Organisation des Speichers

Gültigkeitsbereiche von Variablen

Variablennamen haben feste Gültigkeitsbereiche

- Was ist die Ausgabe des folgenden Programms?

```
void foo(int x) {
    x += 42;
    int y = 146;
    cout << x << y << endl;
}

int main() {
    int x = 42; int y = 23;
    cout << x << y << endl;
    foo(x);
    cout << x << y << endl;

    return 0;
}
```

Gültigkeitsbereiche von Variablen

Variablennamen haben feste Gültigkeitsbereiche

- Die Variablen `x` und `y` innerhalb von `foo` sind nicht identisch mit `x` und `y` im Hauptptogramm.

Gültigkeitsbereiche von Variablen

Variablennamen haben feste Gültigkeitsbereiche

- Die Variablen `x` und `y` innerhalb von `foo` sind nicht identisch mit `x` und `y` im Hauptptogramm.
- Beim Aufruf der Funktion wird der Parameter `x` kopiert.
- Auch bei der Deklaration von `y` wird eine neue Variable erzeugt.

Gültigkeitsbereiche von Variablen

Variablennamen haben feste Gültigkeitsbereiche

- Die Variablen `x` und `y` innerhalb von `foo` sind nicht identisch mit `x` und `y` im Hauptptogramm.
- Beim Aufruf der Funktion wird der Parameter `x` kopiert.
- Auch bei der Deklaration von `y` wird eine neue Variable erzeugt.
- Die ursprünglichen `x` und `y` sind innerhalb der Funktion `foo` nicht sichtbar.
- Vorteil: Namenskonflikte werden vermieden.

Gültigkeitsbereiche von Variablen

Variablennamen haben feste Gültigkeitsbereiche

- Das Gleiche gilt für `if`-Blöcke und Schleifen:

```
int x = 42;

for (int y=0;y<5;y++) {
    int x = y * 2;
    cout << x;
}
cout << x << endl;
```

Gültigkeitsbereiche von Variablen

Variablennamen haben feste Gültigkeitsbereiche

- Variablen sind nur in ihrem jeweiligen Block sichtbar:

```
int x = 42; {
    int x = 2; {
        int x = 4; {
            int x = 6;
            cout << x << endl;
        }
        cout << x << endl;
    }
    cout << x << endl;
}
cout << x << endl;
```

Stack und Heap

Der Speicher ist in zwei Bereiche unterteilt: Stack und Heap

Stack und Heap

Der Speicher ist in zwei Bereiche unterteilt: Stack und Heap

Stack

- Enthält lokale Variablen, die in Funktionen und Blöcken deklariert wurden.

Stack und Heap

Der Speicher ist in zwei Bereiche unterteilt: Stack und Heap

Stack

- Enthält lokale Variablen, die in Funktionen und Blöcken deklariert wurden.
- Ist in Stack-Frames unterteilt, die die jeweils gültigen Variablen enthalten.
- Stack-Frames werden beim Aufruf von Funktionen erzeugt und beim Rücksprung gelöscht.

Stack und Heap

Der Speicher ist in zwei Bereiche unterteilt: Stack und Heap

Stack

- Enthält lokale Variablen, die in Funktionen und Blöcken deklariert wurden.
- Ist in Stack-Frames unterteilt, die die jeweils gültigen Variablen enthalten.
- Stack-Frames werden beim Aufruf von Funktionen erzeugt und beim Rücksprung gelöscht.

Heap

- Enthält globale Variablen.

Stack und Heap

Beispiel: Stack-Frames

```
void bar(int x) {
    x += 77;
    cout << x << endl;
}

void foo() {
    int x = 23;
    bar(x);
    cout << x << endl;
}

int main() {
    int x = 42;
    foo();
    cout << x << endl;
    return 0;
}
```

Stack und Heap

Beispiel: Stack-Frames

```
void bar(int x) {
    x += 77;
    cout << x << endl;
}

void foo() {
    int x = 23;
    bar(x);
    cout << x << endl;
}

int main() {
    int x = 42;
    foo();
    cout << x << endl;
    return 0;
}
```

Inhalt des Stacks:

Stack und Heap

Beispiel: Stack-Frames

```
void bar(int x) {
    x += 77;
    cout << x << endl;
}

void foo() {
    int x = 23;
    bar(x);
    cout << x << endl;
}

int main() {
    int x = 42;
    foo();
    cout << x << endl;
    return 0;
}
```

Inhalt des Stacks:

Stack und Heap

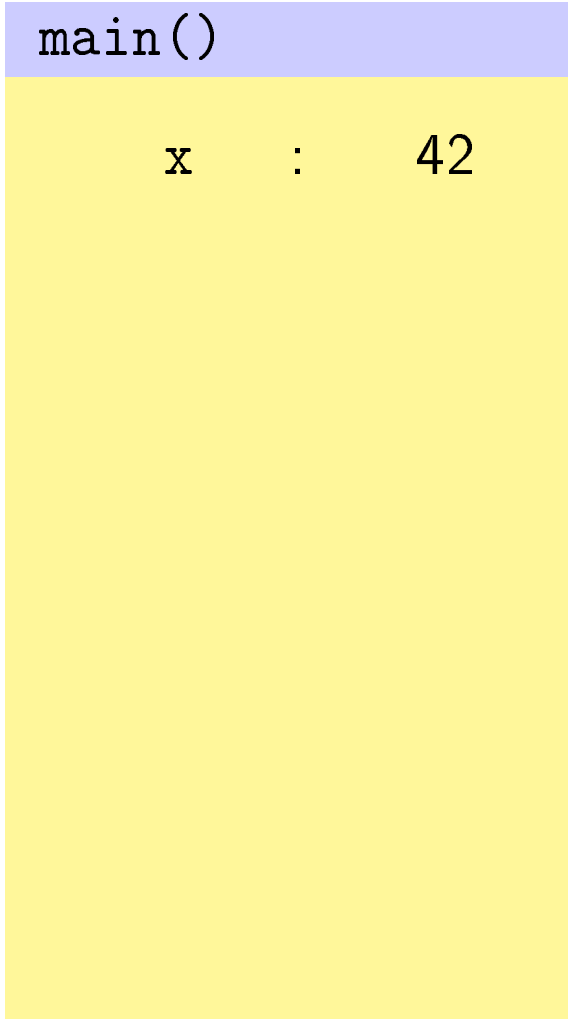
Beispiel: Stack-Frames

```
void bar(int x) {
    x += 77;
    cout << x << endl;
}

void foo() {
    int x = 23;
    bar(x);
    cout << x << endl;
}

int main() {
    int x = 42;
    foo();
    cout << x << endl;
    return 0;
}
```

Inhalt des Stacks:



Stack und Heap

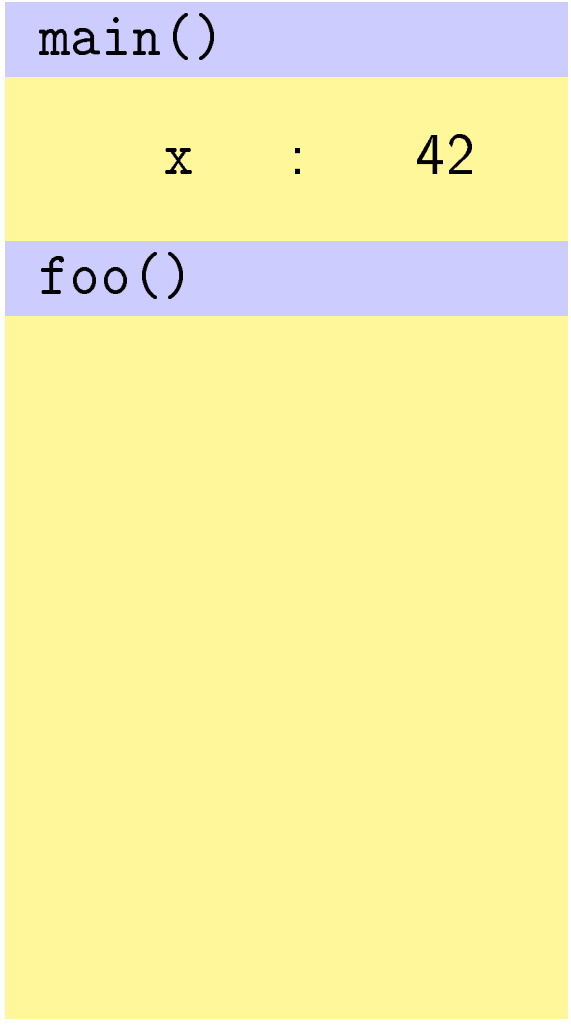
Beispiel: Stack-Frames

```
void bar(int x) {
    x += 77;
    cout << x << endl;
}

void foo() {
    int x = 23;
    bar(x);
    cout << x << endl;
}

int main() {
    int x = 42;
    foo();
    cout << x << endl;
    return 0;
}
```

Inhalt des Stacks:



Stack und Heap

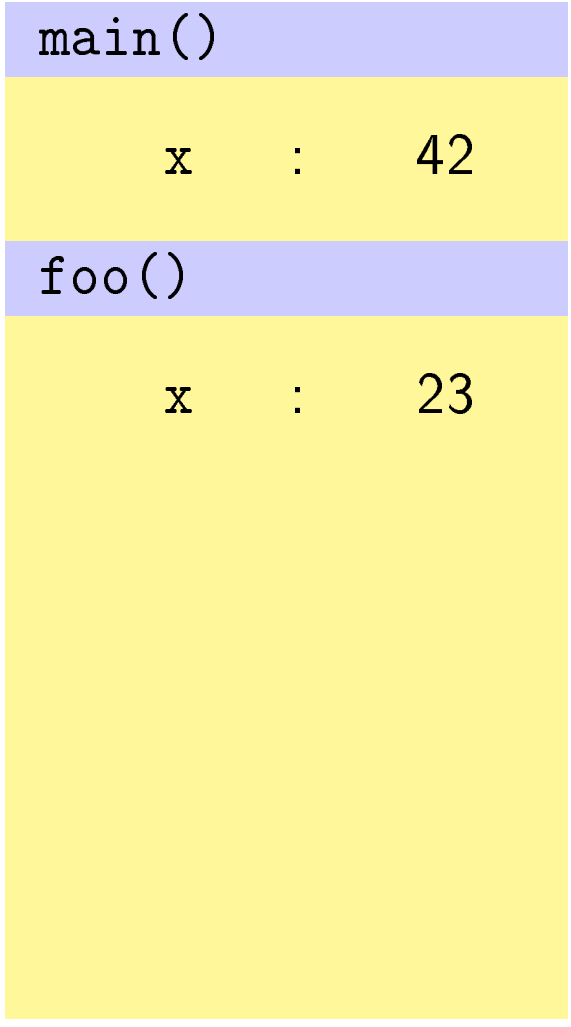
Beispiel: Stack-Frames

```
void bar(int x) {
    x += 77;
    cout << x << endl;
}

void foo() {
    int x = 23;
    bar(x);
    cout << x << endl;
}

int main() {
    int x = 42;
    foo();
    cout << x << endl;
    return 0;
}
```

Inhalt des Stacks:



Stack und Heap

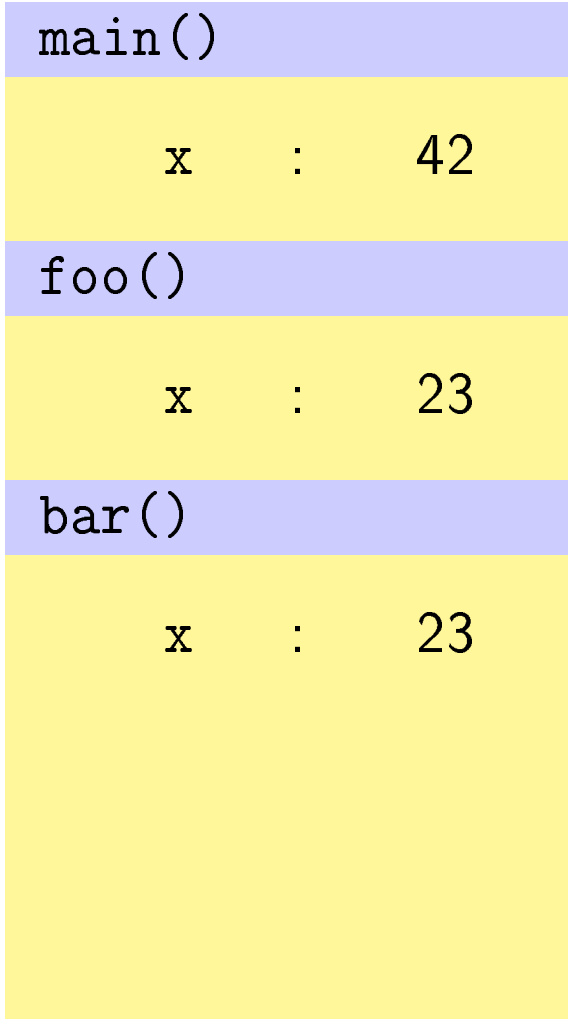
Beispiel: Stack-Frames

```
void bar(int x) {
    x += 77;
    cout << x << endl;
}

void foo() {
    int x = 23;
    bar(x);
    cout << x << endl;
}

int main() {
    int x = 42;
    foo();
    cout << x << endl;
    return 0;
}
```

Inhalt des Stacks:



Stack und Heap

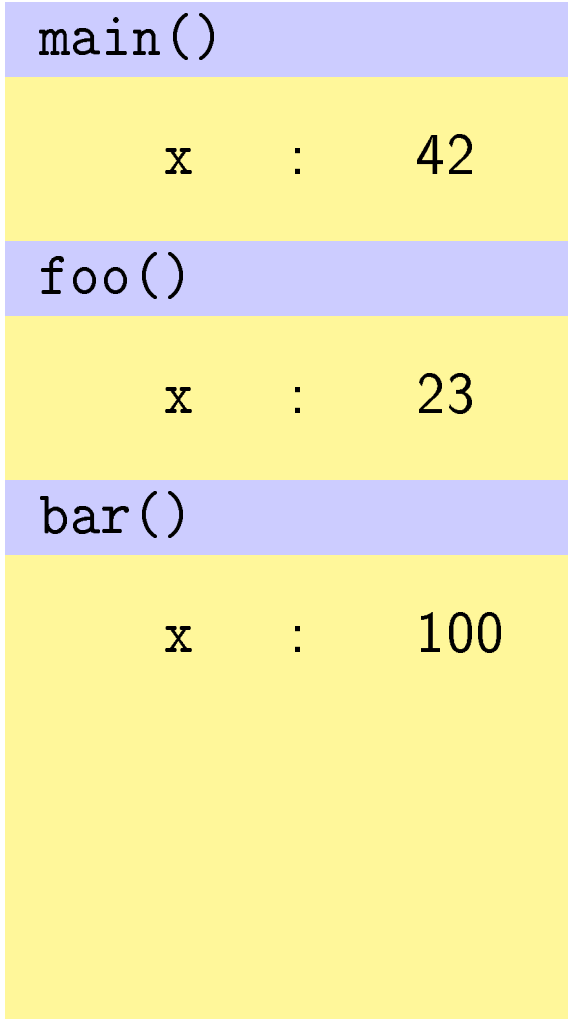
Beispiel: Stack-Frames

```
void bar(int x) {
    x += 77;
    cout << x << endl;
}

void foo() {
    int x = 23;
    bar(x);
    cout << x << endl;
}

int main() {
    int x = 42;
    foo();
    cout << x << endl;
    return 0;
}
```

Inhalt des Stacks:



Stack und Heap

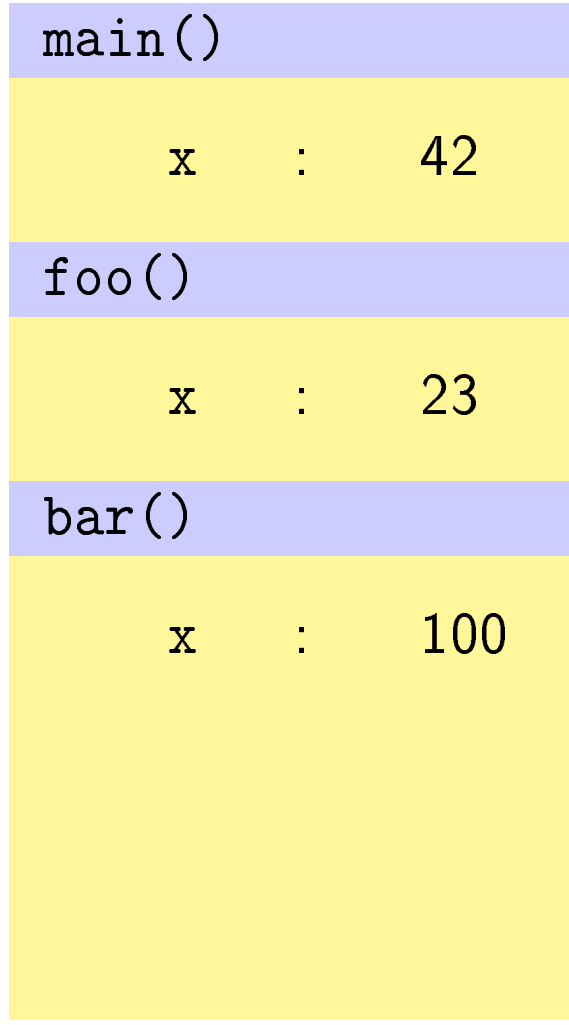
Beispiel: Stack-Frames

```
void bar(int x) {
    x += 77;
    cout << x << endl;
}

void foo() {
    int x = 23;
    bar(x);
    cout << x << endl;
}

int main() {
    int x = 42;
    foo();
    cout << x << endl;
    return 0;
}
```

Inhalt des Stacks:



Stack und Heap

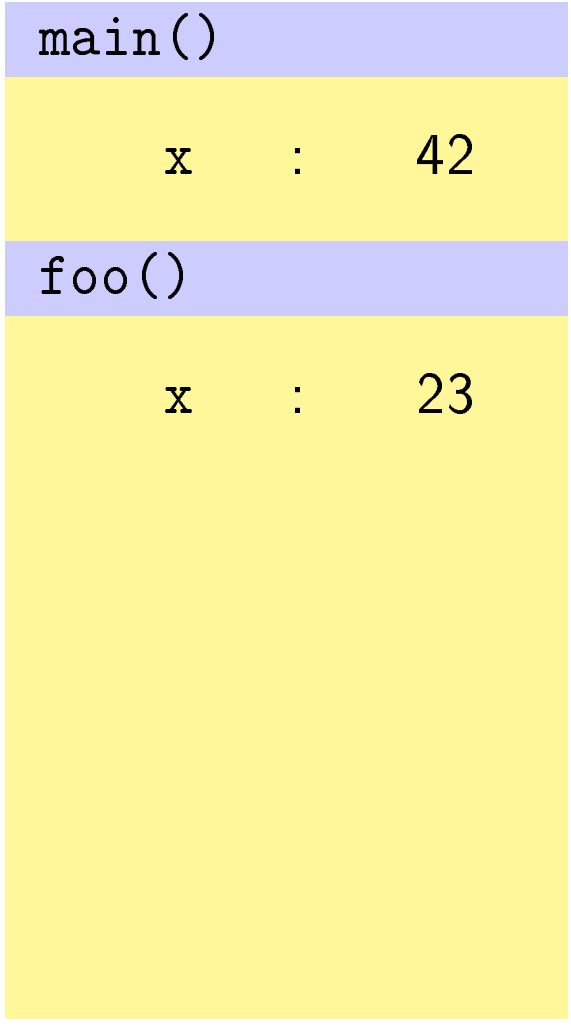
Beispiel: Stack-Frames

```
void bar(int x) {
    x += 77;
    cout << x << endl;
}

void foo() {
    int x = 23;
    bar(x);
    cout << x << endl;
}

int main() {
    int x = 42;
    foo();
    cout << x << endl;
    return 0;
}
```

Inhalt des Stacks:



Stack und Heap

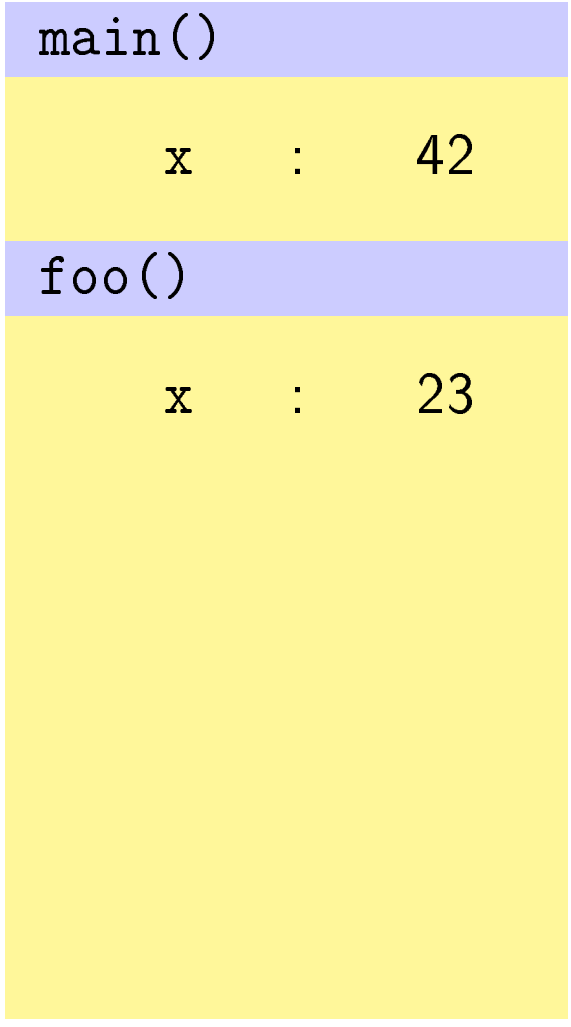
Beispiel: Stack-Frames

```
void bar(int x) {
    x += 77;
    cout << x << endl;
}

void foo() {
    int x = 23;
    bar(x);
    cout << x << endl;
}

int main() {
    int x = 42;
    foo();
    cout << x << endl;
    return 0;
}
```

Inhalt des Stacks:



Stack und Heap

Beispiel: Stack-Frames

```
void bar(int x) {
    x += 77;
    cout << x << endl;
}

void foo() {
    int x = 23;
    bar(x);
    cout << x << endl;
}

int main() {
    int x = 42;
    foo();
    cout << x << endl;
    return 0;
}
```

Inhalt des Stacks:

main()

x : 42

Stack und Heap

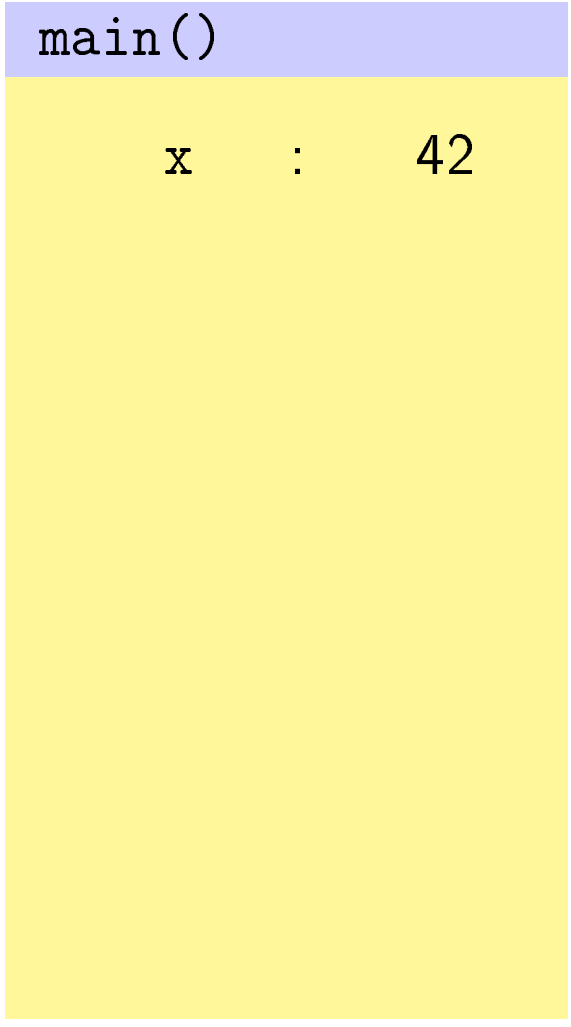
Beispiel: Stack-Frames

```
void bar(int x) {
    x += 77;
    cout << x << endl;
}

void foo() {
    int x = 23;
    bar(x);
    cout << x << endl;
}

int main() {
    int x = 42;
    foo();
    cout << x << endl;
    return 0;
}
```

Inhalt des Stacks:



Stack und Heap

Beispiel: Stack-Frames

```
void bar(int x) {
    x += 77;
    cout << x << endl;
}

void foo() {
    int x = 23;
    bar(x);
    cout << x << endl;
}

int main() {
    int x = 42;
    foo();
    cout << x << endl;
    return 0;
}
```

Inhalt des Stacks:

Pointer

Manchmal will man Variablen aus einem übergeordneten Scope weiter verwenden.

Pointer

Manchmal will man Variablen aus einem übergeordneten Scope weiter verwenden.

Lösung: Pointer

- Ein Pointer ist ein Name für eine andere Variable.
- Technisch: Eine Variable, in der eine Speicheradresse steht.

Pointer

Manchmal will man Variablen aus einem übergeordneten Scope weiter verwenden.

Lösung: Pointer

- Ein Pointer ist ein Name für eine andere Variable.
- Technisch: Eine Variable, in der eine Speicheradresse steht.

Verwendung in C++:

- Deklaration mit *:

```
int * x;
void foo(int * x) { ... }
```

- Benutzung mit Dereferenzierungsoperator * und Adressoperator &:

```
int x = 3;          int * y = &x;
printf("%d",*y);
```

Pointer

Einfaches Beispiel:

```
int main() {
    int x = 3;
    int * y = &x;
    cout << x << *y << endl;

    x = 4;
    cout << x << *y << endl;

    *y = 5;
    cout << x << *y << endl;

    return 0;
}
```

Pointer

Komplexeres Beispiel mit Funktionsaufrufen:

```
void bar(int * x) {
    *x += 77;
}

void foo(int * x) {
    *x += 23;
    bar(x);
}

int main() {
    int x = 42;
    foo(&x);
    cout << x << endl;
    return 0;
}
```

Pointer

Komplexeres Beispiel mit Funktionsaufrufen:

```
void bar(int * x) {
    *x += 77;
}

void foo(int * x) {
    *x += 23;
    bar(x);
}

int main() {
    int x = 42;
    foo(&x);
    cout << x << endl;
    return 0;
}
```

Inhalt des Stacks:

Pointer

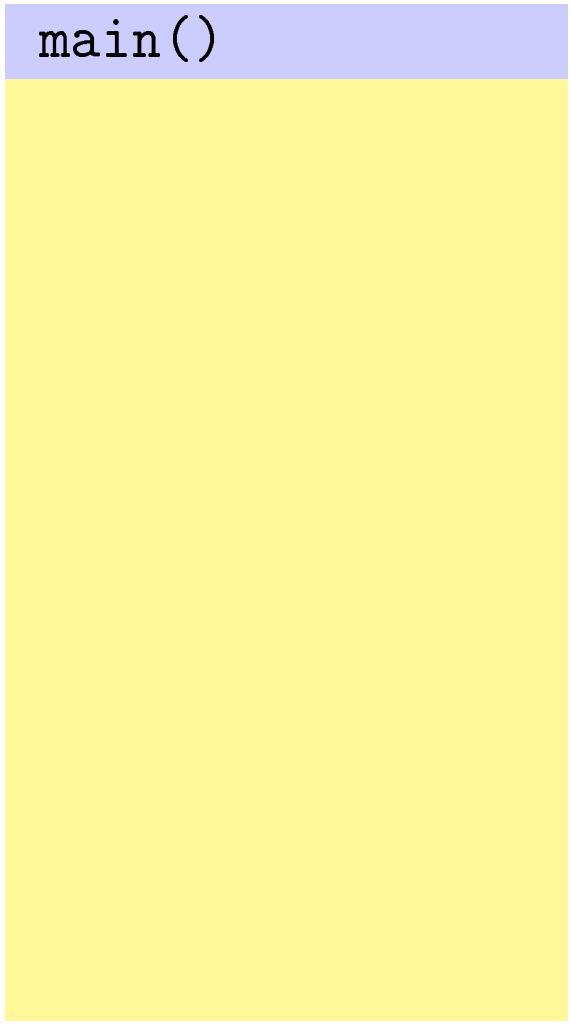
Komplexeres Beispiel mit Funktionsaufrufen:

```
void bar(int * x) {
    *x += 77;
}

void foo(int * x) {
    *x += 23;
    bar(x);
}

int main() {
    int x = 42;
    foo(&x);
    cout << x << endl;
    return 0;
}
```

Inhalt des Stacks:



Pointer

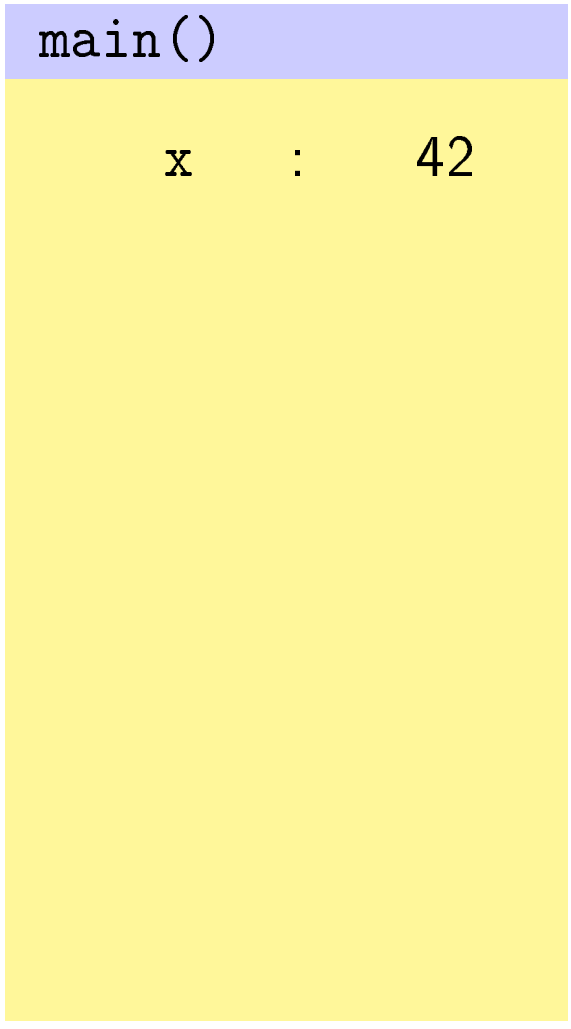
Komplexeres Beispiel mit Funktionsaufrufen:

```
void bar(int * x) {
    *x += 77;
}

void foo(int * x) {
    *x += 23;
    bar(x);
}

int main() {
    int x = 42;
    foo(&x);
    cout << x << endl;
    return 0;
}
```

Inhalt des Stacks:



Pointer

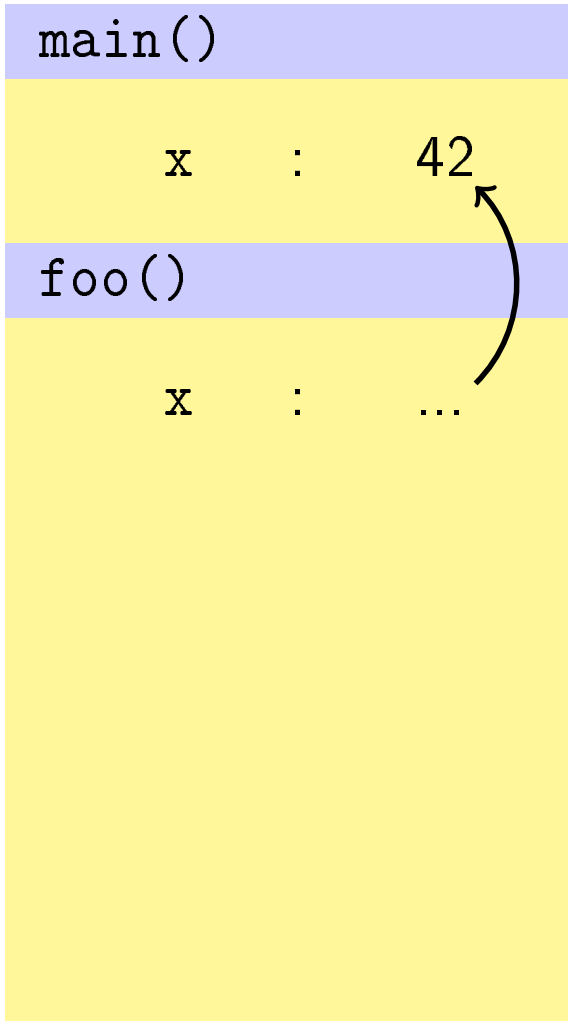
Komplexeres Beispiel mit Funktionsaufrufen:

```
void bar(int * x) {
    *x += 77;
}

void foo(int * x) {
    *x += 23;
    bar(x);
}

int main() {
    int x = 42;
    foo(&x);
    cout << x << endl;
    return 0;
}
```

Inhalt des Stacks:



Pointer

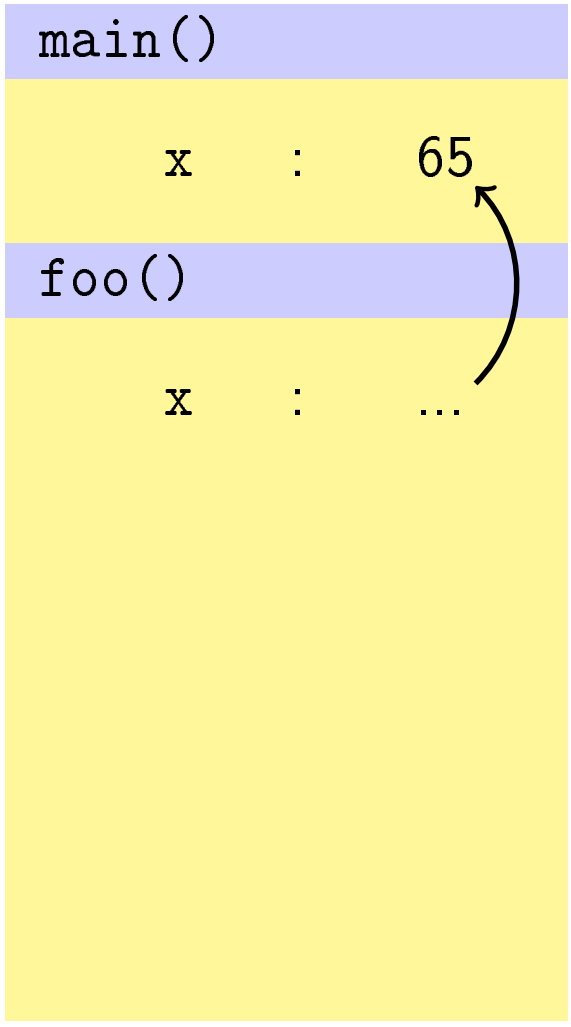
Komplexeres Beispiel mit Funktionsaufrufen:

```
void bar(int * x) {
    *x += 77;
}

void foo(int * x) {
    *x += 23;
    bar(x);
}

int main() {
    int x = 42;
    foo(&x);
    cout << x << endl;
    return 0;
}
```

Inhalt des Stacks:



Pointer

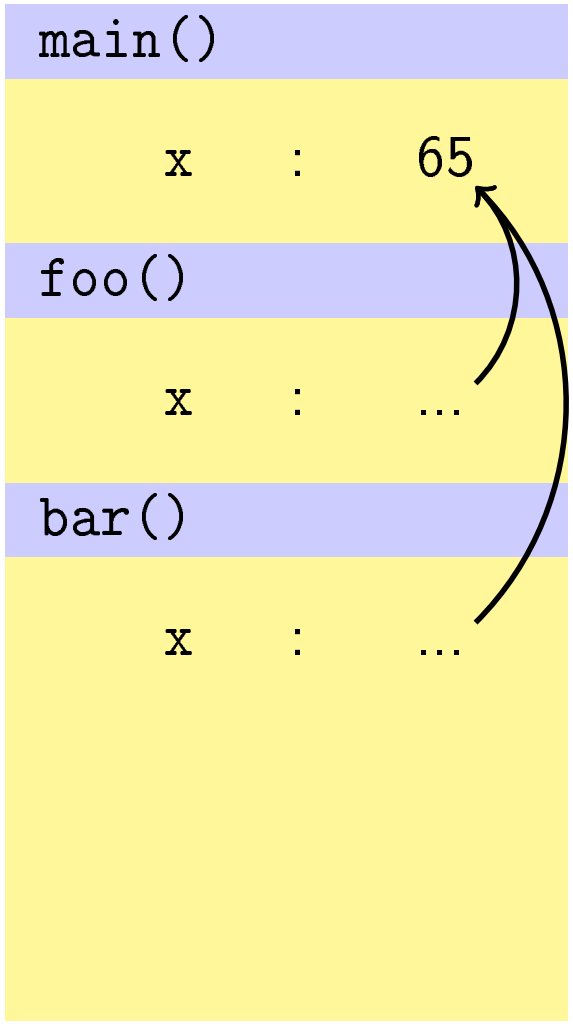
Komplexeres Beispiel mit Funktionsaufrufen:

```
void bar(int * x) {
    *x += 77;
}

void foo(int * x) {
    *x += 23;
    bar(x);
}

int main() {
    int x = 42;
    foo(&x);
    cout << x << endl;
    return 0;
}
```

Inhalt des Stacks:



Pointer

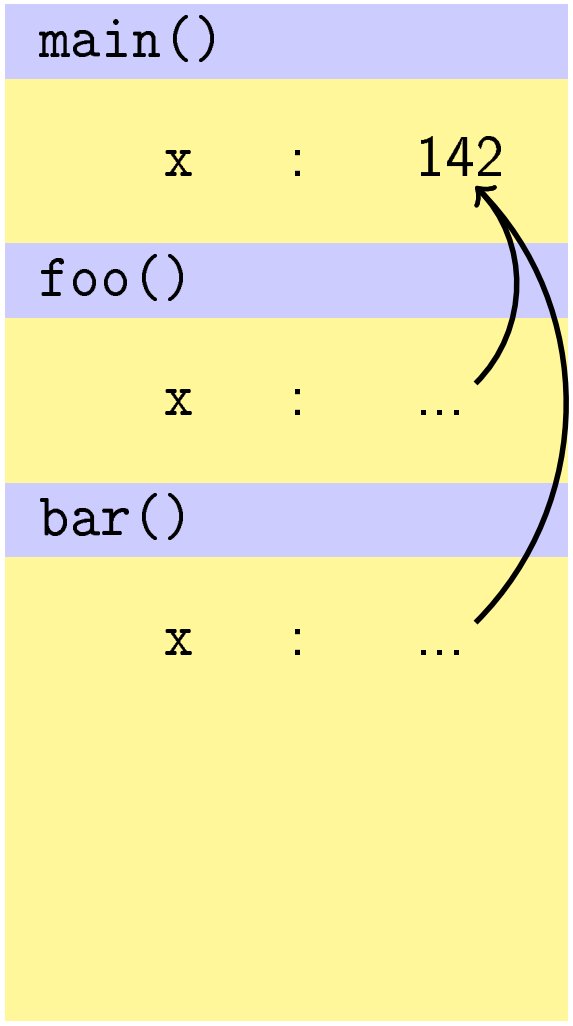
Komplexeres Beispiel mit Funktionsaufrufen:

```
void bar(int * x) {
    *x += 77;
}

void foo(int * x) {
    *x += 23;
    bar(x);
}

int main() {
    int x = 42;
    foo(&x);
    cout << x << endl;
    return 0;
}
```

Inhalt des Stacks:



Pointer

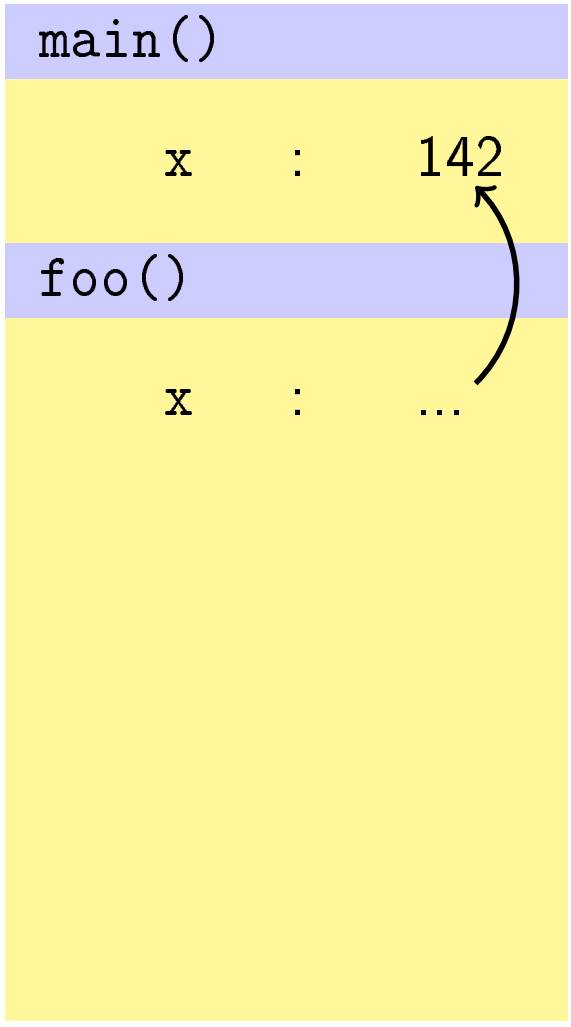
Komplexeres Beispiel mit Funktionsaufrufen:

```
void bar(int * x) {
    *x += 77;
}

void foo(int * x) {
    *x += 23;
    bar(x);
}

int main() {
    int x = 42;
    foo(&x);
    cout << x << endl;
    return 0;
}
```

Inhalt des Stacks:



Pointer

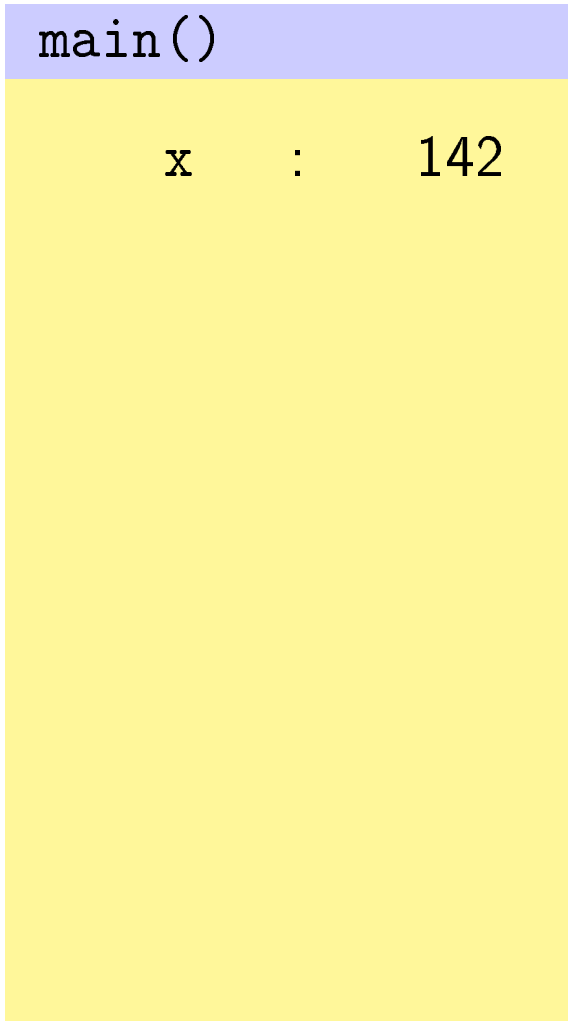
Komplexeres Beispiel mit Funktionsaufrufen:

```
void bar(int * x) {
    *x += 77;
}

void foo(int * x) {
    *x += 23;
    bar(x);
}

int main() {
    int x = 42;
    foo(&x);
    cout << x << endl;
    return 0;
}
```

Inhalt des Stacks:



Pointer

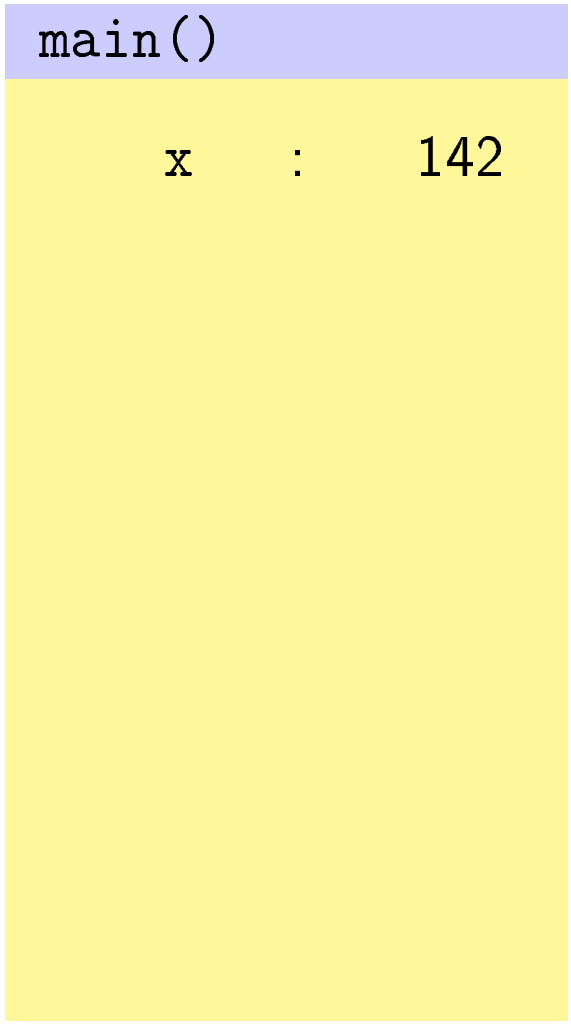
Komplexeres Beispiel mit Funktionsaufrufen:

```
void bar(int * x) {
    *x += 77;
}

void foo(int * x) {
    *x += 23;
    bar(x);
}

int main() {
    int x = 42;
    foo(&x);
    cout << x << endl;
    return 0;
}
```

Inhalt des Stacks:



Pointer

Komplexeres Beispiel mit Funktionsaufrufen:

```
void bar(int * x) {
    *x += 77;
}

void foo(int * x) {
    *x += 23;
    bar(x);
}

int main() {
    int x = 42;
    foo(&x);
    cout << x << endl;
    return 0;
}
```

Inhalt des Stacks:

Pointer

Pointer in der Praxis

- Verwendung als Namen, deren Zuordnung veränderlich ist.
- Übergabe als Parameter, damit Funktionen sie verändern.
- Verwendung als Zeiger in Arrays.

Pointer

Pointer in der Praxis

- Verwendung als Namen, deren Zuordnung veränderlich ist.
- Übergabe als Parameter, damit Funktionen sie verändern.
- Verwendung als Zeiger in Arrays.

Arrays sind Pointer ...

- `int a[3]` erzeugt einen Pointer.
- `a[0]` entspricht `*a`.

Pointer

Pointer in der Praxis

- Verwendung als Namen, deren Zuordnung veränderlich ist.
- Übergabe als Parameter, damit Funktionen sie verändern.
- Verwendung als Zeiger in Arrays.

Arrays sind Pointer ...

- `int a[3]` erzeugt einen Pointer.
- `a[0]` entspricht `*a`.

... und man kann mit ihnen rechnen (Pointerarithmetik):

- `a[1]` entspricht `*(a+1)`.
- Addition mit Pointern addiert eine Zahl an Speicherstellen.

Pointer

Beispiel für Arrays und Pointerarithmetik:

```
int main() {

    int a[2] = {42,23};

    cout << a[0] << endl;
    cout << *a << endl;

    cout << a[1] << endl;
    cout << *(a+1) << endl;
    cout << *a+1 << endl;

    return 0;
}
```

Referenzen

- Pointer sind sehr flexibel...

Referenzen

Pointer sind sehr flexibel...

- ... aber auch gefährlich ...

```
int a = 42;
int * b = a;
b++;
cout << *b // Krawumm!!
```

Referenzen

Pointer sind sehr flexibel...

- ... aber auch gefährlich ...

```
int a = 42;
int * b = a;
b++;
cout << *b // Krawumm!!
```

- ... und vielfältig falsch benutzbar:

```
int *a = nullptr;
cout << a // s.o.
```

Referenzen

Pointer sind sehr flexibel...

- ... aber auch gefährlich ...

```
int a = 42;
int * b = a;
b++;
cout << *b // Krawumm!!
```

- ... und vielfältig falsch benutzbar:

```
int *a = nullptr;
cout << a // s.o.
```

Alternative: Referenzen

- Namen für Objekte (wie Pointer)
- keine Arithmetik
- nicht nachträglich veränderbar
- keine Null-Pointer

Referenzen

Verwendung von Referenzen in C++:

```
int main() {
    int x = 3;
    int & y = x;
    cout << x << y << endl;

    x = 4;
    cout << x << y << endl;

    y = 5;
    cout << x << y << endl;

    return 0;
}
```

Wert- und Referenzparameter

Wir unterscheiden zwei Arten der Parameterübergabe bei Funktionsaufrufen: Wert- und Referenzparameter.

Wert- und Referenzparameter

Wir unterscheiden zwei Arten der Parameterübergabe bei Funktionsaufrufen: Wert- und Referenzparameter.

Wertparameter:

- Der Inhalt (Wert) des Parameters wird in die Funktion kopiert.
- Die ursprüngliche Variable kann von der Funktion nicht verändert werden.
- Standardfall in C++ für die meisten Datentypen.

Wert- und Referenzparameter

Wir unterscheiden zwei Arten der Parameterübergabe bei Funktionsaufrufen: Wert- und Referenzparameter.

Wertparameter:

- Der Inhalt (Wert) des Parameters wird in die Funktion kopiert.
- Die ursprüngliche Variable kann von der Funktion nicht verändert werden.
- Standardfall in C++ für die meisten Datentypen.

Referenzparameter:

- Die Variable wird nicht kopiert, sondern eine Referenz benutzt.
- Die Funktion manipuliert die ursprüngliche Variable.
- Implementierung in C++ mittels Pointern oder Referenzen.
- Standardfall bei C-Arrays

Wert- und Referenzparameter

Beispiel: Zwei Funktionen mit Wert- und Referenzparametern:

```
void foo(vector<int> v) // Wertparameter
{
    for (int el:v) cout << el;
}

void bar(vector<int> & v) // Referenzparameter
{
    for (int el:v) cout << el;
}
```

Wert- und Referenzparameter

Beispiel (Forts.): Aufruf der Funktionen:

```
int main()
{
    // Grossen Vector erzeugen
    vector<int> vec;
    for (int i = 0; i<100000; i++) vec.push_back(i);

    // Beide Aufrufe haben den gleichen Effekt:
    foo(vec);      // langsam (v wird kopiert)
    bar(vec);      // schnell (keine Kopie)

    return 0;
}
```