Technische Universität Braunschweig

Institute of Operating Systems and Computer Networks
Reliable System Software

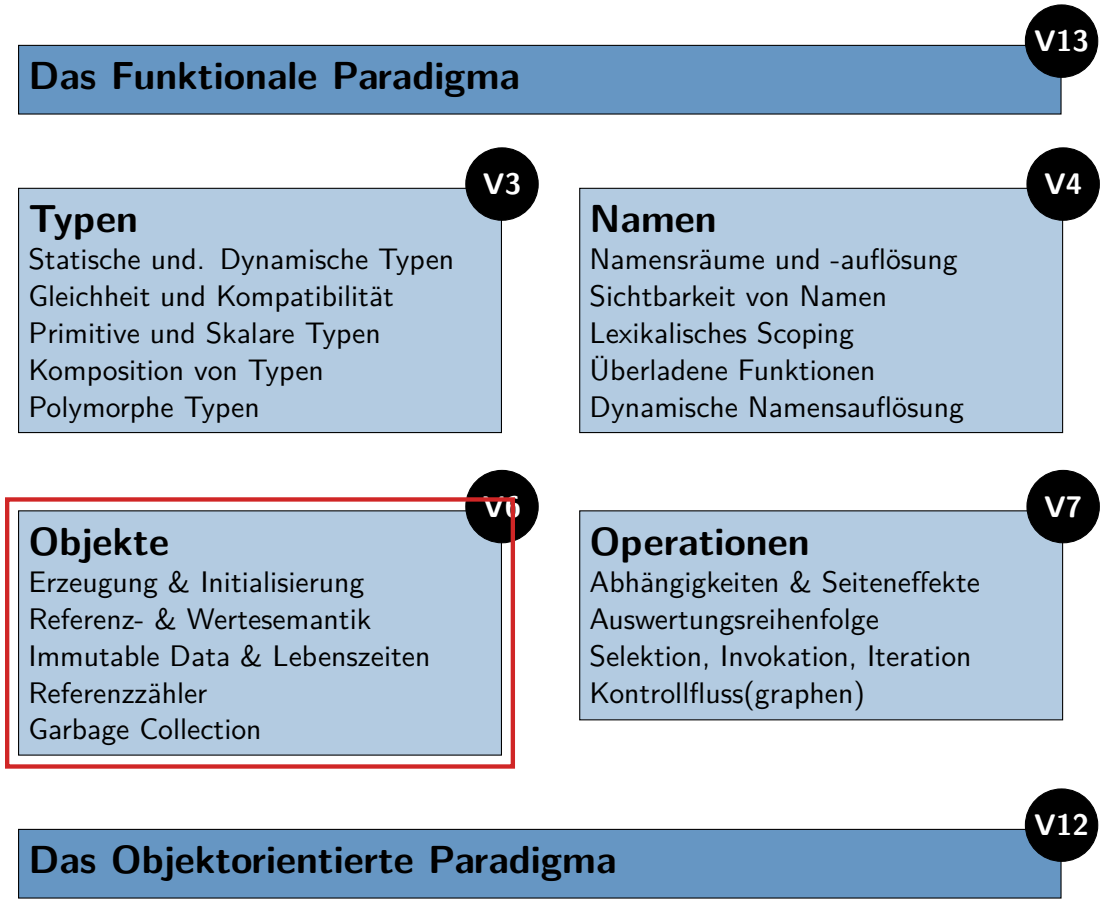
Programmiersprachen und Übersetzer

06 - Objekte

Christian Dietrich

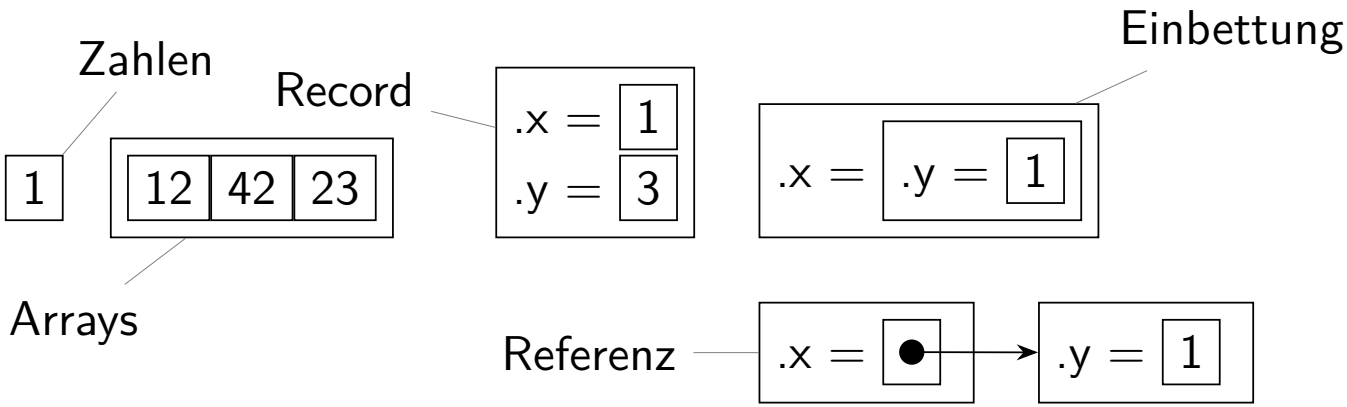
Sommersemester 2024

Einordnung in die Vorlesung: Objekte



- Objekte sind die datentragenden Elemente in einer Laufzeitumgebung
  - Die virtuelle Sprach-Maschine definiert Geburt, Leben und Tod von Objekten.
  - Effiziente Programme gehen sparsam und sorgsam mit Objekten um.

Was ist ein Objekt?



- **Erinnerung 1. Vorlesung**: Maschinenmodell der virtuellen Maschine:
  - Speicher/Objekte: Wie kann man Informationen ablegen und wieder abrufen?
  - Befehle/Operationen: Wie kann man Informationen miteinander kombinieren?

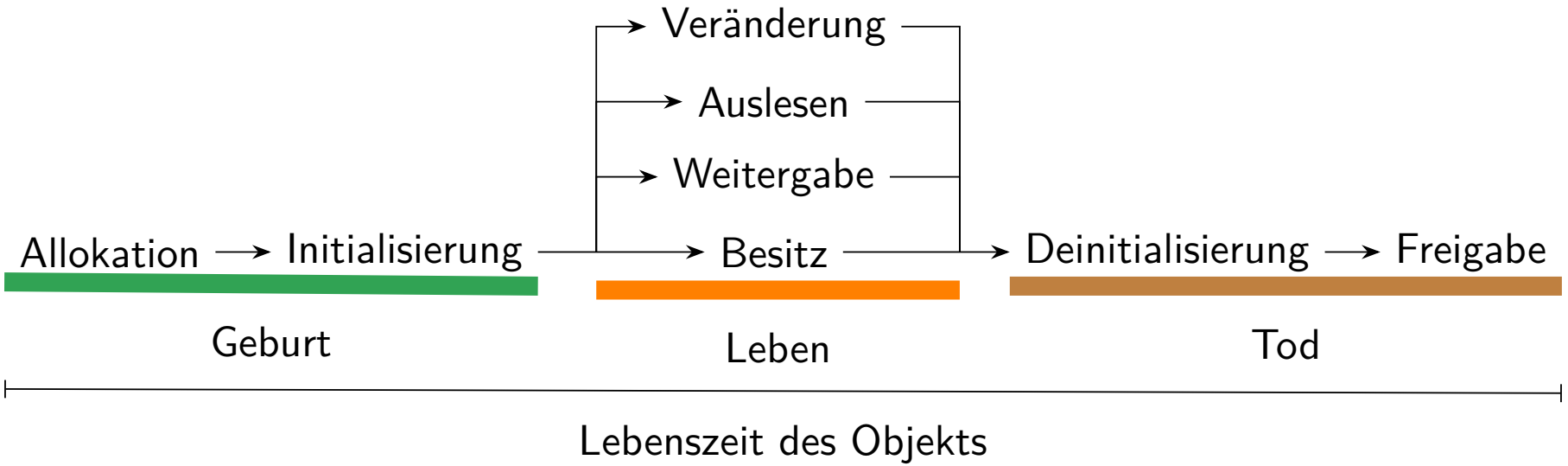
Definition: Objekt

Ein Objekt ist ein existenziell abhängiger Verbund von Informationen.

Wichtig: (1) Variablen $\neq$ Objekte! Variablen können Objekte beinhalten.
(2) Diese Definition umfasst auch primitive Objekte, wie Zahlen!

Lebenszyklus eines Objekts

Vereinfachung: Objekte residieren in zusammenhängendem Speicher



- Geburt: Initialisierung von frisch allokiertem Speicher
  Objektzustand muss so präpariert werden, dass die Typinvarianten gelten.
- Leben: Objekte transportieren Informationen im Programmablauf
  Sprache kann es uns erleichtern, die Typinvarianten zu erhalten, sonst: Bugs.
- Tod: Kontrolliertes Aufräumen des Zustands und Freigabe der Ressourcen
  Der Zeitpunkt und der Verantwortliche für das Aufräumen ist sprachabhängig.

Beispiel: Leben eines Loggers

```c
typedef enum {
  DEBUG, INFO, WARN, ERROR,
} level_t;

typedef struct {
  level_t level;
  int fd;
} log_t;

int main() {
  // Geburt
  log_t *L = log_init();

  // Leben
  log(L, INFO, "message");
  L->level = DEBUG;
  log(L, INFO, "message");

  // Tod
  log_deinit(L);
}
```

C

```c
log_t * log_init() {
  // Allokation: Speicher am Heap
  log_t *l= malloc(sizeof(log_t));
  // Initialisierung des Zustands
  l->level = WARN;
  l->fd = open("/dev/stderr", 0);
  return l;
}
```

```c
void
log(log_t *l, level_t ll, char *m) {
  if (ll >= l->level)
    write(l->fd, m, strlen(m));
}
```

```c
void log_deinit(log_t *l) {
  close(l->fd); // Datei schließen
  free(l); // Speicher freigeben
}
```

- Kaum Sprachunterstützung: `struct log_t` kann man leicht falsch halten!
  - Benutzer kann Invarianten verletzen: `L->fd = 23;`
  - Use-after-free Bugs: `log_deinit(L); log(L, ERROR, "panic");`
  - Vergessen das Objekt frei zu geben → Memory Leak

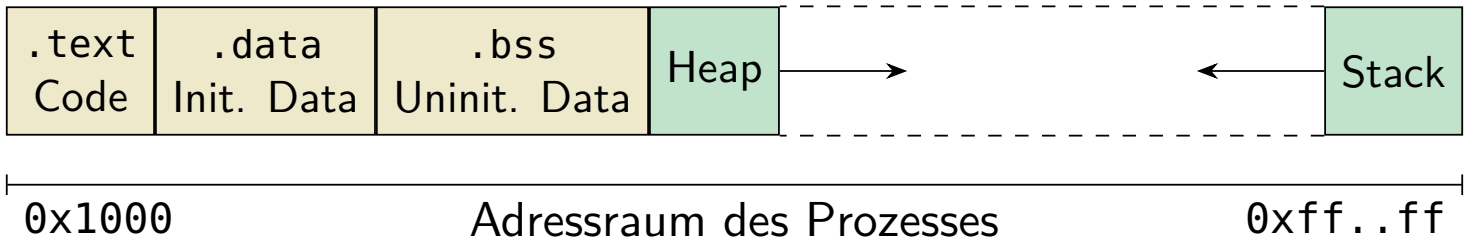
Allokation: Wo kommt der Speicherplatz her?

Meistens residieren Objekte in einem zusammenhängenden Speicherbereich

`allocate(bytes_t N)` $\longrightarrow$ Startadresse des zukünftigen Objekts

- Statische Allokation durch Übersetzer: Lebenszeit = Programmlaufzeit
  - Absolute Adresse des Objekts steht vor der Laufzeit fest
  - Bsp.: globale Variablen, konstante Literale, Maschinencode einer Funktion
  - 2 statische Objekte: `char *global = "foobar";`
- Stackallokation: Objekte können auf den Aufrufstapel
  - Allokation als Einbettung des Call-Frames einer Funktionsinstanz
  - Lebenszeit des Objekts ist $\leq$ Lebenszeit des Call-Frames
  - Beispiele: Argumente, lokale Variablen, `alloca()`
- Heapallokation: Laufzeitsystem betreibt separaten Speicherpool
  - Lebenszeit unabhängig von Aufrufhierarchie, beliebige Allokationszeitpunkte
  - Benötigt komplexeres Speichermanagement und ist daher teurer
  - Speicher muss irgendwann freigegeben werden (manuell oder Garbage Collection)

Allokationen im größeren Kontext



- Im klassisches UNIX-Modell wachsen Stack und Heap aufeinander zu.
  - Das ELF beschreibt die statisch allokierten Objekte (.text, .data, .bss)
  - Realität ist komplexer: Mehrere Stacks/Heaps, Address Space Randomization

  → Genauere Betrachtung in Betriebssysteme (BS)

- Für Programmiersprachen und Übersetzer nehmen wir an, dass...
  - wir globale Objekte im Assembler anlegen können.
  - der Prozess einen Stack hat und wir den Stackpointer kontrollieren.
  - das Laufzeitsystem `void *malloc(size_t)` / `free(void*)` bereitstellt.

Initialisierung und Konstruktoren

Technische Universität Braunschweig

Wie wird aus dem allokierten, noch blanken, Speicherbereich ein Objekt?

- Sprachabhängige Initialisierung etabliert Meta-Informationen

  Was macht ein Stück Speicher für meine Sprache zu einem Objekt?

  - Einige Sprachfeatures erfordern zusätzliche Informationen an jedem Objekt.
  - Zum Beispiel – Dynamische Typinformationen in Form eines Typ-Tags
    – Registrierung des Objekts am Garbage Collector (später mehr)

- Benutzerdefinierte Initialisierung durch **Konstruktoren**

  Was möchte der Benutzer bei der Geburt eines Objekts tun?

  - Initiale Belegungen der Objekt-Attribute
  - Parametrisierte Konstruktion
  - Intention des Benutzers: Etablierung der **semantischen Invarianten**

Sprachregeln für Konstruktoren

```C++
class Object {
  public:
    Object(int x)    { ... }
    Object(string x) { ... }
};
```

C++

```Java
class Object {
  public Object(int x)    { ... }
  public Object(String x) { ... }
};
```

Java

- Definition von eigenen Konstruktoren
  - Definieren wir keine eigenen, werden Default-Konstruktoren erzeugt
  - Konstruktoren haben Parameter und können überladen werden
  - Sehen aus wie Funktionen, haben aber keinen Rückgabewert
- Aufruf von Konstruktoren
  - Konstruktoren werden bei Objekterstellung automatisch aufgerufen.
  - Bei Vererbung: Eltern-Konstruktoren vor Kind-Konstruktoren
    $\Rightarrow$ Kind-Konstruktoren finden ein valides Eltern-Objekt vor.

Konstruktoren in Aktion: Java

```
class Counter {
  final int start;
  int next;       // ④

  public Counter(int n) {
    start = n;    // ③
  }

  public Counter() {
      start = 0;
  }

  public int inc() {
      return start+(next++);
  }
}
class Derived
     extends Counter {
  Derived(int x) {
      super(3+x); // ②
  }
  Derived(String s) { // ①
      this(parseInt(s));
  }
}
```

- Java
  - Konstruktoren heißen wie die Klasse.
  - Nicht-initialisierte Attribute werden automatisch 0 oder `null`.
- Ablauf für `new Derived("123")`
  1. Aufruf anderer Konstruktoren mittels `this()`
  2. Elternkonstruktor mittels `super()`
  3. Konstruktor setzt `final`-Attribut
  4. Autom. Nullung durch `Object`-Konstruktor

Konstruktoren in Aktion: Java

```
class Counter {
  final int start;
  int next;       // ④

  public Counter(int n) {
    start = n;    // ③
  }

  public Counter() {
      start = 0;
  }

  public int inc() {
      return start+(next++);
  }
}
class Derived
     extends Counter {
  Derived(int x) {
      super(3+x); // ②
  }
  Derived(String s) { // ①
      this(parseInt(s));
  }
}
```

- Java
  - Konstruktoren heißen wie die Klasse.
  - Nicht-initialisierte Attribute werden automatisch 0 oder `null`.
- Ablauf für `new Derived("123")`
  1. Aufruf anderer Konstruktoren mittels `this()`
  2. Elternkonstruktor mittels `super()`
  3. Konstruktor setzt `final`-Attribut
  4. Autom. Nullung durch `Object`-Konstruktor
- Wie sähe eine manuelle Konstruktion aus?

```
// Allokation
Derived *this = malloc(...);

// Sprachspezifisches Init
memset(this, 0, sizeof(Derived));
this->vtable = Dervied_vtable;

// Konstruktor (inlined)
int tmp = parseInt("123")
this->start = 3+tmp;
```

Pseudo-C

Was sollte **mein** Konstruktor leisten?

Technische Universität Braunschweig

Sinnvolle Konvention: Konstruktoren stellen semantische Invarianten her

- Entwickler trifft **immerwährende Annahmen** über den Zustand eines Objekts.
- **Jeder** Konstruktor hinterlässt ein Objekt, das die Annahmen einhält.
- Jede weitere Operation auf dem Objekt muss diese Annahmen **erhalten**.

- Beispiel: Bounded Pointer zeigt nur auf die Elemente eines Arrays

```
class bounded_ptr {
  uint8_t *data;
  unsigned length;
  uint8_t *ptr;

  bounded_ptr(unsigned len) {
    data   = malloc(len);
    length = len;
    ptr    = &data[0];
  }

  void set(uint8_t val) {
    *ptr = val;
  }
  ...
};
```

- Invarianten für bounded_ptr
  - `data` zeigt auf ein valides Integer-Array
  - `length` gibt Länge dieses Arrays an
  - `ptr` ist die Adresse eines Array-Elements
- **Bonussternchen**, falls der Benutzer nicht in der Lage ist die Invarianten zu verletzen.

Wie werden Objekte abgelegt und weitergereicht?

Erinnerung (binding time): Ein Objekt wird an einen Namen gebunden.

```
var_foo = new object();
```

```
foo(var_foo);
```

Mittels gebundener Namen können wir Objekte ansprechen und weitergeben.

Wie werden Objekte abgelegt und weitergereicht?

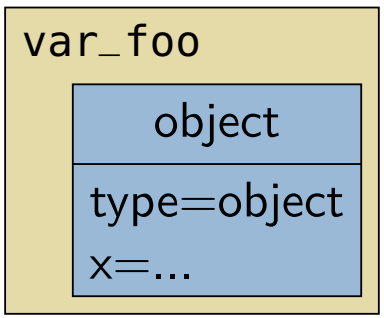
Erinnerung (binding time): Ein Objekt wird an einen Namen gebunden.

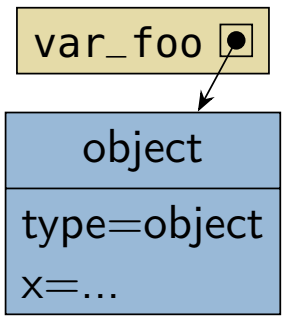
```
var_foo = new object();
```

```
foo(var_foo);
```

Mittels gebundener Namen können wir Objekte ansprechen und weitergeben.

Aber: Ist der Wert einer Variable das Objekt oder eine Objektreferenz?





Wertemodell für Variablen

- Objekte „leben“ in Variablen
- Zuweisung erzeugt eine **Kopie**
- Referenzen sind separate Objekte

⇒ C, C++, Rust

Referenzmodell für Variablen

- Variable speichert nur Referenz
- Zuweisung erzeugt weitere Referenz
- Keine separaten Referenzobjekte

⇒ Java (mostly), Python, Ruby

Wertemodell: Vor- und Nachteile

```
class foo_t { int x; };
...
foo_t A = { .x = 23 };
foo_t B = A;
A.x = 42; // B.x == 23
```

- A und B enthalten verschiedene Objekte
- Initialisierung von B durch Kopie
- Objektinhalte/Speicher werden kopiert

\+ Das Wertemodell ist flexibler und kann effizienter sein.

- Indirektion nur auf Nachfrage; kleine Objekte lassen sich effizient kopieren.
- Referenzen sind explizit im Code sichtbar (als Zeiger-Typ).
- Objekte in lokalen Variablen können auf dem Stack allokiert werden.

– Das Wertemodell ist komplexer und schwieriger zu beherrschen.

- Referenzen machen das Programmiermodell komplexer.
- Häufiges Kopieren *kann* unsichtbaren Overhead erzeugen.
- Initialisierung durch Kopieren (oder Verschieben) ist eigentlich ein Sonderfall
  $\Rightarrow$ C++ kennt 3 Sorten von Konstruktoren: Init-, Copy-, Move-Konstruktoren

Referenzmodell: Vor- und Nachteile

```java
class foo { public int x; }
...
foo A = new foo();
foo B = A;
A.x   = 23;
// B.x == 23
```

Java

- Beide referenzieren das selbe Objekt
- Automatisch Dereferenzierung
- Objekterzeugung nur mittels `new T()`

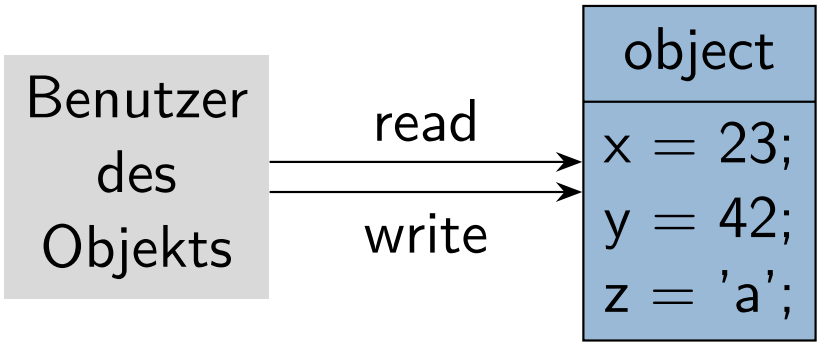
\+ Im Referenzmodell sind Variablen und Objekte orthogonal.

- Objekte existieren **immer** unabhängig von Variablen und Funktionsaufrufen.
- Weitergabe quer zur Aufrufhierarchie ist trivial.
- Keine Probleme mit Kind-Klassen, die mehr Speicher brauchen.
  Problematisch in C++: `Base func() { Derived d; return d; }`

– Im Referenzmodell tragen wir immer die Kosten der Indirektion.

- Ohne Optimierungen müssen alle Objekte am Heap allokiert werden.
- Todeszeitpunkt von Objekten ist unklar.
- Primitive Typen: Jeder Integer hinter einem Pointer?
  Java: Wertemodell für Zahlen, außer wenn nicht (Auto(un-)boxing)

Zugriff auf Objekte

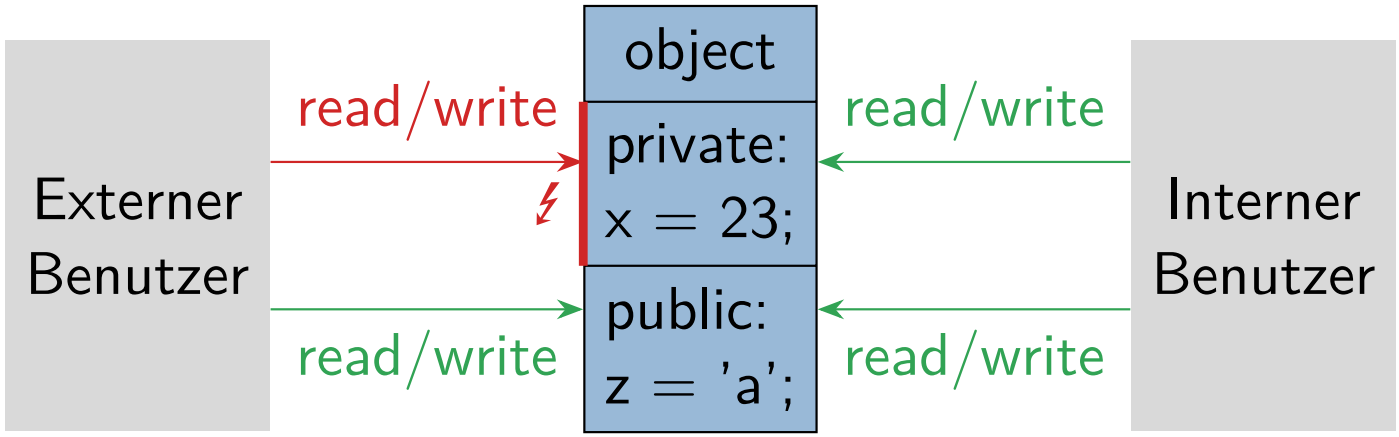


- Wer ist der Benutzer eines Objekts?
  - Eigentlich: Das laufende Programm (aktiv) greift auf ein Objekt (passiv) zu.
  - Nützliche Präzisierungen: Zugreifende Funktion, zugreifender Thread
- Welche Berechtigungen hat der Benutzer?
  - Referenzen sind Befähigungen (Capabilities) ein Objekt zu nutzen.
  - Nicht jede Referenz auf ein Objekt muss gleich mächtig sein.
- Welche Arten von Zugriffen gibt es?
  - Daten aus dem einem Objekt auslesen oder verändern.
  - Direkter Speicherzugriff oder gefiltert über Getter/Setter.
  - Ableitung weiterer Referenzen (z.B. Subobjekte: `&obj.x`).

Gefährlich!

Wer greift **wie** mit **welchem Recht** zu?

Benutzerabhängige Zugriffseinschränkungen

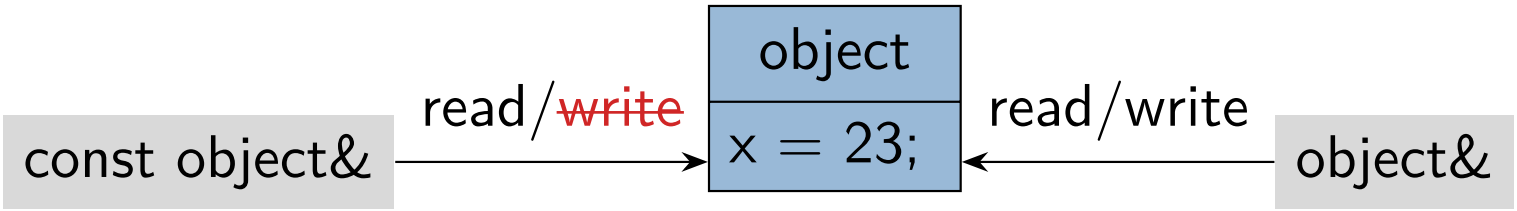


- **Ziel:** Benutzergruppen mit unterschiedlichen Zugriffsrechten
  - Identifikation der unterschiedlichen Gruppen (Wer greift zu?)
  - Einteilung anhand der zugreifenden Instruktionsadresse

  $\Rightarrow$ Wir können den Code mit unbeschränktem Zugriff kontrollieren.

- **Implementierung:** Einschränkung der Sichtbarkeit des Namens
  - Wenn der externe Nutzer einen Namen nicht sieht, kann er nicht zugreifen.
  - **Aber:** Gibt ein interner Nutzer eine Referenz auf ein privates Attribut heraus, bricht die Abstraktion. **Referenzen geben Kontrolle heraus!**

Referenzabhängige Zugriffseinschränkungen



- **Ziel:** Einschränkbare Referenzen erlauben kontrolliertes Sharing.
  - Über eingeschränkte Referenzen dürfen nicht alle Zugriffsarten erfolgen.
  - Referenzen dürfen nur weiter eingeschränkt und nicht geweitet werden.
  - Beispiel: Ableitung einer read-only-Referenz aus einer read-write-Referenz
- **Implementierung (const):** Erweiterung des Typsystems
  - Weiterer Zeiger-Typkonstruktor: const_pointer(T)
  - Implizite Typumwandlung zur read-only-Referenz
  - Asymmetrische Typ-Kompatibilität: const_pointer(T) $\ll$ pointer(T)
- Standardverhalten in verschiedenen Sprachen
  - C++: Read-only ist die Ausnahme (`T*, const T`).
  - Rust: Read-only ist die Regel (`&T, &mut T`)
  - Java: Alle Referenzen dürfen alles.

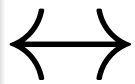
Gefilterter Zugriff über Zugriffsmethoden

Technische Universität Braunschweig

Direkter Speicherzugriff

```
obj.length = -1;
```

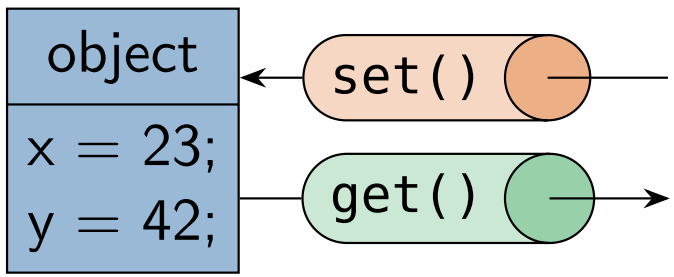
User kann Invarianten verletzen



Zugriffsmethoden

```
obj.setLength(-1);
```

Boilerplate, reine Konvention



- Einige Sprachen erlauben den gefilterten Zugriff auf einzelne Attribute
  - Verwendung sieht aus wie direkter Zugriff
  - `set()` beim Schreiben eines Attributs
  - `get()` zum Lesen eines Attributs
  - Kann **nachträglich** eingefügt werden, ohne die Benutzer zu ändern!

```C#
public class Student {
  private string name;

  public string Name {
    get { return name; }
    set { name = value; }
  }
}
```

Fallstudie: Unveränderliche Objekte

Unveränderliche Objekte (immutable objects)

Gibt es für ein Objekt, im gesamten Programm, keine read-write-Referenzen, so ist sein Inhalt unveränderlich.

- Immutability für ein Objekt bringt spannende Eigenschaften.
  - Thread-Safety: Keine Probleme mit konkurrierenden Schreibzugriffen
  - Werte- und Referenzmodell werden äquivalent.
  - Deduplikation von Objekten ist möglich (=interning).
  - **Aber:** Verändernde Operation müssen das Objekt verändert kopieren.
- Manche Objekte sind natürlicherweise Immutable.
  - Zahlen: Die Zahl 5 kann nicht so verändert werden, dass alle Fünfen im gesamten Programm plötzlich Achten sind.
  - Stringliterale: Der Übersetzer legt literale Zeichenketten (`"Hello"`) nur einmal in die Binärdatei.

Ownership für Objekte

Wem gehört ein Objekt?

Viele Bugs und Probleme rühren daher, dass man sich keine Gedanken darüber gemacht hat, wer die Verantwortung für ein Objekt hat.

- **Denkanweisung**: Wer ist der Besitzer und wo wird Besitz übertragen?
  - Besitzer kann ein Thread, eine Funktion oder ein anderes Objekt sein.
  - Der (letzte) Besitzer ist verantwortlich für die Freigabe des Objekts.
  - Geteilter Besitz von Objekten erfordert immer **erhöhte Aufmerksamkeit**.

```
class Proxy {
  obj_t* ref;
 public:
  void set(obj_t* o) { ref = o; }
  void call()    { ref->call(); }
};
Proxy p; p.set(obj);
delete obj;
p.call();
```

- `p` glaubt Besitzer von `obj` zu sein.
- dangling-reference problem

Besitz ist Schwierig!

- Besitz ist oft nur intentional!
- Referenz impliziert keinen Besitz!
- Modernste Sprachenentwicklungen:
  - Hilfestellungen: `std::unique_ptr<T>`
  - Übersetzer prüft Besitzer statisch: Rusts Borrowchecker

Wann endet die Lebenszeit eines Objekts?

Technische Universität Braunschweig

Ende der Lebenszeit

Wenn ein Objekt nicht mehr benötigt wird, endet seine Lebenszeit und wir können es freigeben.

- Indikatoren für das Ende der Lebenszeit
  - Keine zukünftigen Zugriffe auf das Objekt
  - Letzter Besitzanspruch erlischt ⇒ Notwendig
  - Letzte Referenz auf das Objekt wird ungültig ⇒ Hinreichend
- Explizite Freigabe erfordert Disziplin und provoziert Speicherlecks

```
log_deinit(log);
```

```
delete obj;
```
(C++)

Referenzzählung

- Zählen der existierenden Referenzen
- Exakter Todeszeitpunkt
- Probleme mit Referenzzyklen

Garbage Collector

- Finden der unreferenzierten Objekte
- Entkopplung von Tod und Freigabe
- Kosten treten in Bursts auf

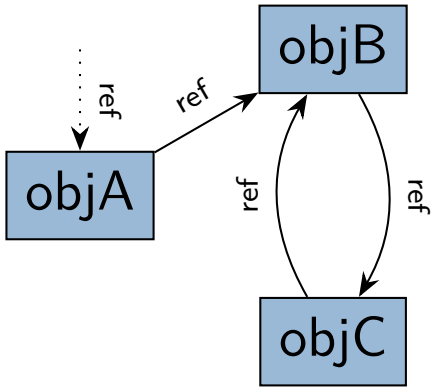
Referenzzählung

```C++
class A {
  unsigned users;
  A()            { users=1; }
  void claim()   { users++; }
  void release() {
    if (--users == 0) {
      delete this;
    }
  }
};
```

- Manuell: Besitzer zeigt Referenzweiter- bzw. -aufgabe an
  - Reihenfolge von `claim()` und `release()` ist kritisch
  - Fällt der Zähler auf 0, wird das Objekt automatisch freigegeben.
  - Referenzieren sich Objekte zyklisch, wird **niemals** freigegeben.

Referenzzählung

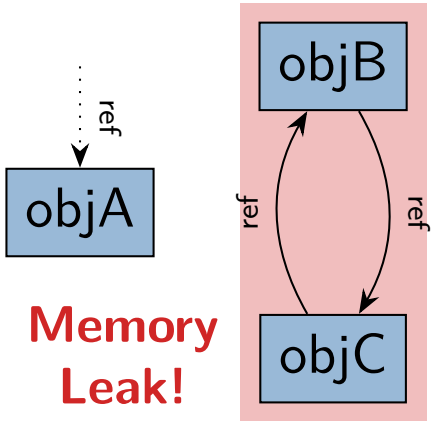
```C++
class A {
  unsigned users;
  A()            { users=1; }
  void claim()   { users++; }
  void release() {
    if (--users == 0) {
      delete this;
    }
  }
};
```



- Manuell: Besitzer zeigt Referenzweiter- bzw. -aufgabe an
  - Reihenfolge von `claim()` und `release()` ist kritisch
  - Fällt der Zähler auf 0, wird das Objekt automatisch freigegeben.
  - Referenzieren sich Objekte zyklisch, wird **niemals** freigegeben.

Referenzzählung

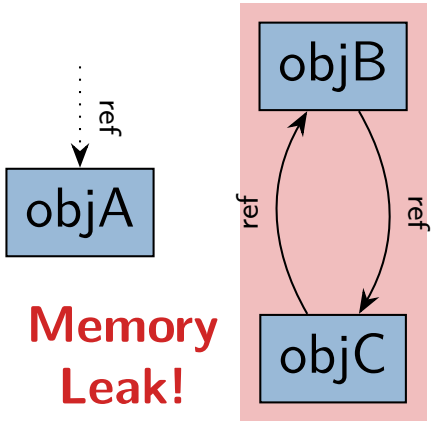
```C++
class A {
  unsigned users;
  A()            { users=1; }
  void claim()   { users++; }
  void release() {
    if (--users == 0) {
      delete this;
    }
  }
};
```



- Manuell: Besitzer zeigt Referenzweiter- bzw. -aufgabe an
  - Reihenfolge von `claim()` und `release()` ist kritisch
  - Fällt der Zähler auf 0, wird das Objekt automatisch freigegeben.
  - Referenzieren sich Objekte zyklisch, wird **niemals** freigegeben.

Referenzzählung

```C++
class A {
  unsigned users;
  A()            { users=1; }
  void claim()   { users++; }
  void release() {
    if (--users == 0) {
      delete this;
    }
  }
};
```



- Manuell: Besitzer zeigt Referenzweiter- bzw. -aufgabe an
  - Reihenfolge von `claim()` und `release()` ist kritisch
  - Fällt der Zähler auf 0, wird das Objekt automatisch freigegeben.
  - Referenzieren sich Objekte zyklisch, wird **niemals** freigegeben.
- Smart Pointer: Automatische Referenzzählung

```C++
{ // Typausdruck: shared_ptr(int) vs ptr(int);
  std::shared_ptr<int> A(new int(23));
  (*A)++;
  {
    std::shared_ptr<int> B = A;
    std::cout << (*B); // -> 24;
  } // <- keine Freigabe
} // <- Freigabe des ints
```

Garbage Collection

Kernidee: Nicht-referenzierte Objekte können freigegeben werden

Schritt 1 Finde alle Objekte, die transitiv von einem root set erreichbar sind.
Schritt 2 Gib alle nicht-erreichbaren Objekte frei.

- Erreichbarkeitsanalyse im Referenzgraphen
  - Jedes existierende Objekt ist ein Knoten im Referenzgraphen.
  - Jede Referenz/Zeiger ist eine gerichtete Kante.
  - Root Set: globale/lokale Variablen, Registerinhalte

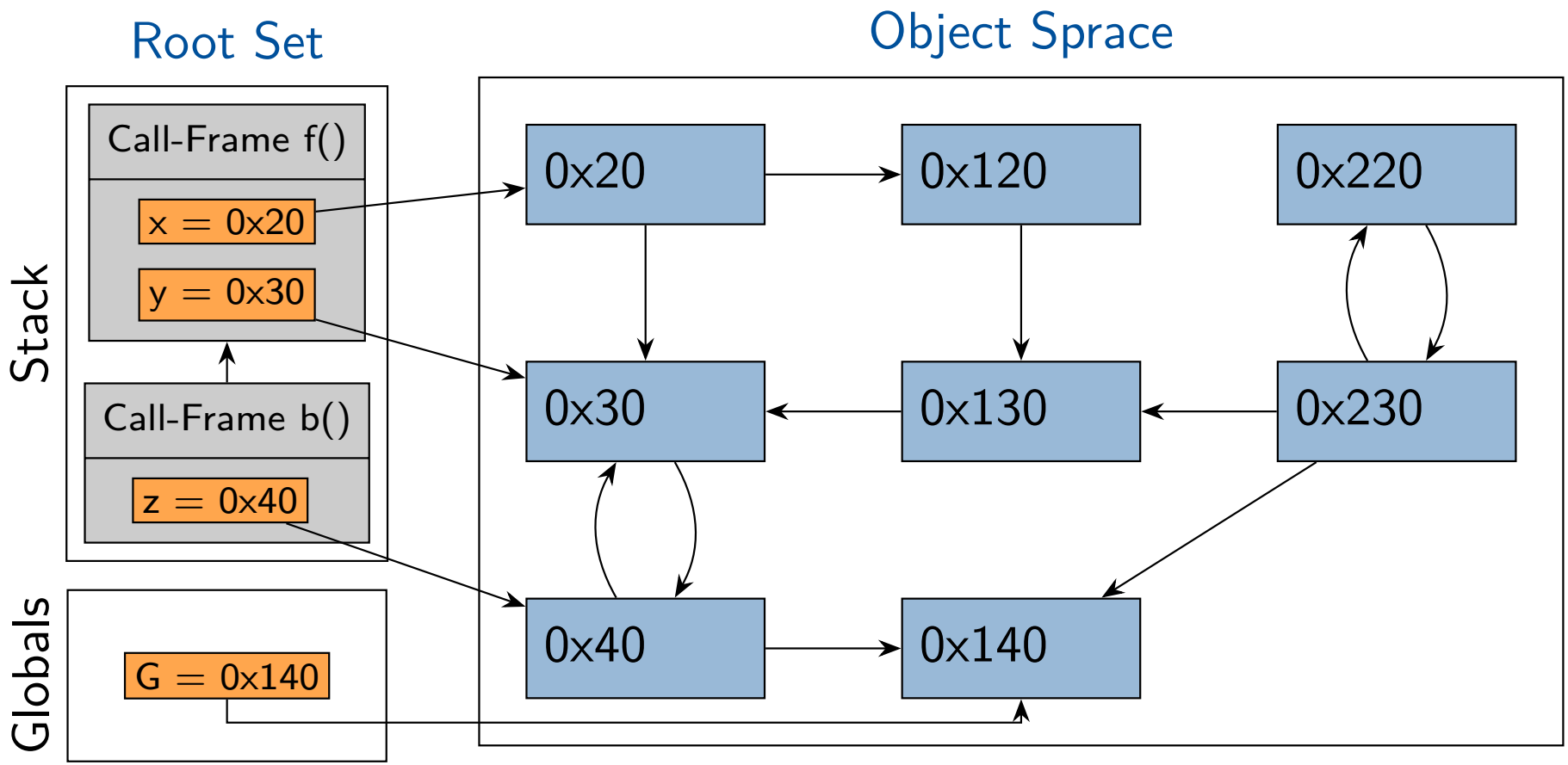
Vorbedingungen und Probleme	Ansatz
Kenntnis aller existierender Objekte	Objektliste
Konsistente Sicht auf den Referenzgraphen	stop-the-world
Erkennung aller ausgehenden Referenzen	Typsicherheit
Iteration über alle Objekte ist teuer	Partitionierte Objektmenge

$\Rightarrow$ GC hauptsächlich in gemanagten Sprachen, wie Java oder Python

Mark-and-Sweep Garbage Collector

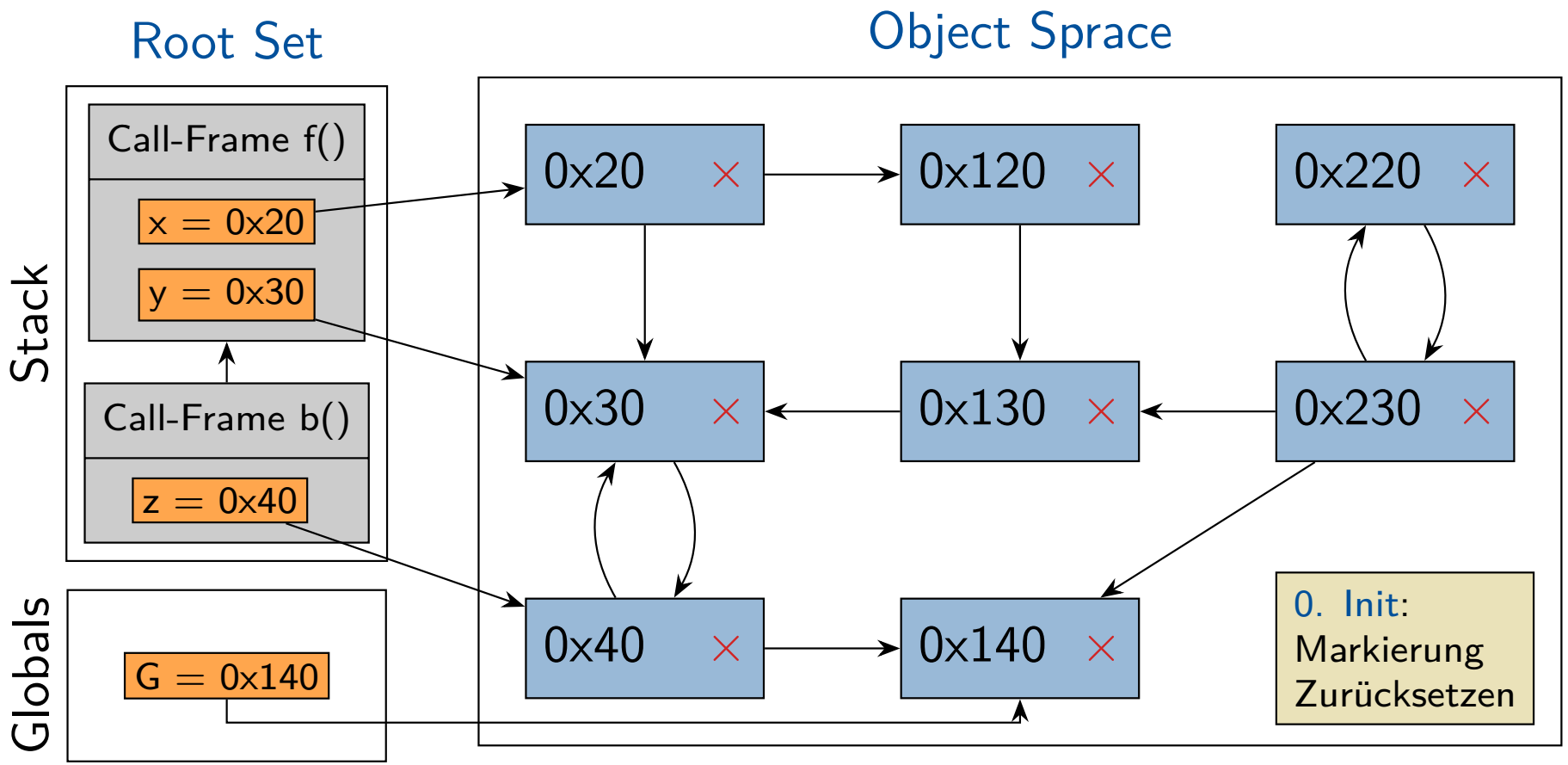
Technische Universität Braunschweig

Der Mark-and-Sweep Garbage Collector hält die Welt an, markiert alle erreichbaren Objekte, und gibt die anderen frei.



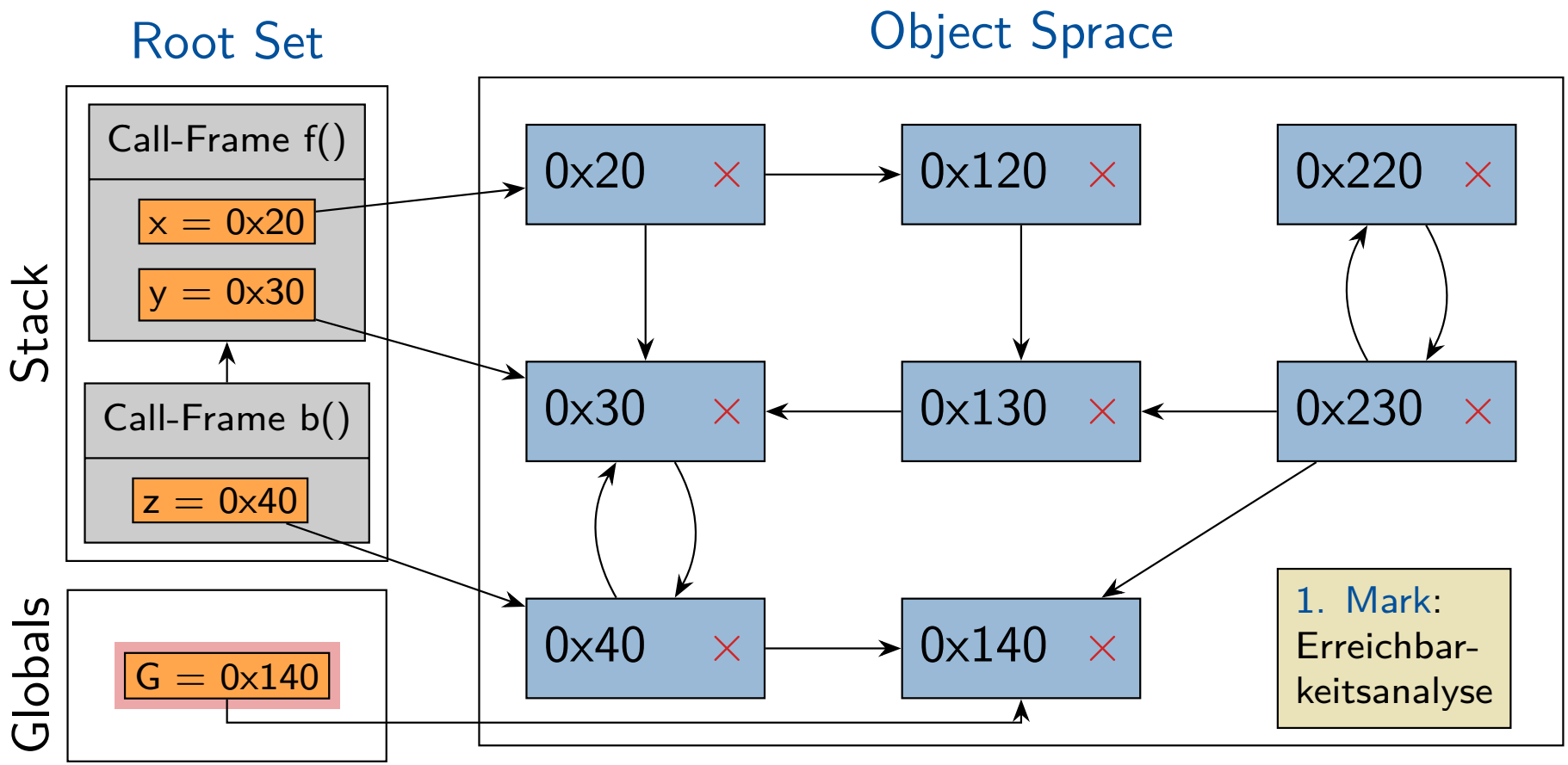
Mark-and-Sweep Garbage Collector

Der Mark-and-Sweep Garbage Collector hält die Welt an, markiert alle erreichbaren Objekte, und gibt die anderen frei.



Mark-and-Sweep Garbage Collector

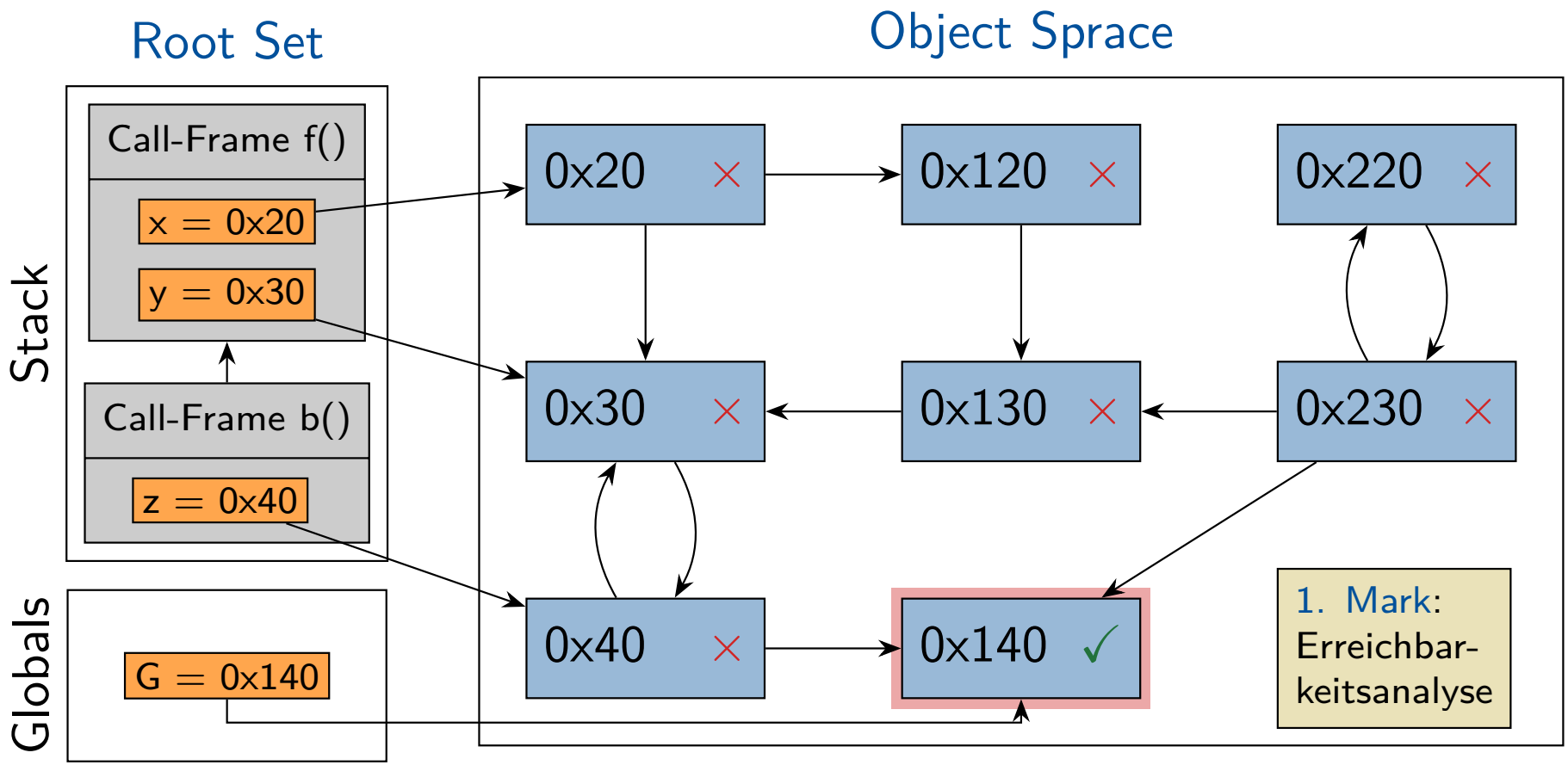
Der Mark-and-Sweep Garbage Collector hält die Welt an, markiert alle erreichbaren Objekte, und gibt die anderen frei.



Mark-and-Sweep Garbage Collector

Technische Universität Braunschweig

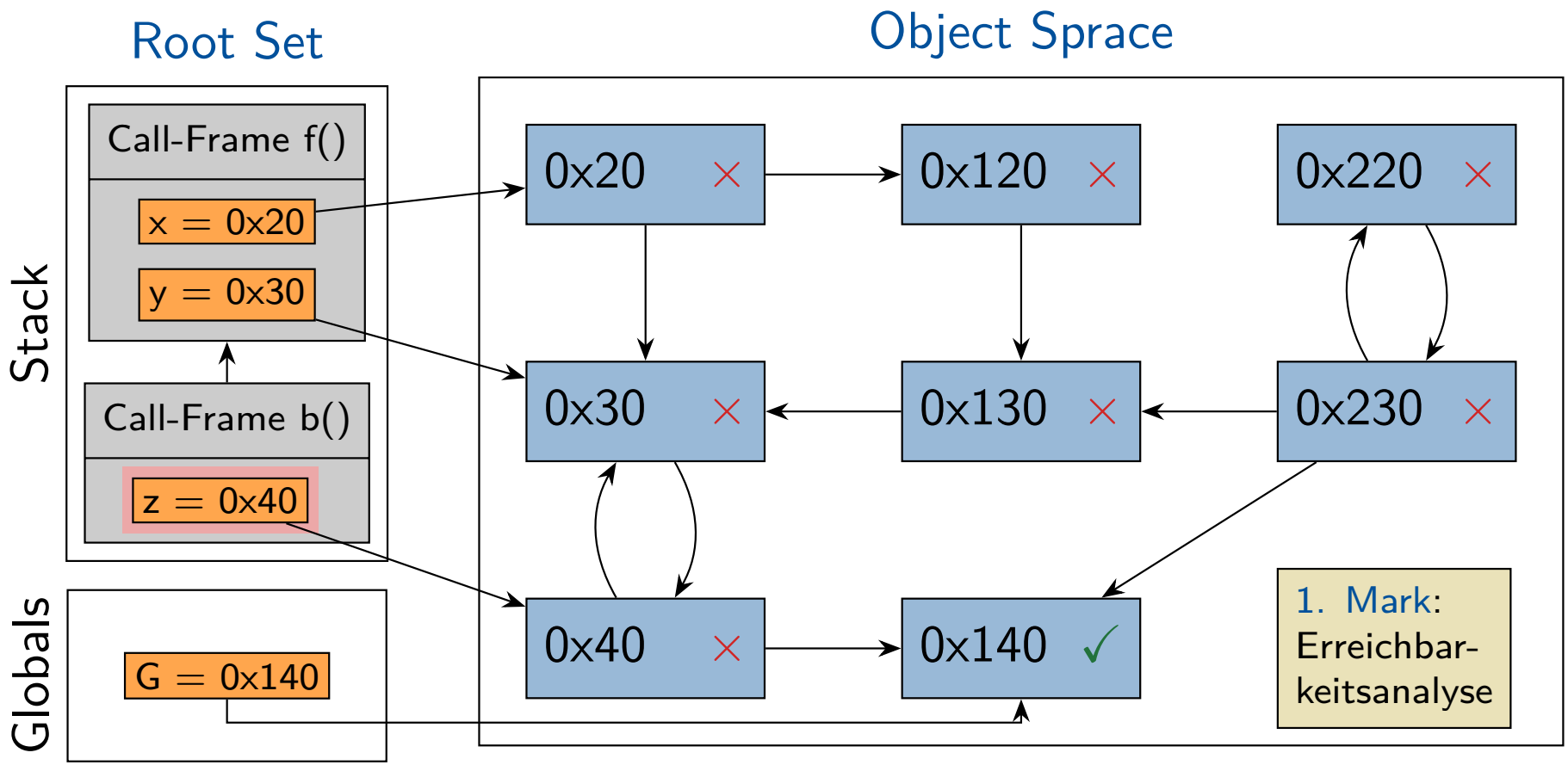
Der Mark-and-Sweep Garbage Collector hält die Welt an, markiert alle erreichbaren Objekte, und gibt die anderen frei.



Mark-and-Sweep Garbage Collector

Technische Universität Braunschweig

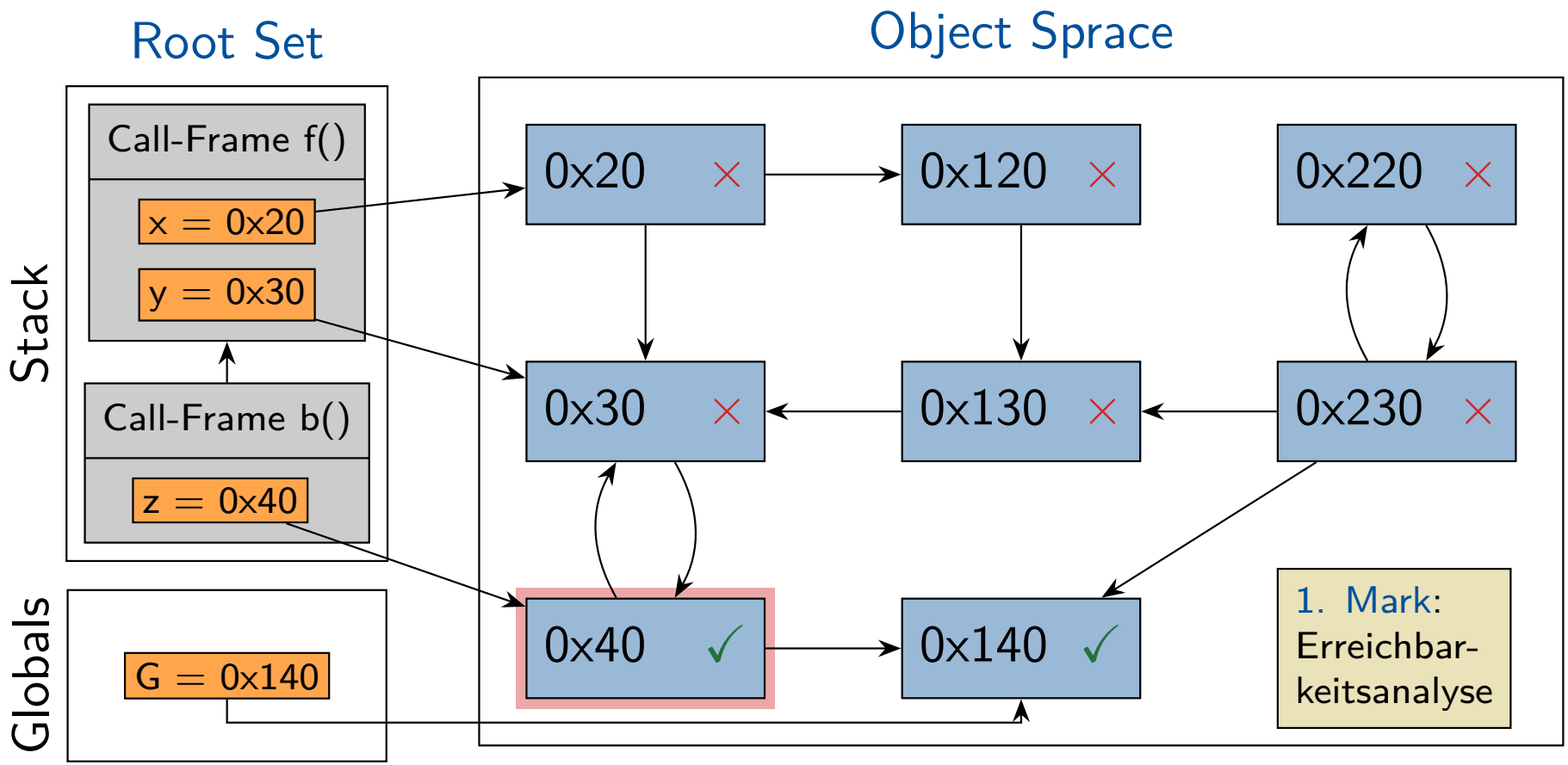
Der Mark-and-Sweep Garbage Collector hält die Welt an, markiert alle erreichbaren Objekte, und gibt die anderen frei.



Mark-and-Sweep Garbage Collector

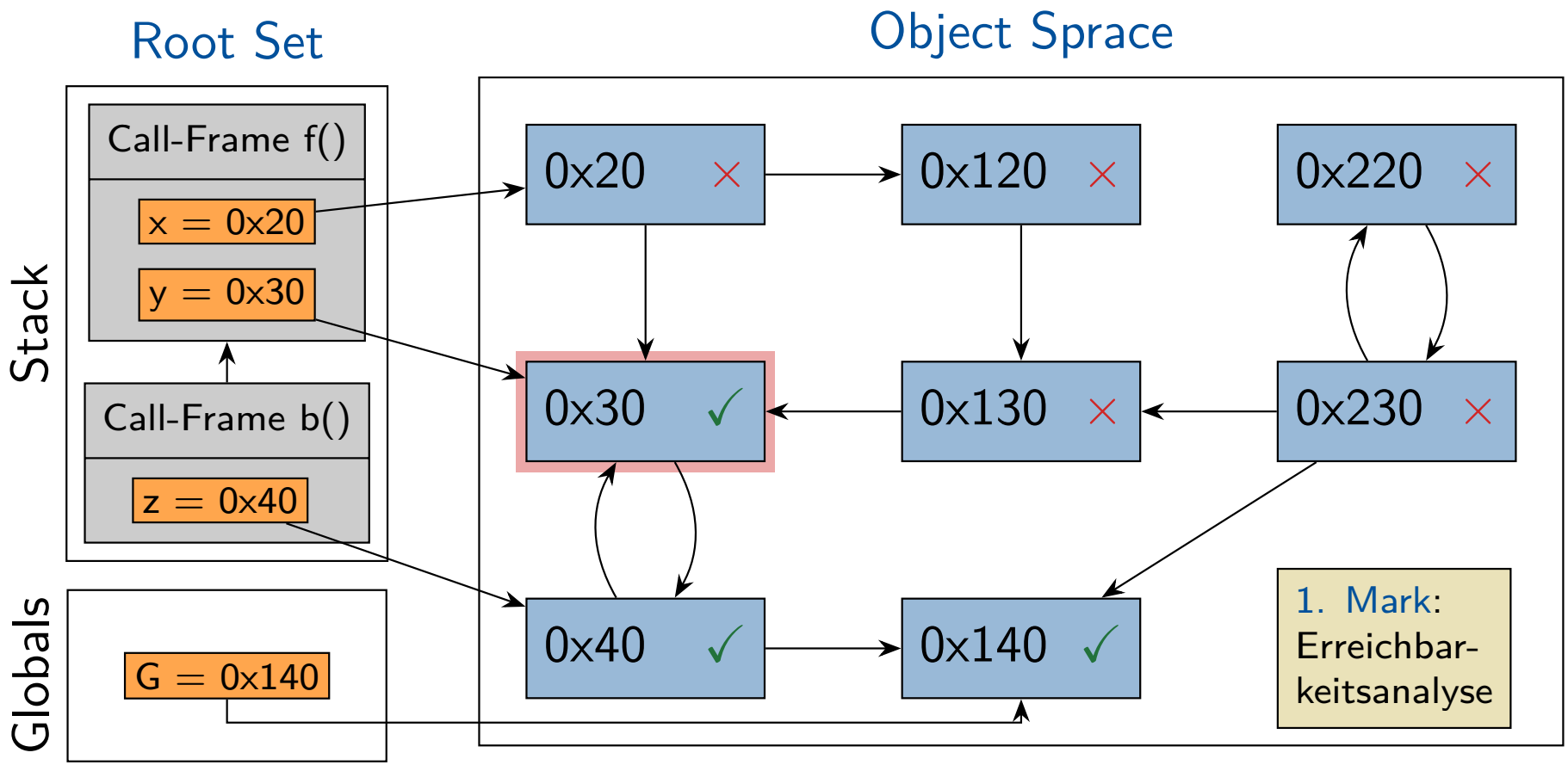
Technische Universität Braunschweig

Der Mark-and-Sweep Garbage Collector hält die Welt an, markiert alle erreichbaren Objekte, und gibt die anderen frei.



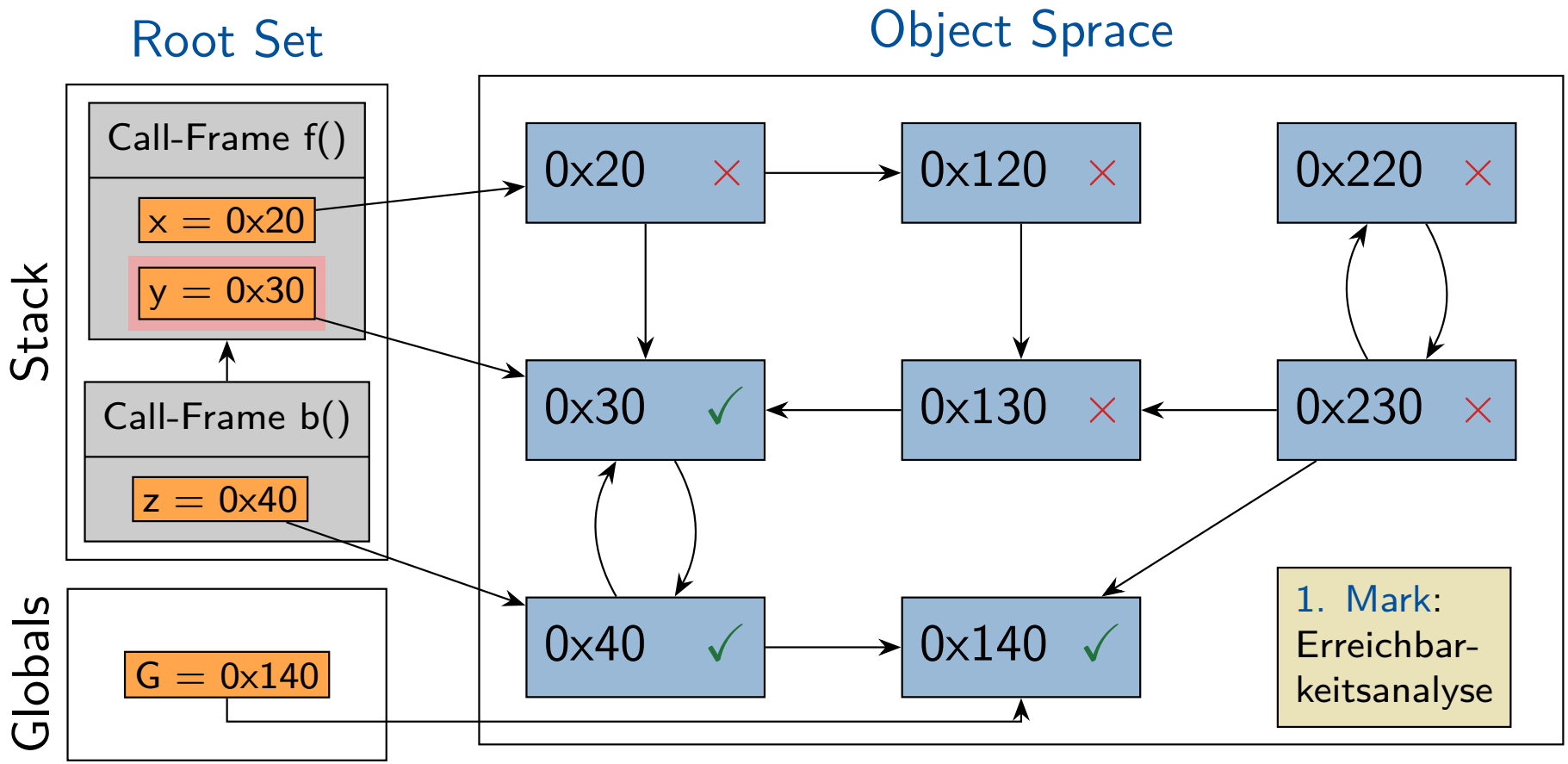
Mark-and-Sweep Garbage Collector

Der Mark-and-Sweep Garbage Collector hält die Welt an, markiert alle erreichbaren Objekte, und gibt die anderen frei.



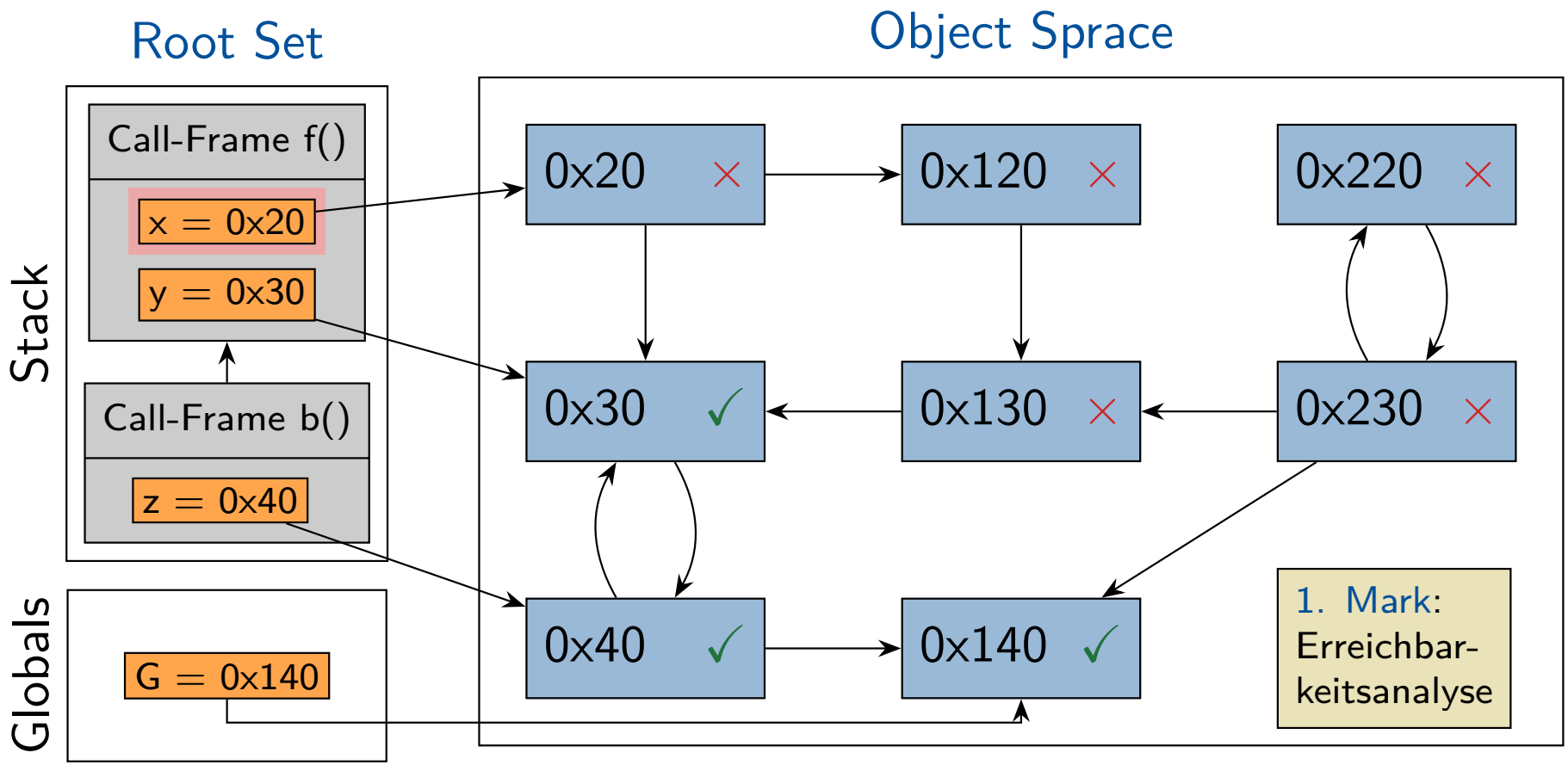
Mark-and-Sweep Garbage Collector

Der Mark-and-Sweep Garbage Collector hält die Welt an, markiert alle erreichbaren Objekte, und gibt die anderen frei.



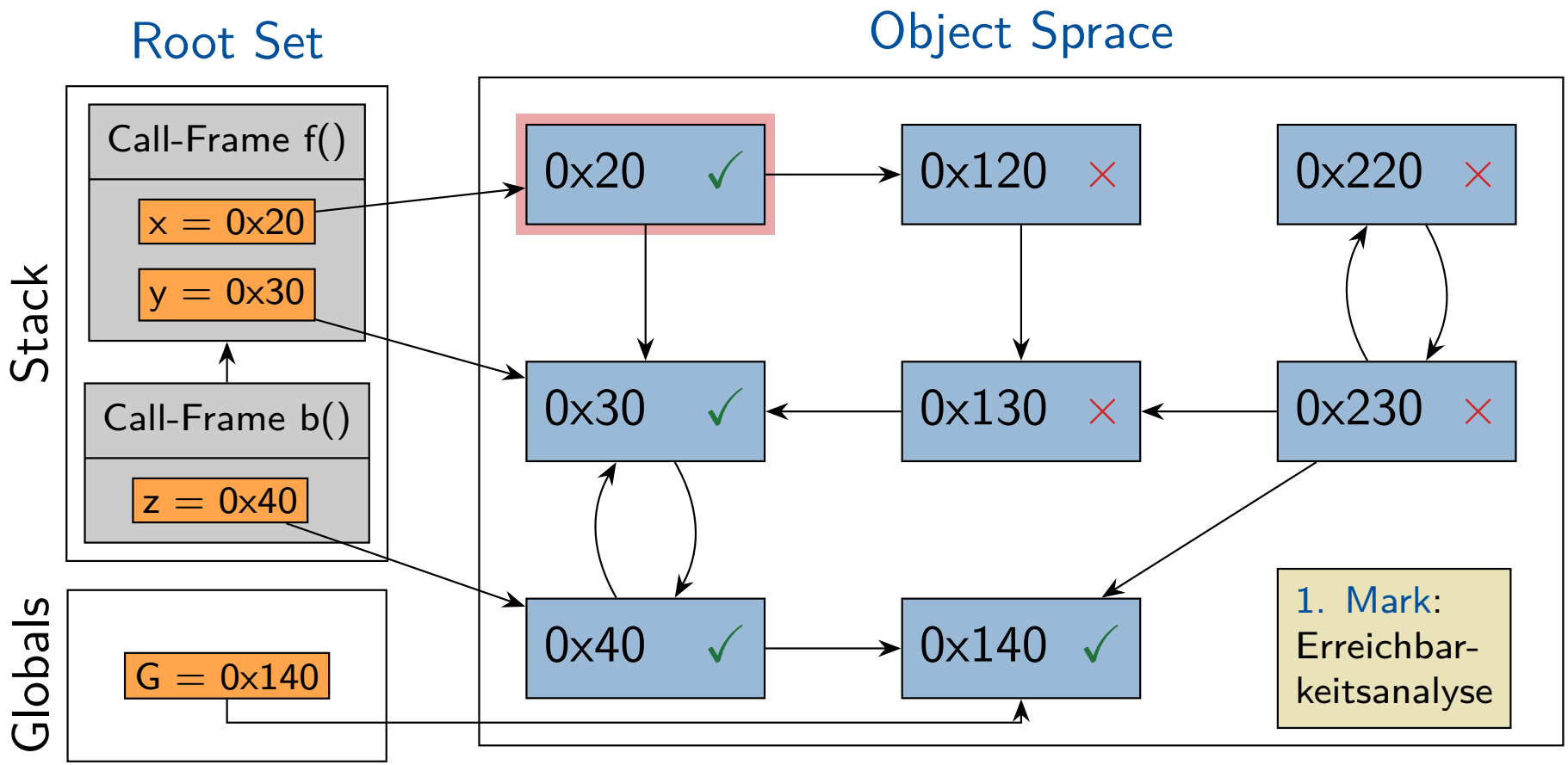
Mark-and-Sweep Garbage Collector

Der Mark-and-Sweep Garbage Collector hält die Welt an, markiert alle erreichbaren Objekte, und gibt die anderen frei.



Mark-and-Sweep Garbage Collector

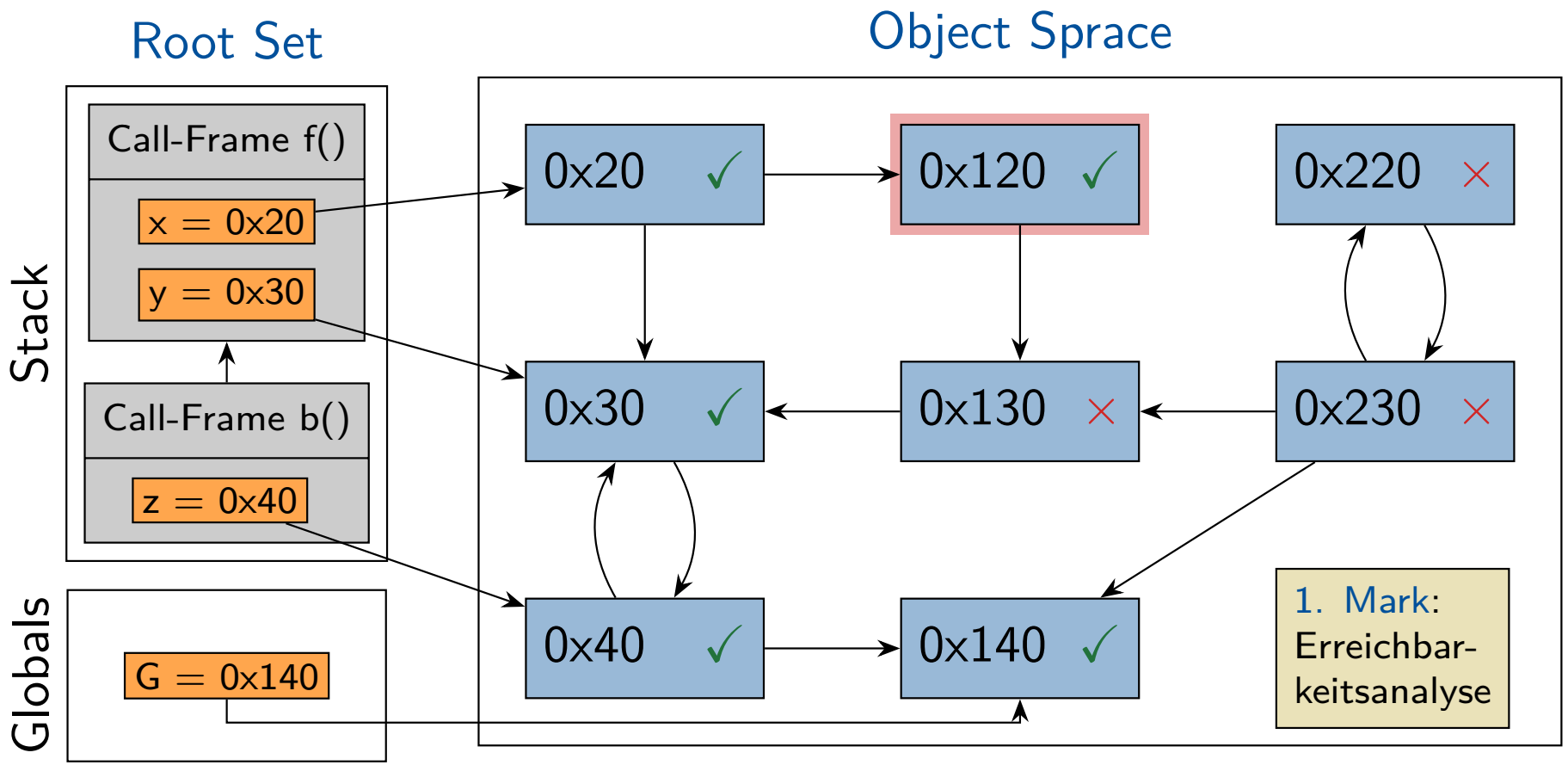
Der Mark-and-Sweep Garbage Collector hält die Welt an, markiert alle erreichbaren Objekte, und gibt die anderen frei.



Mark-and-Sweep Garbage Collector

Technische Universität Braunschweig

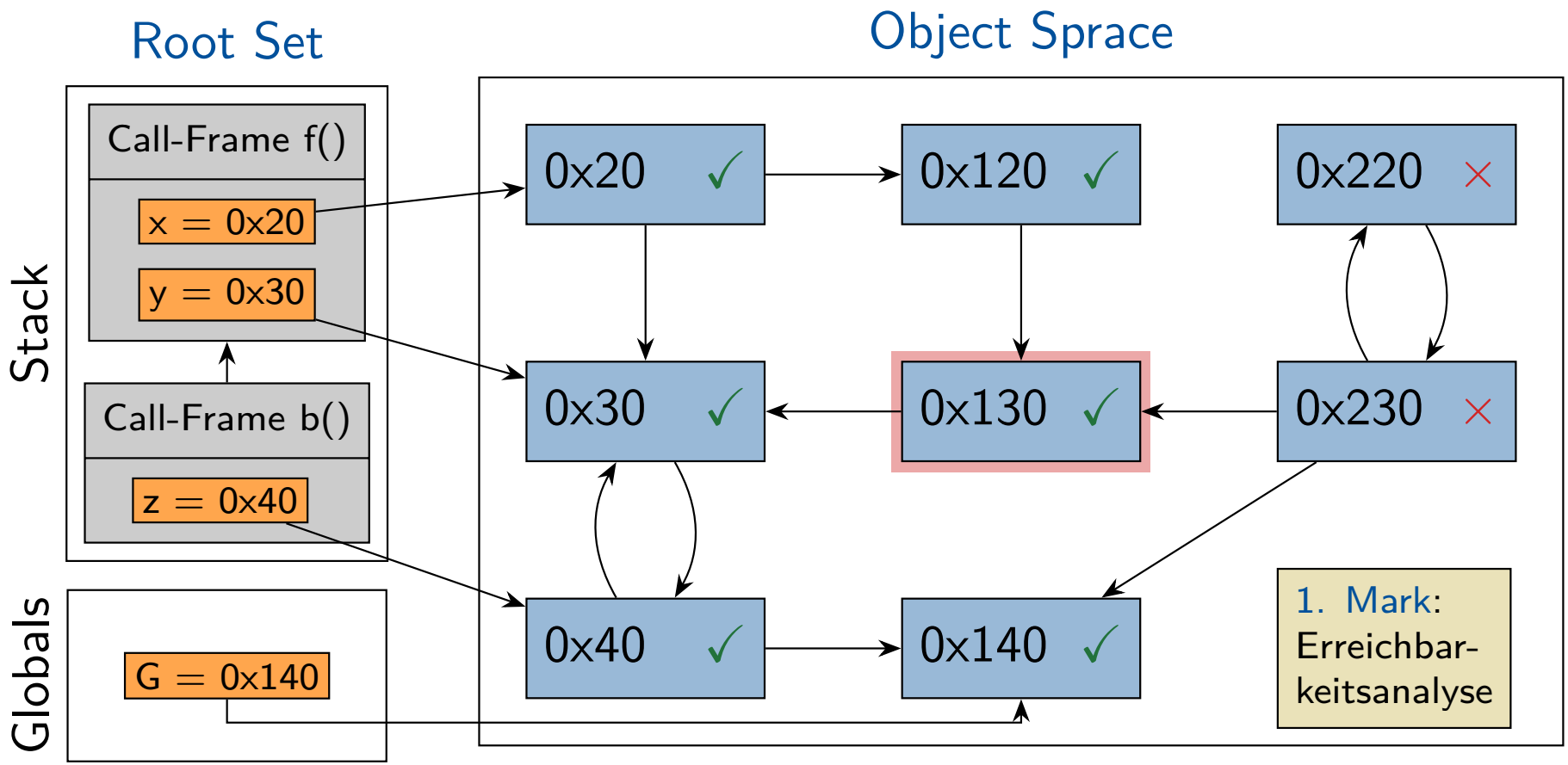
Der Mark-and-Sweep Garbage Collector hält die Welt an, markiert alle erreichbaren Objekte, und gibt die anderen frei.



Mark-and-Sweep Garbage Collector

Technische Universität Braunschweig

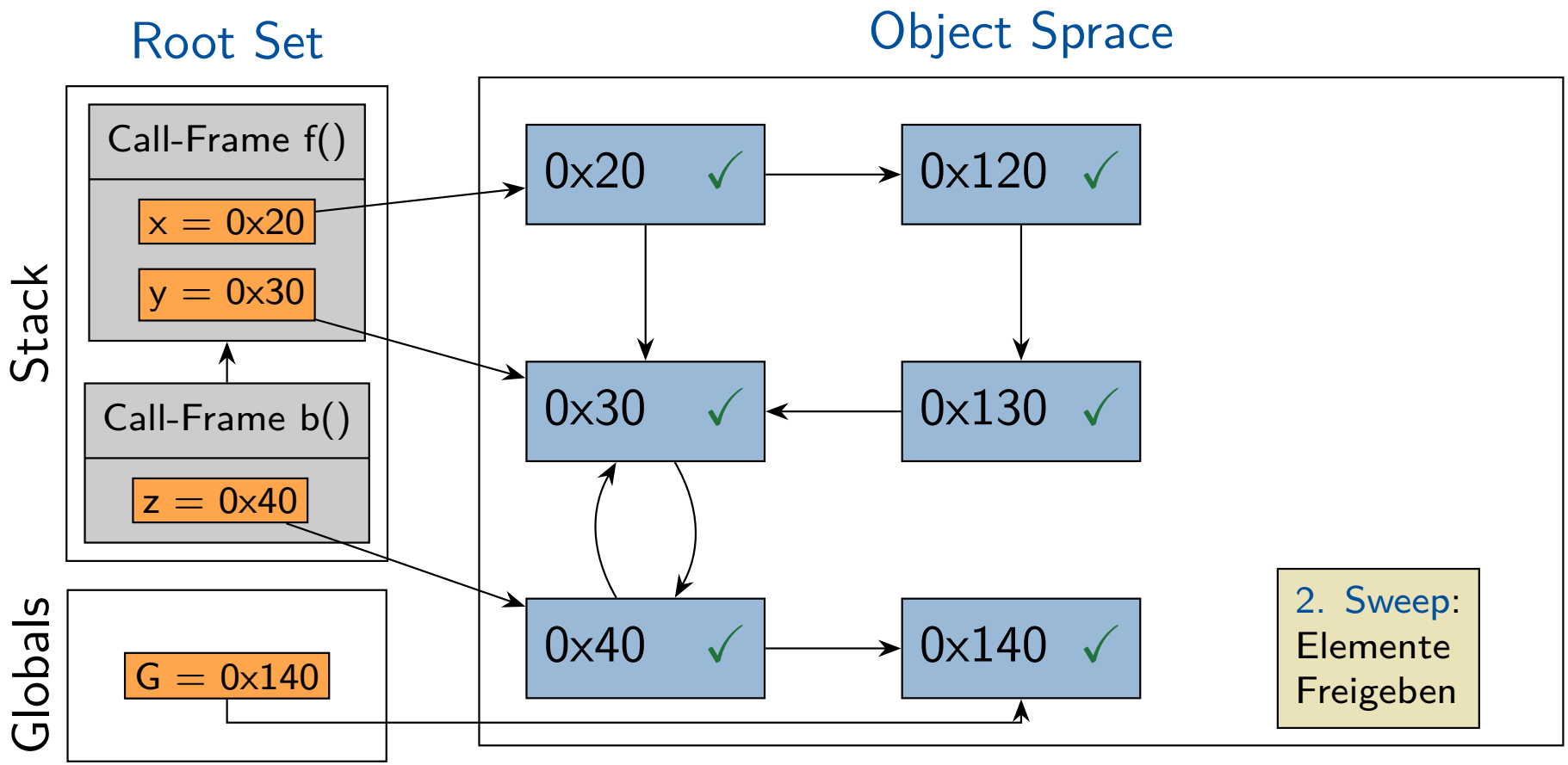
Der Mark-and-Sweep Garbage Collector hält die Welt an, markiert alle erreichbaren Objekte, und gibt die anderen frei.



Mark-and-Sweep Garbage Collector

Technische Universität Braunschweig

Der Mark-and-Sweep Garbage Collector hält die Welt an, markiert alle erreichbaren Objekte, und gibt die anderen frei.



Fallstudie: Der CPython3 Garbage Collector

- Leistungsfähiger GC für eine Sprache bei der alles ein Objekt ist
  - Kombination aus Referenzzählung und Mark-and-Sweep GC
  - 3 Generationen für unterschiedlich alte Objekte
  - Zugriff aus der Python-VM über `gc`-Module

```
import sys
import gc

x = [object()]

print("Refcount:",
      sys.getrefcount (x))
print("x -> *:",
      gc.get_referents(x))
print("* ->: x",
      gc.get_referrers(x))
print("Objects:",
      len(gc.get_objects()))
```

```
// +1 Für den Wurzelnamensraum
// +1 für temp-argument
Refcount: 2

// Ein- und Ausgehende Kanten
x -> *: [<object at 0x1000>]
* ->: x [<namespace: root>]

// Alle Objekte
Objects: 5823
```

Destruktoren

```
class A { ...
  A() {
    global->register(this);
    this->mem = malloc(3);
  }
```

```
class A { ...
  ~A() {
    free(mem);
    global->unregister(this);
  }
```

- Destruktoren sollen die Konstruktor-Seiteneffekte rückgängig machen
  - Seiteneffekte: Registrierung bei anderen Objekten, Ressourcennachforderung
  - Löschung oder Invalidierung aller Referenzen auf das Objekt
  - Destruktoren habe keine Parameter
- Destruktoren und Vererbung: Umgekehrte Konstruktionsreihenfolge
  - Destruktor für Derived muss ein gültiges Derived-Objekt vorfinden
  - `~Derived()` muss vor `~Base()` aufgerufen werden.
- Schwierige Semantik für Sprachen mit Garbage Collection
  - GC: Todeszeitpunkt und Zeitpunkt der Freigabe sind entkoppelt
  - GC müsste den Destruktor aufrufen, dies geschieht aber $IRGENDWANN

⇒ Manuelle Deinitialisierung und `void finalize()`

Fallstudie: Resource Aquisition is Initialization (RAII)

- C++: Kopplung von Scopes und der Lebenszeit lokaler Variablen
  - Wertemodell: Lebenszeit von Variable und enthaltenem Objekt sind gleich
  - Definition einer Variable: $\Rightarrow$ Konstruktor wird aufgerufen
  - Ende des umgebenden Scopes: $\Rightarrow$ Destruktor wird aufgerufen

```C++
class log {
  level_t level;
  int fd;
  public:
    log() : level(DEBUG) {
      fd = open("/dev/stderr");
    }
    ...
    ~log(){
      close(fd);
    }
};

void foo() {
  log L; // Constructor
  L.log(...)
} // implicit: Destructor
```

- Destruktor wird immer aufgerufen!
  - Ausführung erreicht Scope-Ende
  - Vorzeitiges `return`
  - Exceptions (direkt und indirekt)

$\Rightarrow$ Garantierte Ressourcenfreigabe

- Anwendbar für alle Ressourcentypen

```
mutex lock; // Das Lock-Objekt

void foo() {
  lock_guard<mutex> X(lock);

  // Implizites Unlock
}
```

Fallstudie: Resource Aquisition is Initialization (RAII)

- C++: Kopplung von Scopes und der Lebenszeit lokaler Variablen
  - Wertemodell: Lebenszeit von Variable und enthaltenem Objekt sind gleich
  - Definition einer Variable: ⇒ Konstruktor wird aufgerufen

Bjarne Stroustrup (Designer von C++):

„The RAII technique [...] is a clumsy name for a **central concept** [that] happens to be **necessary** for exception handling. [...] the **main tool** for resource management is constructors and destructors.“

Interview mit Bill Venners, 2003

```
};

void foo() {
  log L; // Constructor
  L.log(...)
} // implicit: Destructor
```

C++

```
mutex lock; // Das Lock-Objekt

void foo() {
  lock_guard<mutex> X(lock);

  // Implizites Unlock
}
```

Zusammenfassung

- Objekte sind ein **existentieller Verbund** von Informationen.
  - Objekte transportieren Informationen (typsicher) im Programmablauf.
  - Referenz- und Wertemodell für Variablen
- **Geburt**: In einem Speicherbereich entsteht ein Objekt
  - Allokation des nötigen Speichers auf dem Stack, im Heap, oder statisch.
  - Konstruktoren setzen Felder, fordern Ressourcen an und etablieren Invarianten
- **Leben**: Kontrollierter und gefilterter Zugriff auf die enthaltenen Daten
  - Wer greift wie, mit welchem Recht, auf das Objekt zu?
  - Der Besitz von Objekten verpflichtet zu ihrer Pflege
- **Tod**: Objekt verliert den letzten Besitzer oder die letzte Referenz
  - Manuelles Management, Referenzzähler oder automatische Garbage Collection
  - Destruktoren geben angeforderte Ressourcen wieder frei