SELEKTION

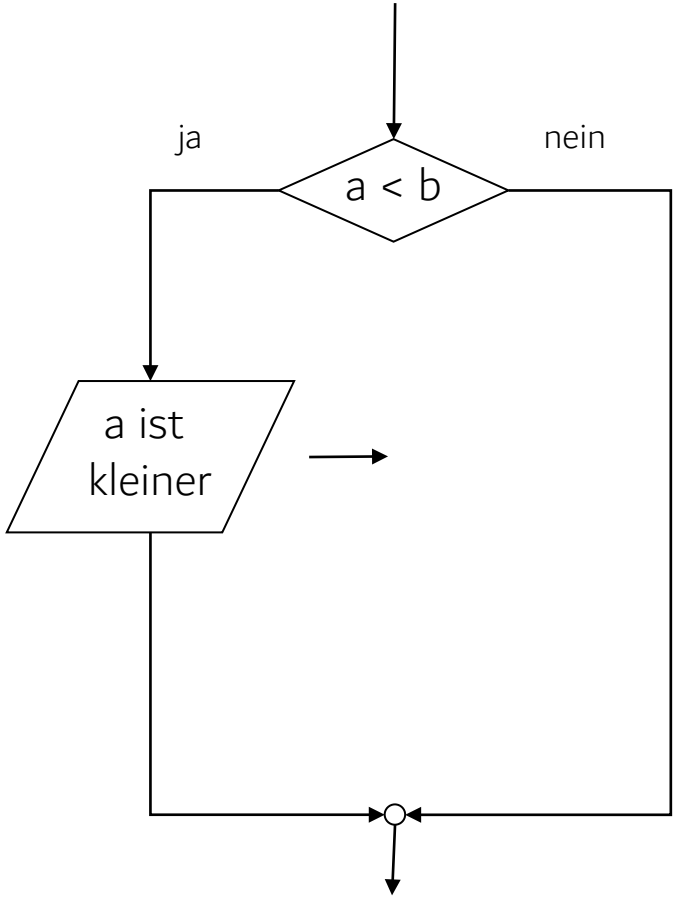
Dozent: Dr. Andreas Jäger

Überblick

- Selektion (Verzweigung)
  - > if …
  - > if … else …

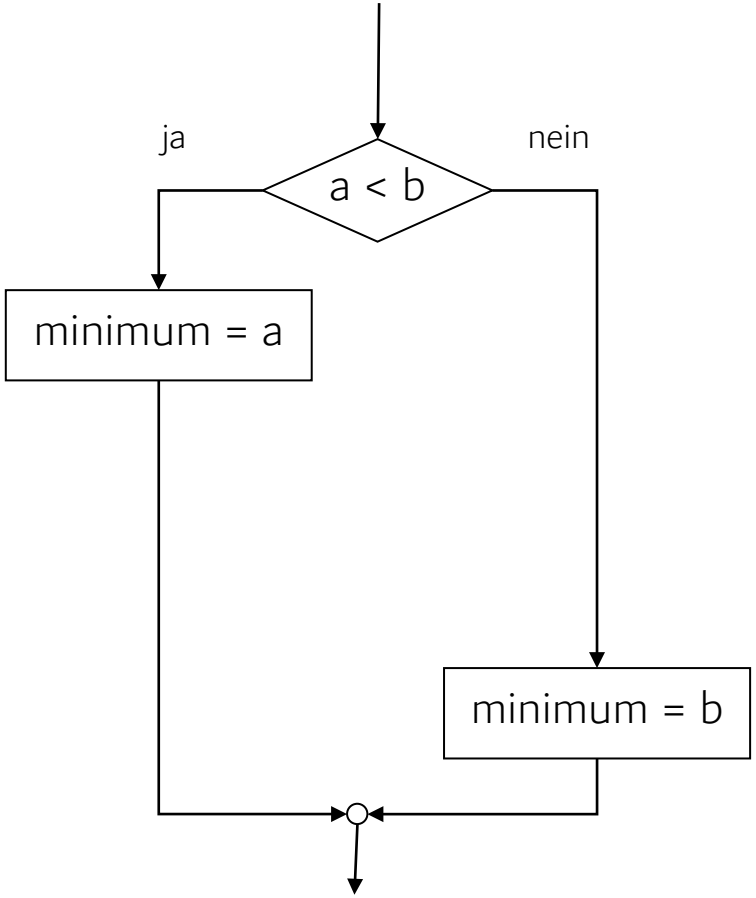
- *„Konstanten“*
  - > Literalkonstanten
  - > **#defines**

Selektion




```
if ( a < b )
{
 cout << "a ist kleiner";
}
```

Selektion




```
if ( a < b )
{
     minimum = a;
}
else
{
    minimum = b;
}
```

If – else mit einer Anweisung

- Bei einer Anweisung können Blockklammern entfallen:

```
if ( semester > regelstudienzeit )
  cout << "Beeilung!!!";
else
  cout << "Don't panic.";
```

- Empfehlung:
  Trotzdem { } verwenden!

KONSTANTEN

Dozent: Dr. Andreas Jäger

konstanten

- Literalkonstanten
- Alle Zahlen und Zeichen, die als Text dargestellt sind:
  - >1
  - >5
  - >-99
  - >‘c‘
  - >‘9‘

konstanten

- Durch Preprozessor ersetzte „Namen“
  > #define PI 3.141592653589793
  > #define KONSTANTE 100

- Kein Semikolon (;)
  da Preprozessor einfach die entsprechende Textstelle durch den Eintrag ersetzt → und ein Semikolon wäre dann meist fehl am Platz.

konstanten “Variablen”

- Konstante Werte, die nicht mehr verändert werden können:

  const double pi = 3.141592653589793;

- Siehe Variablendeklaration

  > zusätzliches Schlüsselwort **const**

  > gleichzeitige Zuweisung

  → konstanter Wert in einer Variable.

ITERATION

Dozent: Dr. Andreas Jäger

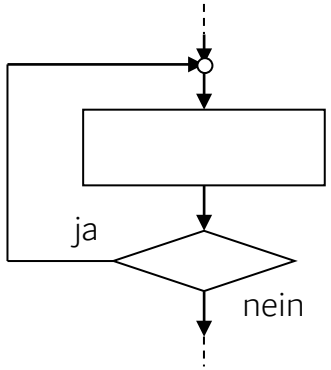
Übersicht

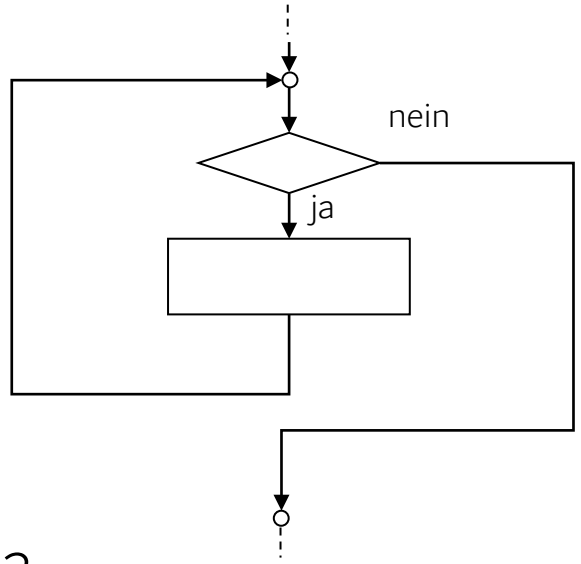
- Die Wiederholung

- Iteration (Schleife)
  - > Kopfgesteuert
  - > Fußgesteuert

- Umsetzung in der Sprache C

Schleifen





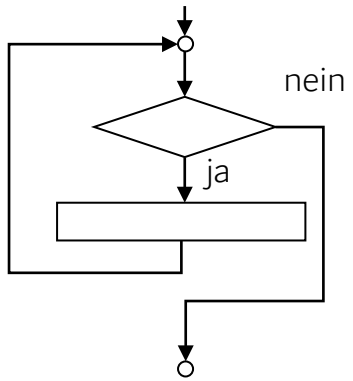
Unterschied?

fußgesteuert

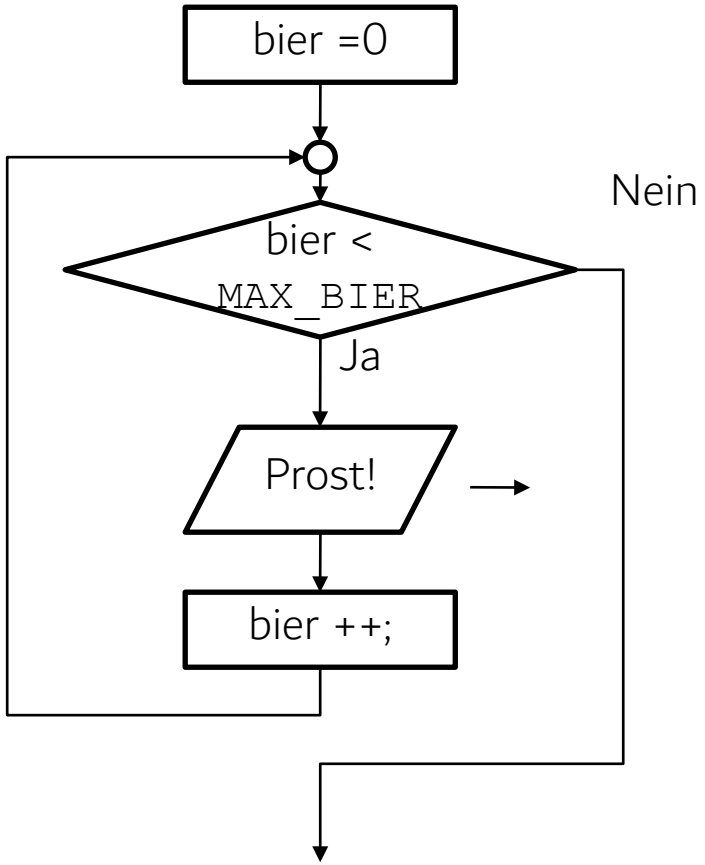
kopfgesteuert

Kopfgesteuert

- Kopfgesteuerte Schleife
  1. Bedingung zu Beginn der Schleife prüfen
  2. Trifft Bedingung nicht zu springe zu Punkt 5
  3. Führe Aktionen aus
  4. Gehe zurück zu Punkt 1.
  5. Weiter…



Kopfgesteuert




```
#define MAX_BIER 5
unsigned long bier = 0;

while ( bier < MAX_BIER )
{

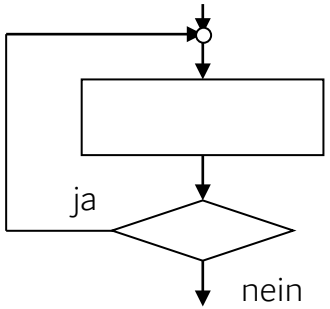
   cout << "Prost!";

   bier++;
}
```

Fußgesteuert

Fußgesteuerte Schleife

1. Führe Aktionen aus
2. Prüfe Schleifenbedingung
3. Trifft Bedingung zu, gehe zurück zu Punkt 1.
4. Weiter…



Zählschleife

- Verwendet als Bedingung Zählvariable
- Zählweise festlegbar
- Sonderfall der kopfgesteuerten Schleife

- **Beispiel:**

```
for (int i=0; i<10; i++)
{
   cout << i << ", ";
}
```

Blockklammern

- Ebenso wie bei der Verzweigung können die Blockklammern weggelassen werden, wenn der Block aus nur einer Anweisung besteht!

```
while ( b > a )
  a = funktion( b );
```

Blockklammern

```
int nMax = 10;
int nMin = 5;
cout << "Ich zähle von "<< nMin
     << " bis " << nMax;

for (int i=nMin; i<=nMax; i++)
   cout << i ;
```

Warum 3 verschiedene?

- Warum gibt es 3 Arten von Schleifen?
  - > Standardfall
    - > Schleife n mal durchführen
  - > Weitere Fälle
    - > Schleife solange durchführen bis etwas anders ist.
    - > Schleife mindestens einmal durchführen und dann solange bis eine Bedingung nicht mehr erfüllt ist.

n-Mal was machen

```
int i;
int n = 5;

for (i=0; i<n; i++)
{
   // Aktion
}
```

Führe durch solange bis

```
bool bFlag;
// in Ansi-C int verwenden!
do
{

   bFlag = false;
   // in Ansi-C: bFlag=0
}
while ( bFlag );
```

Führe durch wenn - solange bis

```
bool bFlag = true;
   // Ansi-C: int bFlag=1;

while ( bFlag )
{
   bFlag = false;    // bFlag = 0;
}
```

Endlosschleife

- ```
  int i,a=9;
  for (i=0; a<10; i++)
  {
     //gefangen in einer Endlosschleife
  }
  ```

- **Fehler**: Falsche Abbruchbedingung
  ```

Endlosschleife

- Na klar kann es Probleme geben.
  > Endlosschleifen / endless loop
- Wie?
  > Durch falsche Abbruchbedingungen und/oder
  > Abbruchbedingung wird gar nicht erst verändert

ARRAYS

Dozent: Dr. Andreas Jäger

Arrays (C++)

Ein Array ist eine Serie von Elementen des selben Datentypes.

Dabei gilt, dass die Datenelemente kontinuierlich im Speicher hinterlegt werden.

Dies bedeutet, dass beispielsweise 10 Werte vom Typ int als Array deklariert werden können,

ohne dass 10 verschiedene Variablen deklariert werden müssen.

Arrays (C++)

Mithilfe eines Arrays kann mit demselben Bezeichner und dem richtigen Index somit auf 10 (n) verschiedene Werte zugegriffen werden.

Syntax Deklaration:

<type> <name> [<elements>];

Syntax Deklaration + definition:

<type> <name> [<elements>] = {<Wert1>,...,<WertN>};

Arrays

- *In C/C++:* definierte Anzahl von Elementen des gleichen Datentyps.
- <*typ*> <*variable*>[<*Anzahl*>];
- Beispiel:

```
unsigned char nLottozahlen[6];
float nMesswerte[25];
```

- Indizierung von 0 bis Anzahl-1
    1. Element ist 0: `nLottozahlen[0] = 7;`

Arrays (C++)

Beispiel:

```
int arr[5];

int arr2[5] = { 16, 2, 77, 40, 12071 };
```

Arrays (C++)

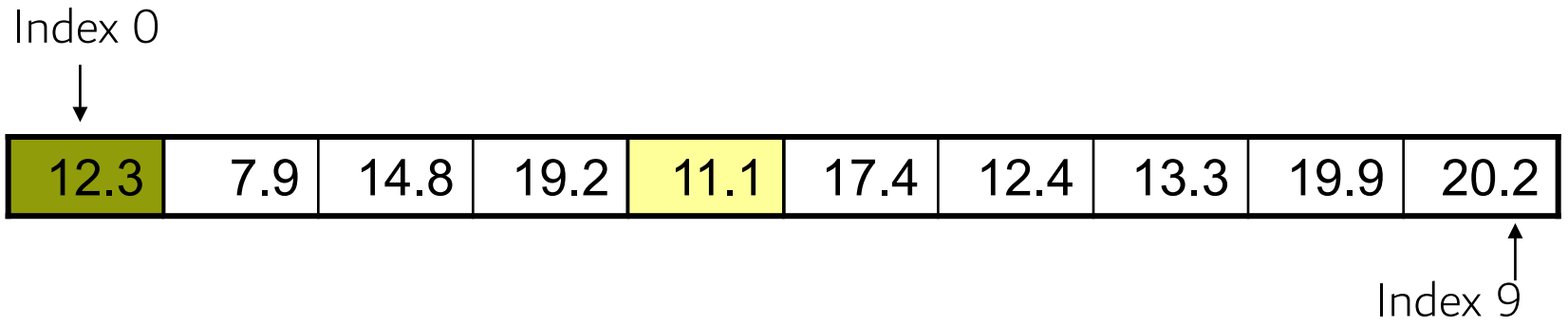
Zugriff auf die Werte eines Arrays:

Auf die Werte aller Elemente in einem Array kann genauso zugegriffen werden wie auf den Wert einer regulären Variablen desselben Typs.

Die Syntax lautet:

<name> [<index>]

Array Zugriff




```
printf( "Die 1. Messung ergibt %lf m.\n", dMesswerte[0] );
```

Die 1. Messung ergibt 12.3 m.

```
printf( "5. Messung = %lf m.\n", dMesswerte[4] );
```

5. Messung = 11.1 m.

Arrays (C++)

Beispiel:

```
Std::cout<<arr[2]<<std::endl;
```

```
>> 2
```

Übung

Schreibt ein Programm, das die Zahlen von 1 bis 99 in einen Array schreibt.

Gebt alle Elemente des Arrays aus.

Übung

a) Recherchiert: Was sind die Fibonacci-Zahlen.

b) Erstellt einen PAP Plan.

c) Generiert die ersten 10 Zahlen mit Hilfe eines Arrays.

d) Info: Die ersten 2 Zahlen der Folge dürfen im Code direkt gesetzt werden. Der Rest muss vom Algorithmus generiert werden.

Dynamische Arrays

- Funktioniert über den Zeiger auf die 1. Speicherstelle
- Reservieren von **n** Elementen:

```
long n;
long* messwerte;      // liste mit Messwerten
cin >> n;             // Eingeben der Anzahl Werte

messwerte = (long*)malloc( n * sizeof(long) );
for (int i=0; i<n; i++)
  messwerte[i] = (i+1.5) * 2.3 + i*i * 1.2;

// n-ten Messwert auslesen
cout << messwerte[ n-1 ];
```

Zeichenketten

- Zeichenketten (strings)
  > Besteht aus einer Anzahl von *char*acters.

  → Char-Array fester Länge

```
char szText[500];
```

  → Dynamische Länge

```
char* pszText =(char*)malloc(50*sizeof(char));
```

Zeichenkette

- Interner Aufbau der Zeichenkette

```
char szText[17];
strcpy( szText, "HELLO WORLD" );
```

Index 0

Character-Array

Nullterminierte Zeichenkette ('\0' = 0)

```
printf( "Länge Text = %n", strlen(szText) );
```

Länge Text = 11

strcpy

Kopiert Zeichen einer Zeichenkette a in die Zeichenkette b

Einfacher Algorithmus

> Für alle Zeichen an Stelle i:
  Beginne bei 0 und setze $b_i = a_i$
  solange a_i nicht Ende der Zeichenkette (a_i!='\0').

Signatur:

```
> strcpy(char* b, const char* a);
```

STRUKTUREN

Dozent: Dr. Andreas Jäger

Strukturen

- Strukturen fassen Datenelemente zusammen

- Datentypen bisher
  - **char**, **short**, **int**, **long**, **double**, ...

- Zusammensetzen von Elementen
  - Erlaubt es komplexere Typen zu generieren
  - Gruppieren von Informationen
  - Zugriffserleichterung

Strukturen

z.B. Vektor mit 3 Elementen (x,y und z)

```
struct vector3d {
    double x;
    double y;
    double z;
};
```

Deklaration

- Außerhalb von Funktionen!
  (am besten nach den #include statements)

```
#include <iostream>
struct vector3d { double x,y,z; };


int main()
{
    // [...]
    return 0;
}
```

```
struct vector3d { double x, y, z; };
```

- Zugriff auf die Attribute

  → Mittels Punktnotation

```
vector3d p;
p.x = 10.1;
p.y = 20.2;
p.z = -17.7;
```

Falls mit einem Zeiger gearbeitet wird (vector3d* pVec), kann der Zugriff auf die Attribute mittels Pfeilnotation - erfolgen:

original: (*pVec).x;

neu: *pVec->x*

Struct erzeugen

Speicher kann auch dynamisch reserviert werden:

```
struct vector3d { double x, y, z; }
// [...]
vector3d* pVec;
pVec = (vector3d*)malloc( sizeof(vector3d) );
if (pVec!=0) {
    pVec->x = 0; pVec->y = 20; pVec->z = -2;
    free( pVec );
}
```

ENUM

Dozent: Dr. Andreas Jäger

C Datatype enum

```
// manchmal  ist es nützlich den Status mithilfe von Attributen zu speichern
// denkbar wäre:
// const int flying = 1;
// const int moving = 2;
// const int resting = 3;
// Nachteil: range der Werte nicht festgelegt
//           Benennung über den Variablennamen
// C bietet hier enums

enum state{ flying, moving, resting };
//identisch zu
enum state { flying = 0, moving = 1, resting = 2 };

state r = flying;
switch (r) {
  case flying:
    std::cout << "flying\n";
    break;
  case moving:
    std::cout << "moving\n";
    break;
  case resting:
    std::cout << "resting\n";
    break;}
```

POINTER

Pointer

- Wir wissen Variablen werden im Speicher angelegt.
- Der „identifier“ (name) erlaubt es uns auf die Variable zuzugreifen.

→Vorteil: Man muss nicht wissen wo was im Speicher hinterlegt wurde.

Wenn eine Variable deklariert wird, wird ein bestimmter Speicherbereich für das Speichern des eigentlichen Wertes reserviert.

Pointer

- Der Compiler behandelt den Speicher hierbei wie eine Ansammlung von Speicherzellen. Diese Zellen haben eine fortlaufende Nummerierung.

- Die Adresse (Start der Zelle) von einer deklarierten Variablen können wir uns in C++ geben lassen.

- Der Operator der hierfür verwendet wird nennt sich der Adressoperator

Address-of operator „&“

Pointer

- Es gibt zudem eine spezielle Variable, die dazu dient, die Adresse einer anderen Variablen zu speichern.
- Dieser Typ von Variable nennt sich Pointer.

Pointer

Deklaration eines Pointers:

```
// <type> * <pointer name>
int* pInt;
```

Pointer

Deklaration + Definition eines Pointers:

```
int i = 4;
int* pInt = &i;
```

Übung

versucht mal *pInt* wie folgt auszugeben. Was fällt auf.

```
int i = 4;
int* pInt = &i;
std::cout << pInt << std::endl;
```

Pointer – Dereferenzierungsoperators

Wie wir gerade herausgefunden haben, enthält der Pointer ja lediglich eine Adresse

Normalerweise intressiert uns aber der Wert von dem worauf er zeigt.

Hierfür gibt es den den Dereferenzierungsoperator:

Dereference operator „*“

Pointer

Verwendung des Dereferenzierungsoperators

```
int i = 4;
int* pInt = &i;
std::cout << *pInt << std::endl;
```

Übung was ist const?

Tip – von rechts nach links lesen:

int*

int const *

int * const

int const * const

Pointer – Reihenfolge beachten

Tip – von rechts nach links lesen:

int* - pointer to int

int const * - pointer to const int

int * const - const pointer to int

int const * const - const pointer to const int

Verwirrend – das erste const darf rechts oder links stehen

const int * == int const *

const int * const == int const * const

Übung was ist const?

int ** -

int ** const -

int * const * -

int const ** -

int * const * const -

Pointer – Reihenfolge beachten

int ** - pointer to pointer to int

int ** const - a const pointer to a pointer to an int

int * const * - a pointer to a const pointer to an int

int const ** - a pointer to a pointer to a const int

int * const * const - a const pointer to a const pointer to an int

Übung

Schreibt ein Programm, das vom Benutzer 2 Integer eingaben verlangt.

Legt 2 pointer an, die auf diese Eingaben zeigen und gebt den Inhalt der Pointer aus.

Übung

Analysiert den folgenden Programmcode. Erklärt die Ausgabe.

```
int main(void) {
char str[] = "Hello Pointer";
char* pc = str;

std::cout << str[0] << ' ' << *pc << ' ' << pc[3] << "\n";
pc += 2;
std::cout << *pc << ' ' << pc[2] << ' ' << pc[5];

return 0;
}
```

Dynamisches allokieren auf dem Heap

In C wird der dynamische Speicher mithilfe einiger Standardbibliotheksfunktionen aus dem Heap zugewiesen.

Die beiden wichtigsten dynamischen Speicherfunktionen sind malloc () und free ().

Dynamisches allokieren auf dem Heap

Die Funktion malloc () verwendet einen einzelnen Parameter (die Größe des angeforderten Speicherbereichs in Byte.)

Es gibt einen pointer auf den zugewiesenen Speicher zurück.
Wenn die Zuordnung fehlschlägt, wird NULL zurückgegeben.

Dynamisches allokieren auf dem Heap

Die Standardbibliotheksfunktion lautet wie folgt:

```
void * malloc (size_t size);
```

Die Funktion free () verwendet den von malloc () zurückgegebenen Zeiger und hebt die Zuordnung des Speichers auf.

Es wird kein Hinweis auf Erfolg oder Misserfolg zurückgegeben. Der Funktionsprototyp sieht folgendermaßen aus:

```
void free (void * Zeiger);
```

Dynamisches allokieren auf dem Heap

Ich empfehle euch weder malloc noch free zu verwenden!

Dynamischer Speicher in C ++

C ++ verfügt über zwei zusätzliche Operatoren

new

delete

mit denen Code klarer, prägnanter und flexibler mit geringerer Fehlerwahrscheinlichkeit geschrieben werden kann.

Dynamischer Speicher in C ++

Der neue Operator kann auf drei Arten verwendet werden:

```
p_var = new Type;
p_var = new Type(<init>);
p_array = new Type [<size>];
```

In den ersten beiden Fällen wird Platz für ein einzelnes Objekt zugewiesen.

Die zweite beinhaltet die Initialisierung.

Der dritte Fall ist der Mechanismus zum Zuweisen von Speicherplatz für ein Array von Objekten.

Dynamischer Speicher in C ++

Der Löschoperator kann auf zwei Arten aufgerufen werden:

```
delete p_var;
```

```
delete [] p_array;
```

(Speicherreservierung wird als Allokation – Freigabe als Deallokation beschrieben)

Der erste Befehl ist für ein einzelnes Objekt.

Mit der zweiten Option wird der von einem Array verwendete Speicherplatz freigegeben.
Es ist sehr wichtig, jeweils den richtigen Deallokator zu verwenden.

Übung

Recherchiert, wie man Pointer dazu verwenden kann dynamische Arrays zu erstellen.

Erstellt einen dynamischen Array, der groß genug ist, den vom Benutzer eingegebenen Namen zu speichern

Dynamischer Speicher in C ++

Hier ist der Code zum dynamischen Zuweisen eines Arrays und zum Initialisieren des vierten Elements:

```
int * pI;

pI = new int [10];

pI [3] = 99;
```

Die Verwendung der Array-Zugriffsnotation ist natürlich. Die Aufhebung der Zuordnung erfolgt folgendermaßen:

```
Delete[] pI
```

Auch hier ist das Zuweisen von NULL zum Zeiger nach der Freigabe eine gute Programmierpraxis.

Referenzen

Eine Referenz ist ein alias.

M.a.W. ein anderer Name (identifier) für eine existierende Variable.

Man kann also sowohl mit dem Variablennamen sowie dem Refernznamen auf den selben Wert zugreifen.

Man kann auch sagen, man hat mehrere Identifier für dieselbe Variable.

References vs Pointers

Referenzen werden manchmal mit Pointer durcheinandergebracht.

Was unterscheidet die beiden typen:

1. Es gibt NULL pointer, aber keine Null-Referenzen. Eine Referenz zeigt also nie ins “leere”.
2. Referenzen sind fix einer Variablen zugeordnet. Pointers können jederzeit einem anderen Objekt zugewiesen werden.
3. Referenzen müssen somit bei der Deklaration zugewiesen werden (initialisiert).

Übung

Was ist die Ausgabe:

```
int i = 4;
int f = 5;

int& rInt = i;
std::cout << rInt << std::endl;

rInt = f;

std::cout << i << std::endl;
```

Übung

Was ist die Ausgabe:

```
int i = 4;
int f = 5;

int& rInt = i;
std::cout << rInt << std::endl; gibt 4 aus

rInt = f; überschreibt den Wert in i (4 mit 5)

std::cout << i << std::endl; gibt 5 aus
```

Übung

Schreibt eine Funktion und überladet diese auf 3 Arten.

Die Funktion soll 2 integer Zahlen addieren.

Die Funktion akzeptiert:

a) 2 Referenzen auf integer Zahlen (“**pass by reference**”)
b) 2 Pointer auf Integer Zahlen

VIELEN DANK
FÜR
IHRE AUFMERKSAMKEIT!