Working with Grace

Pim van Riezen

優雅

Working With Grace

First edition copyright © 2006 Pim van Riezen - PanelSix V.O.F. Rotterdam

Introduction

The Grace library is a C++ toolkit for writing system applications that do stuff on or for the Internet. It contains a modest hierarchy of classes that will help you get your things done faster, leaving you more time for important stuff like enjoying family life or showing general disdain for your software's userbase. The library uses several niffty constructs that make it easy to create legible code, which is a blessing for the maintainability of your software. Even though it's natural that you don't care about the mess you leave for the next guy who inherits your code, at least now you can pray that your predecessor at the next gig used Grace.

During its development, the library has seen extensive abuse within several telco and ISP environments. The simple concepts it delivers will encourage you to think about your problem at a higher level, while still giving you the benefits of natively compiled C++-code.

Of course there is no such thing as a silver bullet for secure or timely software development, but the methods and concepts inside the Grace library can be helpful in preventing the pitfalls that come with designing robust applications that span multiple threads and expose interfaces to the network.

Audience

This handbook will show you around the library, its classes, and how you can use them to create your own tools and system services in less time. It will be tremendously helpful to your understanding if you have some experience with the C and C++ languages. Grace is on the lighthearted side of the Object Oriented political spectrum, puritans beware!

Typographical Conventions

In the code examples, we are using the following conventions:

```
Grace keywords and classes are green and italic.
Language keywords and classes are blue and bold.
Preprocessor keywords are red.
"String literals are orange"
The other code is black.
// Comments are italic and light grey.
```

Most of the distinctions should be visible in black-and-white renderings of this document as well. Examples of command line data use an output format for a bash shell with the prompt '$', like this:

```
$ command --flag=bar
output data
$ anothercommand
output data
```

Structure

The book is set up to progressively disclose functionality and concepts of Grace. It should do a good job complementing the online Doxygen documentation. After working yourself through this book you should be able to create your own programs with Grace and the online documentation. You will be grinding out weapons systems for the upcoming revolution, writing floorspace optimization systems for sweatshops in Thailand or creating a program that will really annoy the Recording Industry Association of America before you know it.

The Data Classes

Within Grace, a number of basic classes are used intensively as argument and return objects for other parts of the library. This makes sense, it's what we call eating your own dogfood: a good way to make sure that your product is tasty and nutricious. First we will go over these classes quickly, then we'll illustrate their typical usage through some example code.

The string Class

The most lamented design decision in the C language is the lack of a native string type. There are support groups for people suffering from this oversight. In place of a string type proper, in C there is only the concept of an array of character bytes. The Grace **string** class is a good alternative for storing static and dynamic string data. Some of its strong points:

- When string data is assigned from another **string** object, the library uses a copy-on-write mechanism to keep memory usage within bounds: Both strings use the same part of memory until one of them is changed and reserves a private copy.
- Many built-in operations can be performed on the object to help you with common tasks when processing all kinds of text data.
- The string data is kept inside a dynamic buffer that allocates new memory on the go, so there is no such thing as a buffer overflow.
- A string can contain full 8 bit data including NUL characters, so it can double up as a general storage buffer for binary data like bitmaps.

Objects of the **string** class can be used seamlessly when calling C functions that expect a regular C string. The other way around things are fine, too. If you define functions or methods to accept a **string** object as an argument, it will accept a C-string just fine.

The statstring Class

Like the regular **string** class, this is a container for string data, but it is optimized for keeping strings that are more static in nature and could be used as an index value for object collections. Alongside the regular string buffer, a calculated hash value is kept inside its objects for this purpose. Within a single thread context, all **statstring** objects that have the same data point to the same memory region, making it a useful companion for large arrays of indexed sets.

The value Class

A powerful container class for keeping variable data. A **value** object can contain any of the following:

- Signed or unsigned integer values (8, 16, 32 or 64 bits)
- Strings
- Boolean values
- IP address values
- Fixed point decimal numbers with 0.001 resolution for **currency** data.

- Double precision floating point values
- Attributes indexed by a **statstring** key (attributes themselves are **value** objects, although it is frowned upon to appoint these attribute objects child nodes or attributes)
- Child **value** objects (either numbered array nodes or indexed by a statstring key)

In short, it has all the ingredients for exchanging data without casting the data specification in iron. There are a many ways to serialize and deserialize **value** objects, with representation formats including:

- GraceXML
- Generic XML with translation schemas
- Apple/NeXT property lists
- Serialized PHP
- Compressed XML as used by Apple/NeXT binary serialization
- Grace Compressed XML (SHoX)
- CSV files
- INI files
- Simple key/value representation
- JSON

In other words, it's extremely easy to connect and talk to the outside world and create easy bindings from your applications for data coming from and going to programs written in other languages.

The currency Class

This is a simple class for fixed point arithmetic, to be used when doing financial calculations that mandate non-use of floating point numbers. It can be formatted for output, externally it is always represented as a cents amount with 2 decimals, internally it currently uses 3 decimals. Beancounters with fixations on regulatory banalities tend to frown upon scary floating point calculations for financial statements; In such a war on your sanity this will be a useful weapon.

Data Classes in Action

Let's take a look at how we're normally using the different data classes. A bit of fictional code:

```
#include <grace/str.h>
#include <grace/value.h>

// Main application method
int myApp::main (void)
{
    value val;
    string mystr;
    statstring statstr;

    mystr = "Hello, world";
    val["messageOfPeace"] = mystr;
    val["ultimateAnswer"] = 42;

    statstr = "messageOfpeace";
    if (mystr != val[statstr])
    {
        ferr.printf ("Something's wrong with this world!\n");
        return 1;
    }

    // Clear the string and put some formatted text in
    mystr.crop ();
    mystr.printf ("Message of Peace: %s\n", val[statstr].cval());
    mystr.printf ("Ultimate Answer: %i\n", val["ultimateAnswer"].ival());

    // Send the string to the output channel.
    fout.puts (mystr);
    return 0;
}
```

In this example we use a **value** object **val** as an associative array containing two key/value pairs. One is a string, the other is an integer. We use a **statstring** object as a key inside the value, but regular string constants work just as well. Then we put some formatted text inside a **string** object and send it to the output channel.

Data Objects and Memory Management

Let's take a look at how Grace data objects are passed to and from functions and methods in a way that frees your mind from worrying about memory management.

```
#include <grace/str.h>
#include "myapplication.h"

class personalName
{
public:
    personalName (void)
    {
        _first = "John";
        _last = "Doe";
    }
    ~personalName (void) { }
```

```
        // Accessor/Setter functions
        const string &first (void) const { return _first; }
        const string &last (void) const { return _last; }
        void first (const string &arg) { _first = arg; }
        void last (const string &arg) { _last = arg; }

        // Combine the two strings into a new string.
        string *makeFullName (void)
        {
            returnclass (string) result retain;
            result = _first;
            result += " ";
            result += _last;

            return &result;
        }

    protected:
        string _first;
        string _last;
    };

    // Main application method.
    int myApp::main (void)
    {
        personalName nom;
        string fname;

        nom.first ("Miles");
        nom.last ("Davis");

        fname = nom.makeFullName ();
        fout.writeln (fname);

        return 0;
    }
```

We defined a class here containing two **string** objects. Accessor methods for reading or writing these protected strings work on the basis of const references. The **makeFullName** method combines the two strings into a new string and returns this to the caller. This is where Grace does some magic for you. All data classes used in grace display special behavior when they are assigned with a pointer to an object of their own class (either through a copy-constructor or the '=' operator): Instead of bluntly copying the other object's data, it steals its data references and silently removed the temporary object from memory.

The **returnclass...retain** macro creates a temporary object and a temporary reference to it. The object is sent to the caller by deriving the pointer from this reference again. We go through this trouble to give us some convenience in dealing with all the object's overloaded operators, which is somewhat tricky if you only have a pointer. The macro also ensures that the temporary object's memory is allocated from a static pool outside the normal memory allocator. This makes the process faster and allows the Grace library to detect leaks if a caller doesn't assign a return-value to an object.

Using the value Class

The **value** class is a really flexible way to keep collections of mixed data together. For many applications passing or returning value objects can make a lot of sense: Using structs or classes for complex data layouts can be both tedious and inflexible. Let's take a look into some of the features of the **value** class and how you use them.

Variant Data

You can stuff any native non-pointer type into a **value** object, and use it in place of one of those types in most situations, so for instance, this will work just fine:

```
value v = 42;
int i = v; // i will contain 42.
```

In cases where there could be ambiguity about the proper cast, there are some explicit methods you can call on a **value**:

```
int value::ival (void)
unsigned int value::uval (void)
double value::dval (void)
long long value::lval (void)
unsigned long long value::ulval (void)
bool value::bval (void)
const string &value::sval (void)
const char *value::cval (void)
```

At any given time, a value object has a stored internal type. If you are interested in the variant type, you can use the **itype** call:

```
void myfunction (const value &val)
{
    switch (val.itype())
    {
       case i_unset:
           printf ("void object\n"); break;
       case i_int:
          printf ("int object\n"); break;
       case i_unsigned:
          printf ("unsigned int object\n"); break;
       case i_double:
          printf ("double (float) object\n"); break;
       case i_long:
          printf ("long long object\n"); break;
       case i_ulong:
          printf ("unsigned long long object\n"); break;
       case i_bool:
          printf ("bool object\n"); break;
       case i_string:
          printf ("string object\n"); break;
       case i_ipaddr:
          printf ("ipv4 object\n"); break;
       case i_date:
          printf ("date object\n"); break;
       case i_currency:
          printf ("currency object\n"); break;
    }
}
```

Array Handling

You can use a value as an automatically growing array. There are two ways to create an array:

```
value myarray;
myarray[0] = "Hack";
myarray[1] = "The";
myarray[2] = "Planet";

value secondarray;
secondarray.newval() = "Hack";
secondarray.newval() = "The";
secondarray.newval() = "Planet";
```

The second implementation is more suited for adding members in a loop. Note that arrays are not sparse, so accessing high numbered members creates a lot of objects. Talking about loops, let's see how we can iterate over an array. Again, there are two ways:

```
for (int i=0; i<myarray.count(); ++i)
{
    fout.printf ("member(%i): %s\n", i, myarray[i].cval());
}

foreach (node, myarray)
{
    fout.printf ("member: %s\n", node.cval());
}
```

The first method is useful if, for some reason, you need access to the counted data. The second method, however, can be a bit faster when you need to access the node you want to look at multiple times.

Dictionary Handling

A value can also have child values with a named key. Everything works mostly the same here, except the indexes you ask for:

```
value mydict;

mydict["question"] = "What is six times eight?";
mydict["answer"] = 42;
mydict["source"] = "Deep Thought";

foreach (node, mydict)
{
   fout.printf ("%s: %s\n", node.id().cval(), node.cval());
}
```

You can also mix the two kinds of members, you can use the **ucount** method to get the count of only the members with no name. These are always at the top of the array:

```
value mydict;

mydict["foo"] = "bar"; // is now equivalent to mydict[0]
mydict.newval() = "test"; // the new mydict[0], foo is moved to the right.
mydict["baz"] = "quux"; // equivalent to mydict[2].
mydict.newval() = "choo"; // the new mydict[1], foo and baz move right.
```

```
fout.printf ("%i %i\n", mydict.ucount(), mydict.count());
// returns 2 4
```

Another lifesaver sometimes is the ability to get an index from the right side of the array by using a negative index:

```
value myrecords;

myrecords.newval();
myrecords[-1]["name"] = "John Doe";
myrecords[-1]["email"] = "john@doe.org";

myrecords.newval();
myrecords[-1]["name"] = "John Q. Lamer";
myrecords[-1]["email"] = "john@lamer.org";
```

More Child Management

Some other ways to manage the array of children:

```
void handle (const value &arg)
{
    value vleft;
    value vright;

    vright = arg;

    // Move the leftmost two nodes from vright into vleft.
    vleft = vright.cutleft (2);

    // Clear the vleft value.
    vleft.clear();

    // Copy the rightmost node as an array into a newly created
    // member in vleft.
    // vleft[0][0] will contain vright[-1].
    vleft.newval() = vright.copyright (1);
}
```

Finally, you can remove individual nodes from a value object:

```
myvalue.rmval ("password");
myvalue.rmindex (-1);
myvalue.rmindex (2);
```

Attempts to address nonexistant children this way will silently fail. If you want to check if a particular child is in the set, use the **exists** method:

```
if (myvalue.exists ("update-url")
{
    runUpdateFromURL (myvalue["update-url"]);
}
```

It is advisable to use this for any optional records, specifically avoid scenarios like the one below:

```
if (myvalue["update-uri"]) // <-- Wrong!
if (myvalue["update-uri"].sval().strlen()) // <-- Also wrong!!
if (myvalue["update-uri"] != 0) // <-- Totally wrong! Stop doing this!
```

It's not that these do not work, but each of them creates the child node inside the value object, which adds the extra overhead of creating the object with no extra gain. If the value object is passed as const, asking for members that do not exist leads to unpredictable behavior or thrown exceptions.

Attributes

As stated before, a **value** can also have attributes. This is useful when you are dealing with data sourced from XML. Some forms of data layout also make the distinction between attributes and child nodes beneficial. Here is how you work them:

```
value myEmployee;
myEmployee("type") = "salaried";
myEmployee("class") = "engineer";

myEmployee["name"] = "Steve Johnson";
myEmployee["department"] = "AQ100";
```

This is the equivalent of the expression:

```
<employee type="salaried" class="engineer">
  <name>Steve Johnson</name>
  <department>AQ100</department>
</employee>
```

Of course, you can also check for the existence of attributes and remove them:

```
if (myvalue.attribexists ("removeme"))
    myvalue.rmattrib ("removeme");
```

If you want to iterate over the attributes, you can use this trusty macro:

```
foreach (attrib, myvalue.attributes())
{
    fout.printf ("attrib %s: %s\n", attrib.id().cval(), attrib.cval());
}
```

Whether or not it makes sense to use attributes in your data layout is up to you, but keep in mind that not all forms of serialization support attributes transparently.

Sorting Objects

Sometimes you want to sort the array inside a **value** object. You can perform sorts on the following criteria:

- Alphabetic sort by key
- Alphabetic sort by value
- Alphabetic sort by value of a child node with provided key
- Natural language sort by value
- Natural language sort by key of child value

Here is a run through the options:

```
myval.sort (labelSort); // Alphabetic by key.
myval.sort (valueSort); // Alphabetic by value.
myval.sort (recordSort, "name"); // Sort on value of myval[*]["name"]
myval.sort (naturalSort); // Natural sort on value
myval.sort (naturalSort, “”); // Natural sort on key
myval.sort (naturalSort, “name”); // Natural sort on myval[*][“name”]
```

The natural sort is case-insensitive and alphabetical, but it ignores words like ‘the’, ‘a’, ‘le’, ‘la’, etc. in the sorting order.

Handling CSV Formats

The **value** class can convert to and from files in the ‘Comma-Separated Value’ format. It supports a number of variations on the idea:

```
arbitrary data:
“John”,42,”Wattson”,”Developer”
“James”,14,”Pimplyface”,”Trainee”

data with headers:
“Name”,”Email”
“Steve”,”steve@initech.com"
“Dave”,”dave@initech.com"

data with headers and an id-field:
“id”,”Name”,”Email”,”AuthorizationLevel”
“john”,”John Doe”,”j.doe@organization.co.uk",10
“steve”,”Steve Wibble”,”s.wibble@organization.co.uk",25
```

The **fromcsv** and **loadcsv** methods can be used to deserialize from any of these formats, either from a string or a disk file:

```
void csvMagic (void)
{
    value in;
    in.loadcsv (“arbitrary.csv”);

    foreach (row, in)
    {
        fout.printf (“--- ROW ---\n”);
        foreach (column, row)
        {
            fout.printf (“    %s\n”, column.cval());
        }
    }

    fout.printf (“\n***\n”);
    in.loadcsv (“withHeaders.csv”, true);

    foreach (row, in)
    {
        fout.printf (“--- ROW ---\n”);
        foreach (column, row)
        {
            fout.printf (“    %s: %s\n”, column.id().cval(), column.cval());
        }
    }

    fout.printf (“\n***\n”);
    in.loadcsv (“withKey.csv”, true, “id”);

    foreach (row, in)
    {
        fout.printf (“--- ROW %s ---\n”, row.id().cval());
        foreach (column, row)
```

```
                {
                    fout.printf ("    %s: %s\n", column.id().cval(), column.cval());
                }
            }
        }
```

Writing CSV files is also possible, but you have to keep in mind the limitations:

- The value-object must be a two-dimensional array.
- If headers are to be written, rows must have a uniform column layout or will be normalized to this point.
- Attributes are discarded.

Within these bounds you can write them in all three variations, for now we'll only illustrate the third:

```
value v;

// First record
v["steve"]["Name"] = "Steve \"Wibble\" Conner";
v[-1]["Email"] = "steve@myorganization.org";
v[-1]["Home"] = "/home/steve";
v[-1]["Password"] = "*";
v[-1]["AuthLevel"] = 42;

// Second record
v["dave"]["Name"] = "Dave DeLong";
v[-1]["Email"] = "dave@myorganization.org";
v[-1]["Home"] = "/home/dave";
v[-1]["Password"] = "*";
v[-1]["AuthLevel"] = 23;

v.savecsv ("people.csv", true, "Username");
```

This will give the following output:

```
"Username","Name","Email","Home","Password","AuthLevel"
"steve","Steve ""Wibble"" Conner","steve@myorganization.org","/home/
steve","*",42
"dave","Dave DeLong","dave@myorganization.org","/home/dave","*",23
```

Loading INI Files

For some situations, windows-style INI-files are just what the doctor ordered. You can parse these files into a **value** object, but there are no methods for saving. There are two supported variations. The first one is two-dimensional:

```
license = "GPL"
owner = "john@buyer.org"

[performance]
maxthreads = 16

[customization]
bannerstring = "Welcome to the Pleasure Dome"
user = "ID:10:T"
```

Loading this into a **value** is pretty straightforward:

```
value v;
v.loadini ("myapp.ini");

fout.printf ("MyApp Licensed to <%s> id <%s>\n", v["owner"].cval(),
                                                 v["license"].cval());
spawnThreads (v["performance"]["maxthreads"].ival());

fout.writeln (v["customization"]["bannerstring"];
if (v["customization"]["user"] == "ID:10:T")
{
    setIdiotModeOn ();
}
```

The second variation is multi-dimensional, sections can contain a path with each path element separated by a colon:

```
[Alerts]
sendalerts = true

[Alerts:Subscribers:main]
routeto = "10.11.12.13"

[Alerts:Subscribers:fallback]
routeto = "10.11.12.14"
```

The **value::loadinitree** method can take care of these critters:

```
void sendAlert (const value &alertData)
{
    static value conf;

    if (! conf) conf.loadinitree ("conf:alerts.ini");
    if (! conf["Alerts"]["sendalerts"]) return;

    foreach (target, conf["Alerts"]["Subscribers"])
    {
        sendAlertMessage (target["routeto"], alertData);
    }
}
```

Native Serialization Using SHoX

The SHoX format is a binary serialization format tailored for **value** objects. Whenever you're looking for an efficient way to temporarily store or transmit data and XML adds no benefits, this format can make sense. It keeps all member data and attributes intact and saves up to 80% space over conventional XML saves and a lot over parsing overhead. There's not much to demonstrate here, the **toshox**, **saveshox**, **fromshox** and **loadshox** methods are pretty self-explanatory. The binary format is endian-safe, by the way.

The NeXT / Apple 'plist' Format

The Objective-C environment used in Mac OS X is an inheritance from the NeXT era. Grace can load and save the XML variant of this serialization format, which does not support attributes. A typical **.plist** file looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<plist>
    <dict>
        <key>foo</key>
        <string>bar</string>
        <key>answers</key>
        <dict>
            <key>ultimate</key>
            <int>42</int>
        </dict>
        <key>stupidformat</key>
        <true/>
    </dict>
</plist>
```

In the ‘native’ XML schema used by Grace, this same data would look like this:

```
<?xml version=”1.0” encoding=”UTF-8”?>
<plist>
    <dict>
        <string id=”foo”>bar</string>
        <dict id=”answers”>
            <integer id=”ultimate”>42</integer>
        </dict>
        <bool id=”stupidformat”>true</bool>
    </dict>
</plist>
```

The **loadplist**, **fromplist**, **saveplist** and **toplist** methods are actually variations on the xml-related methods that use a build-in schema for this specific format. In a later chapter we will take a closer look at the **xmlschema** class and what you can do with it.

Serializing and Deserializing PHP Arrays

If you’re communicating with systems written in PHP, Grace can convert **value** objects to its native serialization format using the **phpserialize** and **phpdeserialize** methods. There are no load/save methods for this format because this particular format ends up going over a network link or a pipe in 99% of the situations. The **phpserialize** method takes one boolean argument. If you set this to **true**, Grace will also serialize attributes, but the PHP code on the other side needs to be aware of this. In cases where a node has attributes, its attribute list will end up in an array keyed **.attrib** and any connected data will move to a child node with the label **.data**.

Simplistic ASCII Serialization

For situations where you want to exchange **value** data without attributes and XML feels like overkill, you can use the **load**, **save**, **encode** and **decode** methods that work with this simple ASCII model:

```
realName = “Pim van Riezen”
ultimateAnswer = 42
tagLines {
  += “Bad Salad”
  += “Taco Chainsaw”
}
contactHandles {
  email {
    work = “pi@panelsix.com"
    private = “pi@madscience.nl"
  }
  undernet = “exel”
  skype = “Skypical”
}
```

String Manipulation

Text processing is an important part of software. Many applications suck in massive amounts of text data and hack it up, process and twist the bits, puff some magic smoke and then spit out yet more text data. The **string** class and its associated manipulations make Grace an excellent choice for this kind of processing.

Array Access

The **string** class has regular array index operators that return a reference to the character inside the string, so code like this is perfectly valid:

```
bool parse (const string &text)
{
      if (text[0] == '!') // expect command character
      {
            string arg = text.mid (1); // Copy from second char onward.
            docommand (arg);
            return true;
      }
      return false;
}
```

You should be careful with manipulating data like this, though. String objects are guarded by a copy-on-write mechanism, which allows you to share string data between multiple objects, where an object that wants to change its data severs its ties from this shared object and creates a private copy. References to the array do not trigger this mechanism and assigning new values to them does not either. Use the **docopyonwrite()** method on such a string beforehand if you really insist on such tomfoolery:

```
void myfunction (const string &param)
{
      string arg = param;
      if (arg[0] == '@') // replace at-sign with exclamation mark
      {
            // if we don't do this, the next command will alter param.
            arg.docopyonwrite ();
            arg[0] = '!'; // change first character to excl. mark.

           // oh, and don't do this, a nul-character does not terminate a
           // string proper:
           arg[1] = '\0';

           // this will work, though:
           arg.crop (1);
      }
}
```

It's much better if you manage to express your operations using the existing manipulation methods, so let's get to those quickly.

String Matching

The simplest way of comparing a string to another string, a value or a c-string is to use the plain old '==' operator. This will perform a byte-for-byte comparison. For more complex operations, there are a number of variants of ye olde libc 'strcmp' family:

```
if (mystring == "foo") { /* old school c literal */ }
if (mystring == myvalue) { /* compare to value */ }

if (mystring.strcmp ("bar") == 0) { /* libc-style compare */ }
if (mystring.strcasecmp ("bar") == 0) { /* libc, ignore case */ }
if (mystring.strncmp ("ba", 2) == 0) { /* libc with length */ }
if (mystring.strncasecmp ("ba", 2) == 0) { /* libc, length+case */ }
if (mystring.globcmp ("ba*")) { /* unix wildcard/glob compare */ }
```

For a comparison you can also use the less-than and greater-than operators, these are case sensitive.

Finding a Sequence Inside a String

For locating specific character sequences, the string class exports libc-style **strchr** and **strstr** methods:

```
if (mystring.strstr ("therfuc") >= 0)
     ferr.writeln ("Bad word detected!");

if (mystring.strchr ('&') >= 0)
     ferr.writeln ("Bad character detected!");
```

Keep in mind that libc returns string pointers, which is impracticle in Grace, so instead you get an integer with an offset. A negative offset means nothing was found. If you want to search beyond the location where you first found something, you can provide an explicit base offset:

```
int lastSpacePos;;
int spacePos = -1;

do
{
      lastSpacePos = spacePos;
      spacePos = mystring.strchr (' ', spacePos+1);
} while (spacePos >= 0);

string lastword;
if (lastSpacePos>=0) lastword = mystring.mid (lastSpacePos+1);
```

What the code above is trying to do, by the way, is a perfect job for the **cutafterlast** method which we'll get to later.

Copying Sub-strings

You can use the position you find through these searching methods to make copies of a sub-set of the string data, for example:

```
// This function will split a string in three parts:
//   1. Everything until the first '<'.
//   2. Everything between the first '<' and the first '>'.
//   3. Everything after the first '>'.
value *copyFirstTag (const string &text)
{
      returnclass (value) res retain;
      int ltpos, rtpos;

      ltpos = text.strchr ('<');
      gtpos = text.strchr ('>');

      if ( (ltpos<0) || (gtpos<0) ) return &res;
```

```
        string left, mid, right;
        left = text.left (ltpos);
        mid = text.mid (ltpos+1, (gtpos-ltpos) -1);
        right = text.mid (rtpos+1);
        res[0] = left;
        res[1] = mid;
        res[2] = right;
        return &res;
    }
```

Copying Using Markers

The string class has a number methods that create copies of the left or right side of a string up to or after the first or last instance of a specified marker character/sequence:

```
    string mystring = “one to three four”;
    string foo;

    // foo will contain “one”
    foo = mystring.copyuntil (‘ ‘);

    // foo will contain “to three four”
    foo = mystring.copyafter (‘ ‘);

    // foo will contain “one to three”
    foo = mystring.copyuntillast (‘ ‘);

    // foo will contain “four”
    foo = mystring.copyafterlast (‘ ‘);

    mystring = “hello<br/>world”;

    // foo will contain “hello”
    foo = mystring.copyuntil (“<br/>”);
```

Cutting up Using Markers

The nondestructive copying methods of the last section have destructive counterparts that can be used to cut up one string into two using the same semantics:

```
    string mystring = “this is a test”;
    string word;

    // word will be “this”, mystring will be “is a test”.
    word = mystring.cutat (‘ ‘);

    // word will be “test”, mystring will be “is a”
    word = mystring.cutafterlast (‘ ‘);
```

Changing the Case

There are situations where you want to normalize the case of a string. The string class has methods for case normalization in two variations: One set creates a normalized copy, the second set replaces the string data inline:

```
    string mystring = “HellO woRld.”;
    string res;

    // res will be “hello world.”
    res = mystring.lower();
```

```
// res will be "Hello world."
res.capitalize();

// res will be "HELLO WORLD."
res.ctoupper();
```

Removing Unwanted Characters

There are several situations where you want to remove some unwanted characters from a string. Grace offers many methods for purging such subversive elements. Let's go over the whole set:

```
string trimme = "  ?hello!  ";
string res;

// Trim chars from the ends: res will be "?hello!".
res = trimme.trim (" ");

// Remove chars from a set: res will be "hello".
res = res.stripchar ("?!");

// set up the translation dictionary
value trans;
trans["h"] = "o";
trans["o"] = "h";

// translate: res will be "oellh".
res.replace (trans);

// translate again: res will be "hello".
res.replace (trans);


// restrict to a set: res will be "hll";
res = res.filter ("bcdfghjklmnpqtstvwxyz");
```

Cropping and Padding

In situations where you want to limit or control the size of a string, you can use the **pad** and **crop** methods to your advantage:

```
string orig = "Hello world.";
string str;

str = orig;
str.pad (15, ' '); // "Hello world.    "

str = orig;
str.pad (-15, ' '); // "   Hello world."

str = orig;
str.crop (5); // "Hello"

str = orig;
str.crop (-6); // "world."
```

Formatting and Encoding

The **printf** method accepts formatted text to append to your string. It mostly follows the libc conventions, but there are a couple of exceptions:

```
string input = "<hello>";
string output;

// 1. Encoding for XML
// &lt;hello&gt;
output.printf ("%Z", input.str());

output.crop ();
input = "The \"string\" escape";

// 2. Traditional backslash-escape style.
// "The \"string\" escape
output.printf ("\"%S\"", input.str());
```

In some cases you just want to do some explicit encoding/decoding, there are some functions for that as well:

```
string base64 = mystring.encode64();
string back = base64.decode64();

// inline backslash-escape
mystring.escape();
mystring.unescape();

// inline xml-escape
mystring.escapexml();
mystring.unescapexml();
```

There's one more codec hiding inside the **strutil** class:

```
#include <grace/strutil.h>

void foo (void)
{
   string in = "Testing & Trying";

   // out will be "Testing+%26+Trying"
   string out = strutil::urlencode (in);

   // back will be "Testing & Trying" again
   string back = strutil::urldecode (out);
}
```

This performs the common URL-encoding as is extensively used in web-environments.

Splitting Strings

There are more useful functions inside the **strutil** class to split a string into a **value** array where you can do further processing.

```
string words = "read my lips";
value wordlist = strutil::splitspace (words);

words = "one|two|three";
wordlist = strutil::split (words, '|');

words = "one two \"two plus one\" four";
wordlist = strutil::splitquoted (words, ' ');

words = "one\ntwo\nthree";
wordlist = strutil::splitlines (words);
```

```
words = “1,\”john\”,133”;
wordlist = strutil::splitcsv (words);
```

Regular Expressions

Executing a regular expression on a string is also simple:

```
string in = “java language considered beneficial”;
string out = strutil::regexp (in, “s/beneficial/hazardous/”);
```

If you are reusing regular expressions in a more than casual fashion, you may also want to look at the **regexpression** class.

Parameter Parsing

If formatting with printf-style arguments is unpractical (for example, because you’re dealing with user input that doesn’t use pre-defined types), it can be convenient to create a string with a value-object as its parameter. The **strutil::valueparse** function can help you with this:

```
void doSomething (const value &v)
{
    string out, tmpl;

    tmpl = “Hello, $firstname$.\nYou are visitor number $#visitor$\n”;
    tmpl = strutil::valueparse (tmpl, v);
}
```

The variables posted between dollar-signs will be replaced by the variable inside the supplied value object. You can give a number of formatting hints, although these are not as rich as those offered by printf, they can help you with escaping and type-safety:

- A number sign will represent the variable as an integer: **$#varname$**
- A slash will escape the string using backslash-escape: **$/varname$**
- A circumflex will escape the string for literal inclusion in html: **^varname**
- A tilde will trigger an indirection to var[var[“varname”]]: **$`varname$**
- A pair of colons can be used for sub-nodes: **$varname::subvarname$**

This form of variable parsing is also used inside the HTML templating system, which will be further documented in a later chapter.

Creating an Application

The application is a basic program unit built with the Grace library. It includes an executable and some extra meta-data like XML schemas and definitions for command line argument parsing.

Inside the Application Bundle

Grace applications are created inside a *bundle* directory. This is a concept from the NeXT/Apple camp, where an executable and all related resources are kept inside a directory with the **.app** extension. The build tools also create a softlink outside the **.app** pointing to only the executable. A basic application bundle has the following layout:

```
myApplication.app/
  exec -> Contents/Linux.i386/myApplication.exe
  Contents/
    Configuration Defaults/
      com.initech.myapplication.conf.xml
    Linux.i386/
      myApplication.exe
    Resources/
      grace.runoptions.xml
      resources.xml
    Schemas/
      com.initech.myapplication.schema.xml
```

Grace applications resolve the path to their bundle directory on startup. If a configuration file is missing on the system it can load the default configuration file from inside the bundle. The **grace.runoptions.xml** file controls the parsing of command line argument options. The **resources.xml** file can be used for storing behavioral flags, text messages or other overridable parameters that are not really part of a common configuration.

The **mkapp** utility can build a **.app** bundle from a Grace project directory. It will collect all resources and your executable and create the proper directory structure.

Creating a Project Directory

The **mkproject** tool creates a project directory and a template for your application development. Projects can currently come in two flavors: A project with either **application** or **daemon** templates. You assign both a project name and a project *domain*. The latter is used to prevent collisions between application names from different developers. It should be a reverse order domain path that distinguishes your development effort from those of others. If your DNS domain name is 'jimsdevshack.com', you could use something like com.jimsdevshack.tools as a domain for a simple tool you're developing.

Some usage examples:

```
$ mkapp MagicStorple com.jimsdevshack.tools
Project <com.jimsdevshack.tools.magicstorple> created

$ mkapp -d MyWebService com.jimsdevshack.svc
Project <com.jimsdevshack.svc.mywebservice> created
```

The first example creates an application template for a tool called 'MagicStorple', the second one creates a daemon template for a service called 'MyWebService'. This is what's inside a newly created project directory:

```
com.jimsdevshack.tools.magicstorple/
  MagicStorple.app/
  Makefile
  configure
  configure.in
  magicstorple.h
  main.cpp
  rsrc/
    grace.runoptions.xml
```

The **magicstorple.h** and **main.cpp** contain some template code you can edit.

Defining Command Line Arguments

The file **grace.runoptions.xml** inside **rsrc** controls parsing of command line arguments. This is what the file could look like:

```
<?xml version="1.0" encoding="utf-8"?>
<grace.runoptions>

  <!-- Mapping from short to long options -->
  <grace.option id="-h">
    <grace.long>--help</grace.long>
  </grace.option>

  <grace.option id="-v">
    <grace.long>--verbose</grace.long>
  </grace.option>

  <grace.option id="-o">
    <grace.long>--outfile</grace.long>
  </grace.option>

  <grace.option id="-t">
    <grace.long>--type</grace.long>
  </grace.option>

  <!-- The hard-wired --help option -->
  <grace.option id="--help">
    <grace.argc>0</grace.argc>
    <grace.help>This help information</grace.help>
  </grace.option>

  <!-- An optional flag -->
  <grace.option id="--verbose">
    <grace.argc>0</grace.argc>
    <grace.help>Verbose output</grace.help>
  </grace.option>

  <!-- An option with a parameter -->
  <grace.option id="--outfile">
    <grace.argc>1</grace.argc>
    <grace.help>Send output to a file instead of stdout</grace.help>
  </grace.option>

  <!-- An option with a parameter and a default value -->
  <grace.option id="--type">
    <grace.argc>1</grace.argc>
    <grace.default>normal</grace.default>
    <grace.help>The type of processing</grace.help>
  </grace.option>

</grace.runoptions>
```

The **application** class will use the hints in this file to parse the command line parameters into a **value** object called **argv**. If confronted with the '--help' flag, it will create a help message from the information in this file. For the other scenarios, this is what handling of the command line arguments would boil down to:

```
int myApp::main (void)
{
    bool useStandardOut = true;
    bool useStandardIn = true;
    bool verboseOutput = false;
    string outputFileName;
    string inputFileName;
    string processingType;

    if (argv.exists ("--outfile"))
        outputFileName = argv["--outfile"];

    if (argv.exists ("--verbose"))
       verboseOutput = true;

    processingType = argv["--type"];

    // The '*'-node contains any arguments that did not fit the runoptions.
    if (argv["*"].count())
    {
        useStandardIn = false;
        inputFileName = argv["*"][0];
    }

    file inFile;
    file outFile;

    if (useStandardIn) inFile = fin;
    else if (! inFile.openread (inputFileName)) return 1;

    if (useStandardOutput) outFile = fout;
    else if (! outFile.openwrite (outputFileName)) return 1;

    return doWork (inFile, outFile, processingType, verboseOutput);
}

int myApp::dowork (file &fi, file &fo, const string &type, bool verbose)
{
    ...
    return 0;
}
```

This code also illustrates the good habit to use your **main** method for initialization and processing and defining a separate work method for performing the actual job.

Path volumes

An annoying aspect of programming in a Unix environment is how hard it is to be sensible about paths, and many applications suffer from compiled-in assumptions about the location of certain resources. Grace prevents these problems on two levels. The first line of defense is the application bundle, which gets rid of predfined paths for static but external resources. The second line of defense is the path volume system, which offers a transparent overlay for collections of distinct directory paths on the filesystem that have similar functionality.

A volume path distinguishes itself from a normal path by its use of the colon for its first element, like **rsrc:resources.xml** or **log:myapp/event.log**. This first element is the volume. Volumes are determined at runtime, either relative to the application or the operating system environment. Users can define volumes as well, or you can define custom volumes from within your application for convenience.

The major point of a volume is that it can represent multiple paths as an overlay. Whenever you open a file for reading that uses a volume path, Grace will go left-to-right through all the paths defined for the volume until it finds the matching file. When opening a file for writing, Grace will try the paths in the volume right-to-left. This allows your application to combine default system configuration and/or plugins while allowing individual users to keep their private configuration files and plugins.

Grace applications have access to the following path volumes (paths are only set for the volume if they exist):

Volume	Paths	Description
`app:`	`determined at runtime`	The location of the .app bundle directory.
`home:`	`copied from $HOME`	The user's home directory.

`homes:`	`one of:`
	`/Users`
	`/home`
	`/Home`
	`/usr/people`
		The location of user home directories. Systems with eccentric layouts should set the _PATH_HOMES environment variable.

`library:`	`/Library`
	`/usr/share`
	`/usr/local/share`
	`homes:``Shared/Library`
	`home:``.library`
		A location for potentially shareable resources for similar application types.

`var:`	`/var`
	`home:``var`
		The traditional Unix /var

`log:`	**`var:`**`log`	Log file storage
`run:`	**`var:`**`run`	Storage for runtime information files.

`tools:`	`everything in $PATH`
	`app:``Contents/Tools`
		This represents the location of executable Unix binaries, as represented by the PATH environment variable, with the addition of fallback tools that may be shipped inside the application bundle.

`rsrc:`	**`app:`**`Contents/Resources`	Application resource files
`schema:`	**`app:`**`Contents/Schemas`	XML Schema and validator files

`conf:`	**`app:`**`Contents/Configuration Defaults`
	`/etc/conf/`**`$APPID`**
	`home:``Library/Preferences/`**`$APPID`**
	`home:``.conf/`**`$APPID`**
		Location for configuration files. The APPID is determined by the constructor of the **application** object.

All Grace classes dealing with files will automatically translate path volumes. For external APIs they can be resolved through the global **fs** object that will be explained in a later chapter. Let's just say that there's no stopping you from using a user-defined alias **pr0n:** for locating downloaded high-resolution multi-angle content spanning several high-volume harddisks.

Unix Environment Variables

All environment variables are available inside the **value** object called **env** inside the application object:

```
int myApp::main (void)
{
    if (env.exists ("EDITOR"))
        fout.printf ("You vile %s user!\n", env["EDITOR"].cval());
    return 0;
}
```

Manipulation of this **value** object will not alter the system environment. Check the documentation for creating new processes if you want to know how to feed your environment to a subprocess.

Handling Files and Directories

One of the core tenets of the Unix philosophy is its blending of multiple concepts on the idea of a file and a filesystem. The **file** and **filesystem** classes help you in getting data from and about objects inside the available filesystem namespace.

Reading Data from a File

A file object is either passed to you, or unopened. You can use the **openread** command to bind the file object to an object on the filesystem for reading. You can use **gets** to read a line of text (terminated by either \n or \r\n):

```
#include "myapp.h"
#include <grace/file.h>

int myApp::main (void)
{
    file f;
    if (! f.openread ("input.txt"))
    {
        ferr.printf ("Cannot open\n");
        return 1;
    }
    try
    {
        while (! f.eof())
        {
            line = f.gets();
            fout.writeln (line);
        }
    }
    catch (...)
    {
        ferr.printf ("%% File exception\n");
    }
    f.close();
    return 0;
}
```

Remember that methods in the **file** class will throw an exception on input/output errors, so it's best to wrap these in a try block. Reading data after the **eof** condition is set will also earn you an exception.

Not all data consists of lines of text, you may find yourself in situations where unreasonable people demand you involve yourself in evil binary files, and that's where the **read** method comes in handy. Here's a demonstration that also shows off the **md5checksum** class:

```
#include "md5sum.h"
#include <grace/file.h>
#include <grace/md5.h>

int md5sumApp::main (void)
{
    md5checksum md5; // The md5 checksum accunulator.
    file in; // The file object we'll use for input.
    string block; // String to store blocks we read.
    string digest; // The generated MD5 digest.
```

```
        // First command line argument.
        string fname = argv["*"][0];

        // Was a filename provided on the command line?
        if (fname)
        {
            // If we can't open it, we'll call it quits.
            if (! in.openread (fname))
            {
                ferr.printf ("Cannot open: %s\n", fname.str());
                return 1;
            }
        }
        else in = fin; // No argument, use stdin.

        try
        {
            while (! in.eof())
            {
                line = in.read (4096);
                md5.append (line);
                if (line.strlen() < 4096) break;
            }
        }
        catch (...) { }

        digest = md5.hex (); // Get MD5 sum as hexadecimal
        fout.writeln (digest); // Output the checksum

        return 0;
    }
```

As you can see, you can assign one **file** object to another using the '=' operator. The **fin**, **fout**, and **ferr** objects, as you may have guessed, are the standard input, output and error channels.

Writing Data to a File

There's something to be said for being able to write to a file as well, of course. Some of the writing-methods you have already met, but let's have a quick tour:

```
    file f;
    if (! f.openwrite ("out.txt"))
        return false;

    try
    {
        f.writeln ("Unformatted, newline will be added");
        f.printf ("Formatted, like %s\n", "string::printf");

        // Let's build a bit of XML in a string.
        string bar;
        value tmp;
        tmp["foo"] = "bar";
        tmp["answer"] = 42;
        bar = tmp.toxml ();

        // Write the string to the file.
        f.puts (bar);
    }
```

```
catch (...)
{
    ferr.writeln (“I/O Error”);
}
f.close ();
```

Note that there is no **write** method. You can always use a **string** object as a write buffer, since these may contain binary data. Keep in mind that write-operations can also throw exceptions if there are i/o errors, hence the try-block.

Getting a Directory Listing

Wibble.

Resolving File Paths

Wobble.

The Filesystem Shell

Wubble.

Loading and Saving Blobs

Hurray fs.load().

Basic Network Communication

Introduction.

Connecting to a TCP Host

All the gory details.

Connecting to a Unix Domain Socket

More details.

Listening to a TCP Port

More lead by example.

The Global kernel Object

Intro about the system class.

Process Control

Yadda

System User Database

...

Password Encryption

...

Time Keeping

...

Network-related Information

...

Multithreading

Introduction

The thread Class

…

Sending Events to a Thread

…

Grouped Threads

…

Locked Data

…

Synchronization

…

Conditionals

…

The daemon Class

Some background innfo,

Creating the Project Directory

…

Defining The Configuration File

…

About Logging

…

Configuration Handlers

…

Keeping to the Foreground

…

Shutting Down

…

Creating a Simple XML Schema

Introduction about XML schemas.

Making an XML Template

...

Building a Schema and Validator

...

The HTTP Client

Intro

Getting Data from a URL

…

Posting Data

…

Headers and Options

…

Using SSL/HTTPS

…

The HTTP Service

Introduction

Implementing a URI Handler

…

Basic Authentication

…

Custom Authentication

…

Logging

…

Serving Files

…

Handling Filetypes

…

Error Documents

…

Redirects

…

Simple Virtual Host Handling

…

An Example Web Service

…

The SMTP Client

Introduction

Preparing a Message

…

Sending a Message

…

The SMTP Service

qweqwe

Address Verification

…

Delivery Routing

…

Handling a Message

…

Managing Child Processes

qweqwe

Running a System Process

…

Creating a process Class

…

Advanced File I/O

Non-blocking methods and methods with timeouts.

Reading Data with Timeouts

…

Writing Data with Timeouts

…

Advanced XML Schemas

Introduction

Using Tag Types as Index

…

Mapping Types and Using Attribute Keys

…

Mixing Tag and Attribute Keys

…

Wrapping the Value

…

Container Arrays

…

Advanced Wrapping

…

More Wrapping Madness

…

Push Access Protocol Example

…

Boolean Values as Objects

…

Value Validators

Introduction

Using the Validator Class

blabla

The <match.data> Object

…

The <match.id> Object

…

The <match.child> Object

…

The <match.rule> Object

…

The <match.attrib> Object

…

The <match.mandatory> Object

…

HTML Template Processing

Introduction…

Variables

…

Sections

…

Loops

…

Conditionals

…

Case Selection

…

Using Templates in HTTP

…

Creating a Command Line Interface Shell

Werwer

Setting Commands

…

Setting the Prompt

…

Adding Help Information

…

Defining Dynamic Data Sources

…

Binding Keys

…

The dictionary Template Class

qweqwe

The array Template Class

Werwerwer

DNS Resolution

Werwer

Runtime Tunable Parameters

Werwer

Timing and Initial Buffer Sizes

…

System Limits

…

Appendix A: The mkproject Tool

qweqwe

File Matching

…

Appendix B: The mkapp Tool

qwewqe