数独游戏

```cpp
#include<windows.h>
int main() {
    SetConsoleCP(CP_UTF8);
    constexpr char CORNER[] = "\u254B";//制表符粗竖和水平 ╋
    constexpr char LINE[] = "\u2501";//制表符粗横线 ━
    constexpr char PIPE[] = "\u2503";//制表符粗竖 ┃
    constexpr char ARROW[] = "^";

    std::cout << CORNER << std::endl;
    std::cout << LINE << std::endl;
    std::cout << PIPE << std::endl;
    std::cout << ARROW << std::endl;
    system("pause");
```

单例模式（懒汉、饿汉）

单例模式的特点

1. 单例类只能有一个实例
2. 单例类必须自己创建自己的唯一实例
3. 单例类必须给其他对象提供这一实例

懒汉模式（线程不安全）

懒汉模式，顾名思义就是懒，没有对象需要调用它的时候不去实例化，有人来向它要对象的时候再实例化对象，因为懒，比我还懒

```
//懒汉式单例类.在第一次调用的时候实例化自己
public class Singleton{
    private Singleton() {}
    private static Singleton single=null;
    //静态工厂方法
    public static Singleton getInstance() {
         if (single == null) {
             single = new Singleton();
         }
        return single;
    }
}
```

也能看出来，并不能多线程使用，所以还有一个线程安全的写法

懒汉模式（线程安全）

我的理解就是，加了个关键字，对，就是那个每次都读不出来的那个词

synchronized,其实我还是能读出来的（傲娇脸）

```
public class Singleton {
    private static Singleton single=null;
    private Singleton (){}
    public static synchronized Singleton getInstance() {
    if (instance == null) {
        instance = new Singleton();
    }
    return single;
    }
}
```

网上有些博客说，这个看着100分，很遗憾，其实效率很低，99%的情况下不需要同步

饿汉模式

饿汉模式，就是它很饿，它的对象早早的就创建好了（懒汉是有人管它要了再创建）

直接看代码

```
//饿汉式单例类.在类初始化时，已经自行实例化
public class Singleton {
    private Singleton() {}
    private static final Singleton single = new
Singleton();
    //静态工厂方法
    public static Singleton getInstance() {
        return single;
    }
}
```

单例中懒汉和饿汉的本质区别在于以下几点：

1. 饿汉式是线程安全的,在类创建的同时就已经创建好一个静态的对象供系统使用,以后不在改变。懒汉式如果在创建实例对象时不加上synchronized则会导致对对象的访问不是线程安全的。
2. 从实现方式来讲他们最大的区别就是懒汉式是延时加载,他是在需要的时候才创建对象,而饿汉式在虚拟机启动的时候就会创建，饿汉式无需关注多线程问题、写法简单明了、能用则用。但是它是加载类时创建实例(上面有个朋友写错了)、所以如果是一个工厂模式、缓存了很多实例、那么就得考虑效率问题，因为这个类一加载则把所有实例不管用不用一块创建。

控制台\033方式设置字体颜色

简介

这个字体颜色几乎可以在任何能在控制台或者终端输入语句的编程语言中使用，本文使用的是C语言演示的控制台。
在此，不介绍Windows程序控制台中使用Windows.h库中的setconsoletextattribute函数，仅介绍\033控制字符(ESC)的方法。该方法可以直接适用于printf()函数中。

其中，\033(八进制)即ESC符号，Windows中为\027(十进制)，\x1b(十六进制)

注：不同编辑器控制台的颜色会不一样，比如Hbuilderx中的Node.js控制台的颜色，可以跳转最后查看。

格式

开始格式：

```
\033 [ 参数1 ; 参数2 ; 参数3 m      //以字母m结尾
```

内容格式：

```
正常的`printf`中的参数及内容
```

参数效果

总览

控制码	效果
\033[0m	关闭所有属性,恢复默认设置
\033[1m	设置`字体`高亮度
\033[2m	低亮（减弱）显示
\033[4m	下划线
\033[5m	闪烁
\033[7m	反显
\033[8m	消隐
\033[30m~\033[39m	`字体`颜色
\033[40m~\033[49m	`背景`颜色

单值控制码

不能与其他控制码联用

控制码	效果
\033[nA	光标上移n行
\033[nB	光标下移n行
\033[nC	光标右移n列
\033[nD	光标左移n列
\033[y;xH	设置光标位置（y行,x列）
\033[2J	清屏
\033[K	清除从光标到行尾的内容
\033[s	保存光标位置
\033[u	恢复光标位置
\033[?25l	隐藏光标
\033[?25h	显示光标

字体颜色（前景色）

控制码	字体效果
\033[30m	黑色
\033[31m	红色
\033[32m	绿色
\033[33m	黄色
\033[34m	蓝色
\033[35m	紫色
\033[36m	浅蓝色
\033[37m	白色
\033[38m	无
\033[39m	无

背景颜色

\033[40m	黑色
\033[41m	红色
\033[42m	绿色
\033[43m	黄色
\033[44m	蓝色
\033[45m	紫色
\033[46m	浅蓝色
\033[47m	白色
\033[48m	无
\033[49m	无

C++中ifstream一次读取整个文件

c++中一次读取整个文件的内容的方法：

1. ## 读取至char*的情况

```
std::ifstream t;
int length;
t.open("file.txt");      // open input file
t.seekg(0, std::ios::end);    // go to the end
length = t.tellg();           // report location (this is
the length)
t.seekg(0, std::ios::beg);    // go back to the beginning

buffer = new char[length];    // allocate memory for a
buffer of appropriate dimension
t.read(buffer, length);       // read the whole file into
the buffer
t.close();                    // close file handle

// ... do stuff with buffer here ...
```

2. 读取至std::string的情况：

   - 第一种方法：

     ```
     #include <string>
     #include <fstream>
     #include <streambuf>

     std::ifstream t("file.txt");
     std::string str((std::istreambuf_iterator<char>(t)),

                     std::istreambuf_iterator<char>());
     ```

   - 第二种方法

     ```
     #include <string>
     #include <fstream>
     #include <sstream>
     std::ifstream t("file.txt");
     std::stringstream buffer;
     buffer << t.rdbuf();
     std::string contents(buffer.str());
     ```

C++基本输入及读取整行

C++标准库提供了一组丰富的输入/输出功能。C++的I/O发生在流中，流是字节序列：

- 如果字节流是从设备（如键盘、磁盘驱动器、网络连接等）流向内存，这叫做输入操作。
- 如果字节流是从内存流向设备（如显示屏、打印机、磁盘驱动器、网络连接等），这叫做输出操作。

标准输入流

cin>>

预定义的cin 是 iostream 类的一个实例。cin 对象附属到标准输入设备，通常是键盘；cin 与流提取运算符`>>`结合使用。

cin默认使用空白（空格、制表符、换行符）来确定字符串的结束位置：

```
#include <iostream>
using namespace std;

void testInput(){
    int nAge;
    double height;
    string name;
    cout<<"Input Age, height, name: "; // 12 1.65 mike
josn
    cin>>nAge>>height>>name;
    cout<<"Age: "<<nAge<<", Height: "<<height<<", Name: "
<<name<<endl;
    // 12 1.65 mike
}
```

读取name时，只读取前面一部分（因空格结束了字符串的读取）。

cin.get

cin.get()可以读取每个字符返回，也可把读取的内容放到字符参数中：

```
int_type get();
basic_istream& get (char_type& c);
```

cin.get()依次读取每一个字符，直到结束（Ctrl+z）：

```
void testGet(){
    cout<<"Input: ";
    int nGet;
    while( (nGet=cin.get())!=EOF){ // 输入回车时，才依次读取（回车符本身也作为一个字符读取到
        cout<<(int)nGet<<endl;
    }
}
```

整行读取

std::getline可以读取一行放入string中，cin.get与cin.getline可以读取一行放入到字符数组中。

std::getline

std::getline从输入流中读取字符到string中，直到遇到分隔符（默认`\n`）：分隔符不读入缓冲区

```
istream& getline (istream& is, string& str, char delim);
istream& getline (istream& is, string& str);
```

若指定分隔符（如设为空格），可用于字符串分割：

```
#include <iostream>
#include <sstream>
using namespace std;

void testLine(){
    cout<<"Input: ";
    string strIn;
    std::getline(cin, strIn);
//    cout<<strIn;

    string strWord;
    istringstream is(strIn);
    while(std::getline(is, strWord, ' ')){
        cout<<strWord<<endl;
```

```
    }
}
```

cin.getline

cin.getline从输入流中读取字符到字符数组，直到遇到分隔符（默认\n）或数组最大长度-1；末尾会自动添加NULL；通过gcount可获取读取字符数（包括分隔符）：

```
basic_istream& getline (char_type* s, streamsize n );
basic_istream& getline (char_type* s, streamsize n,
char_type delim);
```

字符数组中不包含分隔符（且已从输入缓冲区中删掉了），但gcount包括分隔符（即：len(ary)+1）：

```
    char strIn[1024];
    for(int i=0; i<5; ++i) {
        cin.getline(strIn, 1024);
        cout << "count: " << cin.gcount() << endl;
    }
```

cin.get

cin.get从输入流中读取字符到字符数组，直到遇到分隔符（默认\n）或数组最大长度-1；末尾会自动添加NULL：

```
basic_istream& get (char_type* s, streamsize n);
basic_istream& get (char_type* s, streamsize n, char_type
delim);
```

cin.get读取时，会把分隔符留在缓冲区中，若不主动去掉，则会一直读取空字符串：

```
char strIn[1024];
for(int i=0; i<5; ++i) {
    cin.get(strIn, 1024).get();
}
```

若没有后面的get()，则在读取一行后，后续一直读取空（因换行符还在缓冲区中）。

C++ 日期 & 时间

C++ 标准库没有提供所谓的日期类型。C++ 继承了 C 语言用于日期和时间操作的结构和函数。为了使用日期和时间相关的函数和结构，需要在 C++ 程序中引用 头文件。

有四个与时间相关的类型：**clock_t、time_t、size_t** 和 **tm**。类型 clock_t、size_t 和 time_t 能够把系统时间和日期表示为某种整数。

结构类型 **tm** 把日期和时间以 C 结构的形式保存，tm 结构的定义如下：

```
struct tm {
  int tm_sec;   // 秒，正常范围从 0 到 59，但允许至 61
  int tm_min;   // 分，范围从 0 到 59
  int tm_hour;  // 小时，范围从 0 到 23
  int tm_mday;  // 一月中的第几天，范围从 1 到 31
  int tm_mon;   // 月，范围从 0 到 11
  int tm_year;  // 自 1900 年起的年数
  int tm_wday;  // 一周中的第几天，范围从 0 到 6，从星期日算起
  int tm_yday;  // 一年中的第几天，范围从 0 到 365，从 1 月 1 日算起
  int tm_isdst; // 夏令时
};
下面是 C/C++ 中关于日期和时间的重要
```

下面是 C/C++ 中关于日期和时间的重要函数。所有这些函数都是 C/C++ 标准库的组成部分，您可以在 C++ 标准库中查看一下各个函数的细节。

序号	函数 & 描述
1	**time_t time(time_t *time);** 该函数返回系统的当前日历时间，自 1970 年 1 月 1 日以来经过的秒数。如果系统没有时间，则返回 -1。
2	**char *ctime(const time_t *time);** 该返回一个表示当地时间的字符串指针，字符串形式 *day month year hours:minutes:seconds year\n\0*。
3	**struct tm *localtime(const time_t *time);** 该函数返回一个指向表示本地时间的 **tm** 结构的指针。
4	**clock_t clock(void);** 该函数返回程序执行起（一般为程序的开头），处理器时钟所使用的时间。如果时间不可用，则返回 -1。
5	**char * asctime (const struct tm * time);** 该函数返回一个指向字符串的指针，字符串包含了 time 所指向结构中存储的信息，返回形式为：day month date hours:minutes:seconds year\n\0。
6	**struct tm *gmtime(const time_t *time);** 该函数返回一个指向 time 的指针，time 为 tm 结构，用协调世界时（UTC）也被称为格林尼治标准时间（GMT）表示。
7	**time_t mktime(struct tm *time);** 该函数返回日历时间，相当于 time 所指向结构中存储的时间。
8	**double difftime (time_t time2, time_t time1);** 该函数返回 time1 和 time2 之间相差的秒数。
9	**size_t strftime();** 该函数可用于格式化日期和时间为指定的格式。

当前日期和时间

下面的实例获取当前系统的日期和时间，包括本地时间和协调世界时（UTC）。

```
#include <iostream>
#include <ctime>

using namespace std;

int main( )
{
   // 基于当前系统的当前日期/时间
   time_t now = time(0);

   // 把 now 转换为字符串形式
   char* dt = ctime(&now);

   cout << "本地日期和时间：" << dt << endl;
```

```
    // 把 now 转换为 tm 结构
    tm *gmtm = gmtime(&now);
    dt = asctime(gmtm);
    cout << "UTC 日期和时间："<< dt << endl;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
本地日期和时间：Sat Jan  8 20:07:41 2011

UTC 日期和时间：Sun Jan  9 03:07:41 2011
```

使用结构 tm 格式化时间

tm 结构在 C/C++ 中处理日期和时间相关的操作时，显得尤为重要。tm 结构以 C 结构的形式保存日期和时间。大多数与时间相关的函数都使用了 tm 结构。下面的实例使用了 tm 结构和各种与日期和时间相关的函数。

在练习使用结构之前，需要对 C 结构有基本的了解，并懂得如何使用箭头 -> 运算符来访问结构成员。

```
#include <iostream>
#include <ctime>

using namespace std;

int main( )
{
    // 基于当前系统的当前日期/时间
    time_t now = time(0);

    cout << "1970 到目前经过秒数:" << now << endl;

    tm *ltm = localtime(&now);

    // 输出 tm 结构的各个组成部分
    cout << "年: "<< 1900 + ltm->tm_year << endl;
    cout << "月: "<< 1 + ltm->tm_mon<< endl;
    cout << "日: "<<  ltm->tm_mday << endl;
    cout << "时间: "<< ltm->tm_hour << ":";
```

```
    cout << ltm->tm_min << ":";
    cout << ltm->tm_sec << endl;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
1970 到目前时间:1503564157
年: 2017
月: 8
日: 24
时间: 16:42:37
```

C 库函数 - time()

描述

C 库函数 **time_t time(time_t *seconds)** 返回自纪元 Epoch（1970-01-01 00:00:00 UTC）起经过的时间，以秒为单位。如果 **seconds** 不为空，则返回值也存储在变量 **seconds** 中。

声明

下面是 time() 函数的声明。

```
time_t time(time_t *seconds)
```

参数

- **seconds** -- 这是指向类型为 time_t 的对象的指针，用来存储 seconds 的值。

返回值

以 time_t 对象返回当前日历时间。

实例

下面的实例演示了 time() 函数的用法。

```
#include <stdio.h>
#include <time.h>

int main ()
{
  time_t seconds;

  seconds = time(NULL);
  printf("自 1970-01-01 起的小时数 = %ld\n", seconds/3600);

  return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
自 1970-01-01 起的小时数 = 373711
```

C 库函数 - ctime()

描述

C 库函数 **char *ctime(const time_t *timer)** 返回一个表示当地时间的字符串，当地时间是基于参数 **timer**。

返回的字符串格式如下： **Www Mmm dd hh:mm:ss yyyy** 其中，*Www* 表示星期几，*Mmm* 是以字母表示的月份，*dd* 表示一月中的第几天，*hh:mm:ss* 表示时间，*yyyy* 表示年份。

声明

下面是 ctime() 函数的声明。

```
char *ctime(const time_t *timer)
```

参数

- **timer** -- 这是指向 time_t 对象的指针，该对象包含了一个日历时间。

返回值

该函数返回一个 C 字符串，该字符串包含了可读格式的日期和时间信息。

实例

下面的实例演示了 ctime() 函数的用法。

```
#include <stdio.h>
#include <time.h>

int main ()
{
   time_t curtime;

   time(&curtime);

   printf("当前时间 = %s", ctime(&curtime));

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
当前时间 = Mon Aug 13 08:23:14 2012
```

C 库函数 - localtime()

描述

C 库函数 **struct tm *localtime(const time_t *timer)** 使用 timer 的值来填充 **tm** 结构。**timer** 的值被分解为 **tm** 结构，并用本地时区表示。

声明

下面是 localtime() 函数的声明。

```
struct tm *localtime(const time_t *timer)
```

参数

- **timer** -- 这是指向表示日历时间的 time_t 值的指针。

返回值

该函数返回指向 **tm** 结构的指针，该结构带有被填充的时间信息。下面是 tm 结构的细节：

```
struct tm {
   int tm_sec;         /* 秒，范围从 0 到 59
*/
   int tm_min;         /* 分，范围从 0 到 59
*/
   int tm_hour;        /* 小时，范围从 0 到 23
*/
   int tm_mday;        /* 一月中的第几天，范围从 1 到 31
              */
   int tm_mon;         /* 月份，范围从 0 到 11
*/
   int tm_year;        /* 自 1900 起的年数                */
   int tm_wday;        /* 一周中的第几天，范围从 0 到 6
         */
   int tm_yday;        /* 一年中的第几天，范围从 0 到 365
                */
   int tm_isdst;       /* 夏令时                          */

};
```

实例

下面的实例演示了 localtime() 函数的用法。

```
#include <stdio.h>
#include <time.h>

int main ()
{
   time_t rawtime;
   struct tm *info;
   char buffer[80];

   time( &rawtime );

   info = localtime( &rawtime );
   printf("当前的本地时间和日期：%s", asctime(info));

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
当前的本地时间和日期：Thu Aug 23 09:12:05 2012
```

C 库函数 - clock()

描述

C 库函数 **clock_t clock(void)** 返回程序执行起（一般为程序的开头），处理器时钟所使用的时间。为了获取 CPU 所使用的秒数，您需要除以 CLOCKS_PER_SEC。

在 32 位系统中，CLOCKS_PER_SEC 等于 1000000，该函数大约每 72 分钟会返回相同的值。

声明

下面是 clock() 函数的声明。

```
clock_t clock(void)
```

参数

- **NA**

返回值

该函数返回自程序启动起，处理器时钟所使用的时间。如果失败，则返回 -1 值。

实例

下面的实例演示了 clock() 函数的用法。

```
#include <time.h>
#include <stdio.h>

int main()
{
   clock_t start_t, end_t;
   double total_t;
   int i;

   start_t = clock();
   printf("程序启动，start_t = %ld\n", start_t);

   printf("开始一个大循环，start_t = %ld\n", start_t);
   for(i=0; i< 10000000; i++)
   {
   }
   end_t = clock();
   printf("大循环结束，end_t = %ld\n", end_t);

   total_t = (double)(end_t - start_t) / CLOCKS_PER_SEC;
```

```
    printf("CPU 占用的总时间：%f\n", total_t  );
    printf("程序退出...\n");

    return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
程序启动，start_t = 2614
开始一个大循环，start_t = 2614
大循环结束，end_t = 28021
CPU 占用的总时间：0.025407
程序退出...
```

C 库函数 - asctime()

描述

C 库函数 **char *asctime(const struct tm *timeptr)** 返回一个指向字符串的指针，它代表了结构 **struct timeptr** 的日期和时间。

声明

下面是 asctime() 函数的声明。

```
char *asctime(const struct tm *timeptr)
```

参数

timeptr 是指向 tm 结构的指针，包含了分解为如下各部分的日历时间：

```
struct tm {
   int tm_sec;         /* 秒，范围从 0 到 59
*/
   int tm_min;         /* 分，范围从 0 到 59
*/
   int tm_hour;        /* 小时，范围从 0 到 23
*/
   int tm_mday;        /* 一月中的第几天，范围从 1 到 31
            */
   int tm_mon;         /* 月份，范围从 0 到 11
*/
   int tm_year;        /* 自 1900 起的年数                  */
   int tm_wday;        /* 一周中的第几天，范围从 0 到 6
         */
   int tm_yday;        /* 一年中的第几天，范围从 0 到 365
               */
   int tm_isdst;       /* 夏令时                            */

};
```

返回值

该函数返回一个 C 字符串，包含了可读格式的日期和时间信息 **Www Mmm dd hh:mm:ss yyyy**，其中，*Www* 表示星期几，*Mmm* 是以字母表示的月份，*dd* 表示一月中的第几天，*hh:mm:ss* 表示时间，*yyyy* 表示年份。

实例

下面的实例演示了 asctime() 函数的用法。

```
#include <stdio.h>
#include <string.h>
#include <time.h>

int main()
{
   struct tm t;
```

```
    t.tm_sec    = 10;
    t.tm_min    = 10;
    t.tm_hour   = 6;
    t.tm_mday   = 25;
    t.tm_mon    = 2;
    t.tm_year   = 89;
    t.tm_wday   = 6;

    puts(asctime(&t));

    return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
Sat Mar 25 06:10:10 1989
```

C 库函数 - gmtime()

描述

C 库函数 **struct tm *gmtime(const time_t *timer)** 使用 **timer** 的值来填充 **tm** 结构，并用协调世界时（UTC）也被称为格林尼治标准时间（GMT）表示。

声明

下面是 gmtime() 函数的声明。

```
struct tm *gmtime(const time_t *timer)
```

参数

- **timeptr** -- 这是指向表示日历时间的 time_t 值的指针。

返回值

该函数返回指向 tm 结构的指针，该结构带有被填充的时间信息。下面是 timeptr 结构的细节：

```
struct tm {
   int tm_sec;         /* 秒，范围从 0 到 59
*/
   int tm_min;         /* 分，范围从 0 到 59
*/
   int tm_hour;        /* 小时，范围从 0 到 23
*/
   int tm_mday;        /* 一月中的第几天，范围从 1 到 31
              */
   int tm_mon;         /* 月份，范围从 0 到 11
*/
   int tm_year;        /* 自 1900 起的年数                */
   int tm_wday;        /* 一周中的第几天，范围从 0 到 6
         */
   int tm_yday;        /* 一年中的第几天，范围从 0 到 365
                */
   int tm_isdst;       /* 夏令时                          */

};
```

实例

下面的实例演示了 gmtime() 函数的用法。

```
#include <stdio.h>
#include <time.h>

#define BST (+1)
#define CCT (+8)

int main ()
{

   time_t rawtime;
```

```
   struct tm *info;

   time(&rawtime);
   /* 获取 GMT 时间 */
   info = gmtime(&rawtime );

   printf("当前的世界时钟：\n");
   printf("伦敦：%2d:%02d\n", (info->tm_hour+BST)%24, info->tm_min);
   printf("中国：%2d:%02d\n", (info->tm_hour+CCT)%24, info->tm_min);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
当前的世界时钟：
伦敦：14:10
中国：21:10
```

C 库函数 - mktime()

描述

C 库函数 **time_t mktime(struct tm *timeptr)** 把 **timeptr** 所指向的结构转换为自 1970 年 1 月 1 日以来持续时间的秒数，发生错误时返回-1。

声明

下面是 mktime() 函数的声明。

```
time_t mktime(struct tm *timeptr)
```

参数

- **timeptr** -- 这是指向表示日历时间的 time_t 值的指针，该日历时间被分解为以下各部分。下面是 timeptr 结构的细节：

```
struct tm {
   int tm_sec;         /* 秒，范围从 0 到 59
   */
   int tm_min;         /* 分，范围从 0 到 59
   */
   int tm_hour;        /* 小时，范围从 0 到 23
     */
   int tm_mday;        /* 一月中的第几天，范围从 1 到 31
                  */
   int tm_mon;         /* 月份，范围从 0 到 11
     */
   int tm_year;        /* 自 1900 起的年数
*/
   int tm_wday;        /* 一周中的第几天，范围从 0 到 6
           */
   int tm_yday;        /* 一年中的第几天，范围从 0 到 365
                  */
   int tm_isdst;       /* 夏令时
*/
};
```

返回值

该函数返回自 1970 年 1 月 1 日以来持续时间的秒数。如果发生错误，则返回 -1 值。

实例

下面的实例演示了 mktime() 函数的用法。

```
#include <stdio.h>
#include <time.h>

int main () {
    int ret;
    struct tm info;
    char buffer[80];

    info.tm_year = 2021 - 1900;
```

```
    info.tm_mon = 7 - 1;
    info.tm_mday = 4;
    info.tm_hour = 0;
    info.tm_min = 0;
    info.tm_sec = 1;
    info.tm_isdst = -1;

    ret = mktime(&info);
    if( ret == -1 ) {
        printf("Error: unable to make time using
mktime\n");
    } else {
        strftime(buffer, sizeof(buffer), "%c", &info );
        printf(buffer);
    }

    return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
Sun Jul  4 00:00:01 2021
```

```
/* 输入日期判断是周几 */
#include <stdio.h>      /* printf, scanf */
#include <time.h>       /* time_t, struct tm, time, mktime
*/

int main ()
{
    time_t rawtime;
    struct tm * timeinfo;
    int year, month ,day;
    const char * weekday[] = { "周日", "周一","周二", "周
三","周四", "周五", "周六"};

    /* 用户输入日期 */
    printf ("年: "); fflush(stdout); scanf ("%d",&year);
    printf ("月: "); fflush(stdout); scanf ("%d",&month);
    printf ("日: "); fflush(stdout); scanf ("%d",&day);
```

```
    /* 获取当前时间信息，并修改用户输入的输入信息 */
    time ( &rawtime );
    timeinfo = localtime ( &rawtime );
    timeinfo->tm_year = year - 1900;
    timeinfo->tm_mon = month - 1;
    timeinfo->tm_mday = day;

    /* 调用 mktime: timeinfo->tm_wday  */
    mktime ( timeinfo );

    printf ("那一天是：%s\n", weekday[timeinfo->tm_wday]);

    return 0;
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
年：2018
月：7
日：26
那一天是：周四
```

C 库函数 - difftime()

描述

C 库函数 **double difftime(time_t time1, time_t time2)** 返回 **time1** 和 **time2** 之间相差的秒数 **(time1 - time2)**。这两个时间是在日历时间中指定的，表示了自纪元 Epoch（协调世界时 UTC：1970-01-01 00:00:00）起经过的时间。

声明

下面是 difftime() 函数的声明。

```
double difftime(time_t time1, time_t time2)
```

参数

- **time1** -- 这是表示结束时间的 time_t 对象。
- **time2** -- 这是表示开始时间的 time_t 对象。

返回值

该函数返回以双精度浮点型 double 值表示的两个时间之间相差的秒数 (time1 - time2)。

实例

下面的实例演示了 difftime() 函数的用法。

```
#include <stdio.h>
#include <time.h>
#ifdef _WIN32
#include <Windows.h>
#else
#include <unistd.h>
#endif

int main ()
{
   time_t start_t, end_t;
   double diff_t;

   printf("程序启动...\n");
   time(&start_t);

   printf("休眠 5 秒...\n");
   sleep(5);

   time(&end_t);
   diff_t = difftime(end_t, start_t);

   printf("执行时间 = %f\n", diff_t);
   printf("程序退出...\n");
```

```
   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
程序启动...
休眠 5 秒...
执行时间 = 5.000000
程序退出...
```

C 库函数 - strftime()

描述

C 库函数 **size_t strftime(char *str, size_t maxsize, const char *format, const struct tm *timeptr)** 根据 **format** 中定义的格式化规则，格式化结构 **timeptr** 表示的时间，并把它存储在 **str** 中。

声明

下面是 strftime() 函数的声明。

```
size_t strftime(char *str, size_t maxsize, const char
*format, const struct tm *timeptr)
```

参数

- **str** -- 这是指向目标数组的指针，用来复制产生的 C 字符串。
- **maxsize** -- 这是被复制到 str 的最大字符数。
- **format** -- 这是 C 字符串，包含了普通字符和特殊格式说明符的任何组合。这些格式说明符由函数替换为表示 tm 中所指定时间的相对应值。格式说明符是：

说明符	替换为	实例
%a	缩写的星期几名称	Sun

说明符	替换为	实例
%A	完整的星期几名称	Sunday
%b	缩写的月份名称	Mar
%B	完整的月份名称	March
%c	日期和时间表示法	Sun Aug 19 02:56:02 2012
%d	一月中的第几天（01-31）	19
%H	24 小时格式的小时（00-23）	14
%I	12 小时格式的小时（01-12）	05
%j	一年中的第几天（001-366）	231
%m	十进制数表示的月份（01-12）	08
%M	分（00-59）	55
%p	AM 或 PM 名称	PM
%S	秒（00-61）	02
%U	一年中的第几周，以第一个星期日作为第一周的第一天（00-53）	33
%w	十进制数表示的星期几，星期日表示为 0（0-6）	4
%W	一年中的第几周，以第一个星期一作为第一周的第一天（00-53）	34
%x	日期表示法	08/19/12
%X	时间表示法	02:50:06
%y	年份，最后两个数字（00-99）	01
%Y	年份	2012

说明符	替换为	实例
%Z	时区的名称或缩写	CDT
%%	一个 % 符号	%

- **timeptr** -- 这是指向 tm 结构的指针，该结构包含了一个被分解为以下各部分的日历时间：

```
struct tm {
   int tm_sec;         /* 秒，范围从 0 到 59
*/
   int tm_min;         /* 分，范围从 0 到 59
*/
   int tm_hour;        /* 小时，范围从 0 到 23
*/
   int tm_mday;        /* 一月中的第几天，范围从 1 到 31
            */
   int tm_mon;         /* 月份，范围从 0 到 11
*/
   int tm_year;        /* 自 1900 起的年数                    */
   int tm_wday;        /* 一周中的第几天，范围从 0 到 6
         */
   int tm_yday;        /* 一年中的第几天，范围从 0 到 365
               */
   int tm_isdst;       /* 夏令时                              */

};
```

返回值

如果产生的 C 字符串小于 size 个字符（包括空结束字符），则会返回复制到 str 中的字符总数（不包括空结束字符），否则返回零。

实例

下面的实例演示了 strftime() 函数的用法。

```
#include <stdio.h>
#include <time.h>

int main ()
{
   time_t rawtime;
   struct tm *info;
   char buffer[80];

   time( &rawtime );

   info = localtime( &rawtime );

   strftime(buffer, 80, "%Y-%m-%d %H:%M:%S", info);
   printf("格式化的日期 & 时间 : |%s|\n", buffer );

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
格式化的日期 & 时间 : |2018-09-19 08:59:07|
```

关于getch()引发的思考，以及在Linux下实现的方式

这两天在玩linux系统，就准备用c写个小游戏出来，然而基本所有游戏都需要用到getch()函数，因为这是一个无缓冲，不回显的输入函数。

关于缓冲输入和非缓冲输入

> **缓冲输入**：文件缓冲输入通常表现为延迟回显。即您所键入的字符被收集并存储在一个被称为缓冲区的（buffer）的临时存储区域中。按下回车键则可使您所键入的字符快对程序可用。
> **非缓冲输入**：输入字符立即回显。即该内容对程序立即可用。

关于输入的回显

> **回显**：显示正在执行的批处理命令及执行的结果等。

总结

缓冲输入可修改，需要按下回车键才能对程序用；非缓冲输入不可修改，输入即可用。

回显输入会显示输入的东西，不回显的输入不会显示输入的东西。（是不是很直白hhh）

所以写游戏基本都用无缓冲不回显的输入函数。

但是。。。getch()这个函数不是C的标准函数函数。在Win下可以包含conio.h这个头文件以调用，而linux下的编译器并不提供这个头文件，这就很难受了。

那就只能百度了，发现了相应的**解决方法**。

```
#include <termio.h>
int getch(void)
{
	struct termios tm, tm_old;
	int fd = 0, ch;

	if (tcgetattr(fd, &tm) < 0) {//保存现在的终端设置
		return -1;
	}

	tm_old = tm;
	cfmakeraw(&tm);//更改终端设置为原始模式，该模式下所有的输入数据以字节为单位被处理
	if (tcsetattr(fd, TCSANOW, &tm) < 0) {//设置上更改之后的设置
		return -1;
	}

	ch = getchar();
	if (tcsetattr(fd, TCSANOW, &tm_old) < 0) {//更改设置为最初的样子
```

```
            return -1;
        }

        return ch;
}
```

总体的思路就是设置终端的属性
设置为原始模式，这种模式下输入就是无缓冲的，
设置过去，输入完之后然后再更改回来
主要就是两个函数
tcgetattr()和tcsetattr()

如何在Linux系统中实现getch()

几天我们老师给我们布置了一个小作业，写一个可以测反应时间的小程序，甚至还给了个输入示例

```
# include < iostream >
# include < conio.h >
using namespace std;
int main()
{
    char a;
    a=getch();
    cout<<a<<endl;
    return 0;
}
```

我：？？？

然后我就不理解了，这这这不歧视Mac和Linux用户吗?

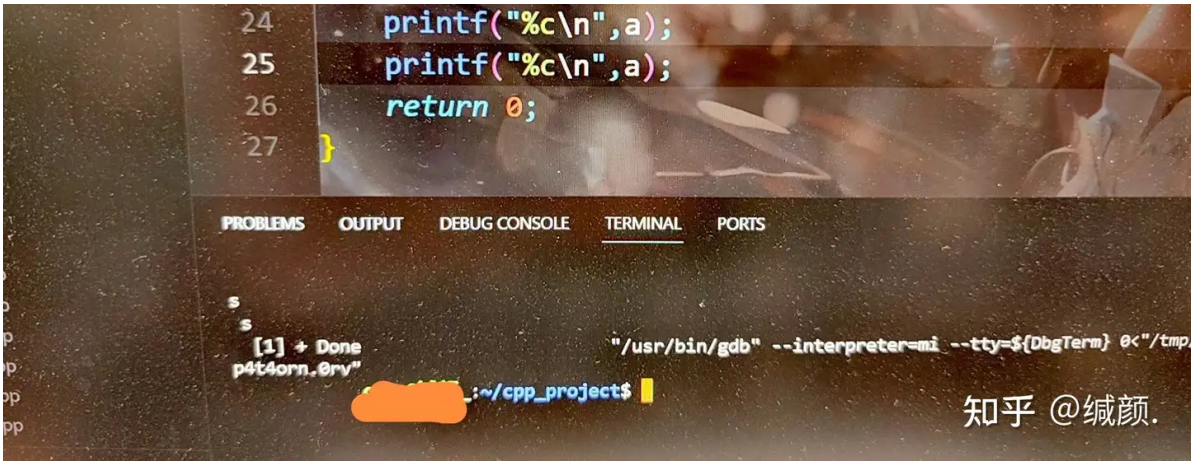
getch()函数并非标准库函数，它所在的头文件< conio.h>仅存在于windows中

那啥办呢?

于是我开始疯狂搜索了一个小时，结果发现某些博主....

我直接怀疑这种东西我助教也看不懂

正当我打算把这坨东西直接扔在作业上并老老实实注释这是我从网上嫖的时，我的终端出现了一些不可理谕的事故



这咋还自动缩进呢？？？

好吧，又经过半个小时的搜索，我终于找到了一个简单的方式

```
# include < iostream >
using namespace std;
int main()
{
    system("stty -icanon");   //无缓冲
    system("stty -echo");     //无回显
```

```
    char a;
    a=getchar();
    cout<<a<<endl;
    system("stty icanon");    //恢复
    system("stty echo");
    return 0;
}
```

其中system为系统指令直接禁用缓冲区与回显

使用这几个语句可以轻松实现无缓冲无回显输入

getch()函数的使用方法及其返回值问题

getch()函数依赖于头文件 conio.h .会在windows平台下从控制台无回显地取一个字符，并且返回读取到的字符。

然而，我在实际用这个函数才发现getch()这个函数并不简单。

> getch函数从控制台读取单个字符而不回显，函数不能去读取CTRL+C，当读取一个功能键或方向键，函数必须调用两次（这就说明可以用这个函数去监控功能键和方向键），第一次调用返回0或0xe0，第二次返回实际的键代码

例如：

```
#include <stdio.h>
#include <conio.h>
int main()
{
    while (true)
    {
        int tmp = _getch();
        printf(" ……\n");     //测试每键入一次，打印几次
    }
    return 0;
}
```

在这个简单的小程序中，我测试了几个键盘的按键。

- 在a~z的英文字母、数字键、以及Tab、space、ESC、Backspace、Enter等几个常用键上，printf（）只会打印一次，也就是说，getch（）会立即返回真实的键码值，并且被tmp变量接收。
- 在键盘上输入上下左右的方向键，F1~F9、Delete等功能键时，printf（）会打印两次。

为了探究getch()的真相，我改写了以上函数。

```
int main()
{
    while (true)
    {
        int tmp = _getch();
        int tmp2 = _getch();
        printf(" tmp=%d\n tmp2=%d\n",tmp,tmp2);
    }
    return 0;
}
```

在键盘上依次输入上下左右得到如下键值：

- 上　tmp=224　tmp2=72
- 下　tmp=224　tmp2=80
- 左　tmp=224　tmp2=75
- 右　tmp=224　tmp2=77

上下左右方向键，getch()第一次返回 224（0xe0），第二次返回真实键值

在键盘上依次输入F1~F10、F11、F12、Delete得到如下键值：

- F1　　tmp=0　tmp2=59
- F2　　tmp=0　tmp2=60
- F3　　tmp=0　tmp2=61
- F10　　tmp=0　tmp2=68

F1~F10，getch()第一次返回 0，第二次返回该键的真实键值（59~68）

- F11　　tmp=224　tmp2=133
- F12　　tmp=224　tmp2=134
- Delete　tmp=224　tmp2=83

那么问题就来了，getch()一会返回一个值，一会返回两个值，究竟要怎样写才不会出错呢?

```
#include <conio.h>
int main()
{
    char tmp;   //读取键值，或过滤功能键的第一个返回值
    char tmp2;  //接受功能键
    while (true)
    {
        tmp = _getch();

        if (tmp == 0 || tmp == -32)//表示读取的是功能键或者方向键，丢掉第一个返回值，读取第二个返回值
        {
            switch (tmp2 = _getch())    //接收功能键返回值
            {
            case 72://上
                printf("This is ↑\n");
                break;
            case 59://F1
                printf("This is F1\n");
                break;
            default:
                break;
            }
        }
        else            //普通按键，如字母、数字、space，Esc等按键
        {
            switch (tmp)
            {
            case 32://空格
                printf("This is Space\n");
                break;
            case 27://Esc
                printf("This is Esc\n");
                break;
            default:
                break;
```

```
            }
        }

    }
    return 0;
}
```

注意1：在以上代码中 tmp 为char类型，可接受的值为 -128~127之间，所以原本 0xe0的返回值(10进制为224) 会被转换为 -32。

转换原理为 超出char范围的(即127以后的数字)，把差值从 char类型的另一侧极限值重新开始计算(即-128往后排)

例如：224——> 超出97(224-127)——> -128+97-1=-32

详见char的越界赋值即其原理剖析

解决办法：

- 可以把tmp定义 unsigned char类型
- 通过计算、或者测试，得到可用的键值。

注意2：用getch()函数时，编译器可能会给出如下错误

```
严重性    代码    说明    项目    文件    行    禁止显示状态
错误    C4996    'getch': The POSIX name for this item is deprecated. Instead, use the ISO C and C++ conformant name: _getch. See online help for details.
```

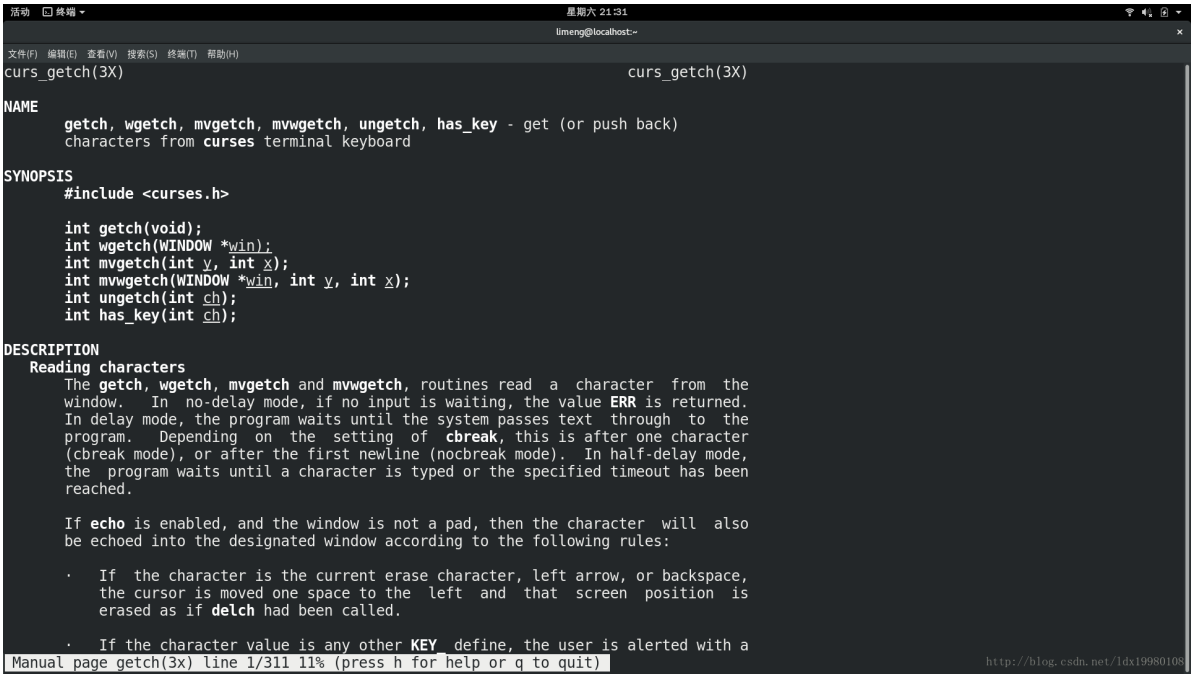
解决办法：把getch()换成编译器要求的_getch()函数即可

linux关于getch()与getche()的问题

getchar()、getche()、getch() 函数，它们都用来从控制台获取字符，getchar() 会等待用户按下回车键才开始读取，而 getche()、getch() 会立即读取。这是因为 getchar() 带有缓冲区，用户输入的数据会暂时保存到缓冲区，直到按下回车键才开始读取；而 getche()、getch() 不带缓冲区，只能立即读取。

在windows下，getche()、getch() 都在conio.h头文件定义好了，可以直接拿来用。

但是在linux下没有conio.h这个头文件，通过man getch可知道getch()是存在的，但是getche()不存在。



使用 getch 等函数，必须使用 curses 库，在 gcc 编译是用 -l curses 加进这个库。
要调用 getch ，程序初始化时应当调用 initscr，否则将出现误。程序结束时，要调用 endwind。

```
//代码
#include<stdio.h>
#include<stdlib.h>
#include<curses.h>
#include<unistd.h>
int  main(void)
{
    initscr();
    getch();
    sleep(1);
    printf("123456");
    getchar();
    endwin();
    return 0;
}
```

运行

```
[limeng@localhost ~]$ vim b.c
[limeng@localhost ~]$ gcc b.c -lncurses
[limeng@localhost ~]$ ./a.out
```

输入1
结果
1123456(光标在此等待，按任意键回到原来的界面)
//这里的结果说明curses.h的getch(),是有回显的

按照我的理解，当调用initscr()时，相当于打开了一个新的虚拟窗口，我输入一个字符，回显出来，停顿一秒后打印出字符串"123456",然后等待输入，按任意键后，关闭了刚开的虚拟界面(此时程序已经结束)，回到了原先的界面，等待执行其他命令。

由此可见，getch在不同系统中的功能是不同的，实现方式也不同，在windows下没有回显，而在linux下却有回显。

那么，怎么在linux实现和windows下getch()和getche()的功能呢?
我们可以写一个conio.h和conio.c文件

```
//conio.c
#include "conio.h"
#include<stdio.h>

char getch()
{
char c;
system("stty -echo");  //不回显
system("stty -icanon");//设置一次性读完操作，如使用getchar()读操作，不需要按enter
c=getchar();
system("stty icanon");//取消上面的设置
system("stty echo");//回显
return c;
}

char getche()
{
    char c;
```

```
    system("stty -icanon");
    c=getchar();
    system("stty icanon");
    return c;
}
```

```
//conio.h
#ifndef _CONIO_H
#define _CONIO_H

#include <stdio.h>
#include <stdlib.h>

char getch();

char getche();

#endif
```

在需要用到”conio.h”的文件中只需要包含进去就可以啦！！！

在Linux中，按上下左右键为什么变成^[[A^[[B^[[C^[[D?

你用getchar可以发现上下左右的编码有三位，分别是十进制27，91和65-68，按顺序对应了上下右左，查阅ascii码可知91是[，65-68是ABCD，27为esc键，可能会输出为^[，所以如果你的shell不支持读取这样的3字节编码，就会输出成^[[A，B，C，D这样的乱码了

其他特殊键位一般也都是27 91开头，大部分也是3字节，但有些甚至是四字节，比如pageup/pagedown，但都有固定格式（可能是某个标准规定的吧）

Windows 系统中 std::cout 输出中文乱码

下面一段代码:

```
#include <iostream>

int main() {
    std::cout << "你好，世界！"<<std::endl;
    return 0;
}
```

代码文件保存为 UTF-8 编码，编译执行，会发现控制台中输出乱码。

在中文版 Windows 系统中，控制台的默认字符编码为 GBK，如果代码中的字符串采用 UTF-8 编码，自然会因为编码不对而输出乱码。

解决方法是在程序中手动设置控制台输出编码为 UTF-8：

```
#include <iostream>
#include <windows.h>

int main() {
    SetConsoleOutputCP(CP_UTF8); // 将控制台编码设置为 UTF-8

    std::cout << "你好，世界！"<<std::endl;
    return;
}
```

C语言printf()、sprintf()、vsprintf()的区别与联系

printf() 在控制台应用程序中最为常用，使用也很简单。其参数为格式化字符串

函数原型：printf(const char *format,[argument]);

例如：

```
int a=1,b=2;
printf("a=%d,b=%d",a,b);
```

输出：

```
a=1,b=2
```

sprintf() 用于将输出存到字符缓冲中

函数原型：sprintf(char *buffer, const char *format, [argument]);

例如：

```
int a=1,b=2;
char s[10];
sprintf(s,"a=%d,b=%d",1,2);
puts(s);
```

输出：

```
a=1,b=2
```

vsprintf() 与sprintf() 功能类似。

需要引入相关头文件 #include <stdarg.h>

函数原型： vsprintf(char *buffer, char *format, va_list param);

例如：

```
void Myprintf(const char* fmt,...);

int a=1,b=2;
char s[10];
Myprintf("a=%d,b=%d",a,b);

void Myprintf(const char* fmt,...)
{
  char s[10];
  va_start(ap, fmt);
  vsprintf(s,fmt,ap);
  va_end(ap);
  puts(s);
}
```

输出：

```
a=1,b=2
```

因此可以用 vsprintf() 来实现 sprintf()。

其中 va_start、va_end、va_list 是什么东东一会儿再解释，先往下看。

既然sprintf() 与 vsprintf() 功能类似，为什么不统一下，而且 vsprintf() 用法如此麻烦呢?

假设你想封装一个子函数 Myprintf() ,其功能是将格式化字符串输出两遍，若用 sprintf() 实现：

(错误的)代码：

```
void Myprintf(const char* fmt,...)
{
  char s[10];
  sprintf(s,fmt);
  puts(s);
  puts(s);
}
```

当你尝试这么调用该函数时：

```
int a=1,b=2;
Myprintf("a=%d,b=%d",a,b);
```

输出：

```
a=?,b=?
a=?,b=?
```

我也不知道它会输出几，但一定不会是 a=1,b=2 。

这显然是错误的!

为什么呢?

当你这么调用 Myprintf() 时,其中的sprintf() 实际上是这样的：
sprintf(s,"a=%d,b=%d")

而不是：sprintf(s,"a=%d,b=%d",a,b) !!

因为这么调用的话只能解析到第一个字符串而无法解析到后面的参数。

所以类似这种封装用 sprintf() 是无法实现的，**使用 sprintf() 只能原始的为它输入所有的参数而不能以传参的方式给**它。

若用 vsprintf()那么这个问题就可以迎刃而解。

函数如下：

```
void Myprintf(const char* fmt,...)
{
  char s[10];
  va_list ap;
  va_start(ap,fmt);
  vsprintf(s,fmt,ap);
  va_end(ap);
  puts(s);
  puts(s);
}
```

当你这么调用时：

```
Myprintf("a=%d,b=%d",a,b);
```

输出：

```
a=1,b=2
a=1,b=2
```

成功了！哇哈哈哈哈！！想想也是有点小激动！！！

现在来解释一下这个 vsprintf():

首先你要知道函数执行时，函数的参数是倒序压入栈中的，vsprintf() 为了能够解析你传给它的多个参数，你必须告诉它参数从哪里开始。

vadefs.h 头文件中这么定义： typedef char * va_list，于是我们定义了一个 va_list ap 来保存参数起始地址。

va_start(ap,fmt) 就找出这个函数在栈中排列的一堆参数的起始地址，然后直接浏览栈中参数，并用 vsprintf() 实现格式化字符串的读取，最后 vs_end(ap) 释放ap，就像释放指针一样。通俗地说就是因为 vsprintf() 比 sprintf() 更加接近底层(栈)，因此能实现这个目的，也是因此能用 vsprintf() 来实现 sprintf()。

打的手好酸~0.0 希望大家发现错误帮我指正，不幸感激！最后给懒得敲代码的童鞋附上这段演示代码：

```
#include <stdarg.h>
void Myprint(const char* fmt,...);

int main()
{
    int a=1,b=2;
    Myprintf("a=%d,b=%d",a,b);
    return 0;
}

void Myprintf(const char* fmt,...)
{
  char s[10];
  va_list ap;
  va_start(ap,fmt);
  vsprintf(s,fmt,ap);
  va_end(ap);
  puts(s);
  puts(s);
}
```

c语言printf、sprintf、vsprintf用法和区别

printf、sprintf、vsprintf 通常用于格式化字符串，通俗来说就是字符串打印或显示格式转换。

printf、sprintf 需要包含 <stdio.h> 头文件，vsprintf 需要包含 <stdarg.h>。

使用下面例程，介绍他们的用法、区别和关系。

我的VS2019在运行c程序时候出现点小问题，提示函数非安全。需要使用sprintf_s、vsprintf_s。

替换后，实验结果有一些不对劲。换了下平台，在树莓派上编译运行，结果正常。以下实验现象均为树莓派运行输出。

使用 printf、sprintf、vsprintf 分别输出 a、b 的值。

- printf:

```
#include <stdio.h>

int main(int argc, char* argv)

{

    int a = 10,b = 20;

    printf("a=%d,b=%d\r\n", a, b);

    return 0;

}
```

输出:

```
a=10,b=20
```

实际运行现象： printf最为简单，只需要添加字符串，和%格式说明，即可输出用户内容。

- sprintf

```
#include <stdio.h>

#include <stdarg.h>

void myPrintf(char* fmt, ...)

{
```

```
    char str[20];

    va_list ap;

    va_start(ap, fmt);

    vsprintf(str, fmt, ap);

    va_end(ap);

    printf(str);
}

int main(int argc, char* argv)

{

    int a = 30,b = 20;

    myPrintf("a=%d,b=%d\r\n", a, b);

    return 0;

}
```

输出：

```
a=30,b=20
```

实际运行现象：

vsprintf使用不确定参数的方式封装，可直接输入数组和未知的参数。从而精简 sprintf需要先创建数组再传入参数的操作。

vsprintf 与 sprintf 功能相似，都是将格式化内容输出到指定数组。不同的是使用方法和用途有些不一样。

看过上面的例程，有没有感觉 sprintf 也能替换 vsprintf，实现相同的功能，何必搞出 va_list、va_end 一堆的麻烦事？？？

那么就来试着替换一下。（实际这种用法是错误的）

```
#include <stdio.h>

void myPrintf(char* fmt, ...)

{

    char str[20];

    sprintf(str, fmt);

    printf(str);

}

int main(int argc, char* argv)

{

    int a = 30,b = 20;

    myPrintf("a=%d,b=%d\r\n", a, b);

    return 0;

}
```

输出：

```
a=20,b=2125550836
```

实际运行现象：

欸！！！有趣的来了，这tm输出的完全不对啊，我输入的明明是30和20，输出的是20和2125550836。

所以 **sprintf 不适用于这种封装的传参，sprintf不能以不确定传参的方式来格式参数。**

而 va_start(ap,fmt) 从实现方式来讲，它使va_list类型变量ap指向被传递给函数的可变参数表中的第一个参数，然后在栈中浏览参数，最后由于va_end(ap)释放掉ap。

最后总结：对于传参中参数类型和个数不确定的格式转换，请使用vsprintf。

typora-自动保存

这样：

- 好消息：以后不会丢了，能够自动保存了，只要恢复一下就好了。
- 坏消息：丢掉了的再找不回了。

Linux Makefile 生成 *.d 依赖文件以及 gcc -M -MF -MP 等相关选项说明

1. 为什么要使用后缀名为 .d 的依赖文件?

在 Makefile 中， 目标文件的依赖关系需要包含一系列的头文件。

比如 main.c 源文件内容如下：

```
#include "stdio.h"
#include "defs.h"

int main(int argc, char *argv[])
{
    printf("Hello, %s!\n", NAME);
    return 0;
}
```

defs.h 头文件内容如下：

```
#ifndef _DEFS_H_
#define _DEFS_H_

#define NAME    "makefile"

#endif /* _DEFS_H_ */
```

那么依赖关系如下（依赖的文件省略了绝对路径）：

```
main.o : main.c stdio.h defs.h ...
```

假设目标文件 main.o 的依赖关系缺少了 defs.h 文件，当 defs.h 文件中的内容改变后，根本不会重新编译目标文件，这是致命的，因为目标文件内部引用了 defs.h 文件中的宏定义。

如果是一个比较大型的工程，我们必需清楚每一个源文件都包含了哪些头文件，并且在加入或删除某些头文件时，也需要一并修改 Makefile，这是一个费力不讨好的工作。

为了避免这种繁重而又容易出错的事情，可以使用 C/C++ 编译器的 “-M” 系列选项，即自动获取源文件中包含的头文件，并生成一个依赖关系。

由编译器自动生成依赖关系，这样做的好处有如下两点：

- 不必手动书写若干目标文件的依赖关系，由编译器自动生成
- 不管是源文件还是头文件有更新，目标文件都会重新编译

2. 参数说明

- -M

  生成文件的依赖关系，同时也把一些标准库的头文件包含了进来。本质是告诉预处理器输出一个适合 make 的规则，用于描述各目标文件的依赖关系。对于每个源文件，预处理器输出 一个 make 规则，该规则的目标项 (target) 是源文件对应的目标文件名，依赖项 (dependency) 是源文件中 “#include” 引用的所有文件，生成的规则可以是单行，但如果太长，就用"换行符续成多行。规则显示在标准输出，不产生预处理过的 C 程序。

  注意：该选项默认打开了 -E 选项， -E 参数的用处是使得编译器在预处理结束时就停止编译。

  例如：

  ```
  gcc -M main.c
  ```

  则在终端上输出如下：

```
 main.o: main.c defs.h \
 /usr/include/stdio.h \
 /usr/include/features.h \

 /usr/include/sys/cdefs.h /usr/include/gnu/stubs.h \

 /usr/lib/gcc-lib/i486-suse-
linux/2.95.3/include/stddef.h \
 /usr/include/bits/types.h \
 /usr/include/bits/pthreadtypes.h \
 /usr/include/_G_config.h /usr/include/wchar.h \
 /usr/include/bits/wchar.h /usr/include/gconv.h \
 /usr/lib/gcc-lib/i486-suse-
linux/2.95.3/include/stdarg.h \
 /usr/include/bits/stdio_lim.h
```

- -MM

  生成文件的依赖关系，和 -M 类似，但不包含标准库的头文件

  例如：

  ```
  gcc -MM main.c
  ```

  则在终端上输出如下：

  ```
  main.o: main.c defs.h
  ```

三个函数的对比

函数	缓冲区	头文件	回显
getchar()	有缓冲区	stdio.h	有回显
getch()	无缓冲区	conio.h	无回显
getche()	无缓冲区	conio.h	有回显