OPEN SOURCE SOFTWARE DEVELOPMENT SERIES

An Introduction to Design Patterns in C++ with Qt™

Second Edition

Alan Ezust • Paul Ezust

Foreword by Lars Knoll, Director Qt Research and Development

An Introduction to Design Patterns in C++ with Qt™, 2nd Edition

In general, command-line arguments can be any of the following:

- **switches**, such as `-verbose` or `-t`
- **parameters** (typically filespecs), which are simple strings not associated with switches
- **switched parameters** such as the gnu compiler's optional `-o` switch, which *requires* an accompanying parameter, the name of the executable file to generate

The following line contains examples of all three kinds of arguments:

```
g++ -ansi -pedantic -Wall -o myapp someclass.cpp someclass-demo.cpp
```

Example 6.25 shows how a C program might deal with command-line arguments.

EXAMPLE 6.25 src/derivation/argumentlist/argproc.cpp

```
[ . . . . ]
#include <cstring>

bool test = false;
bool verbose = false;

void processFile(char* filename) {
[ . . . . ]
}

/*
  @param argc - the number of arguments
  @param argv - an array of argument strings
*/
int main (int argc, char* argv[]) {
  // recall that argv[0] holds the name of the executable.
  for (int i=1; i < argc; ++i) {                                    1

     if (strcmp(argv[i], "-v")==0) {
        verbose = true;
     }
     if (strcmp(argv[i], "-t") ==0) {
        test = true;
     }
  }
  for (int i=1; i < argc; ++i) {                                    2
     if (argv[i][0] != '-')
```

```
        processFile(argv[i]);
  }
}
[ . . . . ]
```

1 First process the switches.

2 Make a second pass to operate on the non-switched arguments.

Qt enables you to avoid the use of arrays, pointers, and `<cstring>` by using more object-oriented constructs.

In Example 6.26, you can see how code like this could be greatly simplified through the use of higher-level classes: `QString` and `QStringList`.

6.7.1 Derivation and `ArgumentList`

`ArgumentList` provides an example of a reusable class with a specific purpose derived from a more general-purpose Qt class. It reuses `QString` and `QStringList` to simplify the processing of command-line arguments.

Operationally, `ArgumentList` is a class that is initialized with the `main()` function's `int` and `char**` parameters, which capture the command-line arguments. Conceptually, `ArgumentList` is a list of `QString`s. Structurally, it is derived from `QStringList`, with some added functionality. A Java programmer would say that `ArgumentList` is *extended* from `QStringList`. Example 6.26 contains the class definition for `ArgumentList`.

EXAMPLE 6.26 src/derivation/argumentlist/argumentlist.h

```
#ifndef ARGUMENTLIST_H
#define ARGUMENTLIST_H

#include <QStringList>

class ArgumentList : public QStringList {
  public:
    ArgumentList();

    ArgumentList(int argc, char* argv[]) {
        argsToStringlist(argc, argv);
    }
```

```
    ArgumentList(const QStringList& argumentList):
       QStringList(argumentList) {}
    bool getSwitch(QString option);
    QString getSwitchArg(QString option,
                         QString defaultRetVal=QString());
  private:
    void argsToStringlist(int argc,  char* argv[]);
};
#endif
```

Because it is publicly derived from `QStringList`, `ArgumentList` supports the full interface of `QStringList` and can be used wherever a `QStringList` is expected. In addition to its constructors, `ArgumentList` defines a few additional functions:

- `argsToStringList()` extracts the command-line arguments from the given array of `char` arrays and loads them into a `QStringList`. This function is `private` because it is part of the implementation of this class, not part of the public interface. It is needed by the constructors but not by client code.
- `getSwitch()` finds and removes a switch from the string list, if that switch exists. It returns `true` if the switch is found and `false` otherwise.
- `getSwitchArg()` finds and removes a switch and its accompanying argument from the string list and returns the argument if the switch is found. It does nothing and returns a `defaultValue` if the switch is not found.

Example 6.27 shows the implementation code for these functions.

EXAMPLE 6.27 src/derivation/argumentlist/argumentlist.cpp

```
#include <QCoreApplication>
#include <QDebug>
#include "argumentlist.h"
ArgumentList::ArgumentList() {
   if (qApp != NULL)                                          1
      *this = qApp->arguments();
}


void ArgumentList::argsToStringlist(int argc, char * argv []) {
   for (int i=0; i < argc; ++i) {
      *this += argv[i];
   }
}
```

```
bool ArgumentList::getSwitch (QString option) {
   QMutableStringListIterator itr(*this);
   while (itr.hasNext()) {
     if (option == itr.next()) {
        itr.remove();
        return true;
     }
   }
    return false;
}

QString ArgumentList::getSwitchArg(QString option, QString defaultValue) {
   if (isEmpty())
      return defaultValue;
   QMutableStringListIterator itr(*this);
   while (itr.hasNext()) {
      if (option == itr.next()) {
         itr.remove();
         if (itr.hasNext()) {
            QString retval = itr.next();
            itr.remove();
            return retval;
         }
         else {
            qDebug() << "Missing Argument for " << option;
            return QString();
         }
      }
   }
   return defaultValue;
}
```

1 A global pointer to the current `QApplication`.

In the client code shown in Example 6.28, all argument processing code has been removed from `main()`. No loops, `char*`, or `strcmp` are to be found.

EXAMPLE 6.28 src/derivation/argumentlist/main.cpp

```
#include <QString>
#include <QDebug>
#include "argumentlist.h"
```

```
void processFile(QString filename, bool verbose) {
   if (verbose)
      qDebug() << QString("Do something chatty with %1.")
                        .arg(filename);
   else
      qDebug() << filename;
}

void runTestOnly(QStringList & listOfFiles, bool verbose) {
   foreach (const QString &current, listOfFiles) {
      processFile(current, verbose);
   }
}

int main( int argc, char * argv[] ) {
   ArgumentList al(argc, argv);                                    1
   QString appname = al.takeFirst();                               2
   qDebug() << "Running " << appname;
   bool verbose = al.getSwitch("-v");
   bool testing = al.getSwitch("-t");                              3
   if (testing) {
      runTestOnly(al, verbose);                                    4
      return 0;
   } else {
      qDebug() << "This Is Not A Test";
   }
}
```

1. Instantiate the `ArgumentList` with command-line args.
2. Inherited from `QStringList`—first item in the list is the name of the executable.
3. Now all switches have been removed from the list. Only filenames remain.
4. `ArgumentList` can be used in place of `QStringList`.

Following are some sample outputs from running the program in Example 6.28:

```
src/derivation/argumentlist> ./argumentlist
Running  "./argumentlist"
This Is Not A Test
src/derivation/argumentlist> ./argumentlist item1 "item2 item3" item4 item5
Running  "./argumentlist"
This Is Not A Test
src/derivation/argumentlist> ./argumentlist -v -t "foo bar" 123 space1 "1 1"
```

```
Running  "./argumentlist"
"Do something chatty with foo bar."
"Do something chatty with 123."
"Do something chatty with space1."
"Do something chatty with 1 1."
src/derivation/argumentlist>
```

6.7.2 Exercises: Processing Command-Line Arguments

Write a birthday reminder application called `birthdays`.

- Store name/birthday pairs in any format you like, in a file called `birthdays.dat`.
- `birthdays` with no command-line arguments opens the file and lists all birthdays coming up in the next 30 days, in chronological order.
- `birthdays -a "john smith" "yyyy-mm-dd"` should add an entry to the file.
- `birthdays -n 40` shows birthdays coming up in the next 40 days.
- `birthdays` *`nameSpec`* searches for a birthday paired with *nameSpec*.

6.8 Containers

Qt's container classes collect **value types** (things that can be copied), including pointers to object types[7] (but not object types themselves). **Qt containers** are defined as template classes that leave the collected type unspecified. Each data structure is optimized for different kinds of operations. In Qt there are several template container classes to choose from.

- `QList<`*T*`>` is implemented using an array, with space preallocated at both ends. It is optimized for index-based random access and, for lists with less than a thousand items, it also gives good performance with operations like `prepend()` and `append()`.
- `QStringList` is a convenience class derived from `QList<QString>`.

[7] Chapter 8, "QObject, QApplication, Signals, and Slots," discusses `QObject` and object types.