LANDESBERUFSSCHULE 4 SALZBURG

Informatik

Funktionen überladen, Default-Argumente, Speicherklassen, inline, const

LBS 4

Dieses Skript dient als zusätzliche Lernunterlage für Informatik

Inhalt

Überladen von Funktionen/Methoden

Wenn Funktionen mit unterschiedlichen Argumenten das „gleiche“ tun sollen, so muss man in traditionellen Programmiersprachen Funktionen mit unterschiedlichen Namen definieren.

C++ unterstützt wie auch viele andere OOP-Sprachen das Überladen von Methoden. Überladen bedeutet hier, dass zwei Funktionen den gleichen Namen, aber andere Argumente besitzen.

Soll beispielsweise die Summe von zwei ganzen Zahlen und zwei Gleitpunktzahlen gebildet werden, so kann man zwei Funktionen mit gleichen Namen aber verschiedenen Parametern definieren.

Beispiel: sum()

```
int32_t sum( int32_t x, int32_t y);
double sum(double x, double y);
```

Hier werden zwei verschiedene Funktionen mit dem gleichen Namen ***sum*** deklariert. Der Compiler unterscheidet überladene Funktionen anhand ihrer Signatur.

Die Signatur einer Funktion besteht aus der Anzahl und Typ der Parameter. Beim Aufruf vergleicht der Compiler die Signaturen der überladenen Funktionen und entscheidet sich für die passende.

Beispiel: C++ Code

```
//Überladene Funktionen

int32_t sum( int32_t x, int32_t y){ return x+y;}
double sum(double x, double y){return x+y;}

int main(){

int32_t value_int;
double value_double;

value_int = sum(1,4);              //int32 Bit-Version von sum.
value_double = sum(1.4,1.9);       //double-Version von sum.
return 0;
}
```

Default Argumente in Funktionen/Methoden

Für die Argumente einer Funktion können Standardwerte festgelegt werden. (Default-Argumente) Beim Aufruf einer Funktion müssen nicht alle Argumente angegeben werden. Für die fehlenden Argumente setzt der Compiler dann die Standardwerte ein.

Beispiel: C++ Code

```
//Defaultwerte

double summe(double s1, double s2 = 0, double s3 = 4){return s1+s2+s3;}

int main(){

double end_summe;
end_summe = summe(1,2,2);      //end_summe hat den Wert 5

end_summe = summe(1);          //end_summe hat den Wert 5

return 0;
}
```

Speicherklassen

Die Speicherklasse eines Objektes ist festgelegt durch

- die Position der Deklaration innerhalb der Quelldatei
- den Speicherklassen-Spezifizierer

Den Zugriff auf ein Objekt kann auf drei verschiedene Arten festgelegt werden:

- lokal Lebensdauer innerhalb eines Blockes
- modulglobal innerhalb eines Modules (Klassen, …….)
- programmglobal gesamte Programm

Es gibt verschiedene Typen von Spezifizierern:

extern:

Ein globales Objekt ist außerhalb von Funktionen, Klassen und Methoden definiert. Dadurch ist es im gesamten Programm verfügbar. Wenn ein Objekt in einer anderen Quelldatei verwendet werden soll, dann muss das Objekt mit dem Spezifizierer extern deklariert werden.

```
extern long position;        //Deklaration
```

Ein globales Objekt muss genau einmal definiert werden, kann aber öfters deklariert werden. Normalerweise wird das Objekt in einer Headerdatei deklariert die dann in allen benötigten Quellcodedateien inkludiert wird.

static:

Statische Objekte haben eine permanente Lebensdauer. Sie werden nicht am STACK sondern im globalen Speichersegment gehalten. Der Unterschied zu globalen Objekten ist, dass der Zugriff beschränkt ist.

Es gibt zwei Unterscheidungen.

- Definition außerhalb jeder Funktion
  - in diesem Fall ist Sie modulglobal
- Definition innerhalb eines Blockes
  - das Objekt ist nur in diesem einen Block sichtbar

```
static long position = 3;
```

Wenn statische Objekte nicht initialisiert werden, dann erhalten Sie den Wert 0.

register:

Um die Programmausführung zu beschleunigen können die *auto* Variablen im Register der CPU statt im Stack gehalten werden. In der Deklaration muss dazu das Schlüsselwort *register* verwendet werden.

Der Compiler kann die Anweisung register ignorieren, weil die Anzahl der Register sehr gering ist.

```
register int32_t my_integer = 1;
```

Inline-Funktionen

Beim Aufruf von Funktionen erfolgt ein Programmsprung. Das Hin- und Rückspringen beeinträchtigt die Programmlaufzeit. Für sehr kleine Funktionen die öfters aufgerufen werden müssen, kann man die Funktion als *inline*-Definieren.

Der Compiler fügt dann den Code der Funktion an die Stelle des Aufrufes ein. Damit findet kein Programmsprung statt.

Das Schlüsselwort heißt *inline*. Wenn eine Funktion zu viele Anweisungen enthält, kann der Compiler das Schlüsselwort ignorieren.

```
inline int64_t sum(int64_t x, int64_t y){ return x+y;};
```

const-Objekte und Methoden

Wenn ein Objekt als *const* deklariert wird, so kann das Programm nur lesend auf das Objekt zugreifen. Das Objekt muss dazu in der Definition initialisiert werden.

```
const Konto giro(„Param1“, 34, 12);
```

Das Objekt giro kann danach nicht mehr geändert werden. Methoden des Objektes können auch nicht mehr aufgerufen werden.

Read-Only-Methoden

Methoden die nur lesend auf Daten zugreifen und für konstante Objekte aufrufbar sein sollen, müssen als solche gekennzeichnet werden.

Das geschieht in der Definition und Deklaration der Methoden durch das Schlüsselwort const.

```
int getAmount() const;
```

Dadurch wird die Methode als Read-Only deklariert und ist über ein konstantes Objekt aufrufbar.

Überladen von Operatoren

Das überladen von Operatoren funktioniert ähnlich wie das Überladen von Funktionen. Ein selbstdefinierter Operator entspricht einer Funktion wo einem vorhandenen Operator eine neue Funktion zugewiesen wird.

Man kann nur bekannte Operatoren überladen, es ist nicht möglich selbstdefinierte Symbole zu erstellen. Diese Operatoren **::**, **.***, **.**, **?** können nicht überladen werden

Die Operatorfunktion unterscheidet sich von herkömmlichen Funktionen durch:

- der Aufruf wird in einen Funktionsaufruf umgewandelt
  - s = operator+(x,y) --> x.operator(y) → s= x+y
- der Funktionsname besteht aus dem Schlüsselwort „operator“ und dem Operatorzeichen.
  - operator+(a,b);

Beispiel:

```
// Zahlen von Objekten addieren *.hpp
class Calculate {
private:
   int64_t m_value;
public:
   int64_t getM_value() const;
   void setM_value(int64_t m_value);

//Operator + überladen
   friend Calculate operator+( cost Calculate &first , const Calculate &second );

   bool operator==(const Calculate &rhs) const; //Operator == überladen
   bool operator!=(const Calculate &rhs) const; //Operator != überladen
   Calculate(int64_t m_value);
   virtual ~Calculate();
};

// *.cpp

Calculate operator+(const Calculate& first ,const Calculate& second){
   Calculate temp(0);
   temp.m_value = first.m_value + second.m_value;
   return temp;
}

// main

int main() {
   auto number_1 = Calculate(12);
   auto number_2 = Calculate(25);
   Calculate result = number_1 + number_2;
   std::cout << result.getM_value() << std::endl;
   return 0;
}
```