1. Elementares C++

C - **enum**s mit Problemen:

- konvertierbar nach int
- exportieren ihre Aufzählungsbezeichner in den umgebenden Bereich (name clashes)
- schwach typisiert (z.B. keine forward Deklaration möglich)

enum classes ("strong enums") sind stark typisiert und 'scoped':

```
enum Alert { green, yellow, election, red }; // traditional enum
enum class Color { red, blue };              // scoped and strongly typed enum
                 // no export of enumerator names into enclosing scope
                 // no implicit conversion to int
enum class TrafficLight { red, yellow, green };

Alert a = 7;             // error (as ever in C++)
Color c = 7;             // error: no int->Color conversion
int a2 = red;            // ok: Alert->int conversion
int a3 = Alert::red;     // error in C++98; ok in C++0x
int a4 = blue;           // error: blue not in scope
int a5 = Color::blue;    // error: not Color->int conversion
Color a6 = Color::blue; // ok
```

Typ der Repräsentation kann spezifiziert werden

```
enum class Color : char { red, blue }; // compact representation enum class

TrafficLight { red, yellow, green };
    // by default, the underlying type is int

enum E { E1 = 1, E2 = 2, Ebig = 0xFFFFFFF0U };
// how big is an E?
// (whatever the old rules say;
// i.e. "implementation defined")

enum EE : unsigned long { EE1 = 1, EE2 = 2, EEbig = 0xFFFFFFF0U };
// now we can be specific
```

C++11

forward Deklaration möglich

```
enum class Color_code : char; // (forward) declaration
void foobar(Color_code* p); // use of forward declaration
```

1.2. Datentypen (Felder)

```
int p[];
```

nur in Argumentlisten von Funktionen:
int-Feld unbekannter == beliebiger Länge, Größeninformation ist separat bereitzustellen

```
double m[3][4];
```

12 doubles hintereinander !

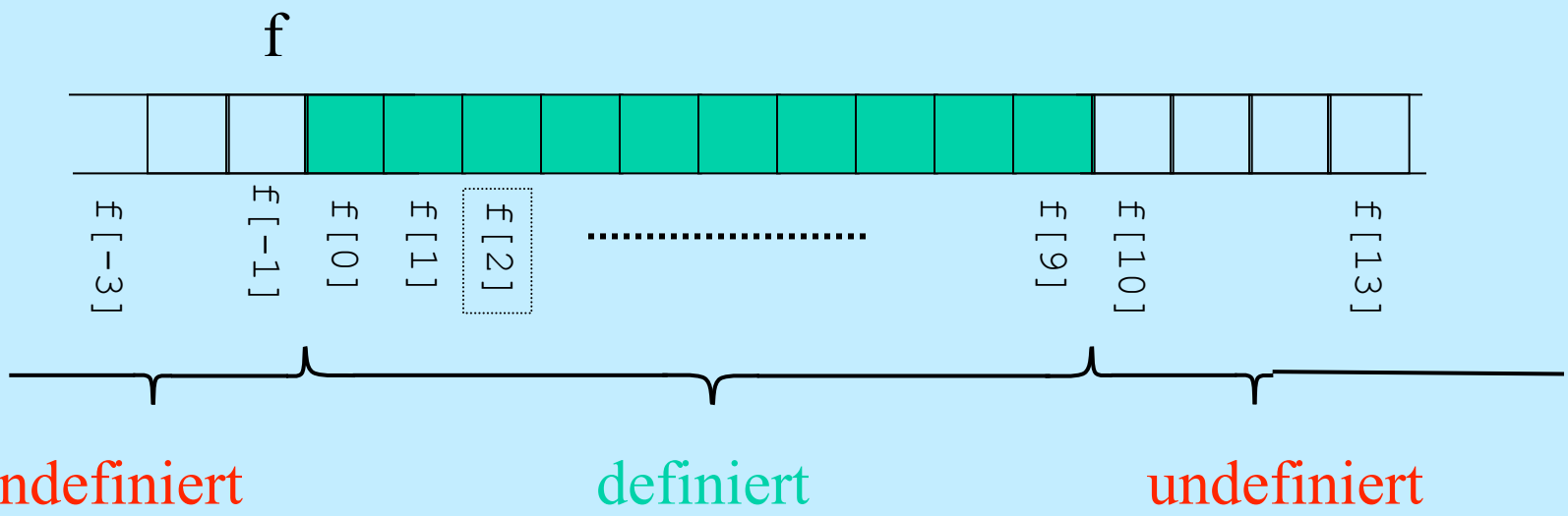
Typedefs: Synonyme für (u.U.) komplexe Typkonstrukte

```
typedef double V4[4];
V4 m[3]; // entspricht obigem Feld
```

1.2. Datentypen (Felder)

es gibt keine Prüfung auf Einhaltung der Feldgrenzen bei Zugriffen:

```
T f [10];
```

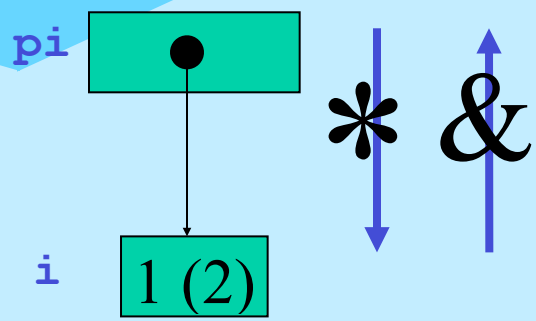


1.2. Datentypen

Zeiger: Indirektion per Speicheradresse!

```
int* pi; // ein u.U. NICHT initialisierter Zeiger

int i=1;
pi = & i; // Adressoperator !
```



Zeiger sind vollständig typisiert:

```
double x;
// ERROR: pi = & x;
```

Umkehroperation zu & ist die sog. Dereferenzierung:

```
*pi = 2;
```

1.2. Datentypen (Zeiger)

```
int* pi = new int; // ein neues anonymes int auf dem Heap
                   // pi ist (bislang) der einzig Zugang
                   // no more C: int* pi = malloc(sizeof(int));
int* ap = pi; // 2 Verweise, 1 Objekt !

pi = 0; // ausgezeichneter Zeigerwert <==> KEIN Objekt

ap = 0; // letzte Referenz weg: KEINE garbage collection
        // sondern ein memory leak
```

daher zuvor:

```
delete ap; // kein leak !
           // no more C: free(pi);
```

1.2. Datentypen (Zeiger)

ACHTUNG: nach `delete zeiger;` ist u.U. in `zeiger` immer noch die gleiche Adresse enthalten, jeder Zugriff darüber ist jedoch undefiniert !
Empfehlung: `delete pi; pi=0;`

auch Felder können dynamisch erzeugt werden:

```
int* pf = new int [100]; // pf zeigt auf erstes von 100 int's
```

Zeiger auf Felder sind mit `delete[]` zu deallokieren:

```
delete[] pf; pf=0;
```

1.2. Datentypen (Zeiger)

Zeiger auf Klassentypen (~ Java-Objektsemantik)

```
class X {
public:
  X() {std::cout<<"hi\n";}
  ~X(){std::cout<<"bye\n";}
};

void foo() { X* px = new X; } // hi & leak
void bar() { X* px = new X; delete px; } // hi & bye
```

auch: `new X();` möglich aber nicht zu empfehlen !

In jeder (nicht static) Memberfunktion ist `this` ein Zeiger auf das Objekt, an dem der Aufruf der Funktion erfolgte (anders als in Java !)

1.2. Datentypen (Zeiger)

Java - **new** vs. C++ - **new**

```
class X {}

class Main {
        public static void main(String s[]) {
                // X x = new X; nicht ohne leere Parameterliste
                X x = new X();
                // int i = new int; new nur für Klassen erlaubt
                // int i = new int(); auch so nicht
                int i[] = new int[10]; // Felder sind Objekte
        }
}
```

1.2. Datentypen (Zeiger)

Java - new vs. C++ - new

```
class X {};

int main() {
                X *x1 = new X;          // besser so
                X *x2 = new X();        // als so
                int *i1 = new int;      // di
                int *i2 = new int();    //   to
                // int i[] = new int[10]; so nicht:
                // Feldvariablen sind Konstanten & die Größe
                // von i ist unbestimmt
                int *i = new int[10]; // ein Zeiger kann
                // eins oder viele referenzieren !
}
```

nullptr

C++11

```
void foo(int) {std::cout<<"foo(int)\n";}
void foo(int*) {std::cout<<"foo(int*)\n";}
…
foo(0);
foo(NULL);
foo(nullptr);

int* pi = nullptr;

if (pi == nullptr) pi = new int;
```

Warum besser keine Klammern bei parameterlosen Konstruktorrufen ?

```
T* pt = new T(); // ok

T t = T(); // ok, aber redundant
T t ();    // auch ok, aber kein Objekt vom Typ T !
```

?

```
void foo(); // Funktionsdeklaration !!!
T    t  (); //   dito

T t;
```

1.2. Datentypen (Zeiger)

Felder sind konstante Zeiger:

```
T someTs [30];

T* pt = someTs;
T* qt = &someTs[0];
```

```
using std::cout;...
cout<<1["]<<2["]<<endl;
Korrektes C++ ?
wenn ja, was wird ausgegeben ?
```

`pt[i]` ist nur eine abkürzende Notation von `*(pt+i)`

Die Zeigerarithmetik erfolgt modulo `sizeof(T)`

`pt` ist ein Zeiger auf's erste `T` im Feld

`pt+1` ist ein Zeiger auf's zweite `T` im Feld ...

1.2. Datentypen

Zeichenketten:

Zeichenkettenliterale wie »üblich«

```
"eine Zeichenkette\nmit Doppelapostroph \" und Backslash \\"
```

werden als 0-terminierte `char`-Felder realisiert !

```
char* hello = "Hello, World";
```