Page 1

Allocating Dynamic Memory¶

To allocate dynamic memory on the heap means to make a contiguous memory area accessible to the program at runtime and to mark this memory as occupied so that no one else can write there by mistake.
To reserve memory on the heap, one of the two functions `malloc` (stands for *Memory Allocation*) or `calloc` (stands for *Cleared Memory Allocation*) is used. The header file `stdlib.h` or `malloc.h` must be included to use the functions.
Here is the syntax of `malloc` and `calloc` in C/C++:

```
pointer_name = (cast-type*) malloc(size);
pointer_name = (cast-type*) calloc(num_elems, size_elem);
```

`malloc` is used to dynamically allocate a single large block of memory with the specified size. It returns a pointer of type `void` which can be cast into a pointer of any form.
`calloc` is used to dynamically allocate the specified number of blocks of memory of the specified type. It initializes each block with a default value '0'.
Both functions return a pointer of type `void` which can be cast into a pointer of any form. If the space for the allocation is insufficient, a NULL pointer is returned.

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
   void *p = malloc(sizeof(int));
   printf("address=%p, value=%d\n", p, *p);

   return 0;
}
```

Page 2

In the code example on the right, a block of memory the size of an integer is allocated using `malloc`.
The `sizeof` command is a convenient way of specifying the amount of memory (in bytes) needed to store a certain data type. For an `int`, `sizeof` returns 4. However, when compiling this code, the following warning is generated on my machine:

```
warning: ISO C++ does not allow indirection on operand of type 'void *' [-Wvoid-ptr-dereference]
```

```
printf("address=%p, value=%d", p, *p);
```

In the virtual workspace, when compiling with `g++`, an error is thrown instead of a warning.
The problem with `void` pointers is that there is no way of knowing the offset to the end of the allocated memory block. For an int, this would be 4 bytes but for a double, the offset would be 8 bytes. So in order to retrieve the entire block of memory that has been reserved, we need to know the data type and the way to achieve this with `malloc` is by casting the return pointer:
`int *p = (int*)malloc(sizeof(int));`
This code now produces the following output without compiler warnings: `address=0x1003001f0, value=0`
Obviously, the memory has been initialized with 0 in this case. However, you should not rely on pre-initialization as this depends on the data type as well as on the compiler you are using.
At compile time, only the space for the pointer is reserved (on the stack). When the pointer is initialized, a block of memory of `sizeof(int)` bytes is allocated (on the heap) at program runtime. The pointer on the

Page 3

Quiz¶

Modify the example in a way that memory for 3 integers is reserved.

HIDE SOLUTION

```
    // reserve memory for several integers
    int *p2 = (int*)malloc(3*sizeof(int));
    printf("address=%p, value=%d\n", p2, *p2);
```

Page 4

Memory for Arrays and Structs¶

Since arrays and pointers are displayed and processed identically internally, individual blocks of data can also be accessed using array syntax:

```
    int *p = (int*)malloc(3*sizeof(int));
    p[0] = 1; p[1] = 2; p[2] = 3;
    printf("address=%p, second value=%d\n", p, p[1]);
```

Until now, we have only allocated memory for a C/C++ data primitive (i.e. `int`). However, we can also define a proprietary structure which consists of several primitive data types and use `malloc` or `calloc` in the same manner as before:

```
struct MyStruct {
    int i;
    double d;
    char a[5];
};

MyStruct *p = (MyStruct*)calloc(4,sizeof(MyStruct));
p[0].i = 1; p[0].d = 3.14159; p[0].a[0] = 'a';
```

After defining the struct `MyStruct` which contains a number of data primitives, a block of memory four times the size of `MyStruct` is created using the `calloc` command. As can be seen, the various data elements can be accessed very conveniently.

Page 5 has video:

Page 6

The size of the memory area reserved with `malloc` or `calloc` can be increased or decreased with the `realloc` function.
`pointer_name = (cast-type*) realloc( (cast-type*)old_memblock, new_size );`
To do this, the function must be given a pointer to the previous memory area and the new size in bytes. Depending on the compiler, the reserved memory area is either (a) expanded or reduced internally (if there is still enough free heap after the previously reserved memory area) or (b) a new memory area is reserved in the desired size and the old memory area is released afterwards.
The data from the old memory area is retained, i.e. if the new memory area is larger, the data will be available within new memory area as well. If the new memory area is smaller, the data from the old area will be available only up until the site of the new area - the rest is lost.
In the example on the right, a block of memory of initially 8 bytes (two integers) is resized to 16 bytes (four integers) using `realloc`. Note that realloc has been used to increase the memory size and then decrease it immediately after assigning the values 3 and 4 to the new blocks. The output looks like the following:

```
address=0x100300060, value=1
address=0x100300064, value=2
address=0x100300068, value=3
address=0x10030006c, value=4
```

Interestingly, the pointers `p+2` and `p+3` can still access the memory location they point to. Also, the original data (numbers 3 and 4) is still there. So `realloc` will not erase memory but merely mark it as "available" for future allocations. It should be noted however that accessing a memory location *after* such an operation must be avoided

#include <stdio.h>
#include <stdlib.h>

```
int main()
{
    // reserve memory for two integers
    int *p = (int*)malloc(2*sizeof(int));
    p[0] = 1; p[1] = 2;

    // resize memory to hold four integers
    p = (int*)realloc(p,4*sizeof(int));
    p[2] = 3; p[3] = 4;

    // resize memory again to hold two integers
    p = (int*)realloc(p,2*sizeof(int));

    printf("address=%p, value=%d\n", p+0, *(p+0)); // valid
    printf("address=%p, value=%d\n", p+1, *(p+1)); // valid

    printf("address=%p, value=%d\n", p+2, *(p+2)); // INVALID
    printf("address=%p, value=%d\n", p+3, *(p+3)); // INVALID

    return 0;
}
```

Page 7

Freeing up Memory¶

If memory has been reserved, it should also be released as soon as it is no longer needed. If memory is reserved regularly without releasing it again, the memory capacity may be exhausted at some point. If the RAM memory is completely used up, the data is swapped out to the hard disk, which slows down the computer significantly.

The `free` function releases the reserved memory area so that it can be used again or made available to other programs. To do this, the pointer pointing to the memory area to be freed is specified as a parameter for the function. In the `free_example.cpp`, a memory area is reserved and immediately released again.

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    void *p = malloc(100);
    free(p);

    return 0;
}
```

Page 8

Some things should be considered with dynamic memory management, whose neglect in some cases might result in unpredictable program behavior or a system crash - in some cases unfortunately without error messages from the compiler or the operating system:

1. `free` can only free memory that was reserved with `malloc` or `calloc`.
2. `free` can only release memory that has not been released before. Releasing the same block of memory twice will result in an error.

In the example on the right, a pointer `p` is copied into a new variable `p2`, which is then passed to `free` AFTER the original pointer has been already released.

```
free(41143,0x1000a55c0) malloc: *** error for object 0x1003001f0: pointer being freed was not allocated.
```

In the workspace, you will see this error:

```
*** Error in './a.out': double free or corruption (fasttop): 0x0000000000755010 ***
```

The pointer `p2` in the example is invalid as soon as `free(p)` is called. It still holds the address to the memory location which has

been freed, but may not access it anymore. Such a pointer is called a "dangling pointer".

3. Memory allocated with `malloc` or `calloc` is not subject to the familiar rules of variables in their respective scopes. This means that they exist independently of block limits until they are released again or the program is terminated. However, the pointers which refer to such heap-allocated memory are created on the stack and thus only exist within a limited scope. As soon as the scope is left, the pointer variable will be lost - but not the heap memory it refers to.