Page 1:

Comparing `malloc` with `new`¶

The functions `malloc` and `free` are library function and represent the default way of allocating and deallocating memory in C. In C++, they are also part of the standard and can be used to allocate blocks of memory on the heap.
With the introduction of classes and object oriented programming in C++ however, memory allocation and deallocation has become more complex: When an object is created, its constructor needs to be called to allow for member initialization. Also, on object deletion, the destructor is called to free resources and to allow for programmer-defined clean-up tasks. For this reason, C++ introduces the operators `new` / `delete`, which represent the object-oriented counterpart to memory management with `malloc` / `free`.

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

class MyClass
{
private:
    int *_number;

public:
    MyClass()
    {
        std::cout << "Allocate memory\n";
        _number = (int *)malloc(sizeof(int));
    }
    ~MyClass()
    {
        std::cout << "Delete memory\n";
        free(_number);
    }
```

```
    void setNumber(int number)
    {
        *_number = number;
        std::cout << "Number: " << _number << "\n";
    }
};


int main()
{
    // allocate memory using malloc
    // comment these lines out to run the example below
    MyClass *myClass = (MyClass *)malloc(sizeof(MyClass));
    myClass->setNumber(42); // EXC_BAD_ACCESS
    free(myClass);

        // allocate memory using new
    MyClass *myClass = new MyClass();
    myClass->setNumber(42); // works as expected
    delete myClass;

    return 0;
}
```

PAge 2:

f we were to create a C++ object with `malloc`, the constructor and destructor of such an object would not be called. Consider the class on the right. The constructor allocates memory for the private element `_number` (yes, we could have simply used `int` instead of `int*`, but that's for educational purposes only), and the destructor releases memory again. The setter method `setNumber` finally assigns a value to `_number` under the assumption that memory has been allocated previously.
In main, we will allocate memory for an instance of MyClass using both `malloc`/ `free` and `new`/`delete`.

With `malloc`, the program crashes on calling the method `setNumber`, as no memory has been allocated for `_number` - because the constructor has not been called. Hence, an `EXC_BAD_ACCESS` error occurs, when trying to access the memory location to which `_number` is pointing. With `_new`, the output looks like the following:

```
Allocate memory
Number: 42
Delete memory
```

`Page 3:`

Before we go into further details of `new`/`delete`, let us briefly summarize the major differences between `malloc`/`free` and `new`/`delete`:

1. **Constructors / Destructors** Unlike `malloc( sizeof(MyClass) )`, the call `new MyClass()` calls the constructor. Similarly, `delete` calls the destructor.
2. **Type safety** `malloc` returns a void pointer, which needs to be cast into the appropriate data type it points to. This is not type safe, as you can freely vary the pointer type without any warnings or errors from the compiler as in the following small example: `MyObject *p = (MyObject*)malloc(sizeof(int));`

In C++, the call `MyObject *p = new MyObject()` returns the correct type automatically - it is thus type-safe.

3. **Operator Overloading** As `malloc` and `free` are functions defined in a library, their behavior can not be changed easily. The new and delete operators however can be overloaded by a class in order to include optional proprietary behavior. We will look at an example of overloading `new` further down in this section.

Page 4:

Creating and Deleting Objects¶

As with `malloc` and `free`, a call to `new` always has to be followed by a call to `delete` to ensure that memory is properly deallocated. If the programmer forgets to call `delete` on the object (which happens quite often, even with experienced programmers), the object resides in memory until the program terminates at some point in the future causing a *memory leak*.
Let us revisit a part of the code example to the right:

```
myClass = new MyClass();
myClass->setNumber(42); // works as expected
delete myClass;
```

The call to `new` has the following consequences:

1. Memory is allocated to hold a new object of type `MyClass`
2. A new object of type `MyClass` is constructed within the allocated memory by calling the constructor of `MyClass`

The call to `delete` causes the following:

1. The object of type `MyClass` is destroyed by calling its destructor
2. The memory which the object was placed in is deallocated

Page 5:

Optimizing Performance with `placement new`¶

In some cases, it makes sense to separate memory allocation from object construction. Consider a case where we need to reconstruct an object several times. If we were to use the standard `new`/ `delete` construct, memory would be allocated and freed unnecessarily as only the content of the memory block changes but not its size. By separating allocation from construction, we can get a significant performance increase.
C++ allows us to do this by using a construct called *placement new*: With `placement new`, we can pass a preallocated memory and construct an object at that memory location. Consider the following code:

```
void *memory = malloc(sizeof(MyClass));
MyClass *object = new (memory) MyClass;
```

The syntax `new (memory)` is denoted as *placement new*. The difference to the "conventional" `new` we have been using so far is that that no memory is allocated. The call constructs an object and places it in the assigned memory location. There is however, no `delete` equivalent to `placement new`, so we have to call the destructor explicitly in this case instead of using `delete` as we would have done with a regular call to `new`:

```
object->~MyClass();
free(memory);
```

Important: Note that this should never be done outside of `placement new`.
In the next section, we will look at how to overload the new operator and show the performance difference between `placement new` and `new`

`Page 6:`

Overloading `new` and `delete`¶

One of the major advantages of `new`/`delete` over `free`/`malloc` is the possibility of overloading. While both `malloc` and `free` are function calls and thus can not be changed easily, `new` and `delete` are operators and can thus be overloaded to integrate customized functionality, if needed.
The syntax for **overloading the new operator** looks as follows:

```
void* operator new(size_t size);
```

The operator receives a parameter `size` of type `size_t`, which specifies the number of bytes of memory to be allocated. The return type of the overloaded `new` is a void pointer, which references the beginning of the block of allocated memory.
The syntax for **overloading the delete operator** looks as follows:

```
void operator delete(void*);
```

The operator takes a pointer to the object which is to be deleted. As opposed to `new`, the operator `delete` does not have a return value.
Let us consider the example on the right.

#include <iostream>

```
#include <stdlib.h>

class MyClass
{
   int _mymember;

public:
   MyClass()
   {
      std::cout << "Constructor is called\n";
   }

   ~MyClass()
   {
      std::cout << "Destructor is called\n";
   }

   void *operator new(size_t size)
   {
      std::cout << "new: Allocating " << size << " bytes of
memory" << std::endl;
      void *p = malloc(size);

      return p;
   }

   void operator delete(void *p)
   {
      std::cout << "delete: Memory is freed again " << std::endl;
      free(p);
   }
};

int main()
{
   MyClass *p = new MyClass();
   delete p;
}
```

Page 7: is a video

Page 8:

In the code to the right, both the `new` and the `delete` operator are overloaded. In `new`, the size of the class object in bytes is printed to the console. Also, a block of memory of that size is allocated on the heap and the pointer to this block is returned. In `delete`, the block of memory is freed again. The console output of this example looks as follows:

```
new: Allocating 4 bytes of memory
Constructor is called
Destructor is called
delete: Memory is freed again
```

As can be seen from the order of text output, memory is instantiated in `new` before the constructor is called, while the order is reversed for the destructor and the call to `delete`.

Page 9:

Quiz :¶

What will happen to the console output of the overloaded new operator when the data type of `_mymember` is changed from int to double?

1. Nothing.
2. There will be a memory leak as not enough memory is allocated on the heap.
3. The output will show a changed size in bytes (8 instead of 4).

HIDE SOLUTION

The correct answer is 3.

Overloading new[] and delete[]¶

In addition to the `new` and `delete` operators we have seen so far, we can use the following code to create an array of objects:

```
void* operator new[](size_t size);
void operator delete[](void*);
```

Let us consider the example on the right, which has been slightly modified to allocate an array of objects instead of a single one.

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

class MyClass
{
   int _mymember;

public:
   MyClass()
   {
      std::cout << "Constructor is called\n";
   }

   ~MyClass()
   {
      std::cout << "Destructor is called\n";
   }

   void *operator new[](size_t size)
   {
      std::cout << "new: Allocating " << size << " bytes of
memory" << std::endl;
      void *p = malloc(size);

      return p;
   }
```

```
    void operator delete[](void *p)
    {
        std::cout << "delete: Memory is freed again " << std::endl;
        free(p);
    }
};

int main()
{
    MyClass *p = new MyClass[3]();
    delete[] p;
}
```

Page 11:
In main, we are now creating an array of three objects of MyClass. Also, the overloaded `new` and `delete` operators have been changed to accept arrays. Let us take a look at the console output:

```
new: Allocating 20 bytes of memory
Constructor is called
Constructor is called
Constructor is called
Destructor is called
Destructor is called
Destructor is called
delete: Memory is freed again
```

Interestingly, the memory requirement is larger than expected: With `new`, the block size was 4 bytes, which is exactly the space required for a single integer. Thus, with three integers, it should now be 12 bytes instead of 20 bytes. The reason for this is the memory allocation overhead that the compiler needs to keep track of the allocated blocks of memory - which in itself consumes memory. If we change the above call to e.g. `new MyClass[100]()`, we will see that the overhead of 8 bytes does not change

```
new: Allocating 408 bytes of memory
Constructor is called
…
Destructor is called
delete: Memory is freed again
```