Work in progress

Vom Quellcode zum „Programm in Ausführung“

Ute Bormann, TI2

2023-10-13

Inhalt

1. Der Prozess-Adressraum
2. Der Linker
3. Die Symboltabelle

Teil 1:
Der Prozess-Adressraum

- Ein Beispiel-Programm

```
int ndigit[10] = {0,0,0,0,0,0,0,0,0,0};

void count() {
  char c;
  while (cin.get(c)) {
    if (c>='0' && c<='9') {
      ndigit[c-'0']++;
    }
  }
}
```

```
void print() {
  int i;
  for (i=0; i<10; i++) {
    cout << ndigit[i] << ' ';
  }
  cout << endl;
}

main() {
  count();
  print();
}
```

Programm soll ausgeführt werden:

Einfacheres Beispiel:

```
int twoton (int i) {  //2^i
  int r=1;
  while (i>0) {
    i--;
    r<<=1;
  }
  return r;
}
```

Programm soll ausgeführt werden:

a) wird vom Compiler in Maschinencode übersetzt

Einfacheres Beispiel:

```
int twoton (int i) {  //2^i
  int r=1;
  while (i>0) {
    i--;
    r<<=1;
  }
  return r;
}
```

⇒ Assemblerprogramm (vereinfacht):

```
twoton_Fi:   mov 1,%g2
             cmp %o0,0
             ble .LL3
      .LL4:  add %o0,-1,%o0
             sll %g2,1,%g2
             cmp %o0,0
             bg .LL4
      .LL3:  mov %g2,%o0
             retl
```

Programm soll ausgeführt werden:

a) wird vom Compiler in Maschinencode übersetzt

Einfacheres Beispiel:

```
int twoton (int i) {  //2^i
  int r=1;
  while (i>0) {
    i--;
    r<<=1;
  }
  return r;
}
```

⇒ Assemblerprogramm (vereinfacht):

```
twoton_Fi:  mov 1,%g2
            cmp %o0,0
            ble .LL3
.LL4:       add %o0,-1,%o0
            sll %g2,1,%g2
            cmp %o0,0
            bg .LL4
.LL3:       mov %g2,%o0
            retl
```

Programm soll ausgeführt werden:

a) wird vom Compiler in Maschinencode übersetzt

b) wird in Speicher geladen und abgearbeitet

⇒ Labels auf Adressen abbilden (Annahme: 32-Bit-Maschine)

Einfacheres Beispiel:

```
int twoton (int i) {  //2^i
  int r=1;
  while (i>0) {
    i--;
    r<<=1;
  }
  return r;
}
```

⇒ Assemblerprogramm (vereinfacht):

0	twoton_Fi:	mov 1,%g2	4 Bytes
4		cmp %o0,0	
8		ble .LL3 28	
12	.LL4:	add %o0,-1,%o0	
16		sll %g2,1,%g2	
20		cmp %o0,0	
24		bg .LL4 12	
28	.LL3:	mov %g2,%o0	
32		retl	

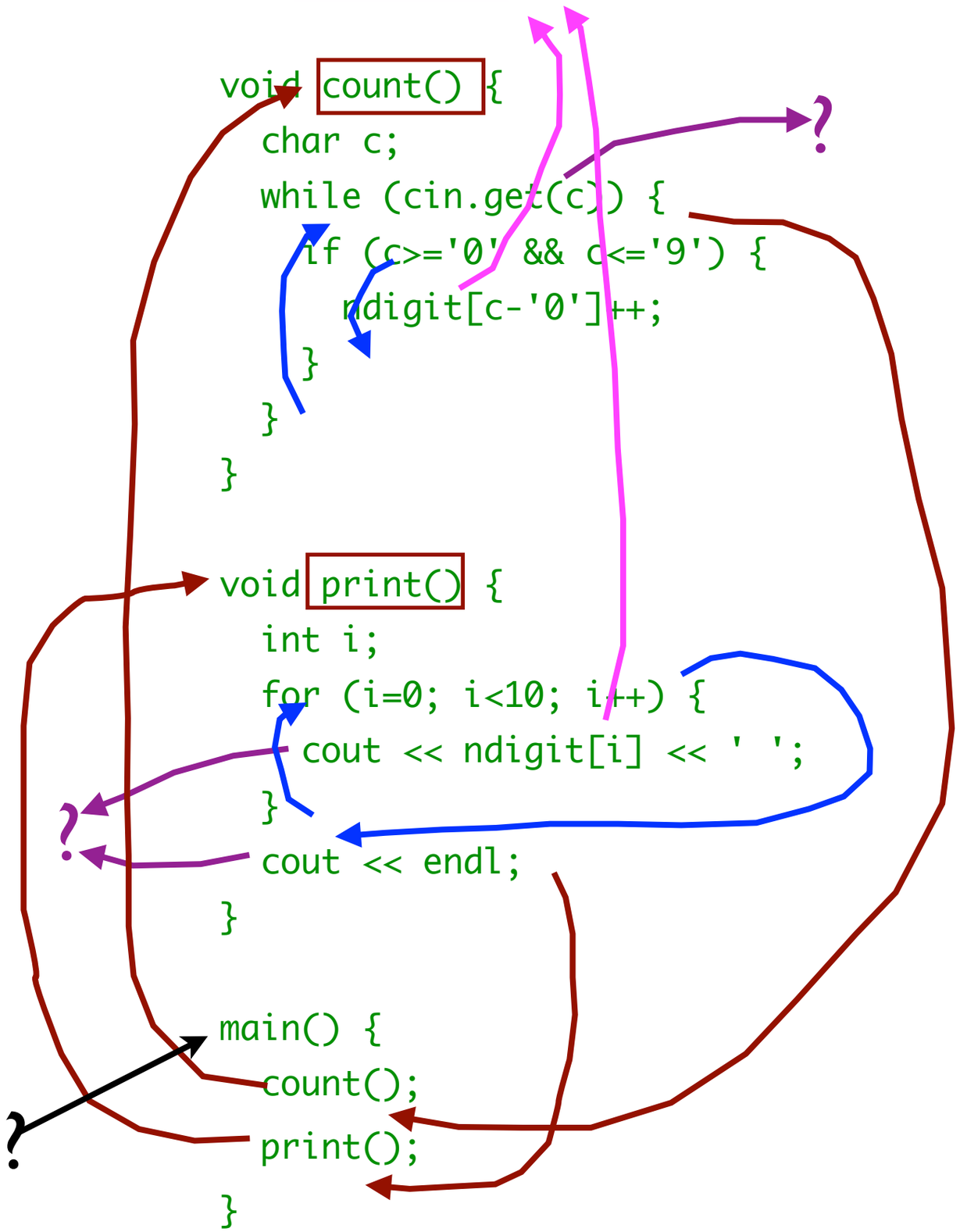
- Ein Beispiel-Programm

```
int ndigit[10] = {0,0,0,0,0,0,0,0,0,0};

void count() {
  char c;
  while (cin.get(c)) {
    if (c>='0' && c<='9') {
      ndigit[c-'0']++;
    }
  }
}
```

```
void print() {
  int i;
  for (i=0; i<10; i++) {
    cout << ndigit[i] << ' ';
  }
  cout << endl;
}

main() {
  count();
  print();
}
```

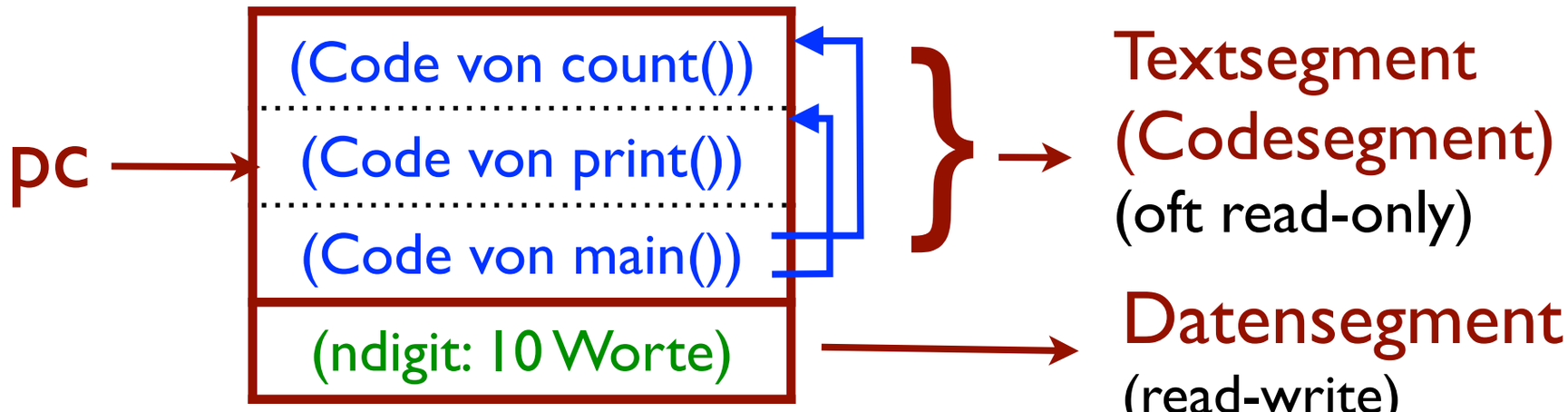
- Ein Beispiel-Programm

```
int ndigit[10] = {0,0,0,0,0,0,0,0,0,0};

void count() {
  char c;
  while (cin.get(c)) {
    if (c>='0' && c<='9') {
      ndigit[c-'0']++;
    }
  }
}

void print() {
  int i;
  for (i=0; i<10; i++) {
    cout << ndigit[i] << ' ';
  }
  cout << endl;
}

main() {
  count();
  print();
}
```



Programm soll ausgeführt werden

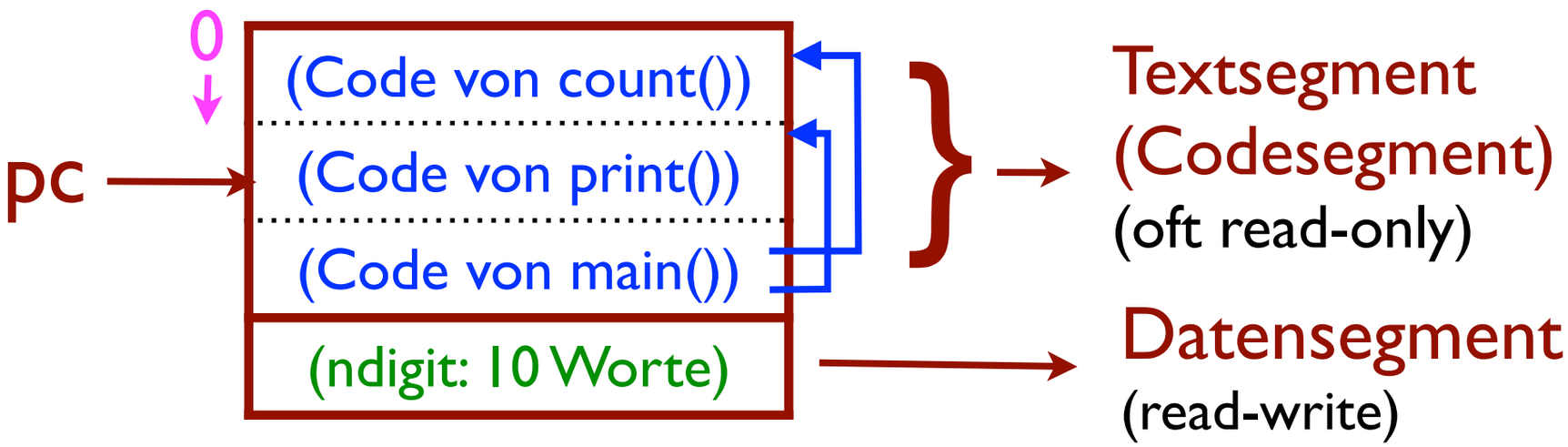
a) wird vom Compiler in Maschinencode übersetzt
b) wird in Speicher geladen und abgearbeitet



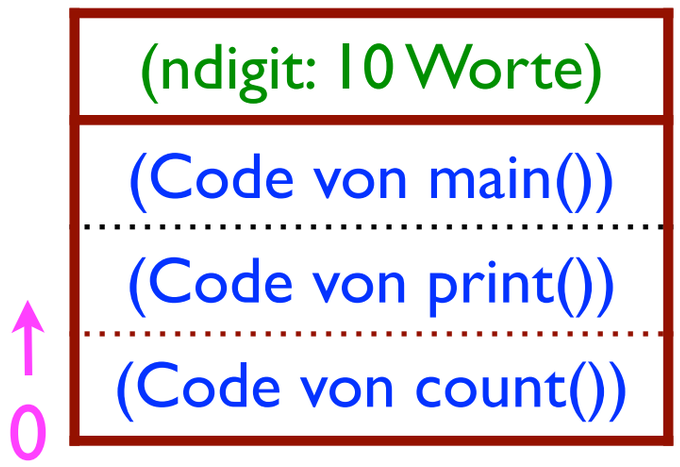
⇒ Programm läuft in einem Adressraum

Programm soll ausgeführt werden

a) wird vom Compiler in Maschinencode übersetzt
b) wird in Speicher geladen und abgearbeitet



oder



⇒ Programm läuft in einem Adressraum

- Jedoch: Im Modell beginnt Zählung oft von „unten“

- Während der Laufzeit zusätzlicher Platz für temporäre Informationen erforderlich
  - Retten des Status bei Prozeduraufrauf
  - Parameter der Prozedur
  - Lokale Variablen der Prozedur

⇒ Extra-Bereich ⇒ Stack

alternativ: Register nutzen

- Stack wächst/schrumpft dynamisch nach Bedarf

SP: Stack Pointer

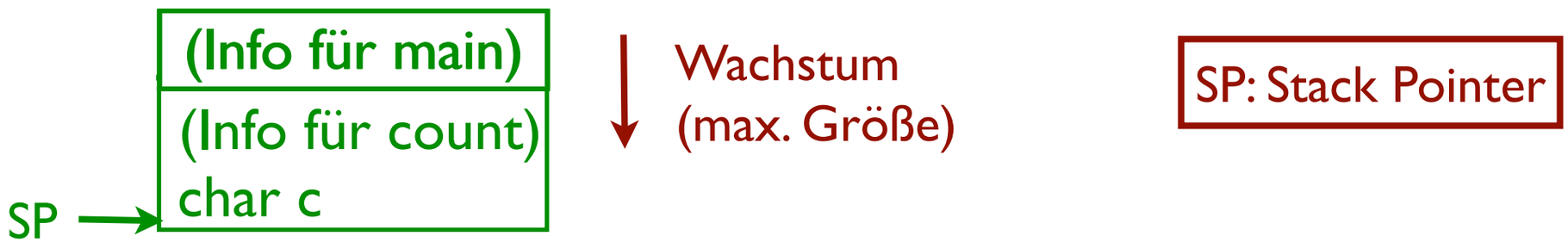
⇒ Stack frame / Activation Record

- Während der Laufzeit zusätzlicher Platz für temporäre Informationen erforderlich
  - Retten des Status bei Prozeduraufrauf
  - Parameter der Prozedur
  - Lokale Variablen der Prozedur

⇒ Extra-Bereich ⇒ Stack

alternativ: Register nutzen

- Stack wächst/schrumpft dynamisch nach Bedarf



⇒ Stack frame / Activation Record

- Während der Laufzeit zusätzlicher Platz für temporäre Informationen erforderlich
  - Retten des Status bei Prozeduraufrauf
  - Parameter der Prozedur
  - Lokale Variablen der Prozedur

⇒ Extra-Bereich ⇒ Stack

alternativ: Register nutzen

- Stack wächst/schrumpft dynamisch nach Bedarf

SP: Stack Pointer

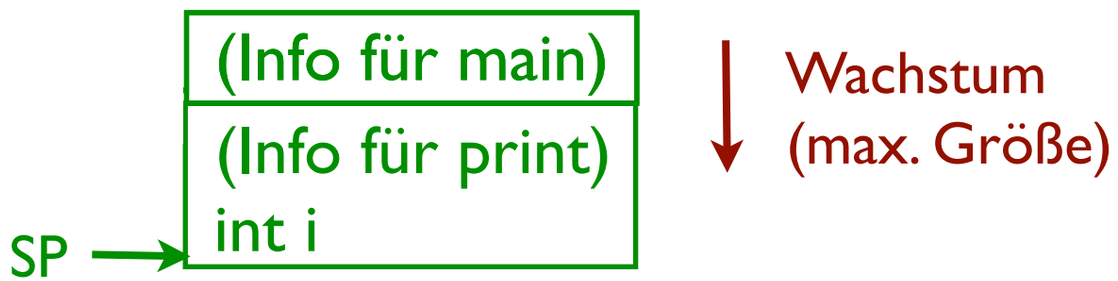
⇒ Stack frame / Activation Record

- Während der Laufzeit zusätzlicher Platz für temporäre Informationen erforderlich
  - Retten des Status bei Prozeduraufrauf
  - Parameter der Prozedur
  - Lokale Variablen der Prozedur

⇒ Extra-Bereich ⇒ Stack

alternativ: Register nutzen

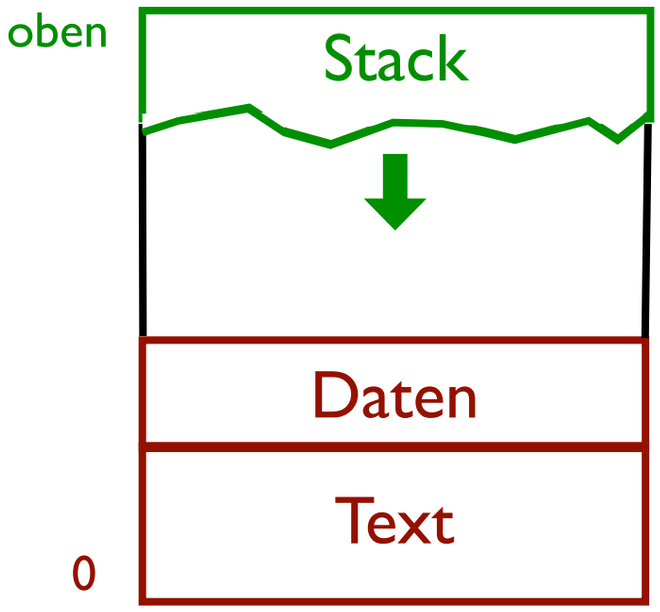
- Stack wächst/schrumpft dynamisch nach Bedarf



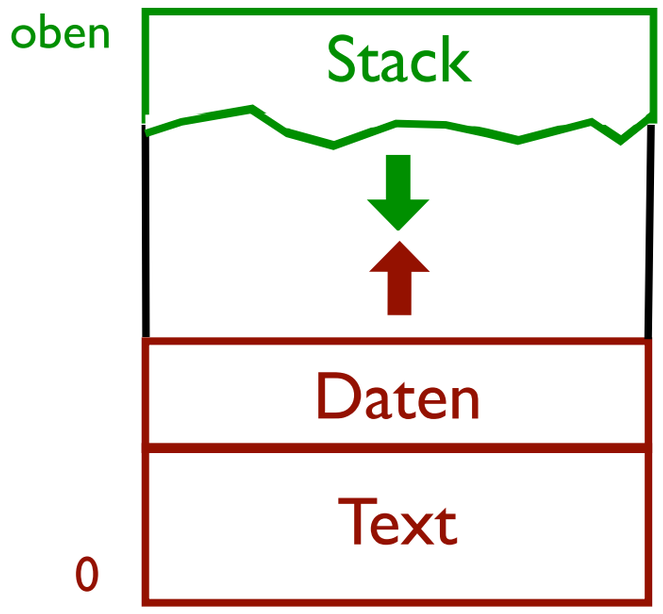
SP: Stack Pointer

⇒ Stack frame / Activation Record

- Stack muss von Text- und Datensegment entkoppelt werden:

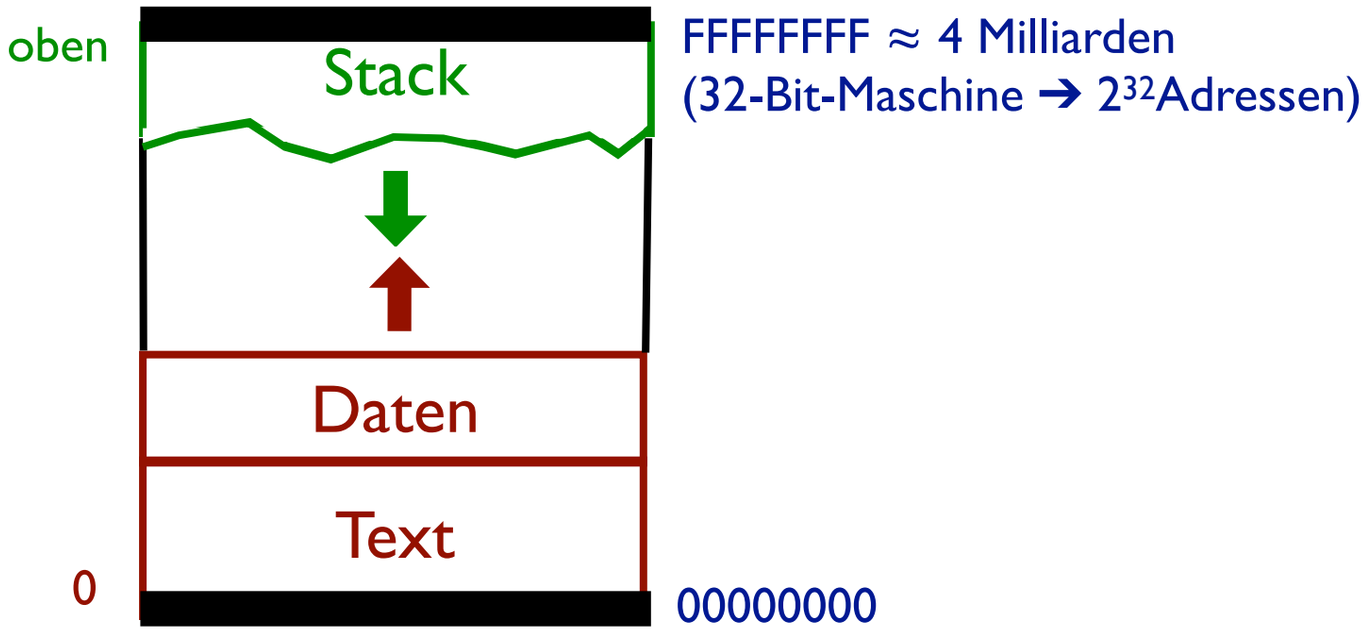


- Stack muss von Text- und Datensegment entkoppelt werden:



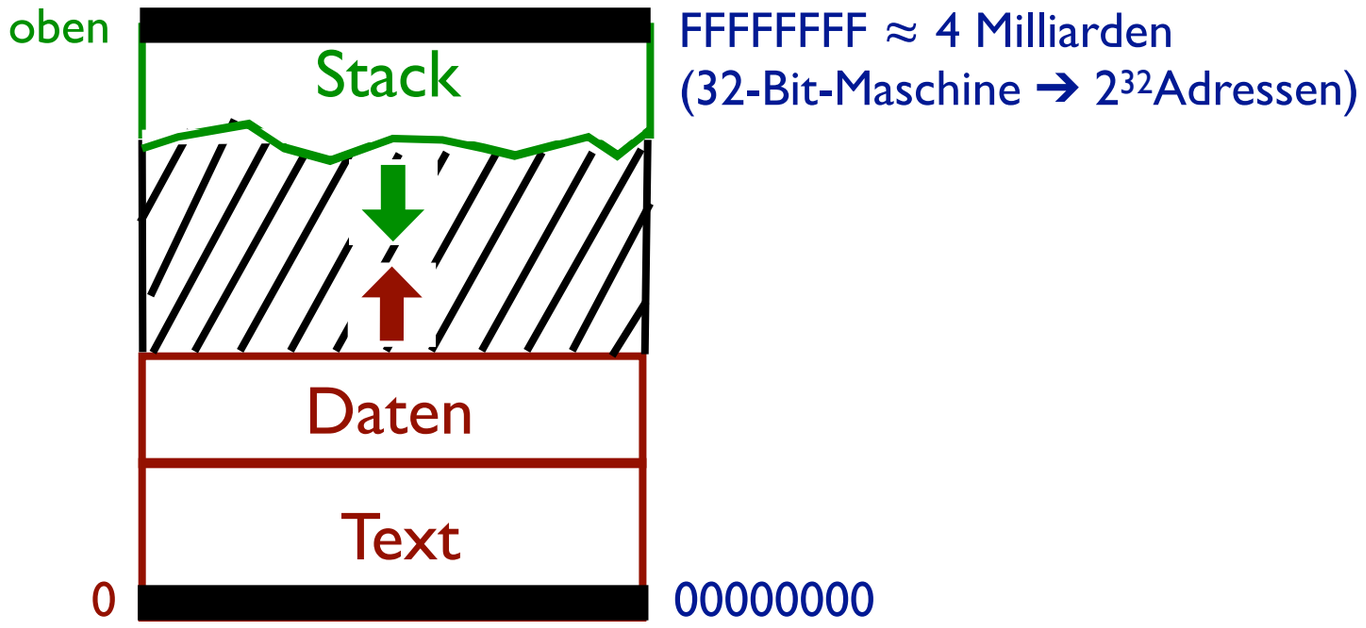
- Programme erzeugen u.U. weitere Daten während der Laufzeit (new…)
  ⇒ Vergrößerung des Datensegments

- Stack muss von Text- und Datensegment entkoppelt werden:



- Programme erzeugen u.U. weitere Daten während der Laufzeit (new…)
  ⇒ Vergrößerung des Datensegments
- Adressraum hat gewisse Größe (z.B. 2^{32} Bytes)
- „Nullpointer“ vermeiden, ggf. auch reservierte Adressen im oberen Bereich… (in Beispielen oft ignoriert)

- Stack muss von Text- und Datensegment entkoppelt werden:

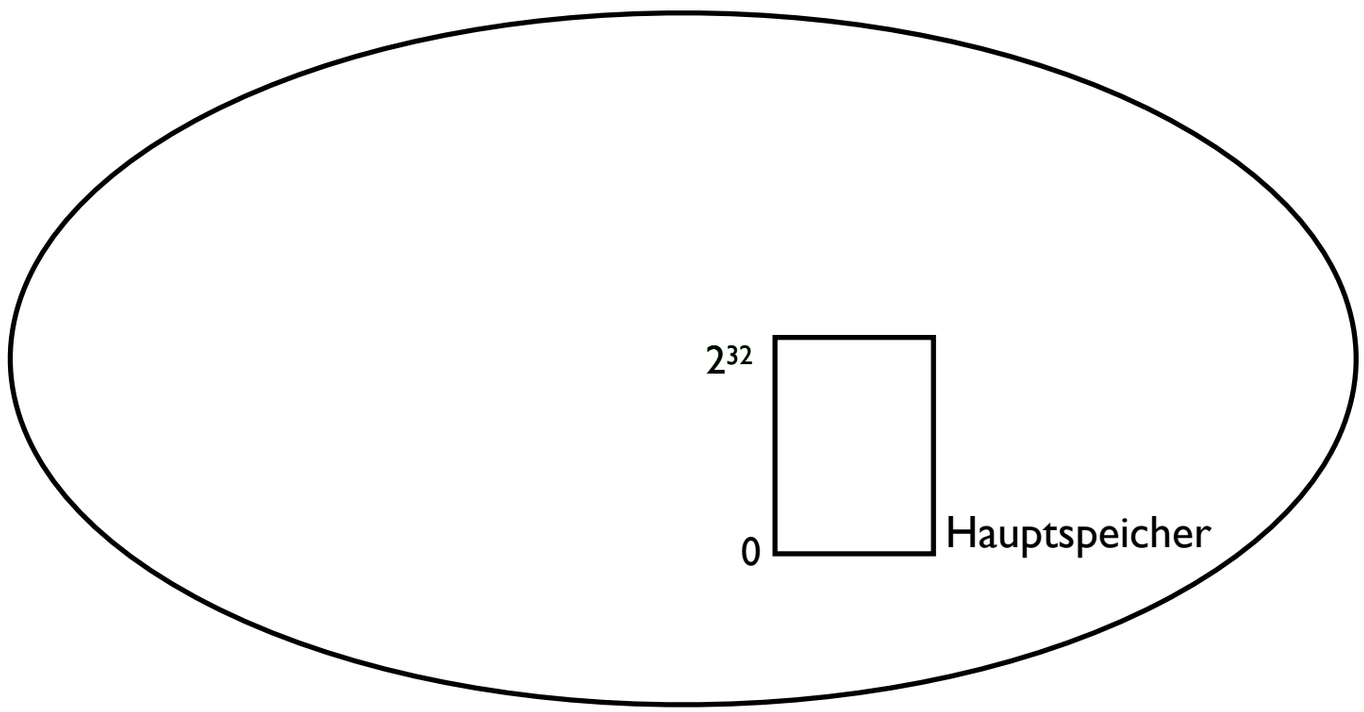


- Programme erzeugen u.U. weitere Daten während der Laufzeit (new…)
  ⇒ Vergrößerung des Datensegments
- Adressraum hat gewisse Größe (z.B. 2^{32} Bytes)
- „Nullpointer“ vermeiden, ggf. auch reservierte Adressen im oberen Bereich… (in Beispielen oft ignoriert)
- Restliche Adressen sind „ungenutzt“ (= ungültig)

  ⇒ virtueller Adressraum (prozesslokal)

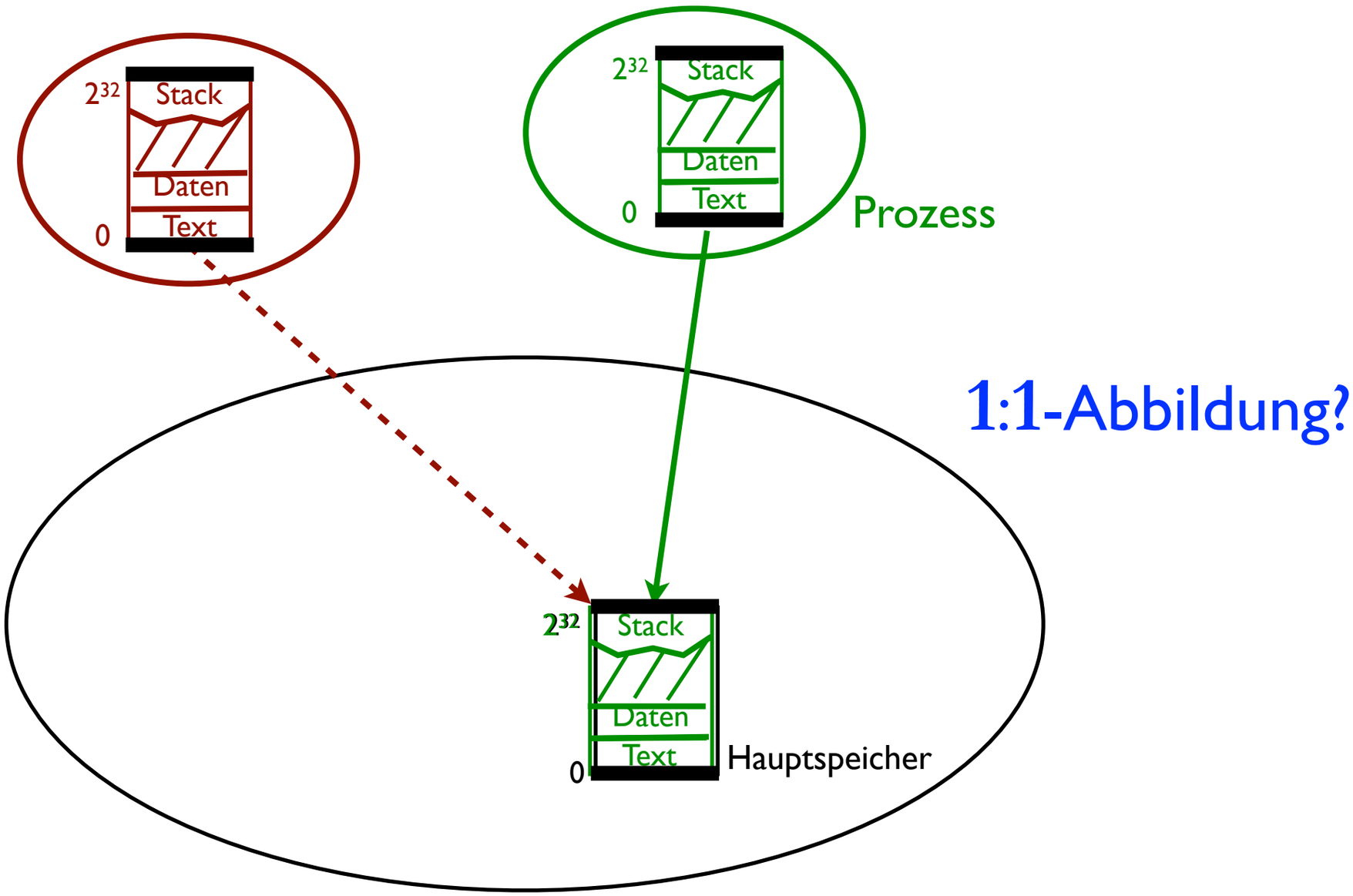
Benutzung des Betriebssystems (vereinfacht)

- Betriebssystem verwaltet realen Hauptspeicher
- Virtueller Adressraum des Prozesses muss darauf abgebildet werden



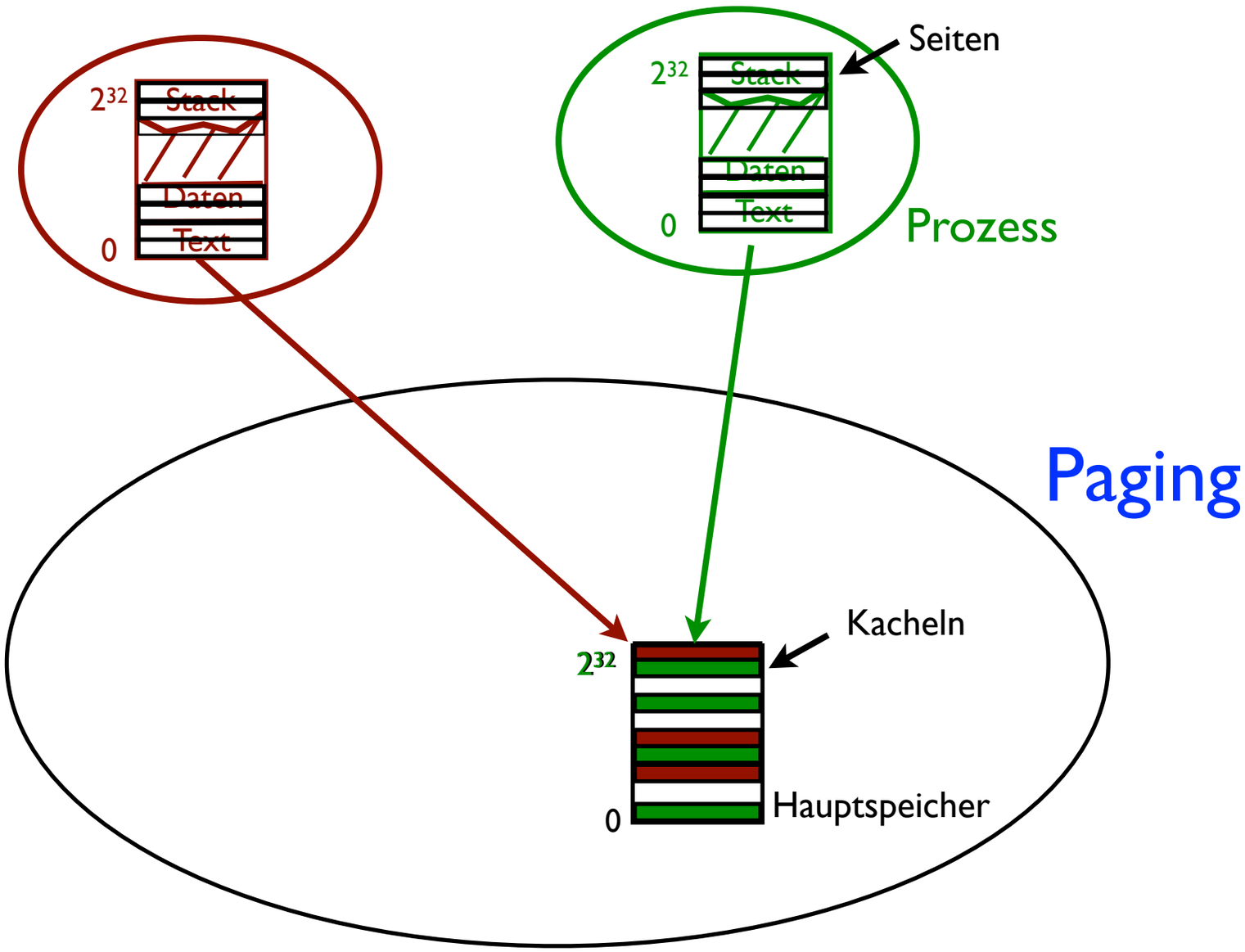
Benutzung des Betriebssystems (vereinfacht)

- Betriebssystem verwaltet realen Hauptspeicher
- Virtueller Adressraum des Prozesses muss darauf abgebildet werden

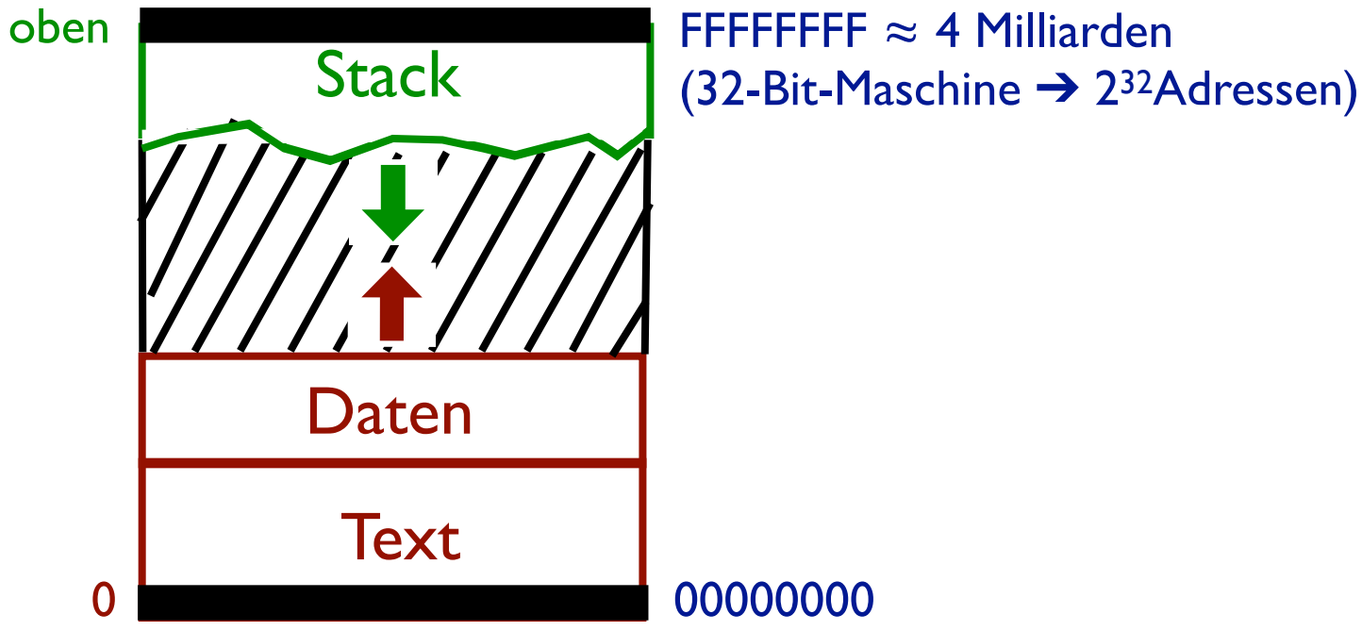


Benutzung des Betriebssystems (vereinfacht)

- Betriebssystem verwaltet realen Hauptspeicher
- Virtueller Adressraum des Prozesses muss darauf abgebildet werden



- Stack muss von Text- und Datensegment entkoppelt werden:



- Programme erzeugen u.U. weitere Daten während der Laufzeit (new…)
  ⇒ Vergrößerung des Datensegments
- Adressraum hat gewisse Größe (z.B. 2^{32} Bytes)
- „Nullpointer“ vermeiden, ggf. auch reservierte Adressen im oberen Bereich… (in Beispielen oft ignoriert)
- Restliche Adressen sind „ungenutzt“ (= ungültig)

  ⇒ nicht realer Hauptspeicher, damit mehrere Prozesse aktiv sein können
  ⇒ virtueller Adressraum (prozesslokal) (+ Abbildung auf Hauptspeicher)

- Ein Beispiel-Programm

```
#include <iostream.h>
int ndigit[10] = {0,0,0,0,0,0,0,0,0,0};

void count() {
  char c;
  while (cin.get(c)) {
    if (c>='0' && c<='9') {
      ndigit[c-'0']++;
    }
  }
}

void print() {
  int i;
  for (i=0; i<10; i++) {
    cout << ndigit[i] << ' ';
  }
  cout << endl;
}

main() {
  count();
  print();
}
```

Kleine Aufgabe

```
#include <unistd.h>
#include <stdio.h>

extern int errno;

void loeschen(char *filename) {
  if (unlink(filename) == -1) {
    printf("Fehler %d\n", errno);
  } else {
    printf("Datei %s geloescht\n", filename);
  }
}


int main(int argc, char **argv) {
  for (int i = 1; i < argc; i++)
    loeschen(argv[i]);
}
```

Welche Zugriffe/Sprünge sind beim Kompilieren noch nicht auflösbar? Was fehlt zum Auflösen?

- Ein Beispiel-Programm

```
#include <iostream.h>
int ndigit[10] = {0,0,0,0,0,0,0,0,0,0};

void count() {
  char c;
  while (cin.get(c)) {
    if (c>='0' && c<='9') {
      ndigit[c-'0']++;
    }
  }
}

void print() {
  int i;
  for (i=0; i<10; i++) {
    cout << ndigit[i] << ' ';
  }
  cout << endl;
}

main() {
  count();
  print();
}
```

+ Bibliothekfunktionen
  (z.B. cin.get())

+ „Laufzeitsystem“
⇒ Ansprung von main()

  (z.B. crt0.o)

Bibliotheken (Libraries)

- Ansammlung von allgemein verfügbaren „Modulen“
- Module enthalten Prozeduren und/oder Datenobjekte, auf die aus Programmen heraus zugegriffen werden kann

  ⇒ „Rad“ muss nicht immer neu erfunden werden

  z.B. cin.get() Eingabe

  cout <<... Ausgabe

  ⇒ Bestandteile der C++-Bibliothek

- Übrigens: „Systemaufrufe“ (read(), write()) sind auch Bibliotheksroutinen

  ⇒ werden intern auf „echte“ Systemaufrufe abgebildet

Bibliotheken (Libraries)

- Ansammlung von allgemein verfügbaren „Modulen“
- Module enthalten Prozeduren und/oder Datenobjekte, auf die aus Programmen heraus zugegriffen werden kann

  ⇒ „Rad“ muss nicht immer neu erfunden werden

  z.B. cin.get() Eingabe

  cout <<... Ausgabe

  ⇒ Bestandteile der C++-Bibliothek

- Übrigens: „Systemaufrufe“ (read(), write()) sind auch Bibliotheksroutinen

  ⇒ werden intern auf „echte“ Systemaufrufe abgebildet

- Bibliotheksroutinen müssen zur Ausführung im selben Adressraum angesiedelt werden wie „Anwendungsprogramm“

  ⇒ gemeinsam übersetzen?

Fragen – Teil 1

- In welche Bereiche (*Segmente*) ist der *(virtuelle) Adressraum* eines Programms in Ausführung in Unix unterteilt, und welche Eigenschaften kennzeichnen sie?
- Wozu wird der *Stack* verwendet?
- Welchem Zweck dienen Bibliotheken (*Libraries*)?

Teil 2:
Der Linker

Getrennte Übersetzung von Programmen

- Gemeinsames Übersetzen von riesigen Programmen zu umständlich/zeitaufwendig

  ⇒ Teile einzeln übersetzen (bei Bedarf)

- Ein Beispiel-Programm

count.cc

```
int ndigit[10] = {0,0,0,0,0,0,0,0,0,0};
void count() {
  //… wie bisher
}
```

print.cc

```
void print() {
  // … wie bisher
}
```

main.cc

```
main() {
  count();
  print();
}
```

- Ein Beispiel-Programm

count.cc

```
#include <iostream>
#include “count.hh”
int ndigit[10] = {0,0,0,0,0,0,0,0,0,0};
void count() {
  //… wie bisher
}
```

count.hh

```
void count();
```

print.cc

```
#include <iostream>
#include “count.hh”
#include “print.hh”
void print() {
  // … wie bisher
}
```

print.hh

```
void print();
```

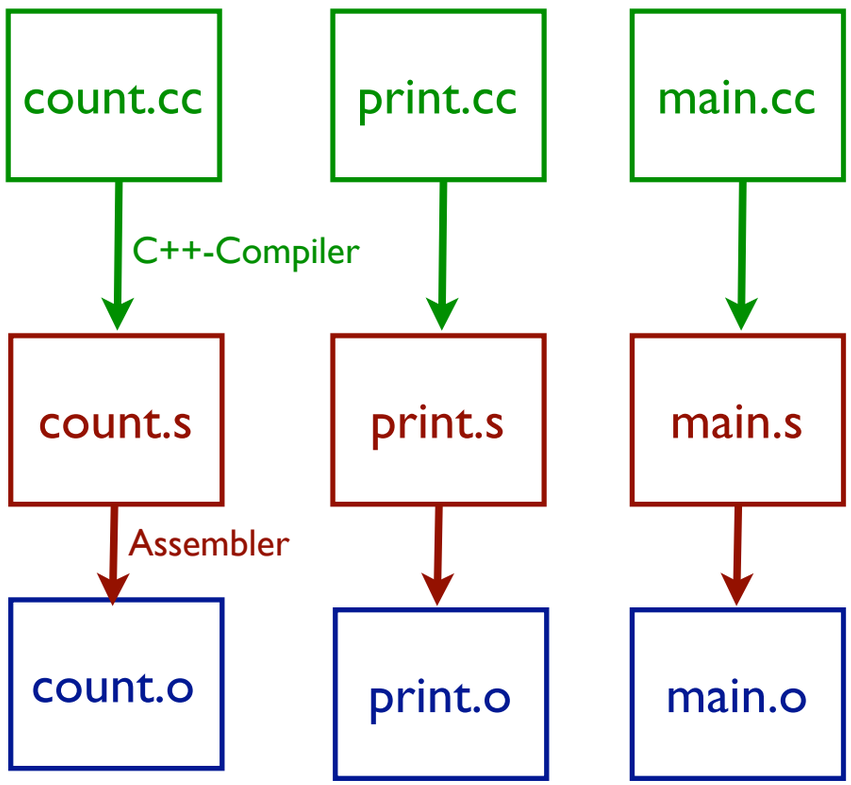
main.cc

```
#include “count.hh”
#include “print.hh”
main() {
  count();
  print();
}
```

Getrennte Übersetzung von Programmen

- Gemeinsames Übersetzen von riesigen Programmen zu umständlich/zeitaufwendig

  ⇒ Teile einzeln übersetzen (bei Bedarf)

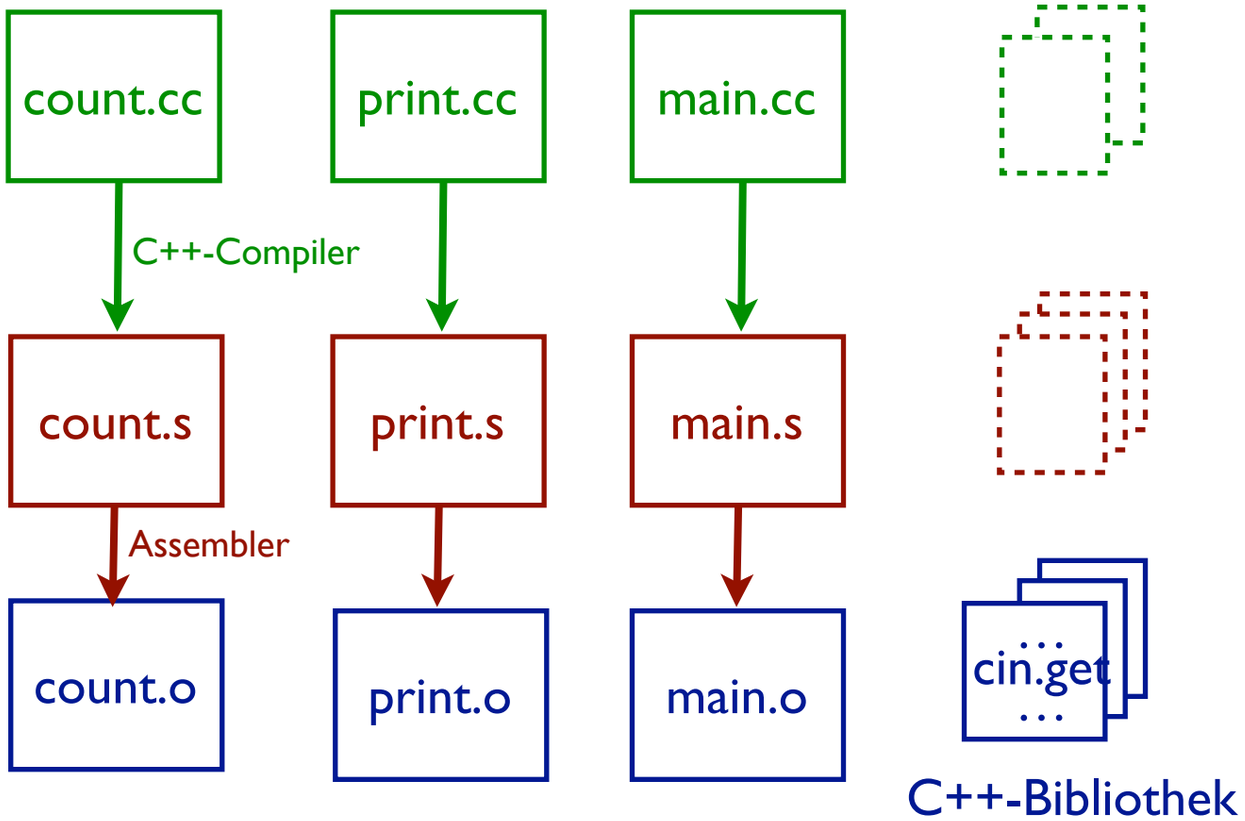


Getrennte Übersetzung von Programmen

- Gemeinsames Übersetzen von riesigen Programmen zu umständlich/zeitaufwendig

  ⇒ Teile einzeln übersetzen (bei Bedarf)

  ⇒ Bibliotheken liegen i.d.R. bereits übersetzt vor

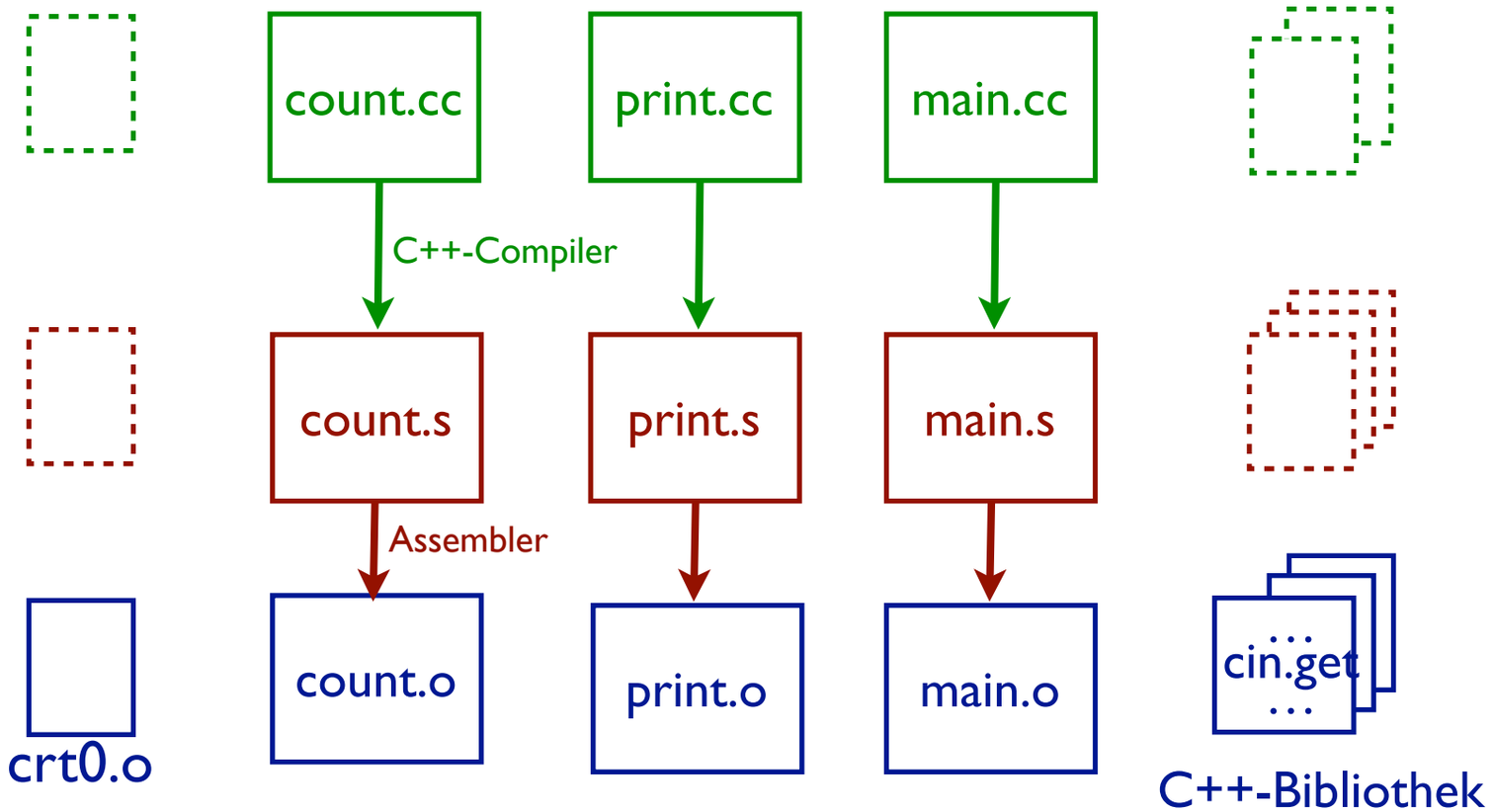


Getrennte Übersetzung von Programmen

- Gemeinsames Übersetzen von riesigen Programmen zu umständlich/zeitaufwendig

  ⇒ Teile einzeln übersetzen (bei Bedarf)

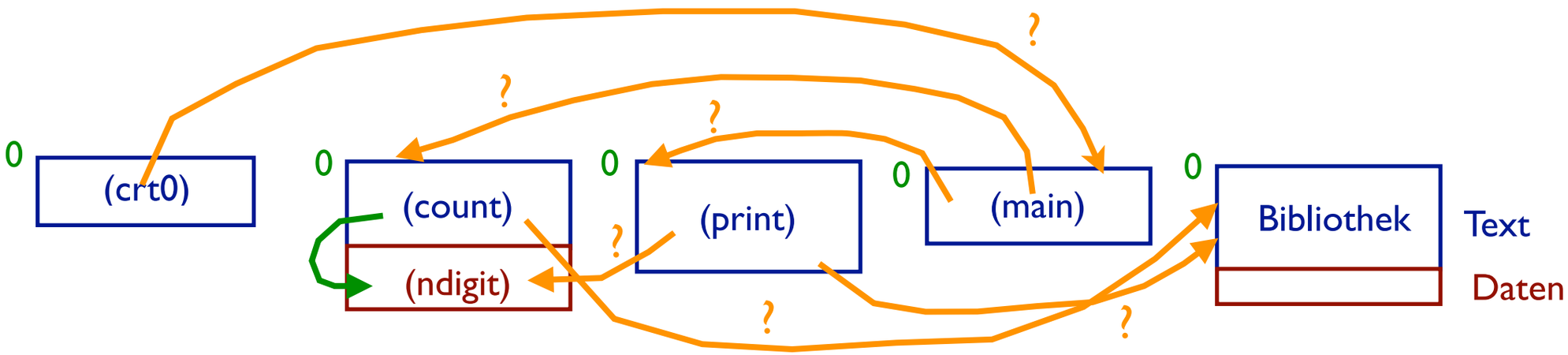
  ⇒ Bibliotheken liegen i.d.R. bereits übersetzt vor



- Jede .o-Datei (Object File) hat eigenen virtuellen Adressraum

  ⇒ Querbezüge nicht erkennbar

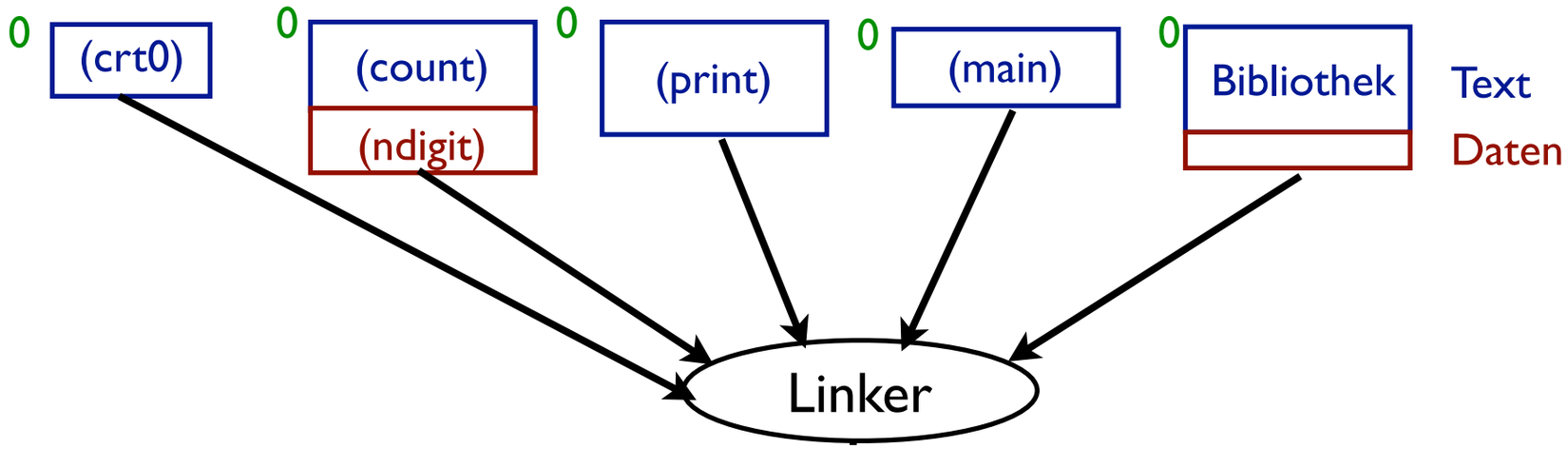
  (z.B. Aufruf von count() in main())



Der Linker (ld)

- Schaffung eines gemeinsamen Adressraums durch Zusammenbinden (Linken) der .o-Dateien (Object Files)

  ⇒ „Executable“ (ausführbares Programm)



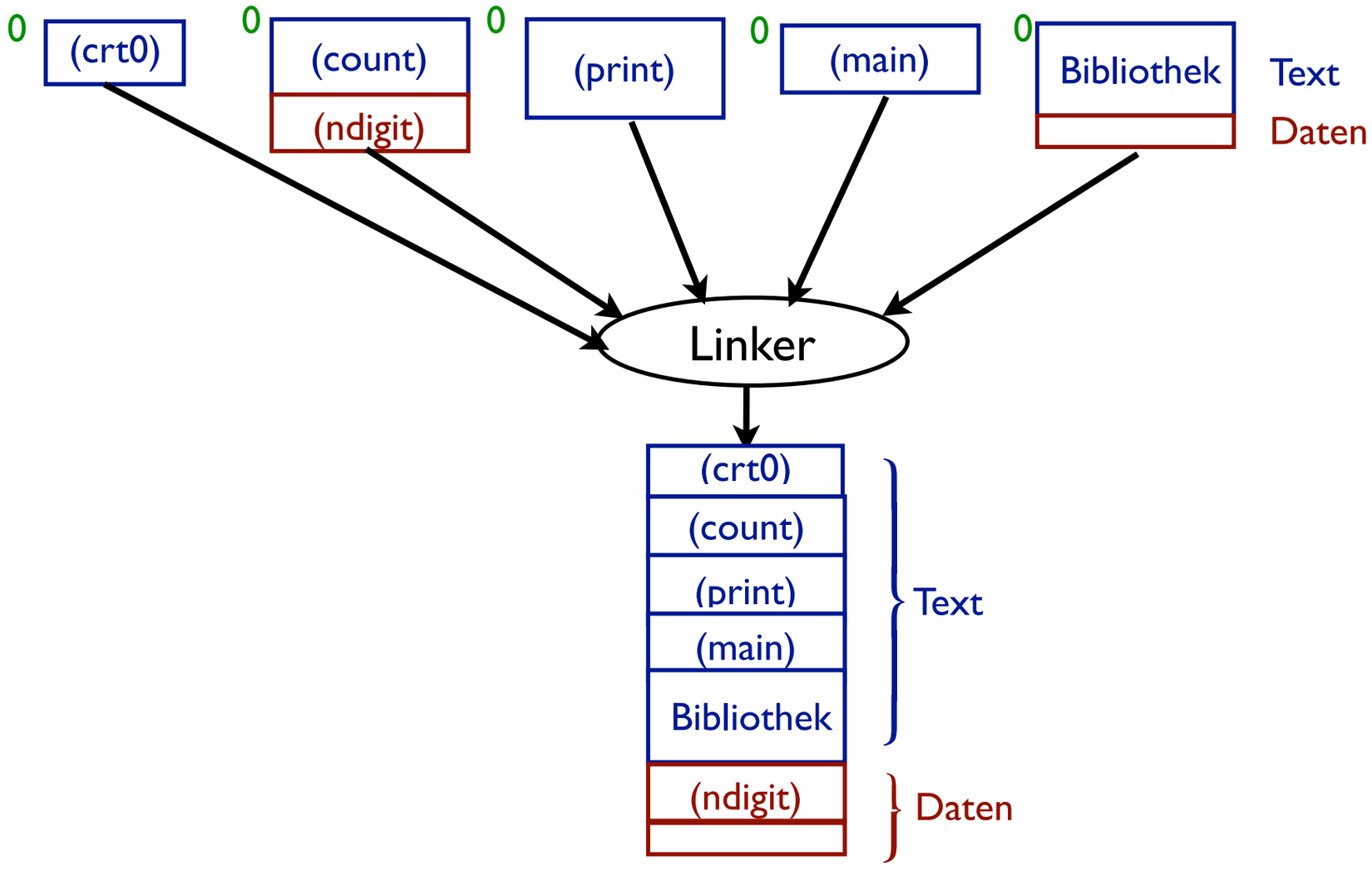
Zur Erinnerung: „C-Compiler“ `cc` ist Shell-Skript:

```
/lib/ccom $1 | as
ld /lib/crt0.o a.out -lc

⇒ cc bla.c
```

Der Linker (ld)

- Schaffung eines gemeinsamen Adressraums durch Zusammenbinden (Linken) der .o-Dateien (Object Files)

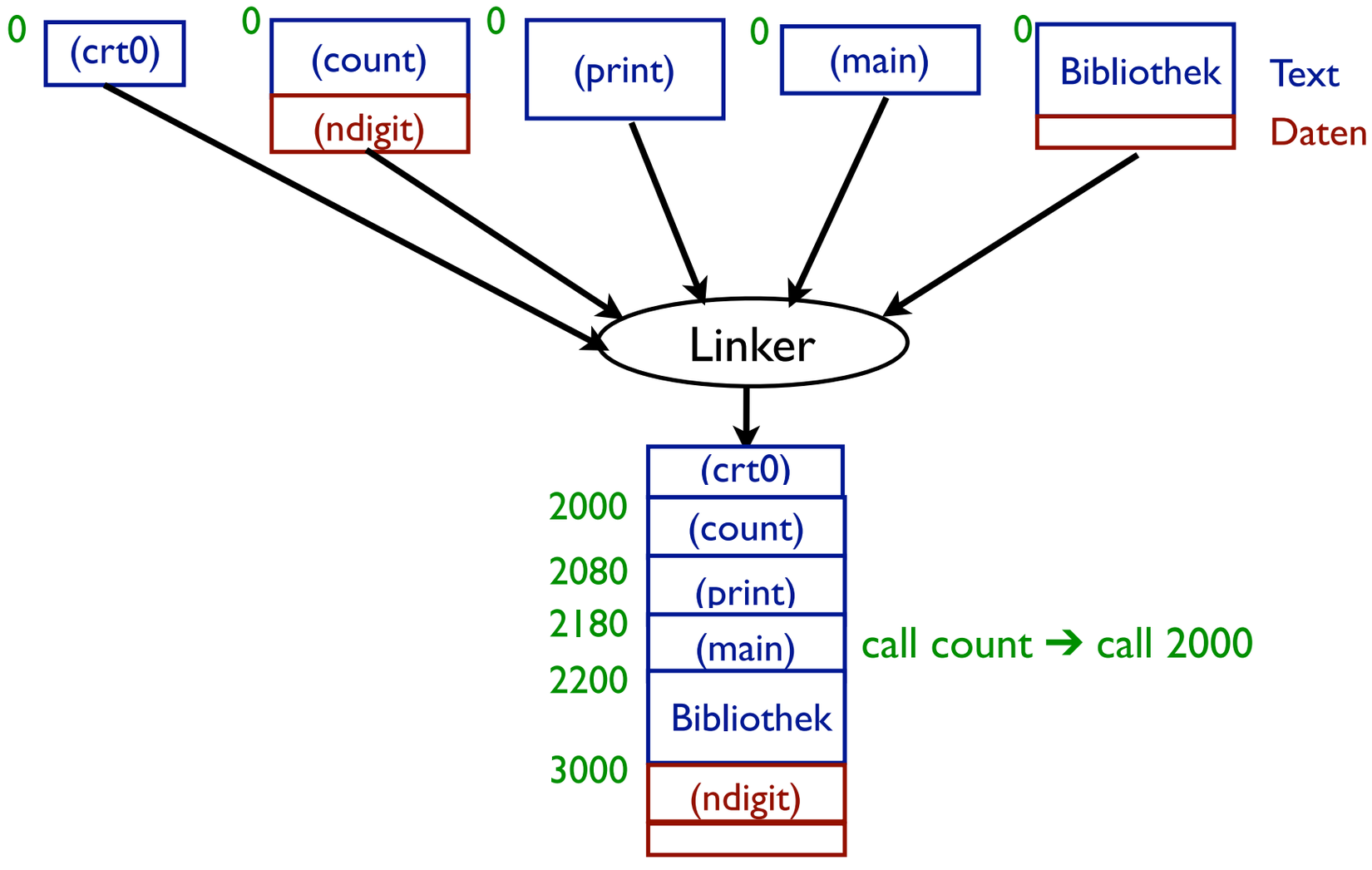
  ⇒ „Executable“ (ausführbares Programm)



Der Linker (ld)

- Schaffung eines gemeinsamen Adressraums durch Zusammenbinden (Linken) der .o-Dateien (Object Files)

  ⇒ „Executable“ (ausführbares Programm)

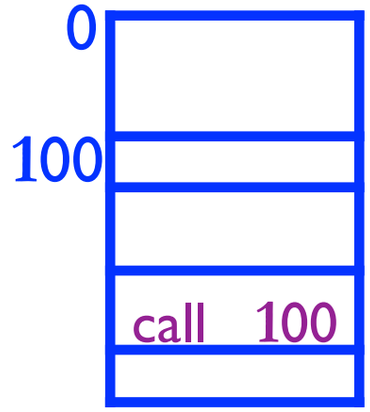


- Querbezüge ausdrückbar durch Angabe der entsprechenden Adressen

Angabe der aufgelösten Adressen

a) „absolute“ Adressen

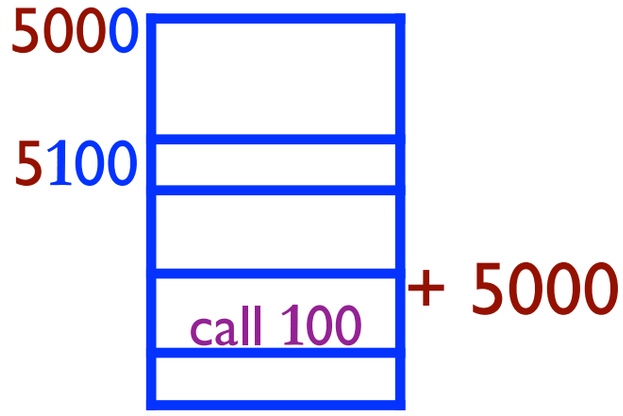
main: ... call print ... ➔ call 100



Angabe der aufgelösten Adressen

a) „absolute“ Adressen

main: ... call print ... → call 100

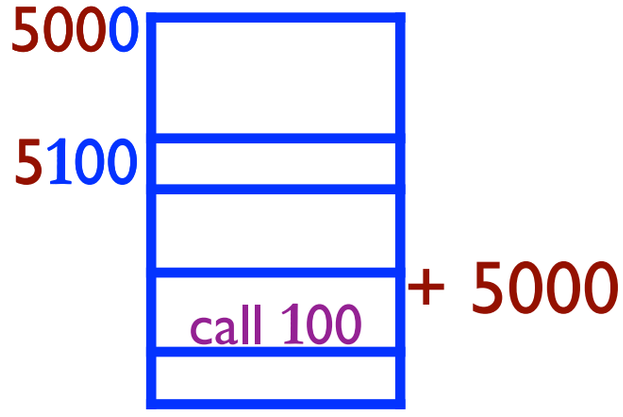


- Problem: Tatsächliche Anfangsadresse beim Linken nicht immer bekannt/ verwendbar
  - (Zusätzlicher Lader: „Offset“ addieren → Relocation)

Angabe der aufgelösten Adressen

a) „absolute“ Adressen

main: ... call print ... ➔ call 100



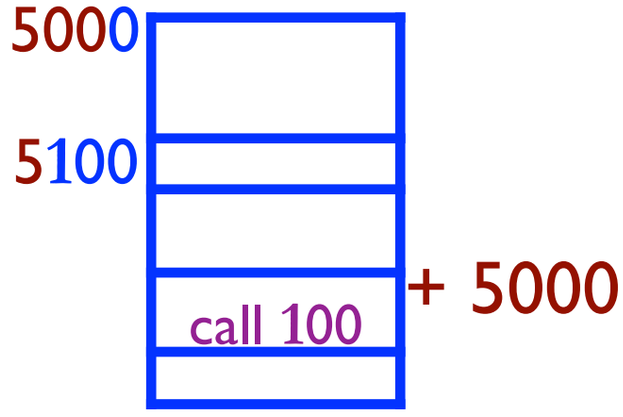
- Problem: Tatsächliche Anfangsadresse beim Linken nicht immer bekannt/ verwendbar
  - (Zusätzlicher Lader: „Offset“ addieren ➔ Relocation)
  - Shared Libraries ➔ gleichzeitig in mehreren Adressräumen benutzen

    ⇒ derselbe Code, u.U. jedoch unterschiedliche Adressen

Angabe der aufgelösten Adressen

a) „absolute“ Adressen

main: ... call print ... ➔ call 100



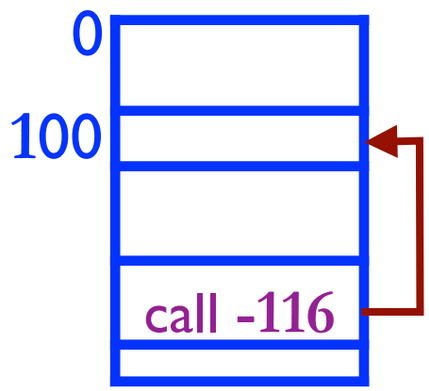
- Problem: Tatsächliche Anfangsadresse beim Linken nicht immer bekannt/ verwendbar
  - (Zusätzlicher Lader: „Offset“ addieren ➔ Relocation)
  - Shared Libraries ➔ gleichzeitig in mehreren Adressräumen benutzen

    ⇒ derselbe Code, u.U. jedoch unterschiedliche Adressen

b) Besser: „PC-relative “ Adressen

- Relativ zum aktuellen Befehlszähler (Position independent code, PIC)

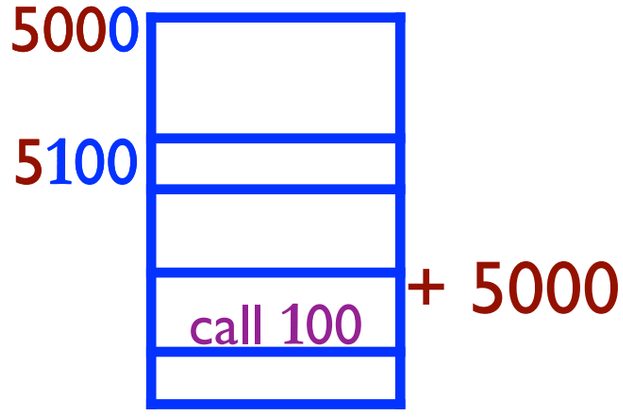
main: ... call print ... ➔ call -116



Angabe der aufgelösten Adressen

a) „absolute“ Adressen

main: ... call print ... ➔ call 100



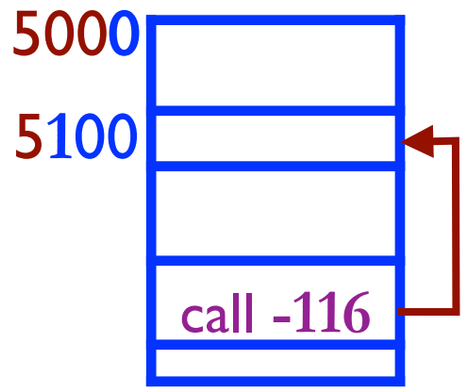
- Problem: Tatsächliche Anfangsadresse beim Linken nicht immer bekannt/ verwendbar
  - (Zusätzlicher Lader: „Offset“ addieren ➔ Relocation)
  - Shared Libraries ➔ gleichzeitig in mehreren Adressräumen benutzen

    ⇒ derselbe Code, u.U. jedoch unterschiedliche Adressen

b) Besser: „PC-relative “ Adressen

- Relativ zum aktuellen Befehlszähler (Position independent code, PIC)

main: ... call print ... ➔ call -116



Fragen – Teil 2

- Welche Aufgabe erfüllt ein *Linker*?
- Welchen Vorteil hat es, Bibliotheken mit *Position Independent Code* zu versehen?

Teil 3:
Symboltabelle und a.out-Format

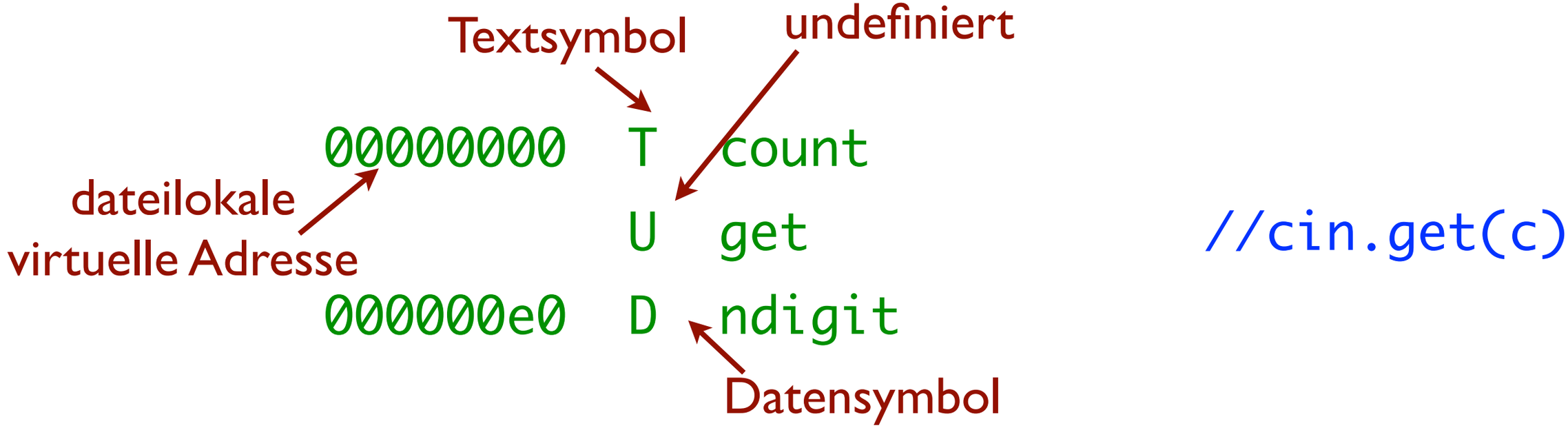
Symboltabelle (vereinfacht)

- Linker benötigt Informationen über aufzulösende Referenzen (vom Compiler)
  - Zuordnung von definierten Symbolen zu Adressen
  - Auflistung undefinierter Symbole
- Beispiel:

```
00000000  T  count
          U  get                    //cin.get(c)
000000e0  D  ndigit
```

Symboltabelle (vereinfacht)

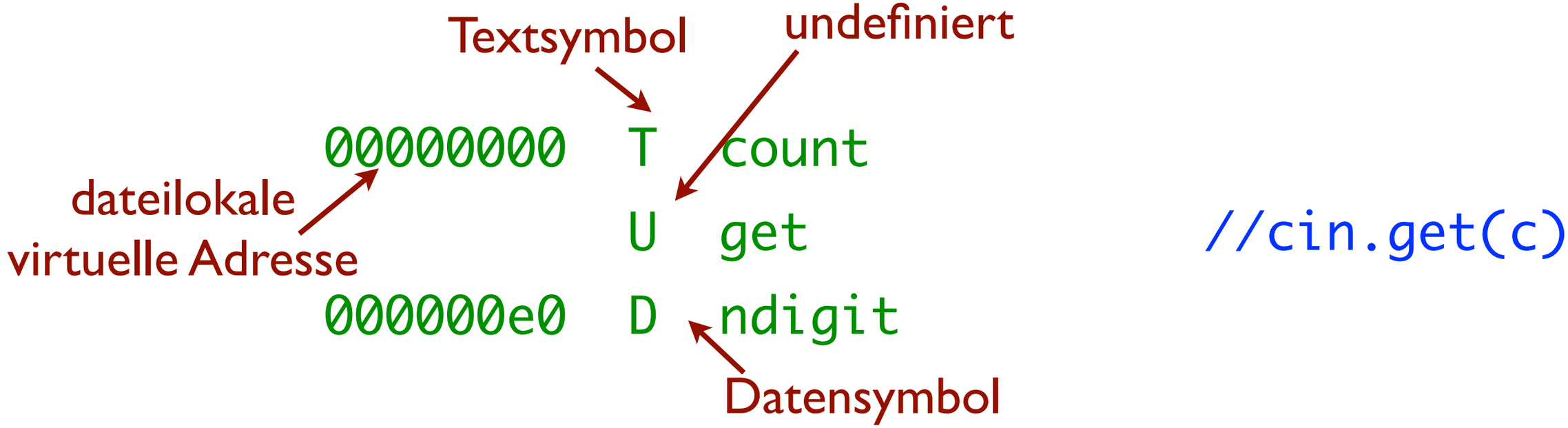
- Linker benötigt Informationen über aufzulösende Referenzen (vom Compiler)
  - Zuordnung von definierten Symbolen zu Adressen
  - Auflistung undefinierter Symbole
- Beispiel:



Symboltabelle (vereinfacht)

- Linker benötigt Informationen über aufzulösende Referenzen (vom Compiler)
  - Zuordnung von definierten Symbolen zu Adressen
  - Auflistung undefinierter Symbole

- Beispiel:



- Problem: C++ lässt überladene Funktionen zu

  ⇒ Funktionen mit unterschiedlichen Parametern, aber gleichen Namen

  ⇒ muss in Symboltabelle unterschieden werden

  ⇒ „Name Mangling“

Name Mangling (vereinfacht)

Erweitern der Symbolnamen durch Compiler

- Unterscheidung globale Funktion (F)
  vs. klassenspezifische Funktion (Länge Klassenname, z.B. 7istream)

```
count__F
get__7istream     //cin.get
```

Name Mangling (vereinfacht)

Erweitern der Symbolnamen durch Compiler

- Unterscheidung globale Funktion (F)
  vs. klassenspezifische Funktion (Länge Klassenname, z.B. 7istream)
- Angabe der Parametertypen, z.B. ii (zwei Integer)

```
power__Fii
power__Fdd
count__Fv
get__7istreamRc   //cin.get(c)
```

Name Mangling (vereinfacht)

Erweitern der Symbolnamen durch Compiler

- Unterscheidung globale Funktion (F)
  vs. klassenspezifische Funktion (Länge Klassenname, z.B. 7istream)
- Angabe der Parametertypen, z.B. ii (zwei Integer)

```
power__Fii
power__Fdd
count__Fv
get__7istreamRc   //cin.get(c)
```

- Notation für Operatoren (__op)

```
__pl__7complexd      +
__ls__7ostreami      <<
__ls__7ostreamc      <<
```

- Kein Name Mangling für Datenobjekte nötig

Symboltabelle (vereinfacht)

- Linker benötigt Informationen über aufzulösende Referenzen (vom Compiler)
  - Zuordnung von definierten Symbolen zu Adressen
  - Auflistung undefinierter Symbole
- Beispiel:

```
00000000  T  count__Fv
          U  get__7istreamRc   //cin.get(c)
000000e0  D  ndigit
```

- Problem: C++ lässt überladene Funktionen zu

  ⇒ Funktionen mit unterschiedlichen Parametern, aber gleichen Namen

  ⇒ muss in Symboltabelle unterschieden werden

  ⇒ „Name Mangling“

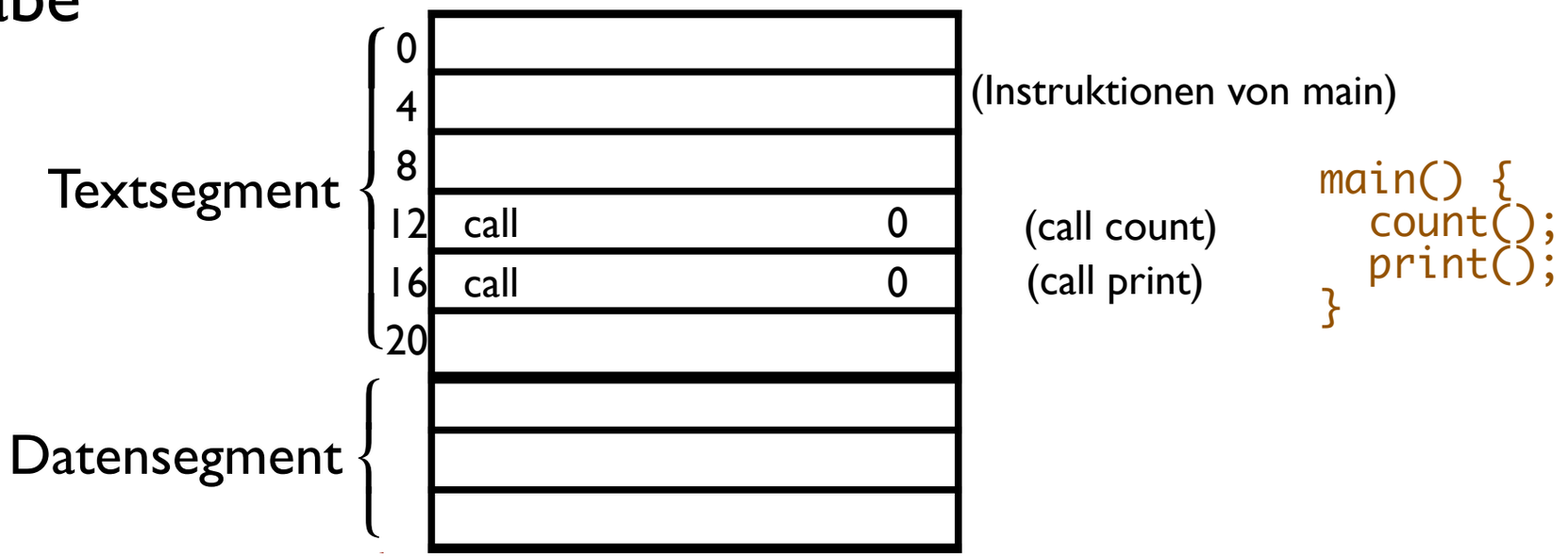
Das a.out-Format

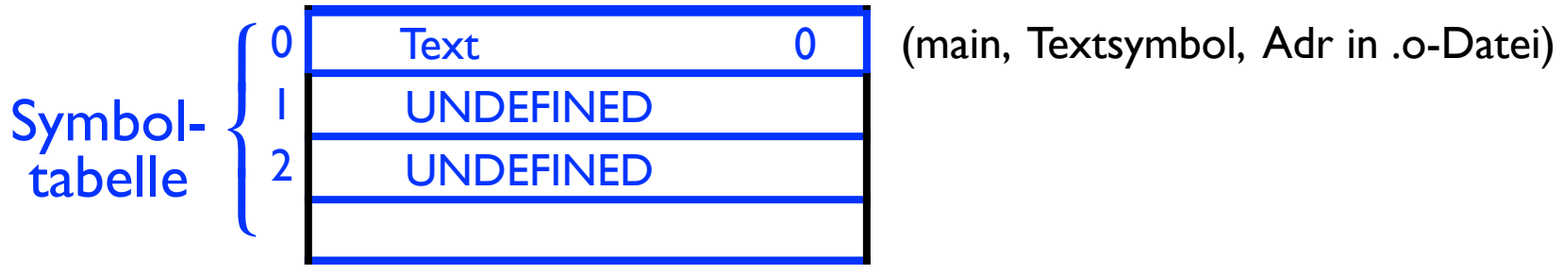
(Unix, Beispiel SunOS4)

Beispiel: main.o

- Object Files unterstützen Linker bei seiner Aufgabe

  ⇒ spezielles Format (a.out)





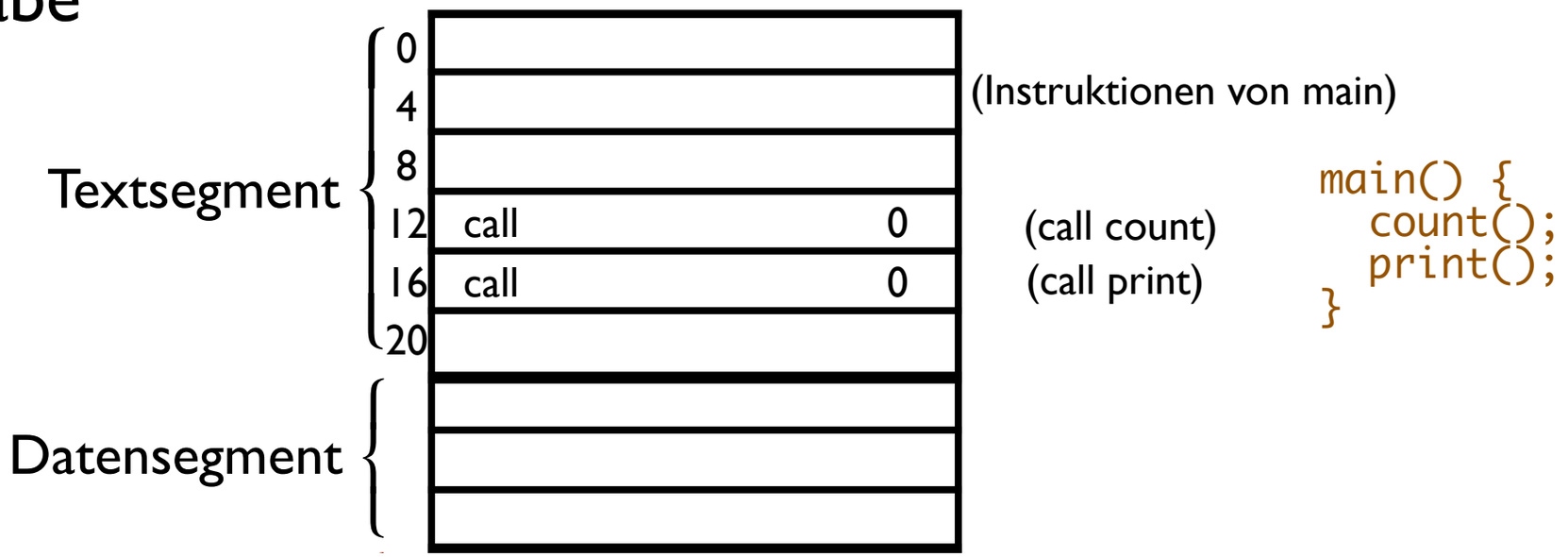
Das a.out-Format

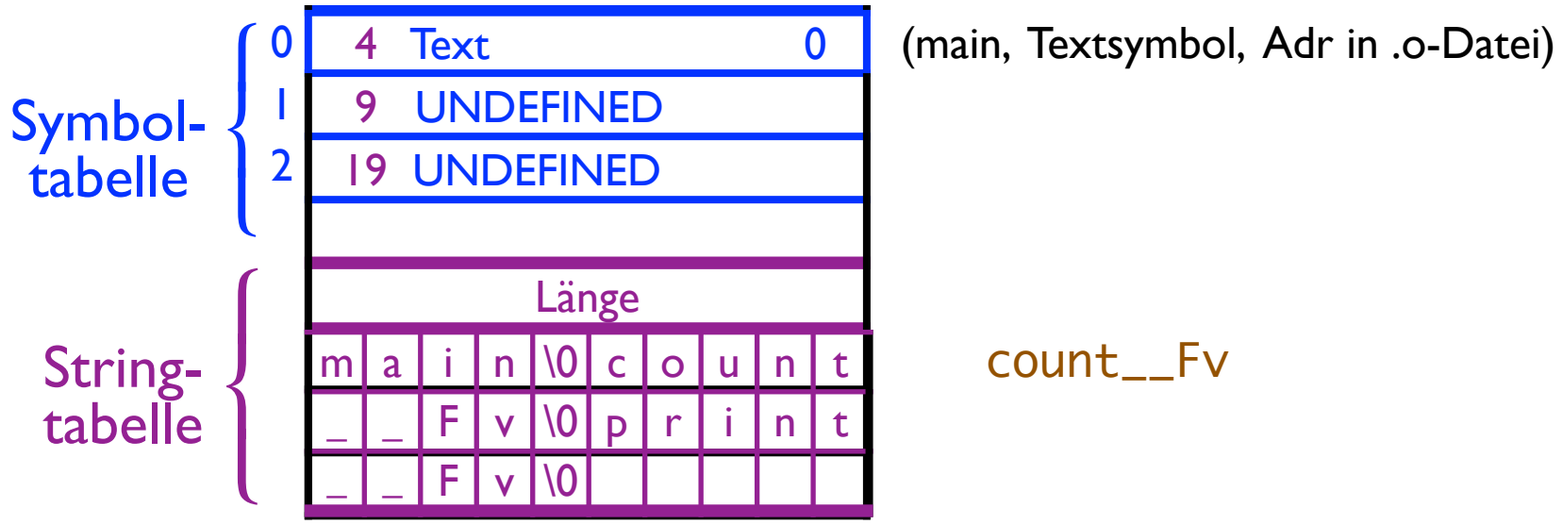
(Unix, Beispiel SunOS4)

Beispiel: main.o

- Object Files unterstützen Linker bei seiner Aufgabe

  ⇒ spezielles Format (a.out)





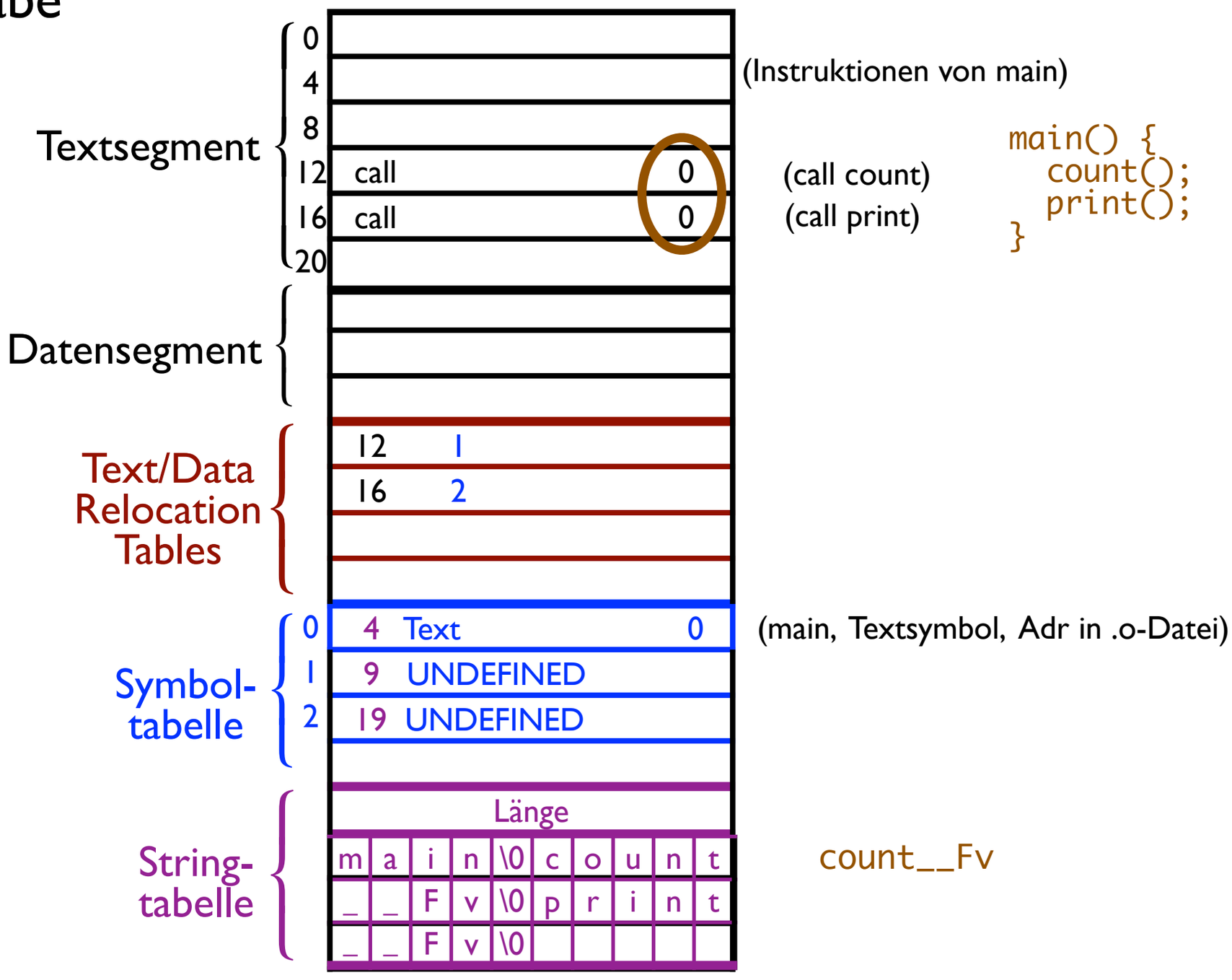
Das a.out-Format

(Unix, Beispiel SunOS4)

Beispiel: main.o

- Object Files unterstützen Linker bei seiner Aufgabe

  ⇒ spezielles Format (a.out)



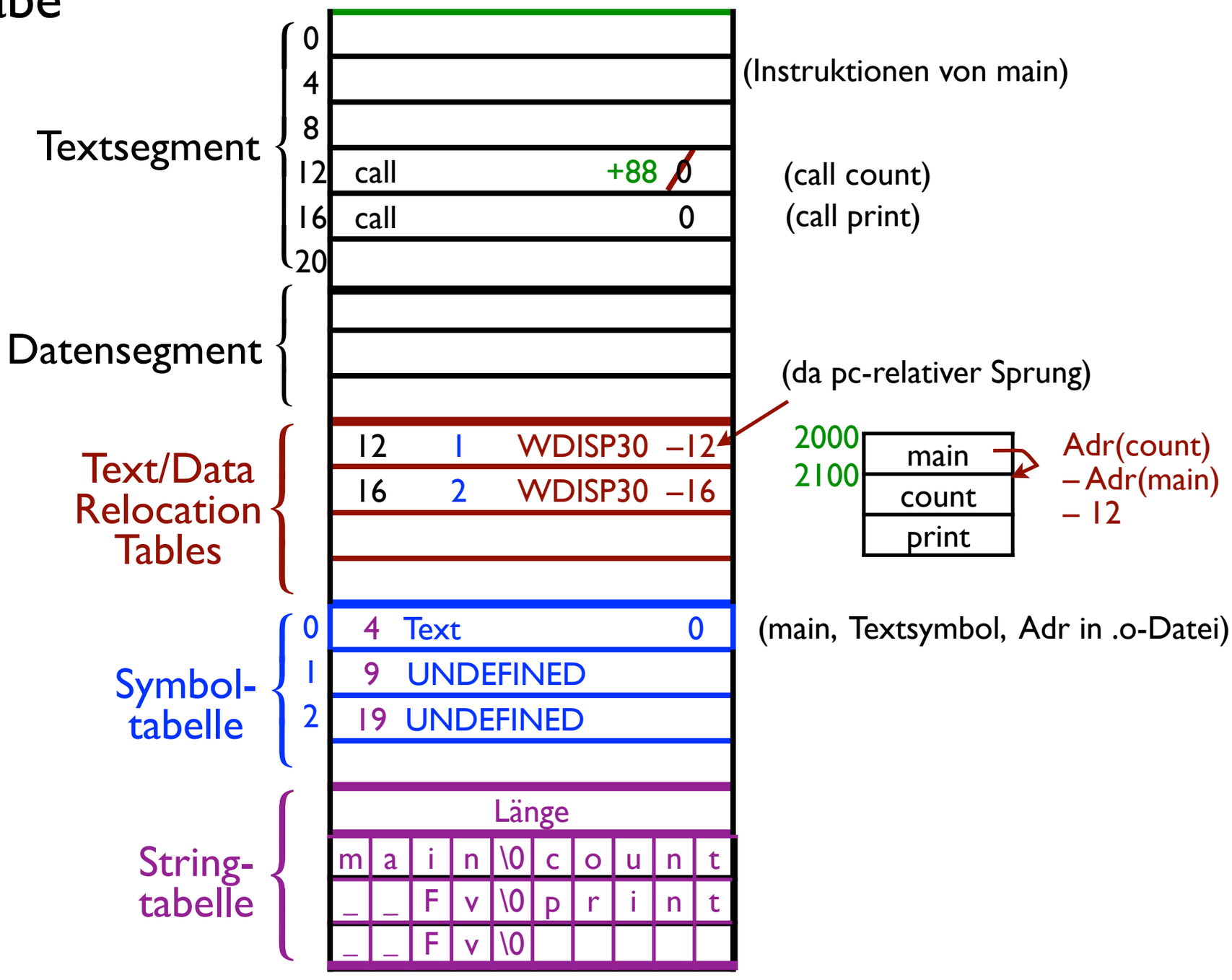
Das a.out-Format

(Unix, Beispiel SunOS4)

Beispiel: main.o

- Object Files unterstützen Linker bei seiner Aufgabe

  ⇒ spezielles Format (a.out)

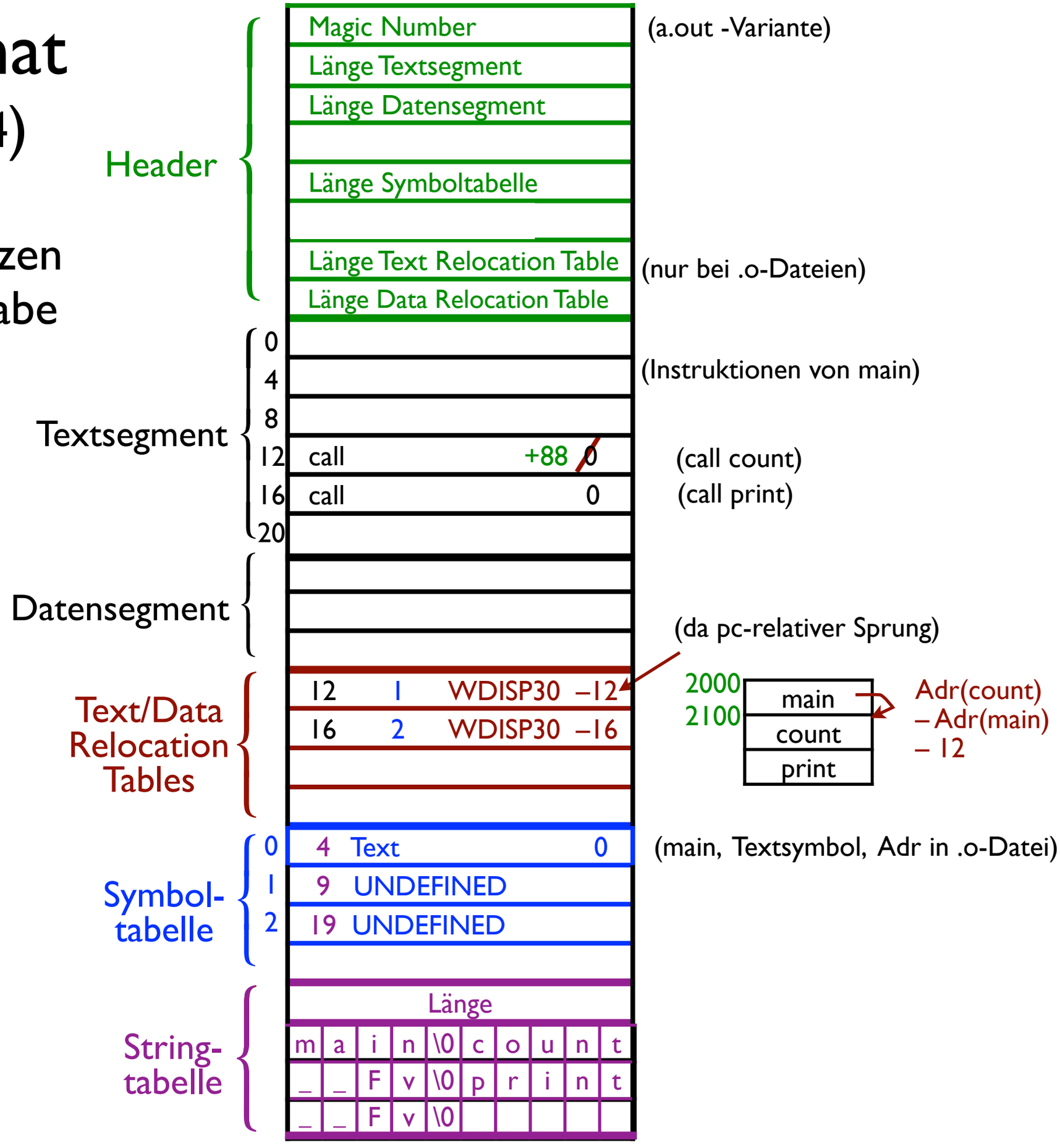


Das a.out-Format

(Unix, Beispiel SunOS4)

- Object Files unterstützen Linker bei seiner Aufgabe

  ⇒ spezielles Format (a.out)

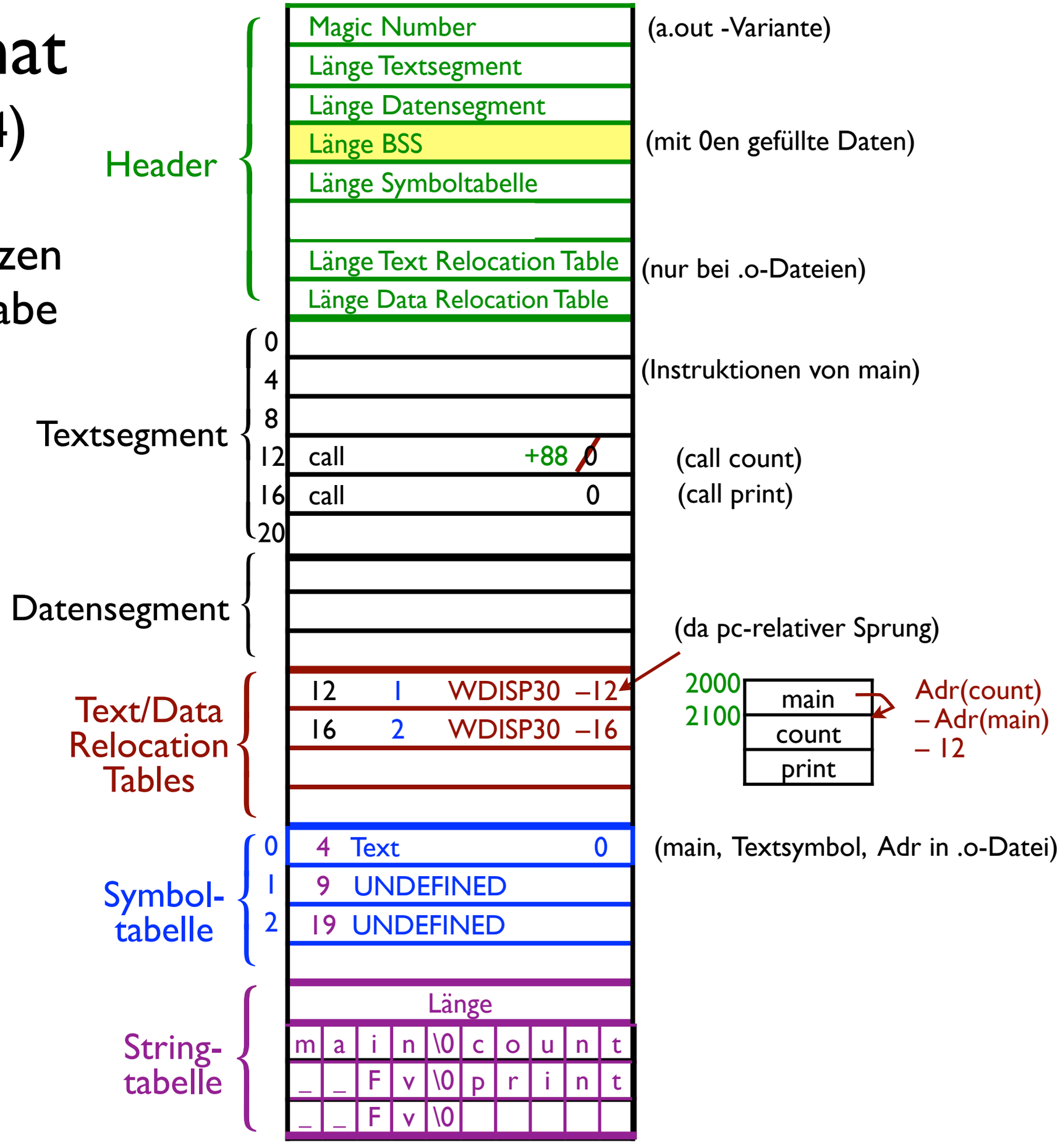


Das a.out-Format

(Unix, Beispiel SunOS4)

- Object Files unterstützen Linker bei seiner Aufgabe

  ⇒ spezielles Format (a.out)

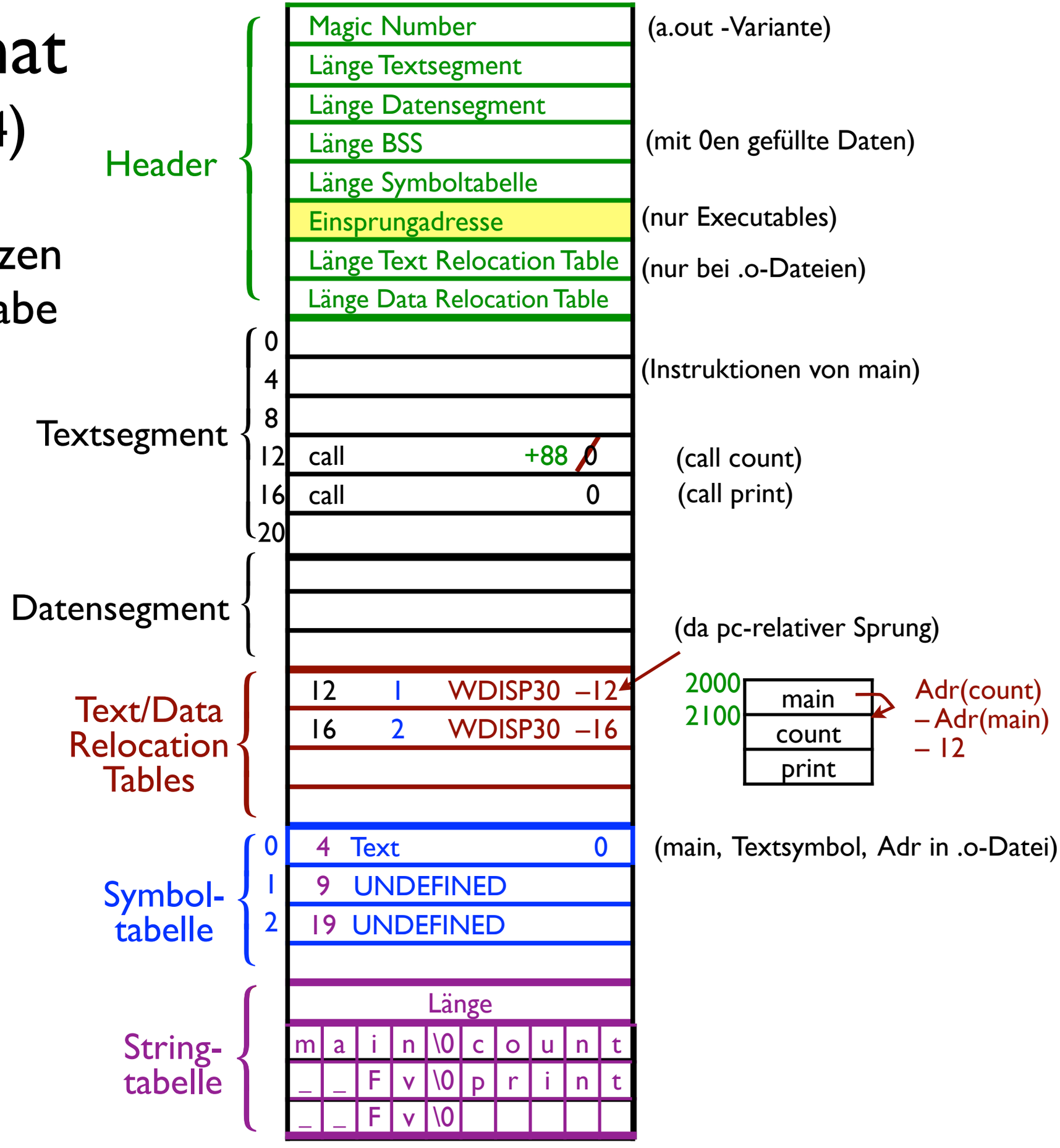


Das a.out-Format

(Unix, Beispiel SunOS4)

- Object Files unterstützen Linker bei seiner Aufgabe

  ⇒ spezielles Format (a.out)



Erläuterungen: a.out-Format von main.o

- Textsegment
  - enthält Sprünge, die noch nicht ausgefüllt werden können, da Adresse von count()/print() nicht bekannt (Platzhalter: 0)
- Stringtabelle:
  - enthält alle „sichtbaren“ Symbole (Namen von exportierten/importierten Prozeduren bzw. Datenstrukturen)
  - jeweils durch Nullbytes abgeschlossen
  - ermöglicht feste Länge von Symboltabelleneinträgen trotz variabler Stringlänge
- Symboltabelle:
  - enthält Eintrag für jedes „sichtbare“ Symbol, bestehend aus:
  - Name des Symbols durch Verweis auf Startbyte in Stringtabelle
  - Typ (+Adresse):
    - Text (Im Textsegment an angegebener Adresse definiert; hier: 0)
    - Data (Im Datensegment an angegebener Adresse definiert)
    - UNDEFINED (Extern definiert, muss noch dazu gebunden werden; hier: count/print)

- Text/Data Relocation Tabelle:

  enthält Relocation-Information für den Linker

  - Eintrag im Text-/Daten-Segment, der noch mit richtiger Adresse versehen werden muss
    (hier: Instruktionen an Adressen 12 und 16 ⇒ call count/print)

  - Angabe des zu ersetzenden Symbols über Verweis auf entsprechenden Symboltabelleneintrag (hier: count ⇒ 1, print ⇒ 2)

  - Angabe eines „Modus“, wie nachfolgende Adressinfo umgesetzt werden soll
    (hier: „WDISP30“ ⇒ spezielle Form der pc-relativen Adressierung)

  - Angabe eines „Adressmodifizierers“
    (hier: pc befindet sich auf Adresse 12 bzw. 16, wenn Sprung erfolgt)

    ⇒ Adresse = Adr(count) – Adr(main) – 12

    (Bei Instruktion „call count“ einzutragende pc-relative Sprungadresse zu count ⇒ muss der Linker ausrechnen)

Fragen – Teil 3

1. Wozu wird beim Assemblieren eine *Symboltabelle* angelegt?

Zusammenfassung

- Übersetzung von Programmen
- Virtueller Adressraum von Prozessen
- Bibliotheken
- Der Linker
- Symboltabelle, a.out-Format

Vom Quellcode zum Programm in Ausführung – Fragen

1. In welche Bereiche (*Segmente*) ist der *(virtuelle) Adressraum* eines Programms in Ausführung in Unix unterteilt, und welche Eigenschaften kennzeichnen sie?
2. Wozu wird der *Stack* verwendet?
3. Welchem Zweck dienen Bibliotheken (*Libraries*)?
4. Welche Aufgabe erfüllt ein *Linker*?
5. Welchen Vorteil hat es, Bibliotheken mit *Position Independent Code* zu versehen?
6. Wozu wird beim Assemblieren eine *Symboltabelle* angelegt?