     // error


int *pint = (int*)pv;

// void指针的值（地址）可以赋值给任何类型的指针变量

// 但需要类型强制转换
```

指向常量的指针

```
const Type * ptr;
```

- 不能通过指针来修改所指对象的值，但指针本身可以改变，可以指向另外的对象。

```
const char *name1 = "John"; //指向常量的指针
char s[]="abc";
name1 = s;                  //正确，name1本身的值可以改变
*name1 = '1';               //错误，试图修改常量
name1[2]='3';               //错误，试图修改常量
```

指针类型的常量

Type * const ptr;

➤ 声明指针是常量，指针的值（地址值）不能被改变。

例：

```
char str_1[] = "abcdef";
```

```
char str_2[] = "ghijkl";
```

```
char * const s = str_1;
```

```
s[1] = 'A';
```

```
s = str_2; // 错误，指针常量值不能改变
```

```
const char * const cp = str_1; // 指针和指向对象都是常量
```

指针变量的算术运算

➤ 指针与整数的加减运算

□ 指针 p 加上或减去 n ，等于是指针当前指向位置的前方或后方第 n 个数据对象的地址。

□ 这种运算的结果值取决于指针指向对象的数据类型

➤ 指针加一，减一运算

□ 指向后一个或前一个数据对象的地址

$px++$

$px--$

注：* 和 ++、-- 优先级相同，自右向左运算

$*px++ == *(px++)$

指针变量的关系运算

➤ 关系运算

- 指向相同类型数据的指针之间可以进行各种关系运算

> < == >= <=

- 指向不同数据类型的指针，以及指针与一般整数变量之间的关系运算是无意义的。

- 指针可以和零之间进行等于或不等于的关系运算。例如：
 $p==0$ 或 $p!=0$

➤ 赋值运算

- 向指针变量赋的值必须是地址常量或变量，不能是普通整数。但可以赋值为整数0，表示空指针。

数组与指针

- 数组名是指向数组第一个数组元素的常量指针
- 类型是数组元素的类型

```
int a[10];
```

```
int *pa = a;           // 或 pa = &a 或 pa=&a[0]
```

通过指针或数组名引用数组元素是等效的

a[0] , pa[0] , *pa, *a 引用的是同一数组元素

a[m], pa[m], *(pa+m), *(a+m) 引用的是同一数组元素

...

不能写 a++, 因为a是数组首地址是常量。

指针数组

➤ 数组的元素是指针型

```
Point *pa[2]; //由两个指向Point类型的指针组成的数组
```

➤ 多维数组

```
int pi[2][10]; // 由两个一维数组构成
```

pi[0]、pi[1] 是一个指向一维数组首地址指针 (常量)

```
pi[0] == &(pi[0][0]);
```

```
pi[1] == &(pi[1][0]);
```

指向函数的指针

➤ 声明形式

返回数据类型 (*函数指针名)(参数表);

➤ 调用

函数指针名(参数表)

含义:

- ❑ 数据指针指向数据存储区，而函数指针指向的是程序代码存储区
- ❑ 每一个函数名是一个常量函数指针

例：函数指针

```
#include <iostream>
using namespace std;

void print_float(float data_to_print);
void (*function_pointer)(float);

int main()
{
    float pi = (float)3.14159;
    function_pointer = print_float;
    function_pointer(pi); // 等于 print_float(pi)
    ...
}
```

对象指针

➤ 声明形式

类名 *对象指针名;

□ 例

```
Point A(5,10);
```

```
Piont *ptr;
```

```
ptr = &A;
```

➤ 通过指针访问对象成员

对象指针名->成员名

□ 例

```
ptr->getx() ;    // 相当于 (*ptr).getx();
```

this指针

- 隐含于每一个类的成员函数中的特殊指针。
- 明确地指出了成员函数当前所操作的数据所属的对象
 - 当通过一个对象调用成员函数时，该对象的地址赋给 this 指针，成员函数对对象的数据成员进行操作时，就隐含使用了 this 指针

例如：Point类的构造函数体中的语句：

$X = xx$ 相当于 $this \rightarrow X = xx$

- 调用对象的成员函数（除了静态成员函数外）时，都隐含地将 this 对象指针作为成员函数的一个参数

指向类成员（非静态）的指针

➤ 通过指向成员的指针来访问类的公有成员

➤ 指向成员变量的指针

☐ 声明

类型说明符 类名::*指针名;

☐ 赋值/初始化

指针名 = &类名::数据成员名;

☐ 访问数据成员

对象名.* 类成员指针名

对象指针名->*类成员指针名

➤ 指向成员函数的指针

☐ 声明

类型说明符 (类名::*指针名)(参数表);

☐ 赋值/初始化

指针名 = 类名::函数成员名;

☐ 访问成员函数

(对象名.* 类成员指针名)(参数表)

(对象指针名->*类成员指针名)(参数表)

指向类成员（非静态）的指针

```
class CA
{
public:
    int x;
    int y;

    void f(int i);
    void g(int i);
    ...
};
void CA::f(int i)
{
    x = i;
}
void CA::g(int i)
{
    y = i;
}

int main()
{
    int    CA::* pData      = &CA::x;
    void (CA::*pFunc)(int) = CA::f;
```

```
    CA a, b;

    a.*pData = 1;
    b.*pData = 2;
    cout << a.x << " " << b.x << endl;

    pData = &CA::y;
    a.*pData = 3;
    b.*pData = 4;
    cout << a.y << " " << b.y << endl;

    (a.*pFunc)(5);
    (b.*pFunc)(6);
    cout << a.x << " " << b.x << endl;

    pFunc = CA::g;
    (a.*pFunc)(7);
    (b.*pFunc)(8);
    cout << a.y << " " << b.y << endl;
    ...
}
```

指向类的静态成员的指针

- 对类的静态成员的访问不依赖于对象
- 可以用普通的指针来指向和访问静态成员

```
class Point
{
public:
    static int GetC();
    // ... 其他外部接口

    static int countP; //静态数据说明
    // ... 其他公共数据

private:    //私有成员
    ...
};

static int GetC()
{
    return countP;
}
//...其他外部接口实现

int Point::countP = 0; //静态数据定义
```

```
int main()
{
    int *count = &Point::countP;
    (*count) ++;
    cout << *count <<endl;

    int (*pGetC)() = Point::GetC;
    cout << pGetC() <<endl;

    Point A(3,4);
    cout << pGetC() <<endl;

    ...
}
```

继承、派生与多态性

北京 电大学

网络与交换技术国家重点实验室

宽带网研究中心

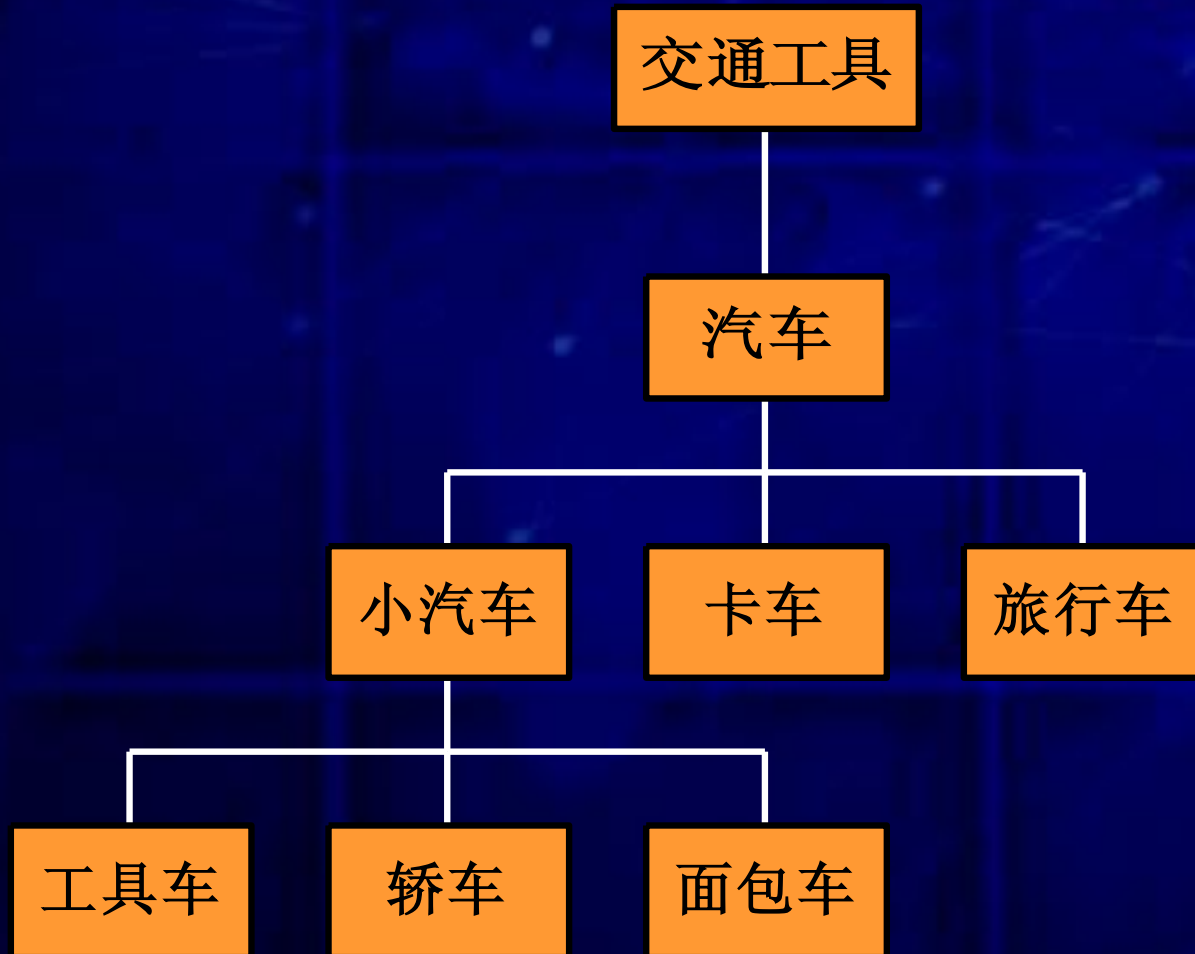
本章主要内容

- 类的继承与派生
- 类成员的访问控制
- 简单继承与多重继承
- 派生类的构造、析构函数
- 类成员的标识与访问

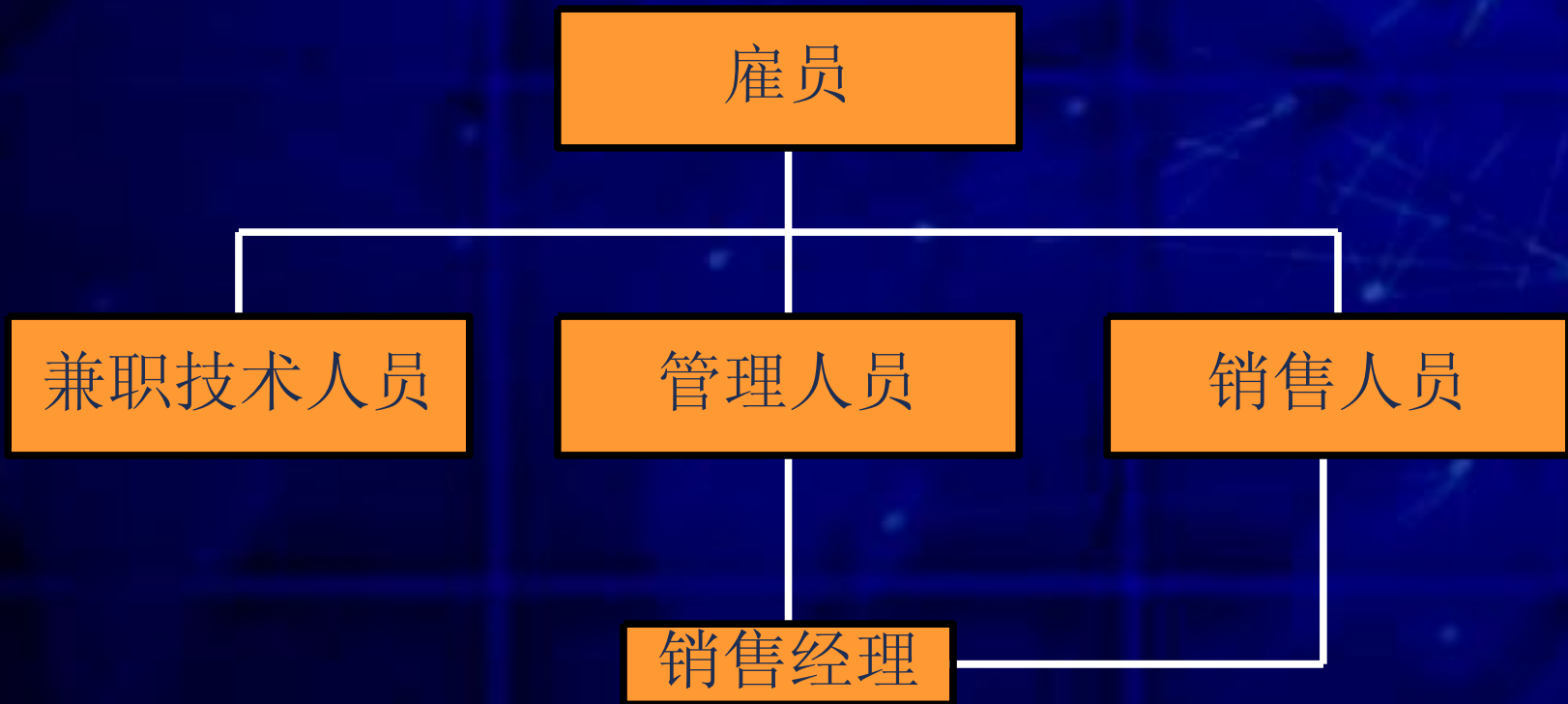
类的继承与派生

- 保持已有类的特性而构造新类的过程称为继承。
- 在已有类的基础上新增自己的特性而产生新类的过程称为派生。
- 被继承的已有类称为基类 或父类
- 派生出的新类称为派生类
 - ❑ 派生类将自动继承基类的所有特性 属性和方法
 - ❑ 派生类可以定义新的特性 属性和方法
 - ❑ 派生类可以对继承的方法定义新实现
- 派生是一个从抽象到具体的过程

现实世界对象的从属与继承关系



现实世界对象的从属与继承关系



继承与派生的目的

➤ 继承的目的

□ 实现代码 用 派生 可以继承原有 中具有普 性
的解决方法

□ 最大程度利用原有的成果

➤ 派生的目的

□ 当新的 出现 原有程序无法解决 或不能完全解决
时 要对原有程序 行改 具体解决新

➤ 继承和派生 过 C++ 的类来实现

派生类的声明

```
class 派生类名 继承方式 基类名  
{  
    成员声明  
};
```

继承方式

➤ 不同继承方式的影响主要体现在

- 过派生类**对象**对基类成员的访 权

➤ 三种继承方式

- 公有继承 public
- 私有继承 private (缺省)
- 保护继承 protected

公有继承(public)

- 基类的成员的访问属性 public/protected/private 在派生类中访问属性保持不变
- 派生类中的成员函数可以直接访问基类中的 public 和 protected 成员 但不能访问基类的 private 成员。
- 对外部来说 通过派生类的对象只能访问基类的 public 成员

例：公有继承

```
class CBase
{
    public:
        int x;
    protected:
        int y;
    private:
        int z;
    ...
};

class CDerived:
    public CBase
{
    public:
        int a;
        void f(int i);
    ...
};
```

```
void CDerived::f(int i)
{
    x = i;        // ok
    y = i;        // ok
    z = i;        // error
}

int main()
{
    CDerived d;

    d.x = 1;      // ok
    d.y = 2;      // error
    d.z = 3;      // error
    d.a = 4;      // ok
    d.f(5);       // ok
    ...
};
```


私有继承(private)

- 基类的 public/protected/private 成员在派生类中的访问属性 变成 private
- 派生类中的成员函数可以直接访问 基类中的 public 和 protected 成员 但不能直接访问 基类的 private 成员
- 对外 来说 通过派生类的对象不能直接访问 基类中的任何成员

例：私有继承

```
class CBase
{
    public:
        int x;
    protected:
        int y;
    private:
        int z;
    ...
};

class CDerived:
    private CBase
{
    public:
        int a;
        void f(int i);
    ...
};
```

```
void CDerived::f(int i)
{
    x = i;        // ok
    y = i;        // ok
    z = i;        // error
}

int main()
{
    CDerived d;

    d.x = 1;      // error
    d.y = 2;      // error
    d.z = 3;      // error
    d.a = 4;      // ok
    d.f(5);       // ok
    ...
};
```

保护继承(protected)

- 基类的 public/protected 成员在派生类中的访问属性变成 protected
- 派生类中的成员函数可以直接访问基类中的 public 和 protected 成员 但不能直接访问基类的 private 成员
- 对外部来说 通过派生类的对象不能直接访问基类中的任何成员
- protected 成员的作用
 - ❑ 对类的外部 protected 成员与 private 成员的性质相同
 - ❑ 对于其派生类来说 protected 成员与 public 成员的性质相同
 - ❑ 既实现了对外数据隐藏 又方便继承 实现代码重用

类型兼容规则

➤ 从某个基类派生出的public 派生类的对象 可以作为基类对象来使用

□ 例如 人类 ~~==派生==~~ ➡ 学生类

学生对象 完全可以当作 人类对象 来使用

□ 但是 相反的方向则禁止

➤ 具体表现在

□ 派生类的对象可以被赋值给基类对象

□ 派生类的对象可以用来初始化基类的引用

□ 指向基类的指 也可以用来指向派生类对象

□ 注 过基类对象名、指 只能访 基类的成员

例：类型兼容

```
class CB0 //基类
{
public:
    void display() {...}
    void func_b0() {...}
    ...
};

class CB1 : public CB0
{
public:
    void display() {...}
    void func_b1() {...}
    ...
};

class CD : public CB1
{
public:
    void display() {...}
    void func_d() {...}
    ...
};
```

```
int main()
{
    CD d, *pD = &d;
    CB0 *pB0 = &d;
    CB1 *pB1 = &d;

    pD->display();
    pD->func_b0();
    pD->func_b1();
    pD->func_d();

    pB1->func_b0();
    pB1->display();
    pB1->func_b1();
    pB1->func_d(); // error

    pB0->display();
    pB0->func_b0();
    pB0->func_b1(); // error
    pB0->func_d(); // error
}
```

单继承与多重继承

➤ 单继承

- ❑ 派生类只从一个基类派生。

➤ 多 继承

- ❑ 派生类从多个基类派生
- ❑ 派生类继承多个基类的所有的属性和行为成员

多重继承时派生类的声明

class 派生类名 继承方式1 基类名1

 继承方式2 基类名2

...

{

 成员声明

}

注意 每一个“继承方式” 只用于 制对紧 其后之基类的继承

例：多重继承

```
class A
{
    public:
        void setA(int);
        void showA();
        ...;
};

class B
{
    public:
        void setB(int);
        void showB();
        ...
};

class C : public A, private B
{
    public:
        void setC(int a, int b);
        void showC();
        ...
};
```

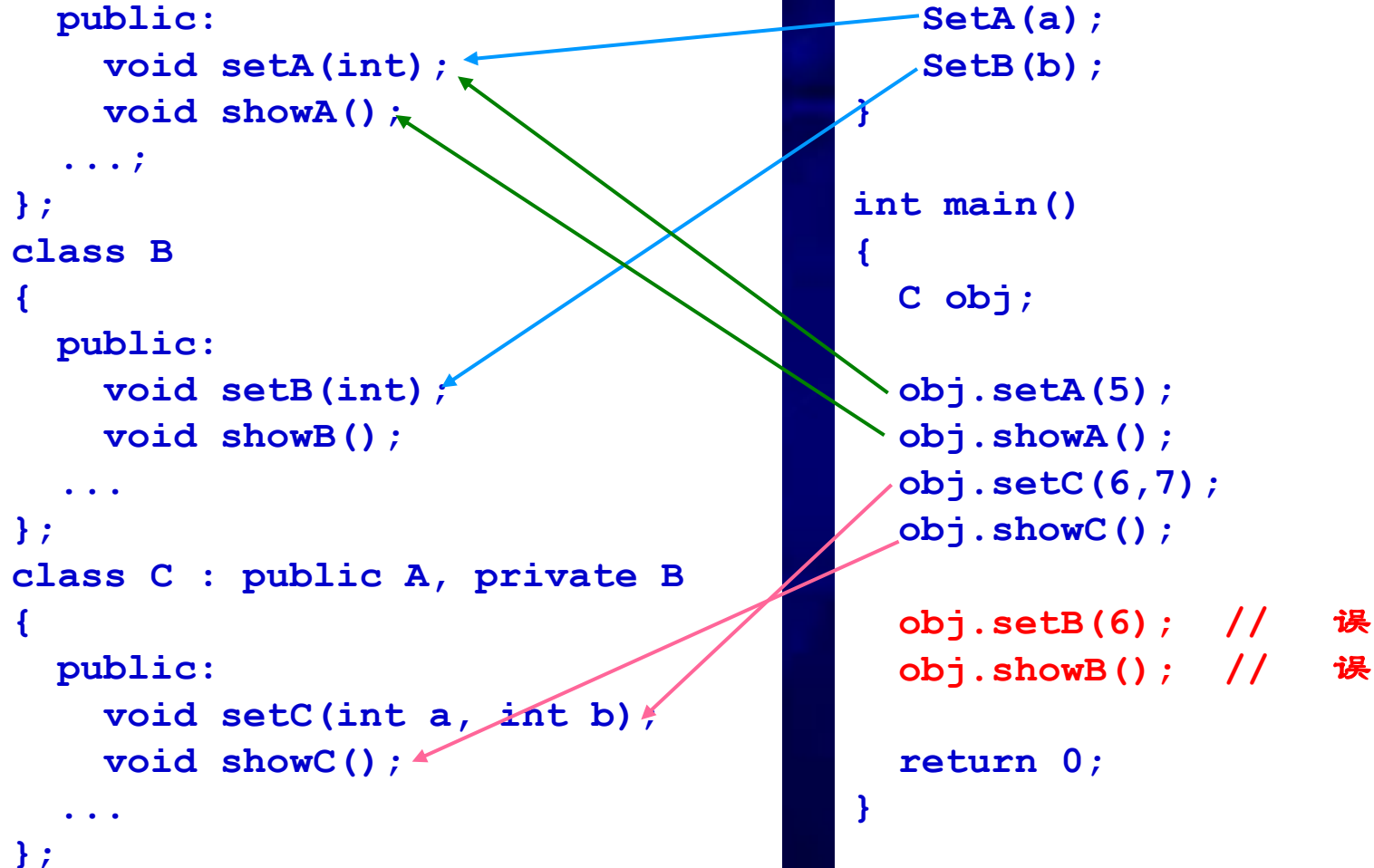
```
void C::setC(int a, int b)
{
    SetA(a);
    SetB(b);
}

int main()
{
    C obj;

    obj.setA(5);
    obj.showA();
    obj.setC(6,7);
    obj.showC();

    obj.setB(6); // 误
    obj.showB(); // 误

    return 0;
}
```



派生类的构造函数

➤ 基类的构造函数不被直接继承成为派生类的构造函数
派生类中要声明自己的构造函数

□ 在执行派生类构造函数时 基类构造函数将被自动执行

➤ 派生类的构造函数

□ 要对本类中新增成员 行初始化

□ 对继承的基类成员的初始化 可以在自动调用基类构造函数的时候 由基类构造函数完成

□ 派生类构造函数也可以对继承的基类成员 重新初始化

➤ 派生类的构造函数 要向基类构造函数传 参数

□ 编写派生类的拷贝构造函数时 也可能 要向基类拷贝构造函数传 参数

派生类的构造函数

派生类名::派生类名 (参数表)

基类名1(参数表1) 基类名2(参数表2) ...

成员1(参数表) 成员2(参数表) ...

{

成员的初始化

}

例：派生类的构造函数

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

class B
{
    public:
        B()      {...};
        B(int y) {...};
        ...
};

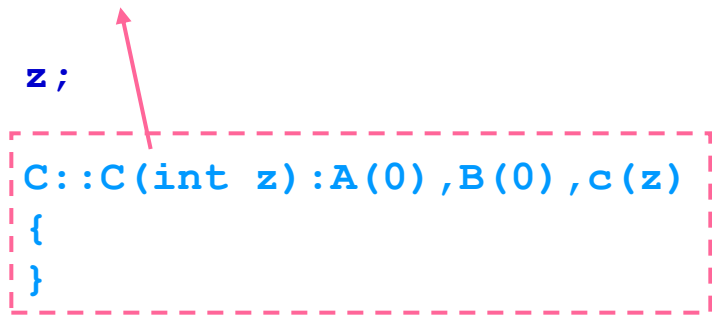
class C : public A, public B
{
    public:
        C();
        C(int z);
        C(int x, int y, int z);
    private:
        int c;
};
```

```
C::C()      // 调用A、B的 认构 函数
{
    c = 0;
}

C::C(int z) : A(0), B(0)
{
    c = z;
}

C::C(int z) : A(0), B(0), c(z)
{
}

C::C(int x, int y, int z) :
    A(x), B(y), c(z)
{
}
```



构造函数的调用次序

- 1 调用基类构造函数调用序按照它们被继承时声明的序从左向右。
- 2 调用成员对象的构造函数调用序按照它们在类中声明的序。
 - 对简单对象 如 int 等的初始化也相当于调用其构造函数
- 3 派生类的构造函数体中的内容。

派生类的析构函数

- 基类的析构函数也不被派生类继承 派生类自行声明 但派生类的析构函数会自动调用基类的析构函数
- 声明方法与一般类的析构函数相同。
- 不要显式地调用基类的析构函数 系统会自动式调用。
- 析构函数的调用次序与构造函数相反
 - 派生类、成员对象、基类



同名隐藏规则

当派生类与基类中有相同成员时

- 若未加限定，则通过派生类对象访问的是派生类中的同名成员
- 如要通过派生类对象访问基类中被覆盖的同名成员，应使用基类名限定。

例：同名隐藏

```
#include <iostream>
using namespace std;
```

```
class B1 //基类B1
{
```

```
    public:
        int nV;
        void func() {...};
        ...
};
```

```
class D1: public B1 // 派生类
```

```
{
    public:
        int nV; //同名数据成员
        void func(); //同名函数成员
};
```

```
void D1::func()
{
```

```
    B1::nV = 1;
```

```
    B1::func();
```

```
    nV = 0;
```

```
    func();
}
```

```
int main()
{
```

```
    D1 d;
```

```
    d.nV = -1;
```

```
    d.func();
```

```
    d.B1::nV = 1;
```

```
    d.B1::func();
```

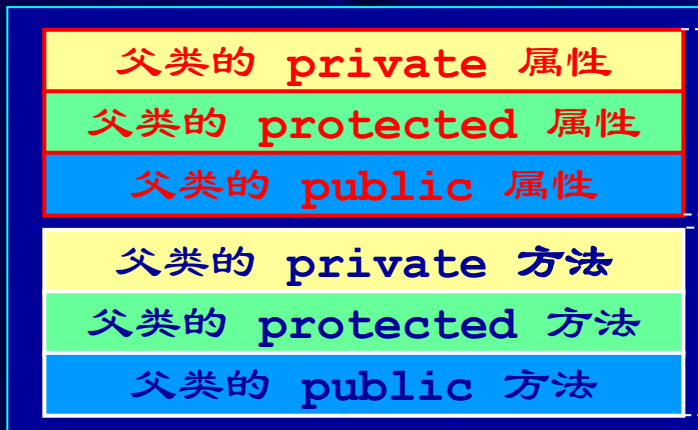
```
    ...
```

```
};
```


类继承对象的内存映像

➤ 派生类继承了父类的所有方法和属性

父
类
对
象



❑ 派生类中包含父类的所有成员

父类的内存映像

❑ 父类的成员成为了派生类的成员

❑ 在派生类中 对父类成员的访

权 与其自身成员一样 受到访

控制权 的 制

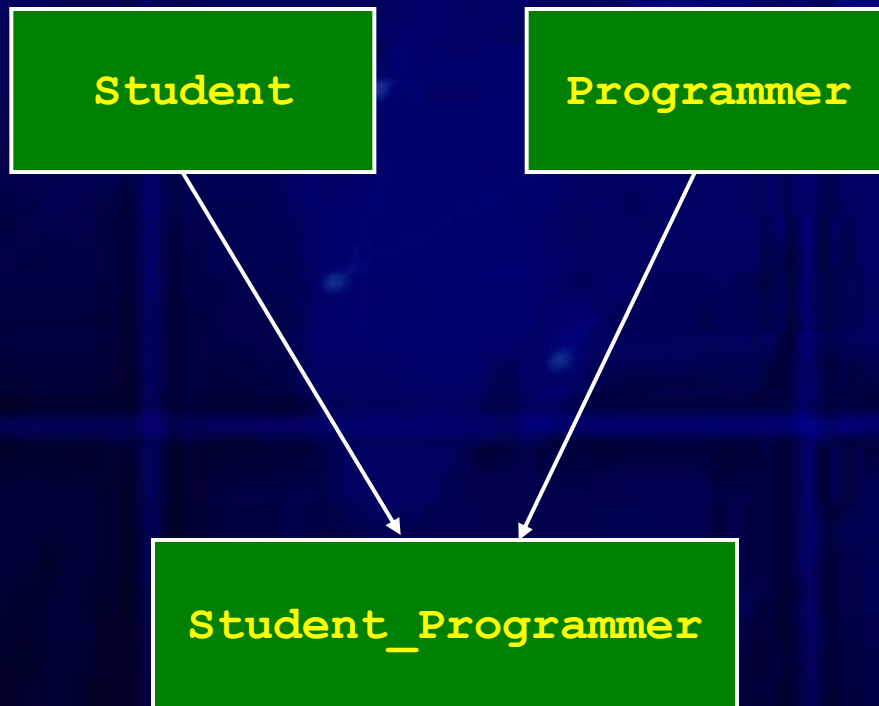


派生类对象

多重继承

➤ 派生类从两个以上的父类继承

□ 派生类将继承和拥有所有父类的属性和方法



多重继承

➤ 派生类从两个以上的父类继承

❑ 派生类将继承和拥有所有父类的属性和方法

```
class Student
{
protected:
    char * pName;
    ...
};

class Programmer
{
protected:
    char * pName;
    ...
};

class Student_Programmer :
public Student,
public Programmer
{
    ...
};
```

student 的属性和方法

pName

programmer 的属性和方法

pName

Student_Programmer
自身的属性和方法

多重继承

➤ 可能存在的二义性

```
class Student
{
protected:
    char * pName;
    ...
};
class Programmer
{
protected:
    char * pName;
    ...
};
class Student_Programmer :
public Student,
public Programmer
{
public:
    char * GetName();
    ...
};
```

```
char * Student_Programmer::GetName()
{
    return pName;
}
```

二义性

编译器不知 到底访
哪一个 pName 成员

多重继承

➤ 可能存在的二义性

```
class Student
{
protected:
    char * pName;
    ...
};
class Programmer
{
protected:
    char * pName;
    ...
};
class Student_Programmer :
public Student,
public Programmer
{
public:
    char * GetName();
    ...
};
```

```
char * Student_Programmer::GetName()
{
    return Student::pName;
// 或者
// return Programmer::pName;
}
```

必 加类 定符 “::” 以
消 二义性。

当不存在二义性时 则类
定符不是必 的。

在访 存在二义性的方法
时也必 使用 定符来
行修 。

二义性问题

- 在多继承时 基类与派生类之 或基类之 出现同名成员时 将出现访 时的二义性 不确定性 二义性将导致编译器报
- 当派生类从多个基类派生 而 些基类又从同一个基类派生 则在访 此共同基类中的成员时 将产生二义性—— 用虚基类来解决。

多态性

多态性的概念

- 多态性是 面向对象程序设计的 要特征之一。
- 多态性是指发出同样的消息被不同类型的对象接收时有可能导致完全不同的行为。
- 多态的实现
 - ❑ 函数 载/ 算符 载
 - ❑ 虚函数

运算符重载 (overloading)

```
class complex //复数类声明
{
public:
    complex(double r=0.0,double i=0.0) //构造函数
    {
        real=r;
        imag=i;
    }
    void display()
    {
        cout << "(" << real << "," << imag << ")"; //显示复数的值
    }
private:
    double real;
    double imag;
};
```

用"+"、"-","<<"能够实现复数的加减 算、输出吗

答案 可以 载"+"、"-","<<" 算符

运算符重载的实质

➤ 算符 载与函数 载本质相同

□ 只不过 载的是 C++ 中保留的 算符 $+ - * /$

➤ 算符 载是对已有的 算符赋予多 含义

➤ 必要性

□ C++ 中 定义的 算符其 算对象只能是基本数据类型
而不 用于用户自定义类型 如类

➤ 实现机制

□ 将指定的 算表达式转化为对 算符函数 的调用 算
对象转化为 算符函数的输入参数

□ 编译系统对 载 算符的 择 循与函数 载 择相
同的原则

规则 and 限制

- 可以 载C++中 下列 算符外的所有 算符
. * :: ?:
- 只能 载C++语言中已有的 算符 不可臆 新的。
- 不改变原 算符的优先级和结合性。
- 不能改变操作数个数。
- 经 载的 算符 其操作数中至少应该有一个是自
定义类型 class 或 struct

操作符重载的两种形式

➤ 将操作符 重载为类成员函数

- ❑ 操作符函数是类成员函数

➤ 重载为友元函数

- ❑ 操作符函数是类的友元函数

- ❑ 若操作符函数只访 类中的 public 成员 则无 是友元

运算符重载函数

➤ 声明形式

```
函数类型 operator 算符 形参  
{  
    .....  
}
```

➤ 载为类成员函数时

参数个数 = 原操作数个数 - 1 后置++、-- 外

➤ 载为友元函数时

参数个数 = 原操作数个数

且至少应该有一个是自定义类型的形参

运算符成员函数的设计

➤ 二元 算符 以+为例

$a + b$

其中 a 为 A 类对象 b 可以使各种类型

➤ + 被 载为 A 类的一个成员函数 $A::operator+(Type\ b)$

□ 算符成员函数与普 成员函数没有本质差别

➤ 表达式 $a + b$ 相当于

$a.A::operator+(b)$

例：复数类的 +- 重载

```
class complex //复数类声明
{
private:
    double real, imag;
public:
    complex(double r=0.0,double i=0.0) {...}; //构 函数
    void display() {...}; //显示复数的值
    complex operator+(complex c2)
    {
        complex c;
        c.real = real + c2.real;
        c.imag = imag + c2.imag;
        return c;
    };
    complex operator-(complex c2)
    {
        return complex(real-c2.real, imag-c2.imag);
    };
};
```

$c1+c2$ $c1-c2$

运算符成员函数的设计

➤ 前置一元 算符 `+a` `-a` `++a` `--a` 等

□ 以 `-a` 为例 `a` 是 `A` 类的对象

`operator-` 被 载为 `A` 类的成员函数 且没有参数

□ 表达式 `-a` 相当于 `a.A::operator-()`

□ 同样 `--a` 相当于 `a.A::operator--()`

`++a` 相当于 `a.A::operator++()`

```
class complex //复数类声明
{
public:
    ...
    complex operator-()
    {
        return complex(-real, -imag);
    };
};
```


运算符成员函数

➤ 后置一元 算符 ++和-- 要特殊处理

- ❑ 以 `a--` 为例 `a` 是 `A` 类的对象
`operator--` 被 载为 `A` 类的成员函数 有一个整型参数 但未用
- ❑ 表达式 `a--` 相当于 `a.A::operator--(0)`
- ❑ 同样 `a++` 相当于 `a.A::operator++(0)`

```
class complex //复数类声明
{
public:
    ...
    complex & operator--()      // --c
    {   real--;   imag--;
        return *this;
    };
    complex operator--(int)     // c--
    {   complex temp = *this;
        --*this;
        return temp;
    };
};
```


重载赋值类运算符=(+=, -=, ...)

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

Class Table{
    Name* p;
    int sz;
    Table(int s=20) {
        p= new Name[sz=s];
    }
    Table& operator=(const Table&);
    //...
};

Table x, y, z;
若要实现 x=y=z;
```

```
Table& Table::operator=(const Table&t)
{
    if(this !=&t) //当心自赋值 t=t
    {
        delete[]p; //删 老元素
        p=new Name[sz=t.sz];
        for(int i=0; i<sz;i++)
            p[i]=t.p[i];
    }
    return *this;
}
```

运算符友元函数

- 如果要重载一个运算符使之能够用于操作某类对象的私有成员 可以将此运算符重载为该类的友元函数。
- 函数的形参代表依自左至右次序排列的各操作数。
- 后置单目运算符 ++ 和 -- 的载函数 要特殊处理 形参列表中要增加一个 int 但不必写形参名。

运算符友元函数

➤ 二元 算符 $+/*-$ 等 载

$a + b$ 等同于 $\text{operator}+(a, b)$

➤ 前置一元 算符 $+ - ++ --$ 等 载

$++a$ 等同于 $\text{operator}++(a)$

➤ 后置单目 算符 $(++\text{和}--)$ 载

$a++$ 等同于 $\text{operator}++(a, 0)$

例：复数类的重载

```
class complex //复数类声明
{
    friend complex operator+(complex c1,complex c2);
    friend complex operator-(complex c1,complex c2);
    friend complex operator++(complex& c1);          // ++c
    friend complex operator++(complex& c1,int);      // c++
private:
    double real, imag;
public:
    ...
};

complex operator+(complex c1,complex c2)
{
    c1.real += c2.real;
    c1.imag += c2.imag;
    return c1;
}
```

例：复数类的重载

```
complex operator-(complex c1,complex c2)
{
    c1.real -= c2.real;
    c1.imag -= c2.imag;
    return c1;
}

complex operator++(complex& c1)           // ++c1
{
    c1.real ++;
    c1.imag ++;
    return c1;
}

complex operator++(complex& c1,int)
{
    complex temp = c1;
    ++c1;
    return temp;
}
```

例：复数类 << 的重载

```
// 将复数 a 输出到流式IO, 表示成 (re,im) 形式输出
ostream& operator <<(ostream& Ostr, complex c)
{
    return Ostr << "(" << c.re << "," << c.im << ")";
}

int main()
{
    complex a,b;
    a.re = 10.;
    a.im = 15.;
    b.re = 25.5;
    b.im = -3.5;

    cout << "a=" << a << endl;
    cout << "b=" << b << endl;
    cout << "a+b" << a+b << endl;

    return 0;
}
```

C++ 类的多态性

➤ 类的多态性是 面向对象程序设计的 要支柱

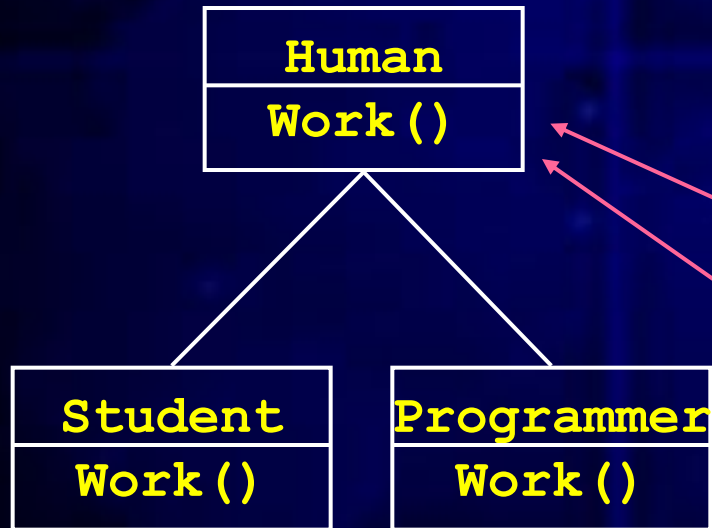
□ 多态性体现在 当我们向某种类型的对象发 相同消息时
即调用对象的方法 允许不同对象 行不同的操作

□ 支持动态联编

⊗ 在程序 行的过程中 确定具体调用对象的哪一个方法

□ 不提供多态性 以提 代码 用的效率

多态性的必要性



```
Student a;  
Programmer b;  
Human * p;
```

```
p = &a;  
p->Work();
```

```
p = &b;  
p->Work();
```

与自然语
义不符

多态性的必要性

```
class Phone
{
    ...
public:
    int call(int iNumber);

    int dial(int Number);
    int Conversation();
    int Hangup();
    ...
};

int Phone::call(int iNumber)
{
    ...

    dial(int Number);
    Conversation();
    Hangup();

    ...
}
```

```
int Phone::dial(int Number)
{
    ...
}
int Phone::Conversation()
{
    ...
}
int Phone::Hangup()
{
    ...
}

int main()
{
    Phone my_phone;
    my_phone.call(62281342);
    ...
}
```

原有

多态性的必要性

```
class MobilePhone : public Phone
{
    ...
public:
    int call(int iNumber);

    int dial(int Number);
    // int Conversation(); 不用 写
    int Hangup();
    ...
};
```

```
int MobilePhone::call(int iNumber)
{
    ...

    dial(int Number);
    Conversation();
    Hangup();

    ...
}
```

一个流程虽然与父类的流程相同
但 MobilePhone::call() 也必须
写 否则将会使用父类实现
调用父类方法 Phone::dial()、
Phone::Conversation() ...

```
int MobilePhone::dial(int Number)
{
    ... // 与父类不同
}
```

```
int MobilePhone::Hangup()
{
    ... // 与父类不同
}
```

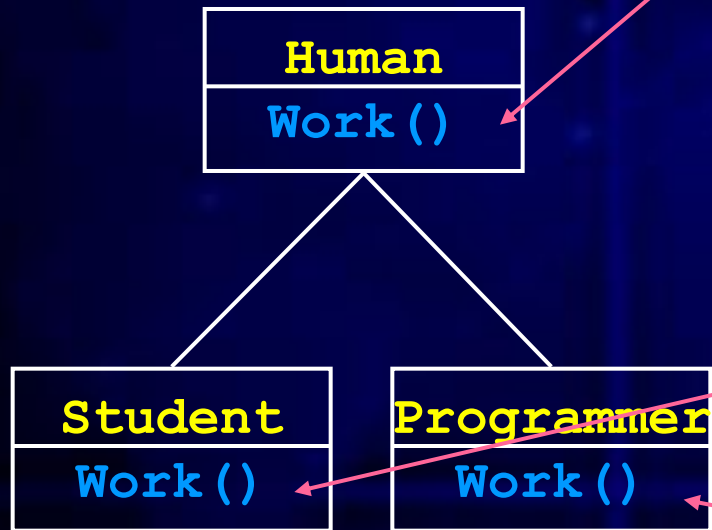
```
int main()
{
    MobilePhone my_phone;
    my_phone.call(13900012345);
    ...
}
```

MobilePhone::Call()

派生的
新

多态性支持——虚函数

将 `Work()` 定义
为虚成员函数



```
Student a;  
Programmer b;  
Human * p;
```

```
p = &a;  
a.Work();  
p->Work();
```

```
p = &b;  
b.Work();  
p->Work();
```

虚成员函数

- 虚函数是动态绑定的基础。
- 在调用对象虚函数时 根据对象所属的派生类 决定调用哪个函数实现 动态联编
- 是 态的成员函数。
- 在类的声明中 在函数原型之前写 virtual。
- virtual 只用来说明类声明中的原型 不能用在函数实现时。
- 具有继承性 基类中声明了虚函数 派生类中无论是否说明 同原型函数 自动为虚函数。
- 在派生类中 可以对基类的虚函数 行 写 覆 盖 而不是 载

多态性支持——虚函数

```
class Phone
{
public:
    virtual int call(int iNumber);
    virtual int dial(int Number);
    virtual int Conversation();
    virtual int Hangup();
    ...
};

int Phone::call(int iNumber)
{
    ...
    dial(int Number);
    Conversation();
    Hangup();
    ...
}

int Phone::dial(int Number)
{
    ...
}

int Phone::Conversation()
{
    ...
}

int Phone::Hangup()
{
    ...
}
```

```
class MobilePhone : public Phone
{
public:
    virtual int dial(int Number);
    virtual int Hangup();
    ...
};

int MobilePhone::dial(int Number)
{
    ...
}

int MobilePhone::Hangup()
{
    ...
}
```

只 要 写 MobilePhone 的
特殊 分的代码其他 从基类
中继承 最大 度的 用

多态性支持——虚函数

```
int main()
{
    MobilePhone my_phone;
    my_phone.call(13900012345);
    ...
}
```

将调用基类的 `Phone::Call()`

```
int Phone::call(int iNumber)
{
    ...
    dial(int Number);
    Conversation();
    Hangup();
    ...
}
```

在基类的 `Phone::Call()` 中将自动调用
派生类的方法实现 `MobilePhone::dialCall()`
和 `MobilePhone: Hangup()`

静态绑定与动态绑定

➤ 绑定 联编

- 程序自身彼此关联的过程 确定程序中的操作调用与执行该操作的代码 的关系。

➤ 静态联编

- 联编工作出现在编译 阶段 用对象名或者类名来 定义要调用的函数。

➤ 动态联编

- 联编工作在程序 运行时执行 在程序 运行时才确定将要调用的函数。

虚函数与动态联编

➤ 先前联编/ 静态联编 Early Binding/Static Binding

- ❑ 由编译程序在编译时确定调用特定类的特定方法
- ❑ 效率

➤ 后联编/动态联编 Late Binding/Dynamic Binding

- ❑ 编译程序在编译时无法确定调用哪一个类的方法 种情况下 要在行时动态地确定具体调用哪一个类的方法
- ❑ 灵活性好、 度的抽象性 但 要 外的 行开

```
main()
{
    Phone * my_phone;
    my_phone = new MobilePhone;

    my_phone->fee();
    ...
}
```

若 `fee()` 不是虚函数 则被静态联编为调用 `Phone::fee()`
若 `fee()` 是虚函数 则 要动态联编 在 行时根据目标对象的类型 调用其相应的 `fee()`

抽象类

带有纯虚函数的类称为抽象类:

```
class 类名
```

```
{
```

```
    virtual 类型 函数名(参数表)=0;
```

```
    //纯虚函数, 只定义方法调用的格式
```

```
    ...
```

```
}
```

抽象类

➤ 作用

- ❑ 抽象类为抽象和设计的目的而声明 将有关的数据和行为组织在一个继承层次结构中 保证派生类具有要求的行为。
- ❑ 对于暂时无法实现的函数 可以声明为纯虚函数 留给派生类去实现。

➤ 注意

- ❑ 抽象类只能作为基类来使用。
- ❑ 不能声明抽象类的对象。
- ❑ 构造函数不能是虚函数 析构函数可以是虚函数。

例：抽象类

```
#include <iostream>
using namespace std;
class B0                                //抽象基类B0声明
{ public:                                //外  接口
    virtual void display( )=0;          //纯虚函数成员
};
class B1: public B0
{
    public:
        void display(){cout<<"B1::display() "<<endl;}    //虚成员函数
};

class D1: public B1
{
    public:
        void display(){cout<<"D1::display() "<<endl;}    //虚成员函数
};
```