

9.3

BNF：巴科斯范式 描述编程语言的文法，自然语言存在不同程度的二义性。这种模糊、不确定的方式无法精确定义一门程序设计语言。必须设计一种准确无误地描述程序设计语言的语法结构，这种严谨、简洁、易读的形式规则描述的语言结构模型称为文法。该范式由他定义Algol60语言时提出

编程的两个视角：科学、艺术

编程范式

- 面向过程：分析出解决问题所需要的步骤，然后用函数把这些步骤一步一步实现，使用的时候一个一个依次调用就可以了（命令式思想：即程序员一步步告诉计算机应该做什么）
- 面向对象：把构成问题事务分解成各个对象，建立对象的目的不是为了完成一个步骤，而是为了描述某个事物在整个解决问题的步骤中的行为
- 函数式
 - 函数副作用：当调用函数时，除了返回函数值之外，还对主调用函数产生附加的影响。例如修改全局变量（函数外的变量）或修改参数
 - 函数式编程：为了消除这种副作用的编程方式，告诉计算机要干什么，而不是怎么干
 - 参考：<https://blog.csdn.net/archimelan/article/details/81940858>
- 逻辑式：将人类的知识告诉给机器，然后让机器自己决定计算结果（AI的第二阶段）

9.10

进攻式编程：开发阶段主动暴露问题，从而在发布前尽可能多的解决问题

防御式编程：开发阶段预防可能出现的所有问题，避免错误数据对系统造成影响

创建OO的背景

Simula 1时期

simulation：仿真，用于模拟某些不能真实做的（炸毁三峡大坝产生的后果）、有较大随机性的实验

- 仿真语言为仿真研究人员提供了专门用于建模、仿真实验和仿真结果统计、分析、显示的程序语句
- 不可测原理
- 仿真语言打破了严格的先进后出制度
- 采取的措施类似现在虚拟机的工作

Simula 1：**不是program language**，仅仅为了完成simulation引入的约束/语言

Simula 67时期

- 考虑到Simula1的不可重用性，提出了class subclass inheriting的思想
- 抽象函数的概念
- 通用程序设计语言
- 有了OO思想

C++的诞生

BS最开始为了完成博士论文需要一门语言做支撑

1.simula: 性能差

最终被市场淘汰: 用的人少, 无法为复杂的系统提供支持

2.BCPL: debug难

此时他还是用原有语言完成了论文, 但希望有一门语言能综合Simula编程体验好, BCPL性能高的特点

贝尔实验室阶段

C++思想逐步完善

- Cpre: 预处理程序, C加上Simula的类的机制
- 此后逐渐诞生了创建一门语言的想法

1983年诞生

此时C++源码先通过cpp预处理后再通过Cfront翻译成C语言, 最后通过C编译器来使程序运行

- 用Cfront不用Cpre的原因: Cpre不懂C语法, Cfront懂, 发现语法错误会传回source code, 但Cpre将方言部分翻译成c后交给cc, 此时若发现错误才传回source code
- 正交性: 矛盾体, 有你没我, 有我没你。但这些可以容忍, 如果共存后效率能提高

初始化与赋值

对于复合类型的初始化和赋值有区别, 基本类型则没有区别:

https://blog.csdn.net/qq_38211852/article/details/80629691

异常

- 可以预见的
- 无法避免的 (a/b, 无法避免输入b=0; 无法避免文件路径不存在)
- 处理异常不是让异常消失, 而是让异常产生后能受到控制, 不会让系统手足无措

9.12

类型

类型确定了取值范围和相应的操作

类型分类


强 & 弱

- 强类型: 使用时必须指定类型, 不能隐式转换
- 弱类型: 可以隐式转换

动态 & 静态

- 动态类型: 运行时才检查类型
- 静态类型: 编译时就检查了类型

- 例如
 - java: `int x='666'/3`; 编译时就报错
 - python: `int x='666'/3`; 运行时才会报错

1568424056794

表达式

`cout << a`的 `<<` 是一个双目操作符

副作用：对使用的变量不但引用，对它们的值还加以改变

- `a+b*c+(++b)`和`a+(++b)+b*c`结果不一样

左值和右值的区别：<https://zhidao.baidu.com/question/167692164.html>

- 左值也可以称为左值表达式
- `++i`是左值表达式，`i++`是右值表达式
- 逗号表达式`,,,;` 的值作为该表达式的值

9.17

c++枚举

- 直接输出枚举常量会在屏幕上显示对应的值，而不是枚举的名称
- 不能直接给枚举类赋一个int值，而可以`today=weekday(4)`，其中`weekday`是定义的枚举类

switch-case

1.不同的代码段内对相同的变量做判断，则可以集中成一个switch函数，避免重复 2.case的值可以写成枚举类型，而不是复杂难以记忆的数 3.switch实现与优化（不同的编译器有不同的处理方式）；参考：

<https://www.jb51.net/article/142122.htm>

- 1.逐条件判断：把各个case按顺序进行判断，直到找到正确的case或者跳出
 - 适用于case条件很少的情况
- 2.跳转表实现：跳转表存放各个case执行语句的地址，把switch的条件值作为偏移量加上首地址就能找到对应case的地址
 - 特点是只用进行一次cmp操作
 - 适用于case条件较多，但是case的值比较连续的情况（不连续也可以用，但对空间损耗比较大）
 - 详细过程参考ppt备注
- 3.二分查找法：编译器先将所有case值排序后按照二分查找顺序写入汇编代码，在程序执行时则采二分查找的方法在各个case值中查找条件值，如果查找到则执行对应的case语句，如果最终没有查找到则执行default语句
 - 适用于case较多，且值分布较离散的情况

4.优化switch-case

- 表驱动
 - 表驱动的应用：<https://blog.csdn.net/zhouyulu/article/details/6860893>

- 菜单调节。一个模块，有几十个菜单参数可以调节，每个菜单调节的步进、范围不同，但都是“触发消息、调节数值”流程。
- 按键响应。多个按键，属于“按键，执行对应处理函数”流程。
- 鼠标操控。不同状态下移动鼠标，属于“状态判断、响应鼠标处理函数”流程。
 - 相对if else来说效率提升（对于switch不一定）
 - 使代码清晰
 - 相对于if-else和switch来说更易维护：<https://www.cnblogs.com/cnlian/p/6064888.html>
- 表驱动的进一步优化：将内容整合成配置文件，以后只要修改配置文件，而不需要重新编译程序

函数

EBP是当前函数的存取指针，即存储或者读取数时的指针基地址；ESP就是当前函数的栈顶指针

对于ESP、EBP的理解：调用一个函数时，先将堆栈原先的基址（EBP）入栈，以保存之前任务的信息。然后将栈顶指针的值赋给EBP，将之前的栈顶作为新的基址（栈底），然后再这个基址上开辟相应的空间用作被调函数的堆栈。函数返回后，从EBP中可取出之前的ESP值，使栈顶恢复函数调用前的位置；再从恢复后的栈顶可弹出之前的EBP值，因为这个值在函数调用前一步被压入堆栈。这样，EBP和ESP就都恢复了调用前的位置，堆栈恢复函数调用前的状态。

值传递

1.esp指针下移32位，开辟main函数的栈空间 2-3.栈中保存参数1和2 4.调用函数func，call相当于先push eip（保存待返回的地址），再jmp_func（跳转到被调函数的首条指令） 10.保存main函数的ebp地址 11.将当前函数的栈底设为esp的值 12.开辟被调函数的栈空间 13-15.寄存器存传来的参数值，相加后存入eax 16.函数执行完毕，栈顶指针回到ebp 17.ebp出栈，回到ebp_main的位置 18.返回main函数调用func后的第一条指令的地址 5.将eax中的值传入ebp指向的内存空间 ...：继续main函数的逻辑

引用传递

注意引用传递没有类似值传递中为新空间赋值的指令，且mov (edi),0x1表示修改了某个地址对应内存单元内的值

call by name：需要用到该参数的时候才会计算参数表达式的值

```
void p(int x){++i;++x;}

int main(){
    int a[10],i=1;
    a[1]=1;
    a[2]=2;
    p(a[i]);
}
```

- call by value: i=2; 其余值不变
- call by reference: i=2, a[1]=2, 其余值不变
- call by name: i=2, a[2]=3（当计算++x是才传入参数，由于i的值加1，所以传入的参数是a[2]）

__cdecl & __stdcall & __fastcall

- 不同的函数调用约定（参数入栈顺序是从右至左还是从左至右、是由调用者还是被调用者把参数弹出栈）
- 参考：<https://blog.csdn.net/luoweifu/article/details/52425733>

9.24

复杂的表驱动1

```
enum LOG{
    LOGIN,
    LOGOUT,
    ...
}

struct info{
    LOG id; //消息id
    string s; //消息实体
}

struct info Event[]={//定义结构体数组表
    {LOGIN,"login success"},
    {LOGOUT,"logout success"},
    ...
};

String ShowMsg(LOG e){
    return Event[e].s;
}
```

复杂的表驱动2

```
//相对于1需要修改的地方
void (MSG_FUN*)(string); //函数指针，保存的是某个函数的内存地址
struct info{
    LOG id;
    string s;
    MSG_FUN fun_ptr;
}

string ShowMsg(LOG e){
    return Event[e].fun_ptr(Event[e].s); //所有string都能用于fun_ptr这个接口
}

//修改Event数组的内容
{
    {LOGIN,"login success",login_fun},
    {LOGOUT,"logout success",logout_fun},
    ...
}
```

```
}
```

函数调用

每个函数都有栈空间：称为frame（active frame为当前运行函数的栈空间） main()调用g()的注意点（参考PPT p46）

- main还在活跃时保存ebp、eip再转给g()执行
- caller-调用者 callee-被调用者
- mov：空间已经存在，只是填值
- f(a,b)参数一般从右往左进栈

_cdecl：被调用者不清栈，而是调用者清栈 _stdcall：被调用者清栈；函数执行权交还同时退掉参数1,2 P46（一般用ret 8，实际作用和ebp+8相同）

- 好处：节省空间
- 坏处：对于可变参数的函数无法计算ebp还参数个数；但对于调用者来说是知道的，此时只能用_cdecl

上述两者相当于两种不同契约，该约定由compiler、linker共同管理("法官"的角色)

printf("%f %d %s")也是一种约定（如何编写函数？ switch case？）

- 这种约定没有人监督，可能会被攻击

函数原型

如果没有原型，编译器解释f(1,2)时不知道1, 2的类型更不知道函数返回值的类型，如果有了函数原型int f(int,int)，编译器先掌握了函数信息，就知道f(1,2)参数个数正确，1、2是int类型（如果是其他的会转换），返回时检索对应的字节数以返回int类型

编译链接 编译只编译当前模块 g(){ //a.cpp f() //b.cpp } 编译每个编译单元(.cpp)时是相互独立的，即每个cpp文件之间是不知道对方的存在的,.cpp编译成.obj后，link期时a.obj才会从b.obj中获得f()函数的信息 link是将编译的结果连接成可执行代码，主要是确定各部分的地址.将编译结果中的地址符号全换成实地址（call指令在a.cpp被编译时只是call f的符号，而不知道f确切的地址）

extern

lib文件：<https://wenku.baidu.com/view/9a6acf23482fb4daa58d4b35.html> c语言写的，c编译器编译的函数放在lib库，c++不能直接调用，而是要extern

原因：符号表机制

符号表：与编译的各个阶段都有交互，存有函数名、地址等信息；编译时会创建一个函数符号表 <name,address>，对应的符号后面的地址还没确定（link期决定），call name根据name找到符号表对应的地址，再执行

- 对于c语言来说，编译得到的符号表内函数f在符号表里的name就是f（不存在函数重载）
- 对于c++来说，因为有重载，所以f(int)和f(float)在符号表里的name是不同的
- c++对于c语言的函数f会按c++的方式生成函数表中的nameA，但c编译好的函数表内f对应的nameB和nameA不一致，导致c++无法找到该函数

- 符号表参考:<https://blog.csdn.net/wdjjwb/article/details/86233389>
<https://www.xuebuyuan.com/1131926.html>
- 加上extern "C"就表示按c的规则去翻译函数名f, 参考:
<https://www.cnblogs.com/yuxingli/p/7821102.html>

函数重载

函数重载匹配: http://www.360doc.com/content/18/0510/22/54097382_752843509.shtml

```
void bar(int i) {
    cout << "bar(1)" << endl;
}
void bar(const char c) {
    cout << "bar(2)" << endl;
}
void func(int a) {
    cout << "func(1)" << endl;
}
void func(char c) {
    cout << "func(2)" << endl;
}
void func(long long ll) {
    cout << "func(3)" << endl;
}
void hum(int i, ...) {
    cout << "hum(1)" << endl;
}
void hum(int i, int j) {
    cout << "hum(2)" << endl;
}
}
int main() {
    char c = 'A';
    bar(c);
    short s=1;
    func(s);
    hum(12, 5);
    hum(10, 12, 1);
    system("pause");
}
//输出结果为
//bar(2)
//func(1)
//hum(2)
//hum(1)
```

对于func(s)来说, 是将short转为int型从而调用func(2); ...参数的意思是在函数调用时不检查形参类型是否与实参类型是否相同, 也不检查参数个数 (当传入参数个数为两个时, 还是会调用hum(2))

重载后的调用可能会造成二义性: https://www.cnblogs.com/iloverain/p/5796932.html?utm_source=itdadao&utm_medium=referral

默认参数

默认参数声明

- 默认参数是严格按照**从右至左**的顺序使用
 - `void f(int a,int b=2,int c=2)` //正确
 - `void f(int a,int b=2,int c)` //错误
- 默认参数只能在原型或定义其中一处指定，不能同时指定

b.cpp `void f(int,int,char='a')`不会去找a.cpp中实现的函数?

inline 详细: <https://blog.csdn.net/fuzhongmin05/article/details/54615865>

源代码表示为函数形式，但内部实现机制不是函数调用，而是直接将内联函数代码复制到主文件执行

- 好处：避免函数传递的时间和空间开销
- 坏处：破坏复用；每调用一次内联函数就会将那段代码复制到主文件中，内存增加，内存调用时原本一页的内容可能出现在第一页+第二页的一部分，**造成操作系统的“抖动”**
- 内联的意思就是是消除函数的额外开销

```
inline int max(int a, int b)
{
    return a > b ? a : b;
}
//则调用: cout<<max(a, b)<<endl;
//在编译时展开为: cout<<(a > b ? a : b)<<endl;
//从而消除了把 max写成函数的额外执行开销
```

- 函数是否内联由编译器决定，使用inline只是发出一个请求希望内联
 - 复杂的函数、函数指针不能内联(指针所指向的函数在编译器也许会指定/但是在运行期也会发生改变,这种逻辑就不能内联了)
 - 一般针对使用频率高、小段、简单的代码才使用内联函数，比如构造函数（默认用内联）
- inline必须和**函数体**放在一起，而**不是和原型**放在一起；且函数体必须出现在调用之前，否则程序可以编译，但不会出现inline（若要使用inline，必须看到完整的内部结构，不能只看到函数声明）

```
◦ inline int f(int a,int b){
    return a+b;
}
int main(){
    f(1,2); //正确使用内联
}
```

- 函数在头文件中被多次include 的重定义问题:<https://blog.csdn.net/chunyexiyu/article/details/43673059>（可以用inline解决）


```

○ //common.h
inline int f(int a,int b){
    return a+b;
}
//a.cpp
#include "common.h"
void h(){
    f(2,3);
}
//b.cpp
#include "common.h"
void g(){
    f(1,2);
}

```

- 此时如果f被拒绝则会相当于重定义，导致报错（这也是inline被拒绝后被视作普通函数可能造成错误的情况）
- 可以将函数实现写在a.cpp，而声明写在同名的a.h文件中，这样b.cpp只需要include a.h就能调用a.cpp中实现的函数

9.26

程序结构

- 逻辑结构：相互关联的函数/文件形成的抽象上的图结构
- 物理结构：考虑到每个函数存储的位置，不一定存在同一个cpp
- 一个源程序不论有多少个源文件，只有一个源文件能包含main函数
- lib：obj的容器，参考：<https://wenku.baidu.com/view/9a6acf23482fb4daa58d4b35.html>

作用域

- 程序级：一个程序可能有很多源文件，程序级就是说整个程序可见
- 文件级：同一个文件中可见
- 函数级：同一个函数内可见
- 块级：同一个代码块内可见，比如while循环中定义的一个变量

extern另一个用法 修饰变量或函数：用来说明此变量/函数是在别处定义的，要在此处引用，而不会新开辟一块内存

- 该变量或函数必须是唯一的，如果多个cpp文件中都有定义则会报错
- 为什么可以把声明放在头文件中？
 - 编译阶段，B虽然找不到该函数或变量（理解为符号表的信息不完整），但连接时会从A生成的目标代码中找到此函数
 - **a.cpp和a.h并没有联系**，link时会从所有编译好的文件中找B需要的符号，如果几个不同文件中实现了同一个函数/定义了同一个全局变量，就会报错
- 一般不用extern修饰常量，因为常量默认作用域在一个文件内
 - 使用extern const：<https://blog.csdn.net/liuhhaiffeng/article/details/77234103>

函数间的调用

- a.cpp中实现了 `int f(int a){return a;}`, b.cpp声明`int f(int)`, 再直接调用
 - `extern int f(int)`和`int f(int)`: 函数的声明默认是extern的, 带有关键字仅仅是语义上按时这个函数可能在其他源文件里有定义
- 实现不变, `extern int f(int)`或`int f(int)`可放入.h文件内由b.cpp引入, 效果和前者一样

static

- 修饰全局变量时, 表明一个全局变量只对定义在同一文件中的函数可见。
 - 其他文件中可以定义相同名字的变量
- 修饰局部变量时, 表明该变量的值不会因为函数终止而丢失。
 - 该变量在全局数据区分配内存(局部变量在栈分配内存)
 - 始终驻留在全局数据区直到程序运行结束, 但作用域为局部作用域, 不能再函数外访问
- 修饰函数时, 表明该函数只在同一文件中调用。
 - 其他文件中可以定义相同名字的函数
- 修饰类的数据成员, 表明对该类所有对象这个数据成员都只有一个实例。即该实例归 所有对象共有

头文件: 除了可以放常量、变量/函数的声明/内联函数, 还可以放类的声明 (类的定义在.cpp实现):

https://blog.csdn.net/qj_35779286/article/details/94169434

- 头文件预编译: 把一个project中经常使用的、不会被频繁修改的头文件预先编译, 以后编译该工程时, 不再编译这部分代码和头文件, 仅仅使用预编译的结果(.pch文件), 这样可以加快编译速度

namespace

问题: 如何解决全局变量/函数名的冲突?

- 编写的函数带上姓名: `Zhang_f`: 不方便操作, 每定义一个就要添加姓名前缀
- 带参数的宏
 - `define Pre(x) Zhang_**x Zhang_f=>Pre(f) Zhang_f(1)=>Pre(f)(1)`
- 将全局变量/函数名纳入一个类中: 违背了类的设计思想, 将类当成了工具而不是直接用于解决问题
- namespace: 借助了类的思想


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

namespace A
{
    int a = 0;
    namespace D
    {
        int b = 1;
    }
}
namespace B
{
    int a = 2;
```

```

    namespace C
    {
        int b = 4;
    }
}
int main() {
    using namespace B;
    using namespace C;
    cout << b; //必须using B,C否则b会报错
    getchar();
}

```

- using-declaration: 指定使用L中的哪些变量/函数 using L:k
- using-directive: 使用L中的全部内容 using namespace L
 - L后期添加的内容可能会和现有内容造成冲突
- 1569555119925
 - 自定义namespace被其他文件调用: <https://blog.csdn.net/Kwansy/article/details/82937900>
 - namespace A,B都定义了变量b, 则使用b会报错 (ambiguous)
- ##拼接 #@转为char #转为string

10.8

数组

- 特征
 - 相同类型, 比如: int A[12]表示定义了一个长度为12的类型都是int的数组 (sizeof(B) = 12 * sizeof(int))
 - **连续存储**, 0 ~ n - 1, C++没有iterator, 必须连续存储
- 一维数组
 - 函数接口写法, void print(int B[], int n), 获得数组长度的一般方法: sizeof(B) / size(B[0])
 - 元素个数通过参数显式给出, 不能通过sizeof取得
 - 调用方式 print(A, n)
 - 字符数组特例
 - 接口不需要传递元素个数n, void print(char a[])
- 多维数组
 - 定义
 - T B[c1] [c2]代表T[c2] B[c1], 不要把二维数组想象成矩阵!

- ```

int B[2][6];
typedef int T[6];
T b[2]; //等价于int B[2][6]
int C[2][3][2];
typedef int T2[2];
typedef T2 T1[3];
T1 C[2]; //等价于int C[2][3][2]

```

- 无论维度多大，都是相同类型元素的连续排列；使用多维数组的原因是与应用空间有关

- 参数传递

- 缺省第一维，void f(int a[] [3], int n)

## struct

- 可以做赋值操作--同类型，大块数据的传输

- struct按不同顺序排列sizeof会不一样

- ```

struct A {
    char a;
    int b;
    short c;
}
cout << sizeof(A) << endl; //此时sizeof(A) = 12
struct B {
    short c;
    char a;
    int b;
}
cout << sizeof(B) << endl; //此时sizeof(B) = 8, 因为short占2个字节，四个字节并未占满，所以还可以填入一个char，但是再遇到int（4个字节）就不能再填充了，只能从写一个四字节地址开始，所以共占8个字节

```

- struct必须进行声明

```

//方式一
struct A {};
//方式二
struct {} B;
//方式三
typedef struct {} C;

```

union

- 共享存储空间（公共的容器），能够节省空间，union变量占用的内存长度等于最长的成员的内存长度，所有成员的起始地址是一样的，故下面的&test.b &test.a &test.c是一样的

```

    union B {
        char b;
        int a;
        short c;
    } test;
    cout << sizeof(B) << endl; //输出存储空间最大的一位，即输出max(sizeof(b),
    sizeof(a), sizeof(c))

```

union的内存分配问题<https://blog.csdn.net/xiajun07061225/article/details/7295355>

公用变量体中起作用的是最后一个存放的成员，存入一个新成员，原有的成员失去作用

<https://www.jb51.net/article/56009.htm>

不能把union变量作为函数参数，也不能使函数带回union变量

- 应用

- 矩阵

```

    union Matrix {
        struct {
            double _a11, _a12, _a13;
            double _a21, _a22, _a23;
            double _a31, _a32, _a33;
        }
    }
    Matrix m;
    int i,j;
    for (i = 0; i < 3; i++)
        for (j = 0; j < 3; j++)
            m._element[i][j] = (i + 1) * (j + 1)

```

指针

- 最基本的作用：管理地址信息

- 管理数据
- 调用代码

- 定义

- 格式：<基类型> *<指针变量> 例：int *p;任何一种类型都能赋值给void *，void *是所有指针类型的公共接口

```

void *p;
char *cp;
int *ip;

```

```
p = cp; //任何类型的指针都能赋值给void类型，因为它不会检查内部数据类型（不会进房间，只看门牌号）
```

- 使用typedef定义一个指针类型

```
typedef int* pointer;
pointer p, q; //p, q均是指针变量
```

• 操作符

- &取地址，*间接取内容

```
int x = 9;
int *p;
p = &x;
*p = 1000; //p存储的是x的内容，*p间接获得x的内容
//x的地址是0X5212，x的内容是9，p的地址是0X7B77，p的内容是x的地址0X5212，*p就是x的内容
```

- NULL和nullptr区别，在C++中，NULL代表0，nullptr代表空指针

```
#include<iostream>
using namespace std;
void func(int num) {
    cout << "This is a NULL example" << endl;
}
void func(int *num) {
    cout << "This is a nullptr example" << endl;
}
int main() {
    func(NULL);
    func(nullptr);
    getchar();
}
// 输出结果为
// This is a NULL example
// This is a nullptr example
```

• 指针的运算

- 整型 +/-

- 结果类型不变，数值：sizeof(基类型) * 整型数值

```
int A[12];
int *p = NULL;
p = &A[0];
for (int i = 0; i < 12; i++) {
    cout << *p << endl;
    p++;
    //另一种输出 cout << *(p + i) << endl;
}
```

- 同类型指针相减

- 结果类型：整型，数值：偏移量(值差 / sizeof(基类型))

```
int *p, *q, offset;
int A[12];
p = &A[0];
q = &A[3];
offset = q - p; //offset = 3
```

- 指针的比较

- 同类型指针比较一般只用 == 和 !=

- 指针的输出

- 对于cout来说char *是特殊的，发生了重载

```
int x = 1;
int *p = &x;
cout << p << endl; // 输出p的值即x的地址
cout << *p << endl; // 输出p指向的元素值，即输出x

char *p = "ABCD";
cout << p << endl; // 输出p指向的字符串，即ABCD
cout << *p << endl; // 输出p指向的字符，即A
cout << (int *)p // 指向p的值
```

- **void ***

- 只管理地址信息
- 指针类型的公共接口，任何操作须做强制类型转化

```
void *any_pointer;
int x;
int y;
any_pointer = &x;
```

```
any_pointer = &y;

((int *)any_pointer);//正确, 需要做类型转化
*any_pointer;//错误
```

memset()函数, memset(数组名,要填充的值,字节数), memset()的函数原型是void *memset(void *s, int v, size_t n);

• 常量指针

- 一般形式 const <类型> * <指针变量>

```
const int c = 0;
const int *cp;
int y = 1;
int *q;
cp = &c;//√
q = &y;//√
*cp = 1;//×
*q = 2;//√
cp = &y;//√
q = &c;//×
```

const int *cp只有读的权限, 没有写的权限, 所以 *cp = 1是错误的

可以理解为const限制操作为读操作, 一定程度上保护了数据的安全

不能把一个有读写权力的指针赋值给只有读权力的地址

const_cast取消常量属性, 但这是不安全的, 所以最后一个例子应该改写为q = const_cast<int *>(&c);

```
const int c=128;
int * q = const_cast<int *>(&c);
*q = 111;
cout << " c " << &c << c << endl;
cout << " q " << &q << q << endl;
cout << "*q " << q << *q << endl;
//c      0012FF74      128
//q      0012FF70      0012FF74
//*q     0012FF74      111
```

编译器编译时已经把c换成字面常量128

```
void(int A[],int n);//此时的A是一个指针常量
void g(){
    int a[12];
```



```
f(a,12);
//此时的a已经转化为一个int *const表达式，失去了原来数组的含义
}
```

```
int a[12];
int *p = &a[0];
for(int i = 0; i < 12; i++){
    *(p++) = 0;
    *(a + i) = 0;
    *(a++) = 0; //该写法错误
}
```

二维数组

```
int a[6][2];
int *p = a[0]; //p指向一个一维数组
for(int i = 0; i < 12; i++){
    *(p+i) = 8; //此写法不对，发生越界
}
```

• 指针常量

- 一般形式 <类型> * const <指针变量> 必须在定义时初始化

```
int x,y;
int * const p = &x;
p = &y; //错误
*p = 1;
```

p被创建后只能固定地指向x，即p中存储的是x的地址，不能再存储其他变量的地址

malloc()函数，原型void* malloc(size_t size)，size以字节为单位

```
char *tmp = (char *)malloc(width)
```

• 指针数组

- 数组中的指针为指针

```
char *s1[] = {"C++", "PASCAL", "FORTRAN"};
```

- **printf宏定义说明**

- `*typedef char va_list`: 为了和下面保持对应, 所以将char *定义为va_list
- `#define _INTSIZEOF(x) ((sizeof(x) + sizeof(int) - 1) & ~(sizeof(int) - 1))`: 考虑地址对齐 (char 1个byte int 4个byte double 8个byte), 此处以int为标准, 如果 $1 \leq \text{sizeof}(x) \leq 4$, 则 $\text{INTSIZEOF}(n)=4$, 若 $5 \leq \text{sizeof}(n) \leq 8$, 则 $\text{INTSIZEOF}(n)=8$, 以此类推

假设char *的长度为x

$x = Q * n + r (-n < r \leq 0)$ 目的是求出Q, 这样Q占用长度为n的空间, n是2的整数次幂

$x + n - 1 = Q * n + r' (0 \leq r' < n)$

故 $Q = [(x + n - 1) / n] * n$

如何使用位运算表示

考虑到 $n = 2^m$, 所以相当于x先右移m位再左移m位, 相当于和 $111000\dots 0$ (m个0)做与运算

$111000\dots 0$ (m个0)可以表示为 $\sim(n - 1)$

- `#define va_start(ap,v) (ap = (va_list)&v + _INTSIZEOF(v))`: 这里的&v是最后一个固定参数的起始地址, 再加上其实际占用大小后, 就得到了第一个可变参数的起始内存地址
- `#define va_arg(ap,t) (*(t *) (ap += _INTSIZEOF(t) - _INTSIZEOF(t)))`: 必须先转化为t类型的指针再取值, 否则直接取值不知道指针类型所以不知道取几个单元的内存作为值
- `#define va_end(ap) (ap = (va_list)0)`: 使ap不再指向堆栈, 而是跟NULL一样, 成为空指针

- **字符串类型的swap函数**

- ```
void swap(char **a, char **b){
 char *tmp = *a;
 *a = *b;
 *b = tmp;
}
//附: int型的swap函数
void swap_int(int *a, int *b){
 int tmp = *a;
 *a = *b;
 *b = tmp;
}
```

## 11.5

### 面向对象

#### 类

- 类的定义和声明是分开的, 类声明的头文件仅仅给出接口部分

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

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

## • 构造函数

- 对象的**初始化**，构造函数可以被定义，也可以只声明

```
class Line
{
public:
 void setLength(double len);
 double getLength(void);
 Line(); // 这是构造函数

private:
 double length;
};

// 成员函数定义，包括构造函数
Line::Line(void)
{
 cout << "Object is being created" << endl;
}
void Line::setLength(double len)
{
 length = len;
}
double Line::getLength(void)
{
 return length;
}

// 程序的主函数
int main()
```

```

{
 Line line;
 // 设置长度
 line.setLength(6.0);
 cout << "Length of line : " << line.getLength() << endl;
 return 0;
}
//输出结果为
//Object is being create
//6.0

```

#### ◦ 描述

- 与类同名，**无返回类型**，构造函数可以有参数，初始化要带参数，比如上面的Line类的构造函数可以改为Line(double len)，构造函数的定义改为将length初始化为len
- 自动调用，不可以直接调用

```

class A{
public:
 A();
 A(int i);
 A(char *p);
};
A a1 = A(1); //等价于A a1(a) //等价于A a1=1;自动调用A(int i);
A a2 = A(); //调用A()
A a3 = A("abcd"); //等价于A a3("abcd") //等价于 A a3 = "abcd" //调用A
(char *p)
A a[4];
A b[5] = {A(), A(1), A("abcd"), 2, "xyz" };

```

- 可重载
- 当类中没有提供构造函数时，编译系统自动提供
- 一般声明为public，但也可以定义为private

#### • 成员初始化表

- 构造函数的补充
- 先于构造体函数执行，按类数据成员声明次序，减轻compiler的负担

```

class A
{
 int x;
 const int y;
 int& z;
public:

```

```
A(): y(1),z(x), x(0) { x = 100; }
};
```

- 注意

- 再构造函数中尽量使用成员初始化表代替赋值动作
- 数据成员太多时，不要使用，降低了可维护性

- 析构函数

- 结构：~类名()
- 对象消亡时，系统自动调用，释放对象持有的非内存资源
- 一般声明在public中，也可以声明在private中
- 没有返回值，不接受参数（不能重载）

- 拷贝构造函数

- 默认拷贝构造函数执行的是浅拷贝，一旦对象存在了动态成员，浅拷贝会出问题。此时要使用深拷贝
- 参考网址：<https://www.cnblogs.com/alantu2018/p/8459250.html>
- 例

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

```
void main1(){
 string s1("abcd");
 string s2=s1;
```

//浅拷贝：s2和s1这两个指针指向堆里的同一空间，再销毁对象时，两个对象的析构函数将同一个内存空间释放两次，这就是错误所在

```
}
class string{
private:
 char *p;
public:
 string(const string & c){
 p=new char[strlen(c.p)+1];
 strcpy(p,c.p);
 }
 ~string(){
```

```
 delete[] p;
 }
}
void main2(){
 string s1("abcd");
 string s2=s1;
 //深拷贝：s1和s2各自指向一段内存空间，他们指向的空间具有相同的内容
}
```

## 11.12

- 动态对象

- 在heap（堆）中创建
- new/delete，使用new和delete可以自动调用constructor和destructor
- malloc和free不调用析构函数

- const成员

- 静态成员

- 静态成员变量：静态成员变量属于类，不属于某个具体的对象，即使创建多个对象，也只为静态成员变量分配一份内存，所有对象都是这份内存中的数据，参考网址：

<http://c.biancheng.net/view/2227.html>

- 静态成员变量必须在类声明的**外部初始化**

```
type class::name = value;//具体实例见我的clion
```

- 静态成员函数：静态成员函数只能访问静态成员（变量，函数）；静态成员函数与普通成员函数的根本区别在于：普通成员函数有 **this 指针**，可以访问类中的**任意成员**；而静态成员函数没有 this 指针，只能访问静态成员（包括静态成员变量和静态成员函数）。

## 11.19

- 友元

- 分类

- 友元函数
- 友元类
- 友元类成员函数

- ```
void func();
class B;
```

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

特性

友元不具有传递性

```
class Matrix{
    friend void multiply(Matrix &m, Vector &v, Vector &r);
};
class Vector{
    friend void multiply(Matrix &m, Vector &v, Vector &r);
};
//定义在两个类中的友元是单独针对Matrix或者Vector类的，两个不互通
```

- 上述代码不能编译通过，需要进行**前置声明**，在最开头加上一句**void multiply(Matrix &m, Vector &v, Vector &r);**即可
- C++11之后可以访问private成员

原则

- 避免将数据成员放在公开接口中
- 努力将接口完满且最小化

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

作用

- 提高程序设计的灵活性
- 数据保护和对数据的存取效率之间的一个折中方案

- 继承

- 单继承

- 子类无法访问父类的protected元素
- 子类无法继承父类的构造函数，友元函数，析构函数

- ```
class Student{
 int id;
 public:
 char nickname[16];
 void set_ID(int x) { id = x; }
 void SetNickName(char* s) { strcpy(nickname,s);}
 virtual void showInfo()
 { cout << nickname << ":" << id <<endl; }
 //virtual使得函数可以被动态重载
};
//父类的变量子类同样拥有，可以通过sizeof验证
//子类中父类已经定义元素的访问权限与父类相同
class Undergraduated_Student : public Student{
 int dept_no;
 public:
 void setDeptNo(int x) { dept_np = x; }
 void set_ID(int x){
 //函数重定义（静态的，并非动态），但不是同一个名空间
 }
 void showInfo(){
 cout << dept_no << ":" << nickname << ":" << id <<endl;
 }
 private:
 using Student::nickname; //Undergraduated_Student的派生类不能再访问nickname，由于权限变成private
 void SetNickName();
 //编译器：1.找名字（可以调用则调用，否则不调用）；2.在子类中找到了SetNickName的名空间，就不会去父类找（名空间覆盖）；3.子类SetNickName名空间没有匹配char* s的函数，因此调用失败
};
```

- 重定义，重载和重写的区别

- 重载overload：是函数名相同，参数列表不同，重载只是在类的内部存在，但是不能靠返回类型来判断，若基类的虚函数值给出声明没有给出实现，子类在使用该函数时为重载（函数名相同，参数类型不同）
- 重写override：子类重新定义父类中有相同名称和参数的虚函数；被重写的函数不能是static类型必须是virtual；重写函数必须有相同的类型、名称和参数列表；重写函数的访问修饰符可以改变，尽管virtual是private，但是派生类中重写可以改写为public，甚至protected也是可以的（protected的作用是保护变量和函数，类对象不能直接访问自己的保护变量，只能通过类函数来访问；保护变量可以被派生类的成员访问，但是不能被派生类的对象直接访问）



- 重定义redefining：是指派生类的函数屏蔽了与其同名的基类函数；如果派生类的函数和基类的函数同名，但是参数不同，此时，不管有无virtual，基类的函数被隐藏；如果派生类的函数名和参数都和基类相同，但是基类函数没有virtual关键字，此时基类的函数被隐藏

```
class Base{
private:
 virtual void display(){cout << "Base display()" << endl;}
 void say(){cout << "Base say()" << endl;}
public:
 void exec(){display();say();}
 void f1(string a){cout << "Base f1(string)" << endl;}
 void f1(int a){cout << "Base f1(int)" << endl;}//重载，两个f1函数在Base类的内部被重载
}
class DeriveA:public Base{
public:
 void display(){cout << "DeriveA display()" << endl;}//重写
 void f1(int a,int b){cout << "DeriveA f1(int,int)" << endl;}//重定义
 void say(){cout << "DeriveA say()" << endl;}//重定义
}
class DeriveB:public Base{
public:
 void f1(int a){cout << "DeriveB f1(int)" << endl;}//重定义
}
int main(){
 DeriveA a;
 Base *b=&a;
 b->exec();//调用DeriveA的display()方法以及Base的say()方法
 a.exec();//调用DeriveA的display()方法以及Base的say()方法
 a.say();//调用DeriveA的say()方法
 DeriveB c;
 c.f1(1);//调用DeriveB的f1()方法
}
```

- 关于static有几点要说
  - 静态数据成员和静态成员函数都可以被继承
  - 父类的static变量和函数在派生类中依然可用，但是受访问控制（比如，父类的private域中的就不可访问），而且对static变量来说，派生类和父类中的static变量是共用空间的，这点在利用static变量进行引用计数的时候要特别注意
  - static函数没有虚函数一说，static函数实际上是加上了**访问控制的全局函数**，全局函数没有虚函数

```
class A{
public:
 static int num;
};
```

```
int A::num=100;
class B:public A{
public:
 int i;
 B(int m):i(m){}
};
//int B::num=200;
```

A,B中num值相等，num的地址也相等，故父类子类指向是同一个全局数据区的static变量，此外，注释的那句话如果不添加注释会发生编译错误

- 构造函数的执行次序
  - 基类构造函数
  - 派生类对象成员类的构造函数
  - 派生类构造函数
- 析构函数的执行次序
  - 与构造函数相反
- 基类构造函数的调用

```
class A{
 int x;
public:
 A() { x = 0; }
 A(int i) { x = i; }
};

class B: public A{
 int y;
public:
 B() { y = 0; }
 B(int i) { y = i; }
 B(int i, int j):A(i)
 { y = j; }
};

B b1; //执行A::A()和B::B()
B b2(1); //执行A::A()和B::B(int)
B b3(0,1); //执行A::A(int)和B::B(int,int)
```

## 11.21

### • 虚函数

- 类型相容、赋值相容

Q: a、b是什么类型时，a = b合法？

```
A a; B b;
class B: public A{};
a = b;
// 对象切片，取出b的每个对象按位拷贝到a，拷贝之后b属于类A，去掉专属于类型B的属性
```

- 对象的身份发生变化
- 属于派生类的属性已经不存在

```
B pb;
A* pa = &pb;
class B: public A

B b;
A &a=b;
class B: public A
```

- 对象的身份没有发生变化

◦ 前期绑定-->默认

- 编译时刻
- 依据对象的静态类型
- 效率高但是灵活性差

◦ 动态绑定--> 显示指出 virtual

- 运行时刻
- 依据对象的实际类型（动态）
- 灵活性高、效率低

◦ 限制

- 类的成员函数才可以是虚函数
- 静态成员函数不能是虚函数
- 内联成员函数不能是虚函数
- 构造函数不能是虚函数
- 析构函数可以（往往）是虚函数

◦ 后期绑定的实现

```
class A{
 int x,y;
public:
 virtual f();
 virtual g();
```

```

 h();
 };
 class B:public A{
 int z;
 public:
 f();
 h();
 }
 A a;
 B b;
 A *p;
 p=&a;//情况1
 p=&b;//情况2
 //情况1的数据对象为x和y, 虚函数表为A::f和A::g
 //情况2的数据对象为x、y和z, 虚函数表为B::f和A::g

```

```

class A{
public:
 A(){ f();}
 virtual void f();
 void g();
 void h() { f(); g(); }
};
class B: public A{
public:
 void f();
 void g();
};
B b;//A::A(),A::f,B::B()
A *p=&b;
p->f(); //B::f
p->g(); //A::g
p->h(); //A::h,B::f,A::g

```

直到构造函数返回之后，对象方可正常使用

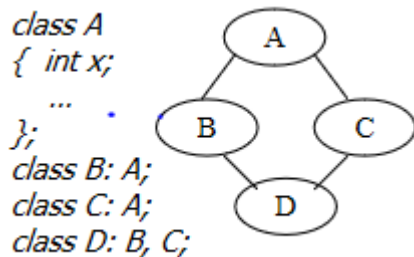
- 纯虚函数和抽象类
  - 纯虚函数
    - 声明时在函数原型后面加上 = 0
    - 往往只给出函数声明，不给出实现
  - 抽象类
    - 至少包含一个纯虚函数
    - 不能用于创建对象
    - 为派生类提供框架，派生类提供抽象基类的所有成员函数的实现
- 绝对不要重新定义继承而来的缺省参数值

## 多继承

- 继承方式等都与单继承一样，只是继承的基类数目大于1
- 多继承下，构造函数的执行次序为类声明的次序，析构函数与构造函数执行次序相反
- 当两个或多个基类中有同名的成员时，如果直接访问该成员，就会产生命名冲突，编译器不知道使用哪个基类的成员。这个时候需要在成员名字前面加上类名和域解析符::，以显式地指明到底使用哪个类的成员，消除二义性。参考网址：<http://c.biancheng.net/view/2277.html>

```
E:\C++\code\cmake-build-debug\code.exe
BaseA constructor
BaseB constructor
Derived constructor
1 2 3 4 5
Derived destructor
BaseB destructor
BaseA destructor
```

## 虚继承



- - 类D拥有两个x成员，B::x和C::x
  - 虚基类：合并，B和C共同指向A，当要访问x的时候，会去A中找x，不存在副本的问题

```
class A;
class B:virtual public A;
class C:public virtual A;
class D:B,C;
```

- 虚基类的构造函数由最新派生出的类的构造函数调用，比如上例中，先调用D的构造函数，再调用C，B
  - 虚基类的构造函数优先非虚基类的构造函数执行
- 虚继承的产生解决了冲突的问题，消除成员访问的二义性，但是需要全局考虑

## 面向对象程序设计

### 多态

- 同一论域中一个元素可以有多种解释
- 程序设计语言
  - 一名多用---函数重载
  - 类属---template
  - OO程序设计---虚函数

## 12.5

---

### 操作符重载

参考网址: <https://www.runoob.com/cplusplus/cpp-overloading.html>

- 操作符重载的动机: 赋予运算符新的功能
- 注意: 至少要包含一个用户自定义类型; **永远不要重载 && 和 ||**; 尽可能让事情有效率, 但是不要过度有效率
- 格式
  - **返回值类型 operator 操作符(形参){}**
- 双目操作符重载
  - 隐含this
  - 使用方法

```
<class name> a,b;
a 操作符 b;
a.operator操作符(b);
两种写法等价
```

- 全局函数
  - 友元函数  
friend 返回值类型 operator 操作符(形参)
  - 限制: = () [] 不能用作全局函数重载
- 单目操作符重载
  - 隐含this
  - 格式  
**返回值类型 operator 操作符 ()**
- 一些操作符重载
  - + 操作符重载

```
class Complex{
 double real,img;
public:
 Complex(){real=0;img=0;}
 Complex(double r,double i):real(r),img(i){}
 friend Complex operator+ (Complex& x,Complex& y){
 Complex temp;
 temp.real=x.real+y.real;
 temp.img=x.img+y.img;
 return temp;
 }
};

int main(){
 Complex a(1,2),b(3,4),c;
 c=a+b;//c.real=5,c.img=5
}
```

---

## 期末复习

---

### 以pass-by-reference-to-const代替pass-by-value

对于下面这段代码，如果使用值传递的话，s1被创建的时候，先调用父类的默认构造函数，再调用子类自己的默认构造函数；当把s1以值传递到函数f中时，先调用父类的默认构造函数，再构造子类的拷贝构造函数，执行完之后，临时变量s被析构，先调用子类的析构函数，再调用父类的析构函数

```
class person {
public:
 person() {
 cout << "father construct func" << endl;
 }
 person(const person& p) {
 cout << "father copy func" << endl;
 }
 ~person() {
 cout << "father xigou func" << endl;
 }
private:
 string name;
 string address;
};

class student :public person {
public:
 student() {
 cout << "son construct func" << endl;
 }
 student(const student& p) {
 cout << "son copy func" << endl;
 }
}
```

```
 }
 ~student() {
 cout << "son xigou func" << endl;
 }
private:
 string school;
 string schoolAddr;
};
void f(student p) {
 cout << "hh" << endl;
}
int main()
{
 student s1;
 f(s1);
 system("pause");
 return 0;
}
```