

Zu Beachten

In diesem Summary werden bei Member-Funktionen die PRE- und POST-Conditions aus Platzgründen oftmals weggelassen. Bei eigenen Programmen müssen diese aber unbedingt mit angegeben werden.

Datentypen

<code>class</code>	Datencontainer mit Kapselung
<p>Eine Klasse besteht aus Daten und Funktionen, genannt Member, und erlaubt deren Kapselung via Zugriffskontrolle: Auf Member im privaten Teil (<code>private</code>) einer Klasse kann nur durch die Klasse selbst, d.h., deren Member-Funktionen zugegriffen werden, nicht direkt von ausserhalb.</p> <p>Zugriff von ausserhalb der Klasse muss über öffentliche (<code>public</code>) Member erfolgen. Per default sind die Member einer Klasse privat.</p> <p>Einziger Unterschied gegenüber <code>structs</code>: Member in <code>structs</code> sind per default öffentlich (<code>public</code>).</p> <p>Die Deklarationsreihenfolge von Membern ist irrelevant.</p> <p>In den meisten Fällen ist es empfehlenswert, Datenmember nur im <code>private</code>-Bereich zu haben. Beispielsweise kann dadurch verhindert werden, dass der User ihnen problematische Werte zuweist.</p>	
<pre>class Insurance { public: double get_rate (); // public member-function ... private: bool is_up_to_date; // private data member double rate; // private data member double update_rate (); // private member-function }; int main () { Insurance insur1; // get an instance Insurance insur2; // get another instance std::cout << insur1.get_rate(); // get_rate() on insur1 std::cout << insur2.get_rate(); // get_rate() on insur2 std::cout << insur1.is_up_to_date; // Error return 0; }</pre>	

Programmier-Befehle - Woche 12

Memberfunktion	Funktionalität auf Klassen
<p>Memberfunktionen stellen Funktionalität auf einer Klasse bereit. Sie ermöglichen den kontrollierten Zugang zu den privaten Daten und privaten Memberfunktionen. Die <i>Deklaration</i> einer Memberfunktion erfolgt immer in der Klassendefinition, die <i>Definition</i> der Memberfunktion ist auch extern möglich (ermöglicht vorkompilierte Libraries). Dann muss allerdings die Zugehörigkeit zur Klasse explizit erwähnt werden mittels der ::-Schreibweise.</p> <p>Der Aufruf einer Memberfunktion ist <code>obj.mem_func(arg1, arg2, ..., argN)</code>. Der Teil <code>obj.</code> kann weggelassen werden, falls aus der Class heraus auf einen Member des aufrufenden Objekts (siehe Eintrag <code>*this</code>) zugegriffen wird.</p>	
<pre>// Internal Definition vs. External Definition class Insurance { public: void set_rate_i (const double v) { rate = v; } // int. void set_rate_e (const double v); ... private: double rate; ... }; void Insurance::set_rate_e (const double v) {rate = v;} // ext. ----- // Call from Inside vs. Call from Outside class Insurance { public: double get_rate () { if (!is_up_to_date) update_rate(); // from inside return rate; } double get_cost () {return get_rate() * ...;} // from inside ... private: bool is_up_to_date; double rate; double update_rate () { rate = ...; } }; ... Insurance insurance; ... std::cout << insurance.get_rate(); // from outside</pre>	

Programmier-Befehle - Woche 12

<code>*this</code>	Zugriff auf implizites Argument
<p>Memberfunktionen einer Klasse haben ein implizites Argument, nämlich das aufrufende Objekt. Und <code>this</code> ist ein Zeiger darauf. Via <code>*this</code> kann man darauf zugreifen.</p> <p>Bei Zugriffen von innerhalb einer Klasse aus auf Daten-Member oder Member-Funktionen wird das implizite Argument automatisch verwendet. Man muss es dann also nicht unbedingt explizit angeben (siehe Eintrag Memberfunktion). Man muss <code>*this</code> aber mindestens explizit verwenden, falls z.B. eine Referenz auf das implizite Argument zurückgegeben werden soll.</p>	
<pre>// General example class Human { public: void set (const int a) { age = a; } // or (*this).age = a; void print1 () const { std::cout << (*this).age; } void print2 () const { std::cout << age; } // equivalent private: int age; }; ... Human me; me.set(99); me.print1(); // 99 me.print2(); // 99 ----- // Another example class Complex { public: // Note: In most applications // a reference should be returned. Complex& operator+= (const Complex& b) { real += b.real; imag += b.imag; return *this; } ... // other members private: float real; float imag; };</pre>	

Programmier-Befehle - Woche 12

<code>const</code> Memberfunktion	Unverändernde Memberfunktion
<p>Das <code>const</code> bezieht sich auf <code>*this</code>. Es verspricht, dass durch die Funktionsausführung das implizite Argument nicht im Wert verändert wird.</p>	
<pre>class Insurance { public: double get_value() const { return value; // same: return (*this).value; } ... private: double value; };</pre>	

<code>typedef</code> -Member	Kapseln von Typendefinitionen
<p>Klassen erlauben die Kapselung von Typendefinitionen. Dies ermöglicht beispielsweise eine (kompatible) Redefinition dieser Typen zu einem späteren Zeitpunkt, ohne dass die User der Klasse dann ihre Projekte umschreiben müssen. (Die Änderung erfolgt an einer einzigen Stelle.)</p> <p>Bei Verwendungen ausserhalb der Klasse muss mit der <code>::</code>-Schreibweise auf den <code>typedef</code>-Member zugegriffen werden.</p>	
<pre>class My_Class { public: typedef double My_Double; My_Double d; }; int main() { My_Class::My_Double d = 0; d += 1; // can be used exactly like double return 0; }</pre>	

Konstruktor	Datencontainer Initialisierung
<p>Konstruktoren sind spezielle Memberfunktionen einer Klasse, die den Namen der Klasse tragen. Sie werden bei der Variablendeklaration aufgerufen.</p> <p>Sie werden analog zu Funktionen überladen und bei der Variablendeklaration wie eine Funktion aufgerufen. Damit das funktioniert, muss der Konstruktor öffentlich (public) sein.</p> <p>Einige Konstruktoren haben spezielle Zusatzfunktionen: z.B. der Default-Konstruktor (kein Argument) wird automatisch erzeugt, falls eine Klasse keinen Konstruktor definiert, oder Konstruktoren mit genau einem Argument ermöglichen die Definition benutzerdefinierter Konversionen.</p>	
<pre>class Insurance { public: Insurance(double v, int r) // general constructor : value (v), rate (r) // initialize data members { update_rate(); } Insurance() // default constructor : value (0), rate (0) // initialize data members {} ... private: double value; double rate; void update_rate(); }; ... // General Constructor Insurance i1 (10000, 10); // default-Constructor, direct call Insurance i3; // identical: Insurance i3 (); ... ----- class Complex { public: // Conversion Constructor (float --> Complex) Complex(const float i) : real (i), imag (0) {} private: float real; float imag; };</pre>	

Dynamische Datentypen

<code>new, delete</code>	Objekt mit dynamischer Lebensdauer erstellen und löschen.
<p>Mit <code>new</code> wird ein Objekt erstellt, indem der nötige Speicherplatz reserviert wird, und dann ein gegebener <i>Konstruktor</i> aufgerufen wird. Bei <code>delete</code> wird zuerst ein Destruktor aufgerufen, bevor der Speicherplatz freigegeben wird.</p> <p>Der Rückgabewert von <code>new</code> ist ein <i>Pointer</i> auf das neu erstellte Objekt, und <code>delete</code> verlangt einen <i>Pointer</i>. Wird mit <code>delete</code> ein Objekt gelöscht, so sollte man immer <i>alle Pointer</i>, die auf das Objekt zeigen, auf 0 setzen.</p> <p>Jedes <code>new</code> braucht ein <code>delete</code>. Sonst existieren die erstellten Objekte bis zum Ende des Programms, was je nach Laufdauer eine grosse Speicherverwendung ist.</p>	
<pre>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 = 0; ptr2 = 0; // has to be done !separately! ...</pre>	

<code>new ... [], delete []</code>	Ranges mit dynamischer Lebensdauer und Länge erstellen und löschen.
<pre>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</pre>	

Programmier-Befehle - Woche 12

Copy-Konstruktor	Kopier-Initialisierung
Der Copy-Konstruktor hat Argumenttyp <code>const My_Class&</code> .	
<pre>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]; } ... private: Customer* cust; // pointer to an array containing customers int length; // length of cust ... };</pre>	

<code>operator=</code>	Kopier-Zuweisung
<p><code>operator=</code> ist eng verwandt mit dem Copy-Konstruktor. Der Unterschied ist, dass der Copy-Konstruktor nur bei der Initialisierung aufgerufen wird, <code>operator=</code> hingegen nur <i>nach</i> der Initialisierung. z.B.</p> <pre>my_class a (5, 6); // Call a general constructor my_class b = a; // Call copy-constructor c = b; // Call operator=</pre> <p><code>operator=</code> 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 <code>operator=</code> meistens zuerst das aktuell vorhandene Objekt gelöscht werden, bevor die Kopie erstellt werden kann. Dies ist beispielsweise beim Stack aus der Vorlesung relevant.</p> <p><code>operator=</code> gibt im Normalfall eine Referenz auf seinen linken Operanden zurück.</p> <p>Faustregel: Meistens führt <code>operator=</code> zuerst die Aufgaben des Destructors, und dann die Aufgaben des Copy-Konstruktors aus.</p>	

(...)

(...)

```
// 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
<pre>// for Customer-struct see example on Copy-Constructor class Insurance { public: ~Insurance () { delete[] cust; } // free dynamic space ... };</pre>	