C PROGRAMMING
Lecture 5

1st semester 2023-2024

Pointers

- Review of variables
- Parameter passing
- Pointer declaration
- Pointer types
- Pointer arithmetics
- Examples

Remember variables

- Just to remember, what is a variable?
- A variable in a program is something with a name, the value of which can vary.
- It is assigned a specific block of memory to hold the value of that variable (address)
- The size of that block depends on the range over which the variable is allowed to vary (type)
- Imagine two "values" associated with the object. One is the value of the integer stored there and the other the "value" of the memory location, (address). Some refer these two values rvalue (right value) and lvalue (left value).
- Rvalues cannot be used on the left side of the assignment statement.
    - `10 = i;  //  is illegal`

Remember variables

```
 int nr1, nr2;
nr1 = 1;
nr2 = 2;   // observe this line
nr1 = nr2; // then observe this line
```

- the compiler interprets nr2 in first assignment as the address of the variable (its lvalue) and copies the value 2 to that address. Next line, the variable nr2 is interpreted as its rvalue (on the right hand side of the assignment operator '='). There nr2 refers to the value stored at the memory location of nr2, in this case 2. So, the value is copied to the address (the lvalue) of nr1.

Parameter passing

```
#include <stdio.h>
int main()
{
   void func(int nr);
   int val=1;
   printf("%d\n", val);    // first call for printf
   func(val);
   printf("%d\n", val);    // second call for printf
   func(val);
}
void func(int val)
{
   val++;
   printf("%d\n", val);    // third call for printf
}
```

Parameter passing

- The displayed result will be:
  1
  2
  1
  2
- Value of val is stored in a temporary location, as a copy

Parameter passing: arrays

```
#include <stdio.h>
int main()
{
   void func(int nr[], int max);
   int cnt, num[3];
   for(cnt=0;cnt<3;cnt++)
      num[cnt]=cnt;
   func(num,3);
   for(cnt=0;cnt<3;cnt++)
      printf("%d  ", num[cnt]);
}
void func(int num[], int max)
{
   int i;
   for(i=0;i<max;i++)
      num[i]=num[i]+1;
}
```

Parameter passing: arrays

- The result displayed is:
  1 2 3

- When array name is used, the address of the first element is passed
- The value of an array name is an address
- Copy of the address is passed to the function

Pointers - Introduction

- A pointer is a variable that can have as value a storage address
- Suppose we have an integer val. The unary operator & gives the address of its operand.
- The value after applying & operator (i.e. address) can be assigned to a pointer variable:

```
ptr = &val;
```

- By this we say “ptr points to val”
- the & operator retrieves the lvalue (address) of val, even though val is on the right hand side of the assignment operator, and copies that to the contents of ptr

Pointers - Declaration

- Pointers, being variables, must be declared
- In C, a pointer variable can point to values of one type only (except special case, see later)
- Declaration of a pointer variable is done by preceding its name with an asterisk. The declaration of pointer variable must specify the type:

```
int *ptr;
```

Pointers - Declaration

- ptr is the name of the variable .
- * tells the compiler that ptr is a pointer variable, i.e. reserve however many bytes is required to store an address in memory.
- int says that the pointer variable stores the address of an integer.
- Similar to “regular”variables, ptr has no value after declaration. If the declaration is outside of any function, it is initialized to a value guaranteed in such a way that it doesn’t point to any C object or function. A pointer initialized in this manner is called a "null" pointer.

Pointers - Declaration

```
#include <stdio.h>
int nr1, nr2;
int *ptr;
int main()
{
    nr1 = 1;
    nr2 = 2;
    ptr = &nr1;
    printf("\n");
    printf("nr1 value %d and is at %p\n", nr1, &nr1);
    printf("nr2 value %d and is at %p\n", nr2, &nr2);
    printf("ptr value %p and is at %p\n", ptr, &ptr);
    printf("The value of the integer pointed to by ptr
is %d\n", *ptr);
    return 0;
}
```

Pointers - Declaration

```
#include <stdio.h>

int main() {
    int number = 42;
    int *pointerToNumber; // Declare a pointer to an integer

    pointerToNumber = &number; // Assign the address of 'number' to the
pointer

    printf("Value of number: %d\n", number);
    printf("Address of number: %p\n", &number);
    printf("Value of number via pointer: %d\n", *pointerToNumber);
    printf("Address stored in the pointer: %p\n", pointerToNumber);

    // Modify the value through the pointer
    *pointerToNumber = 100;

    printf("Value of number: %d\n", number);

    return 0;
}
```

Pointers - type

- When we declare a variable, the compiler is given
- the name of the variable
- the type of the variable.
- For example, we declare a variable of type integer with the name val:

```
int val;
```

- For the "int" part of this statement the compiler reserves (depending on the platform) 4 bytes of memory to hold the value of the integer.
- It also sets up a symbol table. In that table it adds the symbol val and the relative address in memory where those 4 bytes were reserved.

Pointers - Operators

- `int nr1;`
- `int *ptr;`
- `ptr = &nr1;`
- the & operator retrieves the address of nr1
- the dereferencing operator is * and it is used as follows:

```
*ptr = 2;
```

- puts 2 to the address pointed to by ptr. When we use * we refer to the value pointed by ptr, not the value of the pointer itself.

Pointers - Operators

```
#include <stdio.h>
int main() {
    int x = 10; // Declare an integer variable
    int *ptr;   // Declare a pointer to an integer

    ptr = &x;   // Assign the address of x to the pointer

    // Print the value of x and the value pointed to by ptr
    printf("Value of x: %d\n", x);
    printf("Value pointed to by ptr: %d\n", *ptr);

    // Modify the value of x through the pointer
    *ptr = 20;

    // Print the updated value of x
    printf("Value of x: %d\n", x);

    return 0;
}
```

Pointers and Arrays

```
#include <stdio.h>
#include <stdlib.h>
int main() {
  int vec[3];
  vec[0] = 1;
  vec[1] = 2;
  vec[2] = 3;
  printf("vec[2]=%d\n", vec[2]); //  same as
          // printf("vec[2]=%d\n", *(vec+2));
  return 0;
}
```

Pointers and Arrays

- When declaring an array a block of memory is allocated large enough to hold the intended number of that type (eg: integers) the variable is a pointer that points to the first element in the array. Indexing with the expression printf("vec[2]=%d\n", vec[2]);
- C is using pointer arithmetic to step into the array the appropriate number of times, and reading the value in the memory location it ends up in.
- When accessing 3rd element of the array using vec[2] then C is looking at the address pointed to by vec (the first element of the array), and steps 2 integers forward reading the value it finds there.

Pointers and Arrays

- As we said, first element in array is the same with the array name:

  `ptr = &vec[0];` is equivalent to

  `ptr = vec;`

- But, vec is not a pointer, is the address of the first element of vec !!!

  `vec = ptr;`

- is illegal, vec is a constant (unmodifiable lvalue)

Pointers

```
#include <stdio.h>

int main() {
    int numbers[] = {1, 2, 3, 4, 5};
    int *ptr = numbers;

    for (int i = 0; i < 5; i++) {
        printf("Value at index %d: %d\n", i, *ptr);
        ptr++;
    }

    return 0;
}
```

Pointers

```
#include <stdio.h>

void increment(int *ptr) {
    (*ptr)++;
}

int main() {
    int x = 5;
    printf("Before: x = %d\n", x);
    increment(&x); // Pass the address of x to the function
    printf("After: x = %d\n", x);
    return 0;
}
```

Pointers

```
#include <stdio.h>

void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main() {
    int x = 5, y = 10;
    printf("Before swapping: x = %d, y = %d\n", x, y);
    swap(&x, &y);
    printf("After swapping: x = %d, y = %d\n", x, y);
    return 0;
}
```

Pointers - Operations

A pointer, constructed either as a variable or function parameter, contains a value: an address

Pointers Arithmetic

- In C there are some arithmetic operations allowed when using pointers:

Consider ptr a pointer to a type t and i an integer. The expressions ptr + i and ptr - i have as result a pointer with value greater or less by i*sizeof(type) then ptr. For example:

```
int *ptr;
```

`ptr + 3` means address of p + 12 bytes

- Consider the case:

```
int vec[3];
int *ptr;
ptr = vec;
```

- We know that vec[i] is the same as *(ptr+i). Wherever one writes a[i] it can be replaced with *(a + i). This is NOT saying that pointers and arrays are the same thing, they are not. We only say that to identify a given element of an array we have the choice of two syntaxes, one using array indexing and the other using pointer arithmetic, with identical results.

Pointers Arithmetic

- Looking at(a + i), the rules of C state that addition is commutative. That is (a + i) is identical to (i + a). Thus we could write *(i + a) just as easily as *(a + i).
- But *(i + a) could have come from i[a] ! What about:

```
#include <stdio.h>
int main(void)
{
    char a[10]="A string";
    a[3]='x';
    2[a]='y';
    puts(a);
    return 0;
}
```

Pointers Arithmetic

```
#include <stdio.h>

int main() {
    int numbers[] = {1, 2, 3, 4, 5};
    int *ptr = numbers; // Pointer to the first element of the array

    // Increment the pointer to move to the next element
    ptr++;
    printf("Second element: %d\n", *ptr);

    // Decrement the pointer to move back to the first element
    ptr--;
    printf("First element: %d\n", *ptr);

    return 0;
}
```

Pointers Arithmetic

```
#include <stdio.h>

void incrementByValue(int value) {
    value++;
}

void incrementByReference(int *value) {
    (*value)++;
}

int main() {
    int x = 5;

    incrementByValue(x);
    printf("After incrementByValue: x = %d\n", x);

    incrementByReference(&x);
    printf("After incrementByReference: x = %d\n", x);

    return 0;
}
```

Pointers Arithmetic

```
#include <stdio.h>

int main() {
    char *message = "Hello, World!";
    char *ptr = message;

    // Access characters in a string
    while (*ptr != '\0') {
        printf("%c\n", *ptr);
        ptr++;
    }

    return 0;
}
```