# Kinect使用:

GetDefaultKinectSensor () -------- IKinectSensor

IKinectSensor :: get\_DepthFrameSource() ------ IDepthFrameSource

IDepthFrameSource:: get\_FrameDescription()------ IFrameDescription，可用于获得帧分辨率

IDepthFrameSource :: OpenReader()------IDepthFrameReader

IDepthFrameReader:: AcquireLatestFrame()------IDepthFrame

IDepthFrame :: CopyFrameDataToArray()-------Mat

assert(SUCCEEDED(hr))：

HRESULT hr是用来判断com接口方法的返回值，当hr=0时，表示接口方法调用成功，返回值为S\_OK；当hr=1时，表示接口方法调用成功，但返回值为S\_FALSE；当hr<0时，表示接口方法调用中遇到错误，调用失败。

SUCCEEDED(hr);是用来判断com接口方法的返回值hr是否正确运行，当hr小于0时表示接口方法遇到错误，于是SUCCEEDED的结果就是FALSE，否则就是TRUE

assert()是调试下的一个确保性的判断，保证其参数值为TRUE，如果参数为FALSE将弹出错误提示窗口。

assert(SUCCEEDED(hr))就是确保hr是个非负整数。等价于assert(hr>=0)

while (hr < 0 || pDepthFrame == NULL)

hr = m\_pDepthFrameReader->AcquireLatestFrame(&pDepthFrame);/\*由于有时候获取不到，因此循环获取最近的帧\*/

USHORT distMin = 0;//获取最大、最小深度距离信息

USHORT distMax = 0;

assert (hr >= 0);

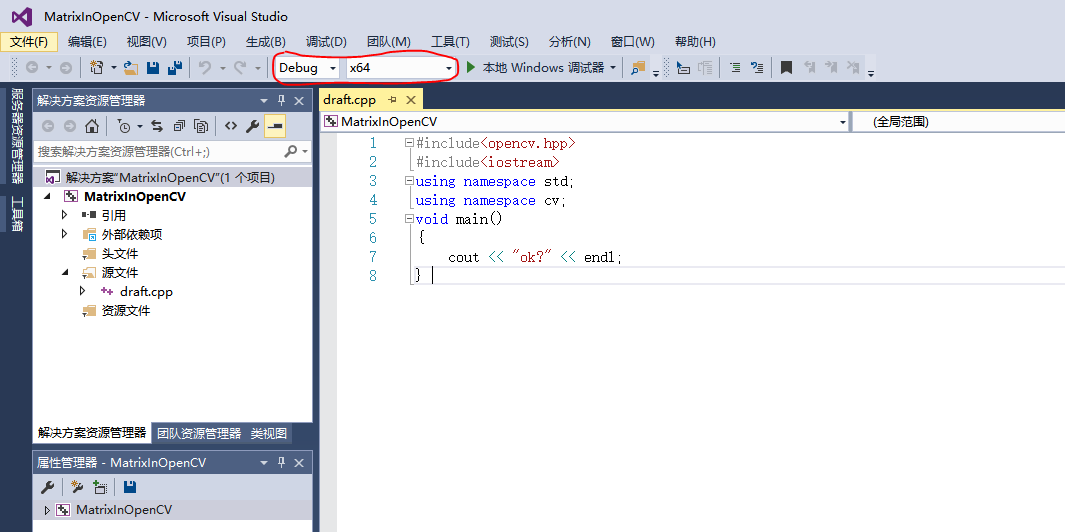
hr = pDepthFrame->get\_DepthMinReliableDistance(&distMin);

assert (hr >= 0);

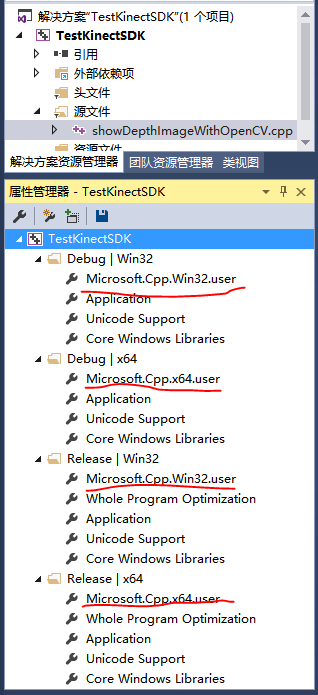
hr = pDepthFrame->get\_DepthMaxReliableDistance(&distMax);

# 安装OpenCV、VS环境配置：

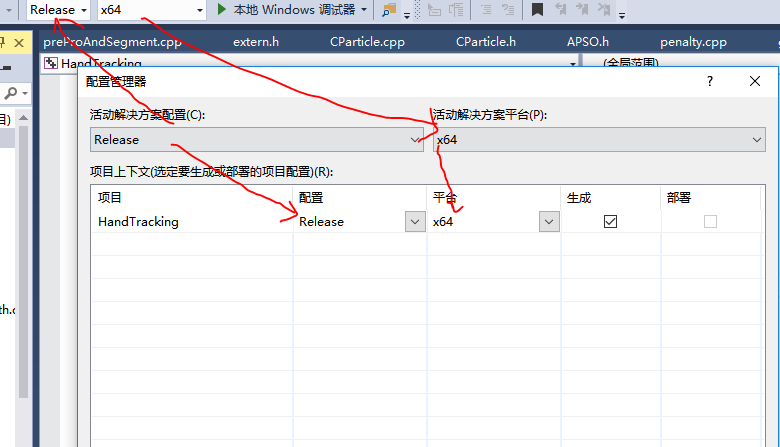
1. 早期版本的某些OpenCV SDK在build文件夹下可能有两个文件x86和x64。x86对应编译程序时使用x86编译器， x64对应编译程序时使用x64编译器.



1. OpenCV3.2和OpenCV3.3的SDK，其build文件夹只有X64文件夹。因此编译选项只能是Debug x64 或者Release x64。要想使用Debug x86 或者Release x86编译选项，需要使用CMake软件生成相应配置环境下的X86文件夹。
2. 在VS2015中配置开发环境：属性管理器下有四个文件夹，在每个文件夹下的xxxxxxx.user双击打开属性页面，然后在该页面配置开发环境，这样就实现了“在属性管理器中进行一次配置，就相当于进行了通用的配置过程，以后新建的工程就不用再额外的进行重新配置了”。
3. 属性管理器下的四个文件夹，各自对应一种编译选项。因此配置开发环境时，每个编译选项对应的xxxxxxx.user都要配置。配置Debug|Win32或Release|Win32时，路径选“…bulid\x86…”路径；配置Debug|Win64或Release|Win64时，路径选“…bulid\x64…” 路径。



1. 点击下图编译选项设置处的下拉箭头，打开配置管理器。可以看到，Debug、Release的专业术语是活动解决方案配置（在项目上下文框中简称为配置），x86、x64是活动解决方案平台（在项目上下文框中简称为平台）。



1. 具体教程：

<http://blog.csdn.net/sinat_27953939/article/details/68948313> opencv3.2在vs2015开发环境搭建+cmake3.8编译生成opencv x86版本库。其中【通用属性】 ->【链接器】->【输入】->【附加的依赖项】，添加的lib即OpenCV SDK解压后opencv\build\x86\vc10\lib（x86代表使用x86编译器，因此也可能x64，视具体情况而定；vc10代表的是VS的版本，因此也可能是vc14，视具体情况而定）下所有lib库文件的名字。

# Visual Studio

## VC-VS对应版本

vc6 -   VC6.0

vc7 -   VS2002

vc7.1 - VS2003

vc8 -  VS2005

vc9 -  VS2008

vc10   -   VS2010

vc11   -   VS2012

vc12   -   VS2013

vc13   -   VS2014

vc14   -   VS2015

Visual C++ 2005（也即Visual C++ 8.0）

Visual C++ 2008（也即Visual C++ 9.0）

Visual C++ 2010（也即Visual C++ 10.0）

Visual C++ 2012（也即Visual C++ 11.0

Visual C++ 2013（也即Visual C++ 12.0）

Visual C++ 2015（也即Visual C++ 14.0）

## VS2015编译运行程序：

* 一个解决方案中可以包含多个项目

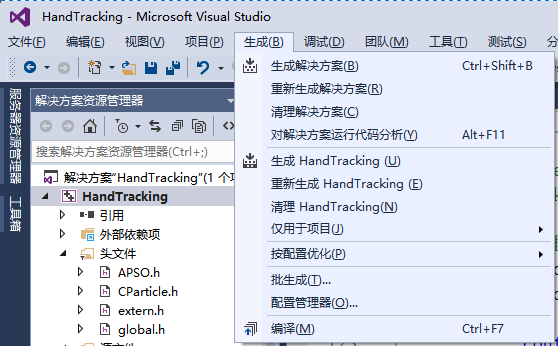
生成-生成解决方案：只对改动过的文件重新生成，没有改动过的文件不会重新生成

生成-重新生成解决方案：对所有的文件都重新生成

只改动某些.cpp之类的文件的时候可以用生成，省了编译没有改动的那些文件的时间；但是改动了某些.h之类的文件最好用重新生成，因为有可能有些文件包含.h文件也需要重新编译

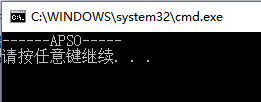
* 运行程序的操作顺序：

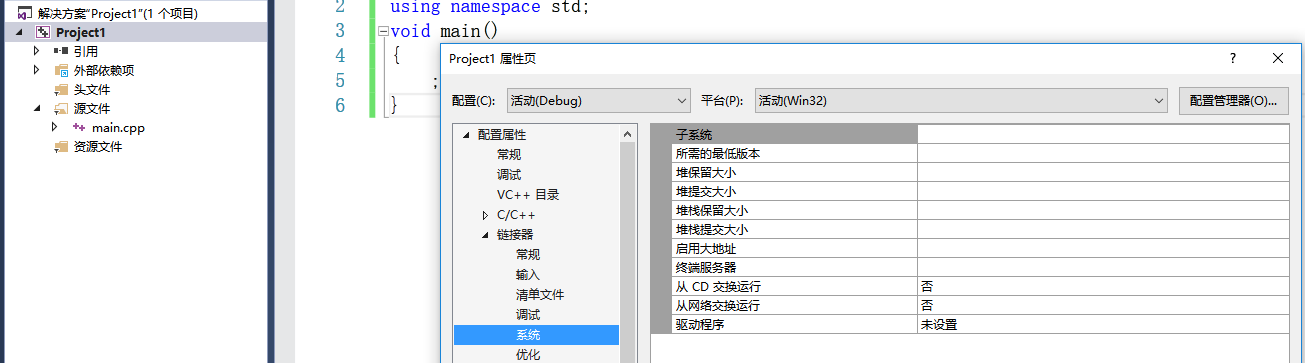
1. ctrl+F7：编译
2. 重新生成解决方案（或生成解决方案），重新生成项目（或生成项目）
3. ctrl+F5: 开始执行（不调试）

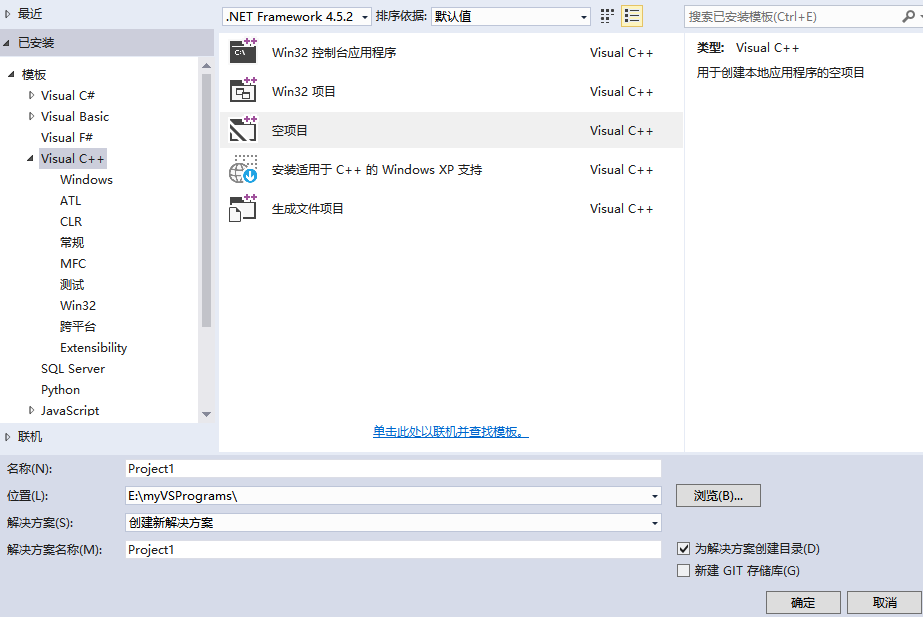


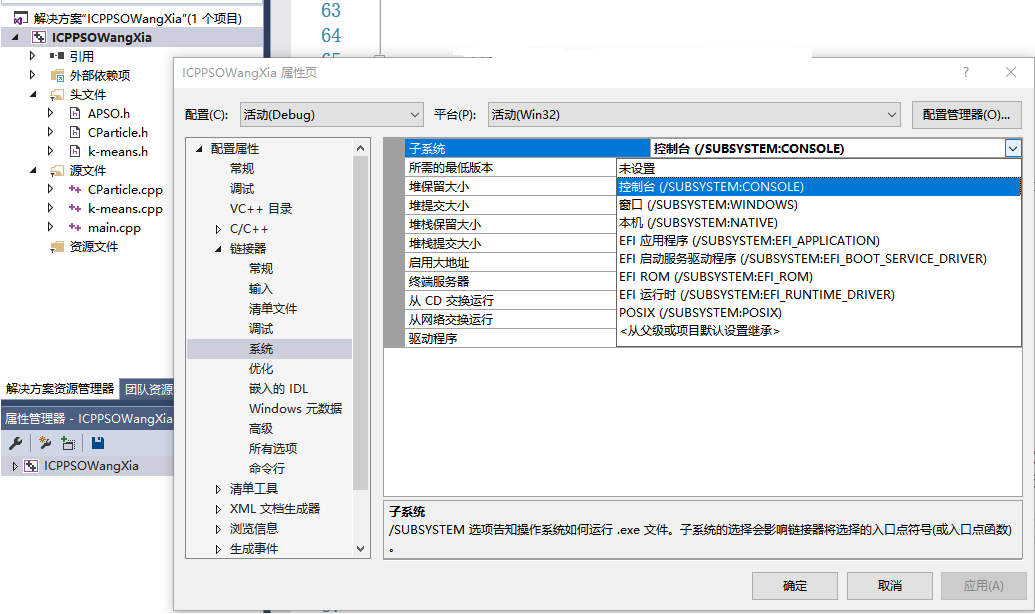
## VS中Win32控制台应用程序：

VS中新建项目时，出现以下选项，包括Win32控制台应用程序、Win32项目、空项目，区别如下：

* win32控制台程序在操作系统的入口函数是main函数，通常是调用C:\windows\System32目录下的某个exe文件形成系统32位的DOS窗口，即通常所见的黑底灰白字的窗口。程序结束后，窗口保留，提示“请按任意键继续. . .”
* win32应用程序在操作系统的入口函数是winMain函数，可以搭建在MFC等库下，通常用来生成系统窗口，即可以自定义按钮和菜单的窗口，或者直接用来编写后台程序，并不生成窗口。
* 空项目也形成32位DOS窗口，程序结束后，DOS窗口直接关闭，所以运行空项目时DOS窗口经常一闪而过，甚至看不到运行结果
* 这3种类型的程序除在新建项目时选择外，也可以在建好了项目后改变项目类型，可以在"菜单\项目\属性\链接\系统\子系统"下进行选择，win32项目的选项是“subsystem:windows"，而Win32控制台应用程序的选项是”subsystem:console，空项目的选型是空白







## VS中Debug和Release模式的区别：

Debug 和 Release 并没有本质的区别，他们只是VC预定义提供的两组编译选项的集合，编译器只是按照预定的选项行动。如果我们愿意，我们完全可以把Debug和Release的行为完全颠倒过来。当然也可以提供其他的模式，例如自己定义一组编译选项，然后命名为MY\_ABC等。

Debug通常称为调试版本，通过一系列编译选项的配合，编译的结果通常包含调试信息，而且不做任何优化，便于开发人员调试

Release通常称为发布版本，是为用户使用的，不用于调试，不保存调试信息，同时，它往往进行了各种优化，以期达到代码最小和速度最优。

Debug 和 Release 的真正秘密，在于一组编译选项。下面列出了分别针对二者的一些重要选项：

**Debug 版本：**

* + /MDd /MLd 或 /MTd 使用 Debug runtime library（调试版本的运行时刻函数库）
  + /Od 关闭优化开关
  + /D "\_DEBUG" 相当于 #define \_DEBUG，打开编译调试代码开关（主要针对assert函数）
  + /ZI 创建 Edit and continue（编辑继续）数据库，这样在调试过程中如果修改了源代码不需重新编译
  + /GZ 可以帮助捕获内存错误
  + /Gm 打开最小化重链接开关，减少链接时间

**Release 版本：**

* + /MD /ML 或 /MT 使用发布版本的运行时刻函数库
  + /O1 或 /O2 优化开关，使程序最小或最快
  + /D "NDEBUG" 关闭条件编译调试代码开关（即不编译assert函数）
  + /GF 合并重复的字符串，并将字符串常量放到只读内存，防止被修改

**运行时库Runtime Library**

Runtime Library，运行时库，C++程序在运行时所需要的库文件（如C++标准库里的函数、类等），有静态库和动态库可供选择。

VC++中有六种Runtime Library:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 类型 | 简称 | 含义 | 对应的库名称 | 备注 |
| Single-Threaded | /ML | Release版的单线程静态库 | libc.lib | VS2003以后被废弃 |
| Single-Threaded Debug | /MLd | Debug版的单线程静态库 | libcd.lib | VS2003以后被废弃 |
| Multi-threaded | /MT | Release版的多线程静态库 | libcmt.lib |  |
| Multi-threaded Debug | /MTd | Debug版的多线程静态库 | libcmtd.lib |  |
| Multi-threaded DLL | /MD | Release版的多线程动态库 | msvcrt.lib+msvcrtxx.dll  (动态库及其导出库) |  |
| Multi-threaded DLL Debug | MDd | Debug版的多线程动态库 | msvcrtd.lib+msvcrtxxd.dll  （动态库及其导出库） |  |

(1) Release版多线程静态库 /MT

选择多线程静态库，则：(1)库被链接到可执行程序exe中，(2)可以在多线程程序中使用。 会获得轻微的性能提升，但最终生成的二进制代码因链入庞大的运行时库实现而变得非常臃肿。

以下代码Debug编译选项下，配置属性—C/C++—代码生成—运行库，选择MDd，最终生成的exe文件48k，如果选MTd，则最终生成的exe文件1414k

#include<iostream>

#include<windows.h>

**using** **namespace** std;

**int** main()

{

    constexpr **int** len = 128;

**wchar\_t** sysdir[len];

    GetSystemDirectory(sysdir, len) ;//C:\Windows\system32，无论x86还是x64编译选项

    wcout << sysdir << endl;

    GetWindowsDirectory(sysdir, len);//C:\Windows

    wcout << sysdir << endl;

**return** 0;

}

(2) Release版多线程动态库 /MD

多线程动态运行时库将所有的 C 库函数保存在一个单独的动态链接库 MSVCRTxx.DLL 中， MSVCRTxx.DLL 处理了多线程问题。链接时将按照传统VC链接dll的方式使用MSVCRxx.DLL的导入库MSVCRT.lib；在运行时要求PC上安装了相应版本的VC运行时库（当然把这些运行时库dll放在应用程序目录下也是可以的）。 因/MD和/MDd方式不会将运行时库链接到可执行文件内部，可有效减少可执行文件尺寸。

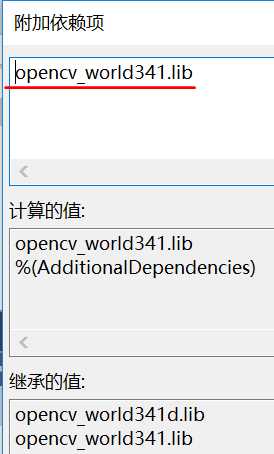
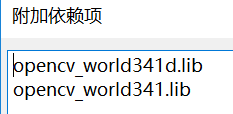
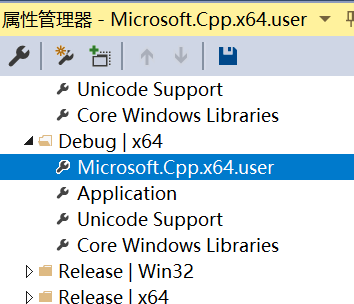
/MDd 、 /MLd 或 /MTd 选项使用 Debug runtime library( 调试版本的运行时刻函数库 ) ，与 /MD 、 /ML 或 /MT 分别对应。 Debug 版本的 Runtime Library 包含了调试信息，并采用了一些保护机制以帮助发现错误，加强了对错误的检测，因此在运行性能方面比不上 Release 版本。

结论：/MD和/MDd将是潮流所趋，/ML和/MLd方式请及时放弃，/MT和/MTd在非必要时最好也不要采用了。

**常见问题：**

OpenCV在release模式下不能imshow图片（在debug模式下正常）的解决方法：

正确的解决方法： 配置属性->C/C++->代码生成->运行库Debug下默认是“多线程调试 DLL (/MDd)”， release下默认是“多线程 DLL (/MD)”，不要乱动必须保持这样。出现release模式下不能imshow图片的原因是，一劳永逸地配置OpenCV库时，如下在链接器-输入-附加依赖项里先添加debug下的lib文件，再添加release 下的lib。这样，编译选项为Debug时没有问题，但当编译选项为Release时，即会出错。因此在Release编译模式下，需要在当前项目的属性页里的链接器-输入-附加依赖项里重新添加release 下的lib。



## VS中解决方案和工程：

一个 Solution (解决方案)可以有多个 Project

应用场景：当一个项目由多个不同的组件(模块)构成时，为每一个组件创建一个project，所有组件的project在同一个解决方案下。

设置启动工程，在有多个工程的 Solution 中要设置启动工程(也就是要第一个开始执行的工程)，不然你按 F5 运行时不知道从哪个工程开始执行。选择某个工程名右键鼠标->Set as Startup Project

## 控制台应用程序和windows程序的切换：

* 项目Configuration Properties->C/C++->Preprocessor,然后右边栏的Preprocessor Definitions对应的项中，控制台程序是\_CONSOLE、windows程序是\_WINDOWS，对QT工程，这两个预处理宏都没有
* 项目Configuration Properties->Linker->System,然后右边栏的SubSystem对应的项中，控制台程序是CONSOLE(/SUBSYSTEM:CONSOLE)、windows程序是Windows(/SUBSYSTEM:WINDOWS)

## VS性能分析与优化：

profiler:探查器

elapsed：已用的

分析🡪性能探查器🡪性能向导🡪检测

**Num of Calls:**(函数)调用次数

**Elapsed Inclusive Time:**已用非独占时间

**Elapsed Exclusive Time:**已用独占时间

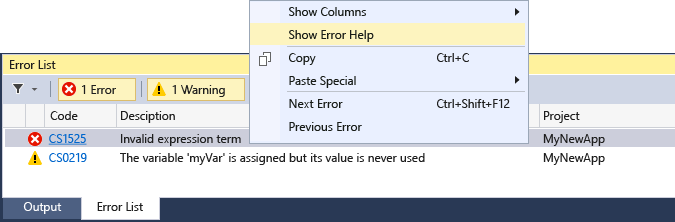
非独占样本数是指的包括了子函数执行时间的总执行时间

独占样本数是不包括子函数执行时间的函数体执行时间，函数执行本身花费的时间，不包括子(函数)树执行的时间

## VS Debugger:

* VS查看函数重载，光标在（|）里面 按Ctrl Shift + 空格
* 在相应的 Error 或者 Warning 条目上右击，会出现一个菜单，点击 Show Error Help，将自动执行该 Error 或者 Warning 的 Bing 搜索。

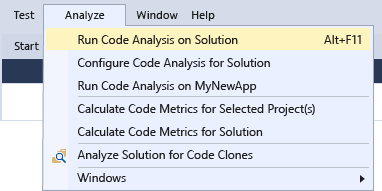
也可以点击 Code 列中的错误代码超链接，如 CS1525，同样会执行该错误代码的 Bing 搜索



* 静态分析

解决了Build过程中ErrorList出现的所有Error之后，可以尝试执行代码静态分析。代码静态分析会去分析代码中可能的导致运行时错误等问题，并给出Warning。

不要觉得没Erorr了，Build成功了，就万事大吉了，在一开始的时候，尽量解决代码中出现的Warning，可以省去将来不必要的麻烦。可以点击如下菜单，执行代码静态分析。



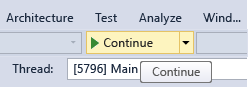
Warning 同样出现在 Error List 窗口，点击 「 Warning」选项卡查看。

* F5调试：

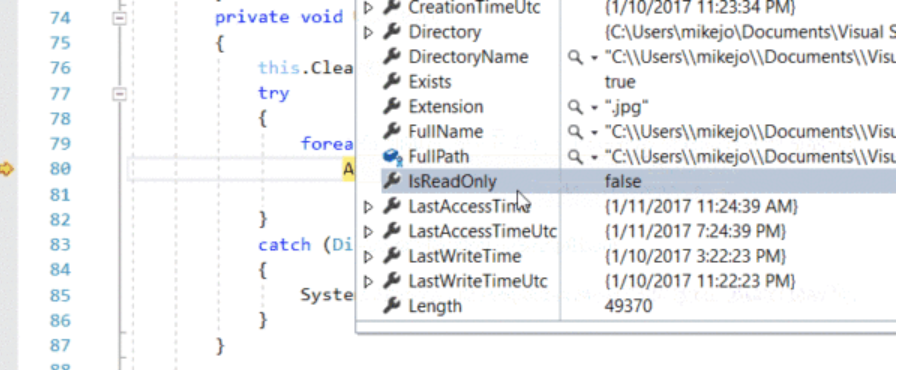
按下 Break 或 Stop 按钮来暂停或停止程序。



按下 Continue 按钮继续执行程序。



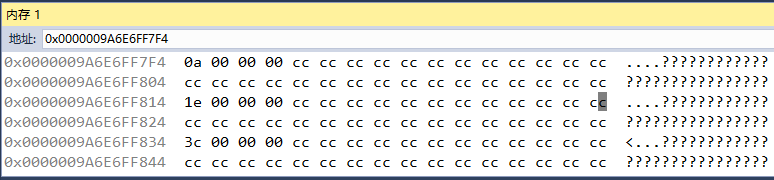
* 设置断点，程序执行到断点处，停下来的时候，可以查看代码中的变量，堆栈调用情况等，看下它们是不是在预期的范围内。将鼠标悬停在变量上面，可以看到它此刻的值。



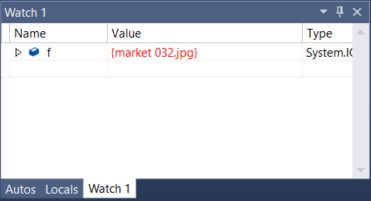
* 调试->窗口->内存，Memory  窗口显示的是应用程序的内存空间，可以查看指定地址中的内容。

在地址栏中输入地址的时候，除了用具体的 32 bit 地址，也可以用指针（前提是该指针在当前范围内），如对变量a查看内存，可以输入&a

右上方的列表示一行显示多少个 Byte 的数据，可以选择 8，16，32，64 等。



* Watch窗口（监视窗口）显示的是自己想要查看的变量或者表达式。



在变量处右击选择「Add Watch」，即可把变量添加至 Watch 窗口，进而在程序执行的时候，查看该变量值的变化情况

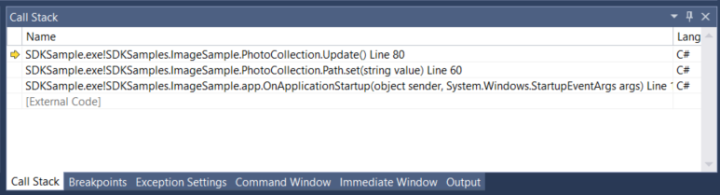
借助Watch窗口可以让VS[调试器正确显示UTF-8字符串](http://www.cppblog.com/sunicdavy/archive/2012/02/14/165566.html)

默认的, VC调试器只能正常显示GBK字符串, 而UTF-8字符串及其他格式则无法显示

需要将要显示的字符串拉到Watch中,并在变量后面添加「,s8」即可显示。如

[image](http://www.cppblog.com/images/cppblog_com/sunicdavy/Windows-Live-Writer/165611ed03f8_D62F/image_4.png) --> [image](http://www.cppblog.com/images/cppblog_com/sunicdavy/Windows-Live-Writer/165611ed03f8_D62F/image_2.png)

* Autos窗口（自动窗口）显示的是当前行以及前面一行（C#，VB）或者三行（C++）的变量和当前值
* Locals窗口（局部变量）显示的是当前范围内定义的变量，通常是当前正在执行的函数或方法

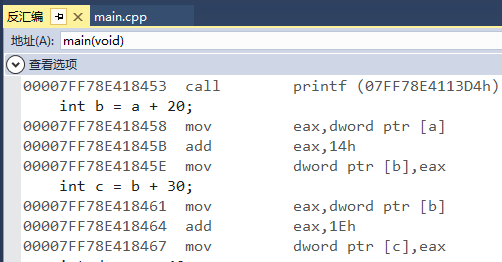


* Call Stack 窗口显示的是函数或者方法的调用顺序。

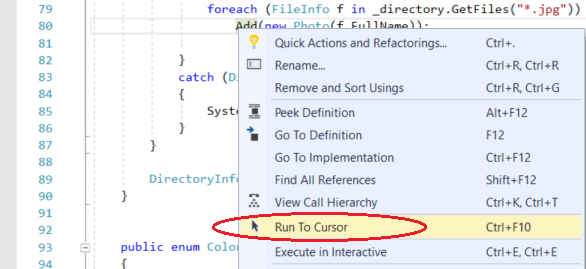
如上图，第一行黄色箭头指向的是当前正在执行的函数 （本例中为 Update)。第二行代表的是第一行的被调函数（本例中，Path.set 调用 Update），以此类推。

在调用堆栈窗口双击某个函数，则可直接跳转到源码处

* 查看汇编代码：调试->窗口->反汇编



* 在某代码行右键选择 **Run to Cursor**，将启动 Debugger，运行到此处暂停



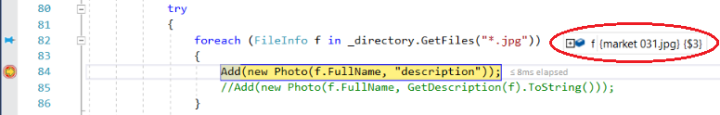
* Data Tips

Debug 的过程中，通过 Data Tips 查看某些变量的值，常规的做法是，把光标放在想要查看的变量上，等 Data Tips 出现。

如果想要频繁查看变量值的话，这样会比较麻烦。其实，可以按下 Data Tips 窗口旁边类似于钉子的按钮，这样就可以把 Data Tips 窗口 Pin 住。

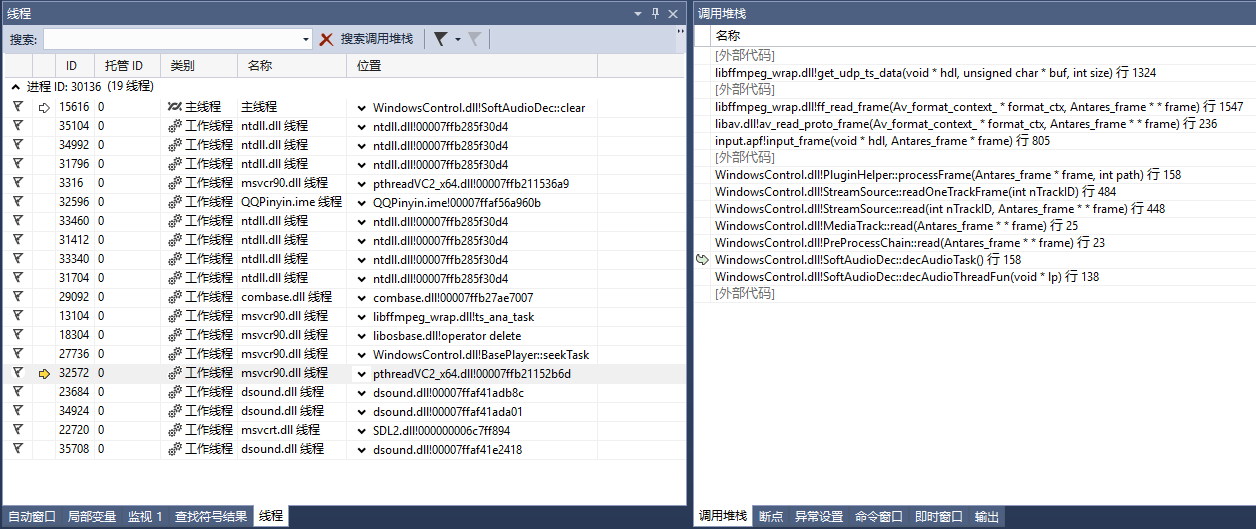
即使重启 Debugger，它依然是保持 Pin 的状态。另外，可以同时 Pin 住多个变量的 Data Tips。

这个功能跟把变量放进 Watch 窗口查看，效果是一样的。



VS可以在Debug的过程中编辑代码，而不必退出Debugger再编辑重新编译运行

## VS多线程调试：



调试时，在pthread\_join(m\_audioThread, NULL);处点击下一过程，程序卡死

m\_audioThread对应的线程入口函数为decAudioTask()，则肯定是decAudioTask线程不能结束导致卡死

decAudioTask()中调用了其它函数，为了确定是哪个调用导致线程卡死，调试->窗口->线程打开线程窗口，如上图左，列出了进程的所有线程

在线程窗口列表逐个双击线程，则切换到该线程，查看调用堆栈，如果发现某个线程对应的调用堆栈有decAudioTask()函数，则说明这是导致卡死的线程

查看调用堆栈，发现最终调用的函数为get\_udp\_ts\_data()、ff\_read\_frame()……，依次转到这些函数确定是哪个函数没有退出导致卡死

## vs编译含inline的C文件

使用vs编译c文件时，可能遇到出现 inline”之后应输入“(”错误，

解决方案是：

在该头文件中加入

#if defined(WIN32) && !defined(\_\_cplusplus)

#define inline \_\_inline

#endif

## VS与文件编码问题：

（1）警告warning C4819: The file contains a character that cannot be represented in the current code page (936). Save the file in Unicode format to prevent data loss

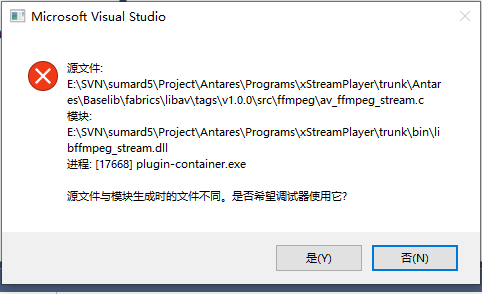
如果不解决此警告，可能出现编译不通过，或者调试断点不能命中等奇怪的问题

解决方法：打开出现warning的文件，Ctrl+A全选，然后在文件菜单：file->advanced save options ，在弹出的选项中选择新的编码方式为：UNICODE codepage 1200 ，点击确定，问题即可解决。

（2）报错如下，可能是文件字符编码或换行符的难题



点击上图中的“浏览并查找av\_ffmpeg\_stream.c”，则



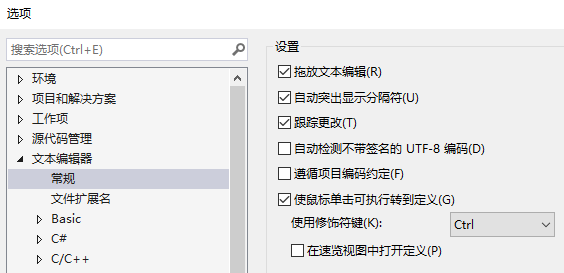
（3）VS在解决方案资源管理器右键项目配置属性，打开的属性不一定与项目当前编译选项配置，如当前为x64，打开的属性页为x86，需要手动调整属性页为x64的属性页

（4）为什么avformat-open-input突然不能传入中文路径了？

文件编码格式有问题，虽然VS页面显示的中文正常，但保存，用VSCode打开该.cpp文件，无论是哪种编码格式，都显示乱码

（5）将GB2312编码的源文件拷贝到VS工程目录下，并在解决方案资源管理器里添加到VS工程，重新生成解决方案后，VS自动修改文件编码为UTF-8无签名

解决方法：**Tools/Options/Text Editor** and uncheck "**Follow project coding conventions**"



# OpenCV：

## OpenCV中Mat矩阵：

CV\_PI

Mat mat(320, 240, CV\_8UC1); //元素未被初始化

OpenCV中Mat对象，可以直接用cout<<打印

double aArray[3][3] = { {10,5,3},{6,4,7},{1,0,9} };

double bArray[3][4] = { {1,3,8,10},{7,5,4,20},{10,6,0,30}}; //C++中二维数组的定义

Mat A(3, 3, CV\_64F, aArray);

Mat B(3,4,CV\_64F,bArray); //用二维数组初始化Mat对象

cout << "A matrix is:"<<A << endl<<endl;

cout << "B matrix is:" << B << endl;

cout << "A\*B=" << endl << " " << A\*B << endl;

cout << "A矩阵的逆=" << endl << " " << A.inv() << endl;

cout << "按位除A/B" << endl << " " << A / A << endl;

CV\_[每一项的位数] [S|U|F]C[通道数]：S表示有符号的，U表示无符号的，F表示浮点数

CV\_8UC3： 8 位无符号，三个通道

CV\_32FC1：表示32位浮点数，1通道

CV\_8UC3：表示8位无符号整型，3通道

1. //例子1：创建100\*90的矩阵，矩阵元素为3通道32位浮点型
2. cv::Mat M(100, 90, CV\_32FC3);
3. Mat dst;
5. // 【1】创建与src同类型和大小的矩阵(dst)
6. dst.create( src1.size(), src1.type() );

Mat mat(10,3,CV\_64FC1,a); //OpenCV的PCA输入必须要是单信道32位浮点数格式或是单信道64位浮点数格式的,参数为CV\_32FC1或是CV\_64FC1

Mat的存储是逐行存储的，矩阵中的数据类型包括：

Mat\_<uchar>对应的是CV\_8U，

Mat\_<char>对应的是CV\_8S，

Mat\_<int>对应的是CV\_32S，

Mat\_<float>对应的是CV\_32F，

Mat\_<double>对应的是CV\_64F

对于Mat 对象：

data指向数据区的首地址 uchar\* data

对于二维矩阵，step[0]是一行占用的字节数，step[1]是一个元素占用的字节数

对于二维矩阵第i行第j列元素的地址： M.data + M.step[0] \* i + M.step[1] \* j

访问二维Mat的元素：

unsigned short int arc[3][2] = { {3,4},{5,6},{7,8} };

Mat mat(3, 2, CV\_16UC1,arc);

cout << mat.at<unsigned short>(1, 1) << endl;

OpenCV访问矩阵Mat元素：

int a[3][5] = { {1,2,3,4,5},{ 11,12,13,14,15 },{ 111,112,113,114,115 } };

Mat map(3, 5, CV\_32SC1, a);

1. at()方式

这种方法适用于随机访问。在需要连续扫描所有点时并不推荐，奇慢无比

int sum1 = 0;

for (int i = 0; i < map.rows; i++)

for (int j = 0; j < map.cols; j++)

sum1+=map.at<int>(i, j);用指针直接定位到(i, j)

1. 借助data用指针直接定位到(i, j)

int sum2 = 0;

for (int i = 0; i < map.rows; i++)

for (int j = 0; j < map.cols; j++)

sum2 += \*((int\*)(map.data + i\*map.step[0] + j\*map.step[1]));

1. 用data先获取行指针、再获取列指针的方式

int sum3 = 0;

for (int i = 0; i < map.rows; i++)

{

int\* p = (int\*)(map.data + i\*map.step[0]);

for (int j = 0; j < map.cols; j++)

sum3 += p[j];

}

在debug模式下，比较而言，用data先获取行指针、再获取列指针的方式，比用指针直接定位到(i, j)效率高，at最慢。在release模式下，各种方法的执行效率相同，且基本都比debug模式略快

多通道矩阵Mat使用at(i,j)返回的是Vec类型，如Vec2b、 Vec2s、 Vec2w、 Vec2i、 Vec2f、 Vec2d，2可以是3、4、5、6等，表示几个通道

typedef Vec<uchar,2> Vec2b; //2个元素的向量，元素类型是unsigned char

typedef Vec<uchar,3> Vec3b;

typedef Vec<short,2> Vec2s;

typedef Vec<short,3> Vec3s;

typedef Vec<ushort, 2> Vec2w;

typedef Vec<ushort, 3> Vec3w;

typedef Vec<int, 2> Vec2i;

typedef Vec<int, 3> Vec3i;

typedef Vec<float, 2> Vec2f;

typedef Vec<float, 3> Vec3f;

typedef Vec<double, 4> Vec4d;

typedef Vec<double, 6> Vec6d;

使用Mat::at()访问RGB图像(CV\_8UC3)的3个通道：

Vec3b即Vec<unchar,3>，Vec是个模板类，用来实现向量，可使用[]操作符访问Vec元素，使用+、-等进行向量运算

Mat imgRGB = imread("C:/Users/myth/Pictures/forTest/bupt.jpg");

Vec3b v= imgRGB.at<Vec3b>(i, j); //i行j列的一个像素点的B、G、R分量，存为Vec3b类型

结果如[110, 103, 106]，数值类型

访问B分量：

cout<<v[0]<<endl;

或者cout << imgRGB.at<Vec3b>(i, j)[0] << endl;

结果显示n，unsigned char类型，是个字符 字符n对应的ASCII码是110

要显示单个通道的值，cout << (int)(imgRGB.at<Vec3b>(i, j)[0]) << endl;

img.at(i, j) = Vec3b(255, 255, 255); //将(i,j)处的像素点置为(255，255，255)

## OpenCV中矩阵normalize

normalize(src, dst, double alpha=1, double beta=0, NORM\_MINMAX,-1);

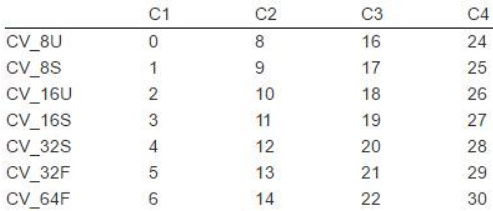
NORM\_MINMAX：矩阵的数值被平移或缩放到一个指定的范围，线性归一化

alpha和beta是要归一化至的范围

-1表示dst的type与src的type一致

## OpenCV中Mat::type()

mat有一个type()的函数可以返回该Mat的类型。类型表示了矩阵中元素的类型以及矩阵的通道个数，具体的有以下值：



CV\_8U和CV\_8UC1等价、CV\_16U和CV\_16UC1等价、CV\_32F和CV\_32FC1等价

cout << CV\_8UC1 << " " << CV\_8U << endl; //0 0

cout << CV\_16UC1 << " " << CV\_16U << endl; //2 2

cout << CV\_32F << " " << CV\_32FC1 << endl; //5 5

## OpenCV中Mat最大/小值及位置

* 查找最大值、最小值并定位：

double minValue, maxValue;

Point minPoint, maxPoint;

minMaxLoc(distMat,&minValue,&maxValue,&minPoint,&maxPoint);

minMaxIdx(distHist,0,&max);//只找最大值，则另一个参数置0

* 仅查找最大值、最小值：

double minValue, maxValue;

minMaxIdx(copy\_original\_depthMap, &minValue, &maxValue);

## OpenCV的Mat::setTo()函数：

* 将矩阵所有元素设为3 ：mat.setTo(3);
* 将矩阵小于10的元素设为3：mat.setTo(3,mat<10)
* 将矩阵小于10或大于20的元素设为3：mat.setTo(3,mat<10|mat>20)
* 将矩阵大于5且小于10的元素设为3：mat.setTo(3,mat>5&mat<10)
* matA.setTo(3,matA>5&matA<10)，矩阵matA和matB尺寸相同

## OpenCV的Mat::copyTo()函数

copyTo()的高阶用法同Mat::setTo()函数

## OpenCV中waitKey()

OpenCV中，waitKey函数的原型int waitKey(int delay=0)

延迟delay毫秒，如果在此期间有按键按下，则立即结束并返回按下按键的ASCII码，否则返回-1。

如果delay为0，则除非有按键，不然不会返回值

经常程序里面出现if( waitKey(10) >= 0 ) 是说10ms中按任意键进入此if块

while (1) { if (waitKey(0) == '\r') { break; } //除非按回车键，不然一直停留

## OpenCV中distanceTransform()：

distanceTransform：单通道图，返回非0像素点到0像素点的最小距离

void distanceTransform (InputArray src, OutputArray dst, int distanceType, int maskSize)

1. src
2. 输入 8-比特、单通道图像.
3. dst
4. 包含计算出的距离的输出图像(32-比特、浮点数、单通道).

距离类型：CV\_DIST\_L1， CV\_DIST\_C，CV\_DIST\_L2（欧氏距离）

maskSize ：距离变换掩模的大小，可以是 3 或 5. 对 CV\_DIST\_L1 或 CV\_DIST\_C 的情况，参数值被强制设定为 3, 因为 3×3 mask 给出和 5×5 mask 一样的结果，而且速度还更快。

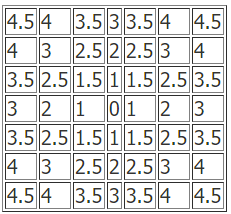
距离类型定义了三个参数：a：水平/垂直位移；b：对角线位移c：knight's 移动（国际象棋里的马步）

OpenCV 使用 [Borgefors86] 推荐的值：

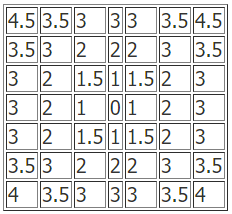
* CV\_DIST\_C (3×3): a=1, b=1
* CV\_DIST\_L1 (3×3): a=1, b=2
* CV\_DIST\_L2 (3×3): a=0.955, b=1.3693
* CV\_DIST\_L2 (5×5): a=1, b=1.4, c=2.1969

下面用户自定义距离的距离域示例 (黑点 (0) 在白色方块中间），

用户自定义 3×3 mask (a=1, b=1.5)：



用户自定义 5×5 mask (a=1, b=1.5, c=2)：



## OpenCV的imshow()函数：

imshow()显示的Mat类型是CV\_32F时，期望所有的像素值在[0.0，1.0]范围，如果超过1.0，则都按1.0处理

imshow()显示的Mat类型是CV\_8U时，期望所有的像素值在[0，255]范围，如果超过255，则都按255处理

一般配合使用convertTo()的缩放因子

Mat distMap;

distMat.convertTo(distMap, CV\_8U, 255/maxValue);

imshow("distMap", distMap); //while (1) { if (waitKey(0) == '\r') break; }

distMat是CV\_32FC1类型的Mat，maxValue是其最大值

## OpenCV的imread()：

Mat image = imread("C:/users/myth/Desktop/test.jpg");/\*默认参数flag将图像转化为三通道BGR彩色图像\*/

Mat imGray;

cvtColor(image, imGray, CV\_RGB2GRAY); /\*使用函数CV\_RGB2GRAY将彩色图像转换为灰度图\*/

Mat img = imread("C:/users/myth/Desktop/test.jpg ", 0); //参数flag=0表示载入灰度图；如果原始图片为RGB，则读取后自动转化为灰度图

if (!img.data) { cout << "不能找到图片！" << endl; return; }

imshow("原灰度图", img); //while (1) { if (waitKey(0) == '\r') break; }

## OpenCV中Mat::convertTo()改变矩阵元素数据类型

Mat distMap;

distMat.convertTo(distMap, CV\_8U, 255/maxValue);

distMat是CV\_32FC1类型的矩阵，变为CV\_8UC1类型的矩阵

显示深度图时：

Mat mat(height,width,CV\_16UC1);

Mat img(height, width, CV\_8UC1);

mat.convertTo(img, CV\_8UC1, 255.0 / 4500);

OpenCV中cvtColor()函数:

Mat image = imread("C:/users/myth/Desktop/test.jpg");/\*默认参数flag将图像转化为三通道BGR彩色图像\*/

Mat imGray;

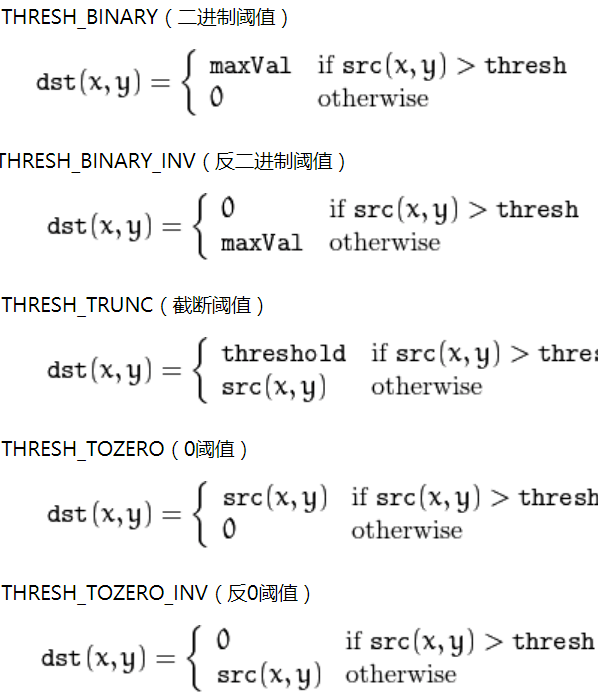
cvtColor(image, imGray, CV\_RGB2GRAY); /\*使用函数CV\_RGB2GRAY将彩色图像转换为灰度图\*/



## OpenCV灰度图转换为二值图：

double threshold ( InputArray src, OutputArray dst, double thresh, double maxval, int type ); 项目中自己设置maxval默认的值为255，thresh由用户调节

THRESH\_OTSU：采用大津法OTSU对灰度图二值化



## OpenCV形态学运算：

getStructuringElement函数的第一个参数表示内核的形状，我们可以选择如下三种形状之一：矩形: MORPH\_RECT；交叉形: MORPH\_CROSS；椭圆形: MORPH\_ELLIPSE

而getStructuringElement函数的第二和第三个参数分别是内核的尺寸以及锚点的位置

## OpenCV中边缘检测：

**Sobel()和Scharr()**

使用方法：

1. 尝试3×3的Scharr()：采用先x、再y、后合并的方式；
2. 尝试5×5的Sobel()：采用先x、再y、后合并的方式；
3. 尝试7×7的Sobel()：采用先x、再y、后合并的方式；
4. 比较以上3种效果，选最好的

/\*Sobel需要保证 ksize>order\*/

ksize取值1、3、5、7

order取值1、2、3

具体分析：

1. Sobel(gray,mat1, -1 / CV\_16S,1,1)的效果与先x方向滤波得到一张图、再y方向滤波得到一张图、最后合并得到最终的结果相比，不如后者。大多数时候使用Sobel()的方式为后者，如：

Sobel(gray, mat5x, -1 / CV\_16S, 1, 0); //x方向

Sobel(gray, mat5y, -1 / CV\_16S, 0, 1); //y方向

addWeighted(mat5x, 0.5, mat5y, 0.5, 0, mat5); //合并

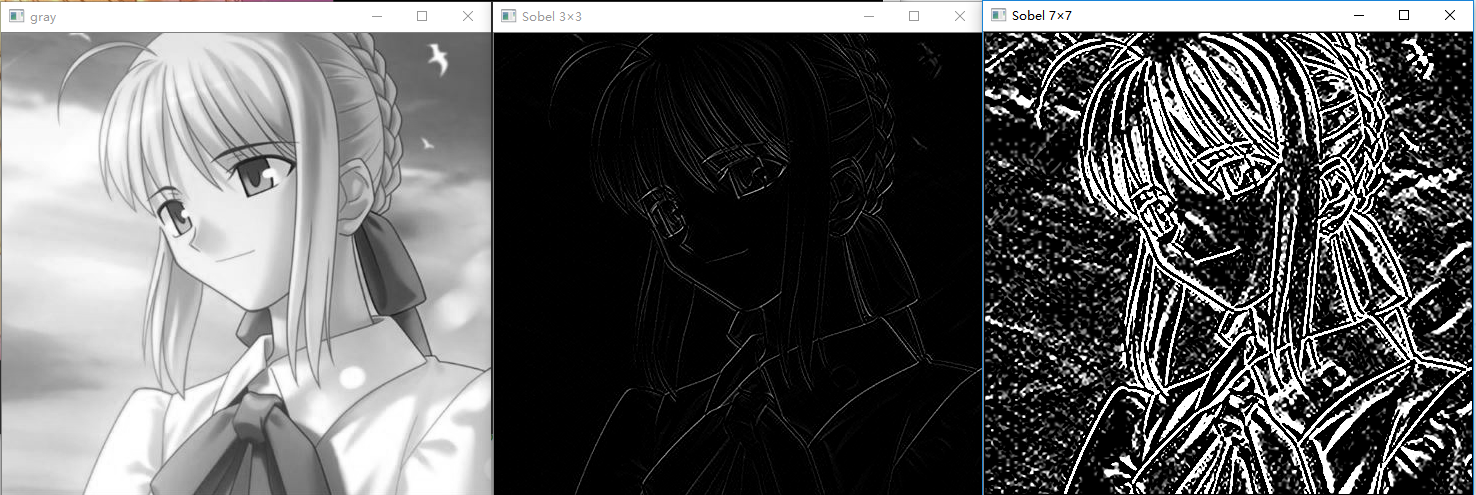
下图最左边是mat5x图，中间是mat5y图，最右边是x方向Sobel()和y方向Sobel()合并的结果：



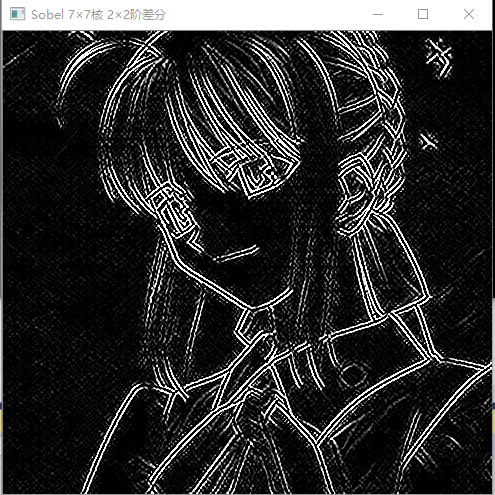
1. Sobel(gray,mat1, -1 / CV\_16S,1,1)和Sobel(gray,mat1, -1 / CV\_16S,7,7)的效果Sobel(gray,mat1, -1 / CV\_16S,1,1); //默认内核大小3×3

Sobel(gray, mat2, -1 / CV\_16S, 1, 1,7); //内核大小7×7

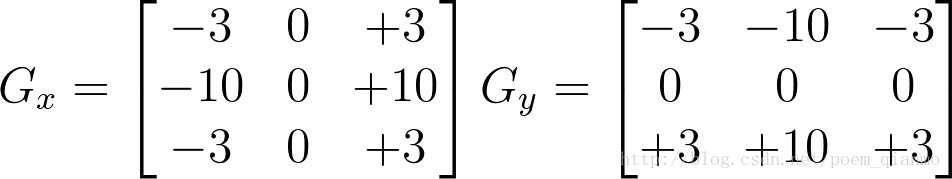
下图左最边原始灰度图，中间3×3的Sobel核，最右边7×7的Sobel核



1. 内核并非越大越好，内核大可以检测出更多边缘，但也易误检测出现许多噪点。改变差分阶数使用Sobel(gray, mat4, -1 / CV\_16S, 2, 2, 7)的结果为：



1. 当Sobel内核大小为3时，误差较大，这时会用Scharr()函数代替，结果更精确，该函数仅作用于大小为3的内核，如下：



1. 使用Scharr()滤波的方法为：

Scharr(gray,mat3x, -1 / CV\_16S,1,0); //x方向

Scharr(gray, mat3y, -1 / CV\_16S, 0, 1); //y方向

addWeighted(mat3x, 0.5, mat3y, 0.5, 0, mat3); //合并

下图左最边是mat3x图，中间是mat3y图，最右边是x方向Scharr()和y方向Scharr()合并的结果



**Canny()**

简单的Canny()用法，直接调用Canny处理RGB图

高阶的canny用法，转成灰度图，降噪，用canny，最后将得到的边缘作为掩码，拷贝原图到效果图上，得到彩色的边缘图

Mat img = imread("C:/Users/myth/Pictures/forTest/edgeDetect.jpg");

if (!img.data) { cout << "can't read" << endl; return; }

imshow("origin", img);

Mat mat1;

Canny(img, mat1, 150, 100, 3); //阈值1和阈值2两者的小者用于边缘连接，而大者用来控制强边缘的初始段，推荐的高低阈值比在2:1到3 : 1之间

imshow("Canny", mat1); while (1) { if (waitKey(0) == '\r') break; }

Mat gray, smooth,edge,result(img.size(),img.type(),Scalar(0));

cvtColor(img,gray,CV\_RGB2GRAY);

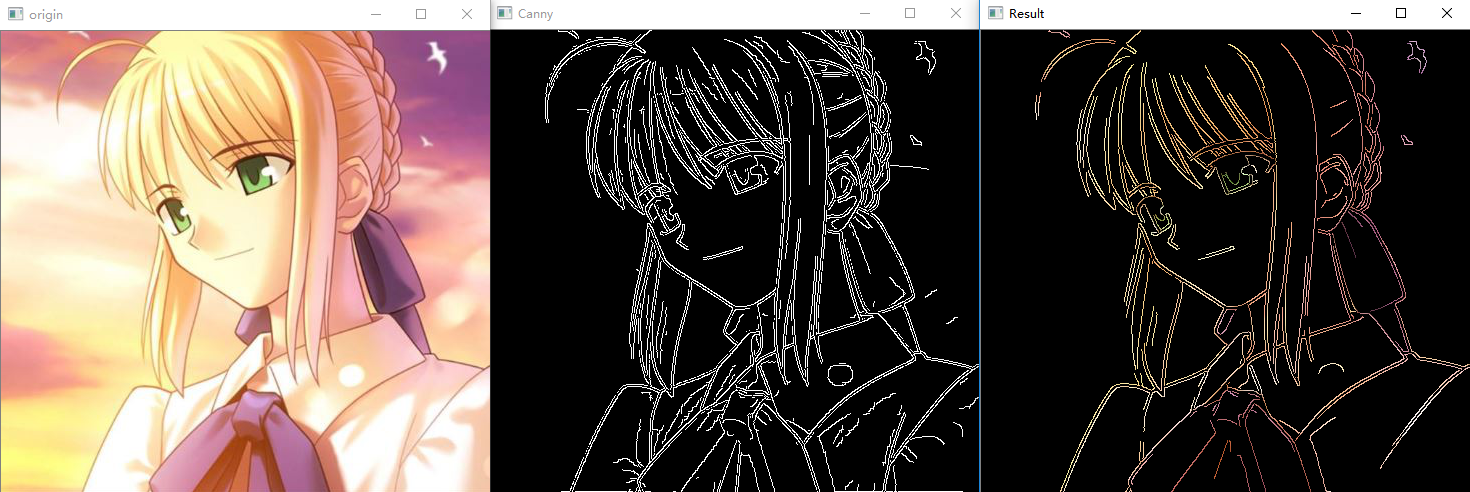
GaussianBlur(gray,gray,Size(3,3),1.5);

Canny(gray,edge,150,100);

img.copyTo(result,edge);

imshow("Result", result); while (1) { if (waitKey(0) == '\r') break; }

下图中间是简单的Canny()用法，最右边是高阶用法：



## OpenCV创建滑动条：

void on\_TrackbarNumChange(int, void \*);//回调函数

void on\_ElementSizeChange(int, void \*);//回调函数

createTrackbar("腐蚀/膨胀", "g\_dstImage", &g\_nTrackbarNumer, 1, on\_TrackbarNumChange);

createTrackbar("内核尺寸", "g\_dstImage", &g\_nStructElementSize, 21, on\_ElementSizeChange);

/\*轨迹条名称为“内核尺寸”， 跟踪条所依附的窗口的名字为"g\_dstImage"，值存入g\_nTrackbarNumer，轨迹条最大值为21（最小值总是0），每次拖动轨迹条后触发on\_ElementSizeChange()函数\*/

回调函数必须是(int,void\*)参数

## OpenCV直方图calcHist()：

calcHist()输出的直方图矩阵类型是CV\_32F，其他函数如calcBackProject()中表示直方图的参数Mat hist必须是CV\_32FC1类型

一维直方图，calcHist()输出一维矩阵，如i号元素表示灰度值落在i号区间的像素点个数

二维直方图，calcHist()输出二维矩阵，如

dstHist中i行j列的元素存储的是，同时落在第一维度（色度维度）的i号bin内、第二维度（饱和度维度）的j号bin内的像素点的数目

dstHist行数是第一维度的划分数，列数是第二维度的划分数，CV\_32FC1类型

要求R、G、B分量各自的直方图：

calcHist(images, 1, channel\_r, Mat(), dstHist\_r, dim, histSize\_r, histRange\_r, true, false);

calcHist(images, 1, channel\_g, Mat(), dstHist\_g, dim, histSize\_g, histRange\_g, true, false);

calcHist(images, 1, channel\_b, Mat(), dstHist\_b, dim, histSize\_b, histRange\_b, true, false);

#pragma once 防止多次包含一个头文件

## OpenCV掩模：

掩模的创建：必须是CV\_8UC1类型，0元素和非0元素

掩模用于copyTo()

掩模用于set()

string path = "C:\\Users\\myth\\Pictures\\forTest\\yanlingji.jpeg";

**const** Mat src = imread(path);

**if** (src.empty()) { cout << "nor read" << path << endl; **return**; }

assert(src.type() == CV\_8UC3);

imshow("src", src); cv::waitKey(0);

cv::Rect rect(200, 100, 300, 200);

Mat roi = src(rect);  //roi的与原图像指向相同的像素，修改roi的元素，则原图像也改变

imshow("roi", roi); cv::waitKey(0);

Mat mask(src.size(), CV\_8UC1, cv::Scalar(0));

mask(rect).setTo(255);

imshow("mask", mask); cv::waitKey(0);

Mat dst;

//作为copyTo第二个参数的mask必须是CV\_8UC1,size相同，非0元素位置被拷贝，其余位置不变，这里为0

src.copyTo(dst,mask);

assert(dst.type() == CV\_8UC3);

imshow("dst", dst); cv::waitKey(0);

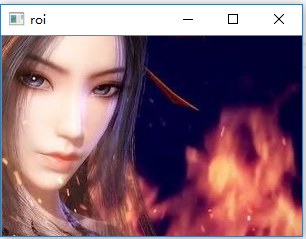
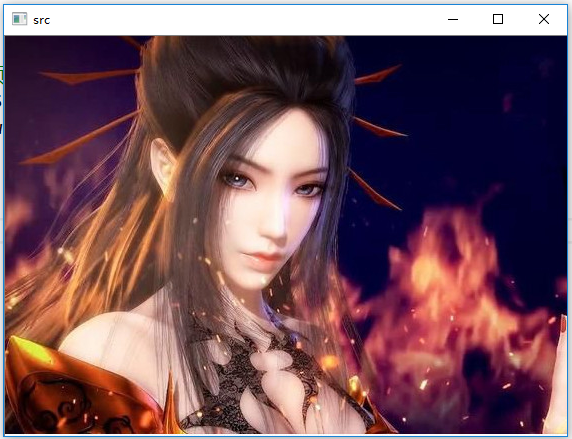
Mat temp;

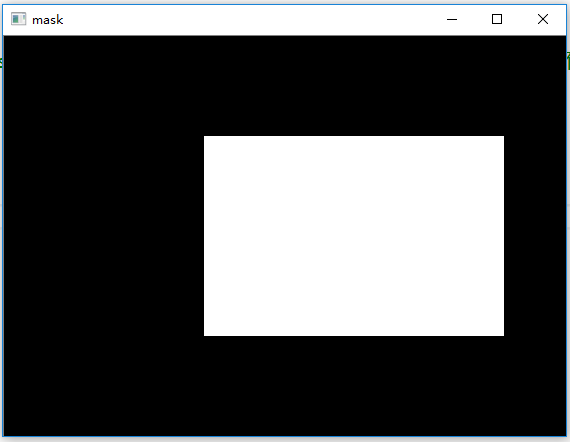
src.copyTo(temp);

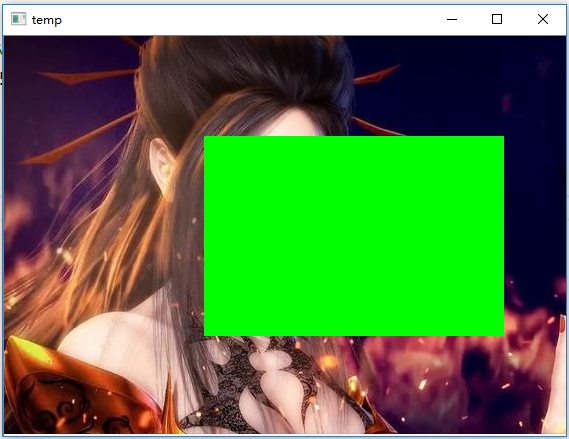
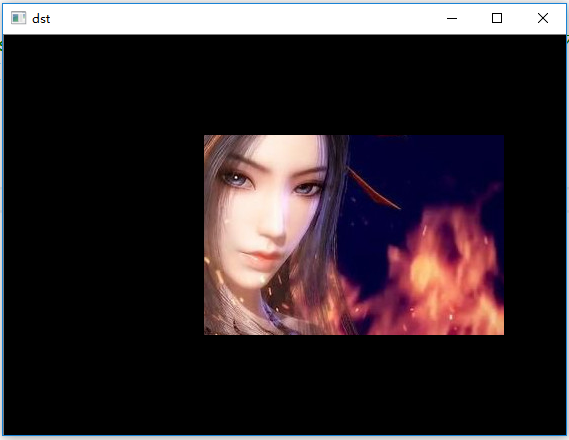
//作为set第二个参数的mask必须是CV\_8UC1,size相同，非0元素位置被设置，其余位置不变

temp.setTo(cv::Scalar(0,255,0), mask);

imshow("temp", temp); cv::waitKey(0);







## OpenCV中vector转Mat：

vector转为Mat：size为vector::size()行1列

如果vector<int>，则Mat::type()为CV\_32SC1

如果是vetcor<Point>，则Mat::type()为CV\_32SC2

转换后修改Mat的元素，则vector也会随之改变

vector<**float**>vec;

**for** (**int** i = 0; i< 10; i++)vec.push\_back((**float**)i);

Mat img = Mat(vec);

cout << "size=" << img.rows << endl; //10

cout << "size=" << img.cols << endl; //1

cout << "type=" << img.type() << endl; //CV\_32FC1

cout << Point(10, 90) << endl; //[10, 90]

vector<Point> pts{ Point(2,3),Point(10,20),Point(7,8) };

Mat m(pts);

cout << m.type() << endl; //CV\_32SC2

assert(m.size() == Size(1, 3));

cout << m << endl;

/\*

[2, 3;

10, 20;

7, 8]

\*/

m.setTo(cv::Scalar(2, 3));

cout << m << endl;

/\*

[2, 3;

2, 3;

2, 3]

\*/

**for** (auto i : pts) cout << i << "  "; cout << endl;

//[2, 3]  [2, 3]  [2, 3]

Mat cvt(3, 2, CV\_32SC1, m.data);

assert(cvt.size() == Size(2, 3));

assert(cvt.type() == CV\_32SC1);

//将输入的vector<cv::Point> pointset转换为一个CV\_64FC1的len行2列的Mat，Mat的每行为一个点

vector<cv::Point> pointset;

**int** len = (**int**)pointset.size();

Mat dst;

Mat ma(pointset);

assert(ma.size() == cv::Size(1, len) && ma.type() == CV\_32SC2);

Mat mb(len, 2, CV\_32SC1, ma.data);

mb.convertTo(pts\_mat, CV\_64FC1);

assert(dst.size() == cv::Size(2, len) && dst.type() == CV\_64FC1);

## OpenCV其它：

OpenCV中绘制轮廓函数void drawContours()的参数thickness设为-1就是填充轮廓

OpenCV3维点坐标：typedef Point3\_<int> Point3i

高斯滤波void GaussianBlur(InputArray src, OutputArray dst, Size ksize, double sigmaX, double sigmaY=0, int borderType=BORDER\_DEFAULT ) ;

GaussianBlur(mat, mat3, Size(5, 5),1.5);只设置sigmaX，则sigmaY取默认值0，当sigmaY≤0时，sigmaY=sigmaX

Size(宽，高)

标准差不变，仅增大核尺寸，当ksize增大到一定程度后，则图像不会更模糊（也即更平滑），因为高斯函数的能量集中在[-3σ,3σ]内。如GaussianBlur(mat, mat3, Size(19, 19),1.5)跟GaussianBlur(mat, mat3, Size(5, 5),1.5)差异不大，要想更平滑，联合调节尺寸和σ，如GaussianBlur(mat, mat3, Size(19, 19),3)

VideoCapture::get()函数可以获取视频总帧数、帧率、帧宽、帧高等属性

for(auto i:v)：遍历容器元素：[c++11](https://www.baidu.com/s?wd=c%2B%2B11&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1d9nymvrAc4P1bdnHN-m16z0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnHmsPWb3PH6vnWn1PHnkPHbsPs)的新特性，v是vector类型，i用来在遍历过程中获得容器里的每一个元素

例如：vector<int> v={1,2,3,4};

for(auto i:v)

cout<<i; //结果就是1234

或者使用迭代器遍历容器：

for (vector<int>::iterator i = v.begin(); i != v.end(); i++)

{cout<<\*i<<endl;}

vector中：

iterator begin()：返回vector容器中起始元素的迭代器

iterator end()： 指向最后一个元素的下一个位置

迭代器是指向vector中元素的指针

vector<char>::iterator iter1;

vector<char>::iterator iter2;

iter1 = v1.begin();

cout << \*iter1 << endl;

iter2 = v1.end()-1; /\*注意v1.end()指向的是最后一个元素的下一个位置，所以访问最后一个元素的正确操作为：v1.end() - 1;\*/

cout << \*iter2 << endl;

STL算法中for\_each是个模板函数，原型：

template<class InputIterator, class Function>

Function for\_each(InputIterator first, InputIterator last, Function fn)

{

while (first!=last) {

fn (\*first);

++first;

}

return fn; //

}

示例：

void printFun(int i)

{

cout << i << " ";

}

auto print = [](int i){ cout << i << " "; } //lambda 表达式

vector<int> v{1,2,3,4,5};;//容器的初始化

for\_each(v.begin(), v.end(), printFun);//输出1 2 3 4 5

for\_each(v.begin(), v.end(), print);//输出1 2 3 4 5

vector中存储的是指针时，容器清空：

vector<Particle\*> swarm;

for\_each(swarm.begin(), swarm.end(), [](Particle\* p){delete p;}); //for\_each模板函数，lambda表达式

swarm.clear();

**Lua：**

**C++调用lua中的函数int add(int，int)：**

lua\_getglobal(L, "add"); //把lua脚本中变量add的值压入堆栈

int a = 100;

int b = 200;

lua\_pushnumber(L, a);

lua\_pushnumber(L, b);

if (lua\_pcall(L, 2, 1, 0) != 0) {

std::cerr << "call func failed, got error: " << lua\_tostring(L, -1) << std::endl;

}

int sum = lua\_tonumber(L, -1); //函数计算结果置于栈顶

lua\_pop(L, 1);

lua\_pcall(lua\_State \*L,int nargs,int nresults,int errfunc)

nargs 参数个数

nresults 返回值个数

errFunc 错误处理函数，0表示无，表示错误处理函数在栈中的索引

lua\_call

调用一个函数void lua\_call(lua\_State\*L,int nargs,int nresults);//nargs是参数的个数 nresults是返回值个数

调用一个函数请遵循以下协议

首先,要调用的函数应该被压入堆栈;接着把需要传递给这个函数的参数按正序压栈;这是指第一个参数首先压栈.

最后调用一下 lua\_call;

当函数调用完毕后,所有的参数以及函数本身都会出栈。而函数的返回值这时则被压入堆栈。

返回值的个数将调整为 nresults 个，除非 nresults 被设置成 LUA\_MULTRET. 在这种情况下，所有的返回值都被压入堆栈中。函数第一个返回值首先压栈，因此在调用结束后,最后一个返回值将被放在栈顶

lua\_pcall 表示 protected mode。简单理解，lua\_call 没有返回值，我们不能知道函数调用的状态。而 lua\_pcall 有返回值，并且支持设定错误处理函数

**读lua脚本中的变量于C++中：**

放于栈顶，转为C++中变量

lua\_getglobal(L, "version"); //lua脚本中数据入栈

const char \*version = lua\_tostring(L, -1); //栈中数据转为C++变量

**将C++中的变量设置到lua虚拟机中**

lua\_pushstring(L, "hello world"); //C++中数据入栈

lua\_setglobal(L, "greet"); //栈顶数据转为lua虚拟机变量，并弹栈。

lua\_setglobal(L, "greet") 的功能是将栈顶数据命名为greet，并弹出。这样，在 lua 虚拟机就新建了名字为 greet，值为字符串 "hello world" 的全局变量了。

**Lua索引：**

由于我们一般都是对栈顶做操作，使用负数的索引会更方便。

lua\_tostring(L, -1) 就可以把索引号为 -1 的数据（即栈顶的数据）转换成string类型输出。

**Lua中的table：**

增加键值对：

void lua\_settable (lua\_State \*L, int index);

先作一个等价于 t[栈顶之下的值] = 栈顶值 的操作，这里t为指定的table；

然后把键和值都从堆栈中弹出。

lua\_newtable(L); // t = {} 创建没有名字的table，这里用t代替

lua\_pushstring(L, "key1");

lua\_pushstring(L, "val1");

lua\_settable(L, -3); // t["key1"] = "val1"

或

lua\_newtable(L); //创建无名table入栈

lua\_pushstring(L, "val3");

lua\_setfield(L, -2, "key3"); // t["key3"] = "val3"，弹出栈顶元素

lua\_setglobal(L, "my\_table"); // 给这个匿名的table一个名字，弹栈

访问key对应的value：

lua\_getglobal(L, "my\_table"); //table入栈

void lua\_gettable (lua\_State \*L, int index);

作用：table[栈顶]替换栈顶（即弹出之前栈顶，再value入栈），index指定table

lua\_pushnumber(L, 3); //将数值3放于栈顶。如果放置字符串，则调lua\_pushstring

// table一开始是在栈顶，即-1处的，但上面的语句压入了一个值，table变为-2。

lua\_gettable(L, -2)用table[3]替换原栈顶，table仍为-2

简单方法，获取table的某个key对应的value：

lua\_getglobal(L, "my\_table");

lua\_getfield(L, -1, "key3"); //将my\_table[“key3”]置于栈顶

HlsContext是个table

**lua中table的key-value，value可能是函数**

经常看到这样的代码:

tb = {}

function tb.print()

print(“hello”) //print为lua的内建函数

end

其实刚才那段代码也就是

tb.print = function()

end

在tb这个table对象里面有一个key，名为print，对应的value是一个function

.和:调用函数参数的传递:

tb.print和tb:print的含义略微不同，:调用函数会多传递进去一个self进去,差不多相当于OOP里面的成员函数调用

**lua\_pushvalue(L, -4)**

把在栈中位置为-4的元素copy之后插入于栈顶中！！！

局部全局变量、函数：

Lua 变量有三种类型：全局变量、局部变量、表中的域。

Lua 中的变量全是全局变量，那怕是语句块或是函数里，除非用 local 显式声明为局部变量。

局部变量的作用域为从声明位置开始到所在语句块结束。

变量的默认值均为 nil。

**元表和\_\_index：**

（1）

首先看看如下情况：

local tab = {

name="haha"

}

print(tab.date)

print访问了tab中并不存在的索引，因此结果必定是nil。

那么，会不会有例外呢？答案是有。实际上，当遇到上述的访问了table中并不存在的索引变量时，解释器还会坐另外一步工作：查找\_\_index元方法

（2）

我们把程序稍微修改一下：

local tab = {

name = "haha"

}

local mt = {

\_\_index = function(t, key) --定义了访问空索引时如何操作

print("no such an index!")

end

}

setmetatable(tab, mt)

print(tab.date)

以上代码的功能是：如果试图访问tab.date这个不存在的变量，由于\_\_index的存在，将会执行那句print

（3）

local mt = {

\_\_index = function(t, key)

return "empty"

end

}

此时，执行print(tab.date)语句的结果便是”empty”，因为我们已经规定了它的行为：对空的索引，返回值一律为”empty”。

（4）

local tab\_old = {

name = "haha",

date = "2017.1.7"

}

local tab = {

name = "haha"

}

local mt = {

\_\_index = tab\_old

}

setmetatable(tab, mt)

print(tab.date)

当程序试图访问tab.date时，由于并没有这个索引，就去寻找与tab相关的index元方法了，而index就关联到了tab\_old这个表格，因此程序便会在tab\_old中寻找date索引。

**table：**

local o={}

o.age=21

o.store=99

print(o.age)

print(o.store)

**for循环：**

for var=exp1,exp2,exp3 do

<执行体>

end

var 从 exp1 变化到 exp2（包括exp2），每次变化以 exp3 为步长递增 var，并执行一次 "执行体"。exp3 是可选的，如果不指定，默认为1。

运行：

array = {"Lua", "Tutorial"} //lua数组下标从1开始

for i= 0, 2 do

print(array[i])

end

print(#array) //返回数组元素个数

输出：

nil

Lua

Tutorial

2

运行

array = {"Lua", "Tutorial"}

print("ele num=" .. #array)

for i= 1, 2 do

print("i= " .. i .. ", array[i]=" .. array[i]) //..用于字符串拼接

end

for k, v in pairs(array) do

print(k, v)

end

输出

ele num=2

i= 1, array[i]=Lua

i= 2, array[i]=Tutorial

1 Lua

2 Tutorial

**if语句：**

boolean 类型只有两个可选值：true（真） 和 false（假），Lua 把 false 和 nil 看作是 false，其他的都为 true，数字 0 也是 true

video\_idx=false //于video\_idx=nil效果相同

if video\_idx then

print(true)

else

print(false)

end

**lua\_rawgeti和lua\_rawseti：**

从table中得到相应下标的元素

lua\_rawgeti(L, index, key)

index表示table在栈中的位置

key表示元素在table中的位置

把值放到table相应下标位置

lua\_rawseti(L, index, key)

index表示table在栈中的位置

key表示元素在table中的位置

**lua\_gettop和lua\_settopd等**

下列函数来完成通常的堆栈维护工作：

int  lua\_gettop (lua\_State \*L);

void lua\_settop (lua\_State \*L, int index);

void lua\_pushvalue (lua\_State \*L, int index);

void lua\_remove (lua\_State \*L, int index);

void lua\_insert (lua\_State \*L, int index);

void lua\_replace (lua\_State \*L, int index);

函数lua\_gettop返回堆栈中的元素个数，它也是栈顶元素的索引。注意一个负数索引-x对应于正数索引gettop-x+1。lua\_settop设置栈顶（也就是堆栈中的元素个数）为一个指定的值。如果开始的栈顶高于新的栈顶，顶部的值被丢弃。否则，为了得到指定的大小这个函数压入相应个数的空值（nil）到栈上。特别的，lua\_settop(L,0)清空堆栈。你也可以用负数索引作为调用lua\_settop的参数；那将会设置栈顶到指定的索引。利用这种技巧，API提供了下面这个宏，它从堆栈中弹出n个元素：

#define lua\_pop(L,n)  lua\_settop(L, -(n)-1)

函数lua\_pushvalue压入堆栈上指定索引的一个抟贝到栈顶；lua\_remove移除指定索引位置的元素，并将其上面所有的元素下移来填补这个位置的空白；lua\_insert移动栈顶元素到指定索引的位置，并将这个索引位置上面的元素全部上移至栈顶被移动留下的空隔；最后，lua\_replace从栈顶弹出元素值并将其设置到指定索引位置，没有任何移动操作。注意到下面的操作对堆栈没有任何影响：

lua\_settop(L, -1);   /\* set top to its current value \*/

lua\_insert(L, -1);   /\* move top element to the top \*/

# OpenGL：

OpenGL规范描述了绘制2D和3D图形的抽象[API](https://zh.wikipedia.org/wiki/API)。尽管这些API可以完全通过软件实现，但它是为大部分或者全部使用[硬件加速](https://zh.wikipedia.org/wiki/%E7%A1%AC%E4%BB%B6%E5%8A%A0%E9%80%9F)而设计的

OpenGL库实际的开发者通常是显卡厂商。你所购买的每个显卡支持的是特定的OpenGL版本

GLEW 是Opengl扩展库，使用它可以很方便的调用OpenGL较新的特性。GLEW能自动识别你的平台所支持的全部OpenGL高级扩展涵数

GLFW是继GLUT，FreeGLUT之后，当前最新的用来创建OpenGL上下文，以及操作窗口的第三方库。OpenGL没有窗口管理的功能



实验室台式机：

------------------GLEW库测试--------------

Status: Using GLEW 1.12.0

------------------OpenGL测试--------------

OpenGL实现厂商的名字：ATI Technologies Inc.

渲染器标识符：AMD Radeon R5 340

OpenGL实现的版本号：4.5.13447 Compatibility Profile Context 20.19.0.32837

OGLU工具库版本：1.2.2.0 Microsoft Corporation

windows系统以库的形式提供opengl功能，opengl32.lib、opengl32.dll、GL.h，使用时需要#pragma comment(lib, "opengl32.lib")

这样就可以使用GL.h中定义的

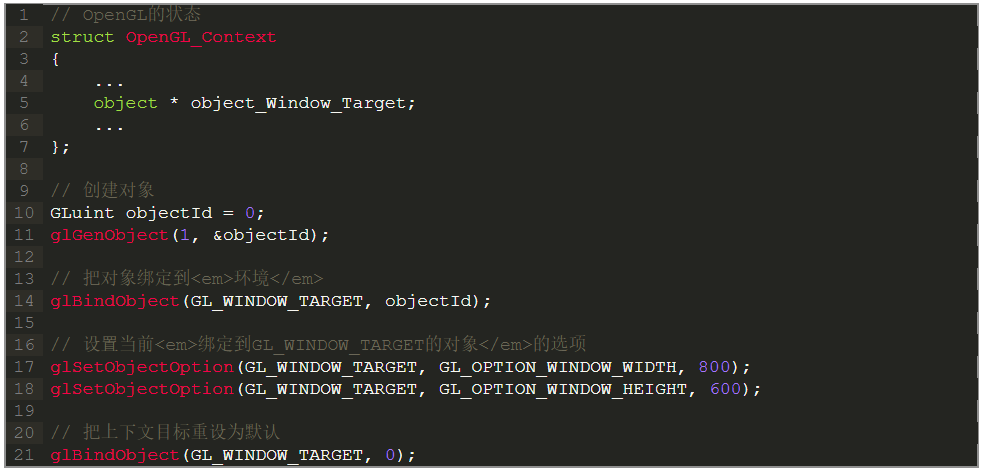
glClearColor(0.2f,0.3f,0.3f,1.0);

glClear(GL\_COLOR\_BUFFER\_BIT);等函数

GLAD库对win系统opengl进行了封装，使用该库后就不再需要手动链接opengl32.lib了

OpenGL本身是个大状态机。OpenGL的状态通常被称为OpenGL上下文（context）。当使用OpenGL的时候，我们会遇到多种状态改变函数，这些函数改变环境；也会遇到多个状态使用函数，这些函数基于当前OpenGL的状态执行一些操作。

注意，当时用OpenGL的时候，建议使用OpenGL定义的自有类型。在写float的时候在前面加GL；对于int、char、bool也同样处理。各种操作系统可能对于各自的类型有不同的内存布局，而OpenGL的GL自有类型的内存布局是跨平台的。所以使用OpenGL的自有类型可以保证你的应用可以跨平台



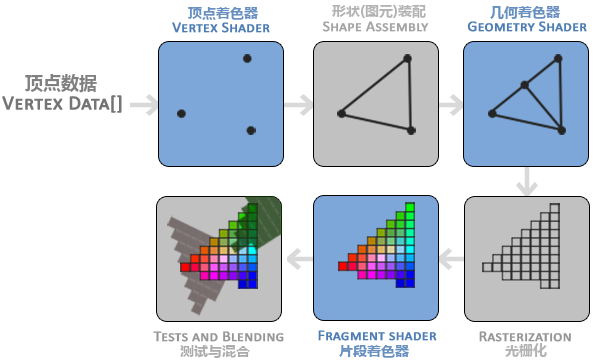
我们先创建一个对象，把它的引用储存为一个id（真实的对象数据在场景后台储存）。然后我们把对象绑定到环境（contex）的目标（target）位置（例子窗口对象的目标位置定义为GL\_WINDOW\_TARGET）。下一步我们设置窗口选项，最后把窗口目标的当前对象id设置为0解绑对象。我们设置的选项被储存在对象的引用objectId上面，把对象解绑

**双缓冲（Double buffer）**当一个应用以单缓冲方式绘制的时候，图像会产生闪烁的问题。这是因为最后的图像输出不是被立即绘制出来的，而是一个像素一个像素绘制出来的，通常是以从左到右从上到下这样的方式。由于这些图像不是立即呈现在用户面前，而是一步一步地生成结果，这就产生很多不真实感。为了规避这些问题，窗口应用使用双缓冲的方式进行渲染。**前缓冲**包含最终的输出图像，它被显示在屏幕上，与此同时，所有的渲染命令绘制**后缓冲**。所有的渲染命令执行结束，我们就把后缓冲交换到前缓冲，这样图像就会立即显示到用户面前了，前面提到的不真实感就这样被解决了。

图形输送管道接收一组3D坐标，然后把它们转变为你屏幕上的有色2D像素。

每个环节能简单地并行执行，在GPU上为每一个（输送管道的）阶段运行各自的小程序（这些小程序叫做着色器），加速处理数据。

在下面，你会看到一个图形输送管道的每个阶段的抽象表达。要注意蓝色部分代表的是我们可以自定义的着色器。



一旦你的顶点坐标已经在顶点着色器中处理过，它们就应该是标准化设备坐标了，标准化设备坐标是一个x、y和z值在-1.0到1.0的一小段空间

标准化设备坐标接着会变换为屏幕空间坐标（screen-space coordinates），这是使用你通过glViewport函数提供的数据，进行视口变换（viewport transform）完成的。最后的屏幕空间坐标被变换为像素输入到像素着色器

* 顶点数组对象：Vertex Array Object，VAO
* 顶点缓冲对象：Vertex Buffer Object，VBO
* 索引缓冲对象：Element Buffer Object，EBO或Index Buffer Object，IBO

shader是用GLSL语言编写的

in、out关键字，解决显存中shader之间的数据传输

以下代码如果片段着色器最终输出颜色为绿色vec4(0.0f, 1.0f, 0.0f, 1.0f);

const char\* vertex\_src =

"#version 400 core\n"

"layout(location=0) in vec3 pos;\n"

"out vec4 vertex\_color;\n"

"void main()\n"

"{\n"

"vertex\_color=vec4(0.0f, 1.0f, 0.0f, 1.0f);\n"

"gl\_Position = vec4(pos.x+0.3,pos.y,pos.z,1.0f);\n"

"}\n";

//顶点着色器输出vertex\_color

//片段着色器以之作为输入

//in、out解决gpu内部数据传输的问题，shader之间

const char\* frag\_src =

"#version 400 core\n"

"in vec4 vertex\_color;\n"

"out vec4 frag\_color;\n"

"void main()\n"

"{\n"

"frag\_color = vertex\_color;\n"

"}\n";

uniform关键字从cpu向gpu传输数据：

以下代码可以实现颜色动态变化

const char\* vertex\_src =

"#version 400 core\n"

"layout(location=0) in vec3 pos;\n"

"void main()\n"

"{\n"

"gl\_Position = vec4(pos.x,pos.y,pos.z,1.0f);\n"

"}\n";

const char\* frag\_src =

"#version 400 core\n"

"uniform vec4 out\_color;\n"

"out vec4 frag\_color;\n"

"void main()\n"

"{\n"

"frag\_color = out\_color;\n"

"}\n";

int outcolor\_location = glGetUniformLocation(shaderprog, "out\_color");

while (!glfwWindowShouldClose(wind)) //事件循环

{

glClearColor(0.2f, 0.3f, 0.3f, 1.0);

glClear(GL\_COLOR\_BUFFER\_BIT); //窗口背景色

glUseProgram(shaderprog);

float timeval = glfwGetTime();

float gval = sin(timeval);

glUniform4f(outcolor\_location, 1.0f, gval, 0.0f, 1.0f);

//glDrawArrays(GL\_TRIANGLES, 0, 6);//由显存中0号顶点开始画6个

glDrawElements(GL\_TRIANGLES, 6, GL\_UNSIGNED\_INT, 0);

glfwSwapBuffers(wind); //双缓冲，将后缓冲区交换到前

glfwPollEvents(); //检测事件，执行相应处理函数（用户可设置回调函数）

}

glTexSubImage2D修改纹理：

// 功能：提供修改图像的功能,因为修改一个纹理比重新创建一个纹理开销小很多,对于一些视频捕捉程序可以先将视频图像存储在更大的初始图像中

// (图像大小要是2^n,opengl2.0后没有这个限制),创建一个渲染用的纹理,然后反复调用glTexSubImage2D函数修改纹理(修改的图像区域不用是2^n)

glTexSubImage2D (GLenum target, GLint level, GLint xoffset, GLint yoffset, GLsizei width, GLsizei height, GLenum format, GLenum type, **const** GLvoid \*pixels)

// target:必须是glCopyTexImage2D中对应的target可用值

// level:mipmap等级

// xoffset,yoffset是要修改的图像左上角偏移,width,height是要修改的图像宽高像素修改的范围在原图之外并不受影响

// format,type:表示图像的数据格式和类型

// pixels:子图像的纹理数据

必须在纹理上至少调用glTexImage2D()一次，然后才能使用glTexSubImage2D()。glTexImage2D()分配图像数据，重新创建纹理

**protobuf:**

protoc -I=C:\Users\myth\Desktop --cpp\_out=C:\Users\myth\E\garbage addressbook.proto

如果没有将protoc添加到path环境变量，则需要进入protoc.exe文件所在目录再输入上边语句。若将protoc添加到path环境变量，则在任意位置都可以直接protoc

第一个参数是.proto文件的目录，第二个参数是生成.pb.h和.pb.cc文件的存储目录，第三个参数是.proto文件路径

getter函数具有与字段名一模一样的名字，并且是小写的，而setter函数都是以set\_前缀开头

clear\_前缀的函数，用来将字段重置（un-set）到空状态（empty state）

has\_前缀的函数，对每一个单一的（required或optional的）字段来说，如果字段被置（set）了值，该函数会返回true

// required int32 age = 2;

inline bool Person::has\_age() const {

return (\_has\_bits\_[0] & 0x00000004u) != 0;

}

inline void Person::set\_has\_age() {

\_has\_bits\_[0] |= 0x00000004u;

}

inline void Person::clear\_has\_age() {

\_has\_bits\_[0] &= ~0x00000004u;

}

inline void Person::clear\_age() {

age\_ = 0;

clear\_has\_age();

}

inline ::google::protobuf::int32 Person::age() const {

return age\_;

}

inline void Person::set\_age(::google::protobuf::int32 value) {

set\_has\_age();

age\_ = value;

}

重复的字段也有一些特殊的函数——如果你看一下重复字段phone 的那些函数，就会发现你可以：   
（1）得到重复字段的\_size（换句话说，这个Person关联了多少个电话号码）。

（2）通过索引（index）来获取一个指定的电话号码。

（3）通过指定的索引（index）来更新一个已经存在的电话号码。

（4）向消息（message）中添加另一个电话号码，然后你可以编辑它（重复的标量类型有一个add\_前缀的函数，允许你传新值进去）

// repeated int32 phone = 4;

int phone\_size() const;

void clear\_phone();

static const int kPhoneFieldNumber = 4;

::google::protobuf::int32 phone(int index) const;

void set\_phone(int index, ::google::protobuf::int32 value);

void add\_phone(::google::protobuf::int32 value);

const ::google::protobuf::RepeatedField< ::google::protobuf::int32 >&

phone() const;

只有add\_phone添加一个字段，.phone\_size()才会增加1

string serializedStr;

stu.SerializeToString(&serializedStr);

cout << "serialization result: " << serializedStr << endl; ////序列化后的字符串内容是二进制内容，非可打印字符，预计输出乱码

cout << endl << "debugString: " << stu.DebugString() << endl; //string DebugString() const; //将消息内容以可读的方式输出

Student deserializedStudent;

if (!deserializedStudent.ParseFromString(serializedStr))

{

cerr << "Fail to parse student" << endl;

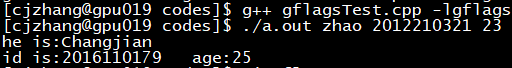
return;

}

.prototxt文件的注释符号为#

# gflag使用：

1. #include<iostream>
2. #include<gflags/gflags.h>
3. DEFINE\_string(name,"Changjian","your name");
4. DEFINE\_uint64(id,2016110179,"your id");
5. DEFINE\_int32(age,25,"your age");
6. **int** main(**int** argc,**char**\*\*argv)
7. {
8. gflags::ParseCommandLineFlags(&argc,&argv,**true**);
9. std::cout<<"he is:"<<FLAGS\_name<<std::endl;
10. std::cout<<"id is:"<<FLAGS\_id<<"   age:"<<FLAGS\_age<<std::endl;
11. **return** 0;
12. }



cv::Mat a(2, 3, CV\_16UC1);

cv::Mat b(cv::Size(2, 3), CV\_16UC1);

std::cout << a.rows << " " << a.cols << std::endl; //2,3

std::cout << b.rows << " " << b.cols << std::endl; //3,2

unsigned short\* data = new unsigned short[6]{11,22,33,44,55,66};

cv::Mat chan(2, 3, CV\_16UC1, data);

std::cout << chan << std::endl;

data[2] = 0;

std::cout << chan << std::endl;

std::cout << data << std::endl; //0000021C037698D0

std::cout <<(int\*)chan.data<< std:: endl; //0000021C037698D0

运行结果：

[11, 22, 33;

44, 55, 66]

[11, 22, 0;

44, 55, 66]

用cout打印一个指针指向的内存地址编号

ptr为指针。

如果指针指向非char类数据，直接用cout << ptr << endl;

如果指针指向char类数据，用cout << (int \*) ptr << endl;

# 贪心与动态规划算法：

**1、最少纸币凑面额**

　　假设有足够的1、5、10、20、50、100元面值的钞票，目标是凑出某个金额w，需要用到尽量少的钞票。

　　依据生活经验，显然可以采取这样的策略：能用100的就尽量用100的，否则尽量用50的……依次类推。在这种策略下，666=6×100+1×50+1×10+1×5+1×1，共使用了10张钞票。

　　这种策略称为“**贪心**”：假设我们面对的局面是“需要凑出w”，**贪心策略会尽快让w变得更小**。能让w少100就尽量让它少100，这样我们接下来面对的局面就是凑出w-100。长期的生活经验表明，贪心策略是正确的。

但是，如果我们换一组钞票的面值，贪心策略就也许不成立了。如果一个奇葩国家的钞票面额分别是1、5、11，那么我们在凑出15的时候，贪心策略会出错：  
　　15=1×11+4×1 （贪心策略使用了5张钞票）

15=3×5 （正确的策略，只用3张钞票）

为什么会这样呢？贪心策略错在了哪里？

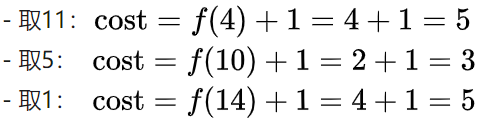
鼠目寸光。

刚刚已经说过，贪心策略的纲领是：“尽量使接下来面对的w更小”。这样，贪心策略在w=15的局面时，会优先使用11来把w降到4；但是在这个问题中，凑出4的代价是很高的，必须使用4×1。如果使用了5，w会降为10，虽然没有4那么小，但是凑出10只需要两张5元。  
　　在这里我们发现，贪心是一种**只考虑眼前情况**的策略。

　　那么，现在我们怎样才能避免鼠目寸光呢？

　　如果直接暴力枚举凑出w的方案，明显复杂度过高。太多种方法可以凑出w了，枚举它们的时间是不可承受的。我们现在来尝试找一下性质。

　　重新分析刚刚的例子。w=15时，我们如果取11，接下来就面对w=4的情况；如果取5，则接下来面对w=10的情况。我们发现这些问题都有相同的形式：“给定w，凑出w所用的最少钞票是多少张？”接下来，我们用f(n)来表示“凑出n所需的最少钞票数量”。



显而易见，cost值最低的是取5的方案。**我们通过上面三个式子，做出了正确的决策**

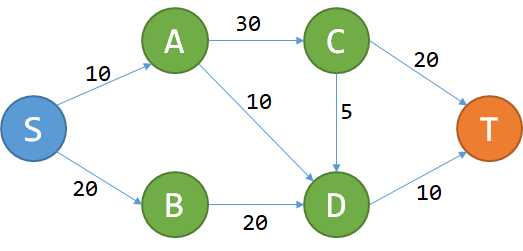
f(n)只与f(n-11)、f(n-5)、f(n-1)相关，即



　　这个式子是非常激动人心的。我们要求出f(n)，只需要求出几个更小的f值；既然如此，从小到大把所有的f(i)求出来不就好了？O(n)复杂度

**2、DP的典型应用：DAG最短路**

　　问题很简单：给定一个城市的地图，所有的道路都是单行道，而且不会构成环。每条道路都有过路费，问您从S点到T点花费的最少费用。

边上的数字表示过路费。

记从S到P的最少费用为f(P).



其中R为有路通到P的所有的点，Wr->p为R到P的过路费。

**3、DP的典型应用：最长上升子序列**

最长上升子序列（LIS）问题：给定长度为n的序列a，从a中抽取出一个子序列，这个子序列需要单调递增。问最长的上升子序列（LIS）的长度。  
　　e.g. 1,5,3,4,6,9,7,8的LIS为1,3,4,6,7,8，长度为6。

我们记 f(x)为以ax结尾的LIS长度，那么答案就是max{f(x)}

考虑比x小的每一个p：如果ax>ap，那么f(x)可以取f(p)+1.  
　　解释：我们把 ax 接在 ap 的后面，肯定能构造一个以  ax 结尾的上升子序列，长度比以  ap 结尾的LIS大1.那么，我们可以写出状态转移方程了：



至此解决问题。两层for循环，复杂度 O(n2)

int lis\_n2(int\* a, int n) //O(n^2)复杂度

{

vector<int> f(n, 1);

for (int x = 0; x < n; x++)

{

for (int p = 0; p < x; p++)

{

if(a[p]<a[x]) f[x] = std::max(f[x], f[p] + 1);

}

}

return \*std::max\_element(f.begin(),f.end());

}

# VS的项目生成



运行后在Output Directory下出现生成的lib、dll等

ProjectDir，即项目目录，即右键在文件资源管理器中打开文件夹显示的目录



同时项目配置了Post-Build Event





在项目名上右键，生成项目后，自动执行postbuild.bat，结果如下：



删除非空文件夹请用rmdir /s /q命令

# QT调用VC++ 动态库：

Qt库VC++编译器生成的，则工程可以静态加载动态库，即使用testdll.h头文件和testdll.lib引入库文件。

Qt库是MinGW版本，GCC编译器生成，与VC++编译器不是同个体系，则工程不可以使用testdll.h头文件和testdll.lib引入库文件，必须动态加载动态库

**int** test()

{

**typedef** IPlayerControl\*(\*FUNC)(**const** **char**\*, **const** **char**\*, **const** **char**\*, **void**\*);

    QLibrary mylib("E:\\VSC++Projects\\TestC++Syntax\\x64\\Debug\\WindowsControl.dll");

    mylib.load();

    FUNC createWindowsControl = (FUNC)mylib.resolve("createWindowsControl");

**const** **char**\* pluginPath = "";

**const** **char**\*libPath = pluginPath;

**const** **char**\* cfgPath = pluginPath;

**void**\* hWnd = nullptr;

    IPlayerControl\* p = createWindowsControl(pluginPath, libPath,

        cfgPath, hWnd);

    cout << "p is:" << (**void**\*)p << endl;  // p is:0000020DBBDD86E0

}

dumpbin.exe在

C:\Program Files (x86)\Microsoft Visual Studio\2017\Community\VC\Tools\MSVC\14.16.27023\bin\Hostx64\x64目录下

查看dll的导出函数：

dumpbin.exe /exports libass\_stream.dll

查看dll的依赖库：

dumpbin.exe /dependents E:\VSC++Projects\TestC++Syntax\x64\Debug\libass\_stream.dll



如果有Image has the following delay load dependencies，列出的为运行时动态加载的dll。   
如果有Image has the following dependencies，列出的为载入程序时加载的dll。

LoadLibraryEx函数跟LoadLibrary函数装载dll的机制不一样，前者在装载dll遇到与该dll依赖的其他dll时会自动装载，而后者不会。

LoadLibraryEx函数加载dll时，如果该dll或其依赖的某个dll不存在，则返回NULL，GetLastError()得到的错误码为126

const wchar\_t\* root = L"E:\\VSC++Projects\\TestC++Syntax\\x64\\Debug\\libass\_stream.dll";

HMODULE hmod;

hmod = LoadLibraryEx(root, NULL, LOAD\_WITH\_ALTERED\_SEARCH\_PATH);

if (hmod == NULL)

{

wcout << "Failed to load " << root << " error code is " << GetLastError() << endl;

}

else

{

wcout << "success to load " << root << endl;

FreeLibrary(hmod); //释放动态链接库

}

运行一个testplayer.exe，不能正常结束，资源管理器中也找不到记录，强制结束的方法：

在cmd中taskkill /im testplayer.exe /f

/f是强制杀死命令

tasklist命令，列出系统当前运行的所有进程



input.vcxproj里有工程的配置属性

decoder.apf解码的函数栈

decoder\_frame->raw\_frame\_decode->av\_decoder\_open + av\_decoder\_frame-->ff\_decoder\_frame-->ff\_decoder\_video或ff\_decoder\_audio-->video\_pack\_dec-->avcodec\_decode\_video2

# MFC：

CString Temp("8");//to\_string(8)

int ComNum = \_ttoi(Temp); //CString转int

　CString 和 LPCTSTR 可以说通用。 原因在于CString定义的自动类型转换（C++操作符重载）。

UpData(TRUE): 把控件内容装入控件变量。就是说：当用户对控件作了操作，控件内容发生了改变，可以通过该函数加TURE参数实现同步。

UpData(FALSE): 用控件变量的值更新控件。就是说：当程序中变量由于执行流程而发生了改变，需要在控件上体现出来，可以用该函数加FALSE参数实现同步。

* 给edit控件绑定类成员变量：

在控件上右键，添加变量，然后

（1）添加Cedie型变量



（2）或者添加int型变量



设置最小值、最大值后，如果输入的值不在此范围，则自动给出提示

* 删除控件

在给控件添加事件处理程序后，要删除控件，则右键、类向导



其中，对象ID为要删除的控件的ID、消息中加黑显示的为已给该控件添加的事件

# exp、ilk、pdb文件：

exp文件就是导出文件（export file）。我们假设我们生成两个dll（or just executables）。但是他们都需要调用一些对方中函数，问题出现了。当我们生成a.dll的时候我们需要b.lib；但是b.lib在对应的b.dll生成之前没有生成，而b.dll的生成又需要a.lib。正因如此，微软的解决办法是使用exp文件，或者叫导出文件。在生成两个dll之前，使用lib.exe(library mangager tool库管理工具)来创建一个.lib和.exp，即，DLL A 的a.lib 和a.exp，现在linker使用a.lib和DLL B 自己的东西去生成b.dll和b.lib。当你回来链接DLL A的时候你就有了b.lib。这里linker需要知道a.dll中需要导出处啥，这些信息都被缓存到了a.exp文件中。linker不需要def文件或者/EXPORT选项，它仅仅是加载a.exp中的信息。

增量链接，是一个链接选项Project Properties -> Configuration Properties -> Linker (General) -> Enable Incremental Linking。启用增量链接，则生成目录下会有一个.ilk文件

不选用增量链接时，每次修改或新增代码后进行链接时把原来的.exe删了，重新链接成一个新的.exe。而选用增量链接时，在对代码做小的改动时会把新成的函数或数据穿插到已有的.exe中（采用增量链接后链接器不会再将所有函数紧挨着放在一块儿了，而是在函数之间加上padding，这个时候函数要想添几句指令就有余地了），而不重新生成.exe，只有做了大量修改时才可能会重新编排，这样就可以提高链接的速度

一般VS的默认设置会把Debug版的Incremental Linking设置成Yes (/INCREMENTAL)，而把Release版的设置成No (/INCREMENTAL:NO)

借助源码调试第三方库：

.pdb，pdb全称program database，程序数据库，记录数据和调试信息，调试时使用

使用VS调试程序时，默认加载程序以及程序依赖的dll库产生的所有pdb文件，但经常VS找不到依赖库的pdb文件，于是提示 “无法查找或打开pdb文件”



如上图所示，由于pdb文件，又名“符号文件”，所以“已加载符号”，也就是成功加载了动态链接库对应的pdb文件

kernel32.dll、msvcp140d.dll等为C++运行时库，VS调试时无法加载到对应的PDB文件

发布的debug库典型如下：

使用时，

#include "E:/VSC++Projects/Format/include/time\_format.h"

#pragma comment(lib,"E:/VSC++Projects/Format/lib/FormatDLL.lib")

dll、pdb必须放置于exe找得到的位置

単步调试，如果在原位置（生成dll的工程中cpp文件的位置）未找到源文件time\_format.cpp，则VS自动弹窗让选择位置，如上图。