Making your C++ code robust

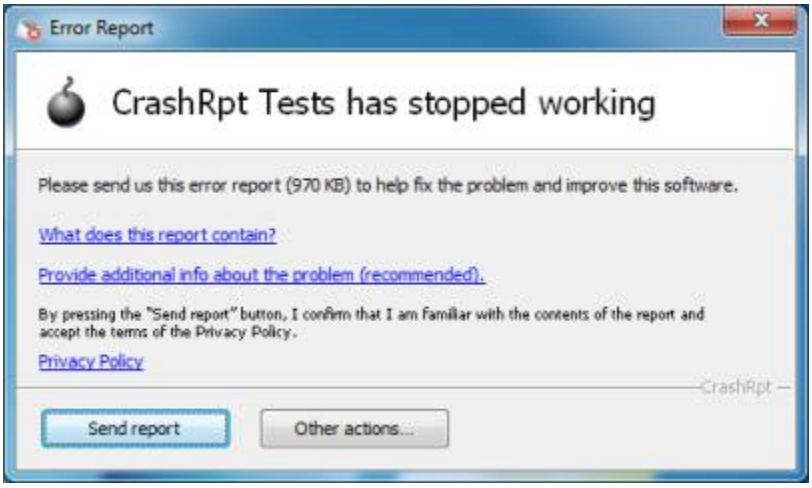
- **Introduction**

在实际的项目中,当项目的代码量不断增加的时候，你会发现越来越难管理和跟踪其各个组件，如其不善，很容易就引入 **BUG**。因此、我们应该掌握一些能让我们程序更加健壮的方法。

这篇文章提出了一些建议，能有引导我们写出更加强壮的代码，以避免产生灾难性的错误。即使、因为其复杂性和项目团队结构，你的程序目前不遵循任何编码规则，按照下面列出的简单的规则可以帮助您避免大多数的崩溃情况。

- **Background**

先来介绍下作者开发一些软件**(CrashRpt),**你可以 http://code.google.com/p/crashrpt/网站上下载源代码。**CrashRpt** 顾名思义软件崩溃记录软件（库），它能够自动提交你电脑上安装的软件错误记录。它通过以太网直接将这些错误记录发送给你，这样方便你跟踪软件问题，并及时修改，使得用户感觉到每次发布的软件都有很大的提高，这样他们自然很高兴。



图一、**CrashRpt** 库检测到错误弹出的对话框

在分析接收的错误记录的时候,我们发现采用下文介绍的方法能够避免大部分程序崩溃的错误。例如、局部变量未初始化导致数组访问越界，指针使用前未进行检测（**NULL**）导致访问访问非法区域等。

我已经总结了几条代码设计的方法和规则，在下文一一列出，希望能够帮助你避免犯一些错误，使得你的程序更加健壮。

- ***Initializing Local Variables***

使用未初始化的局部变量是引起程序崩溃的一个比较普遍的原因，例如、来看下面这段程序片段：

```
  // Define local variables
  BOOL bExitResult; // This will be TRUE if the function exits
successfully
  FILE* f; // Handle to file
  TCHAR szBuffer[_MAX_PATH];   // String buffer
```

```
  // Do something with variables above...
```

上面的这段代码存在着一个潜在的错误，因为没有一个局部变量初始化了。当你的代码运行的时候，这些变量将被默认负一些错误的数值。例如 **bExitResult** 数值将被负为**-135913245** ， **szBuffer** 必须以**"\0"**结尾,结果不会。因此、局部变量初始化时非常重要的，如下正确代码：

```
  // Define local variables

  // Initialize function exit code with FALSE to indicate failure
assumption
  BOOL bExitResult = FALSE; // This will be TRUE if the function
exits successfully
  // Initialize file handle with NULL
  FILE* f = NULL; // Handle to file
  // Initialize string buffer with empty string
  TCHAR szBuffer[_MAX_PATH] = _T("");   // String buffer
  // Do something with variables above...
```

注意：有人说变量初始化会引起程序效率降低，是的，确实如此，如果你确实非常在乎程序的执行效率，去除局部变量初始化，你得想好其后果。

- ***Initializing WinAPI Structures***

许多 **Windows API** 都接受或则返回一些结构体参数，结构体如果没有正确的初始化，也很有可能引起程序崩溃。大家可能会想起用 **ZeroMemory** 宏或者 **memset()**函数去用 **0** 填充这个结构体**(**对结构体对应的元素设置默认值**)**。但是大部分 **Windows API** 结构体都必须有一个 **cbSIze** 参数,这个参数必须设置为这个结构体的大小。

看看下面代码，如何初始化 **Windows API** 结构体参数:

```
  NOTIFYICONDATA nf; // WinAPI structure
  memset(&nf,0,sizeof(NOTIFYICONDATA)); // Zero memory
  nf.cbSize = sizeof(NOTIFYICONDATA); // Set structure size!
  // Initialize other structure members
  nf.hWnd = hWndParent;
  nf.uID = 0;
  nf.uFlags = NIF_ICON | NIF_TIP;
  nf.hIcon = ::LoadIcon(NULL, IDI_APPLICATION);
  _tcscpy_s(nf.szTip, 128, _T("Popup Tip Text"));

  // Add a tray icon
  Shell_NotifyIcon(NIM_ADD, &nf);
```

注意:千万不要用 **ZeroMemory** 和 **memset** 去初始化那些包括结构体对象的结构体,这样很容易破坏其内部结构体,从而导致程序崩溃.

```
  // Declare a C++ structure
  struct ItemInfo
  {
    std::string sItemName; // The structure has std::string object
inside
    int nItemValue;
  };

  // Init the structure
  ItemInfo item;
  // Do not use memset()! It can corrupt the structure
  // memset(&item, 0, sizeof(ItemInfo));
  // Instead use the following
  item.sItemName = "item1";
  item.nItemValue = 0;
```

这里最好是用结构体的构造函数对其成员进行初始化.

```
  // Declare a C++ structure
  struct ItemInfo
  {
    // Use structure constructor to set members with default values
    ItemInfo()
    {
      sItemName = _T("unknown");
      nItemValue = -1;
    }

    std::string sItemName; // The structure has std::string object
inside
    int nItemValue;
  };
  // Init the structure
  ItemInfo item;
  // Do not use memset()! It can corrupt the structure
  // memset(&item, 0, sizeof(ItemInfo));
  // Instead use the following
  item.sItemName = "item1";
  item.nItemValue = 0;
```

- **Validating Function Input**

在函数设计的时候,对传入的参数进行检测是一直都推荐的。例如、如果你设计的函数是公共 **API** 的一部分,它可能被外部客户端调用，这样很难保证客户端传进入的参数就是正确的。

例如，让我们来看看这个 **hypotethical DrawVehicle()** 函数，它可以根据不同的质量来绘制一辆跑车，这个质量数值（**nDrawingQaulity** ）是 **0~100**。**prcDraw** 定义这辆跑车的轮廓区域。

看看下面代码，注意观察我们是如何在使用函数参数之前进行参数检测：

```
BOOL DrawVehicle(HWND hWnd, LPRECT prcDraw, int nDrawingQuality)
{
  // Check that window is valid
  if(!IsWindow(hWnd))
    return FALSE;

  // Check that drawing rect is valid
  if(prcDraw==NULL)
    return FALSE;

  // Check drawing quality is valid
  if(nDrawingQuality<0 || nDrawingQuality>100)
    return FALSE;

  // Now it's safe to draw the vehicle

  // ...

  return TRUE;
}
```

- ***Validating Pointers***

在指针使用之前，不检测是非常普遍的，这个可以说是我们引起软件崩溃最有可能的原因。如果你用一个指针，这个指针刚好是 **NULL**,那么你的程序在运行时,将报出异常。

```
CVehicle* pVehicle = GetCurrentVehicle();

// Validate pointer
if(pVehicle==NULL)
{
  // Invalid pointer, do not use it!
  return FALSE;
```

```
}
```

- ***Initializing Function Output***

如果你的函数创建了一个对象，并要将它作为函数的返回参数。那么记得在使用之前把他复制为**NULL**。如不然，这个函数的调用者将使用这个无效的指针，进而一起程序错误。如下错误代码：

```
int CreateVehicle(CVehicle** ppVehicle)
{
  if(CanCreateVehicle())
  {
    *ppVehicle = new CVehicle();
    return 1;
  }

  // If CanCreateVehicle() returns FALSE,
  // the pointer to *ppVehcile would never be set!
  return 0;
}
```

正确的代码如下；

```
int CreateVehicle(CVehicle** ppVehicle)
{
  // First initialize the output parameter with NULL
  *ppVehicle = NULL;

  if(CanCreateVehicle())
  {
    *ppVehicle = new CVehicle();
    return 1;
  }

  return 0;
}
```

- ***Cleaning Up Pointers to Deleted Objects***

在内存释放之后,无比将指针复制为**NULL**。这样可以确保程序的没有那个地方会再使用无效指针。其实就是，访问一个已经被删除的对象地址，将引起程序异常。如下代码展示如何清除一个指针指向的对象：

```
// Create object
```

```
CVehicle* pVehicle = new CVehicle();

delete pVehicle; // Free pointer
pVehicle = NULL; // Set pointer with NULL
```

- ***Cleaning Up Released Handles***

在释放一个句柄之前，务必将这个句柄复制伪 **NULL** （**0** 或则其他默认值）。这样能够保证程序其他地方不会重复使用无效句柄。看看如下代码，如何清除一个 **Windows** ＡＰＩ的文件句柄：

```
HANDLE hFile = INVALID_HANDLE_VALUE;

// Open file
hFile = CreateFile(_T("example.dat"), FILE_READ|FILE_WRITE,
FILE_OPEN_EXISTING);
if(hFile==INVALID_HANDLE_VALUE)
{
  return FALSE; // Error opening file
}

// Do something with file

// Finally, close the handle
if(hFile!=INVALID_HANDLE_VALUE)
{
  CloseHandle(hFile);   // Close handle to file
  hFile = INVALID_HANDLE_VALUE;   // Clean up handle
}
```

下面代码展示如何清除 **File *** 句柄：

```
// First init file handle pointer with NULL
FILE* f = NULL;

// Open handle to file
errno_t err = _tfopen_s(_T("example.dat"), _T("rb"));
if(err!=0 || f==NULL)
  return FALSE; // Error opening file

// Do something with file
```

```
// When finished, close the handle
if(f!=NULL) // Check that handle is valid
{
  fclose(f);
  f = NULL; // Clean up pointer to handle
}
```

- ***Using delete [] Operator for Arrays***

如果你分配一个单独的对象，可以直接使用 **new** ，同样你释放单个对象的时候,可以直接使用 **delete** . 然而,申请一个对象数组对象的时候可以使用 **new,**但是释放的时候就不能使用 **delete** ,而必须使用 **delete []**:

```
// Create an array of objects
CVehicle* paVehicles = new CVehicle[10];

delete [] paVehicles; // Free pointer to array
paVehicles = NULL; // Set pointer with NULL
or
// Create a buffer of bytes
LPBYTE pBuffer = new BYTE[255];

delete [] pBuffer; // Free pointer to array
pBuffer = NULL; // Set pointer with NULL
```

- ***Allocating Memory Carefully***

有时候，程序需要动态分配一段缓冲区，这个缓冲区是在程序运行的时候决定的。例如、你需要读取一个文件的内容，那么你就需要申请该文件大小的缓冲区来保存该文件的内容。在申请这段内存之前，请注意，**malloc() or new** 是不能申请 **0** 字节的内存，如不然，将导致 **malloc() or new** 函数调用失败。传递错误的参数给 **malloc()** 函数将导致 **C** 运行时错误。如下代码展示如何动态申请内存：

```
// Determine what buffer to allocate.
UINT uBufferSize = GetBufferSize();

LPBYTE* pBuffer = NULL; // Init pointer to buffer

// Allocate a buffer only if buffer size > 0
if(uBufferSize>0)
 pBuffer = new BYTE[uBufferSize];
```

为了进一步了解如何正确的分配内存，你可以读下 Secure Coding Best Practices for Memory Allocation in C and C++这篇文章。

- ***Using Asserts Carefully***

Asserts 用语调试模式检测先决条件和后置条件。但当我们编译器处于 **release** 模式的时候，**Asserts** 在预编阶段被移除。因此，用 **Asserts** 是不能够检测我们的程序状态,错误代码如下：

```
#include <assert.h>

 // This function reads a sports car's model from a file
 CVehicle* ReadVehicleModelFromFile(LPCTSTR szFileName)
 {
   CVehicle* pVehicle = NULL; // Pointer to vehicle object

   // Check preconditions
   assert(szFileName!=NULL); // This will be removed by preprocessor
in Release mode!
   assert(_tcslen(szFileName)!=0); // This will be removed in
Release mode!

   // Open the file
   FILE* f = _tfopen(szFileName, _T("rt"));

   // Create new CVehicle object
   pVehicle = new CVehicle();

   // Read vehicle model from file

   // Check postcondition
   assert(pVehicle->GetWheelCount()==4); // This will be removed in
Release mode!

   // Return pointer to the vehicle object
   return pVehicle;
 }
```

看看上述的代码，**Asserts** 能够在 **debug** 模式下检测我们的程序,在 **release** 模式下却不能。所以我们还是不得不用 **if()**来这步检测操作。正确的代码如下；

```
CVehicle* ReadVehicleModelFromFile(LPCTSTR szFileName, )
```

```
  {
    CVehicle* pVehicle = NULL; // Pointer to vehicle object

    // Check preconditions
    assert(szFileName!=NULL); // This will be removed by preprocessor
in Release mode!
    assert(_tcslen(szFileName)!=0); // This will be removed in
Release mode!

    if(szFileName==NULL || _tcslen(szFileName)==0)
      return NULL; // Invalid input parameter

    // Open the file
    FILE* f = _tfopen(szFileName, _T("rt"));

    // Create new CVehicle object
    pVehicle = new CVehicle();

    // Read vehicle model from file

    // Check postcondition
    assert(pVehicle->GetWheelCount()==4); // This will be removed in
Release mode!

    if(pVehicle->GetWheelCount()!=4)
    {
      // Oops... an invalid wheel count was encountered!
      delete pVehicle;
      pVehicle = NULL;
    }

    // Return pointer to the vehicle object
    return pVehicle;
  }
```

- ***Checking Return Code of a Function***

断定一个函数执行一定成功是一种常见的错误。当你调用一个函数的时候，建议检查下返回代码和返回参数的值。如下代码持续调用 **Windows API** ,程序是否继续执行下去依赖于该函数的返回结果和返回参数值。

```
HRESULT hres = E_FAIL;
    IWbemServices *pSvc = NULL;
    IWbemLocator *pLoc = NULL;

    hres =  CoInitializeSecurity(
        NULL,
        -1,                          // COM authentication
        NULL,                        // Authentication services
        NULL,                        // Reserved
        RPC_C_AUTHN_LEVEL_DEFAULT,   // Default authentication
        RPC_C_IMP_LEVEL_IMPERSONATE, // Default Impersonation
        NULL,                        // Authentication info
        EOAC_NONE,                   // Additional capabilities
        NULL                         // Reserved
        );

    if (FAILED(hres))
    {
        // Failed to initialize security
        if(hres!=RPC_E_TOO_LATE)
           return FALSE;
    }

    hres = CoCreateInstance(
        CLSID_WbemLocator,
        0,
        CLSCTX_INPROC_SERVER,
        IID_IWbemLocator, (LPVOID *) &pLoc);

    if (FAILED(hres) || !pLoc)
    {
        // Failed to create IWbemLocator object.
        return FALSE;
    }
```

```
hres = pLoc->ConnectServer(
     _bstr_t(L"ROOT\\CIMV2"), // Object path of WMI namespace
     NULL,                    // User name. NULL = current user
     NULL,                    // User password. NULL = current
     0,                       // Locale. NULL indicates current
     NULL,                    // Security flags.
     0,                       // Authority (e.g. Kerberos)
     0,                       // Context object
     &pSvc                    // pointer to IWbemServices proxy
     );

if (FAILED(hres) || !pSvc)
{
    // Couldn't conect server
    if(pLoc) pLoc->Release();
    return FALSE;
}
hres = CoSetProxyBlanket(
   pSvc,                         // Indicates the proxy to set
   RPC_C_AUTHN_WINNT,            // RPC_C_AUTHN_xxx
   RPC_C_AUTHZ_NONE,             // RPC_C_AUTHZ_xxx
   NULL,                         // Server principal name
   RPC_C_AUTHN_LEVEL_CALL,       // RPC_C_AUTHN_LEVEL_xxx
   RPC_C_IMP_LEVEL_IMPERSONATE, // RPC_C_IMP_LEVEL_xxx
   NULL,                         // client identity
   EOAC_NONE                     // proxy capabilities
);
if (FAILED(hres))
{
    // Could not set proxy blanket.
    if(pSvc) pSvc->Release();
    if(pLoc) pLoc->Release();
    return FALSE;
}
```

- ***Using Smart Pointers***

如果你经常使用用享对象指针，如 **COM** 接口等，那么建议使用智能指针来处理。智能指针会自动帮助你维护对象引用记数，并且保证你不会访问到被删除的对象。这样，不需要关心和控制接口的生命周期。关于智能指针的进一步知识可以看看 **Smart Pointers - What, Why, Which?** 和 **Implementing a Simple Smart Pointer in C++**这两篇文章。

如面是一个展示使用 **ATL's CComPtr template** 智能指针的代码，该部分代码来至于 **MSDN**。

```
#include <windows.h>
#include <shobjidl.h>
#include <atlbase.h> // Contains the declaration of CComPtr.
int WINAPI wWinMain(HINSTANCE hInstance, HINSTANCE, PWSTR pCmdLine,
int nCmdShow)
{
    HRESULT hr = CoInitializeEx(NULL, COINIT_APARTMENTTHREADED |
        COINIT_DISABLE_OLE1DDE);
    if (SUCCEEDED(hr))
    {
        CComPtr<IFileOpenDialog> pFileOpen;
        // Create the FileOpenDialog object.
        hr = pFileOpen.CoCreateInstance(__uuidof(FileOpenDialog));
        if (SUCCEEDED(hr))
        {
            // Show the Open dialog box.
            hr = pFileOpen->Show(NULL);
            // Get the file name from the dialog box.
            if (SUCCEEDED(hr))
            {
                CComPtr<IShellItem> pItem;
                hr = pFileOpen->GetResult(&pItem);
                if (SUCCEEDED(hr))
                {
                    PWSTR pszFilePath;
                    hr = pItem->GetDisplayName(SIGDN_FILESYSPATH,
&pszFilePath);
                    // Display the file name to the user.
                    if (SUCCEEDED(hr))
                    {
                        MessageBox(NULL, pszFilePath, L"File Path",
MB_OK);
                        CoTaskMemFree(pszFilePath);
```

```
                    }
                }
                // pItem goes out of scope.
            }
            // pFileOpen goes out of scope.
        }
        CoUninitialize();
    }
    return 0;
}
```

- ***Using == Operator Carefully***

先来看看如下代码;

```
  CVehicle* pVehicle = GetCurrentVehicle();

  // Validate pointer
  if(pVehicle==NULL) // Using == operator to compare pointer with
NULL
     return FALSE;

  // Do something with the pointer
  pVehicle->Run();
```

上面的代码是正确的,用语指针检测。但是如果不小心用**“=”**替换了**“==”**，如下代码;

```
 CVehicle* pVehicle = GetCurrentVehicle();

  // Validate pointer
  if(pVehicle=NULL) // Oops! A mistyping here!
     return FALSE;

  // Do something with the pointer
  pVehicle->Run(); // Crash!!!
```

看看上面的代码,这个的一个失误将导致程序崩溃。

这样的错误是可以避免的，只需要将等号左右两边交换一下就可以了。如果在修改代码的时候，你不小心产生这种失误，这个错误在程序编译的时候将被检测出来。