- Examples

Let's see examples of how the 'default' and 'delete' keywords can be used to our benefit.

WE'LL COVER THE FOLLOWING

Constructors and destructors using `default`

```
#include <iostream>

class SomeType{
  public:

  // state the compiler generated default constructor
  SomeType() = default;

  // constructor for int
  SomeType(int value){
    std::cout << "SomeType(int) " << std::endl;
  };

  // explicit Copy Constructor
  explicit SomeType(const SomeType&) = default;

  // virtual destructor
  virtual ~SomeType() = default;

};

int main(){

  std::cout << std::endl;

  SomeType someType;
  SomeType someType2(2);
  SomeType someType3(someType2);

  std::cout << std::endl;
}
```

Explanation

- In this example, we can see how `default` can be used to get the default implementations of constructors and destructors from the compiler.

- Since we have defined a parameterized constructor in line 10, the compiler will not automatically create a default constructor.

- Hence, we have to define it ourselves using `default`, as done in line 7.

- `default` can automatically handle the copy constructor in line 15 and the destructor in line 18.

- The `explicit` keyword is used in the copy constructor to avoid implicit conversions during copying.

- We need the `virtual` destructor in case there is a derived class inheriting `SomeType`.

Restricting operations using `delete`

> The following code generates an error and the reasons why are mentioned below:

```
#include <iostream>

class NonCopyableClass{
  public:

  // state the compiler generated default constructor
  NonCopyableClass()= default;

  // disallow copying
  NonCopyableClass& operator = (const NonCopyableClass&) = delete;
  NonCopyableClass (const NonCopyableClass&) = delete;

  // disallow copying
  NonCopyableClass& operator = (NonCopyableClass&&) = default;
  NonCopyableClass (NonCopyableClass&&) = default;
};
```

```
class TypeOnStack {
  public:

    void * operator new(std::size_t)= delete;
};

class TypeOnHeap{
  public:
    ~TypeOnHeap()= delete;
};

void onlyDouble(double){}
template <typename T>
void onlyDouble(T)=delete;

int main(){

  NonCopyableClass nonCopyableClass;

  TypeOnStack typeOnStack;

  TypeOnHeap * typeOnHeap = new TypeOnHeap;

  onlyDouble(3.14);

  // force the compiler errors

  NonCopyableClass nonCopyableClass2(nonCopyableClass); // cannot copy

  TypeOnStack * typeOnHeap2 = new TypeOnStack; // cannot create on heap

  TypeOnHeap typeOnStack2; // cannot create on stack

  onlyDouble(2011); // int argument not accepted

}
```

Explanation

- Here, we are prohibiting certain operations by using `delete`.

- In lines 10 to 15, we have restricted the copy operation for `NonCopyableClass` objects and references. By assigning `delete`, we tell the compiler that the operation will not be accepted. Hence, an error will be thrown.

- With `delete`, we can also prevent objects from being created on the heap or stack.

- In the `TypeOnStack` class, we assign `delete` to the `operator new` on line 21. This means that an object of this class cannot occupy space on the heap.

This means that an object of this class cannot occupy space on the heap.

- Conversely, the `TypeOnHeap` class is not allowed to make objects on the stack. We simply define a destructor that calls `delete` in line 26.
- Lastly, there is the `onlyDouble()` function. We have explicitly told the compiler to accept only `double` arguments.
- For any other arguments, `onlyDouble()` will generate an error.
- Lines 45 to 51 show various examples of the operations that have been restricted by `delete`. None of them will work.

To test our knowledge of `delete` and `default`, try out the exercise in the next lesson.