



School of Computing and Informatics

BCS362 – GENERIC PROGRAMMING WITH C++

Pointers

1 Introduction

A pointer is a variable that stores memory address of another variable. Normally, a variable directly contains a specific value. A pointer contains the memory address of a variable that, in turn, contains a specific value. In this sense, a variable name directly references a value, and a pointer indirectly references a value. Referencing a value through a pointer is called indirection.

Figure 1 typically represent a pointer as an arrow from the variable that contains an address to the variable located at that address in memory.

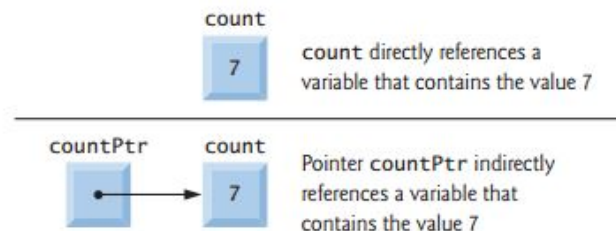


Figure 1: Directly and Indirectly referencing a variable

Just like other variable in C++, pointers must be declared before being used. The general syntax for pointer declaration is

```
1 data_type *variable_name;
```

For instance, the line

```
1 double *ptr;
```

declares a pointer named *ptr* which will store a reference to a memory location that can store a [double](#). Each variable being declared as a pointer must be preceded by an asterisk (*). For example, the declaration

```
1 float *ptr1, *ptr2, *ptr3
```

Pointers should be initialized to 0, NULL or an address of the corresponding type either when they're declared or in an assignment. A pointer with the value 0 or NULL points to nothing and is known as a null pointer. Symbolic constant NULL is defined in several standard library headers to represent the value 0. Initializing a pointer to NULL is equivalent to initializing a pointer to 0, but in C++, 0 is used by convention. When 0 is assigned, it's converted to a pointer of the appropriate type. The value 0 is the only integer value that can be assigned directly to a pointer variable without first casting the integer to a pointer type. [Note: In

the new standard, you should use the constant `nullptr` to initialize a pointer instead of 0 or NULL. Several C++ compilers already implement this constant.]

1.1 Pointer operators

The address operator (&) is a unary operator that obtains the memory address of its operand. For example,

```
1 float a = 10;
2 float *aPtr;
3 aPtr = &a;
```

Line 1 declares a variable of type float and assign it a value, line 2 declares a pointer that can store an address of memory location holding a float and line 3 takes the memory address (reference) of the variable declared in line 1 and assign it to a pointer declared in line 2.

The * operator, commonly referred to as the indirection operator or dereferencing operator, returns a synonym (i.e., an alias or a nickname) for the object to which its pointer operand points. For instance,

```
1 cout << *aPtr << endl;
```

will display the value stored in variable a, just like

```
1 cout << a << endl;
```

would.

Using * in this manner is called dereferencing a pointer. A dereferenced pointer may also be used on the left side of an assignment statement, as in

```
1 float a;
2 float *aPtr;
3 aPtr = &a;
4 *aPtr = 10;
```

where line 4 takes the value 10 and store it in a variable a (Take the value 10 and store it in variable whose address is in *aPtr). You can still read the value from the keyboard as in

```
1 float a;
2 float *aPtr;
3 aPtr = &a;
4 cin >> *aPtr;
```

which places the input in the variable a.

Dereferencing an uninitialized pointer could cause a fatal execution-time error, or it could accidentally modify important data and allow the program to run to completion, possibly with incorrect results.

Example

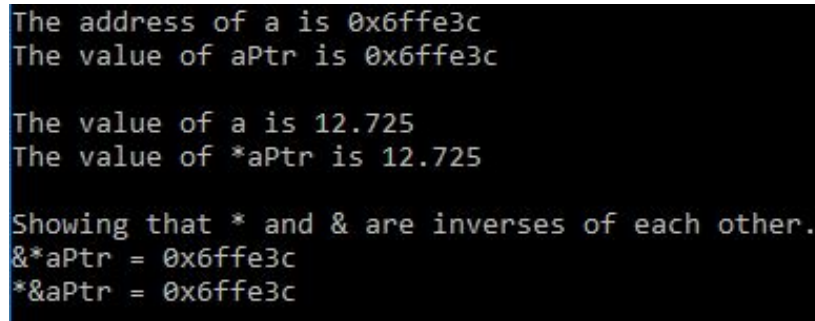
```
1 #include <iostream>
2 using namespace std;
3
4 int main(){
5     float a; // a is an integer
6     a = 12.725;
7     float *aPtr;
8     aPtr = &a;
9     // assigned 7 to a
10    cout << "The address of a is " << &a
11    << "\nThe value of aPtr is " << aPtr;
12    cout << "\n\nThe value of a is " << a
13    << "\nThe value of *aPtr is " << *aPtr;
```

```

14  cout << "\n\nShowing that * and & are inverses of "
15  << "each other.\n&*aPtr = " << &*aPtr
16  << "\n*&aPtr = " << *&aPtr << endl;
17
18  } // end main

```

Figure 2 shows the output of this code, of course on my system, you may get difference values for references on your system.



```

The address of a is 0x6ffe3c
The value of aPtr is 0x6ffe3c

The value of a is 12.725
The value of *aPtr is 12.725

Showing that * and & are inverses of each other.
&*aPtr = 0x6ffe3c
*&aPtr = 0x6ffe3c

```

Figure 2: Output of the code above

The & and * operators are inverses of one another when they're applied consecutively to aPtr in either order, they cancel one another out yielding the same result (the value in aPtr).

1.2 Pass-by Reference with Pointers

There are three ways in C++ to pass arguments to a function: pass-by-value, pass-by-reference with reference arguments and pass-by-reference with pointer arguments.

You can use pointers and the indirection operator (*) to accomplish pass-by-reference for example

```

1  #include <iostream>
2  using namespace std;
3  void cubeByReference( int *); // prototype
4
5  int main()
6  {
7      int number = 5;
8      cout << "The original value of number is " << number;
9      cubeByReference( &number ); // pass number by reference
10     cout << "\nThe new value of number is " << number << endl;
11 } // end main
12
13 void cubeByReference( int *nPtr )
14 {
15     *nPtr = *nPtr * *nPtr * *nPtr; // cube *nPtr
16     /*****what will be the output if the line above is return *nPtr * *nPtr * *nPtr
17     and return type changed to int both
18     at function declaration and function definition . Explain. *****/
19 }

```

1.3 Using const with Pointers

Recall that const enables you to inform the compiler that the value of a particular variable should not be modified. Many possibilities exist for using (or not using) const with function parameters. How do you choose the most appropriate of these possibilities? Let the principle of least privilege be your guide. Always give a function enough access to the data in its parameters to accomplish its specified task, but no more.

1.3.1 Nonconstant Pointer to nonconstant data

The highest access is granted by a nonconstant pointer to nonconstant data – the data can be modified through the dereferenced pointer, and the pointer can be modified to point to other data. Such a pointers declaration as

```
1 int *ptr
```

1.3.2 Nonconstant Pointer to constant data

A nonconstant pointer to constant data is a pointer that can be modified to point to any data item of the appropriate type, but the data to which it points cannot be modified through that pointer. Such a pointer might be used to receive an array argument to a function that will process each array element, but should not be allowed to modify the data. Any attempt to modify the data in the function results in a compilation error. The declaration for such a pointer places `const` to the left of the pointers type, as in

```
1 const int *ptr;
```

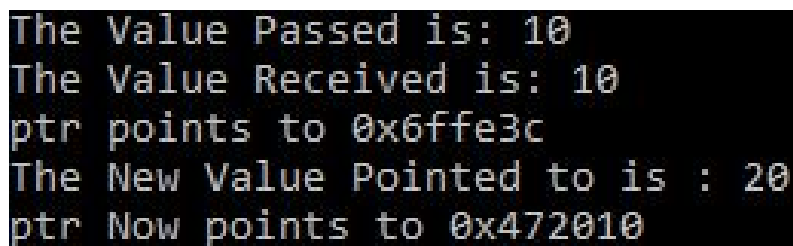
The declaration is read from right to left as `ptr` is a pointer to an integer constant. The code below demonstrate error for attempting to modify data referenced by the pointer.

```
1 void test(const int *ptr )
2 {
3     *ptr = 45; //error: Cannot modify a const object
4 }
```

The `ptr` could be re-assigned with another address, but the value in the referenced memory cannot be changed.

```
1 #include<iostream>
2 using namespace std;
3 int x = 20;
4
5 void re_assign(const int *ptr )
6 {
7     cout << "The Value Received is: " << *ptr << endl;
8     cout << "ptr points to " << ptr << endl;
9     ptr = &x; //Lets change the location ptr points to
10    cout<< "The New Value Pointed to is : " << *ptr << endl;
11    cout << "ptr Now points to " << ptr << endl;
12 }
13 int main()
14 {
15     int a = 10;
16     cout<< "The Value Passed is: " << a << endl;
17     re_assign(&a);
18 }
```

Figure 3 shows the output of this code



```
The Value Passed is: 10
The Value Received is: 10
ptr points to 0x6ffe3c
The New Value Pointed to is : 20
ptr Now points to 0x472010
```

Figure 3: Output of the code above

1.3.3 Constant Pointer to nonconstant data

A constant pointer to nonconstant data is a pointer that always points to the same memory location; the data at that location can be modified through the pointer, but the reference stored by the pointer cannot be changed. For example an array name, which is a constant pointer to the beginning of the array.

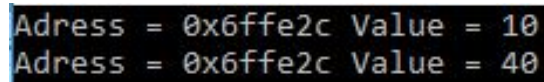
A constant pointer to nonconstant data can be used to receive an array as an argument to a function that accesses array elements using array subscript notation. Pointers that are declared `const` must be initialized when they're declared, but if the pointer is a function parameter, it's initialized with the pointer that's passed to the function. The code below shows how you can declare a constant pointer

```
1 int x;  
2 int *const ptr = &x;
```

The code below demonstrates how you cannot change the reference stored in a constant pointer.

```
1 #include <iostream>  
2 using namespace std;  
3 int main()  
4 {  
5     int x = 10;  
6     int y = 40;  
7     int *const ptr = &x; //constant pointer must be initialized  
8     cout << "Address = " << ptr << " Value = " << *ptr << endl;  
9     *ptr = y; //Allowed: Since *ptr is not constant  
10    cout << "Address = " << ptr << " Value = " << *ptr << endl;  
11  
12    //ptr = &y; //Error: You cannot change address stored by ptr, it's a constant pointer  
13    //uncomment line 12 above, and see the error flagged by your compiler  
14 }
```

Figure 4 shows the output of this code



```
Address = 0x6ffe2c Value = 10  
Address = 0x6ffe2c Value = 40
```

Figure 4: Output of the code above

Notice how the address remains the same (we cannot change it) while we can change the value.

1.3.4 Constant Pointer to constant data

The minimum access privilege is granted by a constant pointer to constant data. Such a pointer always points to the same memory location, and the data at that location cannot be modified via the pointer. This is how an array should be passed to a function that only reads the array, using array subscript notation, and does not modify the array. This pointer can be declared as

```
1 int x = 20;  
2 const int *const ptr = &x;
```

The code below shows that you cannot change the address stored by the pointer nor the value the pointer points to.

```
1 #include <iostream>  
2 using namespace std;  
3 int main()  
4 {  
5     int x = 20;  
6     int y = 40;  
7     const int *const ptr = &x;  
8  
9     *ptr = 40; //Error: You cannot change the value in x, it's a constant
```

```

10 ptr = &y; //Error: You cannot change the address stored by ptr
11 }

```

This code will give two errors. Compile it and see.

Assignment

Two functions are declared as

```

1 void selectionSort(int *const array, const int size);
2 void swap(int *const p1, int *const p2);

```

The function *selectionSort* receives a reference to an array and size of the array, it uses the function *swap* to return sorted array (sorted in ascending order) Write the definition of these functions such that they perform their intended task. Write a main function, pass an array to *selectionSort* to test if indeed it does sorting of the array. Display array elements before and after sorting.

1.4 Pointer Expressions and Pointer Arithmetic

A pointer may be incremented (++) or decremented (--), an integer may be added to a pointer (+ or +=), an integer may be subtracted from a pointer (- or -=), or one pointer may be subtracted from another of the same type.

Assume that array `int v[5]` has been declared and that its first element is at memory location 3000. Assume that pointer `vPtr` has been initialized to point to `v[0]` (i.e., the value of `vPtr` is 3000). Figure 5 diagrams this situation for a machine with four-byte integers. Variable `vPtr` can be initialized to point to array `v` with either of the following statements (because the name of an array is equivalent to the address of its first element):

```

1 int *vPtr = v;
2 int *vPtr = &v[ 0 ];

```

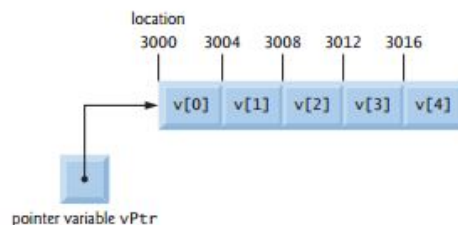


Figure 5: `vPtr` current address

In conventional arithmetic, the addition $3000 + 2$ yields the value 3002. This is normally not the case with pointer arithmetic. When an integer is added to, or subtracted from, a pointer, the pointer is not simply incremented or decremented by that integer, but by that integer times the size of the object to which the pointer refers. The number of bytes depends on the objects data type. For example, the statement

```

1 vPtr += 2;

```

would produce 3008 (from the calculation $3000 + 2 * 4$), assuming that an `int` is stored in four bytes of memory. In the array `v`, `vPtr` would now point to `v[2]` as shown in Figure 6 If `vPtr` had been incremented to 3016, which points to `v[4]`, the statement

```

1 vPtr += 2;

```

would set `vPtr` back to 3000 the beginning of the array. If a pointer is being incremented or decremented by one, the increment (++) and decrement (--) operators can be used. Each of the statements

```

1 ++vPtr;
2 vPtr++;

```

increments the pointer to point to the next element of the array. Each of the statements

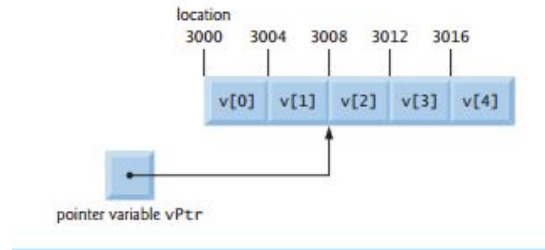


Figure 6: New address for vPtr

```
1 --vPtr;
2 vPtr--;
```

decrements the pointer to point to the previous element of the array.

Pointer variables pointing to the same array may be subtracted from one another. For example, if vPtr contains the address 3000 and vPtr2 contains the address 3008, the statement

```
1 x = vPtr2 - vPtr;
```

would assign to x the number of array elements from vPtr to vPtr2 in this case, 2. Pointer arithmetic is meaningless unless performed on a pointer that points to an array. We cannot assume that two variables of the same type are stored contiguously in memory unless they're adjacent elements of an array. Code example

```
1 #include<iostream>
2 using namespace std;
3 int main()
4 {
5     int v[4];
6     int *vPtr = v; //Pointer to the first element of the array can also be int *vPtr = &v[0]
7
8     cout << "First Address :" << vPtr << endl;
9     vPtr++;
10    cout << "Second Address :" << vPtr << endl;
11    vPtr++;
12    cout << "Third Address :" << vPtr << endl;
13    int *x = v + 1; //Take name of array (pointer to first element, add 1 * number of bytes enough to store an int on
14    cout << "Second Address :" << x << endl;
15
16 }
```

Figure 7 show output of the code above.

```
First Address :0x6ffe30
Second Address :0x6ffe34
Third Address :0x6ffe38
Second Address :0x6ffe34
```

Figure 7: Output of code above

Notice how addresses of two adjacent cells are 4 bytes away (Because my good machine stores integers as a 32 bit number)

1.5 Relationship between Pointers and Arrays

Arrays and pointers are intimately related in C++ and may be used almost interchangeably. An array name can be thought of as a constant pointer. Pointers can be used to do any operation involving array

subscripting.

Assume the following declarations:

```
1 int b[ 5 ]; // create 5-element int array b
2 int *bPtr; // create int pointer bPtr
```

Because the array name (without a subscript) is a (constant) pointer to the first element of the array, we can set bPtr to the address of the first element in array b with the statement

```
1 bPtr = b; // assign address of array b to bPtr
```

This is equivalent to assigning the address of the first element of the array as follows:

```
1 bPtr = &b[ 0 ]; // also assigns address of array b to bPtr
```

Array element b[3] can alternatively be referenced with the pointer expression

```
1 *( bPtr + 3 )
```

The 3 in the preceding expression is the offset to the pointer. When the pointer points to the beginning of an array, the offset indicates which array element should be referenced, and the offset value is identical to the subscript. This notation is referred to as pointer/ offset notation. The parentheses are necessary, because the precedence of * is higher than that of +. Without the parentheses, the preceding expression would add 3 to a copy *bPtrs value (i.e., 3 would be added to b[0], assuming that bPtr points to the beginning of the array). Just as the array element can be referenced with a pointer expression, the address

```
1 &b[ 3 ];
```

can be written with the pointer expression

```
1 bPtr + 3;
```

The array name (which is implicitly const) can be treated as a pointer and used in pointer arithmetic. For example, the expression

```
1 *( b + 3 );
```

also refers to the array element b[3]. In general, all subscripted array expressions can be written with a pointer and an offset. In this case, pointer/offset notation was used with the name of the array as a pointer. The preceding expression does not modify the array name in any way; b still points to the first element in the array.

Pointers can be subscripted exactly as arrays can. For example, the expression

```
1 bPtr[ 1 ];
```

refers to the array element b[1]; this expression uses pointer/subscript notation.

Remember that an array name is a constant pointer; it always points to the beginning of the array. Thus, the expression

```
1 b += 3;
```

causes a compilation error, because it attempts to modify the value of the array name (a constant) with pointer arithmetic.

1.6 Pointer-Based String Processing

A string is a series of characters treated as a single unit. A string may include letters, digits and various special characters such as +, -, *, /and \$. String literals, or string constants, in C++ are written in double quotation marks.

1.6.1 Pointer-Based Strings

A pointer-based string is an array of characters ending with a null character (`'\0'`), which marks where the string terminates in memory. A string is accessed via a pointer to its first character. The value of a string literal is the address of its first character, but the `sizeof` of a string literal is the length of the string including the terminating null character. Pointer-based strings are like arrays; an array name is also a pointer to its first element.

A string literal may be used as an initializer in the declaration of either a character array or a variable of type `char *`. The declarations

```
1 char color[] = "blue";
2 const char *colorPtr = "blue";
```

each initialize a variable to the string "blue". The first declaration creates a five-element array `color` containing the characters 'b', 'l', 'u', 'e' and '\0'. The second declaration creates pointer variable `colorPtr` that points to the letter b in the string "blue" (which ends in '\0') somewhere in memory. String literals have static storage class (they exist for the duration of the program) and may or may not be shared if the same string literal is referenced from multiple locations in a program. The effect of modifying a string literal is undefined; thus, you should always declare a pointer to a string literal as `const char *`.

A string can be read into a character array using stream extraction with `cin`. For example, the following statement reads a string into character array `word[20]`:

```
1 cin >> word;
```

The string entered by the user is stored in `word`. The preceding statement reads characters until a white-space character or end-of-file indicator is encountered. The string should be no longer than 19 characters to leave room for the terminating null character. The `setw` stream manipulator can be used to ensure that the string read into `word` does not exceed the size of the array. For example, the statement

```
1 cin >> setw( 20 ) >> word;
```

specifies that `cin` should read a maximum of 19 characters into array `word` and save the 20th location in the array to store the terminating null character for the string. The `setw` stream manipulator applies only to the next value being input. If more than 19 characters are entered, the remaining characters are not saved in `word`, but they will be in the input stream and can be read by the next input operation.

In some cases, it's desirable to input an entire line of text into a character array. For this purpose, the `cin` object provides the member function `getline`, which takes three arguments: a character array in which the line of text will be stored, a length and a delimiter character. For example, the statements

```
1 char sentence[ 80 ];
2 cin.getline( sentence, 80, '\n' );
```

declare array `sentence` of 80 characters and read a line of text from the keyboard into the array. The function stops reading characters when the delimiter character '\n' is encountered, when the end-of-file indicator is entered or when the number of characters read so far is one less than the length specified in the second argument. The last character in the array is reserved for the terminating null character. If the delimiter character is encountered, it's read and discarded.

A character array representing a null-terminated string can be output with `cout` and `jj`. The statement

```
1 cout << sentence;
```

prints the array `sentence`. Note that `cout` `jj`, like `cin` `jj`, does not care how large the character array is. The characters of the string are output until a terminating null character is encountered; the null character is not printed. [Note: `cin` and `cout` assume that character arrays should be processed as strings terminated by null characters; `cin` and `cout` do not provide similar input and output processing capabilities for other array types.]

1.7 Function pointer

A function pointer is a variable that stores the address of a function that can later be called through that function pointer.

A pointer to a function contains the functions address in memory. We know that an arrays name is actually the address in memory of the first element. Similarly, a functions name is actually the starting address in memory of the code that performs the functions task. Pointers to functions can be passed to functions, returned from functions, stored in arrays, assigned to other function pointers and used to call the underlying function.

The syntax for creating a non-const function pointer is one of the ugliest things you will ever see in C++:

```
1 // fcnPtr is a pointer to a function that takes no arguments and returns an integer
2 int (*fcnPtr)();
```

In the above snippet, fcnPtr is a pointer to a function that has no parameters and returns an integer. fcnPtr can point to any function that matches this type.

The parenthesis around *fcnPtr are necessary for precedence reasons, as int *fcnPtr() would be interpreted as a forward declaration for a function named fcnPtr that takes no parameters and returns a pointer to an integer.

To make a const function pointer, the const goes after the asterisk:

```
1 int (*const fcnPtr)();
```

If you put the const before the int, then that would indicate the function being pointed to would return a const int.

1.7.1 Assigning a function to a function pointer

Function pointers can be initialized with a function (and non-const function pointers can be assigned a function):

```
1 #include<iostream>
2 using namespace std;
3 int square(int x)
4 {
5     return x * x;
6 }
7
8 int cube(int x)
9 {
10    return x * x * x;
11 }
12
13 int main()
14 {
15     int (*fcnPtr)(int) = square; // fcnPtr points to function square
16     cout<< "square() = " << (*fcnPtr)(5)<< endl; //calling function square
17     cout << "The address fcnPtr is " << reinterpret_cast<void*>(fcnPtr) << endl;
18     fcnPtr = cube; // fcnPtr now points to function cube
19     cout<< "cube() = " << fcnPtr(5)<< endl; //calling function square
20     cout << "The address of fcnPtr is now " << reinterpret_cast<void*>(fcnPtr) << endl;
21
22
23     return 0;
24 }
```

Line 16 and 19 shows two different ways we can call a function using function pointers:-

1. Explicit dereference - Line 16
2. Implicit dereference - Line 19

Figure 8 shows the output of this code.

```
square() = 25
The address fcnPtr is 0x401530
cube() = 125
The address of fcnPtr is now 0x401540
```

Figure 8: Output of function pointers code

As you can see, the implicit dereference method looks just like a normal function call – which is what you'd expect, since normal function names are pointers to functions anyway! However, some older compilers do not support the implicit dereference method, but all modern compilers should.

One interesting note: Default parameters won't work for functions called through function pointers. Default parameters are resolved at compile-time (that is, if you don't supply an argument for a defaulted parameter, the compiler substitutes one in for you when the code is compiled). However, function pointers are resolved at run-time. Consequently, default parameters can not be resolved when making a function call with a function pointer. You'll explicitly have to pass in values for any defaulted parameters in this case.

Note that the type (parameters and return type) of the function pointer must match the type of the function. Here are some examples of this:

```
1 // function prototypes
2 int foo();
3 double goo();
4 int hoo(int x);
5
6 // function pointer assignments
7 int (*fcnPtr1)() = foo; // okay
8 int (*fcnPtr2)() = goo; // wrong -- return types don't match!
9 double (*fcnPtr4)() = goo; // okay
10 fcnPtr1 = hoo; // wrong -- fcnPtr1 has no parameters, but hoo() does
11 int (*fcnPtr3)(int) = hoo; // okay
```

1.7.2 Passing functions as arguments to other functions

One of the most useful things to do with function pointers is pass a function as an argument to another function. Functions used as arguments to another function are sometimes called callback functions.

Consider a case where you are writing a function to perform a task (such as sorting an array), but you want the user to be able to define how a particular part of that task will be performed (such as whether the array is sorted in ascending or descending order). Let's take a closer look at this problem as applied specifically to sorting, as an example that can be generalized to other similar problems.

All sorting algorithms work on a similar concept: the sorting algorithm iterates through a list of numbers, does comparisons on pairs of numbers, and reorders the numbers based on the results of those comparisons. Consequently, by varying the comparison, we can change the way the function sorts without affecting the rest of the sorting code.

Here is our selection sort function

```
1 #include <algorithm> // for std::swap, use <utility> instead if C++11
2
```

```

3 void SelectionSort(int *array, int size)
4 {
5     // Step through each element of the array
6     for (int startIndex = 0; startIndex < size; ++startIndex)
7     {
8         // smallestIndex is the index of the smallest element we've encountered so far.
9         int smallestIndex = startIndex;
10
11        // Look for smallest element remaining in the array (starting at startIndex+1)
12        for (int currentIndex = startIndex + 1; currentIndex < size; ++currentIndex)
13        {
14            // If the current element is smaller than our previously found smallest
15            if (array[smallestIndex] > array[currentIndex]) // COMPARISON DONE HERE
16                // This is the new smallest number for this iteration
17                smallestIndex = currentIndex;
18        }
19
20        // Swap our start element with our smallest element
21        std::swap(array[startIndex], array[smallestIndex]);
22    }
23 }

```

Lets replace that comparison with a function to do the comparison. Because our comparison function is going to compare two integers and return a boolean value to indicate whether the elements should be swapped, it will look something like this:

```

1 bool ascending(int x, int y)
2 {
3     return x > y; // swap if the first element is greater than the second
4 }

```

And heres our selection sort routine using the ascending() function to do the comparison:

```

1 #include <algorithm> // for std::swap, use <utility> instead if C++11
2
3 void SelectionSort(int *array, int size)
4 {
5     // Step through each element of the array
6     for (int startIndex = 0; startIndex < size; ++startIndex)
7     {
8         // smallestIndex is the index of the smallest element we've encountered so far.
9         int smallestIndex = startIndex;
10
11        // Look for smallest element remaining in the array (starting at startIndex+1)
12        for (int currentIndex = startIndex + 1; currentIndex < size; ++currentIndex)
13        {
14            // If the current element is smaller than our previously found smallest
15            if (ascending(array[smallestIndex], array[currentIndex])) // COMPARISON DONE HERE
16                // This is the new smallest number for this iteration
17                smallestIndex = currentIndex;
18        }
19
20        // Swap our start element with our smallest element
21        std::swap(array[startIndex], array[smallestIndex]);
22    }
23 }

```

Now, in order to let the caller decide how the sorting will be done, instead of using our own hard-coded comparison function, well allow the caller to provide their own sorting function! This is done via a function pointer.

Because the callers comparison function is going to compare two integers and return a boolean value, a pointer to such a function would look something like this:

```
1 bool (*comparisonFcn)(int, int);
```

So, we'll allow the caller to pass our sort routine a pointer to their desired comparison function as the third parameter, and then we'll use the caller's function to do the comparison.

Here's a full example of a selection sort that uses a function pointer parameter to do a user-defined comparison, along with an example of how to call it:

```
1 #include <algorithm> // for std::swap, use <utility> instead if C++11
2 #include <iostream>
3
4 // Note our user-defined comparison is the third parameter
5 void selectionSort(int *array, int size, bool (*comparisonFcn)(int, int))
6 {
7     // Step through each element of the array
8     for (int startIndex = 0; startIndex < size; ++startIndex)
9     {
10         // bestIndex is the index of the smallest/largest element we've encountered so far.
11         int bestIndex = startIndex;
12
13         // Look for smallest/largest element remaining in the array (starting at startIndex+1)
14         for (int currentIndex = startIndex + 1; currentIndex < size; ++currentIndex)
15         {
16             // If the current element is smaller/larger than our previously found smallest
17             if (comparisonFcn(array[bestIndex], array[currentIndex])) // COMPARISON DONE HERE
18                 // This is the new smallest/largest number for this iteration
19                 bestIndex = currentIndex;
20         }
21
22         // Swap our start element with our smallest/largest element
23         std::swap(array[startIndex], array[bestIndex]);
24     }
25 }
26
27 // Here is a comparison function that sorts in ascending order
28 // (Note: it's exactly the same as the previous ascending() function)
29 bool ascending(int x, int y)
30 {
31     return x > y; // swap if the first element is greater than the second
32 }
33
34 // Here is a comparison function that sorts in descending order
35 bool descending(int x, int y)
36 {
37     return x < y; // swap if the second element is greater than the first
38 }
39
40 // This function prints out the values in the array
41 void printArray(int *array, int size)
42 {
43     for (int index=0; index < size; ++index)
44         std::cout << array[index] << " ";
45     std::cout << '\n';
46 }
47
48 int main()
49 {
50     int array[9] = { 3, 7, 9, 5, 6, 1, 8, 2, 4 };
51
52     // Sort the array in descending order using the descending() function
53     selectionSort(array, 9, descending);
54     printArray(array, 9);
55 }
```

```

56 // Sort the array in ascending order using the ascending() function
57 selectionSort(array, 9, ascending);
58 printArray(array, 9);
59
60 return 0;
61 }

```

Comment the content below. Ask students to do this in Practical Two The caller can even define their own strange comparison functions:

```

1 bool evensFirst(int x, int y)
2 {
3     // if x is even and y is odd, x goes first (no swap needed)
4     if ((x % 2 == 0) && !(y % 2 == 0))
5         return false;
6
7     // if x is odd and y is even, y goes first (swap needed)
8     if (!(x % 2 == 0) && (y % 2 == 0))
9         return true;
10
11     // otherwise sort in ascending order
12     return ascending(x, y);
13 }
14
15 int main()
16 {
17     int array[9] = { 3, 7, 9, 5, 6, 1, 8, 2, 4 };
18
19     selectionSort(array, 9, evensFirst);
20     printArray(array, 9);
21
22     return 0;
23 }

```

1.7.3 Providing default functions

If you're going to allow the caller to pass in a function as a parameter, it can often be useful to provide some standard functions for the caller to use for their convenience. For example, in the selection sort example above, providing the ascending() and descending() function along with the selectionSort() function would make the caller's life easier, as they wouldn't have to rewrite ascending() or descending() every time they want to use them.

You can even set one of these as a default parameter:

```

1 // Default the sort to ascending sort
2 void selectionSort(int *array, int size, bool (*comparisonFcn)(int, int) = ascending);

```

In this case, as long as the user calls selectionSort normally (not through a function pointer), the comparisonFcn parameter will default to ascending.

1.7.4 Making function pointers prettier with typedef or type aliases

The syntax for pointers to functions is ugly. However, typedefs can be used to make pointers to functions look more like regular variables:

```

1 typedef bool (*validateFcn)(int, int);

```

This defines a typedef called validateFcn that is a pointer to a function that takes two ints and returns a bool.

Now instead of doing this:

```

1 bool validate(int x, int y, bool (*fcnPtr)(int, int)); // ugly

```

You can do this:

```
1 bool validate(int x, int y, validateFcn pfcn) // clean
```

Which reads a lot nicer! However, the syntax to define the typedef itself can be difficult to remember.

In C++11, you can instead use type aliases to create aliases for function pointers types:

```
1 using validateFcn = bool(*)(int, int); // type alias
```

This reads more naturally than the equivalent typedef, since the name of the alias and the alias definition are placed on opposite sides of the equals sign.

Using a type alias is identical to using a typedef:

```
1 bool validate(int x, int y, validateFcn pfcn) // clean
```

1.7.5 Using std::function in C++11

Introduced in C++11, an alternate method of defining and storing function pointers is to use std::function, which is part of the standard library <functional> header. To define a function pointer using this method, declare a std::function object like so:

```
1 #include <functional>
2 bool validate(int x, int y, std::function<bool(int, int)> fcn); // std::function method that returns a bool and takes two
   int parameters
```

As you see, both the return type and parameters go inside angled brackets, with the parameters inside parenthesis. If there are no parameters, the parentheses can be left empty. Although this reads a little more verbosely, its also more explicit, as it makes it clear what the return type and parameters expected are (whereas the typedef method obscures them).

Updating our earlier example with std::function:

```
1 #include<iostream>
2 #include<functional>
3 using namespace std;
4 int square(int x)
5 {
6     return x * x;
7 }
8
9 int cube(int x)
10 {
11     return x * x * x;
12 }
13
14 int main()
15 {
16     function<int(int)> fcnPtr = square;
17     cout << "square() = " << fcnPtr(5) << endl; //calling function square
18     cout << "The address fcnPtr is " << &fcnPtr << endl;
19     fcnPtr = cube; // fcnPtr now points to function cube
20     cout << "cube() = " << fcnPtr(5) << endl; //calling function square
21     cout << "The address of fcnPtr is now " << &fcnPtr << endl;
22
23     return 0;
24 }
```

And Figure 9 shows the output on this code

Do you notice something from the output of this code?

```
square() = 25  
The address fcnPtr is 0x401530  
cube() = 125  
The address of fcnPtr is now 0x401540
```

Figure 9: Output of function pointers using functional code above

1.7.6 Conclusion

Function pointers are useful primarily when you want to store functions in an array (or other structure), or when you need to pass a function to another function.