Reviews Pointers: C++ (Part 2)

Contents

- Using const with Pointers
- Relation between Pointers and Arrays
- Arrays of Pointers
- Function Pointers

Using const with Pointers

- Many possibilities exist for using (or not using) const with function parameters
- How to choose the most appropriate one: the principle of least privilege

- A function that takes a one-dimensional array and its size as arguments for printing purpose: The array's size does not change; even not the array itself.

- Four ways to pass a pointer to a function
  - a nonconstant pointer to nonconstant data,
  - a nonconstant pointer to constant data
  - a constant pointer to nonconstant data and
  - a constant pointer to constant data.

- Each combination provides a different level of access privilege

Using const with Pointers

- 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.
  - Example: int *countPtr does not include const.

- 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

```
const int *countPtr;
```

Using const with Pointers

```
// Fig. 8.10: fig08_10.cpp
// Attempting to modify data through a
// nonconstant pointer to constant data.

void f( const int * ); // prototype

int main()
{
   int y;

   f( &y ); // f attempts illegal modification
} // end main

// xPtr cannot modify the value of constant variable to which it points
void f( const int *xPtr )
{
   *xPtr = 100; // error: cannot modify a const object
} // end function f
```

Using const with Pointers

- 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.
  - Example: An array name (constant pointer to the beginning of the array)

```
int * const ptr = &x; // const pointer must be initialized
```

- The minimum access privilege is granted by a **constant pointer to constant data**.
  - Example: 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)

```
// ptr is a constant pointer to a constant integer.
// ptr always points to the same location; the integer
// at that location cannot be modified.
const int *const ptr = &x;
```

Pointers and Arrays

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

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

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

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

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

```
&b[ 3 ]
```

can be written with the pointer expression

```
bPtr + 3
```

```
*( b + 3 )
```

also refers to the array element `b[ 3 ]`.

Pointer-based String Processing

Four notations for referring to array elements

```
Array b printed with:

Array subscript notation
b[0] = 10
b[1] = 20
b[2] = 30
b[3] = 40
```

```
Pointer/offset notation where the pointer is the array name
*(b + 0) = 10
*(b + 1) = 20
*(b + 2) = 30
*(b + 3) = 40
```

```
Pointer subscript notation
bPtr[0] = 10
bPtr[1] = 20
bPtr[2] = 30
bPtr[3] = 40
```

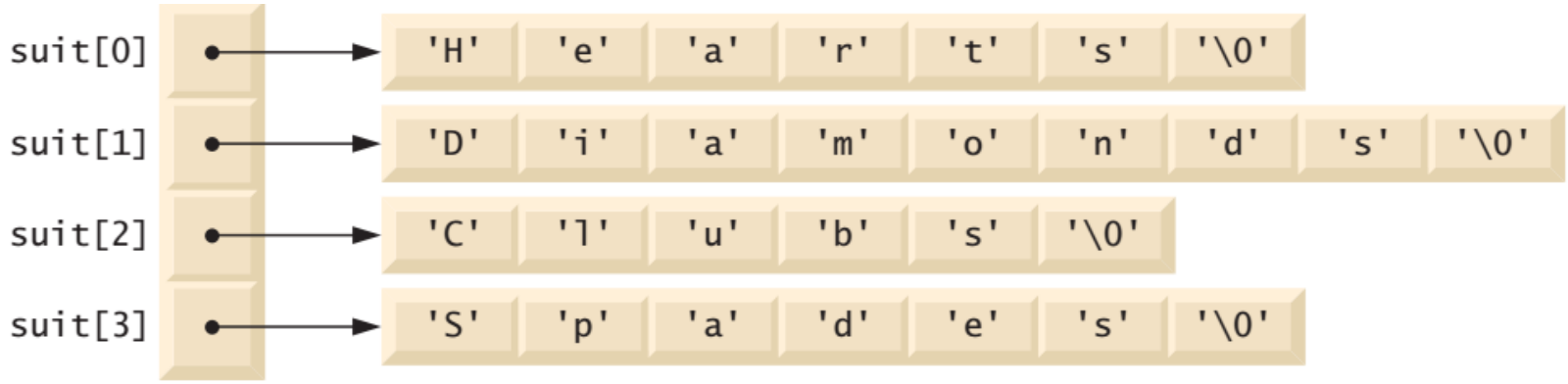
```
Pointer/offset notation
*(bPtr + 0) = 10
*(bPtr + 1) = 20
*(bPtr + 2) = 30
*(bPtr + 3) = 40
```

```cpp
// Using subscripting and pointer notations with arrays.
#include <iostream>
using namespace std;

int main()
{
   int b[] = { 10, 20, 30, 40 }; // create 4-element array b
   int *bPtr = b; // set bPtr to point to array b

   // output array b using array subscript notation
   cout << "Array b printed with:\n\nArray subscript notation\n";

   for ( int i = 0; i < 4; i++ )
      cout << "b[" << i << "] = " << b[ i ] << '\n';

   // output array b using the array name and pointer/offset notation
   cout << "\nPointer/offset notation where "
      << "the pointer is the array name\n";

   for ( int offset1 = 0; offset1 < 4; offset1++ )
      cout << "*(b + " << offset1 << ") = " << *( b + offset1 ) << '\n';

   // output array b using bPtr and array subscript notation
   cout << "\nPointer subscript notation\n";

   for ( int j = 0; j < 4; j++ )
      cout << "bPtr[" << j << "] = " << bPtr[ j ] << '\n';

   cout << "\nPointer/offset notation\n";

   // output array b using bPtr and pointer/offset notation
   for ( int offset2 = 0; offset2 < 4; offset2++ )
      cout << "*(bPtr + " << offset2 << ") = "
         << *( bPtr + offset2 ) << '\n';
} // end main
```

Arrays of Pointers

- A common use of such a data structure is to form an array of pointer-based strings, referred to simply as a **string array**.
- Each entry in the array is a string, but in C++ a string is essentially a pointer to its first character
- So each entry in an array of strings is simply a pointer to the first character of a string.

```
const char * const suit[ 4 ] =
   { "Hearts", "Diamonds", "Clubs", "Spades" };
```



Even though the suit array is fixed in size, it provides access to character strings of any length

Function Pointers

- A **pointer to a function** contains the function's address in memory.
- Like array's name, a function's name is actually the starting address in memory of the code that performs the function's 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.

Sorting using Function Pointers

```
Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 1

Data items in original order
   2   6   4   8  10  12  89  68  45  37
Data items in ascending order
   2   4   6   8  10  12  37  45  68  89
```

```
Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 2

Data items in original order
   2   6   4   8  10  12  89  68  45  37
Data items in descending order
  89  68  45  37  12  10   8   6   4   2
```

```
// Multipurpose sorting program using function pointers.
#include <iostream>
#include <iomanip>
using namespace std;

// prototypes
void selectionSort( int [], const int, bool (*)( int, int ) );
void swap( int * const, int * const );
bool ascending( int, int ); // implements ascending order
bool descending( int, int ); // implements descending order
```

```
// multipurpose selection sort; the parameter compare is a pointer to
// the comparison function that determines the sorting order
void selectionSort( int work[], const int size,
                    bool (*compare)( int, int ) )
{
   int smallestOrLargest; // index of smallest (or largest) element

   // loop over size - 1 elements
   for ( int i = 0; i < size - 1; i++ )
   {
      smallestOrLargest = i; // first index of remaining vector

      // loop to find index of smallest (or largest) element
      for ( int index = i + 1; index < size; index++ )
         if ( !(*compare)( work[ smallestOrLargest ], work[ index ] ) )
            smallestOrLargest = index;

      swap( &work[ smallestOrLargest ], &work[ i ] );
   } // end if
} // end function selectionSort
```

```
bool ( *compare )( int, int )
```

```
bool *compare( int, int )
```

```
( *compare )( work[ smallestOrLargest ], work[ index ] )
```

```
// determine whether element a is less than
// element b for an ascending order sort
bool ascending( int a, int b )
{
   return a < b; // returns true if a is less than b
} // end function ascending
```

```
// determine whether element a is greater than
// element b for a descending order sort
bool descending( int a, int b )
{
   return a > b; // returns true if a is greater than b
} // end function descending
```

```
// swap values at memory locations to which
// element1Ptr and element2Ptr point
void swap( int * const element1Ptr, int * const element2Ptr )
{
   int hold = *element1Ptr;
   *element1Ptr = *element2Ptr;
   *element2Ptr = hold;
} // end function swap
```

```c
int main()
{
   const int arraySize = 10;
   int order; // 1 = ascending, 2 = descending
   int counter; // array index
   int a[ arraySize ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
```

```
   cout << "Enter 1 to sort in ascending order,\n"
      << "Enter 2 to sort in descending order: ";
   cin >> order;
   cout << "\nData items in original order\n";

   // output original array
   for ( counter = 0; counter < arraySize; counter++ )
      cout << setw( 4 ) << a[ counter ];

   // sort array in ascending order; pass function ascending
   // as an argument to specify ascending sorting order
   if ( order == 1 )
   {
      selectionSort( a, arraySize, ascending );
      cout << "\nData items in ascending order\n";
   } // end if

   // sort array in descending order; pass function descending
   // as an argument to specify descending sorting order
   else
   {
      selectionSort( a, arraySize, descending );
      cout << "\nData items in descending order\n";
   } // end else part of if...else

   // output sorted array
   for ( counter = 0; counter < arraySize; counter++ )
      cout << setw( 4 ) << a[ counter ];

   cout << endl;
} // end main
```