Einführung

Speicher

Mit einer n-Bit breiten Adresse können 2^n Zeilen, bzw. Bytes adressiert werden.

Adresse	Inhalt der Speicherzellen							
0	0	1	1	0	0	1	0	1
1	1	1	1	1	1	1	1	1
.	.	.	.	.	.	.	.	.
2^n-1	0	0	0	0	0	0	1	1

Das Byte und seine Vielfachen

Kilobyte: 2^{10} Bytes = 1024 Bytes
Megabyte: 2^{20} Bytes = 1024 kB = 1 048 576 Bytes
GigaByte: 2^{30} Bytes = 1024 MB = 1 073 741 824 Bytes
TeraByte: 2^{40} Bytes = 1024 GB = 1 099 511 627 776 Bytes

Maschinencode

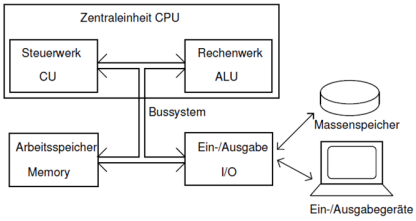
- Jeder Prozessor versteht ausschliesslich seine Maschinenbefehle
- Instruktionssatz: die Menge aller Maschinenbefehle eines Prozessors
- Maschinenbefehl: für jeden Befehl (z.B. Addition) wird ein eindeutiges Bitmuster definiert
- Ein Prozessor könnte direkt mit Maschinenbefehlen programmiert werden, die Nachteile davon sind jedoch 1. sehr unhandlich und 2. sehr unleserlich (man sieht nur eine Riesenmenge von Bitmustern)

Assemblerprogrammierung

- Jedem Maschinenbefehl wird ein aussagekräftiger Name (Mnemonic) zugewiesen
- Die Programme werden dadurch besser lesbar, die Programmierung ist aber immer noch mühsam
- Der Assembler übersetzt den Assemblercode in Maschinenbefehle

Grundlagen

Aufbau einfaches Rechnersystem



- Steuerwerk (Control Unit, CU)
- Rechenwerk (Arithmetic Logic Unit, ALU)
- Arbeitsspeicher (Memory)
- Ein. / Ausgabegeräte (Input/output, I/O)

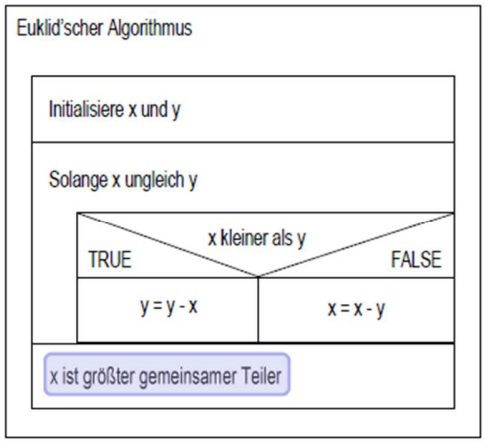
Übersetzungsvorgang

1. Kompilation (gcc –c hello.c) (Kompilieren ohne Linken)
    a. Präprozessor bearbeitet Quelltext und ersetzt #include <..> durch dessen Headerfiles.
    b. Eigentlicher Compiler übersetzt nun den Quellcode in Maschinencode macht .o Datei.
2. Das Linken (gcc –o hello hello.c)

Der Linker verbindet den in 1. Erzeugte Objektcode mit vorgefertigten Programmteilen, welche ebenfalls Maschinencode enthalten. Dazu gehören andere Objektdateien oder Bibliotheken.

Trace Tabelle und Struktogramm

Trace: Dient dazu den/die Werte von Variablen zu jedem Zeitpunkt zu dokumentieren.
Struktogramm: Dient dazu den Programmablauf zu Dokumentieren.



Verarbeitungsschritt	x	y
Initialisierung x = 24, y = 9	24	9
x = x – y	15	9
x = x – y	6	9
y = y – x	6	3
x = x – y	3	3
ggT ist: 3		

Sprunganweisungen

break: do while –, while –, for – Schleife und switch – Anweisung abbrechen
continue: in den nächsten Schleifendurchgang (Schleifenkopf) springen bei do while –, while – und for – Schleife
return: aus Funktion an aufrufende Stelle zurückspringen
goto: **innerhalb einer Funktion** an eine Marke (Label) springen

Datentypen und Variablen

Die Deklaration von Variablen legt nur Art und Typ der Variable, bzw. die Schnittstelle der Funktion fest ohne Speicherplatz zu reservieren.
Die Definition legt die Art und den Typ der Variablen bzw. Funktionen fest und reserviert Speicher.

Datentyp	Anzahl Bytes	Wertebereich (dezimal)[78]
`char`	1	-128 bis +127 oder 0 bis +255 (compilerabhängig)
`unsigned char`	1	0 bis +255
`signed char`	1	-128 bis +127
`unsigned short int`	2 in der Regel	0 bis +65 535
`short int`	2 in der Regel	-32 768 bis +32 767
`int`	4 in der Regel	-2 147 483 648 bis +2 147 483 647
`unsigned int`	4 in der Regel	0 bis +4 294 967 295
`long int`	4 in der Regel	-2 147 483 648 bis +2 147 483 647
`unsigned long int`	4 in der Regel	0 bis +4 294 967 295
`float`	4 in der Regel	$-3.4*10^{38}$ bis $+3.4*10^{38}$
`double`	8 in der Regel	$-1.7*10^{308}$ bis $+1.7*10^{308}$
`long double`	10 bis 16 in der Regel	implementationsabhängig

Pointer und Arrays

Pointer

Ein Pointer ist eine Variable, welche die Adresse einer im Speicher befindlichen Variablen oder Funktion aufnehmen kann.
Ein Nullpointer (int* ptr = 0;) muss nicht zwingend auf die Adresse 0 zeigen!

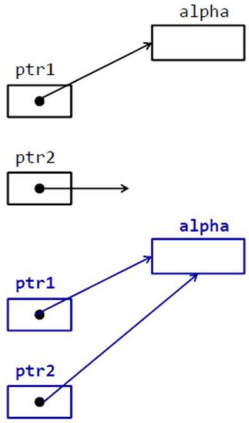
Darstellung von Pointern:

```
float alpha;

float* ptr1 = &alpha;

float* ptr2;

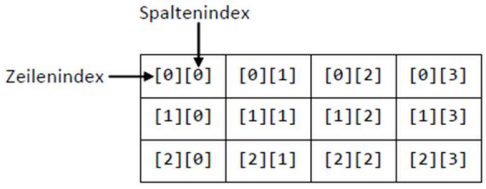
ptr2 = ptr1;
```



Mit *ptr1 = 23 wird 23 in die Variable alpha geschrieben

Array's

- Ein Array bietet eine kompakte Zusammenfassung von mehreren Variablen des gleichen Typs.
- Ein String ist in C ein Array von Zeichen (char-Array)
- Ein String muss in C immer mit dem Nullzeichen '\0' abgeschlossen werden. Dieses benötigt eine Stelle im Array!
- Globale Arrays werden automatisch mit 0 initialisiert
  - globale Arrays sollten aber nur ausnahmsweise verwendet werden
- Lokale Arrays werden nicht automatisch initialisiert
  - der Inhalt eines lokalen Arrays ist bei der Definition undefiniert
- Werden bei der Initialisierung weniger Werte angegeben als der Array Elemente hat, so werden die restlichen Elemente mit 0 belegt.
- Mehrdimensionale Array's



Const bei Pointer (konstanter String)

```
char str[] = "Ein String";
const char* text = str;          // text ist ein Pointer auf eine char-Konstante
char ch = text[1];               // erlaubt (== 'i')
text[1] = 's';                   // nicht erlaubt
text = "Ein anderer String";     // erlaubt
str[4] = 'A';                    // erlaubt
```

Const bei Pointer (konstanter Pointer)

```
char str[] = "Ein String";
char* const text = str;          // text ist ein konstanter Pointer auf ein char
char ch = text[1];               // erlaubt (== 'i')
text[1] = 's';                    // erlaubt
text = "Ein anderer String";     // nicht erlaubt
str[4] = 'A';                    // erlaubt
```

Const bei Pointer (konstanter Pointer auf konstanter String)

```
char str[] = "Ein String";
const char* const text = str;    // text ist ein konstanter Pointer auf eine char-Konstante"
char ch = text[1];                // erlaubt (== 'i')
text[1] = 's';                    // nicht erlaubt
text = "Ein anderer String";     // nicht erlaubt
str[4] = 'A';                    // erlaubt
```

const bei Pointern in Funktionsköpfen

```
void foo(const int* ptr)
{
  *ptr = 14; // nicht erlaubt
}
```

ptr ist ein Pointer auf eine int-Konstante
Wird verwendet bei read-only Pointerparametern

Beispiel 2 für Funktionspointer (crash.c)

```
// Datei: crash.c
#include <stdio.h>
int foo(void)
{
  return 0;
}

int main(void)
{
  int (*p)(void);
  p = foo;                                  //ermittle Adresse der Funktion foo()
  printf ("Adresse von foo: %p \n", p);
  *(int*)p = 17;                            // Schreibzugriff in Codebereich --> Crash
  return 0;
}
```

Lexikalische Konventionen

Schlüsselwörter

auto	break	case	char	const	continue	default	do	double	Else
enum	extern	float	for	goto	if	int	long	register	return
short	signed	sizeof	static	struct	switch	typedef	union	unsigned	void
volatile	while								

Anweisungen, Ausdücke und Operatoren

Operatoren und Operanden

unär:	hat einen einzigen Operanden, z.B. Minusoperator (Vorzeichenwechsel)
binär:	hat zwei Operanden
Postfix-Operatoren:	sind unäre Operatoren, die nach (post) ihrem Operanden stehen.
Beispiel:	printf("%d", i++); i auf Bildschirm, anschliessend inkrementiert.
Präfix-Operatoren:	sind unäre Operatoren, die vor (prä) ihrem Operanden stehen.
Beispiel:	printf("%d", ++i); i wird zuerst inkrementiert, dann auf Bildschirm geschrieben.

Auswertungsreihenfolge

1. Wie in der Mathematik werden als erstes **Teilausdrücke in Klammern** ausgewertet.
2. Dann werden **Ausdrücke mit unären Operatoren** ausgewertet. Unäre Operatoren werden *von rechts nach links* angewendet, d.h.
    2.1 **zuerst** werden die **Postfix-Operatoren** auf ihre Operanden
    2.2 und **dann** die **Präfix-Operatoren** auf ihre Operanden angewendet.
3. Abschliessend werden Teilausdrücke mit mehrstelligen Operatoren gemäss der **Priorität** der Operatoren ausgewertet.
4. Bei gleicher Priorität der Operatoren entscheidet die **Assoziativität** (von links nach rechts oder von rechts nach links)

Beispiel 1: *p++
Zuerst wird der Pointer p inkrementiert, der *Rückgabewert von p++ ist p!* Dieser Rückgabewert (das noch nicht inkrementierte p) wird nun dereferenziert.
Beispiel 2: *++p
Zuerst wird der Pointer inkrementiert, der Rückgabewert von ++p ist der inkrementierte Pointer, dann wird (das inkrementierte) p dereferenziert.
Beispiel 3: (*p)++
Zuerst wird der Pointer dereferenziert, anschliessend wird *p, d.h. der Inhalt der Speicherzelle, auf welche p zeigt, inkrementiert.

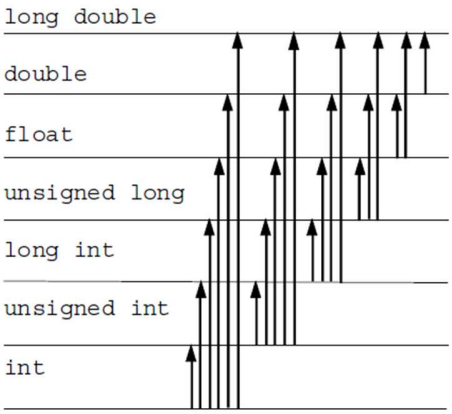
Scope
- Compiler arbeitet dateiweise
- Namen in einer anderen Datei sind dem Compiler nicht bekannt
- (Globale) Variablen, welche in einer anderen Datei definiert werden, können mit Hilfe des `extern`-Statements bekannt gemacht werden
- Durch das `extern`-Statement wird kein Speicherplatz reserviert.

```
extern int Foo_globalVariable;
```

- Funktionsprototypen und Definitionen, die von anderen Modulen genutzt werden können (Schnittstellen), werden in einer Headerdatei definiertb
- Durch `#include` der Headerdatei wird der Header geladen und die Namen somit bekannt gemacht.

Type cast (Typenumwandlungs-Operatoren)

Implizite Typumwandlung



Explizite Typumwandlung

- Nebst den impliziten (automatischen) Typumwandlungen kann mit Hilfe des cast-Operators eine explizite Typumwandlung bewirkt werden.
- **Der Programmierer ist verantwortlich, dass die Umwandlung keine Probleme ergibt.**
  (z.B. Umwandlung von grosser Zahl in kleineren Typ)
- Syntax:
  `(Zieltyp)Ausdruck`

Beispiel:

```
int a = (int)4.6;       // a == 4
double d = (double)a;   // d == 4.0
```

Strukturen und Union

Definition von Strukturtypen und Strukturvariablen
Verwenden des Schlüsselwortes struct
Allgemeine Form:

```
struct StructName
{
 FeldTyp1 feld1;
 FeldTyp2 feld2;
 FeldTyp3 feld3;
 ...
 FeldTypN feldN;
};
```

• StructName kann frei gewählt werden

• struct StructName ist ein hier selbst definierter Typ, der weiter verwendet werden kann

• Der Datentyp ist definiert durch den Inhalt der geschweiften Klammer

• Der Feldtyp kann wiederum eine Struktur sein

Zugriff auf eine Strukturvariable und deren Felder

Der Zugriff auf ein Feld einer Strukturvariablen erfolgt über

- den Namen der Strukturvariablen,gefolgt von einem Punkt und dem Namen des Feldes
- den Namen des Pointers, gefolgt von einem Pfeil (->) und dem Namen des Feldes

Beispiel:

```
struct Adresse
{
 char strasse[20];
 int hausnummer;
 int plz;
 char ort[20];
};

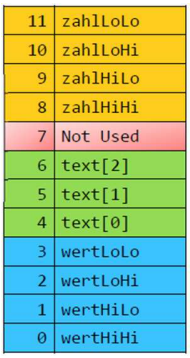
struct Angestellter
{
 int personalnummer;
 char name[20];
 char vorname[20];
 struct Adresse wohnort;
 struct Adresse arbeitsort;
 float gehalt;
};

struct Angestellter mueller;
struct Angestellter bonderer;
struct Angestellter vertrieb[20];
struct Angestellter* pMitarbeiter = &bonderer;
mueller.personalnummer = 34259;
bonderer.wohnort.plz = 7208;
strcpy(mueller.vorname, "Fritz");
printf("%s\n", vertrieb[14].name);
pMitarbeiter->personalnummer = 65433;          // einfache Form bei Pointer
(*pMitarbeiter).personalnummer = 65433;        // alternative Form
pMitarbeiter->arbeitsort.plz = 8640;
```

Lage im Speicher

- Die Felder einer Strukturvariablen werden nacheinander gemäss der Definition in den Speicher plaziert.
- Gewisse Datentypen verlangen unter Umständen (maschinenabhängig), dass Sie auf eine Wortgrenze (gerade Adresse) gelegt werden. Dies nennt man *Alignment*.
- Durch das Alignment kann es vorkommen, dass einzelne Bytes nicht verwendet werden, d.h. im Speicher ausgelassen werden.
- Die exakte Grösse einer Strukturvariablen kann nicht durch Addieren der Grössen der Felder ermittelt werden, nur sizeof() liefert den genauen Wert.

Beispiel für Alignment




```
struct B
{
 int wert;        // Das int Feld Zahl muss auf einer geraden
 char text[3];    // Adresse beginnen. Deshalb wird hier die
 int zahl;        // Adresse 7 nicht verwendet
};
```

Übergabe und Rückgabe von Strukturvariablen

- Strukturvariablen können komplett an Funktionen übergeben werden
- Der Rückgabetyp einer Funktion kann eine Struktur sein. Dabei wird die Strukturvariable direkt komplett übergeben
- Zu beachten ist der Kopieraufwand bei der Übergabe, bzw. Rückgabe eines Wertes. In der Praxis wird deshalb häufig mit Pointern gearbeitet.

```
void foo(struct Angestellter a);                    // grosser Kopieraufwand, nicht ideal

void foo(struct Angestellter* pa);                  // nur Pointerübergabe, effizient

void fooRead(const struct Angestellter* pa)         // nur Pointerübergabe, read only durch const
```

Initialisierung einer Strukturvariablen

```
struct Angestellter maier =
{
 56321,          // personalnummer
 "Maier",        // name[20]
 "Hans",         // vorname[20]
 {
  "Schillerplatz",      // strasse [20]
  14,                   // hausnummer
  75142,                // plz
  "Esslingen"           // ort[20]
 }
};
```

Komplexere Vereinbarungen und Datentypen

Array von Pointern

```
int* alpha[8];          //  [] hat Vorrang vor * //  alpha ist ein Array von 8 int-Pointern
```

Pointer auf ein Array

```
int (*alpha)[8];        // () hat immer Vorrang  // alpha ist ein Pointer auf ein Array von 8 int-Werten
int* (*beta)[8];        // beta ist ein Pointer auf ein Array von 8 int-Pointern
```

Funktion mit Pointer als Rückgabewert

```
int* foo(....);     // () hat immer Vorrang    //  foo ist eine Funktion, welche einen int-Pointer zurückgibt
```

Pointer auf eine Funktion

```
int (*pFunc)(....);     // () hat immer Vorrang, dann Assoziativität // pFunc ist ein Pointer auf eine
                        //Funktion, die ein int zurückgibt

int* (*pFoo)(....);     // pFoo ist ein Pointer auf eine Funktion, die einen int-Pointer zurückgibt
```

Typdef

- Vor allem bei komplexeren Typen (z.B. bei structs) macht es Sinn, einen eigenen (Alias-)Namen für den Typ zu definieren. Das Schlüsselwort struct kann anschliessend bei der Definition von Strukturvariablen weggelassen werden.
- Eigene Typen können mit dem Befehl typedef definiert werden.
- Zwischen dem Schlüsselwort struct und { wird üblicherweise kein zusätzlicher Name definiert.
- typedefs werden im Normalfall global, d.h. ausserhalb einer Funktion definiert.
- Konvention: eigene Typen werden mit einem Grossbuchstaben begonnen

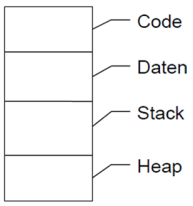
Beispiel:

```
typedef struct
{
  int x;
  int y;
} Point;

int main(void)
{
  Point p; // Definition einer Variablen des Typs Point
  p.x = 34;
  p.y = -25;
}
```

Speicherklassen

Speicherbereiche eines C- Programms



Codesegment: Zugriff: Lesen, ausführen
im Flash EPROM oder im RAM (z.B. beim PC)

Datensegment, Stack, Heap: Zugriff: lesen und schreiben
im RAM, evtl. in Register

Beschreibung der Speichersegmente

Code:	Programm in Maschinencode
Daten:	Globale Variablen, static Variablen
Stack:	Lokale Variablen, Parameter einer Funktion, Rücksprungadressen
Heap:	Dynamische Variablen (Speicherplatz wird erst zur Laufzeit angefordert)

Übersetzungsvorgang

- Quelldateien werden getrennt übersetzt
- Der Compiler erzeugt aus jeder Quelldatei eine Objektdatei mit Maschinencode
- Eine Objektdatei ist nicht ausführbar, da die Adressen der Funktionen (und globalen Variablen), die in einer anderen Datei definiert sind, noch nicht bekannt sind.
- Der Linker löst diese noch offenen Referenzen auf, indem er alle zum ausführbaren Programm gehörenden Objektdateien linked (deutsch: bindet).
- Der Linker meldet genau dann einen Fehler, wenn er eine noch offene Referenz nirgends gefunden hat, bzw. nicht auflösen konnte

Buildprozess

Der Buildprozess beinhaltet alle Schritte, um ein ausführbares Programm zu erhalten:

```
gcc -c foo.
gcc -c goo.c
gcc -c main.c
gcc -o prog.exe foo.o goo.o main.o
```

Externe Variablen

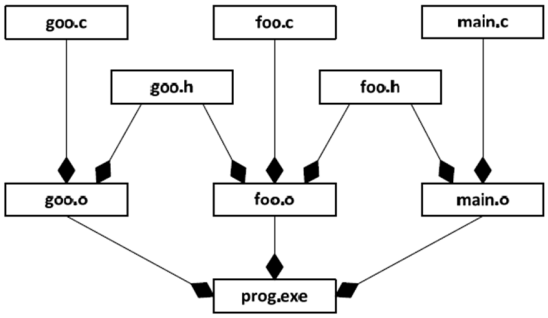
- Eine externe Variable kann nur in einer einzigen Datei definiert werden (ohne extern)
- In den anderen Dateien wird sie mit extern deklariert (bekannt gemacht)
- Eine manuelle Initialisierung ist nur bei der Definition möglich
- Globale Variablen, welche nicht manuell initialisiert werden, werden automatisch mit 0 initialisiert
- extern-Deklarationen werden üblicherweise in einer Headerdatei deklariert. Am Beginn der .c - Datei wird die Headerdatei mit #include eingefügt

static Variablen

- static Variablen sind im Datenbereich, nicht auf dem Stack
- Gültigkeitsbereich ist der Block, in dem die Variable definiert ist
- static Variablen, welche ausserhalb einer Funktion definiert sind (globale Variablen), sind nur in der Datei gültig, in der sie definiert werden
- static Variablen sind nur *einmal* vorhanden (auch in Multithreading- Umgebungen), d.h. ihr Wert wird erhalten, auch wenn die Funktion beendet ist. Beim nächsten Aufruf der Funktion geht es mit dem alten Wert weiter.

make-File

- In einem make-File können Abhängigkeiten definiert werden
- Wenn eine Datei geändert wurde, dann werden alle Operationen ausgeführt mit den Dateien, welche von dieser geänderten Datei abhängen
- Der Befehl (gcc) wird z.B. nur dann ausgeführt, wenn sich an den Dateien, zudenen eine Abhängigkeit besteht, etwas geändert hat




```
# makefile
cc=gcc
CFLAGS= -c -Wall
LFLAGS= -Wall
OBJ=foo.o goo.o main.o
EXE=prog.exe

${EXE}: ${OBJ}
        ${cc} ${LFLAGS} -o ${@} ${OBJ}

foo.o:  foo.h goo.h foo.c
        ${cc} ${CFLAGS} -o ${@} foo.c

goo.o:  goo.h goo.c
        ${cc} ${CFLAGS} -o ${@} goo.c

main.o: foo.h main.c
        ${cc} ${CFLAGS} -o ${@} main.c

clean:
        rm -f ${OBJ} ${EXE}
```

Targets: Diese können mit make angesprungen werden, z.B. make goo.o

Abhängigkeitsliste, d.h. von diesen Dateien ist das Target abhängig

Befehl, der ausgeführt werden muss, falls etwas geändert hat. Wichtig: erstes Zeichen muss ein TAB sein.