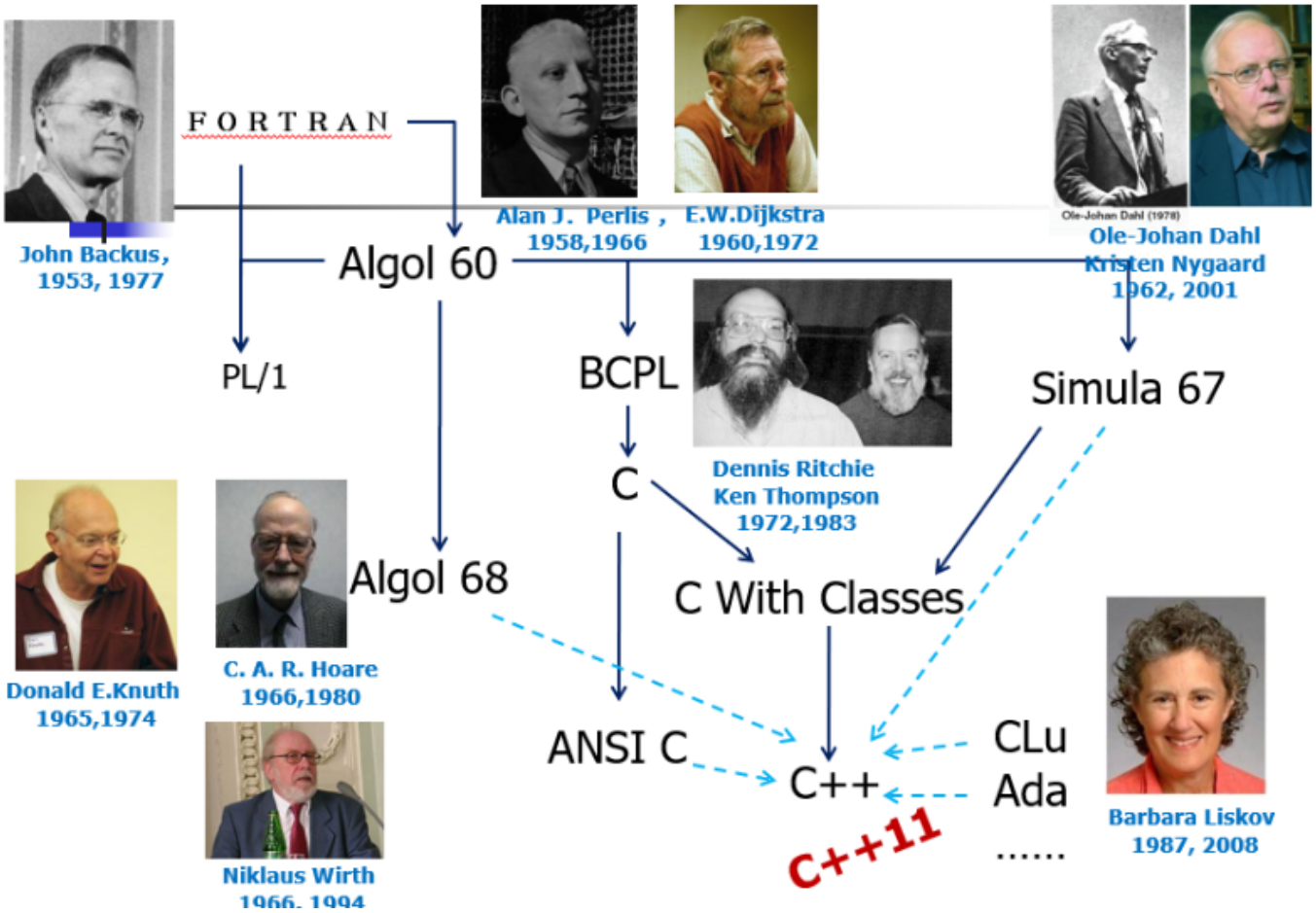
# C++ Collection By $YDJSIR$ 2021

> 以下内容大部分整理自
>
> https://github.com/Software-Knowledge/2020-C-plus-plus-advanced-programming
>
> 与课程PPT等
>
> YDJSIR基本只是改typo与重排，但也加入了很多自己的理解。

# C++历史与理念




## 目的

更高效地进行编码

## 设计理念

- 效率；
- 实用性优于艺术性严谨性；
- 相信程序员允许一个有用的特征比防止各种错误使用更重要（相信程序员）；

## 重要人物

| 人 | 成就 | 备注 |
| --- | --- | --- |
| Kristen Nygaard | Simula67 | 创建 |
| Ole-Johan Dahl | OO 编程 | 创建 |
| Dennis Ritchie、Ken Thompson | C语言 | 创建 |
| Bjarne Stroustrup | C with Classes | 1980形成，并最终发明C++ |

| 人 | 成就 | 备注 |
| --- | --- | --- |
| Rick Mascitti | C++ | 1983正式命名 |
| Dijkstr | 结构化编程 | 创建 |

## 三条主要脉络

### 脉络一：Algol 68：结构化编程的部分的继承

- Niklaus Wirth：发明了PASCAL，很适用于教学；
- C. A. R. Hoare；
- Donald E.Knuth：Dijkstra一同提出goto有害性；
- 继承下来:关于结构化编程的特性；

### 脉络二：系统化编程的继承

- BCPL:贴近计算机，写出高效的程序，很好的想法：将IO作为类成分而不是语言成分，以提高语言可移植性；
- 在BCPL和C之间还有B语言，B语言是将BCPL里面的比较繁杂的部分取出；
- Dennis Ritchie、Ken Thompson，compiler决定程序语义和性质；
- 继承下来:关于系统编程的特性；

### 脉络三：Simula 67 第一个OO的研究(OO部分的继承)

- OO的第一个提出人：Ole-Johan Dahl、Kristen Nygaard；
- 继承下来:关于面向对象编程的特性；
- Barbara Liskov：关于高层复用做出很大的贡献；

## C与C++的关系

1. 代码层面C++完全包含了C语言成分，支持C支持的全部编程技巧，C是建立C++的基础，同时C++还添加了OOP的完全支持。
2. 运行任何C程序都能被C++用基本相同的方法编写，并具有相同的运行效率和空间。
3. 功能C++还引入了重载、内联函数、异常处理等功能，对C中过程化控制及其功能并扩充。

## 为什么不叫D

因为并没有完全抛弃C中的很多东西。而且事实上的确有一门D语言，但是并没有什么人用。C与C++已经被无数应用所验证，仍具有顽强的生命力，将其全盘抛弃后新开一门语言是不划算的。

## C++编译与链接过程

1. C++源代码想通过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。
2. 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。

# C++语言特性

## C++是强类型语言、动/静结合

类型安全不能代替测试。

1. 弱类型允许隐式转换
2. 动：编译时决定类型，静：编译前决定类型。

## 注解

对于char、int、float、double(四种基本数据类型)和修饰符(long、short、signed、unsigned):

1. char只可以被signed和unsigned修饰
2. float不能被修饰
3. double只能用long修饰
4. int可以被四种修饰符组合修饰

## `sizeof`：返回字节为单位的对应单位的大小。

## `typedef`：可以有效地提高系统的可移植性。

## 枚举常量

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，可以today = weekday(4),其中weekday是预定义好的枚举类。

# 运算符

## 自增量运算符

```
int main() {
    int a = 1;
    int b, c, d, e;
    cout << a << endl;//1 a = 1
    b = a++;
    cout << b << endl;//1 b = 1 a = 2
    c = a--;
    cout << c << endl;//2 c = 2 a = 1
    d = ++a;
    cout << d << endl;//2 d = 2 a = 2
    e = --a;
    cout << e << endl;//1 e = 1 a = 1
}
// 注意到++a返回的是左值，他自己，a++返回的是右值，返回的是还没有 + 的值
```

## 条件运算符:`<exp1>?<exp2>:<exp3>`

1. 唯一的三目运算符,不允许进行**重载**
2. 如果`<exp2>`和`<exp3>`的值类型相同且均为左值，则该条件运算符表达式为左值表达式。

3. 可以嵌套(满足就近原则)

# 逗号表达式

按照顺序，连续进行运算,格式形如：<exp1>,<exp2>,...,<expn>,<expn>的值是逗号表达式的值,如<exp n>为左值，则该逗号表达式为左值

```
int a,b,c;
d = (a = 1,b = a + 2,c = b + 3)
//a = 1
//b = 3
//c = d = 6
```

# 异或操作

1. 与全0的二进制串进行运算：不变
2. 与全1的二进制串进行运算：取反
3. 与本身的运算：清零
4. 与同一个对象进行异或运算两次：还原，应用:进行加密

## 补充：交换x和y

```
//允许中间变量
int t = x;
x = y;
y = t
//不允许中间变量
a = a ^ b
b = b ^ a
a = a ^ b
//或者
x = x + y
y = x - y
x = x - y
```

# 左值和右值

## 左值表达式

引用内存位置的表达式称为“左值”表达式，即具有可通过编程方式访问正在运行的程序的存储地址。 左值表示存储区域的“locator”值或“left”值，并暗示它可以出现在等号（=）的左侧。 左值通常是标识符。

引用可修改的位置的表达式称为“可修改的左值”。 可修改的左值不能具有数组类型、不完整类型或包含 const 特性的类型。 要使结构和联合成为可修改的左值，它们必须没有任何包含 const 特性的成员。 标识符的名称表示存储位置，而变量的值是存储在该位置的值。

> 以上内容来自微软文档

### 右值表达式

术语“右值”有时用于描述表达式的值以及将其与左值区分开来。

所有左值都是右值，但并不是所有右值都是左值。

C++11新引入了右值引用，在C++11部分再行展开。

## 常用关键字

this关键字：可以用来访问自己的地址。

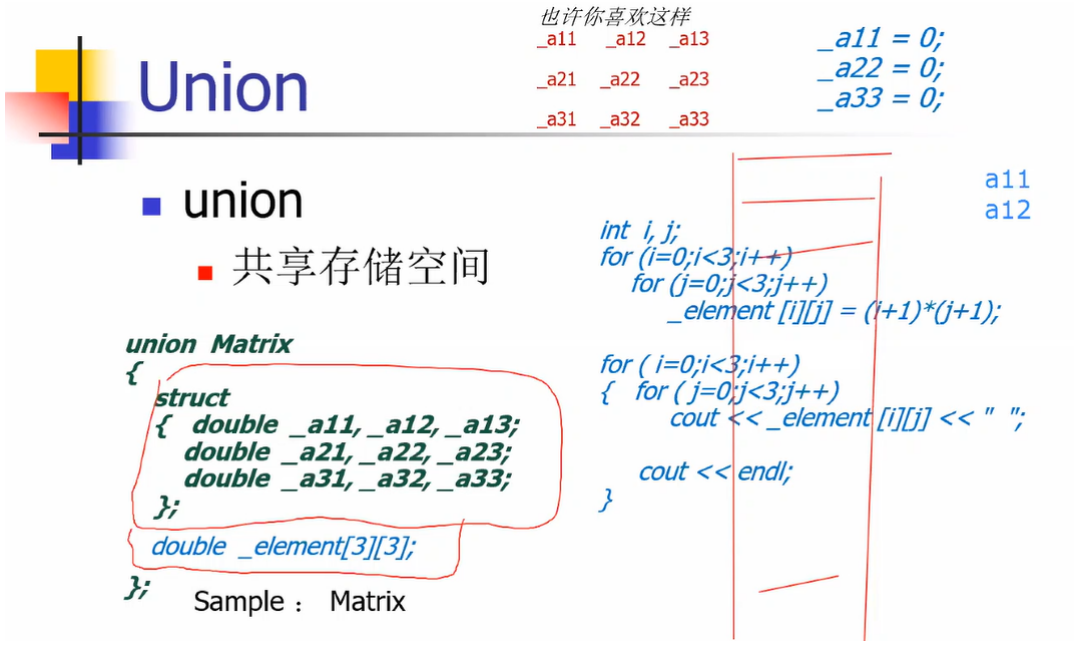
static

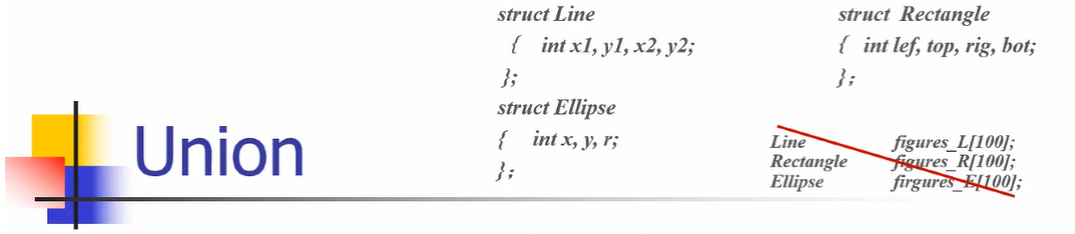
1. 全局有效：函数释放后也不会释放自身空间。
2. static的成员函数，需要在声明的时候进行修饰，但是没有this指针

const对象在对应声明周期中是常量

# struct与union

union

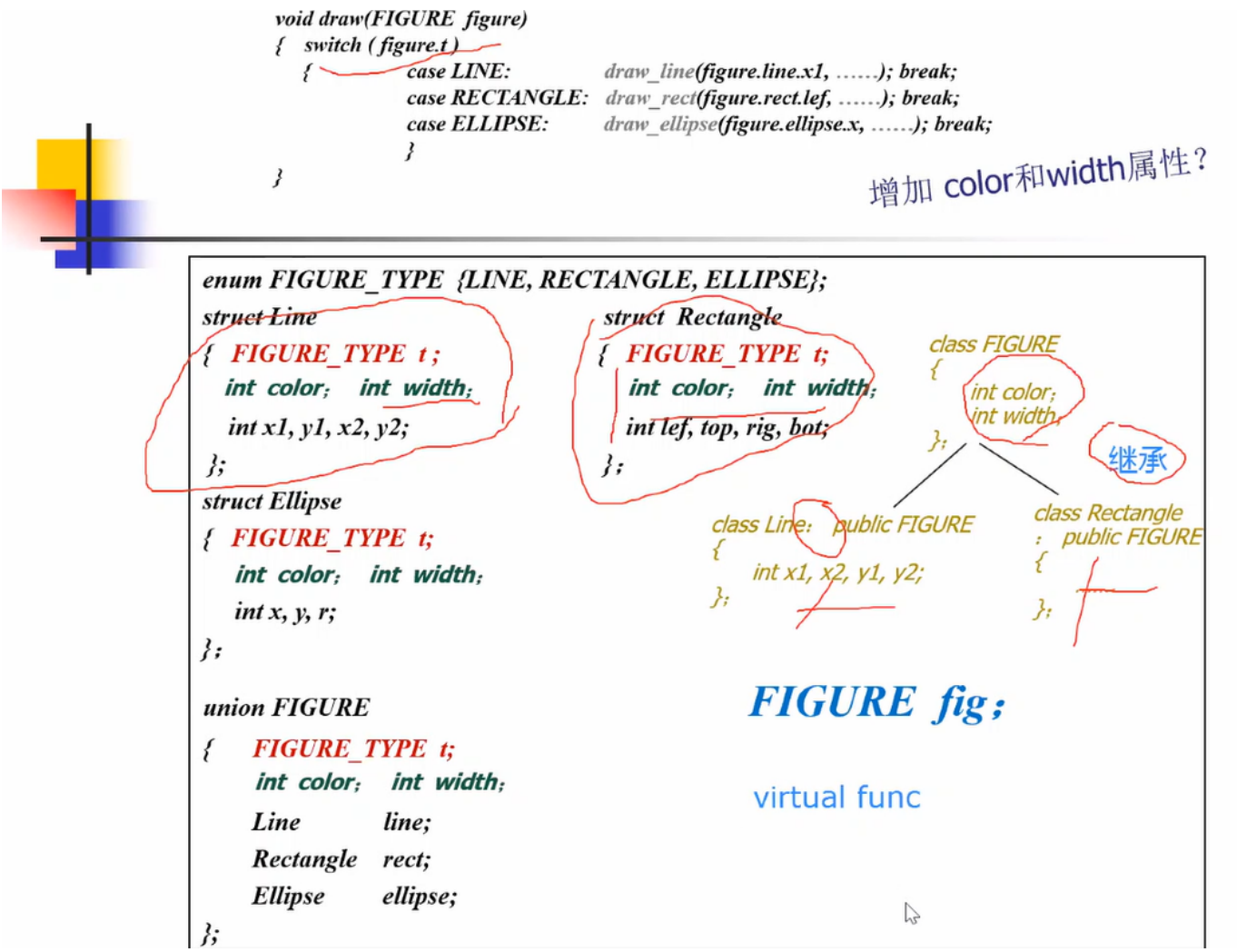



例：定义数组，存储100个图形(直线、矩形、圆)

```
enum FIGURE_TYPE {LINE, RECTANGLE, ELLIPSE};
struct Line
{ FIGURE_TYPE t;
  int x1, y1, x2, y2;
};
struct Ellipse
{ FIGURE_TYPE t;
  int x, y, r;
};
union FIGURE
{
    Line      line;
    Rectangle rect;
    Ellipse   ellipse;
};
struct Rectangle
{ FIGURE_TYPE t;
  int lef, top, rig, bot;
};
union FIGURE
{
    .....
};
FIGURE figures[100];
void main()
{ input( figures, 100 );
  for (int i=0;i<100;i++)
      draw(figures[i]);
}
API：
void draw_line(int,int,int,int);
void draw_rect(int,int,int,int);
void draw_ellipse(int,int,int);
```

# 多态初步




# 指针与数组

## 指针和内存处理

```
//将从pointer开始的长度为size的内存全部清空
void memset (void *pointer, unsigned size) {
    char *p = (char *)pointer;
    for (int k=0;k<size;k++){
      *p++ = 0;
    }
}
//将src位置上和后面size长度的数据拷贝到des的位置上去
void memcpy(void *des, void *src, unsigned size) {
    char *sp = (char *)src;
    char *dp = (char *)des;
    for (int i = 0; i < size; i++) {
        *dp++ = *sp++;
    }
}
//查看从q开始的长度为nbit的数据
void showBytes(void *q, int n)//查看内存
{
    cout << n << endl;
```

```
    unsigned char *p = (unsigned char *)q;
    for (int i=0; i<n; i++){
    cout << (void *)(p+i) << " : "<< setw(2) << hex << (int)*(p+i) << " ";
      if ( (i+1) %4 ==0 ) cout << endl;
    }
    cout << dec << endl;
}
```

# 智能指针

1. 通过将一些需要的信息进行封装的方法，来保证不管出现什么异常，在退出相应操作部分时，自动调用对象的析构函数来保证不会出现内存泄漏的问题。
2. 同样的还有句柄类(C++ 异常中有)

```
template <class T>
class auto_ptr{
    public:
        auto_ptr(T *p=0):ptr(p) {}
        ~auto_ptr() { delete ptr; }
        T* operator->()  const { return ptr;}
        T& operator *()  const { return *ptr; }
    private:
        T*  ptr;
};
//结合智慧指针使用
void processAdoptions(istream& dataSource){
    while (dataSource){
        auto_ptr<ALA> pa(readALA(dataSource));
        pa->processAdoption();//只要对象结束，就会自动delete
    }
}
```

## 实例：GUI应用软件中的某个显示信息的函数

handle class：句柄类，就是处理智能指针

```
void displayInfo(const Information& info){
    WINDOW_HANDLE w(createWindow());//针对windows窗体的一个指针，createWindow:返回一个窗体指针，WINDOW_HANDLE是别名
    display info in window corresponding to w;
    destroyWindows(w);
}
//专门的句柄类，处理窗体问题
class WindowHandle{
    public:
        WindowHandle(WINDOW_HANDLE handler) : w(handler) {}
        ~WindowHandle() { destroyWindow(w);}//析构就会自动释放资源
        operator WINDOW_HANDLE() { return w; }//重载类型转换操作符，转换为WINDOW_HANDLE指针，将句柄类对象和包含的句柄一样的进行使用
```

```
    private:
        WINDOW_HANDLE w;
        WindowHandle(const WindowHandle&);
        WindowHandle & operator = (const WindowHandle&);
};
void displayInfo(const Information& info){
    WindowHandle  w(createWindow())
    //display info in window corresponding to w;
}
```

## 函数指针

1. 函数指针可以使得我们类似传递参数一样传递函数指针。
2. 函数指针允许我们抽象一些操作，同时支持我们实现多态操作。

```
//函数指针的定义，这个函数指针是参数为int的，返回值为double的函数指针。
typedef double (*FP)(int);
double *fp (int);
double f(int x){}
void main(){
    FP fp;
    fp = f;   //相当于fp = &f;为函数指针赋值
    (*fp)(10);//相当于fp(10);
    fp = g;  //Error
}
```

## Lambda表达式

1. 语法:`[capture](parameters) mutable ->return-type{statement}`
    1. `capture`：捕获列表
    2. `parameters`：参数列表
    3. `->` mutable：修饰符，可以取消lambda函数使其不是const函数，使用时参数列表不可省略(就算为空)
    4. `return type`:返回类型
    5. `statement`:函数体，可以使用参数和捕获变量
2. 例子:`auto func = [=, &b](int c)->int {return b += a + c;};`

## 指针和数组

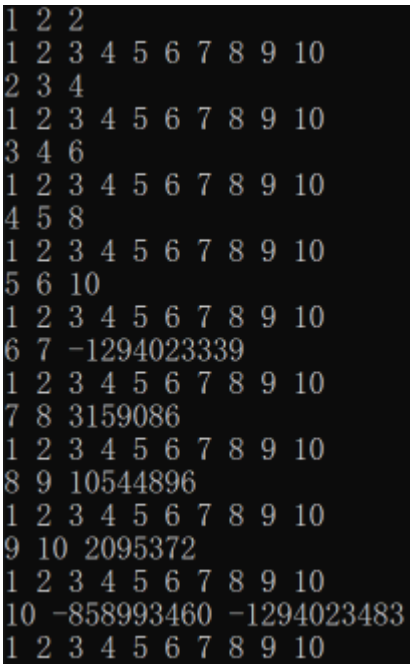
### 一维数组

1. `int *p = a`,a代表数组的首地址。
2. `*(p+i)`:p的位置不移动。
3. `*(p++)`:p向前移动一个位置。
4. `sizeof(a)`:数组的大小。
5. `sizeof(a[0])`:数组的元素的占地大小。
6. 几个等价操作:

1. `a[i] == *(a+i)`
2. `&a[i] == a+i`

```
int a[10] = { 1,2,3,4,5,6,7,8,9,10 };
int *p = &a[0];
for (int i = 0; i < 10; i++) {
    cout << *p++ << " ";//指针移动
    //等价于 cout << *(p++) << " "
    //不等价于 cout << (*p)++ << " ",这样指针不移动
    cout << *p << " ";
    cout << *(p + i) << endl;//指针不动
    for (int j = 0; j < 10; j++) {
        cout << a[j] << " ";
    }
    cout << endl;
}
```

> 上面代码的结果如下，上图中为什么等价，是因为++的优先级高于*，*
>
> C++的优先级和结合性




## 二维数组

1. 有一些操作相对于其组成部分(一维数组越界了，但是对于其本身没有越界)，也就是C++对于这一类越界是默许的，因为这块系统空间在我们的控制中。
2. `int *p = &a[0][0]`:访问二维数组中的T类型的变量

```
int b[20][10];
int *q;
q = &b[0][0];//等价于 q = b[0]
//b[i][j] == *(&b[0][0] + i*10 + j) == *(q + i * 10 + j) == q[i*10 + j]
```

```
T *p;
p = &b[0];// p = b
```

## 降维操作

> 使用线性方式来访问二维数组

## 升维操作

> 对于一维数组，建立逻辑视图，按照多维数组的方式进行访问，可以在传递参数的时候直接进行划分。

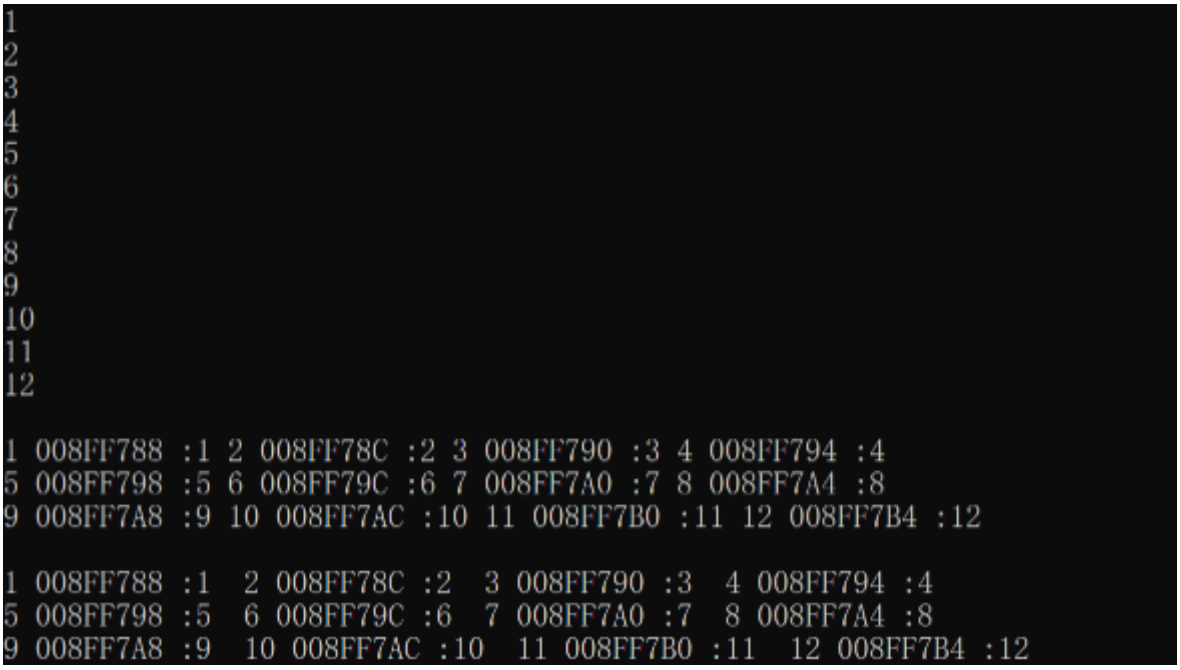
```
void show(int a[],  int n){
    for (int i=0;i<n;i++)  {
        cout << a[i] << " ";
        cout << endl;
    }
    cout << endl;
}
void show(int a[][2], int n){
    for (int i=0;i<n;i++)
        for (int j=0;j<2;j++) {
            cout << a[i][j] << " ";
            cout << *(a+i)+j << " :" << a[i][j] << " ";
            //四个换一行
            if ((i*2+j+1)%4 == 0)
                cout << endl;
        }
    cout << endl;
}
void show(int a[][2][3], int n){
    for (int i=0;i<n;i++)
        for (int j=0;j<2;j++)
            for (int k=0;k<3;k++){
                cout << a[i][j][k] << " ";
                cout << *(*(a+i)+j)+k << " :" << a[i][j][k] << "  ";
                //换行输出
                if ((i*6+j*3+k+1)%4 == 0)
                    cout << endl;
            }
    cout << endl;
}

void main(){
    int b[12];
    for (int i=0;i<12;i++)  b[i] = i+1;
    show(b,12);
    //二维数组
    typedef int T[2];
    show((T*)b,6);//show((int (*)[2])b,6),一定有括号

    //三维数组
```

```
    typedef int T1[3];
    typedef T1 T2[2];
    show((T2*)b,2);//show((int (*)[2][3])b,2)
}
```

> 结果如下图




# 指针数组

1. main函数:`int main(int argc,char * argv[],char * env[])`
    - argc:参数个数(包含命令)
    - argv:命令行参数
    - env:环境参数(为什么这个不必指出长度?因为\0结束，一个结束符)
2. 可变参数(详见C++指针与引用)：主要是利用内存机制，实现print函数

```
void MyPrint(char *s, ...){
    va_list marker;//拿到一个指针,这个指针是字符串开始的位置
    va_start(marker,s);//找到参数的位置，s的位置
    int i=0;
    char c;
    while ((c=s[i]) != '\0'){
        if (c != '%')
            cout << c;
        else{
            i++;
            switch (c=s[i]){
                case 'f': cout << va_arg(marker,double); break;
                case 'd': cout << va_arg(marker,int);break;
                case 'c': cout << va_arg(marker,char);break;
            }
        }
        i++;
```

```
    }
    cout << endl;
    va_end(marker);//将当前指针回归原始状态
}
void main(){
    MyPrint("double: %f integer: %d string: %c ",1.1, 100, 'A');
}
```

# 指针的拷贝和地址传递

```
//传递指针，影响原来的值。
void myswap(int *p1, int *p2) {
    int* tmp = p1;
    p1 = p2;
    p2 = tmp;
}
//传递引用，影响原来的值
void myswap2(int &p1, int & p2) {
    int tmp = p1;
    p1 = p2;
    p2 = tmp;
}
//传递指针的指针，不影响原来的值
void myswap(char **p1, char **p2) {
    char *tmp = *p1;
    *p1 = *p2;
    *p2 = tmp;
}
void main()
{
    char *p1 =(char*) "abcd";
    char *p2 =(char*) "1234";
    int a = 100;
    int b = 200;
    myswap(&a, &b);
    cout << a << " " << b << endl;//100 200
    myswap2(a, b);
    cout << a << " " << b << endl;//200 100
    myswap(&p1, &p2);
    cout << p1 << " " << p2 << endl;//1234 abcd
}
```

# 引用

## 定义

为一块已有的内存空间取一个别名，定义引用变量的时候必须同时声明

可以通过函数副作用，来使得返回值也可以是引用和函数指针。

### 实例

```
//一旦是p的别名，就一定只能是p的别名了
int &a = *p;
//利用函数副作用
void f(int &a);
```

可以使用`const`修饰引用，避免造成不必要的修改

# 动态变量

## 定义

程序员在Heap上主动申请空间进行存储的变量。

## 申请动态变量

申请的过程可能会失败

### `new`

``new <类型名> [<整型表达式>]。首先分配对应大小的内存，然后调用构造函数进行初始化，最后再赋值给对应的值。**注意到new`出来的对象是无名字的，你只能用当时返回的指针去访问它**。

### `malloc`

``int *p = (int *)malloc(sizeof(int));`，不推荐，只是分配了空间，但是并不会调用构造函数。

### 补充

为什么引入`new`和`delete`？因为新的操作符可以解决初始化函数的析构函数的调用的问题

注意到`new`出来的对象数组里面的每一个对象会被调用默认构造函数（你也必须有），但是**原始类型不会被初始化**！你也可以用初始化列表来保证所有的初始化出来都是某个值。

### 分配连续空间实例

```
//使用malloc进行空间分配
int *p = new int[10];
int *p = (int*)malloc(sizeof(int)*10)
int (*p2)[5] = (int (*)[5])p;
//new，如下是new的错误写法
q = new int[2][5];
//new，多维数组
const int ROWS = 3;
const int COLUMNS = 4;
char **chArray2;
//首先分配好行的位置
```

```
chArray2 = new char* [ROWS];
//然后对每一行分配对应的数组
for (int row = 0; row < ROWS; row++ )
    chArray2[row] = new char[ COLUMNS ];
//删除算法和新建算法完全一样
```

## 分配空间的初始化问题

1. `int *p1 = new int[5];` 默认不进行初始化
2. `int *p2 = new int[5]();`进行默认初始化
3. `int *p2 = new int[5]{0,1,2,3,4}`:进行显式对应函数初始化

## 释放动态变量

1. `new - delete | delete[]`:使用new的方式创建的动态变量，通过delete的方式释放
    1. `delete a`:释放数组的第一个元素
    2. `delete[] a`:释放数组中的所有的元素，注意此时归还是从a开始向下归还size大小的空间，所以a必须是数组的首地址才行。
    3. delete会调用变量的析构函数，注意删除原对象之后要将对应的指针置为NULL，避免悬挂指针
2. `malloc free`:只是释放对应的申请的空间，但不会调用析构函数
3. 归还操作前，注意要拷贝一个指针值，不然无法找到归还开始的头部地址。

## 动态变量的应用：单链表和多链表

1. 单链表和双链表要求掌握
2. 注意指针的移动情况

# OOP

## 对象类型的判断

- 运行时判断
- 编译时判断

```
//运行时判断，如方法覆盖等情形
int i;
if(typeid(i) == typeid(int) )
    cout << "i is int" << endl;
//编译时判断，如模板类和改写等情形
template<class T>
void func(T t ){
    cout << "i is not int" << endl ;
}
template<> void func<int>(int i){//特化
    cout << "i is int" << endl ;
}
int i;
func(i)
```

## 空类C++默认提供的方法

```
//了解以下C++编译器默认提供的方法
class Empty {};
class Empty {
    Empty(); // 缺省（默认）构造函数
    Empty(const Empty&); // 缺省（默认）拷贝构造函数
    ~Empty(); // 缺省（默认）析构函数
    Empty& operator=(const Empty&); // 缺省（默认）赋值运算符
    Empty* operator &(); // 缺省取址运算符
    const Empty* operator &() const; // 缺省取址运算符（const版本）
}; // 只有当实际使用这些函数的时候，编译器才会去定义它们。
```

### 三五法则

如果需要析构函数，则一定需要拷贝构造函数、拷贝赋值操作符、移动构造函数和移动赋值运算符（牵一发而动全身，后面两个仅限C++11及以后，且没有只是影响优化）

编译器可能干脆就不生成（后两个）。

## 构造

### 普通构造函数

1. 默认构造函数:无参构造函数
2. 构造函数可以设置为`private`或者`public`（默认）

```
class A{
    public:
        A();
        A(int i);
        A(char *p);
}
A a1 = A(1);
A a1(1);
//注意这种用法在函数回调的时候使用
A a1 = 1;
//以上都是调A(int i)
A a2 = A();
A a2;
//以上都是调A()，注意：不能写成：A a2();
A a3 = A("abcd");
A a3("abcd");
A a3 = "abcd";
//以上都是调A(char *)
A a[4];//调用a[0]、a[1]、a[2]、a[3]的A()
A b[5]={ A(), A(1), A("abcd"), 2, "xyz"};
```

## 拷贝构造函数

相同类型的类对象是通过拷贝构造函数来完成整个复制过程：自动调用：创建对象时，用一同类的对象对其初始化的时候进行调用。

### 默认拷贝构造函数

1. 逐个成员初始化(member-wise initialization)
2. 对于对象成员，该定义是递归的

### 应用场景

```
//赋值拷贝构造
A a;
A b=a;
//传参进行拷贝
f(A a){}
A b;
f(b);
//返回值进行拷贝
A f(){
    A a;
    return a;
}
f();
```

```
//拷贝构造函数
public:
    //const避免出现修改
    A(const A& a);//一定要写引用，不然就递归调用了
```

**拷贝构造函数私有**

目的是让编译器不能调用拷贝构造函数，防止对象按值传递，只能引用传递(对象比较大)

**对象格式**

不然会出现**循环拷贝**问题:如果没有引用的话，传参则会拷贝，那么就会出现循环拷贝，记住：`A(const A& a)`

没有深拷贝的需求的话，使用编译器提供的默认拷贝即可。

```
String &strcpy(String &s){ // 记住这个参数格式，不这样就套娃了（传参也要构造函数）
    delete []str;   // 旧的不去新的不来
    str = newchar[strlen(s.str)+1];
    strcpy(str,s.str);
    return *this;
}
```

**引入右值引用的特殊拷贝构造函数**

```
class MyArray {
public:
    //…
    MyArray &operator=(const MyArray &other) {
        if (this == &other)
            return *this;
        if (arr) {
            delete[] arr;
            arr = NULL;
        }
        size = other.size;
        memcpy(arr, other.arr, size *   sizeof(int));
        return *this;
    }
    MyArray &operator=(ArrayWrapper &&other) {
        size = other.size;
        arr = other.arr;
        other.arr = NULL;
        return *this;
    }
}
int main() {
    MyArray myArr;
```

```
    myArr = MyArr(5);//MyArr是临时对象
}
```

## 移动构造函数

需要C++11的右值引用机制

```
class MyArray {
    int size;
    int *arr;
public:
    MyArray():size(0),arr(NULL){}
    MyArray(int sz):
        size(sz),arr(new int[sz]) {
        //init array here…
    }
    MyArray(const MyArray &other):
        size(other.size),
        arr(new int[other.size]) {
        for (int i = 0; i < size; i++) {        arr[i] = other.arr[i];
        }
    }
    //main函数中第三种声明方式的移动构造
    MyArray (MyArray &&other):
        size(other.size), arr(other.arr) {
        other.arr = NULL;
    }
    ~ MyArray() {
        delete[] arr;
    }
}
MyArray change_aw(const MyArray &other)
{
    MyArray aw(other.get_size());
    //Do some change to aw.
    //….
    return aw;
}

int main() {
    MyArray myArr(5);
    MyArray myArr2 = change_aw(myArr);//调用了两次拷贝，先将myArr传入的时候进行一次拷贝，返回之后再次进行拷贝，比较大的开销
    MyArray &&myArr2 = change_aw(myArr);//右值函数，直接用移动构造函数，右值引用造成的维护困难
    MyArray myArr2 = change_aw(myArr);//有了新的移动构造函数,自动适配，提高拷贝速度在C++中使用移动构造函数
}
```

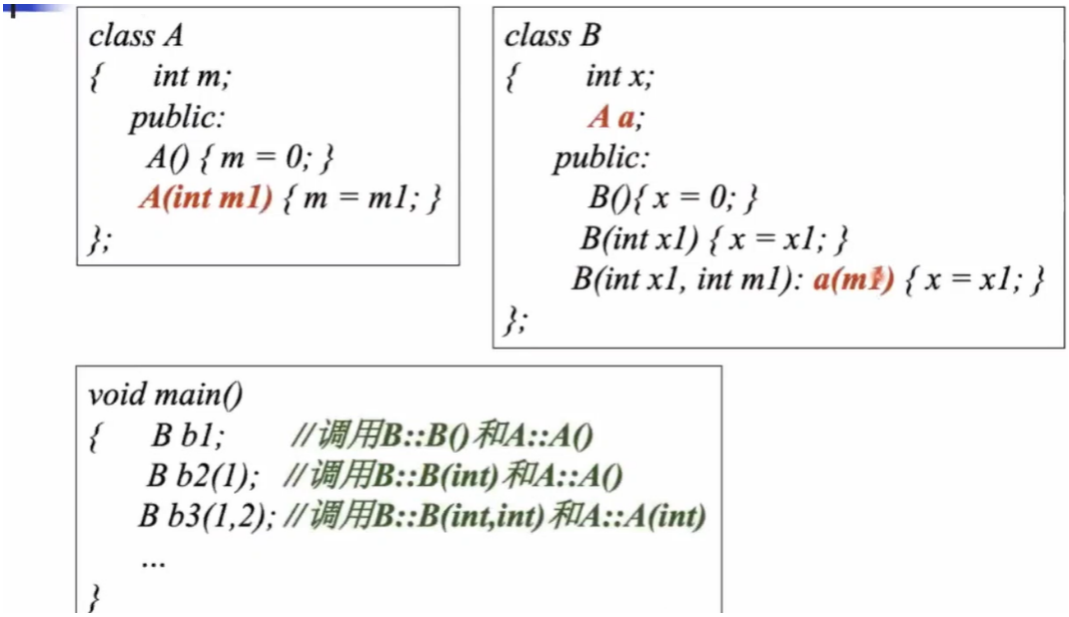
## 成员初始化表

1. 成员初始化表:开辟空间的时候就赋值，而构造函数时在开辟空间结束之后再赋值，先于构造函数执行(按照成员变量声明顺序进行初始化)
2. 成员初始化表可以降低编译器的压力

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

3. 在构造函数中尽量使用成员初始化表取代赋值动作(如果成员变量没有那么多，不然难以维护)

4. 常量往往是通过成员初始化表的方式来完成初始化




C++ 98的时候，除了`static const`可以在声明的时候直接赋值，别的都不行，但是现在是11就可以了。初始化列表也可以用于非`static const`的东西了。

构造函数中尽量使用成员初始化表代替赋值操作

- `const成员` `reference成员`（可以调其构造方法）/对象成员（可以调其构造方法）
- 效率高
- 数据成员太多时除外（防止降低可维护性）

## 初始化顺序

```
class A {
    int m;
public:
    A() {
        m = 0; cout << "A()" << endl;
    }
    A(int m1) {
```

```
        m = m1;
        cout << "A(int m1)" << endl;
    }
};
class B {
    int x;
    A a;//每一次创建类都优先创建
    public:
        B(){
            x = 0; cout << "B()" << endl;
        }
        B(int x1){
            x = x1;
            cout << "B(int x1)" << endl;
        }
        B(int x1, int m1):a(m1){
            x = x1;
            cout << "B(int x1, int m1)" << endl;
        }
        //不能在函数体里写A的构造函数(已经调过了)
};
int main() {
    B b1;// 调用 B::B() 和 A::A()
    cout << "________________" << endl;
    B b2(1);   // 调用 B::B(int) 和 A::A()
    cout << "________________" << endl;
    B b3(1, 2); // 调用 B::B(int,int) 和 A::A(int) …
}
//result:
//A()
//B()
//________________
//A()
//B(int x1)
//________________
//A(int m1)
//B(int x1, int m1)
```

# 析构

1. 格式:~<类名>()
2. 功能:RAII:Resource Acquisition Is Initialization(资源获取即初始化)
3. 调用情况
    1. 对象消亡时，系统自动调用
    2. C++离开作用域的时候回收
    3. 使用delete关键字的时候进行调用
4. Private的析构函数：(强制自主控制对象存储分配)
    1. 回收对象的过程被接管，保证对象在堆上进行创建，但是不能使用delete，那么我们可以在内容提供一个destroy()方法来进行回收
    2. 写在栈或者全局区是不能通过编译的(自动调用，发现调不到)
    3. 强制在堆上进行创建，对很大的对象而言有好处强制管理存储分配

4. 适用于内存栈比较小的嵌入式系统

```
class A{
    public:
        A();
        void destroy(){delete this;}
    private:
        ~A();
};
//析构函数私有，无法声明
A a;
int main(){
    A aa;//析构函数私有，无法声明
};
A *p = new A;//在堆上声明
delete p;//错误
p->destroy();//可能出现p的null空指针问题

//Better Solution
static void free(A *p){ delete p; }
A::free(p);
```

# OS中的变量地址分配

## 栈空间

局部变量、值传递参数

## 堆空间

动态内存分配的位置

# 友元

友元是数据保护和访问效率的折衷方案。

友元声明可以让被声明为友元的类或函数访问当前类的`private`和`protected`的成员。

## 友元函数

1. 一个全局函数是一个友元函数

```
class A{
    int j;
    friend void func(A a);//友元函数
};
//全局函数
void func(A a)
{
```

```
    cout << a.j <<endl;//因为func是A的友元，所以可以访问A的所有成员变量
}
```

## 友元类

友元类：一个类是另一个类的友元

```
friend class B;//编译器会寻找一个类B来完成友元，如果没有会默认创建一个
friend B;//多用于模板类，不引入B\

class A{
    friend class B;     //友元类:B中的每一个函数都可以访问A的成员函数
};
```

## 友元类成员函数

在完整的类的声明完成之前是不能够被声明的。

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

## 友元函数的声明

1. 友元函数在之前可以没有声明
2. 友元函数如果之前还没有声明过，则当做已经声明了
3. 但是友元类函数在完整的类声明出现前不能声明友元函数。
4. 为什么友元函数和友元类成员函数的声明要求是不一样的?
    1. 数据的一致性:避免对应类里面没有这个函数(也就是C的完整定义必须有)
    2. 成员函数依赖于类

## 友元函数注意

1. 友元函数不具有传递性
2. 友元必须显示声明
3. 互为友元的两个类声明时是否需要**前置声明**
    1. 如果A和B不在一个命名空间不能通过编译
    2. 如果A和B在一个命名空间的话可以没有前置声明

```
class vector;//必须有这个才能完成编译，不然找到Vector的类声明
class Matrix{
    friend void multiply(Matrix &m, Vector &v, Vector &r);
};
class Vector{
    friend void multiply(Matrix &m, Vector &v, Vector &r);
```

```
};

//类成员函数的声明顺序
class A{
    int a;
    public:
        friend class B;
    void show(B &b){
        std::cout << b.b;//这里可以吗？不行，不知道B中有b
    }
}
class B{
    int b;
    public:
        friend class A;
        void show(A &a){
            std::cout << a.a;//这里是可以的
        }
}
void A::show (B &b){//只能在这里面实现
    std::cout << b.b;
}

//类和protected
class Base{
    protected :
        int prot_mem;// protected 成员
};
class Sneaky : public Base {//36min
    friend void clobber(Sneaky&);//能访问Sneaky::prot_mem
    friend void clobber(Base&);//不能访问Base::prot_mem，对外不可见
    int j;// j 默认是private

    void clobber(Sneaky &s) {
        s.j = s.prot_mem = 0;
    }//正确：clobber 能访问Sneaky对象的 private和protected成员
    void clobber(Base &b) {
        b.prot_mem = 0;
    }//错误:clobber 不能访问Base的 protected 成员
}
```

## 封装

在头文件中放置类和类属函数的定义，而不放置实现。如果放置实现，则会建议编译器将其作为内联函数及逆行处理。

1. getter 和 setter 一般会作为内联函数写在头文件中。
2. 内联函数一般会比较短

在Cpp文件中放置类和类属函数的实现

声明全局对象的时候如果没有显式初始化，那么他已经完成了默认初始化。

```
//a.h 存储类的头文件
class TDate{
    public:
        //只有函数签名
        void SetData(int y,int m ,int d);
        int IsLeapYear();
    private:
        int year,month,day;
}
//a.cpp 用来存储相应的实现部分
//TDate::命名空间
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);
}
```

## const

### 常量指针和指针常量

1. 常量指针:const <类型> * <指针变量>，不可以修改该指针指向的单元中的值，但是可以修改指向的单元。
2. 在函数式编程中，可以通过对参数中的传递量添加const来保证不会修改原值。
3. 指针常量:<类型>* const<指针变量>，指针在定义的时候初始化，可以修改该指针指向的单元中的值，但是不可以修改指 向的单元。

### print函数的思考

1. 注意变量指针位置上不能传入常量值，但是常量指针上可以传入常量

```
void print(int *p){
    cout << *p << endl;
}

const int c = 8;
print(c) ;//不可以被调用的
print(&c);//C++赋给的权利，在调用的时候除去常量的特性，这个&是强制类型转换，取消常量特性
void print(const int *p){//如此修改就可以大量复用
    //常量使用者和变量使用者都可以使用
    cout << *p << endl;
}
```

### 常量成员函数

1. 声明为const的对象只能调用常成员对象函数
2. 如果是非const的对象，则都可以进行调用
3. 是否const方法真的就不能修改对象里面的值了呢？不是,const只是语法上避免了，但是不是完全不可修改
    1. 关键词`mutable`:表示成员可以再const中进行修改，而不是用间接的方式来做。
    2. 去掉const转换:`(const_cast)<A*>(this)->x`转换后可以修改原来的成员

```
class A{
    int x,y;
    public:
        A(int x1, int y1);
        void f();
        void show() const;//前后要保证一致，const在后面
};
void A::f(){x = 1; y = 1;}//编译器怎么能发现不是const的？转化为防止变量被赋值，见下面，所以const指针不能修改
void f(A * const this);//上面的函数相当于这个

void A::show() const
{cout <<x << y;}
void show(const A* const this);//上面的函数相当于这个，第一个const表示指向对象常量，后一个const表示指针本身是常量

const A a(0,0);//常对象:这个对象是不可以修改的
a.f(); //错误，常对象无法调用非常方法
a.show();//正确
```

## static

### 静态成员变量

1. 一个类只有一个，初始化放在类外部，只能初始化一次
2. 为什么声明为静态，而不是全局？
    1. 避免名污染问题
    2. 避免数据泄漏

```
class A{
    int x,y;
    static int shared;
};
int A::shared=0;//j静态成员的初始化放在类的外部，只能被赋值一次，所以不再头文件中定义，而是在实现中定义，避免重复。并且定义的时候不用再写static
```

### 静态成员函数

1. **只能存取静态成员变量，调用静态成员函数**
2. 遵循类访问控制：在类上直接访问只能是静态成员变量

3. 类也是一种对象，可以通过类直接调用静态方法

### 访问静态成员

1. 通过对象使用:`A a;a.f();`
2. 通过类使用:`A::f();`
3. 例子：查看已经创建的实例数量

```
class A{
    static int obj_count;
    public:
        A(){obj_count++;}//追踪创建了多少个对象
        ~A(){obj_count--;}
        static int get_num_of_obj();//查看已经创建了多少个对象
};
int A::obj_count=0;
int A::get_num_of_obj() { return obj_count; }
```

## 继承

声明的时候不需要声明继承

```
//错误声明
class Undergraduated_Student : public Student;//声明的时候是不用声明继承的
//正确声明
class Undergraduated_Student;
```

派生类中对父类的重名方法是重写,期望被重写的部分的前面添加`Virtual`来保证子类重写。

父类中的所有部分都会被子类的名空间覆盖，但是通过命名空间也可以访问，如果父类中的`public`没有被子类复写，则可以调用

**构造函数**、**析构函数**和**运算符重载函数**是不会被继承的。

```
//显式继承A的构造函数
class B: public A{
    public:
        using A::A;
}
```

### 单继承

**`protected`与`private`继承**

1. 如果没有继承的话，`protected`和`private`的访问权限是相同的
2. 派生类可以访问基类中`protected`的属性的成员。

3. 派生类不可以访问**基类中的对象**的`protected`的属性。
4. **派生类含有基类的所有成员变量**

#### 子类修改访问权限

```
class Student {
    public:
        char nickname[16];
};
class Undergraduated_Student: public Student {}
    private:
        Student::nickname;//这样在才能修改可见性
};
```

#### 继承方式

`public`**继承**：`class A: public B`

1. 原来的`public`是`public`，原来的`private`是`private`。
2. 如果没有特殊需要建议使用`public`。
3. `IS A`关系

`private`**继承**：`class A: private B`

1. `private`：原来所有的都是`private`，但是这个`private`是对于`Undergraduate_Student`大对象而言，所以他自己还是可以访问的。
2. 默认的继承方式
3. `HAS A`关系

#### 继承的初始化顺序

派生类对象的初始化：由基类和派生类共同完成

##### 构造函数的执行次序

1. 基类的构造函数
2. 派生类对象成员类的构造函数(注意！)
3. 派生类的构造函数

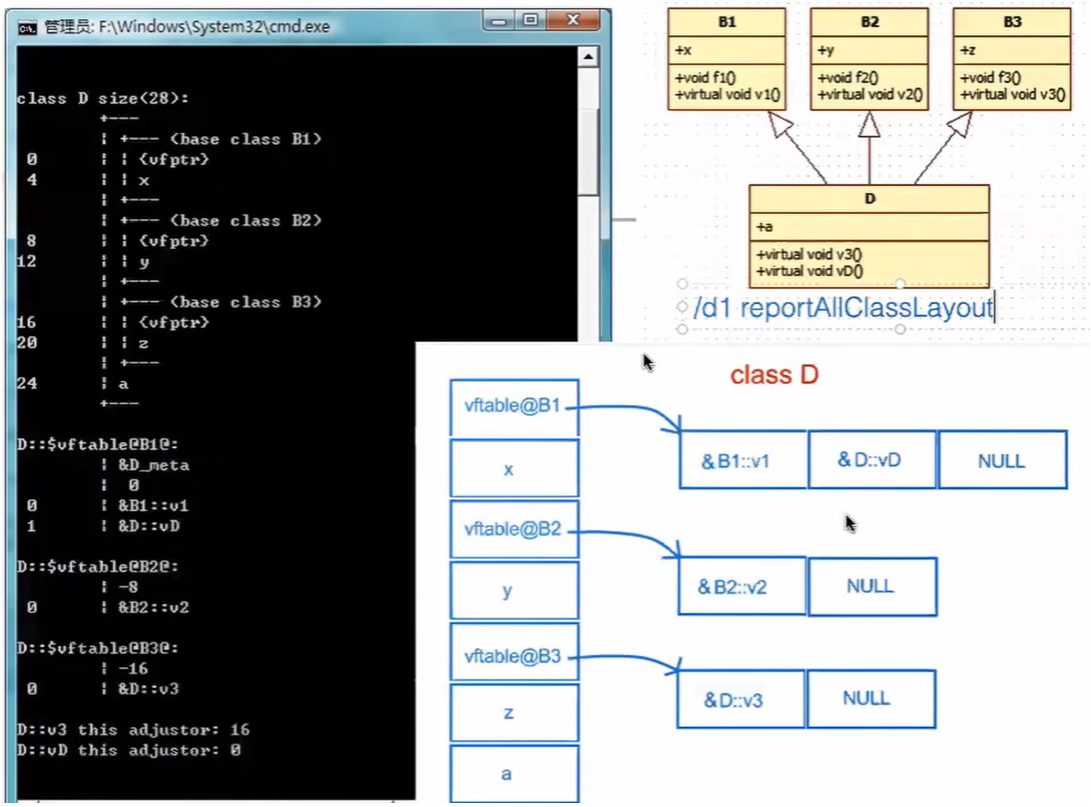
##### 析构函数的执行次序(与构造函数执行顺序相反)

1. 派生类的析构函数
2. 派生类对象成员类的析构函数
3. 基类的析构函数

##### 基类构造函数的调用

- 缺省执行基类默认构造函数
- 如果要执行基类的**非默认构造函数**，则必须在派生类构造函数的成员初始化表中指出

多继承

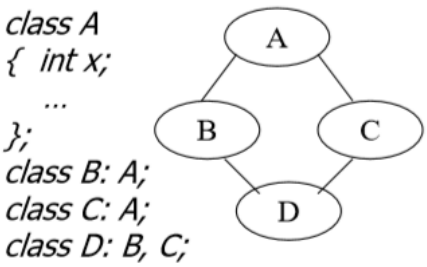



**定义**

```
class <派生类名>: [<继承方式>] <基类名1>,
                  [<继承方式>] <基类名2>, …
{〈成员表〉}
```

名冲突用命名空间来解决，但后面更应用虚基类虚继承解决。

**基类声明顺序(初始化顺序)**




**基类的声明次序**

1. 对基类构造函数/析构函数的调用次序(顶部基类，同层基类按照声明顺序) 上图中就是 ABCD的顺序
2. 对基类数据成员的存储安排

析构函数正好相反。

**虚基类**

1. 如果直接基类有公共的基类，则该公共基类中的成员变量在多继承的派生类中有**多**个副本
2. 如果有一个公共的虚基类，则成员变量只有**一**个副本
3. 类D有两个x成员，B::x,C::x
4. 虚继承:保留一个虚指针
    1. 虚指针指向A
    2. 可以认为是一个组合关系
5. 合并

```
class A;
class B: virtual public A;
class C: public virtual A;
//public virtual 和 virtual public是一致的
class D: B, C;
```

# 虚函数

1. 一个类只有一个虚函数表
2. 虚函数是指一个类中你希望重载的成员函数，但你使用一个基类指针或引用指向一个继承类对象的时候，调用一个虚函数时，实际调用的就是继承类的版本。
3. **如基类中被定义为虚成员函数，则派生类中对其重定义的成员函数均为虚函数**

## 类型相容和赋值兼容

**类型相容**

1. 类型相容是指完全相同的(别名)
2. 一个类型是另一个类型的子类型（`int -> long int`）

**赋值相容（不会丢失信息）**

对于类型相同的变量才有(`a = b`)

**如果类型相同可以直接赋值**

**子类型可以赋值给父类型**

- `A a; B b; class B: public A`
    - 对象的身份发生变化(a和b都代表栈上对应大小的内存),B类型对象变为了A类型的对象
    - 属于派生类的属性已不存在
    - 将派生类对象赋值给基类对象->对象切片
- `A a = b`:调用拷贝构造函数
- `const A &a`:函数必然包含的拷贝构造函数中的参数
- `B* pb; A* pa = pb; class B: public A`

    - 因为是赋值相容的，所以可以指针赋值
    - 这种情况类似Java
- `B b; A & a=b; class B: public A`：对象身份没有发生变化(还是B)

**尽量不要拷贝传参**

传参的时候尽量不要拷贝传参(存在对象切片问题)，而是使用引用传参。

```
class A{
    int x,y;
    public:
        void f();
};
class B: public A{
    int z;
    public:
        void f();
        void g();
};
//把派生类对象赋值给基类对象
A a;
B b;
a = b;      //OK,
b = a;      //Error
a.f();      //A::f()

//基类的引用或指针可以引用或指向派生类对象
A &r_a = b;      //OK
A *p_a = &b;     //OK

B &r_b = a;      //Error
B *p_b = &a;     //Error
//这里通过赋值相容，已经对对象完成了切片
func1(A& a){a.f();}
func1(b);//A::f
```

## 绑定时间

**静态绑定（编译时）**

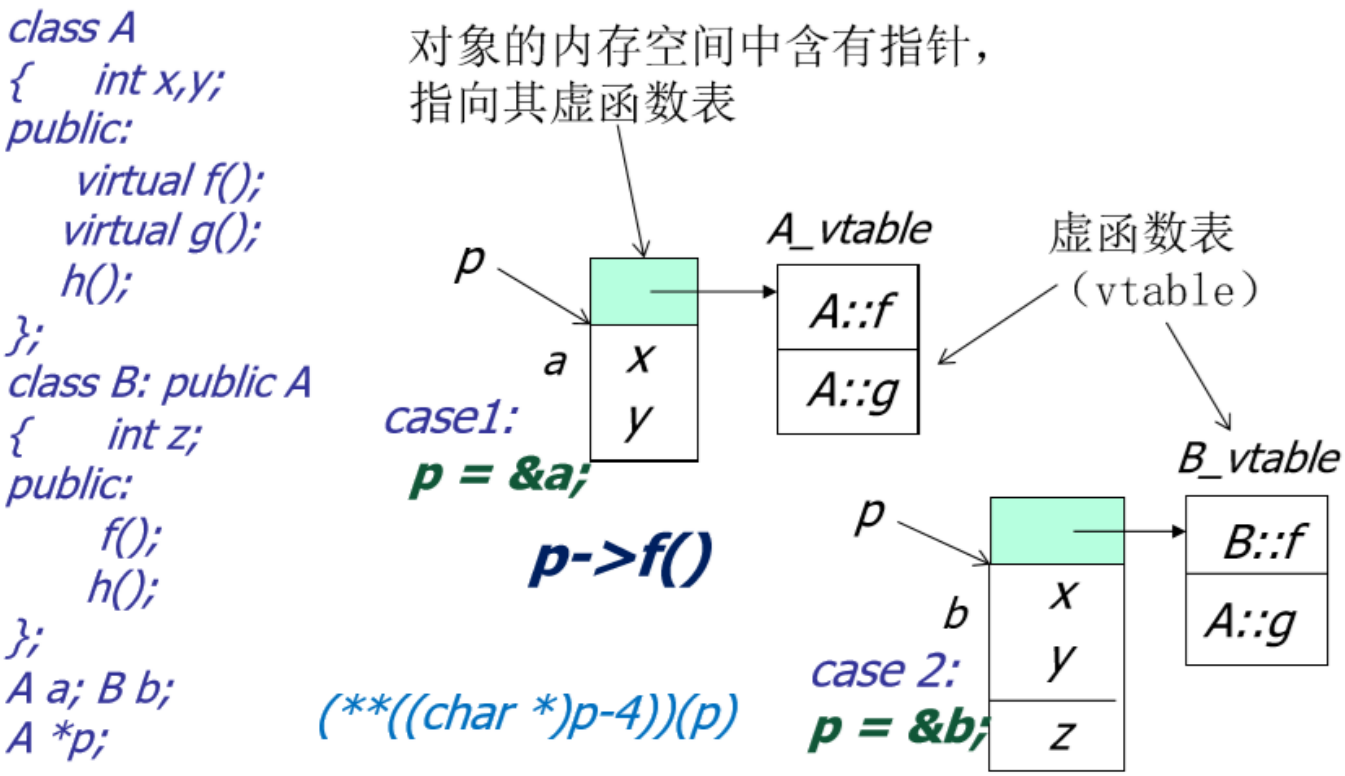
1. 编译时刻确定调用哪一个方法
2. 依据对象的静态类型
3. 效率高、灵活性差
4. 静态绑定根据形参决定

**动态绑定（运行时）（Late Binding）**

1. 晚绑定是指编译器或者解释器在运行前不知道对象的类型，使用晚绑定，无需检查对象的类型，只需要检查对象是否支持特性和方法即可。

2. c++中晚绑定常常发生在使用`virtual`声明成员函数
3. 运行时刻确定，依据对象的实际类型(动态)
4. 灵活性高、**效率低**
5. 动态绑定函数也就是虚函数。
6. 直到构造函数返回之后，对象方可正常使用
7. C++默认的都是静态绑定，Java默认的都是动态绑定

**虚函数表**




- p->f():需要寻找a和b中的f()函数地址
- 如果不能明确虚函数个数，没有办法索引
- 虚函数表(索引表,vtable):大小可变
  - 首先构造基类的虚函数表
  - 然后对派生类中的函数，如果查找了，则会覆盖对应函数来生成虚函数表
- 对象内存空间中含有指针指向虚函数表
- `(**((char *)p - 4))(p)`:f的函数调用(从虚函数表拿数据),p是参数this
- 空间上和时间上都付出了代价
  - 空间:存储虚函数表指针和虚函数表
  - 时间:需要通过虚函数表查找对应函数地址，多调用

```
class A{
    public:
        A() { f();}
        virtual void f();
        void g();
        void h(){
            f();
            g();
```

```
        }
};
class B: public A
{   public:
        void f();
        void g();
};
//直到构造函数返回之后，对象方可正常使用
//函数调用顺序，重要考试题，依据虚函数表
B b;      // A::A(), A::f, B::B(),为什么调用A的f而不是B的？因为名空间以及B没有构造。
A *p= &b;
p->f();   //B::f, 虚函数
p->g();   //A::g, g是静态绑定
p->h();   //A::h, B::f, A::g
```

- 尽量不要在构造函数中调用虚函数
- 此时的虚函数就是和构造函数名空间相同
- `h()`函数是非虚接口
    - 有不同的实现:调用了虚函数和非虚函数
    - 可以替换部分的实现
    - 可以使得非虚函数具有虚函数的特性(让全局函数具有多态:将全局函数做成非虚接口)

### 虚函数限制

1. 类的成员函数才可以是虚函数:全局函数不可以是虚函数
2. 静态成员函数不能是虚函数:静态的成员函数属于类，并不属于一个对象，所以不能虚函数
3. 内联成员函数不能是虚函数:内联成员函数在编译的时候就已经确定了
4. **构造函数不能是虚函数**:
    1. 因为创建类的时候是自动调用的，父类的指针无法直接调用，虚函数没有意义；
    2. 虚函数表是在构造函数中完成的；
5. **析构函数可以(往往)是虚函数**：如果不是虚函数，不好调用到派生类中的析构函数（delete一个父类指针，如果非虚，不能调用到派生类的析构函数）
    1. 如果有继承的话，最好使用虚析构函数，在调用析构的函数，会**先**调用基类的析构函数，所以:
    2. 在析构函数中，只需要析构派生类自己的资源就可以了

### `override`和`final`关键字

- `override`：希望以虚函数的形式写:编译器报错，防止漏写`virtual`问题
- `final`：不可以再次重写

## 10.10.5. 不要定义与继承而来的非虚成员函数同名的成员函数

```
class B {
    public:
        void mf();
};
class D: public B {
    public:
        void mf();
```

```
};
D x;
B* pB = &x;
pB->mf();//B:mf
D* pD = &x;
pD->mf();//D:mf
```

- 这样子的话，同一个对象使用不同指针会有不同的行为。

### 绝对不要重新定义继承而来的缺省参数值!

```
class A{
    public:
        virtual void f(int x = 0) =0;
};
class B: public A{
    public:
        virtual void f(int x = 1)
        { cout << x;}
};
A *p_a;
B b;
p_a = &b;
p_a->f();//0

class C: public A{
    public:
        virtual void f(int x) { cout<< x;}
};
A *p_a1;
C c;
p_a1 = &c;
p_a1->f();//0
//对象中只记录虚函数的入口地址
```

- 虚函数表上只记录了第一个(最近根)的缺省值

## 纯虚函数和抽象类

### 纯虚函数 Java中的接口

1. 声明时在函数原型后面加上 **= 0**:`virtual int f() = 0;`
2. **往往**只给出函数声明，不给出实现：可以给出实现，通过函数外进行定义(但是不好访问，因为查到是0)
3. 子类必须继承接口，并给出实现

```
int f() = 0;
int f(){
    Base::f;//显式调用基类中纯虚函数的定义
}
```

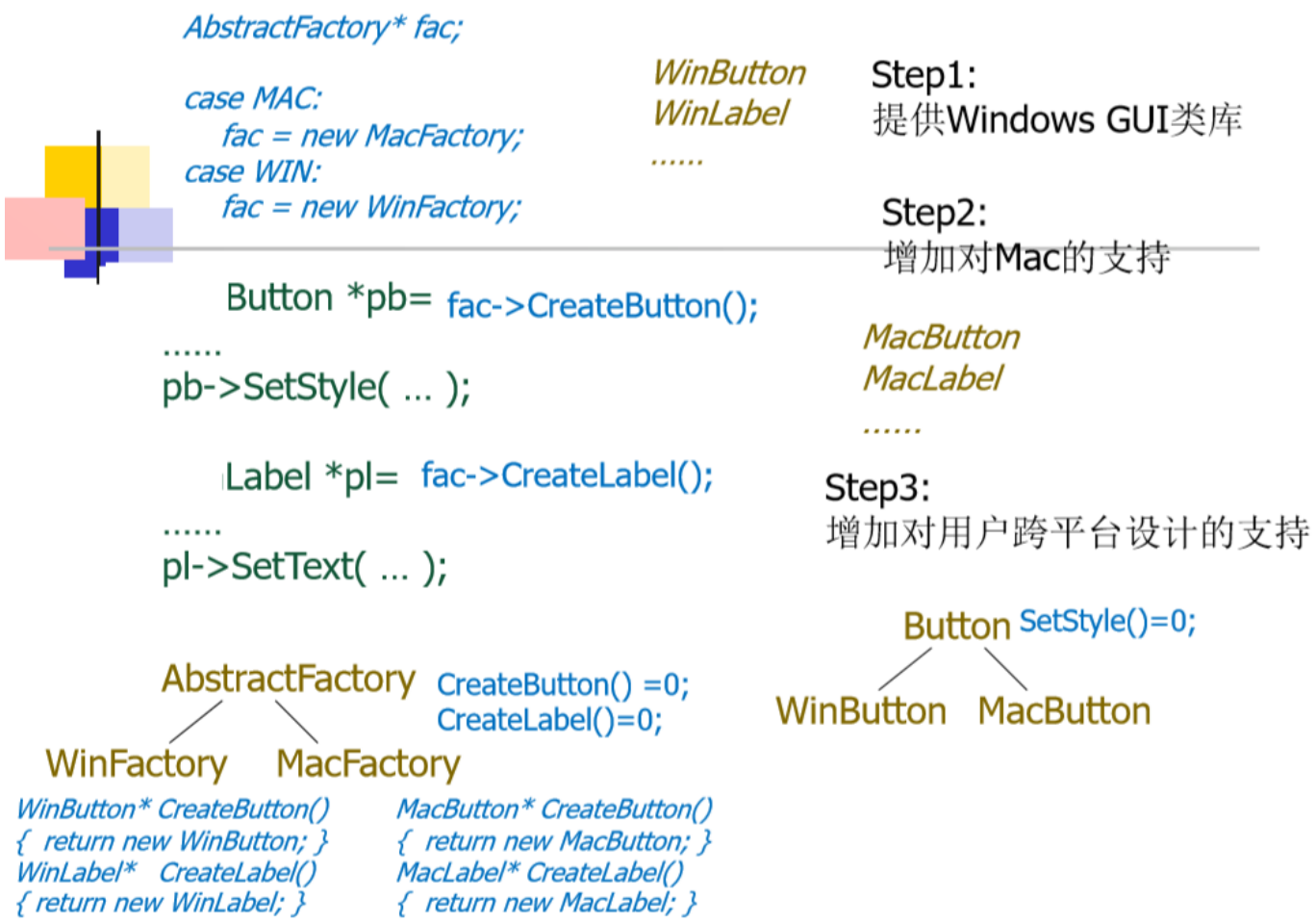
### 抽象类

1. 至少包含一个纯虚函数
2. 不能用于创建对象:抽象类类似一个接口，提供一个框架
3. 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

## 设计模式

### 抽象工厂模式

**实例**




**Step1：提供Windows GUI类库：WinButton**

```
WinButton *pb= new WinButton();
pb->SetStyle();
WinLabel *pl = new WinLabel();
pl->SetText();
```

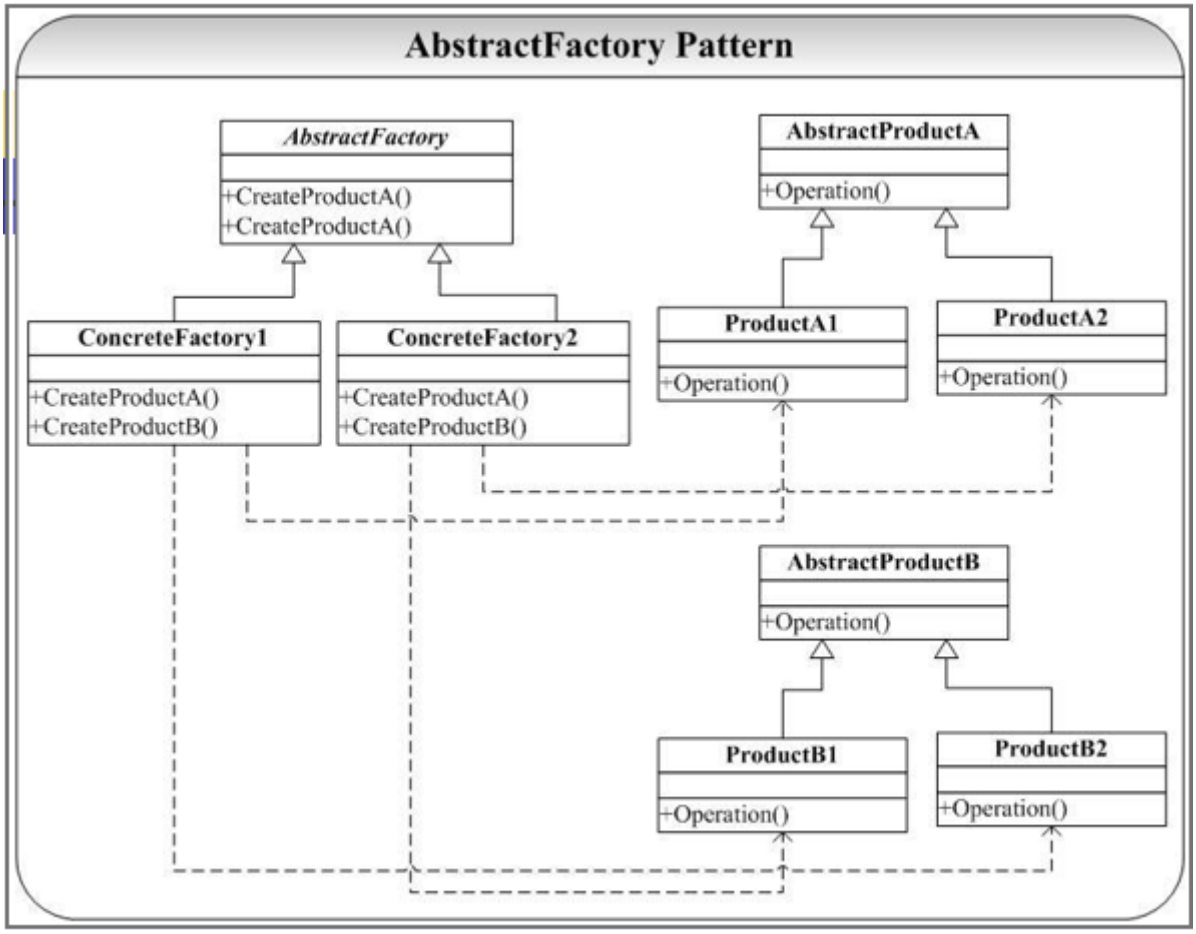
**Step2：增加对Mac的支持:MacButton，MacLabel**

```
MacButton *pb= new MacButton();
pb->SetStyle();
MacLabel *pl = new MacLabel();
pl->SetText();
```

**Step3：增加用户跨平台设计的支持：将Button抽象出来**

```
Button *pb= new MacButton();
pb->SetStyle();
Label *pl = new MacLabel();
pl->SetText();
//创建工厂来保证创建的正确性
class AbstractFactory {
public:
    virtual Button* CreateButton() =0;
    virtual Label* CreateLabel() =0;
};
class MacFactory: public AbstractFactory {
public:
    MacButton* CreateButton() { return new MacButton; }
    MacLabel* CreateLabel() {  return new MacLabel; }
};
class WinFactory: public AbstractFactory {
public:
    WinButton* CreateButton() { return new WinButton; }
    WinLabel*   CreateLabel() { return new WinLabel; }
};
class Button; // Abstract Class
class MacButton: public Button {};
class WinButton: public Button {};
class Label; // Abstract Class
class MacLabel: public Label {};
class WinLabel: public Label {};
AbstractFactory* fac;
switch (style) {
case MAC:
    fac = new MacFactory;
    break;
case WIN:
    fac = new WinFactory;
    break;
}
Button* button = fac->CreateButton();
Label* Label = fac->CreateLabel();
```

**类图**




### 单件模式

```
class  singleton{
    protected://构造函数外部不可以使用
        singleton(){}
        singleton(const singleton &);
    public:
        static singleton *instance() {
            return  m_instance == NULL?
            m_instance = new singleton: m_instance;
        }
        static void destroy()  { delete m_instance; m_instance = NULL; }
    private:
        static singleton *m_instance;//保存对象的指针也是static的
};
singleton *singleton::m_instance= NULL;//初始化
```

## 多态与操作符重载

1. 函数重载：(静态多态)，和虚函数的动态多态不同(一名多用):函数重载包含操作符重载
2. 类属多态：模板：template

### 函数重载

函数重载要求名同、参数不同，而不能只通过返回值的类型进行区分。

**歧义转换**

1. 按照顺序匹配
2. 找到最佳匹配
    1. 原则一:这个匹配每一个参数不必其他的匹配更差
    2. 原则二:这个匹配有一个参数更精确匹配

重载是为了让事情有效率，而不是过分有效率。

## 单目操作符重载

**++与--**

```
class Counter {
    int value;
    public:
        Counter() { value = 0; }
        Counter& operator ++()//++a 左值
        {
            value ++;
            return *this;
        }
        Counter operator ++(int)//a++ 右值，int是dummy argument 哑元变量，就是惯例
        {
            Counter temp = *this;
            value++;
            return temp;
        }
}
```

**<<**

`ostream& operator << (ostream& o, Day& d)`，返回引用保证可以链式调用,如果没有&，那么在第一个return出现了对象拷贝，容易出现临时变量不能返回拷贝的问题。

```
ostream& operator << (ostream& o, Day& d)
{   switch (d)
    {   case SUN: o << "SUN" << endl;break;//直接使用ostream中的<<
        case MON: o << "MON" << endl;break;
        case TUE: o << "TUE" << endl;break;
        case WED: o << "WED" << endl;break;
        case THU: o << "THU" << endl;break;
        case FRI: o << "FRI" << endl;break;
        case SAT: o << "SAT" << endl;break;
    }
    return o;//为什么要return ostream类型的变量:需要连续的使用可链式调用，Cout << 1 <<
2;
}
```

=

A& operator = (A& a)不可以被继承，返回引用对象。在=复制的过程中，尽可能地避免出现自我复制的情况（可以在程序入口检查）。

```
class A {
    int x,y ;
    char *p ;
    public :
        A& operator = (A& a) {
            //赋值
            x = a.x;
            y = a.y;
            delete []p;
            p = new char[strlen(a.p)+1];
            strcpy(p,a.p);
            return *this;//也会出现悬指针
        }//还有问题，就是赋值自身会出现问题
};
A a, b;
a = b;//调用自己的复制

//idle pointer，B被析构的时候会将p释放掉，导致p指向已经被释放掉的指针
//Memory leak,A申请的区域可能没有办法被释放

//更安全的拷贝，先new再delete
char *pOrig = p;
p = new char ...
strcpy();
delete pOrig;
return *this;

//自我复制问题
if(this == &a)
    return *this;
```

[]

char& operator[](int i)

```
class string {
    char *p;
    public :
        string(char *p1){
            p = new char [strlen(p1)+ 1];
            strcpy(p,p1);//#pragma warning(disable:4996)来屏蔽问题
        }
        char& operator[](int i){
            return p[i];
        }
```

```
        const char operator[](int i) const{
            return p[i];
        }
        //可以用两个重载函数吗?是可以的
        virtual ~string() { delete[] p ; }
};
string s("aacd");
s[2] = 'b' ;
//第一个重载加上const可以使得const或者非const对象都可以调用
const string cs('const');
cout << cs[0];
const cs[0] = 'D';//const 版本不想被赋值(返回const的)，非const版本想要被赋值，之后再进行重载的时候就需要同时重载两个
```

`()`

1. 类型转换:`operator double()`
2. 函数调用:`double operator()(double,int,int)`

```
//类型转换
class Rational {
    public: Rational(int n1, int n2) {
        n = n1;
        d = n2;
    }
    operator double() {//类型转换操作符，语法特殊
        return (double)n/d;
    }
    private:
        int n, d;
};

//函数调用
class Func {
    double para;
    int lowerBound , upperBound ;
    public:
        double operator()(double,int,int);
};
Func f;
f(2.4, 0, 8);
```

## 不支持重载的操作符

**无法重载**

不可以重载的操作符:`.`(成员访问操作符)、`.*`(成员指针访问运算符，如下)、`::`(域操作符)、`?:`(条件操作符)、`sizeof`:也不重载

**原因**

前两个为了防止类访问出现混乱。::后面是名称不是变量。?:条件运算符涉及到跳转，如果重载就影响了理解

**不建议重载的操作符号**

永远不要重载&&和||，否则你会丢失短路算法的优势，还可能影响理解。

## 全局函数重载操作符

友元：`friend <ret type> operator #(<arg1>,<arg2>)`

格式：`<ret type> operator #(<arg1>,<arg2>)`

注意:=、()、[]、->不可以作为全局函数重载

1. 单目运算符最好重载为类的成员函数
2. 双目运算符最好重载为类的友元函数

问题：为什么禁止在类外禁止重载赋值操作符?

1. 如果没有类内提供一个赋值操作符，则编译器会默认提供一个类内的复制操作符。
2. 查找操作符优先查找类内，之后查找全局，所以全局重载赋值操作符不可能被用到。

## 双目操作符重载

格式:`<ret type>operator #(<arg>)`

```
<class name> a,b;
a # b;//a -> this
a.operator#(b)
```

**加减乘除**

类内重载也不是不可以，但是你在面对两个操作数类型不同的情形的时候你得写两个，还得考虑执行顺序，这不好。所以还是推荐全局重载。

`friend Complex operator+(Complex& c1 , Complex& c2);` （以加法为例）

```
Complex operator+ (Complex& c1 , Complex& c2 ) {//全局函数重载至少包含一个用户自定义
类型
    Complex temp;
    temp.real = c1.real + c2.real;
    temp.imag = c1.imag + c2.imag;
    return temp;
}//一般返回临时变量
```

返回拷贝，不是引用，效率不太高?为了解决这个问题:可以==返回值优化==，第一个return没有拷贝，直接返回的是一个对象(无拷贝)，先计算，最后生成一个对象返回。（在return语句里面new）

->

为二元运算符，重载的时候按照一元操作符重载描述。A*operator->()

```
//包裹被操作的资源，在意外退出的条件下，自动删除原来的资源
class AWrapper{//不包含逻辑
    A* p;// ? T p; 支持多个类型
    public:
        AWrapper(A *p) { this->p = p;}
        ~AWrapper() { delete p;}
        A*operator->() { return p;}//32min重新听一下
};//RAII 资源获取及初始化
//函数返回，销毁局部指针的时候会直接删除
```

## 结合操作符重载的多维数组结果

```
class Array2D{
    private:
        int *p;
        int num1, num2;
    public:
        class Array1D{//Surrogate 多维，proxy class
            public:
                Array1D(int *p) { this->p = p; }
                int& operator[ ] (int index) { return p[index]; }
                const int operator[ ] (int index) const { return p[index]; }
            private:
                int *p;
        };
        Array2D(int n1, int n2) {
            p = new int[n1 * n2];
            num1 = n1;
            num2 = n2;
        }
        virtual ~Array2D() {
            delete [] p;
        }
        Array1D operator[](int index) {
            return p + index * num2;//return的值和int*相同，构造函数不能声明成显式构造函数。
        }
        //这里为什么是array1D?通过构造函数进行类型转换
        const Array1D operator[] (int index) const {
            return p+index*num2;
        }
};
```

## 重载new和delete操作符

### new的部分

#### 方法

1. 调用系统存储分配，申请一块较大的内存
2. 针对该内存，自己管理存储分配、去配
3. 通过重载new与delete来实现
4. 重载的new与delete是静态成员(隐式的，不需要额外声明，不允许操作任何类的数据成员)
5. 重载的new与delete遵循类的访问控制，可继承(注意派生类和继承类的大小问题，开始5min左右)

有些我们重复新建销毁的，比如Restful的可以单独管理

#### new的其他重载

可以重载成全局函数，也可以重载成类成员函数，支持定向处理(如下面例子)

```
if(size != sizeof(base))
    return ::operator new (size);//调用全局标准库的new进行size的分配，标准库的new永远是可以使用的
operator new;
new A[10];
operator new [];
void * operator new (size_t size, void*)//是不可以被重载的，标准库版本
void * operator new (size_t size, ostream & log);//可以同时写入到日志
void * operator new (size_t size, void * pointer);//定位new，placement new，被调用的时候，在指针给定的地方的进行new(可能预先已经分配好的)，分配比较快，长时间运行不被打断(不会导致内存不足)

//也可以new的时候指定自定义地址
class A{};
char buf[sizeof(A)];
A* a = new(buf) A;//定位new，不用分配内存，直接使用buf指向的区域
```

### delete的部分

1. void operator delete(void *,size_t size)
2. 名：operator delete
3. 返回类型:void
4. 第一个参数:void *(必须)：被撤销对象的地址
5. 第二个参数:可有可无;如果有，则必须为size_t类型：被撤销对象的大小
6. delete 的重载只能有一个
7. 如果重载了delete，那么通过 delete 撤消对象时将不再调用内置的(预定义的)delete
8. 动态删除其父类的所有的。
9. 如果子类中有一个虚继承函数，则size_t大小会根据继承情况进行确定大小

# 模板

模板是一种代码复用机制，模板定义多个类的时候需要显式实例化，如果用不到的化，则不会实例化模板。

模板是不同于重复耦合和函数重载的一种更高效的解决方案

## 实例化

1. **隐式实现**
2. 根据具体模板函数调用

## 函数模板的参数

1. 可有多个类型参数，用逗号分隔:`template <class T1, class T2>`
2. 可带普通参数:
    - **必须列在类型参数之后**:`template <class T, int size>`
    - 调用时需显式实例化，使用默认参数值可以不显式实例化
3. 类型参数和普通参数都可以给出默认参数，但是必须从右侧向左侧给出

```
template <class T1, class T2>
void f(T1 a, T2 b) {}
template <class T, int size>
void f(T a) {T temp[size];}
f<int,10>(1);
```

## 类属函数

1. 使用宏解决:`#define max(a,b) ((a)>(b)?(a):(b))`,只有简单功能，没有类型检查

## 函数模板

```
//int和double都可以使用，编译器编译的并不是之下的代码，而是T转化成具体代码，然后分别编译
template <typename T>
void sort(T A[], unsigned int num) {
    for(int i=1; i<num; i++)
        for (int j=0; j< num - i; j++) {
            if  (A[j] > A[j+1]) {
                T t = A[j];
                A[j] = A[j+1];
                A[j+1] = t;
            }
        }
}
class C {...}
C a[300];
sort(a, 300);//没有重载操作符>
```

## 类属模板

```
//类属模板需要显式实例化
template <class T>
class Stack {
    T buffer[100];
    public:
        void push( T x);
        T pop();
};

template <class T>
void Stack <T> ::push(T x) {...}

template <class T>
T Stack <T> ::pop() {...}

//如下是显式实例化
Stack <int> st1;
Stack <double> st2;
```

## 给出位置

函数模板一般是在头文件中给出完整的定义。

```
//file1.h
template <class T>
class S {
    T a;
    public:
        void f();
};
//file1.cpp
#include "file1.h"
template <class T>
void S<T>::f(){...}

template <class T>
T max(T x, T y){return x>y?x:y;}
void main() {
    int a,b;
    max(a,b);//实例化函数模板
    S<int> x;
    x.f();
//file2.cpp
#include "file1.h"
extern double max(double,double);
void sub(){
    max(1.1,2.2);//error
```

```
    S<float> x;
    x.f();//error
}
//不能通过编译，为什么？file2.cpp找不到max定义，也找不到完整的S代码
```

# 异常处理

## 特征

- 可以预见
- 无法避免

## 作用

### 提高程序鲁棒性(Bobustness)

```
void f(char *str) {//str可能是用户的一个输入
    ifstream file(str);
    if (file.fail()) {
        // 异常处理
    }
    int x;
    file >> x;
}
```

问题：发现异常之处与处理异常之处不一致，怎么处理?函数中的异常要告知调用者

### 常见处理方式

1. 函数参数:
    - 返回值(特殊的，0或者1)
    - 引用参数(存放一些特定的信息)
2. 逐层返回

### 缺陷

程序结构不清楚

相同的异常，不同的地方，需要编写相同的处理了逻辑是不合理的

传统异常处理方式不能处理构造函数出现的异常

## 处理机制

1. C++异常处理机制是，一种专门、清晰描述异常处理过程的机制
2. `try`：监控
3. `throw`：抛掷异常对象，不处理
4. `catch`：捕获并处理

```
try{
    //<语句序列>
```

```
    //监控
}throw//<表达式>，可以是基本类型，拷贝构造函数用来拷贝类
catch(<类型>[<变量>]){//变量不重要可以省略
    //<语句序列> 捕获并处理
    //依次退出，不要抛出指向局部变量的指针，解决:直接抛出对象，自动进行拷贝
}
```

## catch的用法

类型：异常类型，匹配规则同函数重载(精确匹配只有底下三种，int转double都不行)

1. 允许从非常量到常量转换
2. 允许从派生类到基类转换
3. 允许数组和函数转换成指针

变量:存储异常对象，可省

一个try语句块的后面可以跟多个catch语句块，用于捕获不同类型的异常进行处理

```
void f() {
    throw 1;
    throw 1.0;
    throw "abcd";
}
try {
    f();
}catch (int)// 处 理 throw 1;
{...}
catch(double)//throw 1.0
{...}
catch(char *)//throw "abcd"
//字符串优先解释为char *
{...}
```

## 异常处理的嵌套

调用关系

f->g->h

如果所抛掷的异常对象如果在调用链上未被捕获，则由系统的abort处理（程序直接中断）。

```
//第二节课10min
f(){
    try{
        g();
    }catch (int)
     { … }
    catch (char *)
```

```
    { … }
}
g(){
    try{
        h();
    }catch (int)
    { …  }
}
h(){
  throw 1;   //由g捕获并处理
  throw "abcd"; //由f捕获并处理
}
```

# 定义异常类

## 注意catch块排列顺序

这样子保证了继承顺序(重要)，顺序向下检查是否符合条件，一旦符合条件就不再向下查找了。

```
class FileErrors { };
class NonExist:public FileErrors { } ;
class WrongFormat:public FileErrors { } ;
class DiskSeekError:public FileErrors { };

int f(){
    try{
        WrongFormat wf;
        throw wf;
    }catch(NonExists&){...}
    catch(DiskSeekError&){...}
    catch(FileErrors){...}//最后一个可以接住，派生类像基类转换是允许的
}
int f(){
    try{
        WrongFormat wf;
        throw wf;
    }catch(FileErrors){...}//这样子底下都捕获不到
    catch(NonExists&){...}
    catch(DiskSeekError&){...}
}
//Catch exceptions by reference
//尝试多继承，而不是拷贝，避免冗余
```

## 实例

```
class MyExceptionBase {};
class MyExceptionDerived: public MyExceptionBase { };
void f(MyExceptionBase& e) {
    throw e;//调用拷贝构造函数
```

```
}
int main() {
    MyExceptionDerived e;
    try {
        f(e);
    }catch(MyExceptionDerived& e) {
        cout << "MyExceptionDerived" << endl;
    }catch(MyExceptionBase& e) {
        cout << "MyExceptionBase" << endl;
    }
}
//输出:MyExceptionBase，为什么?调用了拷贝构造函数，拷贝构造的结果是MyExceptionBase类型的对象
```

# 特例

1. 无参数 `throw`：将捕获到的异常对象重新抛掷出去`catch(int){throw;}`
2. `catch(...)`：默认异常处理,这三个点是标准语法,捕获所有异常
3. 实现：不影响对象布局:程序状态`<->`析构函数、异常处理器，对程序验证特征的支持
4. 构造函数的初始化表前，放置`try-catch`同样捕获异常1

```
//对程序验证特征的支持
template<class T, class E>
inline void Assert(T exp, E e)
{
    if (DEBUG)
        if (!exp) throw e;
}
```

## 处理多出口导致的碎片问题

如何应对多出口引发的处理碎片问题，如果多个地方`throw`，则意味着这里有多个出口。

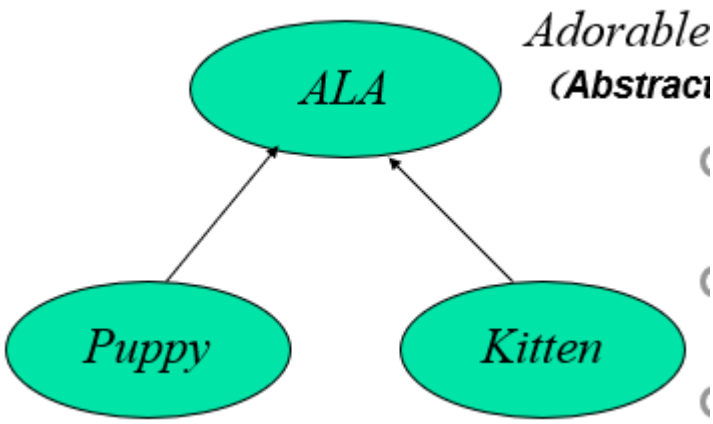
`Java`中在异常处理这一部分提供了`finally`操作，无论在哪里没有抛出最后都会执行finally，将内存缓存进行自己的处理。可是C++中没有`finally`，那怎么进行处理呢?

在C++中，执行完异常处理后，必然执行析构函数。

```
//Know what functions C++ silently writes and calls
class Empty { };
class Empty {
    //以下是C++默认提供给空类的方法
    Empty();
    Empty(const Empty&);
    ~Empty();
    Empty& operator=(const Empty&);
    Empty *operator &();
    const Empty* operator &() const;
};
```

## 异常处理防资源泄露的例子

1. 收养中心每天产生一个文件，包含当天的收养个案信息
2. 读取这个文件，为每个个案做适当的处理




```
class ALA{//Adorable Little Animal
    public:
        virtual void processAdoption() = 0;
};
class Puppy: public ALA{
    public:
        virtual void processAdoption();
};
class Kitten: public ALA{
    public:
        virtual void processAdoption();
};
void processAdoptions(istream& dataSource){
    while (dataSource){
        ALA *pa = readALA(dataSource);
        try{
            pa->processAdoption();//处理可能会出现问题
        }catch (…){
            delete pa;
            throw;
        }
        delete pa;//正常执行也要进行处理，这就是多出口的问题
    }
}
```

结构破碎：被迫重复"清理码"2次delete的pa(不符合集中式处理的想法、同时容易导致维护困难的问题)

集中处理？用析构函数（智能指针）

```
template <class T>
class auto_ptr{
    public:
        auto_ptr(T *p=0):ptr(p) {}
        ~auto_ptr() { delete ptr; }
        T* operator->()  const { return ptr;}
        T& operator *()  const { return *ptr; }
    private:
        T*  ptr;
};
//结合智慧指针使用
void processAdoptions(istream& dataSource){
    while (dataSource){
        auto_ptr<ALA> pa(readALA(dataSource));
        pa->processAdoption();//只要对象结束，就会自动delete
    }
}
```

# C++ 11新特性

## 右值引用

### 简介

在C++中，非const引用可以绑定到左值，而const引用可以绑定到左值和右值，但是没有什么可以绑定到非const r值。右值不可以绑定非常量引用，避免临时变量的修改造成的问题

```
class A{};
A getA(){
    return A();//右值
}
int main() {
    int a = 1;
    int &ra = a; //OK
    const A &ca = getA();//OK
    A &aa = getA();//ERROR，右值不能给左值引用
}
```

### 右值引用只能绑定在右值上

```
class A{
    int val;
    void setVal(int v) {
        val = v;
    }
};
A getA(){
    return A();
}
//知道风险，并且想要改变新对象，就是右值引用&&
int main() {
    int a = 1;
    int &ra = a; //OK
    const A &cra = getA();//OK
    A &&aa = getA();//OK
    aa.setVal(2);//OK
    //…
}
```

## 外部模板

避免不必要的实例化。Avoid of unnecessary instantiation.

```
//myfunc.h
template<typename T>
    void myfunc(T t){}

//test.cpp
#include "myfunc.h"
int foo(int a){
    myfunc(1);
    return 1;
}
//main.cpp
#include "myfunc.h"
//如果没有以下的模板，那么编译器会先去实例化模板，新的方式外部模板可以避免多次实例化的问题
/*Tell compiler: this instance has been
instantiated in another module!*/
extern template void myfunc<int>(int);

int main() {
    myfunc(1);
}
```

# 常量表达式

1. 提供了更一般的常量表达式
2. 允许常量表达式使用用户自定义类型
3. 提供一种方法来确保在编译时完成初始化
4. 必须在编译的时候可以确定常量表达式

## 实例

```
enum Flags { GOOD=0, FAIL=1, BAD=2, EOF=3 };
constexpr int operator| (Flags f1, Flags f2)  {
    return Flags(int(f1)|int(f2));
}//如果不加constexpr则结果被认为是变量不能使用在case中
void f(Flags x) {
    switch (x) {
        case BAD: /* ... */break;
        case EOF: /* ... */ break;
        case BAD|EOF: /* ... */ break;//OK，必须是简单的确认的值
        default: /* ... */ break;
    }
}
void f(Flags x) {
    switch (x) {
        case bad_c(): /* ... */break;
        case eof_c(): /* ... */ break;
        case be_c(): /* ... */ break;
        default: /* ... */ break;
    }
}
```

```
constexpr int bad_c();
constexpr int eof_c();
constexpr int be_c();
```

### 常量对象

1. 所有评估都可以在编译时完成。 因此，提高了运行时间效率。
2. 编译时确定的

```
struct Point {
    int x,y;
    constexpr Point(int xx, int yy) : x(xx), y(yy) { }
};
int main() {
    constexpr Point origo(0,0);//完全常量，在常量表上
    constexpr int z = origo.x;

    constexpr Point a[] = {Point(0,0), Point(1,1), Point(2,2) };
    constexpr int x = a[1].x; // x becomes 1
}
```

## Lambda Function

1. Also names as Lambda Expression.
2. A mechanism for specifying a function object

### 实例1

```
bool cmpInt(int a, int b) {return a < b;}
class CmpInt {
    bool operator()(const int a, const int b) const {
        return a < b;
    }
};
int main() {
    std::vector<int> items { 4, 3, 1, 2 };
    std::sort(items.begin(), items.end(), cmpInt); //Function Pointer
    std::sort(items.begin(), items.end(), CmpInt()); //Function Object (Functor)
    std::sort(items.begin(), items.end(),
        [](int a, int b) { return a < b; } //Lambda Function
    );
    return 0;
}
template <class RandomAccessIterator, class Compare>
  void sort (RandomAccessIterator first, RandomAccessIterator last, Compare comp)
{
    //…
    if ( comp(*it1, *it2) )
```

```
    //…
}
//std::function 是C++对所有可调用的函数的封装
std::function<bool(int, int)> f1(cmpInt);
std::function<bool(int, int)> f2(CmpInt);
std::function<bool(int, int)> f3([](int a, int b) { return a < b;} );
```

实例2

```
vector<string> str_filter(vector<string> &vec, function<bool(string &)> matched){
    vector<string> result;
    for (string tmp : vec) {
        if (matched(tmp))
            result.push_back(tmp);
    }
    return result;
}
//可以会用局部变量，40min
int main(){
    vector<string> vec = {"www.baidu.com", "www.kernel.org", "www.google.com"};
    string pattern = ".com";
    vector<string> filterd = str_filter(vec,
        [&](string &str) {
            if (str.find(pattern) != string::npos)
                return true;
            return false;
        });
}
```

相关语法

| 符号 | 含义 |
| --- | --- |
| `[]` | Capture nothing |
| `[&]` | Capture any referenced variable by reference |
| `[=]` | Capture any referenced variable by making a copy |
| `[=, &foo]` | Capture any referenced variable by making a copy, but capture variable foo by reference |
| `[bar]` | Capture bar by making a copy; don't copy anything else |

# 委托构造

```
#define MAX 256
class X {
    int a;
```

```
    void validate(int x) { if (0<x && x<=MAX) a=x; else throw bad_X(x); }
public:
    X(int x) { validate(x); }
    X() { validate(42); }
    // ...
};
class X {
    int a;
public:
    X(int x) { if (0<x && x<=max) a=x; else throw bad_X(x); }
    X() :X(42) { } // 构造函数里面调用构造函数
    // ...
};
X(int x = 42) ?
```

## 统一初始化

```
//Old style initialization
vector<int> vec;
vec.push_back(1);
//…
//New style initialization
vector<int> vec = {1, 2, 3};
//Compiler will translate {} as initializer_list<int> 新的初始化表
template class vector<T> {
    //..
    vector(initializer_list<T> list) {
        for (auto it = list.begin(); it != list.end(); ++it)
            push_back(*it);
    }
};
int arr[] = {1, 2, 3}; //OK
vector<int> vec = {1, 2, 3};
A a= {1, 2, 3};

class A{
    int x, y, z;
    //Default generated by compiler
    A(initializer_list<int> list) {
        auto it = list.begin();
        x = *it++;
        y = *it++;
        z = *it;
    }
};
//Uniform Initialization achieved!
int arr[] = {1, 2, 3};
vector<int> vec = {1, 2, 3};
A  a = {1, 2, 3};
```

## 空指针nullptr

```
void f(int);//f(0)
void f(char*);

f(0);           // call f(int)
f(nullptr);     // call f(char*)

f(NULL);// call f(int)
```