

## Wiederholung Operatoren

## Überladen von Operatoren, Einleitung

Mit Hilfe des Überladens von Operatoren können fast alle Operatoren für selbstdefinierte Klassen mit einer neuen Bedeutung versehen werden. Um nun den Mechanismus des Operator-Overloading zu verstehen, betrachte man folgende Beispiele:

```
a = b;      --> a.operator= (b);  
a + b;     --> a.operator+ (b);  
a += b;    --> a.operator+= (b);  
!a;        --> a.operator! ();
```

Das Überladen kann auch unter Verwendung von globalen Funktionen erfolgen, z.B.:

```
a = b;      --> operator= (a, b);  
a + b;     --> operator+ (a, b);  
a += b;    --> operator+= (a, b);  
cout << s  --> operator<< (cout, s)
```

## Überladen von Operatoren, Einleitung

Das Überladen kann auch unter Verwendung von globalen Funktionen erfolgen, z.B.:

```
a = b;      --> operator=  (a, b);  
a + b;     --> operator+  (a, b);  
a += b;    --> operator+= (a, b);  
cout << s  --> operator<< (cout, s)
```

# Überladen von Operatoren, Übersicht

## Überladbare Operatoren:

- `->` `->`
- `[]` `()`
- `new` `delete` `delete[]`
- `!` `~` `++` `--` `+` `-` `*` `&` (*unär*)
- `*` `/` `%` `+` `-`
- `<<` `>>`
- `<` `<=` `>` `>=` `==` `!=`
- `&` `^` `|`
- `&&` `||`
- `=` `+=` `-=` `*=` `/=` `%=` `&=` `^=` `|=`
- `<<=` `>>=`
- `,`

## Nicht überladbare Operatoren:

- `.`
- `.*`
- `::`
- `?:`
- `sizeof`
- `static_cast`
- `const_cast`
- `reinterpret_cast`
- `dynamic_cast`
- `typeid`



## Funktionsobjekte (1)

```
class Func {  
public:  
    double operator()(double x)  
        { return x*x; }  
};  
int main() {  
    Func f;  
    cout << f(5) << endl;}
```

```
void Print(double x, Func f)  
{  
    cout << f(x) << endl;  
}  
...  
Print(5, f);
```

Funktionsobjekte sind Objekte, die wie Funktionen benutzt werden. Sie zeichnen sich dadurch aus, dass sie eine Elementfunktion haben, die den **Funktionsoperator „()“** überlädt. Funktionsobjekte können natürlich auch als Parameter von Funktionen verwendet werden, z.B. wie folgt:



## Funktionsobjekte (2)

```
class Func {  
    double a, b, c; // Polynominalkoeffizienten  
public:  
    Func(double a1, double b1=0, double c1=0) {  
        a = a1; b = b1; c = c1;  
    }  
    double operator()(double x) const { return a*x*x + b*x + c; }  
};
```

```
int main() {  
    Func f(3, -2, -7);  
    cout << f(5) << endl;  
}
```

Ein wesentlicher Vorteil von Funktionsobjekten gegenüber Funktionen liegt darin, dass sie über ein *Gedächtnis* in Form von internen Variablen verfügen können, die über einen Konstruktor initialisierbar sind.

Funktionsobjekte sind die objektorientierte Alternative zu globalen Funktionen!

## Einschub: Klammer-Operator

```
inline bool justEqual(int al, int ar) {  
    return al == ar;  
}  
array<int, 7> test = { 1, 1, 2, 3, 3, 3, 5 };  
    // Alle aufeinanderfolgenden Doubletten entfernen  
auto end = unique(test.begin(), test.end(), justEqual);
```

## Was gibt folgendes Programm aus?

```
void uniqueWithFunc() {  
    array<int, 8> test = { 1, 1, 2, 3, 3, 3, 2, 5 };  
    PrintIntervall(test.begin(), test.end());  
    // Alle aufeinanderfolgenden Doubletten entfernen  
    auto end = unique(test.begin(),  
        test.end(), justEqual);  
    PrintIntervall(test.begin(), test.end());  
    PrintIntervall(test.begin(), end);  
}
```

```
inline bool justEqual(int al, int ar) {  
    return al == ar;  
}
```

```
template <typename It>  
void PrintIntervall(It beg, It end) {  
    copy(beg, end,  
        ostream_iterator<int>(cout, ";"));  
    cout << "\n";  
}
```

## Was gibt folgendes Programm im letzten Print-Aufruf aus?

```
void uniqueWithFunc() {  
    array<int, 8> test = { 1, 1, 2, 3, 3, 3, 2, 5 };  
    PrintIntervall(test.begin(), test.end());  
    // Alle aufeinanderfolgenden Doubletten entfernen  
    auto end = unique(test.begin(),  
        test.end(), justEqual);  
    PrintIntervall(test.begin(), test.end());  
    PrintIntervall(test.begin(), end);  
}
```

Ausgabe?

1. 1;2;3;2;5;
2. 1;2;3;5;
3. 1;1;2;3;3;3;2;5;
4. 1;2;3;2;5;3;2;5;

```
template <typename It>  
void PrintIntervall(It beg, It end) {  
    copy(beg, end,  
        ostream_iterator<int>(cout, ";"));  
    cout << "\n";  
}
```

```
inline bool justEqual(int al, int ar) {  
    return al == ar;  
}
```

## Was gibt folgendes Programm im letzten Print-Aufruf aus?

```
void uniqueWithFunc() {  
    array<int, 8> test = { 1, 1, 2, 3, 3, 3, 2, 5 };  
    PrintIntervall(test.begin(), test.end());  
    // Alle aufeinanderfolgenden Doubletten entfernen  
    auto end = unique(test.begin(),  
        test.end(), justEqual);  
    PrintIntervall(test.begin(), test.end());  
    PrintIntervall(test.begin(), end);  
}
```

Ausgabe?

1. 1;2;3;2;5;
2. 1;2;3;5;
3. 1;1;2;3;3;3;2;5;
4. 1;2;3;2;5;3;2;5;

```
template <typename It>  
void PrintIntervall(It beg, It end) {  
    copy(beg, end,  
        ostream_iterator<int>(cout, ";"));  
    cout << "\n";  
}
```

```
inline bool justEqual(int al, int ar) {  
    return al == ar;  
}
```

Ausgabe?

1. 1;2;3;2;5;
2. 1;2;3;5;
3. 1;1;2;3;3;3;2;5;
4. 1;2;3;2;5;3;2;5;

## Das gleiche mit einer Klasse

```
void uniqueWithOperator() {  
    array<int, 8> test = { 1, 1, 2, 3, 3, 3, 2, 5 };  
    PrintIntervall(test.begin(), test.end());  
    auto end = unique(test.begin(),  
        test.end(), JustEqual());  
    PrintIntervall(test.begin(), test.end());  
    PrintIntervall(test.begin(), end);  
}
```

Klammern wichtig  
Instanz wird erzeugt

```
class JustEqual {  
public:  
    bool operator() (int a, int b) {  
        return a == b; }  
};
```

```
template <typename It>  
void PrintIntervall(It beg, It end) {  
    copy(beg, end,  
        ostream_iterator<int>(cout, ";"));  
    cout << "\n";  
}
```

## Was gibt folgendes Programm im letzten Print-Aufruf aus?

```
void uniqueWithOperator() {  
    array<int, 8> test = { 1, 1, 2, 3, 3, 3, 2, 5 };  
    PrintIntervall(test.begin(), test.end());  
    auto end = unique(test.begin(),  
        test.end(), JustEqual());  
    PrintIntervall(test.begin(), test.end());  
    PrintIntervall(test.begin(), end);  
}
```

Ausgabe?

1. 1;2;3;2;5;
2. 1;2;3;5;
3. 1;1;2;3;3;3;2;5;
4. 1;2;3;2;5;3;2;5;

```
class JustEqual {  
public:  
    bool operator() (int a, int b) {  
        return a == b; }  
};
```

```
template <typename It>  
void PrintIntervall(It beg, It end) {  
    copy(beg, end,  
        ostream_iterator<int>(cout, ";"));  
    cout << "\n";  
}
```

Ausgabe?

1. 1;2;3;2;5;
2. 1;2;3;5;
3. 1;1;2;3;3;3;2;5;
4. 1;2;3;2;5;3;2;5;

## Aufgabe 2: Implementieren Sie eine Klasse EqualButNot

```
array<int, 8> test1 = { 1, 1, 2, 3, 3, 3, 2, 5 };  
auto end1 = unique(test1.begin(),  
    test1.end(), EqualButNot(3));  
PrintIntervall(test1.begin(), end1);
```

Ausgabe soll sein:  
1;2;3;3;3;2;5;  
1;1;2;3;2;5;;

```
array<int, 8> test = { 1, 1, 2, 3, 3, 3, 2, 5 };  
auto end = unique(test.begin(),  
    test.end(), EqualButNot(1));  
PrintIntervall(test.begin(), end);
```

```
// Implementieren Sie EqualButNot, sodass unique nur  
// aufeinanderfolgende Dubletten ungleich der Zahl im  
// Konstruktor entfernt
```

## Aufgabe 2: Implementieren Sie eine Klasse EqualButNot

```
array<int, 8> test1 = { 1, 1, 2, 3, 3, 3, 2, 5 };  
auto end1 = unique(test1.begin(),  
    test1.end(), EqualButNot(3));  
PrintIntervall(test1.begin(), end1);
```

```
array<int, 8> test = { 1, 1, 2, 3, 3, 3, 2, 5 };  
auto end = unique(test.begin(),  
    test.end(), EqualButNot(1));  
PrintIntervall(test.begin(), end);
```

Ausgabe soll sein:

1;2;3;3;3;2;5;

1;1;2;3;2;5;;

**13. Woche; Mo. 13.12.2021**

**Vorlesung**

[Lambda-Ausdrücke \(13.12.2020\)](#)

siehe vorherige Woche;

**Übungsaufgaben**

[Aufgabe EqualButNot \(11.12.2021\)](#)

```
// Erweitern Sie EqualButNot, sodass  
// Sie im Konstruktor mehrere Werte übergeben können  
// z.B. als vector
```

# Die verschiedenen Programmierparadigmen von C++

The screenshot displays the Visual Studio IDE with a project named "ArrayNegate". The code in the main editor is as follows:

```
19 // Ohne die folgende Änderung ist ihr Programm nicht
20 // kompilierbar
21 // Implementieren Sie EqualButNot, sodass
22 //unique nur aufeinanderfolgende Dubletten entfernt
23 //Konstruktor entfernt
24 void uniqueButNot() {
25     cout << "\nOriginal Array\n";
26     array<int, 8> test2 = { 1, 1, 2, 3, 3, 3, 3, 3 };
27     PrintIntervall(test2.begin(), test2.end());
28     cout << "\n Mehr ist noch nicht fertig, Sie sind dran";
29
30 #ifdef READY_FOR_AUFGABE02
31     auto end2= unique(test2.begin(),
32                       test2.end(), EqualButNot(9));
33     PrintIntervall(test2.begin(), end2);
34
35     array<int, 8> test1 = { 1, 1, 2, 3, 3, 3, 3, 3 };
36     auto end1 = unique(test1.begin(),
37                       test1.end(), EqualButNot(3));
38     PrintIntervall(test1.begin(), end1);
39
40     array<int, 8> test = { 1, 1, 2, 3, 3, 3, 3, 3 };
41     auto end = unique(test.begin(),
42                      test.end(), EqualButNot(1));
43     PrintIntervall(test.begin(), end);
44
45 #endif // READY_FOR_AUFGABE02
```

The console window on the right shows the following output:

```
Aufgabe 2:
Original Array
1;1;2;3;3;3;2;5;
Mehr ist noch nicht fertig, Sie sind dran
C:\R:\Vorlesungsunterlagen\Vorlesung\Folien\Vorles12\
mit Code "0" beendet.
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.
```

Red circles in the image highlight the `#ifdef READY_FOR_AUFGABE02` preprocessor directive in the code and the corresponding output in the console.

Die verschiedenen Programmierparadigmen von C++

## **Lambda-Ausdrücke**

## Motivation von Lambda-Ausdrücken

```
class Guest{  
public:  
    int entryTime;  
    int usageTime;  
    string name;  
    int earlyTime;  
};
```

Natürlich sollten Attribute **nicht** public sein!

Wir können uns dadurch im Folgenden aber auf das Wesentliche konzentrieren!

Sortierung nach entryTime

```
ostream& operator<<(ostream& str, const Guest& g){  
    str << g.name << ":" << setw(3) << g.entryTime <<  
        " (" << setw(3) << g.earlyTime << ","  
        << setw(3) << g.usageTime << ")";  
    return str;  
}
```

```
bool operator< (const Guest& g1, const Guest& g2){  
    return g1.entryTime < g2.entryTime;  
}
```

## Motivation von Lambda-Ausdrücken (2)

```
- bool operator< (const Guest& g1, const Guest& g2){  
    return g1.entryTime < g2.entryTime;  
}
```

```
- class Guest{  
public:  
    int entryTime;  
    int usageTime;  
    string name;  
    int earlyTime;  
};
```

Sortierung nach entryTime

## Motivation von Lambda-Ausdrücken (2)

```
void sortByName(){  
    vector<Guest> gArr = {  
        { 25, 3, "Hans", 16 },  
        { 13, 4, "Otto", 13 },  
        { 5, 2, "Karl", 3 },  
        { 26, 13, "Anna", 26 }  
    };  
    sort(gArr.begin(), gArr.end());  
    copy(gArr.begin(), gArr.end(),  
        ostream_iterator<Guest>(cout, "\n"));  
}
```

```
Karl: 5 ( 3, 2)  
Otto: 13 ( 13, 4)  
Hans: 25 ( 16, 3)  
Anna: 26 ( 26, 13)
```

Das geht einfach, aber wie verwenden wir ein anderes Sortierkriterium, z.B. alphabetische Sortierung nach „name“?

## Motivation von Lambda-Ausdrücken (3)

```
void sortByName(){  
    vector<Guest> gArr = {  
        { 25, 3, "Hans", 16 },  
        { 13, 4, "Otto", 13 },  
        { 5, 2, "Karl", 3 },  
        { 26, 13, "Anna", 26 }  
    };  
    struct SorterByName  
    : public std::binary_function<Guest, Guest, bool> {  
        bool operator()(const Guest& g1, const Guest& g2) {  
            return g1.name < g2.name;  
        }  
    };  
    // (), da Konstruktor aufgerufen wird  
    sort(gArr.begin(), gArr.end(), SorterByName());  
    copy(gArr.begin(), gArr.end(),  
        ostream_iterator<Guest>(cout, "\n"));  
}
```

```
Anna: 26 ( 26, 13)  
Hans: 25 ( 16, 3)  
Karl: 5 ( 3, 2)  
Otto: 13 ( 13, 4)
```

## Motivation von Lambda-Ausdrücken (4)

```
void sortByName(){  
    vector<Guest> gArr = {  
        { 25, 3, "Hans", 16 },  
        { 13, 4, "Otto", 13 },  
        { 5, 2, "Karl", 3 },  
        { 26, 13, "Anna", 26 }  
    };  
}
```

```
Hans: 25 ( 16, 3)  
Karl: 5 ( 3, 2)  
Otto: 13 ( 13, 4)  
Anna: 26 ( 26, 13)  
  
Anna: 26 ( 26, 13)  
Hans: 25 ( 16, 3)  
Otto: 13 ( 13, 4)  
Karl: 5 ( 3, 2)
```

// Anna als letzte

```
sort(gArr.begin(), gArr.end(), NameSorterIgnore("Anna"));  
copy(gArr.begin(), gArr.end(),  
      ostream_iterator<Guest>(cout, "\n"));
```

cout << "\n";

// Karl als letzter

```
sort(gArr.begin(), gArr.end(), NameSorterIgnore("Karl"));  
copy(gArr.begin(), gArr.end(),  
      ostream_iterator<Guest>(cout, "\n"));
```

## Motivation von Lambda-Ausdrücken (4)

```
...  
// Karl als letzter  
sort(gArr.begin(), gArr.end(), NameSorterIgnore("Karl"));
```

```
Hans: 25 ( 16, 3)  
Karl: 5 ( 3, 2)  
Otto: 13 ( 13, 4)  
Anna: 26 ( 26, 13)  
Anna: 26 ( 26, 13)  
Hans: 25 ( 16, 3)  
Otto: 13 ( 13, 4)  
Karl: 5 ( 3, 2)
```

```
struct NameSorterIgnore  
{  
    : public std::binary_function<Guest, Guest, bool> {  
        NameSorterIgnore(string ign) : m_ign(ign){}  
        bool operator()(const Guest& g1, const Guest& g2) {  
            if (g1.name == m_ign) {return false;}  
            if (g2.name == m_ign) {return true;}  
            return g1.name < g2.name;  
        }  
        string m_ign;  
};
```

Geht das auch anders als jedes Mal eine weitere Klasse zu „erfinden“?

## Einsatz von Lambda-Ausdrücken

```
--  
sort(gArr.begin(), gArr.end(),  
    [](const Guest& g1, const Guest& g2) {  
    return g1.name < g2.name; });
```

Beispiel für Einsatz von einem Lambda-Ausdruck

**Vorteil:** Es muss nicht immer eine eigene Klasse erzeugt werden, sondern an der Stelle, an der Funktionalität benötigt wird, wird sie implementiert.

## Lambda-Ausdrücke

- Lambda Ausdrücke ermöglichen die Definition von (einmalig nutzbaren) anonymen Funktionsobjekten anstatt eine benannte Klasse mit operator() zu definieren (siehe Stroustrup, S.317ff)



## Erfassungsliste

```
..  
string ign = "Anna";  
sort(gArr.begin(), gArr.end(),  
    [](const Guest& g1, const Guest& g2) {  
    if (g1.name == ign) { return false; }  
    if (g2.name == ign) { return true; }  
    return g1.name < g2.name;  
    }  
);
```

Syntaxfehler, ign innen nicht bekannt

## Erfassungsliste (2)

```
string fir = "Karl"; // Karl zuerst
sort(gArr.begin(), gArr.end(),
    [fir](const Guest& g1, const Guest& g2) {
    if (g1.name == fir) { return true; }
    if (g2.name == fir) { return false; }
    return g1.name < g2.name;
});
```

fir wird als Werteparameter  
bekannt gemacht

(fir = first)

```
Karl: 5 ( 3, 2)
Anna: 26 ( 26, 13)
Hans: 25 ( 16, 3)
Otto: 13 ( 13, 4)
```

## Erfassungsliste (3)

```
--  
string fir = "Karl"; // Karl zuerst  
sort(gArr.begin(), gArr.end(),  
    [fir](const Guest& g1, const Guest& g2) {  
    if (g1.name == fir) { g1.earlyTime=-1;return true;}  
    if (g2.name == fir) { g2.earlyTime=-3;return false;}  
    return g1.name < g2.name;  
}  
);
```

Syntaxfehler g1 ist const

## Erfassungsliste mit Referenzparameter

```
string fir = "Karl"; // Karl zuerst
sort(gArr.begin(), gArr.end(),
     [fir](const Guest& g1, const Guest& g2) {
     if (g1.name == fir) { fir="Tony";return true;}
     if (g2.name == fir) { fir="PaPa";return false;}
     return g1.name < g2.name;
     }
);
```

## Erfassungsliste mit Referenzparameter

```
string fir = "Karl"; // Karl zuerst
sort(gArr.begin(), gArr.end(),
    [&fir](const Guest& g1, const Guest& g2) {
    if (g1.name == fir) { fir="Tony";return true;}
    if (g2.name == fir) { fir="PaPa";return false;}
    return g1.name < g2.name;
});
```

Syntaktisch richtig: gute Debugging-Unterstützung der STL sollte hier aber eine Ausnahme werfen, denn

$a1 < a2 \rightarrow$  NICHT  $a2 < a1$

wird verletzt

## Erfassungsliste mit Referenzparameter (2)

```
array<int, 8> test1 = { 1, 1, 2, 3, 3, 3, 2, 5 };
auto end1 = unique(test1.begin(),
    test1.end(), EqualButNot(3));
PrintIntervall(test1.begin(), end1);

array<int, 8> test = { 1, 1, 2, 3, 3, 3, 2, 5 };
auto end = unique(test.begin(),
    test.end(), EqualButNot(1));
PrintIntervall(test.begin(), end);

}
if (g2.name == fir) {
    dum = "PaPa"; return false;
}
return g1.name < g2.name;
}
);
```

zuerst

g1, const Guest& g2) {

};

```
Karl: 5 ( 3, 2)
Anna: 26 ( 26, 13)
Hans: 25 ( 16, 3)
Otto: 13 ( 13, 4)
dum = PaPa
```

## Aufgabe 3: Implementieren Sie die vorherige Aufgabe unter Verwendung eines Lambda-Ausdrucks

Ausgabe soll sein:

1;2;3;3;3;2;5;

1;1;2;3;2;5;;

```
array<int, 8> test1 = { 1, 1, 2, 3, 3, 3, 2, 5 };  
auto end1 = unique(test1.begin(),  
    test1.end(), EqualButNot(3));  
PrintIntervall(test1.begin(), end1);
```

```
array<int, 8> test = { 1, 1, 2, 3, 3, 3, 2, 5 };  
auto end = unique(test.begin(),  
    test.end(), EqualButNot(1));  
PrintIntervall(test.begin(), end);
```

Sie sind dran.

## Referenzliste

- []: Es werden keine Variablen der Umgebung benutzt
- [&]: alle sichtbaren Variablen der Umgebung sind per Referenz zugänglich
- [&x]: nur x per Referenz zugänglich
- [=]: alle sichtbaren Variablen der Umgebung sind per Wert zugänglich und sind const
- [x]: nur x per const Wert zugänglich
- [=, &x, &y]: x, y als Referenz, Rest als Wert
- [&, x, y]: alles als Referenz, nur x und y als const Werte
- [x=42]: Innerhalb des Lambda-Ausdruck ist x eine Konstante mit Wert 42.
- [this]: Wenn Lambda innerhalb einer Klasse definiert wird, hat man per Referenz Zugriff auf die einzelnen Attribute

## mutable

```
- void algo(vector<int>& v){  
    auto count = v.size();  
    std::generate(v.begin(), v.end(),  
                 [count]() {return --count; }  
    );  
}
```

count ist als const Wert übergeben.

## mutable (2)



3 2 1 0

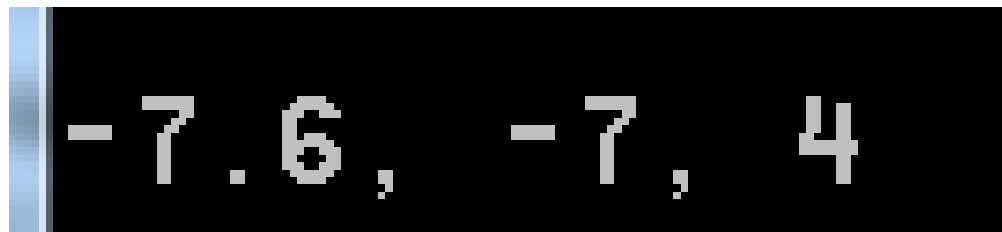
```
void algo(vector<int>& v){
    auto count = v.size();
    std::generate(v.begin(), v.end(),
                 [count]() mutable {return --count; }
    );
}

void callAlgo(){
    vector<int> arr = { 1, 2, 3, -6 };
    algo(arr);
    copy(arr.begin(), arr.end(),
         ostream_iterator<int>(cout, " "));
}
```

generate(b,e,f) weist jedem Element im Bereich von [b..e[ das Element f() zu.

## Rückgabewerte

```
void lambdaReturn(){  
    auto lambda1 = [](double x, double y){  
        return (x < y ? x : y); };  
    cout << lambda1(4.3, -7.6) << ", ";  
    auto lambda2= [](double x, double y)-> int {  
        return (x < y ? x : y); };  
    cout << lambda2(4.3, -7.6) << ", ";  
    cout << lambda2(4, 5);  
}
```



-7.6, -7, 4

## Badeintrittszeitberechnung

Auf den folgenden Folien finden Sie eine Klasse Guest.

Sie dient zur Planung der Badeintrittszeiten von Gästen.

Von jedem Gast kennt man, seinen Namen, wie lange er im Bad braucht und wann er frühestens aufsteht.

Gesucht: Bei gegebener Reihenfolge. Wann kann dann jeder Gast frühestens ins Bad und wie lange ist die Summe der Wartezeiten?

Was ist eine „optimale“ Reihenfolge.

```
class Guest{  
public:  
    int entryTime;  
    int usageTime;  
    string name;  
    int earlyTime;  
};
```

## Rückgabewerte

sum of waiting times 11

```
void calcSumWaitingTimes(){
    vector<Guest> gArr = {
        { 25, 3, "Hans", 16 },
        { 13, 4, "Otto", 13 },
        { 5, 2, "Karl", 3 },
        { 26, 13, "Anna", 26 }
    };
    int sum = 0;
    for_each(gArr.begin(), gArr.end(),
        [&sum](const Guest& g) {
            sum += g.entryTime - g.earlyTime ; }
    );
    cout << "sum of waiting times " << sum;
    cout << "\n";
}
```

```
class Guest{
public:
    int entryTime;
    int usageTime;
    string name;
    int earlyTime;
};
```

Siehe Code in Datei Guest.cxx

## Ermittlung der Badeintrittszeit für gegebene Reihenfolge

```
void calcEntryTimes(){  
    vector<Guest> gArr = {  
        { -1, 3, "Hans", 16 },  
        { -1, 4, "Otto", 13 },  
        { -1, 2, "Karl", 3 },  
        { -1, 13, "Anna", 26 }  
    };  
};
```

```
Hans: 16 ( 16, 3)  
Otto: 19 ( 13, 4)  
Karl: 23 ( 3, 2)  
Anna: 26 ( 26, 13)
```

```
// lastGuest
```

```
Guest lG = { -1, 0, "", 0 };  
for_each(gArr.begin(), gArr.end(),  
    [&lG](Guest& g) {  
        int leaveTime = lG.entryTime + lG.usageTime;  
        g.entryTime = max(leaveTime, g.earlyTime);  
        lG = g;}  
);  
copy(gArr.begin(), gArr.end(),  
    ostream_iterator<Guest>(cout, "\n"));  
cout << "\n";
```

## Ermittlung der Badeintrittszeit für gegebene Reihenfolge (2)

```
void reuseLambda(){
    vector<Guest> gArr = {
        { -1, 3, "Hans", 16 },
        { -1, 4, "Otto", 13 },
        { -1, 2, "Karl", 3 },
        { -1, 13, "Anna", 26 }
    };

    // lastGuest
    Guest lG = { -1, 0, "", 0 };
    auto calcEntryTimes = [&lG](Guest& g) {
        int leaveTime = lG.entryTime + lG.usageTime;
        g.entryTime = max(leaveTime, g.earlyTime);
        lG = g; };
}
```

```
int sum = 0;
auto sumOfWaitingTimes = [&sum](const Guest& g) {
    sum += g.entryTime - g.earlyTime; };
```

```
Name Ent Ear, Usa
Hans: 16 ( 16, 3)
Otto: 19 ( 13, 4)
Karl: 23 ( 3, 2)
Anna: 26 ( 26, 13)
sum of waiting times 26
```

```
for_each(gArr.begin(), gArr.end(), calcEntryTimes);
cout << "Name Ent Ear, Usa\n";
copy(gArr.begin(), gArr.end(),
      ostream_iterator<Guest>(cout, "\n"));
for_each(gArr.begin(), gArr.end(), sumOfWaitingTimes);
cout << "sum of waiting times " << sum;
```

## Ermittlung der Badeintrittszeit für gegebene Reihenfolge (3)

```
// Sortierung nach fruehester Aufstehzeit
sort(gArr.begin(), gArr.end(),
     [](const Guest& g1, const Guest& g2) {
     return g1.earlyTime < g2.earlyTime; });

IG = { -1, 0, "", 0 };
for_each(gArr.begin(), gArr.end(), calcEntryTimes);
cout << "\nEintrittzeiten, wenn nach Aufstehzeit sortiert:\n";
cout << "Name Ent Ear, Usa\n";
copy(gArr.begin(), gArr.end(),
     ostream_iterator<Guest>(cout, "\n"));
sum = 0;
for_each(gArr.begin(), gArr.end(), sumOfWaitingTimes);

cout << "sum of waiting times " << sum;

cout << "\n";
}
```

```
Eintrittzeiten, wenn nach Aufstehzeit sortiert:
Name Ent Ear, Usa
Karl: 3 ( 3, 2)
Otto: 13 ( 13, 4)
Hans: 17 ( 16, 3)
Anna: 26 ( 26, 13)
sum of waiting times 1
```