SDL入门教程（一）：

1、一切从零开始

1.1：SDL是什么？

SDL 是 Simple DirectMedia Layer（简易直控媒体层）的缩写。它是一个跨平台的多媒体库，以用于直接控制底层的多媒体硬件的接口。这些多媒体功能包括了音频、键盘和鼠标（事件）、游戏摇杆等。当然，最为重要的是提供了 2D 图形帧缓冲（framebuffer）的接口，以及为 OpenGL 与各种操作系统之间提供了统一的标准接口以实现 3D 图形。从这些属性我们可以看出，SDL 基本上可以认为是为以电脑游戏为核心开发的多媒体库。

SDL 支持主流的操作系统，包括 Windows 和 Linux。在官方的介绍中，我们可以找到它所支持的其他平台。（SDL supports Linux, Windows, Windows CE, BeOS, MacOS, Mac OS X, FreeBSD, NetBSD, OpenBSD, BSD/OS, Solaris, IRIX, and QNX. ）。SDL 本身从 C 语言开发，并且能很好的在 C++等高级语言中使用。在官方可以看到 SDL 所支持的语言很多。（Including Ada, C#, Eiffel, Erlang, Euphoria, Guile, Haskell, Java, Lisp, Lua, ML, Objective C, Pascal, Perl, PHP, Pike, Pliant, Python, Ruby, Smalltalk, and Tcl. ）

SDL 在 GNU LGPL version 2 下发布，这意味着你可以免费的使用。并且可以免费的用于商业软件的制作（只要你直接使用 SDL 的动态链接库，Windows 下的 SDL.dll）。如果你将 SDL 库编译进了自己的二进制代码中，你需要指明你所使用的 SDL 库的版本以及包括你自己修改的源代码，并说明这些代码的原始出处。这是很宽松的法律，你可以用如此强大的多媒体库完全合法的免费开发商业游戏。

1.2：本教程是为谁准备的？

本教程是为电脑游戏制作的发烧友准备的。因为 Linux 的普及以及不受$M 的牵制，SDL 在过去的几年中，成为了跨平台开发 PC 游戏的首选。即使是在 Windows 平台下，SDL 有具有自身的优势。与 MFC 使用不成熟的 C++外表伪封装的 win32api 以及一家独唱推崇的 COM 风格和.net 相比较，SDL 是更纯粹的 C 风格。无论你是喜欢纯 C 还是 OOP 的 C++，你都可以按照你自己喜欢的方式对 SDL 进行再次封装，只要你自己愿意，可以让自己的程序更接近 C/C++的标准风格，让代码更加优美也更加容易阅读。

事实上，在我看来，有了标准 C++（兴奋的期待 C++0x 的发布），SDL 和 OpenGL 这些标准或免费库的支持，完全可以扔掉与 C++标准并不怎么和谐的“Windows 编程”了。没有谁愿意被别人牵着鼻子走，这个道理在软件行业尤其的明显。

1.3：本教程的写作计划？

我是为那些热爱游戏并希望尝试投入到游戏制作中的人们准备的礼物。也就是说，这是一个玩家写给其他玩家的游戏制作教程。我想，这个教程本身不可能简单，但是也绝对不是“技术狂”的风格。我不希望去深挖一些技术细节，我只希望找出整个知识体系中对于游戏制作最有帮助的部分，同时，我也不会避讳那些有助于帮助我们理解和记忆的看起来晦涩的原理。游戏需要技术来实现，但是做游戏显然不是单纯的做技术研究。我会在自己的学习过程中完善这个教程，如果你觉得更新太过于缓慢，可以在网上搜集到许多已经存在的教程。以下这个教程是SDL官方推荐的，我也推荐给你们：

http://lazyfoo.net/SDL_tutorials/index.php

1.4：我目前的计划和心愿。

我是一个游戏策划，我最大的心愿是做出好玩的游戏。所以，显然，我做游戏的动力来自于对于目前存在的所有游戏的满足度不够，以及已经通过多年策划得到的一个方案希望实现出来以接受市场的考验。通过大量的分析和论证，我目前的项目计划集中在设计一款具有角色扮演性质的大型多人在线同盟共时制回合战略游戏。这个项目的第一个计划就是实现类似英雄无敌III的战场战斗效果，这就是我研究SDL的原因。

对于游戏策划方面的想法，大家可以看我的商博，上面也提到了一些考虑中的商业模式。

http://i.cn.yahoo.com/lf426

八卦一点的，可以看我在sina上的主博，评论和牢骚也是不少滴^^

http://blog.sina.com.cn/fleetlong

最后介绍一下我以及曾经的团队UVi Studio。我本人以及之前的团队（2007 年 1 月——2007 年 7 月）UVi Studio致力于网络游戏的市场分析与策划。简单点说，我们希望从事整个网络游戏产业链上处于开始端（市场分析与策划）以及末端（运营）的工作，而将游戏的具体制作外包。做一个不太恰当的比喻，就如同我们希望做网络游戏领域的“开发商”，而非现在大部分公司采用的“开发”“承建”和“运营”全包的商业模式。现在，我们将不仅仅依靠传统的“资本创业”模式，我们也关注技术创业的可行性，并一直在尝试。这种转变体现在形式上，请记住这个名字：UVi Soft (From 2008)。

无论你是对游戏感兴趣的技术人才，艺术家还是看好游戏市场的资方，如果你对这个项目感兴趣，请在第一时间与我们联系。

2、Visual C++ 下的安装与设置

2.1：获得Visual C++ 2008 Express Edition。

使用盗版不是一个好习惯。与其背上贼名，我更愿意放弃华丽的外表，使用朴实无华的免费软件。你可能知道 Linux 和 GNU，但是也许你还不知道$M 也提供一些“免费的午餐”。Visual Studio 的 Edition 版本就是这样的一种便当。作为商业公司的一种商业手段，免费的 VC 也具有一些吸引人的地方，比如漂亮的界面，与 Windows 完美的集成，没有宽度的文本编辑（主要是跟 Vim 作比较），以及自动提醒的类成员（即所谓 Visual）等。你可以在官方免费获得 VC 2008 Express。

http://www.microsoft.com/express/download/

2.2：获得VC下的SDL开发库。（Development Libraries）

你可以在 SDL 的官方主页找到下载。

http://www.libsdl.org/download-1.2.php

目前的 VC 版本，你会发现……找不到 VC9 的版本。无所谓了，SDL-devel-1.2.13-VC8.zip (Visual C++ 2005 Service Pack 1)可以正常使用的。

2.3：为VC设置SDL库。

解压缩 SDL-devel-1.2.13-VC8.zip，docs 里面包含了官方文档，这将是你学习 SDL 的主要参考资料。找到你在硬盘上安装 VC 的位置，类似：

C:\Program Files\Microsoft Visual Studio 9.0\VC

打开 include 文件夹，在里面建立一个新文件夹，取名为 SDL，打开这个新的文件夹：

C:\Program Files\Microsoft Visual Studio 9.0\VC\include\SDL

然后，将 SDL Development Libraries 中 include 文件夹里面的文件全部拷贝到刚才建立起来的那个新文件夹中。

然后，回到 VC 的.\VC 文件夹下，打开 lib 文件夹：

C:\Program Files\Microsoft Visual Studio 9.0\VC\lib

将 SDL Development Libraries 中 lib 文件夹下的 SDL.lib 和 SDLmain.lib 两个文件拷贝到刚才的那个 VC 的 lib 文件夹下。

最后，把 SDL Development Libraries 中 lib 文件夹下的 SDL.dll 文件拷贝到

C:\WINDOWS\system32

下。当然，另外一个选择是将 SDL.dll 随时绑定到使用了 SDL 库编译的*.exe 文件所在的文件夹中。这通常在发布你的程序的时候使用。

2.4：建立VC下使用SDL库的工程。

1) 启动 Microsoft Visual C++ 2008 Express Edition；
2) File -- New -- Project
3) Project types: Visual C++: Win32 （默认的）
    Templates: Visual Studio installed templates: Win32 Console Application
    Name: <Enter_name> （为你的工程起个名字）
4) Win32 Application Wizard -- Application Settings
    Application type: Console application
    Additional options: Empty project
5) 现在添加一个 C++源文件。右击 Source Files，Add -- New Items...
    选择 Categories 为 Visual C++的 Code；
    Templates 为 Visual Studio installed templates 的 C++ File(.cpp)；
    为源文件起个名字。
6) 设置 Project。Project -- project_name Properties...(Alt+F7)
    Configuration Properties -- C/C++ -- Code Generation -- Runtime Library:
    修改为：Multi-threaded DLL (/MD)
    Configuration Properties -- Linker -- Input -- Additional Dependencies:
    添加：SDL.lib SDLmain.lib
    Configuration Properties -- Linker -- System -- SubSystem
    如果是 Debug 版本，建议不改变原先的 Console (/SUBSYSTEM:CONSOLE)，
    因为调试信息可以出现在 console 的窗口里面。
    Release 可以修改为 Windows (/SUBSYSTEM:WINDOWS)

2.5：第一个使用SDL的C++程序。

将下面代码输入到刚才建立的 cpp 文件中：

```
#include <iostream>
#include "SDL/SDL.h"

int main(int argc, char* argv[])
{
    try {
        if ( SDL_Init(SDL_INIT_EVERYTHING) == -1 )
            throw "Could not initialize SDL!";
    }
    catch ( const char* s )
    {
        std::cerr << s << std::endl;
        return -1;
```

```
    }
    std::cout << "SDL initialized.\n";
    SDL_Quit();

    return 0;
}
```

Build -- Build Solution

Debug -- Start Without Debugging

你就可以看到你的第一个 SDL 程序的运行情况了。

3、MinGW 下的安装与设置

3.1：MinGW 是什么？

MinGW 提供了一套简单方便的Windows下的基于GCC 程序开发环境。MinGW 收集了一系列免费的Windows 使用的头文件和库文件；同时整合了GNU(http://www.gnu.org/)的工具集，特别是GNU 程序开发工具，如经典gcc, g++, make等。MinGW是完全免费的自由软件，它在Windows平台上模拟了Linux下GCC的开发环境，为C++的跨平台开发提供了良好基础支持，为了在Windows下工作的程序员熟悉Linux下的C++工程组织提供了条件。

3.2：为什么使用MinGW？

因为我和很多 IT 人士一样，不愿意吊死在$M 这棵树上。且不说开源这个理念的伟大，仅仅就 Windows 的安全性也已经让人们很不安心了。如果再深入一点从编写程序来看，在C++标准尚不成熟时期产生的 MFC 今天看来基本上就是盖子大叔凭着个人（或许是$M 公司^^）理解所使用的 C++方言。在我们期盼 C++0x 时代到来之前，没有任何实际行动能比严格遵循 C++的 ISO 标准更能表达我们对于 C++这个伟大的哲学体系的尊重。从这个角度来说，遵循 ISO 标准的实现就是好的实现。所以，MinGW 不会比 VC 差，况且，他既是开源软件家族中的一员，也是*nix 平台下事实标准 GCC 的翻版。如果有一天我们开始使用 Linux 平台了，会因为今天透过 MinGW 所了解的 GCC 知识而受益。

3.3：MinGW 的下载和安装。

MinGW的官方主页是：

http://www.mingw.org/

你很容易找到下载的链接。下载文件是放在sourceforge上的，这也表明MinGW本身也是个在开发中的项目。

http://sourceforge.net/project/showfiles.php?group_id=2435

下载页面上有很多令人眼花缭乱的文件，不用害怕，因为这才是软件最真实的一面。当然，这个软件的提供者为普通使用者也提供了最为简单的下载和安装方式。你只需要下载

“MinGW自动下载和安装软件”（Package: Automated MinGW Installer; Release: MinGW-5.1.3）MinGW-5.1.3.exe就可以根据提示简单的安装了。并且，安装后的MinGW甚至可以通过这个文件来自动升级，非常的人性化。

安装时需要注意以下几个方面：

1) 建议选择Current版本，Previous表示以前的版本；Candidate表示当前正在开发的版本，可能正在测试中，还不是非常稳定。

2) 在选择安装模块的时候，根据我们的需要，只需要选择以下三个：MinGW base tools, g++ compiler, MinGW Make。

3) 安装路径请选择一个容易找到的地方，默认的C:\MinGW 是不错的选择。

3.4：MinGW的简单使用。

打开“命令提示符”（console）窗口，我们可以简单的把目录转到比如 C:\MinGW\bin 这样的位置去工作。但是这显然不是一个好的方式，因为这样一定会把 bin 里面搞得很混乱。所以，我们在另外一个“干净”的地方，比如 D:\Project ，把这里设置成 MinGW 的工程工作区。这样，我们需要在系统的 path 里面添加能找到 bin 下面*.exe 文件的路径，所以做一个批处理文件是最简单的方法。我的 startg++.bat 文件这样写的：

```
@set path=C:\mingw\bin;%PATH%;
@echo ---**欢迎使用 MinGW(GCC)系统**---
@echo Update by lf426 ( E-mail: zbln426@163.com ) 2007-12-10
@cmd
```

然后，你可以建立一个该文件的快捷方式放在桌面上，再为这个快捷方式指定一个漂亮的图标，nice!

接下来，我们需要建立一个 cpp 文件，用什么来写 cpp 呢？

3.4.1：（插播）Vim ??

这又是一个有着传统的“悠久”历史，很好很强大的，免费开源的，牛x文本编辑器。官方主页是：

http://www.vim.org/

不用怀疑，你可以很容易的下载Vim的安装程序，并且很容易的安装，不需要任何的“破解”-_-!!

因为Vim很好很强大，所以，太多的内容就留给包括我自己在内的大家自己去学了（我的逻辑Orz）。我们这里简单的只需要了解如下几点：

1) 建立一个名为 abc.def 的文本文件，在console窗口下输入：

```
vim abc.def
```

你就进入了vim的编辑界面。

2) 按"i"键，就可以输入文本了。退回到编辑方式，按Esc，再进入插入方式，再按i，就这样...

3) vim在编辑方式下本来的光标移动是h, j, k, l，不过，上下左右箭头也是可以使用的。

4) 编辑完成后，（编辑方式下）按Shift+Z两次，或者:wq就可以保存退出。使用:q!可以不保存强制退出。

3.4.2：MinGW的简单使用。（续）

继续刚才的话题，我们建立一个叫 hello.cpp 的文件（据说 Hello World 是程序员的咒语...）

```
#include <iostream>

int main(int argc, char* argv[])
{
   std::cout << "Hello World." << std::endl;
   return 0;
}
```

我们可以简单的使用命令：g++ hello.cpp，与最传统的 Unix 系统下的 C 编译器一样，MinGW 为我们生成了一个名为 a 的可执行文件，当然，在 win32 下，名字叫做 a.exe。

可以使用参数 -o 来指定编译的可执行文件的名字：g++ -o hello.exe hello.cpp （或者也可以不写名 exe，直接 g++ -o hello hello.cpp 结果一样）。这样，得到的文件就是 hello.exe。

可以使用参数 -c 来生成与源文件名相同的目标文件：g++ -c hello.cpp ，得到的文件是 hello.o 。

目标文件可以继续连接成可执行文件：g++ hello.o ，得到文件 a.exe 。

多个源文件，目标文件和库文件都可以同时编译和连接。

3.5：获得MinGW下的SDL开发库。（Development Libraries）

下载页面：http://www.libsdl.org/download-1.2.php

下载：SDL-devel-1.2.13-mingw32.tar.gz (Mingw32)

3.6：为MinGW设置SDL库。

找到你在硬盘上安装 MinGW 的位置，类似：

C:\MinGW

打开 include 文件夹，在里面建立一个新文件夹，取名为 SDL，打开这个新的文件夹：

C:\MinGW\include\SDL

然后，将 SDL Development Libraries 中 include 文件夹里面的文件全部拷贝到刚才建立起来的那个新文件夹中。

然后，回到 MinGW 的文件夹下，打开 lib 文件夹：

C:\MinGW\lib

将 SDL Development Libraries 中 lib 文件夹下的 libSDL.dll.a, libSDL.la 和 libSDLmain.a 三个文件拷贝到刚才的那个 MinGW 的 lib 文件夹下。

最后，把 SDL Development Libraries 中 lib 文件夹下的 SDL.dll 文件拷贝到

C:\WINDOWS\system32

下。当然，另外一个选择是将 SDL.dll 随时绑定到使用了 SDL 库编译的*.exe 文件所在的文

件夹中。这通常在发布你的程序的时候使用。

3.7：建立第一个SDL程序。

运行我们之前的那个 startg++.bat 文件，打开 console 窗口，执行命令：vim main.cpp ，将下列代码写入并保存。

```
#include <iostream>
#include "SDL/SDL.h"

int main(int argc, char* argv[])
{
   try {
      if ( SDL_Init(SDL_INIT_EVERYTHING) == -1 )
         throw "Could not initialize SDL!";
   }
   catch ( const char* s ) {
      std::cerr << s << std::endl;
      return -1;
   }
   std::cout << "SDL initialized.\n";
   SDL_Quit();

   return 0;
}
```

执行 g++命令：g++ -o MySDL.exe main.cpp -lmingw32 -lSDLmain -lSDL

参数 -l （小写的 L）使库文件得以与源文件一起编译和连接。（简单的可以认为-labc 即连接了名称符合 libabc.*的所有库文件）

最后说明的两个参数是：-mconsole 和 -mwindows

他们的效果是使可执行程序是带命令行控制窗口的或者是纯 Windows 窗口的，缺省情况下应该是带命令行控制窗口的。

运行 mysdl.exe 可以看到文件的执行情况。

SDL入门教程（二）：

1、SDL的装载，位标（flags）

1.1：SDL_Init！

我们回顾一下前面的那个小程序，所使用到的第一个 SDL 函数是：SDL_Init(SDL_INIT_EVERYTHING);

这是 SDL 的装载函数，也就是说，SDL 的其他函数必须在这个函数将 SDL 装载之后才能够正常调用。我们看一下函数原形：

int SDL_Init(Uint32 flags);

它的返回值是 int，这个我们很熟悉。如果 SDL 成功初始化装载，函数返回 0，如果异常，则返回-1。接下来，这个函数的形参列表不属于标准 C++的内容。

1.1.1：Uint32 是什么？

根据 Uint32 在函数原形中的位置，我们可以猜想到它是一种类类型的名字。因为 SDL 是跨平台的，而且还支持多种计算机语言，当我们在 Windows 的 VC 编译器里面简单的通过 sizeof(int)可以看到 int 类型占用 4 个字节（32 位）的时候，我们并不清楚其他平台，其他编译器和其他语言里面对 int 的大小是如何定义的。所以，为了让大家在任何情况下都能比较直观的阅读类类型的大小，SDL 使用了一个简单的约定：U 就是 unsigned（无符号，意味着非负）的意思，与之对应的 S 代表 signed（有符号，可正可负）；int 还是整数的意思；32 表示占 32 位，类似的，还可以占 1 至 4 个字节，即 8，16，32，64 位。Uint32 的意思就是无符号的，占 32 位的整数类。

1.1.2：flag 是什么意思？

flag 就是旗帜……别 k 我……多想想，旗帜可用于什么呢？旗帜可用于发信号和标记。在计算机里，通常把 flag 叫做位标——其实，bit flag 才是位标，不过这种用法貌似已经是一种习惯。所以，从 flags 字面，我们至少可以理解到三层含义-_-!!!

1) 关键字“位”，这意味着我们需要用二进制的观点看这个值，并且，这个值可以参与位运算；

2) 关键字“标”，这显然是标记的意思。小狗通常采用一些不文明的手段表明某个区域是自己的势力范围，而人类则通常采用插一面代表自己势力的旗帜，所以这就是 flag 的本意。

3) 关键字"s"，这里使用了复数，表明我们可以不止插一面旗。

我们看看这个函数的位标的定义：（就在 SDL.h 文件中）

```
#define    SDL_INIT_TIMER       0x00000001
#define SDL_INIT_AUDIO       0x00000010
#define SDL_INIT_VIDEO       0x00000020
```

```
#define SDL_INIT_CDROM        0x00000100
#define SDL_INIT_JOYSTICK     0x00000200
#define SDL_INIT_EVERYTHING   0x0000FFFF
```

我们通过最常可能用到的 audio 和 video 来简单说明下。SDL_INIT_AUDIO 和 SDL_INIT_VIDEO 显然实际上只用到了 2 个字节，为了少写 8*6 个无谓的 0，我们就简单的看成是 0x10 和 0x20，写成 2 进制，则

SDL_INIT_AUDIO = 0001 0000

SDL_INIT_VIDEO = 0010 0000

位运算或（|）的结果是若两数相对应的位，有一个是 1 则得 1。（与（&）则是两数相对应的位全是 1 才得 1。）

所以，SDL_INIT_AUDIO | SDL_INIT_VIDEO 的结果是 0011 0000，即 0x30（这里要是一不小心觉得貌似 1+2=3，第一，理解错了；第二，是个巧合）。

1.2：SDL_WasInit?

当我们坐上公共汽车，我们真正关心的问题，是这车是开还是停。同样，当 SDL 已经装载，我们更关心的问题是它的运行状态，并且，有哪些“旗子”插在了上面？所以，我们可以问问 SDL：xx_flag was init?

```
int SDL_WasInit(Uint32 flags);
```

这里，我们可以将 SDL_INIT_*的 5 个具体旗子看成一种用法；使用了“|”的复合旗子以及那个 EVERYTHING 的概念看成另外一种用法。当 flag 为某个具体旗子的时候，如果该旗子插上了，则返回该旗子的位标值本身，否则就返回 0。当参数不止一面旗子的时候，则返回这些旗子中插上了的那部分的“|”值。（其实就是返回那些在你指定的旗子中插上的那些，只是电脑看起来很直观，人看起来很不直观——特别在使用非 2 进制表示的时候）。

官方文档里面举了 3 个例子，后面我将举一个我个人觉得比较直观的程序例子。

```
/* Get init data on all the subsystems */
Uint32 subsystem_init;

subsystem_init=SDL_WasInit(SDL_INIT_EVERYTHING);

if(subsystem_init&SDL_INIT_VIDEO)
  printf("Video is initialized.\n");
else
  printf("Video is not initialized.\n");

/* Just check for one specific subsystem */

if(SDL_WasInit(SDL_INIT_VIDEO)!=0)
  printf("Video is initialized.\n");
else
  printf("Video is not initialized.\n");

/* Check for two subsystems */
```

```
Uint32 subsystem_mask=SDL_INIT_VIDEO|SDL_INIT_AUDIO;

if(SDL_WasInit(subsystem_mask)==subsystem_mask)
  printf("Video and Audio initialized.\n");
else
  printf("Video and Audio not initialized.\n");
```

1.3：一段用于演示flags的程序。

需要补充说明的一点是：当 SDL_Init(SDL_INIT_EVERYTHING); 的时候，SDL_WasInit(SDL_INIT_EVERYTHING) 返回的并非 SDL_INIT_EVERYTHING 的原值 0xFFFF，而是 5 个基本旗子的“|”和，即 0x0331，所以，程序里面我定义了一个 const Uint32 EVERYTHING = 0x331;，来正确的反应 EVERYTHING 的实际情况。

```
//////////////////////////////
//本程序用于演示 SDL_Init()函数的 flags
//有关 SDL 的信息请访问 SDL 的官方网站
//http://www.libsdl.org/
//任何疑问和建议请联系我 zbln426@163.com
//再别流年的技术实验室
//http://www.cppblog.com/lf426/
//////////////////////////////

//////////////////////////////
//<iomanip>包含了
//setw(n) 用于设置下次输出的字宽
//setfill(ch) 用于将字宽多出部分用 char ch 填充
//////////////////////////////
#include <iostream>
#include <iomanip>
#include "SDL/SDL.h"
using namespace std;

inline void showHex(int SDLflags);
void testSDLflags(Uint32 SDLflags, char* inf);

int main(int argc, char* argv[])
{
   cout << "*****flags*****" << endl;
   cout << "SDL_INIT_EVERYTHING = ";
   showHex(SDL_INIT_EVERYTHING);
   cout << "SDL_INIT_VIDEO = ";
   showHex(SDL_INIT_VIDEO);
   cout << "SDL_INIT_AUDIO = ";
```

```
    showHex(SDL_INIT_AUDIO);
    cout << "SDL_INIT_TIMER = ";
    showHex(SDL_INIT_TIMER);
    cout << "SDL_INIT_CDROM = ";
    showHex(SDL_INIT_CDROM);
    cout << "SDL_INIT_JOYSTICK = ";
    showHex(SDL_INIT_JOYSTICK);
    cout << endl << endl;

    testSDLflags(SDL_INIT_EVERYTHING, "SDL_INIT_EVERYTHING");
    testSDLflags(SDL_INIT_VIDEO, "SDL_INIT_VIDEO");
    testSDLflags(SDL_INIT_AUDIO, "SDL_INIT_AUDIO");
    testSDLflags(SDL_INIT_VIDEO | SDL_INIT_AUDIO, "SDL_INIT_VIDEO | SDL_INIT_AU
DIO");
    testSDLflags(SDL_INIT_VIDEO | SDL_INIT_AUDIO | SDL_INIT_TIMER | SDL_INIT_CD
ROM | SDL_INIT_JOYSTICK,
            "SDL_INIT_VIDEO | SDL_INIT_AUDIO | SDL_INIT_TIMER | SDL_INIT_CDROM
| SDL_INIT_JOYSTICK");
    return 0;
}

//////////////////////
//该函数用于测试不同的 flags 所产生的效果
//////////////////////
void testSDLflags(Uint32 SDLflags, char* inf)
{
    Uint32 subsystem_init = SDLflags;
    const Uint32 EVERYTHING = 0x331;
    SDL_Init(subsystem_init);

    cout << "SDL_Init(" << inf << ") Loading…" <<endl;


    cout << "*****runtime*****" << endl;
    cout << "SDL_WasInit(SDL_INIT_EVERYTHING) = ";
    showHex(SDL_WasInit(SDL_INIT_EVERYTHING));
    cout << "SDL_WasInit(SDL_INIT_VIDEO) = ";
    showHex(SDL_WasInit(SDL_INIT_VIDEO));
    cout << "SDL_WasInit(SDL_INIT_AUDIO | SDL_INIT_VIDEO) = ";
    showHex(SDL_WasInit(SDL_INIT_AUDIO | SDL_INIT_VIDEO));

    cout << "*****runtime in bool*****" << endl;
    cout << boolalpha;
    Uint32 runtimeThing = SDL_WasInit(SDL_INIT_EVERYTHING);
    cout << "SDL_INIT_EVERYRHING? " << (runtimeThing == EVERYTHING) << endl;
```

```
    cout << "SDL_INIT_VIDEO? " << bool(runtimeThing & SDL_INIT_VIDEO) << endl;
    cout << "SDL_INIT_AUDIO? " << bool(runtimeThing & SDL_INIT_AUDIO) << endl;
    cout << noboolalpha;

    cout << "SDL_Quit…\n\n\n";


    SDL_Quit();

    return;
}

//////////////////////////////
//该函数用于将 flags 打印为 16 进的格式
//////////////////////////////
void showHex(int SDLflags)
{
    cout << hex;
    cout << "0x" << setw(8) << setfill('0') << SDLflags << endl;
    cout << dec;
}
```

运行结果如下：

```
*****flags*****
SDL_INIT_EVERYTHING = 0x0000ffff
SDL_INIT_VIDEO = 0x00000020
SDL_INIT_AUDIO = 0x00000010
SDL_INIT_TIMER = 0x00000001
SDL_INIT_CDROM = 0x00000100
SDL_INIT_JOYSTICK = 0x00000200



SDL_Init(SDL_INIT_EVERYTHING) Loading…
*****runtime*****
SDL_WasInit(SDL_INIT_EVERYTHING) = 0x00000331
SDL_WasInit(SDL_INIT_VIDEO) = 0x00000020
SDL_WasInit(SDL_INIT_AUDIO | SDL_INIT_VIDEO) = 0x00000030
*****runtime in bool*****
SDL_INIT_EVERYRHING? true
SDL_INIT_VIDEO? true
SDL_INIT_AUDIO? true

SDL_Quit…
```

SDL_Init(SDL_INIT_VIDEO) Loading…

*****runtime*****
SDL_WasInit(SDL_INIT_EVERYTHING) = 0x00000020
SDL_WasInit(SDL_INIT_VIDEO) = 0x00000020
SDL_WasInit(SDL_INIT_AUDIO | SDL_INIT_VIDEO) = 0x00000020
*****runtime in bool*****
SDL_INIT_EVERYRHING? false
SDL_INIT_VIDEO? true
SDL_INIT_AUDIO? false

SDL_Quit…

SDL_Init(SDL_INIT_AUDIO) Loading…

*****runtime*****
SDL_WasInit(SDL_INIT_EVERYTHING) = 0x00000010
SDL_WasInit(SDL_INIT_VIDEO) = 0x00000000
SDL_WasInit(SDL_INIT_AUDIO | SDL_INIT_VIDEO) = 0x00000010
*****runtime in bool*****
SDL_INIT_EVERYRHING? false
SDL_INIT_VIDEO? false
SDL_INIT_AUDIO? true

SDL_Quit…

SDL_Init(SDL_INIT_VIDEO | SDL_INIT_AUDIO) Loading…

*****runtime*****
SDL_WasInit(SDL_INIT_EVERYTHING) = 0x00000030
SDL_WasInit(SDL_INIT_VIDEO) = 0x00000020
SDL_WasInit(SDL_INIT_AUDIO | SDL_INIT_VIDEO) = 0x00000030
*****runtime in bool*****
SDL_INIT_EVERYRHING? false
SDL_INIT_VIDEO? true
SDL_INIT_AUDIO? true

SDL_Quit…

SDL_Init(SDL_INIT_VIDEO | SDL_INIT_AUDIO | SDL_INIT_TIMER | SDL_INIT_CDROM |

SDL

_INIT_JOYSTICK) Loading…

*****runtime*****
SDL_WasInit(SDL_INIT_EVERYTHING) = 0x00000331
SDL_WasInit(SDL_INIT_VIDEO) = 0x00000020
SDL_WasInit(SDL_INIT_AUDIO | SDL_INIT_VIDEO) = 0x00000030
*****runtime in bool*****
SDL_INIT_EVERYRHING? true
SDL_INIT_VIDEO? true
SDL_INIT_AUDIO? true

SDL_Quit…

1.4：一段用于演示flags二进制的演示程序

```
#include <iostream>
#include <bitset>
#include "SDL/SDL.h"
using namespace std;

int main(int argc, char* argv[])
{
   cout << "SDL_INIT_EVERYTHING = " << bitset<32>(SDL_INIT_EVERYTHING) << endl;
   cout << "SDL_INIT_VIDEO = " << bitset<32>(SDL_INIT_VIDEO) << endl;
   cout << "SDL_INIT_AUDIO = " << bitset<32>(SDL_INIT_AUDIO) << endl;
   cout << "SDL_INIT_VIDEO | SDL_INIT_AUDIO = " << bitset<32>(SDL_INIT_VIDEO | SD
L_INIT_AUDIO) << endl;
   return 0;
}
```

运行结果如下：

```
SDL_INIT_EVERYTHING = 00000000000000001111111111111111
SDL_INIT_VIDEO = 00000000000000000000000000100000
SDL_INIT_AUDIO = 00000000000000000000000000010000
SDL_INIT_VIDEO | SDL_INIT_AUDIO = 00000000000000000000000000110000
```

2、SDL的安全装载与安全退出

2.1：WasInit可以在Init前使用吗？

我们在介绍 SDL_Init()的时候说过，这个函数是调用 SDL 其他函数之前必须首先调用的函数。但是有一个问题是，我们如何知道 SDL_Init 是否已经被调用过了呢？于是我们接着

认识了 SDL_WasInit()，这个函数可以返回 Init 的状态。如果 WasInit 用在了 Init 之前会出现什么问题，或者说，WasInit 可以在 Init 前使用吗？

这个问题看起来似乎是逻辑混乱，但是事实上，有疑问向电脑问清楚是个好习惯。通过简单的试验，我们可以证实以下结论：SDL_WasInit()可以在 SDL_Init()之前调用，返回值是 0。这很合乎逻辑，没插旗子当然就没旗子，不是吗？

合乎正常的思维方式是：如果你准备穿西装，你最好应该弄清楚你自己身上是不是已经穿了一件西装（这个蹩脚的比喻主要是为了后面那个比喻做准备-_-!!）。所以，我们装载 SDL 之前，最好弄清楚 SDL 是不是已经装载了。结合前面的讨论和试验，我们写一个安全装载 SDL 的函数。

```
void beginSDL()
{
   if ( SDL_WasInit(SDL_INIT_EVERYTHING) != 0 ) {
      std::cout << "SDL is already running.\n";
      return;
   }
   if ( SDL_Init(SDL_INIT_EVERYTHING) == -1 ) {
      throw "Unable to init SDL!";
   }
   std::cout << "SDL is running successfully.\n";
   return;
}
```

2.2：atexit();

我们使用 C++，当然知道<iostream>很好很强大，因为它已经包含了很多 C 的标准库。其中，atexit()是个用起来很方便的函数，它的形参是个函数指针，也就是个函数名，当然，这个被调用的函数需要满足返回值类型是 viod，并且没有形参或者不能接受形参，即：void function();或者 void function(viod);。atexit(function)在 main()正常结束以及异常结束的时候，都会自动调用 function()函数，所以，在我们不可预知程序是否会异常，或者不清楚程序什么时候结束的时候，需要做某些善后工作，atexit()是个很好的选择。

2.3：SDL的退出。

我们前面已经用到了 SDL 的退出函数，其原形是：

```
void SDL_Quit(void);
```

通常，我们在可以预测程序结束的时候直接使用语句：SDL_Quit();

更多的时候，以及 SDL 官方文档里面，也推荐使用：atexit(SDL_Quit);

2.4：SDL_Quit();的容错性。

还是之前的那个问题的翻版，Quit 可以在 Init 之前使用吗？答案依旧是肯定的。SDL_Quit();可以任意调用任意次，甚至不需要 SDL_Init();的先行存在。

这是一种计算机对程序员的包容，但是这种包容并非总是好事情。一个逻辑上的正常思维应该是：你准备脱西装的时候最好弄明白自己身上到底是不是穿着西装。所以，我们在准备退出 SDL 的时候，最好也弄明白 SDL 是不是已经装载了。一个安全退出 SDL 的函数如下。

```
void endSDL()
{
   if ( SDL_WasInit(SDL_INIT_EVERYTHING) == 0 ) {
      std::cout << "SDL was NOT running!\n";
      return;
   }
   SDL_Quit();
   std::cout << "SDL_Quit successfully.\n";
   return;
}
```

2.5：一段用于演示SDL安全装载与安全退出的程序。

```
//////////////////
//File Name: test04.h
//SDL 的安全装载与退出：头文件
//联系我： znln426@163.com
//再别流年的技术实验室
//http://www.cppblog.com/lf426/
//////////////////

#include <iostream>
#include "SDL/SDL.h"

void beginSDL();
void endSDL();

//////////////////
//File Name: test04.cpp
//SDL 的安全装载与退出：实现文件
//联系我： znln426@163.com
//再别流年的技术实验室
//http://www.cppblog.com/lf426/
//////////////////

#include "test04.h"

void beginSDL()
{
   if ( SDL_WasInit(SDL_INIT_EVERYTHING) != 0 ) {
```

```
        std::cout << "SDL is already running.\n";
        return;
    }
    if ( SDL_Init(SDL_INIT_EVERYTHING) == -1 ) {
        throw "Unable to init SDL!";
    }
    std::cout << "SDL is running successfully.\n";
    return;
}

void endSDL()
{
    if ( SDL_WasInit(SDL_INIT_EVERYTHING) == 0 ) {
        std::cout << "SDL was NOT running!\n";
        return;
    }
    SDL_Quit();
    std::cout << "SDL_Quit successfully.\n";
    return;
}

//////////////////
//File Name: main.cpp
//SDL 的安全装载与退出：演示程序
//联系我： znln426@163.com
//再别流年的技术实验室
//http://www.cppblog.com/lf426/
//////////////////

#include "test04.h"

int main(int argc, char* argv[])
{
    try {
        beginSDL();
    }
    catch ( const char* s ) {
        std::cout << s << std::endl;
        return -1;
    }
    std::cout << SDL_WasInit(SDL_INIT_EVERYTHING) << std::endl;

    atexit(endSDL);
```

```
    return 0;
}
```

3、中途装载与退出，SDL错误信息

3.1：SDL flags 的中途装载和中途退出。

如果我们在程序一开始只启动了 video，在运行的某个期间需要启动 audio，之后，又需要关掉 video 只保持 audio 听声音应该怎么做呢？

先说中途装载。我们首先想到的，还是通过 SDL_Init();来装载 audio。这看似合理的，而且通过本人试验，发现事实上也是可行的。但是，我们前面分析过一个逻辑，就是一个程序最好只装载一次 Init，这样更容易看出来 SDL 是从什么时候开始工作的。那么，为了表明之后的装载是在 SDL 某些 flags 已经装载之后的中途装载，SDL 提供了一个基本上类似的函数：

```
int SDL_InitSubSystem(Uint32 flags);
```

与 Init 一样，返回值为 0 则成功装载，-1 则失败。

中途退出某些 flags 的问题，我们显然就不能用 SDL_Quit();了，因为这是所有 flags 全退。SDL 提供了中途退出某些 flags 的函数：

```
void SDL_QuitSubSystem(Uint32 flags);
```

3.2：一段用于演示SDL中途装载和中途退出某些flags的程序。

```
//////////////////////
//SDL 中途装载和中途退出的演示程序
//联系我： znln426@163.com
//再别流年的技术实验室
//http://www.cppblog.com/lf426/
//////////////////

#include <iostream>
#include <iomanip>
#include "SDL/SDL.h"
using namespace std;

void showHex(int SDLflags);
void showBool();

int main(int argc, char* argv[])
{
    cout << "*****flags*****" << endl;
```

```
    cout << "SDL_INIT_EVERYTHING = ";
    showHex(SDL_INIT_EVERYTHING);
    cout << "SDL_INIT_VIDEO = ";
    showHex(SDL_INIT_VIDEO);
    cout << "SDL_INIT_AUDIO = ";
    showHex(SDL_INIT_AUDIO);
    cout << "SDL_INIT_TIMER = ";
    showHex(SDL_INIT_TIMER);
    cout << "SDL_INIT_CDROM = ";
    showHex(SDL_INIT_CDROM);
    cout << "SDL_INIT_JOYSTICK = ";
    showHex(SDL_INIT_JOYSTICK);
    cout << endl << endl;

    cout << "*****runtime*****\n\n";
    SDL_Init(SDL_INIT_VIDEO);

    cout << "SDL_Init(SDL_INIT_VIDEO) calling…" <<endl;

    showBool();
    SDL_InitSubSystem(SDL_INIT_AUDIO);

    cout << "SDL_InitSubSystem(SDL_INIT_AUDIO) calling…" <<endl;

    showBool();
    SDL_QuitSubSystem(SDL_INIT_VIDEO);

    cout << "SDL_QuitSubSystem(SDL_INIT_VIDEO) calling…" <<endl;

    showBool();
    SDL_Quit();

    cout << "SDL_Quit() calling…" <<endl;

    showBool();

    return 0;
}

void showBool()
{
    cout << "*****runtime in bool*****" << endl;
    cout << boolalpha;
    Uint32 runtimeThing = SDL_WasInit(SDL_INIT_EVERYTHING);
    cout << "SDL_WasInit(SDL_INIT_EVERYTHING) = ";
    showHex(runtimeThing);
    cout << "SDL_INIT_VIDEO? " << bool(runtimeThing & SDL_INIT_VIDEO) << endl;
    cout << "SDL_INIT_AUDIO? " << bool(runtimeThing & SDL_INIT_AUDIO) << endl;
```

```
    cout << noboolalpha;
    cout << "\n\n";
}

void showHex(int SDLflags)
{
    cout << hex;
    cout << "0x" << setw(8) << setfill('0') << SDLflags << endl;
    cout << dec;
}
```

演示程序的执行结果：

```
*****flags*****
SDL_INIT_EVERYTHING = 0x0000ffff
SDL_INIT_VIDEO = 0x00000020
SDL_INIT_AUDIO = 0x00000010
SDL_INIT_TIMER = 0x00000001
SDL_INIT_CDROM = 0x00000100
SDL_INIT_JOYSTICK = 0x00000200


*****runtime*****


SDL_Init(SDL_INIT_VIDEO) calling…
*****runtime in bool*****
SDL_WasInit(SDL_INIT_EVERYTHING) = 0x00000020
SDL_INIT_VIDEO? true
SDL_INIT_AUDIO? false



SDL_InitSubSystem(SDL_INIT_AUDIO) calling…
*****runtime in bool*****
SDL_WasInit(SDL_INIT_EVERYTHING) = 0x00000030
SDL_INIT_VIDEO? true
SDL_INIT_AUDIO? true



SDL_QuitSubSystem(SDL_INIT_VIDEO) calling…
*****runtime in bool*****
SDL_WasInit(SDL_INIT_EVERYTHING) = 0x00000010
SDL_INIT_VIDEO? false
```

SDL_INIT_AUDIO? true

SDL_Quit() calling…

*****runtime in bool*****
SDL_WasInit(SDL_INIT_EVERYTHING) = 0x00000000
SDL_INIT_VIDEO? false
SDL_INIT_AUDIO? false

3.3：SDL的错误信息。

在前面的例子中，我们自己提供了出现异常的错误信息，实际上，SDL 也提供了自己的错误信息返回函数：

```
char *SDL_GetError(void);
```

在官方文档中提供的示范例子是 C 风格的表述：

```
if (SDL_Init(SDL_INIT_EVERYTHING) == -1)
{
   printf("SDL_Init Failed: %s\n", SDL_GetError());
   // Unrecoverable error, exit here.
}
```

值得注意的是，这个函数的返回值是 C 风格字符串 char* ，当我们使用 C++风格的异常处理的时候，如果

```
throw SDL_GetError();
```

那么，应该 catch 一个 char*对象

```
catch ( const char* s ) {…}
```

当我们并不是很熟悉 SDL 内置的错误信息的时候，自己写异常信息是更值得推荐的选择。

最后介绍的函数是：

```
void SDL_ClearError(void);
```

它将清除所有错误信息。这用于清除掉你已经处理过的错误信息——否则即使没有异常，你仍然可能收到上次异常的错误信息。

SDL入门教程（三）：

1、如何实现按下ESC退出程序？

1.1：游戏中的退出习惯。

如同我们经常遇到的游戏，一般想退出的时候，我们会习惯性的按下 ESC——即使游戏不会马上退出，也一般会调出一个带有退出选择的菜单。我们希望修房子的时候，最好先计划在哪里修门，所以，我认为应该优先掌握“退出游戏”的方法。简单的说，我们启动了一个 SDL 程序，我们希望按下 ESC 就能退出，怎么实现？

1.2：事件（event）查询初探。

在计算机科学领域，隐喻无处不见。所有的抽象概念，若不是被很好的用形象概念或者已经被理解的抽象概念去解释，其本身很难让人们明白是什么。事件，在这里指的就是计算机所直接感知到的玩家对于其的作用。比如你按下某个键，又松开，移动了鼠标等等。所有的这些行为都被称为事件。在计算机看来，任何事件的发生都是有先后的（如果你了解相对论，你就会知道其实世界上任意两点间并不存在“同时”的概念）。如果计算机工作效率很低，这些事件就会排着队列等待接受处理——这里又用到一个模型的隐喻——队列（queue），这是计算机算法与数据结构知识中很重要的概念，往往意味着其特征是“先进先出”。

我们就从这个事件队列（event queue）的模型去思考吧。要知道，队列是有可能为空的，这就如同银行的服务窗口前面不会总是有人排队一样——当然，我假设你不是总呆在北京，也去过一些小城市^^，那么，没有客户在窗口前需要被服务的时候，这个窗口的工作人员应该是什么状态呢？至少有两种不同的等待方式：一种是什么也不做傻等；一种是边喝点茶看看报纸算算账什么的其他事情边等待。对于计算机来说，第一种停下来等，就是 wait；第二种继续做其他事情的同时等，就是 poll。前者一看就明白，后者被不知道某位前辈高人翻译成“轮询”，好吧，说句实话，我笨，光看“轮询”这个词，完全无法理解是什么意思-_-!!!

SDL 为我们提供了两种等待事件的方式：

```
int SDL_WaitEvent(SDL_Event *event);
int SDL_PollEvent(SDL_Event *event);
```

两个函数的返回值都是 int，形参是 SDL 事件结构（C++里面，就把结构看成类吧。）指针 SDL_Event*（请注意我把 SDL_Event 和*连着写，这意味着在认识上，我把 SDL_Event* 本身看成一种复合类类型。）我们知道，在 C\C++里面，函数至少有三种基本功能。1、像命令似的起了某种作用；2、通过计算得到我们需要的返回值；3、指针（C++里面的引用）参数也可以传值。这里，我们先忽略 SDL_Event 结构的构造，看看这两个函数的作用。首先，他们不会引起某种作用；其次，他们的返回值是 1 或者 0；最后，他们会通过指针参数传值，这是重点。

早期的 C 里面，是没有关键字 true 和 false 的。通常用 1 代表 true，0 代表 false。我个人觉得，在 SDL 里面，1 和 0 的概念最容易与-1 和 0 的概念混淆。我们在学习 SDL_Init 的时候，说到返回值 0 代表成功，-1 代表失败。其实细细想，还是有差别的。0 和 1 是计算机固有的数据表示方式，而-1 是计算机原始方式所无法理解的，所以会代表着异常。所以，

在异常退出的时候，我选择使用 return -1。

这两个函数的返回值，在等到了事件的时候，返回 1，否则返回 0。官方文档里面用类似 while(SDL_PollEvent(&event))的方法引导轮询机制的开始，但是我觉得，对于新手来说，这样的表述不是很直观。与其间接的问窗口的服务员有客户来吗，还不如自己直接看看有没客户（event queue 是不是为空）。所以，在后面的例子里面，我实际上用的是 if (&event != 0)。

1.3：当前窗口。

如果你有兴趣研究 SDL 的官方文档，看到事件介绍（Introduction to Events）部分，也许会对以下问题感觉到奇怪：SDL 的事件查询机制是与 SDL_INIT_VIDEO 同时装载的。为什么呢？

我们知道，我们开发的游戏实际上是运行在操作系统的平台上的。当前的操作系统，都是多任务的操作系统。具体说到 GUI，有个很重要的概念就是你目前操作的是哪个程序，也就是更形象的概念——当前窗口。有些 event 可能是各个窗口，甚至包括系统本身共享的，比如鼠标移动（这不是绝对的，只是有可能）；有些 event 只会被当前窗口接受，就如同你不会希望同时开着两个 Word 文件在编辑，修改一个文件的时候，另外一个也被无情的修改了。所以，SDL 程序运行的时候，只有指定了哪个窗口是这个程序的窗口，并且这个窗口是当前窗口的时候，大部分 event 才能被正确的响应。

注意，console 窗口不是 SDL 程序的运行窗口，它属于操作系统本身。我们要打开 SDL 的程序窗口，需要引入一个新函数：

SDL_Surface *SDL_SetVideoMode(int width, int height, int bitsperpixel, Uint32 flags);

我们这里仅仅是为了打开 SDL 的程序窗口来引入这个函数，只做个简单介绍：1、这个函数本身有作用——打开 SDL 程序窗口；2、前三个参数分别是这个打开窗口的宽，高和位深，最后那个 flags 我们这里只介绍 SDL_SWSURFACE，这个位标表示把返回值的数据建立在系统内存里面。

1.4：一段演示按下ESC（或者点x）退出SDL窗口的程序。

```
//////////////////
//按下 ESC（或者点 x）退出 SDL 窗口
//联系我： znln426@163.com
//再别流年的技术实验室
//http://www.cppblog.com/lf426/
//////////////////

#include <iostream>
#include "SDL/SDL.h"

void pressESCtoQuit();
void doSomeLoopThings();
```

```
int main(int argc,char* argv[])
{
   try {
      if ( SDL_Init(SDL_INIT_VIDEO == -1 ))
         throw SDL_GetError();
   }
   catch ( const char* s ) {
      std::cerr << s << std::endl;
      return -1;
   }
   atexit(SDL_Quit);

   SDL_SetVideoMode(640, 480, 32, SDL_SWSURFACE);
   std::cout << "Program is running, press ESC to quit.\n";
   pressESCtoQuit();
   std::cout << "GAME OVER" << std::endl;

   return 0;
}

void pressESCtoQuit()
{

   std::cout << "pressESCtoQuit() function begin···\n";
   bool gameOver = false;
   while( gameOver == false ){
      SDL_Event gameEvent;
      SDL_PollEvent(&gameEvent);
      if ( &gameEvent != 0 ){
         if ( gameEvent.type == SDL_QUIT ){
            gameOver = true;
         }
         if ( gameEvent.type == SDL_KEYDOWN ){
            if ( gameEvent.key.keysym.sym == SDLK_ESCAPE ){
               gameOver = true;
            }
         }
      }
      doSomeLoopThings();
   }
   return;
}

void doSomeLoopThings()
```

```
{
   std::cout << ".";
   return;
}
```

1.5：两个细节问题。

我们修改 pressESCtoQuit()函数两个小地方：

```
void pressESCtoQuit()
{

   std::cout << "pressESCtoQuit() function begin…\n";

   bool gameOver = false;
   while( gameOver == false ){
      SDL_Event gameEvent;
      while ( SDL_PollEvent(&gameEvent) != 0 ){
         if ( gameEvent.type == SDL_QUIT ){
            gameOver = true;
         }
         if ( gameEvent.type == SDL_KEYUP ){
            if ( gameEvent.key.keysym.sym == SDLK_ESCAPE ){
               gameOver = true;
            }
         }
      }
      doSomeLoopThings();
   }
   return;
}
```

我们把引起轮询机制的 if 换成了 while。我们前面说过，查询&gameEvent 是不是为空，是为了强调 SDL_PollEvent()使用指针参数传了值。而事实上，if 是判断，while 不仅仅是判断，而且还是循环。但是更换这两个关键字似乎并没有出现不同，为什么呢？

我们分析两种情况下的模型。使用 if 的函数，实际上每次循环只查询 event 一次；而使用 while 的时候，内嵌的 while 循环会一直对“悬而未解”的事件队列（event queue）“弹”（这个隐喻意味着某个 event 一旦被处理，就不再属于 queue 的一个元素）出的头值进行处理，直到事件队列为空。这似乎更符合理想的模型。但是实际上，计算机的效率并不是我们假设的那样低，event queue 在一个外循环期间，始终保持两种状态：要么为空，要么最多一个。换句话说，你不可能在一次 gameOver==false 的循环期间，为 event queue 挤进去 1 个之上（>1）的 event，这就是 if 与 while 效果相同的原因。

我们修改的第二个细节是把 SDL_KEYDOWN 换成了 SDL_KEYUP，这是为了提醒大家，按下某个键和按下某个键再松开，是两种不同的 event。

2、显示一张BMP位图。

2.1：准备工作。

找一张*.bmp 格式的图片。我在例子中将使用 640*480 大小的图片。如果你在 windows 下面，你可以打开画图程序自己简单的画一张，或者将其他格式的图片另存为 bmp。然后将图片名字修改为 helloworld.bmp（当然，你也可以在程序的相应部分修改为你目标图片的名字。），这是我们将要显示的图片。

2.2：创建一个SDL的执行窗口。

我们讨论过，SDL 是跨平台的，它的设计者希望使用 SDL 的源程序不要依赖于具体平台，甚至具体的 GUI 和窗口管理器。在前面一节中，我们已经简单使用了 SDL_SetVideoMode()，在这里，我们对它做进一步的介绍——使用这个函数实际上遇到的问题会比我们预想中涉及到的问题多，换句话说，这里的介绍仍然是不完整的。我们当前的目的，只是为了简单的显示一张 BMP 位图。

SDL_Surface *SDL_SetVideoMode(int width, int height, int bitsperpixel, Uint32 flags);

在这里，我们使用的 flag(s)仍然是 SDL_SWSURFACE。它的作用是说明所建立的 surface 是储存在系统内存中的。实际上，SDL_SWSURFACE 是一个“伪位标”，如果你读出它的值，会发现其实是 0！这意味着任何其他位标（以及|组合）与 SDL_SWSURFACE 的&结果都是 0。这个事实的另外一层含义是，surface 的数据“至少”会被储存在系统内存中——对立面的意思是，这些数据有可能储存在显存中（指定使用显存储存数据的位标是 SDL_HWSURFACE，它的值是 1）。

这个函数的返回值是一个 SDL_Surface 的结构指针。如果返回是空指针（C 中习惯用 NULL，而 C++标准将空指针表示为 0），则表示这个函数调用失败了。我们可以通过 SDL_GetError()获得异常的原因。SDL_Surface 结构包含了一个 surface 的数据结构，包括宽，高和每个像素点的具体颜色等等，我们也放在后面具体讨论。这里，我们还是直接把 SDL_Surface 看成一个类，这个函数返回一个 SDL_Surface 类对象的指针。

width 和 height 是你希望建立的窗口的宽与高。如果值为 0，则建立与你当前桌面等宽高的窗口。bitsperpixel 是这个窗口的颜色位深。当前的硬件环境下，相信你的桌面也是 32 位色的。如果这个值为 0，则所建立的窗口使用你当前桌面的位深。

我们试图建立一个 640*480 大小的，32 位色的窗口。并且让返回的 surface 值储存在系统内存里。（后面会介绍使用显存的方法。）需要注意的是，我们必须记下这个返回的 surface 的指针，因为所有的图像操作，最后都是通过修改这个 surface 的数据作用在显示这个 surface 的窗口上，最终呈现在我们眼前的。

```
const int SCREEN_WIDTH = 640;    // 0 means use current width.
const int SCREEN_HEIGHT = 480;    // 0 means use current height.
const int SCREEN_BPP = 32;        // 0 means use current bpp.
const Uint32 SCREEN_FLAGS = SDL_SWSURFACE;    // SDL_SWSURFACE == 0,surface in system memory.

SDL_Surface* pScreen = 0;
```

```
pScreen = SDL_SetVideoMode(SCREEN_WIDTH, SCREEN_HEIGHT, SCREEN_BPP, SCR
EEN_FLAGS);    // Creat a SDL window, and get the window's surface.
try {
    if ( pScreen == 0 )
        throw SDL_GetError();
}
catch ( const char* s ) {
    std::cerr << "SDL_SetVideoMode() failed!\n" << s << std::endl;
    SDL_Quit();
    return -1;
}
```

2.3：装载BMP格式的位图。

SDL_Surface *SDL_LoadBMP(const char *file);

这个函数使用 C 风格字符串的形参，这意味着如果我们使用 std::string objName 传值的时候，需要使用 objName.c_str()（请注意 objName.data()没有'/0'），把 std::string 类转化为 C 风格字符串。这个函数把一个 BMP 位图转化成为 SDL 的 surface 数据结构方式（SDL_Surface 结构），储存在系统内存中（我没找到任何信息可以说明能直接储存到显存中），并返回这个 surface 的指针。如果返回的指针为空，说明函数调用失败了。

```
SDL_Surface* pShownBMP = 0;
pShownBMP = SDL_LoadBMP("helloworld.bmp"); // Load a BMP file, and convert it as a surf
ace.
try {
    if ( pShownBMP == 0 )
        throw SDL_GetError();
}
catch ( const char* s ) {
    std::cerr << "SDL_LoadBMP() failed!\n" << s << std::endl;
    SDL_Quit();
    return -1;
}
```

2.4：块移图面（blit surface）。

int SDL_BlitSurface(SDL_Surface *src, SDL_Rect *srcrect, SDL_Surface *dst, SDL_Rect *dstre
ct);

src 指的是要进行 blit 的源 surface，dst 指的是 blit 这个 surface 要去的目的地——另外一个 surface。我们这里先忽略 SDL_Rect 结构的具体意思，仅仅需要了解的是，如果 srcrect 为空指针，意味着整个源 surface 将被 blit；如果 dstrect 为空指针，意味着源 surface 与目的 surface 的左上角重合（坐标(0,0)）。

blit 是个有渊源的词语，我将来会在术语解释中具体提到。这个词的本意就是块（block）移动（transfer）的缩写 blt，因为这个缩写缺少元音不好读，所以后来加上了 i，就变成 blit。

如果 blit 成功，则返回 0；否则返回-1。

```
SDL_Rect* pSrcRect = 0;    // If pSrcRect is NULL, the entire source surface is copied.

SDL_Rect* pDstRect = 0;    // If pDstRect is NULL, then the destination position (upper left co
rner) is (0, 0).
try {
    if ( SDL_BlitSurface(pShownBMP, pSrcRect, pScreen, pDstRect) != 0 )    // Put the BMP's s
urface on the SDL window's surface.
        throw SDL_GetError();
}
catch ( const char* s ) {
    std::cerr << "SDL_BlitSurface() failed!\n" << s << std::endl;
    SDL_Quit();
    return -1;
}
```

2.5：显示图片。

```
int SDL_Flip(SDL_Surface *screen);
```

源图面被 blit 到目的图面上后，就与目的图面融为一体了。在我们的例子中，ShownBMP 被“画”在了 Screen 上（我这里去掉了 p，是为了说明这里讨论的是 surface 而不是 surface 的指针）。换句话说，Screen 被修改了（似乎也可以用“染指”-_-!!），ShownBMP 则没有改变。

另外一个需要了解的问题是，我们之前对 surface 的种种操作，实际上都是在修改 surface 数据结构中的某些数据，当我们最后需要将这些 surface 显示到屏幕上（我们打开的 SDL 操作窗口上），我们需要使用函数 SDL_Flip()。如果函数调用成功，则返回 0；否则返回-1。

```
try {
    if ( SDL_Flip(pScreen) != 0 )    // Show the SDL window's surface.
        throw SDL_GetError();
}
catch ( const char* s ) {
    std::cerr << "SDL_Flip() failed!\n" << s << std::endl;
    SDL_Quit();
    return -1;
}
```

2.6：这个例子的完整源代码。

```
#include <iostream>
#include "SDL/SDL.h"

void pressESCtoQuit();

int main(int argc, char* argv[])
```

```
{
    try {
        if ( SDL_Init(SDL_INIT_VIDEO) != 0 )
            throw SDL_GetError();
    }
    catch ( const char* s ) {
        std::cerr << "SDL_Init() failed!\n" << s << std::endl;
        return -1;
    }

    const int SCREEN_WIDTH = 640;    // 0 means use current width.
    const int SCREEN_HEIGHT = 480;    // 0 means use current height.
    const int SCREEN_BPP = 32;        // 0 means use current bpp.
    const Uint32 SCREEN_FLAGS = SDL_SWSURFACE;    // SDL_SWSURFACE == 0,surface i
n system memory.

    SDL_Surface* pScreen = 0;
    pScreen = SDL_SetVideoMode(SCREEN_WIDTH, SCREEN_HEIGHT, SCREEN_BPP, SCR
EEN_FLAGS);    // Creat a SDL window, and get the window's surface.
    try {
        if ( pScreen == 0 )
            throw SDL_GetError();
    }
    catch ( const char* s ) {
        std::cerr << "SDL_SetVideoMode() failed!\n" << s << std::endl;
        SDL_Quit();
        return -1;
    }

    SDL_Surface* pShownBMP = 0;
    pShownBMP = SDL_LoadBMP("helloworld.bmp"); // Load a BMP file, and convert it as a surf
ace.
    try {
        if ( pShownBMP == 0 )
            throw SDL_GetError();
    }
    catch ( const char* s ) {
        std::cerr << "SDL_LoadBMP() failed!\n" << s << std::endl;
        SDL_Quit();
        return -1;
    }

    SDL_Rect* pSrcRect = 0;    // If pSrcRect is NULL, the entire source surface is copied.
```

```
    SDL_Rect* pDstRect = 0;    // If pDstRect is NULL, then the destination position (upper left co
rner) is (0, 0).
    try {
        if ( SDL_BlitSurface(pShownBMP, pSrcRect, pScreen, pDstRect) != 0 )    // Put the BMP's s
urface on the SDL window's surface.
            throw SDL_GetError();
    }
    catch ( const char* s ) {
        std::cerr << "SDL_BlitSurface() failed!\n" << s << std::endl;
        SDL_Quit();
        return -1;
    }

    try {
        if ( SDL_Flip(pScreen) != 0 )    // Show the SDL window's surface.
            throw SDL_GetError();
    }
    catch ( const char* s ) {
        std::cerr << "SDL_Flip() failed!\n" << s << std::endl;
        SDL_Quit();
        return -1;
    }

    pressESCtoQuit();
    SDL_Quit();

    return 0;
}

void pressESCtoQuit()
{
    bool gameOver = false;
    while( gameOver == false ){
        SDL_Event gameEvent;
        while ( SDL_PollEvent(&gameEvent) != 0 ){
            if ( gameEvent.type == SDL_QUIT ){
                gameOver = true;
            }
            if ( gameEvent.type == SDL_KEYUP ){
                if ( gameEvent.key.keysym.sym == SDLK_ESCAPE ){
                    gameOver = true;
                }
            }
        }
```

```
    }
    return;
}
```

2.7：补充说明。

1) 这个程序用到了前面课程中建立起来的函数 pressESCtoQuit();
2) 在 VC 的 IDE 中，引用的 bmp 文件可能会需要提供完整的绝对路径，否则直接通过 VC 菜单启动的程序可能找不到实际上就与 exe 文件在同一个文件夹中的 bmp 图片。你可以直接找到编译后的 exe 文件，在 exe 文件夹中直接运行，则不会出现这个问题。或者，你可以修改 VC 默认的资源文件路径，再或者，你可以尊重 VC 的默认约定，将资源文件拷贝到工程目录下（与源文件*.cpp 在同一个文件夹里面）。

3、视频属性信息（VideoInfo）

3.1：获取视频属性信息。

```
const SDL_VideoInfo *SDL_GetVideoInfo(void);
```

我们在前一小节中，为了尽快实现一个 SDL 的运行窗口，跳过了很多细节，也留下了很多问题。其中一个很重要的问题就是：我们到底有没有使用到显卡的硬件加速？因为硬件的差异性，直接使用硬件接口的时候，会出现很多新的问题。这些问题在第（四）章中，我将以自己的操作系统和显卡硬件配置，通过试验得到结论，而在这之前，我们还必须弄清楚其他几个问题。

SDL_GetVideoInfo()将返回当前 SDL 运行窗口的视频属性信息，其数据组织在一个名为 SDL_VidioInfo 的结构中。该函数就是返回这个只读结构的指针。

```
typedef struct{
  Uint32 hw_available:1;
  Uint32 wm_available:1;
  Uint32 blit_hw:1;
  Uint32 blit_hw_CC:1;
  Uint32 blit_hw_A:1;
  Uint32 blit_sw:1;
  Uint32 blit_sw_CC:1;
  Uint32 blit_sw_A:1;
  Uint32 blit_fill:1;
  Uint32 video_mem;
  SDL_PixelFormat *vfmt;
  int current_w;
  int current_h;
} SDL_VideoInfo;
```

初看这个结构似乎有点让人头大，其实分析起来很简单，成员名称也很友好，容易让人记忆。hw_availabale 表示创建硬件 surface 的可行性（1 表示可以，0 表示不可以，后同）；

wm_available 表示是否存在可用的窗口管理器；3-9 行表示了一系列硬件到硬件，软件到硬件加速的可行性；video_men 表示显存大小；vfmt 是当前显示驱动（video device）的像素格式（pixel format）；current_w 和 current_h 是当前窗口的宽和高。

我们在使用在系统内存中建立 SDL 运行窗口的方式来察看这些成员数据。

```
SDL_Init(SDL_INIT_VIDEO);
atexit(SDL_Quit);

SDL_Surface* pScreen = SDL_SetVideoMode(640, 480, 32, SDL_SWSURFACE);
SDL_Flip(pScreen);

const SDL_VideoInfo* myInfo = SDL_GetVideoInfo();
cout << "Is it possible to create hardware surfaces? " << myInfo->hw_available << endl;
cout << "Is there a window manager available? " << myInfo->wm_available << endl;
cout << "Are hardware to hardware blits accelerated? " << myInfo->blit_hw << endl;
cout << "Are hardware to hardware colorkey blits accelerated? " << myInfo->blit_hw_CC << e
ndl;
cout << "Are hardware to hardware alpha blits accelerated? " << myInfo->blit_hw_A << endl;
cout << "Are software to hardware blits accelerated? " << myInfo->blit_sw << endl;
cout << "Are software to hardware colorkey blits accelerated? " << myInfo->blit_sw_CC << en
dl;
cout << "Are software to hardware alpha blits accelerated? " << myInfo->blit_sw_A << endl;
cout << "Are color fills accelerated? " << myInfo->blit_fill << endl;
cout << "Total amount of video memory in Kilobytes? " << myInfo->video_mem << endl;
cout << "Width of the current video mode? " << myInfo->current_w << endl;
cout << "Height of the current video mode? " << myInfo->current_h << endl;
```

结果似乎是可以预料的，创建硬件 surface 和硬件加速都是不可行的。接下来，我们将 flag 从使用系统内存的 SDL_SWSURFACE 换成使用显存的 SDL_HWSURFACE。

3.2：我的显卡不支持硬件加速？？!!

好吧，也许是我运气不好。问题来了！

我的软件环境是 windows server 2003，硬件环境是 GeForce 4 Ti 4200 AGP 8x。当我第一次看到 SDL 反馈给我的信息：创建硬件 surface 不可行！硬件加速不可性！显存大小为 0!!——我快喘不过气来。

我的第一反应是 SDL 不支持我的显卡——后面一想，不对啊，SDL 到今天（2008 年 2 月）还在更新，我显卡可算是古董了——一定是什么地方搞错了，或者还有我没有了解到的问题。

伴随而来的另外一个问题是：我是不是真正把 surface 建立到显存中了——尽快我要求 SDL 这么做。这里，我要第三次提到函数 SDL_SetVideoMode()了。我们之前介绍过，这个函数的返回值是一个 SDL_Surface 结构的指针。而 SDL_Surface 结构中有一个数据成员 flags 储存了这个 surface 的位标信息，其中就包括是否建立到了显存里面（否则就在系统内存中）。

```
typedef struct SDL_Surface {
  Uint32 flags;                    /* Read-only */
```

```
  SDL_PixelFormat *format;              /* Read-only */
  int w, h;                         /* Read-only */
  Uint16 pitch;                      /* Read-only */
  void *pixels;                      /* Read-write */
  SDL_Rect clip_rect;                    /* Read-only */
  int refcount;                     /* Read-mostly */

  /* This structure also contains private fields not shown here */

} SDL_Surface;
```

这里我们继续忽略其他不熟悉的数据成员，直接将 flags 的信息读出来。

```
  cout << "pScreen->flags = ";
  showHex(pScreen->flags);
  cout << boolalpha;
  cout << "SDL_SWSURFACE? " << !(bool((pScreen->flags) & SDL_HWSURFACE)) << endl;
  cout << "SDL_HWSURFACE? " << bool((pScreen->flags) & SDL_HWSURFACE) << endl;
  cout << "SDL_DOUBLEBUF? " << bool((pScreen->flags) & SDL_DOUBLEBUF) << endl;
  cout << noboolalpha;
```

请注两点：函数 showHex()显示 16 进格式，在前面章节有原形和定义；SDL_SWSURFACE 是 0，是伪位标，因为它与 SDL_HWSURFACE 只能二取一的，所以他的实际状态可以用如上方式表示。另外，SDL_DOUBLEBUF 是在开启硬件画 surface 和加速时候很重要的位标，在后面会有介绍。

结果是——很不幸，surface 没有建立在显存中。

3.3：SDL的环境设置。

寻找问题和试验的过程很曲折。我在这里直接说结论，在下一节中，再进行具体的试验。SDL 有个环境设置的概念。这是因为 SDL 是跨平台的，在不同的操作系统和不同的 GUI 之上，SDL 试图建立起一个与以上因素无关的封装。但是因为硬件（主要指显卡）的具体多样性，SDL 在不同的 OS 和不同的 GUI 之上，使用不同的“驱动”，以 windows 为例，SDL 在 windows 上的驱动有 GDI（windib）和 DirectX（directx）两种（Linux 下则更多，请参考官方资料），我们可以通过一个函数知道当前 SDL 使用的驱动版本。

```
char *SDL_VideoDriverName(char *namebuf, int maxlen);
```

如果返回空指针，则表示 SDL_Init()没有装载或出现异常。我们可使用下面这样的语句查看当前 SDL 使用的驱动版本名字。

```
  char driverName[20];
  SDL_VideoDriverName(driverName, 20);
  cout << "SDL_VideoDriverName = " << driverName << endl;
```

结果是：windib。为什么 SDL 官方资料显示，默认值是 directx 呢？（这不是反问，这是我的疑问，请知道答案的同学跟我联系，谢谢）——结论是，windib 无法打开硬件加速，要使用硬件加速，必须使用 directx。官方资料上用了很简短的描述来说明进行 SDL 的环境设置。在后面的章节中，我们将使用：

```
putenv("SDL_VIDEODRIVER=directx");
```

注意：这个函数必须用在 SDL_Init();之前才有实际效果。来设置为 directx 环境。（VC 下为

了解除编译警告，也可使用 SDL_putenv()来代替 putenv()，效果都一样。）之后除了用 SDL_VideoDriverName()获取显示驱动信息，还可以使用：（同样，VC 下可使用 SDL_getent()替换）

const char* myvalue = getenv("name") ;

来获取相关的环境信息（包括显示驱动信息。"name"换为"SDL_VIDEODRIVER"）。

SDL入门教程（四）：

1、SDL动画的软件渲染（Software Render）

1.1：准备工作。

准备一张 640*480 的 bmp 位图（back.bmp）作为背景，另外一张小一点的（比如 100*100 的 front.bmp）作为前景。

渲染（render）是 CG 术语，是将以数据结构储存在计算机中的图片以人眼可观察的图片实现出来的过程。前面例子中显示一张 bmp 图片实际上也是渲染。对于那张 bmp 图片，我们经历了 3 过过程的处理：1、将 bmp 图片转化为 SDL_Surface 的结构格式；2、将转化后的 SDL_Surface 块移（blit）到 SDL 创建的窗口 surface 上；3、将这个 surface 显示出来，实际上就是渲染出来。

当然，前面仅仅是渲染了单帧的图片，我们这里对程序做些修改，从而实际上是连续的渲染多帧动画（虽然表面上我们并没有让画面动起来）。

1.2：演示程序的完整源代码。

```
#include <iostream>
#include "SDL/SDL.h"

SDL_Surface* pScreen = 0;
SDL_Surface* pBack = 0;
SDL_Surface* pFront = 0;

void pressESCtoQuitPlus();
void loopRender();

int main(int argc, char* argv[])
{

   try {
      if ( SDL_Init(SDL_INIT_VIDEO) != 0 )
         throw SDL_GetError();
```

```
    }
    catch ( const char* s ) {
        std::cerr << "SDL_Init() failed!\n" << s << std::endl;
        return -1;
    }

    const int SCREEN_WIDTH = 640;
    const int SCREEN_HEIGHT = 480;
    const int SCREEN_BPP = 32;
    const Uint32 SCREEN_FLAGS = SDL_FULLSCREEN;

    pScreen = SDL_SetVideoMode(SCREEN_WIDTH, SCREEN_HEIGHT, SCREEN_BPP, SCR
EEN_FLAGS);
    try {
        if ( pScreen == 0 )
            throw SDL_GetError();
    }
    catch ( const char* s ) {
        std::cerr << "SDL_SetVideoMode() failed!\n" << s << std::endl;
        SDL_Quit();
        return -1;
    }

    pBack = SDL_LoadBMP("back.bmp");
    try {
        if ( pBack == 0 )
            throw SDL_GetError();
    }
    catch ( const char* s ) {
        std::cerr << "SDL_LoadBMP() failed!\n" << s << std::endl;
        SDL_Quit();
        return -1;
    }

    pFront = SDL_LoadBMP("front.bmp");
    try {
        if ( pFront == 0 )
            throw SDL_GetError();
    }
    catch ( const char* s ) {
        std::cerr << "SDL_LoadBMP() failed!\n" << s << std::endl;
        SDL_Quit();
        return -1;
    }
```

```
    try {
        pressESCtoQuitPlus();
    }
    catch ( const char* s ) {
        std::cerr << "pressESCtoQuitPlus() failed!\n" << s << std::endl;
        SDL_Quit();
        return -1;
    }

    SDL_Quit();

    return 0;
}

void pressESCtoQuitPlus()
{
    bool gameOver = false;
    while( gameOver == false ){
        SDL_Event gameEvent;
        while ( SDL_PollEvent(&gameEvent) != 0 ){
            if ( gameEvent.type == SDL_QUIT ){
                gameOver = true;
            }
            if ( gameEvent.type == SDL_KEYUP ){
                if ( gameEvent.key.keysym.sym == SDLK_ESCAPE ){
                    gameOver = true;
                }
            }
        }
        loopRender();
    }
    return;
}

void loopRender()
{
    SDL_Rect* pSrcRect = 0;
    SDL_Rect* pDstRect = 0;
    if ( SDL_BlitSurface(pBack, pSrcRect, pScreen, pDstRect) != 0 )
        throw SDL_GetError();
    if ( SDL_BlitSurface(pFront, pSrcRect, pScreen, pDstRect) != 0 )
        throw SDL_GetError();
    if ( SDL_Flip(pScreen) != 0 )
```

```
        throw SDL_GetError();
    return;
}
```

1.3：程序改动说明。

1) 使用了全屏幕显示；
2) 使用了改动后的 pressESCtoQuit()函数，用于在全屏模式下方便退出。
3) 将一些变量修改为全局变量，以提供 main()所调用的函数使用。

2、SDL动画的硬件渲染（Hardware Render）

2.1：需要修改的地方。

这里，我们真正的开始使用 SDL 的硬件渲染。首先，我们需要设置驱动的环境（以 windows 为例，我们设置为 directx，Linux 的设置请参考官方网站，我们这里预留为 dga）。另外，如果要启动硬件加速，必须使用全屏模式（SDL_FULLSCREEN），所以，在前面的软件渲染中，我们也使用全屏以作对比。第三，硬件渲染需要打开双缓冲（SDL_DOUBLEBUF），至于为什么我们在最后讨论，我们还是先看看完整的代码。

2.2：硬件渲染演示程序完整的源代码。

```
#define __windows__    // Linux using #define __linux__
#include <iostream>
#include "SDL/SDL.h"

SDL_Surface* pScreen = 0;
SDL_Surface* pBack = 0;
SDL_Surface* pFront = 0;

void pressESCtoQuitPlus();
void loopRender();

int main(int argc, char* argv[])
{
#ifdef __windows__
    SDL_putenv("SDL_VIDEODRIVER=directx");
#endif

#ifdef __linux__
    putenv("SDL_VIDEODRIVER=dga");
#endif
```

```
   try {
      if ( SDL_Init(SDL_INIT_VIDEO) != 0 )
         throw SDL_GetError();
   }
   catch ( const char* s ) {
      std::cerr << "SDL_Init() failed!\n" << s << std::endl;
      return -1;
   }

   const int SCREEN_WIDTH = 640;
   const int SCREEN_HEIGHT = 480;
   const int SCREEN_BPP = 32;
   const Uint32 SCREEN_FLAGS = SDL_FULLSCREEN | SDL_DOUBLEBUF | SDL_HWSUR
FACE;

   pScreen = SDL_SetVideoMode(SCREEN_WIDTH, SCREEN_HEIGHT, SCREEN_BPP, SCR
EEN_FLAGS);
   try {
      if ( pScreen == 0 )
         throw SDL_GetError();
   }
   catch ( const char* s ) {
      std::cerr << "SDL_SetVideoMode() failed!\n" << s << std::endl;
      SDL_Quit();
      return -1;
   }

   pBack = SDL_LoadBMP("back.bmp");
   try {
      if ( pBack == 0 )
         throw SDL_GetError();
   }
   catch ( const char* s ) {
      std::cerr << "SDL_LoadBMP() failed!\n" << s << std::endl;
      SDL_Quit();
      return -1;
   }

   pFront = SDL_LoadBMP("front.bmp");
   try {
      if ( pFront == 0 )
         throw SDL_GetError();
   }
```

```
    catch ( const char* s ) {
        std::cerr << "SDL_LoadBMP() failed!\n" << s << std::endl;
        SDL_Quit();
        return -1;
    }

    try {
        pressESCtoQuitPlus();
    }
    catch ( const char* s ) {
        std::cerr << "pressESCtoQuitPlus() failed!\n" << s << std::endl;
        SDL_Quit();
        return -1;
    }

    SDL_Quit();

    return 0;
}

void pressESCtoQuitPlus()
{
    bool gameOver = false;
    while( gameOver == false ){
        SDL_Event gameEvent;
        while ( SDL_PollEvent(&gameEvent) != 0 ){
            if ( gameEvent.type == SDL_QUIT ){
                gameOver = true;
            }
            if ( gameEvent.type == SDL_KEYUP ){
                if ( gameEvent.key.keysym.sym == SDLK_ESCAPE ){
                    gameOver = true;
                }
            }
        }
        loopRender();
    }
    return;
}

void loopRender()
{
    SDL_Rect* pSrcRect = 0;
    SDL_Rect* pDstRect = 0;
```

```
    if ( SDL_BlitSurface(pBack, pSrcRect, pScreen, pDstRect) != 0 )
        throw SDL_GetError();
    if ( SDL_BlitSurface(pFront, pSrcRect, pScreen, pDstRect) != 0 )
        throw SDL_GetError();
    if ( SDL_Flip(pScreen) != 0 )
        throw SDL_GetError();
    return;
}
```

2.3：问题。

你可能发现除了鼠标指针不显示之外，没有其它问题——这其实不是好现象，因为应该和可能出现的问题，都被我们事先避免了，但是这样让我们离事情的本质越来越远。你可以尝试着关掉SDL_DOUBLEBUF位标看看是什么效果；或者，在前面渲染单帧的程序中使用硬件渲染同时打开双缓冲看看出现什么问题——这些正是我们下一节将要讨论的。

如果你迫不及待的想知道原因，并且英语也过关的话，对于硬件渲染可能会引发的问题，我给你推荐一篇SDL官方也推荐的论文：

http://www.linuxdevcenter.com/pub/a/linux/2003/08/07/sdl_anim.html

但是很不幸的是，我在试验的过程中发现这篇文章有很多问题，当然，也许是我错了。因为我仅仅把SDL作为了一个黑盒子来研究，但是我得到的试验结果，却是不可能错的。

2.4：补充。

目前用Debian试验的时候，发现NVidia的显卡驱动屏蔽掉了dga的。也就是说实际上用不了，或者会设置起来很麻烦。实际上，SDL通过x11 来实现图像，我目前的认识应该是这样的：SDL->x11->NV驱动->显卡。所以，实际上我们虽然没有通过SDL接触到显卡，但实际上还是通过种种渠道调用了显卡，我们应该充分相信NV的工程师比我们牛得多。NV官方解释如下：

http://us.download.nvidia.com/XFree86/Linux-x86/169.04/README/chapter-07.html#id2546686

Why do applications that use DGA graphics fail?

The NVIDIA driver does not support the graphics component of the XFree86-DGA (Direct Graphics Access) extension. Applications can use the XDGASelectInput() function to acquire relative pointer motion, but graphics-related functions such as XDGASetMode() and XDGAOpenFramebuffer() will fail.

The graphics component of XFree86-DGA is not supported because it requires a CPU mapping of framebuffer memory. As graphics cards ship with increasing quantities of video memory, the NVIDIA X driver has had to switch to a more dynamic memory mapping scheme that is incompatible with DGA. Furthermore, DGA does not cooperate with other graphics rendering libraries such as Xlib and OpenGL because it accesses GPU resources directly.

NVIDIA recommends that applications use OpenGL or Xlib, rather than DGA, for graphics rendering. Using rendering libraries other than DGA will yield better performance and improve interoperability with other X applications.

3、SDL的软、硬件渲染的深入试验和分析

3.1：试验——硬件渲染下关闭双缓存。

现象：front 图片出现不断被“撕裂”的效果。

双缓存的概念，是在计算机速度还不足以满足“即时作图”的情况下的一种技术。即，在屏幕（即前台的帧缓存 framebuffer）上显示一帧图片的同时，在后台一个帧缓存的映射中作图。这样，只有当屏幕画面需要改变的时候，后台的缓存才交换到前台来，这样就避免了在前台出现计算机“作图”的过程。

关闭双缓存之后，我们实际上看到的是计算机在不停的“作图”——blit 一次 back，然后 blit 一次 front，然后循环，所以被“撕裂”的画面，实际上是 back 和 front 图片“混合”的效果。

在软件渲染的时候，为什么没有双缓存也不会出现这个问题呢？我的猜想是：图片的像素数据实际上还是储存在内存中的，无论是系统内存还是显存，实际上都是通过一种影射提供给真正“成像”的帧缓存的。所以，实际上，无论软渲染还是硬渲染，图片实际上都是被“双缓存”的，所不同的是，在使用硬件渲染的时候，SDL 试图给我们提供直接访问硬件的接口，导致我们对创建在 hw 下的 surface 的操作，就类似在直接操作帧缓存，但是，我们是不是真的就直接访问了帧缓存呢？

3.2：试验——单帧硬件渲染下打开双缓存。

现象：图片出现闪烁。

闪烁实际上是图片与一个空屏（全黑，0 像素）交互显示的结果。为了验证这个猜想，我们可以尝试 blit 同一个 surface 两次，则可以解决这个问题。这个试验其实更形象的显示了什么是“双”缓存，两次 blit 可以让两个缓存都存储上相同的图片，渲染时候就不会出现闪烁。

3.3：我们可以直接访问缓存的地址吗？

SDL 官方推荐的那篇论文介绍可以通过 pScreen->pixels 查看像素缓存的地址，并且认为如果地址改变则映射交换缓存的实质是传递了数据结构的指针；而如果不变则是直接拷贝了数据结构的数据——事实真的是这样吗？

我们知道，pScreen 是通过 SDL_SetVideoMode()函数获得的。这是一个特殊的 surface，因为它既具有一般 surface 的属性，而且也是 SDL 用于实际显示的窗口 surface。我们建立起 pScreen 的时候，并没有赋予他任何的像素数据。

如果在软件渲染的环境下，pScreen->pixels 可以成功返回一个地址；但是在硬件渲染的环境下，pScreen->pixels 则返回了空指针！这意味着什么？要么就真是一个空指针（因为 pScreen 实际上的像素数据为空），另外一个可能是——这个地址我们不可以访问！我其实倾向于后者的解释，因为软件渲染下，同样为空像素的 pScreen 并不返回空指针。所以，我的猜想是，SDL 中，帧缓存并不能被直接访问，但是我们可以访问帧缓存的映射，并且模拟访问帧缓存的效果。

3.4：对于双缓存现象的另外一种解释。

在 SDL 官方推荐的那篇论文中，作者认为出现“撕裂”现象的原因是硬件加速与软件的不同步。简单来说，在软件渲染情况下，语句被一条条逐步执行，比如：1、blit back；2、blit front；3、flip screen。这 3 个步骤，即使都需要一定的执行时间，也是有先后秩序的。但在硬件渲染环境下，这 3 条指令被程序直接仍给了显卡，有可能前面两次 blit 还没执行完，就 flip 了。所以，作者认为，SDL_DOUBLEBUF 位标的使用实际上是改变了 SDL_Flip()的执行效果：无双缓存的时候，将直接把当时的情况 flip 出来；打开双缓存的时候，flip 指令则成为了递交了请求，然后不暂定的等待（所谓轮询），直到所有的“作图”指令执行完才显示出来——似乎也说得过去，但是我以为如此解释原理过于的复杂的。当然，也可能我对鸟语理解得不对，欢迎大家继续讨论这个话题。

SDL入门教程（五）：

1、构建SDL screen surface类

今天是元宵节，新年过去，又要开始抓紧时间奋斗了。祝大家都能继续追逐自己的梦想。BS 的书上引过一句话，“一个人要是不耕作，就必须写作”，所以，不耕作的我不能停止写作，time is money, my friend!

1.1：整理两种SDL_Surface的关系。

前面一直在用面向过程的思想写程序，因为 OOP 细分到每一个具体的方法，还是过程。OOP 的难点其实在于理清楚不同类之间的关系。说实话，我只是为了自己的理想，为了实现我计划的项目，刚刚开始学习 C++的菜鸟。两个多月的 C++能有什么水平，希望前辈们不要见笑，我会继续努力的。而且，可能因为对于 C++的偏爱，再加上我目前能找到的 SDL 相关教程都是 C 风格的，所以我充满了用 C++来写 SDL 教程挑战的热情。

SDL_Surface 是 SDL 的一个结构。在我们前面的知识中，学习了构建这个结构的两种方法：一种是通过 SDL_SetVideoMode()；一种是 SDL_LoadBMP()。其实，通过 SDL_SetVideoMode()构建的 SDL_Surface 是一种特殊的 surface，因为，实际上，其他的 surface 都是通过 blit 到这个 surface 上，最终通过 flip 这个 surface，才能显示出来。所以，这个 surface 可以看成是 SDL 库中，数据形式的 surface（储存在电脑中）与实体形式的 surface（通过屏幕显示出来）的唯一接口。另外，因为构建这两种 surface 需要的数据成员小同大异，类方法也不尽相同。所以，虽然我也考虑过建立一个基类把两种 surface 作为派生类，但是我最终选择了建立两个类。

1.2：构建SDL screen surface类。

```
class ScreenSurface
{
```

```
private:
   static int screenNum;
   int width;
   int height;
   int bpp;
   Uint32 flags;
   SDL_Surface* pScreen;
public:
   ScreenSurface();
   ScreenSurface(int w, int h, int b = 0, Uint32 f = 0);
   ~ScreenSurface();
   SDL_Surface* point() const;
   bool flip() const;
};
```

我先设定了一个静态 int 作为计数器。我的考虑是，screen surface 实际上只需要建立一个。并且，其他的 surface 实际上都是“依赖”于这个 surface 才能显示出来的。在 SDL 中，多次 SDL_SetVideoMode()的效果实际上是保留了最后一次的 surface 作为了 screen surface，所以，多次 SDL_SetVideoMode()其实是没有实际意义的。计数器 screenNum 用于保证只创建一次 screen surface，多次创建我倾向让程序抛出异常。

构建函数除了调用 SDL_SetVideoMode()函数，还作为 SDL_Init()的启动载入。所以，我专门定义析构函数的目的，是因为 C++会在类对象消亡的时候自动调用析构函数。因为保证了只建立一个 screen surface，并且在创建对象的时候载入了 SDL_Init()，所以，如果析构函数中使用 SDL_Quit()，则可以在手动调用析构函数或者程序结束的时候调用 SDL_Quit()了。

方法 point()返回对象中的 pScreen，其实就是 SDL_Surface 结构的指针。因为 SDL 库是 C 风格的，所以，直接使用指针的函数很多。

方法 flip()用于把 screen surface 最终显示出来。

1.3：screen surface的类方法。

```
int ScreenSurface::screenNum = 0;
```

首先为静态变量附初值。

```
ScreenSurface::ScreenSurface():
width(640), height(480), bpp(32), flags(0)
{
   if ( screenNum > 0 )
      throw "DONOT create more than ONE screen!";
   if ( SDL_Init(SDL_INIT_VIDEO < 0 ) )
      throw SDL_GetError();
   pScreen = SDL_SetVideoMode(width, height, bpp, flags);
   screenNum++;
}
```

```
ScreenSurface::ScreenSurface(int w, int h, int b, Uint32 f):
width(w), height(h), bpp(b), flags(f)
{
   if ( screenNum > 0 )
      throw "DONOT create more than ONE screen!";
   if ( SDL_Init(SDL_INIT_VIDEO < 0 ) )
      throw SDL_GetError();
   pScreen = SDL_SetVideoMode(width, height, bpp, flags);
   screenNum++;
}
```

构造函数。如果创建 1 个以上的 screen surface，则会抛出异常。

```
ScreenSurface::~ScreenSurface()
{
   SDL_Quit();
}
```

析构函数。在对象消亡时，退出 SDL 系统。

```
SDL_Surface* ScreenSurface::point() const
{
   return pScreen;
}
```

返回 screen surface 中 SDL_Surface 结构的指针，主要提供给 SDL 的函数调用。

```
bool ScreenSurface::flip() const
{
   if ( SDL_Flip(pScreen) < 0 )
      return false;
   else return true;
}
```

显示（弹出 flip）screen surface 到屏幕上。

2、构建SDL surface类

2.1：构建普通的surface类。

在所有的 surface 里面，只有 screen surface 是最特殊的。因为第一，screen surface 只有一个；第二，其他所有的普通 surface 都必须被 blit 到 screen surface 上，通过 flip screen surface 才能显示出来。所以，我们可以认为普通的 surface 是“依赖”于一个 screen surface 的。所以，考虑在构建 surface 的时候，除了需要装载的 bmp 文件，还需要指定其所依赖的 screen surface。

```
class DisplaySurface
{
private:
   string fileName;
   SDL_Surface* pSurface;
   SDL_Surface* pScreen;
```

```
public:
   DisplaySurface(string file_name, const ScreenSurface& screen);
   ~DisplaySurface();
   SDL_Surface* point() const;
   bool blit() const;
};
```

2.2：surface的类方法。

```
DisplaySurface::DisplaySurface(std::string file_name, const ScreenSurface& screen):
fileName(file_name)
{
   pSurface = SDL_LoadBMP(file_name.c_str());
   if ( pSurface == 0 )
      throw SDL_GetError();
   pScreen = screen.point();
}
```

构造函数。我们指定一个 bmp 文件，和一个 screen surface 对象来构造 surface。注意，这里我们用到了 C++的 string。我们前面说过，string 的方法 c_str()用于把 C++的 string 对象转化为 C 风格字符串，而这正是 SDL 的函数所需要的。

```
DisplaySurface::~DisplaySurface()
{
   SDL_FreeSurface(pSurface);
}
```

虽然 load 的 bmp 可以在 SDL_Quit()的时候自动释放，不过也许我们也会有需要手动释放的时候。比如，装载一张图片，进行某种处理和转换后，保留处理后的，就可以把原来用于处理的对象释放掉了。

```
SDL_Surface* DisplaySurface::point() const
{
   return pSurface;
}
```

返回对象中，bmp 的 SDL_Surface 指针。同样为了用于 SDL 库的函数。

```
bool DisplaySurface::blit() const
{
   if ( SDL_BlitSurface(pSurface, 0, pScreen, 0) < 0 )
      return false;
   else return true;
}
```

把 surface blit 到 screen surface 上面，左上角重合。到这里，我们前面用到的对于 surface 的所有操作，都包含到类方法中了。

3、对SDL_BlitSurface()的进一步讨论

3.1：矩形区域SDL_Rect。

```
typedef struct{
  Sint16 x, y;
  Uint16 w, h;
} SDL_Rect;
```

因为显示器通常是矩形的，所以，矩形是计算机图形学中最基本的操作区域单元。这个结构很简单，x 和 y 是矩形的左上角坐标。x 从左到右增加；y 从上到下增加。左上角的坐标就是(0,0)——SDL 中就是这样的。w 是矩形的宽，h 是矩形的高。

3.2：进一步了解SDL_BlitSurface()。

```
int SDL_BlitSurface(SDL_Surface *src, SDL_Rect *srcrect, SDL_Surface *dst, SDL_Rect *dstre
ct);
```

4 个参数都是指针——2 个 SDL_Surface 指针，2 个 SDL_Rect 指针。src 是源面，也就是被 blit 的面；dst 是目的面，也就是源面被 blit 到的面。srcrect 是源面上的一个矩形区域，实际上，正是这个矩形区域被 blit，如果是空指针，则整个源面被 blit；dstrect 虽然是个矩形区域指针，但是实际上只用到了这个矩形左上角坐标的数据。所以，实际上，它是源面被 blit 到目的面上的坐标。如果是空指针，则被 blit 到目的面的左上角(0,0)。

3.3：为surface类增加新的方法。

```
class DisplaySurface
{
public:
   bool blit(int at_x, int at_y) const;
   bool blit(int at_x, int at_y, int from_x, int from_y, int w, int h, int delta_x = 2, int delta_y = 2) co
nst;
};
```

我们重载了 blit()方法。

```
bool DisplaySurface::blit(int at_x, int at_y) const
{
   SDL_Rect offset;
   offset.x = at_x;
   offset.y = at_y;

   if ( SDL_BlitSurface(pSurface, 0, pScreen, &offset) < 0 )
      return false;
   else return true;
```

```
}
```
这个方法，让整个源面被 blit 到目的面的(at_x,at_y)坐标上。
```
bool DisplaySurface::blit(int at_x, int at_y,
                          int from_x, int from_y, int w, int h,
                          int delta_x, int delta_y) const
{
   SDL_Rect offset;
   offset.x = at_x - delta_x;
   offset.y = at_y - delta_y;

   SDL_Rect dest;
   dest.x = from_x - delta_x;
   dest.y = from_y - delta_y;
   dest.w = w + delta_x*2;
   dest.h = h + delta_y*2;

   if ( SDL_BlitSurface(pSurface, &dest, pScreen, &offset) < 0 )
      return false;
   else return true;
}
```
这个函数，把源面上，一个左上角坐标是(from_x,from_y)，宽为 w，高为 h 的矩形区域，blit 到目的面的(at_x,at_y)坐标上。

要说明 delta_x 和 delta_y 的作用，我们先思考一个问题：动画效果是如何实现的。

我们假设有个作为背景的 surface，名为 back，我们要让一个名为 front 的 surface 在 back 上动起来。显然，至少有两种方法：

1) 把 front blit 到 back 上，把 back blit 到 screen 上，flit screen，显示出原来的图像；
   把 back 面“搽干净”；改变 front 的坐标，将改变坐标后的 front blit 到 back 上，把 back blit 到 screen 上，flit screen，显示出更新后的图像。
2) 把 back blit 到 screen 上，flit screen，首先显示出 back 图像；
   先把 back 面“搽干净”；把 front blit 到 screen 上（back 是一直保持在 screen 上的，而 front 会被 blit 到 back 的上层），flit screen，显示出更新后的图像。

因为，第二种方法把所有的 surface 都直接往 screen 上 blit，思路更为简单，所以，我们先讨论这种方法。

而对于“搽干净”这个概念，又有两种思路：

1) 全部 back 更新；
2) 更新 back 上“被弄脏”的部分。

实际上，当前的电脑速度对在平面上的 blit 速度问题已经不再是问题了。但是，在不久之前，程序员们还在为了计算机图形的实现速度而绞尽脑汁。blit 一部分应该是比 blit 全部图像要快的。所以，这个重载的 blit()方法多用于对于 back 的 blit。delta_x 和 delta_y 是为了保证 blit 的 back 部分，比 front 大那么一点点。不然的话——实际上大家可以把 delta_x 和 delta_y 设置为 0 看看是什么效果。

4、让图片动起来！

4.1：再讨论简单的SDL event响应。

Uint8 *SDL_GetKeyState(int *numkeys);

要让图片动起来，最好是我们可以“操作”的动。按照一般思路，键盘的“上”“下”“左”“右”是不错的选择。在 FPS 游戏和模拟器中，我们可能更习惯 wsad 四个键，所以，让他们同时起作用吧。这个函数的一般用法，是把参数设置为空指针。我们还是先忽略细节。因为有了两个新的 blit()重载方法的加入，我们设置要移动的图片在 screen 上的坐标是(xpos,ypos)，则：

```
//key event for up, down, left and right.
Uint8* keys;
//moving image's coordinate.
int xpos = 0;
int ypos = 0;
```

控制图片移动的代码如下：

```
//key event to move image.
keys = SDL_GetKeyState(0);
if ( keys[SDLK_UP] || keys[SDLK_w] ){
   ypos -= 1;
}
if ( keys[SDLK_DOWN]|| keys[SDLK_s] ){
   ypos += 1;
}
if ( keys[SDLK_LEFT]|| keys[SDLK_a] ){
   xpos -= 1;
}
if ( keys[SDLK_RIGHT]|| keys[SDLK_d] ){
   xpos += 1;
}
```

代码一目了然，不用多解释。具体对应的按键可参考：

http://www.libsdl.org/cgi/docwiki.cgi/SDLKey

4.2：对于第二种方法的分析。

我们前面讨论了总是把 surface blit 到 screen 上的情况。如果我们的思路是把其他 surface 先 blit 到一个作为背景画布的 surface 上，最后把背景画布 blit 到 screen 上呢？

我们将遇到的第一个疑问是，surface 能不能把自己 blit 到自己上面？试验的结果是否定的，而 SDL 并没有给出官方的异常信息，即 SDL_GetError()没有起作用。所以，异常得我们自己来抛出。另外一个后果是，我们必须建立背景画布的拷贝，把拷贝 blit 到背景画布上才是可行的。

```
class DisplaySurface
{
```

```
public:
   bool blitToSurface(const DisplaySurface& dst_surface,
                      int at_x = 0, int at_y = 0) const;
   bool blitToSurface(const DisplaySurface& dst_surface,
                      int at_x, int at_y,
                      int from_x, int from_y, int w, int h,
                      int delta_x = 2, int delta_y = 2) const;
};
```

这和 blit 到 screen 是类似的，但是实际效果是完全不一样的。因为，没有被 blit 到 screen 上的 surface，都不可能被 flip 显示出来。所以，虽然参数列表是不一样的，意味着我可以继续用 blit()这个名字重载。但是为了表明他们的实际效果有区别，我重新用了方法名。

```
bool DisplaySurface::blitToSurface(const DisplaySurface& dst_surface, int at_x, int at_y) const
{
   SDL_Rect offset;
   offset.x = at_x;
   offset.y = at_y;

   if ( &dst_surface == this )
      throw "Cannot blit surface to itself!";

   if ( SDL_BlitSurface(pSurface, 0, dst_surface.point(), &offset) < 0 )
      return false;
   else return true;
}

bool DisplaySurface::blitToSurface(const DisplaySurface& dst_surface,
                                   int at_x, int at_y,
                                   int from_x, int from_y, int w, int h,
                                   int delta_x, int delta_y) const
{
   SDL_Rect offset;
   offset.x = at_x - delta_x;
   offset.y = at_y - delta_y;

   SDL_Rect dest;
   dest.x = from_x - delta_x;
   dest.y = from_y - delta_y;
   dest.w = w + delta_x*2;
   dest.h = h + delta_y*2;

   if ( &dst_surface == this )
      throw "Cannot blit surface to itself!";

   if ( SDL_BlitSurface(pSurface, &dest, dst_surface.point(), &offset) < 0 )
```

```
        return false;
    else return true;
}
```

5、本章范例的完整源代码

5.1：准备工作。

一张 640*480 大小的 bmp 文件作为背景，命名为：bg.bmp；
一张 128*128 大小的 bmp 文件作为要在背景上移动的图片，命名为：image.bmp。

5.2：头文件SurfaceClass.h

```
//FileName: SurfaceClass.h

#ifndef SURFACE_CLASS_H
#define SURFACE_CLASS_H

#include <iostream>
#include <string>
#include "SDL/SDL.h"

using std::string;

class ScreenSurface
{
private:
    static int screenNum;
    int width;
    int height;
    int bpp;
    Uint32 flags;
    SDL_Surface* pScreen;
public:
    ScreenSurface();
    ScreenSurface(int w, int h, int b = 0, Uint32 f = 0);
    ~ScreenSurface();
    SDL_Surface* point() const;
    bool flip() const;
};

class DisplaySurface
```

```
{
private:
   string fileName;
   SDL_Surface* pSurface;
   SDL_Surface* pScreen;
public:
   DisplaySurface(string file_name, const ScreenSurface& screen);
   ~DisplaySurface();
   SDL_Surface* point() const;
   bool blit() const;
   bool blit(int at_x, int at_y) const;
   bool blit(int at_x, int at_y,
             int from_x, int from_y, int w, int h,
             int delta_x = 2, int delta_y = 2) const;
   bool blitToSurface(const DisplaySurface& dst_surface,
                      int at_x = 0, int at_y = 0) const;
   bool blitToSurface(const DisplaySurface& dst_surface,
                      int at_x, int at_y,
                      int from_x, int from_y, int w, int h,
                      int delta_x = 2, int delta_y = 2) const;
};

#endif
```

5.3：类方法的实现文件：SurfaceClass.cpp

```
#include "SurfaceClass.h"

//VVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVV
//class ScreenSurface

int ScreenSurface::screenNum = 0;

ScreenSurface::ScreenSurface():
width(640), height(480), bpp(32), flags(0)
{
   if ( screenNum > 0 )
      throw "DONOT create more than ONE screen!";
   if ( SDL_Init(SDL_INIT_VIDEO ) < 0  )
      throw SDL_GetError();
   pScreen = SDL_SetVideoMode(width, height, bpp, flags);
   screenNum++;
}
```

```
ScreenSurface::ScreenSurface(int w, int h, int b, Uint32 f):
width(w), height(h), bpp(b), flags(f)
{
   if ( screenNum > 0 )
      throw "DONOT create more than ONE screen!";
   if ( SDL_Init(SDL_INIT_VIDEO < 0 ) )
      throw SDL_GetError();
   pScreen = SDL_SetVideoMode(width, height, bpp, flags);
   screenNum++;
}

ScreenSurface::~ScreenSurface()
{
   SDL_Quit();
}

SDL_Surface* ScreenSurface::point() const
{
   return pScreen;
}

bool ScreenSurface::flip() const
{
   if ( SDL_Flip(pScreen) < 0 )
      return false;
   else return true;
}

//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

//VVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVV
//class DisplaySurface

DisplaySurface::DisplaySurface(std::string file_name, const ScreenSurface& screen):
fileName(file_name)
{
   pSurface = SDL_LoadBMP(file_name.c_str());
   if ( pSurface == 0 )
      throw SDL_GetError();
   pScreen = screen.point();
}

DisplaySurface::~DisplaySurface()
{
```

```
   SDL_FreeSurface(pSurface);
}

SDL_Surface* DisplaySurface::point() const
{
   return pSurface;
}

bool DisplaySurface::blit() const
{
   if ( SDL_BlitSurface(pSurface, 0, pScreen, 0) < 0 )
      return false;
   else return true;
}


bool DisplaySurface::blit(int at_x, int at_y) const
{
   SDL_Rect offset;
   offset.x = at_x;
   offset.y = at_y;

   if ( SDL_BlitSurface(pSurface, 0, pScreen, &offset) < 0 )
      return false;
   else return true;
}

bool DisplaySurface::blit(int at_x, int at_y,
                          int from_x, int from_y, int w, int h,
                          int delta_x, int delta_y) const
{
   SDL_Rect offset;
   offset.x = at_x - delta_x;
   offset.y = at_y - delta_y;

   SDL_Rect dest;
   dest.x = from_x - delta_x;
   dest.y = from_y - delta_y;
   dest.w = w + delta_x*2;
   dest.h = h + delta_y*2;

   if ( SDL_BlitSurface(pSurface, &dest, pScreen, &offset) < 0 )
      return false;
   else return true;
```

```
}

bool DisplaySurface::blitToSurface(const DisplaySurface& dst_surface, int at_x, int at_y) const
{
   SDL_Rect offset;
   offset.x = at_x;
   offset.y = at_y;

   if ( &dst_surface == this )
      throw "Cannot blit surface to itself!";

   if ( SDL_BlitSurface(pSurface, 0, dst_surface.point(), &offset) < 0 )
      return false;
   else return true;
}

bool DisplaySurface::blitToSurface(const DisplaySurface& dst_surface,
                                   int at_x, int at_y,
                                   int from_x, int from_y, int w, int h,
                                   int delta_x, int delta_y) const
{
   SDL_Rect offset;
   offset.x = at_x - delta_x;
   offset.y = at_y - delta_y;

   SDL_Rect dest;
   dest.x = from_x - delta_x;
   dest.y = from_y - delta_y;
   dest.w = w + delta_x*2;
   dest.h = h + delta_y*2;

   if ( &dst_surface == this )
      throw "Cannot blit surface to itself!";

   if ( SDL_BlitSurface(pSurface, &dest, dst_surface.point(), &offset) < 0 )
      return false;
   else return true;
}

//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

5.4：演示文件main.cpp

```
#include "SurfaceClass.h"
```

```
void game();
int main(int argc ,char* argv[])
{
   try {
      game();
   }
   catch ( const char* s ) {
      std::cerr << s << std::endl;
      return -1;
   }

   return 0;
}

void game()
{
   //Create a SDL screen.
   const int SCREEN_WIDTH = 640;
   const int SCREEN_HEIGHT = 480;
   ScreenSurface screen(SCREEN_WIDTH, SCREEN_HEIGHT);

   //Create 2 SDL surface for the screen that just created.
   DisplaySurface backGround("bg.bmp", screen);
   DisplaySurface frontImage("image.bmp", screen);
   //VVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVV
   //way2: If use blitToSurface, must get a copy of backGround.
/*
   DisplaySurface backGroundCopy("bg.bmp", screen);
*/
   //AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

   //Blit backGround surface to screen and flip the screen.
   if ( backGround.blit() == false )
      throw SDL_GetError();
   if ( screen.flip() == false )
      throw SDL_GetError();

   //variable for main loop.
   //key event for up, down, left and right.
   Uint8* keys;
   //moving image's coordinate.
   int xpos = 0;
   int ypos = 0;
```

```
//moving image's size.
const int IMG_WIDTH = 128;
const int IMG_HEIGHT = 128;
//main loop.
bool gameOver = false;
while( gameOver == false ){
   //press ESC or click X to quit.
   SDL_Event gameEvent;
   while ( SDL_PollEvent(&gameEvent) != 0 ){
      if ( gameEvent.type == SDL_QUIT ){
         gameOver = true;
      }
      if ( gameEvent.type == SDL_KEYUP ){
         if ( gameEvent.key.keysym.sym == SDLK_ESCAPE ){
            gameOver = true;
         }
      }
   }
   //key event to move image.
   keys = SDL_GetKeyState(0);
   if ( keys[SDLK_UP] || keys[SDLK_w] ){
      ypos -= 1;
   }
   if ( keys[SDLK_DOWN]|| keys[SDLK_s] ){
      ypos += 1;
   }
   if ( keys[SDLK_LEFT]|| keys[SDLK_a] ){
      xpos -= 1;
   }
   if ( keys[SDLK_RIGHT]|| keys[SDLK_d] ){
      xpos += 1;
   }

   //Hold moving image on the backGround area.
   if ( xpos <= 0 )
      xpos = 0;
   if ( xpos >= SCREEN_WIDTH - IMG_WIDTH )
      xpos = SCREEN_WIDTH - IMG_WIDTH;
   if ( ypos <= 0 )
      ypos = 0;
   if ( ypos >= SCREEN_HEIGHT - IMG_HEIGHT )
      ypos = SCREEN_HEIGHT - IMG_HEIGHT;

   //VVVVVVVVVVVVVVVVVVVVVVVVVVVVV
```

```
        //way1: blit surface to screen
        //Blit a part of backGround ( a rectangular area ) to screen,
        //then the original blitted backGround can be "cleaned".
        if ( backGround.blit(xpos, ypos, xpos, ypos, IMG_WIDTH, IMG_HEIGHT) == false )
            throw SDL_GetError();
        //Blit the image to screen.
        if ( frontImage.blit(xpos, ypos) == false )
            throw SDL_GetError();
        //Flip the screen
        if ( screen.flip() == false )
            throw SDL_GetError();
        //AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

        //VVVVVVVVVVVVVVVVVVVVVVVVVVVVVV
        //way2: blit surface to surface, then blit the last surface to screen
/*
        if ( backGroundCopy.blitToSurface(backGround, xpos, ypos, xpos, ypos, IMG_WIDTH, IM
G_HEIGHT) == false )
            throw SDL_GetError();
        if ( frontImage.blitToSurface(backGround, xpos, ypos) == false )
            throw SDL_GetError();
        if ( backGround.blit() == false )
            throw SDL_GetError();
        if ( screen.flip() == false )
            throw SDL_GetError();
*/
        //AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
    }

    return;
}
```

6、对C++异常机制的思考，代码重写

6.1：用bool作为命令是不是画蛇添足了？

不知道为什么，我总觉得总是用 if 结构来调用命令让人读起程序来很不连贯。所以，我决定重新修改下，并且异常抛出改为使用类对象，这样是不是更 C++一点呢？:)

6.2：修改后的代码。

```
//FileName: SurfaceClass.h
```

```
#ifndef SURFACE_CLASS_H
#define SURFACE_CLASS_H

#include <iostream>
#include <string>
#include "SDL/SDL.h"

using std::string;

class ScreenSurface
{
private:
   static int screenNum;
   int width;
   int height;
   int bpp;
   Uint32 flags;
   SDL_Surface* pScreen;
public:
   ScreenSurface();
   ScreenSurface(int w, int h, int b = 0, Uint32 f = 0);
   ~ScreenSurface();
   SDL_Surface* point() const;
   void flip() const;
};

class DisplaySurface
{
private:
   string fileName;
   SDL_Surface* pSurface;
   SDL_Surface* pScreen;
public:
   DisplaySurface(string file_name, const ScreenSurface& screen);
   ~DisplaySurface();
   SDL_Surface* point() const;
   void blit() const;
   void blit(int at_x, int at_y) const;
   void blit(int at_x, int at_y,
             int from_x, int from_y, int w, int h,
             int delta_x = 2, int delta_y = 2) const;
   void blitToSurface(const DisplaySurface& dst_surface,
                      int at_x = 0, int at_y = 0) const;
```

```
    void blitToSurface(const DisplaySurface& dst_surface,
                       int at_x, int at_y,
                       int from_x, int from_y, int w, int h,
                       int delta_x = 2, int delta_y = 2) const;
};

class ErrorInfo
{
private:
    string info;
public:
    ErrorInfo():info("Unknown ERROR!")
    {}
    ErrorInfo(const char* c_str)
    {
        info = string(c_str);
    }
    ErrorInfo(const string& str):info(str)
    {}
    void show() const
    {
        std::cerr << info << std::endl;
    }
};

#endif

#include "SurfaceClass.h"

//VVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVV
//class ScreenSurface

int ScreenSurface::screenNum = 0;

ScreenSurface::ScreenSurface():
width(640), height(480), bpp(32), flags(0)
{
    if ( screenNum > 0 )
        throw ErrorInfo("DONOT create more than ONE screen!");
    if ( SDL_Init(SDL_INIT_VIDEO < 0 ) )
        throw ErrorInfo(SDL_GetError());
    pScreen = SDL_SetVideoMode(width, height, bpp, flags);
    screenNum++;
}
```

```
ScreenSurface::ScreenSurface(int w, int h, int b, Uint32 f):
width(w), height(h), bpp(b), flags(f)
{
   if ( screenNum > 0 )
      throw ErrorInfo("DONOT create more than ONE screen!");
   if ( SDL_Init(SDL_INIT_VIDEO < 0 ) )
      throw ErrorInfo(SDL_GetError());
   pScreen = SDL_SetVideoMode(width, height, bpp, flags);
   screenNum++;
}

ScreenSurface::~ScreenSurface()
{
   SDL_Quit();
}

SDL_Surface* ScreenSurface::point() const
{
   return pScreen;
}

void ScreenSurface::flip() const
{
   if ( SDL_Flip(pScreen) < 0 )
      throw ErrorInfo(SDL_GetError());
}

//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

//VVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVV
//class DisplaySurface

DisplaySurface::DisplaySurface(std::string file_name, const ScreenSurface& screen):
fileName(file_name)
{
   pSurface = SDL_LoadBMP(file_name.c_str());
   if ( pSurface == 0 )
      throw ErrorInfo(SDL_GetError());
   pScreen = screen.point();
}

DisplaySurface::~DisplaySurface()
{
```

```
    SDL_FreeSurface(pSurface);
}

SDL_Surface* DisplaySurface::point() const
{
    return pSurface;
}

void DisplaySurface::blit() const
{
    if ( SDL_BlitSurface(pSurface, 0, pScreen, 0) < 0 )
        throw ErrorInfo(SDL_GetError());
}


void DisplaySurface::blit(int at_x, int at_y) const
{
    SDL_Rect offset;
    offset.x = at_x;
    offset.y = at_y;

    if ( SDL_BlitSurface(pSurface, 0, pScreen, &offset) < 0 )
        throw ErrorInfo(SDL_GetError());
}

void DisplaySurface::blit(int at_x, int at_y,
                          int from_x, int from_y, int w, int h,
                          int delta_x, int delta_y) const
{
    SDL_Rect offset;
    offset.x = at_x - delta_x;
    offset.y = at_y - delta_y;

    SDL_Rect dest;
    dest.x = from_x - delta_x;
    dest.y = from_y - delta_y;
    dest.w = w + delta_x*2;
    dest.h = h + delta_y*2;

    if ( SDL_BlitSurface(pSurface, &dest, pScreen, &offset) < 0 )
        throw ErrorInfo(SDL_GetError());
}

void DisplaySurface::blitToSurface(const DisplaySurface& dst_surface, int at_x, int at_y) const
```

```
{
   SDL_Rect offset;
   offset.x = at_x;
   offset.y = at_y;

   if ( &dst_surface == this )
      throw ErrorInfo("Cannot blit surface to itself!");

   if ( SDL_BlitSurface(pSurface, 0, dst_surface.point(), &offset) < 0 )
      throw ErrorInfo(SDL_GetError());
}

void DisplaySurface::blitToSurface(const DisplaySurface& dst_surface,
                                   int at_x, int at_y,
                                   int from_x, int from_y, int w, int h,
                                   int delta_x, int delta_y) const
{
   SDL_Rect offset;
   offset.x = at_x - delta_x;
   offset.y = at_y - delta_y;

   SDL_Rect dest;
   dest.x = from_x - delta_x;
   dest.y = from_y - delta_y;
   dest.w = w + delta_x*2;
   dest.h = h + delta_y*2;

   if ( &dst_surface == this )
      throw ErrorInfo("Cannot blit surface to itself!");

   if ( SDL_BlitSurface(pSurface, &dest, dst_surface.point(), &offset) < 0 )
      throw ErrorInfo(SDL_GetError());
}

//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

#include "SurfaceClass.h"

int game(int argc, char* argv[]);
int main(int argc ,char* argv[])
{
   int mainRtn = 0;
   try {
      mainRtn = game(argc, argv);
```

```
    }
    catch ( const ErrorInfo& info ) {
        info.show();
        return -1;
    }

    return mainRtn;
}

int game(int argc ,char* argv[])
{
    //Create a SDL screen.
    const int SCREEN_WIDTH = 640;
    const int SCREEN_HEIGHT = 480;
    ScreenSurface screen(SCREEN_WIDTH, SCREEN_HEIGHT);

    //Create 2 SDL surface for the screen that just created.
    DisplaySurface backGround("bg.bmp", screen);
    DisplaySurface frontImage("image.bmp", screen);
    //VVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVV
    //way2: If use blitToSurface, must get a copy of backGround.
/*
    DisplaySurface backGroundCopy("bg.bmp", screen);
*/
    //AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

    //Blit backGround surface to screen and flip the screen.
    backGround.blit();
    screen.flip();

    //variable for main loop.
    //key event for up, down, left and right.
    Uint8* keys;
    //moving image's coordinate.
    int xpos = 0;
    int ypos = 0;
    //moving image's size.
    const int IMG_WIDTH = 128;
    const int IMG_HEIGHT = 128;
    //main loop.
    bool gameOver = false;
    while( gameOver == false ){
        //press ESC or click X to quit.
        SDL_Event gameEvent;
```

```
while ( SDL_PollEvent(&gameEvent) != 0 ){
   if ( gameEvent.type == SDL_QUIT ){
      gameOver = true;
   }
   if ( gameEvent.type == SDL_KEYUP ){
      if ( gameEvent.key.keysym.sym == SDLK_ESCAPE ){
         gameOver = true;
      }
   }
}
//key event to move image.
keys = SDL_GetKeyState(0);
if ( keys[SDLK_UP] || keys[SDLK_w] ){
   ypos -= 1;
}
if ( keys[SDLK_DOWN]|| keys[SDLK_s] ){
   ypos += 1;
}
if ( keys[SDLK_LEFT]|| keys[SDLK_a] ){
   xpos -= 1;
}
if ( keys[SDLK_RIGHT]|| keys[SDLK_d] ){
   xpos += 1;
}

//Hold moving image on the backGround area.
if ( xpos <= 0 )
   xpos = 0;
if ( xpos >= SCREEN_WIDTH - IMG_WIDTH )
   xpos = SCREEN_WIDTH - IMG_WIDTH;
if ( ypos <= 0 )
   ypos = 0;
if ( ypos >= SCREEN_HEIGHT - IMG_HEIGHT )
   ypos = SCREEN_HEIGHT - IMG_HEIGHT;

//VVVVVVVVVVVVVVVVVVVVVVVVVVVVV
//way1: blit surface to screen
//Blit a part of backGround ( a rectangular area ) to screen,
//then the original blitted backGround can be "cleaned".
backGround.blit(xpos, ypos, xpos, ypos, IMG_WIDTH, IMG_HEIGHT);
//Blit the image to screen.
frontImage.blit(xpos, ypos);
//Flip the screen
screen.flip();
```

```
        //AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

        //VVVVVVVVVVVVVVVVVVVVVVVVVVVVVV
        //way2: blit surface to surface, then blit the last surface to screen
/*
        backGroundCopy.blitToSurface(backGround, xpos, ypos, xpos, ypos, IMG_WIDTH, IMG_H
EIGHT);
        frontImage.blitToSurface(backGround, xpos, ypos);
        backGround.blit();
        screen.flip();
*/
        //AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
    }

    return 0;
}
```

7、鼠标事件演示，代码重用

7.1：演示程序源代码

今天因为一个网上的朋友的请求，做个一个关于鼠标事件的演示程序。实际上，可以完全用到前面我们构造的类和类方法，这里送上主程序，供大家参考。其他两个文件和图片文件均不需要任何改变。

```
#include "SurfaceClass.h"

int game(int argc, char* argv[]);
int main(int argc ,char* argv[])
{
    int mainRtn = 0;
    try {
        mainRtn = game(argc, argv);
    }
    catch ( const ErrorInfo& info ) {
        info.show();
        return -1;
    }

    return mainRtn;
}

int game(int argc ,char* argv[])
{
```

```
//Create a SDL screen.
const int SCREEN_WIDTH = 640;
const int SCREEN_HEIGHT = 480;
ScreenSurface screen(SCREEN_WIDTH, SCREEN_HEIGHT);

//Create 2 SDL surface for the screen that just created.
DisplaySurface backGround("bg.bmp", screen);
DisplaySurface frontImage("image.bmp", screen);

//Blit backGround surface to screen and flip the screen.
backGround.blit();
screen.flip();

//moving image's coordinate.
int xpos = 0;
int ypos = 0;
//mouse's coordinate.
int px = 0;
int py = 0;
//moving image's size.
const int IMG_WIDTH = 128;
const int IMG_HEIGHT = 128;
//main loop.
bool gameOver = false;
while( gameOver == false ){
   //press ESC or click X to quit.
   SDL_Event gameEvent;
   while ( SDL_PollEvent(&gameEvent) != 0 ){
      if ( gameEvent.type == SDL_QUIT ){
         gameOver = true;
      }
      if ( gameEvent.type == SDL_KEYUP ){
         if ( gameEvent.key.keysym.sym == SDLK_ESCAPE ){
            gameOver = true;
         }
      }
      //mouse event
      if ( gameEvent.type == SDL_MOUSEMOTION ) {
         px = gameEvent.motion.x;
         py = gameEvent.motion.y;
      }
   }

   if ( xpos < px )
```

```
            xpos++;
        if ( xpos > px )
            xpos--;
        if ( ypos < py )
            ypos++;
        if ( ypos > py )
            ypos--;

        backGround.blit(xpos, ypos, xpos, ypos, IMG_WIDTH, IMG_HEIGHT);
        frontImage.blit(xpos, ypos);
        screen.flip();
    }

    return 0;
}
```

SDL入门教程（六）：

SDL读取其它格式的图片

1：扩充库（Extension Libraries）

SDL本身所支持的，仅仅是读取bmp格式的图片。要使用其它格式的图片，我们需要使用SDL的扩充库。在下面地址，我们可以下载到相关文件SDL_image-devel-1.2.6-VC8.zip。
http://www.libsdl.org/projects/SDL_image/

与SDL本身的设置一样，将include下的*.h文件拷贝到：
C:\MinGW\include\SDL （MinGW）
C:\Program Files\Microsoft Visual Studio 9.0\VC\include\SDL （VC2008）

将*.lib文件拷贝到：
C:\MinGW\lib （MinGW）
C:\Program Files\Microsoft Visual Studio 9.0\VC\lib （VC2008）

将*.dll文件拷贝到：
C:\WINDOWS\system32

在编译的时候，gcc注意增加共同编译的库文件-lSDL_image，比如，我设置了一个批处理文件g++img.bat内容如下：
g++ -o MySDL.exe main.cpp -lmingw32 -lSDLmain -lSDL -lSDL_image -mwindows

在 VC2008 中，需要在 projec 属性中，Configuration Properties -- Linker -- Input -- Additional Dependencies 下增加 SDL_image.lib。

在程序的头文件中，需要增加：

```
#include "SDL/SDL_image.h"
```

2：更加通用的Display Surface构造函数

我们现在可以回头过来修改我们在SDL入门教程（五）：6、对C++异常机制的思考，代码重写中的Display Surface类的构造函数，使其能够更加通用的读取其它格式的图片。

```
DisplaySurface::DisplaySurface(std::string file_name, const ScreenSurface& screen):
fileName(file_name)
{
   SDL_Surface* pSurfaceTemp = IMG_Load(file_name.c_str());
   if ( pSurfaceTemp == 0 )
      throw ErrorInfo(SDL_GetError());
   pSurface = SDL_DisplayFormat(pSurfaceTemp);
   if ( pSurface == 0 )
      throw ErrorInfo(SDL_GetError());
   SDL_FreeSurface(pSurfaceTemp);
   pScreen = screen.point();
}
```

IMG_Load()可以读取多种格式的图片文件，包括 BMP, PNM, XPM, LBM, PCX, GIF, JPEG, TGA 和 PNG。

3：将图片修改为适合显示的格式

```
SDL_Surface *SDL_DisplayFormat(SDL_Surface *surface);
```

在上面的程序中，我们使用到了函数 SDL_DisplayFormat()。在之前的教程中，我一直没有用到这个函数，是因为我还没有发现用 SDL_LoadBMP()的时候会出现格式兼容性的问题——即使是图片位深与显示不一致。但是使用 IMG_Load()的时候，小小的 bug 出现了。所以，这里我必须使用 SDL_DisplayFormat()，将读取的图片文件转换为适合显示的格式。

如果转换失败，或者内存溢出，这个函数将返回空指针。

SDL入门教程（七）：

SDL抠色（Color Keying）

1：什么是抠色（Color Keying）

我们总是 blit 矩形区域的图片，但是很显然，几乎没有一个游戏的角色图片是矩形的。美工把图片画到一个矩形范围内，如果设定了特定的背景颜色，我们就可以把矩形图片上的角色“抠”下来，相对于背景来说，我们就是把不属于角色的背景颜色扣掉，故称抠色。

我们看看 SDL 抠色函数的原形：

int SDL_SetColorKey(SDL_Surface *surface, Uint32 flag, Uint32 key);

这里有个参数是 Uint32 key，这就是我们要抠掉的颜色。要明白 SDL 如何描述颜色的，我们先看看另外一个函数。

2：RGB映射

Uint32 SDL_MapRGB(SDL_PixelFormat *fmt, Uint8 r, Uint8 g, Uint8 b);

显然，参数 r, g, b 代表了红，绿和蓝。而 fmt 则是代表了这些颜色的格式。我们一般会选择使用作为被抠色的矩形图片的颜色格式。这样的图片是一个 SDL_Surface 结构。

```
typedef struct SDL_Surface {
  Uint32 flags;                    /* Read-only */
  SDL_PixelFormat *format;            /* Read-only */
  int w, h;                        /* Read-only */
  Uint16 pitch;                    /* Read-only */
  void *pixels;                    /* Read-write */
  SDL_Rect clip_rect;                 /* Read-only */
  int refcount;                    /* Read-mostly */

  /* This structure also contains private fields not shown here */

} SDL_Surface;
```

也就是成员数据 format，所以，我们很自然的可以把 RGB 映射看成是 SDL Surface 的一个方法。同样，因为抠色行为也绑定在相应的 Surface 上，所以我们可以想到把这两个函数合起来，作为我们所构建的 SDL Surface 的一个类方法。

请注意成员数据 w 和 h，在之前的程序中，我们直接定义了 frontImage 的大小为常量。我们可以把程序修改得更加健壮一些——让程序自动反馈 frontImage 的大小。

```
  //moving image's size.
  const int IMG_WIDTH = frontImage.point()->w;
  const int IMG_HEIGHT = frontImage.point()->h;
```

3：添加Surface的类方法，抠色

```
class DisplaySurface
{
private:

  //…

public:

  //…

    void colorKey(Uint8 r, Uint8 g, Uint8 b, Uint32 flag = SDL_SRCCOLORKEY);
};
```

其它的成员数据和成员函数不需要做任何的改变。我们只需要增加一个新的类方法

colorKey()。

需要说明的是 flag 位标，它有三种模式：

SDL_SRCCOLORKEY 表示正常抠色；

0 表示清除扣色效果；

SDL_SRCCOLORKEY|SDL_RLEACCEL 表示将扣色后的图片重新编码（通常意味着重复使用时会快些）。

作为背景的颜色，一般选择“无红满绿满蓝”（r=0,g=0xFF,b=0xFF）或者“满红无绿满蓝”（r=0xFF,g=0,b=oxFF）。要直观的了解这两种颜色，最好的方法是直接打开画图程序，用调色版将这两种颜色配出来。（我们这里的例子中使用了“无红满绿满蓝”的背景。）类方法的实现如下：

```
void DisplaySurface::colorKey(Uint8 r, Uint8 g, Uint8 b, Uint32 flag)
{
   Uint32 colorkey = SDL_MapRGB(pSurface->format, r, g, b);
   if ( SDL_SetColorKey(pSurface, flag, colorkey ) < 0 )
      throw ErrorInfo(SDL_GetError());
}
```

SDL 的风格，如果 SDL_SetColorkey()成功则返回 0，否则返回-1。

4：在主程序中使用新的类方法，抠色

因为是类方法，所以使用起来就很直观了。我们在创建需要抠色的 DisplaySurface 对象之后，直接使用类方法就可以了。比如一个使用“无红满绿满蓝”背景的需要抠色的图片 colorkey.bmp，我们使用如下语句就可以轻松实现抠色了。

```
   DisplaySurface frontImage("colorkey.bmp", screen);
   frontImage.colorKey(0, 0xFF, 0xFF);
```

SDL入门教程（八）：

1、裁剪精灵图片（Clip Blitting and Sprite Sheets）

这两天下雨，所以就暂停了寻春的步伐，多写了点教程。  绵阳的春天是美丽的，大家学习和写程序之余，还是应该多看看大自然。给大家推荐我拍的风景照啦，呵呵。http://picasaweb.google.com/firefly.cao

因为不想多上传图片，所以我的例子中的图片文件直接就用Lazy Foo's教程中的附件了。相关资源，大家可以在这里下载：Clip Blitting and Sprite Sheets

1.1：什么是精灵？

如果有很多很多的图片，不想发布游戏的时候包含 n 多小小的图片文件，那么可以考虑把小图片都画在一张大图上，然后“抠”下来用，不也是很方便吗？这张大图就叫精灵图

（Sprite sheet）。每一个抠下来的图片就叫精灵啦。

精灵只是一种技巧，需要的知识，我们前面都已经学习过了。我们稍微修改一下类，就可以很好的演示这个技术实现。

1.2：填充背景的颜色

int SDL_FillRect(SDL_Surface *dst, SDL_Rect *dstrect, Uint32 color);

这个函数是往一个指定 surface 的指定矩形区域中填充指定的颜色。如果指定矩形区域为空指针，则填充整个 surface。值得注意的是，SDL_Surface 结构中包含矩形区域的只读成员数据是 clip_rect。我们为 ScreenSurface 类增加一个方法，默认填充黑色。

```
class ScreenSurface
{
private:

  //…

public:

  //…

  void fillColor(Uint8 r = 0, Uint8 g = 0, Uint8 b = 0) const;
};
```

实现为：

```
void ScreenSurface::fillColor(Uint8 r, Uint8 g, Uint8 b) const
{
   if ( SDL_FillRect(pScreen, 0, SDL_MapRGB(pScreen->format, r, g, b)) < 0 )
      throw ErrorInfo(SDL_GetError());
}
```

1.3：思路

我们先创建一个 ScreenSurface 对象，并将其填充为白色作为背景。装载精灵图之后，先抠掉精灵图的背景色，然后按照区域，分 4 次将 4 个图片小精灵 blit 到背景上。非常简单！需要说明的是，我修改了局部 blit()方法偏移值的默认参数值。因为在之前的章节中，局部 blit()主要用于 blit 部分背景将背景“搽干净”，需要将偏移设为非零；而我们现在使用这个方法的主要目的是抠精灵图片，如果偏移非零的话就会把设定范围之外的图像也抠过来，而这不是我们希望发生的。下节给出完整源代码。

2、裁剪精灵图片的完整源代码

```
 //UVi Soft (2008)
//Long Fei (lf426), E-mail: zbln426@163.com

//FileName: SurfaceClass.h
```

```
#ifndef SURFACE_CLASS_H
#define SURFACE_CLASS_H

#include <iostream>
#include <string>
#include "SDL/SDL.h"
#include "SDL/SDL_image.h"

class ScreenSurface
{
private:
   static int screenNum;
   int width;
   int height;
   int bpp;
   Uint32 flags;
   SDL_Surface* pScreen;
public:
   ScreenSurface();
   ScreenSurface(int w, int h, int b = 0, Uint32 f = 0);
   ~ScreenSurface();
   SDL_Surface* point() const;
   void flip() const;
   void fillColor(Uint8 r = 0, Uint8 g = 0, Uint8 b = 0) const;
};

class DisplaySurface
{
private:
   std::string fileName;
   SDL_Surface* pSurface;
   SDL_Surface* pScreen;
public:
   DisplaySurface(const std::string& file_name, const ScreenSurface& screen);
   ~DisplaySurface();
   SDL_Surface* point() const;
   void blit() const;
   void blit(int at_x, int at_y) const;
   void blit(int at_x, int at_y,
             int from_x, int from_y, int w, int h,
             int delta_x = 0, int delta_y = 0) const;
   void colorKey(Uint8 r = 0, Uint8 g = 0xFF, Uint8 b = 0xFF, Uint32 flag = SDL_SRCCOLORK
EY);
```

```
};

class ErrorInfo
{
private:
   std::string info;
public:
   ErrorInfo():info("Unknown ERROR!")
   {}
   ErrorInfo(const char* c_str)
   {
      info = std::string(c_str);
   }
   ErrorInfo(const std::string& str):info(str)
   {}
   void show() const
   {
      std::cerr << info << std::endl;
   }
};

#endif

//UVi Soft (2008)
//Long Fei (lf426), E-mail: zbln426@163.com

#include "SurfaceClass.h"

//VVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVV
//class ScreenSurface

int ScreenSurface::screenNum = 0;

ScreenSurface::ScreenSurface():
width(640), height(480), bpp(32), flags(0)
{
   if ( screenNum > 0 )
      throw ErrorInfo("DONOT create more than ONE screen!");
   if ( SDL_Init(SDL_INIT_VIDEO < 0 ) )
      throw ErrorInfo(SDL_GetError());
   pScreen = SDL_SetVideoMode(width, height, bpp, flags);
   screenNum++;
}
```

```
ScreenSurface::ScreenSurface(int w, int h, int b, Uint32 f):
width(w), height(h), bpp(b), flags(f)
{
   if ( screenNum > 0 )
      throw ErrorInfo("DONOT create more than ONE screen!");
   if ( SDL_Init(SDL_INIT_VIDEO < 0 ) )
      throw ErrorInfo(SDL_GetError());
   pScreen = SDL_SetVideoMode(width, height, bpp, flags);
   screenNum++;
}

ScreenSurface::~ScreenSurface()
{
   SDL_Quit();
}

SDL_Surface* ScreenSurface::point() const
{
   return pScreen;
}

void ScreenSurface::flip() const
{
   if ( SDL_Flip(pScreen) < 0 )
      throw ErrorInfo(SDL_GetError());
}



void ScreenSurface::fillColor(Uint8 r, Uint8 g, Uint8 b) const
{
    if ( SDL_FillRect(pScreen, 0, SDL_MapRGB(pScreen->format, r, g, b)) < 0 )
       throw ErrorInfo(SDL_GetError());
}

//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

//VVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVV
//class DisplaySurface

DisplaySurface::DisplaySurface(const std::string& file_name, const ScreenSurface& screen):
fileName(file_name)
{
   SDL_Surface* pSurfaceTemp = IMG_Load(file_name.c_str());
   if ( pSurfaceTemp == 0 )
```

```
        throw ErrorInfo(SDL_GetError());
    pSurface = SDL_DisplayFormat(pSurfaceTemp);
    if ( pSurface == 0 )
        throw ErrorInfo(SDL_GetError());
    SDL_FreeSurface(pSurfaceTemp);
    pScreen = screen.point();
}

DisplaySurface::~DisplaySurface()
{
    SDL_FreeSurface(pSurface);
}

SDL_Surface* DisplaySurface::point() const
{
    return pSurface;
}

void DisplaySurface::blit() const
{
    if ( SDL_BlitSurface(pSurface, 0, pScreen, 0) < 0 )
        throw ErrorInfo(SDL_GetError());
}


void DisplaySurface::blit(int at_x, int at_y) const
{
    SDL_Rect offset;
    offset.x = at_x;
    offset.y = at_y;

    if ( SDL_BlitSurface(pSurface, 0, pScreen, &offset) < 0 )
        throw ErrorInfo(SDL_GetError());
}

void DisplaySurface::blit(int at_x, int at_y,
                          int from_x, int from_y, int w, int h,
                          int delta_x, int delta_y) const
{
    SDL_Rect offset;
    offset.x = at_x - delta_x;
    offset.y = at_y - delta_y;

    SDL_Rect dest;
```

```
    dest.x = from_x - delta_x;
    dest.y = from_y - delta_y;
    dest.w = w + delta_x*2;
    dest.h = h + delta_y*2;

    if ( SDL_BlitSurface(pSurface, &dest, pScreen, &offset) < 0 )
        throw ErrorInfo(SDL_GetError());
}

void DisplaySurface::colorKey(Uint8 r, Uint8 g, Uint8 b, Uint32 flag)
{
    Uint32 colorkey = SDL_MapRGB(pSurface->format, r, g, b);
    if ( SDL_SetColorKey(pSurface, flag, colorkey ) < 0 )
        throw ErrorInfo(SDL_GetError());
}

//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

//UVi Soft (2008)
//Long Fei (lf426), E-mail: zbln426@163.com

#include "SurfaceClass.h"

int game(int argc, char* argv[]);
int main(int argc ,char* argv[])
{
    int mainRtn = 0;
    try {
        mainRtn = game(argc, argv);
    }
    catch ( const ErrorInfo& info ) {
        info.show();
        return -1;
    }

    return mainRtn;
}

int game(int argc ,char* argv[])
{
    //Create a SDL screen.
    const int SCREEN_WIDTH = 640;
    const int SCREEN_HEIGHT = 480;
    ScreenSurface screen(SCREEN_WIDTH, SCREEN_HEIGHT);
```

```
    //Fill background with white.(default is black)
    screen.fillColor(0xFF, 0xFF, 0xFF);

    //Load a sprite pic, and colorkey.(default color: R=0,G=0xFF,B=0xFF)
    DisplaySurface sprite("dots.png", screen);
    sprite.colorKey();

    //the 4 dots's coordinate.
    int atX[4] = {0, 540, 0, 540};
    int atY[4] = {0, 0, 380, 380};
    //the 4 dots's clip coordinate.
    int fromX[4] = {0, 100, 0 ,100};
    int fromY[4] = {0, 0, 100, 100};
    //dots's size.
    const int IMG_WIDTH = 100;
    const int IMG_HEIGHT = 100;
    //clip dots.
    for ( int i = 0; i < 4; i++ )
        sprite.blit(atX[i], atY[i], fromX[i], fromY[i], IMG_WIDTH, IMG_HEIGHT);
    //Draw dots and screen.
    screen.flip();

    //press ESC or click X to quit.
    bool gameOver = false;
    SDL_Event gameEvent;
    while( gameOver == false ){
        while ( SDL_PollEvent(&gameEvent) != 0 ){
            if ( gameEvent.type == SDL_QUIT ){
                gameOver = true;
            }
            if ( gameEvent.type == SDL_KEYUP ){
                if ( gameEvent.key.keysym.sym == SDLK_ESCAPE ){
                    gameOver = true;
                }
            }
        }
    }

    return 0;
}
```

SDL入门教程（九）：

1、在SDL图形窗口显示文本

最近几篇教程基本上都是参考着 Lazy Foo 的教程顺序来的。因为我也觉得他的顺序很实用。所不同的是，新的类型我都添加在了之前建立起来的 surface 类的基础之上。所以，如果你觉得单独看这些教程完全搞不明白，最好从头按照顺序来学习。另外，为了复习 C++知识，也为了遵循 C++的理念，我有意的将程序风格向 C++靠拢。如果你更喜欢 C 风格，相信你在其他地方可以找到更适合你的教程。

1.1：一个小细节，SDL窗口的名称

因为涉及到文本的显示了，我们提一个一直以来忽略的问题——SDL 建立起来的窗口的名字。因为我们所建立起来的 Screen Surface 是唯一和特殊的。所以窗口名字这个行为是可以绑定在这个唯一的 Screen Surface 对象上的。SDL 中的相关函数是：

```
void SDL_WM_SetCaption(const char *title, const char *icon);
```

一般 icon 还暂时用不上，我们设置为空指针。我们修改一下 Screen Surface 的数据成员与构造函数。在数据成员里面添加一个 windowName，并且修改构造函数

```
class ScreenSurface
{
private:

   //…

   char* windowName;
public:

  //…

  ScreenSurface(int w, int h, char* window_name = 0, int b = 0, Uint32 f = 0);
};


ScreenSurface::ScreenSurface():
width(640), height(480), bpp(32), flags(0), windowName(0)
{
   if ( screenNum > 0 )
      throw ErrorInfo("DONOT create more than ONE screen!");
   if ( SDL_Init(SDL_INIT_VIDEO < 0 ) )
      throw ErrorInfo(SDL_GetError());
   pScreen = SDL_SetVideoMode(width, height, bpp, flags);
   screenNum++;
}
```

```
ScreenSurface::ScreenSurface(int w, int h, char* window_name, int b, Uint32 f):
width(w), height(h), bpp(b), flags(f)
{
   if ( screenNum > 0 )
      throw ErrorInfo("DONOT create more than ONE screen!");
   if ( SDL_Init(SDL_INIT_VIDEO < 0 ) )
      throw ErrorInfo(SDL_GetError());
   pScreen = SDL_SetVideoMode(width, height, bpp, flags);
   screenNum++;
   if ( window_name != 0 ) {
      windowName = window_name;
      SDL_WM_SetCaption(windowName, 0);
   }
   else
      windowName = 0;
}
```

这样，我们在创建 SceenSurface 的时候，第三个参数如果指定，则可以用字符串表示窗口名称。

1.2：使用*.ttf文件

SDL使用*.ttf文件，仍然需要扩展库的支持。相关的下载和SDL_image的类似，大家可以参考前面的教程。下载地址如下：

http://www.libsdl.org/projects/SDL_ttf/

使用ttf扩展库的程序如下：

(1)装载扩展库：TTF_Init();

(2)打开字库：TTF_OpenFont(const char* ttf_fileName, int ttf_size);

(3)构建显示文本的surface：TTF_RenderText_Solid(TTF_Font* pFont, const char* message, SDL_Color textColor);

(4)显示（blit）文本surface;

(5)关闭字库：TTF_CloseFont();

(6)退出扩展库：TTF_Quit();

(7)释放显示文本的surface：SDL_FreeSurface();

我们考虑下这个TextSurface与之前的DisplaySurface之间的关系，希望通过类将二者有所联系。

1.3：构建TextSurface类

我们分析下 TextSurface 与 DisplaySurface 的关系：他们都依赖于一个 ScreenSurface 对象，至少具有两个一样的私有数据成员 pSurface 和 pScreen；他们有一致的行为 blit()；他们的构造前提条件不同，析构做的“善后”也不一样。

我在水木社区的 CPP 版请教有这样关系的两个类应该是什么关系。有前辈指教说，一个类，用不同的 flag 加以区分。而我并不愿意多增加一个构造函数的参数，所以，我用构

造函数的重载实现构造的不同；用继承类实现方法代码的重用；用继承类的析构函数为 TextSurface 类做额外的析构工作。

考虑到应在第一次建立 TextSurface 对象的时候装载 ttf 扩展库，并在最后一个对象使用完毕后关闭 ttf 扩展库，所以，在基类 DisplaySurface 中添加静态私有成员作为计数器，并添加相应的方法为派生类使用。这些方法，以及专门为派生类创建的基类构造函数，我们并不希望能被外部使用，所以，使用了关键字 proteced。

```
class DisplaySurface
{
private:

   //…

   //for TextSurafce
   static int textNum;
   TTF_Font* pFont;
public:

   //…

protected:
   //for TextSurface
   DisplaySurface(const std::string& msg_name, const std::string& message, const ScreenSurface
& screen,
                  Uint8 r, Uint8 g, Uint8 b,
                  const std::string& ttf_fileName, int ttf_size);
   int tellTextNum() const;
   void reduceTextNum();
   void deleteFontPoint();
};
```

pFont 是 TextSurface 会用到的私有数据，构造基类的时候，直接设置成空指针就可以了。保护成员的实现如下：

```
//for TextSurface
DisplaySurface::DisplaySurface(const std::string& msg_name, const std::string& message, const
ScreenSurface& screen,
                  Uint8 r, Uint8 g , Uint8 b,
                  const std::string& ttf_fileName, int ttf_size):
fileName(msg_name)
{
   if ( textNum == 0 )
      if ( TTF_Init() < 0 )
         throw ErrorInfo("TTF_Init() failed!");

   SDL_Color textColor;
   textColor.r = r;
   textColor.g = g;
   textColor.b = b;
```

```
    pFont = TTF_OpenFont(ttf_fileName.c_str(), ttf_size);
    if ( pFont == 0 )
        throw ErrorInfo("TTF_OpenFont() failed!");

    pSurface = TTF_RenderText_Solid(pFont, message.c_str(), textColor);
    if ( pSurface == 0 )
        throw ErrorInfo("TTF_RenderText_solid() failed!");
    pScreen = screen.point();

    textNum++;
}

int DisplaySurface::tellTextNum() const
{
    return textNum;
}

void DisplaySurface::reduceTextNum()
{
    textNum--;
}

void DisplaySurface::deleteFontPoint()
{
    TTF_CloseFont(pFont);
}
```

有了这些数据成员和方法，我们可以构建 TextSurface 类了。

```
class TextSurface: public DisplaySurface
{
public:
    TextSurface(const std::string& msg_name, const std::string& message, const ScreenSurface& s
creen,
                Uint8 r = 0xFF, Uint8 g = 0xFF, Uint8 b = 0xFF,
                const std::string& ttf_fileName = "lazy.ttf", int ttf_size = 28);
    ~TextSurface();
};
```

可以看到，我们仅仅增添了派生类的构造函数和析构函数，实现如下：

```
//class TextSurface

TextSurface::TextSurface(const std::string& msg_name, const std::string& message, const Screen
Surface& screen,
                Uint8 r, Uint8 g, Uint8 b,
                const std::string& ttf_fileName, int ttf_size):
```

```
DisplaySurface(msg_name, message, screen, r, g, b, ttf_fileName, ttf_size)
{}

TextSurface::~TextSurface()
{
   deleteFontPoint();
   reduceTextNum();
   if ( tellTextNum() == 0 )
      TTF_Quit();
}
```

我们在下节给出完整的代码以及一个用于演示的例子。

2、显示文本的完整代码

注意：ttf字库文件，可以在
C:\WINDOWS\Fonts
下寻找，比如例子中用到的times.ttf；
lazy.ttf请到Lazy Foo的相关教程True Type Fonts后面的示例中获得。

```
//UVi Soft (2008)
//Long Fei (lf426), E-mail: zbln426@163.com

//FileName: SurfaceClass.h

#ifndef SURFACE_CLASS_H
#define SURFACE_CLASS_H

#include <iostream>
#include <string>
#include "SDL/SDL.h"
#include "SDL/SDL_image.h"
#include "SDL/SDL_ttf.h"

class ScreenSurface
{
private:
   static int screenNum;
   int width;
   int height;
   int bpp;
   Uint32 flags;
   SDL_Surface* pScreen;
   char* windowName;
public:
   ScreenSurface();
```

```
    ScreenSurface(int w, int h, char* window_name = 0, int b = 0, Uint32 f = 0);
    ~ScreenSurface();
    SDL_Surface* point() const;
    void flip() const;
    void fillColor(Uint8 r = 0, Uint8 g = 0, Uint8 b = 0) const;
};

class DisplaySurface
{
private:
    std::string fileName;
    SDL_Surface* pSurface;
    SDL_Surface* pScreen;
    //for TextSurafce
    static int textNum;
    TTF_Font* pFont;
public:
    DisplaySurface(const std::string& file_name, const ScreenSurface& screen);
    ~DisplaySurface();
    SDL_Surface* point() const;
    void blit() const;
    void blit(int at_x, int at_y) const;
    void blit(int at_x, int at_y,
              int from_x, int from_y, int w, int h,
              int delta_x = 0, int delta_y = 0) const;
    void colorKey(Uint8 r = 0, Uint8 g = 0xFF, Uint8 b = 0xFF, Uint32 flag = SDL_SRCCOLORK
EY);
protected:
    //for TextSurface
    DisplaySurface(const std::string& msg_name, const std::string& message, const ScreenSurface
& screen,
                   Uint8 r, Uint8 g, Uint8 b,
                   const std::string& ttf_fileName, int ttf_size);
    int tellTextNum() const;
    void reduceTextNum();
    void deleteFontPoint();
};

class TextSurface: public DisplaySurface
{
public:
    TextSurface(const std::string& msg_name, const std::string& message, const ScreenSurface& s
creen,
                Uint8 r = 0xFF, Uint8 g = 0xFF, Uint8 b = 0xFF,
```

```
                const std::string& ttf_fileName = "lazy.ttf", int ttf_size = 28);
    ~TextSurface();
};

class ErrorInfo
{
private:
    std::string info;
public:
    ErrorInfo():info("Unknown ERROR!")
    {}
    ErrorInfo(const char* c_str)
    {
        info = std::string(c_str);
    }
    ErrorInfo(const std::string& str):info(str)
    {}
    void show() const
    {
        std::cerr << info << std::endl;
    }
};

#endif

//UVi Soft (2008)
//Long Fei (lf426), E-mail: zbln426@163.com

#include "SurfaceClass.h"

//VVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVV
//class ScreenSurface

int ScreenSurface::screenNum = 0;

ScreenSurface::ScreenSurface():
width(640), height(480), bpp(32), flags(0), windowName(0)
{
    if ( screenNum > 0 )
        throw ErrorInfo("DONOT create more than ONE screen!");
    if ( SDL_Init(SDL_INIT_VIDEO < 0 ) )
        throw ErrorInfo(SDL_GetError());
    pScreen = SDL_SetVideoMode(width, height, bpp, flags);
    screenNum++;
```

```
}

ScreenSurface::ScreenSurface(int w, int h, char* window_name, int b, Uint32 f):
width(w), height(h), bpp(b), flags(f)
{
   if ( screenNum > 0 )
      throw ErrorInfo("DONOT create more than ONE screen!");
   if ( SDL_Init(SDL_INIT_VIDEO < 0 ) )
      throw ErrorInfo(SDL_GetError());
   pScreen = SDL_SetVideoMode(width, height, bpp, flags);
   screenNum++;
   if ( window_name != 0 ) {
      windowName = window_name;
      SDL_WM_SetCaption(windowName, 0);
   }
   else
      windowName = 0;
}

ScreenSurface::~ScreenSurface()
{
   SDL_Quit();
}

SDL_Surface* ScreenSurface::point() const
{
   return pScreen;
}

void ScreenSurface::flip() const
{
   if ( SDL_Flip(pScreen) < 0 )
      throw ErrorInfo(SDL_GetError());
}



void ScreenSurface::fillColor(Uint8 r, Uint8 g, Uint8 b) const
{
    if ( SDL_FillRect(pScreen, 0, SDL_MapRGB(pScreen->format, r, g, b)) < 0 )
       throw ErrorInfo(SDL_GetError());
}

//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

```
//VVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVV
//class DisplaySurface

int DisplaySurface::textNum = 0;

DisplaySurface::DisplaySurface(const std::string& file_name, const ScreenSurface& screen):
fileName(file_name), pFont(0)
{
   SDL_Surface* pSurfaceTemp = IMG_Load(file_name.c_str());
   if ( pSurfaceTemp == 0 )
      throw ErrorInfo(SDL_GetError());
   pSurface = SDL_DisplayFormat(pSurfaceTemp);
   if ( pSurface == 0 )
      throw ErrorInfo(SDL_GetError());
   SDL_FreeSurface(pSurfaceTemp);
   pScreen = screen.point();
}

DisplaySurface::~DisplaySurface()
{
   SDL_FreeSurface(pSurface);
}

SDL_Surface* DisplaySurface::point() const
{
   return pSurface;
}

void DisplaySurface::blit() const
{
   if ( SDL_BlitSurface(pSurface, 0, pScreen, 0) < 0 )
      throw ErrorInfo(SDL_GetError());
}



void DisplaySurface::blit(int at_x, int at_y) const
{
   SDL_Rect offset;
   offset.x = at_x;
   offset.y = at_y;

   if ( SDL_BlitSurface(pSurface, 0, pScreen, &offset) < 0 )
      throw ErrorInfo(SDL_GetError());
}
```

```
void DisplaySurface::blit(int at_x, int at_y,
                          int from_x, int from_y, int w, int h,
                          int delta_x, int delta_y) const
{
   SDL_Rect offset;
   offset.x = at_x - delta_x;
   offset.y = at_y - delta_y;

   SDL_Rect dest;
   dest.x = from_x - delta_x;
   dest.y = from_y - delta_y;
   dest.w = w + delta_x*2;
   dest.h = h + delta_y*2;

   if ( SDL_BlitSurface(pSurface, &dest, pScreen, &offset) < 0 )
      throw ErrorInfo(SDL_GetError());
}

void DisplaySurface::colorKey(Uint8 r, Uint8 g, Uint8 b, Uint32 flag)
{
   Uint32 colorkey = SDL_MapRGB(pSurface->format, r, g, b);
   if ( SDL_SetColorKey(pSurface, flag, colorkey ) < 0 )
      throw ErrorInfo(SDL_GetError());
}

//for TextSurface
DisplaySurface::DisplaySurface(const std::string& msg_name, const std::string& message, const
ScreenSurface& screen,
                  Uint8 r, Uint8 g , Uint8 b,
                  const std::string& ttf_fileName, int ttf_size):
fileName(msg_name)
{
   if ( textNum == 0 )
      if ( TTF_Init() < 0 )
         throw ErrorInfo("TTF_Init() failed!");

   SDL_Color textColor;
   textColor.r = r;
   textColor.g = g;
   textColor.b = b;

   pFont = TTF_OpenFont(ttf_fileName.c_str(), ttf_size);
   if ( pFont == 0 )
```

```
        throw ErrorInfo("TTF_OpenFont() failed!");

    pSurface = TTF_RenderText_Solid(pFont, message.c_str(), textColor);
    if ( pSurface == 0 )
        throw ErrorInfo("TTF_RenderText_solid() failed!");
    pScreen = screen.point();

    textNum++;
}

int DisplaySurface::tellTextNum() const
{
    return textNum;
}

void DisplaySurface::reduceTextNum()
{
    textNum--;
}

void DisplaySurface::deleteFontPoint()
{
    TTF_CloseFont(pFont);
}

//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

//VVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVV
//class TextSurface

TextSurface::TextSurface(const std::string& msg_name, const std::string& message, const Screen
Surface& screen,
                Uint8 r, Uint8 g, Uint8 b,
                const std::string& ttf_fileName, int ttf_size):
DisplaySurface(msg_name, message, screen, r, g, b, ttf_fileName, ttf_size)
{}

TextSurface::~TextSurface()
{
    deleteFontPoint();
    reduceTextNum();
    if ( tellTextNum() == 0 )
        TTF_Quit();
}
```

```
//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

//UVi Soft (2008)
//Long Fei (lf426), E-mail: zbln426@163.com

#include "SurfaceClass.h"

int game(int argc, char* argv[]);
int main(int argc ,char* argv[])
{
   int mainRtn = 0;
   try {
      mainRtn = game(argc, argv);
   }
   catch ( const ErrorInfo& info ) {
      info.show();
      return -1;
   }

   return mainRtn;
}

int game(int argc ,char* argv[])
{
   //Create a SDL screen.
   const int SCREEN_WIDTH = 640;
   const int SCREEN_HEIGHT = 480;
   ScreenSurface screen(SCREEN_WIDTH, SCREEN_HEIGHT, "Font");
   //Fill background with white.(default is black)
   screen.fillColor(0xFF, 0xFF, 0xFF);

   //Load a textSurface
   TextSurface myText("logo", "UVi Soft", screen, 0, 0, 0xFF, "times.ttf", 80);
   TextSurface lazy("lazy", "by lf426 (zbln426@163.com)", screen, 0xff, 0, 0);
   //Display text
   myText.blit(170, 180);
   lazy.blit(150,400);
   screen.flip();

   //press ESC or click X to quit.
   bool gameOver = false;
   SDL_Event gameEvent;
   while( gameOver == false ){
```

```
    while ( SDL_PollEvent(&gameEvent) != 0 ){
      if ( gameEvent.type == SDL_QUIT ){
        gameOver = true;
      }
      if ( gameEvent.type == SDL_KEYUP ){
        if ( gameEvent.key.keysym.sym == SDLK_ESCAPE ){
          gameOver = true;
        }
      }
    }
  }

  return 0;
}
```

3、文本反馈“按键”信息

3.1：一些小的修改

我觉得写 C++的程序，一是看起来确实比较 C++一点，二是相对于 C 的“精炼”，C++要的是“健壮”。所以，其实我不太满意用 C 风格字符串作为 ScreenSurface 的成员数据，所以做了修改。这也是为了在程序中构建 ScreenSurface 对象的时候可以使用 string。

```
class ScreenSurface
{
private:

  //…

  std::string windowName;
public:

  //…

  ScreenSurface(int w, int h, const std::string& window_name = "NULL", int b = 0, Uint32 f = 0);
};
```

相应的，我们修改了 2 个构造函数。

```
ScreenSurface::ScreenSurface():
width(640), height(480), bpp(32), flags(0), windowName("NULL")
{
  if ( screenNum > 0 )
    throw ErrorInfo("DONOT create more than ONE screen!");
  if ( SDL_Init(SDL_INIT_VIDEO < 0 ) )
    throw ErrorInfo(SDL_GetError());
  pScreen = SDL_SetVideoMode(width, height, bpp, flags);
  screenNum++;
```

```
}

ScreenSurface::ScreenSurface(int w, int h, const std::string& window_name, int b, Uint32 f):
width(w), height(h), bpp(b), flags(f)
{
   if ( screenNum > 0 )
      throw ErrorInfo("DONOT create more than ONE screen!");
   if ( SDL_Init(SDL_INIT_VIDEO < 0 ) )
      throw ErrorInfo(SDL_GetError());
   pScreen = SDL_SetVideoMode(width, height, bpp, flags);
   screenNum++;
   if ( window_name != "NULL" ) {
      windowName = window_name;
      SDL_WM_SetCaption(windowName.c_str(), 0);
   }
   else
      windowName = "NULL";
}
```

第二个地方，我修改了 TextSurface 构造函数的参数顺序，并且将默认的字体改为 Windows 都自带的“新罗马时代”字体 times.ttf。我将字体参数放在最后，将字体大小参数提前了，这样更符合习惯上的使用规律。

```
class TextSurface: public DisplaySurface
{
public:
   TextSurface(const std::string& msg_name, const std::string& message, const ScreenSurface& s
creen,
               Uint8 r = 0xFF, Uint8 g = 0xFF, Uint8 b = 0xFF,
               int ttf_size = 28, const std::string& ttf_fileName = "times.ttf");
   ~TextSurface();
};
```

（在 DisplaySurface 里相应的构造函数也做类似的修改，略）

3.2：回顾SDL事件轮询

SDL_PollEvent()的作用，是事件一旦被触发，就会响应一次，注意它的响应并不是连续不断的。比如你按下某个键，即触发了一次事件。即使你按着不松开，也仅仅是触发了一次，所以 SDL_PollEvent()也只响应一次。

下面的程序，演示键盘事件中，方向键被按下后的反馈信息。

3.3：演示程序

```
//UVi Soft (2008)
//Long Fei (lf426), E-mail: zbln426@163.com

#include "SurfaceClass.h"

int game(int argc, char* argv[]);
int main(int argc ,char* argv[])
{
   int mainRtn = 0;
   try {
      mainRtn = game(argc, argv);
   }
   catch ( const ErrorInfo& info ) {
      info.show();
      return -1;
   }

   return mainRtn;
}

int game(int argc ,char* argv[])
{
   //Create a SDL screen.
   const int SCREEN_WIDTH = 640;
   const int SCREEN_HEIGHT = 480;
   const std::string WINDOW_NAME = "Key Presses";
   ScreenSurface screen(SCREEN_WIDTH, SCREEN_HEIGHT, WINDOW_NAME);
   //Fill background.(default is black)
   screen.fillColor();
   screen.flip();

   //Load a textSurface
   TextSurface upMessage("upMsg", "Up was pressed.", screen);
   TextSurface downMessage("downMsg", "Down was pressed.", screen, 0xFF, 0, 0);
   TextSurface leftMessage("leftMsg", "Left was pressed.", screen, 0, 0xFF, 0);
   TextSurface rightMessage("rightMsg", "Right was pressed.", screen, 0, 0, 0xFF);
   TextSurface otherMessage("otherMsg", "Other key was pressed.", screen, 100, 100, 100, 35);

   //Main loop.Press ESC or click X to quit.
   bool gameOver = false;
   SDL_Event gameEvent;
   int x = 200;
```

```
    int y = 200;
    while( gameOver == false ){
        while ( SDL_PollEvent(&gameEvent) != 0 ){
            if ( gameEvent.type == SDL_KEYDOWN ){
                screen.fillColor();
                switch ( gameEvent.key.keysym.sym ){
                    case SDLK_UP:
                        upMessage.blit(x, y--);
                        break;
                    case SDLK_DOWN:
                        downMessage.blit(x, y++);
                        break;
                    case SDLK_LEFT:
                        leftMessage.blit(x--, y);
                        break;
                    case SDLK_RIGHT:
                        rightMessage.blit(x++, y);
                        break;
                    default:
                        otherMessage.blit(x, y);
                }
                screen.flip();
            }
            if ( gameEvent.type == SDL_QUIT ){
                gameOver = true;
            }
            if ( gameEvent.type == SDL_KEYUP ){
                if ( gameEvent.key.keysym.sym == SDLK_ESCAPE ){
                    gameOver = true;
                }
            }
        }
    }

    return 0;
}
```

4、int转换为std::string

我下面考虑的问题，是用 TextSurface 反馈鼠标事件的信息。我想到的第一个例子，很自然就是反馈鼠标所在坐标的位置。这里涉及到一个基础的问题，即鼠标位置显然不是用字符串表示的。SDL 给我们的反馈信息是 int，我们需要用 TextSurface 将 int 构建成可以被 blit 到 ScreenSurface 上的面，需要做的第一件事情，是将 int 转换为 string。

我的思路是这样的：首先找到 int 的数位数；然后依次从高位读取数字，之后将这个位

去掉（通常减掉是最简单的）；依次记录这些数字，转换成 string，然后将这些数字“加”（字符串的合并）起来。
头文件如下：

```
//UVi Soft (2008)
//Long Fei (lf426), E-mail: zbln426@163.com

//FileName: int_to_string.h

#ifndef INT_TO_STRING_H_
#define INT_TO_STRING_H_

#include <iostream>
#include <string>
#include <vector>

int int_power(int base, int exp);
std::string int_to_string(int num);

#endif
```

其中，int_to_string()是我们需要构建的函数，int_power()是求一个数的整数幂的函数。这么简单的算法，我们就自己写吧。至于用到 vector，按照我的思路，我们需要的数据结构显然应该是“队列”（先进先出）。不过真得感谢 STL，用 vector 显然不是最优化的，但是肯定是最“通俗”的，因为即使是作为非专业的队列（或者栈），vector 也已经为我们提供了必要的方法，比如推入（push_back）。

下面我就把程序说明夹在程序中间了。原因是本人英语太菜，简单描述还能忍，描述算法就有点不能忍了-_-!!!。另外，尽管我英语这水平，我还是希望程序里面别用中文注释的好。这种事情，您见过一次乱码，就总是得恶心一辈子。

```
//UVi Soft (2008)
//Long Fei (lf426), E-mail: zbln426@163.com

#include "int_to_string.h"

int int_power(int base, int exp)
{
   int result = 1;
   for (int i = exp; i > 0; i-- )
      result*=base;
   return result;
}
```

这是个很简单的求幂的算法。其实我们在程序中，只需要用到求 10 的 n 次幂，所以，实际上我们还可以写得更加有针对一点。

```
std::string int_to_string(int num)
{
   bool negative = false;
```

```
if ( num < 0 ){
    negative = true;
    num = -num;
}
```
这开始写转换函数了。首先我们判定 int 是否为负。如果是，我们把它变成其相反数，然后与正数一样转换，最后在前面加上“-”就 OK 了。
```
int bitNum = 1;
for ( int i = num; i > 9; i/=10 )
    bitNum++;
```
bitNum 是这个 int 的数位数。比如 3 就是 1 位，1024 就是 4 位。
```
std::vector<int> eachNum;
for ( int i = bitNum, temp = num; i > 0; i-- ){
    int highBit = int(temp/int_power(10, (i-1)));
    eachNum.push_back(highBit);
    temp-=(highBit*int_power(10, (i-1)));
}
```
我们通过 vector 数组纪录每个数位上的数字，从高位到低位。需要说明的是，n 位的数字是 10 的 n-1 次方幂。比如 1024 是 4 位，而 1000 是 10 的 3 次方幂。所以，我们这里用的是 i-1 而非 i。
```
std::string str;
if ( negative == true )
    str = "-";
for ( std::vector<int>::iterator pTemp = eachNum.begin(); pTemp != eachNum.end(); pTemp++ ){
    switch ( *pTemp ){
        case 0:
            str+="0";
            break;
        case 1:
            str+="1";
            break;
        case 2:
            str+="2";
            break;
        case 3:
            str+="3";
            break;
        case 4:
            str+="4";
            break;
        case 5:
            str+="5";
            break;
        case 6:
```

```
            str+="6";
            break;
        case 7:
            str+="7";
            break;
        case 8:
            str+="8";
            break;
        case 9:
            str+="9";
            break;
        default:
            break;
    }
  }
  return str;
}
```

最后，我们用了 STL 的方法将每个数字转换成 std::string 的字符串，然后将这些字符串合并起来，作为函数的返回值。

我们在下一节中将用 TextSurface 演示这个函数的作用，以及实现我们在本节前面所提出的问题。

5、文本反馈鼠标位置坐标信息

注意事项：

1、times.ttf 文件请到 C:\WINDOWS\Fonts 下寻找并拷贝到资源目录下。

2、如果您使用 VC2008，请用 Release 编译。原因是，似乎涉及到 vector 的操作，Runtime Library 在 debug 的时候必须用 Multi-theaded Debug DLL (MDd)，而 Release 时候才用 Multi-theaded DLL (MD)。而我们亲爱的 SDL 必须始终用 MD，所以，请 Release 吧。

这是一个检验 TextSurface 构造函数和析构函数的好例子。我们每一次的鼠标移动，都会导致一个新的 TextSurface 对象被建立，然后很快的又消亡。

源代码：

```
//UVi Soft (2008)
//Long Fei (lf426), E-mail: zbln426@163.com

#include "SurfaceClass.h"
#include "int_to_string.h"

int game(int argc, char* argv[]);
int main(int argc ,char* argv[])
{
  int mainRtn = 0;
```

```
    try {
        mainRtn = game(argc, argv);
    }
    catch ( const ErrorInfo& info ) {
        info.show();
        return -1;
    }

    return mainRtn;
}

int game(int argc ,char* argv[])
{
    //Create a SDL screen.
    const int SCREEN_WIDTH = 640;
    const int SCREEN_HEIGHT = 480;
    const std::string WINDOW_NAME = "Mouse Motion";
    ScreenSurface screen(SCREEN_WIDTH, SCREEN_HEIGHT, WINDOW_NAME);
    //Fill background.(default is black)
    screen.fillColor();
    screen.flip();

    //Main loop.Press ESC or click X to quit.
    bool gameOver = false;
    SDL_Event gameEvent;
    int x;
    int y;
    while( gameOver == false ){
        while ( SDL_PollEvent(&gameEvent) != 0 ){
            if ( gameEvent.type == SDL_MOUSEMOTION ) {
                x = gameEvent.motion.x;
                y = gameEvent.motion.y;
                std::string mouse_at = "Mouse at: ";
                mouse_at += ("x = " + int_to_string(x) + "; y = " + int_to_string(y) + ";");
                TextSurface text("text", mouse_at, screen);
                screen.fillColor();
                text.blit();
                screen.flip();
            }
            if ( gameEvent.type == SDL_QUIT ){
                gameOver = true;
            }
            if ( gameEvent.type == SDL_KEYUP ){
                if ( gameEvent.key.keysym.sym == SDLK_ESCAPE ){
```

```
                gameOver = true;
            }
        }
    }
}

    return 0;
}
```

SDL入门教程（十）：

1、多语言支持，Win32 下的GetText

自从开始研究SDL的文本显示，我就一直在思考在SDL中显示中文的问题。我们知道韦诺之战（Battle for Wesnoth）使用SDL开发的，并且支持多语言。所以，我一直相信Wesnoth的源代码里面一定有我所需要的答案。网络上是纵说纷纭啊，有些人干脆说，SDL不支持中文；有些人在困难面前回到了MFC的怀抱。而，既然我的目标是跨平台，并且我也相信一定能找到答案，所以，我坚持寻找。终于，完美解决了在SDL中显示中文，甚至多语言的问题。以下的几节，我将全面，详细的说明这些方法。

1.1：po，mo与gettext

线索从Wesnoth的发布游戏与源代码中开始，我们知道，在Wesnoth游戏中，有个名为po的文件夹，多国语言翻译都放在了这个文件夹下面。游戏程序中多为*.mo文件，源代码中多为*.po文件。通过搜索，po与mo的背景浮出水面——它们来自GNU项目gettext。

gettext项目是专门为多语言设计的。我们不需要修改源代码和程序的情况下，可以让程序支持多国语言。程序将根据系统所在的国家和区域选择相应的语言，当然，也可以在执行过程中让玩家自由的选择。既然是开放源代码的，自然也很容易的被移植到win32 下。win32下的这个项目主页如下：

http://gnuwin32.sourceforge.net/packages/gettext.htm

为了方便的使用，我还是建议你下载完整的安装包（Complete package）。然后，你可以看英文说明，也可以凭着直觉去试验，找到哪些库和哪些DLL文件是编译和运行时必须的——当然，我也可以直接告诉你答案。

设置编译环境的问题就不再多说了，不清楚的请看前面的章节。反正都三部分：*.h文件，*.lib文件和*.dll文件，放到相应的文件夹下面并在编译时候指明就可以了。

我们下面将用到的文件有：

libintl.h：请在写源代码的时候#include进来；

libintl.lib：这是编译时候需要的库文件；

libintl3.dll和libiconv2.dll：这是程序运行时候需要的文件，放到*.exe文件可以找到的地方。

1.2：演示程序以及说明

```
#include <iostream>
#include <string>
#include <clocale>
#include "GNU/libintl.h"

int main(int argc, char* argv[])
{
   setlocale(LC_ALL, "");
   bindtextdomain("myText", "E:/My Documents/Visual Studio 2008/po");
   textdomain("myText");
   std::string test = gettext("Hello, World!");
   std::cout << test << std::endl;
   return 0;
}
```

我们先说#include 进来的<clocale>，我用“<>”表示它是标准 C++的一部分。它包含了函数 setlocale()。这个函数在这里的两个参数——常量 LC_ALL 与空字符串""的意思是，在这个程序中的所有语言与区域，都设置为系统默认的语言与区域。

libintl.h 是我们刚才加入的 GNU 的一部分，这意味着在 Linux 系统下，这个头文件是系统本身自带的。它包含了后面三个函数：bindtextdomain()将一个文件夹目录绑定到一个域名上，这个域名也是将来*.mo 文件的文件名；textdomain()表明我们将使用的域名；gettext()中的字符串将是被多语言翻译替换的部分。

将这个程序编译，在没有多语言包的时候，程序也能正常的运行，显示“Hello, World!”。

1.3：为源程序制作po文件和mo文件

如果你已经安装了完整的安装包，找到相关文件夹的 bin 目录，这里有很多工具软件。你可以通过 cmd 的方式一步步的转换，也可以，偷点儿懒，因为有更加现成的工具可以用。但是，第一步，从源代码提取 gettext()的文本，还得靠命令：xgettext。就跟用 g++命令一样，假设我们的源文件名是 main.cpp，我们把它先转换成一个模板文件 a.pot：

```
xgettext -o a.pot main.cpp
```

你可以用 vim 之类的文本编辑器看看*.pot 文件的内容，你会发现，一些说明，以及提取文本的详细信息被纪录了下来。

```
# SOME DESCRIPTIVE TITLE.
# Copyright (C) YEAR THE PACKAGE'S COPYRIGHT HOLDER
# This file is distributed under the same license as the PACKAGE package.
# FIRST AUTHOR <EMAIL@ADDRESS>, YEAR.
#
#, fuzzy
msgid ""
msgstr ""
"Project-Id-Version: PACKAGE VERSION\n"
```

"Report-Msgid-Bugs-To: \n"
"POT-Creation-Date: 2008-03-30 00:24+0800\n"
"PO-Revision-Date: YEAR-MO-DA HO:MI+ZONE\n"
"Last-Translator: FULL NAME <EMAIL@ADDRESS>\n"
"Language-Team: LANGUAGE <LL@li.org>\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=CHARSET\n"
"Content-Transfer-Encoding: 8bit\n"

#: main.cpp:11
msgid "Hello, World!"
msgstr ""
下面，我们使用一个简单的小工具poedit。又一个跨平台的软件，主页在：
http://www.poedit.net/
安装运行后，选择“从POT文件更新类目”，然后打开我们刚才的a.pot，什么都不用修改（当然，你也可以把自己信息都写上去），确保“字符集”是UTF-8 就可以了。然后，在英语下面也上替换的文字吧，保存的时候，相应的mo文件也就建立起来了。
msgid ""
msgstr ""
"Project-Id-Version: \n"
"Report-Msgid-Bugs-To: \n"
"POT-Creation-Date: 2008-03-30 00:24+0800\n"
"PO-Revision-Date: 2008-03-30 00:25+0800\n"
"Last-Translator: lf426 <zbln426@163.com>\n"
"Language-Team: \n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=UTF-8\n"
"Content-Transfer-Encoding: 8bit\n"

#: main.cpp:11
msgid "Hello, World!"
msgstr "浜茬埍鐨勪笘鐣岋紝鎴戞潵浜嗐紒"
这是 po 文件。怎么是乱码？那是因为 windows 不是用 UTF-8 保存的文本文件（默认一般是GB2312）。用 poedit 打开时候是正确显示的。我的文本内容是：“亲爱的世界，我来了！”。如果你用的是 vim，可以通过设置环境变量解决显示乱码的问题，在_vimrc 文件中添加这一句：
set fileencodings=gb2312,ucs-bom,utf-8,chinese

1.4：设置mo文件的目录

下面的工作可能就有些教条了。还记得我们绑定域名的路径吧，我用的是
E:\My Documents\Visual Studio 2008\po
（请注意在 C++程序里面把斜杠反过来！）
*.mo 文件并不是直接放到这个路径下，而是这个路径下的./LL/LC_MESSAGES 或

者./LL_CC/LC_MESSAGES。其中 LL 表示语种，CC 表示国家或区域。具体的请参考 Wesnoth。就我们的中文来说，这个例子放 mo 文件的路径是：
E:\My Documents\Visual Studio 2008\po\zh_CN\LC_MESSAGES

现在运行程序就可以看到文本已经被替换了。如果我们删除 mo 文件或修改 mo 文件名（与绑定域名不一致），程序会继续显示原来的英文。如果我们改变系统环境，只要不是中国中文，程序都还是显示英文。如果我们要更新替换内容，直接用 poedit 更新 po 和 mo 文件就可以了。

1.5：构建StringData类

我们希望字符串的数据单独的保存在一个文件里，这样既方便被 gettext 提取，也方便修改。而且，在程序里面，我们尽量把 gettext 涉及到的一些特殊的设置隐藏了。所以，我们构建 StringDada 类，在程序中需要用到的地方，直接调用它的对象就可以了。

```
//FileName: string_data.h
#ifndef STRING_DATA_H
#define STRING_DATA_H

#include <clocale>
#include <string>
#include <vector>
#include "GNU/libintl.h"

class StringData
{
private:
   std::vector<std::string> data;
public:
   StringData();
   std::string operator [](const unsigned int& n) const;
};

#endif
```

我重载了[]，这样在调用数据的时候更加直观。我们将数据都写在 StringData 的构造函数中，将来 gettext 也只需要提取 StringData 的实现文件就可以了。

```
#include "string_data.h"

StringData::StringData()
{
   setlocale(LC_ALL, "");
   bindtextdomain("StringData", "./po");
   textdomain("StringData");

   //0
   data.push_back(gettext("Up was pressed."));
```

```
    //1
    data.push_back(gettext("Down was pressed."));
    //2
    data.push_back(gettext("Left was pressed."));
    //3
    data.push_back(gettext("Right was pressed."));
    //4
    data.push_back(gettext("Other key was pressed."));
}

std::string StringData::operator [](const unsigned int& n) const
{
    if ( n >= data.size() )
        return 0;
    return data[n];
}
```

1.6：做个gettext的批处理

如果你按照我全面介绍的，安装了 Poedit，也安装了 GnuWin32，那么，我们做个批处理文件让从 string_data.cpp 到 StringData.mo 的转换更加简单吧。（如果安装路径不一样请做相应的修改）。

```
@set path=C:\Program Files\GnuWin32\bin;%PATH%;
xgettext --force-po -o string_data.pot string_data.cpp
msginit -l zh_CN -o StringData.po -i string_data.pot
@set path=C:\Program Files\Poedit\bin;%PATH%;
poedit StringData.po
del string_data.pot
del StringData.po
```

Poedit 打开 StringData.po 的时候会报错，那是因为文件指明的编码不可用，请在“字符集”中选择 UTF-8，另外，在“工程名称以及版本”中写点信息，不要使用默认值就可以了。然后翻译并保存，StringData.mo 文件就生成了。

2、直接通过Unicode让SDL显示中文

2.1：SDL本身可以显示中文吗？

SDL 的扩展库 SDL_ttf 本身具备显示中文的功能吗？网上很多观点，说不能显示的，甚至做了分析解释了原因。但是，事实是，SDL 本身就可以显示中文。如果我们看看 SDL_ttf.c 的源代码，我们可以看到，最终用于构建 SDL_Surface 平面的函数，在三种显示模式（Solid, Shaded, Blended）下，都是其对应的 TTF_RenderUNICODE_Xxx() 函数。我们以 TTF_RenderUNICODE_Blended()为例：

SDL_Surface *TTF_RenderUNICODE_Blended(TTF_Font *font, const Uint16 *text, SDL_Color fg)

可以看到，Unicode 码是通过 Uint16 的数组传递的。在官方文档中，采用这样的形式：

```
// Render some UNICODE text in blended black to a new surface
// then blit to the upper left of the screen
// then free the text surface
//SDL_Surface *screen;
SDL_Color color={0,0,0};
SDL_Surface *text_surface;
Uint16 text[]={'H','e','l','l','o',' ',
            'W','o','r','l','d','!'};
if(!(text_surface=TTF_RenderUNICODE_Blended(font,text,color))) {
   //handle error here, perhaps print TTF_GetError at least
} else {
   SDL_BlitSurface(text_surface,NULL,screen,NULL);
   //perhaps we can reuse it, but I assume not for simplicity.
   SDL_FreeSurface(text_surface);
}
```

请注意对 Uint16 text[]的定义，既然是 Uint16 的数组，也意味着我们可以直接用数字作为这个数组的元素。

2.2：获得字符串的Unicode码

SDL的不能正确显示中文的问题，首先出在SDL_ttf没有提供正确渲染中文的函数，或者再精确点说，没有提供正确渲染GB2312 码的函数。SDL_ttf提供了渲染UTF-8 的函数，但是很不幸，汉字信息并不是通过UTF-8 传递给程序的——即使是我们在po文件中指明了使用UTF-8，程序调用的时候依然被转化成了GB2312——至少在win32 下是这样。

所以，显示中文最简单的办法——同时也是最复杂的实现，即直接给渲染Unicode的SDL_ttf函数传递汉字字符串的Unicode码。我们可以通过MFC的函数得到，而事实上，我根本没装MFC。获得汉字Unicode的方法，一方面可以查表，另外，也可以通过工具软件查找。以下是个实用的小工具，事实上，在以后的研究中，我一直用这个工具检验编码之间转换的正确性。

http://blog.ednchina.com/chinaluou/85656/Message.aspx

这下就简单了。比如我们要输出“你好”，找到它的Unicode码：4F60 597D。因为SDL是通过Uint16 传递Unicode的，所以，对应的数组应该写成：（别忘了最后加一个空元素表示结尾）

```
Uint16 text[] = {0x4F60, 0x597D, 0};
```

这样，SDL 就可以正确的显示中文了。注意：请使用支持中文的字库 TTF 文件。

但是很麻烦，不是吗？我们当然希望汉字是可以自动转换为 SDL 可以渲染的编码，所以，研究还得继续。所有的问题总是能解决的。^^

3、字符集之间的转换，win32 下的libiconv

3.1：GNU的libiconv项目

http://www.gnu.org/software/libiconv/

再一次的，感谢伟大的GNU。我们需要的是Unicode码，在程序中转换，我们需要相应的库。libiconv支持许多字符集，包括我们将用到的GB2312，UTF-8 和UCS-2（Unicode）。具体的，在项目主页上有详细的说明。我们需要新学习直接用的类容并不繁多，同样的，如果你没什么兴趣自己编译源代码，可以直接用在win32 下编译好的头文件，库和动态链接库（DLL）。win32 下的项目主页是：

http://gettext.sourceforge.net/

有趣的是，作者把它作为了我们前面提到的gettext的一部分。在下载页面上，我们直接选择libiconv-win32，同样的，我直接给出所需要的三部分文件的相关信息：

iconv.h：头文件，请在C++代码中#include进来；

iconv.lib：库文件，在编译时候使用；

iconv.dll：动态链接库，请放到exe文件能找到的路径下（通常与exe在同一文件夹下面）

下面，我们看看libiconv的使用方法。

3.2：libiconv的演示程序

我们还是边写程序边做说明：

```
#include <iostream>
#include <string>
#include <iomanip>
#include "GNU/iconv.h"

void showHex(int x);
```

<iomanip>和 showHex 函数，是用来现实 16 进制的。我们在前面用过。

```
int main(int argc, char* argv[])
{
   //src string
   const std::string str = "你好";
   //string size
   const int STR_SIZE = 256;
   //string to be changed
   const unsigned char* src = (const unsigned char*)(str.c_str());
   size_t src_len = strlen((char*)src);
   //string after changed
   unsigned char dst[STR_SIZE] = {0};
   size_t dst_len = sizeof(dst);
   //iconv's arg
   const unsigned char* in = src;
```

```
    unsigned char* out = dst;
    std::cout << "src: " << src << std::endl;
```
我们用来转换的字符串是“纯中文”（为什么我要加引号重点说明，后面会有原因的解释）"你好"。STR_SIZE 是预留的转换内存空间。为什么不用动态存储呢？因为我试过，有错误，可能是 iconv 自身的限制。src 是 C 风格的源字符串，dst 是转换后的 unsigned char 数组。in 和 out 是用于 inconv 的参数。
```
    //GB2312 to UCS-2 (Unicode)
    iconv_t cd;
    cd = iconv_open("UCS-2", "GB2312");
    if ((iconv_t)-1 == cd){
        return -1;
    }
    iconv(cd, (const char**)&in, &src_len, (char**)&out, &dst_len);
    iconv_close(cd);
```
这一段是编码的转换，详细内容请查阅 iconv 的 doc。
```
    //Unicode dst
    std::cout << "dst: ";
    int unicode_len = strlen((char*)dst);
    for (int i = 0; i < unicode_len; i++) {
        showHex(dst[i]);
    }
    std::cout << std::endl;

    return 0;
}

void showHex(int x)
{
    using namespace std;
    cout << hex;
    cout << "0x" << setw(4) << setfill('0') << x << " ";
    cout << dec;
}
```
最后一部分是显示转换后代码。包括函数 showHex()。

3.3：iconv的问题。

我们似乎已经解决问题了。但是有一个问题是，这样转换的编码是 8 位的，即 unsigned char，而 SDL 需要的是 16 位的，即 Uint16；第二个问题是，我说了，这是纯中文，你试试在“你”和“好”之间加段英语是什么效果？结论是，转换不能正常进行，在遇到第一个非汉字的时候，就终止了。

问题还是没解决，研究还得继续。

4、用iconv获得正确的Unicode，使用FriBiDi实现UTF-8到Unicode的正确转换

4.1：为什么iconv不能完全正确的转换Unicode？

我不是先知，教程里面是整理过的思路和逻辑顺序，在我研究这个问题的时候，头绪远远比教程里面乱得多。我完全是从 Wesnoth 的源代码去分析问题的，所以，为什么会扯上 UTF-8 和 FriBidi，那也是因为在源代码中找到了线索。

iconv 不能完全正确的获得 Unicode，也就是我们刚才遇到的纯汉字转换没问题，而有英文就不行了。我并不清楚这是 win32 下的问题，还是在 Linux 下也这样，我也不清楚具体的算法和问题的根本原因，我只是通过试验得到一个算是表面原因的结论：我们知道，GB2312 和 Unicode 汉字都使用 2 个字节（在 UTF-8 中是 3 个字节），英文和数字等用 1 个字节。iconv 在得到两个字节（unsigned char 即一个字节大小）代码的时候可以正确的将 GB2312 转化为 Unicode（或者 UTF-8），但是只有 1 个字节的时候则在转化 Unicode 的时候终止了，幸运的是，如果是转化为 UTF-8 则可以正确的进行，并且也转化为 1 个字节的 UTF-8（只限于英文，数字等）。

所以，我们可以先通过 iconv 将原来的 GB2312 转化为 UTF-8——汉字用 3 个字节（3 个单位的 unsigned char），英文、数字和基本符号用 1 个字节（1 个单位的 unsigned char）。然后，我们需要一个函数，将这种形式的 UTF-8 转换为 SDL 所需要的 Uint16 的 Unicode。什么样的函数可以实现这种转换呢？

4.2：其它编码与Unicode之间的双向转换，GNU FriBidi

http://fribidi.freedesktop.org/wiki/

FriBidi是一个致力于Unicode编码与其它编码相互转换的开源项目，到目前为止，还是一个尚未完成的项目。我在研究 Wesnoth 源代码的时候看到这样的函数：fribidi_utf8_to_unicode()，所以，我想在这个函数中可能应该包含UTF-8 到Unicode的算法——希望不要太复杂。在FriBidi项目中找到这个函数，它在文件fribidi_char_sets_utf8.c下面：

```
int
fribidi_utf8_to_unicode (char *s, int len, FriBidiChar *us)
/* warning: the length of input string may exceed the length of the output */
{
 int length;
 char *t = s;

 length = 0;
 while (s - t < len)
  {
   if (*(unsigned char *) s <= 0x7f)    /* one byte */
  {
   *us++ = *s++;        /* expand with 0s */
```

```
  }
   else if (*(unsigned char *) s <= 0xdf)    /* 2 byte */
  {
   *us++ =
    ((*(unsigned char *) s & 0x1f) << 6) +
    ((*(unsigned char *) (s + 1)) & 0x3f);
   s += 2;
  }
   else           /* 3 byte */
  {
   *us++ =
    ((int) (*(unsigned char *) s & 0x0f) << 12) +
    ((*(unsigned char *) (s + 1) & 0x3f) << 6) +
    (*(unsigned char *) (s + 2) & 0x3f);
   s += 3;
  }
   length++;
  }
 *us = 0;
 return (length);
}
```

其中，我们找到 FriBidiChar 的定义，类似 Uint32 的类型；另外，函数用 char 表示 1 字节的单位。根据我的试验，至少在 VC2008 下是有错误的，我们一直用的是 unsigned char 表示 1 字节的单位，所以，我们需要对这个函数做些修改：

```
int myUTF8_to_UNICODE(Uint16* unicode, unsigned char* utf8, int len)
{
  int length;
  unsigned char* t = utf8;

  length = 0;
  while (utf8 - t < len){

    //one byte.ASCII as a, b, c, 1, 2, 3… ect
    if ( *(unsigned char *) utf8 <= 0x7f ) {
      //expand with 0s.
      *unicode++ = *utf8++;
    }
    //2 byte.
    else if ( *(unsigned char *) utf8 <= 0xdf ) {
      *unicode++ = ((*(unsigned char *) utf8 & 0x1f) << 6) + ((*(unsigned char *) (utf8 + 1)) &
0x3f);
      utf8 += 2;
    }
    //3 byte.Chinese may use 3 byte.
```

```
        else {
            *unicode++ = ((int) (*(unsigned char *) utf8 & 0x0f) << 12) +
                ((*(unsigned char *) (utf8 + 1) & 0x3f) << 6) +
                (*(unsigned char *) (utf8 + 2) & 0x3f);
            utf8 += 3;
        }
        length++;
    }

    *unicode = 0;

    return (length);
}
```

4.3：将汉字，英文，数字和符号都正确的转换为 16 位的 Unicode

有了 iconv 和上面这个函数，我们终于可以将 GB2312 的编码正确的转换为 Unicode 了。

```
//FileName: gb2312_to_Unicode.h
#ifndef GB2312_TO_UNICODE_H_
#define GB2312_TO_UNICODE_H_

#include <iostream>
#include <vector>
#include "GNU/iconv.h"
#include "SDL/SDL.h"

std::vector<Uint16> getUnicode(const std::string& str);

#endif
```

实现文件中包含我们上面写的从 UTF-8 到 Unicode 的函数：

```
#include "gb2312_to_Unicode.h"

int myUTF8_to_UNICODE(Uint16* unicode, unsigned char* utf8, int len);

std::vector<Uint16> getUnicode(const std::string& str)
{
    const int CHAR_SIZE = 256;
    //GB2312 src
    const unsigned char* src = (const unsigned char*)(str.c_str());
    size_t src_len = strlen((char*)src);
    //Unicode dst to get
    unsigned char dst[CHAR_SIZE] = {0};
```

```
    size_t dst_len = sizeof(dst);
    //iconv arg
    const unsigned char* in = src;
    unsigned char* out = dst;

    iconv_t cd;
    //GB2312 to UTF-8
    cd = iconv_open("UTF-8", "GB2312");
    if ((iconv_t)-1 == cd){
        exit (-1);
    }
    //conversion
    iconv(cd, (const char**)&in, &src_len, (char**)&out, &dst_len);

    //UTF-8 to Unicode
    int utf8Len = strlen((char*)dst);
    Uint16 unicodeData[CHAR_SIZE] = {0};
    int unicodeLen = myUTF8_to_UNICODE(unicodeData, dst, utf8Len);
    std::vector<Uint16> unicodeVectorArray;
    for (int i = 0; i < unicodeLen; i++) {
        unicodeVectorArray.push_back(unicodeData[i]);
    }

    iconv_close(cd);
    return unicodeVectorArray;
}
```

函数把一个 std::string 转换位 Uint16 的 vector 数组并返回，这正是 SDL 所需要的 Unicode 格式。

5、SDL完美显示中文

注意：请使用支持中文的 TTF 字库。

5.1：构建可以正确显示中文的SDL_ttf函数

世界终于又充满了光明！任何事情都是有答案的，不知道仅仅是因为我们还没有找到。解决了以上一系列问题，我们终于可以不依赖 MFC，完全使用自由开源的资源，让 SDL 显示中文了！我们通过 TTF_RenderUNICODE_Xxx()来构建这些函数：

```
//FileName: font.h
#ifndef FONT_H_
#define FONT_H_

#include "gb2312_to_Unicode.h"
```

```
#include "SDL/SDL_ttf.h"

SDL_Surface* myTTF_RenderString_Blended(TTF_Font* font, const std::string& str, SDL_Colo
r fg);
SDL_Surface* myTTF_RenderString_Solid(TTF_Font* font, const std::string& str, SDL_Color f
g);
SDL_Surface* myTTF_RenderString_Shaded(TTF_Font* font, const std::string& str, SDL_Color
 fg, SDL_Color bg);

#endif
```

三种显示模式的实现文件：

```
#include "font.h"

SDL_Surface* myTTF_RenderString_Blended(TTF_Font* font, const std::string& str, SDL_Colo
r fg)
{
   SDL_Surface* textbuf;
   //Get Unicode
   std::vector<Uint16> unicodeUnit = getUnicode(str);
   int arraySize = unicodeUnit.size();
   Uint16* perOne = new Uint16[arraySize+1];
   for ( int i = 0; i < arraySize; i++ )
      perOne[i] = unicodeUnit[i];
   perOne[arraySize] = 0;


   //Render the new text
   textbuf = TTF_RenderUNICODE_Blended(font, perOne, fg);

   //Free the text buffer and return
   delete [] perOne;
   return textbuf;
}

SDL_Surface* myTTF_RenderString_Solid(TTF_Font* font, const std::string& str, SDL_Color f
g)
{
   SDL_Surface* textbuf;
   //Get Unicode
   std::vector<Uint16> unicodeUnit = getUnicode(str);
   int arraySize = unicodeUnit.size();
   Uint16* perOne = new Uint16[arraySize+1];
   for ( int i = 0; i < arraySize; i++ )
      perOne[i] = unicodeUnit[i];
```

```
    perOne[arraySize] = 0;


    //Render the new text
    textbuf = TTF_RenderUNICODE_Solid(font, perOne, fg);

    //Free the text buffer and return
    delete [] perOne;
    return textbuf;
}

SDL_Surface* myTTF_RenderString_Shaded(TTF_Font* font, const std::string& str, SDL_Color
 fg, SDL_Color bg)
{
    SDL_Surface* textbuf;
    //Get Unicode
    std::vector<Uint16> unicodeUnit = getUnicode(str);
    int arraySize = unicodeUnit.size();
    Uint16* perOne = new Uint16[arraySize+1];
    for ( int i = 0; i < arraySize; i++ )
        perOne[i] = unicodeUnit[i];
    perOne[arraySize] = 0;


    //Render the new text
    textbuf = TTF_RenderUNICODE_Shaded(font, perOne, fg, bg);

    //Free the text buffer and return
    delete [] perOne;
    return textbuf;
}
```

5.2：修改DisplaySurface的类方法

其它接口都是不需要改动的，我们仅仅把类方法中，原来用于构建文本面的函数换成我们自己的函数就可以了。当然，先把这些函数#include 进来：

```
#include "SurfaceClass.h"
#include "font.h"


//…


DisplaySurface::DisplaySurface(const std::string& msg_name, const std::string& message, const
ScreenSurface& screen,
```

```
                Uint8 r, Uint8 g , Uint8 b,
                int ttf_size, const std::string& ttf_fileName):
fileName(msg_name)
{
   if ( textNum == 0 )
      if ( TTF_Init() < 0 )
         throw ErrorInfo("TTF_Init() failed!");

   SDL_Color textColor;
   textColor.r = r;
   textColor.g = g;
   textColor.b = b;

   pFont = TTF_OpenFont(ttf_fileName.c_str(), ttf_size);
   if ( pFont == 0 )
      throw ErrorInfo("TTF_OpenFont() failed!");

   pSurface = myTTF_RenderString_Blended(pFont, message, textColor);
   if ( pSurface == 0 )
      throw ErrorInfo("myTTF_RenderString_Blended() failed!");
   pScreen = screen.point();

   textNum++;
}
```

5.3：StringData在主程序中的调用

最后，我们演示一下 StringData 在主程序中的调用方法。

```
//must #include "string_data.h"

   //Load a textSurface
   StringData myData;
   const std::string uInfo = myData[0];
   const std::string dInfo = myData[1];
   const std::string lInfo = myData[2];
   const std::string rInfo = myData[3];
   const std::string oInfo = myData[4];
   TextSurface upMessage("upMsg", uInfo, screen);
   TextSurface downMessage("downMsg", dInfo, screen, 0xFF, 0, 0);
   TextSurface leftMessage("leftMsg", lInfo, screen, 0, 0xFF, 0);
   TextSurface rightMessage("rightMsg", rInfo, screen, 0, 0, 0xFF);
   TextSurface otherMessage("otherMsg", oInfo, screen, 100, 100, 100, 35);
```

嘿嘿，就这么简单！

5.4：本章演示程序和完整源代码下载

包含SDL显示中文的演示程序（win32）以及完整的源代码。
http://www.fs2you.com/zh-cn/files/62f0acf0-ff11-11dc-a4f4-0014221b798a/

SDL入门教程（十一）：

1、SurfaceClass类的再设计

1.1：为了按钮做准备

按钮是鼠标事件响应的象征，在 PC 游戏中起着十分重要的作用。这一章节，我们开始通过 SDL 提供的底层函数，自己来设计按钮。

按钮一般有这么几种状态：

out: 鼠标不在按钮上；

over: 鼠标在按钮上；

down: 鼠标按下了；

up: 鼠标松开了；

其中，down 和 up 又可以细分为鼠标是在按钮上按下松开的，还是在按钮外按下松开的。

我们先从按钮的表现形式——图片来分析这些问题。

为了表现不同的状态，我们需要多张图片。至少的，out 与 over 和 down 中的至少一张图片应该是不一样的，这样才能给人以“按下了”的视觉效果。也就是说，如果我们构建按钮类，这个类将会包含我们之前设计的 SurfaceClass 类。这样，根据我们之前讨论的一些 C++ 细节问题，我们必须重新小心的设计 SurfaceClass 类。

1.2：为SurfaceClass设计基类

我们之前使用 DisplaySurface 本身作为 TextSurface 的基类，这是因为我们希望通过基类本身引用派生类，从而不需要为基类和派生类分别写函数。同样的，我们希望在 ButtonClass 中直接使用基类作为成员数据，这样，ButtonClass 可以不需要为派生类重新构建。在之前的 DisplaySurface 中，我们可以看到，大部分方法，比如最为重要的 blit()方法，所涉及到的成员数据仅仅就是 SDL_Surface* pSurface，这一点，TextSurface 类也是一样的。DisplaySurface 类通过图片创建 pSurface，TextSurface 类通过字库创建 pSurface，他们的创建条件不尽相同，TextSurface 需要打开 TTF_Init()，还需要打开字库 TTF_OpenFont()，所以，他们的构造函数与析构函数是不一样的。一种更加清晰的关系，是建立起他们的基类，然后把他们的区别点分开处理，而不是之前采用的 if 来判断处理。

最清晰的办法，是把 BaseSurface 设计成为抽象基类（ABC），但是 ABC 的问题在于，无法创建 ABC 本身的对象，也就是说，我们需要在 ButtonClass 中包含的 BaseSurface 对象无法建立，所以，我们不能把 BaseSurface 设计成为 ABC。我想到的办法，是将 BaseSurface

的默认构造函数仅仅提供给派生类使用，也就是说，因为 BaseClass 的默认构造函数本身是不能建立起必要的数据的，比如 pSurface 和 pScreen，我们暂时都用空指针代替。这里的思想其实还是 ABC 的，让基类仅仅提供算法，即使算法所需要的元素还尚不存在。但是，只要这些元素存在了，BaseSurface 也就可以完成这些算法，所以，我们把 BaseSurface 的复制构造函数设计为公共的，这很重要，这样才为我们通过派生类去初始化基类的构造函数初始化列表提供了可能。

我们把基类命名为 BaseSurface，之前的 DisplaySurface 我们用更形象的名称代替，叫 PictureSurface，然后 TextSurface 不改名。

1.3：如何深度复制SDL_Surface？

因为 PictureSurface 和 TextSurface 的构造函数都涉及到了堆操作，也就是说，SDL_Surface 是建立在堆上的。我们需要通过基类的复制构造函数复制 SDL_Surface，必须找到合适的工具。具体来说，新对象的 pSurface 必须指像一个新建立起来的 SDL_Surface 对象，而非被复制的对象本身。

我最先想到的是用 new...delete 组合，但是失败了。为什么呢？我们研究 SDL_Surafce 结构本身，发现在之中包含着一些其它的指针。比如 SDL_PixelFormat *format 和 void *pixels。也就是说，SDL_Surface 本身的构建也有堆操作，而且，因为 SDL_Surafce 是 C 风格的结构而不是类，它不能提供深度复制的操作，所以，即使我们用 new 为 SDL_Surface 申请了新空间，SDL_Surface 本身也不会为 format 和 pixels 申请新的副本。当被释放掉原本的时候，副本将寻找不到原本的 format 和 pixels，从而产生错误。

我们另外能找到的可以深度复制 SDL_Surface 的函数，是：

```
SDL_Surface *SDL_DisplayFormat(SDL_Surface *surface);
SDL_Surface *SDL_ConvertSurface(SDL_Surface *src, SDL_PixelFormat *fmt, Uint32 flags);
```

这两个函数貌似都是可行的。实际上，基本上也都是可行的。但是涉及到文字的时候，SDL_DisplayFormat 将不能正确的转换 Blended 和 Shaded 的文字，所以，最后只能使用 SDL_ConvertSurafce()。

新构建的SurfaceClass源代码见：

http://www.cppblog.com/lf426/archive/2008/04/14/47038.html

2、设计按钮ButtonClass

2.1：设计框架与基类的接口

有了更加完善的 SurfaceClass 的支持，我们可以进行按钮的设计了。接着上一节的话题，按钮除了要给我们表现出来是否被按下的视觉效果，还要起到实际上的作用。一种最简单的思路，既是鼠标在按钮上一旦按下，程序就马上响应。这种思路很朴素，也很实用。大名鼎鼎的 QuakeIII 的菜单按钮就是这么设计的，这样我们几乎是可以直接使用 SDL 的事件响应，即：事件不为空——鼠标事件的左键按下——响应处理。

但是也许我们已经习惯更加人性话的 GUI 按钮了。比如，如果是不小心点错了，马上响应意味着没有机会改正操作失误。事实上，我们仔细分析当今 GUI 上的按钮，可以发现

按钮的实际效果，是在按下鼠标，并且又松开的时候产生的响应。其实这样说也并不完全准确，更加准确的描述，应该是鼠标既要在按钮上按下，又要在按钮上松开——其实这还是不完整，我想说的是，为了描述这个复杂的状态，我们不得不在 ButtonClass 中引入几个 bool 量，以判断按钮是否真正起作用。

按钮的构成与视觉效果，根据我们之前的知识，大概具有这几类：完全由 PictureSurface 组成，这又分为两类，两张 Picture（out 和 over）或者三张 Picture（再加张 down）；由 TextSurface 组成；由一张精灵图的 PictureSurafce 切分出来。他们的接口几乎是一样的：设置位置和按下时的偏移（setup），扣色（colorkey），添加文字（addText），显示（blit），鼠标事件判断（mouse out, over, down, up 甚至是 up outside）和有效点击（effectiveClick）。所以，我们有理由用基类来规定这些接口。或者说，用 ABC（抽象基类）的纯虚函数硬性规定这些接口。

2.2：鼠标事件判断与有效点击

移动：SDL_MOUSEMOTION
触发的开关是鼠标发生了移动。我们需要判断鼠标是否移动到了按钮上，或者移动到了不是按钮区域的地方；
点击：SDL_MOUSEBUTTONDOWN
触发条件是鼠标按下了，我们需要进一步判断是不是左键按下了（gameEvent.button.button == SDL_BUTTON_LEFT ），然后判断是不是在按钮区域内按下的。
松开：SDL_MOUSEBUTTONUP
触发条件是鼠标松开了，我们需要进一步判断是不是左键松开了（gameEvent.button.button == SDL_BUTTON_LEFT ），然后判断是不是在按钮区域内松开的。
按钮外松开：SDL_MOUSEBUTTONUP
触发条件是鼠标松开了，我们需要进一步判断是不是左键松开了（gameEvent.button.button == SDL_BUTTON_LEFT ），然后判断是不是在按钮区域内松开的。

我们之所以需要做这些判断，是为了构建我们刚才设想中的按钮效果，即有效点击（effectiveClick）。因为有效点击不是通过一次事件的判断完成的，我们通过三个 bool 量在整个按钮的生命周期类描述按钮所接收到的鼠标事件：inBox 鼠标在按钮区域内；clickDown 鼠标在按钮区域类被按下过（并且没有在外面松开）；clickUp 鼠标在按钮区域内松开。我们来看看这段代码吧……我承认，if 得很混乱，但是居然能正常工作，呵呵。

```
bool BaseButton::effectiveClick(const SDL_Event& game_event)
{
   inBox = this->mouseOver(game_event);
   if ( this->mouseDown(game_event) == true ){
      clickDown = true;
      inBox = true;
   }
   if ( this->mouseUp(game_event) == true ){
      if ( clickDown == true )
         clickUp = true;
      inBox = true;
   }
   if ( this->mouseUpOutside(game_event) == true )
      clickDown = false;
```

```
    if ( inBox == true && clickDown == false ){
        this->blitOver();
        return false;
    }
    else if ( inBox == true && clickDown == true ){
        if ( clickUp == true ){
            clickUp = false;
            clickDown = false;
            this->blitOver();
            return true;
        } else {
            this->blitDown();
            return false;
        }
    }
    else {
        this->blitOut();
        return false;
    }
}
```

最有意思的是，我们可以把这个函数构建在基类中——即使 blitxxx()这类的函数都是纯虚函数——但是他们已经代表了算法。在用不同的派生类调用基类的这个方法的时候，blitxxx()会被替换成相应派生类的版本，从而减少了重复写代码的工作。

2.3：ButtonClass的源代码

http://www.cppblog.com/lf426/archive/2008/04/15/47156.html

3、做一个对话框

有了按钮类，我们制作对话框就很轻松了。边写程序边说明吧。

```
bool hand_dialog(const ScreenSurface& screen, const std::string& dialog_text, int size)
{
    const int CENTRE_X = (screen.point()->w) / 2;
    const int CENTRE_Y = (screen.point()->h) / 2;
    const int HALF_SUB_BUTTON_W = 64 / 2;
```

先找出显示屏的中心点，然后是对话框按钮宽度的一半。这些是为了在屏幕中心的对称位置显示 YES 和 NO

```
    PictureSurface quitBG("./images/dialog_bg.png", screen);
    quitBG.blit();
```

载入对话框的背景

```
    TextSurface quitDlg(dialog_text, screen, 215, 195, 122, size);
    int quitDlg_x = CENTRE_X - ((quitDlg.point()->w)/2);
    int quitDlg_y = CENTRE_Y - 50;
    quitDlg.blit(quitDlg_x, quitDlg_y);
```
载入对话框的提示文字
```
    const int YES_X = CENTRE_X - 40 - HALF_SUB_BUTTON_W;
    const int YES_Y = CENTRE_Y + 30;
    Button yesButtonEffect("./images/h3_yes_off.png", "./images/h3_yes_over.png", screen);
    yesButtonEffect.setup(YES_X, YES_Y, 5);
    yesButtonEffect.blitOut();
```
构建 YES 按钮
```
    const int NO_X = CENTRE_X + 40 - HALF_SUB_BUTTON_W;
    const int NO_Y = CENTRE_Y + 30;
    Button noButtonEffect("./images/h3_no_off.png", "./images/h3_no_over.png", screen);
    noButtonEffect.setup(NO_X, NO_Y, 5);
    noButtonEffect.blitOut();
```
构建 NO 按钮
```
    screen.flip();
```
显示屏幕
```
    SDL_Event gameEvent;
    while( true ){
        while ( SDL_PollEvent(&gameEvent) != 0 ){
            if ( gameEvent.type == SDL_KEYDOWN ){
                if ( gameEvent.key.keysym.sym == SDLK_ESCAPE ){
                    return false;
                }
            }
            quitBG.blit();
            quitDlg.blit(quitDlg_x, quitDlg_y);
            if ( yesButtonEffect.effectiveClick(gameEvent) == true )
                return true;
            if ( noButtonEffect.effectiveClick(gameEvent) == true )
                return false;
            screen.flip();
        }
    }
}
```
主循环了。可以通过按下 ESC 或者按下 NO 按钮取消对话框，按下 YES 按钮则表示做出了选择。我们使用 effectiveClick()方法，一次性的将接收事件，判断事件，显示按钮的不同状态集成性的完成了。

SDL入门教程（十二）：

音乐和音效

1：扩展库SDL_mixer

根据网上资料的说法，SDL本身的声音体系做得不是很完善，好在还有一个比较完善的扩展库SDL_mixer，这个库支持包括wav, mp3, ogg和midi的声音和音乐，也算是相当完善了。下载地址在：
http://www.libsdl.org/projects/SDL_mixer/
安装设置参考前面的涉及SDL扩展库安装设置的章节。

2：SDL_mixer的启动和退出，设计MixSoundClass的基类

这个库又是通过 open...close 这样的格式来初始化启动以及退出的。但是我们将用到的音乐和音效将分别用到不同的结构体。SDL_mixer 中音乐和音效的共同点在于都需要打开 Mix_OpenAudio()，使用完后都应该关闭 Mix_CloseAudio()。所以，可以为这两种类设计共同的基类，并设计一个计数器，作为打开和关闭的自动开关。

```
class BaseMixSound
{
private:
   static int MixNUM;
protected:
   BaseMixSound();
public:
   virtual ~BaseMixSound();
};
```

在打开函数中，也就是这个“假”ABC 的构造函数中，我们用到了 SDL_mixer 的函数：

```
Mix_OpenAudio( 22050, MIX_DEFAULT_FORMAT, 2, 4096 )
```

其中，第一个参数是频率，第二个参数是格式，第三个参数是声道，第四个参数是采样率。因为我下不到 SDL_mixer 的说明文档，大概的情况只能描述到这样。试验的情况看，这样的数据是可以比较良好的工作的。
退出的函数是：

```
Mix_CloseAudio()
```

我们放在了析构函数中。

3：音效类EffectSound

```
class EffectSound: public BaseMixSound
{
```

```
private:
   Mix_Chunk* sound;
public:
   EffectSound(const std::string& sound_fileName);
   ~EffectSound();
   void play() const;
};
```

Mix_Chunk 是音效的结构，由函数 Mix_LoadWAV()创建，Mix_FreeChunk()释放。在方法 play()中，我们用到了函数：

Mix_PlayChannel(-1, sound, 0)

第一个参数是指定播放音效的通道，这里设置为-1，则系统会自动寻找使用第一个可以使用的通道；第二个参数是要播放的音效的结构指针；第三个是重复次数，这里为 0 表示不重复，即播放一次。

4：音乐类MusicSound

```
class MusicSound: public BaseMixSound
{
private:
   Mix_Music* music;
public:
   MusicSound(const std::string& music_fileName);
   ~MusicSound();
   void play() const;
   void stop() const;
};
```

Mix_Music 是音乐的结构，由函数 Mix_LoadMUS()创建，Mix_FreeMusic()释放。我们播放音乐用到的函数为：

Mix_PlayMusic(music, -1)

第一个参数是要播放的音乐的结构指针；第二个是播放次数，这里使用-1 将不停止的循环播放，直到被停止。

另外三个行为的函数为：暂停 Mix_PauseMusic()，继续 Mix_ResumeMusic()，结束 Mix_HaltMusic()。

还有两个判断当前音乐状态的函数 Mix_PlayingMusic()和 Mix_PausedMusic()，表示是否在播放，是否暂停了。返回的是 int，0 为假，1 为真。

5：完整的源代码

http://www.cppblog.com/lf426/archive/2008/04/20/47642.html

SDL入门教程（十三）：

1、多线程，从动画说起

1.1：简单动画

游戏离不开动画。我们考虑最简单的情况：将一个角色从一个位置移动到另外一个位置。这个行为表述给电脑就是，将一个 surface 不断的 blit()，从起始位置的坐标，移动到结束位置的坐标。移动速度取决于每次 blit()的坐标差和 blit()的时间间隔（v = ds/dt ）。

我们来设计一个函数实现这个简单的动画。我们需要的数据有：起始坐标（int beginX, int beginY），结束坐标（int endX, int endY），以及作为 SDL 基础的 ScreenSurface 窗口（const ScreenSurface& screen）。一般的考虑是，将这 5 个数据以参数的方式传入函数；但是一种更加通用一点的方式是，将这 5 种数据合并成一个结构，这样函数的参数形式会更加的统一，这正是触发多线程的函数所需要的。在 SDL 中，我们通过函数：

SDL_Thread *SDL_CreateThread(int (*fn)(void *), void *data);

触发多线程，其中所需要的函数指针形式为：

typedef int (*fn)(void*);

而 void*类型的 data 就是函数(*fn)()需要的的数据。我们可以将任意的结构体指针，转化为 void*，作为这个函数的第二个参数需要。

因此，我们可以为我们需要的动画函数定义一个结构作为传递所有数据的载体：

```
struct AmnArg
{
   int beginX;
   int beginY;
   int endX;
   int endY;
   const ScreenSurface& screen;
   AmnArg(int begin_x, int begin_y, int end_x, int end_y, const ScreenSurface& _screen): beginX
(begin_x), beginY(begin_y), endX(end_x), endY(end_y), screen(_screen){}
};
```

这样，我们可以将 AmnArg 对象的指针传递给动画函数——考虑到多线程函数的需要，我们再曲折一点：先将 AmnArg*转换成 void*传递给函数，在函数内部再将其转换回来以供调用。

```
int amn(void* data)
{
   AmnArg* pData = (AmnArg*)data;
   PictureSurface stand("./images/am01.png", pData->screen);
   stand.colorKey();
   PictureSurface bg("./images/background.png", pData->screen);

   const int SPEED_CTRL = 300;
   int speedX = (pData->endX - pData->beginX) / SPEED_CTRL;
```

```
    int speedY = (pData->endY - pData->beginY) / SPEED_CTRL;

    for ( int i = 0; i < SPEED_CTRL; i++ ){
        pData->beginX += speedX;
        pData->beginY += speedY;
        bg.blit(pData->beginX, pData->beginY, pData->beginX, pData->beginY, stand.point()->w, st
and.point()->h, 2, 2);
        stand.blit(pData->beginX, pData->beginY);
        pData->screen.flip();
    }

    return 0;
}
```

注意：我们这里仅仅设定了每次 blit()的位移差（ds）而没有设定时间差（dt）。这并不意味着 dt == 0，事实上，电脑处理数据是需要时间的，包括运算和显示。我们这里事实上将 dt 的设定交给了电脑，也就是说，让电脑以其最快的速度来完成。为什么要这么做呢？这是为了演示多线程的一个现象，卖个关子，后面解释。:)

1.2：动画函数在主程序中的调用

```
#include "SurfaceClass.hpp"
#include "amn.hpp"

int main(int argc ,char* argv[])
{
    //Create a SDL screen.
    const int SCREEN_WIDTH = 640;
    const int SCREEN_HEIGHT = 480;
    const Uint32 SCREEN_FLAGS = 0; //SDL_FULLSCREEN | SDL_DOUBLEBUF | SDL_HW
SURFACE
    const std::string WINDOW_NAME = "Amn Test";
    ScreenSurface screen(SCREEN_WIDTH, SCREEN_HEIGHT, WINDOW_NAME, 0, SCREE
N_FLAGS);

    PictureSurface bg("./images/background.png", screen);
    bg.blit(0);
    screen.flip();

    AmnArg test1(0, 250, 600, 250, screen);
    amn((void*)&test1);

    SDL_Event gameEvent;
    bool gameOver = false;
    while ( gameOver == false ){
```

```
    while ( SDL_PollEvent(&gameEvent) != 0 ){
      if ( gameEvent.type == SDL_QUIT ){
        gameOver = true;
      }
      if ( gameEvent.type == SDL_KEYDOWN ){
        if ( gameEvent.key.keysym.sym == SDLK_ESCAPE ){
          gameOver = true;
        }
      }
      screen.flip();
    }
  }

  return 0;
}
```

当这个程序运行的时候，我们会发现一些很明显的问题：

1、图片移动的时候，界面不接受任何信息。这是因为必须把 amn()执行完毕才会运行到有事件响应的事件轮询循环。

2、如果我们需要另外一张图片移动起来，我们唯一能做的事情，是修改 amn()函数，而不是把 amn()以不同的参数调用两次——如果以不同的参数调用两次，那么移动总是有先后的——是不可能完成“同时”移动的。

1.3：创建线程

如果要将这个程序从主线程（主进程）调用函数修改为通过新创建的线程调用函数，只需要做很小的修改，即将 amn((void*)&test1);修改为：

```
SDL_Thread* thread1 = SDL_CreateThread(amn, (void*)&test1);
```

然后在 return 0;之前加入清理线程的语句：

```
SDL_KillThread(thread1);
```

这样，程序在执行动画的同时，事件轮询就已经开始，我们可以随时结束程序，SDL 界面也不会出现不响应的情况。

2、初识多线程

2.1：竞争条件（Race Conditions）

我们在前面将一个普通函数调用转换成了用线程调用，这意味着我们可以“同时”调用两个以上的线程。例如，我们希望在屏幕的另外一个位置也播放这段简单的动画，我们只需要添加一个线程的调用就可以了。

```
int main(int argc ,char* argv[])
{
  //Create a SDL screen.
```

```
    const int SCREEN_WIDTH = 640;
    const int SCREEN_HEIGHT = 480;
    const Uint32 SCREEN_FLAGS = 0; //SDL_FULLSCREEN | SDL_DOUBLEBUF | SDL_HW
SURFACE
    const std::string WINDOW_NAME = "Amn Test";
    ScreenSurface screen(SCREEN_WIDTH, SCREEN_HEIGHT, WINDOW_NAME, 0, SCREE
N_FLAGS);

    PictureSurface bg("./images/background.png", screen);
    bg.blit(0);
    screen.flip();

    AmnArg test1(0, 250, 600, 250, screen);
    SDL_Thread* thread1 = SDL_CreateThread(amn, (void*)&test1);

    AmnArg test2(0, 0, 600, 0, screen);
    SDL_Thread* thread2 = SDL_CreateThread(amn, (void*)&test2);

    SDL_Event gameEvent;
    bool gameOver = false;
    while ( gameOver == false ){
        while ( SDL_PollEvent(&gameEvent) != 0 ){
            if ( gameEvent.type == SDL_QUIT ){
                gameOver = true;
            }
            if ( gameEvent.type == SDL_KEYDOWN ){
                if ( gameEvent.key.keysym.sym == SDLK_ESCAPE ){
                    gameOver = true;
                }
            }
            screen.flip();
        }
    }

    SDL_KillThread(thread1);
    SDL_KillThread(thread2);
    return 0;
}
```

这段程序看起来似乎没有什么问题，但是运行的时候，不可预知的情况出现了：理论上我们几乎同时调用了两个线程，动画似乎应该是同步播放的，但是实际上，两段动画的播放并不同步，并且每次执行的效果都不一样——有时候上面的图片移动快，有时候下面的图片移动快，并且速度不均匀。

这就是典型的 race conditions 的表现。还记得我说过没有定义 dt 吗，我们让电脑以其所能达到的最快速度决定 dt，换句话说，我们每一个线程都试图“咬死”CPU 的运算，当然，

在实际中多任务的 OS 会帮助 CPU 分配任务，但是如何分配却是不确定的，因为 OS 并不知道哪些任务需要优先执行，所以，两个线程实际上在竞争电脑的性能资源，产生的结果就是不确定的。

2.2：松开“死咬”的CPU

void SDL_Delay(Uint32 ms);

解决 race conditions 的方法就是给 CPU 足够的时间“休息”，而这正好也是我们自己定义 dt 所需要的。SDL_Delay()在这个时候就显得意义重大了。当今电脑的运算速度非常非常快，以至于哪怕我们仅仅给电脑 0.01 秒的时间“休息”（每次循环中），电脑都会显得很轻松了。

```
int amn(void* data)
{
   AmnArg* pData = (AmnArg*)data;
   PictureSurface stand("./images/am01.png", pData->screen);
   stand.colorKey();
   PictureSurface bg("./images/background.png", pData->screen);

   const int SPEED_CTRL = 300;
   int speedX = (pData->endX - pData->beginX) / SPEED_CTRL;
   int speedY = (pData->endY - pData->beginY) / SPEED_CTRL;

   for ( int i = 0; i < SPEED_CTRL; i++ ){
      pData->beginX += speedX;
      pData->beginY += speedY;
      bg.blit(pData->beginX, pData->beginY, pData->beginX, pData->beginY, stand.point()->w, st
and.point()->h, 2, 2);
      stand.blit(pData->beginX, pData->beginY);
      pData->screen.flip();
      SDL_Delay(10);
   }

   return 0;
}
```

说到这里，我们不得不提及之前一直所忽略的一个问题：我们之前凡是涉及循环等待事件轮询的程序总是占用 100%的 CPU，这并不是因为我们真正用到了 100%的 CPU 性能，而是我们让 CPU 陷入了“空等”（Busy Waiting）的尴尬境地。轮询事件得到响应相对于循环等待来说，是发生得非常缓慢的事情，我们在循环中，哪怕是让电脑休息 0.01 秒，事情都会发生本质性的改变：

```
   while ( gameOver == false ){
      while ( SDL_PollEvent(&gameEvent) != 0 ){
         if ( gameEvent.type == SDL_QUIT ){
            gameOver = true;
```

```
        }
        if ( gameEvent.type == SDL_KEYDOWN ){
          if ( gameEvent.key.keysym.sym == SDLK_ESCAPE ){
            gameOver = true;
          }
        }
        screen.flip();
      }
      SDL_Delay(10);
    }
```

当我们重新运行新程序的时候，我们可以看到程序对 CPU 的占用从 100%骤降到了 0%！这当然并不意味着程序就用不上 CPU 了，而是说，这些运算对于我们的 CPU 来说，实在是小菜一碟了，或者从数据上说，处理这些运算的时间与 0.01 秒来比较，都几乎可以忽略不计！

2.3：GUI线程与worker线程

我们的另外一项试验是将事件轮询放到动画线程中，程序就不多写了，大家可以自己试下。我直接说结论：动画线程中无法响应事件轮询。

一般提倡的模式，是将 GUI 事件都写在主线程中，而将纯粹的运算才写到由主线程创建的线程中，后者也就是所谓的 worker 线程。从另外一个概念看，只有主线程控制着“当前窗口”，其它线程也许在后台，也许虽然也是在前台但是并非是我们可见的，所以，轮询事件找不到接口。

对于抛出的线程与主线程之间的通讯，我们可以通过他们共享的数据来进行控制，所以，尽管事件轮询不能直接影响 worker 线程，但是我们仍然是可以通过主线程进行间接影响的。

3、封装多线程

SDL创建多线程的函数SDL_CreateThread()所调用的是函数指针，这意味着我们不可以传入（非静态）成员函数的指针。关于两种函数指针我们之前已经讨论过：函数指针与成员函数指针，我们可以有两种方法能让具有普通函数指针（函数指针以及静态成员函数指针）的函数调用类的私有成员，一是友元函数，另外就是静态成员函数。而能够受到类私有保护的，只有静态成员函数。所以，我们可以通过静态成员函数调用一个对象数据的形式，实现对于创建多线程函数的封装。

另外，我们希望测试在主线程中读写线程数据的效果，所以添加了两个方法show() 和reset()，多线程演示的类源代码如下：

```
#include <iostream>
#include "SurfaceClass.hpp"

class AmnArg
{
private:
  int beginX;
```

```
    int beginY;
    int endX;
    int endY;
    const ScreenSurface& screen;
    //
    static int amn(void* pThat);
public:
    AmnArg(int begin_x, int begin_y, int end_x, int end_y, const ScreenSurface& _screen);
    SDL_Thread* createThrd();
    void show() const;
    void reset();
};
```

其中SurfaceClass.hpp请参考：

http://www.cppblog.com/lf426/archive/2008/04/14/47038.html

实现函数如下：

```
#include "amn.hpp"

AmnArg::AmnArg(int begin_x, int begin_y, int end_x, int end_y, const ScreenSurface& _screen):
beginX(begin_x), beginY(begin_y), endX(end_x), endY(end_y), screen(_screen)
{}

SDL_Thread* AmnArg::createThrd()
{
    return SDL_CreateThread(amn, (void*)this);
}

void AmnArg::show() const
{
    std::cout << "Now x at: " << beginX << std::endl;
}

void AmnArg::reset()
{
    beginX = 0;
}

int AmnArg::amn(void* pThat)
{
    AmnArg* pData = (AmnArg*)pThat;
    PictureSurface stand("./images/am01.png", pData->screen);
    stand.colorKey();
    PictureSurface bg("./images/background.png", pData->screen);

    const int SPEED_CTRL = 300;
```

```
    int speedX = (pData->endX - pData->beginX) / SPEED_CTRL;
    int speedY = (pData->endY - pData->beginY) / SPEED_CTRL;

    for ( int i = 0; i < SPEED_CTRL; i++ ){
        pData->beginX += speedX;
        pData->beginY += speedY;
        bg.blit(pData->beginX, pData->beginY, pData->beginX, pData->beginY, stand.point()->w, st
and.point()->h, 2, 2);
        stand.blit(pData->beginX, pData->beginY);
        pData->screen.flip();
        SDL_Delay(10);
    }

    return 0;
}
```

最后，我们修改了主演示程序，并测试了 show()和 reset()的效果。我们可以看到，直接修改线程数据的 reset()的结果也是不可预知的，所以，我们似乎更应该通过改变线程“流”的效果，而不是直接对数据进行修改。这个我们以后再讨论了。

```
#include "SurfaceClass.hpp"
#include "amn.hpp"

int game(int argc ,char* argv[]);
int main(int argc ,char* argv[])
{
    int mainRtn = 0;
    try {
        mainRtn = game(argc, argv);
    }
    catch ( const ErrorInfo& info ) {
        info.show();
        return -1;
    }
    catch ( const char* s ) {
        std::cerr << s << std::endl;
        return -1;
    }

    return mainRtn;
}

int game(int argc ,char* argv[])
{
    //Create a SDL screen.
    const int SCREEN_WIDTH = 640;
```

```
    const int SCREEN_HEIGHT = 480;
    const Uint32 SCREEN_FLAGS = 0; //SDL_FULLSCREEN | SDL_DOUBLEBUF | SDL_HW
SURFACE
    const std::string WINDOW_NAME = "Amn Test";
    ScreenSurface screen(SCREEN_WIDTH, SCREEN_HEIGHT, WINDOW_NAME, 0, SCREE
N_FLAGS);

    PictureSurface bg("./images/background.png", screen);
    bg.blit(0);
    screen.flip();

    AmnArg test1(0, 250, 600, 250, screen);
    SDL_Thread* thread1 = test1.createThrd();

    AmnArg test2(0, 0, 400, 0, screen);
    SDL_Thread* thread2 = test2.createThrd();

    SDL_Event gameEvent;
    bool gameOver = false;
    while ( gameOver == false ){
        while ( SDL_PollEvent(&gameEvent) != 0 ){
            if ( gameEvent.type == SDL_QUIT ){
                gameOver = true;
            }
            if ( gameEvent.type == SDL_KEYDOWN ){
                if ( gameEvent.key.keysym.sym == SDLK_ESCAPE ){
                    gameOver = true;
                }
                if ( gameEvent.key.keysym.sym == SDLK_SPACE ){
                    test1.show();
                    test2.show();
                }
            }
            screen.flip();
        }
        SDL_Delay(100);
    }

    SDL_KillThread(thread1);
    SDL_KillThread(thread2);

    return 0;
}
```

SDL入门教程（十四）：

1、网络，唠叨以及前言

这一次更新的间隔是很漫长的。之前的教程得到很多朋友的支持，让我感到责任的沉重。一般说来，技术博客的文章通常是自己的学习笔记，但是作为一份被期望“零起步”的教程，我更多时候感到这份东西不仅仅是为我自己写的。

SDL 库作为主要为游戏开发提供简单易用的支持的 API，相信很多朋友学习它的目的，也是跟我一样，希望能做出自己梦想中的游戏吧。作为游戏，我始终相信一个观点：（网络）游戏，是以电脑和网络为媒介，人与人之前的交流。游戏是我们营造“共同经历”的一种方式。作为交流的主要工具，网络起着决定性的作用。所以，本着对网络这一部分的高度重视，我对 SDL_net 寄予厚望。

但是，现实是残酷的。我不可否认 SDL_net 已经对网络编程做了很好的封装，但是问题是，socket 编程本身并不是件简单的事情。所以，实际上的情况是，仅仅靠学习 SDL_net 的函数库，也许我们根本不可能搞明白网络编程究竟是什么东东。必要的基础知识是必须学习的，我逐渐明白这一块我始终是不可能跳过去的。

既然必须要补上这一块的知识，在具体的实现平台上，我们必须做出选择。经过均衡，我还是选择从 BSD socket 入手。这不仅仅是因为 BSD 上的 socket 实现是最早的 TCP/IP 实现，是业内的事实标准；也是因为 Win32 API 实在是不能引起我的好感。

当然，我们不可否认 Windows 有他先进的地方。至少，在 socket 编程方面，基于线程而非 Linux 以进程为基础的核心结构，可以更加有效的控制线程，并且线程的调度也更加均匀。在 Linux 下，无论是 fork 还是 pthread，在我看来其实还是比不上 WinThread 的。但是，这反过来似乎也可以说明，在 Linux 下这两个东西几乎已经“够用”了。况且，从原理上去学习和了解；去了解传统和习惯形成的原因，是有助于我们理解这些通常在学术派的 C/C++ 教科书上不会涉及到的知识：进程，线程，流程控制，通讯，甚至包括操作系统的基本原理。在我看来，这些知识是实际使用电脑编写程序完全无法回避的内容，是一个实践者必须掌握的知识。学术界对于程序的一般定义就是：数据＋算法。而在今天我们看来，多进程，多线程，socket 通讯等等，已经完全改变了传统的编写程序的方式和思路。如果让我来定义，我觉得至少在前两者的基础上，还必须加上流程控制（包含了通讯的意思），这样才能算是当代计算机程序的完善定义。

第十四章的内容，估计会比较长，也会比较难。我会按照从 socket 到 SDL_net 的顺序去写，当然争取还是做到“零起步”的通俗易懂。这一部分，也会有一个单独的名字《Linux socket 编程入门教程》。虽然是用 Linux 的名字，但是大家应该知道所有的 Linux 上 TCP/IP 的实现，甚至 UNIX 上的 TCP/IP 实现，基本上都是源于 BSD 的 TCP/IP 实现的，并且直到今天，这些实现都是相互源代码级兼容的。WinSock 一般的说法也是源自 socket，但是具体的函数会有不同。

接下来的知识对于我来说也是个很大的挑战。因为涉及的面太广了。想全部学完是不现实的，我们必须根据自己的需要，从这些知识中寻找出我们需要的部分，恩，努力吧！

Linux socket 编程入门（一）

TCP server 端：1、建模

绝大部分关于 socket 编程的教程总是从 socket 的概念开始讲起的。要知道，socket 的初衷是个庞大的体系，TCP/IP 只是这个庞大体系下一个很小的子集，而我们真正能用上的更是这个子集中的一小部分：运输层（Host-to-Host Transport Layer）的 TCP 和 UDP 协议，以及使用这两个协议进行应用层（Application Layer）的开发。即使是 socket 的核心部分，网络层（Internet Layer）的 IP 协议，在编程的时候我们也很少会感觉到它的存在——因为已经被封装好了，我们唯一需要做的事情就是传入一个宏。第一节我想介绍的概念就这么多，当然，既然我们已经说了 3 个层了，我想最好还是把最后一个层也说出来，即所谓链路层（Network Access Layer），它包括了物理硬件和驱动程序。这四个层从底到高的顺序是：链路层－－网络层－－运输层－－应用层。

好，说实话我们现在并不清楚所谓 TCP 到底是什么东东，不过我们知道这东东名气很大。或许你早就知道，另外一个声名狼藉建立在 TCP 协议基础上的应用程序，它曾经几乎是统治了一个时代，即使是今天，我们依然无法消除他的影响力的——恩，是的，就是 telnet。

在这个教程中，我使用的环境是 Debian GNU/Linux 4.0 etch。传说中的 stable -_-!!!，恩，我是很保守的人。如果你不是自己 DIY 出来的系统，相信默认安装里面就应该有 telnet（/usr/bin/telnet，要是没装就自己 aptitude install 吧）。telnet 可以与所有遵循 TCP 协议的服务器端进行通讯。通常，socket 编程总是 Client/Server 形式的，因为有了 telnet，我们可以先不考虑 client 的程序，我们先写一个支持 TCP 协议的 server 端，然后用 telnet 作为 client 验证我们的程序就好了。

server 端的功能，我们也考虑一种最简单的反馈形式：echo。就如同你在终端输入 echo 'Hello World'，回车后 shell 就会给你返回 Hello World 一样，我们的第一个 TCP server 就用以实现这个功能。

什么样的模型适合描述这样的一种 server 呢？我相信，一个很 2 的例子会有助于我们记忆 TCP server 端的基本流程。

想象你自己是个小大佬，坐办公室（什么样的黑社会做办公室啊？可能是讨债公司吧^^）你很土，只有一个小弟帮你接电话（因为你自己的号码是不敢对外公开的）。一次通讯的流程大概应该是这样的：小弟那里的总机电话响了；小弟接起电话；对方说是你女朋友 A 妹；小弟转达说，“老大，你马子电话”；你说，接过来；小弟把电话接给你；你和你女朋友聊天半小时；挂电话。

我们来分析一下整个过程中的元素。先分析成员数据（请注意，这里开始用 C++术语了）：你小弟（listenSock），你需要他来监听（listen，这是 socket 编程中的术语）电话；你自己（communicationSock），实际上打电话进行交流的是你自己；你的电话号码（servAddr），否则你女朋友怎么能找到你？你女朋友的电话号码（clntAddr），这个比喻有点牵强，因为事实上你接起电话，不需要知道对方的号码也可以通话（虽然事实上你应该是知道的，你不会取消了来电显示功能吧^^），但是，难道你是只接女朋友电话从来不打过去的牛人吗？这个过程中的行为（成员函数）：你小弟接电话并转接给你（isAccept()）；你自己的通话（handleEcho()）（这个行为确实比较土，只会乌鸦学舌的 echo，呵呵）。

简单的说，就是这些了。根据这个模型，我们可以很容易写出实现我们需要的 echo 功能的 TCP server 的类：

```
class TcpServer
{
private:
   int listenSock;
   int communicationSock;
   sockaddr_in servAddr;
   sockaddr_in clntAddr;
public:
   TcpServer(int listen_port);
   bool isAccept();
   void handleEcho();
};
```

这里面有些简写，比如，sock 实际上就是 socket，addr 就是 address。serv 和 clnt 我想你一定能猜到是 server 和 client 吧。还有一个 socket 中的结构体 sockaddr_in，实际上就是这个意思：socket address internet（网络嵌套字地址），具体解说，请看下回分解。

TCP server 端：2、socket与文件描述符

UNIX 中的一切事物都是文件（everything in Unix is a file!）

当我在这篇教程中提到 UNIX 的时候，其意思专指符合 UNIX 标准的所谓“正统”UNIX 的衍生系统（其实我就用来带指那些买了最初 UNIX 源代码的商业系统）操作系统和类似 Linux，BSD 这些类 UNIX 系统。如果某些要点是 Linux 特有的，或者因为本人孤陋寡闻暂时搞不清楚是 Linux 特有的还是 UNIX 通用的，我就会指明是 Linux，甚至其发行版（我本人在写这篇教程的时候是以 Debian GNU/Linux 4.0 etch 为测试平台的）。

我们学习 UNIX 的时候，恐怕听到的第一句话就是这句：UNIX 中一切都是文件。这是 UNIX 的基本理念之一，也是一句很好的概括。比如，很多 UNIX 老鸟会举出个例子来，“你看，/dev/hdc 是个文件，它实际上也是我的光盘……”UNIX 中的文件可以是：网络连接（network connection），输入输出（FIFO），管道（a pipe），终端（terminal），硬盘上的实际文件，或者其它任何东东。

文件与文件描述符（file & file descriptor）

你可能对上一章中建模类中的 int 还记忆犹新。我们用 int 在描述 socket，实际上，所有的文件描述符都是 int，没错，用的是一个整数类型。如果你觉得这样让你很难接受，那么恭喜你，你跟我一样，也许是深中 C++面向对象思想的毒了^^。因为是 int，所以文件描述符不可能是 C++概念中的对象，因为 int 无法发出行为，但是，这并不代表也不能接受一个动作哈。

PASCAL 之父在批判面向对象思想教条的时候，曾经生动的举了个例子，“在 OOP 的概念中，绝对不应该接受 a+b 这种表达的， OOP 对这个问题的表达应该是 a.add(b)”。fd（file descriptor）可以作为接受动作的对象，但是本身却无法发出动作，这就如同一个只能做宾语不能做主语的名词，是个不完整的对象。但是，请别忘了 Linux 和 socket 本身是 C 语言的产物，我们必须接受在面向过程时代下的产物，正视历史——当然，这与我们自己再进行 OOP 的封装并不矛盾。

我们应该记住 3 个已经打开的 fd，0：标准输入（STDIN_FILENO）；1：标准输出（STDOUT_FILENO）；2：标准错误（STDERR_FILENO）。（以上宏定义在<unistd.h>中）一个最简单的使用 fd 的例子，就是使用<unistd.h>中的函数：write(1, "Hello, World!\n", 20);，在标准输出上显示“Hello, World!”。

另外一个需要注意的问题是，file 和 fd 并非一定是一一对应的。当一个 file 被多个程序调用的时候，会生成相互独立的 fd。这个概念可以类比于 C++中的引用（eg: int& rTmp = tmp;）。

socket 与 file descriptor

文件是应用程序与系统（包括特定硬件设备）之间的桥梁，而文件描述符就是应用程序使用这个“桥梁”的接口。在需要的时候，应用程序会向系统申请一个文件，然后将文件的描述符返回供程序使用。返回 socket 的文件通常被创建在/tmp 或者/usr/tmp 中。我们实际上不用关心这些文件，仅仅能够利用返回的 socket 描述符就可以了。

好了，说了这么多，实际上就解释了一个问题，“为什么 socket 的类型是 int？” -_-!!!

TCP server 端：3、sockaddr与sockaddr_in

收件人地址

一家化妆品公司将一批新产品的样品，准备发给某学校某个班的女生们免费试用。通常情况下，这件邮包的地址上可以这么写：
收件人：全体女生。
地址：A 省 B 市 C 学校，X 级 Y 班。
但是，如果在描述地址的时候这样写呢：
收件人：全体女生。
地址：请打电话 xxxxxxxx，找他们学校一个叫 Lucy 的女生，然后把东西送到她的班上。
这种文字是相当的诡异啊-_-!!!，但是并不等于就没有表述清楚邮包的去向和地址。事实上邮局看到这样的地址一定会发飙的，然而对于电脑，如果你的地址描述形式是他可以接受和执行的，他就会老老实实的按你的要求去做……

所以，如何描述地址不是问题的关键，关键在于这样的表述是不是能够表述清楚一个地址。一种更加通用的表达形式可能是这样的：
收件人：全体女生。
地址：<一种地址描述方式>

事实上，在 socket 的通用 address 描述结构 sockaddr 中正是用这样的方式来进行地址描述的：

```
struct sockaddr
{
  unsigned short sa_family;
  char sa_data[14];
};
```

这是一个 16 字节大小的结构（2+14），sa_family 可以认为是 socket address family 的缩写，也可能被简写成 AF（Address Family），他就好像我们例子中那个“收件人：全体女生”一样，虽然事实上有很多 AF 的种类，但是我们这个教程中只用得上大名鼎鼎的 internet 家族

AF_INET。另外的 14 字节是用来描述地址的。这是一种通用结构，事实上，当我们指定 sa_family=AF_INET 之后，sa_data 的形式也就被固定了下来：最前端的 2 字节用于记录 16 位的端口，紧接着的 4 字节用于记录 32 位的 IP 地址，最后的 8 字节清空为零。这就是我们实际在构造 sockaddr 时候用到的结构 sockaddr_in（意指 socket address internet）：

```
struct sockaddr_in
{
  unsigned short sin_family;
  unsigned short sin_port;
  struct in_addr sin_addr;
  char sin_zero[8];
};
```

我想，sin_的意思，就是 socket (address) internet 吧，只不过把 address 省略掉了。sin_addr 被定义成了一个结构，这个结构实际上就是：

```
struct in_addr
{
  unsigned long s_addr;
};
```

in_addr 显然是 internet address 了，s_addr 是什么意思呢？说实话我没猜出值得肯定的答案，也许就是 socket address 的意思吧，尽管跟更广义的 sockaddr 结构意思有所重复了。哎，这些都是历史原因，也许我是没有精力去考究了。

sockaddr 和 sockaddr_in 在 Linux 中的实现

你可能还记得我之前说过，UNIX 和 Linux 上的 socket 实现都是从 BSD 的 socket 实现演变过来的。事实上，socket 这个词本来的意思，就是 Berkeley Socket interface 的简单说法。Linux 上的 socket 与原本的 socket 的应该是完全兼容的，不过发展到今天，在代码实现上可能有些小的差别。我们就吹毛求疵的来看看这些区别在什么地方。

```
#include <bits/socket.h>

/* Structure describing a generic socket address.  */
struct sockaddr
 {
  __SOCKADDR_COMMON (sa_);   /* Common data: address family and length.  */
  char sa_data[14];       /* Address data.  */
 };

//==============

/* POSIX.1g specifies this type name for the `sa_family' member.  */
typedef unsigned short int sa_family_t;

/* This macro is used to declare the initial common members
  of the data types used for socket addresses, `struct sockaddr',
  `struct sockaddr_in', `struct sockaddr_un', etc.  */
```

```
#define    __SOCKADDR_COMMON(sa_prefix) \
  sa_family_t sa_prefix##family

#define __SOCKADDR_COMMON_SIZE    (sizeof (unsigned short int))
```
可以看到，转了几次 typedef，几次宏定义，实际效果是与标准 socket 一样的。
```
#include <netinet/in.h>

/* Internet address.  */
typedef uint32_t in_addr_t;
struct in_addr
  {
    in_addr_t s_addr;
  };

//=================

/* Structure describing an Internet socket address.  */
struct sockaddr_in
  {
    __SOCKADDR_COMMON (sin_);
    in_port_t sin_port;            /* Port number.  */
    struct in_addr sin_addr;       /* Internet address.  */

    /* Pad to size of `struct sockaddr'.  */
    unsigned char sin_zero[sizeof (struct sockaddr) -
            __SOCKADDR_COMMON_SIZE -
            sizeof (in_port_t) -
            sizeof (struct in_addr)];
  };
```
同样的，看起来挺复杂，实际上与标准 socket 的定义是一样的。

头文件依赖关系

<bits/socket.h>是包含在<sys/socket.h>中的，<netinet/in.h>是包含在<arpa/inet.h>中的，实际上我们在程序中往往就是：
```
#include <sys/socket.h>
#include <arpa/inet.h>
```
值得知道的是，ARPA 是 Advanced research project agency（美国国防部高级研究计划署）的所写，ARPANET 是当今互联网的前身，所以我们就可以想象，为什么 inet.h 会在 arpa 目录下了。

TCP server 端：4、构造函数涉及的概念

话题回到“黑社会办公室”的例子，讲概念已经扯得比较远了，不过，这一节我们还得

讲概念，不过好在有些程序的例子。如果大家不想翻回去看 TcpServer 类的原型，我这里直接给出这个头文件的完整源代码：

```
//Filename: TcpServerClass.hpp

#ifndef TCPSERVERCLASS_HPP_INCLUDED
#define TCPSERVERCLASS_HPP_INCLUDED

#include <unistd.h>
#include <iostream>
#include <sys/socket.h>
#include <arpa/inet.h>

class TcpServer
{
private:
   int listenSock;
   int communicationSock;
   sockaddr_in servAddr;
   sockaddr_in clntAddr;
public:
   TcpServer(int listen_port);
   bool isAccept();
   void handleEcho();
};

#endif // TCPSERVERCLASS_HPP_INCLUDED
```

我们已经解释了为什么 listenSock 和 communicationSock 的类型是 int，以及 sockaddr_in 是什么结构，现在来写这个类的构造函数：

```
TcpServer::TcpServer(int listen_port)
{
   if ( (listenSock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0 ) {
      throw "socket() failed";
   }

   memset(&servAddr, 0, sizeof(servAddr));
   servAddr.sin_family = AF_INET;
   servAddr.sin_addr.s_addr = htonl(INADDR_ANY);
   servAddr.sin_port = htons(listen_port);

   if ( bind(listenSock, (sockaddr*)&servAddr, sizeof(servAddr)) < 0 ) {
      throw "bind() failed";
   }

   if ( listen(listenSock, 10) < 0 ) {
```

```
        throw "listen() failed";
    }
}
```

好，先看看程序培养一下感觉，我们还得说概念。

数据封装（Data Encapsutation）

我们前面说到了网络分层：链路——网络——传输——应用。数据从应用程序里诞生，传送到互联网上每一层都会进行一次封装：

Data>>Application>>TCP/UDP>>IP>>OS(Driver, Kernel & Physical Address)

我们用 socket 重点描述的是协议，包括网络协议（IP）和传输协议（TCP/UDP）。

sockaddr 重点描述的是地址，包括 IP 地址和 TCP/UDP 端口。

socket()函数

我们从 TcpServer::TcpServer()函数可以看到，socket 和 sockaddr 的产生是可以相互独立的。

socket()的函数原型是：

```
int socket(int protocolFamily, int type, int protocol);
```

在 Linux 中的实现为：

```
#include <sys/socket.h>

/* Create a new socket of type TYPE in domain DOMAIN, using
   protocol PROTOCOL.  If PROTOCOL is zero, one is chosen automatically.
   Returns a file descriptor for the new socket, or -1 for errors.  */
extern int socket (int __domain, int __type, int __protocol) __THROW;
```

第一个参数是协议簇（Linux 里面叫作域，意思一样的），还是那句话，我们这篇教程用到的就仅仅是一个 PF_INET（protocol family : internet），很多时候你会发现人们也经常在这里赋值为 AF_INET，事实上，当前，AF_INET 就是 PF_INET 的一个#define，但是，写成 PF_INET 从语义上会更加严谨。这也就是 TCP/IP 协议簇中的 IP 协议（Internet Protocol），网络层的协议。

后面两个参数定义传输层的协议。

第二个参数是传输层协议类型，我们教程里用到的宏，只有两个：SOCK_STREAM（数据流格式）和 SOCK_DGRAM（数据报格式）；（具体是什么我们以后讨论）

第三个参数是具体的传输层协议。当赋值为 0 的时候，系统会根据传输层协议类型自动匹配和选择。事实上，当前，匹配 SOCK_STREAM 的就是 TCP 协议；而匹配 SOCK_DGRAM 就是 UDP 协议。所以，我们指定了第二个参数，第三个就可以简单的设置为 0。不过，为了严谨，我们最好还是把具体协议写出来，比如，我们的例子中的 TCP 协议的宏名称：IPPROTO_TCP。

数据的“地址”

从数据封装的模型，我们可以看到数据是怎么从应用程序传递到互联网的。我们说过，数据的传送是通过 socket 进行的。但是 socket 只描述了协议类型。要让数据正确的传送到某个地方，必须添加那个地方的 sockaddr 地址；同样，要能接受网络上的数据，必须有自

己的 sockaddr 地址。

可见，在网络上传送的数据包，是 socket 和 sockaddr 共同“染指”的结果。他们共同封装和指定了一个数据包的网络协议（IP）和 IP 地址，传输协议（TCP/UDP）和端口号。

网络字节和本机字节的相互转换

sockaddr 结构中的 IP 地址（sin_addr.s_addr）和端口号（sin_port）将被封装到网络上传送的数据包中，所以，它的结构形式需要保证是网络字节形式。我们这里用到的函数是 htons()和 htonl()，这些缩写的意思是：
h: host，主机（本机）
n: network，网络
to: to 转换
s: short，16 位（2 字节，常用于端口号）
l: long, 32 位（4 字节，常用于 IP 地址）
“反过来”的函数也是存在的 ntohs()和 ntohl()。

动作与持续行为

本节最后的一个概念可以跟计算机无关。作为动词，有些可以描述动作，有些是描述一重持续的行为状态的（就如同一般动词和 be 动词一样）。扯到 C++来说，我们可以把持续行为封装到函数内部，只留出动作的接口。事实上，构造函数中的 bind()和 listen()就是这种描述持续状态的行为函数。

TCP server 端：5、创建监听嵌套字

前面一小节，我们已经写出了 TcpServer 的构造函数。这个函数的实际作用，就是创建了 listen socket（监听嵌套字）。这一节，我们来具体分析这个创建的过程。

socket 和 sockaddr 的创建是可以相互独立的

在函数中，我们首先通过 socket()系统调用创建了 listenSock，然后通过为结构体赋值的方法具体定义了服务器端的 sockaddr。（memset()函数的作用是把某个内存段的空间设定为某值，这里是清零。）其他的概念已经在前一小节讲完了。这里需要补充的是说明宏定义 INADDR_ANY。这里的意思是使用本机所有可用的 IP 地址。当然，如果你机器绑定了多个 IP 地址，你也可以指定使用哪一个。

数据流简易模型（SOCK_STREAM）

我们的例子以电话做的比喻，实际上，socket stream 模型不完全类似电话，它至少有以下这些特点：
1、一种持续性的连接。这点跟电话是类似的，也可以想象成流动着液体的水管。一旦断开，这种流动就会中断。
2、数据包的发送实际上是非连续的。这个世界上有什么事物是真正的线性连续的？呵呵，扯远了，这貌似一个哲学问题。我们仅仅需要知道的是，一个数据包不可能是无限大的，所

以，总是一个小数据包一个小数据包这样的发送的。这一点，又有点像邮包的传递。这些数据包到达与否，到达的先后次序本身是无法保证的，即是说，是 IP 协议无法保证的。但是 stream 形式的 TCP 协议，在 IP 之上，做了一定到达和到达顺序的保证。
3、传送管道实际上是非封闭的。要不干嘛叫“网络”-_-!!!。我们之所以能保证数据包的“定点”传送，完全是依靠每个数据包都自带了目的地址信息。

由此可见，虽然 socket 和 sockaddr 可以分别创建，并无依赖关系。但是在实际使用的时候，一个 socket 至少会绑定一个本机的 sockaddr，没有自己的“地址信息”，就不能接受到网络上的数据包（至少在 TCP 协议里面是这样的）。

socket 与本机 sockaddr 的绑定

有时候绑定是系统的任务，特别是当你不需要知道自己的 IP 地址和所使用的端口号的时候。但是，我们现在是建立服务器，你必须告诉客户端你的连接信息：IP 和 Port。所以，我们需要指明 IP 和 Port，然后进行绑定。
int bind(int socket, struct sockaddr* localAddress, unsigned int addressLength);
作为 C++的程序员，也许你会觉得这个函数很不友好，它似乎更应该写成：
int bind_cpp_style(int socket, const sockaddr& localAddress);
我们需要通过函数原型指明两点：
1、我们仅仅使用 sockaddr 结构的数据，但并不会对原有的数据进行修改；
2、我们使用的是完整的结构体，而不仅仅是这个结构体的指针。（很显然光用指针是无法说明结构体大小的）
幸运的是，在 Linux 的实现中，这个函数已经被写为：
```
#include <sys/socket.h>

/* Give the socket FD the local address ADDR (which is LEN bytes long).  */
extern int bind (int __fd, __CONST_SOCKADDR_ARG __addr, socklen_t __len)
    __THROW;
```
看到亲切的 const，我们就知道这个指针带入是没有“副作用”的。

监听：listen()

stream 流模型形式上是一种“持续性”的连接，这就是要求信息的流动是“可来可去”的。也就是说，stream 流的 socket 除了绑定本机的 sockaddr，还应该拥有对方 sockaddr 的信息。在 listen()中，这“对方的 sockaddr”就可以不是某一个特定的 sockaddr。实际上，listen socket 的目的是准备被动的接受来自“所有”sockaddr 的请求。所以，listen()反而就不能指定某个特定的 sockaddr。
int listen(int socket, int queueLimit);
其中第二个参数是等待队列的限制，一般设置在 5-20。Linux 中实现为：
```
#include <sys/socket.h>

/* Prepare to accept connections on socket FD.
   N connection requests will be queued before further requests are refused.
   Returns 0 on success, -1 for errors.  */
extern int listen (int __fd, int __n) __THROW;
```

完成了这一步，回到我们的例子，就像是让你小弟在电话机前做好了接电话的准备工作。需要再次强调的是，这些行为仅仅是改变了 socket 的状态，实际上我想强调的是，为什么这些函数不会造成 block（阻塞）的原因。（block 的概念以后再解释）

TCP server 端：6、创建“通讯 ”嵌套字

这里的“通讯”加上了引号，是因为实际上所有的 socket 都有通讯的功能，只是在我们的例子中，之前那个 socket 只负责 listen，而这个 socket 负责接受信息并 echo 回去。

我们现看看这个函数：

```
bool TcpServer::isAccept()
{
   unsigned int clntAddrLen = sizeof(clntAddr);

   if ( (communicationSock = accept(listenSock, (sockaddr*)&clntAddr, &clntAddrLen)) < 0 ) {
      return false;
   } else {
      std::cout << "Client(IP: " << inet_ntoa(clntAddr.sin_addr) << ") connected.\n";
      return true;
   }
}
```

用 accept()创建新的 socket

在我们的例子中，communicationSock 实际上是用函数 accept()创建的。

```
int accept(int socket, struct sockaddr* clientAddress, unsigned int* addressLength);
```

在 Linux 中的实现为：

```
/* Await a connection on socket FD.
  When a connection arrives, open a new socket to communicate with it,
  set *ADDR (which is *ADDR_LEN bytes long) to the address of the connecting
  peer and *ADDR_LEN to the address's actual length, and return the
  new socket's descriptor, or -1 for errors.

  This function is a cancellation point and therefore not marked with
  __THROW.  */
extern int accept (int __fd, __SOCKADDR_ARG __addr,
        socklen_t *__restrict __addr_len);
```

这个函数实际上起着构造 socket 作用的仅仅只有第一个参数（另外还有一个不在这个函数内表现出来的因素，后面会讨论到），后面两个指针都有副作用，在 socket 创建后，会将客户端 sockaddr 的数据以及结构体的大小传回。

当程序调用 accept()的时候，程序有可能就停下来等 accept()的结果。这就是我们前一小节说到的 block（阻塞）。这如同我们调用 std::cin 的时候系统会等待输入直到回车一样。accept()是一个有可能引起 block 的函数。请注意我说的是“有可能”，这是因为 accept()的 block 与否实际上决定与第一个参数 socket 的属性。这个文件描述符如果是 block 的，accept()就 block，否则就不 block。默认情况下，socket 的属性是“可读可写”，并且，是阻塞的。所以，

我们不修改 socket 属性的时候，accept()是阻塞的。

accept()的另一面 connect()

accept()只是在 server 端被动的等待，它所响应的，是 client 端 connect()函数：

```
int connect(int socket, struct sockaddr* foreignAddress, unsigned int addressLength);
```

虽然我们这里不打算详细说明这个 client 端的函数，但是我们可以看出来，这个函数与之前我们介绍的 bind()有几分相似，特别在 Linux 的实现中：

```
/* Open a connection on socket FD to peer at ADDR (which LEN bytes long).
   For connectionless socket types, just set the default address to send to
   and the only address from which to accept transmissions.
   Return 0 on success, -1 for errors.

   This function is a cancellation point and therefore not marked with
   __THROW.  */
extern int connect (int __fd, __CONST_SOCKADDR_ARG __addr, socklen_t __len);
```

connect() 也使用了 const 的 sockaddr，只不过是远程电脑上的而非 bind()的本机。

accept()在 server 端表面上是通过 listen socket 创建了新的 socket，实际上，这种行为是在接受对方客户机程序中 connect()函数的请求后发生的。综合起看，被创建的新 socket 实际上包含了 listen socket 的信息以及客户端 connect()请求中所包含的信息——客户端的 sockaddr 地址。

新 socket 与 sockaddr 的关系

accept()创建的新 socket（我们例子中的 communicationSock，这里我们简单用 newSock 来带指）首先包含了 listen socket 的信息，所以，newSock 具有本机 sockaddr 的信息；其次，因为它响应于 client 端 connect()函数的请求，所以，它还包含了 clinet 端 sockaddr 的信息。

我们说过，stream 流形式的 TCP 协议实际上是建立起一个“可来可去”的通道。用于 listen 的通道，远程机的目标地址是不确定的；但是 newSock 却是有指定的本机地址和远程机地址，所以，这个 socket，才是我们真正用于 TCP“通讯”的 socket。

inet_ntoa()

```
#include <arpa/inet.h>

/* Convert Internet number in IN to ASCII representation.  The return value
   is a pointer to an internal array containing the string.  */
extern char *inet_ntoa (struct in_addr __in) __THROW;
```

对于这个函数，我们可以作为一种，将 IP 地址，由 in_addr 结构转换为可读的 ASCII 形式的固定用法。

TCP server 端：7、接收与发送

现在，我们通过 accept()创建了新的 socket，也就是我们类中的数据成员 communicationSock，现在，我们就可以通过这个 socket 进行通讯了。

TCP 通讯模型

在介绍函数之前，我们应该了解一些事实。TCP 的 Server/Client 模型类似这样：
ServApp——ServSock——Internet——ClntSock——ClntApp
当然，我们这里的 socket 指的就是用于“通讯”的 socket。TCP 的 server 端至少有两个 socket，一个用于监听，一个用于通讯；TCP 的 server 端可以只有一个 socket，这个 socket 同时“插”在 server 的两个 socket 上。当然，插上 listen socket 的目的只是为了创建 communication socket，创建完备后，listen 是可以关闭的。但是，如果这样，其他的 client 就无法再连接上 server 了。

我们这个模型，是 client 的 socket 插在 server 的 communication socket 上的示意。这两个 socket，都拥有完整的本地地址信息以及远程计算机地址信息，所以，这两个 socket 以及之间的网络实际上形成了一条形式上“封闭”的管道。数据包只要从一端进来，就能知道出去的目的地，反之亦然。这正是 TCP 协议，数据流形式抽象化以及实现。因为不再需要指明“出处”和“去向”，对这样的 socket（实际上是 S/C 上的 socket 对）的操作，就如同对本地文件描述符的操作一样。但是，尽管我们可以使用 read()和 write()，但是，为了完美的控制，我们最好使用 recv()和 send()。

recv()和 send()
int send(int socket, const void* msg, unsigned int msgLength, int flags);
int recv(int socket, void* rcvBuffer, unsigned int bufferLength, int flags);
在 Linux 中的实现为：
#include <sys/socket.h>

```
/* Send N bytes of BUF to socket FD.  Returns the number sent or -1.

   This function is a cancellation point and therefore not marked with
   __THROW.  */
extern ssize_t send (int __fd, __const void *__buf, size_t __n, int __flags);

/* Read N bytes into BUF from socket FD.
   Returns the number read or -1 for errors.

   This function is a cancellation point and therefore not marked with
   __THROW.  */
extern ssize_t recv (int __fd, void *__buf, size_t __n, int __flags);
```

这两个函数的第一个参数是用于“通讯”的 socket，第二个参数是发送或者接收数据的起始点指针，第三个参数是数据长度，第四个参数是控制符号（默认属性设置为 0 就可以了）。失败时候传回-1，否则传回实际发送或者接收数据的大小，返回 0 往往意味着连接断开了。

处理 echo 行为

```
void TcpServer::handleEcho()
{
    const int BUFFERSIZE = 32;
```

```
    char buffer[BUFFERSIZE];
    int recvMsgSize;
    bool goon = true;

    while ( goon == true ) {
        if ( (recvMsgSize = recv(communicationSock, buffer, BUFFERSIZE, 0)) < 0 ) {
            throw "recv() failed";
        } else if ( recvMsgSize == 0 ) {
            goon = false;
        } else {
            if ( send(communicationSock, buffer, recvMsgSize, 0) != recvMsgSize ) {
                throw "send() failed";
            }
        }
    }

    close(communicationSock);
}
```

本小节最后要讲的函数是 close()，它包含在<unistd.h>中

```
#include <unistd.h>

/* Close the file descriptor FD.

   This function is a cancellation point and therefore not marked with
   __THROW.  */
extern int close (int __fd);
```

这个函数用于关闭一个文件描述符，自然，也就可以用于关闭 socket。

下一小节是完整的源代码。默认的监听端口是 5000。我们可以通过

```
$telnet 127.0.0.1 5000
```

验证在本机运行的 echo server 程序。

TCP server 端：8、本章的完整源代码

```
//Filename: TcpServerClass.hpp

#ifndef TCPSERVERCLASS_HPP_INCLUDED
#define TCPSERVERCLASS_HPP_INCLUDED

#include <unistd.h>
#include <iostream>
#include <sys/socket.h>
#include <arpa/inet.h>

class TcpServer
```

```
{
private:
   int listenSock;
   int communicationSock;
   sockaddr_in servAddr;
   sockaddr_in clntAddr;
public:
   TcpServer(int listen_port);
   bool isAccept();
   void handleEcho();
};


#endif // TCPSERVERCLASS_HPP_INCLUDED

//Filename: TcpServerClass.cpp

#include "TcpServerClass.hpp"

TcpServer::TcpServer(int listen_port)
{
   if ( (listenSock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0 ) {
      throw "socket() failed";
   }

   memset(&servAddr, 0, sizeof(servAddr));
   servAddr.sin_family = AF_INET;
   servAddr.sin_addr.s_addr = htonl(INADDR_ANY);
   servAddr.sin_port = htons(listen_port);

   if ( bind(listenSock, (sockaddr*)&servAddr, sizeof(servAddr)) < 0 ) {
      throw "bind() failed";
   }

   if ( listen(listenSock, 10) < 0 ) {
      throw "listen() failed";
   }
}

bool TcpServer::isAccept()
{
   unsigned int clntAddrLen = sizeof(clntAddr);

   if ( (communicationSock = accept(listenSock, (sockaddr*)&clntAddr, &clntAddrLen)) < 0 ) {
```

```
        return false;
    } else {
        std::cout << "Client(IP: " << inet_ntoa(clntAddr.sin_addr) << ") connected.\n";
        return true;
    }
}

void TcpServer::handleEcho()
{
    const int BUFFERSIZE = 32;
    char buffer[BUFFERSIZE];
    int recvMsgSize;
    bool goon = true;

    while ( goon == true ) {
        if ( (recvMsgSize = recv(communicationSock, buffer, BUFFERSIZE, 0)) < 0 ) {
            throw "recv() failed";
        } else if ( recvMsgSize == 0 ) {
            goon = false;
        } else {
            if ( send(communicationSock, buffer, recvMsgSize, 0) != recvMsgSize ) {
                throw "send() failed";
            }
        }
    }

    close(communicationSock);
}
```

演示程序：

```
//Filename: main.cpp
//Tcp Server C++ style, single work

#include <iostream>
#include "TcpServerClass.hpp"

int echo_server(int argc, char* argv[]);

int main(int argc, char* argv[])
{
    int mainRtn = 0;
    try {
        mainRtn = echo_server(argc, argv);
    }
```

```
    catch ( const char* s ) {
        perror(s);
        exit(EXIT_FAILURE);
    }

    return mainRtn;
}

int echo_server(int argc, char* argv[])
{
    int port;
    if ( argc == 2 ) {
        port = atoi(argv[1]);
    } else {
        port = 5000;
    }

    TcpServer myServ(port);

    while ( true ) {
        if ( myServ.isAccept() == true ) {
            myServ.handleEcho();
        }
    }

    return 0;
}
```

其他

SDL在win32 与Linux下的一些差别

最近在学习Linux，主要考虑到跨平台开发。我一直都在思考关于中文输入的问题，在win32 下解决起来那么麻烦，难道SDL的作者会不解决吗——如果是bug的话。事实证明，在Linux平台下，我们自己把GB2312 往Unicode转换的工作就多余了。当前Linux内部用的就是UTF-8（我的系统是Debian 4.0），SDL的TTF函数完全可以正常工作。不过，在Linux下面也会有些问题，现在发现的有这些，大家有发现的继续帮我补充哈。

1、在Linux下，TTF_RenderUTF8_Blended 是可以正常渲染简体中文的。但是不知道为什么，Solid的渲染会报错，共享（动态链接）库的问题。
2、在Linux下，mid文件用SDL_mixer是打不开的。至少我的情况是这样，我装了ALSA的声卡驱动，不知道还需要做些什么才能放midi。

3、直接装了ALSA的驱动播放SDL_mixer是有噪音的。解决方法，也是打开Linux下软件混音(ESD)的方法如下：

混音可以让两种音乐同时播放，首先需要安装libesd-alsa0，用Synaptic或sudo apt-get install等方式都可以进行安装，它就在Ubuntu官方软件库中。然后，用sudo gedit或您喜欢的文本编辑器创建文件

/etc/asound.conf

```
pcm.card0 {
  type hw
  card 0
}

pcm.!default {
  type plug
  slave.pcm "dmixer"
}

pcm.dmixer {
  type dmix
  ipc_key 1025
  slave {
    pcm "hw:0,0"
    period_time 0
    period_size 1024
    buffer_size 4096
    periods 128
    rate 44100
  }
  bindings {
    0 0
    1 1
  }
}
```

接下来，运行"sudo gedit /etc/esound/esd.conf"，将文件改成下面的内容：

```
auto_spawn=1
spawn_options=-terminate -nobeeps -as 2 -d default
spawn_wait_ms=100
# default options are used in spawned and non-spawned mode
default_options=
```

属实挺麻烦的:)

Code::Blocks在Debian下的绿色安装

不想看废话的直接下载用就是了。我已经打包设置好了。^^

简短描述:

媲美于 VC 的 C++ IDE 集成开发环境，具有较完善的自动补全功能。基于官方的 Debian 8.02 最新版本。已包含 libwx2.8。（Debian etch 源上的是 2.6）。无须 root 权限安装，解压即可使用。

安装步骤:
解压到自己安排的位置。cd 到相关目录，运行 sh 脚本 setup.sh 就可以了。将生成 sh 脚本 codeblocks，可直接通过该脚本启动。（实际上就是设置了 LD_LIBRARY_PATH 和 CODEBLOCKS_DATA_DIR）

下载地址：
http://www.fs2you.com/zh-cn/files/7de0ba0c-428f-11dd-812f-00142218fc6e/

下面介绍自己进行绿色安装的过程。
首先到Code::Blocks的项目页下载最新的版本 8.02。
http://www.codeblocks.org/
当然，我用的是Debian，下载的就是这个版本：codeblocks-8.02debian-i386.tar.gz。
解压后是一堆deb包，小心的把这些包解压到某个位置下的usr目录下。(注意，不是默认的/usr目录)。然后这个文件夹的名字不重要，是可以更换的。比如，我就换成了codeblocks。这时候，这个目录下应该有这些文件夹：bin, include, lib, share。其中，可执行文件在bin目录下。
cd到bin目录下，检查codeblocks的依赖库及其路径：
ldd codeblocks
然后可以看到一大堆没有找到的 so：

linux-gate.so.1 => (0xffffe000)
libgtk-x11-2.0.so.0 => /usr/lib/libgtk-x11-2.0.so.0 (0xb7ca1000)
libgdk-x11-2.0.so.0 => /usr/lib/libgdk-x11-2.0.so.0 (0xb7c20000)
libatk-1.0.so.0 => /usr/lib/libatk-1.0.so.0 (0xb7c06000)
libgdk_pixbuf-2.0.so.0 => /usr/lib/libgdk_pixbuf-2.0.so.0 (0xb7bf0000)
libpangocairo-1.0.so.0 => /usr/lib/libpangocairo-1.0.so.0 (0xb7be8000)
libXext.so.6 => /usr/lib/libXext.so.6 (0xb7bd9000)
libXinerama.so.1 => /usr/lib/libXinerama.so.1 (0xb7bd6000)
libXi.so.6 => /usr/lib/libXi.so.6 (0xb7bce000)
libXrandr.so.2 => /usr/lib/libXrandr.so.2 (0xb7bcb000)
libXcursor.so.1 => /usr/lib/libXcursor.so.1 (0xb7bc2000)
libXfixes.so.3 => /usr/lib/libXfixes.so.3 (0xb7bbd000)
libpango-1.0.so.0 => /usr/lib/libpango-1.0.so.0 (0xb7b82000)
libcairo.so.2 => /usr/lib/libcairo.so.2 (0xb7b20000)
libfreetype.so.6 => /usr/lib/libfreetype.so.6 (0xb7ab6000)
libz.so.1 => /usr/lib/libz.so.1 (0xb7aa2000)
libfontconfig.so.1 => /usr/lib/libfontconfig.so.1 (0xb7a77000)
libpng12.so.0 => /usr/lib/libpng12.so.0 (0xb7a54000)
libXrender.so.1 => /usr/lib/libXrender.so.1 (0xb7a4b000)
libX11.so.6 => /usr/lib/libX11.so.6 (0xb795f000)
libgobject-2.0.so.0 => /usr/lib/libgobject-2.0.so.0 (0xb7925000)
libgmodule-2.0.so.0 => /usr/lib/libgmodule-2.0.so.0 (0xb7922000)

libglib-2.0.so.0 => /usr/lib/libglib-2.0.so.0 (0xb7890000)
librt.so.1 => /lib/tls/i686/cmov/librt.so.1 (0xb7887000)
libcodeblocks.so.0 => not found
libwx_gtk2u_richtext-2.8.so.0 => not found
libwx_gtk2u_aui-2.8.so.0 => not found
libwx_gtk2u_xrc-2.8.so.0 => not found
libwx_gtk2u_qa-2.8.so.0 => not found
libwx_gtk2u_html-2.8.so.0 => not found
libwx_gtk2u_adv-2.8.so.0 => not found
libwx_gtk2u_core-2.8.so.0 => not found
libwx_baseu_xml-2.8.so.0 => not found
libwx_baseu_net-2.8.so.0 => not found
libwx_baseu-2.8.so.0 => not found
libpthread.so.0 => /lib/tls/i686/cmov/libpthread.so.0 (0xb7872000)
libdl.so.2 => /lib/tls/i686/cmov/libdl.so.2 (0xb786e000)
libstdc++.so.6 => /usr/lib/libstdc++.so.6 (0xb7789000)
libm.so.6 => /lib/tls/i686/cmov/libm.so.6 (0xb7764000)
libgcc_s.so.1 => /lib/libgcc_s.so.1 (0xb7759000)
libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7627000)
libpangoft2-1.0.so.0 => /usr/lib/libpangoft2-1.0.so.0 (0xb75fc000)
libXau.so.6 => /usr/lib/libXau.so.6 (0xb75f9000)
libexpat.so.1 => /usr/lib/libexpat.so.1 (0xb75d9000)
libXdmcp.so.6 => /usr/lib/libXdmcp.so.6 (0xb75d3000)
/lib/ld-linux.so.2 (0xb7f9c000)

其中 libcodeblocks.so.0 是 Code::Blocks 自带的，已经在 lib 目录下了，应为我们没有指定 LD_LABRARY_PATH 路径而找不到。

其它库是属于 libwx 的。在源中搜索一下：

aptitude search libwx

很不幸，源里面只有 2.4 和 2.6 版本的。所以，我们只好去wx官方去下载 2.8 的库。

http://www.wxwidgets.org/

我们在：

http://apt.wxwidgets.org/dists/etch-wx/main/binary-i386/

下找到我们需要的共享库文件：

libwxbase2.8-0_2.8.7.1-0_i386.deb 和 libwxgtk2.8-0_2.8.7.1-0_i386.deb

（真快，今天已经更新 2.8.8.0 了-_-!!!应该也没问题，不过我没试）

把解压出来的 so 和 s-link 都放到 codeblocks/lib/ 文件夹下面。最后，我们设置 LD_LABRARY_PATH就可以了。

可以直接用这个脚本设置：

```
#!/bin/sh

echo "#!/bin/sh" > codeblocks
echo "LD_LIBRARY_PATH=$PWD/lib CODEBLOCKS_DATA_DIR=$PWD $PWD/bin/codebl
ocks" >> codeblocks
```

chmod +x codeblocks

生成的启动脚本应该类似这样：（$PWD 表示当前目录，也就是 codeblocks 所在的位置。）

#!/bin/sh
LD_LIBRARY_PATH=/home/lf426/app/code/codeblocks/lib CODEBLOCKS_DATA_DIR=/home/lf426/app/code/codeblocks /home/lf426/app/code/codeblocks/bin/codeblocks

通过这个脚本就可以启动 Code::Blocks 了。整个过程不需要 root 权限哦（你不会想当 root 敢死队吧。^^）

Linux下C++ IDE的选择。

SF被封了！NND，本来还说申请了个项目页，大家可以更容易的下载源代码的，现在又恶心了。只有用代理了。Code::Blocks项目也是在SF上的，昨天晚上还好好的。好在这个项目在德国某个网站还有个镜像，大家应该能下得到的。

Linux下的C++IDE环境一直是一个缠人的问题。当然，很多黑客告诉我们vim足矣，可是我们毕竟还是很难以记住那么多函数。我想，win32 API确实更难以记忆吧，所以M$搞出个VC这样的好东东；Linux下的牛人们都是从 70 年代开始玩C的人，他们多多少少更习惯C而不是C++，包括Linus本人还跟BS老大打过口水战。所以，vim对于他们，对于C，可能已经相当够用了——但是，我们是凡人，我们还是希望有个自动补全功能比较强，也不用每次都自己写Makefile的IDE。

其实一开始关注的是Anjuta，因为这是号称Gnome下的原生品，可惜啊，功能实在是太一般般了，都还不如用vim加各种插件呢。后来又试了eclipse 比Anjuta强了不少，这个项目好像是得到哪个大公司资助了，搞得很华丽的样子。可是，基于Java，速度还是慢啊。另外，逼着我装JVM也让我比较不爽。当前Linux下大有Python淘汰Java的趋势。KDevelop听说是不错，可是我还没开始用KDE，貌似非原生品，怕麻烦，忍了。

昨天在论坛上看到大家讨论codeblocks，之前在Windows下关注过，后来传闻说不再维护了，就没放在心上。传闻毕竟是传闻，昨天到codeblocks项目页上看了下，最新的更新是在今年 2 月底呵，有够新的。估计是之前周期太长了吧。慢功出细活，本着对德国人认真态度的敬佩，下载下来试了一下——果然，比eclipse要快很多啊。哈哈，虽然跟VC还是有差距，不过也算是用起来比较舒心的IDE了。所以推荐给大家。下篇文章，我来讲讲在Debian 4.0 ecth下的绿色安装方法。

构建vim的可视化C++编辑平台

Vim是一个强大的文本编辑器。我们在win32 平台下用VC写代码的时候，常常比较习惯VC给予的一些格式控制和代码补全功能，而这些功能都是可以在vim中实现的，并且，已经有前人为我们写好了插件脚本，我们只需要非常简单的设置，就可以迅速的构建出类似VC的C++编译环境。

Vim的配置文件通常在~/.vimrc中。我们首先可以加入这样两句：

```
syntax on           //语法高亮打开，这样我们就可以看倒C/C++的关键字成为了彩色；
```

:set cin!　　//C缩进打开，这样vim会自动判断缩进的设置；

完成这样的设置后，当我们创建或者打开vim所能识别的C/C++源文件的时候，我们就可以看到关键字的颜色以及自动的缩进了。

需要指出的一点是，按照Unix源程序的习惯，我们最好以ASCII编码来写程序，而Debian的内部编码默认是UTF-8。当然，如果我们vim创建的文件没有汉字等其他字符的时候，产生的文件自动是ASCII编码的，但是，某些插件，比如vim的c-support插件会自动添加文件的创建日期，这样，因为我们系统默认语言一般是zh_CN，所以就会产生中文的日期显示。所以，一个解决方法是在运行创建cpp文件的vim的时候，指定使用的语言，或者，干脆做一个alias来简单使用：

alias cvim='LC_ALL=C vim'

这样，就可以确保创建的cpp文件不带非ASCII字符。

接下来，我们简单了解一下ctags。简单说，ctags是为许多计算机语言的源代码文件做索引，以提供给编辑器（比如我们的vim）使用的。我们先看看ctags的帮助：

ctags --help

以下信息是我们将用到的：

-R　Equivalent to --recurse.

--recurse=[yes|no]　Recurse into directories supplied on command line [no].

--languages=[+|-]list

　　Restrict files scanned for tags to those mapped to langauges

　　specified in the comma-separated 'list'. The list can contain any

　　built-in or user-defined language [all].

--fields=[+|-]flags

　　Include selected extension fields (flags: "afmikKlnsStz") [fks].

--extra=[+|-]flags

　　Include extra tag entries for selected information (flags: "fq").

我们用这样的命令在源文件所在的目录运行：（用样，我们可以做一个alias，我们最后来讨论这个问题）

ctags -R --c++-kinds=+p --fields=+iaS --extra=+q

其中参数的含义是：[原文出处]

--c++-kinds=+p　：为C++文件增加函数原型的标签

--fields=+iaS　：在标签文件中加入继承信息(i)、类成员的访问控制信息(a)、以及函数的指纹(S)

--extra=+q　　：为标签增加类修饰符。注意，如果没有此选项，将不能对类成员补全

好了，现在准备知识讲完了。我们先实现C++对象用.或者->调用方法的时候自动产生选择的下拉菜单。我们需要的插件是OmniCppComplete，在vim官方这里下载：

http://www.vim.org/scripts/script.php?script_id=1520

下载到的是一个压缩包，解压到~/.vim/下相应的目录就可以了。另外，我们需要在~/.vimrc中打开装载插件：

:filetype plugin on

（或许还需要打开vim对源代码的识别，不过我是没有用到，Debian 4.0，如果需要的话，再加上:filetype indent on）

另外，我们关闭vim默认的预览窗口：

:set completeopt=longest,menu

这样，当我们创建了对象，用.或者->的时候，就会自动调出类方法的选择窗口了。

另外一个常见的补全是对函数的补全，也有现成的插件可以用code_complete：
http://www.vim.org/scripts/script.php?script_id=1764
这个安装就更简单了，直接把code_complete.vim拷贝到~/.vim/plugin/下面就可以了。
插件的使用方法是，当写完一个函数，并打上左括号(之后，按tab键就可以看倒函数参数列表了，包括重载的函数。

最后，我们不希望记那么多参数，希望使用起来简单一些，我们只需要在bash运行的时候申明我们的两个alias就可以了。在~/.bashrc中添加：

```
alias vctags='ctags -R --c++-kinds=+p --fields=+iaS --extra=+q'
alias cvim='LC_ALL=C vim'
```

关闭终端后重启动，可以用type查看我们的新命令已经生效了：

```
lf426@fleet:~$ type vctags
vctags is aliased to `ctags -R --c++-kinds=+p --fields=+iaS --extra=+q'
```