

# Programming with Haiku

## Lesson 5

Written by DarkWorm



Let's take some time to put together all of the different bits of code that we've been learning about. Since the first lesson, we have examined the following topics:

- Templates
- Namespaces
- Iterators
- The C++ string class
- The STL Associative containers: map, set, multimap, and multiset
- The STL Sequential containers: vector, deque, and list
- The STL Container Adapters: queue and priority\_queue
- C++ input and output streams, a.k.a cout and friends
- Exceptions

There is enough material here that a book could be written to get a good, strong understanding of effective use of the Standard Template Library and the Standard C++ Library. Expert use is not our goal for this context, but having a good working knowledge of these topics will help make our code better when writing applications for Haiku.

### ***Project: Reading Paladin Projects***

For those unfamiliar, Paladin is one of the Integrated Development Environments available for Haiku. It was designed to have an interface similar to BeIDE, which was the main IDE for BeOS back in the day. One feature that sets it apart from other IDEs for Haiku is that its unique project file format is a text file with a specific format. This makes it possible to easily migrate to and away from it.

The format itself is relatively simple. Each file is a list of key/value entry pairs with one entry per line. This makes it possible to read and interpret the project file progressively. It also increases forward compatibility because new keys can be ignored by older versions of the program.

This project will be a cross-platform project reader. We will be using the Standard C++ Library along with the STL to ensure that it can be used on just about any operating system. The main use of this project would be to serve as a start for a program to convert Paladin projects to makefiles, Jamfiles, shell scripts, or other build systems.

```
#include <fstream>
#include <iostream>
#include <list>
#include <map>
#include <string>
#include <strings.h>
#include <vector>

// Because we're doing so much in the std namespace, this will save
// us a *lot* of extra typing.
using namespace std;

// While these could -- and probably should -- be classes, we will just
// use structures for the sake of space and simplicity.
typedef struct
{
    // File paths can be stored as either absolute paths, obviously
```

```

    // beginning the entry with a /, or as a path relative to the
    // project file's location.
    string path;

    // Dependencies are stored as a list of file paths separated by
    // pipe symbols (|).
    string dependencies;
} ProjectFile;

typedef struct
{
    string name;
    bool expanded;
    vector<ProjectFile> files;
} ProjectGroup;

// The actual Project class used by Paladin is much more complicated
// because it also has some properties for maintaining state while the
// program is running, but this structure has all of the data that is used
// for building and maintaining a project.
typedef struct
{
    map<string,string>                properties;

    vector<string>                    localIncludes;
    vector<string>                    systemIncludes;
    vector<ProjectGroup>              groups;
    vector<string>                    libraries;
} PaladinProject;

int
ReadPaladinProject(const char *path, PaladinProject &proj)
{
    // Create an Input File Stream for reading the file
    ifstream file;

    file.open(path, ifstream::in);
    if (!file.is_open())
    {
        // endl is a cross-platform stand-in for an end-of-line
        // character. The EOL character is different on Windows,
        // Haiku/Linux/UNIX, and OS X and this saves us from
        // having to figure it out without sacrificing portability.
        cout << "Couldn't open the file " << path << endl;
        return -1;
    }

    // Empty the project's data to make sure we're not building
    // upon existing baggage.
    proj.properties.clear();
    proj.localIncludes.clear();
    proj.systemIncludes.clear();
    proj.groups.clear();
}

```

```

while (!file.eof())
{
    string strData;

    // While the fstream class has a getline() method, it only
    // works on regular strings. There is a global getline()
    // function in <string> which reads data from a stream into
    // a C++ string. This is the version that we will use.
    getline(file, strData);

    // An empty line shouldn't exist in a Paladin project, but
    // let's handle the case just to prevent headaches.
    if (strData.empty())
        continue;

    size_t pos = strData.find('=');

    // npos is the maximum size for a C++ string. find() will
    // return npos if the string searched for is not found.
    if (pos == string::npos)
        continue;

    string key = strData.substr(0, pos);
    string value = strData.substr(pos + 1, string::npos);

    if (key.compare("GROUP") == 0)
    {
        // Using vectors saves us from having to worry about
        // memory management and pointers. Instead, all that we
        // have to do is create a new group on the stack which
        // will be used to initialize the new element in the
        // groups vector.
        ProjectGroup newGroup;
        newGroup.name = value;
        proj.groups.push_back(newGroup);
    }
    else if (key.compare("EXPANDGROUP") == 0)
    {
        if (!proj.groups.empty())
            proj.groups.back().expanded =
                strcasecmp(value.c_str(), "yes");
    }
    else if (key.compare("SOURCEFILE") == 0)
    {
        // Quite a few dots are used to create the new file, but
        // that's OK. It sure beats messing around with pointers.
        ProjectFile newFile;
        newFile.path = value;
        proj.groups.back().files.push_back(newFile);
    }
    else if (key.compare("DEPENDENCY") == 0)
        proj.groups.back().files.back().dependencies = value;
    else if (key.compare("LIBRARY") == 0)
        proj.libraries.push_back(value);
    else
        proj.properties[key] = value;
}

```

```

        return 0;
    }

    int
    main(int argc, char **argv)
    {
        PaladinProject project;

        if (argc == 2)
            ReadPaladinProject(argv[1], project);
        else
            cout << "Usage: " << argv[0] << " <path>\n";

        map<string,string>::iterator i;
        for (i = project.properties.begin(); i != project.properties.end();
             i++)
            cout << i->first << ": " << i->second << endl;

        return 0;
    }

```

## Going Further

- Use this project as a starting point for printing information about a Paladin project.
- Create a program which reads a Paladin project and spits out a makefile or Jamfile.

## Unit 1 Review Answers

### Lesson 1

1. Templates are a way of making classes and functions work with types in a generalized way.
2. Templates are most often used to create container classes.
3. A template-based class declaration is preceded by the `template <class T>` declaration placed before the `class MyClass` line. A template-based function places this text before the rest of the function's declaration.
4. Function definitions which use templates must be placed in a header file because the templates are generated at compile time. If this is not done, linker errors ensue.
5. There are actually two main differences between deques and vectors. First, it is not safe to use pointers to refer to the elements in a deque, unlike a vector. Second, it is possible to add elements only to the back of a vector, whereas elements can be added to either the back or front of a deque.
6. The main drawback of the list container is that it is not possible to quickly access an arbitrary element – getting to a specific item involves iterating from its beginning.
7. The Standard Template Library uses the `std` namespace.
8. The `using` keyword allows you to refer to members of a namespace without a prefix.

### Lesson 2

1. The `map` container stores elements as key-value pairs. The `set` container is different in that the values are the same as the keys.

2. `equal_range()` returns two iterators which refer to all items matching a value in a `multiset` or `multimap`.
3. FIFO refers to **F**irst-**I**n, **F**irst-**O**ut processing, which is where the items that are the first to go into a container are also the first to leave it, much like a line at a movie theater. LIFO stands for **L**ast-**I**n, **F**irst-**O**ut processing, where the last items into a container are the first to leave. The Towers of Hanoi puzzle exemplifies this.
4. The difference between a queue and a `priority_queue` is that the item with the highest priority is the first item out of a `priority_queue`, which does not necessarily translate to the the first item in. A regular queue is strictly FIFO.

### *Lesson 3*

1. The four standard C++ streams are `cin`, `cout`, `cerr`, and `clog`.
2. `fail()` usually applies to the failure of an individual call, such as a write fail. `bad()` is normally true when there is an error condition in the state of things that applies to more than just an individual method failure.
3. `endl` should be used instead of just `'\n'` because `endl` is portable across operating systems – not every operating system uses `'\n'` as its end-of-line character.
4. The call stack is the series of nested function calls in a program, starting with `main()`.
5. The `boolalpha` formatter translates boolean values to "true" and "false" strings.
6. Exceptions are not normally used in Haiku for performance reasons – unrolling the call stack is slow and the error handling provided by the API is normally sufficient.

### *Lesson 4*

7. The Subversion project was started to correct the problems inherent in the CVS version control system.
8. Setting your name and e-mail address in the `~/.hgrc` file must be done before doing version control with Mercurial.
9. If the `EDITOR` shell variable has been set, Mercurial will use it to edit commit messages.
10. The `hg init` command creates a Mercurial repository.
11. `hg commit` checks in changes to the local repository. `hg push` sends them from the local repository to a remote one.
12. `hg revert` undoes the changes made to one or more files since the last check-in. It can also undo changes back to a specific revision.