Dynamische Datentypen

`new`, `delete`	**Objekt mit dynamischer Lebensdauer** erstellen.

Mit `new` wird ein Objekt erstellt, indem der nötige Speicherplatz reserviert wird, und dann ein gegebener Konstruktor aufgerufen wird. Bei `delete` wird zuerst ein Destruktor aufgerufen, bevor der Speicherplatz freigegeben wird.

Der Rückgabewert von `new` ist ein *Pointer* auf das neu erstellte Objekt. Wird mit `delete` ein Objekt gelöscht, so sollte man immer *alle* Pointer, die auf das Objekt zeigen, auf `nullptr` setzen.

Jedes `new` braucht ein `delete`. Sonst existieren die erstellten Objekte bis zum Ende des Programms, was je nach Laufdauer eine grosse Speicherverschwendung ist.

```
Class My_Class {
public:
    My_Class (const int i) : y (i) { std::cout << "Hello"; }
    int get_y () { return y; }
private:
    int y;
};

...
My_Class* ptr = new My_Class (3); // outputs Hello
My_Class* ptr2 = ptr; // another pointer to the new object
std::cout << (*ptr).get_y(); // Output: 3
delete ptr;
ptr = nullptr;
ptr2 = nullptr; // has to be done !separately!
...
```

`new ...[], delete[]`	**Ranges mit dynamischer Lebensdauer und Länge** erstellen.

```
int n; std::cin >> n;
int* range = new int[n];
// Read in values to the range
for (int* i = range; i < range + n; ++i) std::cin >> *i;
delete range; // ERROR: must say: delete[]
delete[] range; // This works
```

Copy-Konstruktor	**Kopier-Initialisierung**

Der Copy-Konstruktor ist der Konstruktor, dessen Argumenttyp `const My_Class&` ist.

```
struct Customer {
    std::string name;
    int duration;
    int amount_insured;
};

class Insurance {
public:
    Insurance (const Insurance& rhs)
        : length (rhs.length), ... // copy remaining data mbrs
        {
            cust = new Customer [length];
            for (int i = 0; i < length; ++i)
                cust[i] = rhs.cust[i];
        }
    ... // other public members
private:
    Customer* cust; // pointer to an array containing customers
    int length;     // length of cust
    ... // other private members
};
```

operator=	**Kopier-Zuweisung**

Eng verwandt mit `operator=` ist der Copy-Konstruktor. Der Unterschied ist, dass der Copy-Konstruktor nur bei der Initialisierung aufgerufen wird, `operator=` hingegen nur *nach* der Initialisierung. z.B.

```
my_class a (5, 6), c (4, 4); // Call a general constructor
my_class b = a;              // Call copy-constructor
c = b;                       // Call operator=
```

`operator=` kann anders als der Copy-Konstruktor implementiert werden müssen. Ein Beispiel sind Klassen, welche Pointer auf dynamisch generierte Objekte als Member haben. Dann muss bei `operator=` meistens zuerst das aktuell vorhandene Objekt gelöscht werden, bevor die Kopie erstellt werden kann. Dies ist beispielsweise beim Stack aus der Vorlesung relevant.

`operator=` gibt im Normalfall eine Referenz auf seinen linken Operanden zurück.

Faustregel: Meistens führt `operator=` zuerst die Aufgaben des Destructors, und dann die Aufgaben des Copy-Konstruktors aus.

```
// for Customer-struct see example on Copy-Constructor

class Insurance {
public:
    Insurance& operator= (const Insurance& rhs) {
        // Cleanup of current customers
        delete[] cust;

        // Copy over the customers from rhs
        cust = new Customer [length];
        for (int i = 0; i < length; ++i)
            cust[i] = rhs.cust[i];
        length = rhs.length;
        ... // copy other data members
        return *this; // return a reference to left operand
    }
    ... // other members
};
```

Destruktor	**Class abbauen**

```
// for Customer-struct see example on Copy-Constructor

class Insurance {
public:
    ~Insurance () { delete[] cust; } // free dynamic space
    ...
};
```