

Systemnahe Programmierung in C/C++

– Compiler, Zeiger, Speicherverwaltung, Macros –

Knut Stolze

`stolze@informatik.uni-jena.de`

Lehrstuhl für Datenbanken und Informationssysteme
Fakultät für Mathematik und Informatik

2006–10–18

- 1 Einführung
- 2 Compiler
 - Compile-Prozess
 - Compiler
- 3 Zeiger & Arrays
 - Grundlagen
 - Arrays
- 4 Speicherverwaltung
 - Arten von Speicher
 - Funktionen
 - Tipps & Hinweise
- 5 Präprozessor & Macros
- 6 Aufgaben

- 1 Einführung
- 2 Compiler
 - Compile-Prozess
 - Compiler
- 3 Zeiger & Arrays
 - Grundlagen
 - Arrays
- 4 Speicherverwaltung
 - Arten von Speicher
 - Funktionen
 - Tipps & Hinweise
- 5 Präprozessor & Macros
- 6 Aufgaben

- Ursprünglich von Dennis Ritchie (Bell Laboratories) für die Entwicklung von UNIX in den 1970ern entworfen
 - Hatte sehr starken Einfluss auf andere Programmiersprachen (C++, Java)
 - Sehr systemnah
 - Bietet jedoch genügend Abstraktion gegenüber Assembler
 - Sehr weites Einsatzgebiet
 - *Die* Programmiersprache für Systemprogrammierung
 - Wird (wurde) auch für Anwendungsentwicklung eingesetzt
 - Gut portierbar
 - Üblicherweise die *erste* High-Level-Sprache für neue Computer
- ⇒ Abhängig von Problem nicht immer die beste Wahl

- Ursprünglich von Dennis Ritchie (Bell Laboratories) für die Entwicklung von UNIX in den 1970ern entworfen
 - Hatte sehr starken Einfluss auf andere Programmiersprachen (C++, Java)
 - Sehr systemnah
 - Bietet jedoch genügend Abstraktion gegenüber Assembler
 - Sehr weites Einsatzgebiet
 - *Die* Programmiersprache für Systemprogrammierung
 - Wird (wurde) auch für Anwendungsentwicklung eingesetzt
 - Gut portierbar
 - Üblicherweise die *erste* High-Level-Sprache für neue Computer
- ⇒ Abhängig von Problem nicht immer die beste Wahl

Eigenschaften von C

- Systemnähe erlaubt einfache Übersetzung (Compilation)
 - Gute Optimierungsmöglichkeiten; nur wenige Maschineninstruktionen für Sprachelemente
 - Keine komplexe Laufzeitumgebung
 - Bietet Low-level Zugang zu Speicher, Devices, ...
 - Einfach und ausdrucksstark, jedoch etwas kryptisch
 - Kompakter Code
-
- C code kann 20-50% schneller sein als Programme in High-Level Programmiersprachen; allerdings ist Programmierung oft aufwändiger
 - Meist sind nur 10% des Codes performance-kritisch

- Nicht die beste Programmiersprache für Einsteiger

- **ABER:**

Jeder Informatikstudent sollte ein gutes Verständnis dafür haben, was auf Systemebene passiert.

- Wir decken keine Grundlagen der Programmierung ab!

- Nicht die beste Programmiersprache für Einsteiger
- **ABER:**

Jeder Informatikstudent sollte ein gutes Verständnis dafür haben, was auf Systemebene passiert.

- Wir decken keine Grundlagen der Programmierung ab!

- Nicht die beste Programmiersprache für Einsteiger
- **ABER:**

Jeder Informatikstudent sollte ein gutes Verständnis dafür haben, was auf Systemebene passiert.

- Wir decken keine Grundlagen der Programmierung ab!

- 1 Einführung
- 2 Compiler**
 - Compile-Prozess
 - Compiler
- 3 Zeiger & Arrays
 - Grundlagen
 - Arrays
- 4 Speicherverwaltung
 - Arten von Speicher
 - Funktionen
 - Tipps & Hinweise
- 5 Präprozessor & Macros
- 6 Aufgaben

Warum Compiler?

- C (und C++) sind *keine* interpretierten Sprachen
- Quellcode wird vor dem Ausführen in Maschinencode übersetzt
- Maschinencode ist abhängig vom jeweiligen Prozessor bzw. dessen Architektur und Befehlssatz

⇒ Compiler übersetzt Quellcode in Maschinencode

- Quellcode üblicherweise strukturiert in mehrere Dateien
 - Bessere Modularisierung
 - Trennung von Implementierung und Schnittstellen (Header)

Warum Compiler?

- C (und C++) sind *keine* interpretierten Sprachen
- Quellcode wird vor dem Ausführen in Maschinencode übersetzt
- Maschinencode ist abhängig vom jeweiligen Prozessor bzw. dessen Architektur und Befehlssatz

⇒ Compiler übersetzt Quellcode in Maschinencode

- Quellcode üblicherweise strukturiert in mehrere Dateien
 - Bessere Modularisierung
 - Trennung von Implementierung und Schnittstellen (Header)

Warum Compiler?

- C (und C++) sind *keine* interpretierten Sprachen
- Quellcode wird vor dem Ausführen in Maschinencode übersetzt
- Maschinencode ist abhängig vom jeweiligen Prozessor bzw. dessen Architektur und Befehlssatz

⇒ Compiler übersetzt Quellcode in Maschinencode

- Quellcode üblicherweise strukturiert in mehrere Dateien
 - Bessere Modularisierung
 - Trennung von Implementierung und Schnittstellen (Header)

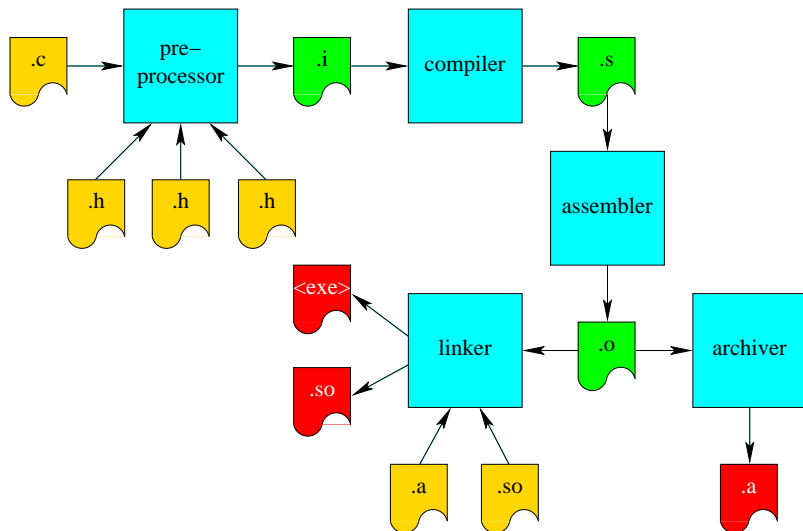
Warum Compiler?

- C (und C++) sind *keine* interpretierten Sprachen
- Quellcode wird vor dem Ausführen in Maschinencode übersetzt
- Maschinencode ist abhängig vom jeweiligen Prozessor bzw. dessen Architektur und Befehlssatz

⇒ Compiler übersetzt Quellcode in Maschinencode

- Quellcode üblicherweise strukturiert in mehrere Dateien
 - Bessere Modularisierung
 - Trennung von Implementierung und Schnittstellen (Header)

Compile-Prozess



Schritte (1)

1 Precompiler

- Bindet Header-Dateien ein
- Löst Macros auf
- In Compiler integriert
- Precompile-Ausgabe mit Option `-E` erzeugen:

```
gcc -E file.c
```

- Für Debugging-Zwecke

2 Compiler

- Übersetzt C Code in Assembler Code
- In Compiler integriert
- Assembler-Ausgabe mit Option `-S` erzeugen:

```
gcc -S file.c
```

- Für Debugging-Zwecke

3 Assembler

- Übersetzt Assembler Code in Maschinencode (Objektcode)
- In Compiler integriert
- Objektcode mit Option `-c` erzeugen:

```
gcc -c file.c
```

1 Precompiler

- Bindet Header-Dateien ein
- Löst Macros auf
- In Compiler integriert
- Precompile-Ausgabe mit Option `-E` erzeugen:

```
gcc -E file.c
```

- Für Debugging-Zwecke

2 Compiler

- Übersetzt C Code in Assembler Code
- In Compiler integriert
- Assembler-Ausgabe mit Option `-S` erzeugen:

```
gcc -S file.c
```

- Für Debugging-Zwecke

3 Assembler

- Übersetzt Assembler Code in Maschinencode (Objektcode)
- In Compiler integriert
- Objektcode mit Option `-c` erzeugen:

```
gcc -c file.c
```

1 Precompiler

- Bindet Header-Dateien ein
- Löst Macros auf
- In Compiler integriert
- Precompile-Ausgabe mit Option `-E` erzeugen:

```
gcc -E file.c
```

- Für Debugging-Zwecke

2 Compiler

- Übersetzt C Code in Assembler Code
- In Compiler integriert
- Assembler-Ausgabe mit Option `-S` erzeugen:

```
gcc -S file.c
```

- Für Debugging-Zwecke

3 Assembler

- Übersetzt Assembler Code in Maschinencode (Objektcode)
- In Compiler integriert
- Objektcode mit Option `-c` erzeugen:

```
gcc -c file.c
```

4 Linker

- Verknüpft mehrere Dateien mit Objektcode (.o), Archive (.a) und Shared Libraries (.so) zu
 - Ausführbarem Programm, oder
 - Shared Library
- Symbole werden verifiziert und ggf. aufgelöst
- Wird oft über Compiler aufgerufen:

```
gcc file1.o file2.o archive.a -o prog
```

- Kann auch direkt aufgerufen werden (üblicherweise Programmname ld)

5 Archivierer

- Kombiniert mehrere Dateien mit Objektcode (.o) in ein Archiv (.a)
- Wird über separates Programm `ar` initiiert

```
ar c archive.a file1.o file2.o
```

4 Linker

- Verknüpft mehrere Dateien mit Objektcode (.o), Archive (.a) und Shared Libraries (.so) zu
 - Ausführbarem Programm, oder
 - Shared Library
- Symbole werden verifiziert und ggf. aufgelöst
- Wird oft über Compiler aufgerufen:

```
gcc file1.o file2.o archive.a -o prog
```

- Kann auch direkt aufgerufen werden (üblicherweise Programmname ld)

5 Archivierer

- Kombiniert mehrere Dateien mit Objektcode (.o) in ein Archiv (.a)
- Wird über separates Programm `ar` initiiert

```
ar c archive.a file1.o file2.o
```

Wahl des Compilers

- Compiler sind plattformabhängig, z. B.
 - Linux: gcc, icc (Intel)
 - AIX: gcc, xlc (Visual Age C Compiler von IBM)
 - HPUX: gcc, aCC
 - Windows: cl, gcc, icc
- Lizenzkosten!
- gcc auf vielen Plattformen verfügbar
 - Herstellerspezifischer Compiler kann bessere Ergebnisse liefern
- Unterschiede i. A. nur im High-Performance-Bereich relevant, z. B. bei Betriebssystemen oder DBMS
- Feine Unterschiede bei Erkennung von Programmierfehlern/-warnungen
 - Wenn möglich, Quellcode mit mehreren Compilern übersetzen und Testen

Compileroptionen

- Differieren von Compiler zu Compiler!!
- Typischerweise:
 - I<dir> Verzeichnisse, in denen Header-Dateien gesucht werden
 - L<dir> Verzeichnisse, in denen Bibliotheken gesucht werden
 - l<lib> Bibliothek <lib> wird beim Linken eingebunden
 - W<warn> Warnungen aktivieren, z. B. -Wall
 - g Debugging-Symbole in Objektdateien hinterlegen
 - D<macro> Definiere Macro <macro>
 - O<level> Optimierung und Optimierungslevel einschalten, z. B. -O2
 - o<file> Ergebnis in Datei <file> schreiben
 - f<option> Generelle Compiler-Option einschalten, z. B. -ffast-math
- Handbuch & man-Pages des Compilers konsultieren

- Compiler, Linker, Archivierer sind auch nur Programme
 - ⇒ Enthalten potentiell auch Fehler
 - Fehler sehr schwer zu finden
- Unterschiede im Verhalten zwischen Debug- und optimierten Objektcode möglich
 - Zwischenergebnisse können in Prozessorregistern gehalten oder in den Hauptspeicher zurückgeschrieben werden
 - Register haben höhere Genauigkeit als 64-bit `double`-Werte
 - Rightarrow* Unterschiedliche Rundung von Ergebnissen
 - Hohe Optimierungslevel beim `gcc` können semantische Abweichungen verursachen, z. B. Code mit Seiteneffekten

- 1 Einführung
- 2 Compiler
 - Compile-Prozess
 - Compiler
- 3 Zeiger & Arrays**
 - **Grundlagen**
 - **Arrays**
- 4 Speicherverwaltung
 - Arten von Speicher
 - Funktionen
 - Tipps & Hinweise
- 5 Präprozessor & Macros
- 6 Aufgaben

- Ursprünge in Prozessorarchitektur
 - Indirekte Adressierung
- Wert eines Zeiger (*Pointer*) verweist auf anderen Wert

Zeiger = Adresse im Hauptspeicher

- Durch * bei Deklaration gekennzeichnet
- * dereferenziert Zeiger
 - Gibt Wert zurück, der an Adresse steht, auf die Zeiger verweist
- Adress-Operator & ermittelt Adresse einer Variablen, die Zeiger zugewiesen werden kann

- Ursprünge in Prozessorarchitektur
 - Indirekte Adressierung
- Wert eines Zeiger (*Pointer*) verweist auf anderen Wert

Zeiger = Adresse im Hauptspeicher

- Durch * bei Deklaration gekennzeichnet
- * dereferenziert Zeiger
 - Gibt Wert zurück, der an Adresse steht, auf die Zeiger verweist
- Adress-Operator & ermittelt Adresse einer Variablen, die Zeiger zugewiesen werden kann

- Ursprünge in Prozessorarchitektur
 - Indirekte Adressierung
- Wert eines Zeiger (*Pointer*) verweist auf anderen Wert

Zeiger = Adresse im Hauptspeicher

- Durch * bei Deklaration gekennzeichnet
- * **dereferenziert** Zeiger
 - Gibt Wert zurück, der an Adresse steht, auf die Zeiger verweist
- Adress-Operator & ermittelt Adresse einer Variablen, die Zeiger zugewiesen werden kann

- Ursprünge in Prozessorarchitektur
 - Indirekte Adressierung
- Wert eines Zeiger (*Pointer*) verweist auf anderen Wert

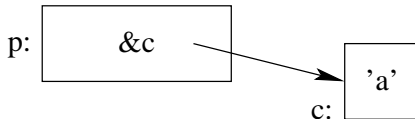
Zeiger = Adresse im Hauptspeicher

- Durch * bei Deklaration gekennzeichnet
- * dereferenziert Zeiger
 - Gibt Wert zurück, der an Adresse steht, auf die Zeiger verweist
- Adress-Operator & ermittelt Adresse einer Variablen, die Zeiger zugewiesen werden kann

Zeiger – Beispiel

```
char c = 'a';  
char *p = &c;  
char n = *p;
```

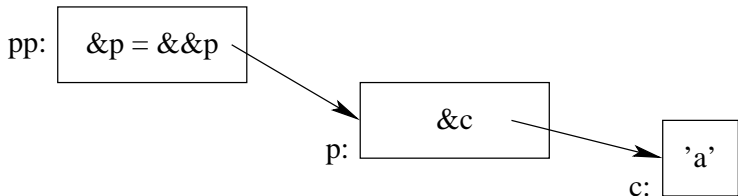
- `p` hält Adresse, an der Wert von `c` ('a') gespeichert ist



Zeiger auf Zeiger

- Referenzierter Wert kann wiederum Zeiger sein

```
char c = 'a';  
char *p = &c;  
char **pp = &&p;  
char n = **pp;
```



- Kann bis zu beliebiger Tiefe weiter geschachtelt werden
 - Bei mehr als 2 Indirektionen sehr schlecht zu durchschauen

Grundlagen Zeigerarithmetik

- Ergebnis von `malloc` zeigt auf **erstes** Element des spezifizierten Datentyps (nach Cast)

```
int *ptr = (int *)malloc(n * sizeof(int));
```

- `ptr` zeigt auf `int`-Wert
- Wie auf nachfolgende $n - 1$ Werte zugreifen?
⇒ Zeiger weiterrücken
- Zeigerarithmetik arbeitet grundsätzlich auf zu Grunde liegenden Datentyp

```
ptr = ptr + 1;
```

- ⇒ Zeiger zeigt auf nächsten `int`-Wert – nicht auf das nächste Byte!
- Bei 32-bit Integers wird Zeiger um 4 Bytes weitergerückt

Zeiger sind keine Zahlen!

Grundlagen Zeigerarithmetik

- Ergebnis von `malloc` zeigt auf **erstes** Element des spezifizierten Datentyps (nach Cast)

```
int *ptr = (int *)malloc(n * sizeof(int));
```

- `ptr` zeigt auf `int`-Wert
- Wie auf nachfolgende $n - 1$ Werte zugreifen?
⇒ Zeiger weiterrücken
- Zeigerarithmetik arbeitet grundsätzlich auf zu Grunde liegenden Datentyp

```
ptr = ptr + 1;
```

- ⇒ Zeiger zeigt auf nächsten `int`-Wert – nicht auf das nächste Byte!
- Bei 32-bit Integers wird Zeiger um 4 Bytes weitergerückt

Zeiger sind keine Zahlen!

Grundlagen Zeigerarithmetik

- Ergebnis von `malloc` zeigt auf **erstes** Element des spezifizierten Datentyps (nach Cast)

```
int *ptr = (int *)malloc(n * sizeof(int));
```

- `ptr` zeigt auf `int`-Wert
- Wie auf nachfolgende $n - 1$ Werte zugreifen?
⇒ Zeiger weiterrücken
- Zeigerarithmetik arbeitet grundsätzlich auf zu Grunde liegenden Datentyp

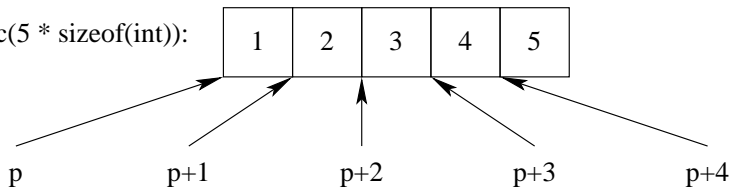
```
ptr = ptr + 1;
```

- ⇒ Zeiger zeigt auf nächsten `int`-Wert – nicht auf das nächste Byte!
- Bei 32-bit Integers wird Zeiger um 4 Bytes weitergerückt

Zeiger sind keine Zahlen!

Zeigerarithmetik – Beispiel

`p = (int *)malloc(5 * sizeof(int));`



- Arrays sind *meist* identisch mit Zeigern

Ausname: sizeof Operator

```
sizeof(int *) == 4  
sizeof(int [5]) == 20
```

- Länge des Arrays nur bei Allokation relevant
 - Wird beim Zugriff zur Laufzeit **nicht** geprüft
- Können bei Allokation auf dem Stack initialisiert werden

```
char arr1[5] = { 'a', 'b', 'c', 'd' };  
int arr2[10] = { 1, 2, 3 };
```

- Zu viele Elemente beim Initialisieren nicht erlaubt
- Zu wenige Elemente \Rightarrow Rest mit 0 aufgefüllt
- Initialisierungsliste bestimmt Arraygröße wenn nicht explizit angegeben

- Arrays sind *meist* identisch mit Zeigern

Ausname: sizeof Operator

```
sizeof(int *) == 4  
sizeof(int [5]) == 20
```

- Länge des Arrays nur bei Allokation relevant
 - Wird beim Zugriff zur Laufzeit **nicht** geprüft
- Können bei Allokation auf dem Stack initialisiert werden

```
char arr1[5] = { 'a', 'b', 'c', 'd' };  
int arr2[10] = { 1, 2, 3 };
```

- Zu viele Elemente beim Initialisieren nicht erlaubt
- Zu wenige Elemente \Rightarrow Rest mit 0 aufgefüllt
- Initialisierungsliste bestimmt Arraygröße wenn nicht explizit angegeben

- Arrays sind *meist* identisch mit Zeigern

Ausname: sizeof Operator

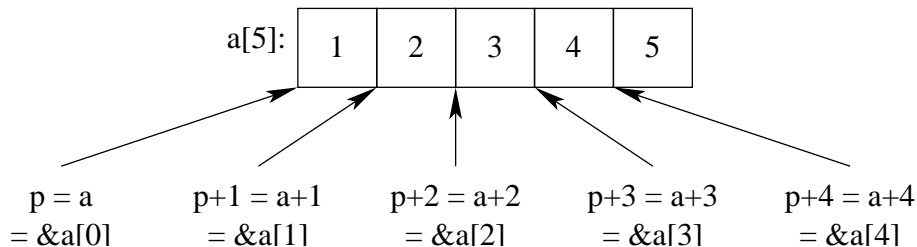
```
sizeof(int *) == 4  
sizeof(int [5]) == 20
```

- Länge des Arrays nur bei Allokation relevant
 - Wird beim Zugriff zur Laufzeit **nicht** geprüft
- Können bei Allokation auf dem Stack initialisiert werden

```
char arr1[5] = { 'a', 'b', 'c', 'd' };  
int arr2[10] = { 1, 2, 3 };
```

- Zu viele Elemente beim Initialisieren nicht erlaubt
- Zu wenige Elemente \Rightarrow Rest mit 0 aufgefüllt
- Initialisierungsliste bestimmt Arraygröße wenn nicht explizit angegeben

Array vs. Zeiger – Beispiel



- 1 Einführung
- 2 Compiler
 - Compile-Prozess
 - Compiler
- 3 Zeiger & Arrays
 - Grundlagen
 - Arrays
- 4 Speicherverwaltung**
 - Arten von Speicher
 - Funktionen
 - Tipps & Hinweise
- 5 Präprozessor & Macros
- 6 Aufgaben

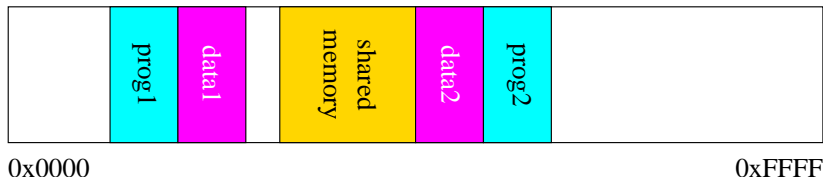
- Hauptspeicher (RAM) ist (meist) **linear** und ein zusammenhängender Bereich
 - Muss unterteilt/strukturiert werden
 - Nicht-linearer Speicher (NUMA) wird vom Betriebssystem verwaltet
- Alle Informationen (Daten und Programme) müssen im Hauptspeicher abgelegt werden
- Virtualisierung von Speicher
 - Erweiterung des physisch vorhandenen Hauptspeichers um Paging/Swap Space
 - Betriebssystem kümmert sich um Paging/Swapping

- Hauptspeicher (RAM) ist (meist) **linear** und ein zusammenhängender Bereich
 - Muss unterteilt/strukturiert werden
 - Nicht-linearer Speicher (NUMA) wird vom Betriebssystem verwaltet
- Alle Informationen (Daten und Programme) müssen im Hauptspeicher abgelegt werden
- Virtualisierung von Speicher
 - Erweiterung des physisch vorhandenen Hauptspeichers um Paging/Swap Space
 - Betriebssystem kümmert sich um Paging/Swapping

- Hauptspeicher (RAM) ist (meist) **linear** und ein zusammenhängender Bereich
 - Muss unterteilt/strukturiert werden
 - Nicht-linearer Speicher (NUMA) wird vom Betriebssystem verwaltet
- Alle Informationen (Daten und Programme) müssen im Hauptspeicher abgelegt werden
- Virtualisierung von Speicher
 - Erweiterung des physisch vorhandenen Hauptspeichers um Paging/Swap Space
 - Betriebssystem kümmert sich um Paging/Swapping

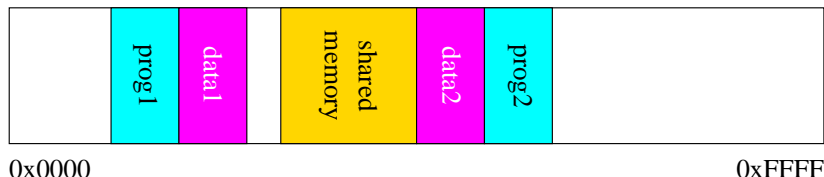
Speicherverwaltung

- Programme können parallel laufen
 - Jedes Programm ist unabhängig von anderen
 - Jede Variable in einem Programm muss in einem zuvor reservierten Speicherbereich abgelegt werden!
 - Speicher muss Programm zugeteilt bzw. von diesem reserviert werden



- Betriebssystem verwaltet, welches Programm welchen Speicherbereich verwendet
 - Speicher wird in Seiten aufgeteilt
 - Indirekte Verwaltung der Seiten

- Programme können parallel laufen
 - Jedes Programm ist unabhängig von anderen
 - Jede Variable in einem Programm muss in einem zuvor reservierten Speicherbereich abgelegt werden!
 - Speicher muss Programm zugeteilt bzw. von diesem reserviert werden



- Betriebssystem verwaltet, welches Programm welchen Speicherbereich verwendet
 - Speicher wird in Seiten aufgeteilt
 - Indirekte Verwaltung der Seiten

Arten von Speicher in einem Programm

1 Statischer Speicher

- Globale Variablen
- Statische Variablen in Funktionen
- Laufzeitumgebung, z. B. Ein-/Ausgabestreams (`stdin/stdout/stderr`)
- Speicher direkt beim Programmstart allokiert; existiert bis zum Programmende

2 Automatischer Speicher

- Funktionsargumente & lokale Variablen
- ⇒ Stack

3 Dynamischer Speicher

- Dynamisch allokierte und freigegebene Speicherbereiche
- ⇒ Heap

4 Konstanter Speicher

- Nicht-änderbare Daten (Konstanten)
- ⇒ *Data segment* im Maschinencode

Arten von Speicher in einem Programm

1 Statischer Speicher

- Globale Variablen
- Statische Variablen in Funktionen
- Laufzeitumgebung, z. B. Ein-/Ausgabestreams (`stdin/stdout/stderr`)
- Speicher direkt beim Programmstart allokiert; existiert bis zum Programmende

2 Automatischer Speicher

- Funktionsargumente & lokale Variablen
- ⇒ Stack

3 Dynamischer Speicher

- Dynamisch allokierte und freigegebene Speicherbereiche
- ⇒ Heap

4 Konstanter Speicher

- Nicht-änderbare Daten (Konstanten)
- ⇒ *Data segment* im Maschinencode

Arten von Speicher in einem Programm

1 Statischer Speicher

- Globale Variablen
- Statische Variablen in Funktionen
- Laufzeitumgebung, z. B. Ein-/Ausgabestreams (`stdin/stdout/stderr`)
- Speicher direkt beim Programmstart allokiert; existiert bis zum Programmende

2 Automatischer Speicher

- Funktionsargumente & lokale Variablen
- ⇒ Stack

3 Dynamischer Speicher

- Dynamisch allokierte und freigegebene Speicherbereiche
- ⇒ Heap

4 Konstanter Speicher

- Nicht-änderbare Daten (Konstanten)
- ⇒ *Data segment* im Maschinencode

Arten von Speicher in einem Programm

1 Statischer Speicher

- Globale Variablen
- Statische Variablen in Funktionen
- Laufzeitumgebung, z. B. Ein-/Ausgabestreams (`stdin/stdout/stderr`)
- Speicher direkt beim Programmstart allokiert; existiert bis zum Programmende

2 Automatischer Speicher

- Funktionsargumente & lokale Variablen
- ⇒ Stack

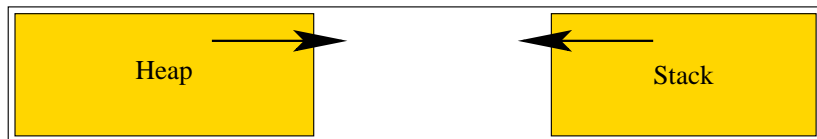
3 Dynamischer Speicher

- Dynamisch allokierte und freigegebene Speicherbereiche
- ⇒ Heap

4 Konstanter Speicher

- Nicht-änderbare Daten (Konstanten)
- ⇒ *Data segment* im Maschinencode

Stack vs. Heap



Gesamter Speicher, der einem Programm zur Verfügung steht

- Stack und Heap **teilen** sich gesamten zur Verfügung stehenden Speicher
 - Pro Prozess
 - Bereiche wachsen gegeneinander

Heap: dynamisch benötigter Speicher

Stack: statisch verwendeter Speicher

- Für alle statischen Informationen während des Programmablaufs
 - Wird zur Laufzeit allokiert – nicht beim Starten!
- Aufrufinformationen
 - Welche Funktion wurde von wo aufgerufen
 - Parameter der aufgerufenen Funktion
 - Rücksprungadresse
 - Lokale Variablen
- Lokale Variablen in der Funktion
 - Z. B. Genügend Speicher für 50 „int“ Werte: `int a[50]`
- Automatisch beim Eintritt in den Block/Funktion allokiert
- Automatisch beim Verlassen des Blocks/der Funktion aufgeräumt, d. h. der Speicher wird wieder freigegeben
- Jeder Block/Funktion hat eigene Kopie (*reentrant*) des Stacks

- Andere Programmiersprachen
 - **C++:** Destruktor von Objekten wird aufgerufen + Speicher wird freigegeben
 - **Java:** Nur dynamischer Speicher für Objekte
 - Lediglich Referenzen auf dem Stack
 - Reference count wird reduziert
 - ⇒ Garbage Collector räumt Objekte selbst auf
- Allokation auf dem Stack ist schneller/performanter als Heap
 - Stack kann nicht fragmentieren
 - Keine Suche nach freien Speicher nötig

Speicher auf dem Stack ist **nicht initialisiert.**

- Andere Programmiersprachen
 - **C++:** Destruktor von Objekten wird aufgerufen + Speicher wird freigegeben
 - **Java:** Nur dynamischer Speicher für Objekte
 - Lediglich Referenzen auf dem Stack
 - Reference count wird reduziert
 - ⇒ Garbage Collector räumt Objekte selbst auf
- Allokation auf dem Stack ist schneller/performanter als Heap
 - Stack kann nicht fragmentieren
 - Keine Suche nach freien Speicher nötig

Speicher auf dem Stack ist **nicht initialisiert.**

- Andere Programmiersprachen
 - **C++:** Destruktor von Objekten wird aufgerufen + Speicher wird freigegeben
 - **Java:** Nur dynamischer Speicher für Objekte
 - Lediglich Referenzen auf dem Stack
 - Reference count wird reduziert
 - ⇒ Garbage Collector räumt Objekte selbst auf
- Allokation auf dem Stack ist schneller/performer als Heap
 - Stack kann nicht fragmentieren
 - Keine Suche nach freien Speicher nötig

Speicher auf dem Stack ist **nicht initialisiert**.

- Für dynamisch allokierten Speicher
- Größe kann nach Bedarf variieren
- Funktion `malloc` fordert Speicherblock an
 - Passender Block muss im Heap gesucht, reserviert und zurückgegeben werden
- Funktion `free` gibt zuvor angeforderten Speicherblock wieder frei
- Speicher des Heaps kann fragmentieren
 - Heap und einzelne Speicherblöcke üblicherweise vom Betriebssystem (*operating system*) verwaltet
 - Manche Programme verwenden eigene Speicherverwaltung
 - Bessere Strukturierung (*memory pools*)
 - Gezielte Kontrolle des gesamten Speichers möglich

Speicher auf dem Heap ist **nicht initialisiert.**

- Für dynamisch allokierten Speicher
- Größe kann nach Bedarf variieren
- Funktion `malloc` fordert Speicherblock an
 - Passender Block muss im Heap gesucht, reserviert und zurückgegeben werden
- Funktion `free` gibt zuvor angeforderten Speicherblock wieder frei
- Speicher des Heaps kann fragmentieren
 - Heap und einzelne Speicherblöcke üblicherweise vom Betriebssystem (*operating system*) verwaltet
 - Manche Programme verwendenen eigene Speicherverwaltung
 - Bessere Strukturierung (*memory pools*)
 - Gezielte Kontrolle des gesamten Speichers möglich

Speicher auf dem Heap ist **nicht initialisiert.**

- Für dynamisch allokierten Speicher
- Größe kann nach Bedarf variieren
- Funktion `malloc` fordert Speicherblock an
 - Passender Block muss im Heap gesucht, reserviert und zurückgegeben werden
- Funktion `free` gibt zuvor angeforderten Speicherblock wieder frei
- Speicher des Heaps kann fragmentieren
 - Heap und einzelne Speicherblöcke üblicherweise vom Betriebssystem (*operating system*) verwaltet
 - Manche Programme verwenden eigene Speicherverwaltung
 - Bessere Strukturierung (*memory pools*)
 - Gezielte Kontrolle des gesamten Speichers möglich

Speicher auf dem Heap ist **nicht initialisiert.**

- Für dynamisch allokierten Speicher
- Größe kann nach Bedarf variieren
- Funktion `malloc` fordert Speicherblock an
 - Passender Block muss im Heap gesucht, reserviert und zurückgegeben werden
- Funktion `free` gibt zuvor angeforderten Speicherblock wieder frei
- Speicher des Heaps kann fragmentieren
 - Heap und einzelne Speicherblöcke üblicherweise vom Betriebssystem (*operating system*) verwaltet
 - Manche Programme verwenden eigene Speicherverwaltung
 - Bessere Strukturierung (*memory pools*)
 - Gezielte Kontrolle des gesamten Speichers möglich

Speicher auf dem Heap ist **nicht initialisiert**.

Fordere einen Speicherblock vom Heap an

```
void *ptr = malloc(size);
```

- `ptr` ist ein **Zeiger** auf den Beginn des Speicherblocks
 - Ein Zeiger ist eine **Adresse** im Hauptspeicher
- `size` ist die Größe des Blocks in Anzahl von Bytes

Fordere einen Speicherblock vom Heap an

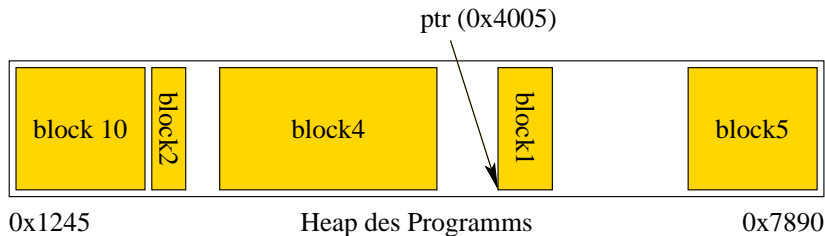
```
void *ptr = malloc(size);
```

- `ptr` ist ein **Zeiger** auf den Beginn des Speicherblocks
 - Ein Zeiger ist eine **Adresse** im Hauptspeicher
- `size` ist die Größe des Blocks in Anzahl von Bytes

Fordere einen Speicherblock vom Heap an

```
void *ptr = malloc(size);
```

- `ptr` ist ein **Zeiger** auf den Beginn des Speicherblocks
 - Ein Zeiger ist eine **Adresse** im Hauptspeicher
- `size` ist die Größe des Blocks in Anzahl von Bytes



- Gibt Zeiger vom Typ `void *` zurück
 - Typ der Daten, die im Speicherblock hinterlegt werden sollen ist `malloc` nicht bekannt
 - Typ muss mittels Cast umgewandelt werden

```
int *ptr = NULL;
...
ptr = (int *)malloc(size);
```

- Es darf **nicht** ausserhalb des allokierten Speicherblocks zugegriffen werden
 - Speicher könnte anderen Prozessen oder anderen Datenstrukturen des gleichen Prozesses gehören
- ⇒ Buffer Overflow
 - Fehler nicht erkannt oder *Segmentation Violation (SIGSEGV)*

- Gibt Zeiger vom Typ `void *` zurück
 - Typ der Daten, die im Speicherblock hinterlegt werden sollen ist `malloc` nicht bekannt
 - Typ muss mittels Cast umgewandelt werden

```
int *ptr = NULL;
...
ptr = (int *)malloc(size);
```

- Es darf **nicht** ausserhalb des allokierten Speicherblocks zugegriffen werden
 - Speicher könnte anderen Prozessen oder anderen Datenstrukturen des gleichen Prozesses gehören
- ⇒ Buffer Overflow
 - Fehler nicht erkannt oder *Segmentation Violation* (SIGSEGV)

- Ähnliche Systemfunktionen:

```
void *calloc(size_t nmemb, size_t size);
```

- Liefert Feld (Array) mit *nmemb* Elementen, wobei jedes Element *size* Bytes groß ist
- Vergleichbar mit `malloc(nmemb * size)`
- Speicherblock mit `0x00` initialisiert

- Andere Programmiersprachen verwenden ähnliche Operatoren

- Typisierung oft gleich mitgeliefert
- Cast ist nicht mehr nötig

```
C++: MyClass *object = new MyClass();
```

```
Java: MyClass object = new MyClass();
```

- Ähnliche Systemfunktionen:

```
void *calloc(size_t nmemb, size_t size);
```

- Liefert Feld (Array) mit *nmemb* Elementen, wobei jedes Element *size* Bytes groß ist
- Vergleichbar mit `malloc(nmemb * size)`
- Speicherblock mit `0x00` initialisiert

- Andere Programmiersprachen verwenden ähnliche Operatoren

- Typisierung oft gleich mitgeliefert
- Cast ist nicht mehr nötig

C++: `MyClass *object = new MyClass();`

Java: `MyClass object = new MyClass();`

- Angeforderte Speicherblöcke müssen wieder freigegeben werden

```
free(ptr);
```

- Sobald nicht mehr gebraucht
- Nur Zeiger verwenden, die `malloc` zurückgibt
 - Nur dynamisch allozierter Speicher
 - Zeiger auf Speicherblock darf bis zum `free` nicht verloren gehen!
 - Teile von Blöcken können nicht freigegeben werden
- Freigabe geschieht automatisch bei Programmende
 - Nicht darauf verlassen!
 - Viele Programme beenden sich nie
- Ein Block kann nur genau 1x freigegeben werden
 - Mehrfache `free` Operationen könnten einen falschen Block freigegeben oder zum Programmabsturz führen
- Block darf nach Freigabe nicht mehr verwendet werden

- Angeforderte Speicherblöcke müssen wieder freigegeben werden

```
free(ptr);
```

- Sobald nicht mehr gebraucht
- Nur Zeiger verwenden, die `malloc` zurückgibt
 - Nur dynamisch allozierter Speicher
 - Zeiger auf Speicherblock darf bis zum `free` nicht verloren gehen!
 - Teile von Blöcken können nicht freigegeben werden
- Freigabe geschieht automatisch bei Programmende
 - Nicht darauf verlassen!
 - Viele Programme beenden sich nie
- Ein Block kann nur genau 1x freigegeben werden
 - Mehrfache `free` Operationen könnten einen falschen Block freigegeben oder zum Programmabsturz führen
- Block darf nach Freigabe nicht mehr verwendet werden

- Angeforderte Speicherblöcke müssen wieder freigegeben werden

```
free(ptr);
```

- Sobald nicht mehr gebraucht
- Nur Zeiger verwenden, die `malloc` zurückgibt
 - Nur dynamisch allozierter Speicher
 - Zeiger auf Speicherblock darf bis zum `free` nicht verloren gehen!
 - Teile von Blöcken können nicht freigegeben werden
- Freigabe geschieht automatisch bei Programmende
 - Nicht darauf verlassen!
 - Viele Programme beenden sich nie
- Ein Block kann nur genau 1x freigegeben werden
 - Mehrfache `free` Operationen könnten einen falschen Block freigegeben oder zum Programmabsturz führen
- Block darf nach Freigabe nicht mehr verwendet werden

- Angeforderte Speicherblöcke müssen wieder freigegeben werden

```
free(ptr);
```

- Sobald nicht mehr gebraucht
- Nur Zeiger verwenden, die `malloc` zurückgibt
 - Nur dynamisch allozierter Speicher
 - Zeiger auf Speicherblock darf bis zum `free` nicht verloren gehen!
 - Teile von Blöcken können nicht freigegeben werden
- Freigabe geschieht automatisch bei Programmende
 - Nicht darauf verlassen!
 - Viele Programme beenden sich nie
- Ein Block kann nur genau 1x freigegeben werden
 - Mehrfache `free` Operationen könnten einen falschen Block freigegeben oder zum Programmabsturz führen
- Block darf nach Freigabe nicht mehr verwendet werden

- Freigegebener Block (oder ein Teil davon) kann beim nächsten `malloc` wieder vergeben werden
 - Speicherverwaltung im Betriebssystem
- Andere Programmiersprachen verwenden intern auch `free`:
 - C++: `delete`-Operator und/oder Garbage Collection
 - Java: Garbage Collection

- Freigegebener Block (oder ein Teil davon) kann beim nächsten `malloc` wieder vergeben werden
 - Speicherverwaltung im Betriebssystem
- Andere Programmiersprachen verwenden intern auch `free`:
 - C++:** delete-Operator und/oder Garbage Collection
 - Java:** Garbage Collection

- Vergrößern von Speicherblöcken ist nicht direkt möglich
 - Andere Speicherblöcke können physisch im Speicher dahinter liegen
- Neuer Speicherblock muss allokiert und Daten umkopiert werden

```
int *ptr = NULL;
int *newPtr = NULL;
ptr = (int *)malloc(init_size);
...
newPtr = (int *)realloc(ptr, new_size);
```

- Einfach Verlängerung des allokierten Blocks, wenn möglich
- Alter Speicherblock wird automatisch freigegeben

Anzahl der Bytes von Datentypen kann von Plattform zu Plattform variieren

- Z. B. unterschiedliche Größen von `int` Werten (32 vs. 64 Bit auf 32/64 Bit Prozessoren)
- Ermitteln der jeweiligen Größe

```
sizeof variable  
sizeof(datatype)
```

- Garantien des C Standards:

```
( sizeof(char) == 1 ) ^  
( sizeof(char) <= sizeof(smallint)  
<= sizeof(int) <= sizeof(long)  
<= sizeof(long long) )
```

- Wenn möglich, auf `<stdint.h>` zurückgreifen

Anzahl der Bytes von Datentypen kann von Plattform zu Plattform variieren

- Z. B. unterschiedliche Größen von `int` Werten (32 vs. 64 Bit auf 32/64 Bit Prozessoren)
- Ermitteln der jeweiligen Größe

```
sizeof variable  
sizeof(datatype)
```

- Garantien des C Standards:

```
( sizeof(char) == 1 ) ^  
( sizeof(char) <= sizeof(smallint)  
<= sizeof(int) <= sizeof(long)  
<= sizeof(long long) )
```

- Wenn möglich, auf `<stdint.h>` zurückgreifen

Sizeof Operator – Arrays

- Arrays sind nicht identisch zu Zeigern für `sizeof`

```
int a[5] = 0 ;  
sizeof a == 20;  
sizeof &a[0] == 4;
```

- `sizeof` gibt Anzahl der Bytes des gesamten Arrays
- Compiler muss zur Compilezeit Größe des Arrays kennen
 - Array darf nicht als Zeiger interpretiert werden
- Größe eines einzelnen Elements nicht immer bekannt

```
sizeof array / sizeof array[0]
```

Sizeof Operator – Padding/Alignment

- Unterschiedliche Optimierungsstrategien der Compiler
- Unterschiedliche Mächtigkeit/Performance der Adressierungsbefehle des Prozessors
 - Zugriff auf Speicheradressen, die ein Vielfaches von 4 Bytes sind, oft schneller als Adressierung einzelner Bytes
 - Padding-Bytes können vom Compiler eingeschoben werden
- Genaue Größe von Strukturen oft nicht bekannt

```
struct s { char c; int i; };
```

- Ergebnis von `sizeof(struct s)` auf 32 bit System: ???
- ⇒ Programme sollten sich nie auf bestimmte Größe verlassen
- `sizeof` verwenden, oder
 - gezielt byte-weise operieren

Sizeof Operator – Padding/Alignment

- Unterschiedliche Optimierungsstrategien der Compiler
- Unterschiedliche Mächtigkeit/Performance der Adressierungsbefehle des Prozessors
 - Zugriff auf Speicheradressen, die ein Vielfaches von 4 Bytes sind, oft schneller als Adressierung einzelner Bytes
 - Padding-Bytes können vom Compiler eingeschoben werden
- Genaue Größe von Strukturen oft nicht bekannt

```
struct s { char c; int i; };
```

- Ergebnis von `sizeof(struct s)` auf 32 bit System: ???

⇒ Programme sollten sich nie auf bestimmte Größe verlassen

- `sizeof` verwenden, oder
- gezielt byte-weise operieren

Sizeof Operator – Padding/Alignment

- Unterschiedliche Optimierungsstrategien der Compiler
- Unterschiedliche Mächtigkeit/Performance der Adressierungsbefehle des Prozessors
 - Zugriff auf Speicheradressen, die ein Vielfaches von 4 Bytes sind, oft schneller als Adressierung einzelner Bytes
 - Padding-Bytes können vom Compiler eingeschoben werden
- Genaue Größe von Strukturen oft nicht bekannt

```
struct s { char c; int i; };
```

- Ergebnis von `sizeof(struct s)` auf 32 bit System: ???

⇒ Programme sollten sich nie auf bestimmte Größe verlassen

- `sizeof` verwenden, oder
- gezielt byte-weise operieren

Sizeof Operator – Padding/Alignment

- Unterschiedliche Optimierungsstrategien der Compiler
- Unterschiedliche Mächtigkeit/Performance der Adressierungsbefehle des Prozessors
 - Zugriff auf Speicheradressen, die ein Vielfaches von 4 Bytes sind, oft schneller als Adressierung einzelner Bytes
 - Padding-Bytes können vom Compiler eingeschoben werden
- Genaue Größe von Strukturen oft nicht bekannt

```
struct s { char c; int i; };
```

- Ergebnis von `sizeof(struct s)` auf 32 bit System: **8 Bytes**

⇒ Programme sollten sich nie auf bestimmte Größe verlassen

- `sizeof` verwenden, oder
- gezielt byte-weise operieren

Sizeof Operator – Padding/Alignment

- Unterschiedliche Optimierungsstrategien der Compiler
- Unterschiedliche Mächtigkeit/Performance der Adressierungsbefehle des Prozessors
 - Zugriff auf Speicheradressen, die ein Vielfaches von 4 Bytes sind, oft schneller als Adressierung einzelner Bytes
 - Padding-Bytes können vom Compiler eingeschoben werden
- Genaue Größe von Strukturen oft nicht bekannt

```
struct s { char c; int i; };
```

- Ergebnis von `sizeof(struct s)` auf 32 bit System: 8 Bytes
- ⇒ Programme sollten sich nie auf bestimmte Größe verlassen
- `sizeof` verwenden, oder
 - gezielt byte-weise operieren

Sizeof Operator – Verwendung

- `sizeof` berechnet Größe eines Wertes oder eines Datentyps

```
size = sizeof(int);  
size = sizeof(struct myStruct);  
size = sizeof myValue;
```

- Klammern bei Datentypen notwendig
- Variablen brauchen nicht geklammert zu werden und sollten es auch nicht
 - Compiler kann besser Fehler erkennen und melden
- Anzahl an Bytes, die die physisch Repräsentation im Speicher benötigt
- `ptr = (int *)malloc(N * sizeof(int));`
- Kann bereits zur Übersetzungszeit errechnet werden und belastet somit nicht die Laufzeit

Sizeof Operator – Verwendung

- `sizeof` berechnet Größe eines Wertes oder eines Datentyps

```
size = sizeof(int);  
size = sizeof(struct myStruct);  
size = sizeof myValue;
```

- Klammern bei Datentypen notwendig
- Variablen brauchen nicht geklammert zu werden und sollten es auch nicht
 - Compiler kann besser Fehler erkennen und melden
- Anzahl an Bytes, die die physisch Repräsentation im Speicher benötigt

```
ptr = (int *)malloc(N * sizeof(int));
```

- Kann bereits zur Übersetzungszeit errechnet werden und belastet somit nicht die Laufzeit

Sizeof Operator – Verwendung

- `sizeof` berechnet Größe eines Wertes oder eines Datentyps

```
size = sizeof(int);  
size = sizeof(struct myStruct);  
size = sizeof myValue;
```

- Klammern bei Datentypen notwendig
 - Variablen brauchen nicht geklammert zu werden und sollten es auch nicht
 - Compiler kann besser Fehler erkennen und melden
 - Anzahl an Bytes, die die physisch Repräsentation im Speicher benötigt
- ```
ptr = (int *)malloc(N * sizeof(int));
```
- Kann bereits zur Übersetzungszeit errechnet werden und belastet somit nicht die Laufzeit

# memset, memcpy, und memmove

## memset

- **Angeforderte Speicherblöcke (malloc) immer initialisieren!**

```
memset(ptr, 0x00, size);
```

- Bei sicherheitskritischen Anwendung *Leeren* vor dem Freigeben

## memcpy

- Kopieren zwischen überlappungsfreien Speicherblöcken

```
memcpy(destination, source, size);
```

## memmove

- Kopieren zwischen überlappenden oder überlappungsfreien Speicherblöcken

```
memmove(destination, source, size);
```

- Nicht so effizient wie memcpy

# memset, memcpy, und memmove

## memset

- Angeforderte Speicherblöcke (`malloc`) immer initialisieren!

```
memset(ptr, 0x00, size);
```

- Bei sicherheitskritischen Anwendung *Leeren* vor dem Freigeben

## memcpy

- Kopieren zwischen **überlappungsfreien** Speicherblöcken

```
memcpy(destination, source, size);
```

## memmove

- Kopieren zwischen überlappenden oder überlappungsfreien Speicherblöcken

```
memmove(destination, source, size);
```

- Nicht so effizient wie `memcpy`

# memset, memcpy, und memmove

## memset

- Angeforderte Speicherblöcke (`malloc`) immer initialisieren!

```
memset(ptr, 0x00, size);
```

- Bei sicherheitskritischen Anwendung *Leeren* vor dem Freigeben

## memcpy

- Kopieren zwischen überlappungsfreien Speicherblöcken

```
memcpy(destination, source, size);
```

## memmove

- Kopieren zwischen **überlappenden** oder überlappungsfreien Speicherblöcken

```
memmove(destination, source, size);
```

- Nicht so effizient wie `memcpy`

Legt Maximum von Ressourcen für einen Nutzer/Prozess fest:

- Maximal nutzbarer Speicherbereich (virtuell)
- Größe des Stacks
- Größe von „core“ Dateien
- Größe des „data segment“ eines Prozesses
- Größe von Dateien, die ein Prozess anlegen kann
- Anzahl der geöffneten Dateien
- Größe des ge„pin“ten Speicherbereichs
- Blockgröße bei Pipes
- Anzahl der Prozesse eines Nutzers
- CPU-Zeit

- Speicherallokation **kann fehlschlagen!!**
  - Ergebnis ist dann NULL-Zeiger
  - ⇒ Ergebnis muss **immer** überprüft werden
- Speicherblock nach `free` nicht weiter verwenden
  - Am besten Zeiger auf NULL setzen:

```
free(ptr);
ptr = NULL;
```

- Kann mit Macros automatisch gemacht werden
- Alle Variablen sofort bei Deklaration initialisieren

```
char *str = NULL;

...
str = (char *)malloc(strlen(orig_str)+1);
if (!str) { ... }
memset(str, 0x00, strlen(orig_str)+1);
```

- Speicherallokation kann fehlschlagen!!
  - Ergebnis ist dann NULL-Zeiger
  - ⇒ Ergebnis muss **immer** überprüft werden
- Speicherblock nach `free` nicht weiter verwenden
  - Am besten Zeiger auf NULL setzen:

```
free(ptr);
ptr = NULL;
```

- Kann mit Macros automatisch gemacht werden
- Alle Variablen sofort bei Deklaration initialisieren

```
char *str = NULL;
...
str = (char *)malloc(strlen(orig_str)+1);
if (!str) { ... }
memset(str, 0x00, strlen(orig_str)+1);
```

- Speicherallokation kann fehlschlagen!!
  - Ergebnis ist dann NULL-Zeiger
  - ⇒ Ergebnis muss **immer** überprüft werden
- Speicherblock nach `free` nicht weiter verwenden
  - Am besten Zeiger auf NULL setzen:

```
free(ptr);
ptr = NULL;
```

- Kann mit Macros automatisch gemacht werden
- Alle Variablen sofort bei Deklaration initialisieren

```
char *str = NULL;
...
str = (char *)malloc(strlen(orig_str)+1);
if (!str) { ... }
memset(str, 0x00, strlen(orig_str)+1);
```

# Potentielle Probleme (1)

- Keine gute Kontrolle über Speicher
  - Keine Gruppierung von Speicher (*Pooling*)
  - Keine Fehlermeldung beim falschen Freigeben
    - ⇒ Programm stürzt eventuell ab
    - ⇒ Vielleicht auch unerwarteter Stelle
- Nur unzureichende Memory-Debugging Facilities
  - ⇒ Spezielle Bibliotheken oder eigene Implementierung nötig
- Buffer Overflow – Zugriff ausserhalb des aktuellen Speicherblocks – verhindern
  - In Programmen oft keine/unzureichende Prüfung von Überläufen
  - Kann sich zu Sicherheitsproblemen ausweiten

# Potentielle Probleme (1)

- Keine gute Kontrolle über Speicher
  - Keine Gruppierung von Speicher (*Pooling*)
  - Keine Fehlermeldung beim falschen Freigeben
    - ⇒ Programm stürzt eventuell ab
    - ⇒ Vielleicht auch unerwarteter Stelle
- Nur unzureichende Memory-Debugging Facilities
  - ⇒ Spezielle Bibliotheken oder eigene Implementierung nötig
- Buffer Overflow – Zugriff ausserhalb des aktuellen Speicherblocks – verhindern
  - In Programmen oft keine/unzureichende Prüfung von Überläufen
  - Kann sich zu Sicherheitsproblemen ausweiten

# Potentielle Probleme (1)

- Keine gute Kontrolle über Speicher
  - Keine Gruppierung von Speicher (*Pooling*)
  - Keine Fehlermeldung beim falschen Freigeben
    - ⇒ Programm stürzt eventuell ab
    - ⇒ Vielleicht auch unerwarteter Stelle
- Nur unzureichende Memory-Debugging Facilities
  - ⇒ Spezielle Bibliotheken oder eigene Implementierung nötig
- Buffer Overflow – Zugriff ausserhalb des aktuellen Speicherblocks – verhindern
  - In Programmen oft keine/unzureichende Prüfung von Überläufen
  - Kann sich zu Sicherheitsproblemen ausweiten

## Potentielle Probleme (2)

- Memory Leaks – angeforderter Speicher wird nie freigegeben
  - Zeiger auf Speicherblock ist verloren gegangen
  - Unbedarftheit des Programmierers
- Funktionen dürfen keinen Zeiger in den Stack zurückgeben
  - Informationen auf dem Stack sind beim Verlassen der Funktion nicht mehr gültig
- `free` auf Objekt auf dem Stack nicht zulässig
  - Führt meist zum Absturz des Programms
  - Irgendwann, irgendwo
- Speicheroperationen kosten Zeit
  - Wenn möglich, mehrere Informationen in einen gemeinsamen Speicherblock legen

## Potentielle Probleme (2)

- Memory Leaks – angeforderter Speicher wird nie freigegeben
  - Zeiger auf Speicherblock ist verloren gegangen
  - Unbedarftheit des Programmierers
- Funktionen dürfen keinen Zeiger in den Stack zurückgeben
  - Informationen auf dem Stack sind beim Verlassen der Funktion nicht mehr gültig
- `free` auf Objekt auf dem Stack nicht zulässig
  - Führt meist zum Absturz des Programms
  - Irgendwann, irgendwo
- Speicheroperationen kosten Zeit
  - Wenn möglich, mehrere Informationen in einen gemeinsamen Speicherblock legen

# Potentielle Probleme (2)

- Memory Leaks – angeforderter Speicher wird nie freigegeben
  - Zeiger auf Speicherblock ist verloren gegangen
  - Unbedarftheit des Programmierers
- Funktionen dürfen keinen Zeiger in den Stack zurückgeben
  - Informationen auf dem Stack sind beim Verlassen der Funktion nicht mehr gültig
- `free` auf Objekt auf dem Stack nicht zulässig
  - Führt meist zum Absturz des Programms
  - Irgendwann, irgendwo
- Speicheroperationen kosten Zeit
  - Wenn möglich, mehrere Informationen in einen gemeinsamen Speicherblock legen

## Potentielle Probleme (2)

- Memory Leaks – angeforderter Speicher wird nie freigegeben
  - Zeiger auf Speicherblock ist verloren gegangen
  - Unbedarftheit des Programmierers
- Funktionen dürfen keinen Zeiger in den Stack zurückgeben
  - Informationen auf dem Stack sind beim Verlassen der Funktion nicht mehr gültig
- `free` auf Objekt auf dem Stack nicht zulässig
  - Führt meist zum Absturz des Programms
  - Irgendwann, irgendwo
- Speicheroperationen kosten Zeit
  - Wenn möglich, mehrere Informationen in einen gemeinsamen Speicherblock legen

- 1 Einführung
- 2 Compiler
  - Compile-Prozess
  - Compiler
- 3 Zeiger & Arrays
  - Grundlagen
  - Arrays
- 4 Speicherverwaltung
  - Arten von Speicher
  - Funktionen
  - Tipps & Hinweise
- 5 Präprozessor & Macros**
- 6 Aufgaben

- Plattformspezifische Definitionen?
- Wie Header-Dateien einbinden?
- Ursprünglich *keine* Inline-Funktionen – Work-around?

## Macros und Präprozessor-Direktiven

- Einfache Textersetzung
  - Parameterisiert vs. nicht-parameterisiert
  - C Präprozessor ersetzt Vorkommen der Macros mit entsprechendem Text
    - Single sweep
- Nicht zur Definition von Datentypen verwenden
  - `typedef` wesentlich geeigneter!

- Plattformspezifische Definitionen?
- Wie Header-Dateien einbinden?
- Ursprünglich *keine* Inline-Funktionen – Work-around?

## Macros und Präprozessor-Direktiven

- Einfache Textersetzung
  - Parameterisiert vs. nicht-parameterisiert
  - C Präprozessor ersetzt Vorkommen der Macros mit entsprechendem Text
    - Single sweep
- Nicht zur Definition von Datentypen verwenden
  - `typedef` wesentlich geeigneter!

- Plattformspezifische Definitionen?
- Wie Header-Dateien einbinden?
- Ursprünglich *keine* Inline-Funktionen – Work-around?

## Macros und Präprozessor-Direktiven

- Einfache Textersetzung
  - Parameterisiert vs. nicht-parameterisiert
  - C Präprozessor ersetzt Vorkommen der Macros mit entsprechendem Text
    - Single sweep
- Nicht zur Definition von Datentypen verwenden
  - `typedef` wesentlich geeigneter!

- Plattformspezifische Definitionen?
- Wie Header-Dateien einbinden?
- Ursprünglich *keine* Inline-Funktionen – Work-around?

## Macros und Präprozessor-Direktiven

- Einfache Textersetzung
  - Parameterisiert vs. nicht-parameterisiert
  - C Präprozessor ersetzt Vorkommen der Macros mit entsprechendem Text
    - Single sweep
- Nicht zur Definition von Datentypen verwenden
  - `typedef` wesentlich geeigneter!

# Definition von Macros

```
#define macro ersetzung
#define macro(parameterliste) ersetzung
```

- Wörtliche Ersetzung von `<macro>` mit `<ersetzung>`
- Parameterliste wird gegen Argumente beim Aufruf des Macros ausgetauscht
- Ersetzung kann leer sein
  - Entfernt Macro aus dem Quelltext
  - Kann gut für Tracing/Debugging eingesetzt werden
- Löschen von Macros

```
#undef macro
```

# Definition von Macros – Beispiel

```
#define MAX(a, b) ((a) > (b) ? (a) : (b))
...
res = MAX(in1, in2);
```

```
res = ((in1) > (in2) ? (in1) : (in2));
```

- Macros können geschachtelt werden
  - Macro A kann Macro B in der Ersetzung verwenden

# Definition von Macros – Beispiel

```
#define MAX(a, b) ((a) > (b) ? (a) : (b))
...
res = MAX(in1, in2);
```

```
res = ((in1) > (in2) ? (in1) : (in2));
```

- Macros können geschachtelt werden
  - Macro A kann Macro B in der Ersetzung verwenden

# Definition von Macros – Beispiel

```
#define MAX(a, b) ((a) > (b) ? (a) : (b))
...
res = MAX(in1, in2);
```

```
res = ((in1) > (in2) ? (in1) : (in2));
```

- Macros können geschachtelt werden
  - Macro A kann Macro B in der Ersetzung verwenden

## 1 Einbinden von Header-Dateien

```
#include "datei"
#include <datei>
```

- Transitives Einbinden möglich

## 2 Header-Datei kann mehrfach eingebunden werden

- Meist nicht erwünscht
- ⇒ Doppelte Deklarationen

## 3 Macros zum Unterbinden mehrfachen Einbindens

```
#if !defined(__DATEI_H__)
#define __DATEI_H__
...
#endif /* __DATEI_H__ */
```

## 1 Einbinden von Header-Dateien

```
#include "datei"
#include <datei>
```

- Transitives Einbinden möglich

## 2 Header-Datei kann mehrfach eingebunden werden

- Meist nicht erwünscht
- ⇒ Doppelte Deklarationen

## 3 Macros zum Unterbinden mehrfachen Einbindens

```
#if !defined(__DATEI_H__)
#define __DATEI_H__
...
#endif /* __DATEI_H__ */
```

## 1 Einbinden von Header-Dateien

```
#include "datei"
#include <datei>
```

- Transitives Einbinden möglich

## 2 Header-Datei kann mehrfach eingebunden werden

- Meist nicht erwünscht
- ⇒ Doppelte Deklarationen

## 3 Macros zum Unterbinden mehrfachen Einbindens

```
#if !defined(__DATEI_H__)
#define __DATEI_H__
...
#endif /* __DATEI_H__ */
```

## 4 Bedingungen

```
#if ...
...
#elif
...
#else
...
#endif
```

- Nur mit Konstanten oder Macros – kein C Code!
- Können logische Verknüpfungen (|| und &&) verwenden

## 5 Macros in Bedingungen

Test ob Macro definiert? `#if defined(MACRO)`

- Alternativ: `#ifdef MACRO`  
⇒ Kann aber nur *einzelnes* Macro testen

Auswertung des Macro-Wertes `#if MACRO`

## 4 Bedingungen

```
#if ...
...
#elif
...
#else
...
#endif
```

- Nur mit Konstanten oder Macros – kein C Code!
- Können logische Verknüpfungen (|| und &&) verwenden

## 5 Macros in Bedingungen

**Test ob Macro definiert?** `#if defined(MACRO)`

- Alternativ: `#ifdef MACRO`  
⇒ Kann aber nur *einzelnes* Macro testen

**Auswertung des Macro-Wertes** `#if MACRO`

## 6 Warnungen und Fehler

```
#warning "meldung"
#error "meldung"
```

- Warnung wird ausgegeben, Präcompile-Vorgang wird fortgesetzt
  - Ausser bei Option `-Werror` (bei gcc)
- Fehlermeldung wird ausgegeben, Präcompile-Vorgang wird abgebrochen

- `__FILE__` Name der Übersetzungseinheit
- **Nicht** die Datei, in der das Macro genutzt wurde.
- `__LINE__` Zeile in der Übersetzungseinheit, in der das Macro auftaucht
- **Nicht** die Datei, in der das Macro genutzt wurde.
  - Auch bei geschachtelten Macros

# Quoting von Macro-Argumenten

- Macros werden *nicht* innerhalb von Strings expandiert
- Macro-Argumente können in Strings umgewandelt werden

```
#define QUOTE(x) #x
printf("%s\n", QUOTE(1+2));
```

```
printf("%s\n", "1+2");
```

# Quoting von Macro-Argumenten

- Macros werden *nicht* innerhalb von Strings expandiert
- Macro-Argumente können in Strings umgewandelt werden

```
#define QUOTE(x) #x
printf("%s\n", QUOTE(1+2));
```

```
printf("%s\n", "1+2");
```

# Probleme von Macros

```
#define MAX(a, b) ((a) > (b) ? (a) : (b))
```

## Reine Textersetzung

- Keine Typverifikation für `a` und `b`
  - signed vs. unsigned
  - Typen überhaupt vergleichbar?
- Klammerung nötig

```
c = MAX(d1 > 0 ? d1 : d2, b);
```

```
c = d1 > 0 ? d1 : d2 > b ? d1 > 0 ? d1 : d2 : b;
```

- Seiteneffekte

```
c = MAX(a++, b);
```

```
c = ((a++) > (b) ? (a++) : (b))
```

# Probleme von Macros

```
#define MAX(a, b) ((a) > (b) ? (a) : (b))
```

## Reine Textersetzung

- Keine Typverifikation für  $a$  und  $b$ 
  - signed vs. unsigned
  - Typen überhaupt vergleichbar?
- Klammerung nötig

```
c = MAX(d1 > 0 ? d1 : d2, b);
```

```
c = d1 > 0 ? d1 : d2 > b ? d1 > 0 ? d1 : d2 : b;
```

- Seiteneffekte

```
c = MAX(a++, b);
```

```
c = ((a++) > (b) ? (a++) : (b))
```

# Probleme von Macros

```
#define MAX(a, b) ((a) > (b) ? (a) : (b))
```

## Reine Textersetzung

- Keine Typverifikation für  $a$  und  $b$ 
  - signed vs. unsigned
  - Typen überhaupt vergleichbar?
- Klammerung nötig

```
c = MAX(d1 > 0 ? d1 : d2, b);
```

```
c = d1 > 0 ? d1 : d2 > b ? d1 > 0 ? d1 : d2 : b;
```

- Seiteneffekte

```
c = MAX(a++, b);
```

```
c = ((a++) > (b) ? (a++) : (b))
```

- 1 Einführung
- 2 Compiler
  - Compile-Prozess
  - Compiler
- 3 Zeiger & Arrays
  - Grundlagen
  - Arrays
- 4 Speicherverwaltung
  - Arten von Speicher
  - Funktionen
  - Tipps & Hinweise
- 5 Präprozessor & Macros
- 6 Aufgaben**

- Aktivieren der Compiler-Warnungen
- Bei selbstgeschriebenen Programmen sind alle Warnungen zu bearbeiten/zu beseitigen
  - Unterdrücken der Warnungen nicht zulässig
- Ausgaben sind mittels `printf` auf die Standardausgabe zu leiten
  - Beispiel in `simple.c` (siehe Aufgabe 1) zu finden

# Aufgabe 1

Übersetzen Sie das vorgegebene Programm `simple.c` auf Ihrem Rechner. Erzeugen Sie dabei Dateien, die

- (a) Preprozessor-Ausgabe,
- (b) Assembler-Code,
- (c) Objekt-Datei, und
- (d) Ausführbares Programm (erzeugt von der Objektdatei).

## Aufgabe 2

Schreiben Sie ein einzelnes Programm, welches folgende Speicherblöcke anlegt. Strukturieren Sie hierbei das Programm so, dass jede Speicheranforderung in einer eigenen Routine abgewickelt wird.

- (a) Dynamische Allokation von 123 Integer-Werten,
- (b) Allokation eines String für 67 Textzeichen auf dem Stack,
- (c) Eine globale Variable vom Typ `struct s { int i; char c; };`,
- (d) Legen Sie ein Array für 10 Strings (jeweils mit beliebiger Länge) an.

Schreiben Sie jeweils mindestens einen (passenden) Wert in den jeweiligen Speicherbereich.

# Aufgabe 3

Definieren Sie ein Macro, das den absoluten Wert einer Integer-Zahl ermittelt. Verwenden Sie dieses Macro in einem selbstgeschriebenen Programm, das eine (beliebige) Zahl akzeptiert, den absoluten Wert bildet und das Ergebnis auf der Standardausgabe ausgibt. Die vorgegebene Zahl ist mittels eines Macros als Compiler-Option beim Übersetzen des Programms zu übergeben.

# Aufgabe 4

Definieren Sie die folgenden drei Macros:

- (a) Zum Ermitteln der Länge eines Strings (ohne die Funktion `strlen` zu verwenden); d. h. implementieren Sie die Funktionalität selbst,
- (b) Zum Freigeben von zuvor angeforderten Speicher und gleichzeitig den übergebenen Zeiger auf `NULL` zu setzen,
- (c) Zum Umwandeln einer Variable in ihren String mit dem Variablennamen.

Das erste der Macros soll nur auf der Linux-Plattform wie beschrieben gültig sein (Macro `LINUX` ist definiert). Auf AIX ist das erste Macro auf `strlen` abzubilden, und auf allen anderen Plattformen ist ein Fehler von Präprozessor auszugeben.

# Aufgabe 5

Nutzen Sie die Macros aus Aufgabe 4 in einem selbstgeschriebenen Programm, das:

- (a) einen beliebigen, vorgegebenen String in einen dynamisch allokierten Puffer kopiert,
- (b) den Inhalt dieses Puffers ausgibt,
- (c) zu Debugging-Zwecken eine beliebige Variable (mit dessen Namen und Integer-Wert) auf der Standard-Ausgabe ausgibt,
- (d) den Puffer freigibt, und
- (e) den Zeiger-Wert auf die Standardausgabe ausgibt.

Testen Sie das Programm. Erzeugen Sie zusätzlich die Präprozessor-Ausgabe.