从零开始构建一个贪吃蛇游戏（C语言）

by kai_Ker

Chapter.01 优雅地使用 WASD 进行控制

既然是一个游戏，我们自然需要一种交互方式，比如通常都会使用的 “WASD” 键位。就我们目前所学，使用 `getchar()` 似乎是一种方法：

```c
// ./src/01-01-getchar.c
#include <stdio.h>
int main() {
    char ch;
```

```
    while (EOF != (ch = getchar())) {
        switch (ch) {
        case 'w':
            printf("pressed: w\n"); break;
        case 'a':
            printf("pressed: a\n"); break;
        case 's':
            printf("pressed: s\n"); break;
        case 'd':
            printf("pressed: d\n"); break;
        }
    }
    return 0;
}
```

但是很明显地，每一次输入都需要按一次**回车**，这怎能算一个游戏呢？于是我们需要介绍另外几个与输入输出相关的函数。

新的控制台输入函数

首先是需要使用的头文件：`<conio.h>`，'con' 代指 "console"，“控制台”之意，'io' 代指 "input-output"。

> 警告：头文件 `<conio.h>` 并不是C的标准头文件，下面介绍的几个函数也不是C的标准库函数，因此它们在 Mac 或者 Linux 系统下是无法直接使用的。如果你是 Mac 用户，可以参考下面这两篇文章；如果你是 Linux 用户，相信你一定可以找到解决方法的。
> 如何在mac/linux实现使用getch()/getche()函数
> mac下的一个类似“_kbhit()”实现

函数一：`_kbhit()`

这个函数的函数名是 "keyboard hit" 的缩写，注意开头有一个**下划线**。当有按键被按下时，函数返回**非0值**，否则返回 **0**。注意，这是一个**非阻塞**函数，它的调用并不会使程序停止，因此我们需要将其放置在一个**循环**中，让其反复调用。

```
// ./src/01-02-kbhit.c
#include <conio.h>
#include <stdio.h>
int main() {
    while (1) {
        if (_kbhit()) {
            printf("some key was hit\n");
            break;
        }
    }
```

```
    return 0;
}
```

当按下键盘上某个键时，程序会打印这句话，并且跳出循环。

函数二：`_getch()`

这个函数同样需要注意开头的**下划线**。它可以直接接收并返回按下的字符，并且**不在**屏幕上显示，同时也**不需要回车**。但它是一个**阻塞**型函数，会**让程序停住**，等待输入。

```c
// ./src/01-03-getch.c
#include <conio.h>
#include <stdio.h>
int main() {
    printf("press one key: ");
    char ch = _getch();
    printf("\nyou pressed: %c", ch);
    return 0;
}
```

按下字母 'e' 时，屏幕上只会有如下内容：

```
press one key:
you pressed: e
```

检测输入而不是等待输入 ⭐

但是，我们希望的是，蛇能够一直向前走（也即**不中断程序的运行**），同时我们能够控制蛇的**方向**。换言之，我们要将蛇的运行放在一个**循环**中，并且每次**去检测**我们有没有按下某个键。于是很自然地，我们可以将这两个函数结合使用，且看：

```c
// ./src/01-04-control.c
#include <conio.h>
#include <stdio.h>
int main() {
    while (1) {
        if (_kbhit()) {
            // 当某个按键被按下，进入这里，利用 _getch() 读取按键
            switch (_getch()) {
            case 'w':
                printf("pressed: w\n"); break;
            case 'a':
                printf("pressed: a\n"); break;
            case 's':
```

```
                printf("pressed: s\n"); break;
            case 'd':
                printf("pressed: d\n"); break;
            }
        }
        // 如果该次循环时没有按键按下，则不会进入 if 语句，自然会进行下一轮循环
    }
    return 0;
}
```

课后思考

1. 如果程序中**只使用** `_getch()` 函数，而不使用 `_kbhit()` 进行判断，会发生什么？写成游戏后会是什么样子的?
2. 如果程序中**只使用** `_kbhit()` 函数，而不使用 `_getch()` 接收字符，会发生什么？将示例程序 `./src/01-02-kbhit.c` 中的 `break;` 去掉试一试，并尝试解释一下原因。

Chapter.02 管理你的控制台

在控制台上输出字符没有啥难的，但是，蛇需要**移动**，移动就必须要擦除一些内容，使用现有知识是很难做到的。下面介绍一些方法，但**未必**每一个都是需要用到的。它们看上去很复杂，但不用担心，学习一下样例，直接**调用**相应的**函数**就好啦，其背后的原理不必深究。

清空控制台

在头文件 `<stdlib.h>` 中，有一个这样的函数： `int system(char const* Command);` ，将字符串作为参数传入，可以执行该指令（Command）。

在 Windows 下，控制台清屏的命令是 `cls` ，在 Mac / Linux 下，清屏的命令可能是 `clear` 。因此，在 Windows 下，我们可以这样做：

```
// ./src/02-01-clear.c
#include <stdio.h>
#include <stdlib.h>
int main() {
    printf("qwertyuiop");
    system("cls");
    return 0;
}
```

移动光标位置

在当代常用 C 语言编译器中，并没有直接提供移动控制台光标的函数，我们可以利用 Windows API 自己写一个（不用纠结细节，直接用就好啦）：

```c
// ./src/02-02-gotoxy.c
#include <stdio.h>
#include <windows.h>

void gotoxy(int x, int y) {
    COORD coord = {x, y};
    SetConsoleCursorPosition(GetStdHandle(STD_OUTPUT_HANDLE), coord);
}

int main(void) {
    // 左上角坐标为 (0, 0)，向右为 x 正方向，向下为 y 正方向
    gotoxy(6, 10);
    printf("Hello world");
    return 0;
}
```

运行结果如下：

当然了，因为自定义的 `gotoxy()` 函数中使用了很多头文件 `<windows.h>` 中的函数，自然只能在 Windows 下运行。

隐藏光标

这同样是一个不必纠结细节的方法，纯纯是为了更加美观，没别的用处：

```c
// ./src/02-03-hide.c
#include <windows.h>

void HideCursor() {
    CONSOLE_CURSOR_INFO cursor_info = {1, 0};
    SetConsoleCursorInfo(GetStdHandle(STD_OUTPUT_HANDLE), &cursor_info);
```

```
}

int main() {
    HideCursor();  // 调用编写的函数
    return 0;
}
```

调整并锁定控制台窗口尺寸

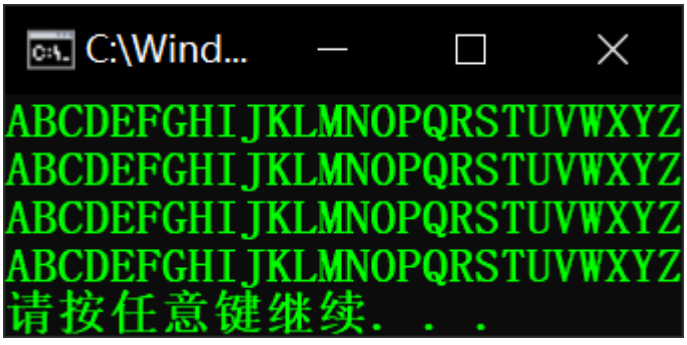
首先是调整尺寸的方法，我们可以使用 `system()` 函数来执行命令：`mode con cols=%d lines=%d`，其中，两个 `%d` 需要被替换为显示字符的列数与行数。于是，我们可以使用 `<stdio.h>` 中的 `sprintf()` 函数：

```c
// ./src/02-04-resize.c
#include <stdio.h>
#include <stdlib.h>

void SetConsoleSize(int cols, int lines) {
    char cmd[50];
    sprintf(cmd, "mode con cols=%d lines=%d", cols, lines);
    system(cmd);
}

int main() {
    SetConsoleSize(26, 5);  // 调用编写的函数
    for (int i = 0; i < 4; ++i) {
        for (int j = 0; j < 26; ++j) {
            putchar('A' + j);
        }
        putchar('\n');
    }
    system("pause");  // 暂停程序
    return 0;
}
```

运行结果如下，窗口就变得这么大了：



但是，在这种情况下，用户可以自己调整窗口大小而破坏我们设计好的尺寸，我们可以利用 Windows API 来将控制台窗口设置为禁止调整大小的模式：

```c
// ./src/02-05-fixed.c
#include <stdio.h>
#include <windows.h>

void SetFixedConsoleSize(int cols, int lines) {
    char cmd[50];
    sprintf(cmd, "mode con cols=%d lines=%d", cols, lines);
    system(cmd);
    // 这个函数需要三个参数，注意括号匹配关系，学会地正确复制粘贴
    SetWindowLongPtrA(GetConsoleWindow(),     // 参数一
                      GWL_STYLE,              // 参数二
                      (GetWindowLongPtrA(GetConsoleWindow(), GWL_STYLE)
                          & ~WS_SIZEBOX
                          & ~WS_MAXIMIZEBOX)  // 参数三
                     );
}

int main() {
    SetFixedConsoleSize(20, 20);
    return 0;
}
```

课后思考

1. 想一想你所构思的贪吃蛇的游戏，需要用到这一章所介绍的哪些方法？它们需要怎样被调用（比如是初始化时调用一次，还是蛇每动一格都要再调用，等等）？
2. 这一章介绍了许多方法，并将其中的大部分都封装成了一个函数，尝试调用这些函数，修改其中的参数，看看运行效果。并且思考，将它们写成函数有什么好处。

Chapter.03 将代码封装为一个函数

第二章中，我们提供了很多函数，现在我们来看一看使用函数的好处。

清晰简明的代码逻辑

首先来看一段例程，虽然这些函数具体是怎么实现的，以及这个程序想干什么我们不太清楚，但通过 `main` 函数中调用的一些方法，我们可以很直观地观察到代码的逻辑。

```c
int main() {
    USR_DATA usr_data = GetUsrData(usr_id);
    PrintWelcomeString(usr_data);
    while(GetInput(&input) != EOF) {
        if (HandleInput(input, usr_data) == MSG_EXIT){
```

```
            break;
        }
    }
    PrintByeString(usr_data);
    return 0;
}
```

比如，它首先根据用户的 `id` 获取到用户的数据（ `usr_data` ），打印欢迎信息，然后循环获取、处理用户输入，等等。

这样一来，无论是编写代码还是阅读代码，我们可以清晰及时地把握到代码的含义以及运行的逻辑，然后再去关心每个函数的具体实现。这样由宏观到微观的顺序，可以强化我们对于代码整体的把控。

干掉重复代码

对于重复代码，复制粘贴一时爽，改起bug火葬场。但如果写在函数里面，不就只要改一处了嘛。

代码与数据解耦合

将代码块提取为一个函数，可以形成一个**相对独立**的运行环境，通过传入参数与返回值与外界进行“交流”，避免污染外部环境（比如一不小心修改了循环变量）。同时，这些函数也可以在不同的地方被使用，甚至被不同的程序使用，增强了代码的可移植性。

但是，我们应进一步考虑**传入参数的“合法性”**，比如计算除法的函数应该对除数进行非0的校验，以防程序员失误或者用户“点炒饭”的可能。