

1. Elementares C++

Wo und wie lange leben Objekte ?

	globale Objekte	lokale Objekte	dynamische Objekte
entstehen durch ...	globale Objektvereinbarung: <code>T o;</code>	blocklokale Objektvereinbarung: <code>{ .. T o; .. }</code>	durch expliziten Aufruf von <code>new</code> : <code>T*op=new T[N];</code>
Objekte sind initialisiert	<i>builtin</i> -Typen: ja, auf 0 Klasstypen: durch Aufruf eines Konstruktors (*)	<i>builtin</i> -Typen: nein ! Klasstypen: durch Aufruf eines Konstruktors (*)	<i>builtin</i> -Typen: nein ! Klasstypen: durch Aufruf eines Konstruktors (*)
werden vernichtet ...	automatisch nach (!) Programmende	automatisch beim Verlassen des Blockes Sonderfall: temporaries (**)	durch expliziten Aufruf von <code>delete</code> : <code>delete[] pi;</code>
residieren im ...	globalen Datenbereich (bereits vom Compiler geplant und vor Programmstart)	Stack (dehnt sich dynamisch und sequentiell aus)	Heap (dehnt sich dynamisch und nicht sequentiell aus)

(* u.U. ohne nutzerspezifische Initialisierung (s. default ctor)

(** am nächsten sequence point (typischerweise ;)

1. Elementares C++

1.5. Strukturierte Anweisungen

Switch-Anweisung (C++): `switch (expression) statement`

statement i.allg. strukturiert mittels case: / default: aber mit mehr Freiheiten als in Java

Beispiel: Duff's Device

(Tom Duff 1983)

```
void send  
(register short *to,  
register short *from,  
register count)  
{ do *to = *from++; while(--count>0); }  
  
// to: some device register
```

```
void send (register short *to, register short *from,  
          register count) {  
    register n = (count+7)/8;  
    switch (count%8){  
        case 0: do{          *to = *from++;  
        case 7:              *to = *from++;  
        case 6:              *to = *from++;  
        case 5:              *to = *from++;  
        case 4:              *to = *from++;  
        case 3:              *to = *from++;  
        case 2:              *to = *from++;  
        case 1:              *to = *from++;  
    } while(--n > 0);  
}
```

1. Elementares C++

1.5. Strukturierte Anweisungen

Switch-Anweisung (C++):

Initialisierungen dürfen nicht 'übersprungen' werden:

```
switch (i) {  
    int v1 = 2; // ERROR: jump past initialized variable  
case 1:  
    int v2 = 3;  
    // ....  
case 2:  
    if (v2 == 7) // ERROR: jump past initialized variable  
        // ....  
}
```

1. Elementares C++

1.5. Un : -) Strukturierte Anweisung

goto - **the don't use statement**

```
loop:                goto skip:
// .....           // .....
goto loop;          skip: //.....
```

Initialisierungen dürfen auch nicht 'übersprungen' werden:

1. Elementares C++

1.5. Strukturierte Anweisungen

Exception Handling : syntaktisch wie in Java (kein `finally`)

```
try {  
    // things that may throw or not  
}  
catch ( Exception1 e ) {  
    // handle e  
}  
catch ( Exception2 e ) {  
    // handle e  
} .....
```

bei Auftreten einer Ausnahme wird der `try`-Block verlassen und zu einem passenden (ggf. übergeordneten) `catch`-Block verzweigt, **zuvor werden alle Destruktoren lokaler Objekte gerufen, die erfolgreich konstruiert wurden !**

1. Elementares C++

1.5. Strukturierte Anweisungen: Exception Handling

```
#include <iostream>

using std::cout; using std::endl;

class X { public:
    X(int i=0) {cout<<"X("<<i<<"")\n";}
    ~X() {cout<<"~X()\n";}
};

void foo(int i) {
    try { X local;
        if (i==1) throw "oops";
        else if (i==2) throw 42;
    }
    catch (const char* why) {
        cout<<why<<endl;
    }
}
```

1. Elementares C++

1.5. Strukturierte Anweisungen: Exception Handling

```
//... cont.  
int main() {  
    try {  
        X x1(1);  
        foo(1);  
        X x2(2);  
        foo(2);  
    }  
    catch (int r) {  
        cout<<"exception: code "<<r<<endl;  
    }  
    catch (...) {  
        cout<<"something thrown: don't know what\n";  
    }  
}
```

```
X(1)  
X(0)  
~X()  
oops  
X(2)  
X(0)  
~X()  
~X()  
~X()  
exception: code 42
```

1. Elementares C++

1.5. Strukturierte Anweisungen: Exception Handling

- Exceptions sind Objekte beliebigen Typs
- stack unwinding ruft Destruktoren aller erfolgreich konstruierten Objekte
- in einem `catch`-Block kann eine Exception mittels `throw;` 're-thrown' werden

```
.... catch (...) {  
                void handleAll(); // proto  
....           handleAll();  
.... }
```

```
void handleAll() {  
    try { throw; }  
    catch (double x) { cout << x << endl; } // z.B.  
}
```


1. Elementares C++

1.5. Strukturierte Anweisungen: Exception Handling

- wird eine Exception nirgends 'gefangen', so endet das Programm durch aufruf von `std::terminate()` (* (dies ruft wiederum `std::abort()`)
- mittels `std::set_terminate()` kann man dieses Verhalten ändern:

Prototyp `void (*set_terminate(void (*term_handler)())) ();`
?????

oder leichter nachvollziehbar:

```
typedef void (*TH) ();  
TH set_terminate(TH);
```

(* es ist implementation-defined,
ob dabei stack-unwinding stattfindet !!!

1. Elementares C++

1.5. Strukturierte Anweisungen: Exception Handling

- `std::terminate()` wird auch gerufen, wenn während der Behandlung einer Ausnahme eine weitere Ausnahme auftritt
- Funktionen können mit sog. exception specifications ausgestattet sein, (entspricht den throws-Klauseln von Java aber)

Java: `void foo(); // lässt keine Exceptions 'raus'`

C++: `void foo(); // lässt beliebige Exceptions 'raus'`

`void foo () throw (dies, das, nochwasanderes);`

Java: vollständige Flussanalyse zur Compile-Zeit

C++: keinerlei Flussanalyse, aber Überwachung zur Laufzeit

- tritt eine Exception auf, die nicht spezifiziert wurde, wird `std::unexpected()` (dies ruft wiederum `std::terminate()`) gerufen

1. Elementares C++

1.5. Strukturierte Anweisungen: Exception Handling

- mittels `std::set_unexpected()` kann man dieses Verhalten ändern
- Aber: Herb Sutter (Exceptional C++, Item 13) und auch Boost (Exception-specification rationale):

Never write an exception specification !

- Destruktoren sollten **NIEMALS** Ausnahmen erzeugen:

```
X::~~X() throw ();
```

WARUM



1. Elementares C++

neu in C++11: noexcept

statisches no-throw:

```
void i_will_never_throw (int i) noexcept{ ... } // noexcept(true)  
template <typename T> bool compare(T t1, T t2) noexcept( noexcept(t1==t2))  
    { return t1 == t2; }
```

sämtliche Exception-Vorkehrungen können im Code entfernt werden

falls doch eine Exception auftritt: terminate