Глава 13

Структуры данных

©Перевод сделан Тюрюмовым А.Н.

Хранение и обработка информации является важнейшей частью любого приложения. wxWidgets предоставляет широкий выбор структур данных, начиная от самых простых, хранящих информацию о размере и расположении, и, заканчивая сложными, такими как массивы и ассоциативные массивы. В этой главе мы рассмотрим различные полезные структуры данных, сосредоточившись на их часто используемых методах. Об остальных методах и особенностях работы с этими структурами можно прочитать в документации.

Заметим, что описание теории структур данных и их реализации выходит за рамки данной книги. Однако общие возможности описываемых структур данных должны быть понятны даже без понимания их реализации.

13.1 Почему не STL?

Для начала ответим на наиболее часто задаваемый вопрос о классах для данных в wxWidgets: «Почему wxWidgets не использует стандартную библиотеку классов (STL)?». Главной причиной является историческая — wxWidgets появилась в 1992 году, задолго до того как реализация стандартной библиотеки классов была перенесена на различные платформы и компиляторы. По мере развития wxWidgets многие внутренние структуры стали поддерживать методы похожие на STL API. Ожидается, что в один прекрасный момент эти структуры будут просто заменены на их STL-эквиваленты.

Несмотря на это обстоятельство, вы можете использовать функциональность STL в ваших wxWidgets-приложениях уже сейчас. Для этого установите в файле setup.h значение переменной `wxUSE_STL` равным 1 (или используете параметр `enable-stl` на стадии конфигурирования перед сборкой библиотеки). Если вы это сделаете, то классы wxString и многие контейнеры будут построены на базе классов из стандартной библиотеки. Обратите внимание, что использование wxWidgets совместно с STL может увеличить размер и время компиляции приложения, особенно при использовании компилятора GCC.

13.2 Строки

Преимущества от использования строк на базе классов хорошо известны. wxWidgets использует свой собственный строковый класс `wxString`, который используется как внутри библиотеки, так и для передачи параметров и возвращения результата. `wxString` поддерживает почти все стандартные операции, которые вы ожидаете найти в строковом классе: динамическое управление памятью, создание из C строк и символов, операторы присваивания, доступ к одиночному символу, объединение и сравнение строк, извлечение подстрок, преобразование регистра, удаление символов, операции поиска и замены, аналоги функции `printf` и т.д.

Несмотря на то, что `wxString` является еще одним строковым классом он поддерживает некоторые дополнительные полезные возможности. `wxString` полностью поддерживает юникод, а также включает методы по преобразованию строки в различные кодировки.

Использование `wxString` дает возможность передавать строки в библиотеку или получать обратно без дополнительного процесса преобразования. Кроме того `wxString` реализует 90% методов стандартного класса `std::string` и значит каждый знакомый с `std::string` может сразу без дополнительного обучения использовать `wxString`.

13.2.1 Использование wxString

Использовать класс `wxString` очень легко. Везде, где вы бы обычно использовали бы `std::string` или вашу любимую реализацию строк, вы должны использовать `wxString`. Все функции, получающие строковые аргументы, должны использовать `const wxString&` (которая делает передачу строк внутрь функции быстрее, так как `wxString` использует механизм подсчета ссылок), а все функции, возвращающие строки, должны возвращать `wxString`, что позволяет безопасно получать из функции даже локальные переменные.

Так как программисты на C и C++ обычно хорошо знакомы с большинством строковых методов, то мы не будем помещать здесь большого и детального описания строкового API. За данной информацией обратитесь к документации wxWidgets, которая содержит впечатляющий список доступных методов.

Обратите внимание, что иногда `wxString` содержит два или более метода с одинаковой функциональностью. Например, `Length`, `Len` и `length` возвращают длину строки. Во всех подобных случаях рекомендуется использовать `std::string`-совместимые методы. Это сделает ваш код более понятным для других C++ программистов и позволит использовать его в других программах простой заменой `wxString` на `std::string`. Возможно в будущем wxWidgets начнет использовать `std::string`, поэтому применение таких методов позволит сделать ваши программы более совместимыми со следующими версиями (хотя старые методы wxString будут поддерживаться некоторое время в целях обратной совместимости).

13.2.2 wxString, символы и символьные литералы

wxWidgets использует тип `wxChar`, который является синонимом типов `char` или `wchar_t`, в зависимости от типа сборки (ANSI или юникод). Как уже упоминалось

ранее, нет нужды разделять тип для строк `char` или `wchar_t`, так как `wxString` сохраняет строку, используя подходящий тип. Вот почему работая напрямую со строками `wxString`, желательно использовать тип `wxChar` вместо `char` или `wchar_t`. Следование данной рекомендации дает гарантии, что ваш код будет работать в ANSI и юникодных сборках без использования запутанных директив препроцессора.

Если вы используете wxWidgets в юникодном режиме, то стандартные строковые литералы имеют неправильный тип (`char*`). Чтобы иметь возможность использовать литералы в юникодном режиме их необходимо преобразовать в константы с широкими символами, что обычно делается с помощью специальной директивы `L`, которая указывается перед литералом. wxWidgets вводит макрос `wxT` (являющийся синонимом `_T`), чтобы получать правильный литерал вне зависимости от текущего режима сборки. Если юникодный режим не используется, то `_T` преобразуется в пустой макрос, но при сборке в юникодном режиме добавляется необходимый символ `L` для преобразования литерала в литерал с широкими символами. Например:

```
wxChar ch = wxT('*');
wxString s = wxT("Hello, world!");
wxChar* pChar = wxT("My string");
wxString s2 = pChar;
```

За более детальной информацией об использовании юникода в ваших приложениях, обратитесь к главе 16 «Написание локализованных приложений».

13.2.3 Преобразование wxString в указатель языка C

Иногда необходимо получить доступ к данным wxString для низкоуровневой обработки на языке C. Для этого в wxWidgets есть несколько методов:

- `mb_str` возвращает строку языка C с типом `const char*`. В юникодном режиме для этого происходит соответствующее преобразование, поэтому могут быть потеряны некоторое данные.
- `wc_str` возвращает широкосимвольное представление строки с типом `wchar_t*`. В ANSI-режиме строка для этого преобразуется в юникод.
- `c_str` возвращает указатель на реальные данные строки (типа `const char*` в режиме ANSI и `const wchar_t*` в юникодном режиме). Никакого преобразования при этом не происходит.

Вы можете производить преобразование между `std::string` и `wxString`, используя метод `c_str` как показано ниже:

```
std::string str1 = wxT("hello");
wxString str2 = str1.c_str();
std::string str3 = str2.c_str();
```

Одной из типичных ошибок при использовании `wxString` является надежда на неявное преобразование `wxString` в `const char *`. Советуем вам всегда использовать метод `c_str`, чтобы явно указать библиотеке, что необходимо выполнить преобразование. Опасность неявного преобразования демонстрируется в следующем примере:

```
// преобразовать входную строку к верхнему регистру, вывести результат
// на экран и возвратить результат (код неверный!)
const char *SayHELLO(const wxString& input)
{
    wxString output = input.Upper();
    printf("Hello, %s!\n", output);
    return output;
}
```

В этих четырех строках содержится две опасных ошибки. Первая происходит во время вызова оператора `printf`. Неявное преобразование `wxString` в строки языка C производится компилятором в случае функции типа puts, так как известно, что аргументом этой функции является значение типа `const char *`. Однако это не верно в случае функции с переменным числом аргументов, когда тип аргументов не известен. Так и происходит в нашем примере, а в результате во время вызова `printf` может произойти все что угодно (включая правильное отображение строки на экране), хотя в большинстве случаев программа аварийно завершит свою работу. Правильным решением является использование `c_str`:

```
printf(wxT("Hello, %s!\n"), output.c_str());
```

Вторая ошибка связана с тем, что из функции возвращается значение переменной `output`. Здесь будет использовано неявное преобразование, поэтому данный код скомпилируется, однако он возвращает указатель на буфер, принадлежащий локальной переменной, а она будет удалена при выходе из функции. Эта проблема решается просто: сделайте так, чтобы функция возвращала `wxString` вместо строк языка C. Конечная версия работающего кода выглядит следующим образом:

```
// преобразовать входную строку к верхнему регистру, вывести результат
// на экран и возвратить результат (исправленный код)
wxString SayHELLO(const wxString& input)
{
    wxString output = input.Upper();
    printf(wxT("Hello, %s!\n"), output.c_str());
    return output;
}
```

13.2.4 Стандартные строковые функции языка C

Т.к. большинство программ активно использует символьные строки, то стандартная библиотека языка C содержит множество функций для работы с ними. Однако не все из них обладают интуитивным поведением (например функция `strncpy` не всегда ставит завершающий ноль в результирующую строку) или являются безопасными с точки зрения возможности переполнения буфера. Более того, некоторые полезные функции не являются частью стандарта. Вот почему в дополнение к обычным функциям класса `wxString` существует несколько глобальных функций: `wxIsEmpty` проверяет является ли строка пустой (возвращает `true` для указателя `NULL`), `wxStrlen` корректно обрабатывает `NULL` и возвращает для него 0 и `wxStricmp`, которая является

платформонезависимой чувствительной к регистру функцией сравнения, аналогичная функциям `stricmp` или `strcasecmp`.

Заголовочный файл "wx/string.h" также определяет функции `wxSnprintf` и `wxVsnprintf`, которые могут быть использованы вместо потенциально опасной стандартной функции `sprintf`. Данные функции используют `snprintf`, которая в большинстве случаев проверяет размеры буфера. Вы также можете использовать `wxString::Printf`, не беспокоясь об уязвимостях стандартной функции `printf`.

13.2.5 Операции преобразования числел

Программисту часто требуется преобразовывать число в строку и обратно, особенно при обработке пользовательского ввода или отображении результатов.

`ToLong(long* val, int base=10)` пытается преобразовать строку в число со знаком в указанной системе счисления. Метод возвращает `true` в случае успеха (результат операции будет записан в переменную на которую указывает `val`), или `false`, если строка не представляется в виде корректного числа в выбранной системе счисления `base`. Основание системы счисления (`base`) может принимать значения от 2 до 36 включительно или 0, которое означает использование стандартных правил языка C: если число начинается с 0x, предполагается, что оно шестнадцатиричное; если начинается с 0, то считается восьмеричным, и десятичным во всех остальных случаях.

`ToULong(unsigned long* val, int base=10)` работает аналогично `ToLong`, но преобразовывает строку в беззнаковое число.

`ToDouble(double* val)` пытается преобразовать строку в число с плавающей точкой. Возвращается `true` в случае успеха (результат будет записан в переменную на которую указывает `val`) или `false`, если строка не является таким числом.

`Printf(const wxChar* pszFormat, ...)` похож на стандартную функцию языка C `sprintf` и позволяет поместить информацию в `wxString`, используя стандартное форматирование для строк языка C. Метод возвращает число записаных байт.

`static Format(const wxChar* pszFormat, ...)` возвращает строку `wxString`, содержащую результат вызова `Printf` с параметрами. Преимуществом `Format` над `Printf` является возможность добавления результата `Format` к существующей строке:

```
int n = 10;
wxString s = "Some Stuff";
s += wxString::Format(wxT("%d"),n );
```

`operator«` может использоваться для добавления значений типа `int`, `float` или `double` к строке `wxString`.

13.2.6 wxStringTokenizer

`wxStringTokenizer` поможет вам разбить строку на несколько лексем, заменяя и расширяя возможности функции `strtok` языка C. Для этого необходимо создать объект класса `wxStringTokenizer`, передать ему требующую разбиения строку и разделитель лексем (по умолчанию, в качестве символа-разделителя лексем используется пробел). Далее необходимо вызывать функцию `GetNextToken` (получить следующую лексему) до тех пор, пока `HasMoreTokens` (есть ли лексемы) не возвратит `false`.

```
wxStringTokenizer tkz(wxT("first:second:third:fourth"), wxT(":"));
while (tkz.HasMoreTokens() ) {
    wxString token = tkz.GetNextToken();
    // здесь идет обработка лексемы
}
```

По умолчанию, когда символом-разделителем является пробел, `wxStringTokenizer` ведет себя аналогично функции `strtok`. Однако класс также возвратит пустые лексемы (если такие будут). Эта возможность полезна при разборе строго форматированных данных, когда есть фиксировано число полей, но некоторые из них могут быть пустыми, как например в случае файла с запятыми или символами табуляции в качестве разделителей.

Поведение `wxStringTokenizer` зависит от последнего параметра конструктора, который может принимать одно из следующих значений:

- `wxTOKEN_DEFAULT`: Используется поведение, описанное ранее: аналогично `wxTOKEN_STRTOK` в случае, когда строка с разделителями состоит только из пробелов, и как с `wxTOKEN_RET_EMPTY` иначе.
- `wxTOKEN_RET_EMPTY`: В этом режиме возвращаются пустые лексемы в середине строки. Поэтому строка a::b: разобьется на три лексемы: a, «» (пустая строка) и b.
- `wxTOKEN_RET_EMPTY_ALL`: В этом режиме также возвращаются пустые завершающие лексемы (после последнего символа-разделителя). Следовательно, a::b: будет разбит на четыре лексемы — три такие как в случае `wxTOKEN_RET_EMPTY` и пустая лексема в конце.
- `wxTOKEN_RET_DELIMS`: В этом режиме разделитель в конце лексемы присоединяется к самой лексеме (кроме последней лексемы, у которой такой символ отсутствует). В противном случае поведение похоже на `wxTOKEN_RET_EMPTY`.
- `wxTOKEN_STRTOK`: В этом случае класс ведет себя также как и стандартная функция `strtok`. Пустые лексемы никогда не возвращаются.

Класс `wxStringTokenizer` имеет две полезные функции:

- `CountTokens` возвращает число оставшихся лексем в строке, и 0, когда лексем уже не осталось.
- `GetPosition` возвращает текущую позицию разборщика в оригинальной строке.

13.2.7 wxRegEx

Класс `wxRegEx` можно использовать для работы с регулярными выражениями. Он поддерживает регулярные выражения для поиска и замены в строках. Класс `wxRegEx` может использовать системную библиотеку (если она доступна и имеет поддержку регулярных выражений POSIX, как в случае современных вариантов Unix, включая Linux и Mac OS X) или встроенную библиотеку Генри Спенсера (Henry Spencer). Регулярные выражения, удовлетворяющие стандарту POSIX, делятся на

два множества: расширенные (extended) и простые (basic). Встроенная библиотека также вводит третий режим (продвинутый), который не доступен при использовании системной библиотеки.

На платформах, где доступна системная библиотека, она как правило используется в обычной сборке. В юникодной сборке обычно используется встроенная библиотека. Однако используемую библиотеку возможно изменить при сборке wxWidgets. При использовании системной библиотеки, в юникодной сборке выражения и данные переводятся в 8-битовую кодировку перед тем как передаются библиотеке.

Используйте `wxRegEx` так, как если бы вы использовали любой другой POSIX-совместимый процессор регулярных выражений. За описанием продвинутых возможностей класса и специфики его использования, обратитесь к документации библиотеки и описанию API.

13.3 wxArray

wxWidgets предлагает структуру для реализации динамических массивов на базе класса `wxArray`, который похож на стандартные массивы языка C, где доступ к элементам осуществляется за постоянное время. Однако, эти массивы динамические, а значит могут самостоятельно выделять память в случае недостаточной текущей вместимости для добавления нового элемента. Добавление элемента обычно осуществляется за константное время, но для этого массиву приходится часть памяти всегда держать в резерве. `wxArray` осуществляет проверку на выход за границы, что в случае ошибки приводит к возникновению утверждения (assert) в отладочной сборке или возвращению произвольного значения в финальной сборке (в этом случае ваша программа может получить произвольное значение в операциях с массивами).

13.3.1 Типы массивов

В wxWidgets существует три типа массивов. Все эти типы являются наследниками от `wxBaseArray`, который работает с нетипизированными данными, а поэтому не может быть использован непосредственно. Макросы `WX_DEFINE_ARRAY`, `WX_DEFINE_SORTED_ARRAY` и `WX_DEFINE_OBJARRAY` используются для создания новых классов-наследников от него. На такие классы обычно ссылаются по именам `wxArray`, `wxSortedArray` и `wxObjArray`, но важно понимать, что классов с такими именами не существует.

`wxArray` подходит для хранения простых типов и указателей, которые не считаются полноценными объектами. Если в массиве хранятся указатели на объекты, то они автоматически не уничтожаются при удалении указателя из массива. Кроме того заметим, что все функции массива объявлены как подставляемые (`inline`), поэтому без последствий (как в скорости выполнения, так и в размере выполняемого файла) можно объявлять очень много различных типов для массивов. У класса есть одно серьезное ограничение: он может использоваться только для хранения простых типов (`bool`, `char`, `short`, `int`, `long` и их беззнаковых вариантов) или указателей (на любой тип). Данные типа `float` или `double` не могут хранится в `wxArray`.

`wxSortedArray` — это вариант класса `wxArray`, который необходимо использовать, когда операция поиска по массиву выполняется очень часто. При использовании

wxSortedArray необходимо дополнительно объявить функцию для сравнения двух элементов массива. Элементы в массиве сортируются в соответствии с этой функцией. Если вы ищите в массиве во много раз чаще, чем добавляете в него новую информацию, то wxSortedArray может дать огромный прирост в скорости по сравнению с wxArray. Заметим, что на типы, хранимые в wxSortedArray действуют те же ограничения, что и на типы в классе wxArray. Таким образом, в данных массивах можно хранить только простые типы данных или указатели.

Класс wxObjArray трактует элементы массива как объекты. Он может удалять эти элементы, когда они удаляются из массива, корректно вызывая при этом деструктор. Он также копирует объекты, используя конструктор копирования самого объекта. Определение wxObjArray состоит из двух частей. Во-первых, вы должны объявить новый класс wxObjArray, используя макрос WX_DECLARE_OBJARRAY. Во-вторых, вы должны подключить файл <wx/arrimpl.cpp>, который определяет реализацию шаблона, и определить реализацию вашего класса с помощью макроса WX_DEFINE_OBJARRAY в точке, где уже полностью определен класс элементов, содержащихся в массиве. Эта технология будет продемонстрирована на примере чуть позднее.

13.3.2 wxArrayString

wxArrayString — это эффективный контейнер, предназначенный для хранения объектов типа wxString и имеет те же возможности, что и wxArray. Он также очень маленький и занимает в памяти не намного больше места, чем стандартный массив языка C типа wxString[] (wxArrayString использует знание о внутренней структуре класса wxString, чтобы достичь этого). Все методы, доступные в классе wxArray также доступны и в wxArrayString.

Этот класс используется аналогично остальным динамическим массивам, с той лишь разницей, что не нужно его объявлять с помощью WX_DEFINE_ARRAY, и вы можете использовать тип wxArrayString непосредственно. Когда строка добавляется или вставляется в массив, создается копия строки, а исходная строка может быть безопасно удалена. Таким образом не нужно беспокоится об управлении памятью при использовании данного класса, так как он всегда освобождает память, которую использует.

Ссылка, возвращаемая методами Item, Last или operator[], не является константной, а поэтому вы можете сразу модифицировать объект:

```
array.Last().MakeUpper();
```

Существует также сортированный вариант wxArrayString, который называется wxSortedArrayString. Он поддерживает те же самые методы, но содержит все текстовые строки, отсортированные в алфавитном порядке. wxSortedArrayString использует бинарный поиск при выполнеии метода Index, что очень эффективно, если вы редко добавляете строки в массив и часто осуществляете по ним поиск. Методы Insert и Sort не должны вызываться у wxSortedArrayString, так как это может нарушить порядок элементов в массиве.

13.3.3 Создание и уничтожение массивов и управление памятью

Классы массивов являются полноценными объектами C++ и реализуют конструктор копирования и оператор присваивания. Копирование `wxArray` копирует все его содержимое, а копирование `wxObjArray` вызывает конструктор копирования для всех его элементов. Однако в целях повышения эффективности работы с памятью, ни один из этих классов не имеет виртуального деструктора. Это не очень важно для `wxArray` (который имеет тривиальный деструктор), но это означает, что вы должны избегать удаление `wxObjArray` через указатель на `wxBaseArray` и не наследовать ваш собственный класс от этих классов-массивов.

Автоматическое управление памятью в массиве элементарно: массив начинает работу с предварительного выделения некоторого количества памяти (определяемого `WX_ARRAY_DEFAULT_INITIAL_SIZE`); когда при добавлении элементов требуемая память превысит выделенное значение, массив делает перераспределение, увеличивая вместимость массива на 50% от его текущего объема, но не более `ARRAY_MAXSIZE_INCREMENT`. Метод `Shrink` освобождает любую незанятую память. Метод `Alloc` может быть очень полезен, если вы точно знаете, сколько элементов вы разместите в массиве. Используя этот метод вы зарезервировать память для заданного числа элементов, что позволит избежать слишком частого перераспределения памяти.

13.3.4 Пример использования массивов

Следующий пример показывает все важные моменты использования `wxObjArray` для хранения данных пользовательского типа. Использование `wxArray` при хранении указателей работает точно также в терминах синтаксиса и семантики, за исключением того, что `wxArray` никогда не берет во владение хранимые объекты, а потому они должны уничтожаться пользователем.

```
// Наши данные, которые мы сохраняем в списке (класс покупателей)
class Customer
{
public:
    int CustID;
    wxString CustName;
};

// эта часть может быть в файле объявления или реализации (.cpp)
// объявляем наш класс для массива:
// этот макрос объявляет и частично реализует класс CustomerArray
// (который является наследником от wxArrayBase)
WX_DECLARE_OBJARRAY(Customer, CustomerArray);

// главным требованием является помещение данной директивы ПОСЛЕ
// полного объявления класса Покупатель (для WX_DECLARE_OBJARRAY
// неполной (forward) декларации вполне достаточно), но обычно
```

```
// данный макрос помещают в файле с реализацией, а не в заголовочный файл
#include <wx/arrimpl.cpp>
WX_DEFINE_OBJARRAY(CustomerArray);

// Используется для сортировки при сравнении объектов
int arraycompare(Customer** arg1, Customer** arg2)
{
    return ((*arg1)->CustID < (*arg2)->CustID);
}

// Демонстрация возможностей массива
void ArrayTest()
{
    // Создаем экземпляр нашего массива
    CustomerArray MyArray;

    bool IsEmpty = MyArray.IsEmpty(); // должно быть true

    // Создаем несколько покупателей
    Customer CustA;
    CustA.CustID = 10;
    CustA.CustName = wxT("Bob");

    Customer CustB;
    CustB.CustID = 20;
    CustB.CustName = wxT("Sally");

    Customer* CustC = new Customer();
    CustC->CustID = 5;
    CustC->CustName = wxT("Dmitri");

    // Добавляем двух покупателей в массив
    MyArray.Add(CustA);
    MyArray.Add(CustB);

    // Добавляем последнего покупателя в заданное место массива
    // Массив не владеет объектом CustC, он владеет только его копией
    MyArray.Insert(*CustC, (size_t)0);

    int Count = MyArray.GetCount(); // должно возвратить 3

    // Если объект не найден, то метод возвратит wxNOT_FOUND
    int Index = MyArray.Index(CustB); // должно возвратить 2

    // Process each customer in the array
    // Обрабатываем каждого покупателя в архиве
    for (unsigned int i = 0; i < MyArray.GetCount(); i++)
```

```
    {
        Customer Cust = MyArray[i]; // or MyArray.Item(i);

        // Обрабатываем покупателя
    }

    // Сортируем покупателей с помощью функции сортировки
    MyArray.Sort(arraycompare);

    // Удаляем Покупателя A из массива, но не удаляем его
    // Remove Customer A from the array, but do not delete
    Customer* pCustA = MyArray.Detach(1);
    // Мы должны удалять объект сами, если используем Detach
    delete pCustA;

    // метод Remove также удаляет и хранимый объект
    MyArray.RemoveAt(1);

    // Очищаем массив, удаляя все объекты
    MyArray.Clear();

    // Массив никогда не владел этим объектом
    delete CustC;
}
```

13.4 wxList и wxNode

Класс `wxList` представляет двусвязный список, который может хранить данные произвольного типа. wxWidgets требует, чтобы вы явно определили новый тип списка для каждого отдельного типа данных, что позволяет библиотеке проводить строгую проверку типов для хранимых в списке значений. Класс `wxList` также позволяет дополнительно определить тип значений для ключа, с помощью которого можно осуществить примитивный поиск (обратитесь к разделу о типе `wxHashMap`, если вам необходима структура с быстрым доступом к произвольному месту).

`wxList` использует абстрактный класс `wxNode` (узел списка). Если вы объявляете новый тип списка, то также создается тип для узлов, являющихся наследником от `wxNodeBase` — он позволяет осуществить строгую проверку типа. Самые важные методы класса: `GetNext`, `GetPrevious` и `GetData` возвращают следующий и предыдущий узел, а также данные текущего узла соответственно.

Заметим, что удаление из списка работает не совсем так, как от нее ожидается. По умолчанию, при удалении узла не удаляются данные, которые в нем содержатся. Используя метод `DeleteContents`, можно изменить это поведение - список удалит данные вместе с узлом. Если вам требуется полностью очистить список и все данные в нем, то не забудьте вызывать `DeleteContents` с параметром `true` перед вызовом `Clear`.

Для иллюстрации возможностей класса, мы приведем маленький, но всесторон-

ний пример кода, который дополнительно продемонстрирует процедуру создания нового типа списка. Заметим, что макрос `WX_DECLARE_LIST` обычно располагается в заголовочном файле, тогда как макрос `WX_DEFINE_LIST` должен почти всегда располагаться в файле с реализацией.

```
// Наши данные, которые мы сохраняем в списке (класс покупателей)
class Customer
{
public:
    int CustID;
    wxString CustName;
};

// Эта часть должна быть в заголовочном файле или в файле с реализацией
// объявляем тип для списка:
// этот макрос объявляет и частично реализует класс CustomerList
// (который является наследником от wxListBase)
WX_DECLARE_LIST(Customer, CustomerList);

// главным требованием является помещение данной дерективы ПОСЛЕ
// полного объявления класса Покупатель (для WX_DECLARE_LIST
// неполной (forward) декларации вполне достаточно), но обычно
// данный макрос помещают в файле с реализацией, а не в заголовочный файл
#include <wx/listimpl.cpp>
WX_DEFINE_LIST(CustomerList);

// Используется для сортировки при сравнении объектов
int listcompare(const Customer** arg1, const Customer** arg2)
{
    return ((*arg1)->CustID < (*arg2)->CustID);
}

// Демонстрация возможностей списка
void ListTest()
{
    // Создаем экзмпляр нашего списка
    CustomerList* MyList = new CustomerList();

    bool IsEmpty = MyList->IsEmpty(); // должно быть true

    // Создаем несколько покупателей
    Customer* CustA = new Customer;
    CustA->CustID = 10;
    CustA->CustName = wxT("Bob");

    Customer* CustB = new Customer;
    CustB->CustID = 20;
```

```
CustB->CustName = wxT("Sally");

Customer* CustC = new Customer;
CustC->CustID = 5;
CustC->CustName = wxT("Dmitri");

// Добавляем двух покупателей в список
MyList->Append(CustA);
MyList->Append(CustB);

// Добавляем последнего покупателя в заданное место списка
MyList->Insert((size_t)0, CustC);

int Count = MyList->GetCount(); // должно возвратить 3

// Если объект не найден, то метод возвратит wxNOT_FOUND
int index = MyList->IndexOf(CustB); // должно возвратить 2

// Вместе с классом создается тип для узла, который
// содержит объект, определенного при создании типа
CustomerList::Node* node = MyList->GetFirst();

// Обходим узлы и обрабатываем покупателей
while (node)
{
    Customer* Cust = node->GetData();

    // Обрабатываем покупателя

    node = node->GetNext();
}

// Возвращает узел на определенном месте
node = MyList->Item(0);

// Сортируем покупателей с помощью функции сортировки
MyList->Sort(listcompare);

// Удаляем узел Покупателя A из списка
MyList->DeleteObject(CustA);
// Объект CustA НЕ удален при удалении узла
delete CustA;

// Возвращаем узел, который содержит данные клиента
node = MyList->Find(CustB);

// Устанавливаем, что надо удалять данные при удалении узла
```

```
    MyList->DeleteContents(true);

    // Удаляем узел и данные которые он содержит (CustB)
    MyList->DeleteNode(node);

    // Удаляем список и все связанные с ним данные
    // (DeleteContents установлен в true)
    MyList->Clear();

    delete MyList;
}
```

13.5 wxHashMap

`wxHashMap` — это простой, типизированный и достаточно эффективный класс для создания отображений (hash maps), интерфейс которого отчасти повторяет интерфейс соответствующего STL-контейнера. В частности, он моделирует часть поведения стандартного класса `std::map` и нестандартного `std::hash_map`. Используя специальные макросы для создания хэш-карт, вы можете выбирать тип для ключей и значений, включая `int`, `wxString` или `void*` (для хранения произвольных классов).

Существует три макроса для объявления отображений. Объявление отображения `CLASSNAME`, типом `wxString`, используемым в качестве ключа, и типом `VALUE_T` для значений, выглядит следующим образом:

```
WX_DECLARE_STRING_HASH_MAP(VALUE_T, CLASSNAME);
```

Объявление отображения `CLASSNAME`, с ключами `void*` и значениями типа `VALUE_T`:

```
WX_DECLARE_VOIDPTR_HASH_MAP(VALUE_T, CLASSNAME);
```

Объявление отображения `CLASSNAME` с заданным ключем и значением в общем случае имеет вид

```
WX_DECLARE_HASH_MAP(KEY_T, VALUE_T, HASH_T, KEY_EQ_T, CLASSNAME);
```

где `HASH_T` и `KEY_EQ_T` — типы, необходимые для хэширования и сравнения ключей. В wxWidgets реализовано три предопределенных хэширующих функции: `wxInteger` — для целочисленных типов (`int`, `long`, `short` и их беззнаковых версий), `wxStringHash` — для строк (`wxString`, `wxChar*`, `char*`) и `wxPointerHash` для указателей на любые типы. Аналогично определены три предиката равенства: `wxIntegerEqual`, `wxStringEqual` и `wxPointerEqual`.

Следующий пример показывает использование методов класса `wxHashMap`, а также создание и использование различных отображений:

```
// Наши данные, которые мы храним в отображении (класс покупателей)
class Customer
{
    public:
        int CustID;
        wxString CustName;
};

// Объявляем класс для отображения
// Этот макрос декларирует и реализует отображение CustomerList
WX_DECLARE_HASH_MAP(int, Customer*, wxIntegerHash,
                    wxIntegerEqual, CustomerHash);

void HashTest()
{
    // Объявляем экземпляр нашего класса
    CustomerHash MyHash;

    bool IsEmpty = MyHash.empty(); // должно быть true

    // Создаем несколько покупателей
    Customer* CustA = new Customer;
    CustA->CustID = 10;
    CustA->CustName = wxT("Bob");

    Customer* CustB = new Customer;
    CustB->CustID = 20;
    CustB->CustName = wxT("Sally");

    Customer* CustC = new Customer;
    CustC->CustID = 5;
    CustC->CustName = wxT("Dmitri");

    // Добавляем покупателей в отображение
    MyHash[CustA->CustID] = CustA;
    MyHash[CustB->CustID] = CustB;
    MyHash[CustC->CustID] = CustC;

    int Size = MyHash.size(); // должно возвратить 3

    // метод count возвращает 0 или 1, т.е. существует ли
    // число 20 в отображении
    int Present = MyHash.count(20); // должно быть 1

    // Итератор для нашего отображения
    CustomerHash::iterator i = MyHash.begin();
```

```
    // метод End возвращает псевдоэлемент, следующий за последним
    while (i != MyHash.end())
    {
        // first - это ключ, а second - его значение
        int CustID = i->first;
        Customer* Cust = i->second;

        // обрабатываем покупателя

        // переходим к следующему элементу
        i++;
    }

    // Удаляем покупателя
    MyHash.erase(10);

    // Объект CustA НЕ УДАЛЯЕТСЯ при удалении из отображения
    delete CustA;

    // Возвращает итератор на узел с определенным ключом
    CustomerHash::iterator i2 = MyHash.find(21);

    // метод Find возвращает hash::end, если ключ не найден
    bool NotFound = (i2 == MyHash.end()); // должно быть true

    // На этот раз поиск должен быть успешным
    i2 = MyHash.find(20);

    // Удаляем элемент, используя итератор
    MyHash.erase(i2);
    delete CustB;

    // В качестве побочного эффекта данного действия будет
    // вставлено значение NULL для ключа 30
    Customer* Cust = MyHash[30]; // в Cust запишется NULL

    // Очищает структуру от всех элементов
    MyHash.clear();

    delete CustC;
}
```

13.6 Хранение и обработка дат и времени

wxWidgets предоставляет достаточно комплексный класс `wxDateTime` для хранения даты и времени с множеством доступных операций, таких как форматирование,

часовые пояса, различные арифметические операции и т.д. Статические функции дают информацию о текущей дате и времени, а также методы получения информации о високосном годе. Советуем использовать этот класс, даже если вам необходимо просто хранить информацию о времени и дате. Вспомогательные классы `wxTimeSpan` и `wxDateSpan` дают возможность легко изменять существующие объекты `wxDateTime`.

13.6.1 wxDateTime

Класс `wxDateTime` содержит слишком много методов, чтобы все их рассматривать здесь. Полную документацию по API вы можете найти в документации wxWidgets. В этом разделе мы дадим описание только самым часто используемым методами и операциям.

Хотя при работе класс времени предполагает, что время считается по гринвичу (Greenwich Mean Time — GMT), вам не надо об этом заботится, так как обычно вы будете работать в одном часовом поясе (локальном времени). Таким образом, все конструкторы и модификаторы wxDateTime, составляющие дату или время из компонентов (например из часов, минут и секунд) предполагают, что эти значения даются для локального времени. Все методы, возвращающие компоненты даты или времени (месяц, день, час, минуту, секунду и т.п.) также дают правильные значения для локального часового пояса, поэтому вам ничего не нужно делать, чтобы получать правильные результаты для вашей локального часового пояса. Чтобы узнать об управлении часовым поясом, обратитесь к документации.

13.6.2 Конструкторы и модификаторы wxDateTime

Объект `wxDateTime` может быть сконструирован из значения типа `time_t` (время Unix), информации о дате, информации о времени или полной информации о дате и времени. Для каждого конструктора существует соответствующий метод `Set`, который изменяет существующий объект, так, чтобы он имел определенные время или дату. Аналогично, существуют модификаторы для индивидуальных компонентов, такие как `SetMonth` или `SetHour`, которые меняют только одну компоненту даты или времени.

`wxDateTime(time_t)` создает объект с датой и временем в соответствии со значением времени Unix.

`wxDateTime(const struct tm&)` создает объект, используя данные из стандартной структуры `tm` языка C.

`wxDateTime(wxDateTime_t hour, wxDateTime_t minute = 0, wxDateTime_t second = 0, wxDateTime_t millisec = 0)` создает объект, руководствуясь заданной информацией о времени.

`wxDateTime(wxDateTime_t day, Month month = Inv_Month, int year = Inv_Year, wxDateTime_t hour = 0, wxDateTime_t minute = 0, wxDateTime_t second = 0, wxDateTime_t millisec = 0)` создает объект, руководствуясь заданной информацией о времени и дате.

13.6.3 Аксессоры wxDateTime

Аксессоры для `wxDateTime` обычно имеют понятные имена: `GetYear`, `GetMonth`, `Getday`, `GetWeekDay`, `GetHour`, `GetMinute`, `GetSecond`, `GetMillisecond`, `GeTDayOfYear`, `GetWeekOfYear`, `GetWeekOfMonth` и `GetYearDay`. `wxDateTime` также имеет следующие методы:

- `GetTicks` возвращает дату и время в формате времени Unix (число секунд прошедших с полночи 1 января 1970 года).

- `IsValid` указывает находится ли в объекте допустимая дата (можно сконструировать объект, который содержит недопустимую дату).

13.6.4 Получаем сегодняшнюю дату

Класс `wxDateTime` содержит два статических метода для получения текущего времени:

- `wxDateTime::Now` создает объект `wxDateTime` с текущей датой с точностью до секунды.

- `wxDateTime::UNow` создает объект `wxDateTime` с текущей датой с точностью до милисекунды.

13.6.5 Ввод и форматирование даты

Функции данного раздела преобразовывают объект `wxDateTime` в текст и из текста. Преобразование даты в текст достаточно тривиально: вы можете использовать форматирование даты и времени в соответствии с текущей локалью (`FormatDate` and `FormatTime`), использовать международный стандарт представления даты, определенный в ISO 8601 (`FormatISODate` и `FormatISOTime`) или определить произвольный стандарт, используя метод `Format`.

Создание даты из строки гораздо более интересная задача, потому что существует множество различных форматов. В самом простом случае ее можно считать с помощью `ParseFormat`, который может разобрать любую дату в определенном формате. `ParseRfc822Date` считывает дату в формате из RFC 822, который определяет формат даты для электронной почты в Интернете.

Самыми интересными функциями являются `ParseTime`, `ParseDate` и `ParseDateTime`, которые пытаются считать дату и/или время в «произвольном» формате, позволяя их определить различными путями. Эти функции можно использовать для разбора пользовательского ввода, который обычно не соответствует никакому предопределенному формату. Например, `ParseDateTime` может разбирать строки наподобие «tomorrow»[1], «March first»[2] и даже «next Sunday»[3].

[1] Завтра (англ.)
[2] Первое марта (англ.)
[3] Следующее воскресенье (англ.)

13.6.6 Сравнение дат

Два объекта `wxDateTime` можно легко сравнить, используя одну из многих функций сравнения. Все эти методы возвращают `true` или `false`.

Следующие методы сравнивают с другим объектом класса `wxDateTime`: `IsEqualTo` (равен), `IsEarlierThan` (раньше чем), `IsLaterThan` (позже чем), `IsSameDate` (одна и та же дата) и `IsSameTime` (одно и то же время).

Следующие методы делают сравнения используя два других объекта `wxDateTime`: `IsStrictlyBetween` (точно между) и `IsBetween` (между). Разница между этими двумя методами в том, что `IsStrictlyBetween` возвращает `false`, если `wxDateObject` равен одному из краевых значений промежутка, тогда как `IsBetween` вернет в таком случае `true`.

13.6.7 Арифметика дат

Библиотека wxWidgets содержит два очень продвинутых класса для выполнения арифметических операций с объектами `wxDateTime`: `wxTimeSpan` и `wxDateSpan`. `wxTimeSpan` просто содержит разницу в милисекундах и всегда дает быстрый и предсказуемый результат. С другой стороны время имеет и другие важные единицы измерения, такие как недели и месяцы. `wxDateSpan` пытается осуществить арифметические операции наиболее естественным путем, однако результат таких операций не всегда хорошо определен. Например, сложение 31 января и 1 месяца даст результат 28 (или 29) февраля, который является последним днем февраля, а не 31 февраля (этот день в феврале не существует). Конечно в данном случае мы получаем то, что ожидаем, однако вы очень удивитесь когда увидите, что вычитание того же самого интервала из 28 февраля даст результат 28 января (а не 31 января с которой мы начинали).

Можно выполнять множество различных операций с датами, но не все из них имеют смысл. Например, умножение даты на число является некорректной операцией, однако умножение временного интервала на число корректно.

- Сложение: Значения класса `wxTimeSpan` или `wxDateSpan` можно прибавить к `wxDateTime` и в результате получить новый объект `wxDateTime`. Также можно сложить два интервальных класса и получить другой объект того же самого класса.

- Вычитание: Вычитать можно те же самые объекты, как и в операции сложения. Кроме того можно взять разность двух объектов класса `wxDateTime`, которая будет иметь тип `wxTimeSpan`.

- Умножение: Объекты `wxTimeSpan` и `wxDateSpan` могут быть умножены на целое число. Результатом данной операции является объект того же самого типа.

- Унарный минус: Можно взять отрицание объектов `wxTimeSpan` или `wxDateSpan`. Результатом является интервал с тем же самым значением, но противоположным направлением во времени.

Следующий небольшой пример показывает использование `wxDateSpan` и `wxTimeSpan` для изменения времени, сохраненного в объекте класса `wxDateTime`. Обратитесь к официальной документации за полным списком доступных методов.

```
void TimeTests()
{
    // Взять текущую дату и время
    wxDateTime DT1 = wxDateTime::Now();

    // Промежуток в 2 недели и 1 день (т.е. в 15 дней)
    wxDateSpan Span1(0, 0, 2, 1);

    // Вычитаем 15 дней от сегодня
    wxDateTime DT2 = DT1 - Span1;

    // Статическое создание однодневной разницы
    wxDateSpan Span2 = wxDateSpan::Day();

    // Промежуток 3 теперь равен 14 дням
    wxDateSpan Span3 = Span1 - Span2;

    // 0 дней (промежуток определен как 2 недели)
    int Days = Span3.GetDays();

    // 14 дней (2 недели)
    int TotalDays = Span3.GetTotalDays();

    // 2 две недели в прошлое
    wxDateSpan Span4 = -Span3;

    // Промежуток в 3 месяца
    wxDateSpan Span5 = wxDateSpan::Month() * 3;

    // 10 часов, 5 минут и 6 сукунд
    wxTimeSpan Span6(10, 5, 6, 0);

    // Добавляем определенный промежуток к DT2
    wxDateTime DT3 = DT2 + Span6;

    // Span7 в 3 раза дольше, чем Span6, но направлен в прошлое
    wxTimeSpan Span7 = (-Span6) * 3;

    // SpanNeg должно быть true, т.к. промежуток отрицателен
    bool SpanNeg = Span7.IsNegative();

    // Статическое создание одночасовой разницы
    wxTimeSpan Span8 = wxTimeSpan::Hour();

    // Один час не длиннее 30+ часов (сравниваются абсолютные значения)
    bool Longer = Span8.IsLongerThan(Span7);
}
```

13.7 Вспомогательные структуры данных

wxWidgets использует несколько внутренних структур данных для передачи в качестве параметров и возвращения результатов из открытых методов объектов. Прикладные программисты могут также использовать эти вспомогательные функции в своих проектах.

13.7.1 wxObject

Класс `wxObject` является базовым для всех классов библиотеки wxWidgets. Именно он отвечает за реализацию получения информации о типе времени выполнения, подсчета ссылок, объявление виртуального деструктора и дополнительные отладочные версии операторов `new` и `delete`. Класс `wxClassInfo` хранит мета-данные о классе и используется в некоторых методах класса `wxObject`.

```
MyWindow* window = wxDynamicCast(FindWindow(ID_MYWINDOW), MyWindow);
```

Метод `IsKindOf` получает в качестве параметра указатель на `wxClassInfo` и возвращает `true`, если объект относится к данному типу. Например,

```
bool tmp = obj->IsKindOf(CLASSINFO(wxFrame));
```

Методу `Ref` передается параметр класса `const wxObject&`, который заменяет текущие данные объекта на ссылку на указанные в параметре. Число ссылок на текущий хранимый объект уменьшается, возможно освобождаются данные, а число ссылок на объект из параметра увеличивается.

`UnRef` уменьшает число ссылок на ассоциированные с ним данные и удаляет данные в случае, если число ссылок уменьшилось до нуля.

13.7.2 wxLongLong

Класс `wxLongLong` представляет 64-битные целые числа. Когда возможно используется родной 64-битный тип платформы, во всех остальных случаях используется эмулирующий его код. В результате вы можете использовать этот тип также как и любой другой (встроенный) целочисленный тип. Заметим, что число типа `wxLongLong` является числом со знаком. Если вам необходимо хранить беззнаковые числа, то используйте для этого тип `wxULongLong`, который имеет почти похожий API, кроме некоторых логических операторов (таких как взятие модуля числа). Для этих классов определены обычные математические операции, а также несколько дополнительных методов:

- `Abs` возвращает абсолютное значение `wxLongLong`. Если данный метод вызвать у константного объекта, то возвратится копия числа. Для неконстантного объека метод изменит сам объект и возвратит ссылку на него.
- `ToLong` возвращает результат преобразования в тип `long`. Метод вызывает исключение в debug-версии, если при этом происходит потеря точности.
- `ToString` возвращает строковое представление значения числа.

13.7.3 wxPoint и wxRealPoint

Тип `wxPoint` используется библиотекой для хранения пары целочисленных координат (точки) на экране или в окне. Как легко понять по названию, класс хранит пару значений `x` и `y`. Данные члены класса являются открытыми, а поэтому с ними можно работать непосредственно. Для `wxPoint` реализованы операторы + и -, которые позволяют прибалять или вычитать из текущий координат значения типа `wxPoint` или `wxSize`. `wxRealPoint` хранит вещественные (`double`) координаты вместо целочисленных, а также реализует операторы + и -, но только для вещественных точек.

Создать объект `wxPoint` очень несложно:

```
wxPoint myPoint(50, 60);
```

13.7.4 wxRect

Этот класс используется для хранения информации о прямоугольной области и обычно используется библиотекой wxWidgets для рисования или в элементах управления, таких как `wxDC` или `wxTreeCtrl`. Класс `wxRect` содержит поля `x` и `y`, а также `width` (ширина) и `height` (высота), которые являются открытыми. Прямоугольники можно складывать и вычитать друг из друга. Также класс поддерживает несколько других полезных методов.

`GetRight` возвращает координату `x` правой границы прямоугольника.

`GetBottom` возвращает координату `y` нижней границы прямоугольника.

`GetSize` возвращает размер прямоугольника (высоты и ширину) в виде объекта `wxSize`.

`Inflate` увеличивает размер прямоугольника на заданную величину, которая может быть одинаковой для обоих направлений (один параметр) или разной для разных (два параметра).

Метод `Inside` определяет, располагается ли данная точка внутри прямоугольника. Точку можно определить как пару координат или как объект `wxPoint`.

`Intersects` принимает в качестве параметра другой прямоугольник `wxRect` и определяет перекрываются ли они.

`Offset` перемещает прямоугольник на заданное смещение. Смещение можно задать пару значений или как объект `wxPoint`.

Объект `wxRect` можно создать тремя способами. Следующие три объекта определяют один и тот же прямоугольник:

```
wxRect myRect1(50, 60, 100, 200);
wxRect myRect2(wxPoint(50, 60), wxPoint(150, 260));
wxRect myRect3(wxPoint(50, 60), wxSize(100, 200));
```

13.7.5 wxRegion

Класс `wxRegion` представляет простую или сложную область на холсте или окне. Данный класс использует технологию подсчета ссылок, а поэтому операции копирования и присваивания выполняются для него очень быстро. В основном `wxRegion` используется для определения областей отсечения или обновления.

Метод Contains возвращает true, если данная пара координат, wxPoint, прямоугольник или wxRect находятся внутри данной области.

GetBox возвращает объект класса wxRect, представляющий прямоугольник, содержащий область.

Intersect возвращает true, если выбранный прямоугольник wxRect или wxRegion пересекает область.

Offset перемещает область на определенное смещение. Смещение определяется в виде пары координат x и y.

Методы Subtract (вычитание), Union (объединение) или Xor (исключающее или) меняет регион различными способами, предлагая для этого десяток перегруженных функций. У всех этих методов есть версии, принимающие в качестве параметров wxRegion или wxPoint. Обратитесь к документации, чтобы получить полный список доступных методов.

Следующий пример иллюстрирует четыре самых используемых метода инициализации wxRegion. Все следующие строки создают объекты, представляющие одну и ту же область:

```
wxRegion myRegion1(50, 60, 100, 200);
wxRegion myRegion2(wxPoint(50,60), wxPoint(150, 260));
wxRect myRect(50, 60, 100, 200);
wxRegion myRegion3(myRect);
wxRegion myRegion4(myRegion1);
```

Можно использовать класс wxRegionIterator, чтобы пройтись по всем прямоугольникам в данной области, например для перерисовки «поврежденной» области экрана в обработчике соответствующего события, как показано в следующем примере:

```
// Вызывается когда окно необходимо перерисовать
void MyWindow::OnPaint(wxPaintEvent& event) {
    wxPaintDC dc(this);

    wxRegionIterator upd(GetUpdateRegion());
    while (upd) {
        wxRect rect(upd.GetRect());

        // перерисовываем прямоугольник
        ... тут идет код ...
        upd++;
    }
}
```

13.7.6 wxSize

Класс wxSize используется внутри wxWidgets для хранения целочисленных размеров окна, элементов управления, объектов на холсте и т.д. Данный объект также часто возвращается методами, которые возвращают информацию о размере.

GetHeight и GetWidth возвращают высоту и ширину.

SetHeight и SetWidth принимают целочисленный параметр, определяющий новое значение для высоты или ширины.

Метод Set устанавливает новые параметры для размера.

Объет очень легко создать, указав высоту и ширину:

```
wxSize mySize(100, 200);
```

13.7.7 wxVariant

Класс wxVariant является контейнером для любого типа. Значение данного типа может изменяться в процессе выполнения программы, возможно вместе с типом. Этот класс может применяться для уменьшения размера кода в некоторых задачах, таких как редактор различных типов данных или реализации протокола удаленного вызова процедур.

wxVariant может сохранять значения типа bool, char, double, long, wxString, wxArrayString, wxList, wxDateTime, указатель на void и массив wxVariant. Однако, любое приложение может расширить возможности хранения в wxVariant. Для этого необходимо создать наследника от wxVariantData и использовать вариант конструктора класса wxVariant с параметром wxVariantData или оператор присваивания, чтобы присвоить классу wxVariant значение вашего типа. Настоящие значения для пользовательских типов необходимо получать через специальные методы объекта wxVariantData. Это отличается от использования стандартных типов для которых можно вызывать встроенные функции, такие как GetLong.

Помните, что не все типы данных могут быть автоматически преобразованы в другие типы. Например, отсутствует преобразование из логического типа в объект wxDateTime и из целочисленного в wxArrayString. Интуиция должна вам подсказать какого рода преобразования существуют. Вы всегда можете узнать текущий хранимый тип, используя для этого метод GetType. Далее приведен простой пример использования wxVariant:

```
wxVariant Var;

// Сохраняем wxDateTime и получаем wxString
Var = wxDateTime::Now();
wxString DateAsString = Var.GetString();

// Сохраняем wxString, а получаем вещественное число
Var = wxT("10.25");
double StringAsDouble = Var.GetDouble();

// Тип должен быть "string" (строка)
wxString Type = Var.GetType();

// Это недопустимое преобразование, так как нельзя преобразовать строку
// в число, поэтому метод возвратит 0
char c = Var.GetChar();
```

13.8 Итоги

Структуры данных, существующие в wxWidgets позволяют вам легко передавать-/получать структурированные данные в/из библиотеки wxWidgets и внутри своего приложения. Мощные функции и классы для обработки информации, такие как `wxRegEx`, `wxStringTokenizer`, `wxDateTime` и `wxVariant` покроют большинство ваших потребностей в хранении и обработке данных без привлечения сторонних библиотек.

Далее мы рассмотрим средства, предоставляемые wxWidgets для работы с файлами и потоками.