# directory



# 源程序的结构、编译、链接

## 源程序的结构

- 头文件与编译指令
- 辅助函数定义
- 主函数定义

## 编译、链接

- 源程序
- 编译器：`g++ -c test.cpp`(只编译，不链接)
    - 第一遍：执行**语法分析**和**静态类型检查**，将**源代码**解析为**语法分析树**的结构
    - 第二遍：由**代码生成器**遍历**语法分析树**，把树的每个**节点**转换为**汇编语言或机器代码**，生成**目标模块(文件)**(.o或.obj文件)。
    - 一个源程序对应一个目标文件，还需要把一系列目标文件关联起来(链接)。
- 链接器：`g++ -o test.out test.o`(链接程序，把几个东西关联在一起)
    - 把一组**目标模块**链接为**操作系统**可以执行的**可执行程序**
    - 处理目标模块中的**函数**或**变量引用**，必要时**搜索库文件**处理所有的**引用**
- 可执行程序（与平台相关的机器指令）
- 注：参考《c++编程思想》

# 多个源文件的编译与链接

- 直接编译 (g++帮我们省略了一些步骤)
    - `g++ main.cpp func.cpp -o test`
        - 输入：源程序
        - 输出：可执行文件
    - 再运行`./test 1 2`
- 分步编译(实际运行步骤)
    - `g++ -c main.cpp -o main.o`
    - `g++ -c func.cpp -o func.o`
    - `g++ main.o func.o -o test`
    - `./test 1 2`

### 链接

- 链接: 将各个目标文件中的各段代码进行**地址定位**，生成**与特定平台相关的**可执行文件
- 外部函数的**声明**（一般声明在头文件中）只是令程序顺利通过编译，此时并不需要搜索到外部函数的实现（或**定义**）。
- 在**链接**过程中，外部函数的实现（或**定义**）才会被寻找和添加进程序，一旦没有找到函数实现，就无法成功链接。
- 声明和实现不一致，则链接错误。

### 头文件

- 使用原因：有时辅助函数(如全局函数)会**在多个源文件中被使用**
- 头文件(.h)作用：
  - 避免反复编写同一段声明
  - 统一辅助函数的声明，避免错误
- #include 预编译指令
  - 将被包含的文件代码，**直接复制**到当前文件
  - 一般被用于包含头文件（实际也能包含任意代码）

## 声明与定义

### 函数

- 声明：`int ADD(int a, int b);`或`int ADD(int, int);`
  - 变量名可省略
- 定义（实现）：`int ADD(int a, int b) {return a + b;}`
- 一个函数可以多次声明，但只能有一次实现
  - 多次实现会导致链接错误

### 变量

- **声明**：告诉编译器关于变量名称、类型、大小等信息，在声明阶段不会给变量分配任何的内存。
- **定义**：在变量声明后，给它分配上内存。可以看成"定义 = 声明 + 内存分配"。
- 定义也是声明，extern声明不是定义
- 变量在使用前就要被定义或者声明。
- 在一个程序中，变量只能定义一次，却可以声明多次。
- 定义分配存储空间，而声明不会。

```
extern int x;        // 声明，不是定义
int x;               // 声明，也是定义，未初始化
extern double pi=3.141592654;  // 定义
extern double max(double d1,double d2);  // 函数声明
```

- 注意：
  - 1. 不要把变量定义放入.h文件，这样容易导致重复定义错误。 永远不要在.h文件中定义变量。定义变量和声明变量的区别在于定义会产生内存分配的操作，是汇编阶段的概念；而声明则只是告诉包含该声明的模块在连接阶段从其它模块寻找外部函数和变量

    - 2. 尽量使用static关键字把变量定义限制于该源文件作用域，除非变量被设计成全局的。
    - 3. 可以在头文件中声明一个变量，在用的时候包含这个头文件就声明了这个变量。

**extern关键字**

```
extern int x;     //声明变量
extern int arr[100]; //声明数组变量
```

- 变量的声明：extern关键字
- extern关键字也可以用于函数声明`extern int ADD(int a, int b);`
    - 但extern对于函数声明不是必须的。
- extern通常**用在全局变量在不同文件内的共享**

链接

- 为什么只在头文件(.h)进行函数声明而不实现（定义）函数体?
    - 在链接时因发现**多个相同的函数实现**而发生错误
    - 如果把定义放进头文件中，每包含一次头文件，标识符对应函数就被定义一次，重复定义在多文件的编译连接时容易出问题。
- 若头文件中定义**全局变量**且多个cpp文件包含此头文件，在链接时会因重复定义而发生错误

# 宏定义的使用

- #define是C++语言中的一个预编译指令，它用来将一个标识符定义为一个字符串，该标识符被称为宏名，被定义的字符串称为替换文本。
- 简单的宏替换
    - 在程序被编译前，先进行宏替换（即宏名用被定义的字符串替换），宏替换是简单的替换。
    - 在C++中，这种替换一般被const取代，进而能保证类型的正确性。
- 带参数的宏替换

```
#define PI 3.1415926535 // 不知道类型
const double PI = 3.1415926535
```

- 带参数的宏定义
    - 带参数的宏定义`#define sqr(x) ((x)*(x))`
    - 这种替换一般被内联函数取代，进而能保证类型的正确性

```
#include<iostream>
using namespace std;
#define M(y) y * y
inline double sqr(double x) {return x * x;}
```

```
int main() {
    const int x = 1 + 1;
    cout << x * x << endl; // 4
    int y =  M(1 + 1);
    cout << y << endl; // 3
    cout << sqr(1 + 1);
}
```

### 防止头文件被重复包含

- 方法一 #ifndef

```
#ifndef __BODYDEF_H__
#define __BODYDEF_H__
 // 头文件内容
#endif
```

```
//func.h
#ifndef FUNC_H
#define FUNC_H
int ADD(int a, int b);
#endif
```

- 方法二 #pragma once
    - 保证物理上的同一个文件不会被编译多次

```
#pragma once
// 头文件内容
```

## 1.8 编写Make工具的脚本程序

- g++ -o：指定生成文件名称
- g++ -c：要求只编译不链接

```
# 注释以#开头
ex5.out: ex5_main.o func.o
    g++ ex5_main.o func.o -o ex5.out

ex5_main.o: ex5_main.cpp func.h
    g++ -c ex5_main.cpp -o ex5_main.o

func.o: func.cpp func.h
    g++ -c func.cpp -o func.o
```

```
clean:
    del *.o *.out
  # windows 下把rm换成del
  # Linux 下写rm
```

- 可用来提高效率的几个MAKE宏
  - $@代表目标的全名（含后缀）
  - $^：所有的依赖文件
  - $*代表无后缀的目标名
  - $<代表规则中的源程序名，第一个依赖文件
  - %：通配符，[%.o: %.cpp]
  - 判断与循环
    - Makefile中可使用Bash语法，完成判断与循环

```
ifeq ($(CC),gcc)
  libs=$(libs_for_gcc)
else
  libs=$(normal_libs)
endif
```

## 1.9 使用程序主函数的命令行参数

- int argc
- char** argv

```
// ex3.cpp
#include <iostream>
#include <cstdlib>  // atoi()
int main(int argc, char** argv)
{
   if (argc != 3)  {
        std::cout << "Usage: " << argv[0]
                        << " op1 op2" << std::endl;
        return 1;
   } // 原则：总是考虑边界和异常的情况
   int a, b;
//  std::cin >> a >> b;
    a = atoi(argv[1]);
    b = atoi(argv[2]);
    std::cout << a + b << std::endl;
    return 0;
}
```

## 1.10 GDB调试工具

- g++ -g a.cpp –o a.out编译程序

- `-g` 在可执行程序中包含标准调试信息
- `gdb a.out` 调试a.out程序
- 在gdb内不产生歧义可以简写前几个字母(红色部分)
- `run` 运行程序
- `break + 行号` 设置断点
    - `break 10 if (k==2)` 可根据具体运行条件断点
    - `delete break 1` 删除1号断点
- `watch x` 当x的值发生变化时暂停
- `continue` 跳至下一个断点
- `step` 单步执行(进入)
- `next` 单步执行(不进入)
- `print x` 输出变量/表达式x
- GDB中输入 `p x=1`，程序中x的值会被手动修改为1
- `display x` 持续监测变量/表达式x
- `list`列出程序源代码
- `quit` 退出
- `回车` 重复上一条指令