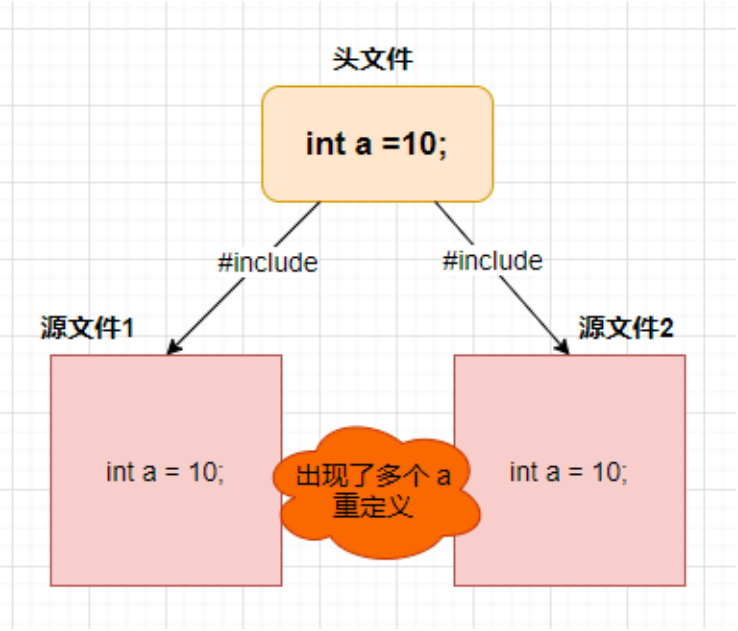
# 第6章-探索程序

author：岳石磊 copyright：科林明伦 内部资料禁止外泄

## 1. 头文件-源文件

头文件（.h）和源文件（.cpp）两者的区别：

1. 默认情况下，头文件不参与编译，而每个源文件自上而下独立编译。
2. 通常我们将声明的变量、类型、函数、宏、结构体和类的定义等放于头文件（.h文件），将变量的定义初始化、函数的定义实现放于源文件中，这样方便于我们去管理、规划，更重要的是避免了重定义的问题。




类中的成员函数在对应的源文件中定义时，一定要加上**类名作用域**。

```
//test.h
class CTest {
    void fun();
};
```

```
//test.cpp
void CTest::fun();
```

静态常量成员一定要在**源文件**中进行定义初始化（而不是头文件中）。
常函数： 保留const关键字。

```
//test.h
void fun()const;

//test.cpp
void CTest::fun()const {}
```

静态成员函数：去掉static 关键字。

```
//test.h
static void fun();

//test.cpp
void CTest::fun() {}
```

虚函数： 去掉virtual关键字。

```
//test.h
virtual void fun();

//test.cpp
void CTest::fun() {}
```

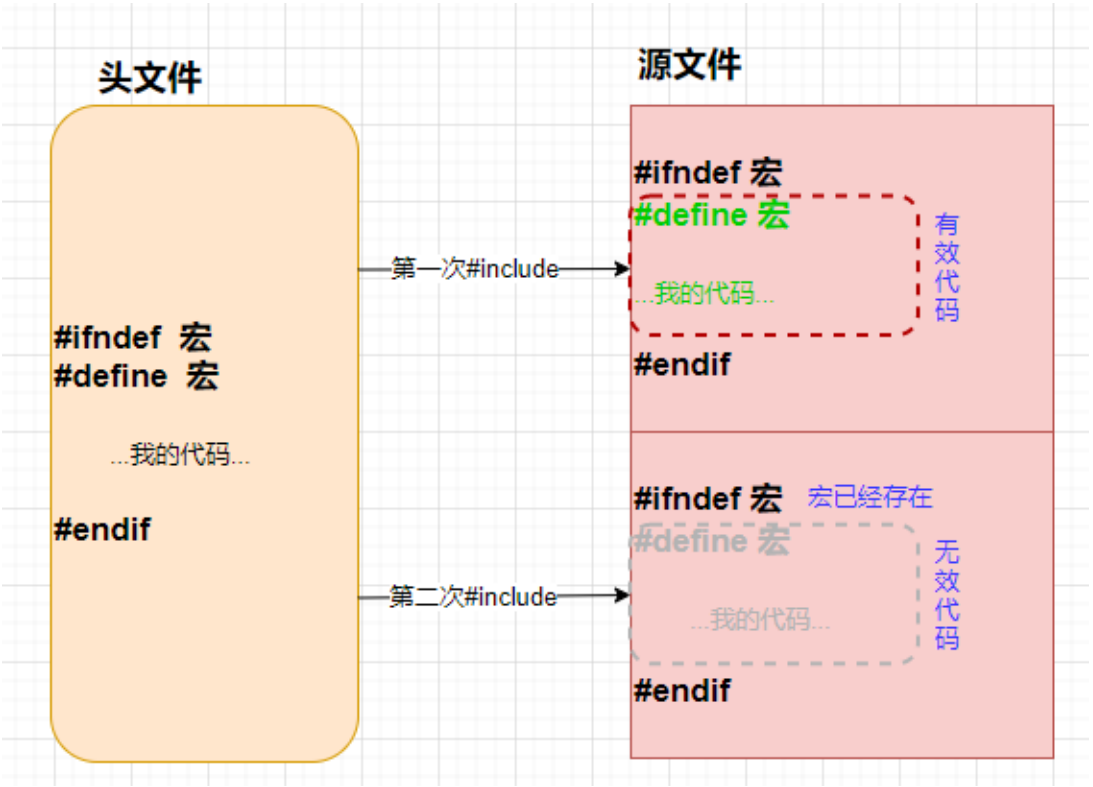
纯虚函数： 不需要实现。

```
//test.h
virtual void fun() = 0;
```

## 2. 头文件重复包含

`#pragma once` 的作用：直接告诉编译器这个文件在源文件中只包含一次，相对来说效率比较高。
宏判断 `#ifndef #define ... #endif` ，基于逻辑宏判断，在大量头文件时，编译速度降低，耗时增加。而且需要考虑宏重名的问题，一般情况下宏的名字与当前文件名对应，但是并不能保证一定不重

名，如果不同路径下存在相同的文件，也可能会重复。




## 3. 程序生成过程

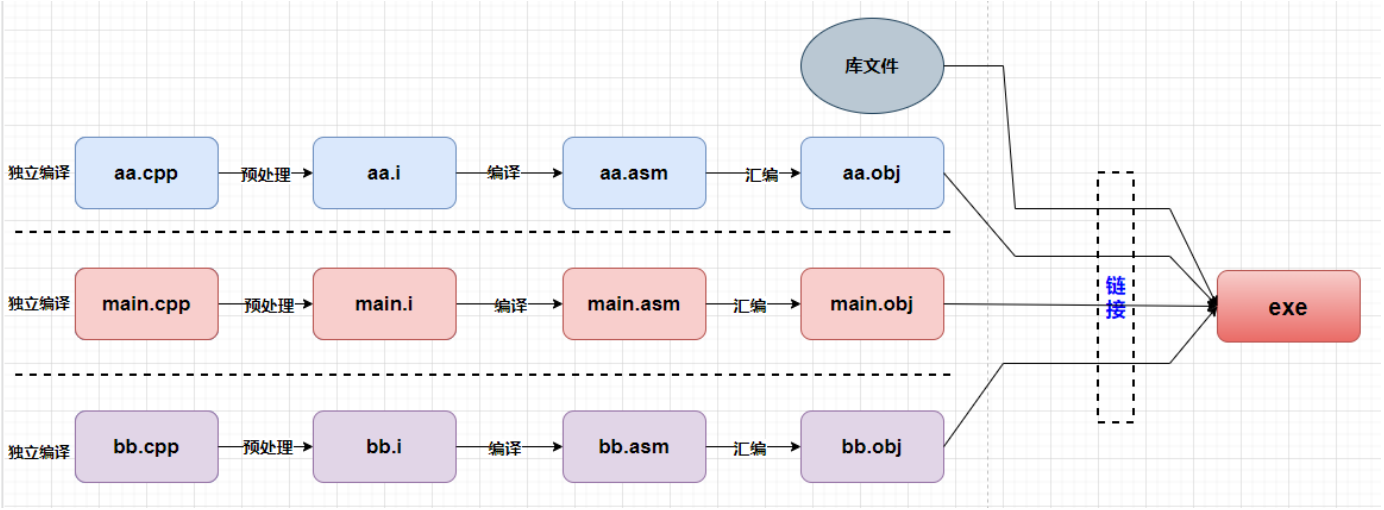
1.**预处理**Preprocessing
将源文件（.cpp）初步处理，生成预处理文件（.i）：

1. 解析 #include 头文件展开替换。
2. 宏定义指令：#define 宏的替换，#undef等。
3. 预处理指令：解析 #if、#ifndef、#ifdef、#else、#elif、#endif 等。
4. 删除所有注释。

2.**编译**Compilation
将预处理后的文件（.i）进行一系列词法分析、语法分析、语义分析及优化，产生相应的汇编代码文件（.asm）。

3.**汇编**Assembly
将编译后的汇编代码文件（.asm）汇编指令逐条翻译成目标机器指令，并生成可重定位目标程序的.obj文件，该文件为二进制文件，字节编码是机器指令。

4.**链接**Linking
通过链接器将多个目标文件（.obj）和 库文件 链接在一起生成一个完整的可执行程序。




## 4. 编译期-运行期

**编译期** 是指把源程序交给编译器编译、生成的过程，最终得到可执行文件。
**运行期** 是指将可执行文件交给操作系统执行、直到程序退出，把在磁盘中的程序二进制代码放到内存中执行起来，执行的目的是为了实现程序的功能。

编译期确定：

```
1  //main.cpp
2  int main() {
3
4  #ifdef __cplusplus
5  #define NN    10
6  #else
7  #define NN    20
8  #endif
9
10     int a = NN + 2;  //这个在编译期就确定了
11
12     return 0;
13 }
```

汇编代码如下：

```
    : int main() {

push    ebp
mov ebp, esp
push    ecx
mov DWORD PTR [ebp-4], -858993460       ; ccccccccH
mov ecx, OFFSET __65FCE45B_main11@cpp
call    @__CheckForDebuggerJustMyCode@4

    : #ifdef __cplusplus
    : #define NN    10
    : #else
    : #define NN    20
    : #endif
    :    int a = NN + 2;  //这个在编译期就确定了

mov DWORD PTR _a$[ebp], 12          ; 0000000cH

    :    return 0;
```

运行期确定：

```cpp
//main.cpp
int main() {
    int NN=0;
    cin>>NN;
    if (NN)
    {
        NN = 10;
    }
    else {
        NN = 20;
    }

    int a = NN + 2; //这个在运行期才能确定

    return 0;
}
```

编译期错误

```
int  len1 = 10;
const int  len2 = 10000000000;

int ARR1[len1];  //error C2131: 表达式的计算结果不是常数,编译期分配内存,
//因为必须要确定len的大小，但它是变量在编译期无法确定其具体值

int ARR2[len2];   //error C2148: 数组的总大小不得超过 0x7fffffff 字节
```

运行期错误

```
int  len3 = 10000000000;
int* pArr = new int[len3]; //程序崩溃：动态申请空间，可以为变量，运行时确定器数组大小

int arr[10] = { 0 };
arr[12] = 10;        //程序崩溃：数组越界，在编译期是检查不出来的，真正运行时可能会报错
```

**类和对象**：类是编译期的概念，包含了【访问权限】、【成员作用域】。而对象的作用域是运行期，它包括类的【实例】、【引用】和【指针】。

```
class CFather{
public:
    virtual void fun(){
        cout<<"CFather::fun"<<endl;
    }
};

class CSon:public CFather{
private:
    virtual void fun(){
        cout<<"CSon::fun"<<endl;
    }
};

CFather * pFa =new CSon;
pFa->fun();   //成功调用子类的虚函数，CSon::fun
```

编译器在检查代码时，他认为pFa->fun()调用的是父类中public属性的函数，那自然是通过编译的。但是在运行期时由于多态的作用，结果调用的是子类的fun函数，即使子类的fun函数是 private 但 由于访问修饰符是编译期的限制，所以在运行时无效，子类的fun函数自然也能调用。

## 5. 宏

宏起到**替换**作用（预处理阶段），一般写法：

```
#define N   10
```

一个标识符被宏定义后，在用到宏 N的地方替换为10，在程序编译前预处理阶段进行替换，替换后才进行编译。

宏是可以传参数的，在宏名字后面加 (PARAM)，参数的作用也是一个替换。

```
#define N(PARAM)   int a = PARAM;
```

一般情况下，宏替换当前这一行的内容，替换多行可以使用 \ 这个字符
作用：用来连接当前行和下一行。
注意：一般最后一行不加 \，\ 后面不能有任何字符，包括空格、tab、注释等。

```
#define N()\
    for(int i=0;i<10;i++){\
        cout<<i<<"   ";\
    }
```

使用宏替换需要注意，宏及参数并不会像函数参数一样自动计算，也不做表达式求解。

```
#define N   2+3
int a = N*2;   //2+3*2 = 8 不是 10

#define N(A,B)   A*B
int c = N(1+2,3);   //1+2*3 = 7  并不是9
```

可以加上 ()来解决。

```
#define N   (2+3)
int a = N*2;   //(2+3)*2 = 10

#define N(A,B)   (A)*(B)
int c = N(1+2,3);   //(1+2)*3 = 9
```

#undef 宏： 取消宏定义，限制宏的作用范围

```
#define N  10
    int a = N;
#undef N  //作用范围到此为止

    a = N;  //error C2065: “N”: 未声明的标识符
    int N = 20;  //N 并不是宏，而是定义的变量名
```

**优点**:

1. 使用宏可以替换在程序中经常使用的常量或表达式，在后期程序维护时，不用对整个程序进行修改，只需要维护、修改一份宏定义的内容即可。
2. 宏在一定程度上可以代替简单的函数，这样就省去了调用函数的各种开销，提高程序的运行效率。

**缺点**:

1. 不方便调试。
2. 没有类型安全的检查
3. 对带参的宏而言，由于是直接替换，并不会检查参数是否合法，也并不会计算求解，存在一定的安全隐患。

## 6. 宏的其他用法

# 用于将宏参数转为字符串，即加上双引号。

```
#define N(PARAM) #PARAM
N(123)  // ===  "123"
N(abc)  // ===  "abc"
```

#@ 用于将宏参数转为字符，加上单引号。

```
#define N(PARAM) #PARAM
N(1)  // ===  '1'
N(a)  // ===  'a'
```

## 用于拼接，常用语宏参数与其他内容的拼接。

```
#define N(PARAM) int a##PARAM;
N(1)  // ===  int a1;
N(2)  // ===  int a2;
```

## 7. inline 内联

内联函数C++为了提高程序的运行速度所做的一项改进，普通函数和内联函数主要区别不在于编写方式，而在于C++编译器如何将他们组合到程序中的。编译器将使用相应的函数代码替换到内联函数的调用处，所以程序无需跳转到另一个位置执行函数体代码，所以会比普通的函数稍快，代价是需要占用更多的内存，空间换时间的做法。
执行函数之前需要做一些准备工作，要将实参、局部变量、返回地址以及若干寄存器都压入栈中，然后才能执行函数体中的代码，代码执行完毕后还要将之前压入栈中的数据都出栈。这个过程中涉及到空间和时间的开销问题，如果函数体的中代码比较多，逻辑也比较复杂，那么执行函数体占用大部分时间，而函数调用、释放空间过程花费的时间占比很小可以忽略；如果函数体的中代码非常少，逻辑也非常简单，那么相比于函数体代码的执行时间 函数调用机制所花费的时间就不能忽略了。

```
int add(int a,int b){
    return a+b;
}
int c = add(1,2);
```

所以为了消除函数调用的时间开销，C++提供一种提高效率的方法 inline**函数**，上例中的add函数可以变为内联函数，如下 ，内联函数在**编译时**将函数调用处用函数体替换（类似于宏）。

```
inline int add(int a,int b){
    return a+b;
}
int c = add(1,2);  //替换后: int c = 1+2;
```

注意：

1. inline是一种空间换时间的做法，内联在一定程度上能提高函数的执行效率，这并不意味着所有函数都要成为内联函数，如果函数调用的开销时间远小于函数体代码执行的时间，那么效率提高的并不多，如果该函数被大量调用时，每一处调用都会复制一份函数体代码，那么将占用更多的内存会增加，得不偿失。所以一般函数体代码比较长，函数体内出现循环（for、while），switch等不应为内联函数。
2. 并非我们加上 inline关键字， 编译器就一定会把它当做内联函数进行替换。定义 inline 函数只是程序员对编译器提出的一个建议，而不是强制性的，编译器有自己的判断能力，它会根据具的情况决定是否把它认为是内联函数。编译器不会把递归函数视为内联函数的。
3. 类、结构中在的类内部声明并定义的函数默认为内联函数，如果类中只给出声明，在类外定义的函数，那么默认不是内联函数，除非我们手动加上 inline 关键字。

```
class CTest{
    void fun1(){}   //默认内联
    void fun2();    //声明
};
void CTest::fun2(){}  //默认不是内联函数

CTest tst;
```

```
8  tst.fun1();  //内联替换
9  tst.fun2();  //未替换
```

---

```
1  void fun1() {
2      int a = 10;
3  }
4  inline void fun2() {
5      int b = 20;
6  }
7
8  int main() {
9      fun1();
10     fun2();
11     return 0;
12 }
```

汇编文件如下：

```
    : int main() {

push    ebp
mov ebp, esp
push    ecx

    :    fun1();
                调用函数
call    ?fun1@@YAXXZ                    ; fun1

    :    int b = 20;
                        代码替换
mov DWORD PTR _b$1[ebp], 20             ; 00000014H

    :    fun2();
    :    return 0;
```