全局变量

全局变量

- 定义在函数外面的变量是全局变量
- 全局变量具有全局的生存期和作用域
  - 它们与任何函数都无关
  - 在任何函数内部都可以使用它们

全局变量初始化

- 没有做初始化的全局变量会得到0值
  - 指针会得到NULL值
- 只能用编译时刻已知的值来初始化全局变量
- 它们的初始化发生在main函数之前

被隐藏的全局变量

- 如果函数内部存在与全局变量同名的变量，则全局变量被隐藏

静态本地变量

- 在本地变量定义时加上static修饰符就成为静态本地变量
- 当函数离开的时候，静态本地变量会继续存在并保持其值
- 静态本地变量的初始化只会在第一次进入这个函数时做，以后进入函数时会保持上次离开时的值

静态本地变量

- 静态本地变量实际上是特殊的全局变量
- 它们位于相同的内存区域
- 静态本地变量具有全局的生存期，函数内的局部作用域
  - static在这里的意思是局部作用域（本地可访问）

返回指针的函数

- 返回本地变量的地址是危险的
- 返回全局变量或静态本地变量的地址是安全的
- 返回在函数内malloc的内存是安全的，但是容易造成问题
- 最好的做法是返回传入的指针

tips

- 不要使用全局变量来在函数间传递参数和结果
- 尽量避免使用全局变量
  - 丰田汽车的案子
- ＊使用全局变量和静态本地变量的函数是线程不安全的

大程序

多个.c文件

- main()里的代码太长了适合分成几个函数
- 一个源代码文件太长了适合分成几个文件
- 两个独立的源代码文件不能编译形成可执行的程序

项目

- 在Dev C++中新建一个项目，然后把几个源代码文件加入进去
- 对于项目，Dev C++的编译会把一个项目中所有的源代码文件都编译后，链接起来
- 有的IDE有分开的编译和构建两个按钮，前者是对单个源代码文件编译，后者是对整个项目做链接

编译和链接的过程

- 在IDE里点击“构建”按钮，背后发生的是什么
- 从外部来看，构建的过程是由：
  - 编译预处理
  - 编译成汇编代码
  - 汇编成目标代码
  - 链接成可执行程序
- 这样四个步骤组成的

编译过程

- 在命令行编译的时候加上选项—save-temps
  - gcc a.c --save-temps
- 就能保留编译链接过程中所产生的中间过程文件
  - 编译预处理 --> .i
  - 编译成汇编代码 --> .s
  - 汇编成目标代码 --> .o
  - 链接成可执行程序 --> .exe（或a.out）

- 这些文件都可以再用gcc单独操作，产生下一步结果（但是选项复杂）
- gcc -c的选项可以只编译，产生目标文件，但是不链接

编译单元

- 一个项目中的多个.c文件之间一定存在相互调用的关系
  - 如a.c里的f()调用了b.c里的g()
- 一个.c文件是一个编译单元
- 编译器每次编译只处理一个编译单元
  - 编译器不会打开其他文件来参考、检查
  - 即使这些文件在一个项目中，编译器也不看
- 因此需要通过头文件来告诉某个编译单元别的单元里有什么

头文件

函数原型

- 如果不给出函数原型，编译器会猜测你所调用的函数的所有参数都是int，返回类型也是int
- 编译器在编译的时候只看当前的一个编译单元，它不会去看同一个项目中的其他编译单元以找出那个函数的原型
- 如果你的函数并非如此，程序链接的时候不会出错
- 但是执行的时候就不对了
- 所以需要在调用函数的地方给出函数的原型，以告诉编译器那个函数究竟长什么样

头文件

- 把函数原型放到一个头文件（以.h结尾）中，在需要调用这个函数的源代码文件（.c文件）中#include这个头文件，就能让编译器在编译的时候知道函数的原型

#include

- #include是一个编译预处理指令，和宏一样，在编译之前就处理了
- 它把那个文件的全部文本内容原封不动地插入到它所在的地方
  - 所以也不是一定要在.c文件的最前面#include

“”还是<>

- #include有两种形式来指出要插入的文件
  - “”要求编译器首先在当前目录（.c文件所在的目录）寻找这个文件，如果没有，到编译器指定的目录去找
  - <>让编译器只在指定的目录去找
- 编译器自己知道自己的标准库的头文件在哪里
- 环境变量和编译器命令行参数也可以指定寻找头文件的目录

#include的误区

- #include不是用来引入库的
- stdio.h里只有printf的原型，printf的代码在另外的地方，某个.lib(Windows)或.a(Unix)中
- 现在的C语言编译器默认会引入所有的标准库
- #include <stdio.h>只是为了让编译器知道printf函数的原型，保证你调用时给出的参数值是正确的类型

头文件

- 在使用和定义这个函数的地方都应该#include这个头文件
- 一般的做法就是任何.c都有对应的同名的.h，把所有对外公开的函数的原型和全局变量的声明都放进去

不对外公开的函数

- 在函数前面加上static就使得它成为只能在所在的编译单元中被使用的函数
- 在全局变量前面加上static就使得它成为只能在所在的编译单元中被使用的全局变量

声明

变量的声明

- int i;是变量的定义
- extern int i;是变量的声明

声明和定义

- 声明是不产生代码的东西
  - 函数原型
  - 变量声明
  - 结构声明
  - 宏声明
  - 枚举声明
  - 类型声明
  - inline函数
- 定义是产生代码的东西

头文件

- 只有声明可以被放在头文件中
  - 是规则不是法律
- 否则会造成一个项目中多个编译单元里有重名的实体
  - *某些编译器允许几个编译单元中存在同名的函数，或者用weak修饰符来强调这种存在

重复声明

- 同一个编译单元里，同名的结构不能被重复声明
- 如果你的头文件里有结构的声明，很难这个头文件不会在一个编译单元里被#include多次
- 所以需要“标准头文件结构”

标准头文件结构

```
#ifndef __LIST_HEAD__
#define __LIST_HEAD__

#include "node.h"

typedef struct _list {
    Node* head;
    Node* tail;
} List;

#endif
```

- 运用条件编译和宏，保证这个头文件在一个编译单元中只会被#include一次
- #pragma once也能起到相同的作用，但是不是所有的编译器都支持

在头文件中引用其他头文件

- 如果在node.h里声明了Node，那么在list.h里，因为用到了Node，一般就要#include "node.h"
- 这样会造成复杂的依赖关系，即list.h依赖于node.h，一旦node.h修改了，所有#include了list.h的编译单元都需要重新编译
- 一般尽量避免在头文件中引用其他头文件，除非必须

```
#ifndef _LIST_H_
#define _LIST_H_

#include "node.h"

typedef struct {
    Node *head;
    Node *tail;
} List;

#endif
```

前向声明

```
#ifndef __LIST_HEAD__
#define __LIST_HEAD__

struct Node;

typedef struct _list {
    struct Node* head;
    struct Node* tail;
} List;

#endif
```

- 因为在这个地方不需要具体知道Node是怎样的，所以可以用struct Node来告诉编译器Node是一个结构

宏

编译预处理指令

- #开头的是编译预处理指令
- 它们不是C语言的成分，但是C语言程序离不开它们
- #define用来定义一个宏

#define

- #define <名字> <值>
- 注意没有结尾的分号，因为不是C的语句
- 名字必须是一个单词，值可以是各种东西
- 在C语言的编译器开始编译之前，编译预处理程序（cpp）会把程序中的名字换成值
  - 完全的文本替换
- gcc —save-temps

宏

- 如果一个宏的值中有其他的宏的名字，也是会被替换的
- 如果一个宏的值超过一行，最后一行之前的行末需要加\
- 宏的值后面出现的注释不会被当作宏的值的一部分

没有值的宏

- #define _DEBUG
- 这类宏是用于条件编译的，后面有其他的编译预处理指令来检查这个宏是否已经被定义过了

预定义的宏

- __LINE__
- __FILE__
- __DATE__
- __TIME__
- __STDC__

带参数的宏

像函数的宏

- #define cube(x) ((x)*(x)*(x))
- 宏可以带参数

错误定义的宏

- #define RADTODEG(x) (x * 57.29578)
- #define RADTODEG(x) (x) * 57.29578

带参数的宏的原则

- 一切都要括号
  - 整个值要括号
  - 参数出现的每个地方都要括号
- #define RADTODEG(x) ((x) * 57.29578)

带参数的宏

- 可以带多个参数
  - #define MIN(a,b) ((a)>(b)?(b):(a))
- 也可以组合（嵌套）使用其他宏

分号?

```
#define PRETTY_PRINT(msg) printf(msg);


if (n < 10)
    PRETTY_PRINT("n is less than 10");
else
    PRETTY_PRINT("n is at least 10");
```

带参数的宏

- 在大型程序的代码中使用非常普遍
- 可以非常复杂，如“产生”函数
  - 在#和##这两个运算符的帮助下
- 存在中西方文化差异
- 部分宏会被inline函数替代

operator

- #define PRINT_INT(n) printf(#n “=%d”, n)

##operator

```
#define MYCASE(item,id) \
case id: \
  item##_##id = id;\
break


switch(x) {
    MYCASE(widget,23);
}
```

http://wengkai.github.io/cmacro/cmacro.html

其他编译预处理指令

- 条件编译
- error
- …