Third round tasks as a `ZIP file`.

This section focuses on one of the most important concepts of the C language: **Pointers**. Pointers allow processing of **arrays**, **strings** and various other C-programming variables. After reading through this section, you will understand pointers and operations allowed on pointers, how to deal with array data using pointers, as well as how to do simple programs in which pointers are used. In future sections, pointers are used in practice constantly, so in this course you will be practicing how to use them in abundance.

Pointers¶

Pointers are a fundamental part of C programming, and typically necessary in any non-trivial software functionality. The ability to directly operate with pointers and pointers' memory is one of the differentiating factors between C and many other (higher-level) programming languages. Unfortunately, the ability to directly operate with (virtual) memory also introduces various ways for errors that are sometimes hard to trace.

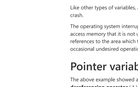
Background¶

Variables declared in the program code allocate memory space that depends on the type of variable. For example, a *char* variable uses one byte (8 bits) of memory, and *int (usually)* uses four bytes (32 bits) of memory. For local variables, the system allocates the needed memory space automatically when the variable is declared, and releases the memory when execution exits the program block or function where the variable was declared. After exiting a function or program block (the section framed with curly brackets), the compiler does not allow using the variables introduced inside the block, but returns an error to inform about this attempt.

Consider a simple example as in the following:

```
1  void main(void)
2  {
3      char a = 10;
4      char b = 12;
5      int c = 123456;
6      char *d = &a;
7  }
```

The above function does not do anything else, but declares a few variables. After the four variable declarations, the content of memory is like follows:



Variables **a** and **b** use one byte of memory each, and variable **c** takes 4 bytes. As the variables are declared, their initial values are set. If this was not done, the memory would be allocated in the similar way, but its content would be unspecified. Variable **d** is a **pointer** to a char-type value in memory, and it is initialized to **point to the location** of variable **a** using an address operator (&). Even though it points to a char-type value, variable d takes four bytes from the memory, because it is a pointer: its value is a memory address (in this example we assume that addresses are 32 bits long).

The picture also shows imaginary memory addresses allocated for the variables. Normally the programmer does not need to know about numeric addresses, but they are shown here to clarify the functionality of pointers. Despite this example, one should not generally make specific assumptions about how variables are placed in the memory, because compilers sometimes take liberties in memory usage.

When function exits, the memory allocated for local variables is released. This is not a problem for the basic data types, because compiler raises an error if the program tries to use the variable from outside the scope it was declared. However, when referring to the variable via pointers, the compiler cannot protect the programmer from referring to invalid memory location. Therefore careless use of pointers may cause problems, for example when referring to memory that has already been released for other use.

Use of obsolete variables¶

As mentioned before, variables declared/defined inside a block can be used only inside that block. In practice, this means that the memory area allocated for that variable can be used for other purposes outside the block. If a pointer refers to the freed area for the rest of the program without changing/using the contents will not cause any issues. However, careless use of such an pointer would have damaging consequences, because the content of the referenced memory area will be modified some other part of the program also. This is common mistake done by beginners. It is very difficult to debug this kind of wrong memory usage errors, because the program might run without errors but with very weird-outputs that seem to change randomly.

Like other types of variables, an uninitialized pointer has unknown value. Use of such pointer will likely cause the program to crash.

The operating system intercepts the program with "segmentation fault" - signal when it detects that the program tries to access memory that it is not used in the first place. In many cases, however, error is caused due to an incorrect memory references to the area which the operating system has already been allocated for the program. In this case, the result is only an occasional undesired operation. To detect this situation, there are some tools which are widely used.

Pointer variable basics¶

The above example showed an sample of how a pointer variable is declared and initialized in the case char data type, using the **dereferencing operator** (`*`) after the data type. Similar format can be used with other data types. There could be, for example, int *e or float *f. Despite different data types, all these variables allocate similar space in memory, because their content is actually a memory address that points to location that contains a value of given data type. You might be confused now; the dereferencing operator is the same as the multiplication operator in C-language! However, the meaning of the operation can be seen from the context.

Some discussion on the internet: "Why pointer sign and multiplication are the same in C?"

C allows different uses of typical data by declaring pointers: `char* d;`, `char *d;`, and `char *d;` are all valid and equivalent notations. Usually it is good coding style to consistently use one of the above throughout the program.

Let's extend the previous example by a few lines, and take one more pointer, e, into use:

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      char a = 10;
6      char b = 12;
7      int c = 123456;
8      char *d = &a;
9      char *e;
10
11     e = d;
12     printf("*e: %d  e: %p\n", *e, e);
13
14     *d = 13;
15     printf("*e: %d  d: %p  *e: %d  a: %d  c: %d\n", *d, d, *e, a, c);
16     if (*d > b)
17         printf("New value is greater than b!\n");
18 }
```

Pointer variables can be assigned as any other variables, and such assignment (as on line 11 for variable *e*) will copy the address. After the assignment, both *e* and *d* point to the same location (address of variable *a*).

The value referenced by a pointer can be accessed using the `*` (dereferencing) operator, as shown for example on line 12 with the printf call. The dereferencing operator can be used in any expression (part of function call, assignments, comparisons, ...). Because pointer *e* references the variable *a*, the printf call outputs the value 10 for the first field. The latter printf field (`%p`) is an example of printing out the value of pointer in hexadecimal format (not very often needed). Note the two different *printf* parameters and their difference: one has an asterisk and one doesn't. They are two different values.

The dereferencing operator can also be used to modify the memory the pointer is referencing. It is also done with the assignment operator, as long as the dereferencing operator is used correctly. The operator can thus be used as a part of expressions in different situations, just like other operators: as a part of a longer computation, function parameter etc.

On line 15 we print out the values of multiple variables, so that we can see what the preceding lines have done. You should try out the program and modify it in different ways.

At the end of the *main* function, the content of variables looks like this:

With the pointer variable we have also modified the contents of variable *a*, which we can now access with both the variables *d* and *e*.

Let's continue the the program, and add **erroneous** lines:

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      char a = 10;
6      char b = 12;
7      char *d = &a;
8
9      *d = 13;
10     printf("*d: %d  d: %p  a: %d\n", *d, d, a);
11     if (*d > b)
12         printf("New value is greater than b!\n");
13     d = 14;
14     printf("d: %d\n", d);
15     *d = 15;
16     printf("bye bye! d is now: %d\n", *d);
17
18     return 0;
19 }
```

Compiling the program causes warnings. The warning on line 13 means that the assignment operator is trying to set a pointer to equal an integer, which is most likely wrong. Warning on line 14 says that we are trying to use the `%d` format specifier on a pointer, which is also most likely incorrect. Nevertheless, the compiler will compile the program and produce a runnable program. **Warnings should thus always be checked and fixed**. This saves a lot of time in debugging.

The pointer variable is set to reference the memory address 14, which in most systems isn't directly available to programs. This in itself does not crash the program but when that memory location is referenced on line 15, the OS gives a segmentation fault signal.

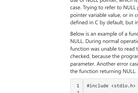
The operating system stops the program execution at that spot due to erroneous memory access but cannot tell what line it happened on; the OS does not see the lines of the program as it was compiled into machine language. The end result is that the "bye bye!" message is not printed.

Even though in the previous example unary `*` and `&` operators are used in the context of variables, they can be used in expressions in other contexts as well. For example, `*(a + 2)` accesses the memory location a little bit after the location of *d*. This has many uses, as we shall soon see.

- *unary operator*: an operator that only operates on a single expression i.e. only has one operand (after the operator). E.g. `*(arr)`.
- *binary operator*: an operator that operates on two expressions i.e. has two operands. E.g. `(arr) * (ins)`

It is also important to notice that the `*` character has two different meanings in the context of pointers: when a variable is defined (line 7) and when it is dereferenced (e.g. lines 9 and 10).

The following screenshot from StackOverflow summarizes the interpretation of the unary `*` and `&` operators:



Task 3.1: SegFault¶

Fix the above program, so that there are no warnings during compilation, the program operates correctly until the end, and set the memory location indicated by *d* to value 14 at first, and then to 15, as this what the above program was really attempting to do. Do not touch the print formatting, but you may need to change the expressions used in the *printf* parameters, for example by adding operators.

Points 1 / 1 My submissions 2 Deadline Friday, 18 June 2021, 19:59
To be submitted alone

This course has been archived (Saturday, 31 December 2022, 20:00).

Seg Fault

Select your files for grading

main.c

Choose File No file chosen

Submit

Video on pointers:

Pointers in functions¶

Pointers can be used much like other variables. They can also be used as a parameters in function call, and as return values from a function. The **scanf** function uses pointers as parameters, because the function needs to copy the value typed by user to the context where scanf was called, and pointer is practically the only way to do it. If a basic (non-pointer) data type is used directly as a function parameter, the function could not modify its value in the caller's context, because the parameters are passed as value, and copied for the function's local use.

Below is a short example, "**my_readint**" function, that reads a character from user, converts it to integer, and copies its value to the location indicated by the pointer. The function returns 1, if it changed the pointed value, or 0, if it did not get a valid number, and therefore did not change the content of memory.

```
1  #include <stdio.h>
2
3  int my_readint(int *value)
4  {
5      char c;
6      int ret;
7      ret = scanf("%c", &c);  // read one ASCII-character
8      if (ret == 1 && c >= '0' && c <= '9') {
9          int num = c - '0';  // convert it to the corresponding number
10         *value = num; // assign it to the location indicated by the pointer
11         return 1;
12     }
13     return 0;
14 }
15
16 int main(void)
17 {
18     int a;
19     int *ptr_a = &a;
20     if (my_readint(ptr_a))
21         printf("reading succeeded: %d\n", a);
22     else
23         printf("not a valid number\n");
24
25     // another way to do the same thing
26     my_readint(&a);
27 }
```

There are two calls to *my_readint* from the main function (lines 20 and 26). In both calls to *my_readint*, the parameter is set to point to the location of variable '**a**', but in two alternative ways (the latter just shows that address operator can be used as part of other expressions, such as function calls).

Inside the function, argument '**value**' is a pointer to an integer, that was set by the caller of the function. The function reads a character from user, makes an assumption that user types a digit, and converts the ASCII digit into a integer number by subtracting ASCII code of '0' from the value entered by user (line 9). The resulting integer value is assigned to the memory location pointed by 'value' (line 10), which in this case points to variable 'a' in the main function. With pointers, a function can modify the variables and other memory outside the function, which would not be possible otherwise.

Using pointers as parameters is useful, for example to be able to return values from inside the function by also other means than just return parameter. This is needed, if a function needs to return more than one values to the caller.

Pointer can also be used as the return value type of a function, as seen in the below example. The example also illustrates the use of NULL pointer, which is a special pointer value that is used to indicate an error, unused pointer, or some other special case. Trying to refer to NULL pointer with dereference operator will always cause an error, but it is ok to assign NULL as a pointer variable value, or in conditions compare a pointer value with NULL for some special program action. NULL is not defined in C by default, but included in **stddef.h** header, which you'll need to include in order to use it.

Below is an example of a function that reads an integer from the user to a given address, but only if the given pointer is not NULL. During normal operation, the function returns the pointer it got as a parameter. A NULL pointer is returned, if the function was unable to read the integer into the pointer. One error case is that the given pointer was NULL. This needs to be checked, because the program would crash due to an invalid memory access. If a NULL pointer was passed as a scanf parameter. Another error case is that scanf failed to read the number, for example due to invalid user input. Both cases result in the function returning NULL.

```
1  #include <stdio.h>
2
3  int *read_int(int *number)
4  {
5      int ret;
6      if (number == NULL) { // check that the given pointer is valid
7          return NULL; // exit the function, if the pointer is invalid
8      }
9      ret = scanf("%d", number);
10     if (ret != 1) { // check if scanf read correctly 1 integer
11         return NULL;
12     }
13     return number;
14 }
15
16 int main(void)
17 {
18     int a;
19     int *ptr_a = &a;
20     int ret = -1000;
21     if (read_int(ptr_a))
22         printf("reading succeeded: %d\n", a);
23     else
24         printf("not a valid number\n");
25
26     // a second way to do the same
27     read_int(&a);
28     printf("r: %d\n", ret);
29 }
```

Task 3.2: Read Int¶

Change the above read_int - function so that it returns directly the integer (int) that was read and not the pointer (*int). If the number cannot be read, the function returns -1. In addition to this, modify the code according to the following instructions:

- Lines 21-24: if the read_int return value is not equal to -1, print the "reading succeeded"- branch. Otherwise print "not a valid number"
- Line 27: read_int return value is stored in integer *ret*, so that it prints correctly on the next line.

Do not change the printf formatting.

Points 1 / 1 My submissions 1 Deadline Friday, 18 June 2021, 19:59
To be submitted alone

This course has been archived (Saturday, 31 December 2022, 20:00).

Read Int

Select your files for grading

main.c

Choose File No file chosen

Submit