目录

三、class

3.1 C语言-面向过程

(1)定义：定义结构体、对结构体的操作函数

(2)使用：传入结构体指针、对应的参数

```
//定义
typedef struct point {
    float x;
    float y;
}+Point;
void print(const Point *p){
    printf("%f %f\n",p->x,p->y);
}
void move(Point* p,int dx,int dy){
    p->x += dx; p->y += dy;
}
//使用
Point a;
a.x = 1;a.y = 2;
move(&a,2,2);
print(&a);
```

3.2 C++语言-面向对象

(1)定义：将**对象**和**操作**绑定到一起

(2)使用：操作Point对象

```
//定义
class Point{
public:
    void init(int x,int y);
    void move(int dx,int dy);
    void print()const;

private:
    int x;
    int y;
}
//实现
void Point::init(int ix,int iy){
    x = ix; y = iy;
}
void Point::move(int dx,int dy){
    x += dx; y += dy;
}
void Point::print() const{
    cout << x << ' ' << y << endl;
}
//使用
Point a;
a.init(1,2);
a.move(2,2);
a.print();
```

3.3 :: resolver

< Class Name >::< function name >

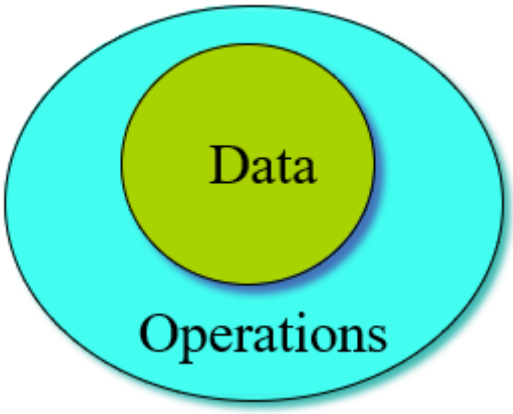
::< function name >

```
void S::f(){
    ::f();//这里调用的f()是全局的函数f(),而非自己
    ::a++;//全局的a++
    a--;//S里的a++
}
```

Objects = Attributes + Services

(1)通过接口修改数据，而非直接修改数据

(2)直接访问变量，变量的声明者对变量没有控制权，无法控制操作者的行为



3.4 Container容器

container：是一个能够保存其它对象的**object**

stash：是一个可以存储变量、运行时可以扩展的**container**

要求：

(1)无类型**typeless**

(2)存储相同类型的**type**

(3)操作：**add()**、**fetch()**

(4)当需要扩展时，可以扩展内存

3.4.1 声明

```
#ifndef STASH2_H
#define STASH2_H
class Stash{
    int size;    //每个空间的大小
    int quantity;//空间的个数

    int next;    //下一个为空的空间
    //进行动态内存访问
    unsigned char* storage;
    void inflatte(int increase);//增长内存
public:
    Stash(int size);
    ~Stash();
    int add(void * element);
    void* fetch();
    int count();
};
#endif
```

3.4.2 定义

```
#include "Stash2.h"
#include "../require.h"
#include <iostream>
#include <cassert>
using namespace std;
const int increment = 100;
```

```cpp
Stash::Stash(int sz) {//构造函数,函数名和类名相同
    size = sz;
    quantity = 0;
    storage = 0;
    next = 0;
}
int Stash::add(void* element) {
    if(next >= quantity) //剩余内存不够,增加内存
        inflate(increment);
    //将元素复制到storage[]中,开始另一个新内存
    int startBytes = next * size;
    unsigned char* e = (unsigned char*)element;
    for(int i = 0; i < size; i++)
        storage[startBytes + i] = e[i];
    next++;
    return (next - 1); //当前元素存储的空间编号
}

void* Stash::fetch(int index) {
    require(0 <= index, "Stash::fetch (-)index");
    if(index >= next)
        return 0;//访问为空
    //返回访问的内存地址
    return &(storage[index * size]);
}

int Stash::count() {
    return next; //Stash中的元素个数
}

void Stash::inflate(int increase) {
    require(increase > 0,
        "Stash::inflate zero or negative increase");
    int newQuantity = quantity + increase;
    int newBytes = newQuantity * size;
    int oldBytes = quantity * size;
    unsigned char* b = new unsigned char[newBytes];
    for(int i = 0; i < oldBytes; i++)
        b[i] = storage[i]; //将旧内存中的元素复制到新内存中
    delete [](storage); //删除旧内存
    storage = b; //storage指向新内存
    quantity = newQuantity;
}

Stash::~Stash() {//析构函数
    if(storage != 0) {
        cout << "freeing storage" << endl;
        delete []storage;
    }
}
```

3.5 this

this：指向对象本身的指针

```
class MyClass{
    int x;
    int y;
public:
    MyClass(){x = 0; y = 0;}
    void foo(int x, int y){
        this->x = x;
        this->y = y;//在运行时,this会被看为指向该对象的一个指针
    }
};
```

3.6 Object & class

对象**Object**：数据(**properties or status**)+操作(**functions**)

class定义**object**，**object**是**class**的一个实例

3.7 Definition of a class

头文件***.h**：

```
#include "stdafx.h" //VS的预编译头文件
#pragma once //表示当前头文件只编译一次，防止重复引用
```

class的声明放入***.h**

class的函数体、静态成员变量放入***.cpp**

3.8 compile unit

(1)**compile**只会看到***.cpp**文件，生成 ***.obj**文件

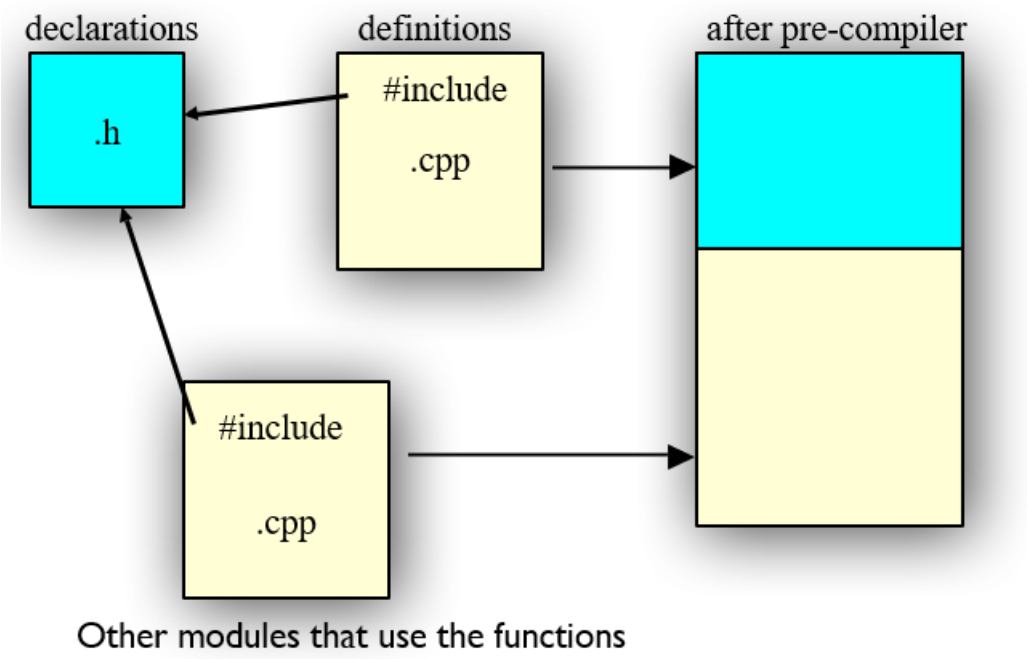
(2)**linker**会链接所有的***.obj**文件，生成一个 ***.exe**文件

(3)如果要使用其它***.cpp**的文件，需要使用 ***.h**

3.9 The header files

(1)如果函数定义在头文件，那么必须在任何使用该函数和定义该函数的地方**include**该头文件

(2)如果**class**定义在头文件，那么必须在任何使用**class**和定义**class**的地方**include**该头文件

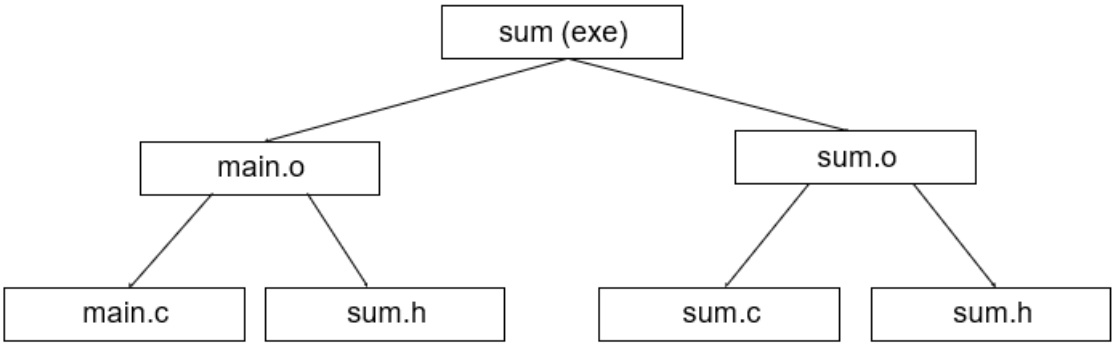


3.10 Makefile

3.10.1 Project的结构

可以看作是一个有向无环图**DAG**

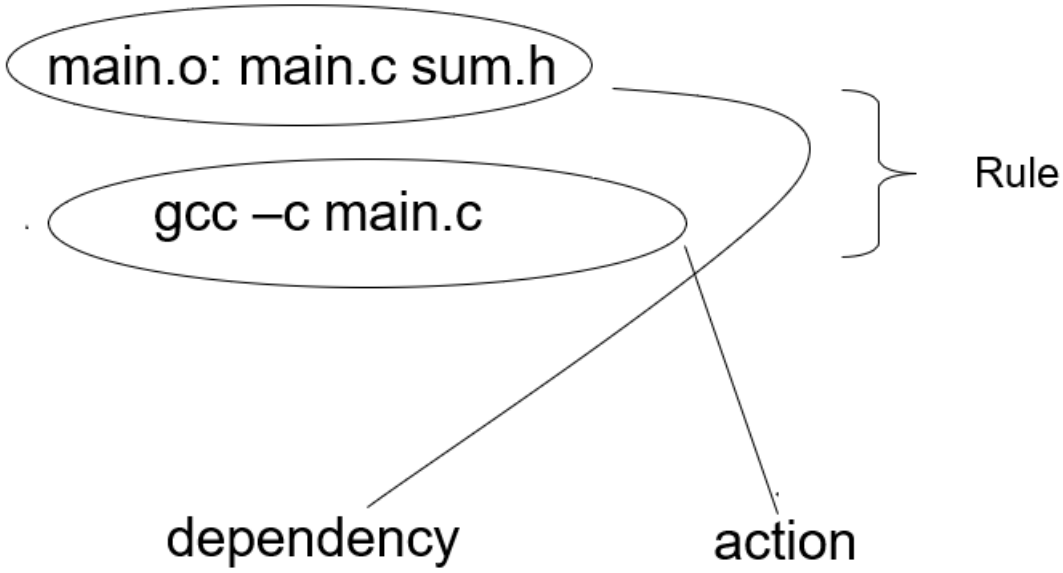
以一个工程为例--包含main.c、sum.c、sum.h，sum.h在所有.c文件中都包含



此时的**makefile**为：

```
sum: main.o sum.o
    gcc -o sum main.o sum.o
main.o: main.c sum.h
    gcc -c main.c
sum.o: sum.c sum.h
    gcc -c sum.c
```

3.10.2 makefile的语法规则：



简化语法：

```
sum: main.o sum.o
    gcc -o $@ main.o sum.o # $@ ==> sum
main.o  sum.o: sum.h
    gcc -c $*.c # $* ==> main、sum
```

3.10.3 makefile的过程

(1)构建项目依赖树

(2)创建第一条指令的目标，当且仅当一下两种情况成立之时：

(a)目标文件不存在

(b)目标文件比它的一个依赖项更早生成

(3)**makefile**的作用是保证最小的编译数量，因此需要正确写明

不要写成下面这样(这样写会直接重新编译所有文件)

```
prog: main.c sum1.c sum2.c
gcc -o prog main.c sum1.c sum2.c
```

另一个例子

```
# 定义变量
BASE = /home/blufox/base
CC   =   gcc  #编译器
CFLAGS = -O -Wall  #编译时的宏
EFILE = $(BASE)/bin/compare_sorts
NCLS =  -I$(LOC)/include  #头文件的位置
LIBS =  $(LOC)/lib/g_lib.a $(LOC)/lib/h_lib.a
LOC = /usr/local
OBJS = main.o   another_qsort.o  compare.o  quicksort.o

$(EFILE): $(OBJS)
    @echo “linking …” #回显
```

```
    @$(CC) $(CFLAGS) -o $@ $(OBJS) $(LIBS)

$(OBJS): compare_sorts.h
    $(CC) $(CFLAGS) $(INCLS) -c $*.c

# 清除临时文件
clean:
    rm *~ $(OBJS)
```

3.10.4 makefile的其它参数

(1)我们可以在一个**makefile**里定义多个对象

(2)**clean**：删除临时文件

```
OBJS = main.o   another_qsort.o  compare.o  quicksort.o
clean:
    rm *~ $(OBJS)
```

(3)**make**：创建***.exe**文件

(4)**make clean**：删除临时文件

3.10.5 makefile里面的自定义参数

```
make
PAR1 = 1
PAR2 = soft1
#定义参数PAR1=1，PAR2="soft1"
```

注意：将值分配给 **makefile** 中的变量会覆盖从命令行传递的任何值，例如：

```
make PAR = 1
PAR = 2
#makefile中的PAR值将是2，覆盖从命令行发送的值
```

3.10.6 makefile里面的条件语句

```
ifeq(value1, value2)
    body of if
else
    body of else
endif
```

例：

```
sum: main.o sum.o
    gcc -o sum main.o sum.o

main.o: main.c sum.h
    gcc -c main.c

#判断将哪个文件编译为sum.o
ifeq ($(USE_SUM), 1)
```

```
    sum.o: sum1.c sum.h
        gcc -c sum1.c -o $@
else
    sum.o: sum2.c sum.h
        gcc -c sum2.c -o $@
endif
```