Softwarepraktikum C++ / LEDA

Andrea Jaax

12. November 2012

Inhaltsverzeichnis

Inhalt und Ziel des Praktikums

- Einführung in C++
  - Einfache und komplexe Datenstrukturen
- Kennenlernen der C++ Klassenbibliothek LEDA (**L**ibrary of **E**fficient **D**ata Types and **A**lgorithms)
  - Komfortable Graph-Datentypen, Graphalgorithmen
  - geometrische Algorithmen
  - Visualisierung von Algorithmen

Organisation

2 Phasen:

1. Praktikum mit begleitenden Übungen
   - mindestens 50 % der Punkte aus den Übungen
   - höchstens einmal 0 Punkte
2. Projektarbeit
   - Ausarbeitung des Projekts
   - kurzer Vortrag

Organisation

- In den Übungen herrscht Anwesenheitspflicht!!
- Abgabe der Übungen per Email bis Montags 10:00 Uhr.
- Wir bestehen auf Gruppenabgaben!
- Die Bewertung des Projekts erfolgt Gruppenweise!

Termine

- 18.10.2012 - 13.12.2012 jeweils Donnerstags 8:30 Uhr Praktikum
- 01.11.2012: Allerheiligen
- 13.12.2012 oder 20.12.2012: Projektverteilung
- Mitte Januar: Zwischenstand
- Anfang Februar: Abgabe Code / Ausarbeitung / Vortrag
- Mitte / Ende Februar: Vorträge

Übungsabgaben

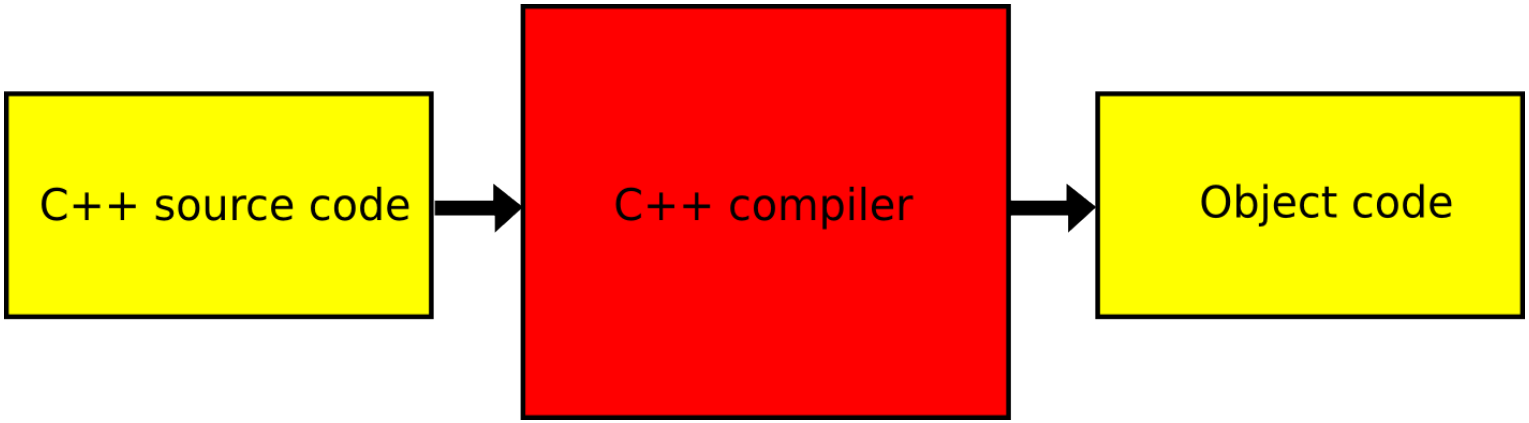
Bitte beachten:

- Programme, die sich nicht übersetzen lassen, geben keine Punkte!
- Programme, die (z.B. aus dem Internet) abgeschrieben wurden, geben keine Punkte!
- Bitte Programmcode gut kommentieren.
- Bitte nur eine .cpp-Datei pro Programm abgeben!
- Auch auf mündliche Anweisungen achten! ;-)

Hallo Welt Programm

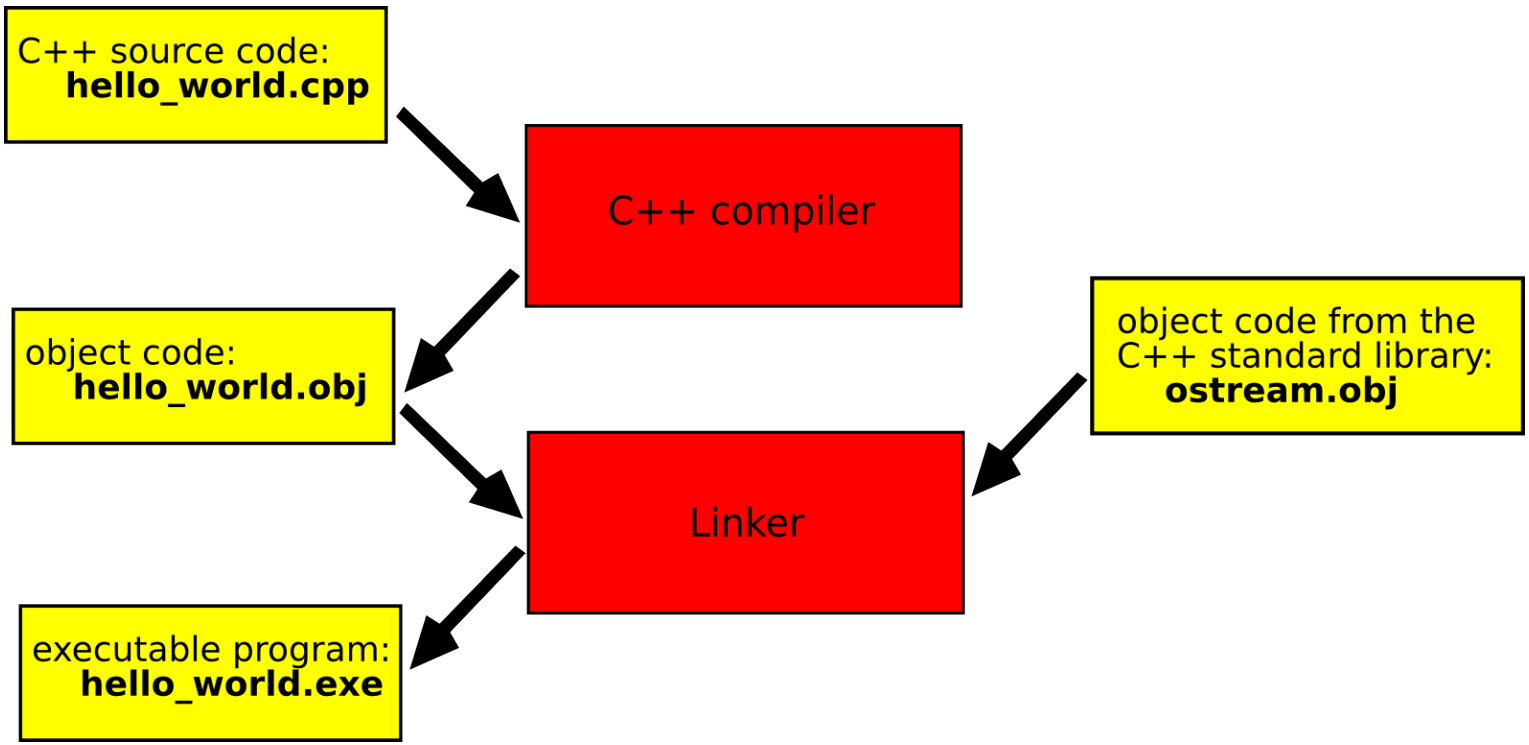
```
1 #include "std_lib_facilities.h"
2
3 int main()
4 {
5     cout << "Hallo Welt!\n";
6     return 0;
7 }
```

Kompilieren



Der Compiler kontrolliert, ob euer Quelltext korrekt geschrieben ist, jedes Word eine definierte Bedeutung besitzt oder ob es andere ersichtliche Fehler gibt, die bereits ohne Ausführen des Programms erkennbar sind.

Linken



Ein Programm besteht meistens aus mehreren seperaten Teilen, welche zusammengeführt werden müssen.

Eine einzelne Quelldatei (*.cpp) übersetzen

std_lib_facilities.h | hello_world.cpp

```
1 // This program outputs the message "Hello, World!" to the monitor
2
3 #include "std_lib_facilities.h"
4
5 int main()    // C++ programs start by executing the function main
6 {
7   cout << "Hello, World!\n";    //output "Hello, World!"
8   return 0;
9 }
```

```
jaax@golem:...~/Lehre/SS11/SWP/Programme
Datei  Bearbeiten  Ansicht  Terminal  Hilfe
jaax@golem:~/Lehre/SS11/SWP/Programme> g++ hello_world.cpp
jaax@golem:~/Lehre/SS11/SWP/Programme> a.out
Hello, World!
jaax@golem:~/Lehre/SS11/SWP/Programme>
```

nützliche Compiler-Flags

-o	(Output) Wenn die ausführbare Datei anders benannt werden soll als a.out, dann sollten Sie mit dieser Option den Namen der Ausgabedatei angeben.
-c	(Compile) Kompilieren ohne zu linken.
-g	(Debug) Fügt Standard-Debug-Informationen zum Debuggen hinzu.
-Wall	Aktiviert allgemeine sinnvolle Warnungen, die vom g++-Compiler unterstützt werden.
-v	Zeigt alle Schritte und Kommandos in der Kommandozeile an, die der Compiler gerade ausführt.

Einfache Ausgabe

```
1 #include "std_lib_facilities.h"
2
3 int main()
4 {
5    cout << "Dieser Text steht in der Kommandozeile!";
6
7    cout << "Dieser Text schliesst sich direkt an.";
8 }
```

Einfache Ausgabe

Zwei Möglichkeiten für einen Zeilenumbruch:

- Escape-Sequenz \n
- Manipulator **endl** (leert zusätzlich den Ausgabespeicher)

```
1 #include "std_lib_facilities.h"
2
3 int main()
4 {
5    cout << "Zeile 1 \n";
6    cout << "Zeile 2" << endl;
7    cout << "Zeile 3";
8 }
```

Eingabe

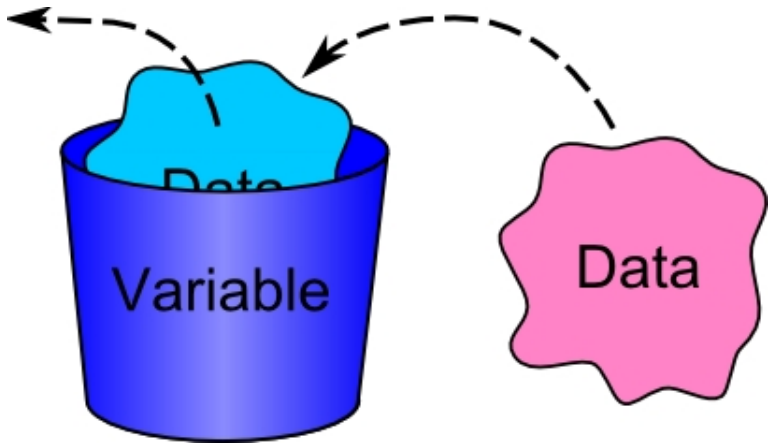
Für eine Eingabe muss zuerst eine Variable angelegt werden, in die das Gespeicherte abgelegt werden kann:

```
1 #include "std_lib_facilities.h"
2
3 int main()
4 {
5    cout << "Please␣enter␣your␣first␣name:␣\n";
6    string first_name;
7    cin >> first_name;
8    cout << "Hello␣" << first_name << "!\n";
9 }
```

Variablen

Eine Variable hat

- einen Namen (hier Variable),
- einen Typ (hier Eimer),
- und einen Wert (hier Data).



Variablen

Deklaration einer Variablen:

```
1 Datentyp Name;
```

Variablen

Deklaration einer Variablen:

```
1 Datentyp Name;
```

Es ist auch möglich mehrere Variablen gleichzeitig zu deklarieren:

```
1 Datentyp Variable1, Variable 2, Variable 3;
```

Variablen

Deklaration einer Variablen:

```
1 Datentyp Name;
```

Es ist auch möglich mehrere Variablen gleichzeitig zu deklarieren:

```
1 Datentyp Variable1, Variable 2, Variable 3;
```

Initialisierung einer Variablen:

```
1 int Zahl=100;                    // Moeglichkeit 1
2 int Zahl(100);                   // Moeglichkeit 2
```

Sichtbarkeit und Lebensdauer von Variablen

Bei der Sichtbarkeit und Lebensdauer von Variablen unterscheiden wir hauptsächlich zwischen drei Typen von Variablen:

- globalen Variablen,
- lokalen Variablen und
- statischen Variablen.

Sichtbarkeit und Lebensdauer von Variablen

```
1  int global = 5;
2  int main(void)
3  {
4    int lokal = 2;
5
6    cout << "Globale Variable: " << global << endl;
7    cout << "lokale Variable: " << lokal << endl;
8    {
9      global = 10;
10     int lokal = 8;
11
12     cout << "Globale Variable: " << global << endl;
13     cout << "lokale Variable: " << lokal << endl;
14   }
15   cout << "Globale Variable: " << global << endl;
16   cout << "lokale Variable: " << lokal << endl;
17 }
```

Sichtbarkeit und Lebensdauer von Variablen

Was passiert nun?

```
1  int global = 5;
2  int main(void)
3  {
4    int lokal = 2;
5
6    cout << "Globale Variable: " << global << endl;
7    cout << "lokale Variable: " << lokal << endl;
8    {
9      global = 10;
10     int lokal = 8;
11     int lokal_2 = 6;
12
13     cout << "Globale Variable: " << global << endl;
14     cout << "lokale Variable: " << lokal << endl;
15   }
16   cout << "Globale Variable: " << global << endl;
17   cout << "lokale Variable: " << lokal << endl;
18   cout << "lokale Variable 2: " << lokal_2 << endl;
19 }
```

Sichtbarkeit und Lebensdauer von Variablen

Lokale Variablen:

- Variablen, die innerhalb eines Blockes definiert werden.
- jede Variable ist nur in dem Block gültig, in dem sie vereinbart wurde, sowie in allen darin enthaltenen Blöcken;
- es sei denn, in einem Unter-Block wird eine Variable gleichen Namens definiert. Dann bezieht sich in diesem Unter-Block der Bezeichner auf die im Unter-Block angelegte Variable.

Sichtbarkeit und Lebensdauer von Variablen

Globale Variablen:

- werden ausserhalb jedes Blockes definiert und gelten ab der Stelle, an der sie deklariert werden.
- Wird jedoch in einem Block eine Variable gleichen Namens angelegt, gilt ab hier bis zum Ende des Blocks nicht mehr die globale Variable, sondern die im Block deklarierte.

Sichtbarkeit und Lebensdauer von Variablen

Statische Variablen:

```
1 ...
2 {
3   static int variable = 5;
4 }
5 ...
```

Eigenschaften:

- festen Speicherbereich während der gesamten Ausführungszeit
- Speicherbereich im RAM wird nach dem Schließen des Blocks, in dem sie deklariert wurde, nicht zerstört
- Wird der gleiche Block noch mal betreten, so besitzt die Variable immer noch den selben Wert, der ihr zuletzt zugewiesen wurde

Konstanten

- Variablen, die ihren Wert nicht verändern.
- Konstanten werden mit **const** gekennzeichnet.
- const bezieht sich auf das was links von ihm steht. Wenn links neben const nichts steht, dann bezieht sich const auf das rechts von ihm.

```
1 int const Zahl(400);                    // oder
2 const int Zahl(400);
```

Die wichtigsten Datentypen

C++ stellt sehr viele Typen bereit, aber mit folgenden kann schon ein sehr gutes Programm geschrieben werden:

- **int** für ganze Zahlen
- **double** für Gleitkommazahlen
- **char** für individuelle Buchstaben wie z.B. “.”
- **string** für character strings
- **bool** für logische Variablen

Datentypen

Ganze Zahlen:

Typ	**Speicherplatz**	**Wertebereich**
char	1 Byte	-128 bis +127 bzw. 0 bis 255
signed char	1 Byte	-128 bis +127
unsigned char	1 Byte	0 bis 255
short	2 Byte	-32768 bis +32767
unsigned short	2 Byte	0 bis 65.535
int	4 Byte	-2.147.483.648 bis +2.147.483.647
unsigned int	4 Byte	0 bis 4.294.967.295
long	8 Byte	-9.223.372.036.854.775.808 bis + 9.223.372.036.854.775.807
unsigned long	8 Byte	0 bis 18.446.744.073.709.551.615

Datentypen

Gleitkommazahlen:

Typ	Speicherpl	Wertebereich	kl pos Zahl	Genauigkeit
float	4 Byte	$\pm 3,4 * 10^{38}$	$1,2 * 10^{-38}$	6 Stellen
double	8 Byte	$\pm 1,7 * 10^{308}$	$2,3 * 10^{-308}$	15 Stellen
long double	10 Byte	$\pm 1,1 * 10^{4932}$	$3,4 * 10^{-4932}$	19 Stellen

Überläufe bei Ganzzahlen

Passt das Ergebnis einer Berechnung nicht in den Wertebereich einer Zahl, so wird dieser Fehler nicht vom System angezeigt!

Aufgabe:
Lassen Sie sich das Ergebnis der folgenden Gleichung ausgeben:

$$100000 * 100000/100000$$

Was fällt Ihnen auf?

Input und Typen

Die Input-Operation >> ist Typsensitiv!

```
1  #include "std_lib_facilities.h"
2
3  int main()
4  {
5    cout << "Please␣enter␣your␣first␣name␣and␣age\n";
6    string first_name;
7    int age;
8    cin >> first_name;
9    cin >> age;
10   cout << "Hello,␣" << first_name <<
11     ",␣your␣age␣is␣" << age << "\n";
12 }
```

Input und Typen

Was passiert, wenn Sie als Input z.B. zuerst das Alter und anschließend den Namen eingeben?

Operationen

```
int count;
cin » count; // » reads an integer into count
string name;
cin » name; // » reads a string into name

int c2 = count + 2; // + adds integers
string s2 = "Hallo "+ name; // + appends characters

int c3 = count - 2; // - subtracts integers
string s3 = -"Hallo "+ name; // error: - isn't defined for strings
```

nützliche Operatoren

	bool	char	int	double	string
Zuweisung	=	=	=	=	=
Addition			+	+	
Konkatenation					+
Subtraktion			-	-	
Multiplikation		*	*		
Division			/	/	
Modulo			%		
Inkrementieren			++	++	
Dekrementieren			–	–	
Inkrem. um *n*			+=n	+=n	
Add. am Ende					+=
Dekrem. um *n*			-=n	-=n	

nützliche Operatoren

	bool	char	int	double	string
Multiplik. und Zuweisung			*=	*=	
Division und Zuweisung			/=	/=	
Modulo und Zuweisung			%=	%=	
Gleichheit	==	==	==	==	==
Ungleichheit	!=	!=	!=	!=	!=
größer als	>	>	>	>	>
größer gleich	>=	>=	>=	>=	>=
kleiner als	<	<	<	<	<
kleiner gleich	<=	<=	<=	<=	<=

Lösen von Rechenaufgaben:

```
1  #include "std_lib_facilities.h"
2
3  int main()
4  {
5    int Ergebnis;
6
7    // Ergebnis berechnen
8    Ergebnis = ((3+3*4)/5-1)*512-768;
9
10   // Aufgabe und Ergebnis ausgeben
11   cout << "((3+3*4)/5-1)*512-768 = "
12     << Ergebnis << endl;
13 }
14
15 Ausgabe:
16 ((3+3*4)/5-1)*512-768 = 256
```

Lösen von Rechenaufgaben:

Wie in der Mathematik gilt auch in C++ die Regel:
Klammern vor Punkt vor Strich.

Zusammengesetzte Operatoren

C++ ist eine Sprache für schreibfaule Menschen:

```
1  Zahl  = 22;
2
3  Zahl += 5;                    // Zahl = Zahl + 5;
4
5  Zahl -= 7;                    // Zahl = Zahl - 7;
6
7  Zahl *= 2;                    // Zahl = Zahl * 2;
8
9  Zahl /= 4;                    // Zahl = Zahl / 4;
```

Inkrement und Dekrement

Inkrementieren = Wert einer Variablen um 1 erhöhen
Dekrementieren = Wert einer Variablen um 1 herunterzählen

Die beiden Variablen gibt es jeweils in der Präfix und der Postfix Variante:

```
1 int i = 1, j = 1;
2 int a, b, c, d;
3
4 a = i++;                    // a = 1, i = 2
5 b = i++;                    // b = 2, i = 3
6 c = ++j;                    // j = 2, c = 2
7 d = ++j;                    // j = 3, d = 3
```

Postfix (häufig):

1. Wert des Ausdrucks = (ursprünglicher!) Wert der Variablen
2. Variable hochzählen (herunterzählen)

Postfix (häufig):

1. Wert des Ausdrucks = (ursprünglicher!) Wert der Variablen
2. Variable hochzählen (herunterzählen)

Präfix (seltener):

1. Variable hochzählen (herunterzählen)
2. Wert des Ausdrucks = (veränderter!) Wert der Variablen

Bedingte Befehlsausführung

```
1 if($Bedingung$)
2 {
3     $Anweisungen$
4 }
5 else
6 {
7     $Anweisungen$
8 }
```

Bedingte Befehlsausführung

Beispiel:

Berechne das Maximum zweier Zahlen a und b:

```
1 #include "std_lib_facilities.h"
2
3 int main()
4 {
5   if( a < b )
6     max = b;
7   else
8     max = a;
9 }
```

Vergleichsoperatoren

Es gibt 6 verschiedene Vergleichsoperatoren:

==	identisch
<=	kleiner gleich
>=	größer gleich
<	echt kleiner
>	echt größer
!=	nicht gleich

Logische Operatoren

und folgende logische Operatoren:

!	Logisches Nicht
& &	Logisches Und
||	Logisches Oder

Wiederholte Befehlsausführung

```
1 for(
2       $Initialisierungsteil$;
3       $Bedingungsteil$;
4       $Anweisungsteil$
5     )
6 {
7    $Schleifenrumpf$
8 }
```

Wiederholte Befehlsausführung

Beispiel:

Gebe alle Zahlen bis zu einer eingegebenen Grenze aus:

```
1  #include "std_lib_facilities.h"
2
3  int main()
4  {
5    int Grenze;
6
7    cin >> Grenze;
8
9    for(int i = 1; i <= Grenze; ++i) // for-Schleife
10     cout << i << endl;             // Ausgabe von i
11 }
```

Die break-Anweisung

Die break-Anweisung wird innerhalb von Schleifen verwendet, um diese sofort zu beenden.

```
1  int main()
2  {
3    int Grenze, i=1;
4    cin >> Grenze;
5
6    // Endlosschleife
7    for(;;){
8      cout << i << endl;
9      ++i;
10     // Abbrechen, wenn Bedingung erfuellt
11     if(i > Grenze) break;
12   }
13 }
```

Die continue-Anweisung

Mit der continue-Anweisung kann der Rest des aktuellen Schleifendurchlauf übersprungen werden.

Gebe alle ungeraden Zahlen bis zu der eingegebenen Grenze aus:

```
1  int main()
2  {
3    int Grenze;
4    cin >> Grenze;
5
6    for(int i = 1; i <= Grenze; ++i)
7    {
8      // alle geraden Zahlen ueberspringen
9      if (i % 2 == 0) continue;
10     cout << i << endl;
11   }
12 }
```

Wiederholte Befehlsausführung

“while”-Schleife:

```
1 while($Bedingung$)
2 {
3     $Anweisungen$
4 }
```

“do-while”-Schleife:

```
1 do
2 {
3     $Anweisungen$
4 }
5   while($Bedingung$);
```

Auswahl

Die switch-Anweisung:

```
1  switch($Ganzzahlige Variable$)
2  {
3      case $Ganzzahl$: $Anweisungsblock$
4
5      case $Ganzzahl$: $Anweisungsblock$
6
7      $...$
8
9      default: $Anweisungsblock$
10 }
```

Jeder Anweisungsblock muss durch eine break-Anweisung unterbrochen werden!!

Funktionen

- Funktionen erlauben es Code auszulagern und in kleine, zusammengehörende Teile zu zerschneiden.
- Wir müssen also nicht jedesmal die Funktion komplett schreiben, sondern sie einfach aufrufen.
- Funktionen bieten die Möglichkeit Code-Redundanz zu vermeiden.

Funktionen

Funktionsdeklaration und -definition im Programm:

```
1  int f(int x);              // Funktionsdeklaration
2
3  int main()
4  {
5    int a = f(3);            // Funktionsaufruf
6    ...                      // a hat jetzt den Wert 9
7  }
8
9  int f(int x)               // Funktionsdefinition
10 {
11   return x * x;
12 }
```

Funktionen

Übergabe der Argumente:

- call-by-value
  Bei call-by-value (Wertübergabe) wird der Wert des Arguments in einen Speicherbereich kopiert, auf den die Funktion mittels Parametername zugreifen kann.

```
1 int f(int x);
```

- call-by-reference
  Im Gegensatz zu call-by-value wird bei call-by-reference die Speicheradresse des Arguments übergeben, also der Wert nicht kopiert.

```
1 int f(int &x);
```

Funktionen

Default-Parameter:

Wenn beim Aufruf einer Funktion nicht alle Parameter explizit angegeben wurden, werden diese mit dem Default-Parameter belegt.

```
1 int summe(int a, int b, int c = 1, int d = 0) {
2   return a + b + c + d;
3 }
4
5 int main() {
6   int x = summe(2, 3, 4, 5);
7   x = summe(2, 3, 4);
8   x = summe(2, 3);
9 }
```

Funktionen

Default-Parameter:

```
1  int summe(int a, int b, int c = 1, int d = 0)
2  {
3    return a + b + c + d;
4  }
5
6  int main()
7  {
8    int x = summe(2, 3, 4, 5);     //x == 14
9    x = summe(2, 3, 4);            //x == 9
10   x = summe(2, 3);               //x == 6
11 }
```

Aufgabe 1

Adresse:

Schreiben Sie ein Programm,
das Ihren Namen und Ihre vollständige Adresse
auf dem Bildschirm ausgibt.

Aufgabe 2

Schaltjahre:

In unserem Kalender sind zum Ausgleich der astronomischen und der kalendarischen Jahreslänge in regelmäßigen Abständen Schaltjahre eingebaut.
Zur exakten Festlegung der Schaltjahre dienen die folgenden Regeln:
(1) Ist die Jahreszahl durch 4 teilbar, so ist das Jahr ein Schaltjahr.
Diese Regel hat allerdings eine Ausnahme:
(2) Ist die Jahreszahl durch 100 teilbar, so ist das Jahr kein Schaltjahr.
Diese Regel hat wiederum eine Ausnahme:
(3) Ist die Jahreszahl durch 400 teilbar, so ist das Jahr doch ein Schaltjahr.

Schreiben Sie ein Programm, mit dessen Hilfe man feststellen kann, ob ein bestimmtes Jahr ein Schaltjahr ist oder nicht!

Aufgabe 3

Jahrestag:

Erstellen Sie ein Programm, das zu einem eingegebenen Datum (Tag, Monat und Jahr) berechnet, um den wievielten Tag des Jahres es sich handelt. Berücksichtigen Sie dabei die Schaltjahrregel!

Aufgabe 4

Rekursion:

Definieren und implementieren Sie die

- Fakultätsfunktion
- Fibonaccifunktion
  Erinnerung: $f_n = f_{n-1} + f_{n-2}$ für $n \geq 2$ mit $f_0 = 0$ und $f_1 = 1$
- und die Ackermannfunktion (x und y sind natürliche Zahlen mit x = 0 und y = 0):
  - $A(0, y) = y + 1$
  - $A(x + 1, 0) = A(x, 1)$
  - $A(x + 1, y + 1) = A(x, A(x + 1, y))$

Aufgabe 5

Größte und kleinste Zahl:

Schreiben Sie ein Programm, das eine vom Benutzer festgelegte Anzahl von Zahlen einliest und anschließend die größte und kleinste der eingegebenen Zahlen auf dem Bildschirm ausgibt.

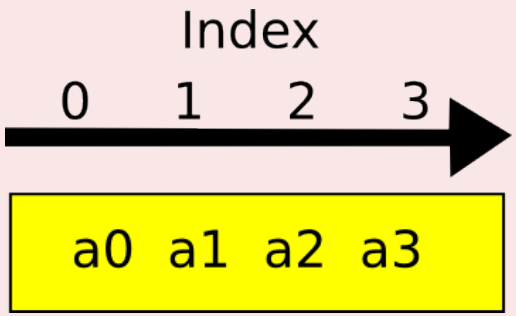
Werte speichern

Erstellen Sie ein Programm, das 100 Zahlen einliest und die Zahlen in umgekehrter Reihenfolge wieder ausgibt.

Arrays

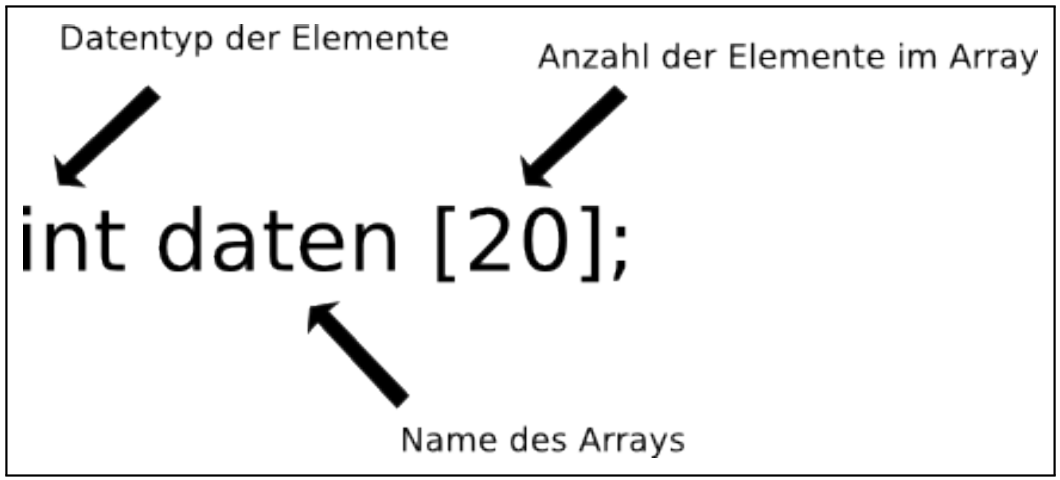
Definition

Arrays *sind Reihungen von Datenelementen gleichen Typs. Über ein Index kann auf jedes Datenelement unmittelbar zugegriffen werden. Der einfachste Fall ist eine eindimensionale, lineare Anordnung der Datenelemente:*



Arrays

Notation eines Arrays in C++:



Ein einzelnes Element des Arrays wird über seinen Index angesprochen:

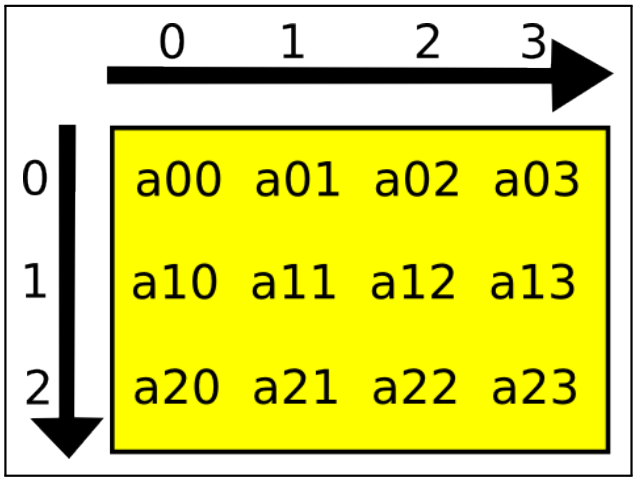
```
1 daten[7] = 123
```

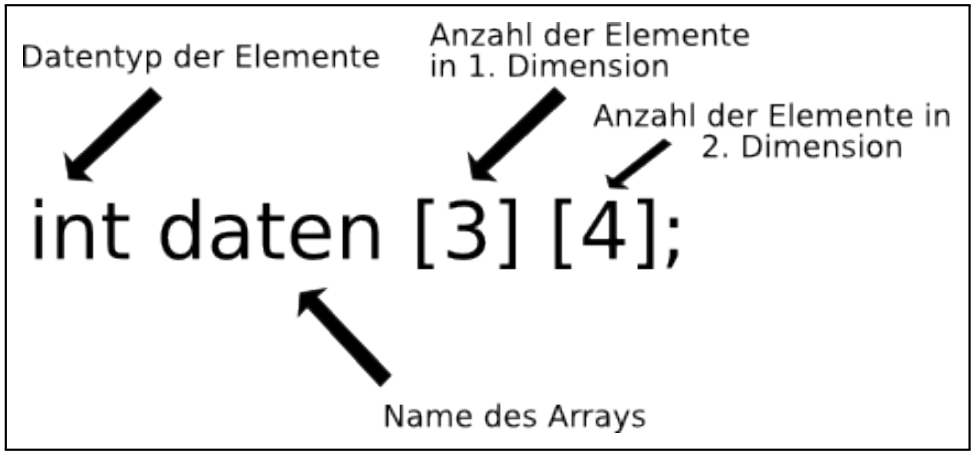
Arrays

Lösung der Aufgabe mit Hilfe von Arrays:

```
1  int main() {
2    int daten[100];
3    int i;
4
5    for(i=0; i<100; i++) {
6      cout << "Bitte␣geben␣Sie␣die␣" << i << "-te␣Zahl␣ein:␣";
7      cin >> daten[i];
8      cout << endl;
9    }
10   for(i=99; i>=0; i--)
11     cout << "Die␣" << i << "-te␣Zahl␣ist:␣" << daten[i];
12 }
```

Zweidimensionale Arrays





Bei der Definition muss angegeben werden, wie viele Dimensionen und welche Ausdehnung das Array in jeder Dimension haben soll.

Zweidimensionale Arrays

Arrays können direkt bei der Definition initialisiert werden:

```
1 int daten_1dim[20] = { 1, 2, 3, 4, 5};
2
3 int daten_2dim[3][4] =
4 {
5   { 0, 1, 2, 3},
6   { 1, 2, 3, 4},
7   { 2, 3, 4, 5},
8 };
```

Felder (Klasse array)

Sortieren durch Minimumsuche:

A = [6, 5, 4, 3, 2, 1]

-> Minimum an Stelle 5 mit Wert 1

-> tausche Element an Stelle 0 mit Minimum

A = 1, [5, 4, 3, 2, 6]

-> Minimum an Stelle 4 mit Wert 2

-> tausche Element an Stelle 1 mit Minimum

A = 1, 2, [4, 3, 5, 6]

-> Minimum an Stelle 3 mit Wert 3

-> tausche Element an Stelle 2 mit Minimum

A = 1, 2, 3, [4, 5, 6]

-> Minimum an Stelle 3 mit Wert 4

-> Minimum schon an richtiger Stelle

Felder (Klasse array)

Sortieren durch Minimumsuche:

```
1  #include "std_lib_facilities.h"
2  int main()
3  { int A[10] = {10, 9, 8, 7, 6, 5, 4, 3, 2, 1};
4    for(int i=0; i<10; i++)
5    { char min = A[i];
6      int min_pos = i;
7      for(int j=i+1; j<10; j++)
8      { if(A[j] < min)
9        { min = A[j];
10         min_pos = j; } }
11     A[min_pos] = A[i];
12     A[i] = min; }
13   for(int i=0; i<10; i++) cout << A[i] << "␣";
14   cout << endl;
```

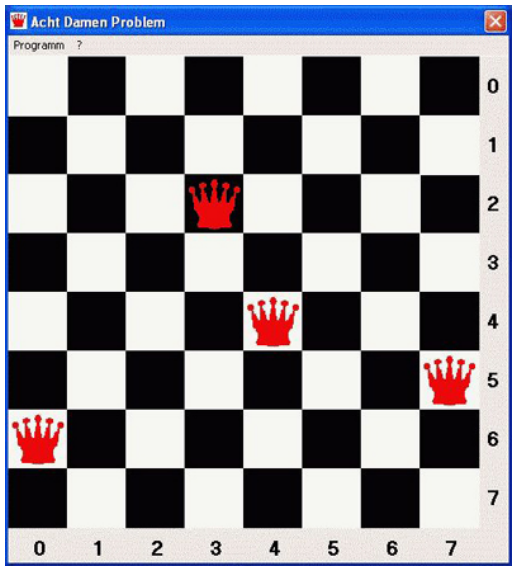
Felder (Klasse array)

```
1   for(int i=0; i<n; i++)
2   { char min = A[i];
3     int min_pos = i;
4     for(int j=i+1; j<n; j++)
5       if(A[j] < min) {
6         min = A[j];
7         min_pos = j;
8       }
9     A[min_pos] = A[i];
10    A[i] = min;
11  }
12  for(int i=0; i<n; i++) cout << A[i] << " ";
13  cout << endl;
14 }
```

Felder (Klasse array)

Damenproblem:

Die Dame ist die schlagkräftigste und spielstärkste Figur im Schach!



Felder (Klasse array)

Damenproblem:

Die Aufgabe des Damenproblems lautet, n Damen auf einem n x n Schachbrett so zu positionieren, dass keine Dame eine andere schlagen kann. Dies bedeutet:

- höchstens (genau) eine Dame in jeder Zeile
- höchstens (genau) eine Dame in jeder Spalte
- höchstens eine Dame in jeder Diagonalen

Im klassischen Fall des 8 x 8 Schachbretts gibt es 92 verschiedene Lösungen!

Felder (Klasse array)

Lösen des Damenproblem:

Datenstruktur: **int** x[8];
x[3] = 2 $\Longleftrightarrow$ 4. Dame steht in Spalte 2

Wir stellen uns nun immer die folgende Frage:
Kann die i-te gesetzte Dame eine der zuvor gesetzten Damen schlagen?

Felder (Klasse array)

Lösen des Damenproblem:

1. Stehen zwei Damen in der gleichen Zeile?
   Durch unser Modell ausgeschlossen, da sich jede Dame immer nur in einer Zeile bewegt.
2. Stehen zwei Damen (Index i, k) in der gleichen Spalte?

$$x[i] == x[k]$$

3. Stehen zwei Damen in einer Diagonalen?

$$|x[i] - x[k]| == |i - k|$$

Felder (Klasse array)

Lösen des Damenproblem:

Befehl der testet, ob der Platz (ix, iy) bereits von einer Königin 0...(iy-1) geschlagen wird

```
1  int x[8];
2
3  int is_free (int ix, int iy)
4  {
5    int i;
6    for(i=0; i<iy; i++)
7      if((x[i]==ix)  (abs(x[i]-ix)==abs(i-iy))) return 0;
8    return 1;
9  }
```

Felder (Klasse array)

Lösen des Damenproblem:

```
1  // druckt das Feld
2  void print()
3  {
4  int i, j;
5  cout << "+-----------------------------------------+" << endl;
6  for(i=0; i<8; i++){
7    cout << "|" ;
8      for(j=0; j<8; j++)
9        if(j==x[i]) {
10         cout << "<x>|";
11       }
12       else cout << "   |";
13     cout << "\n";
14     cout << "+-----------------------------------------+" << endl;
15 }
16 cout << "\n";
17 }
```

Felder (Klasse array)

Lösen des Damenproblem:

Befehl, der alle möglichen Lösungen testet:

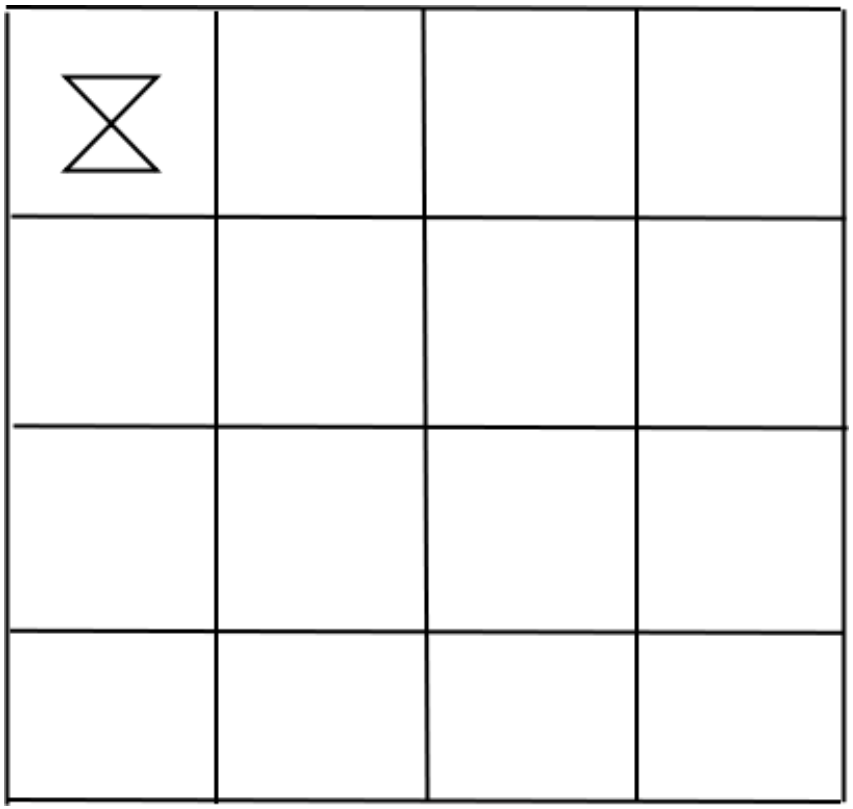
n=8: Feld wird gedruckt; n $<$ 8: mit is_free(i,n) wird getestet, ob Dame n auf Platz (i,n) platziert werden kann

```
1  void trying(int n)
2  {
3    int i;
4    if (n==8){
5         print();
6         }
7    else{
8      for(i=0; i<8; i++)
9        if(is_free(i,n)) {
10         x[n]=i;
11         trying(n+1);
12       }
13   }
14 }
15
16 int main ()
17 {
18   trying(0);
19   return 0;
20 }
```

Felder (Klasse array)

Lösen des Damenproblem:

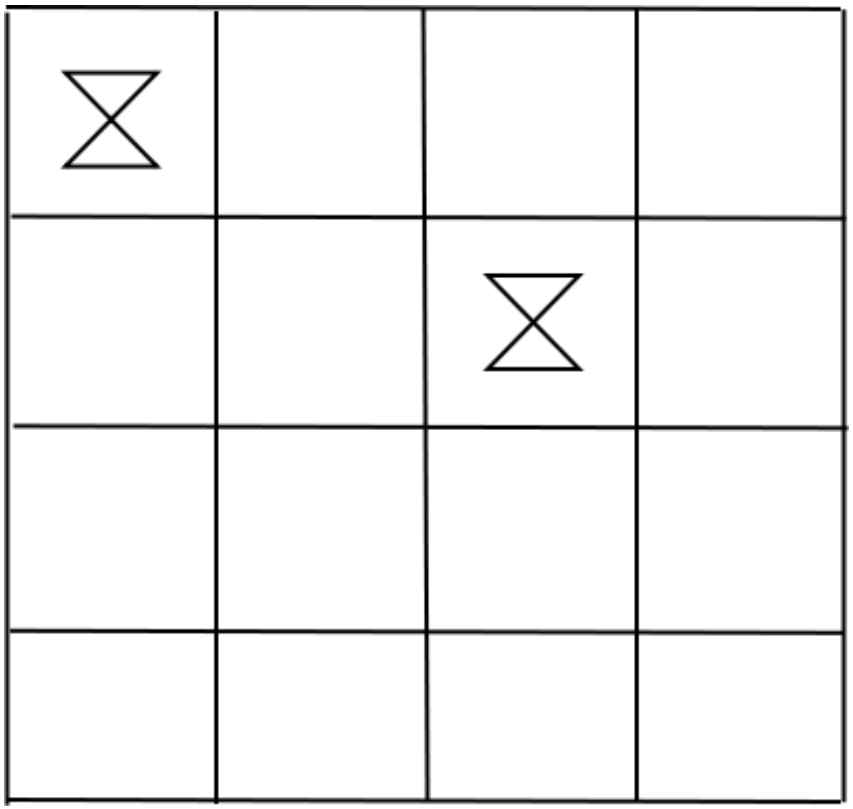
1. Schritt:



Felder (Klasse array)

Lösen des Damenproblem:

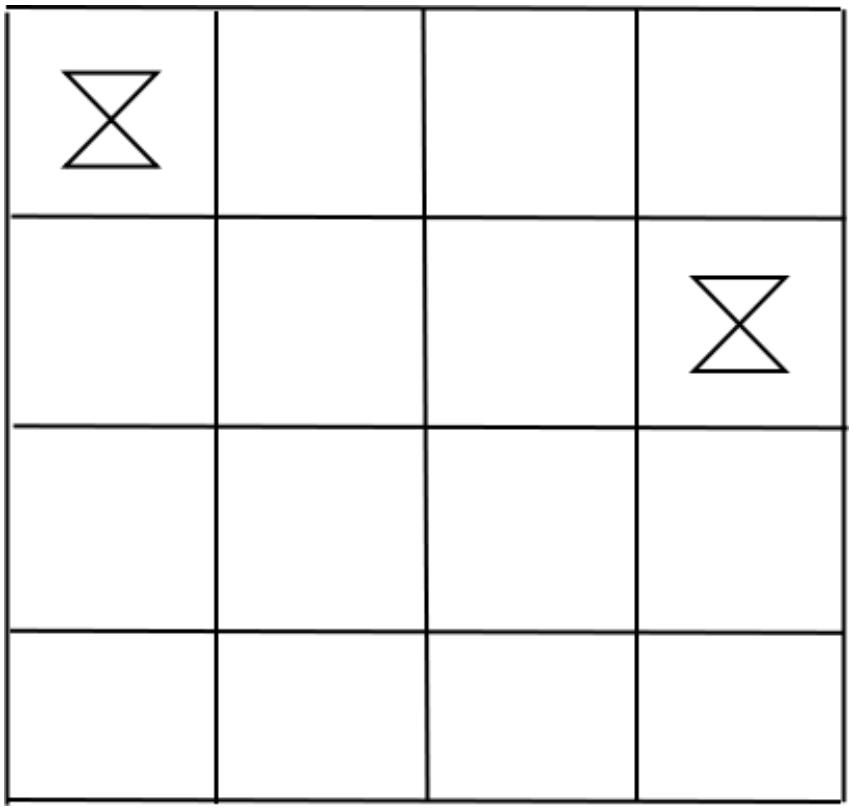
2. Schritt:



Felder (Klasse array)

Lösen des Damenproblem:

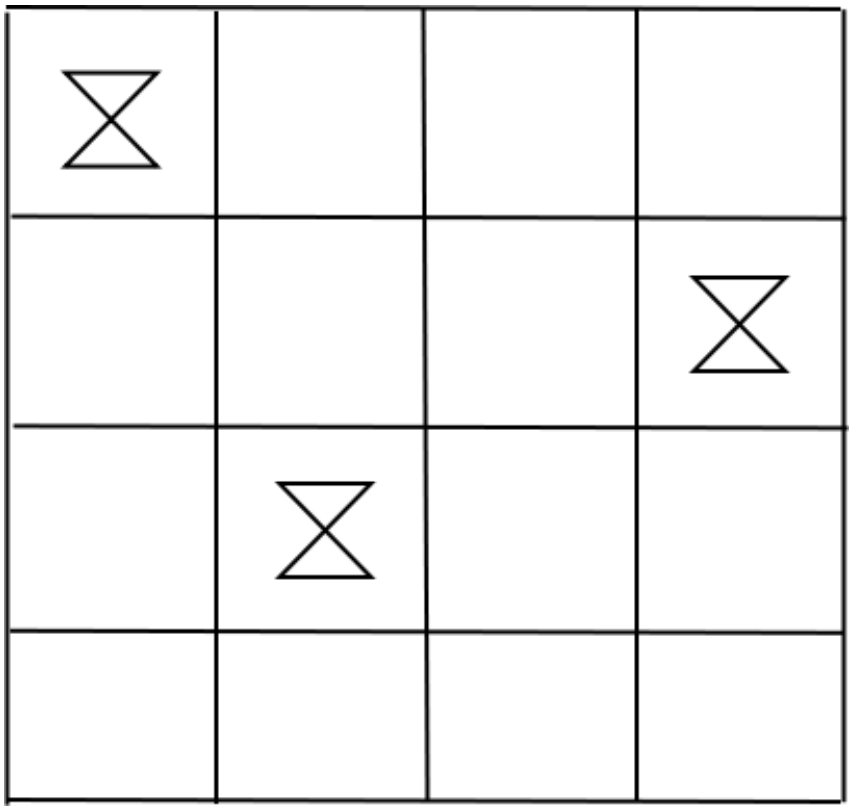
3. Schritt:



Felder (Klasse array)

Lösen des Damenproblem:

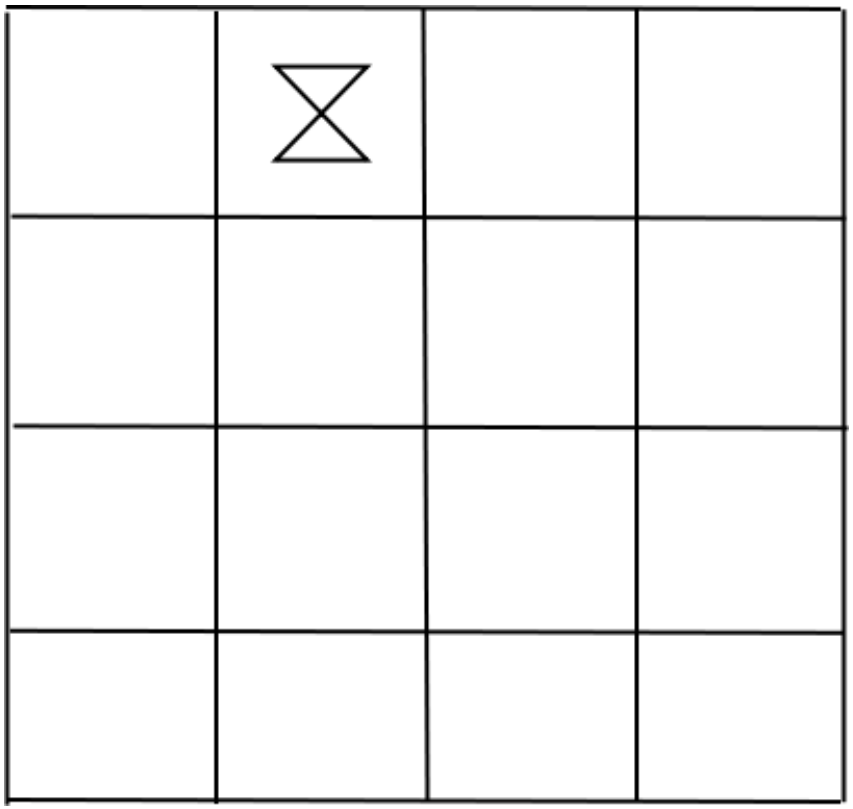
4. Schritt:



Felder (Klasse array)

Lösen des Damenproblem:

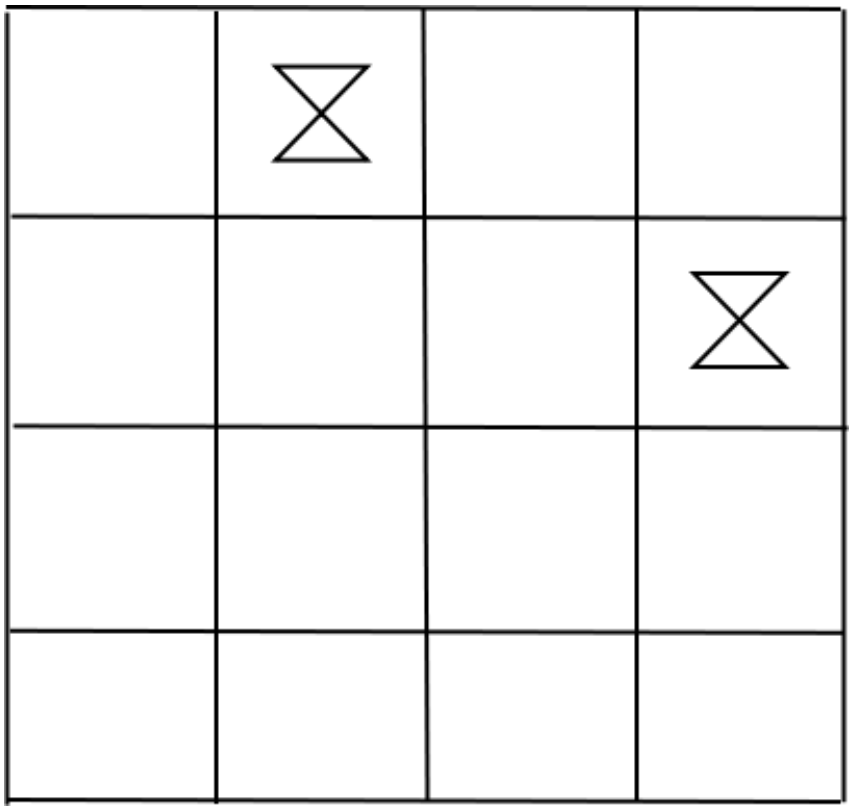
5. Schritt:



Felder (Klasse array)

Lösen des Damenproblem:

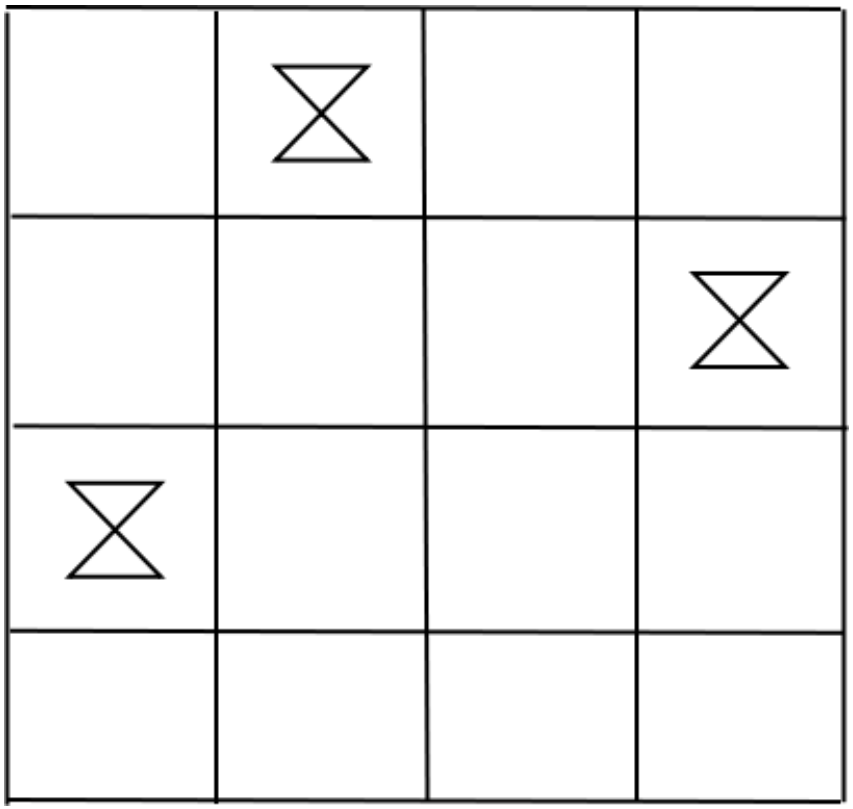
6. Schritt:



Felder (Klasse array)

Lösen des Damenproblem:

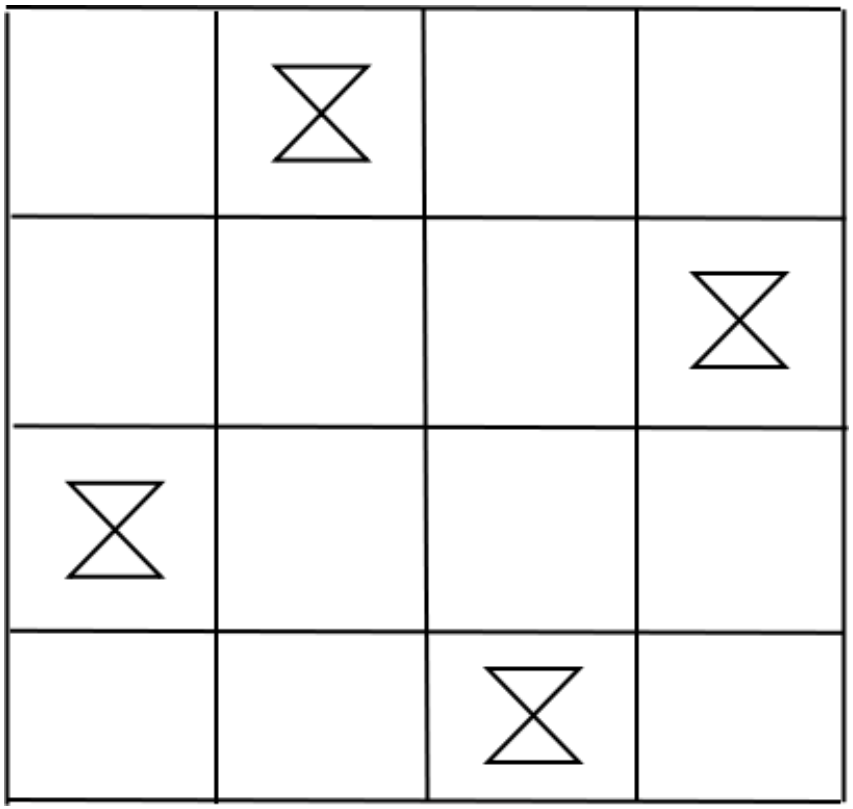
7. Schritt:



Felder (Klasse array)

Lösen des Damenproblem:

8. Schritt:

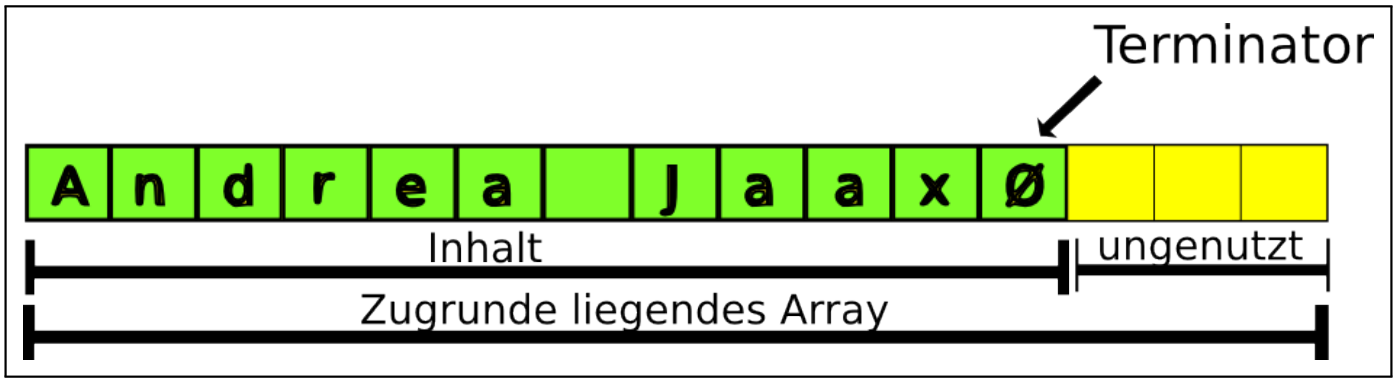


Zeichenketten

Definition

Ein String ist eine Zeichenkette, d.h. eine Folge von mehreren (oder auch gar keinem) Zeichen.

Ein String kann als eindimensionales Array von Zeichen (char) mit der zusätzlichen Eigenschaft, dass das letzte Zeichen NULL ist, angesehen werden.



Zeichenketten

Zu beachten:

- Der String befindet sich in einem Array fester Länge. Bei Manipulation des Strings dürfen die Grenzen des zugrunde liegenden Arrays nicht überschritten werden.
- Wegen des Terminators muss das Array, das den String beherbergt, mindestens ein Element mehr haben als der String Zeichen enthält.

$\Rightarrow$ leider recht umständlich!

Klasse string

Lösung:
Verwende die Klasse string aus der C++ Standardbibliothek!

Kompatibilität:
Ein Objekt vom Typ string verhält sich wie ein char-Array, z.B. kann auf ein einzelnes Zeichen eines Strings durch die eckigen Klammern zugegriffen werden.

Klasse string

Vorteile:

- Beim Anlegen eines Objekts der Klasse string muss nicht angegeben werden, wie viele Zeichen reserviert werden sollen.
- Wird mehr Platz benötigt, sorgt das Objekt selbst dafür, dass es den erforderlichen Speicherbereich bekommt.
- Strings können einfach zugewiesen und verglichen werden.

Klasse string

Verbinden:
Strings können aneinander gehängt werden, indem ein einfaches Pluszeichen zwischen sie gestellt wird.

Vergleiche:
Strings können mit den einfachen Vergleichsoperatoren (in lexikalischer Reihenfolge) verglichen werden.

Elementarfunktionen:

length()	liefert die Länge des Strings
insert(n,s)	fügt den String s an Position n ein
erase(p,n)	entfernt ab Position p n Zeichen
find(s)	liefert die Position, an der sich der String s befindet

Klasse string

Palindrom:
Ein Wort, das vorwärts wie rückwärts gelesen gleich ist, wird Palindrom genannt.

Vernachlässigen wir dabei Leerzeichen, Satzzeichen und Groß-/Kleinschreibung, so können wir die Definition auf ganze Sätze ausdehnen. Ein berühmtes Satzpalindrom ist:
“Nie grub Ramses Marburg ein”.

Klasse string

Palindrom-Test:

Schreiben Sie ein Programm, das einen String von der Standardeingabe liest und testet, ob dieser ein Palindrom ist.

Klasse string

Palindrom-Test:

```
1  #include "std_lib_facilities.h"
2
3  int main() {
4    string s;
5
6    getline (cin,s);
7
8    string  r, t;
9
10
11   for(int i = 0 ; i < s.length() ; i++)
12   {
13     if(s[i] == ' ') continue;
14     else t += s[i];
15   }
16
17   for(int i = s.length() - 1 ; i >= 0 ; i--)
18   {
19     if(s[i] == ' ') continue;
20     else r += s[i];
21   }
22
23   if(t == r)
24     cout << s << " ist ein Palindrom\n";
25   else
26     cout << s << " ist KEIN Palindrom\n";
```

Aufgabe 6

doppelte Leerzeichen:

Schreiben Sie ein Programm, das aus einem Text alle mehrfach vorkommenden Leerzeichen entfernt.

Aufgabe 7

Buchstaben zählen 1:

Schreiben Sie ein Programm, das einen Text und einen Buchtstaben einliest und ermittelt, wie oft der Buchstabe in dem Text vorkommt.

Aufgabe 8

Buchstaben zählen 2:

Erstellen Sie ein Programm, das einen Text einliest und dann eine Statistik über die Häufigkeit des Vorkommens von Buchstaben erstellt. Der Einfachheit halber beschränken wir uns auf die Kleinbuchstaben a-z.

Aufgabe 9

Matrixdruck:

Bei einem Matrixdrucker werden die Zeichen des Zeichensatzes in einer Punktmatrix abgelegt und dann Punkt für Punkt ausgegeben. Das folgende Beispiel zeigt die Ziffer '3' in einer Punktmatrix mit 5 Zeilen und 4 Spalte:

```
1 ***
2    *
3  **
4    *
5 ***
```

Schreiben Sie mit Hilfe eines 3 dim. Arrays ein Programm, das eine vom Benutzer eingegebene mehrstellige Zahl wie ein Matrixdrucker auf dem Bildschirm ausgibt.
(dots[n][z][s], n = darzustellende Ziffer (0-9), z = Zeile in der Matrixdarstellung (0-4), s = Spalte in der Matrixdarstellung (0-3))

Zeiger und Adressen

Ein paar Definitionen:

Zeiger (engl. Pointer): Variable, in der die Adresse einer anderen Variablen gespeichert ist.

Dereferenzierung: Über einen Zeiger kann auf die Daten der referenzierten Variablen zugegriffen werden.

Dereferenzierungsoperator *: Zum Zugriff auf die referenzierte Variable stellt man der Zeigervariablen den Dereferenzierungsoperator * voran.

Zeiger und Adressen

Über den Adressoperator wird die Verbindung zwischen dem Zeiger und der zu referenzierenden Variablen hergestellt.

```
1 int x;
2 int *pointer;
3
4 pointer = &x;
5
6 *pointer = 1;
7 *pointer = *pointer + 1;
```

Zeiger und Adressen

```
1  void tausche( int *a, int *b)
2  {
3     int t;
4     t = *a;
5     *a = *b;
6     *b = t;
7  }
8
9  void main()
10 {
11    int x(1), y(2);
12    cout << string("Vorher: %d %d", x, y) << endl;
13    tausche( &x, &y);
14    cout << string("Nachher: %d %d", x, y) << endl;
15 }
```

Zeigerarithmetik

Was Passiert wenn wir den Wert eines Zeigers ändern?

```
1 void main()
2 {
3   int *p = 0;
4
5   cout << string("p + 0 = %d", p + 0) << endl;
6   cout << string("p + 0 = %d", p + 1) << endl;
7   cout << string("p + 0 = %d", p + 2) << endl;
8   cout << string("p + 0 = %d", p + 3) << endl;
9 }
```

Zeigerarithmetik

In diesem Programm legen wir einen Zeiger p an und initialisieren ihn mit dem Wert 0. Dann geben wir die Adresswerte von p, p+1, p+2 und p+3 als Integer-Werte aus:

```
1 p + 0 = 0;
2 p + 1 = 4;
3 p + 2 = 8;
4 p + 3 = 12;
```

Arrays und Zeiger

Zusammenhang zwischen Arrays und Zeigern:

array⟨int⟩ a[8]
a +0 → a[0]
+1 → a[1]
+2 → a[2]
+3 → a[3]
+4 → a[4]
+5 → a[5]
+6 → a[6]
+7 → a[7]

a Array

$\Longleftrightarrow$

a ist Zeiger auf 1. Element des Arrays

a = &a[0]

a + i = &a[i]

$\Longrightarrow$ a[i] und *(a+i) sind gleichwertig

Arrays und Zeiger

Gleichwertige Schleifendurchläufe:

```
1  int A[5];
2  int i;
3  int *p;
4
5  for(i=0; i<5; i++)
6    A[i] = 0;
7
8  for(i=0; i<5; i++)
9    *(A+i) = 0;
10
11 for(i=0, p = A; i<5; i++, p++)
12   *p = 0;
```

Referenzen

Betrachte erneut:

```
1  void tausche_pointer( int *a, int *b){
2    int t;
3    t = *a;   *a = *b;   *b = t;
4  }
5
6  void main(){
7    int x(1), y(2);
8    cout << string("Vorher: %d %d", x, y) << endl;
9    tausche_pointer( &x, &y);
10   cout << string("Nachher: %d %d", x, y) << endl;
11 }
```

$\Rightarrow$ Zeiger werden immer direkt dereferenziert!

Referenzen

Eine **Referenz** ist ein konstanter Zeiger, der bei jeder Verwendung automatisch dereferenziert wird.

```
1  void tausche_referenz( int& a, int& b)
2  {
3    int t;
4    t = a;   a = b;   b = t;
5  }
6
7  void main()
8  {
9    int x(1), y(2);
10   cout << string("Vorher: %d %d", x, y) << endl;
11   tausche_referenz( x, y);
12   cout << string("Nachher: %d %d", x, y) << endl;
13 }
```

Referenzen

Referenzen können auch als Variablen in einem Programm benutzt werden. Sie stellen sozusagen Aliasnamen für andere Variablen dar:

```
1 int i;
2 int& ref = i;
3
4 i = 1;
5 cout << i << " " << ref << endl;
6 i++;
7 cout << i << " " << ref << endl;
8 ref++;
9 cout << i << " " << ref << endl;
```

Aufbau von Klassen

```
1  class point                    // Name der Klasse
2  {
3                                 // ausserhalb
4                                 // gehoeren zum privaten Bereich!
5
6    private:                     // privater Bereich
7                                     Hier stehen die privaten Member
8
9
10   public:                      // oeffentlicher Bereich
11                                    Hier stehen die oeffentlichen Member
12
13 };
```

Klassen können Daten und Funktionen als Elemente erhalten!

Aufbau von Klassen

```
1 class point
2 {
3   public:
4     double x;
5     double y;
6 };
```

Auf die Member kann durch den "."-Operator zugegriffen werden.

Aufbau von Klassen

```
1 class point
2 {
3   public:
4     double x;
5     double y;
6 };
```

Auf die Member kann durch den "."-Operator zugegriffen werden.

So kann jeder Benutzer die Daten beliebig ändern und z.B. einen ungewollten negativen Wert als Koordinate zuweisen.

Aufbau von Klassen

```
1 class point
2 {
3   private:
4     double x;
5     double y;
6 };
```

Aufbau von Klassen

```
1 class point
2 {
3   private:
4     double x;
5     double y;
6 };
```

Ein Zugriffsversuch wei z.B. $p.x = 0$ wird vom Compiler als ungültig zurückgewiesen.

Aufbau von Klassen

Lösung des Problems durch Funktions-Member!

Funktions-Member einer Klasse haben uneingeschränkten Zugriff auf alle Daten ihrer Klasse!

```
1  class point
2  {
3    private:
4      double x;
5      double y;
6
7    public:
8      double get_x() { return x;}
9      double get_y() { return y;}
10     void set( double xx, double yy) { x = xx; y = yy; }
11 };
```

In der Funktion können wir nun z.B. zuerst testen ob die Zahl positiv ist und nur dann den Wert ändern!

Funktions-Member

Implementierung außerhalb der Klasse: Die Member-Funktion muss wie ein Funktionsprototyp in der Klasse deklariert werden und wird dann ausserhalb der Klasse implementiert.

```
1  class point
2  {
3    private:
4      double x;
5      double y;
6
7    public:
8      point() { x = 0; y = 0; }
9      double get_x() { return x;}
10     double get_y() { return y;}
11     void set( double xx, double yy) { x = xx; y = yy; }
12     void zeichne();
13     ~point();
14 };
15
16 void punkt::zeichne()
17 {
18   // zeichne Punkt
19 }
```

Konstruktor und Destruktor

```
1  class point
2  {
3    private:
4      double x;
5      double y;
6
7    public:
8      point() { x = 0; y = 0; }
9      point( double xx, double yy ) { set( xx, yy); }
10     double get_x() { return x;}
11     double get_y() { return y;}
12     void set( double xx, double yy) { x = xx; y = yy; }
13     ~point();
14 };
```

Konstruktor und Destruktor

Ein Konstruktor...

- ...ist eine Member-Funktion einer Klasse.
- ...kann parameterlos sein oder Parameter unterschiedlicher Art und Anzahl haben.
- ...hat keinen Returntyp!
- ...muss nicht existieren.
- ...muss sich von anderen Konstruktoren der gleichen Klasse durch die Parametersignatur unterscheiden.
- ...wird immer dann aufgerufen, wenn ein Objekt dieser Klasse instantiiert wird.

Konstruktor und Destruktor

Membervariablen können direkt initialisiert werden:

```
1  class point
2  {
3    private:
4      double x;
5      double y;
6
7    public:
8      point() { x = 0; y = 0; }
9      point( double xx, double yy ) : x(xx), y(yy) { }
10     double get_x() { return x;}
11     double get_y() { return y;}
12     void set( double xx, double yy) { x = xx; y = yy; }
13     ~point();
14 };
```

Konstruktor und Destruktor

Ein Destruktor...

- ...kann innerhalb oder außerhalb der Klasse implementiert werden.
- ...wird nach der Destruktion des umschließenden Objekts aufgerufen.
- ...gibt gegebenfalls Speicher frei und räumt auf.

Dynamische Datenstrukturen

In C++ können Variablen

- automatisch,
- statisch oder
- dynamisch

angelegt werden.

Dynamische Datenstrukturen

Neu: dynamische Variablen

- werden während der Laufzeit angelegt
- leben bis zum Programmende oder bis die Variable explizit gelöscht wird.
- Für den Zugriff auf dynamische Datenstrukturen verwenden wir Zeiger.

Dynamische Datenstrukturen

Datenstrukturen, die erst zur Laufzeit vom Programm nach Bedarf auf- bzw. abgebaut werden.

```
1  class point
2  {
3    private:
4      double x;
5      double y;
6    public:
7      point() { x = 0; y = 0; }
8      point( double xx, double yy ) : x(xx), y(yy) { }
9      void set( double xx, double yy) { x = xx; y = yy; }
10     ...
11 };
12
13 // lege Zeiger zum Zugriff auf die Variable an
14 point *p;
15 // new allokiert den benötigten Speicher und instantiiert die Variable
16 p = new point(10,12);
17
18 // Zugriff auf Member eines Pointers
19 p->set(0,0);
20 ...
21 delete(p);
```

Instantiierung von Arrays

Um ein Array von Objekten einer Klasse instantiieren zu können, muss die Klasse einen Default-Konstruktor (oder einen selbst erstellten Konstruktor ohne Parameter) besitzen!

```
1 point p_array[10]; // statisch
2
3 point *p_array; // dynamisch
4 p_array = new point[10];
5 ... Verwendung des Arrays ...
6 delete[] p_array;
```

Friends

Man kann Klassen Privilegien beim Zugriff auf andere Klassen einräumen.
Beispiel:

```
1 class point
2 {
3    friend class rechteck;
4    private:
5       ...
6    public:
7       ...
8 };
```

Auf diese Weise können alle Member-Funktionen von rechteck auf alle - also auch die privaten - Daten der Klasse point zugreifen.

Friends

Friend-Funktionen verwendet man gerne, wenn man Funktionen erstellen will, die auf mehreren Objekten u.U. verschiedener Klassen arbeiten.

Beispiel: Berechne das Maximum von zwei Punkten.

Die Maximumsberechnung könnte man in einer Member-Funktion der Klasse point durchführen:

```
1 p3 = p1.max( p2);
```

Wünschenswert wäre:

```
1 p3 = max(p1, p2);
```

Friends

Dies ist möglich, in dem wir der Funktion den Status eines Friends der Klasse point einräumen:

```
1 class point
2 {
3   private:
4     int x;
5     int y;
6   public:
7   friend point max( point p1, point p2);
8 };
```

Friends

Dann erstellen wir eine kleine Hilfsfunktion und implementieren die Funktion max:

```
1 point max( point p1, point p2)
2 {
3   if( p1.x > p2.x || (p1.x == p2.x && p1.y > p2.y) )
4     return p1;
5   else
6     return p2;
7 }
```

Operatoren auf Klassen

Operatoren auf Klassen können genau wie Member-Funktionen angelegt werden.
Beispiel:

```
1 class point
2 {
3   private:
4     int x;
5     int y;
6   public:
7     bool operator==(point& p);
8 };
```

Includes

Mit einer Include-Direktive können komplette Dateien vor der Übersetzung in den Programmcode eingefügt werden.

```
1 #include <stdio.h>
2 #include <LEDA/window.h>
3 #include "xyz.h"
4 ...
```

Symbolische Konstanten

Durch symbolische Konstanten werden Werte, die an unterschiedlichen Stellen im Code einheitlich verwendet werden sollen, an zentraler Stelle definiert und gepflegt.

```
1 #define PI 3.14
2 #define MAXIMUM 10
3 #define ZWEI_PI (PI + PI)
```

Symbolische Konstanten sind keine Variablen, sondern nur Platzhalter für einen Ersatztext!

Makros

Minifunktion ohne zusätzliche Laufzeitkosten für den Funktionsaufruf zu verursachen.

```
1 #define PI 3.14
2
3 #define KREIS_FLAECHE(r) (PI*(r)*(r))
4
5 #define MAX(a,b) (a) > (b) ? (a) : (b)
```

Funktion:
KREIS_FLAECHE(a++)

Makro:
KREIS_FLAECHE(a++) -> (PI*(a++)*(a++))

Aufgabe 10

Klasse Zeit:

Entwerfen und implementieren Sie eine Klasse zeit! Diese Klasse soll aus den int-Variablen stunde und minute bestehen und die folgenden Funktionalitäten bieten:

- ein Standard-Konstruktor soll vorhanden sein
- ein Parameter-Konstruktor mit 2 Parametern soll vorhanden sein, der 1. für die Stunde, der 2. für die Minuten. Dieser soll die Zeitangabe normieren, so dass die Minuten < 60 sind.
- Setzen einer bestimmten Zeit
- Ausgabe der Zeit
- Addition zweiter Zeiten
- Bestimmung der Differenz zweier Zeiten

Schreiben Sie ein Testprogramm, das alle Funktionen dieser Klasse intensiv testet!

Aufgabe 11

Listen:

Implementieren Sie eine Klasse int_list mit folgenden Funktionen:

- **int_item L.first()** returns the first item of L (nil if L is empty)
- **int_item L.last()** returns the last item of L (nil if L is empty)
- **int_item L.get_item(int i)** returns the item at position i (the first position is 0) Precondition $0 <= i <$ L.length().
- **int_item L.succ(int_item it)** returns the successor item of item it, nil if it = L.last(). Precondition it is an item in L.
- **int_item L.pred(int_item it)** returns the predecessor item of item it, nil if it = L.first()
- **const int& L.contents(int_item it)** returns the contents L[it] of item it. Precondition it is an item in L.
- **int_item L.push(const int& x)** adds a new item $< x >$ at the front of L and returns it
- **int_item L.append(const int& x)** appends a new item $< x >$ to L and returns it

Aufgabe 11

Listen:

und noch mehr Funktionen:

- **int L.pop()** deletes the first item from L and returns its contents
- **int L.del_item(int_item it)** deletes the item it from L and returns its contents L[it]
- **void L.move_to_front(int_item it)** moves it to the front end of L
- **void L.move_to_rear(int_item it)** moves it to the rear end of L
- **void L.conc(list<int>& L1)** appends list L1 to list L and makes L1 the empty list. Precondition: L! = L1
- **void L.split(int_item it, list<int>& L1, list<int>& L2)** splits L at item it into lists L1 and L2. More precisely, if it! = nil and L = x1,..., xk-1, it, xk+1,..., xn then L1 = x1,..., xk-1 and L2 = it, xk+1,..., xn. If it = nil then L1 is made empty and L2 a copy of L. Finally L is made empty if it is not identical to L1 or L2.