CSCU9Y4 Programming Language Paradigms Practical 5

The 'default' C/C++ IDE in the labs is Netbeans, with gcc/g++ available at the command line. Alternatively, if outside the labs, http://www.learn-c.org/en/ is a great option.

There's a lot of text explaining pointers in this week's lab. Read the text, try the examples and verify your understanding.

Pointers in C/C++ programming

Variables can be explained as locations in the computer's memory which are accessed by their identifier (their name). This way, the program does not need to care about the physical address of the data in memory; it simply uses the identifier whenever it needs to refer to the variable. Recall, this is the declaration-reference binding (or lifetime).

For a C/C++ program, the memory of a computer is like a succession of memory cells, each one byte in size, and each with a unique address. These single-byte memory cells are ordered in a way that allows data representations larger than one byte to occupy memory cells that have consecutive addresses.

This way, each cell can be easily located in the memory by means of its unique address. For example, the memory cell with the address 1776 always follows immediately after the cell with address 1775 and precedes the one with 1777, and is exactly one thousand cells after 776 and exactly one thousand cells before 2776.

When a variable is declared, the memory needed to store its value is assigned a specific location in memory (its memory address). Generally, C/C++ programs do not actively decide the exact memory addresses where its variables are stored. Fortunately, that task is left to the environment where the program is run - generally, an operating system that decides the particular memory locations on runtime. However, it may be useful for a program to be able to obtain the address of a variable during runtime in order to access data cells that are at a certain position relative to it.

Reference (or "address of") operator: &

The address of a variable can be obtained by preceding the name of a variable with an ampersand sign (&), known as reference operator, and which can be literally translated as "address of". For example:

```
foo = &myvar;
```

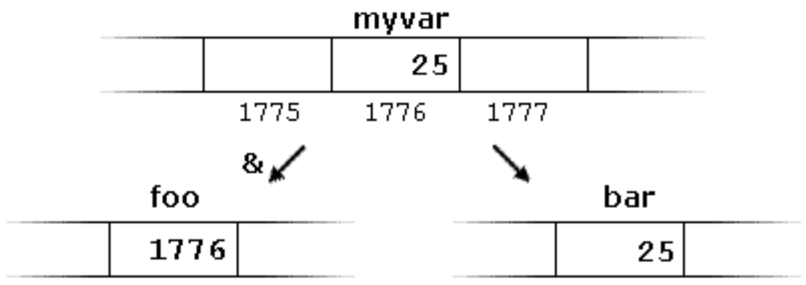
This would assign the address of variable `myvar` to `foo`; by preceding the name of the variable `myvar` with the reference operator (&), we are no longer assigning the content of the variable itself to `foo`, but its address.

The actual address of a variable in memory cannot be known before runtime, but let's assume, in order to help clarify some concepts, that `myvar` is placed during runtime in the memory address 1776.

In this case, consider the following code fragment:

```
myvar = 25;
foo = &myvar;
bar = myvar;
```

The values contained in each variable after the execution of this are shown in the following diagram:



First, we have assigned the value 25 to `myvar` (a variable whose address in memory we assumed to be 1776).

The second statement assigns `foo` the address of `myvar`, which we have assumed to be 1776.

Finally, the third statement, assigns the value contained in `myvar` to bar. This is a standard assignment operation.

The main difference between the second and third statements is the reference operator (&).

The variable that stores the address of another variable (like foo in the previous example) is what in C is called a *pointer*. Pointers are a very powerful feature of the language that has many uses in lower level programming. A bit later, we will see how to declare and use pointers.

Dereference (or 'indirection') operator: *

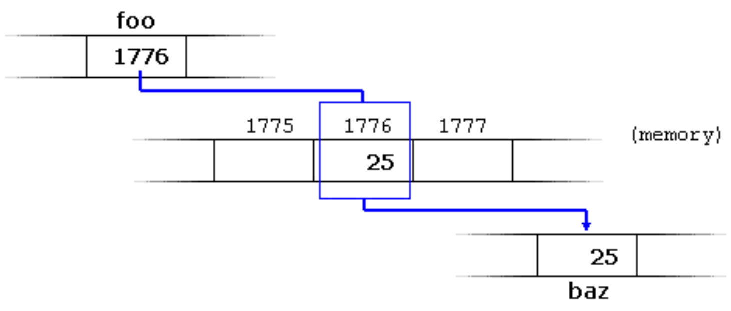
As just seen, a variable which stores the address of another variable is called a pointer. Pointers are said to "point to" the variable whose address they store.

An interesting property of pointers is that they can be used to access the variable they point to directly. This is done by preceding the pointer name with the dereference operator (*). The operator itself can be read as "value pointed to by".

Therefore, following with the values of the previous example, the following statement:

```
 baz = *foo;
```

This could be read as: "assign to `baz` the value contained in the location pointed to by `foo`", and the statement would actually assign the value 25 to `baz`, since `foo` is 1776, and the value in the location 1776 (following the example above) would be 25.



It is important to clearly differentiate that foo refers to the value 1776, while `*foo` (with an asterisk * preceding the identifier) refers to the value stored at address 1776, which in this case is 25. Notice the difference of including or not including the dereference operator (I have added an explanatory comment of how each of these two expressions could be read):

```
baz = foo;   // baz equal to foo (1776)
baz = *foo;  // baz equal to value pointed to by foo (25)
```

The reference and dereference operators are thus complementary:

- &is the reference operator, and can be read as "address of"
- *is the dereference operator, and can be read as "value pointed to by"

Thus, they have sort of opposite meanings: a variable referenced with & can be dereferenced with *.

Earlier, we performed the following two assignment operations:

```
myvar = 25;
foo = &myvar;
```

Right after these two statements, all of the following expressions would give true as result:

```
myvar == 25
&myvar == 1776
foo == 1776
*foo == 25
```

The first expression is quite clear, considering that the assignment operation performed on `myvar` was `myvar=25`. The second one uses the reference operator (&), which returns the address of `myvar`, which we assumed it to have a value of 1776. The third one is somewhat obvious, since the second expression was true and the assignment operation performed on foo was `foo=&myvar`. The fourth expression uses the dereference operator (*) that can be read as "value pointed to by", and the value pointed to by foo is indeed 25.

So, after all that, you may also infer that for as long as the address pointed by `foo` remains unchanged, the following expression will also be true:

```
*foo == myvar
```

Declaring pointers

Due to the ability of a pointer to directly refer to the value that it points to, a pointer has different properties when it points to a char than when it points to an int or a float. Once dereferenced, the type needs to be known. And for that, the declaration of a pointer needs to include the data type the pointer is going to point to.

The declaration of pointers follows this syntax: ‘`type * name;`’ where type is the data type pointed to by the pointer. This type is not the type of the pointer itself, but the type of the data the pointer points to. For example:

```
int * number;
char * character;
double * decimals;
```

Aside: The following are equivalent (your life will be easier if you choose your favourite here and stick to a single convention):

```
    type * name;
    type* name;
    type *name;
```

The numbered declarations above are for pointers. Each one is intended to point to a different data type; however, being pointers, they occupy the same amount of space in memory (the size in memory of a pointer depends on the platform where the program runs). Nevertheless, the data to which they point to do not occupy the same amount of space nor are of the same type: the first one points to an int, the second one to a char, and the last one to a double. Therefore, although these three example variables are all pointers, they actually have different types: int*, char*, and double* respectively, depending on the type they point to.

Note that the asterisk (*) used when declaring a pointer only means that it is a pointer (it is partof its type compound specifier), and should not be confused with the dereference operator seen a bit earlier, but which is also written with an asterisk (*). **They are two different things represented with the same sign.**

Let's see an example on pointers:

```
// my first pointer
#include <stdio.h>

int main ()  {
  int firstvalue, secondvalue;
  int * mypointer;

  mypointer = &firstvalue;
  *mypointer = 10;
  mypointer = &secondvalue;
  *mypointer = 20;
  printf("firstvalue is %d\n", firstvalue);
  printf("secondvalue is %d\n", secondvalue);
  return 0;
}
```

The output is

```
    firstvalue is 10
    secondvalue is 20
```

Notice that even though neither `firstvalue` nor `secondvalue` are directly set any value in the program, both end up with a value set indirectly through the use of `mypointer`. This is how it happens:

- First, `mypointer` is assigned the address of `firstvalue` using the reference operator (&). Then, the value pointed to by `mypointer` is assigned 10. Because, at this moment, `mypointer` is pointing to the memory location of `firstvalue`, this in fact modifies the value of `firstvalue`.

- In order to demonstrate that a pointer may point to different variables during its lifetime in a program, the example repeats the process with `secondvalue` and that same pointer, `mypointer`.

Here is a second example, a little bit more elaborated. Try to get the results by hand before executing the programs. Draw pictures of the memory allocation: it helps!

```
// more pointers
#include <stdio.h>

int main ()  {
  int firstvalue = 5, secondvalue = 15;
  int * p1, * p2;

  p1 = &firstvalue;     // p1 = address of firstvalue
  p2 = &secondvalue;  // p2 = address of secondvalue
  *p1 = 10;          // value pointed to by p1 = 10
  *p2 = *p1;      // value pointed to by p2 = value pointed by p1
  p1 = p2;           // p1 = p2 (value of pointer is copied)
  *p1 = 20;          // value pointed by p1 = 20
  printf("firstvalue is %d\n", firstvalue);
  printf("secondvalue is %d\n", secondvalue);
  return 0;
}
```

The output is

```
firstvalue is 10
secondvalue is 20
```

Each assignment operation includes a comment on how each line could be read: i.e., replacing ampersands (&) by "address of", and asterisks (*) by "value pointed to by".
Add some more printf statements to check the values as you go along if you're not sure.
You can print an address (a pointer value) in hexadecimal with

```
printf("p1 is Ox%x\n",p1);
```

You should see addresses that look something like this: "`0xbfe55918`". The initial characters "`0x`" tell you that hexadecimal notation is being used; the remainder of the digits give the address itself.

Notice that there are expressions with pointers p1 and p2, both with and without the dereference operator (*). The meaning of an expression using the dereference operator (*) is very different from one that does not. When the * operator precedes the pointer name, the expression refers to the value being pointed to, while when a pointer name appears without the * operator, it refers to the value of the pointer itself (i.e., the address of what the pointer is pointing to).

Another thing that may call your attention is the line:

```
int * p1, * p2;
```

This declares the two pointers used in the previous example. But notice that there is an asterisk (*) for each pointer, in order for both to have type int* (pointer to int). This is required due to the precedence rules. Note that if, instead, the code was:

```
int * p1, p2;
```

`p1` would indeed be of type `int*`, but `p2` would be of type int. Spaces do not matter at all for this purpose. But anyway, simply remembering to put one asterisk per pointer is enough for most pointer users interested in declaring multiple pointers per statement. Or even better: use a different statement for each variable.

Checkpoint Question 1 (this is repeated at the end for convenience)

For the second example program, what are the values of variables firstvalue and secondvalue for each of its intermediate steps from Line 8 to Line 13? You should also explain why they are those values.

Pointer initialization

Pointers can be initialized to point to specific locations at the very moment they are defined:

```
int myvar;
int * myptr = &myvar;
```

The resulting state of variables after this code is the same as after:

```
int myvar;
int * myptr;
myptr = &myvar;
```

When pointers are initialized, what is initialized is the address they point to (i.e., myptr), never the value being pointed (i.e., *myptr). Therefore, the code above shall not be confused with:

```
int myvar;
int * myptr;
*myptr = &myvar;
```

The asterisk (*) in the pointer declaration (line 2) only indicates that it is a pointer, it is not the dereference operator (as in line 3). Both things just happen to use the same sign: *. As always, spaces are not relevant, and never change the meaning of an expression.

Pointers can be initialized either to the address of a variable (such as in the case above), or to the value of another pointer:

```
int myvar;
int *foo = &myvar;
int *bar = foo;
```

<u>Question 2 (repeated at the end)</u>

Please complete the following program to demonstrate the working of pointers. Like Question 1, you should also explain the reason why you get the outputs.

```
int main() {
   int *pc, c;
   c = 22;

   // Task 1: output address and value of c.
   // Task 2: assign pointer pc the address of c. Output address and value of pc.
   // Task 3: assign the value 11 to c. Output address and value of pc.
   // Task 4: assign the value 2 to the value pointed to by pc. Output address and value of c.

   return 0;
}
```

Pointer arithmetic

To conduct arithmetical operations on pointers is a little different than to conduct them on regular integer types. To begin with, only addition and subtraction operations are allowed; the others make no sense in the world of pointers. But both addition and subtraction have a slightly different behavior with pointers, according to the size of the data type to which they point.

When fundamental data types were introduced, we saw that types have different sizes. For example: char always has a size of 1 byte, short is generally larger than that, and int and long are even larger; the exact size of these being dependent on the system. For example, let's imagine that in a given system, char takes 1 byte, short takes 2 bytes, and long takes 4.

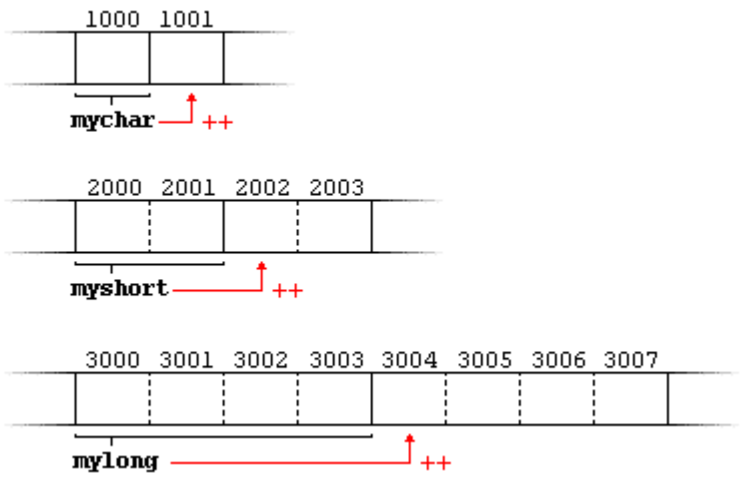
Suppose now that we define three pointers in this compiler:

```
char *mychar;
short *myshort;
long *mylong;
```

and that we know that they point to the memory locations 1000, 2000, and 3000, respectively. Therefore, if we write:

```
++mychar;
++myshort;
++mylong;
```

`mychar`, as one would expect, would contain the value 1001. But not so obviously, `myshort` would contain the value 2002, and `mylong` would contain 3004, even though they have each been incremented only once. The reason is that, when adding one to a pointer, the pointer is made to point to the following element of the same type, and, therefore, the size in bytes of the type it points to is added to the pointer. Check this out: implement the code and print the addresses associated with each pointer.



This is applicable both when adding and subtracting any number to a pointer. It would happen exactly the same if we wrote:

```
mychar = mychar + 1;
myshort = myshort + 1;
mylong = mylong + 1;
```

Question 3

Write a short C program that declares and initializes (to any value you like) a `double`, an `int`, and a `char`. Next declare and initialize a pointer to each of the three variables. Your program should then print the address of, and value stored in, and the memory size (in bytes) of each of the six variables. Use the `sizeof` operator to determine the memory size allocated for each variable.

Checkpoint

Now demonstrate to a tutor that you have done the following:

Question 1

For the second example program, what are the values of variables firstvalue and secondvalue for each of its intermediate steps from Line 8 to Line 13? You should also explain why they are those values.

Question 2

Demonstrate your example program with pc and c. Like Question 1, you should also explain the reason why you get the outputs.

Question 3

Demonstrate your example program with a `double`, an `int`, and a `char`.

More on increment and decrement and operator precedence

Regarding the increment (++) and decrement (--) operators, they both can be used as either prefix or suffix of an expression, with a slight difference in behavior: as a prefix, the increment happens before the expression is evaluated, and as a suffix, the increment happens after the expression is evaluated. This also applies to expressions incrementing and decrementing pointers, which can become part of more complicated expressions that also includes dereference operators (*). Remembering operator precedence rules, we can recall that postfix operators, such as increment and decrement, have higher precedence than prefix operators, such as the dereference operator (*). Therefore, the following expression:

```
*p++
```

is equivalent to `*(p++)`. And what it does is to increase the value of `p` (so it now points to the next element), but because ++ is used as postfix, the whole expression is evaluated as the value pointed originally by the pointer (the address it pointed to before being incremented).

Essentially, these are the four possible combinations of the dereference operator with both the prefix and suffix versions of the increment operator (the same being applicable also to the decrement operator):

```
*p++     // same as *(p++): increment pointer, and dereference un-incremented address
*++p     // same as *(++p): increment pointer, and dereference incremented address
++*p     // same as ++(*p): dereference pointer, and increment the value it points to
(*p)++   // dereference pointer, and post-increment the value it points to
```

A typical -but not so simple- statement involving these operators is:

```
*p++ = *q++;
```

Because ++ has a higher precedence than *, both `p` and `q` are incremented, but because both increment operators (++) are used as postfix and not prefix, the value assigned to `*p` is `*q` before both `p` and `q` are incremented. And then both are incremented. It would be roughly equivalent to:

```
*p = *q;
++p;
++q;
```

Like always, parentheses reduce confusion by adding legibility to expressions.

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