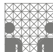
64-040 Modul IP7: Rechnerstrukturen

12 Assemblerprogrammierung (2)
Datenstrukturen und Speicherverwaltung

Norman Hendrich

Universität Hamburg
MIN Fakultät, Department Informatik
Vogt-Kölln-Str. 30, D-22527 Hamburg
hendrich@informatik.uni-hamburg.de

WS 2013/2014



Inhalt

1. Assembler: Speicherverwaltung
   - Elementare Datentypen
   - Arrays
   - Strukturen
   - Objektorientierte Konzepte
   - Linker und Loader
   - Dynamische Speicherverwaltung
   - Puffer-Überläufe

Elementare Datentypen

- Ganzzahl (Integer)
  - wird in allgemeinen Registern gespeichert
  - abhängig von den Anweisungen: *signed/unsigned*
  - Intel, gas, Bytes, C:

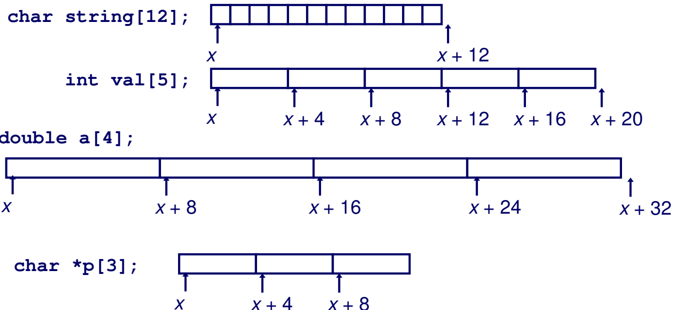
Intel	gas	Bytes	C
byte	b	1	[unsigned] char
word	w	2	[unsigned] short
double word	l	4	[unsigned] int

- Gleitkomma (Floating Point)
  - wird in Gleitkomma-Registern gespeichert
  - Intel, gas, Bytes, C:

Intel	gas	Bytes	C
Single	s	4	float
Double	l	8	double
Extended	t	10/12	long double

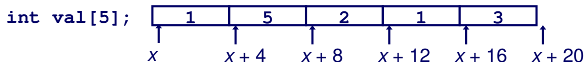
Array: Allokation / Speicherung

- `T A[N];`
  - Array `A` mit Daten von Typ `T` und `N` Elementen
  - fortlaufender Speicherbereich von `N`×sizeof(`T`) Bytes



Array: Zugriffskonvention

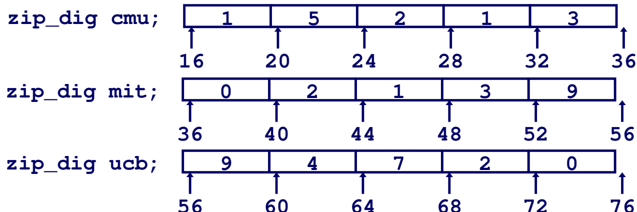
- `T A[N];`
  - Array `A` mit Daten von Typ `T` und `N` Elementen
  - Bezeichner `A` zeigt auf erstes Element des Arrays: Element 0



Reference	Type	Value
`val[4]`	`int`	3
`val`	`int *`	x
`val+1`	`int *`	$x + 4$
`&val[2]`	`int *`	$x + 8$
`val[5]`	`int`	??
`*(val+1)`	`int`	5
`val + i`	`int *`	$x + 4\,i$

Beispiel: einfacher Arrayzugriff

```
typedef int zip_dig[5];

zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
```



Beispiel: einfacher Arrayzugriff (cont.)

- Register `%edx` : Array Startadresse
  `%eax` : Array Index
- Adressieren von 4×`%eax`+`%edx`

⇒ Speicheradresse `(%edx,%eax,4)`

```
int get_digit
  (zip_dig z, int dig)
{
  return z[dig];
}
```

Memory Reference Code

```
  # %edx = z
  # %eax = dig
  movl (%edx,%eax,4),%eax # z[dig]
```

- keine Bereichsüberprüfung („*bounds checking*“)
- Verhalten außerhalb des Indexbereichs ist Implementierungsabhängig

Beispiel: Arrayzugriff mit Schleife

- Originalcode

```
int zd2int(zip_dig z)
{
  int i;
  int zi = 0;
  for (i = 0; i < 5; i++) {
    zi = 10 * zi + z[i];
  }
  return zi;
}
```

- transformierte Version: gcc
  - Laufvariable `i` eliminiert
  - aus Array-Code wird Pointer-Code
  - in „do-while“ Form
  - Test bei Schleifeneintritt unnötig

```
int zd2int(zip_dig z)
{
  int zi = 0;
  int *zend = z + 4;
  do {
    zi = 10 * zi + *z;
    z++;
  } while(z <= zend);
  return zi;
}
```

Beispiel: Arrayzugriff mit Schleife (cont.)

- Register `%ecx : z`
  `%eax : zi`
  `%ebx : zend`
- `*z + 2*(zi+4*zi)`
  ersetzt `10*zi + *z`
- `z++` Inkrement: $+4$

```
int zd2int(zip_dig z)
{
  int zi = 0;
  int *zend = z + 4;
  do {
    zi = 10 * zi + *z;
    z++;
  } while(z <= zend);
  return zi;
}
```

```
  # %ecx = z
  xorl %eax,%eax             # zi = 0
  leal 16(%ecx),%ebx         # zend  = z+4
.L59:
  leal (%eax,%eax,4),%edx    # 5*zi
  movl (%ecx),%eax           # *z
  addl $4,%ecx               # z++
  leal (%eax,%edx,2),%eax    # zi = *z + 2*(5*zi)
  cmpl %ebx,%ecx             # z : zend
  jle .L59                   # if <= goto loop
```

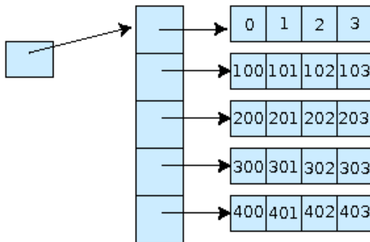
Zwei- und mehrdimensionale Arrays

$(N \times M)$ Matrizen? drei grundsätzliche Möglichkeiten:

- Array von Pointern auf Zeilen-Arrays von Elementen — Java
  - sehr flexibel, auch für nicht-rechteckige Layouts
  - Sharing/Aliasing von Zeilen möglich
- Array von $N \times M$ Elementen und passende Adressierung
  - row-major Anordnung — C, C++
  - column-major Anordnung — Matlab, FORTRAN
- bei Verwendung/Mischung von Bibliotheksfunktionen aus anderen Sprachen unbedingt berücksichtigen

Java: Array von Pointern auf Arrays von Elementen

```
class MatrixDemo {
  int matrix[][]; // matrix[i]->

  public MatrixDemo( int NROWS, int NCOLS ) {
    matrix = new int[NROWS][NCOLS];
    for( int r=0; r < matrix.length; r++ ) {
      for( int c =0; c < matrix[r].length; c++ ) {
        matrix[r][c] = 100*r + c;
      }
    }
    // int[] row0 = matrix[0];
    // int   m23 = matrix[2][3];
  }
  public int get( int r, int c ) {
    return matrix[r][c];
  }
}
```



Zweidimensionale Arrays in C

```
int n_rows = 4; int n_cols = 5;
int matrix[4][5];  // 00 01 02 03 04  10 11 12 13 14  20 .. 34
int schach[8][8] = { 0,1,2,3,4,5,6,7, 10,11,12,13,... , 20, 21, 77 };

int m00 = matrix[0][0]; // *(matrix[0] + 0);
int m01 = matrix[0][1]; // *(matrix[0] + 1);
int m20 = matrix[2][0]; // *(matrix[2] + 0);
int m34 = matrix[1][4]; // *(matrix[3] + 4);

int *elem = &(matrix[2][2]);
elem++;         // nächste Spalte (bzw. Wraparound);
elem+= n_cols; // nächste Zeile
```

- die sogenannte „row-major“ Anordnung, Spalten fortlaufend
- „column-major“ ist transponiert: 00 10 20 ... 01 11 21 ... 34

Mehrdimensionale Arrays: entsprechend

- d-dimensionales $N_1 \times N_2 \times \cdots \times N_d$ Array
  - Element addressiert mit Tupel $(n_1, n_2, \ldots, n_d)$, mit d (zero-offset) Indizes $n_k \in [0, N - K - 1]$
- row-major Anordnung: letzte Dimension ist fortlaufend

$$n_d + N_d \cdot (n_{d-1} + N_{d-1} \cdot (n_{d-2} + N_{d-2} \cdot ()))) = \sum_{k=1}^{d} \left(\prod_{\ell=k+1}^{d} N_\ell \right) n_k$$

- column-major Anordnung: erste Dimension is fortlaufend im Speicher

$$n_1 + N_1 \cdot (n_2 + N_2 \cdot (n_3 + N_3 \cdot (\cdots + N_{d-1} n_d) \cdots))) = \sum_{k=1}^{d} \left(\prod_{\ell=1}^{k-1} N_\ell \right) n_k$$

- oder Arrays von Arrays von Arrays auf Arrays auf Elemente

Strukturen (*Records*)

- Allokation eines zusammenhängenden Speicherbereichs
- Elemente der Struktur über Bezeichner referenziert
- verschiedene Typen der Elemente sind möglich

```
struct rec {
  int i;
  int a[3];
  int *p;
};
```

Memory Layout

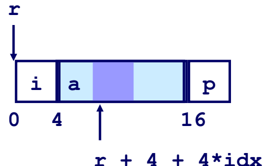
```
void
set_i(struct rec *r,
      int val)
{
  r->i = val;
}
```

Assembly

```
# %eax = val
# %edx = r
movl %eax,(%edx)    # Mem[r] = val
```

Strukturen: Zugriffskonventionen

- Zeiger auf Byte-Array für Zugriff auf Struktur(element) `r`
- Compiler bestimmt Offset für jedes Element




```
struct rec {
  int i;
  int a[3];
  int *p;
};
```

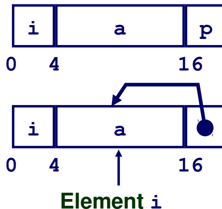
```
int *
find_a
 (struct rec *r, int idx)
{
  return &r->a[idx];
}
```

```
# %ecx = idx
# %edx = r
leal 0(,%ecx,4),%eax    # 4*idx
leal 4(%eax,%edx),%eax # r+4*idx+4
```

Beispiel: Strukturreferenzierung

```
struct rec {
  int i;
  int a[3];
  int *p;
};
```

```
void
set_p(struct rec *r)
{
  r->p =
   &r->a[r->i];
}
```




```
 # %edx = r
movl  (%edx),%ecx         # r->i
leal  0(,%ecx,4),%eax     # 4*(r->i)
leal  4(%edx,%eax),%eax   # r+4+4*(r->i)
movl  %eax,16(%edx)       # Update r->p
```

Alignment: Ausrichtung der Datenstrukturen

- Datenstrukturen an Wortgrenzen ausrichten
  double- / quad-word
- sonst Problem
  - ineffizienter Zugriff über Wortgrenzen hinweg
  - virtueller Speicher und Caching

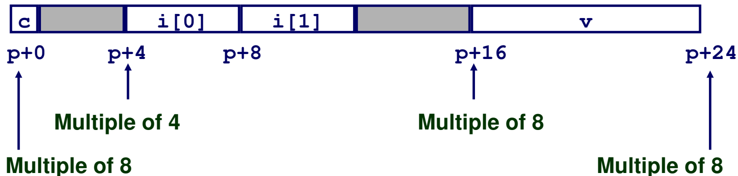
⇒ Compiler erzeugt „Lücken“ zur richtigen Ausrichtung

- typisches Alignment (IA32)

Länge	Typ		Windows	Linux
1 Byte	`char`	keine speziellen Verfahren		
2 Byte	`short`	Adressbits:	...0	...0
4 Byte	`int, float, char *`	Adressbits:	...00	...00
8 Byte	`double`	Adressbits:	...000	...00
12 Byte	`long double`	Adressbits:	n.a.	...00

Beispiel: Structure Alignment

```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```



Umsetzung von Objektorientierung?

- Klassen/Objekte verbinden Daten und Methoden
  - polymorphe Funktionen — name-mangling
  - Metadaten, run-time type-information — rtti
  - Vererbung und dynamischer Funktionsaufruf — vtable
- Grundidee
  - Datenstrukturen wie in Assembler/C
  - Schema zur Erzeugung eindeutiger Namen
  - zusätzliche Pointer auf Typ/Klassen-Information
  - zusätzliche Pointer auf Funktionstabelle(n)
  - Methodenaufrufe bekommen `this`-Pointer als Argument

  - gute Performance erfordert effiziente Implementierung
  - Details normalerweise vor dem Programmierer verborgen

Objekte: Exemplare von Klassen

- kombinieren Daten mit den zugehörigen Methoden

- Datenelemente wie C/Assembler Strukturen angeordnet
- ein Pointer auf die rtti-Datenstruktur
  - Debug-Infos: Name der Klasse, Datenelemente
  - Pointer auf Basisklasse(n)
  - Interfaces und Vererbungsinformation
- ein Pointer auf die vtable-Tabelle
  - Array mit allen Methoden der Klasse
  - Name-Mangling erhält Typ-Infos der Parameter
- aus Effizienzgründen diese Pointer ggf. mit negativem Offset
  - Speicherverwaltung berücksichtigt dies

Polymorphe Funktionen: Name-Mangling

- Programmierer arbeitet mit Klassen und deren Methoden
- polymorphe Funktionen, abhängig vom Typ der Parameter

```
class polymorph { public:
  float f( int i )   { return 2.0f*i; }
  float f( float f ) { return 1.5f*f; } ...
}
```

- aber: Assembler und Linker erwarten globale Funktionen

- Name-Mangling („name decoration“) im Compiler
  - Funktionsname gebildet aus Prefix + Name + Typkennung
  - Prefix bildet Klassennamen/Namespace ab
  - Typkennung zur eindeutigen Unterscheidung der Argumente
    `_ZN9polymorph1fEi _ZN9polymorph1fEf`
  - Java: siehe Java Native Interface und `javah`-Tool

Methodenaufruf: this-Pointer

- bisher: Funktionen/Code vollkommen separat von Daten
- woher weiss eine Methode, zu welchem Objekt sie gehört?
- wie kommt eine Methode an Exemplarvariablen heran?

- Trick: Compiler übergibt `this` als erstes Argument
  - implizit, muss normalerweise nicht geschrieben werden
  - Pointer auf das aktuelle Objekt
  - Referenz auf Daten über `this->x`
  - Referenz auf Methoden über `this->vtable[offset]`
  - zusätzliche Funktionsparameter anschließend wie gewohnt
  - `Point3D.f( int i, int j )` wird intern zu
    `Point3D.f( Point3D *this, int i, int j )`

Methodenaufruf: this-Pointer

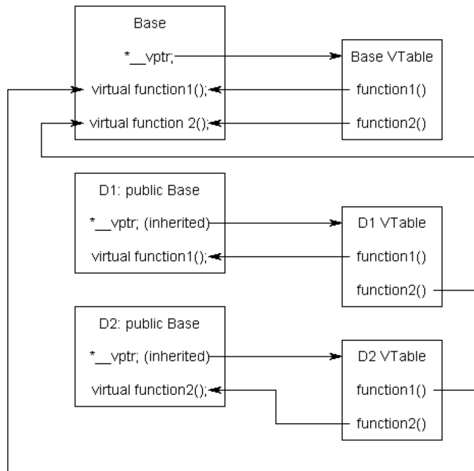
```
#include <stdio.h>

class Point3D {
  private: int x; int y; int z;
  public:
    Point3D( int _x, int _y, int _z ) { x = _x; y = _y; z = _z; }
    int getX() { return x; }
};

int main( int argc, char** argv ) {
  Point3D p( 42, 2, 3 );
  printf( "%d\n", p.getX() );
}
```

```
0000000000400572 <_ZN7Point3D4getXEv>:
  400572: 55                        push   %rbp
  400573: 48 89 e5                  mov    %rsp,%rbp
  400576: 48 89 7d f8               mov    %rdi,-0x8(%rbp)
  40057a: 48 8b 45 f8               mov    -0x8(%rbp),%rax
  40057e: 8b 00                     mov    (%rax),%eax
  400580: 5d                        pop    %rbp
  400581: c3                        retq
```

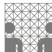
Virtual Table: Dynamischer Methodenaufruf

- Compiler kennt und sammelt alle Methoden einer Klasse
  - inklusive aller Methoden der Basisklassen
- erzeugt vtable Array mit Pointer auf die Funktionen
  - Aufruf der Funktionen als `*((this->vtable)+offset)()` wobei der Offset die jeweilige Methode auswählt
    - wieder `this`-Pointer als erster Parameter
    - weitere Parameter anschließend auf dem Stack
    - ein zusätzlicher Speicherzugriff gegenüber direktem Aufruf der Funktion
  - vererbte Methoden zeigen auf Code der Basisklasse
  - überschriebene Methoden zeigen auf Code der Unterklasse
  - `super.f()` durch Zugriff auf vtable der Basisklasse

Virtual Table: Vererbung



learncpp.com: 12.5 the virtual table



Zusammenfassung: Datentypen

- Arrays
  - fortlaufend zugeteilter Speicher
  - Adressverweis auf das erste Element
  - keine Bereichsüberprüfung (*Bounds Checking*)
- Compileroptimierungen
  - Compiler wandelt Array-Code in Pointer-Code um
  - verwendet Adressierungsmodi um Arrayindizes zu skalieren
  - viele Tricks, um die Array-Indizierung in Schleifen zu verbessern
- Strukturen
  - Bytes werden in der ausgewiesenen Reihenfolge zugeteilt
  - ggf. Leerbytes, um die richtige Ausrichtung zu erreichen
- Objekte
  - wie Strukturen, zwei extra Pointer auf Typ-Infos und vtable
  - Methodenaufruf über vtable mit this-Pointer

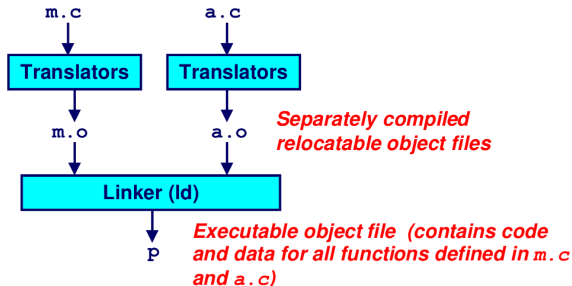
Linker und Loader

- Statisches Linken
- Object-Dateien (ELF)
- Statische Funktionsbibliotheken
- Loading
- Dynamische Funktionsbibliotheken (shared libraries)

Bisher: gesamtes Programm in einer Quelldatei

- Probleme:
  - schlechte Effizienz: jede kleine Änderung erfordert volle Neu-Compilierung des Programms
  - keine Modularisierung: wie können wichtige Funktionen wiederverwendet werden? (z.B. `malloc`, `printf`)
- Lösung:
  - Statisches Binden („static linking“)

Static Linking: Konzept



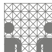
- Quelltext auf mehrere Dateien aufgeteilt
- einzeln in (verschiebbaren: „PIC“) Objektcode compiliert
- Linker baut daraus eine ausführbare Datei

Static Linking: Aufgaben des Linkers

- Zusammenführen der einzelnen (.o) Objektdateien in eine vollständige kombinierte Objektdatei
- Suchen der referenzierten Funktionen — external references
- Relozieren aller Speicherreferenzen — relocate symbols
  - für Daten — `int *xp=&x;`
  - und Funktionen — printf();
  - nicht aufgerufene Funktionen werden eliminiert

- Compiler(-driver) kümmert sich um Aufruf der einzelnen Tools

  Präprozessor (cpp), Compiler (`cc1`),
  Assembler (gas) und Linker (`ld`)
  - „Finetuning“ und Reihenfolge über Kommandozeilen-Parameter



Static Linking: Vorteile

- Programm aus übersichtlichen Modulen zusammengesetzt
- erlaubt den Aufbau von Funktionsbibliotheken, z.B. mathematische Funktionen, Standard C-Library, Datenstrukturen, TCP/IP, Grafik, ...

- schnellere Entwicklung: nur geänderte Quelltexte müssen neu compiliert werden, Linken ist viel schneller als Compilieren
- kompakte Programme: das ausführbare Programm enthält nur die tatsächlich benutzten Funktionen aus den Bibliotheken

Unix: Executable and Linkable Format (ELF)

- Unix/Linux Standard für Objektdateien
- einheitliches Dateiformat für
  - relozierbare Objektdateien (.o)
  - ausführbare Objektdateien („.exe“)
  - „shared“ Objektdateien (.so)

- ELF im Prinzip prozessor/architektur-unabhängig
- aber gegebene Objektdatei ist natürlich architektur-spezifisch
  - enthält Maschinenbefehle für Zielarchitektur
  - Infos sind im Header codiert

- Microsoft nutzt COFF/PE („portable executable“) `.exe` `.dll`
- Java Class-Format `.class`

ELF Object File Format

ELF header

magic number, Typ (.o, .so, .exe), Maschine, Byte-Order, usw.

Program Header Tabelle

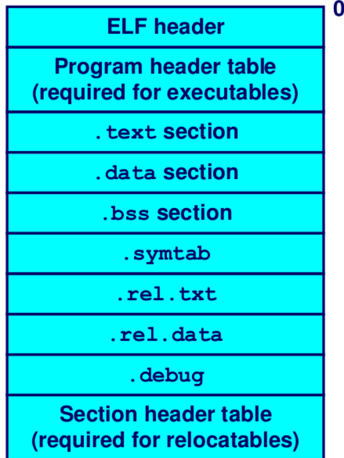
`.text` Programmcode

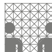
`.data` Statische Variablen

Daten mit Initialwerten (`ans=42;`)

`.bss` Daten

unitialisierte statische Daten
„block started by symbol“
„better save space“





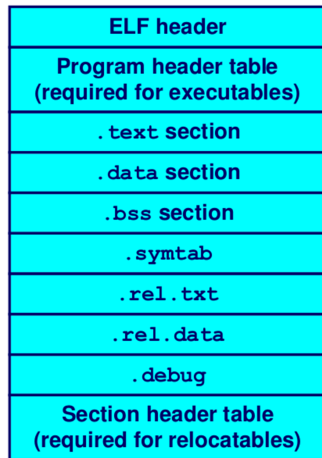
ELF Object File Format

`.symtab` Symboltabelle
: Namen aller Funktionen und statischen Variablen, Sektionsnamen und Offets

`.rel.text` Relocation-Infos
: alle Maschinenbefehle, die beim beim Linken angepasst werden müssen Adressen aller (Sprung-) Befehle, die beim Linken angepasst werden müssen

`.rel.data` Relocation-Infos
: Adressen aller Pointer, die beim Linken angepasst werden müssen

`.debug`
: Hilfsinformationen fürs Debugging



Beispiel-Quellcode

m.c

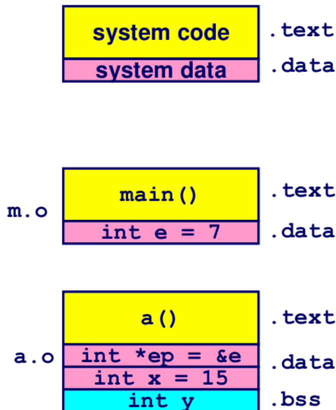
```
int e=7;

int main() {
  int r = a();
  exit(0);
}
```

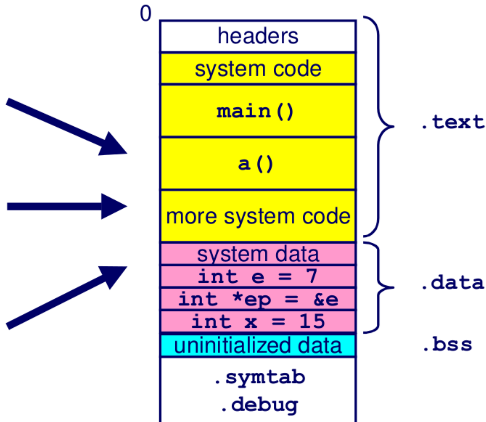
a.c

```
extern int e;

int *ep=&e;
int x=15;
int y;

int a() {
  return *ep+x+y;
}
```

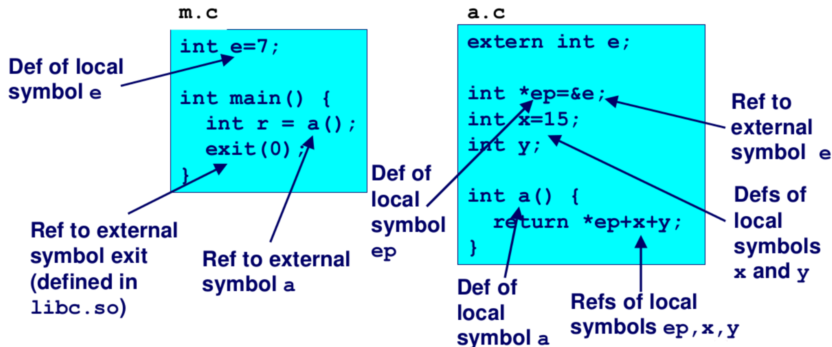
- zwei Funktionen: `main()`, `a()`
- zusätzlicher System-Code, Initialisierung und `exit()`
- vier globale Variablen: `e`, `*ep`, `x`, `y`

Erzeugte ELF-Datei





Zuordnung der externen Referenzen



Beispiel: `int e=7;` definiert und initialisiert Symbol e,
`int *ep=&e;` definiert Symbol ep und initialisiert mit der Adresse von e

m.o Relocation-Infos

m.c

```
int e=7;

int main() {
  int r = a();
  exit(0);
}
```

```
Disassembly of section .text:

00000000 <main>: 00000000 <main>:
   0:   55                  pushl  %ebp
   1:   89 e5               movl   %esp,%ebp
   3:   e8 fc ff ff ff      call   4 <main+0x4>
                            4: R_386_PC32    a
   8:   6a 00               pushl  $0x0
   a:   e8 fc ff ff ff      call   b <main+0xb>
                            b: R_386_PC32    exit
   f:   90                  nop
```

```
Disassembly of section .data:

00000000 <e>:
   0:   07 00 00 00
```

a.o Relocation-Infos für .text

a.c

```
extern int e;

int *ep=&e;
int x=15;
int y;

int a() {
  return *ep+x+y;
}
```

```
Disassembly of section .text:

00000000 <a>:
   0:   55                  pushl  %ebp
   1:   8b 15 00 00 00      movl   0x0,%edx
   6:   00
                            3: R_386_32      ep
   7:   a1 00 00 00 00      movl   0x0,%eax
                            8: R_386_32      x
   c:   89 e5               movl   %esp,%ebp
   e:   03 02               addl   (%edx),%eax
  10:   89 ec               movl   %ebp,%esp
  12:   03 05 00 00 00      addl   0x0,%eax
  17:   00
                            14: R_386_32     y
  18:   5d                  popl   %ebp
  19:   c3                  ret
```

a.o Relocation-Infos für .data

a.c

```
extern int e;

int *ep=&e;
int x=15;
int y;

int a() {
  return *ep+x+y;
}
```

```
Disassembly of section .data:

00000000 <ep>:
   0:    00 00 00 00
                          0: R_386_32     e
 00000004 <x>:
   4:    0f 00 00 00
```

Erzeugtes ausführbares Programm .text

```
08048530 <main>:
 8048530:       55                      pushl  %ebp
 8048531:       89 e5                   movl   %esp,%ebp
 8048533:       e8 08 00 00 00          call   8048540 <a>
 8048538:       6a 00                   pushl  $0x0
 804853a:       e8 35 ff ff ff          call   8048474 <_init+0x94>
 804853f:       90                      nop

08048540 <a>:
 8048540:       55                      pushl  %ebp
 8048541:       8b 15 1c a0 04          movl   0x804a01c,%edx
 8048546:       08
 8048547:       a1 20 a0 04 08          movl   0x804a020,%eax
 804854c:       89 e5                   movl   %esp,%ebp
 804854e:       03 02                   addl   (%edx),%eax
 8048550:       89 ec                   movl   %ebp,%esp
 8048552:       03 05 d0 a3 04          addl   0x804a3d0,%eax
 8048557:       08
 8048558:       5d                      popl   %ebp
 8048559:       c3                      ret
```

Erzeugtes ausführbares Programm .data

m.c

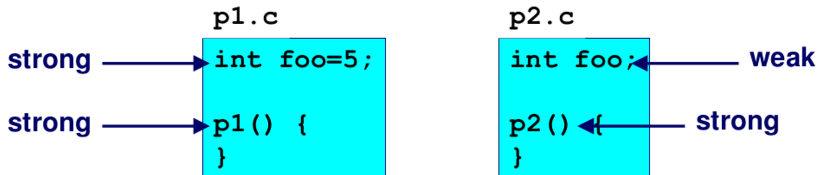
```
int e=7;

int main() {
  int r = a();
  exit(0);
}
```

a.c

```
extern int e;

int *ep=&e;
int x=15;
int y;

int a() {
  return *ep+x+y;
}
```

```
Disassembly of section .data:

0804a018 <e>:
 804a018:       07 00 00 00

0804a01c <ep>:
 804a01c:       18 a0 04 08

0804a020 <x>:
 804a020:       0f 00 00 00
```

Starke und schwache Symbole



strong: alle Prozeduren und initialisierte globale Daten

weak: nicht-initialisierte globale Daten

1. jedes starke Symbol darf nur einmal auftreten
2. ein schwaches Symbol wird einem starken Symbol zugewiesen
3. der Linker kann sich eines von mehreren schwachen aussuchen

Linker-Quiz: Separate Quelldateien (C)

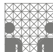
Module 1	Module 2	
`int x;` `p1() {}`	`p1() {}`	Link time error: two strong symbols (`p1`)
`int x;` `p1() {}`	`int x;` `p2() {}`	References to `x` will refer to the same uninitialized int. Is this what you really want?

`int x;`
`int y;`
`p1() {}`
	`double x;`
	`p2() {}`
		Writes to `x` in `p2` might overwrite `y`! Evil!

`int x=7;`
`int y=5;`
`p1() {}`
	`double x;`
	`p2() {}`
		Writes to `x` in `p2` will overwrite `y`! Nasty!

`int x=7;` `p1() {}`	`int x;` `p2() {}`	References to `x` will refer to the same initialized variable.

Nightmare scenario: two identical weak structs, compiled by different compilers with different alignment rules.



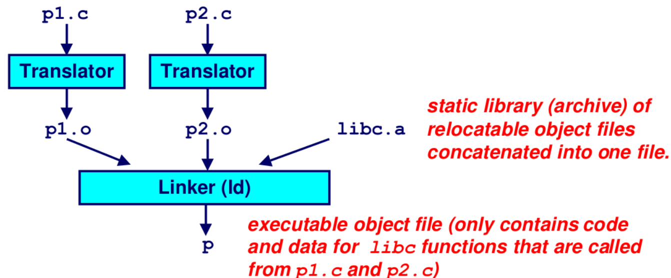
Funktionsbibliotheken

- Zugriff auf häufig benötigte Funktionen?
  - Math, Strings, I/O, Threads, Speicherverwaltung, usw.

  - alle Funktionen in einer Quelldatei ist keine Lösung
  - jede Funktion in separater Quelldatei ist sehr mühsam

- statische Funktionsbibliotheken (.a Archiv-Dateien)
  - Sammlung von compilierten Funktionen mit Index
  - Linker sucht (strong) Symbole im Index
  - gefundene Funktionen und Daten werden ins Programm eingebunden

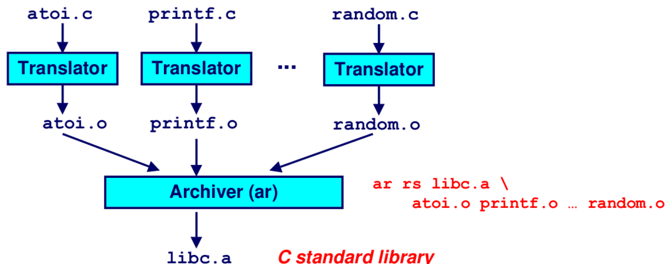
Funktionsbibliotheken



Ausführbares Programm gebaut aus

- relozierbaren Modulen (.o), compiliert aus den Quelltexten (.c)
- vordefinierten Funktionsbibliotheken (.a)
- nur die verwendeten Funktionen landen im Programm

Statische Funktionsbibliotheken zusammenbauen



- alle Funktionen der Bibliothek einzeln compilieren
- Archiver (ar) erzeugt den benötigten Index
- erzeugte ELF Datei (.a) mit Objektcode für alle Funktionen
- inkrementelles Update möglich (einzelne .c nach .o compilieren)

Wichtige Bibliotheken

- `libc.a`: die C „Standard-Bibliothek“
  - 900 Funktionen, ca. 8 MByte
  - I/O, Speicherverwaltung, Strings, Datum und Zeit, Zufallszahlen, Integer-Arithmetik, Signale
- `libm.a`: die C „Mathematik-Bibliothek“
  - 226 Funktionen, ca 1 MByte
  - Gleitkommafunktionen (sin, cos, tan, log, exp, sqrt, ...)

- Funktionen anzeigen:

```
ar -t /usr/lib/libm.a | sort                      (32-bit)
ar -t /usr/lib/x86_64-linux-gnu/libm.a | sort
```

- Java/Python/usw. benutzen eigene Bibliotheken, die wiederum auf libc/libm aufbauen

Funktionsbibliotheken verwenden

- Linker bekommt Liste der .o und .a Dateien vom Compiler
- alle Dateien werden nach fehlenden Referenzen durchsucht
- gefundene Referenzen werden sofort gelinkt („reloziert“)
- jede fehlende Referenz führt zum Abbruch

⇒ Reihenfolge der Module/Bibliotheken ist wichtig

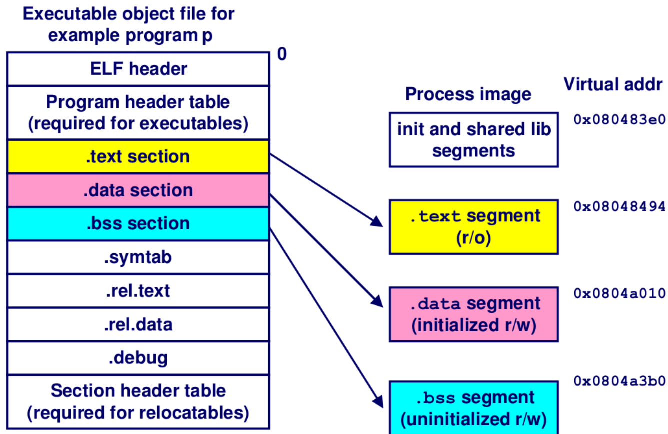
⇒ Bibliotheken gehören ans Ende der Kommandozeile

Unix-Konvention

- Bibliotheken heissen `libXYZ.a`
- Linker-Kommandozeile ohne „`lib`“, sondern nur `-lXYZ`
- Suchverzeichnisse mit `-L <dir>` Option angeben

```
gcc a.c b.c c.o d.o -L . -lbluetooth -lpthread -lm -lc
```

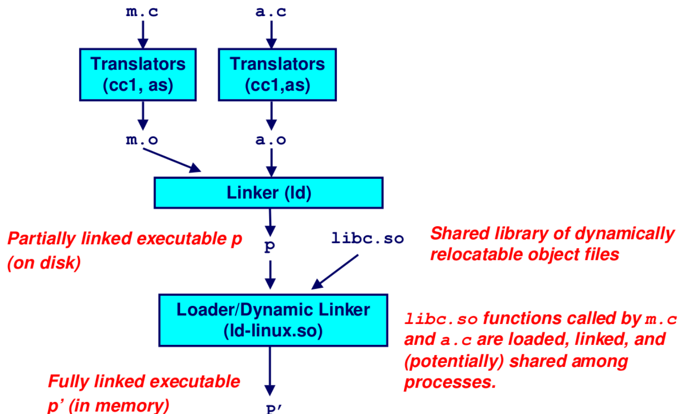
Loader: ELF-Module/Programme laden und ausführen



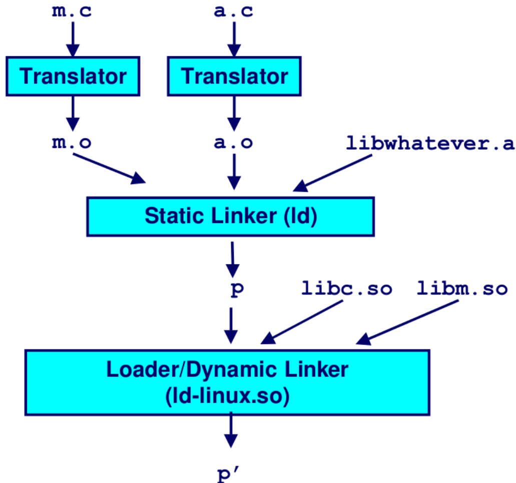
Dynamische Bibliotheken, „Shared Libraries“

- Programm wird zur Objektdatei compiliert
- Bibliotheken werden erst beim Laden dazugelinkt
- die Bibliotheken können von mehreren Prozessen gleichzeitig benutzt werden, liegen aber (maximal) einmal im Speicher
- signifikant effizienter als separat statische gelinkte Programme
- Symbole werden entweder sofort (wie beim statischen Binden) oder „lazy“ referenziert (erst beim ersten Aufruf)
- Versionierung: unter Unix/Linux ist es möglich, mehrere Versionen einer Bibliothek zu verwenden, `libopencv_core.so.2.3.1`

Dynamically Linked Shared Libraries



Das Gesamtsystem



ELF: Speicheraufteilung in Regionen

- Header: Meta-Informationen
- Stack: Funktionsaufrufe
- Heap: dynamische angeforderte Daten
- statische (globale) Daten
- Code-Bereiche
- Debug- und Relocation-Infos

- bisher noch nicht erklärt: wie funktioniert die dynamische Speicherverwaltung im Heap?

Dynamic Memory Allocation

- nicht alle Daten können statisch alloziert werden
  - Speicher ist begrenzt
  - viele Daten/Arrays werden nur zeitweise benötigt
  - viele Algorithmen basieren auf dynamischen Bäumen/Graphen
  - usw.

- Datenstrukturen dynamisch anlegen
  - erst wenn die Daten benötigt werden
  - Speicher nach Benutzung wieder freigeben
  - Assembler, C/C++ benutzen die `malloc`-Bibliotheksfunktionen
  - Ursache für viele Programmierfehler

  - moderne Sprachen (Java, C# usw.) bieten automatische Heap-Verwaltung mit einem „garbage-collector"
  - bequem, aber oft auch langsamer, weniger Kontrolle

Bryant: „Harsh Reality: Memory Matters“

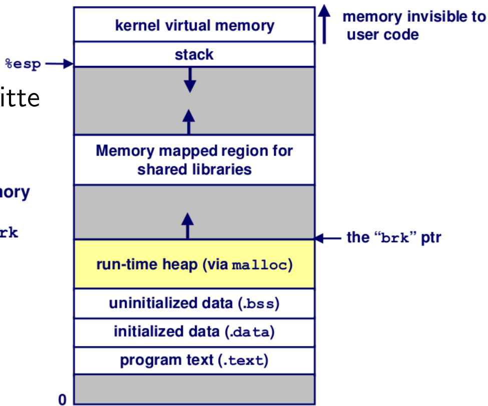
- viele Applikationen sind durch den verfügbaren Speicher begrenzt, z.B. komplexe Graph-Algorithmen

- Programmierfehler im Umgang mit dynamisch angefordertem Speicher sind häufig und schwer zu beseitigen
  - Effekt wird häufig erst spät und weit entfernt bemerkt
  - siehe wöchentliche Linux/Windows/Application Updates

- Performance eines Programms hängt entscheidend von effektivem Umgang mit dem Speicher ab
  - Cache und Virtual Memory empfindlich gegen falsche Datenstrukturen und Zugriffsmuster
  - effiziente Programmierung kann Wunder wirken

Linux: Speicherbereiche für ein Programm

- Kernel bei höchsten Adressen
- Stack wächst nach unten
- Shared-Bibliotheken in der Mitte



Allocators request additional heap memory from the operating system using the sbrk function.

- Heap (dynamische Daten)
- globale statische Daten
- Programmcode
- Startup-Code ab Adresse 0

Malloc: Funktionen

```
void* malloc( size_t size )
```

- liefert Pointer auf Speicherbereich mit mindestens `size` Bytes, typisch ausgerichtet an 8-Byte Adressen
- Aufruf mit `size == 0` liefert `NULL`
- liefert `NULL`, wenn nicht erfolgreich

```
void free( void *p )
```

- gibt den Speicherbereich `*p` ans Betriebssystem zurück
- Pointer `p` von vorherigem Aufruf von `malloc` oder `realloc`

```
void* realloc( void *p, size_t size)
```

- ändert die Größe des Speicherbereichs `*p`
- wenn erfolgreich, bleibt der Inhalt des Speicherbereichs unverändert, bis zum Minimum der alten und neuen Größe

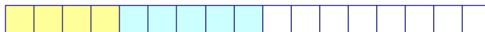
Malloc: Beispielcode

```
void foo(int n, int m) {
  int i, *p;

  /* allocate a block of n ints */
  if ((p = (int *) malloc(n * sizeof(int))) == NULL) {
    perror("malloc");
    exit(0);
  }
  for (i=0; i<n; i++)
    p[i] = i;

  /* add m bytes to end of p block */
  if ((p = (int *) realloc(p, (n+m) * sizeof(int))) == NULL)
{
    perror("realloc");
    exit(0);
  }
  for (i=n; i < n+m; i++)
    p[i] = i;

  /* print new array */
  for (i=0; i<n+m; i++)
    printf("%d\n", p[i]);

  free(p); /* return p to available memory pool */
}
```

Malloc: Beispiel

```
p1 = malloc(4)
```

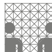
```
p2 = malloc(5)
```



```
p3 = malloc(6)
```

```
free(p2)
```

```
p4 = malloc(2)
```



Malloc: Anforderungen

- Programme können jederzeit `malloc` und `free` aufrufen
- die Anzahl oder Größe der angeforderten Blöcke kann nicht von der Speicherverwaltung beeinflusst werden
- Anfragen müssen sofort und möglichst schnell erfüllt werden
- dies erfordert ausreichende freie Speicherbereiche

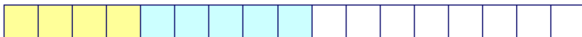
- einmal allozierte Blöcke stehen für weitere Anfragen nicht mehr zur Verfügung, es sei denn, sie werden mit `free()` wieder freigegeben

- Vertiefung: eigenes `malloc` implementieren und testen :-)

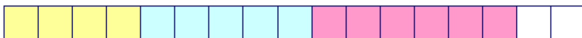
Problem: Fragmentierung

`p1 = malloc(4)`

`p2 = malloc(5)`



`p3 = malloc(6)`

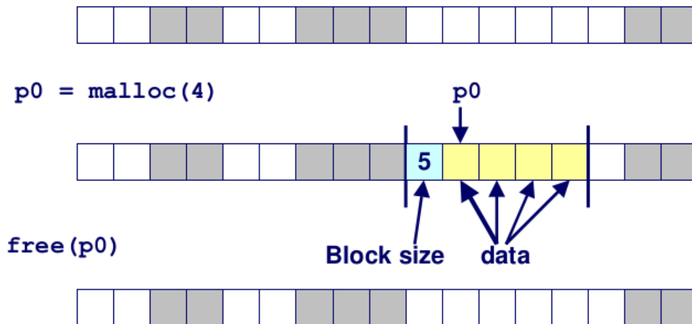


`free(p2)`

`p4 = malloc(6)`

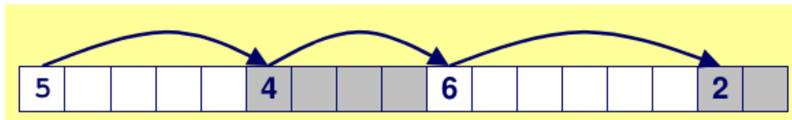
oops. Wir haben nur Platz für höchstens `malloc(5)`.

Idee zur Implementierung von `free()`



- Länge eines Blocks im Header gespeichert
- mindestens ein extra-Wort pro Block

Freie Blöcke finden

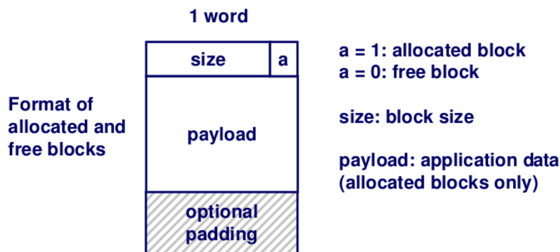


- Länge eines Blocks im Header gespeichert
  - Zusatz-/Verwaltungsdaten außerhalb des angeforderten Blocks
  - `malloc` und `free` kennen das Speicherlayout, und können Blöcke suchen bzw. zurückgeben

  - doppelte verkettete Listen (vorwärts/rückwärts) und Graphen sind effizienter als die gezeigte einfache Liste
  - Details: Bryant O'Hallaron, Knuth

Freie Blöcke finden: Datenstruktur

- wie erkennt man, ob ein Block belegt ist?



- erfordert 1-bit extra,
- z.B. das niederwertigste Bit im `size` Feld bei wortweiser Allozierung (32-bit) und byte-weiser Adressierung

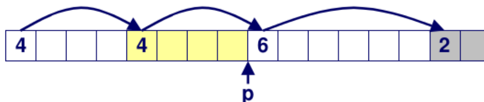
Freie Blöcke finden

- first fit: Liste vom Anfang an durchsuchen, erster passender Block wird zurückgeliefert. Linearer Zeitbedarf.

```
p = start;
while ((p < end) ||      \\ not passed end
       (*p & 1) ||       \\ already allocated
       (*p <= len));     \\ too small
```

- next fit: startet die Suche vom zuletzt gefundenen Block. Fragmentierung häufig schlechter als bei first-fit.
- best fit: gesamte Liste durchsuchen, Block mit kleinstem Verschnitt zurückliefern. Weniger Fragmentierung, aber langsamer als first-fit.

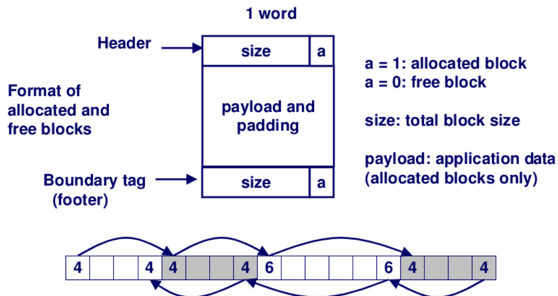
Freie Blöcke finden




```
void addblock(ptr p, int len) {
  int newsize = ((len + 1) >> 1) << 1;  // add 1 and round up
  int oldsize = *p & -2;                // mask out low bit
  *p = newsize | 1;                     // set new length
  if (newsize < oldsize)
    *(p+newsize) = oldsize - newsize;   // set length in remaining
}                                       //   part of block
```

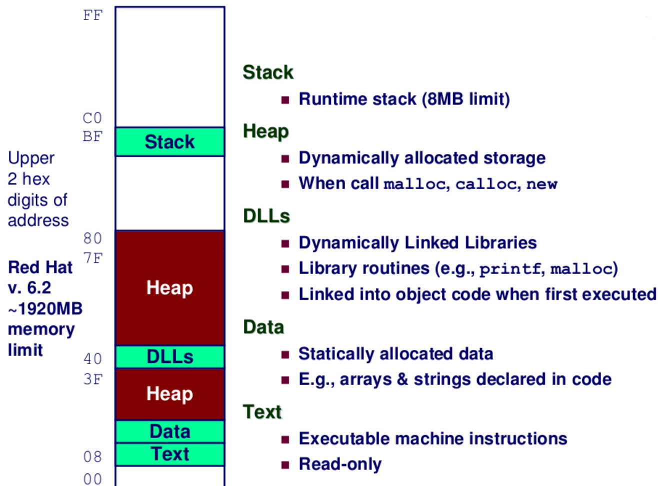
```
addblock(p, 2)
```

Bidirectional Coalescing

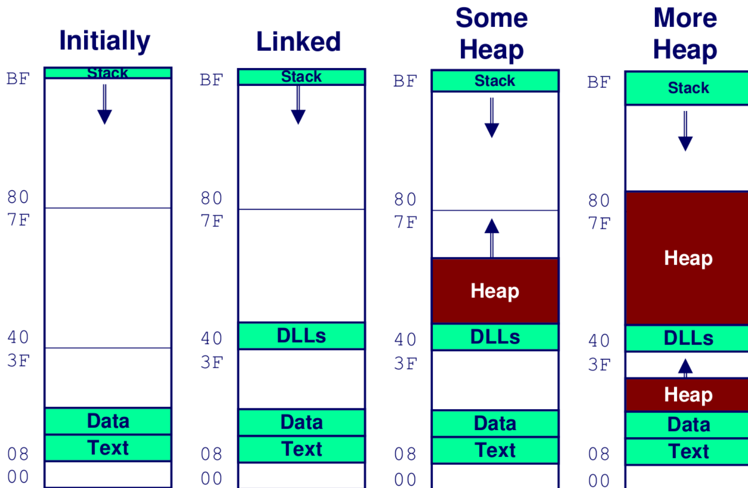


- size/allocated-Infos doppelt am Beginn und Ende des Nutzdaten-Blocks. Liste kann vorwärts und rückwärts schnell durchlaufen werden.
- schnelles Verschmelzen benachbarter freier Blöcke

Linux: Speicherlayout



Linux: Speicherverwaltung



Linux: Beispiel für Text und Stack

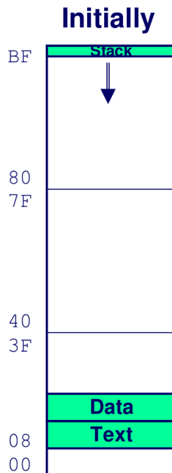
```
(gdb) break main
(gdb) run
  Breakpoint 1, 0x804856f in main ()
(gdb) print $esp
  $3 = (void *) 0xbffffc78
```

Main

- **Address** `0x804856f` **should be read** `0x0804856f`

Stack

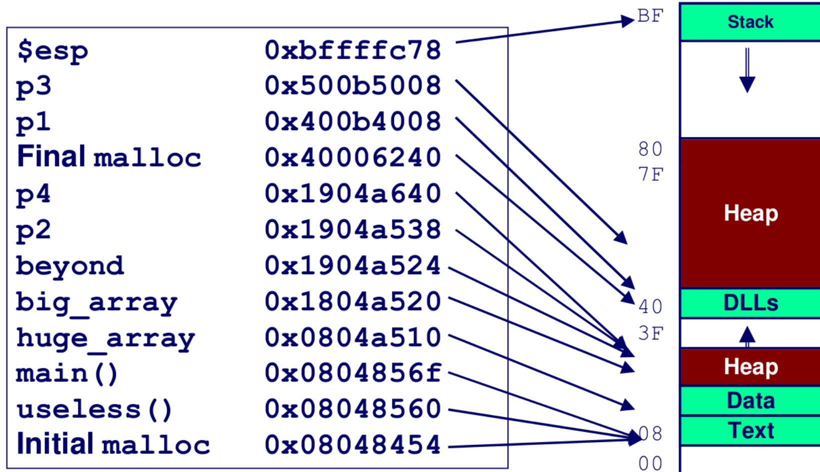
- **Address** `0xbffffc78`



Beispiel für Malloc

```
char big_array[1<<24];  /*  16 MB */
char huge_array[1<<28]; /* 256 MB */

int beyond;
char *p1, *p2, *p3, *p4;

int useless() {  return 0; }

int main()
{
 p1 = malloc(1 <<28);  /* 256 MB */
 p2 = malloc(1 << 8);  /* 256 B  */
 p3 = malloc(1 <<28);  /* 256 MB */
 p4 = malloc(1 << 8);  /* 256 B  */
 /* Some print statements ... */
}
```

Beispiel: Speicherbereiche



Operatoren in C

sortiert nach Operator-Priorität

Operators	Associativity
() [] -> .	left to right
! ~ ++ -- + - * & (type) sizeof	right to left
* / %	left to right
+ -	left to right
<< >>	left to right
< <= > >=	left to right
== !=	left to right
&	left to right
^	left to right
|	left to right
&&	left to right
||	left to right
?:	right to left
= += -= *= /= %= &= ^= != <<= >>=	right to left
,	left to right

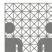
Syntax für Pointer in C

`int *p`	p is a pointer to int
`int *p[13]`	p is an array[13] of pointer to int
`int *(p[13])`	p is an array[13] of pointer to int
`int **p`	p is a pointer to a pointer to an int
`int (*p)[13]`	p is a pointer to an array[13] of int
`int *f()`	f is a function returning a pointer to int
`int (*f)()`	f is a pointer to a function returning int
`int (*(*f())[13])()`	f is a function returning ptr to an array[13] of pointers to functions returning int
`int (*(*x[3])())[5]`	x is an array[3] of pointers to functions returning pointers to array[5] of ints

Gruselkabinett: Memory-related Bugs

- ungültige Pointer dereferenzieren
- nicht existierende Variablen referenzieren
- nicht-initialisierten Speicher lesen
- Speicherbereiche überschreiben
- freie Blöcke referenzieren
- Blöcke mehrfach freigeben
- Blöcke nicht freigeben: Speicherlecks

- Details: Bryant/O'Hallaron Buch
- Java: die meisten (dieser) Fehler sind unmöglich



Ungültige Pointer dereferenzieren

der „klassische“ `scanf`-Bug

```
scanf(“%d”, val);
```

lokale Variablen „verschwinden“ nach dem Rücksprung:

```
int *foo () {
   int val;
   return &val;
}
```

tückisch: direkt nach dem Rücksprung liegen die Daten noch auf dem Stack, werden aber von späteren Funktionsaufrufen überschrieben

Nicht initialisierten Speicher lesen

per `malloc` allozierter Speicher ist nicht initialisiert

`calloc` aufrufen oder Bereich explizit initialisieren

```
/* return y = Ax */
int *matvec(int **A, int *x) {
   int *y = malloc(N*sizeof(int));
   int i, j;

   for (i=0; i<N; i++)
      for (j=0; j<N; j++)
         y[i] += A[i][j]*x[j];
   return y;
}
```

Speicherbereiche überschreiben

versehentlich falsche Größe beim `malloc`

```
int **p;

p = malloc(N*sizeof(int));

for (i=0; i<N; i++) {
   p[i] = malloc(M*sizeof(int));
}
```

off-by-one Fehler

```
int **p;

p = malloc(N*sizeof(int *));

for (i=0; i<=N; i++) {
   p[i] = malloc(M*sizeof(int));
}
```

Speicherbereiche überschreiben

Maximalgröße von Puffern nicht beachtet

```
char s[8];
int i;

gets(s);  /* reads "123456789" from stdin */
```

sehr häufiger Fehler, Einfallstor für Schadsoftware

Missverständnis der Pointerarithmetik

```
int *search(int *p, int val) {

   while (*p && *p != val)
      p += sizeof(int);

   return p;
}
```

Speicherbereiche mehrfach freigeben

```
x = malloc(N*sizeof(int));
<manipulate x>
free(x);

y = malloc(M*sizeof(int));
<manipulate y>
free(x);
```

```
x = malloc(N*sizeof(int));
<manipulate x>
free(x);
...
y = malloc(M*sizeof(int));
for (i=0; i<M; i++)
   y[i] = x[i]++;
```

Speicherbereiche nicht freigeben: Speicherlecks

```
foo() {
    int *x = malloc(N*sizeof(int));
    ...
    return;
}
```

- nach dem Rücksprung bleibt der Speicher belegt, aber es gibt keinen (gültigen) Pointer mehr

Speicherbereiche nur teilweise freigeben

```
struct list {
   int val;
   struct list *next;
};

foo() {
   struct list *head =
           malloc(sizeof(struct list));
   head->val = 0;
   head->next = NULL;
   <create and manipulate the rest of the list>
   ...
   free(head);
   return;
}
```

gets aus Standard C Library

Puffer wird übergeben, aber Anzahl der gelesenen Zeichen ist nicht limitiert

```
/* Get string from stdin */
char *gets(char *dest)
{
    int c = getc();
    char *p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getc();
    }
    *p = '\0';
    return dest;
}
```

Verwundbarer Code

Puffer liegt auf dem Stack, ist viel zu klein

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
int main()
{
  printf("Type a string:");
  echo();
  return 0;
}
```

Verwundbarer Code: Beispiele

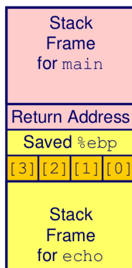
```
unix>./bufdemo
Type a string:123
123
```

```
unix>./bufdemo
Type a string:12345
Segmentation Fault
```

```
unix>./bufdemo
Type a string:12345678
Segmentation Fault
```

Verwundbarer Code: Beispiele

Array überschreibt den Stack



%ebp
buf

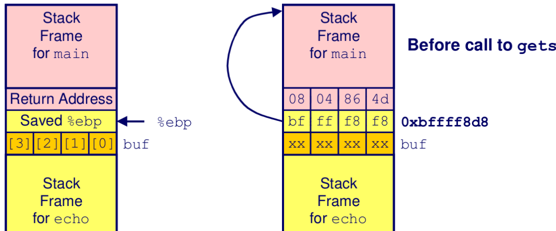
```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
    pushl %ebp              # Save %ebp on stack
    movl %esp,%ebp
    subl $20,%esp           # Allocate space on stack
    pushl %ebx              # Save %ebx
    addl $-12,%esp          # Allocate space on stack
    leal -4(%ebp),%ebx      # Compute buf as %ebp-4
    pushl %ebx              # Push buf on stack
    call gets               # Call gets
    . . .
```

Verwundbarer Code: Beispiele

Array überschreibt den Stack

```
unix> gdb bufdemo
(gdb) break echo
Breakpoint 1 at 0x8048583
(gdb) run
Breakpoint 1, 0x8048583 in echo ()
(gdb) print /x *(unsigned *)$ebp
$1 = 0xbffff8f8
(gdb) print /x *((unsigned *)$ebp + 1)
$3 = 0x804864d
```

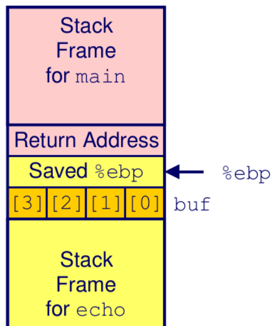


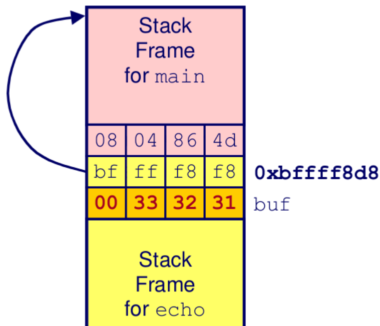

```
8048648:  call  804857c <echo>
804864d:  mov   0xffffffe8(%ebp),%ebx # Return Point
```

Verwundbarer Code: Beispiele

Array überschreibt den Stack

vor dem Aufruf von `gets`, Eingabe "123"

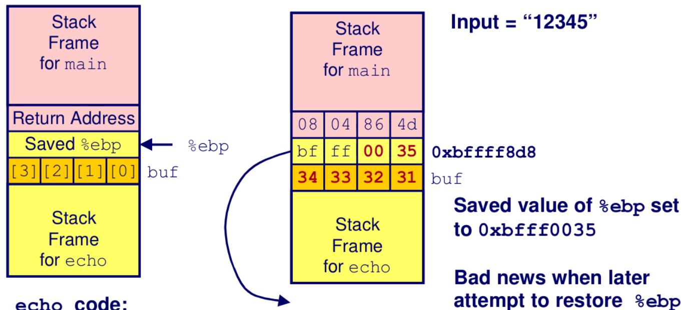




alles ok.

Verwundbarer Code: Beispiele

Array überschreibt den Stack

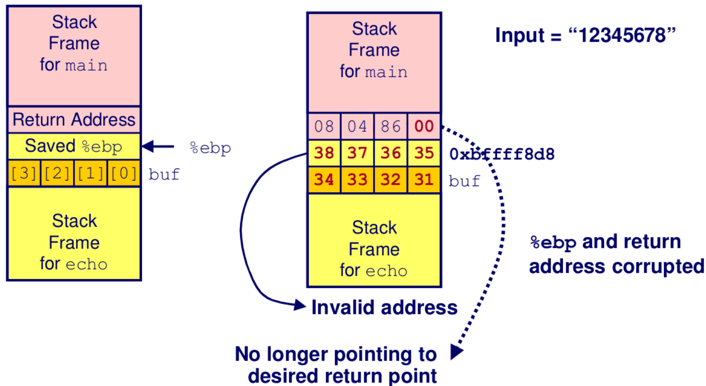


echo code:

```
8048592: push   %ebx
8048593: call   80483e4 <_init+0x50>  # gets
8048598: mov    0xffffffe8(%ebp),%ebx
804859b: mov    %ebp,%esp
804859d: pop    %ebp      # %ebp gets set to invalid value
804859e: ret
```

Verwundbarer Code: Beispiele

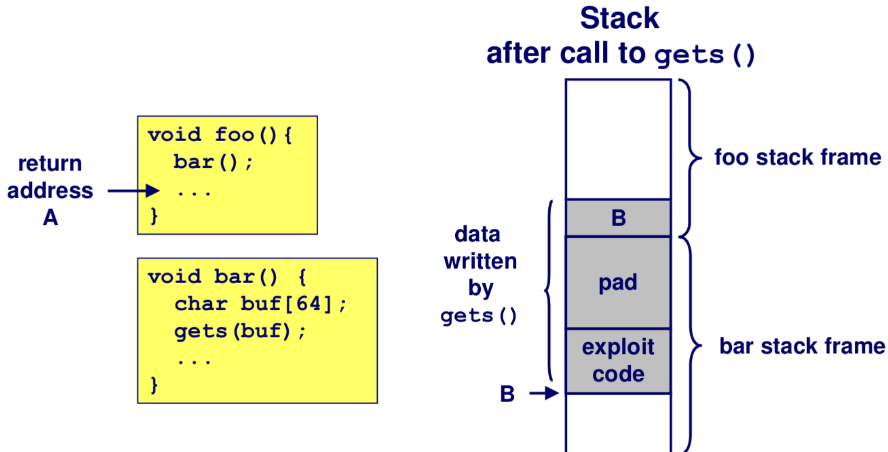
Return-Adresse überschrieben




```
8048648:  call  804857c <echo>
804864d:  mov   0xffffffe8(%ebp),%ebx  # Return Point
```

Verwundbarer Code: Beispiele

Return-Adresse überschrieben: Rücksprung in Schad-Code



Puffer-Überläufe: Sicherheitslücken

- Buffer-Overflow Bugs erlauben es Angreifern, beliebigen Code auf den angegriffenen Rechnern auszuführen!
- der Code wird vom Angreifer mitgeliefert

```
GET
  /default.ida?NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN....NNNN
  NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN%u9090%u6858%ucbd3%u780
  1%u9090%u6858%ucbd3%u7801%u9090%u6858%ucbd3%u7801%u9090%u909
  0%u8190%u00c3%u0003%u8b00%u531b%u53ff%u0078%u0000%u00=a
HTTP/1.0" 400 325 "-" "-"
```

$\Rightarrow$ sorgfältig programmieren

$\Rightarrow$ verwundbare Bibliotheksfunktionen vermeiden

Assemblerebene: Zusammenfassung

- Umsetzung von Programmen mit Kontrollstrukturen
- Funktionsaufrufe, Parameter, lokale Variablen

- Speicherlayout von strukturierten Daten und Arrays
- Dynamische Speicherverwaltung im Heap
- Umsetzung objektorienter Konzepte

- ELF-Dateiformat und statisches Linking
- Programmcode, Stack, Heap, statische Variablen
- Funktionsbibliotheken