

Typumwandlung

• Umwandlungen

- o Ganze Zahlen

- Zieltyp **unsigned**

- alle Bits aus der Quelle, die ins Ziel passen, werden kopiert
- der Rest (höherwertige Bits) wird ggf. ignoriert

```
unsigned char uc = 1023; // binär 11 1111 1111
      |           |
      8 bit      10 bit          ⇒ uc = 255
```

- Zieltyp **signed**

- Wertübernahme, wenn im Ziel darstellbar; **sonst undefiniert!**

```
signed char sc = 1023; // plausible Resultate 127 oder -1
```

Typumwandlung

• Umwandlungen

- o Fließkommazahlen

- float → double

- ⇒ passt immer

- double → float

- ⇒ Wertübernahme, wenn im Ziel darstellbar; **sonst undefiniert!**

- float/double → Ganzzahl

- ⇒ Ungenauigkeiten und möglicher Datenverlust

```
int i = 2.6; → i = 2; char c = 2.3e8; → c = -128;
```

Der Compiler warnt vor möglichem Datenverlust!
Warnungen des Compiler nicht ignorieren!

Typumwandlung

Trauen Sie nicht **vorbehaltlos** den Ergebnissen des Rechners!

Bsp:

$$333.75y^6 + x^2(11x^2y^2 - y^6 - 121y^4 - 2) + 5.5y^8 + \frac{x}{2y}$$

für $x = 77617$, $y = 33096$

Resultat bei doppelter Genauigkeit (double): -1.18059e+021

→ exakt: **-54767 / 66192 = -0.827396...**

Typumwandlung

Vorbemerkung:

Die Regeln von C++ garantieren, dass Typfehler unmöglich sind.
Theorie: Wenn Programm sauber kompiliert, dann keine Durchführung von ungültigen / unsauberen Operationen an Objekten.

- Wertvolle Garantie! → nicht leichtfertig aufgeben!

Aber: explizite Typumwandlung (cast) untergräbt das Typsystem!

explizite Typumwandlung:

C Stil:

```
(T) Ausdruck // wandelt Ausdruck in den Typ T um
```

```
T(Ausdruck) // wandelt Ausdruck in den Typ T um
```

mißbilligt (deprecated)
Nicht verwenden!

Explizite Typumwandlung (C++)

- `const_cast<T>(Ausdruck)`
→ beseitigt Konstanz von Objekten
- `dynamic_cast<T>(Ausdruck)`
→ zum „Downcasten“ bei polymorphen Quelltypen
→ umwandeln in einen abgeleiteten Typ
→ Fehlschlag bei * ergibt Nullpointer, bei & Ausnahme `bad_cast`
- `reinterpret_cast<T>(Ausdruck)`
→ verwendet auf niedriger Ebene (Uminterpretation des Bitmusters)
→ Ziel muss mindestens soviele Bits wie Quelle haben, sonst ... ☹️🐛
- `static_cast<T>(Ausdruck)`
→ zum Erzwingen von impliziten Typumwandlungen

Typumwandlung

Wenn im Code viele Casts notwendig sind,
dann stimmt meistens etwas mit dem Design des Programms nicht!

Wenn im Code ein Cast notwendig ist,
dann die Cast-Operation von C++ verwenden, weil

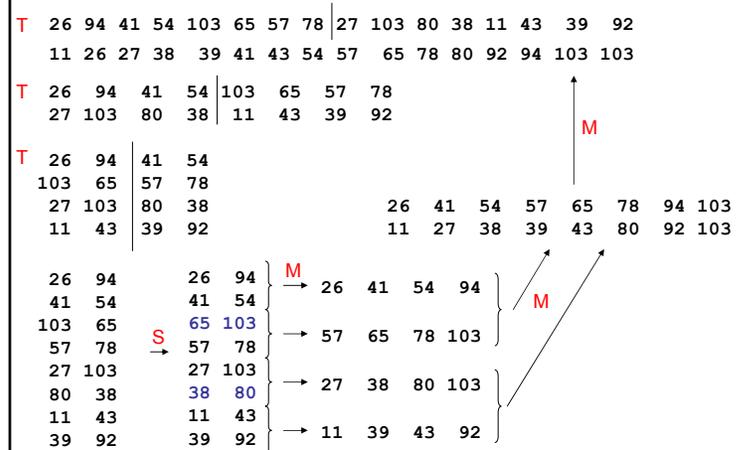
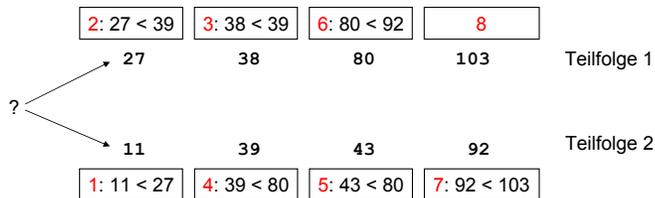
1. minimale automatische Typprüfung möglich (statisch / dynamisch);
2. man sich mehr Gedanken darüber macht, was man eigentlich tut;
3. für Außenstehende präziser angezeigt wird, was Sie tun.

Wenn im Code ein Cast notwendig ist,
dann die Cast-Operation in einer Funktion verbergen.

Mergesort

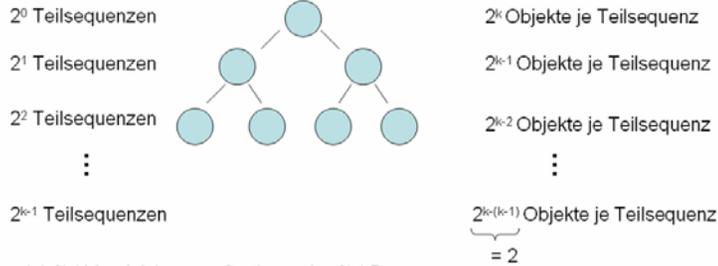
Beobachtung:

Sortieren ist einfach, wenn man zwei sortierte Teilfolgen hat.



Laufzeitanalyse

Annahme: Anzahl Objekte $n = 2^k \Leftrightarrow k = \log_2 n$



- (a) 2^{k-1} Vergleiche zum Sortieren der 2^{k-1} Paare
- (b) auf Ebene e : $(2^{k-e}-1)$ Vergleiche zum Mischen von 2 der 2^e Sequenzen
 $\Rightarrow (2^{k-e}-1) * 2^{e-1} = 2^{k-1} - 2^{e-1}$ Vergleiche auf Ebene $e = 1, \dots, k-1$
 $\Rightarrow 2^{k-1} + (k-1)*2^{k-1} - \text{Summe}(2^{0-1}; 1..k-1) = (k-1)*2^{k-1} + 1 < k*2^k = n \log_2 n$

Mergesort

- Eingabe: unsortiertes Feld von Zahlen
- Ausgabe: sortiertes Feld
- Algorithmisches Konzept: „Teile und herrsche“ (*divide and conquer*)
 - Zerlege Problem solange in Teilprobleme bis Teilprobleme lösbar
 - Löse Teilprobleme
 - Füge Teilprobleme zur Gesamtlösung zusammen

Hier:

1. Zerteile Feld in Teilfelder bis Teilproblem lösbar (\rightarrow bis Feldgröße = 2)
2. Sortiere Felder der Größe 2 (\rightarrow einfacher Vergleich zweier Zahlen)
3. Füge sortierte Teilfelder durch Mischen zu sortierten Feldern zusammen

Mergesort

- Programmwurf

1. Teilen eines Feldes \rightarrow einfach!
2. Sortieren
 - a) eines Feldes der Größe 2 \rightarrow einfach!
 - b) eines Feldes der Größe $> 2 \rightarrow$ rekursiv durch Teilen & Mischen
3. Mischen \rightarrow nicht schwer!

Annahme:
Feldgröße ist
Potenz von 2

Mergesort: Version 1

```
void Msort(const int size, int a[]) {
    if (size == 2) { // sortieren
        if (a[0] > a[1]) Swap(a[0], a[1]);
        return;
    }
    // teilen
    int k = size / 2;
    Msort(k, &a[0]);
    Msort(k, &a[k]);
    // mischen
    Merge(k, &a[0], &a[k]);
}

void Swap(int& a, int& b) {
    int c = b; b = a; a = c;
}
```

} sortieren (einfach)

} sortieren durch
Teilen & Mischen

} Werte vertauschen
per Referenz



Mergesort: Version 1

```
void Merge(const int size, int a[], int b[]) {
    int* c = new int[2*size];
    // mischen
    int i = 0, j = 0;
    for (int k = 0; k < 2 * size; k++)
        if ((j == size) || (i < size && a[i] < b[j]))
            c[k] = a[i++];
        else
            c[k] = b[j++];
    // umkopieren
    for (int k = 0; k < size; k++) {
        a[k] = c[k];
        b[k] = c[k+size];
    }
    delete[] c;
}
```

← dynamischen Speicher anfordern

← dynamischen Speicher freigeben



Mergesort: Version 1

```
void Print(const int size, int a[]) {
    for (int i = 0; i < size; i++) {
        cout << a[i] << "\t";
        if ((i+1) % 8 == 0) cout << endl;
    }
    cout << endl;
}

int main() {
    const int size = 32;
    int a[size];
    for (int k = 0; k < size; k++) a[k] = rand();
    Print(size, a);
    Msort(size, a);
    Print(size, a);
}
```

Hilfsfunktion für Testprogramm

Programm zum Testen



Mergesort: Version 1

Ausgabe:

41	18467	6334	26500	19169	15724	11478	29358
26962	24464	5705	28145	23281	16827	9961	491
2995	11942	4827	5436	32391	14604	3902	153
292	12382	17421	18716	19718	19895	5447	21726
41	153	292	491	2995	3902	4827	5436
5447	5705	6334	9961	11478	11942	12382	14604
15724	16827	17421	18467	18716	19169	19718	19895
21726	23281	24464	26500	26962	28145	29358	32391

OK, funktioniert für int ... was ist mit char, float, double ... ?

⇒ Idee: Schablonen!



Mergesort: Version 2

```
template <class T> void Msort(const int size, T a[]) {
    if (size == 2) { // sortieren
        if (a[0] > a[1]) Swap<T>(a[0], a[1]);
        return;
    }
    // teilen
    int k = size / 2;
    Msort<T>(k, &a[0]);
    Msort<T>(k, &a[k]);
    // mischen
    Merge<T>(k, &a[0], &a[k]);
}

template <class T> void Swap(T& a, T& b) {
    T c = b; b = a; a = c;
}
```



Mergesort: Version 2

```
template <class T> void Merge(const int size, T a[], T b[]) {
    T* c = new T[2*size];
    // mischen
    int i = 0, j = 0;
    for (int k = 0; k < 2 * size; k++) {
        if ((j == size) || (i < size && a[i] < b[j]))
            c[k] = a[i++];
        else
            c[k] = b[j++];
    }
    // umkopieren
    for (int k = 0; k < size; k++) {
        a[k] = c[k];
        b[k] = c[k+size];
    }
    delete[] c;
}
```



Mergesort: Version 2

```
template <class T> void Print(const int size, T a[]) { ... }
```

```
int main() {
    const int size = 32;
    int a[size];
    for (int k = 0; k < size; k++) a[k] = rand();
    Print<int>(size, a);
    Msort<int>(size, a);
    Print<int>(size, a);
    float b[size];
    for (int k = 0; k < size; k++) b[k] = rand() * 0.01f;
    Print<float>(size, b);
    Msort<float>(size, b);
    Print<float>(size, b);
}
```

Konstante vom Typ float (nicht double)



Mergesort: Version 2

Ausgabe:

41	18467	6334	26500	19169	15724	11478	29358
26962	24464	5705	28145	23281	16827	9961	491
2995	11942	4827	5436	32391	14604	3902	153
292	12382	17421	18716	19718	19895	5447	21726
41	153	292	491	2995	3902	4827	5436
5447	5705	6334	9961	11478	11942	12382	14604
15724	16827	17421	18467	18716	19169	19718	19895
21726	23281	24464	26500	26962	28145	29358	32391
147.71	115.38	18.69	199.12	256.67	262.99	170.35	98.94
287.03	238.11	313.22	303.33	176.73	46.64	151.41	77.11
282.53	68.68	255.47	276.44	326.62	327.57	200.37	128.59
87.23	97.41	275.29	7.78	123.16	30.35	221.9	18.42
7.78	18.42	18.69	30.35	46.64	68.68	77.11	87.23
97.41	98.94	115.38	123.16	128.59	147.71	151.41	170.35
176.73	199.12	200.37	221.9	238.11	255.47	256.67	262.99
275.29	276.44	282.53	287.03	303.33	313.22	326.62	327.57