CSCU9Y4 Programming Language Paradigms Practical 6

Pointers and arrays

The concept of arrays is related to that of pointers. In fact, arrays work very much like pointers to their first elements, and, actually, an array can always be implicitly converted to the pointer of the proper type. For example, consider these two declarations:

```
int myarray [20];
int * mypointer;
```

The following assignment operation would be valid:

```
mypointer = myarray;
```

Here, `myarray` is a synonym for `&myarray[0]`. After the assignment above, `mypointer` and `myarray` would be equivalent and would have very similar properties. The main difference being that `mypointer` can be assigned a different address, whereas `myarray` can never be assigned anything, and will always represent the same block of 20 elements of type int. Therefore, the following assignment would not be valid:

```
myarray = mypointer;
```

Let's see an example that mixes arrays and pointers:

```
1   // more pointers
2
3   int main ()
4   {
5     int numbers[5];
6     int * p;
7     p = numbers;  *p = 10;
8     p++;  *p = 20;
9     p = &numbers[2];  *p = 30;
10    p = numbers + 3;  *p = 40;
11    p = numbers;  *(p+4) = 50;
12    for (int n=0; n<5; n++)
13      printf("%d, ", numbers[n]);
14    return 0;
15  }
```

The output is

```
10, 20, 30, 40, 50,
```

Why? Make sure you understand what all the different assignment statements do.

Pointers and arrays support the same set of operations, with the same meaning for both. The main difference being that pointers can be assigned new addresses, while arrays cannot. Essentially, this is because brackets in an array expression are a dereferencing operator known as

offset operator. They dereference the variable they follow just as * does, but they also add the number between brackets to the address being dereferenced. For example:

```
 a[5] = 0;       // a [offset of 5] = 0
 *(a+5) = 0;     // pointed by (a+5) = 0
```

These two expressions are equivalent and valid, not only if a is a pointer, but also if a is an array. Remember that if an array, its name can be used just like a pointer to its first element.

Pointers as function parameters (1)

Recall that if we have call by value then we can implement call by reference by using pointers. The example from the lectures is a swap function. It's repeated here:

```
void swap(int* first, int* second)
{
  int temp;
  temp = *first;
  *first = *second;
  *second = temp;
}

int main ()
{
  int dee, dum;
  dee = 42; dum = 56;

  swap(&dee,&dum);
  printf("Dee is %d\n",dee);
  printf("Dum is %d\n",dum);
  return 0;
}
```

Run this and convince yourself that you understand how it works. Compare this with a version without pointers:

```
void swap(int first, int second)
{
  int temp;
  temp = first;
  first = second;
  second = temp;
}

int main ()
{
  int dee, dum;
  dee = 42; dum = 56;
```

```
swap(dee,dum);
printf("Dee is %d\n",dee);
printf("Dum is %d\n",dum);
return 0;
}
```

This version fails to actually swap the values of dee and dum.

Pointers and const

Pointers can be used to access a variable by its address, and this access may include modifying the value pointed to. But it is also possible to declare pointers that can access the pointed value to read it, but not to modify it. For this, it is enough with qualifying the type pointed by the pointer as const. For example:

```
1 int x;
2 int y = 10;
3 const int * p = &y;
4 x = *p;         // ok: reading p
5 *p = x;         // error: modifying p, which is const-qualified
```

Here p points to a variable, but points to it in a const-qualified manner, meaning that it can read the value pointed, but it cannot modify it. Note also, that the expression &y is of type int*, but this is assigned to a pointer of type const int*. This is allowed: a pointer to non-const can be implicitly converted to a pointer to const. But not the other way around! As a safety feature, pointers to const are not implicitly convertible to pointers to non-const.

Pointers as function parameters (2)

One of the use cases of pointers to const elements is as function parameters: a function that takes a pointer to non-const as parameter can modify the value passed as argument, while a function that takes a pointer to const as parameter cannot.

```
1   // pointers as arguments:
2
3   void increment_all (int* start, int* stop)
4   {
5     int * current = start;
6     while (current != stop) {
7       ++(*current);  // increment value pointed
8       ++current;     // increment pointer
9     }
10  }
11
```

```
12  void print_all (const int* start, const int* stop)
13  {
14    const int * current = start;
15    while (current != stop) {
16      printf("%d\n",*current);
17      ++current;     // increment pointer
18
19    }
20  }
21
22  int main ()
23  {
24    int numbers[] = {10,20,30};
25    increment_all (numbers,numbers+3);
26    print_all (numbers,numbers+3);
27    return 0;
28  }
```

The output is

```
11
21
31
```

Note that `print_all` uses pointers that point to constant elements. These pointers point to constant content they cannot modify, but they are not constant themselves: i.e., the pointers can still be incremented or assigned different addresses, although they cannot modify the content they point to.

And this is where a second dimension to constness is added to pointers: Pointers can also be themselves const. This is specified by appending const to the pointed type (after the asterisk):

```
1  int x;
2  int * p1 = &x;  // non-const pointer to non-const int
3  const int * p2 = &x;   // non-const pointer to const int
4  int * const p3 = &x;   // const pointer to non-const int
5  const int * const p4 = &x;    // const pointer to const int
```

The syntax with const and pointers is definitely tricky, and recognizing the cases that best suit each use tends to require some experience. In any case, it is important to get constness with pointers (and references) right sooner rather than later, but you should not worry too much about grasping everything if this is the first time you are exposed to the mix of const and pointers.

To add a little bit more confusion to the syntax of const with pointers, the const qualifier can either precede or follow the pointed type, with the exact same meaning:

```
const int * p2a = &x;  // non-const pointer to const int
```

```
int const * p2b = &x;  // also non-const pointer to const int
```

As with the spaces surrounding the asterisk, the order of const in this case is simply a matter of intensely-debated style.

Checkpoint

Now attempt these questions and demonstrate to a tutor that you have completed them:

Question 1

Write a program that initialises an array of double and then copies the contents of the array into two other arrays. Here is an example showing how the functions should be called, given the following declarations:

```
double source[4] = {1,2.3,4.5,6.7};
double destination1[4];
double destination2[4];
copy_array(source, destination, 4);
copy_ptr(source, destination, 4);
```

Hints: Declare all arrays in the main program.
To make the first copy, make a function `copy_array()`, which uses the array notation (the square brackets []) to access the elements of the array.
To make the second copy, write a function `copy_ptr()`, which uses the pointer notation and pointer incrementing to access the elements of the arrays.
Have each function take as function arguments the name of the target array and the number of elements to be copied. Test the two copy functions in the main program.

Question 2

Write a function `dif()`, which uses the pointer notation in the function body and returns the difference between the largest and smallest elements in an array of double. Test the function in the main program.

Question 3

Write a function `add()`, which uses the pointer notation in the function body and adds the elements of two one-dimensional arrays of same size. The results should be stored in a third array. The function should take as arguments of the names of the three arrays and the array dimension. Test the function in the main program.

Optional extras:

Write a program that initialises a two-dimensional array of double and uses the second copy functions (i.e. the one using pointer in Question 1) to copy it to a second two-dimensional array. Use the one-dimensional copy function to copy the sub-arrays of the two-dimensional array, one-by-one.

Hint: Remember how the two-dimensional arrays are stored in the memory!

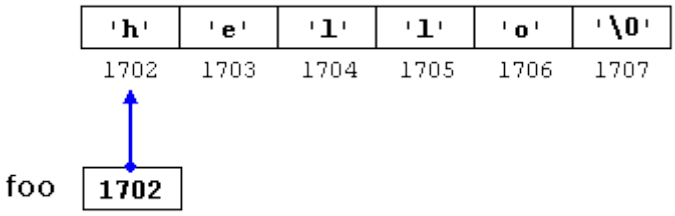
Pointers and string literals

As pointed earlier, string literals are arrays containing null-terminated character sequences. In earlier sections, string literals have been used to be directly inserted into cout, to initialize strings and to initialize arrays of characters.

But they can also be accessed directly. String literals are arrays of the proper array type to contain all its characters plus the terminating null-character, with each of the elements being of type const char (as literals, they can never be modified). For example:

```
const char * foo = "hello";
```

This declares an array with the literal representation for "hello", and then a pointer to its first element is assigned to foo. If we imagine that "hello" is stored at the memory locations that start at address 1702, we can represent the previous declaration as:



Note that here foo is a pointer and contains the value 1702, and not 'h', nor "hello", although 1702 indeed is the address of both of these.

The pointer foo points to a sequence of characters. And because pointers and arrays behave essentially in the same way in expressions, foo can be used to access the characters in the same way arrays of null-terminated character sequences are. For example:

```
1    *(foo+4)
2    foo[4]
```

Both expressions have a value of 'o' (the fifth element of the array).

References: material taken from http://www.cplusplus.com/doc/tutorial/pointers/