Betriebssysteme

Kapitel 1.2: Die Programmiersprache C
Grundstruktur, Typen und Typ-Deklarationen

Stand: WS 10/11

Prof. Dr. Wolfgang Küchlin

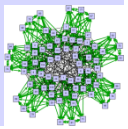
Dipl.-Inform., Dr. sc. techn. (ETH)

Arbeitsbereich Symbolisches Rechnen
Wilhelm-Schickard-Institut für Informatik
Mathematisch- Naturwissenschaftliche Fakultät

Universität Tübingen

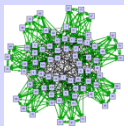
Steinbeis Transferzentrum
Objekt- und Internet-Technologien (OIT)

Wolfgang.Kuechlin@uni-tuebingen.de
http://www-sr.informatik.uni-tuebingen.de

SR 

Die Sprache C

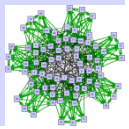
- Die Programmiersprache C wurde 1971 als Grundlage für das Betriebssystem UNIX in den USA entwickelt (UNIX ist zu über 90% in C geschrieben). 1978 wurde von Brian Kernighan und Dennis Ritchie eine eindeutige Sprachdefinition entwickelt. Mittlerweile ist C von ANSI und ISO standardisiert.
- C ist eine Hardware-unabhängige Sprache, die aber in der Ausführung der Effizienz von Assembler-Programmen sehr nahe kommt, oder diese mit hoch optimierenden Compilern – besonders auf RISC-Architekturen – noch übertrifft. Assembler-Befehle können in C-Programme eingebettet werden. Dies wird z.B. in den UNIX-Bibliotheksfunktionen oder in Linux Device-Drivers benutzt.
- Sehr viele Hardware-nahe Anwendungen sind heute in C geschrieben, auch eingebettete Systeme, die lange in Assembler programmiert werden mussten, da keine ausreichend leistungsfähigen Compiler zur Verfügung standen.
- Heute sind C und ihre Nachfolger C++ und Java die dominierenden Programmiersprachen. Jedes C Programm ist auch ein C++-Programm (ohne Klassen); Java ist in der Syntax an C angelehnt.
- B. Kernighan und D. Ritchie. The C Programming Language. Prentice Hall 1978.
- C Tutorial. Online unter http://www.rn-wissen.de/index.php/C-Tutorial



Von Java nach C

- Alle C Funktionen sind auf einer Sichtbarkeits-Ebene (keine geschachtelten Deklarationen), mit main() als Einstieg
- „Simulation“ in Java möglich mit Hilfsklasse „Program“ und Deklaration aller Funktionen als public static

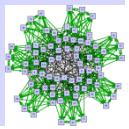
```
public class Program {
  public static
    int plus(int a, int b) {
      return (a+b);
  }
  public static void main(String[ ] args) {
    int x=5;
    int y=6;
    int z, v;
    z=plus(x,y);
    System.out.println(x + " + " + y + " = " + z);
    v=plus(plus(x,y),z);
    System.out.println("( " + x + " + " + y + ") + " + z + " = " + v);
  }
}
```



Formatierter Input / Output

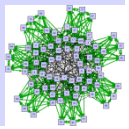
- Input durch scanf(char* format, arg1, arg2, …)
- Output durch printf (char* format, arg1, arg2, …)
- format ist ein zu druckender String mit integrierten Format-Anweisungen (z.B. %d: drucke eine Variable als Dezimalzahl)
  - printf(‘ ‘Dies ist der Wert von x: %d und dies der von y: %d.“, x, y);
- Einige Formate
  - d, i: Dezimalzahl
  - c: Zeichen
  - s: String
  - f: double

```
// Java: System.out.println(x + " + " + y + " = " + z);
printf( ‘ ‘%d + %d = %d“, x, y, z);
// Java: System.out.println("( " + x + " + " + y + ") + " + z + " = " + v);
printf( ‘ ‘(%d + %d) + %d = %d“, x, y, z, v);
```



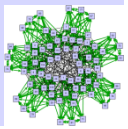
Typen und Typ-Deklarationen

- Abgeleitete Typen (derived types)
  - aus Basistypen können komplexere Typen abgeleitet werden
  - Zeiger (pointer), Reihungen (arrays), Verbunde (structures, records), Funktionstypen (function types)
- Sei T ein Typ.
  - mittels der **Deklarations-Operatoren** *, &, [ ], ( ) erhält man abgeleitete Typen
  - T* pointer to (object of type) T
  - T& reference of (object of type) T
  - T[ ] array of (objects of type) T
  - T( ) function returning (object of type) T



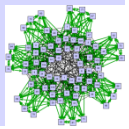
Typ-Deklarationen und Ausdrucks-Operatoren

- Den Deklarations-Operatoren *, &, [ ], ( ) für Typen entsprechen die **Ausdrucks-Operatoren**
  - * Dereferenzieren, Pointer-Inhalt
  - & Adresse (L-value) von
  - [ ] Array-Zugriff
  - ( ) Funktions-Aufruf
- Sei v eine Variable
  - Ist v vom Typ T*, so ist *v vom Typ T
  - Ist v vom Typ T[ ], so ist v[0] vom Typ T
  - Ist v vom Typ T( ), so ist v() vom Typ T
  - Ist pf ein Zeiger auf eine Funktion, so ist *pf die Funktion. (*pf)(x) ruft eine Funktion mit Argument x auf



Deklaration von Variablen von abgeleiteten Typen

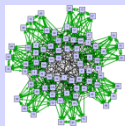
- Prinzip: Die Deklaration einer Variable vom Typ T (T x;) spiegelt die spätere Benutzung wider
- Vorgehensweise zur Typdeklaration
  - Sei B ein Basistyp. Eine Variable x erhält einen von B abgeleiteten Typ durch das **Typ-Deklarationsverfahren**
    1. Schreibe zunächst die Deklaration **B x;**
    2. Denke x als Variable vom gewünschten Typ. Dekoriere nun x solange mit den (Ausdrucks-)Operatoren *, &, [ ], ( ) bis der Basistyp herauskommt. (Der Basistyp einer Funktion ist der des Funktions-Ergebnisses).
    3. Das Resultat ist die richtige Deklaration von x, wenn man die Ausdrucks-Operatoren als Deklarations-Operatoren interpretiert.
    4. Lässt man den Variablennamen weg, hat man den Typ-Ausdruck
    5. Stellt man typedef davor, hat man statt der Variable einen Namen für den abgeleiteten Typ.



Deklaration von Variablen von abgeleiteten Typen

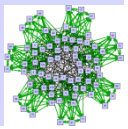
- Bemerkung:
  - * und & sind Präfix-Operatoren, [ ] und () sind Postfix-Operatoren. [ ] und () binden stärker als * und &. Im Zweifelsfall unbedingt Klammern schreiben!!
- Beispiele

Gwünschter Typ	Nach Verfahren	Empfohlene Schreibweise	Typ-Ausdruck
Pointer to int	int *pi	int* pi	int*
Pointer to array of 10 int	int (*pai)[10]	int (*pai)[10]	int (*)[10]
Array of 10 pointers to int	int *(api[10])	int* api[10]	int *[10]
Pointer to pointer to char	char *(*ppc)	char** ppc	char**
Function: int → int pointer	int *(ipfi(int))	int* ipfi(int)	int* (int)
Pointer to function: int → int	int (*pifi)(int)	int (*pifi)(int)	int (*)(int)



Reihungen und Zeiger

- Reihungen sind gegeben durch einen Zeiger auf den Anfang (also auf das Element mit Index 0).
  - der Typ int[ ] ist ein Synonym für den Typ int*
- Beispiel: Nach einer Deklaration int x[10] ist x vom Typ int*.
  - x[0] ist synonym für (wird umgewandelt in) *x.
  - x[1] ist synonym für *(x+1)
- Reihungsgrenzen können beim Zugriff nicht überwacht werden
  - ein direkter Zugriff über den Zeiger mit explizit angegebener Zeiger-Arithmetik ist immer möglich.
  - Beliebt: p=i; x = *p++ statt x=p[i]; i=i+1;



Structures (Verbunde)

- Eine structure ist ein Verbund verschiedener Daten
  - aber ohne direkt zugeordnete Methoden (keine Klassen)
- Beispiel:

```
struct tnode {
char* word;
int count;
struct tnode* left;
struct tnode* right;
}
```

- Typischer Zugriff auf Komponenten
  - struct tnode* p = new tnode;

    p = p → left; // Abkürzung für p = (*p).left