Using the hush ADT's

by SAM
sam@cs.vu.nl

version 1.0

June 20, 1996

1 Introduction

This document will explain the basic usage of the most useful hush ADT's: strings, lists and dictionaries. Every ADT is explained using an example that will show the usage of the most common member functions. A lot of these ADT's are based on templates. Inexperienced C++ programmers shouldn't panic now: using these classes doesn't require any knowledge about templates. If you just follow the syntax used in the examples, you'll be alright.

2 Strings

The class `string` is the replacement of `char*`. It contains the usual operations on strings, but without the usual administration that comes with using `char*`.

The example:

```
// string.cc
// Example of use of class string

#include <hush/string.h>
#include <iostream.h>
#include <stdio.h>


int main()
{
    char buf[128];                  // buffer used for sprintf
    string s1;                      // creating an empty string

    s1 = "hello world!";            // setting a string
    cout << "s1 = " << s1 << endl;  // printing a string

    sprintf(buf, "s1 as char*: %s", (char*) s1);
                                    // casting to (char*) (can also
```

```
                                // be done with (const char*))
    cout << buf << endl;

    string s2;                  // another empty string

    s2 = s1;                    // copying a string

    cout << "s1 = " << s1 << ", s2 = " << s2 << endl;

    cout << "length of s1 = " << s1.length() << endl;
                                // retrieving the length of a string

    char ch = s1[5];
    cout << "the 6th character of s1 = " << ch << endl;
                                // character indexing

    if (s1 == s2)               // comparison between 2 strings
        cout << "s1 is equal to s2" << endl;
    else
        cout << "s1 is not equal to s2" << endl;

    if (s1 == "Hello World!!")  // comparison between string and char*
        cout << "s1 is equal to Hello World!!" << endl;
    else
        cout << "s1 is not equal to Hello World!!" << endl;

// suppose we got a list of something contained in a string:

    string stritems = "item1 item2 item3 item4 item5";

    cout << "stritems = " << stritems << endl;

// we can iterate over this list in the following way:

    stritems.split(" ");        // split the string in seperate substrings,
                                // using ' ' (space) as separator
    cout << "stritems = " << stritems << endl;
                                // string stays the same
    iter<string> striter = stritems;// create an itererator over strings, more
                                // specifically, the list we just declared
    string* strptr = 0;         // IMPORTANT!!! this is A POINTER to a
                                // string that will be used for iteration
    while (strptr = striter())
    {
        cout << "item: " << *strptr << endl;
    }

    return 0;
}
```

2.1 Iterators

Most of the example should be obvious. The use of iterators requires some explanation. An iterator is an object that is used to iterate over all elements of some other object. The iterator hides the exact structure of this other object. It doesn't matter whether the other object has a list structure, a tree structure, an acyclic graph structure or something else: the iterator is always used in the same way. With the hush ADT's, an interator is created in the following manner:

iter<*element-type*> *iterator-name* = *object-to-be-iterated-over* **;**

The *element type* reflects the type of elements the iterator will return: iterating over a 'split' string will result in seperate strings, iterating over a list of integers will result in seperate integers, etc. The assignment from an object to an iterator tells the iterator over which object it should iterate. Each time the function `operator()` is called, it will return *a pointer to* the next element. When there is no next element, `operator()` will return a null-pointer (0 or `NULL`).

3 Unordered lists

The example:

```
// list.cc
// Example of use of list

#include <hush/list.h>
#include <iostream.h>


// ALWAYS pass complex value parameters with const& !
void printlist(const list<int>& l)
{
    iter<int> listiter = l;             // see explanation in main
    int* intptr = 0;

    while (intptr = listiter())
    {
        cout << *intptr << " ";
    }
    cout << endl;
}


int main()
{
    list<int> l1;                       // create an empty list of ints
    int i1 = 27, i2 = 3, i3 = 81;       // some integers

    cout << "length of l1 is " << l1.length() << endl;
                                        // retrieving the length of a list

// insert some elements
    l1.insert(&i1);                     // IMPORTANT: list stores only
                                        // POINTERS to elements, so something
                                        // like l1.insert(3) won't work!
                                        // see the effect of this later on
```

```
    l1.insert(&i2);
    l1.insert(&i3);

    cout << "length of l1 is " << l1.length() << endl;

// now we will iterate over the whole list:
    iter<int> listiter = l1;            // create an iterator for a list, more
                                        // specifically, an iterator for our
                                        // created list
    int* intptr = 0;                    // this POINTER to an integer will
                                        // be used for the iteration
    while (intptr = listiter())
    {
        cout << "item = " << *intptr << endl;
    }

    cout << "2nd element of l1 = " << *(l1[1]) << endl;
                                        // retrieving an element by index

// an example of datastructure sharing between lists:
    list<int> l2 = l1;                  // creates an alias of l1

    printlist(l2);

    int* eltptr = l2.head();            // retrieve the first element
    *eltptr = 7;                        // change it
    printlist(l2);                      // of course, l2 has changed, but
    printlist(l1);                      // the elements in l1 have changed too!

// since lists only store POINTERS to elements, the elements that were added
// will be changed:
    cout << "i1 is now " << i1 << endl;

// and vice versa:
    i2 = 128;
    printlist(l1);

// show effect of aliasing:
    l1.remove(l1.head());               // remove some elements
    l1.remove(l1.head());
    printlist(l1);                      // l1 is a lot shorter
    printlist(l2);                      // l2 changes too

    return 0;
}
```

4 Ordered lists

Ordered lists have the same member functions as unordered lists, but elements are inserted in some (user-defined) order. Because some ordering is required, you have to provide member functions to enforce an ordering if you want to use your own elements in an ordered list. These functions are `operator<` and `operator==`. Some other 'external' functions has to be provided also to print an element to a stream, see the `operator<<` in the example.

The example:

```
// olist.cc

#include <hush/list.h>
#include <iostream.h>


// define our own class which can be used in the ordered list
class myclass
{
public:
    myclass(int i) { i_ = i; }
    int value() const { return i_; }
    int operator<(const myclass& m) const { return (i_ < m.value()); }
    int operator==(const myclass& m) const { return (i_ == m.value()); }

private:
    int i_;
};

// you MUST define a stream-operator for your own class (but it's only used
// when you evoke 'print' on an ordered list):
ostream& operator<<(ostream& os, const myclass& mc)
{
    os << mc.value();
    return os;
}

int main()
{
    myclass a(27), b(3), c(72);
    olist<myclass> l1;                  // create an ordered list with
                                        // elements of our own class

    l1.insert(&a);                      // add elements in unsorted order
    l1.insert(&b);
    l1.insert(&c);

    iter<myclass> mciter = l1;
    myclass* mcptr = 0;

    while (mcptr = mciter())
    {
        cout << "item = " << mcptr -> value() << endl;
    }

    l1.print(cout);                     // print the entire list on cout,
                                        // great for debugging

    return 0;
}
```

5 Dictionaries

A dictionary is a mapping between strings (`char*`) and some (user-defined) type. An instance of a type is stored with a string as 'index' and retrieved with that string. Dictionaries are very handy for storing objects by tags. The example should speak for itself.

The example:

```
// dictionary.cc

#include <hush/dictionary.h>
#include <hush/string.h>
#include <iostream.h>

// define our own class which can be used in the dictionary
class myclass
{
public:
    myclass(int i) { i_ = i; }
    int value() const { return i_; }

private:
    int i_;
};


int main()
{
    myclass a(27), b(3), c(72);         // create some instances of myclass

    dictionary<myclass> dict;           // create an empty dictionary that
                                        // will contain instances of myclass

// store some objects (only POINTERS are stored!)
    dict["foo"] = &a;
    dict["bar"] = &b;
    dict["cow"] = &c;

    if (!dict["apple"])
        cout << "no such thing" << endl;

    dict["apple"] = &c;
    dict["car"]   = &b;
    dict["event"] = &a;

    myclass* mcptr1 = dict["foo"];
    myclass* mcptr2 = dict["bar"];

// retrieve some objects by name:
    cout << "foo: " << mcptr1 -> value() << endl;
    cout << "bar: " << mcptr2 -> value() << endl;

// print the complete dictionary:
    iter<string> it = dict;             // create an iterator over strings
```

```
    string* strptr = 0;                 // pointer to myclass for iteration
    myclass* mcptr = 0;

    while(strptr = it())
    {
        mcptr = dict[*strptr];
        cout << *strptr << " = " << mcptr -> value() << endl;
    }

    return 0;
}
```