# 常量和常量类：

## 类中的成员变量是一个常量：

Class A{

private:

const int a;

int b;

public:

void A():a(100) {……}

//当类的成员变量是一个const修饰的常量时只能在构造函数的初始化列表中进行初始化赋值

};

## 类实例是一个常对象

class A {

  private:

    const int a;

    int b;

  public:

    A();

    ~A();

    void setB(int n);

    void getA() const;

    void getB() const { cout << "b:" << b << endl; }

};

A::A() : a(100), b(11) {

    cout << "a:" << a << endl;

    cout << "b:" << b << endl;

}

A::~A() {}

void A::setB(int n) {

    b = n;

    return;

}

void A::getA() const {

    cout << "a:" << a << endl;

    cout << "b:" << b << endl;

}

int main(int argc, char \*argv[]) {

    A a;

    const A aa;

    // aa.setB(111);

    aa.getA();

    aa.getB();

    return 0;

}

在这个程序中，main函数中A aa；定义了一个类A的实例化对象aa; 并且使用const前缀修饰实例化的对象是**常量对象**；

这意味着实例化的对象aa不能再被修改，且使用aa进行”.”调用的成员函数，只能是那些有const后缀的成员函数——aa.setB因为函数有赋值操作且函数没有const后缀修饰，会编译报错；

***const A aa***；

这一句就限定了实例化的aa，不能调用任何含有对类的对象有赋值修改操作的成员函数。也就是不能对其内部成员进行修改；

# 预编译,编译和链接

## 预编译g++ -E ：

#include <iostream>

using namespace std;

int main(int argc, char \*argv[]) {

    cout << "hello!" << endl;

    return 0;

}

// 如下命令预编译cpp文件为另外一个a.cpp

//  g++ -E .\预编译hello.cpp -o a.cpp

// 预编译的结果就是把源文件头上include的.h以及相关的库文件替换为对应的.h或者库文件的实际内容

使用g++ -E .\预编译hello.cpp -o a.cpp预编译后,a.的内容如下：



可以看到文件的内容增加了更多，其实预编译就是把include的.h或者库文件直接在文件中替换为其实际的内容。

从\*.c/\*/cpp –> \*.o的过程成为编译，

从\*.o->\*.exe的过程需要经过链接，所谓的连接就是把各个\*.o文件链接到一起然后生成\*.exe

# 动态库文件的生成以及使用

## 生成动态库

如下动态库的源文件mylib.c

int max(int a,int b)

{

        if(a>b)

                return a;

        return b;

}

如下是生成.so文件通用 Makefile模板：

CC=gcc

SRCS=mylib.c\

OBJS=$(SRCS:.c=.o)

MYLIB=libmylib.so

start:$(OBJS)

        $(CC) -o $(MYLIB) $(OBJS) -shared

.c.o:

        $(CC) -o $@ -c $< -fPIC

clean:

    rm -fr \*.o

其中：

* SRCS=mylinb.c\

自己编写的动态库的源文件，如果有新的源文件，直接在SRCS后追加文件名即可；

* $(CC) -o $(MYLIB) $(OBJS) –shared

这是把.o文件编译成为.so共享库文件，添加-shared后缀告诉编译器变异的是一个库文件；

* $(CC) -o $@ -c $< -fPIC

这是把-c文件编译为.o文件,**添加-fPIC即使告诉编译器生成的库文件中函数不添加偏移量**，这是生成动态库文件所必须的编译选项；

* MYLIB=libmylib.so

MYLIB是makefile编译的目标so文件；

libmylib.so是生成的.so动态库文件名称，**在liuux约定下，动态库so文件必须以lib开头，以.so结尾**；

## 使用动态库

### 编写头文件

添加能够被c和c++同时使用的，动态库中函数的.h头文件：

#ifndef MYLIB\_H

#define MYLIB\_H

#ifdef \_\_cplusplus

#define

extern "C"

{

#endif

int max(int a,int b);

#ifdef \_\_cplusplus

}

#endif

#endif

* extern "C"

c下编写的函数要在cpp中使用就需要用extern “C”包含函数的申明语句；

* #ifdef \_\_cplusplus

用于判断是否是cpp的编译器(app的编译器会自带\_\_cplusplus宏)

如果判断是cpp的编译器，那么就给函数预编译添加上extern “C”的包含语句；

### 调用动态库的主文件（.c源文件）：

#include<stdio.h>

#include "mylib.h"

int main()

{

    printf("%d\n",max(10,20));

    return 0;

}

在这个文件中添加了#include “mylib.h”的头文件引用。

编译c程序的通用makefile如下：

CC=gcc

SRCS=calllib.c\

OBJS=$(SRCS:.c=.o)

EXE=myapp

start:$(OBJS)

    $(CC) -o $(EXE) $(OBJS) -L. -lmylib

.c.o:

    $(CC) -o $@ -c $<

clean:

    rm -fr \*.o

* SRCS=calllib.c\

主程序的源文件，如果要新增源文件，直接在SRCS后追加即可；

* EXE=myapp

编译后生成的主程序的文件名；

* $(CC) -o $(EXE) $(OBJS) -L. –lmylib

-L. –lmylib 分别是指定编译时的链接的库文件的路径，以及路径下的库文件的名称；

-L. 就是在当前路径下连接库文件，–lmylib就是连接的库文件名称为libmylib.so(linux约定编译的时候会

自动按照lib开头，so结尾的文件查找库，所以只写上去掉lib前缀和.so后缀的库文件部分名称即可）

### 调用动态库的主文件（.cpp源文件）：

和.c文件差异不大，就是使用的不同的编译器

#include<stdio.h>

#include "mylib.h"

int main()

{

    printf("%d\n",max(10,20));

    return 0;

}

编译cpp程序的通用makefile如下：

CC=g++

SRCS=calllib.cpp\

OBJS=$(SRCS:.cpp=.o)

EXE=myapp

start:$(OBJS)

    $(CC) -o $(EXE) $(OBJS) -L. -lmylib

.c.o:

    $(CC) -o $@ -c $<

clean:

rm -fr \*.o

只是使用的编译器CC的名称不同，其他的和c相同； 叭叭叭

### 添加使用动态库的环境变量

Linux默认是只有so文件被放在在系统默认的lib路径下，so中的函数才能被调用，如果需要自已定义调用动态库的路径，需要添加环境变量；

所以当我们写了一个动态库以后：

* 方法1：把.so拷贝到系统的库文件看路径下,如/lib,或者/usr/lib等；
* 方法2：在~/.profile中添加如下环境变量：

export PATH=$PATH:.

export LD\_LIBRARY\_PATH=$LD\_LIBRARY\_PATH:.

在编辑追加完这两项环境变量后，还需要在命令行执行如下命令使其生效：

$ . ~/.profile