4.1 Void Pointers and Callbacks

Sue was working on her Skeet project (Project 2) and noticed that somehow OpenGL knew to call her `callback()` function every time a new frame in the game is drawn. How did OpenGL know about this function? Why did it take a `void` pointer as a parameter? This is all quite confusing…

Objectives

By the end of this chapter, you will be able to:

- Define "callback" and explain why it is a useful programming pattern
- Describe situations where `void` pointers can help the programmer

Prerequisites

Before reading this chapter, please make sure you are able to:

- Create a variable that is a pointer to a function (Chapter 1.5)
- Pass a pointer to a function as a parameter (Chapter 1.5)
- Cast one data type into another (Procedural Programming in C++, Chapter 1.3)

What are void pointers & callbacks and why you should care

A **callback** is a function pointer passed to another function as a parameter with the expectation that the function pointer will be executed. Perhaps this is best explained by analogy. Consider a general planning for a battle with the enemy. Up on the front lines, he anticipates that the enemy will try some attack. If this happens, he wants his army to respond a certain way. To deal with this contingency, he puts some orders in a sealed envelope for one of his lieutenants. If the lieutenant observes the enemy trying the attack, the lieutenant is to respond with the instructions in the sealed envelope. In this analogy, the general is the caller, the function initiating the exchange and providing the instructions. The lieutenant is the callee, the function receiving the instructions and the function to carry out the instructions at a pre-determined time. The instructions in the sealed envelope are the callback. These are instructions provided by the caller to be executed by the callee. While this analogy with the general, the lieutenant, and the sealed envelope may seem contrived, it is actually quite common in practice. Consider a graphics library (such as OpenGL) enabling a program to display images on the screen and to receive keyboard events. If such a keyboard event were to occur, how would the graphics library notify the client program? The most convenient way for this to happen is for the graphics library (lieutenant) to call a function (sealed envelope) specified by the client (general). This way the program can handle the event as needed.

A **void pointer** is a pointer to an unknown data type. This is the same as saying "I have a pointer to something; I am just not sure what!" This, too, is best explained by an analogy. A mailman delivers letters to a collection of mailboxes. He does not need to know (and frankly he should not look at) what is in each envelope. In essence, the mailman deals with `void` pointers. A `void` pointer cannot be dereferenced; it first must be casted into a known data type. This may seem a bit pointless. Why would one want to carry around a pointer one cannot dereference? The answer is that the mailman (in our analogy) does not know how to dereference the `void` pointer, but the client who receives the letter does.

The combination of `void` pointers and callback functions enable programmers to create algorithms and data-structures that work with any data type. These algorithms and data-structures work with data in the form of `void` pointers, relying on callback functions to know how to work with the data.

Callback functions

There are three players in any callback scenario: the caller, the callee, and the callback function itself:

The **caller** is the function initiating the callback (the general in our analogy). The caller issues a function call to the callee passing the callback function as a parameter.

```
void caller()                        // the function initiating the callback scenario
{
   callee(&callback);                // the callback function is sent to callee as a
}                                    //    parameter in the form of a function pointer
```

The **callee** is the function receiving the call from the caller (the lieutenant). The callee is needed by the caller to carry out some task involving the callback function. To do this the caller needs to accept the callback function as a function pointer. Presumably this callback function will need to be executed at some point. Either it will need to be executed immediately before control is returned to the caller, or it will be saved for some later event.

```
void callee(void (*callback)())      // the callee takes a callback as a parameter
{
   callback();                       // the callee executes the callback
}                                    //    on behalf of the caller
```

The **callback** is the function to be executed by the callee on the caller's behalf (the sealed envelope). The callback function is a normal function that gets sent to the callee by the caller as a parameter.

```
void callback()                      // just an ordinary function meant to perform
{                                    //    some operation on behalf of the caller
}                                    //    at a point in time specified by the callee
```

Using callback functions

Typically, the caller does not care *how* the callee responds to the caller's request; the caller just needs to know that the job was done. Under certain situations, however, this simple model is not rich enough to capture what needs to transpire. In these situations, the caller not only specifies what needs to be done, but also how. One of the main ways this is done is through callbacks.

Recall from Chapter 1.5 that a function pointer can be passed as a parameter to another function. Usually this is done with the expectation that the callee will execute the passed function under certain circumstances. Consider the following example of a function receiving input from the user:

```
/********************************************
 * PROMPT
 * Ask the user for a number meeting certain
 * criteria.
 ******************************************/
float prompt(bool (*pValidate)(float), const string & message)
{
   float response;
   cout << message << ": ";
   cin  >> response;

   while (!pValidate(response))      // call the callback provided by the client
   {
      cout << "Invalid number.  Please enter another: ";
      cin  >> response;
   }

   return response;
}
```

This function will keep prompting the user for a response, not quitting until a valid number is entered. The question is: under what condition is a response valid? That depends on the callback function `pValidate()` provided by the caller.

The first client of this function is a program asking the user for a GPA. The client (`caller1()`) will need to provide a callback function (`validateGPA()`) to pass to the `prompt()` function:

```
// callback for caller1()
bool validateGPA(float input)
{
   return (input >= 0.0 && input <= 4.0);
}

// first caller, passing validateGPA() as a parameter to the callee prompt()
void caller1()
{
   float gpa = prompt(&validateGPA, "Please enter a GPA");
   cout << "The GPA is: " << gpa << endl;
}
```

The second client is a program asking the user for his salary. This client (`caller2()`) will need to provide a different callback function (`validateIncome()`) for the `prompt()` function:

```
// callback for caller2()
bool validateIncome(float input)
{
   return (input >= 0.00);
}

// second caller, passing validateIncome() as a parameter to the callee prompt()
void caller2()
{
   float income = prompt(&validateIncome, "What is your yearly income");
   cout << "The income is: $" << income << endl;
}
```

By passing a callback function pointer to the `prompt()` function, it has become much more versatile than it would otherwise because it can work with many different types of values.

Sue's Tips

The whole point of a callback function is that the callee can perform an operator on behalf of the caller without knowing anything about that operation. Therefore, it is vital that the callback function be as cohesive as possible and have absolutely no dependencies on the callee. If the callee has to know anything about the inner workings of the callback function, the entire purpose of the callback is defeated!

Example 4.1 – Prompt function

Demo

This example will demonstrate how to pass a callback function to the callee it can perform a specific action according to the caller's specification.

Problem

Write a generic prompt function performing error-recovery and validation according to the specification sent by the caller. The prompt function should also accept zero parameters, one parameter (the prompt string), two parameters (prompt and validation callback), or three parameters (prompt, callback, and error message).

Solution

The first step is to identify the prototype. In order to handle the four ways the `prompt()` function can be called, default parameters will be specified for all three parameters:

```
float prompt(const char * sPrompt = "Enter a value", // prompt string
             bool (*pValidate)(float) = NULL,        // validation callback
             const char * sReprompt = NULL);         // reprompt string
```

Next, the function will be specified. We will borrow heavily from `1-1-getIndex.cpp` which handles unexpected characters in the input stream. Pay special attention to how the callback function is used.

```
float prompt(const char *sPrompt, bool (*pValidate)(float), const char *sReprompt)
{
   bool done = false;
   float value;

   do
   {
      // instructions
      cout << sPrompt << ": ";
      cin >> value;

      if (cin.fail())                               // was a non-float entered?
      {
         if (sReprompt != NULL)
            cout << sReprompt << endl;
         cin.clear();                               // clear the error state
         cin.ignore(256, '\n');                     // clear the buffer
      }

      else if (NULL != pValidate && !pValidate(value)) // are we valid?
      {
         if (sReprompt != NULL)
            cout << sReprompt << endl;
      }
      else                                          // otherwise, we are good!
         done = true;
   }
   while (!done);
   return value;
}
```

See Also

The complete solution is available at 4-1-prompt.html or:

```
/home/cs165/examples/4-1-prompt.cpp
```

Unit 4

Example 4.1 – OpenGL callbacks

Demo

Another common use for callback functions occurs when a library needs to notify the client of an event. This is quite common with networking libraries, the operating system, and graphics libraries.

Problem

Write a program to accept callback events from OpenGL in the form of keyboard input and drawing requests. Register these callbacks with OpenGL and then write the callback code.

Solution

We register callback events with the GLUT interface to OpenGL when the graphics window is first created. This occurs in the `Interface::initialize()` method of `uiInteract.cpp`:

```
void Interface :: initialize(int argc, char ** argv, const char * title)
{
... code removed for brevity ...
   // register the callbacks so OpenGL knows how to call us
   glutDisplayFunc(   drawCallback    );     // register the draw callback
   glutIdleFunc(      drawCallback    );     // drawCallback gets called twice
   glutKeyboardFunc(  keyboardCallback);     // keyboard callback
   glutSpecialFunc(   keyDownCallback );     // cursor key callback
   glutSpecialUpFunc( keyUpCallback   );     // another cursor key callback
... code removed for brevity ...
}
```

Notice how we are passing normal functions to the callback registration functions. We are not passing methods. However, our `Interface` class needs to be made aware of these events. To handle this, we need to instantiate an `Interface` object and call the appropriate method:

```
void keyDownCallback(int key, int x, int y)
{
   // Even though this is a local variable, all the members are static
   // so we are actually getting the same version as in the constructor.
   Interface ui;
   ui.keyEvent(key, true /*fDown*/);
}
```

This means that as soon as the user presses a key on the keyboard, OpenGL will call `keyDownCallback()` so the client can respond to the event. We call this an "asynchronous" callback because it does not occur when the callee is excuted but rather at some time in the future.

See Also

The complete solution is available at:

```
/home/cs165/prj4/uiInteract.cpp
```

Void pointers

Along with the standard built-in data types (`int`, `float`, `bool`, `char`, etc.), there is a special data type called "`void`." This is the "type-less" data type, or a data type that has no type. While you can't declare a variable of type `void`, you often want a function to have the `void` return type.

```
void display();                     // return nothing
```

So what is the point of having a data type which can't be used in a variable declaration? The answer lies with pointers. While we commonly know the type of data an item is pointing to (pointer to a character, for example), this is not always the case. The need frequently arises when a function does not need to know and cannot know the type of data a pointer is pointing to. In these cases, we use a `void` pointer:

```
{
   void * p;                        // we point to something, but we don't know what!
}
```

When we do this, we are stating "I don't need to know what this thing is pointing to." Under normal circumstances, we can access the data in a pointer with the dereference operator. For example, consider the case where a pointer refers to the first element in a string. In this case, we can retrieve the data by dereferencing the pointer:

```
{
   char text[256] = "c-string";
   char * pChar = text;

   cout << *pChar << endl;          // this will display a lowercase 'c'
}
```

With `void` pointers, we cannot dereference it directly because the resulting data type is `void`. To get around this, we need to cast the pointer to a known data type:

```
{
   char text[256] = "void pointer";
   void * pVoid = (void *)text;     // cast it to a void * when assigning an address
                                    //   to a void pointer
   cout << *(char *)pVoid << endl;  // cast it back to a known data type when we
}                                   //   want to access the data in the void pointer
```

Therefore it is possible to store data in a `void` pointer as long as someone knows how to cast it back into a known data type. When working with `void` pointers, there are typically two steps:

1. Cast the data from a known data type into a `void *`
2. Cast the data back from `void *` into the known data type

Sam's Corner

When assigning one data type onto another, a cast is usually required:

```
{
   float valueFloat   = 3.14;
   int   valueInteger = (int)floatValue;   // the compiler might complain without
}                                          //    the (int) cast
```

However, we do not have to use a cast when assigning a pointer to a `void` pointer:

```
{
   void * pVoid = "Text";                  // no need to cast a constant char *
}                                          //     into a void *
```

Example 4.1 – Swap

Demo

This example will demonstrate how to accept void pointers as parameters to a function. The function then does not need to know anything about the data being manipulated.

Problem

Write a swap function that works with any pointer data type. The function should then swap the pointers in memory. Note that it does not swap the data that the pointers refer to.

Solution

The purpose of the `swap()` function is to swap pointers. Therefore, the pointers have to be passed by reference. This may look odd at first. Think of it this way: the function needs to be able to change the address of the pointer being passed as a parameter.

```
void swap(void * & pLhs, void * & pRhs)
{
   void * pTemp = pLhs;
   pLhs = pRhs;
   pRhs = pTemp;
}
```

Now to call this function, we need to pass pointers that we can change.

```
int main()
{
   char buffer1[256];       // this is a constant pointer.  We can change the
   char buffer2[256];       //    data but not the address of buffer1
   char * p1 = buffer1;     // p1 shares the address of buffer1 but we
   char * p2 = buffer2;     //    have the ability to change the address

   // prompt the user
   cout << "First message:  ";
   cin.getline(buffer1, 256);
   cout << "Second message: ";
   cin.getline(buffer2, 256);

   // before we swap
   cout << "Before: \"" << p1 << "\", \"" << p2 << "\"\n";

   // swap them
   swap(p1, p2);            // note that we do not need to cast the char *
                            //    to a void pointer
   // display after the swap
   cout << "After:  \"" << p1 << "\", \"" << p2 << "\"\n";

   return 0;
}
```

Challenge

As a challenge, can you change the driver program to swap pointers to `floats`? Note that you will need to allocate the `floats` that are swapped.

See Also

The complete solution is available at 4-1-swap.html or:

```
/home/cs165/examples/4-1-swap.cpp
```

Example 4.1 – Stack

Demo

This example will demonstrate how to use `void` pointers in a container (classes designed to hold data). In this case, the container does not need to know what type of data is being stored. Instead, it just needs to give the data back to the client when he asks for it.

Problem

Write a class to implement a Stack, a container following the "first-in, last-out" pattern. As with "4.0 Stack," this class will support the push operation (place an item on the end of the list) and the pop operation (remove an item from the end of the list).

Solution

We will start from the stack of `float`s from `4-0-stack.cpp`. The only thing we need to do is replace the `float` stand-ins with `void` pointers.

```
class Stack
{
   public:
      // create the stack with a zero size
      Stack() : size(0) {}
      // add an item to the stack if there is room.  Otherwise throw
      void push(void * value) throw(bool)
      {
         if (size < MAX)
            data[size++] = value;   // must support the assignment operator
         else
            throw false;
      }
      // pop an item off the stack if there is one.  Otherwise throw
      void * pop() throw(bool)
      {
         if (size)
            return data[--size];    // must support the assignment operator
         else
            throw false;
      }
   private:
      void * data[MAX];             // a stack of void pointers
      int    size;                  // number of items currently in the stack
};
```

When we test the program, we need to cast our data type into a "`void *`" when we push it on and cast it back to something useful when we pop it off.

```
{
   Stack stack;
   stack.push((void *)"text");
   cout << (char *)stack.pop() << endl;
}
```

Challenge

As a challenge, modify the driver program so it works with pointers to `string`s. Note that you will need to push on a pointer to a `string` so it will need to be allocated with `new`.

See Also

The complete solution is available at 4-1-stack.html or:

```
/home/cs165/examples/4-1-stack.cpp
```

Callbacks with void pointers

Recall that a callback is a function that is passed to a callee so it can be executed on the caller's half. The callee does not need to know anything about the function being passed. Recall that a `void` pointer is a pointer to an unknown data type so, when passed to a function, the function does not need to know anything about the data being passed. It turns out that combining callbacks and `void` pointers give the programmer even more power to work with generic algorithms and data-structures.

Consider the typical `void` pointer scenario: a function taking a `void` pointer as a parameter but is unable to work with the data because nothing is known about the data type of the `void` pointer. However, if the caller also provides a callback function that knows how to work with the `void` pointer, we have the power to implement truly type-independent data-structures and algorithms.

To illustrate this point, recall the bubble sort example from Chapter 4.0. Ideally we should be able to use the bubble sort generic algorithm with just about any data type. It should work with text, numbers, playing cards, and many other data types. As long as the data type supports a few operations, we should be able to use it. Specifically, we need to be able to swap two items and compare two items. This means that if we are to write a generic sort algorithm, we will need to pass several parameters:

- `array:` The list of items that are to be sorted. This will be an array of `void` pointers of course
- `num:` The number of items in the `array`. This will always be an integer
- `compare():` A callback function knowing how to compare two items in the list. Since `array` contains `void` pointers, the `compare()` function will need to know how to cast the `void` pointer into a known data type
- `swap():` A callback function knowning how to swap to items in the list.

Recall from Chapter 4.0 the necessity of indentifing the operations required to implement a generic algorithm or generic data-structure. When using callback functions and `void` pointers, each operation gets passed as a callback to the function or class.

Thus the callback function and the `void` pointer work as a pair: the callback is the only function needing to be able to cast the `void` pointer back to the data in which it really represents. In this scenario, the role of the callee is not to interpret the `void` pointer, but rather to just hand it over to the callback.

Sue's Tips

A common objection to the callback/void-pointer pairing is that often more than one parameter needs to be sent to the callback. It turns out, however, than *you only ever need a single parameter*. Since that `void` pointer can refer to a pointer to a structure or to a class, arbitrarily rich data can be sent through that single parameter.

Example 4.1 – Binary search

Demo

This example will demonstrate how to use `void` pointers in conjunction with a callback function in order to implement a generic algorithm. This algorithm should work with any data type as long as there is a `compare()` function that works with it.

Problem

Write a binary search algorithm that can work with any data type. Only four parameters are needed: the array of `void` pointers, the number of items in the array, a callback function to compare two values, and the value being searched for.

Solution

We will start with the float version of the binary search algorithm from Chapter 4.0. We will replace the data type with `float` and pass a compare callback function as a parameter.

```
bool binarySearch(const void * array[],            // an array of void pointers
                  int size,                         // number of items in array
                  int (*pCompare)(const void *, const void *),
                  const void * search)              // the search value
{
   int iFirst = 0;
   int iLast  = size - 1;

   // loop through the list
   while (iLast >= iFirst)
   {
      int iMiddle = (iLast + iFirst) / 2;

      int compare = pCompare(array[iMiddle], search);
      if (compare == 0)                             // 0 means they are the same
         return true;
      if (compare > 0)                              // positive means bigger
         iLast = iMiddle - 1;
      else                                          // negative means smaller
         iFirst = iMiddle + 1;
   }

   return false;
}
```

To test this function, we will need to cast our data into `void` pointers:

```
   if (binarySearch((const void **)values,
                    10,
                    (int  (*)(const void *, const void *))strcmp,
                    (const void *)search))
      cout << "\tThe name is in the list\n";
   else
      cout << "\tThe name is not in the list\n";
```

Challenge

As a challenge, can you change the driver program to search through an array of string objects rather than using c-strings. Note that you will need to write your own comparision function and you will need to pass to `binarySearch()` an array of pointers to `string` objects.

See Also

The complete solution is available at 4-1-binarySearch.html or:

```
/home/cs165/examples/4-1-binarySearch.cpp
```

Example 4.1 – OpenGL part 2

Demo

This example will demonstrate how the callback mechanism is much more powerful and versatile when paired with a `void` pointer. When a graphics library such as OpenGL notifies the client that it is time to draw, the client needs the program state to complete the operation. This program state comes in as a `void` pointer.

Problem

Write a program to display a rotating polygon on the screen using the OpenGL library. First, create a class representing the state of the program (a polygon in this case), next write a callback function to be executed by OpenGL when it is time to draw, and finally, pass a `void` pointer (with the program state) and a callback function (the draw functionality) to OpenGL.

Solution

The first step is to write a class representing the state of the program. For a game, this will be the game class. In this example, it will be the position, angle, and number of sides of the polygon.

```
class Ball
{
public:
   Ball() : sides(3), rotation(0), pt() { }

   // this is just for test purposes.  Don't make member variables public!
   Point pt;          // location of the polygon on the screen
   int sides;         // number of sides in the polygon.  Initially three
   int rotation;      // the angle or orientation of the polygon
};
```

Next the callback function which will handle any input (from pUI), any physics that need to be computed (rotating the polygon), and the drawing routines.

```
void callBack(const Interface *pUI, void * p)
{
   Ball * pBall = (Ball *)p;  // cast the void pointer into a known type
```

... code removed for brevity ...

```
   if (pUI->isSpace())
      pBall->sides++;         // now use our Ball class as we normally would
   pBall->rotation++;

   // draw
   drawCircle(pBall->pt, /*position*/
              20, /* radius */
              pBall->sides /*segments*/,
              pBall->rotation /*rotation*/);
}
```

Notice how the first thing the callback function does is to cast the `void` pointer into a known data type. The last thing to do is to pass the callback function and the void pointer to OpenGL.

```
int main(int argc, char ** argv)
{
   Interface ui(argc, argv, "Test");    // initialize OpenGL
   Ball ball;                           // initialize the game state
   ui.run(callBack, &ball);             // set everything into action, passing a
                                        //    void pointer and callback to OpenGL

   return 0;
}
```

See Also

The complete solution is available at:

```
/home/cs165/prj2/uiTest.cpp
```

Problem 1 - 4

Consider the STL container `vector`. It is an array that grows to accommodate as much data as the client puts in it.

```
Vector
- data
- capacity
- size
+ Vector
+ pushBack
+ operator []
+ getSize
- resize
```

1. Write the class definition for `Vector` where any data type can be stored.

2. Write the code for the array-index operator.

3. Write the code for the `pushBack()` method.

4. Write the code for the `resize()` method.

Please see page 280 *for a hint.*