Wo und wie lange leben Objekte ?

	globale Objekte	lokale Objekte	dynamische Objekte
entstehen durch ...	globale Objektvereinbarung: `T o;`	blocklokale Objektvereinbarung: `{ .. T o; ..}`	durch expliziten Aufruf von new: `T*op=new T[N];`
Objekte sind initialisiert	*builtin*-Typen: ja, auf 0 Klassentypen: durch Aufruf eines Konstruktors (*	*builtin*-Typen: nein ! Klassentypen: durch Aufruf eines Konstruktors (*	*builtin*-Typen: nein ! Klassentypen: durch Aufruf eines Konstruktors (*
werden vernichtet ...	automatisch nach (!) Programmende	automatisch beim Verlassen des Blockes Sonderfall: ***temporaries*** (**	durch expliziten Aufruf von delete: `delete[] pi;`
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 ;)**

kurzer Einschub: C++ std Container

Folgende Containertypen existieren schon länger (in C++98):

Typ	Header	Charakteristik	Iteratoren
`vector`	`<vector>`	dynamische Felder mit wahlfreiem Zugriff, erweiterbar am Ende	`RandomAccess`
`deque`	`<deque>`	dynamische Felder mit wahlfreiem Zugriff, erweiterbar am Anfang und am Ende	`RandomAccess`
`list`	`<list>`	doppelt verkette Listen ohne wahlfreiem Zugriff, nicht sortiert	`Bidirectional`
`set`	`<set>`	Mengen mit impliziter Sortierung, keine Duplikate	`Bidirectional`
`multiset`	`<set>`	Mengen mit impliziter Sortierung, Duplikate erlaubt	`Bidirectional`
`map`	`<map>`	Mengen von Schlüssel/Wert- Paaren mit impliziter Sortierung nach dem Schlüssel, keine Duplikate	`Bidirectional`
`multimap`	`<map>`	Mengen von Schlüssel/Wert- Paaren mit impliziter Sortierung nach dem Schlüssel, Duplikate erlaubt	`Bidirectional`

Folgende Containertypen kommen neu hinzu (in C++11):

Typ	Header	Charakteristik	Iteratoren
`array`	`<array>`	Felder fester Länge mit wahlfreiem Zugriff	`RandomAccess`
`forward_list`	`<forward_list>`	einfach (vorwärts) verkette Listen ohne wahlfreiem Zugriff, nicht sortiert	`forward`
`unordered_set`	`<unorderd_set>`	unsortierte Mengen mit Hash, keine Duplikate	`forward`
`unordered_ multiset`	`<unorderd_set>`	unsortierte Mengen mit Hash, Duplikate erlaubt	`forward`
`unordered_ map`	`<unorderd_map>`	unsortierte Mengen von Schlüssel/Wert-Paaren mit Hash, keine Duplikate	`forward`
`unordered_ multimap`	`<unorderd_map>`	unsortierte Mengen von Schlüssel/Wert-Paaren mit Hash, Duplikate erlaubt	`forward`

kurzer Einschub:
C++ std Container

http://www.cplusplus.com/reference/stl/

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);
    }
}
```

1.5. 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
       // ....
}
```

1.5. Un : -) Strukturierte Anweisung

goto - the don't use statement

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

```
goto skip:
// ....
skip: //....
```

Initialisierungen dürfen auch nicht 'übersprungen' werden:

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 !

1.5. 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 "ooops";
                else if (i==2) throw 42;
        }
        catch (const char* why) {
                cout<<why<<endl;
        }
}
```

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()
ooops
X(2)
X(0)
~X()
~X()
~X()
exception: code 42
```

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.
}
```

1.5. Strukturierte Anweisungen: Exception Handling

- wird eine Exception nirgends 'gefangen', so endet dass 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 !!!**

1.5. 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