ProgOptions: Руководство программиста

DocumentId:GradSoft-PR-r-26.09.2000-v1.0.3

May 8, 2003

Contents

1 Введение

ProgOptions представляет собой компоненту для обработки опций командной строки.

Используя ProgOptions можно легко и быстро сделать следующие вещи:

1. установить факт наличия в полученной Вами командной строке определенных (описанных) опций;

2. выделить аргументы опций (- при этом ProgOptions понимает несколько вариантов синтаксиса, например

```
--a 0x!
```

распознается как опция "a" с аргументом "0x!", также как

```
--a="Say Ox!"
```

распознается как опция "a" с аргументом "Say Ox!");

3. как только факт наличия какой-либо опции установлен, автоматически выполнить некоторые связанные с ней действия.

Этот документ представляет собой неформальное описание, полная спецификация пакета приводится в API reference (`www.gradsoft.kiev.ua/common/ToolBox/ProgOptions/API/index.html`).

2 Общее описание механизма работы

2.1 Базовый способ использования

Базовый способ использования ProgOptions предполагает, что командная строка, с которой запущена Ваша программа, уже разобрана на слова и передана Вам в виде пары (argc,argv), где argc (типа int) - количество аргументов в командной строке, argv (типа char**) - массив аргументов. В этом случае ProgOptions можно использовать так:

- Создать объект ProgOptions, при необходимости передавая конструктору класса следующие параметры:

  1. const char* optPrefix - подстрока, с которой должны начинаться все различаемые объектом опции (по умолчанию " --");
  2. const char* pkgPrefix - дополнительная подстрока, которую должны содержать предопределенные опции help и config (о них см.ниже). По умолчанию "";
  3. bool allowUnknownOptions - решение пользователя относительно того, что делать если интерпретация какой-либо опции невозможна: продолжать интерпретацию других опций (true) или прекратить работу (false). По умолчанию false.

- Описать интерпретируемые опции, их действие, а также дополнительные help-сообщения при помощи методов ProgOptions::put() и ProgOptions::setAdditionalHelp

- Получить набор опций в виде char** из внешего мира (например из аргументов main), после чего вызвать метод разбора опций ProgOptions::parse(int,char**); после вызова parse(int,char**) аргументы командной строки будут разобранны во внутренние структуры ProgOptions и, если это было задано, будут вызваны callback - функции пользователя.

- После этого, используя `ProgOptions::is_set("option")` можно узнать установлена ли опция "option", а для опций с аргументом - узнать значения аргументов, используя метод ProgOptions::argument("option")

Пример:

```
#include <GradSoft/ProgOptions.h>  // самое главное

GradSoft::ProgOptions options;   // создать объект с параметрами по умолчанию:

void init()
{
 options.put("qqq", "qqq option", false );           // описать опцию qqq
                                                     // (false - опция без аргумента)
 options.put("zzz", "option with argument", true );  // описать опцию zzz
                                                     // (true - опция с аргументом)

 options.setAdditionalHelp(true,                     //
   "This program illustate usage of GradSoft library"// определить дополнительные
 );                                                  // help-сообщения
 options.setAdditionalHelp(false,                    //
   "and this is shown at the end of usage screen"    //
 );                                                  //
};

int main(int argc, char** argv)
{
 init();                                        // все подготовить (см.выше)

 if (!options.parse(argc,argv)) return 1;      // разобрать опции, доступные через argv

 if (options.is_set("qqq")) {      // проверить, установлена ли опция --qqq
   cout << "qqq is set" << endl;
 }
                                   // проверить, установлена ли опция --zzz,
                                   // и если да - получить ее аргумент
 if (options.is_set("zzz")) {
   cout << "zzz is set with argument:" << options.argument("zzz") << endl;
 }
 return 0;
}
```

Запуск данной программы с командной строкой:

```
--qqq --zzz zz-arg
```

приведет к следующему результату:

```
qqq is set
zzz is set with argument:zz-arg
```

Альтернативный запуск с опцией:

```
--"Пошел ты!"
```

приведет к сообщению об ошибке:

```
<executable>:unknown option
```

где <executable> - имя программы

2.2 Предопределенные опции help и config

В дополнение к описанному программистом набору опций, ProgOptions идентифицирует предопределенные опции `<optPrefix><pkgPrefix>help` и `<optPrefix><pkgPrefix>config`. Результатом запуска с опцией `<optPrefix><pkgPrefix>help` является вывод в стандартный поток сообщений об ошибках предопределенного help-а (который суть перечень опций с их описаниями, заданными на этапе вызова put()), а также дополнительной информации, заданной с помощью метода ProgOptions::setAdditionalHelp(). Результатом запуска с опцией `<optPrefix><pkgPrefix>config <fname>` является разбор дополнительного набора опций, заданных в файле конфигурации `fname` (подробнее о файле конфигурации см. в разделе 4)

Пример:

Запустим описанную программу с опицией `--help` (UNIX):

```
./a.out --help
This program illustate usage of ProgOptions library
  --qqq         qqq option
  --zzz <argument>  option with argument
and this is shown at the end of usage screen
```

Внимание: Использование предопределенной опции `--config <filename>` требует некоторой поддержки со стороны программиста в том случае, если разбором опций занимается кто-нибудь кроме него (например, это может быть функция ORB, ORB_init()). Дело в том, что метод ProgOptions::parse(argc,argv) копирует опции, полученные в виде argv, и опции, полученные в файле <filename>, в отдельный, специально для этого созданный, вектор аргументов, ссылку на который можно получить при помощи метода argv(), и далее использует его. Первоначальный же вектор argv не меняется, и если Вы передадите его, скажем, функции ORB_init(), то опции, переданные в файле, естественно туда не попадут. Поэтому, если Вы хотите, чтобы при помощи `--config <filename>` можно было устанавливать опции командной строки, распознаваемые внешним ПО, Вы должны

1. разобрать вектор аргументов при помощи parse(argc,argv);
2. передать внешнему ПО тот вектор аргументов (и его длину), который возник в результате выполнения метода ProgOptions::parse().

При этом, поскольку внешее ПО может удалять распознанные им опции, наиболее правильно передавать ему **копию** вектора аргументов, для чего можно использовать wrapper-класс ProgOptions::ArgsHolder:

```
ProgOptions options("--","",true);
if (!options.parse(argc,argv)) return 1;

ProgOptions::ArgsHolder argsHolder;
argsHolder.takeArgv(options);

CORBA::ORB_var orb = CORBA::ORB_init(argsHolder.argc,argsHolder.argv);
```

Скопированный при помощи argsHolder::takeArgv(const ProgOptions&) вектор аргументов будет автоматически уничтожен в деструкторе argsHolder.
Внимание: на сегодняшний день передать имя конфигурационного файла при помощи знака присваивания нельзя, т.е. синтаксис `--config=<config file>` не работает. Используйте только `--config <config file>`.

Теоретически, опции help и config можно переопределить при помощи метода ProgOptions::put, и тогда они станут обычными, ничем не отличающимися от других, опциями.

3 callback функции

Еще один механизм обработки опций, предоставляемый ProgOptions, - это callback функции. Свою callback функцию можно связать с каждой определенной пользователем опцией, и она будет автоматически вызваться во время parse как только опция будет обнаружена.

Осуществить связь можно при помощи параметра |callback| метода `ProgOptions::put`. Например, добавим в нашей программе следующий код:

```
bool zz1Callback(const GradSoft::ProgOptions* options, const char* argument,
                 void* )
{
 cout << "zz1:callback called with argument " <<
            ((!argument) ? "NULL" : argument) << endl;
 return true;
}

void init()
{
 .....
 options.put("zz1", "option with argument and callback", true, zz1Callback );
};
```

Теперь попробуем вызвать программу с опцией "zz1":

```
/a.out --zz1 xx
zz1:callback called with argument xx
```

Отметим, что callback-функция вызывается при каждом нахождении связанной с ней опции, поэтому с помощью callback можно организовать обработку нескольких одинаковых опций с разными аргументами:

```
/a.out --zz1 xx --zz1 yy
zz1:callback called with argument xx
zz1:callback called with argument yy
```

4 Дополнительные возможности

4.1 Установка свойств ProgOptions

- optPrefix, pkgPrefix (char*)– опции, различаемые ProgOptions должны иметь вид `<optPrefix><pkgPrefix><optionName>` .
  - Значение по умолчанию: `--` и пустая строка, соответственно
  - Способ задания: параметры конструктора.
- allowUnknownOptions (bool) – если это свойство установленно в true, то ProgOptions игнорирует те элементы командной строки, которые не соответствуют синтаксису опций, либо соответствую синтаксису, но не заданны в put. Если это свойство установлнно в false, то ProgOptions::parse в этом случае возвращает false.
  - Значение по умолчанию: false
  - Способ задания:
    - Параметр конструктора
    - `ProgOptions::setAllowUnknownOptions(bool);`
  - чтение: `bool ProgOptions::getAllowUnknownOptions()`

4.2 Использование файла конфигурации

Как было описано выше, использование ProgOptions::parse приводит к тому, что набор опций будет считан из файла конфигурации, если конечный пользователь употребит предопределенную опцию командной строки `--config <filename>` где <filename> - полное имя файла конфигурации.

Дополнительная возможность состоит в том, что config-файл можно прочитать независимо от решения конечного пользователя - т.е. по собственному решению программиста, - используя метод ProgOptions::parseFile(const char* configFname, const char* executable="unspecified"), где configFname - имя файла. Например,

```
parseFile("D:\Demo\config.ini");
```

разберет текст, найденный в файле `D:\Demo\config.ini`, и то, что он посчитает за опции, интерпретирует так же, как это бы сделал метод ProgOption::parse(argc,argv) с подходящим argv. (Второй параметр метода, executable, это имя выполняемого файла, которое надо задать, если вызов ProgOptions::parseFile не предупреждается вызовом ProgOptions::parse.)

Еще одна возможность состоит в том, что опции, которые были проанализированы при помощи методов ProgOptions::parse, ProgOptions::parseFile и ProgOptions::parseString (см.ниже) можно объединить и сохранить для последующего использования при помощи метода `ProgOptions::saveToFile`. В частности, вызов

```
parse(argc,argv);
...
parseFile("D:\Demo\config.ini");
...
saveToFile("D:\Demo\config1.ini");
```

приведет к тому, что все опции, полученные через argv и считанные из файла `D:\Demo\config.ini` (как распознаные, так и не распознанные) будут записаны в форматированный файл `D:\Demo\config1.ini` который можно будет снова прочесть при помощи ProgOptions::parseFile.

ВАЖНО:

Оба метода для работы с файлами, ProgOptions::parseFile и ProgOptions::saveToFile, возвращают булевское значение, говорящее успешности или неуспешности операции. В случае ProgOptions::saveToFile возврат false означает, что произошла системная ошибка, информация о которой доступна в переменной среды errno. Следовательно, рекомендуемый способ записи опций выглядит так:

```
ProgOptioins options;
  ...
if (!options.saveToFile(configFname)) {
  perror(options.argv(0));
  ...
}
```

В случае ProgOptions::parseFile использовать подобный способ действий не следует, т.к. указанный метод с необходимостью возвращает false в трех разных случаях:

1. произошла системная ошибка при чтении файла,
2. формат файла неправильный,
3. разбор прочитанных опций невозможен.

При этом в случае системной ошибки или неправильного формата файла ProgOptions::parseFile самостоятельно сигнализирует о сбое, выводя сообщение на стандартный поток сообщений об ошибках программы.

4.3 Формат файла конфигурации

Формат файла конфигурации не является обременительным, однако, все-таки, не произволен.

1. Файл состоит из слов (=опций), символов-разделителей и комментариев.
2. Словом считается:
   (a) непрерывная последовательность "видимых" символов;
   (b) любая последовательность видимых символов, пробелов и символов табуляции, заключенная в кавычки (при интерпретации кавычки игнорируются);
   (c) непрерывная последовательность таких последовательностей.

   Пример:

   ```
   aaa_bbb "aaa bbb" aaa" bbb ccc"
   ```

   - три разных слова, которые будут интерпретированы как `'aaa_bbb'`, `'aaa bbb'`, `'aaa bbb ccc'`
3. К комментариям относятся:
   (a) маркеры `'/*'`, `'*/'` и все, что стоит между ними;
   (b) маркер `'//'` и часть строки после него;
   (c) строка, начинающаяся с символа `'#'`.

   Примеры:

   ```
   # This string will be ignored

     aaa /* This part of string will be ignored */ bbb

     ccc // This part of string will be ignored too
   ```
4. Первое слово файла должно находиться среди первых 10000 символов первой строки и в "чистом" виде должно представлять собой маркер `'@ProgOptions Config File'`. Таким образом, оно может быть задано одним из следующих способов:

   ```
   @"ProgOptions Config File"

   "@ProgOptions Config File"
   ```

   или каким-то другим.

5. Эффекты:

   (a) если кавычки открыты и не закрыты, границей слова считается конец строки;

   (b) если символ кавычек должен быть членом слова, его можно вставить как '\"', при этом косая черта игнорируется;

4.4 Пример файла конфигурации

Пример файла конфигурации:

```
@"ProgOptions config file"

# The list of options:

    --a          /* single option */
    --b c        // option with argument
    --d="a b c"  // another option with argument
```

4.5 Метод ProgOptions::parseString

Еще одна возможность ProgOptions - это способность разобрать строку, заданную в виде массива символов. Метод ProgOptions::parseString(const char* item, const char* executable="unspecified") разбирает строку так же, как метод ProgOptions::parseFile разбирает файл за исключением того, что:

1. не нужен маркер файла '@ProgOptions config file';
2. символ `'#'` не имеет никакого специального значения.

5 Требования к программному окружению

Если Вы работаете под управлением Windows NT, Вам необходимо:

1. определить макрос WIN32 перед включением файла ProgOptions.h
2. использовать "новые" библиотеки и, соответственно, заголовочные файлы iostream, fstream и т.п. вместо аналогичных iostream.h, fstream.h и т.п.

6 Перечень изменений

31.01.2002 - описана поддержка опции `--config <filename>`

24.01.2002 - добавлено предупреждение о том, что синтаксис `--config=<filename>` не работает

03.01.2002 - изменены:

1. пример использования (раздел 2.1): заголовочный файл `GradSoft/ProgOptions` больше не используется;
2. описание порядка разбора конфигурационного файла (раздел 4.3): конструкции '//','/*','*/' можно использовать в тексте опций;
3. требования к программному окружению (раздел 5): "старые" библиотеки больше не используются;

17.02.2001 - коррекция, добавлены формальные аттрибуты эксплуатационной документации, добавлена подглава о установках свойств.

29.06.2000 - первая версия.