Глава 15

# Управление памятью, отладка приложения и поиск ошибок



Поиск ошибок является одной из важнейших частей процесса разработки программы. В данной главе описываются механизмы, предоставляемые библиотекой, для обнаружения проблем с памятью, а также средства, способствующие использованию принципа «защитного программирования» — обнаружения проблем так рано, как это только возможно. Все это делает отладку программы значительно более легким делом, а само приложение — более надежным. Также вы узнаете когда стоит создавать объект в куче, а когда можно использовать стек. Мы обсудим использование информации о типе времени выполнения (Run-Time Type Information, RTTI), механизм модулей и поддержку библиотекой исключений языка C++. В конце главы есть несколько общих советов касающихся отладки.

## 15.1 Основы управления памятью

При программировании на языке C++ у вас есть возможность создавать объекты либо на стеке, либо в куче, используя оператор `new`. Объект, созданный на стеке, доступен только в пределах своей области видимости. В момент выхода из нее будет автоматически вызван деструктор объекта, после чего объект просто перестанет существовать. С другой стороны, объект, созданный в куче, будет доступен до тех пор, пока вы явно не удалите его оператором `delete` или программа не завершится.

### 15.1.1 Создание и удаление объекта окна

Как правило, оконные объекты, такие как `wxFrame` или `wxButton`, создаются в куче, используя оператор `new`. Обычно такие объекты должны существовать неопределенное время до тех пор, пока пользователь не решит закрыть окно. Учтите, что wxWidgets автоматически уничтожает дочерние объекты при уничтожении родителя. Таким образом, нет необходимости напрямую удалять элементы управления, размещенные в окне, достаточно уничтожить само диалоговое окно, используя `Destroy`. Аналогично, при удалении фрейма автоматически удаляются

все дочерние объекты внутри него. Однако, если вы создали окно верхнего уровня (например, фрейм) как дочернее другому окну верхнего уровня (другому фрейму), то родительский фрейм не уничтожит такой дочерний элемент. Исключением из данного правила является MDI-интерфейс (Multiple Document Interface), где дочерние фреймы не являются независимыми окнами и поэтому удаляются родителем.

Диалоги можно создавать на стеке, однако это верно только для модальных диалогов: вызовите `ShowModal` для передачи управления в цикл обработки сообщений диалога, таким образом вся работа пользователя с диалогом закончится до того, как объект выйдет за пределы видимости и будет уничтожен.

Механизм закрытия и удаления оконных фреймов и диалогов вполне может служить источником недоразумений. Для удаления фрейма или модального диалога приложение должно использовать `Destroy`. Эта функция реализует отложенное удаление, то есть окно будет фактически удалено только после того, как будет обработана вся его очередь сообщений. Такое поведение намеренно сделано для того, чтобы избежать посылки сообщений несуществующему окну. Однако, для модальных диалогов сначала необходимо вызвать метод `EndModal`, чтобы выйти из цикла обработки сообщений. Обработчик сообщения (к примеру для кнопки «OK») не должен уничтожать диалог, так как при создании модального диалога на стеке, он будет уничтожаться дважды: в первый раз в обработчике сообщения и еще раз, когда объект покинет область видимости. Когда пользователь закрывает модальный диалог возникает событие `wxEVT_CLOSE_WINDOW` и соответствующий обработчик должен вызвать метод `EndModal` (не не должен разрушать диалог). По умолчанию поведение при нажатии кнопки закрытия в заголовке окна — это эмуляция события `wxID_CANCEL`, обработчик которого обычно закрывает диалог. Затем диалог уничтожается при выходе из области видимости. Таким образом работают стандартные диалоги, например `wxFileDialog` или `wxColourDialog`, позволяя вам, тем самым, получить значения полей диалога после его закрытия и возврата управления. Вы можете разработать такой модальный диалог, который уничтожит себя в обработчике события, но тогда вы не сможете создать такой диалог в стеке или получать значения его данных после закрытия.

Ниже показаны два варианта использования `wxMessageDialog`:

```
// 1) Создаем диалог в стеке, без явного удаления
wxMessageDialog dialog(NULL, _("Press OK"), _("App"), wxOK|wxCANCEL);
if (dialog.ShowModal() == wxID_OK)
{
    // 2) Создаем диалог в куче: должны уничтожить используя Destroy()
    wxMessageDialog* dialog = new wxMessageDialog(NULL,
        _("Thank you! "), _("App"), wxOK);
    dialog->ShowModal();
    dialog->Destroy();
}
```

Немодальные диалоги и фреймы обычно при закрытии удаляют себя сами. Они не могут быть созданы в стеке, так как сразу же выйдут из области видимости и будут уничтожены.

Если вы храните указатель на оконный объект, то обязательно обнуляйте его при разрушении окна. Код для решения данной задачи можно поместить в деструктор окна или обработчик его закрытия. Например:

```
void MyFindReplaceDialog::OnCloseWindow(wxCloseEvent& event)
{
    wxGetApp().SetFindReplaceDialog(NULL);
    Destroy();
}
```

### 15.1.2 Создание и копирование объектов для рисования

Объекты рисования, такие как `wxBrush`, `wxPen`, `wxColour`, `wxBitmap` и `wxImage`, можно создавать как в стеке, так и в куче. Код рисования обычно создает временные переменные на стеке. Так как классы объектов для рисования реализованы с использованием счетчиков ссылок, то это позволяет использовать сами объекты (а не указатели на них) с небольшими накладными расходами на копирование: то есть при присваивании объекта кисти (`wxBrush`) другой переменной в действительности происходит копирование только ссылки на данные. Однако у такого поведения есть и неожиданные побочные эффекты, проявляющиеся тем, что изменение одной переменной может приводить к изменению другой. Чтобы избежать подобных связей между объектами, создавайте копии, используя подходящий конструктор и явно присваивайте свойства исходного объекта копии. Ниже приводится несколько примеров «настоящего» копирования, а также использования подсчета ссылок.

```
// Копирование с подсчетом ссылок
wxBitmap newBitmap = oldBitmap;

// Настоящее копирование
wxBitmap newBitmap = oldBitmap.GetSubBitmap(
    wxRect(0, 0, oldBitmap.GetWidth(), oldBitmap.GetHeight()));

// Копирование с подсчетом ссылок
wxFont newFont = oldFont;

// Настоящее копирование
wxFont newFont(oldFont.GetPointSize(), oldFont.GetFamily(),
    oldFont.GetStyle(), oldFont.GetWeight(),
    oldFont.GetUnderlined(), oldFont.GetFaceName());
```

### 15.1.3 Инициализация объекта приложения

Объект приложения вполне может быть создан до того, как wxWidgets проинициализирует все свои внутренние объекты, такие как базу цветов или шрифт, используемый по умолчанию. Поэтому будьте осторожны и не используйте подобного типа объекты в конструкторе класса приложения. Вместо этого, в конструкторе инициализируйте такие члены класса значением по умолчанию, а правильное значение присвойте им в методе `OnInit`. Например,

```
MyApp::MyApp()
{
    // Не устанавливайте значение в данном месте!
    // m_font = *wxNORMAL_FONT;
}

bool MyApp::OnInit()
{
    m_font = *wxNORMAL_FONT;
    ...
    return true;
}
```

### 15.1.4 Очистка памяти

Можно перегрузить метод `wxApp::OnExit` и разместить в нем код для очистки данных приложения, а также удаления ненужных объектов. Данный метод вызывается уже после удаления всех окон приложения, но перед тем, как wxWidgets приступит к удалению своих внутренних структур. Однако, в некоторых случаях очистку необходимо выполнять в обработчике закрытия главного окна приложения. Например, в этом месте необходимо удалить поток, который может осуществлять чтение/запись данных в окно приложения.

## 15.2 Обнаружение утечек памяти и прочих ошибок

В идеальном мире при завершении приложения все объекты должны быть корректно удалены либо самим приложением, либо библиотекой wxWidgets. Вы не должны оставлять не удаленной использованную память, даже учитывая то, что операционная система автоматически освободит память вместо вас после завершения программы. Несмотря на искушение махнуть рукой на очистку объектов, вам действительно стоит добиваться удаления всего того, что вы создали. Часто подобные утечки памяти — это симптомы проблем с вашим кодом, которые могут привести к потерям больших объемов памяти во время работы программы. Чем больше прошло времени с момента написания проблемного кода, тем труднее его обнаружить, поэтому постарайтесь привыкнуть к тому, чтобы никогда не оставлять в программе даже малейших утечек памяти.

Как можно проверить, есть ли в вашей программе утечки памяти? Во-первых, можно найти различные сторонние решения для обнаружения подобных проблем (memory-checking tools). Кроме того, сама библиотека предоставляет простой встроенный механизм такой проверки. Для его использования в отладочной конфигурации вашего проекта, установите ряд опций в файле setup.h (в системе Windows) или изменить опции сборки (для других платформ или при использовании компилятора CGG в Windows).

Опции, которые необходимо установить в файле setup.h:

```
#define wxUSE_DEBUG_CONTEXT 1
#define wxUSE_MEMORY_TRACING 1
```

```
#define wxUSE_GLOBAL_MEMORY_OPERATORS 1
#define wxUSE_DEBUG_NEW_ALWAYS 1
```

При конфигурировании используются следующие ключи:

```
--enable-debug --enable-mem_tracing --enable-debug_cntxt
```

Но данный механизм имеет ряд ограничений: он не работает с MinGW и Cygwin (по крайней мере, на момент написания книги), а также нельзя использовать `wxUSE_DEBUG_NEW_ALWAYS`, если вы используете STL или компилятор CodeWarrior.

При включенном `wxUSE_DEBUG_NEW_ALWAYS` каждый встречающийся оператор `new` как в самой wxWidgets, так и в вашем коде будет переопределен как `new(__TFILE__,__LINE__)`, что приведет к использованию специальных процедур для выделения и освобождения памяти. Кроме того можно явно использовать модифицированную версию оператора, просто вызывая `WXDEBUG_NEW` вместо обычного оператора `new`.

Самый простой способ использования внутренней системы обнаружения утечек памяти не требует никаких действий. Запустите программу под управлением отладчика и закройте ее, а отладчик сам сообщит вам об обнаруженных ошибках, например вот так:

```
There were memory leaks. (найдены утечки памяти)

- Memory dump -
.\memcheck.cpp(89): wxBrush at 0xBE44B8, size 12
..\..\src\msw\brush.cpp(233): non-object data at 0xBE55A8, size 44
.\memcheck.cpp(90): wxBitmap at 0xBE5088, size 12
..\..\src\msw\bitmap.cpp(524): non-object data at 0xBE6FB8, size 52
.\memcheck.cpp(93): non-object data at 0xBB8410, size 1000
.\memcheck.cpp(95): non-object data at 0xBE6F58, size 4
.\memcheck.cpp(98): non-object data at 0xBE6EF8, size 8

- Memory statistics -
1 objects of class wxBitmap, total size 12
5 objects of class nonobject, total size 1108
1 objects of class wxBrush, total size 12

Number of object items: 2
Number of non-object items: 5
Total allocated size: 1132
```

Показанный отчет говорит о том, что были созданы, но не были освобождены `wxBrush`, `wxBitmap` и еще один объект неизвестного библиотеке класса (объект не содержит информации о типе). В некоторых средах разработки можно будет быстро перейти к строке кода, выделяющий память, всего лишь дважды кликнув на соответствующую строку в сообщении об ошибке. Эта возможность является отличным первым шагом на пути к поиску проблемного места. Чтобы добиться лучших результатов всегда добавляйте информацию времени выполнения (run-time

type information, RTTI) к любому классу-наследнику от `wxObject`. Для этого просто добавьте `DECLARE_CLASS(класс)` в определение класса и `IMPLEMENT_CLASS(класс, базовый_класс)` в любом месте файла с реализацией.

Вышеописанная система слежения за памятью также пытается отловить попытки перезаписи памяти и ошибки двойного удаления. Функции, отвечающие за выделение и освобождение памяти, ставят специальные метки для «хороших» и уже удаленных блоков памяти. Как только приложение попробует удалить блок памяти без соответствующей метки, то вы сразу получите сообщение о проблеме. То же самое произойдет и при попытке удалить блок, помеченный как уже удаленный. Такие сообщения облегчают обнаружение проблем, связанных с порчей памяти.

Используя статические методы класса `wxDebugContext`, можно получить список текущих объектов в памяти, вызвав `PrintClasses`, или показать количество объектов и не-объектов (это объекты без информации о времени выполнения), вызвав `PrintStatistics`. Используя `SetCheckpoint` можно сказать `wxDebugContext`, что необходимо показывать статистику для объектов, объявленных после данного момента, игнорируя выделения памяти, произошедшие ранее. Более подробную информацию можно получить из справки по классу `wxDebugContext` и примера samples/memcheck.

Вместо использования простой системы обнаружения утечек, которую предлагает wxWidgets, можно использовать коммерческие продукты, такие как BoundsChecker, Purify, AQtime или бесплатные альтернативы StackWalker, ValGrind, Electric Fence или MMGR от Fluid Studios. Если вы используете Visual C++, то wxWidgets будет использовать стандартный для данного компилятора механизм обнаружение ошибок, который не сообщает вам имя класса, но сообщает номер соответствующей строки кода. Для лучших результатов убедитесь, что `wxUSE_DEBUG_NEW_ALWAYS` установлен в `1` в setup.h. Так как данный механизм переопределяет оператор `new`, то в некоторых случаях возможно понадобится отключить его, если появятся проблемы с другими библиотеками.

## 15.3 Инструменты для защитного программирования

Достаточно часто проблема становится видимой только спустя некоторое время после возникновения настоящего источника проблемы. Если ошибочное значение не будет обнаружено вовремя, то программа может успеть выполнить тысячи строк кода перед тем как упасть или выдать мистический результат. Можно потратить уйму времени, пытаясь найти реальную причину ошибки. Однако, стоит вам просто добавить проверки для своевременного обнаружения таких «плохих» значений, например, проверки значений аргументов функции, как ваш код сразу станет куда более стабильным. Тем самым вы можете оградить себя (и своих пользователей) от огромного количества потенциальных проблем. Такая методика написания кода называется «защитным программированием». Ваши классы и функции смогут защитить себя как против «неправильного» использования вышестоящим кодом, так и против внутренних логических ошибок. Учитывая то, что большинство таких проверок удаляются компилятором из релизной сборки, то вам не прийдется платить за них увеличенным размером файла с программой или скоростью выполнения.

Как вы уже догадались, в коде самой библиотеки wxWidgets выполняется огромное количество разнообразных проверок для обеспечения надежности, а значит вы можете использовать те же макросы и функции в своем коде. Существует три основных семейства макросов — вариации на тему `wxASSERT`, `wxFAIL` и `wxCHECK`. `wxASSERT` показывает сообщение об ошибке, если условие аргумента ложно. Данная проверка осуществляется только в отладочной сборке. `wxFAIL` всегда генерирует сообщение об ошибке, таким образом она эквивалентна конструкции `wxASSERT(false)`. Данная проверка также отсутствует в релизной версии программы. `wxCHECK` проверяет условие на истинность и обеспечивает выход из функции с возвратом указанного значения, если условие не выполняется. В отличии от двух других групп макросов, `wxCHECK` присутствует и выполняется (правда уже без показа сообщения) и в финальной версии программы. Все указанные макросы имеют модификации, позволяющие вам выводить собственное сообщение об ошибке.

Давайте рассмотрим несколько примеров использования таких макросов:

```
// Найдем сумму двух положительных чисел
int AddPositive(int a, int b)
{
    // Проверим, что число a положительное
    wxASSERT(a > 0);

    // Проверим, что число b положительное, а в случае
    // ошибки напишем свое сообщение
    wxASSERT_MSG(b > 0, wxT("The second number must be positive!"));

    int c = a + b;
    // Вернем -1, если результат отрицателен или равен 0
    wxCHECK_MSG(c > 0, -1, wxT("Result must be positive!"));

    return c;
}
```

Если вместо простого возврата из функции при провале проверки вам хочется выполнить произвольные операции, то можно использовать макросы `wxCHECK2` и `wxCHECK2_MSG`. Макрос `wxCHECK_RET` просто прерывает функцию, не возвращая ничего и используется, как правило, в функциях, возвращающих `void`. Информацию по редко используемым макросам `wxCOMPILE_TIME_ASSERT` и `wxASSERT_MIN_BITSIZE` легко найти в справке.

На рисунке 15.1 показан диалог с сообщением об ошибке, который появляется, если условие проверки не выполняется (равно `false`). Пользователь может остановить программу (ответ «Yes»), проигнорировать ошибку (ответ «No») или проигнорировать это и все дальнейшие предупреждения (ответ «Cancel»). Если программа выполняется под управлением отладчика, то остановка приведет к вызову прерывания отладчика и вы сразу перейдете к строчке кода, в которой была обнаружена проблема, и сможете посмотреть значения переменных в данной точке.

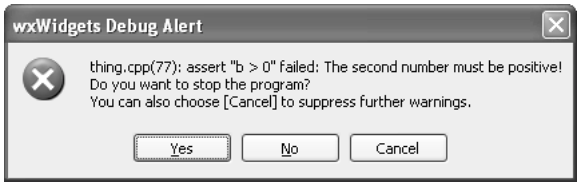


Рис. 15.1: Сообщение об ошибке

## 15.4 Сообщения об ошибках

Иногда появляется необходимость вывести некоторое сообщение в консоли или в диалоге, чтобы помочь при отладке или в ситуации, когда ошибка не может быть нормально обработана самим приложением. Для таких целей wxWidgets предоставляет набор функций логирования, использующих различные способы вывода сообщений. Например, если во время создание `wxBitmap` большого размера обнаружился недостаток свободной памяти, то программа может использовать `wxLogError` для вывода сообщения об ошибке с использованием всплывающего диалога (рисунок 15.2). Или если вы захотите вывести значения аргументов функции в окно вывода отладочной информации, то просто используйте `wxLogDebug`. Место, где сообщение реально появится (в диалоге, в окне отладчика или в стандартном потоке ошибок) зависит от имени используемой функции, а также от установленной в данный момент активной цели (wxLog target).

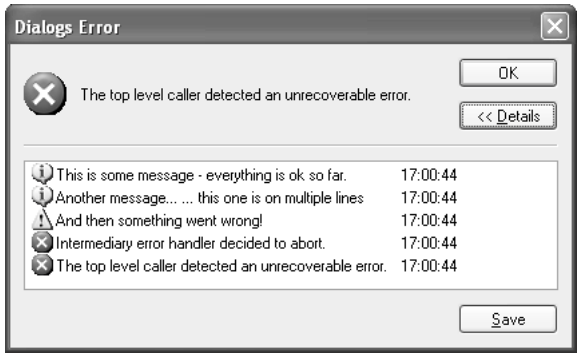


Рис. 15.2: Окно диалога с логом

Все функции логирования имеют одинаковый синтаксис, похожий на синтксис `printf` и `vprintf`. Они принимают строку формата в качестве первого аргумента, за которым следует произвольное количество аргументов или указатель на список аргументов. Например:

```
wxString name(wxT("Calculation"));
int nGoes = 3;

wxLogError(wxT("%s does not compute! You have %d more goes."),
           name.c_str(), nGoes);
```

Библиотека поддерживает следующие функции логирования:

wxLogError используется для сообщений об ошибках, которые должны быть показаны пользователю. По умолчанию, для информирования пользователя об ошибке функция использует всплывающее диалоговое окно. Почему тогда просто не использовать wxMessageBox? Во-первых, вы можете локально запретить вывод сообщений об ошибках, используя wxLogNull. Во-вторых, все ошибки, переданные данной функции, добавляются в специальную очередь и показываются пользователю во время простоя в одном диалоге. Таким образом, в случае возникновения нескольких последовательных ошибок пользователю не придется нажимать «OK» в сообщении о каждой из них.

wxLogFatalError подобна wxLogError, но дополнительно прерывает выполнение программы, используя стандартную функцию abort и выдавая 3 в качестве кода завершения программы. В отличии от всех остальных функций, ее вывод не может перенаправлен изменением цели логирования.

wxLogWarning подобна wxLogError, но сообщения выглядят для пользователя как информация, а не как ошибка.

wxLogMessage предназначена для всех нормальных, информационных сообщений. По умолчанию сообщения выводится во всплывающем окне.

wxLogVerbose используется для подробного режима вывода сообщений. Обычно, сообщения переданные этой функции подавляются, но вы можете изменить данное поведение просто вызвав wxLog::SetVerbose, если пользователь захочет узнать больше деталей о ходе работы программы.

wxLogStatus предназначена для показа статусной информации, сообщения отображаются в статусбаре активного или указанного фрейма при условии, что он существует.

wxLogSysError по большей части используется самой библиотекой для логирования каких-то определенных сообщений, так и информации о коде ошибки последней операции (значение errno или результат работы GetLastError в зависимости от платформы) и соответствующего сообщения об ошибке. Вторая форма данной функции принимает код ошибки в качестве первого аргумента.

wxLogDebug используется для вывода отладочной информации. Сообщения отображаются только в отладочной версии программы (если определен __WXDEBUG__). При работе в системе Windows вы должны запустить программу под управлением отладчика или использовать сторонние утилиты (например, DebugView от http://www.sysinternals.com) для того, чтобы увидеть такие сообщения.

wxLogTrace подобно wxLogDebug работает только в отладочной версии. Причина, по которой эта функциональность вынесена в отдельную функцию кроется в том, что обычно существует большое количество сообщений трассировки, поэтому имеет смысл отделить их от всей остальной отладочной информации. Более того, вторая версия этой функции принимает первым аргументом специальную маску, которая позволяет вам в дальнейшем ограничивать количество генерируемых сообщений. Например, wxWidget внутри использует маску mousecapture, поэтому если вы добавите эту строку к маске трасировки, используя wxLog::AddTraceMask, то увидите сообщения трассировки при захвате мыши.

```
void wxWindowBase::CaptureMouse()
{
    wxLogTrace(wxT("mousecapture"), wxT("CaptureMouse(%p) "), this);
```

```
    ...
}
void MyApp::OnInit()
{
    // Добавьте mousecapture к маске трасировки
    wxLog::AddTraceMask(wxT("mousecapture"));
    ...
}
```

Вы можете удивиться — почему бы не использовать стандартные средства ввода/вывода языка C или потоки библиотеки C++? Если кратко, то это хорошие обобщенный механизмы, но, к сожалению, они плохо адаптируется к работе с wxWidgets. У встроенных классов для ведения лога имеется три основных преимущества.

Во-первых, `wxLog` является кроссплатформенным механизмом. Распространенная практика — использование возможностей `printf` или потоков `cout` и `cerr` в C++ для вывода информации. Это замечательно работает в Unix-системах, но не работает в Windows, где такие сообщения просто не будут никуда выведены, так как устройство вывода для графических приложений ни с чем не связано. Таким образом можно рассматривать `wxLogMessage` как замену для `printf`.

Вы можете легко перенаправить вывод логов в `cout` просто написав:

```
wxLog *logger = new wxLogStream(&cout);
wxLog::SetActiveTarget(logger);
```

Более того, существует возможность перенаправить вывод, посылаемый в `cout`, в текстовое поле `wxTextCtrl`, используя класс `wxStreamToTextRedirector`.

Второе, `wxLog` — гибче. Вывод функций `wxLog` может быть перенаправлен или подавлен в зависимости от важности сообщений, чего очень трудно или даже невозможно добиться, используя традиционные методы. Например, можно выводить в лог только ошибки или только ошибки и предупреждения, отфильтровывая все остальные информационные сообщения.

И наконец, `wxLog` более проработанный. Обычно, сообщение об ошибке должно быть показано пользователю в случае невозможности выполнить некоторую операцию. Давайте рассмотрим простой, но часто встречающийся случай ошибки работы с файлом: представьте, что вы пишите данные в файл и оказывается, что сделать это невозможно, так как отсутствует достаточного свободного места. Ошибка может быть обнаружена внутри кода библиотеки wxWidgets (например, в `wxFile::Write`), так как вызывающий код реально может и не знать в чем причина возникновения ошибки, он знает только то, что данные не могут быть записаны на диск. Однако, так как wxWidgets использует `wxLogError`, то правильный код ошибки и соответствующее сообщение будет показано пользователю.

Теперь давайте разберемся как работать с функциями логирования в случае, если вам необходимо нечто большее, чем стандартное поведение.

В wxWidgets существует такое понятие как цель логирования (log target) — по сути это просто класс-наследник от `wxLog`, который реализует виртуальные функции базового класса, которые вызываются в случае, когда необходимо логировать сообщение. В каждый момент времени только одна цель может быть активной.

Обычное использование объекта логирования заключается в его установке в качестве активной цели при помощи вызова `wxLog::SetActiveTarget`, после чего объект будет использоваться всеми дальнейшими вызовами функций логирования.

Для создания нового класса цели логирования достаточно унаследовать его от `wxLog` и реализовать один из методов `DoLog` или `DoLogString` (или оба сразу). Метод `DoLogString` перегружается, если вас устраивает стандартное оформление сообщений в `wxLog` (добавление в начало сообщения строки «Error» или «Warning> и текущего времени), а вы просто хотите перенаправлять сообщение в другое место. Метод `DoLog` перегружается, если хочется сделать нечто большее, но тогда вам придется самостоятельно обрабатывать различные уровни сообщений. Обратитесь к файлу src/common/log.cpp в качестве примера того, как wxWidgets делает это.

Существует несколько готовых классов-наследников от `wxLog`, которые можно использовать без изменений. Кроме того просмотр исходных текстов данных классов может оказаться полезным примером при реализации собственных целей логирования.

`wxLogStderr` записывает сообщения в передаваемый ей файл (`FILE*`) или в `stderr`, если файл не указан.

`wxLogStream` делает то же самое, что и `wxLogStderr`, но использует `ostream` и `cerr` вместо `FILE*` и `stderr`.

`wxLogGui` является стандартной целью логирования для приложений, написанных с помощью wxWidgets, и используется по умолчанию. Она использует наиболее подходящую для текущей платформы обработку сообщений.

`wxLogWindow` создает «консоль для логов», которая выводит все сообщения, создаваемые приложением, а также передает их в предыдущую цель логирования. Окно данной консоли имеет меню, позволяющее пользователю очистить лог, закрыть его или сохранить все сообщения в файл.

`wxLogNull` можно использовать для временного подавления вывода функций логирования. В качестве примера, попробуем открыть не существующий файл, что должно привести к появлению сообщения об ошибке, которое мы по некоторой причине в этом месте не хотим видеть. Все, что нам необходимо сделать — это создать объект класса `wxLogNull` на стеке. В результате, в области жизни данного объекта любой вывод логирующих функций будет подавлен. Вы можете использовать дополнительную пару фигурных скобок для создания нужной области жизни объекта. Например:

```
wxFile file;
// wxFile.Open() Обычно выдает сообщение если файл
// не может быть открыт, которое сейчас нам не нужно
{
    wxLogNull logNo;
    if ( !file.Open("bar") )
      ... обратываем ошибку самостоятельн ...
}  // Вызывается ~wxLogNull, восстанавливая старое поведение
wxLogMessage("..."); // Все работает как прежде
```

Можно объединять несколько целей в цепочку: например, можно перенаправлять сообщения куда-нибудь (например в файл), но при этом также обрабатывать их

как обычно. Для этого необходимо использовать wxLogChain и wxLogPassThrough. Например:

```
// Явно устанавливаем цель логгирования
wxLogChain *logChain = new wxLogChain(new wxLogStderr);

// Теперь все сообщения будут посылаться в stderr
// и при этом обрабатыватся как обычно

// Не удаляйте logChain явно, используйте
// вместо этого SetActiveTarget
delete wxLog::SetActiveTarget(new wxLogGui);
```

### 15.4.1 wxMessageOutput против wxLog

Иногда wxLog оказывается неподходящим классом из-за его высокоуровневой функциональности, такой как автоматическое добавление времени и отложенный показ сообщений. Класс wxMessageOutput и его наследники предлагают низкоуровневую замену printf, которую можно использовать как в консольных приложениях, так и в программах с графическим пользовательским интерфейсом. Используйте wxMessageOutput::Printf, если вам необходим printf. Например, для записи ошибки:

```
#include "wx/msgout.h"

wxMessageOutputStderr err;
err.Printf(wxT("Error in app %s.\n"), appName.c_str());
```

Также можно использовать wxMessageOutputDebug для вывода сообщений в окно отладчика или в стандартный поток ошибок, в зависимости от текущей платформы или от того, запущена ли программа под управлением отладчика. Графические приложения могут использовать wxMessageOutputMessageBox, чтобы незамедлительно показать сообщения в диалоге, вместо стандартного помещения в очередь, как это делает wxLog.

Как и wxLog, класс wxMessageOutput поддерживает концепцию текущей активной цели. Используйте wxMessageOutput::Set для установки и wxMessageOutput::Get для получения текущей активной цели. Эта цель устанавливается в соответствующий объект при инициализации библиотеки: экземпляр wxMessageOutputStderr используется для консольных приложений, а wxMessageOutputMessageBox для графических программ. wxWidgets сама использует данных механизм, например в классе wxCmdLineParser, следующим образом:

```
wxMessageOutput* msgOut = wxMessageOutput::Get();
if ( msgOut )
{
    wxString usage = GetUsageString();

    msgOut->Printf( wxT("%s%s"), usage.c_str(), errorMsg.c_str() );
```

```
}
else
{
    wxFAIL_MSG( _T("no wxMessageOutput object?") );
}
```

## 15.5 Поддержка информации времени выполнения (Run-Time Type Information)

Как и большинство других фреймворков, wxWidgets обеспечивает более качественную работу с RTTI (информацией времени выполнения о типе), по сравнению со стандартным механизмом RTTI языка C++. Данная возможность может быть полезна для принятия решений, основанных на типе объекта во время выполнения программы или, как мы увидели в последнем разделе, для более подробных сообщениях об ошибках. Также это дает возможность динамически создавать объекты, просто передавая имя класса в виде строки. Необходимым условием поддержки RTTI в классе является его наследование от `wxObject`.

Если вам не нужна возможность динамического создания объектов, то используйте макрос `DECLARE_CLASS(класс)` в объявлении класса и `IMPLEMENT_CLASS(класс, базовый_класс)` в файле с реализацией. Если вы допускаете динамическое создание экземпляров класса с использованием RTTI, то используйте `DECLARE_DYNAMIC_CLASS(класс)` в определении и `IMPLEMENT_DYNAMIC_CLASS(класс, базовый_класс)` в файле с реализацией. В последнем случае вы также должны убедиться, что класс имеет конструктор по умолчанию, в противном случае компилятор выдаст предупреждение при попытке скомпилировать функции, автоматически генерируемые wxWidgets для динамического создания объекта класса.

Пример использования RTTI для динамического создания объекта класса:

```
class MyRecord: public wxObject
{
DECLARE_DYNAMIC_CLASS(MyRecord)
public:
    MyRecord() {}
    MyRecord(const wxString& name) { m_name = name; }

    void SetName(const wxString& name) { m_name = name; }
    const wxString& GetName() const { return m_name; }
private:
    wxString m_name;
};

IMPLEMENT_DYNAMIC_CLASS(MyRecord, wxObject)

MyRecord* CreateMyRecord(const wxString& name)
{
```

```
    MyRecord* rec = wxDynamicCast(
        wxCreateDynamicObject(wxT("MyRecord")), MyRecord);
    if (rec)
        rec->SetName(name);
    return rec;
}
```

Когда код вызывает функцию `CreateMyRecord` с определенным именем, функция `wxCreateDynamicObject` создает требуемый объект, после чего `wxDynamicCast` подтверждает, что действительно был создан объект типа `MyRecordit` или возвращает `NULL` в противном случае. С первого взгляда кажется, что такое создание класса не очень полезно, однако такая возможность полезна, например, при загрузке объектов из сложного файла, содержащего различные типы. Данные класса могут хранится вместе с его именем. При чтении файла мы сначала по имени создаем нужный объект, после чего уже сам объект может прочесть свои данные.

Есть еще ряд доступных вам макросов RTTI.

`CLASSINFO(класс)` возвращает указатель на связанный с классом объект типа `wxClassInfo`. Такой объект можно использовать с `wxObject::IsKindOf` для проверки типа класса:

```
if (obj->IsKindOf(CLASSINFO(MyRecord)))
{
    ...
}
```

Используйте `DECLARE_ABSTRACT_CLASS(класс)` и `IMPLEMENT_ABSTRACT_CLASS(класс, базовый_класс)` для абстрактных классов.

Используйте `DECLARE_CLASS2(класс)` и `IMPLEMENT_CLASS2(класс, базовый_класс1, базовый_класс2)` при наличии двух базовых классов.

Используйте `DECLARE_APP(класс)` и `IMPLEMENT_APP(класс)`, чтобы сделать класс приложения известным wxWidgets.

`wxConstCast(ptr, класс)` — это макрос, разворачивающийся в `const_cast<класс *>(ptr)`, если компилятор поддерживает `const_cast` или в старый синтаксис приведения типов в стиле C, в противном случае.

`wxDynamicCastThis(класс)` — эквивалент для `wxDynamicCast(this, класс)`, но последний может являться источником странных предупреждений на некоторых компиляторах (так как тестирует не указывает ли `this` на `NULL`, что всегда должно быть истиной). Чтобы избежать такие предупреждения рекомендуется использовать `wxDynamicCastThis`.

`wxStaticCast(ptr, класс)` в отладочной версии сначала дополнительно проверяет правильность приведения типов (вы получите `assert` в случае если `wxDynamicCast(ptr, класс) == NULL`), а затем возвращает результат выполнения `static_cast<класс*>(ptr)`.

`wx_const_cast(T, x)` то же самое, что и `const_cast<T>(x)` в случае поддержки данной конструкции компилятором или `(T)x` для старых компиляторов. В отличии от `wxConstCast` данный макрос преобразует к типу `T`, не к `*T`. Кроме того порядок аргументов в точности соответствует конструкции для стандартного приведения типов.

`wx_reinterpret_cast(T, x)` то же самое, что и `reinterpret_cast<T>(x)` в случае поддержки данной конструкции компилятором или `(T)x` для старых компиляторов.

`wx_static_cast(T, x)` то же самое, что и `static_cast<T>(x)` в случае поддержки данной конструкции компилятором или `(T)x` для старых компиляторов. В отличии от `wxStaticCast` не делается ни каких дополнительных проверок, а порядок аргументов в точности совпадает со стандартным `static_cast`, `T` является полным именем типа (`*` автоматически не добавляется).

## 15.6 Использование wxModule

Систем модулей является достаточно простым механизм, позволяющим приложениям (а также частям самой wxWidgets), определять функции инициализации и очистки, которые будут автоматически вызваны библиотекой при запуске и при завершении работы. Это позволяет избавить приложение от необходимости размещать большое количество кода инициализации и очистки в обработчиках `OnInit` и `OnExit`.

Для того чтобы определить новый модуль, унаследуйте класс от `wxModule`, реализуйте методы `OnInit` и `OnExit`, а также добавьте `DECLARE_DYNAMIC_CLASS` и `IMPLEMENT_DYNAMIC_CLASS` к определению и реализации класса. Во время инициализации, wxWidgets автоматически создаст объекты для каждого класса, унаследованного от `wxModule`, а также вызовет метод `OnInit` у всех созданных объектов. При завершении работы, wxWidgets вызовет метод `OnExit` у каждого модуля.

Например:

```
// Модуль выполняющий инициализацию и очистку DDE
class wxDDEModule :: public wxModule
{
DECLARE_DYNAMIC_CLASS(wxDDEModule)
public:
    wxDDEModule() {}
    bool OnInit() { wxDDEInitialize(); return true; };
    void OnExit() { wxDDECleanUp(); };
};

IMPLEMENT_DYNAMIC_CLASS(wxDDEModule, wxModule)
```

## 15.7 Загрузка динамических библиотек

Если вам необходимо использовать функции, находящиеся в динамических библиотеках, то у вас есть возможность для этого использовать специальный класс `wxDynamicLibrary`. Конструктору или методу `Load` необходимо передать имя файла динамической библиотеки, а также специальный флаг `wxDL_VERBATIM`, если вы хотите предотвратить автоматическое добавление соответствующего расширения файла (.dll в системе Windows или .so в Linux). После успешной загрузки библиотеки, можно вызывать функции по имени, используя метод `GetSymbol`. Ниже приведен

пример кода, который загружает и инициализирует библиотеку общих элементов управления в системе Windows:

```
#include "wx/dynlib.h"

INITCOMMONCONTROLSEX icex;
icex.dwSize = sizeof(icex);
icex.dwICC = ICC_DATE_CLASSES;

// Загружаем comctl32.dll
wxDynamicLibrary dllComCtl32(wxT("comctl32.dll"), wxDL_VERBATIM);

// Определяем тип ICCEx_t
typedef BOOL (WINAPI *ICCEx_t)(INITCOMMONCONTROLSEX *);

// Получаем ссылку на InitCommonControlsEx
ICCEx_t pfnInitCommonControlsEx =
    (ICCEx_t) dllComCtrl32.GetSymbol(wxT("InitCommonControlsEx"));

// Вызываем функцию инициализации общих элементов управления
if ( pfnInitCommonControlsEx )
{
    (*pfnInitCommonControlsEx)(&icex);
}
```

Строку с `GetSymbol` можно записать более кратко, используя специальный макрос `wxDYNLIB_FUNCTION`:

```
wxDYNLIB_FUNCTION (ICCEx_t, InitCommonControlsEx, dllComCtl32);
```

`wxDYNLIB_FUNCTION` позволяет указать имя типа всего один раз (в качестве первого параметра), после чего автоматически создаст переменную данного типа, названную по имени функции с помощью добавления префикса `pfn`.

Если библиотека была успешно загружена конструктором класса или методом `Load`, то деструктор автоматически вызовет метод `Unload` для выгрузки библиотеки из памяти. Используйте `Detach` если хотите сохранить дескриптор библиотеки после уничтожения объекта `wxDynamicLibrary`.

## 15.8 Обработка исключений

Библиотека wxWidgets была создана задолго до появления исключений в языке C++, да и в настоящее время собирается на компиляторах с различным уровнем поддержки исключений, поэтому сама библиотека не использует исключения. Однако, вы вполне можете использовать их в коде вашей программы, а библиотека сможет помочь вам в этом.

Существует несколько вариантов использования исключений в программах, написанных с использованием wxWidgets. Во-первых, вы можете просто их не

использовать. Сама библиотека некогда не бросает исключений, поэтому, если вы сами явно не вызываете `throw` в своем коде, то можете вообще не заботится о перехвате и обработке исключений. Это простейший из всех возможных вариантов, но, к сожалению, не лучший с точки зрения обработки всех возможных ошибок.

Другой вариант — использовать исключения только в качестве сигнала о действительно фатальных ошибках. В таком случае вы вероятно не будете восстанавливать состояние и работоспособность программы после их возникновения, а значит вас устроит поведение программы по умолчанию — просто прекратить работу приложения. В противном случае, вы можете переопределить метод `OnUnhandledException` в вашем классе-наследнике от `wxApp` для выполнения задачи по корректному завершению приложения. Учтите, что любая информация о произошедшем исключении теряется к моменту вызова данной функции. Поэтому, если вам необходима эта информация, то вы должны переопределить метод `OnRun` и завернуть вызов кода базового класса в блок `try/catch`. Это позволит вам отловить любое исключение, сгенерированное во время работы главного цикла обработки событий. Для работы с исключениям, которые могут возникнуть во время старта или завершения программы, необходимо добавить блоки `try/catch` в методы `OnInit` и `OnExit`.

И наконец, возможно вы захотите продолжить выполнение программы даже после возникновения определенных исключений. Если эти исключения могут произойти только в обработчиках событий некоторого класса (или в его наследниках), то можно поместить ваш код обработки исключительных ситуаций в метод `ProcessEvent` данного класса. Если это не целесообразно, то можно переопределить метод `wxApp::HandleEvent`, что позволит обработать любые исключения, брошенные любым обработчиком событий.

Чтобы включить поддержку исключений в wxWidgets необходимо установить флаг `wxUSE_EXCEPTIONS` равным `1` перед сборкой библиотеки. Флаг равен `1` по умолчанию, но если вдруг это не так, то нужно отредактировать файл include/wx/msw/setup.h в Windows или запустить configure с ключом --enable-exceptions в Unix. Если вы не планируете использовать исключения в своей программе, то установите этот флаг в `0` или используйте --disable-exceptions в Unix. В результате получится немного более компактная и быстрая версия библиотеки. Также, при использовании Visual C++ и библиотеки собранной с флагом `wxUSE_ON_FATAL_EXCEPTION` равным `1`, в случае необработанного исключения будет вызвана переопределенная вами функция `wxApp::OnFatalException` вместо показа стандартного диалога Windows об исключении (General protection fault window). И наоборот, если вы предпочитаете стандартное поведение и сброс программы в отладчик, то установите этот флаг равным `0`.

Примеры использования исключений в wxWidgets можно найти в папке samples/except в стандартной поставке библиотеки.

## 15.9 Советы по отладке приложения

Защитное программирование, сообщения об ошибка и другие техники написания качественного кода не помогут вам полностью избавится от необходимости использовать отладчик. Отладчик позволяет пройти весь код по шагам, просмотреть

значение переменных и в точности узнать где код начинает вести себя неправильно или падает. Итак, вам необходимо иметь как минимум две версии вашего проекта — отладочную и релизную. Отладочная версия содержит большее количество проверок, не использует оптимизацию компилятора и включает в себя информацию необходимую для отладки — имя исходного файла, текущую строку в этом файле и так далее. Символ препроцессора `__WXDEBUG__` всегда определен в отладочной версии программы, поэтому вы всегда можете проверить его, чтобы узнать в каком режиме работает программа. Но некоторые функции, такие как `wxLogDebug`, всегда будут удалены из релизной версии программы, таким образом вам понадобится реже проверять значение этого символа в вашей программе.

Удивительно большое количество программистов пытаются работать не используя отладчик, но усилия потраченные на изучение доступных вам средств в итоге полностью окупаются. В Windows среда Visual C++ включает в себя очень хороший отладчик. Если вы используете GCC (в Windows или в Unix), то вынужденны использовать базовый отладчик GDB, используя его через консоль или редактор. Но вы всегда вы можете выбрать одну из многих IDE, предоставляющую более дружественный интерфейс к GDB. Информацию о них, вы найдете в Дополнении E «Сторонние утилиты для wxWidgets».

Библиотека wxWidgets позволяет одновременное использование нескольких конфигураций. В Windows вы можете передать флаги `BUILD=debug` или `BUILD=release` в makefile или, если вы используете утилиту configure, задать параметры --enable-debug или --disable-debug. Некоторые IDE не поддерживают одновременную работу с несколькими конфигурациями сборки. В таком случае необходимо менять настройки и перекомпилировать в одном режиме, а потом опять менять настройки для того, чтобы скомпилировать программу в другой конфигурации. Избегайте использования подобных IDE.

### 15.9.1 Поиск ошибок в оконной системе X11

В очень редких случаях ваше wxGTK приложение может падать с ошибкой в оконной системе X11. При этом программа будет немедленно завершена, не давай вам возможность увидеть трассировку стека, что весьма затрудняет поиск подобных ошибок. Для решения данной проблемы необходимо установить свой собственный обработчик ошибок, как показано в следующем примере.

```
#if defined(__WXGTK__)
#include <X11/Xlib.h>

typedef int (*XErrorHandlerFunc)(Display *, XErrorEvent *);

XErrorHandlerFunc gs_pfnXErrorHandler = 0;

int wxXErrorHandler(Display *display, XErrorEvent *error)
{
    if (error->error_code)
    {
          char buf[64];
```

```
          XGetErrorText (display, error->error_code, buf, 63);

          printf (
           "** X11 error in wxWidgets for GTK+: \
%s\n  serial %ld error_code %d request_code %d minor_code %d\n",
           buf,
           error->serial,
           error->error_code,
           error->request_code,
                   error->minor_code);
    }

    // Активируйте следующий участок кода, чтобы передать
    // управление обработчику ошибок по умолчанию
#if 0
    if (gs_pfnXErrorHandler)
        return gs_pfnXErrorHandler(display, error);
#endif
    return 0;
}
#endif
  // __WXGTK__

bool MyApp::OnInit(void)
{
#if defined(__WXGTK__)
    // устанавливаем обработчик ошибок для X
    gs_pfnXErrorHandler = XSetErrorHandler( wxXErrorHandler );
#endif
    ...
    return true;
}
```

Теперь приложение вызовет ошибку памяти сразу после возникновения ошибки в подсистеме X11, поэтому, если передать опцию -sync приложению во время запуска, сбой должен произойти достаточно близко к месту, где передается некорректное значение в одну из функций X11.

### 15.9.2 Упростите проблему

Если вам встретилась ошибка, которую вы никак не можете найти и устранить, то разумной стратегией будет попробовать воспроизвести ошибку в наименьшем возможном приложении. Можно взять за основу примеры из стандартной поставки wxWidgets и добавить код, демонстрирующий проблему. Или можно скопировать исходные тексты вашей программы и последовательно удалять лишний код до того минимума на котором проблема все еще воспроизводится. Если вы уверены, что

проблема в wxWidgets, то добавление маленькой части кода к стандартному примеру поможет вам или разработчикам wxWidgets воспроизвести и устранить проблему.

### 15.9.3 Отладка релизной версии

Время от времени случается так, что ваше приложение работает в отладочной версии, но не работает в релизной. Такое может произойти, например, в следствии незначительных отличий версий библиотек времени исполнения, используемых компилятором. Если вы используете Visual C++, то можете создать новую конфигурацию проекта, идентичную отладочной, но с определенным символом `NDEBUG`. Таким образом ваше приложение и библиотека будет содержать отладочную информацию, но использоваться будет релизная версия библиотек времени исполнения. Это как минимум позволит устранить возможную разницу в версиях библиотек времени выполнения.

Обычно, конечные пользователи получают релизную версию вашего приложения, не содержащую отладочную информацию. Но если у пользователей случаются проблемы, которые вы не можете воспроизвести на своей машине, то можно послать им отладочную версию (для Windows вы должны использовать статическую линковку с библиотеками времени выполнения, чтобы избежать проблем с незаконным распространением отладочных DLL). Также можно собрать приложение с отладочными символами для системной программы Dr. Watson, запущенной на компьютерах ваших пользователей и предназначенной для сбора информации об ошибках программ. Обратитесь к документации вашего компилятора, чтобы узнать каким образом можно использовать файлы, создаваемые программой Dr. Watson в случае ошибок или падения вашего приложения.

Если вы используете MinGW, то можно воспользоваться утилиту Dr. MinGW, которая может отлавливать исключения в вашей программе и выдавать удобную трассировку в случае, если код приложения содержит отладочную информацию (опция -g). Данную утилиту можно найти на сайте http://www.mingw.org. Если у вас терпеливые и отзывчивые пользователи, то вы можете попросить их установить данную утилиту, а в случае падения вашего основного приложения — послать вам созданный отчет.

В системе Unix выполняемый файл с отладочной информацией при падении создает файл с crash-дампом (поведение зависит от настроек операционной системы пользователя — за дополнительной информацией обратитесь к документации по команде ulimit). Вы можете использовать полученный crash-дамп вместе с оригинальным файлом, содержащим отладочную информацию, для того, чтобы посмотреть в отладчике в каком конкретно месте вашего кода произошел сбой. Однако, файл crash-дампа может быть весьма большим, поэтому не все пользователи будут готовы отправить его вам.

Кроме того, ваше приложение может записывать в файл с логом информацию о ключевых моментах работы программы. Также обратите внимание на документацию wxWidgets по классу `wxDebugReport`, который умеет создавать отчеты о падении программы, пригодные к отправке по электронной почте. Похожая функциональность реализована в классе `wxCrashPrint` с http://wxcode.sf.net (для Linux) и `BlackBox` с http://www.codeproject.com/tools/blackbox.asp (для Windows).

## 15.10 Итоги

В данной главе мы раскрыли различные аспекты управления памятью и проверки ошибок. Теперь вы знаете, когда стоит использовать оператор `new` или создавать объекты на стеке, как ваше приложение может корректно завершить работу, как обнаружить утечки памяти и как использовать макросы в парадигме «защитного программирования». Теперь вы должны понимать как выбрать между использованием классов `wxLogDebug` и `wxLogError`. Вы научились использовать исключения языка C++ в программах на wxWidgets, а также получили несколько советов по отладке приложений. В следующем главе рассказывается как научить ваше приложение говорить на нескольких языках.