I might even go so far as to advise someone whose only goal is to learn C++ to start with Java. D. Eck, 2001

Entwurfsziele von Programmiersprachen

elementare / strukturierte Datentypen

Das Variablenkonzept von C / C++

Adressoperationen in C

Dynamischer Speicher / Datenstrukturen in C / C++

Grundelemente von C++

Entwurfsziele

C: „*C reflektiert die Struktur der heutigen Rechner, daher sind C-Programme üblicherweise effizient genug, so dass kein Zwang besteht, etwas in Assembler zu codieren.“* Kernighan, Richie 1977

C++: *„C++ wurde entworfen, um die Fähigkeiten von Simula bei der Organisation von Programmen mit der Geschwindigkeit und Flexibilität von C bei der Systemprogrammierung zu verbinden.“* Stroustrup, 1994

Java: „*Java is a general-purpose concurrent class-based object-oriented programming language, specifically designed to have as few implementation dependencies as possible.“* Gosling, Joy, Steele, 1996

Elementare Datentypen

C/C++:

ganzzahlig: `bool`[1], `char`, `short`, `int`[2], `long` (`signed` / `unsigned`)

Gleitkomma: `float`, `double`, `long double`

Adressdatentypen: Zeiger, Referenzen[1]

Bereich und Genauigkeit der Datentypen ist rechnerabhängig.

Java:

ganzzahlig: `byte`, `char`, `short`, `int`, `long`

Gleitkomma: `float`, `double`

logisch: `boolean`

Bereich und Genauigkeit der Datentypen ist rechnerunabhängig.

[1] gibt es nur in C++.

[2] Die Unterstreichung kennzeichnet die numerischen Grundtypen.

Strukturierte Datentypen

C: (Array)[1], `struct`

Die Deklaration von Array/struct-Variablen legt den Speicherplatz fest. Bei Arrays steht der Variablenname für die Adresse des 1. Elements, bei structs steht der Name für die gesamte Datenstruktur (z.B. bei Zuweisung, Parameterübergabe)

C++: (Array)[1], `struct` / `class`

wie C. struct/class können Methoden definieren und erlauben Vererbung. In C++ gibt es keine vordefinierte Klassenhierarchie,

Java: Array, `class`, `interface`

Arrays und Klassen beschreiben strukturierte Datentypen. Sie sind Teil der mit `Object` beginnenden Klassenhierarchie.

[1] nach der C Sprachdefinition wird ein Array nicht als eigener Datentyp betrachtet.

Das Variablenkonzept in C / C++

Java kennt Wertdatentypen (elementare Datentypen) und Referenzdatentypen (Arrays und Objekte)

In **C** und **C++** gibt es:

- normale Variable (Wertsemantik)
- Zeigervariable (beziehen sich auf Adressen von Daten)
- Zeigervariable (stehen für Arrays)
- Zeigervariable (beziehen sich auf dynamischen Speicher)
- Referenzvariable stellen Alias für andere Variable/Objekte dar (nur C++)

Speicherorganisation

Globale Variable	(C, C++)
Klassenvariable	(C++, Java)
Instanzvariable	(C++, Java)
lokale Variable	(C, C++, Java)

dynamischer Speicher

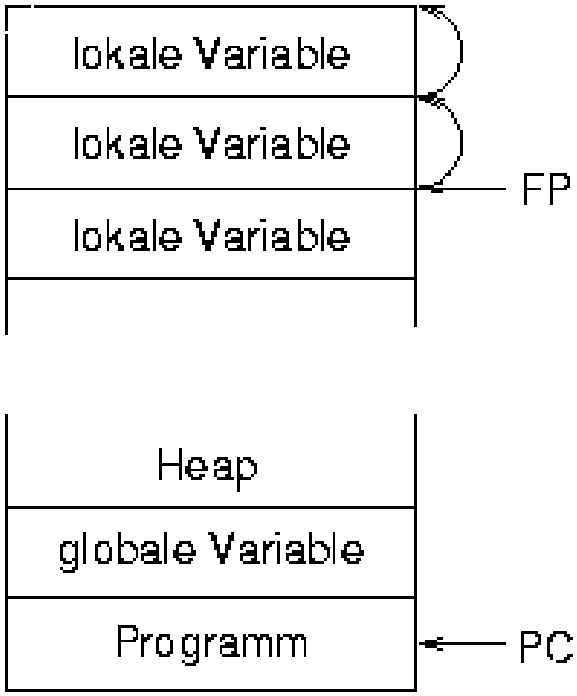
manuell:	(C, C++)
automatisch:	(Java)

In Java liegen alle Objekte im dynamischen Speicher.

In C/C++ ist alles möglich.

Das Unix-Speicherlayout

C-Programme werden von den Werkzeugen des Betriebssystems verwaltet. Das Unix-Layout hat sich allgemein durchgesetzt. Es berücksichtigt den linearen (virtuellen) Adressraum.



Das Java-Layout wird durch die virtuelle Maschine festgelegt. Die Grundzüge sind ähnlich zu C. Aber: Programmcode wird bei Java dynamisch geladen!

PC = program counter
= nächste Instruktion

FP = frame pointer
= Bereich der lokalen Variablen

Zeigeroperationen (nur für Profis)

Deklaration: *Typ* * p*var1*, * p*var2*;
Adressoperator: *&var*
Dereferenzierung: **pvar*
Adressarithmetik: *pvar + intVar*;
pvar - intVar;
pvar1 - pvar2;

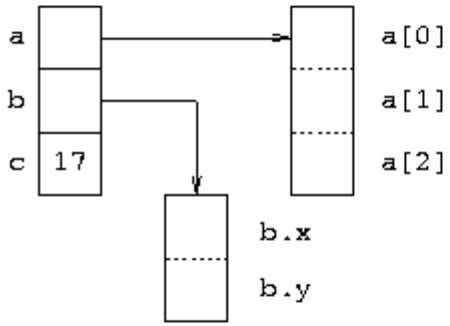
```
int var = 17;              /* Deklaration, Adresse, ,Dereferenzierung */
int* pvar = &var;
*pvar = 8;

pvar = pvar + 1;           /* Unsinnige Arithmetik */
*pvar = 18;

int a[20];                 /* Arrays sind Zeiger + Speicherreservierung */
int* p = a;
*(p+2) = 6;
a[2] = 6;
```

Frage: `int* p` oder `int *p` ?
Vgl.: `String[] a` und `String a[]`.

Wenn es geht, sollte man Zeiger vermeiden.

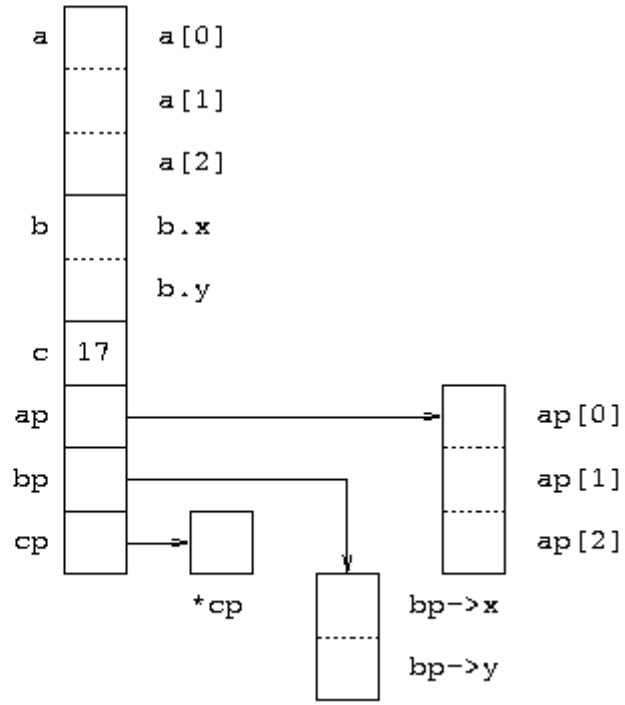
Java Variable

```
int a = new int[3];

Obj b = new Obj();

int c = 17;
```



C / C++ Variable

```
int a[3];

Obj b;

int c = 17;

int* ap = new int[3];

Obj* bp; = new Obj();

int* cp;
```



In C braucht man Zeiger

```
void swap(double* x, double* y) {
    double z = *x;
    *x = *y;
    *y = z;
}

double a = 7;
double b = 8;
double* c = &a;
swap(&b , c);    /* Wieso der Unterschied */


double arraySum(double* a, int n) {
    double s = 0.0;
    int i;
    for (i=0; i<n;i++) s += a[i];
    return s;
}

double recSum(double* a, int n) {
    if (n == 0)
         return 0.0;
    else
        return a[0] + recSum(a+1, n-1);
}
```

Dynamischer Speicher(1), Arrays

```
#include <stdlib.h>

int n;
int* a;

scanf("%d", &n);  /* &n, das hatten wir doch! */

a = (int*)malloc(n * sizeof(int));
a[0] = ...

free(a);
```

Definition:

reserviert n Byte und gibt die Adresse zurück:

```
void* malloc(unsigned int nBytes);
```

gibt einen Speicherbereich wieder frei:

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

Dynamische Datenstrukturen (2), Beispiel Liste

```
struct Node {
    char* value;
    struct Node*;
};

struct Node* first = NULL;
```

```
typedef struct Node_ {
    char* value;
    struct Node_*;
} Node;

Node* first = NULL;
```

NULL ist kein Sprachwort, sondern ein Macro, das so definiert ist:

```
#define NULL ((void*)0)   /* steht in stdlib.h ? */

int contains(struct Node* first, char* value) {
    struct Node* p = first;
    while (p != NULL && strcmp(value, p->value) != 0) p = p->next;
    return p != NULL;
}
```

Zugriff auf Komponente eines Structs: `(*p).value` ≡ `p->value`

Dynamische Datenstrukturen (3), Einfügen, Löschen

```
void addFirst(struct Node** first, char* value) {
    struct Node* n = (struct Node*)malloc(sizeof(Node));
    n->value = strdup(value);      // strdup legt dynamisch an
    n->next = *first;
    *first = n;
}

void remove(struct Node** first, char* value) {
    struct Node* q = NULL;
    struct Node* p = *first;
    while (p != NULL && strcmp(value, p->value) != 0) {
        q = p;
        p = p->next;
    }
    if (p != NULL) {
        if (q == NULL)
            *first = p->next;
        else
            q->next = p->next;
        free(p->value);            // nichts vergessen !!
        free(p);
    }
}
```

Probleme mit Zeigern:

- man benutzt undefinierte Zeiger
- man vergisst Speicher wieder freizugeben (Speicherleck)
- man benutzt Zeiger auf Objekte, die bereits freigegeben wurden
- man benutzt Zeiger auf lokale Objekte
- man überschreitet den alloziierten Speicherbereich

Konsequenzen:

- Fehler zeigen sich nur selten da wo sie gemacht wurden
- das Auftreten von Fehlern kann vom Betriebssystem abhängen
- das Auftreten von Fehlern kann von zufälligen Umständen abhängen
- Fehler können zu ganz merkwürdigen Effekten führen

Probleme mit Zeigern bekommt man nur mit sehr gutem Programmierstil in den Griff!

Fazit zu C

1. Bei der Systemprogrammierung kommt man an C nicht vorbei.

2. Ein C-Programmierer muss wissen was passiert.

3. Die Grundstruktur der Sprache ist einfach (bis primitiv).

4. C++ ist in vieler Hinsicht besser als C - aber auch komplizierter.

5. Java ist eine ganz andere Programmier-Klasse als C.

6. ... aber auch Java kommt nicht ohne C aus.

C++ ist eine Multiparadigma-Sprache

- Aufwärtskompatibel zu C
- Funktioniert nahtlos mit C zusammen
- Systemprogrammiersprache (verbessertes C)
- Elegante Erweiterung des Typsystems
- Klassenbasierte Sprache
- Generative Klassen (Template)
- Objektorientierung

„man soll nur für die Features zahlen, die man nutzt“

Einfache Verbesserungen von C++

- Referenzvariable und Referenzübergabe

```
void swap(double& a, double& b) {
    double t = a;
    a = b;
    b = t;
}

double x = 6.5;
double y = 7.8;
swap(x, y);
```

- verbessertes Typssystem (genauer als C, bool, const)
- Überladen von Funktionen (wie Java)
- beliebige Deklarationsreihenfolge (wie Java)
- Exceptions (ähnlich wie Java)
- Namensräume (entspricht teilweise dem Java Paketkonzept)

Überladen von Operatoren

```
Bruch operator+(const Bruch& a, const Bruch& b) { ... }

Bruch c = Bruch(1,2) + Bruch(3,4);
        // operator+(Bruch(1,2), Bruch(3,4));
```

Vereinfachte Ein-/Ausgabe basierend auf Operatorüberladen

```
#include <iostream>

int n;
cout << "Zahl eingeben: ";
cin >> n;
cout << n << " * 5: " << n * 5 << endl;
```

Dynamischer Speicher in C++

```
/** dynamisches Erzeugen von Objekten */
int* array = new int[n];
Bruch* b = new Bruch(3, 4);
Bruch* bArray = new Bruch[n]; // legt n-Bruch-Objekte an,
                              // erfordert Default-Konstruktor

/** das ist (fast) wie Java: */
Bruch** bpArray = new Bruch*[n];
bpArray[0] = new Bruch(7, 8);

array[0] = 7;
b->kuerzen();
bArray[0].kuerzen();
bpArray[0]->kuerzen();

/** unbenutzte Objekte müssen explizit freigegeben werden */
delete[] array;
delete b;
delete[] bArray;
delete bpArray[0];
delete[] bpArray;
```

Eine einfache C++-Klasse (1)

```
// SimpleCalc.h
#ifndef SIMPLE_CALC_H
#define SIMPLE_CALC_H

class SimpleCalc {
  // Statistikberechnungen
private:
    int count;     // Anzahl der eingegebenen Werte.
    double sum;    // Summe der eingegebenen Werte.
    double sqSum;  // Die Summe der Quadrate der Eingabewerte.

public:
    SimpleCalc();      // Konstruktor

    void enter(double dataItem);  // Eingabe der nächsten Zahl.

    int getCount();    // Anzahl der eingegebenen Zahlen.

    double getMean();  // bisheriger Mittelwert.

    double getSD();    // Standardabweichung.

    void clear();      // Loeschen des Speichers
};
// C++-Klassen enden mit ;
#endif
```

Eine einfache C++-Klasse(2)

```
// SimpleCalc.cc
#include "SimpleCalc.h"
#include "math.h"

SimpleCalc::SimpleCalc() {
    count = 0;
    sum = 0;
    sqSum = 0;
 }

void SimpleCalc::enter(double dataItem) {
    count++;
    sum += dataItem;
    sqSum += dataItem * dataItem;
}

...
```

Die cc-Datei definiert die Methoden. Dabei muss der Klassenname angegeben werden. (alternativ: inline-Funktionen)

Eine einfache C++-Klasse (3)

```
// anwendung.cc
#include <iostream>
#include "SimpleCalc.h"

using namespace std;

int main(int argc, char** argv) {
    cout << "Mittelwert von 3 Zahlen" << endl;
    SimpleCalc sc;
    // SimpleCalc* sc = new SimpleCalc();
    for (int i = 0; i < 3; i++) {
        double zahl;
        cout << (i+1) << ". Zahl: ";
        cin >> zahl;
        sc.enter(zahl);
        // sc->enter(zahl);
    }
    cout << "Mittelwert: " << sc.mean() << endl;
    // cout << "Mittelwert: " << sc->mean() << endl;
    // delete sc;
    return 0;
}
```

C++ kennt Vererbung und abstrakte Methoden / Klassen

```
#include <string>  // in C++ gibt es viele konkurrierende Bibliotheken

class Shape {                        // abstrakte Klasse
public:
    Shape(string& objName);
    int compareTo(Shape& other);   // das ist eine final-Methode
    virtual string getName();      // das ist eine normale Methode
    virtual double getArea() = 0;  // das ist eine abstrakte Methode
private:
    string name;
};

class Rectangle: public Shape {    // Ableitung
public:
    Rectangle(string& objName, double l, double b);
    virtual string getName();
    virtual double getArea();      // jetzt die Implementierung
private:
    double laenge, breite;
};
```

Polymorphie in C++

```
int Shape::compareTo(Shape& other) {
    double a = getArea();          // this->getArea()
    double b = other.getArea();
    return (a==b)? 0: (a<b)? -1: +1; // schlecht: Rundungsfehler!
}

Rectangle::Rectangle(string& objName, double l, double b)
    : Shape(objName) {             // vgl. super(objName)
    laenge = l;
    breite = b;
}

// Shape array[100]; geht nicht !!! (warum?)
Shape* array[100];

array[0] = new Rectangle("r1", 17.6, 5.4);
array[1] = new Circle("c1", 23.6);
array[2] = 0;    // vgl. null

int i = 0;
while (array[i] != 0) cout << array[i]->getArea() << endl;
```

Templates in C++ = Schablonen für Funktionen und Klassen (generische Programmierung)

```
template<class T> void swap(T& x, T& y) {
    T hilf = x;
    x = y;
    y = hilf;
}
```

```
int a = 5;
int b = 6;
swap(a,b)
// int,int -> T=int
```

```
template<class X> struct Node {
   Node* next;
   X value;
};

template<class T> class List {
private:
    Node<T>* first;       // in C++ ist definiert struct Typnamen!
public:
    ...
};

List<int> iList;          // Liste für int
iList.addLast(34);

List<string> sList;
sList.addLast("Hallo");   // Liste für string
```

Bewertung von C++

- C++ ist eine multiparadigmen-Erweiterung von C.
- C++ erlaubt effiziente Programmierung von Abstrakten Datentypen (Templates, Operator-Überladen, Inline).
- C++ erlaubt effiziente klassenbasierte Programmierung (Klassen verhalten sich wie C-structs).
- C++ ermöglicht objektorientierte Programmierung.
- Für C++ gibt es umfangreiche Softwarebibliotheken (z.B. STL).

- C++ besitzt keine vordefinierte Klassenhierarchie.
- Der C++-Standard ist nicht allgemein bekannt.
- Unterschiedliche Compiler sind kaum kompatibel.
- Fortgeschrittene Eigenschaften von C++ werden nur zurückhaltend genutzt.
- Es gibt keine allgemein akzeptierte Standardbibliothek.

C++ ist eine Programmiersprache für Profis.