Learning to Program with Haiku

Lesson 14

Written by DarkWyrm

If you've been waiting excitedly for the chance to learn how to write a program that shows something on the screen instead of just printing stuff on the Terminal, today's the day! We will spend most of our time understanding the code, but let's talk about tools and organization first.

Tools of the Trade: Integrated Development Environments

All of our programs so far have been simple. Console programs, as they are called, don't have to be, but to learn the language, simple is good. A text editor is all we've needed to do these simple programs which can all fit into one file. Regular applications can have hundreds of files. Many experienced C++ programmers are content with a text editor and Terminal, but having more isn't a bad thing, especially for novice coders.

For the rest of these lessons, you will be encouraged – but not required, however – to use an Integrated Development Environment (IDE). IDEs tie together the compiler, the code editor, the debugger, and other tools so that a developer can work more efficiently. For the purposes of these lessons, use of the Paladin IDE for Haiku will be assumed. If you prefer using another one, that's just fine. Let's get started.

Our First Project

Run Paladin and you will be greeted by the start window. Choose to create a new project. At the top of the New Project window is a menu where you can choose what kind of project to create. Choose Empty Application. While we could choose GUI with Main Window to automate what we are about to do, the act of typing the source code will help us learn the basics of Haiku GUI programming better. Set your project name to whatever you like. The Target Name is the name that the executable will have when your project is built and is often the same as the project name. When you're ready to move on, click Create Project.

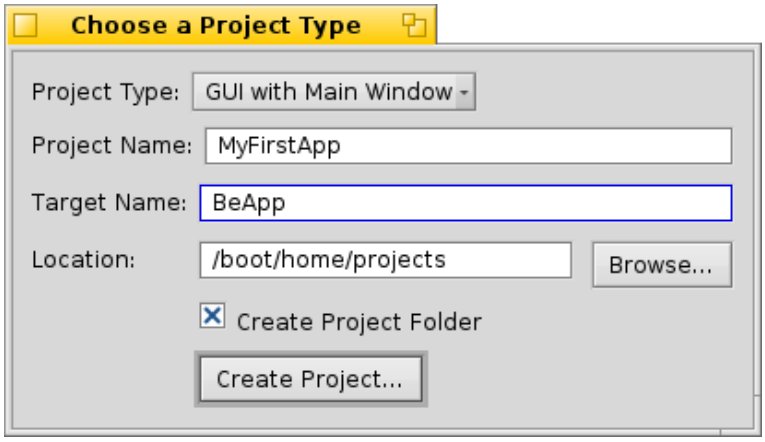


Illustration 1: The Paladin New Project window

As of this writing, if you are using a GCC4-based build of Haiku, you will need to add a library to your project to be able to build it properly. In your project's window, choose Change System Libraries from the Project menu. Scroll down and check the box beside libsupc++.so. Close the window and from the Project menu, choose Add New File. When asked the name of your file, enter App.cpp, check the box to create the corresponding header file, and click OK.

In general, there is one class per pair of files and the base of the files' names is the same as the class that goes in it. Our first file, App.cpp will have one class: App. The definition for the App class will go into the header file App.h. All of the code for the class will go into App.cpp. If we had a couple of really small related classes, it would be acceptable to put them both into the same file, but the general rule of thumb is one class per file pair for organization's sake.

Let's start with adding what we need to App.h. Click ONCE on App.cpp in the project window to select it, hold Alt and press Tab. You'll notice that App.h appears in the editor. The Alt-Tab key combination is a fast way of jumping between a file's header and its source file, but it only works if both files have the same base name, such as App.cpp and App.h. Start by entering the code below:

```
// This is what is called the header guard. It prevents the header from being
// included more than once, which can cause build errors. If you're using
// Paladin, this was entered for you. What you define isn't terribly important,
// but it should be unique to each header file in your project, so following this
// convention is probably a good idea.
#ifndef APP_H
#define APP_H

// This header holds the definition for the BApplication class which we will
// need for our project
#include <Application.h>


class App : public BApplication
{
public:
	App(void);
};

// This line ends the header guard and should always be the last line in our
// header file.
#endif
```

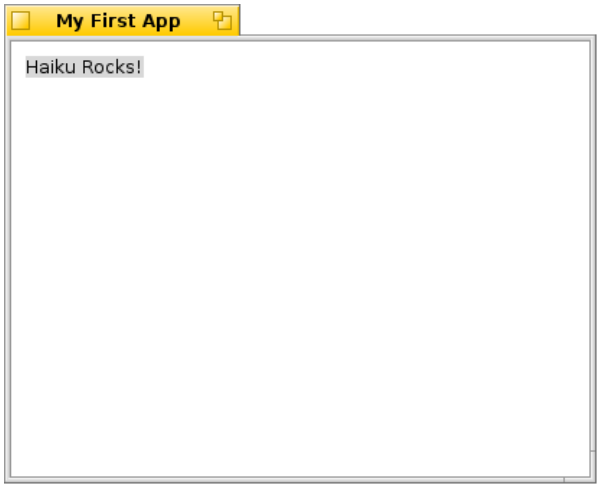
Every GUI program in the Haiku operating system creates a subclass of BApplication. This class sets up communications with the app_server, the part of the system responsible for actually drawing everything on the screen, handling font information, and a host of other tasks. Without this connection, we can't do very much.

Save your work if you haven't done so already and switch to App.cpp (Alt + Tab) to enter in this code:

```
// Include the header containing the definition for our App class. This is done
// for you by Paladin.
#include "App.h"

// Include the two headers for the controls we're going to put into our window
#include <StringView.h>
#include <Window.h>

// Define the constructor for our App class.
App::App(void)
	:	BApplication("application/x-vnd.dw-MyFirstApp")
{
	// Create a rectangle to define the size and location of the window
	BRect frame(100,100,500,400);

	// Our window will have the regular style and the title "My First App".
	// When we close it, it will tell our program to quit.
	BWindow *myWindow = new BWindow(frame,"My First App",B_TITLED_WINDOW,
					B_QUIT_ON_WINDOW_CLOSE);
```

```
	// We'll reuse our BRect variable to set the location of our label. The
	// actual width and height don't actually matter, though.
	frame.Set(10,10,11,11);

	// Create a static text label which has the text “Haiku Rocks!”
	BStringView *label = new BStringView(frame,”mylabel”,”Haiku Rocks!”);

	// Tell the label to resize itself to fit the text that we've given it,
	// saving us a lot of work in having to figure it out ourselves.
	label->ResizeToPreferred();

	// Attach the label to our window
	myWindow->AddChild(label);

	// Show our window
	myWindow->Show();
}


int
main(void)
{
	// Create the instance of our App class. Each Haiku program has just one.
	// Creating it sets up the connection to the app_server.
	App *app = new App();

	// This actually puts our application into motion. We will not be exiting
	// this function until our program quits.
	app->Run();

	// Free the memory that we got off the heap. We could create it on the
	// stack, but BApplication objects can get kinda big, so it's better to use
	// the heap.
	delete app;

	// Token return call to quiet the compiler. ;-)
	return 0;
}
```

Save your work and either click on Run from the Build menu or press Alt + R to build and run your project. If you've typed everything correctly, you should see this on the screen:



If you run into errors, carefully check to make sure that you've typed everything just the same as the code above.

We've written our first working Haiku app! Not only does it show something on the screen, but we can resize the window, hide it, or close it and quit the program. It's not terribly useful, but that's OK. We had to learn a lot just to get this far and still understand what's going on.

One small bit of code was not explained in the comments, namely these two lines:

```
App::App(void)
	:	BApplication(“application/x-vnd.dw-MyFirstApp”)
```

The part inside the quotation marks doesn't seem at all familiar. This is what's called a MIME type. MIME stands for **M**ultipurpose **I**nternet **M**ail **E**xtensions. Every file in Haiku has a type, including all Haiku programs. Graphical Haiku programs like ours have a unique type. Substitute your company's name, your online handle, or your initials for the `dw` and the name of your program for `MyFirstApp` – just make sure that the signature begins with `"application/x-vnd."` This will ensure no mixups with any programs you publish online.

This project was a barebones application to show the minimum code needed to create a simple window. It might seem like a lot, but in comparison to a Windows program written in C++, it’s quite short. Next time, we’ll make a window that does more, but for now, play around with the code and see what kinds of things you can do with it – tinkering leads to better understanding.