模块化

- 工程代码
- 链表操作
- 模块封装
- 数据与操作的分离
- 代码注入

原理代码vs工程代码

- 从去抖动程序看代码的工程化
  http://wengkai.github.io/arduino/debounce.html
- 网上搜到的、教科书上的代码往往只是原理性的
- 仅能说明工作原理，并不考虑在工程中的实际情况

Debounce.c

Arduino的去抖动代码

https://www.arduino.cc/en/Tutorial/Debounce

- 很多全局变量

```
// Variables will change:
int ledState = HIGH;          // the current state of the
int buttonState;              // the current reading from
int lastButtonState = LOW;    // the previous reading from

// the following variables are unsigned longs because the
// milliseconds, will quickly become a bigger number than
unsigned long lastDebounceTime = 0;  // the last time the
unsigned long debounceDelay = 50;    // the debounce time

void loop() {
        ...
```

- 在`loop()`中会使用这些全局变量

提取工作函数

- Arduino的`loop()`函数是会不断循环执行的函数，相当于C的`main()`函数
- 首先将其中判断按钮按下的代码抽离出来，形成一个函数`isKeyPressed()`
- 于是就会发现上面的这些全局变量其实只在这个函数中使用
- 但是它们是持久存储的：需要在函数的每次执行之间保持值

```
boolean isKeyPressed()
{
  static int btnState = HIGH;
  static unsigned long lastDebounceTime = 0;  // the last
  static boolean isValid = false;
  static const unsigned long debounceDelay = 50;
  int reading = digitalRead(btnMode);
  int ret = false;
```

Debounce-1.c

处理多个按钮

- 这样的函数只能处理一个按钮，如果需要处理多个按钮，就需要写多个这样的函数，这显然是不现实的
- 把这些静态本地变量提取出来，形成一个结构，作为函数的参数，每次调用的时候传进去

```
typedef {
    int btnPin;
    int btnState;
    int pressedValue;
    unsigned long lastDebounceTime;
    boolean isValid;
    unsigned long debounceDelay;
} Button;
boolean isKeyPressed(Button *btn);
```

Debounce-2.c

初始化 `Button` 结构

- 为了使用这个 `Button` 结构，就必须为它配一个初始化函数

```
Button* init_button(Button* btn, int pin, boolean pullup)
{
  btn->btnPin = pin;
  btn->btnState = pullup?HIGH:LOW;
  btn->pressedValue = pullup?LOW:HIGH;
  pinMode(pin, pullup?INPUT_PULLUP:INPUT);
  btn->lastDebounceTime = 0;
  btn->isValid = false;
  btn->debounceDelay = 50;
}
```

链表

- 实现一个程序，读入无确定数量的非负整数，读到`-1`表示结束
- 将这些数中的偶数除以2
- 按照输入的顺序输出计算后的结果
- 因为会读到的数的数量无法预知，所以是典型的采用链表实现的程序

w5-1.c

七宗罪之一：命名混乱

- 这个结构表达的是链表中的每一个结点，却起名为List
- 起名字很重要！
- 名不符实会让程序员越做越心慌，也很容易让别人在阅读代码的时候难以理解，或容易误解

```
typedef struct _Node {
        int value;
        struct _Node *next;
} Node;
```

w5-2.c

PS：为何不声明为指针类型

```
typedef struct _Node {
        int value;
        struct _Node *next;
} *Node;
```

- 有的书喜欢这样做：因为代码中出现的都是`Node`的指针，于是他们干脆把`Node`定义为指针类型，这样`Node p;`实际上就是指针，但是又不会出现`*`
- 这是非常害人的！
- 因为别人在阅读你的代码的时候，看到`Node p`，没有看到`*`，就会误以为`p`是数据本身而不会想到是指针

w5-3.c

七宗罪之二：30行的连续代码

- 在main()中有30行代码，没有任何间隔
- 其实这30行代码正好是4段：
  - 读入数据，构造链表
  - 遍历链表，将偶数/2
  - 遍历链表输出
  - 删除链表中的每个结点
- 如果在其中加入恰当的注释，可以将代码分割成四段，有助于阅读理解

  w5-4.c

七宗罪之三：30行的大函数

- 虽然一个函数可以有几行没有定数，但是一般有这样两个原则：
  - 尽量在一屏之内（所以有人要把屏幕竖起来，还用很小的字体 ）
  - 一个函数内的代码无法再拆分成有意义的片段
- 显然这个30行的main()是由四个片段组成的，正好可以拆成4段
- 通常不应该在main()里放实际的代码，main()就是用来启动和表达整体逻辑的，里面应该只有几行简单的函数调用

w5-5.c

七宗罪之四：全局变量

- 在丰田汽车代码的故事里我们已经知道全局变量是毒药
- 在这个链表代码里，全局变量阻止了两件好事：
  - 如果我们需要第二个链表，该怎么办？
  - 这四个函数都和那两个全局变量紧密捆绑在一起，如何能把它们重用在其他程序中？
- 如果不用全局变量，就意味着：
  - 这四个函数都需要有参数来表达那个链表
  - 表示链表的两个全局变量`head`和`tail`要有个别的地方放
- 我们先考虑一个更简单一些的例子

链表的add_head()函数

```
Node *p = (Node*)malloc(Node);
p->value = v;
p->next = NULL;
if ( head ) {
        head->next = p;
} else {
        head = p;
}
```

- 把这段代码做成函数的困难，是如何定义参数和返回

链表的add_head()函数

- `void add_head(Node *head, int v)`
  - `head`是会在函数中被改变的！这个绝对不行！
- `void add_head(Node **pHead, int v)`
  - 传入指向`head`的指针，可以用，不过不够优雅
- `Node* add_head(Node *head, int v)`
  - 返回`head`，可能是新的，所以调用的地方必须记得要`head = add_head(head, v)`，可是万一忘了呢？依赖程序员的自觉和高素质是很危险的！

List结构

- 更好的方案是定义一个结构List：

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

- 于是`add_head()`就是：

```
void add_head(List* list, int v)
{
    Node *p = (Node*)malloc(Node);
    p->value = v;
    p->next = NULL;
    if ( list->head ) {
        list->head->next = p;
    } else {
        list->head = p;
    }
}
```

使用List结构

- `List` 的变量本身可以是本地变量，也可以是动态分配的内存

```
int main()
{
    List list = {NULL};
    int x;
    scanf("%d", &x);
    while ( x>-1 ) {
        add_head(&list, x);
        scanf("%d", &x);
    }
    ...
}
```

- 在 `add_head` 里申请了动态内存，却没有直接在那里释放，这似乎违背了之前讲过的原则：不在函数内申请动态内存后返回出去
- 这是因为链表函数库作为一个整体，会有释放动态申请的内存的地方

双头链表

```c
typedef struct _Node {
        int value;
        struct _Node *next;
} Node;

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

void add_tail(List* list, int value);
void read(List* list);
void div2(const List* list);
void prt(const List* list);
void clean(List* list);
```

w5-6.c

read函数

```
void read(List* list)
{
        int x;
        scanf("%d", &x);
        while ( x!= -1 ) {
                add_tail(list, x);
                scanf("%d", &x);
        }
}
```

- `read()`函数是和这个程序的业务逻辑密切相关的，它能被重用的可能性不大
- 把链表中添加结点的部分抽离出一个`add_tail()`函数，使得这部分代码将来可以被重用

add_tail函数

```
void add_tail(List* list, int value)
{
        Node *p = (Node*)malloc(sizeof(Node));
        p->value = value;
        p->next = NULL;
        if ( list->tail ) {
                list->tail->next = p;
        } else {
                list->head = p;
        }
        list->tail = p;
}
```

div2和prt函数

```
void div2(const List* list)
{
        for ( Node *p = list->head; p; p=p->next ) {
                if ( p->value %2 == 0 ) {
                        p->value /= 2;
                }
        }
}

void prt(const List* list)
{
        for ( Node *p = list->head; p; p=p->next ) {
                printf("%d ", p->value);
        }
        printf("\n");
}
```

- 这两个函数都不会修改`List`结构，所以加上了`const`修饰
  - `div2()`会修改链表，但是不会修改`head`或`tail`

clean函数

```
void clean(List* list)
{
        for ( Node *p = list->head; p; ) {
                Node *q = p->next;
                free(p);
                p = q;
        }
        list->head = list->tail = NULL;
}
```

- 删除了全部结点之后，把`List`结构里的`head`和`tail`都复位是个好的习惯
- 万一后面还有代码会接触到这个结构变量呢
- 这个`List`变量本身不需要被释放，它是一个本地变量

代码重用

- 这里的`add_tail`、`prt`、`clean`函数和具体的业务无关，和具体的链表无关，它们可以不经任何修改用于其他程序
- 实现代码重用首先重在思想，在写代码的时候要想着将来，而不是局限于满足眼前的需要
- 以下三条基本原则可以遵循：
  - 分离数据和操作，让函数通过参数获得要处理的数据，绝不直接访问全局变量
  - 分离业务逻辑和通用操作，就像从`read`中分离出`add_tail`，通用操作可以打包成库
  - 隐藏细节。一开始定义的`List`结构里只有`head`，后来加上了`tail`，但是`add_tail`这些函数的原型和`List`里有什么无关

隐藏细节-->接口

```
void add_tail(List* list, int value);
void prt(const List* list);
void clean(List* list);
```

- 这些函数的原型，和链表的具体实现方式无关，链表可能是：
  - 单向、双向；有无尾指针；有无哨兵结点
- 这些选项的组合，但是无论是哪种链表，都可以用上述的三个函数的原型来操作，不同的只是`List`结构里的成员变量，以及函数内部的具体代码
- 保持这样的函数原型不变，更换不同的链表类型时，使用这三个函数的代码就不需要修改（*将w5-6.c的链表改成没有尾指针，需要改哪些部分？*）
- 这三个函数的原型就是接口。通过接口隔离了代码的具体实现，实现了松耦合

div2和prt函数

```
void div2(const List* list)
{
        for ( Node *p = list->head; p; p=p->next ) {
                if ( p->value %2 == 0 ) {
                        p->value /= 2;
                }
        }
}

void prt(const List* list)
{
        for ( Node *p = list->head; p; p=p->next ) {
                printf("%d ", p->value);
        }
        printf("\n");
}
```

- 这两个函数长得很像，都是遍历整个链表，对每个结点做点什么

另一种代码重用

- 将重复使用的代码提取成函数，是代码重用的一种方式
- 这样的函数可以做成库，供今后的程序使用
- 但是遍历链表是特殊的，因为遍历的代码是会重复用到的，而不同的是遍历到每一个结点时做的动作
- 这样就无法采用函数来重用
  - `div2()`和`prt()`里相同的循环无法被重用
- 有两个解决方案：
  - 枚举器
  - 代码注入

枚举器

- 当采用不同的方式的链表时，如有无哨兵结点、块链表还是单数据链表，遍历的代码是不相同的
- 如果想要让应用代码能以相同的函数原型操作链表，可以采用枚举器的方式
- C++的STL就是用的枚举器方式

```
for ( Iterator<int> p = v.begin(); p!=v.end(); p++ ) {
        *p...
}
```

- 这种方式能解决对不同形式的链表采用相同的代码接口的要求
- 但是不能很好地解决遍历代码重用的问题，而且C语言实现不易

代码注入

- 既然遍历的代码是一样的，不同的只是循环内针对每个结点要做的动作
- 那么能否把这个动作在调用遍历函数的时候送进去呢?

```
static void iterate(const List *list, void (*f)(Node*))
{
        for ( Node *p = list->head; p; p=p->next ) {
                f(p);
        }
}
```

- 这个函数的第二个参数是函数指针，指向一个参数为`Node*`，没有返回值的函数
- 在遍历到每一个结点时，调用这个函数来处理每个结点

代码注入的 `div2()`

```
static void div2it(Node *p)
{
        if ( p->value %2 == 0 ) {
                p->value /= 2;
        }
}

void div2(const List* list)
{
        iterate(list, div2it);
}
```

- `div2it()` 就是满足 `iterate()` 要求的函数，在 `div2()` 里将这个函数传递进 `iterate()` 里去做具体的计算
- 虽然改造了遍历的方式，但是维持了 `div2()` 函数原型没有变，因此这个修改没有扩散到 `main()` 里
- `prt()` 也以相同的方式做了修改

w5-7.c

作业

- 实现一个基于链表的容器库（整数队列）
- 具有如下功能：
  - 添加一个整数在队尾
  - 从队首取走一个整数
  - 偷看队首的整数值
  - 遍历整个队列
  - 返回队列的长度
  - 获得队列中第n个位置上的整数值
  - 删除第n个位置上的整数值
  - 查找整数x是否在队列中，返回位置或-1（找不到时）
  - 删除整个队列