# Using dynamically allocated arrays

While most instances of JUCE objects can be stored in regular C++ arrays, JUCE offers a handful of arrays that are more powerful, somewhat comparable to the C++ Standard Library classes, such as `std::vector`. The JUCE `Array` class offers many features; these arrays can be:

- Dynamically sized; items can be added, removed, and inserted at any index
- Sorted using custom comparators
- Searched for particular content

The `Array` class is a template class; its main template argument, `ElementType`, must meet certain criteria. The `Array` class moves its contents around by copying memory during resizing and inserting elements, this could cause problems with certain kinds of objects. The class passed as the `ElementType` template argument must also have both a copy constructor and an assignment operator. The `Array` class, in particular, works well with primitive types and some commonly used JUCE classes, for example, the `File` and `Time` classes. In the following example, we create an array of integers, add five items to it, and iterate over the array, sending the contents to the console:

```
int main (int argc, char* argv[])
{
  Logger *log = Logger::getCurrentLogger();

  Array<int> array;

  for (int i = 0; i < 5; ++i)
    array.add (i * 1000);

  for (int i = 0; i < array.size(); ++i) {
    int value = array[i];
    log->writeToLog ("array[" + String (i) + "]= " + String
(value));
  }

  return 0;
}
```

This should produce the output:

```
array[0]= 0
array[1]= 1000
array[2]= 2000
array[3]= 3000
array[4]= 4000
```

Notice that the JUCE `Array` class supports the C++ indexing subscript operator `[]`. This will always return a valid value even if the array index is out of bounds (unlike a built-in array). There is a small overhead involved in making this check; therefore, you can avoid the bounds checking by using the `Array::getUnchecked()` function, but you must be certain that the index is within bounds, otherwise your application may crash. The second `for()` loop can be rewritten as follows to use this alternative function, because we have already checked that out indices will be in-range:

```
...
  for (int i = 0; i < array.size(); ++i) {
    int value = array.getUnchecked (i);
    log->writeToLog("array[" + String (i) + "] = " +
                    String (value));
  }
...
```

# Finding the files in a directory

The JUCE library uses the `Array` objects for many purposes. For example, the `File` class can fill an array of `File` objects with a list of child files and directories it contains using the `File::findChildFiles()` function. The following example should post a list of files and directories in your user `Documents` directory to the console:

```
int main (int argc, char* argv[])
{
  Logger *log = Logger::getCurrentLogger();

  File file =
    File::getSpecialLocation (File::userDocumentsDirectory);

  Array<File> childFiles;

  bool searchRecursively = false;
  file.findChildFiles (childFiles,
                       File::findFilesAndDirectories,
                       searchRecursively);

  for (int i = 0; i < childFiles.size(); ++i)
    log->writeToLog (childFiles[i].getFullPathName());

  return 0;
}
```

Here, the `File::findChildFiles()` function is passed the array of `File` objects, to which it should add the result of the search. It is also told to find both files and directories using the value `File::findFilesAndDirectories` (other options are the `File::findDirectories` and `File::findFiles` values). Finally, it is told not to search recursively.

# Tokenizing strings

Although it is possible to use `Array<String>` to hold an array of JUCE `String` objects, there is a dedicated `StringArray` class to offers additional functionality when applying array operations to string data. For example, a string can be **tokenized** (that is, broken up into smaller strings based on whitespace in the original string) using the `String::addTokens()` function, or divided into strings representing lines of text (based on newline character sequences found within the original string) using the `String::addLines()` function. The following example tokenizes a string, then iterates over the resulting `StringArray` object, posting its contents to the console:

```
int main (int argc, char* argv[])
{
  Logger *log = Logger::getCurrentLogger();

  StringArray strings;
  bool preserveQuoted = true;
  strings.addTokens("one two three four five six",
                    preserveQuoted);

  for (int i = 0; i < strings.size(); ++i) {
    log->writeToLog ("strings[" + String (i) + "]=" +
                     strings[i]);
  }

  return 0;
}
```

# Arrays of components

User interfaces comprising banks of similar controls, such as buttons and sliders, can be managed effectively using arrays. However, the JUCE `Component` class and its subclasses do not meet the criteria for storage as an object (that is, by value) in a JUCE `Array` object. These must be stored as arrays of pointers to these objects instead. To illustrate this, we need a new Introjucer project with a basic window as used throughout Chapter 2, *Building User Interfaces*. Create a new Introjucer project, such as this, name it `Chapter03_02`, and open it into your IDE. To the end of the `MainWindow` constructor in `Main.cpp`, add the following line:

```
setResizable (true, true);
```

In the `MainComponent.h` file change the code to:

```
#ifndef __MAINCOMPONENT_H__
#define __MAINCOMPONENT_H__

#include "../JuceLibraryCode/JuceHeader.h"

class MainContentComponent : public Component
{
public:
  MainContentComponent();
  ~MainContentComponent();

  void resized();

private:
  Array<TextButton*> buttons;
};

#endif
```

Notice that the `Array` object here is an array of pointers to `TextButton` objects (that is, `TextButton*)`. In the `MainComponent.cpp` file change the code to:

```
#include "MainComponent.h"

MainContentComponent::MainContentComponent()
{
  for (int i = 0; i < 10; ++i)
  {
    String buttonName;
    buttonName << "Button " << String (i);
    TextButton* button = new TextButton (buttonName);
    buttons.add (button);
    addAndMakeVisible (button);
  }

  setSize (500, 400);
}

MainContentComponent::~MainContentComponent()
{
}

void MainContentComponent::resized()
{
  Rectangle<int> rect (10, 10, getWidth() - 20, getHeight() - 20);

  int buttonHeight = rect.getHeight() / buttons.size();

  for (int i = 0; i < buttons.size(); ++i) {
    buttons[i]->setBounds (rect.getX(),
                           i * buttonHeight + rect.getY(),
                           rect.getWidth(),
                           buttonHeight);
  }
}
```

Here, we create 10 buttons and using a `for()` loop, adding these buttons to an array, and basing the name of the button on the loop counter. The buttons are allocated using the `new` operator (rather than the static allocation used in Chapter 2, *Building User Interfaces*), and it is these pointers that are stored in the array. (Notice also, that there is no need for the `&` operator in the function call to `Component::addAndMakeVisible()` because the value is already a pointer.) In the `resized()` function, we use a `Rectangle<int>` object to create a rectangle that is inset from the `MainContentComponent` object's bounds rectangle by 10 pixels all the way around. The buttons are positioned within this smaller rectangle. The height for each button is calculated by dividing the height of our rectangle by the number of buttons in the button array. The `for()` loop then positions each button, based on its index within the array. Build and run the application; its window should present 10 buttons arranged in a single column.

There is one major flaw with the preceding code. The buttons allocated with the `new` operator are never deleted. The code should run fine, although you will get an assertion failure when the application is exited. The message into the console will be something like:

```
*** Leaked objects detected: 10 instance(s) of class TextButton
JUCE Assertion failure in juce_LeakedObjectDetector.h:95
```

To solve this, we could delete the buttons in the `MainComponent` destructor like so:

```
MainContentComponent::~MainContentComponent()
{
  for (int i = 0; i < buttons.size(); ++i)
    delete buttons[i];
}
```

However, it is very easy to forget to do this kind of operation when writing complex code.

# Using the OwnedArray class

JUCE provides a useful alternative to the `Array` class that is dedicated to pointer types: the `OwnedArray` class. The `OwnedArray` class always stores pointers, therefore should not include the `*` character in the template parameter. Once a pointer is added to an `OwnedArray` object, it takes ownership of the pointer and will take care of deleting it when necessary (for example, when the `OwnedArray` object itself is destroyed). Change the declaration in the `MainComponent.h` file, as highlighted in the following:

```
...
private:
  OwnedArray<TextButton> buttons;
};
```

You should also remove the code from the destructor in the `MainComponent.cpp` file, because deleting objects more than once is equally problematic:

```
...
MainContentComponent::~MainContentComponent()
{
}
...
```

Build and run the application, noticing that the application will now exit without problems.

This technique can be extended to using broadcasters and listeners. Create a new GUI-based Introjucer project as before, and name it `Chapter03_03`. Change the `MainComponent.h` file to:

```
#ifndef __MAINCOMPONENT_H__
#define __MAINCOMPONENT_H__

#include "../JuceLibraryCode/JuceHeader.h"

class MainContentComponent :  public Component,
                              public Button::Listener
{
public:
MainContentComponent();

void resized();
void buttonClicked (Button* button);

private:
OwnedArray<Button> buttons;
Label label;
};

#endif
```

This time we use an `OwnedArray<Button>` object rather than an `OwnedArray<TextButton>` object. This simply avoids the need to typecast our button pointers to different types when searching for the pointers in the array, as we do in the following code. Also, notice here that we added a `Label` object to our component, made our component a button listener, and that we do not need a destructor. Change the `MainComponent.cpp` file to:

```
#include "MainComponent.h"

MainContentComponent::MainContentComponent()
{
  for (int i = 0; i < 10; ++i) {
    String buttonName;
    buttonName << "Button " << String (i);
    TextButton* button = new TextButton (buttonName);
    button->addListener (this);
    buttons.add (button);
    addAndMakeVisible (button);
  }

  addAndMakeVisible (&label);
  label.setJustificationType (Justification::centred);
  label.setText ("no buttons clicked", dontSendNotification);

  setSize (500, 400);
}

void MainContentComponent::resized()
{
  Rectangle<int> rect (10, 10,
                       getWidth() / 2 - 20, getHeight() - 20);

  int buttonHeight = rect.getHeight() / buttons.size();

  for (int i = 0; i < buttons.size(); ++i) {
    buttons[i]->setBounds (rect.getX(),
                           i * buttonHeight + rect.getY(),
                           rect.getWidth(),
                           buttonHeight);
  }

  label.setBounds (rect.getRight(),
                   rect.getY(),
                   getWidth() - rect.getWidth() - 10,
                   20);
}

void MainContentComponent::buttonClicked (Button* button)
{
  String labelText;
  nt buttonIndex = buttons.indexOf (button);
  labelText << "Button clicked: " << String (buttonIndex);
  label.setText (labelText, dontSendNotification);
}
```

Here, we add the label in the constructor, reduce the width of the bank of buttons to occupy only the left half of the component, and position the label at the top in the right-half. In the button listener callback, we can obtain the index of the button using the `OwnedArray::indexOf()` function to search for the pointer (incidentally, the `Array` class also has an `indexOf()` function for searching the items). Build and run the application and notice that our label reports which button was clicked. Of course, the elegant thing about this code is that we need only change the value in the `for()` loop when the buttons are created in our constructor to change the number of buttons that are created; everything else works automatically.

## Other banks of controls

This approach may be applied to other banks of controls. The following example creates a bank of sliders and labels, keeping each corresponding component updated with the appropriate value. Create a new GUI-based Introjucer project, and name it `Chapter03_04`. Change the `MainComponent.h` file to:

```
#ifndef __MAINCOMPONENT_H__
#define __MAINCOMPONENT_H__

#include "../JuceLibraryCode/JuceHeader.h"

class MainContentComponent :  public Component,
                              public Slider::Listener,
                              public Label::Listener
{
public:
  MainContentComponent();

  void resized();
  void sliderValueChanged (Slider* slider);
  void labelTextChanged (Label* label);

private:
  OwnedArray<Slider> sliders;
  OwnedArray<Label> labels;
};

#endif
```

Here, we have arrays of sliders and labels and our component is both a label listener and a slider listener. Now, update the `MainComponent.cpp` file to contain the include directive, the constructor, and the `resized()` function:

```
#include "MainComponent.h"

MainContentComponent::MainContentComponent()
{
  for (int i = 0; i < 10; ++i) {
    String indexString (i);
    String sliderName ("slider" + indexString);
    Slider* slider = new Slider (sliderName);
    slider->setTextBoxStyle (Slider::NoTextBox, false, 0, 0);
    slider->addListener (this);
    sliders.add (slider);
    addAndMakeVisible (slider);

    String labelName ("label" + indexString);
    Label* label = new Label (labelName,
                              String (slider->getValue()));
    label->setEditable (true);
    label->addListener (this);
    labels.add (label);
    addAndMakeVisible (label);
  }

  setSize (500, 400);
}

void MainContentComponent::resized()
{
  Rectangle<int> slidersRect (10, 10,
                              getWidth() / 2 - 20,
                              getHeight() - 20);
  Rectangle<int> labelsRect (slidersRect.getRight(), 10,
                             getWidth() / 2 - 20,
                             getHeight() - 20);

  int cellHeight = slidersRect.getHeight() / sliders.size();

  for (int i = 0; i < sliders.size(); ++i) {
    sliders[i]->setBounds (slidersRect.getX(),
                           i * cellHeight + slidersRect.getY(),
                           slidersRect.getWidth(),
                           cellHeight);
    labels[i]->setBounds (labelsRect.getX(),
                          i * cellHeight + labelsRect.getY(),
                          labelsRect.getWidth(),
                          cellHeight);
  }
}
```

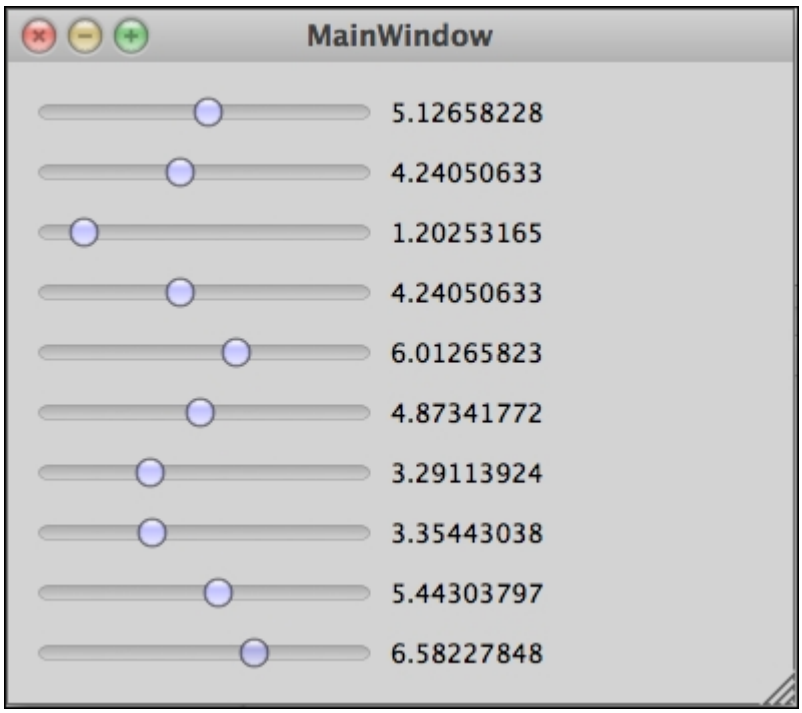
Here, we use a `for()` loop to create the components and add them to the corresponding arrays. In the `resized()` function, we create two helper rectangles, one for the bank of sliders and one for the bank of labels. These are positioned to occupy the left half and right half of the main component respectively.

In the listener callback functions, the index of the broadcasting component is looked up in its array, and this index is used to set the value of the other corresponding component. Add these listener callback functions to the `MainComponent.cpp` file:

```
void MainContentComponent::sliderValueChanged (Slider* slider)
{
  int index = sliders.indexOf (slider);
  labels[index]->setText (String (slider->getValue()),
                          sendNotification);
}

void MainContentComponent::labelTextChanged (Label* label)
{
  int index = labels.indexOf (label);
  sliders[index]->setValue (label->getText().getDoubleValue());
}
```

Here, we use the `String` class to perform the numerical conversions. After moving some of the sliders, the application window should look similar to the following screenshot:




Hopefully, these examples illustrate the power of combining JUCE array classes with other JUCE classes to write elegant, readable, and powerful code.