Informatik Übungsstunde - Woche 13

Eric Ceglie

Der Befehl `delete`

Wir wissen bereits, wie wir mit dem Befehl `new` Speicher für neue Elemente reservieren können. Mit dem Befehl `delete` können wir diesen Speicher wieder freigeben. Falls dies gemacht wird, sollten danach immer *alle* Pointer, die ursprünglich auf das nun gelöschte Objekt gezeigt haben manuell auf `nullptr` gesetzt werden.
Faustregel: In einem C++ Programm braucht jedes `new` ein `delete`. Sonst existieren die erstellten Objekte bis zum Ende des Programms, was je nach Laufdauer eine grosse Speicherverschwendung wäre.

Beispiel:

```
Class My_Class{
private:
        int y;

public:
        My_Class(const int i) : y(i){
                std::cout << "Hello\n";
        }

        int get_y(){
                return y;
        }
};

int main(){
        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; // delete the object
        ptr = nullptr;
        ptr2 = nullptr; // has to be done !separately!

        return 0;
}
```

Erstellen wir mit `new ...[]` einen Bereich von Speicherstellen, so können wir analog mit dem Befehl `delete[]` den ganzen Bereich wieder freigeben.

Beispiel:

```
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
```

Beachte, dass in diesem Fall `delete range;` zu einem Fehler führen würde, da wir nicht `new` sondern `new ...[]` benutzt haben!

Destruktor

Der Destruktor eines Objekts wird automatisch aufgerufen, falls der Gültigkeitsbereich eines Objekts endet (z.B. wenn `delete` auf ein Objekt angewendet wird). Der Destruktor einer Klasse wird mit `~Typ()` im `public` Bereich deklariert.

Beispiel:

```
class My_Class{
  public:
  My_Class(){
    std::cout << "Constructor called\n";
  }

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

int main() {
  {
    My_Class p; // Constructor called
  } // Destructor called when p goes out of scope

  std::cout << "-----[1]-----\n";

  {
    // No object is created since pp is pointer-typed.
    // Hence, no con- and destructor calls.
    My_Class* pp;
  }

  std::cout << "-----[2]-----\n";
```

```
  // Causes five con- and (at the end of main) destructor calls
  std::vector<My_Class> vec(5);

  std::cout << "-----[3]-----\n";


  // No object creation, no con- and destructor calls
  std::vector<My_Class*> pvec(5);

  std::cout << "-----[4]-----\n";

  return 0;
}
```

Output:

```
Constructor called
Destructor called
-----[1]-----
-----[2]-----
Constructor called
Constructor called
Constructor called
Constructor called
Constructor called
-----[3]-----
-----[4]-----
Destructor called
Destructor called
Destructor called
Destructor called
Destructor called
```

Copy-Konstruktor und `operator=`

Der Copy-Konstruktor ist der Konstruktor, dessen Argumenttyp `const My_Class&` ist und wird gebraucht, wenn eine *neue* Kopie eines Objekts erstellt werden möchte.
Eng verwandt mit dem Copy-Konstruktor ist das Überladen des Zuweisungsoperators `operator=` in einer Klasse. Der Unterschied ist, dass der Copy-Konstruktor nur bei der Initialisierung aufgerufen wird, `operator=` hingegen nur *nach* der Initialisierung.
Beachte, dass Copy-Konstruktor und `operator=` im Allgemeinen andere Implementationen haben.

Beispiel:

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

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

Beispiel:

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

                length = rhs.length;

                // Copy over the customers from rhs
                cust = new Customer[length];
                for (int i = 0; i < length; ++i){
                        cust[i] = rhs.cust[i];
                }

                ... // copy other data members
                return *this; // return a reference to left operand
        }
...
// other members
};
```

Dreierregel

Die *Dreierregel* (englisch *Rule of Three*) bezeichnet in C++ eine Empfehlung, die besagt, dass in einer Klasse, die eine der folgenden drei Memberfunktionen definiert, auch die jeweils anderen beiden definiert werden sollten:

- Destruktor
- Copy-Konstruktor
- Zuweisungsoperator (`operator=`)

Begründung: Die drei oben aufgelisteten Memberfunktionen werden normalerweise vom Compiler automatisch generiert. Diese automatische generierten Funktionen haben dabei eine in C++ spezifisch fesgelegte Bedeutung (es werden alle nicht-statischen Datenelemente in der Reihenfolge ihrer Deklaration kopiert bzw. in umgekehrter Reihenfolge freigegeben).
Falls eine Klasse jedoch eine andere Semantik hat, die *nicht* auf diese Weise kopiert oder freigegeben werden kann, kann jede dieser Funktionen durch eine eigene Definition ersetzt werden. In den meisten

Fällen erfordern solche Klassen dann aber, dass *alle drei* dieser Funktionen eine eigene, benutzerdefinierte Implementierung benötigen.