Lecture 6 examples

November 26, 2018

1 Pointers and addresses

What is your pointer what is your number ...

1.1 1. New data type: pointer

- used to declare variables
- stores addresses of other variables
- can stor an address of other pointers

We will start by recalling sizeoff() to illustrate that all poiners have the same size

```
In [96]: #include <stdio.h>

         int main()
         {
             int a=3;
             double b=5.0;
             printf("Size of an int is %ld, and the size of a double is %ld\n", sizeof(a), siz

             int *pi;
             double *pd;
             printf("The sizes of pointers to an int is %ld and to a double %ld, so the same!\

             void *vp;
             printf("The sizes of pointer to void is %ld\n", sizeof(vp));
             //Please note that in this example we use %ld to print the size,
             //this is because this configuration is 64 bit, and a pointer would be equivalent
             //This behaviour is compiler and hardware dependent
         }
```

```
Size of an int is 4, and the size of a double is 8
The sizes of pointers to an int is 8 and to a double 8, so the same!
The sizes of pointer to void is 8
```

The size of all pointers in this example is 8B. Note that this depands on the compiler and hardware so during the laboratories you might see the result of sizeof being an int type, and the size 4B.

1.2 2. Initialize your pointers with addresses of variables, the & operator

- recall our use of function scanf()
- use & to retrive an address from a variable
- &variable_name returns an address of variable_name

```
In [97]: #include <stdio.h>

         int main()
         {
             int a = 3;
             int *pi = &a; // Here I assign address of a to be stored by pi

             //Use a cast to suppress warnings
             printf("Address of a is %ld, and the address pointed by pi %ld\n", (long int)&a,
             printf("And address of pi is %ld\n", (long int)&pi);
         }
```

```
Address of a is 140731347479324, and the address pointed by pi 140731347479324
And address of pi is 140731347479328
```

Note, that we printed addreses as long ints, and used casting tu suppress warnings.

1.3 3. Retrive / modify the value from pointer using Use * to retrive the value that is stored under the address stored by a pointer

- *p - returns the value

```
In [98]: #include <stdio.h>

         int main()
         {
             int a = 3;
             int b = 5;
             int *p;
             printf("a=%d, b=%d\n", a, b);
             p = &a;
             printf("p points to a value of %d\n", *p); //p points at a, so *p returns 3
             p = &b;
             printf("p points to a value of %d\n", *p); // p now points at b, so *p returns 5
         }
```

```
a=3, b=5
p points to a value of 3
p points to a value of 5
```

- the * operator can also be used to manipulate the value that is stored under the address pointed by p

- *p = 5 will set the value, of whatever is pointed by p to 5

```
In [99]: #include <stdio.h>

         int main()
         {
             int a = 3;
             printf("a=%d\n", a);

             int *p = &a;
             printf("p points to a value of %d\n", *p);

             *p = 10;
             printf("p points to a value of %d\n", *p);
             printf("a=%d\n", a);

             a = 20;
             printf("p points to a value of %d\n", *p);
             printf("a=%d\n", a);
         }
```

```
a=3
p points to a value of 3
p points to a value of 10
a=10
p points to a value of 20
a=20
```

1.4 4. Printing of addreses, the new format specifier, %p

- Prints pointers in a hexadecimal format, i.e. using 16 digits
- 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f
- 0x at the front is just an information that the number is printed in hexadecimal system

```
In [24]: #include <stdio.h>

         int main()
         {
             int a = 3;
             int *pi = &a;
             printf("Address of a is %p, and the address pointed by pi %p\n", &a, pi);

             printf("And address of pi is %p\n", &pi);
         }
```

```
Address of a is 0x7fff01aa20ac, and the address pointed by pi 0x7fff01aa20ac
And address of pi is 0x7fff01aa20b0
```

1.5 5. What is a pointer to a pointer?

- pointers can point to pointers

```
In [100]: #include <stdio.h>

          int main()
          {
              int a = 3;
              int *pi = &a;
              int **ppi = &pi;

              printf("a = %d\n", a);
              printf("pi = %p, *pi = %d\n", pi, *pi);
              printf("ppi = %p, *ppi = %p, **ppi=%d\n", ppi, *ppi, **ppi);
              // this can get pretty evil
          }
```

```
a = 3
pi = 0x7ffe8d91c944, *pi = 3
ppi = 0x7ffe8d91c948, *ppi = 0x7ffe8d91c944, **ppi=3
```

1.6 6. Pointer arythmetics, +,-

- Pointers are more than just a way of storring addresses of variables
- Thay serve as a basic in accesing data stored in memory
- It needs to be precisly understood what does it mean to add 1 to a pointer - this depands on the type of pointer
- To add, or substract means to move up or down the ampunt of bytes necessary to store a variable of a given type
  - 4B in case of int
  - 8B for doubles
  - 1B for characters, and so on

```
In [102]: #include <stdio.h>

          int main()
          {
              int *p = (int *)5;
              // we initialize the pointer with an address 5, normally we would initialize it 

              printf("And address of p is %p\n", p);
              p = p + 1; // We add 1 to p, since we work on integers the pointer now points to
              printf("And address of p is %p\n", p);
          }
```

```
And address of p is 0x5
And address of p is 0x9
```

So for int +1 adds 4. The reason is that the size of an int is 4B!

```
In [104]: #include <stdio.h>

          int main()
          {
              double *p = (double *)5;
              printf("And address of p is %p\n", p);
              p = p + 1; // And for a double this is 12, or d
              printf("And address of p is %p\n", p);
          }
```

So for a double +1 adds 8. The reason is that the size of an int is 8B!

- Note that d is equivalent to 13 in hexadecimal notation

So a +/- 1 means move the pointer up/down the memory line by the size of a variable to whhich it points.

1.7 7. Pointer to void

- we can not declare a variable of type void, put we can point to it
- we can not perform arithmetics, since the size of void is not known

The example below will not compile!

```
In [ ]: #include <stdio.h>
        int main(){
            void a; // this will not compile
        }
```

But this will:

```
In [111]: #include <stdio.h>
          int main(){
              void *a;
          }
```

Store an address of an integer using a void pointer, and than print it. Note that, whan printing we need to cast the pointer to the correct type (why?).

```
In [117]: #include <stdio.h>

          int main()
          {
              int a = 9;
              void *p = &a;

              printf("The value of a = %d. The address of a is &a = %p. And p points to p = %p
              printf("We can print the value pointed by p, but we need to cast it to (int *).\
          }
```

```
The value of a = 9. The address of a is &a = 0x7ffca43349fc. And p points to p = 0x7ffca43349f
We can print the value pointed by p, but we need to cast it to (int *).
The value is: *p=9
```

1.8 8. Let's do something bad! Store two ints in a double!

- Here we illustrate some consequences of using pointers
- We will attempt to store two ints in a single double
- Please mind, that in general this is not a good idea!

```
In [124]: #include <stdio.h>

          int main()
          {
              double d = 9;
              printf("The value of d is: d = %lf\n", d);

              int *p = (int *)&d;
              printf("Address of d is: &d = %p. And p points to p = %p\n", &d, p);
              printf("p = %p, (p+1) = %p\n", p, p+1);

              *p = 5;
              *(p+1) = 1000;

              printf("*p = %d\n*(p+1) = %d\n", *p, *(p+1));
              printf("The value of d is now: d = %lf\n", d);
          }
```

```
The value of d is: d = 9.000000
Address of d is: &d = 0x7ffcd2efa5a8. And p points to p = 0x7ffcd2efa5a8
p = 0x7ffcd2efa5a8, (p+1) = 0x7ffcd2efa5ac
*p = 5
*(p+1) = 1000
The value of d is now: d = 0.000000
```

1.9 9. Recall functions and function arguments

- pass by value
- pass with a pointer
- how to avoid global variables

Argument is passed **by value** - and is not modified by the function. The function works on a **copy**.

```
In [125]: #include <stdio.h>

          void fun(int a){
```

```
                   printf("\t a=%d, &a=%p\n", a, &a);
                   a = 500;
                   printf("\t a=%d\n", a);
               }

               int main()
               {
                   int b = 9;
                   printf("b=%d &b=%p\n", b, &b);
                   fun(b);
                   printf("b=%d\n", b);
               }
```

```
b=9 &b=0x7ffcc1bc21c4
         a=9, &a=0x7ffcc1bc21ac
         a=500
b=9
```

With global variable

```
In [126]: #include <stdio.h>

               int a;

               void fun(){
                   printf("\t a=%d, &a=%p\n", a, &a);
                   a = 500;
                   printf("\t a=%d\n", a);
               }

               int main()
               {
                   a = 9;
                   printf("a=%d &a=%p\n", a, &a);
                   fun();
                   printf("a=%d\n", a);
               }
```

```
a=9 &a=0x7f877fe2a034
         a=9, &a=0x7f877fe2a034
         a=500
a=500
```

The argument passed to the function is now **an address** to the variable, so all work is performed over the same region in the memory. The modyfications carry over, and are not lost!

```
In [128]: #include <stdio.h>
```

```c
void fun(int *a){
    printf("\t a=%d, &a=%p\n", *a, a);
    *a = 500;
    printf("\t a=%d\n", *a);
}

int main()
{
    int b = 9;
    printf("b=%d &b=%p\n", b, &b);
    fun(&b); // like scanf("", &b)
    printf("b=%d\n", b);
}
```

```
b=9 &b=0x7ffe78c0fb34
         a=9, &a=0x7ffe78c0fb34
         a=500
b=500
```