

# 程序结构

2020年7月24日 10:41

## 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

```
3 int f(void);
4
5 int gAll = 12;
6
7 int main(int argc, char const *argv[])
8 {
9     printf("in %s gAll=%d\n", __func__, gAll);
10    f();
11    printf("agn in %s gAll=%d\n", __func__, gAll);
12    return 0;
13 }
14
15 int f(void)
16 {
17     printf("in %s gAll=%d\n", __func__, gAll);
18     gAll += 2;
19     printf("agn in %s gAll=%d\n", __func__, gAll);
20     return gAll;
21 }
```

```
in main gAll=12
in f gAll=12
agn in f gAll=14
agn in main gAll=14
[Finished in 0.8s]
```

## 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

```
3 int f(void);
4
5 int gAll=f();
6
7 int main(int argc, char const *argv[])
8 {
9     printf("in %s gAll=%d\n", __func__, gAll);
10    f();
11    printf("agn in %s gAll=%d\n", __func__, gAll);
12    return 0;
13 }
14
15 int f(void)
16 {
17     printf("in %s gAll=%d\n", __func__, gAll);
18     gAll += 2;
19     printf("agn in %s gAll=%d\n", __func__, gAll);
20     return gAll;
21 }
```

```
/Users/wengkai/cc/12.1.c:5:10: error: initializer element
int gAll=f();
      ~~~
1 error generated.
[Finished in 0.1s with exit code 1]
```

```
3 int f(void);
4
5 int gAll=12;
6 int g2 = gAll;
7
8 int main(int argc, char const *argv[])
9 {
10    printf("in %s gAll=%d\n", __func__, gAll);
11    f();
12    printf("agn in %s gAll=%d\n", __func__, gAll);
13    return 0;
14 }
15
16 int f(void)
17 {
18    printf("in %s gAll=%d\n", __func__, gAll);
19    gAll += 2;
20    printf("agn in %s gAll=%d\n", __func__, gAll);
21    return gAll;
22 }
```

编译器不聪明，编译器看到把一个变量给g2

```
/Users/wengkai/cc/12.1.c:6:10: error: initializer element
int g2 = gAll;
      ~~~
1 error generated.
[Finished in 0.1s with exit code 1]
```

```
3 int f(void);
4
5 const int gAll=12;
6 int g2 = gAll;
7
8 int main(int argc, char const *argv[])
9 {
10    printf("in %s gAll=%d\n", __func__, gAll);
11    f();
12    printf("agn in %s gAll=%d\n", __func__, gAll);
13    return 0;
14 }
15
16 int f(void)
17 {
18    printf("in %s gAll=%d\n", __func__, gAll);
19    // gAll += 2;
20    printf("agn in %s gAll=%d\n", __func__, gAll);
21    return gAll;
22 }
```

虽然可以，但是不推荐这样初始化

[Finished in 0.1s]

```
3 int f(void);
4
5 int g2 = gAll;
6
7 const int gAll=12;
8
9 int main(int argc, char const *argv[])
10 {
11    printf("in %s gAll=%d\n", __func__,
12    f();
13    printf("agn in %s gAll=%d\n", __func__,
```

## 被隐藏的全局变量

## 被隐藏的全局变量

```
9 int main(int argc, char const *argv[])
10 {
11     printf("in %s gAll=%d\n", __func__, gAll);
12     f();
13     printf("agn in %s gAll=%d\n", __func__, gAll);
14     return 0;
15 }
16
17 int f(void)
18 {
19     printf("in %s gAll=%d\n", __func__, gAll);
20     // gAll += 2;
21     printf("agn in %s gAll=%d\n", __func__, gAll);
22     return gAll;
23 }
```

/Users/wengkai/cc/12.1.c:5:10: error: use of undeclared identifier 'gAll';  
1 error generated.  
[Finished in 0.1s with exit code 1]

- 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

```
3 int f(void);
4
5 int gAll=12;
6
7 int main(int argc, char const *argv[])
8 {
9     f();
10    f();
11    f();
12    return 0;
13 }
14
15 int f(void)
16 {
17     int all = 1;
18     printf("in %s all=%d\n", __func__, all);
19     all += 2;
20     printf("agn in %s all=%d\n", __func__, all);
21     return all;
22 }
```

```
in f gAll=12
agn in f gAll=12
in f gAll=12
agn in f gAll=12
in f gAll=12
agn in f gAll=12
[Finished in 0.1s]
```

```
3 int f(void);
4
5 int gAll=12;
6
7 int main(int argc, char const *argv[])
8 {
9     f();
10    f();
11    f();
12    return 0;
13 }
14
15 int f(void)
16 {
17     static int all = 1;
18     printf("in %s all=%d\n", __func__, all);
19     all += 2;
20     printf("agn in %s all=%d\n", __func__, all);
21     return all;
22 }
```

```
in f all=1
agn in f all=3
in f all=3
agn in f all=5
in f all=5
agn in f all=7
[Finished in 0.2s]
```

## 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）

```
3 int f(void);
4
5 int gAll=12;
6
7 int main(int argc, char const *argv[])
8 {
9     f();
10    return 0;
11 }
12
13 int f(void)
14 {
15     int k = 0;
16     static int all = 1;
17     printf("&gAll=%p\n", &gAll);
18     printf("&all=%p\n", &all);
19     printf("&k=%p\n", &k);
20     printf("in %s all=%d\n", __func__, all);
21     all += 2;
22     printf("agn in %s all=%d\n", __func__, all);
23     return all;
24 }
```

&gAll=0x3800c  
&all=0x38010  
&k=0xbffc9d4c  
in f all=1  
agn in f all=3

## \*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

```

7 int main(int argc, char const *argv[])
8 {
9     int *p = f();
10    printf("p=%d\n", *p);
11    g();
12    printf("p=%d\n", *p);
13
14    return 0;
15 }
16
17 int* f(void)
18 {
19     int i=12;
20     return &i;
21 }
22
23 void g(void)
24 {
25     int k = 24;
26     printf("k=%d\n", k);
27 }

```

1 warning generated.  
 \*p=12  
 k=24  
 \*p=24  
 [Finished in 0.2s]

tips

还可以得到12?  
 不存在意为不再受控, 不是不在那里,  
 无人保证12所在的地方还可以继续保持12  
 变成24? ?  
 i, k 地址是一样的, 有风险!!  
 函数结束后本地变量的地址分配给别的变量了!!  
 房东把房子租给别人了, 你还有房子的钥匙  
 当f()运行结束后, 变量i的内存被自动返还。  
 因为i与k都是存储在 栈中, 所以, i返还后的内存被k利用了。

- 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

## 编译预处理指令

```

// const double PI = 3.14159;
#define PI 3.14159

```

一个宏pi, pi是宏的名字, 3.14159是宏的值  
 编译预处理, 把所有的pi变成3.14159 (会生成临时文件来处理)  
 就是原始的文本替换 可以用注释  
 空格什么什么标点符号会被当成宏的值

- #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

```

1 #include <stdio.h>
2
3 #define PI 3.14159
4 #define FORMAT "%f\n"
5 #define PI2 2*PI // pi * 2
6 #define PRT printf("%f ", PI); \
7     printf("%f\n", PI2)
8
9 int main(int argc, char const *argv[])
10 {
11     // printf(FORMAT, PI2*3.0);
12     PRT;
13     return 0;
14 }
15

```

3.141590 6.283180  
 [Finished in 0.1s]

## 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

```
1 #include <stdio.h>
2
3 int main(int argc, char const *argv[])
4 {
5     printf("%s:%d\n", __FILE__, __LINE__);
6     printf("%s,%s\n", __DATE__, __TIME__);
7     return 0;
8 }
```

- \_\_LINE\_\_
- \_\_FILE\_\_
- \_\_DATE\_\_
- \_\_TIME\_\_
- \_\_STDC\_\_

```
/Users/wengkai/cc/12.4.c:5
Aug 3 2014,14:08:45
[Finished in 0.1s]
```

## 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

```
1 #include <stdio.h>
2
3 #define RADTODEG1(x) (x * 57.29578)
4 #define RADTODEG2(x) (x) * 57.29578
5
6 int main(int argc, char const *argv[])
7 {
8     printf("%f\n", RADTODEG1(5+2));
9     printf("%f\n", 180/RADTODEG2(1));
10     return 0;
11 }
```

```
119.591560
10313.240400
[Finished in 0.1s]
```

```
int main(int argc, char const *argv[])
{
    printf("%f\n", (5+2 * 57.29578));
    printf("%f\n", 180/(1) * 57.29578);
    return 0;
}
```

M00C:cc\ \$tail 12.6.c

```
#define RADTODEG1(x) (x * 57.29578)
#define RADTODEG2(x) (x) * 57.29578
```

```
int main(int argc, char const *argv[])
{
    printf("%f\n", RADTODEG1(5+2));
    printf("%f\n", 180/RADTODEG2(1));
    return 0;
}
```

)M00C:cc\ \$

## 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

## 多个.c文件

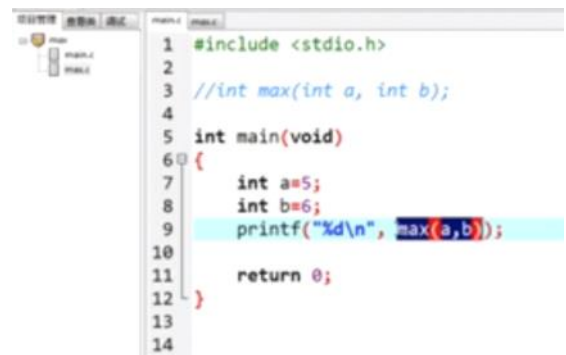
- main()里的代码太长了适合分成几个函数
- 一个源代码文件太长了适合分成几个文件
- 两个独立的源代码文件不能编译形成可执行的程序

## 编译单元

- 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

## 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



```
1 #include <stdio.h>
2
3 //int max(int a, int b);
4
5 int main(void)
6 {
7     int a=5;
8     int b=6;
9     printf("%d\n", max(a,b));
10
11     return 0;
12 }
13
14
```

运行正确??

编译器猜max是什么东西

如果把max里面类型换了?? 编译通过??

有问题，执行会出错

## 函数原型

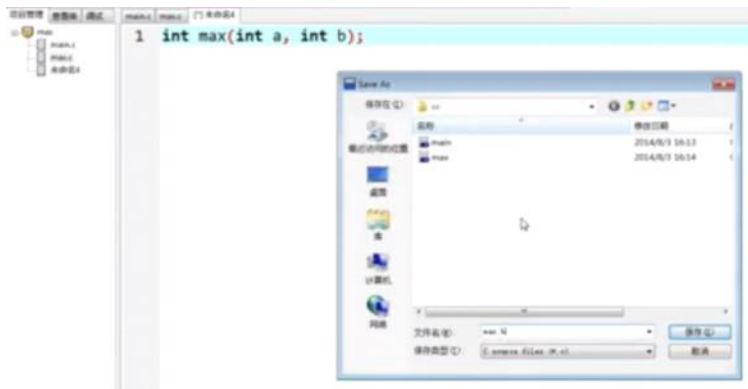
- 如果不给出函数原型，编译器会猜测你所调用的函数的所有参数都是int，返回类型也是int
- 编译器在编译的时候只看当前的一个编译单元，它不会去看同一个项目中的其他编译单元以找出那个函数的原型
- 如果你的函数并非如此，程序链接的时候不会出错
- 但是执行的时候就不对了
- 所以需要在调用函数的地方给出函数的原型，以告诉编译器那个函数究竟长什么样

## 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

max.h





这样编译器会发现出错（类型不一致）

```
#include "max.h"

double max(double a, double b)
{
    return a>b?a:b;
}
```

```
1 #include <stdio.h>
2 #include "max.h"
3
4 int main(void)
5 {
6     int a=5;
7     int b=6;
8     printf("%d\n", max(a,b));
9
10    return 0;
11 }
```

## #include

头文件：一个桥梁，一个合同

- #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
- 所以也不是一一定要在.c文件的最前面#include

included只是把另一个文件中的代码原样插到文件中

```
extern int __vsprintf_chk (char * re:
    const char * restrict, va_list
# 491 "/Applications/Xcode.app/Conten
usr/include/stdio.h" 2 3 4
# 1 "main.c" 2

# 1 "./max.h" 1
double max(double a, double b);
# 2 "main.c" 2
```

```
int main(void)
{
    int a=5;
    int b=6;
    printf("%f\n", max(a,b));

    return 0;
}
```

## “”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

```
1 double max(double a, double b);
2 extern int gAll;
```

```
1 #include "max.h"
2
3 int gAll = 12;
4
5 double max(double a, double b)
6 {
7     return a>b?a:b;
8 }
```

```
1 #include <stdio.h>
2 #include "max.h"
3
4 int main(void)
5 {
6     int a=5;
7     int b=6;
8     printf("%f\n", max(a,gAll));
9
10    return 0;
11 }
```

## #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

- #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

抄下来，不产生代码，就是编译器知道了记住了有这个东西

## 变量的声明

- int i;是变量的定义
- extern int i;是变量的声明

## 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

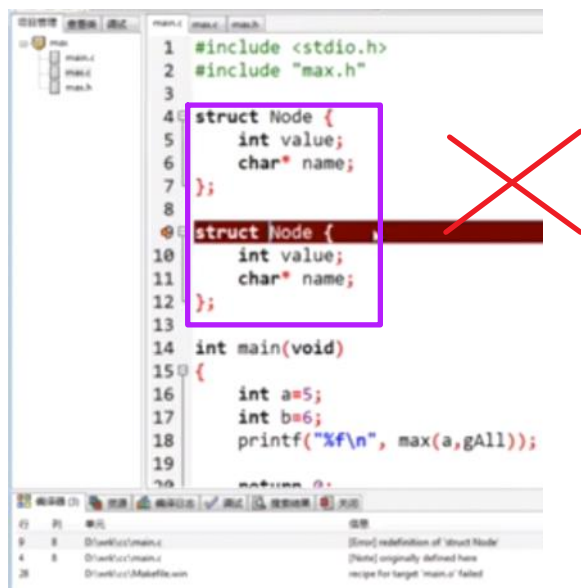
## 声明和定义

- 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

## 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

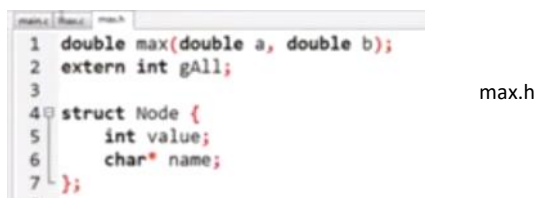
## 头文件



- 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”



解决方法max.h

```

1 double max(double a, double b);
2 extern int gAll;
3
4 struct Node {
5     int value;
6     char* name;
7 };
8

```

max.h

```

1 #include "max.h"
2

```

min.h

```

1 #include <stdio.h>
2 #include "max.h"
3 double max(double a, double b);
4 extern int gAll;
5
6 struct Node {
7     int value;
8     char* name;
9 };
10 #include "min.h"
11 #include "max.h"
12 double max(double a, double b);
13 extern int gAll;
14
15 struct Node {
16     int value;
17     char* name;
18 };
19

```



- 所以需要“标准头文件结构”

解决方法max.h

```

1 #ifndef _MAX_H_
2 #define _MAX_H_
3
4 double max(double a, double b);
5 extern int gAll;
6
7 struct Node {
8     int value;
9     char* name;
10 };
11
12 #endif
13

```

```

1 #include <stdio.h>
2 #include "max.h"
3 #ifndef _MAX_H_
4 #define _MAX_H_
5
6 double max(double a, double b);
7 extern int gAll;
8
9 struct Node {
10     int value;
11     char* name;
12 };
13
14 #endif
15
16 #include "min.h"
17 #ifndef _MAX_H_
18 #define _MAX_H_
19
20 double max(double a, double b);
21 extern int gAll;
22
23 struct Node {
24     int value;
25     char* name;
26 };
27
28 #endif

```

## 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

## \* 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

小明忘了在程序开头写#include <stdio.h>，但是main()中的printf("hello\n");还是通过了编译而且运行正确。说明为什么会这样，并举出例子说明什么情况下不#include相应的头文件会通过编译但是不能正确运行

因为头文件只是包含原型，真正的可执行文件并不在当前目录下。

程序没找到文件会有个猜测原型。当猜测的原型在实际运行中发现冲突时就会出错