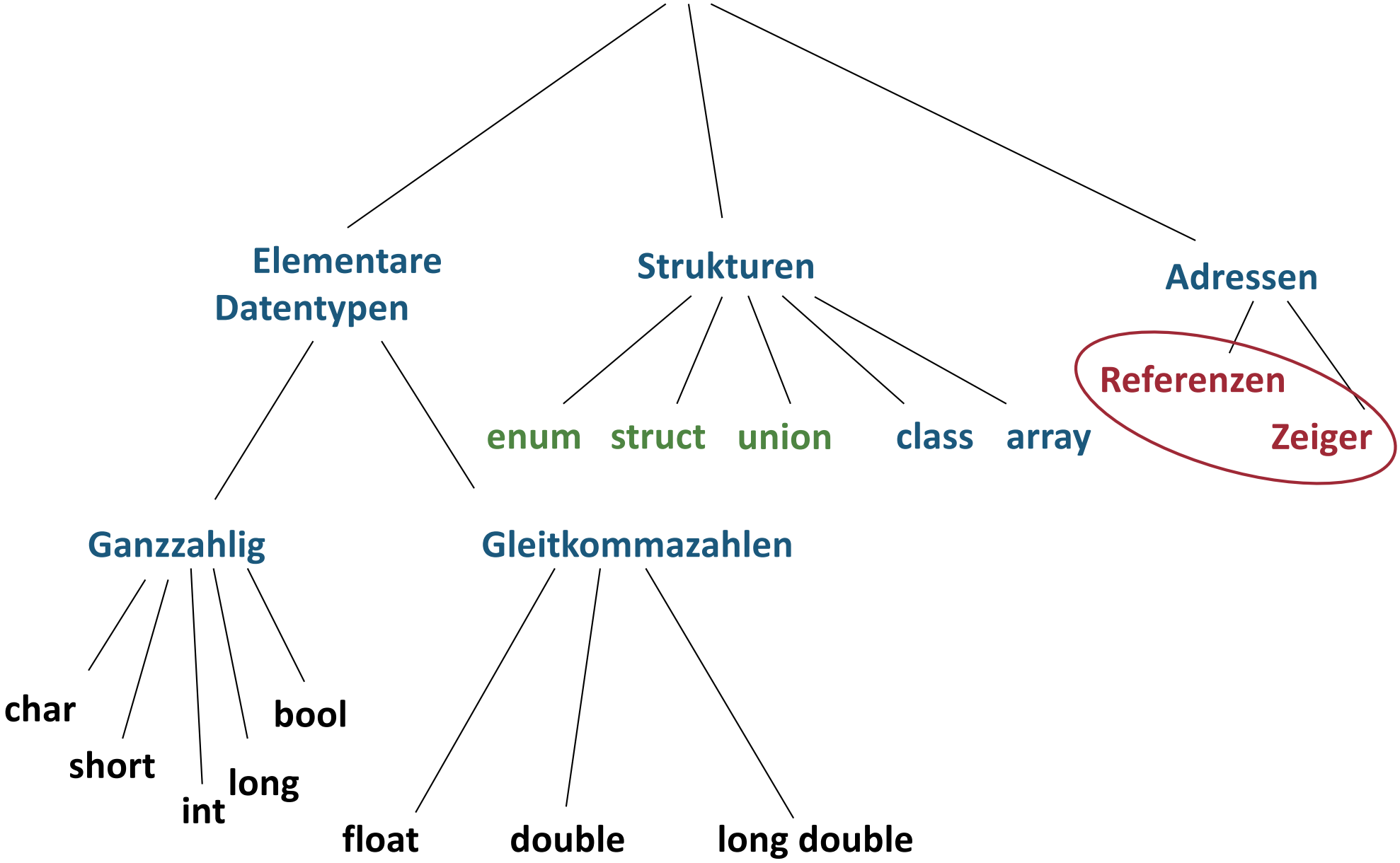
K04

Zeiger & Co

1. Zeiger
   - Verwendung und Zeigerarithmetik
2. Referenzen
3. Arrays
4. Zeigertabellen

Wiederholung: Das Typsystem in C++



Zeiger – kurz und knapp

In C++ (und einigen anderen Programmiersprachen wie z.B. C) wird ein

Typ dessen Wertebereich Adressen sind

als Zeiger (engl. Pointer) bezeichnet.

Beachte: An der Adresse können Daten, Programmcode, als auch wiederum eine Adresse liegen.

Wozu braucht es Zeiger? (i)

- Problem:
  - Jede Variable ist fest mit einem Speicherbereich verbunden.
  - Ziel: gesucht ist ein Programmstück, das beliebige Speicherbereiche verarbeiten kann, ohne vorher extra eine Kopie davon zu machen (vorausgesetzt, der Typ ist für die Verarbeitung geeignet).
- Beispiel (in Pseudocode)
  - Annahme: **`Polynom`** ist **`struct`** mit 100 Koeffizienten

```
Polynom ptmp, p1, p2;
if (bedingung)
        ptmp = p1; // kopiert 100 Koeffizienten!
else
        ptmp = p2; // macht wieder dasselbe!

bearbeite ptmp
wieder zurück kopieren; // kopiert 100 Koeffizienten!
```

Wozu braucht es Zeiger? (ii)

- Beispiel (in Pseudocode)

```
Polynom p1, p2;
Polynom-Zeiger ptmp;

if (bedingung)
        ptmp zeigt jetzt auf p1
else
        ptmp zeigt jetzt auf p2

Bearbeite das worauf ptmp zeigt
```

- Solche Zeiger existieren auch in Java, man sieht sie jedoch nicht.

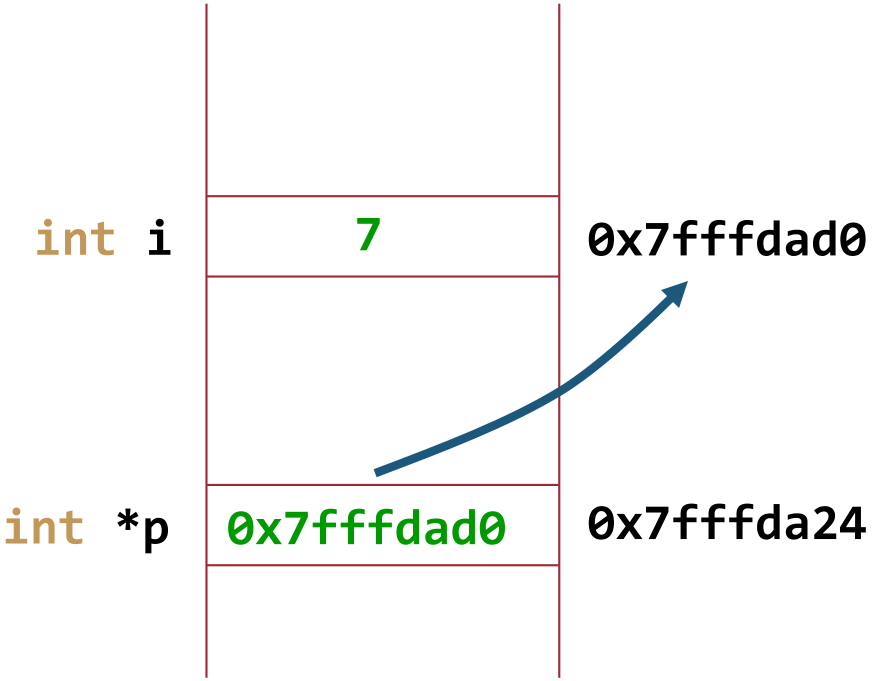
Genauer: Was ist ein Zeiger?

Erinnerung: Variable

- Variable = Name für Speicherbereich = Name für Anfangsadresse
- Typ (z.B. `int`) definiert, wie Bits interpretiert werden sollen, die an dieser Adresse gespeichert sind
- Jede Variable ist genau einem Adressbereich fest zugeordnet

Zeiger

- Variable, wie alle anderen auch
  - Steht irgendwo im Speicher an bestimmter Adresse
  - Hat einen Wert
  - Bedeutung des Wertes = Adresse eines Speicherbereiches
- m.a.W: ein Zeiger macht keinen Sinn ohne etwas worauf er zeigen kann



Eigenschaften von Zeigern

- Mit Zeigern kann man indirekt auf den Wert einer anderen Variable zugreifen, ohne deren Namen zu verwenden (bzw. zu kennen)!
- Zeiger selbst ist immer gleich gross, unabhängig von der Grösse des Typs, auf den der Zeiger verweist (Wortbreite der CPU-Architektur)
- Fähigkeiten 'normaler' Variablen:
  - Arithmetik (hier: Adressberechnungen)
  - Zuweisung, Vergleiche
- Vorteil: Direkter Zugriff auf den Speicher des Computers
  - Unverzichtbar zur schnellen, maschinennahen Programmierung
- Nachteile:
  - Keine Überprüfung auf korrekte Handhabung
  - "gewöhnungsbedürftige" Syntax

Syntax zur Zeigerdeklaration

Die allgemeine Syntax zur Deklaration eines Zeiger lautet:

```
DatenTyp * variablenName [ = Wert];
```

- Wobei **DatenTyp** ein zuvor deklarierter oder elementarer Typ ist
- Zu jedem Typ **T** gibt es einen Zeiger-Typ **T***

```
int  i = 42;
int* p1, j;      // Achtung: j ist KEIN Zeiger auf int
                 // sondern eine Variable vom Typ int
int* p2, *p3;    // p2 und p3 sind Zeiger auf int
```

Adressoperator

- Wie gelangt man an die Adresse einer Variablen?

  ```
  pointervar = &var;
  ```

  wobei **var** vom Typ **T** ist und daher **pointervar** vom Typ **T***

- Der neue Operator `&` heisst Adressoperator (&: *ampersand*)
  - Es ist ein unärer Operator, der die Speicheradresse des Operanden zurückliefert
- Beispiele:

```
int* pi;
int i = 17;
pi = &i;
```

```
struct S {...};
S s;
S* ps = &s
```

```
float f;
float*   pf = &f;
float** ppf = &pf;
```

```
float f;
float** ppf = &&f;
        // geht nicht!
```

Dereferenzierungsoperator

- Wie erhält man an das Objekt an der Adresse auf die ein Zeiger zeigt?
- Man benötigt einen unären Operator der das Objekt zurück gibt, auf das der Operand (Zeiger) verweist.
- Syntax:

```
var = *ptr-expr
```

  wobei **`ptr-expr`** ein Ausdruck ist, der einen Wert vom Typ **`T*`** liefert. Das Resultat von ***`ptr-expr`** ist dann vom Typ **`T`**.

- „Gegenstück“ zum **&**-Operator.
- Wird auch Stern-Operator genannt.
- Achtung:
  - Je nach Kontext kann ***** auch Multiplikation sein!
  - Nicht zu verwechseln mit ***** Modifier zur Deklaration eines Zeigers!

Adress- u. Dereferenzierungsoperator

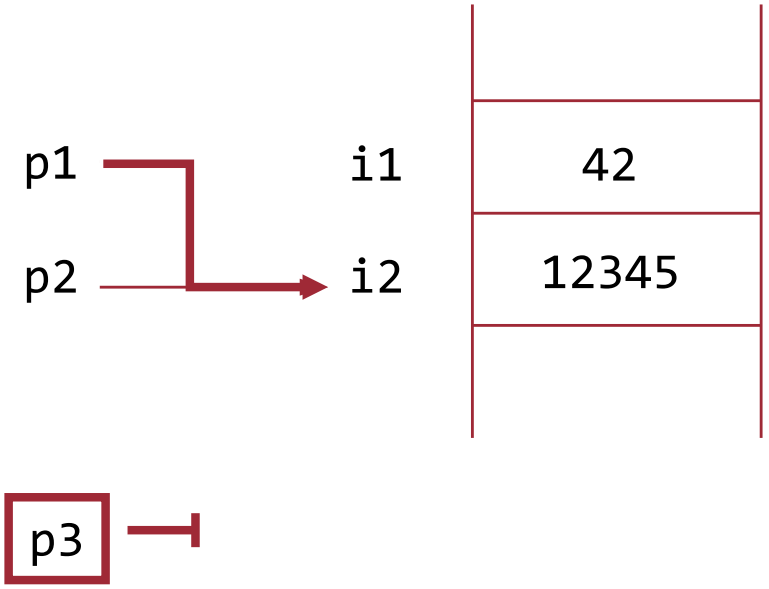
Beispiele:

```
int i=1, j=0;

int* p = &i;          // p zeigt auf i

i = i + *p            // verdoppelt i

p = &j                // p zeigt jetzt auf j

*p = 42               // j ist jetzt 42
                      // (i bleibt unverändert, war 2)


int x = j**p          // 42 * 42
```

Zeiger, Deref'- und Addressoperator

Beispiele:

```
int i1 = 0, i2 = 100;
int *p1 = &i1, *p2 = &i2;

*p1 = 42;   // as i1=42

*p2 = *p1; // as i2=i1

p1 = p2;

*p1 = 12345;

int* p3;
```



p3 == NULL Spezieller Wert 0x00000000

… und noch ein Beispiel

```
int   a = 42;
int*  b = &a;
*b = 27;
std::cout << a << std::endl;
```

Ergebnis?

27

```
int a = 42, b = 137;
int* c = &a;
int* d = &b;
c = d;
std::cout << a << "\n" << *c
              << std::endl;
```

Ergebnis?

42

137

```
int a = 42;

int*  c = &a;
int** d = &c;

std::cout << *d << std::endl;
```

Ergebnis?

0x????????

Wert in c
= Adresse von a
Wert in d
= Adresse von c

Zeiger auf `struct`

- Kommt sehr häufig vor, insbesondere Zeiger auf Klassen bzw. Objekte (kommt später).
- Zugriff auf Membervariablen einer Struktur mit

  `(*ps).id` oder (kürzer) **`ps->id`**

  wobei **`ps`** ein Zeiger auf eine Struktur mit Member (Feld) **`id`** ist

```
struct Student {
        int id;
        char name[80];
};
Student s = {10, "Klara"};
Student* ps;
cout << s.id << endl;          // output id=10
ps = &s;
cout << (*ps).id << endl;      // output id=10
cout <<  ps->id << endl;       // output id=10
```

Vergleichsoperatoren für Zeiger

- Zeiger kann man auf Gleichheit (==) und Ungleichheit (!=) vergleichen
  - Wie bei allen anderen Typen auch
  - Gleichheit bedeutet: zeigt auf dieselbe Variable (d.h. Inhalt ist dieselbe Speicheradresse)

- Alle anderen Vergleiche wie <, >, <=, >=, ... sind auch erlaubt, werden jedoch selten verwendet

Zeigerarithmetik: Addition (i)

- Mit Zeigern kann man rechnen (bzw. mit den Adressen, die sie enthalten)
- Ausdruck der Form:

```
pointer + k
```

wobei `k` eine Ganzzahl ist (z.B. **`int`**) und **`pointer`** vom Typ `T*` sei.

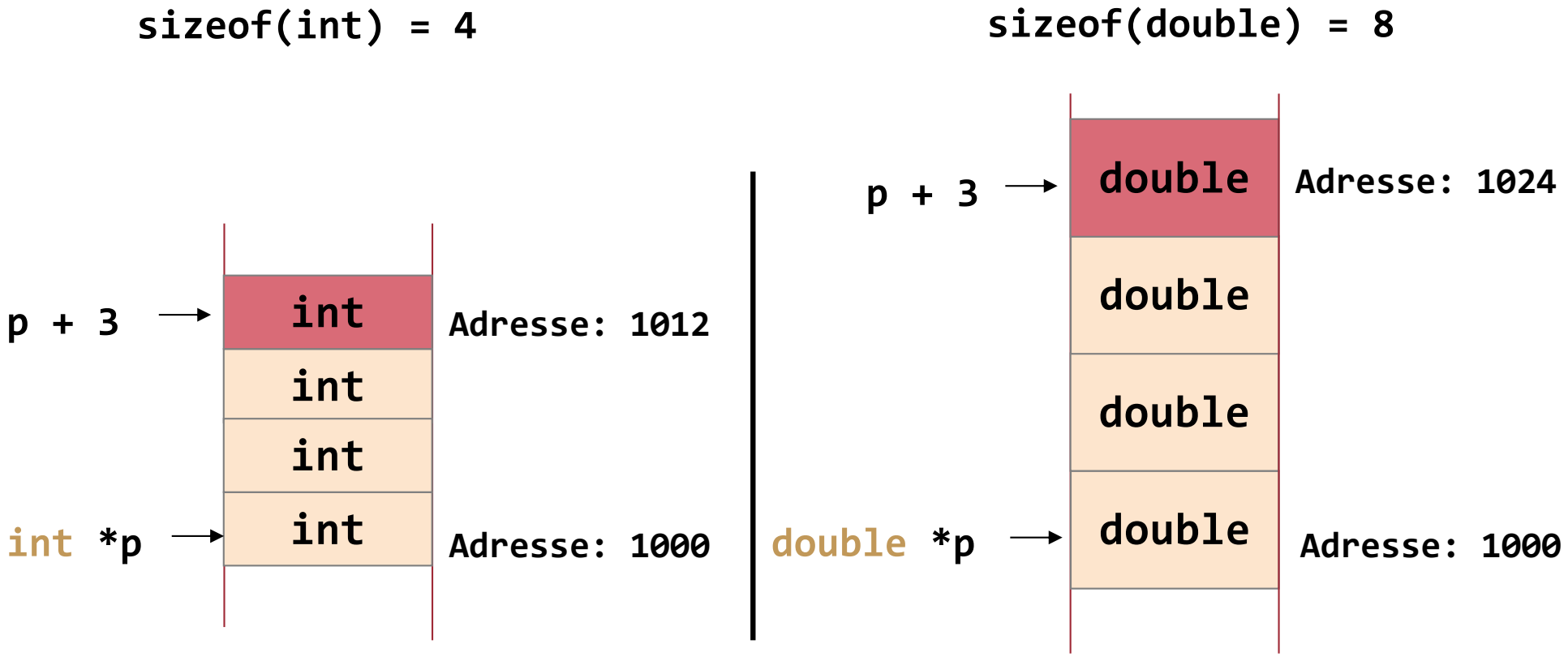
Achtung! Bedeutung ist nicht: **`Adresse + k`**

Sondern: **`Adresse + k×sizeof(T)`**

- Dies ist die low-level-Methode, um mit Arrays zu arbeiten – dazu später mehr.

Zeigerarithmetik: Addition (ii)

Beispiel:



Zeigerarithmetik: Subtraktion (i)

- Subtraktion von Zeigern:

```
pointer1 – pointer2
```

- Bedeutung:

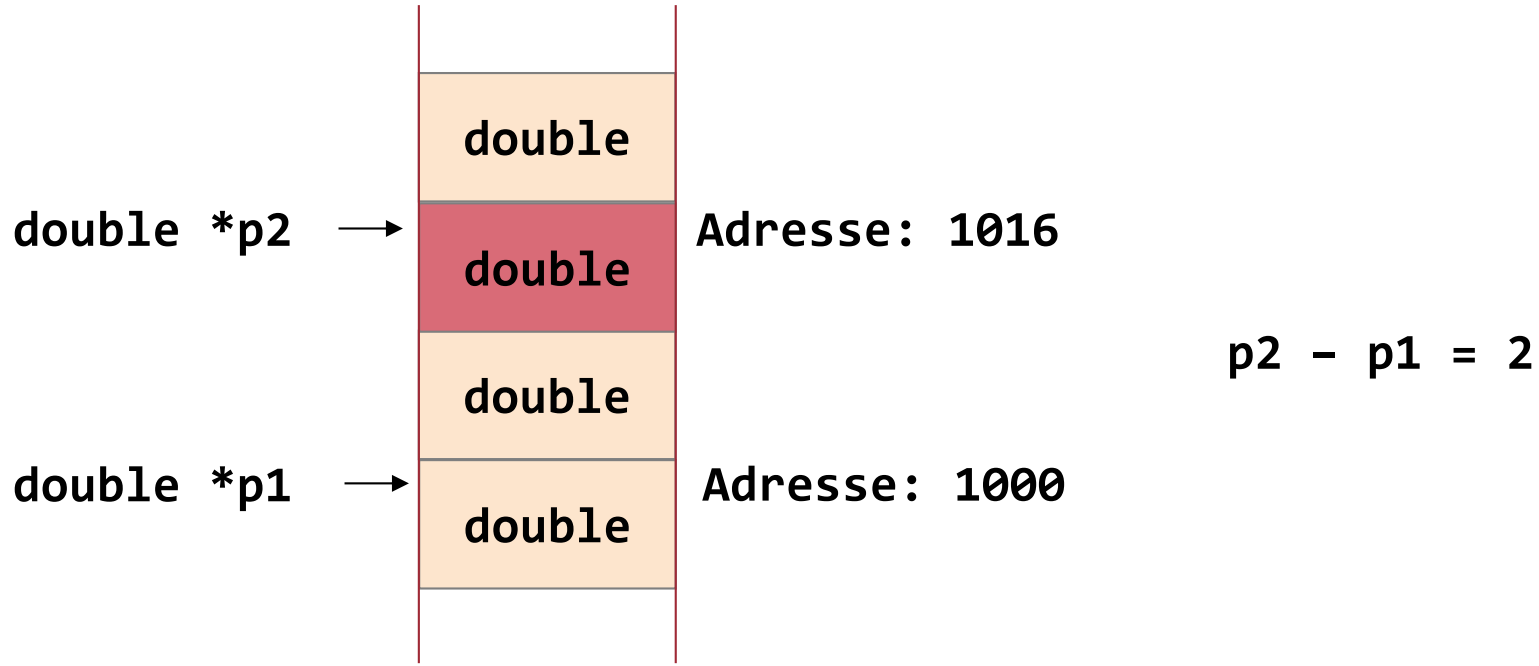
```
(Adresse1 - Adresse2) / sizeof(T)
```

- Ergebnis ist die **Distanz** in **T**-Elementen zwischen beiden Pointern

Zeigerarithmetik: Subtraktion (ii)

Beispiel:

```
sizeof(double) = 8
```



... noch ein Zeigerarithmetik-Beispiel

```
sizeof(int) = 4
int* p = &a;
```

p (aa8200)	aa8192
b (aa8196)	9
a (aa8192)	16

```
*p = 200;
```

p (aa8200)	aa8192
b (aa8196)	9
a (aa8192)	200

```
*(p+1) = 300;
```

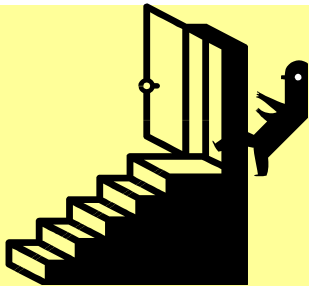
p (aa8200)	aa8192
b (aa8196)	300
a (aa8192)	200

- Zeiger **p** weist auf ein **int**, d.h. durch Addition von 1 erhöhen wir die Adresse um die Grösse eines **int**. Der C/C++ Ausdruck für die Ermittlung der Grösse eines Typs T lautet: **sizeof(**T**)**; hier also **sizeof(int)**.

Nullzeiger

- Problem: wie unterscheidet man gültige Zeiger von ungültigen Zeigern, d.h. von Zeigern, die auf nichts verweisen?
- Adresse **0** bzw. Wert **NULL** ist genau dafür reserviert.
- Was passiert, wenn man **NULL**-Pointer dereferenziert?
  - Core Dump (relativ einfacher Bug)
  - Häufig verursacht durch uninitialisierte Zeiger

```
int* c = NULL;
std::cout <<  c << std::endl; // OK
std::cout << *c << std::endl; // core dump
```



`void`-Zeiger

- **`void*`** ist ein Zeigertyp, der auf Variablen irgendeines Typs zeigen kann.
- Automatische Konvertierung in eine Richtung:
  - Automatisch von **`T*`** nach **`void*`** (speziell → allgemein)
  - Nur mit explizitem Cast von **`void*`** nach **`T*`** (allgemein → speziell)

```
char* foo = "Dies ist ein Test";
char* bar;
void* somePtr;
somePtr = foo;      // automatic cast
bar = somePtr;      // error
bar = static_cast <char*>(somePtr);          // explizit,
                                             // gefährlich!
```

Zeiger und **const** (i)

- Bei Deklaration eines Zeigers sind zwei Objekte beteiligt: Zeiger und referenziertes Objekt
- Beide können unabhängig voneinander konstant sein

Beispiel: **Konstantes referenziertes Objekt** (Objekt ist read-only)

```
const char* p1 = "123";   // p1 Zeiger auf
                          // konstantes char-array
char const* p2 = "123";   // äquivalente Syntax

char c = *p1;  // OK
*p1 = 'X';     //Fehler, das Objekt *p1 ist konstant!

p1 += 1;       // OK, Zeiger wird verändert

char* z = p2;  // Fehler! Warum?
const char* z = p2;  // Besser
```

Zeiger und **const** (ii)

Beispiel: **Konstanter Zeiger** (Zeiger ist read-only)

```
char* const p3 = new char;  // p3 konstanter Zeiger
                            // auf ein char
*p3 = '\n';  // OK
++p3;        // Fehler, der Zeiger ist konstant
```

- Tipp: Deklaration von rechts nach links lesen

  `int* const iz;` „`iz` is a constant pointer to `int`“

  `const int* iz;` „`iz` is a pointer to a constant `int`“

Beispiel: **Zeiger und Objekt konstant**

```
const char* const p4; // p4 konstanter Zeiger auf
                      // konstante Variable (ungewöhnlich)

*p4 = 'a';   // Fehler, Inhalt von *p4 ist konstant
 p4+= 1;     // Fehler, Zeiger p4 ist konstant
```

Zeiger und ihre „dunkle Seite“

```
int  a = 42;
int* p;
p = &a;

std::cout << "an der Adresse " << p
          << "steht (als integer) "
          << *p << std::endl;

float* q;
q = p;
q = reinterpret_cast<float*>(p);

std::cout << "an der Adresse " << q
          << " steht (als float) "
          << *q << std::endl;
```

p enthält eine Adresse. Der Speicherinhalt dort soll als Integer interpretiert werden

p enthält jetzt die Adresse, an der die Variable **a** im Speicher liegt.

q enthält eine Adresse. Der Speicherinhalt dort soll als float interpretiert werden

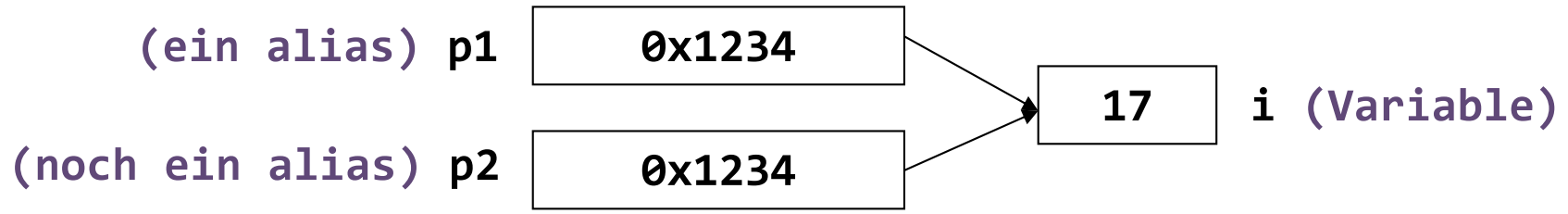
hier meckert der Compiler – zu Recht!

das muss der Compiler akzeptieren

```
an der Adresse 0xbffff578 steht (als integer) 42
an der Adresse 0xbffff578 steht (als float) 5.88545e-44
```

Aliasing

- Dieselbe Variable kann über verschiedene Wege (Zeiger oder Referenzen) erreicht werden
- Dies nennt man Aliasing



- Dies kann für den Compiler bei der Optimierung zu einem Problem werden. Beispiel:

Es besteht eine strenge Ordnung zw. den beiden Zeilen. Kein Vertauschen möglich!

```
int  i=10;
int* ip1 = &i;
int* ip2 = &i;
*ip1 = 17;
foo (*ip2);
```

K04

Zeiger & Co

1. Zeiger
   - Verwendung und Zeigerarithmetik
2. **Referenzen**
3. Arrays
4. Zeigertabellen

Wozu Referenzen?

- „Problem“ der Zeiger:
  - Wert eines Zeigers (Adresse, auf die der Zeiger verweist) kann sich beliebig ändern

- Lösung: neues Sprachkonstrukt **Referenz**
- Referenzen sind alternative Bezeichner für existierende Speicherbereiche
  - Können **NICHT** umgehängt (also verändert) werden
  - Müssen zur Konstruktionszeit zugewiesen werden
    - Ergo: sie sind niemals undefiniert
  - Arrays von Referenzen sind nicht möglich
  - Referenz = anderer Name (Alias) für ein Objekt

Syntax der Referenz

- Deklaration & Initialisierung:

```
Typ & refname = varname;
```

Beachte: Keine Deklaration ohne Initialisierung!

- Verwendung:

```
refname
```

Beachte: Kein Dereferenzierungs-Operator *!

Beispiele:

```
int  i = 17, j = 99;
int& ri = i;   // ri Referenz auf i
i += ri;       // verdoppelt nun i
ri = j;        // jetzt ist i == 99
```

Referenzen auf `struct`

- Referenzen sind bei **`struct`** (und Klassen) bequem:

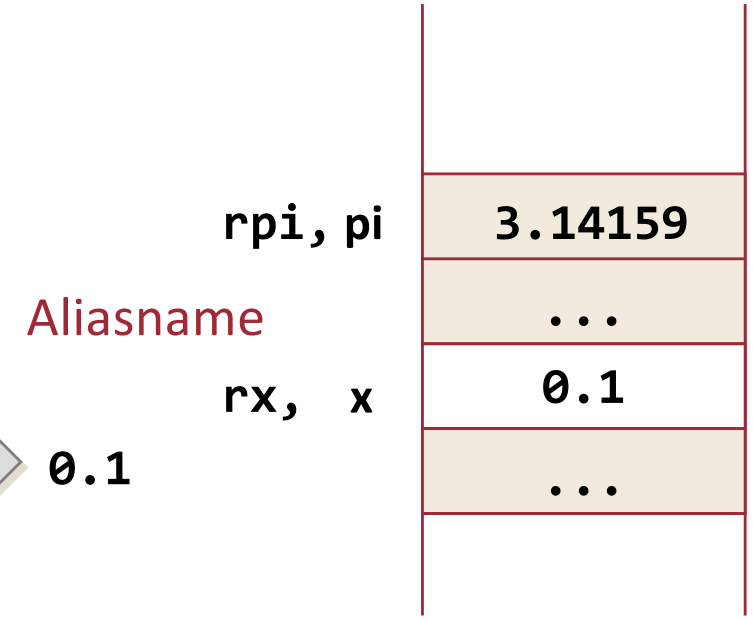
```
struct S {
        float x;
        float y;
};

struct S s;
s.x = 1.0;              // direkter Member-Zugriff

struct S* ps = &s;
ps->x = 1.0;            // Member-Zugriff über Zeiger
(*ps).x = 1.0;          // Member-Zugriff über Zeiger


struct S& rs = s;
rs.x = 1.0;             // Member-Zugriff über Referenz
```

Beispiel: Referenzenzuweisung

```
...
float x = 42.1f;
...
   float& rx = x;
   rx /=  421.0f;

cout << "x = " << x;
   const double pi = 3.14159;
   const double& rpi = pi;
   rpi = 2.345;
```

Read-only Referenz!

0.1



Kombination von Zeiger & Referenz

Was macht folgender Code?

```
int   j = 1;
int* pj = &j;
cout << "j "   << j   << endl;
cout << "pj "  << pj  << endl;
cout << "*pj " << *pj << endl;
// Zeiger auf Referenz nicht OK.
// int&   *t = pj;
// Referenz auf Zeiger ist OK.
int* &pt = pj;
cout << "pt "  << pt << endl;

pt = NULL;
cout << "pt "  << pt << endl;
cout << "pj "  << pj << endl;
```



```
j 1
pj 0xbffff300
*pj 1


pt 0xbffff300

pt 0
pj 0
```

Man sollte jedoch ein solches „Gemisch“ aus Zeigern und Referenzen möglichst vermeiden.

Vergleich Zeiger versus Referenz

Zeiger

- Zeigt auf ein anderes Objekt und ist im Speicher explizit vorhanden
- Kann man auf **`NULL`** testen
- Kann auf beliebig viele verschiedene Objekte zeigen
- Void Pointer möglich
- Bei Auswertung explizit als Zeiger zu erkennen

Referenz

- Ist ein zweiter Name (Alias) für ein Objekt und existiert nicht explizit im Speicher
- Muss immer auf etwas zeigen
- Kann nur für ein Objekt ein Alias sein (kann nicht nachträglich geändert werden)
- Referenz auf void Pointer möglich
- Bei Auswertung nicht als Referenz zu erkennen

Vergleich Standardvariable, Zeiger und Referenz bzgl. Funktionsaufrufen

Was passiert hier genau? Was insbesondere auf dem Stack?

```
int inc(int x) {
   return ++x;
}

int i = 1;
int j = inc(i);
cout<<"i "<<i<<endl;
cout<<"j "<<j<<endl;
```

```
int* inc(int* x) {
   ++(*x); return x;
}

int i  = 1;
int* p = inc(&i);
cout<<"&i "<<&i<<endl;
cout<<"p  "<<p <<endl;
cout<<"i  "<<i <<endl;
cout<<"*p "<<*p<<endl;
```

```
int inc(int& x) {
   return ++x;
}

int i = 1;
int j = inc(i);
cout<<"i "<<i<<endl;
cout<<"j "<<j<<endl;
```

Ausgabe:

```
i 1
j 2
```

```
&i 0x7fff5e9729ec
p  0x7fff5e9729ec
i  2
*p 2
```

```
i 2
j 2
```

Zusammenfassung – Syntax

Verwendung	* (star)	& (ampersand)

als Modifier	`int* p;`
	`void function(int* p);`
	Deklaration eines Zeigers
		`int& a = b;`
		`void function(int& a);`
		Deklaration einer Referenz

als unärer Operator	`std::cout << *p;` Dereferenzierung: Der Speicherinhalt ab Adresse **p** wird dem Zeigertyp entsprechend interpretiert und ausgegeben.	`std::cout << &a;` Adress-Operator: Gibt Adresse aus, an der Variable **a** im Speicher gespeichert ist.
als binärer Operator	`std::cout << a * b;` Muliplikation von **a** und **b**	`std::cout << (a & b);` Bitweises logisches "Und" von **a** und **b**

Die Bedeutung des binären Operators ist abhängig vom Kontext!