Programmiertechnik 1

Unit 8: Programmiersprache C –
Funktionen und Programmstruktur

Ablauf

- Grundlagen
- Prinzip Funktionsaufruf
- Stack
- call-by-value, call-by-ref, call-by-copy
- Rückgabewerte
- externe Variablen
- Scope - header files und Übersetzungseinheiten
- Initialisierung
- Rekursion
- C Präprozessor

Unterprogramme

- Funktionen
  - Zerlegen einer Berechnung in Teilprobleme
    - Zur Verbesserung der Lesbarkeit
    - Zur mehrfachen Verwendung im gleichen Programm (Vermeidung von Codedopplungen)
    - Zur Wiederverwendung in anderen Programmen (Bibliothek)
  - C: Deklaration vs. Definition
    - Deklaration: *Rückgabetyp Funktionsname* (*Argumentdeklarationen*);
    - Definition: *Rückgabetyp Funktionsname* (*Argumentdeklarationen*)

      {

      Deklarationen und Anweisungen

      }
  - Variablen, die innerhalb der Funktion belegt werden, sind nur bis zum Ende der Funktion gültig
    - Lokale Variablen
    - Sollen Variablen den Wert auch außerhalb der Funktion behalten, so muss der Funktion die Adresse der Variablen übergeben werden (call-by-value)

Unterprogramme - Beispiel

```
# include <stdio.h>

void print_sterne ( int anzahl ) {
    int i;
    for (i = 0; i < anzahl; i++)
      printf(”*”);
}
...
for (zeile = 1; zeile <= 5; zeile++) {
    print_sterne( zeile );
    printf(”\n”);
}
```

Prozedurale Abstraktion

- Zerlegung des Programms in Teilschritte mithilfe von Prozeduren: prozedurale Abstraktion
  - Unterprogramme, die „wie Anweisungen“ verwendet werden
  - Prozeduren liefern kein Ergebnis
- Anders als Pascal unterscheidet C nicht zwischen Funktionen und Prozeduren
  - Prozeduren sind Funktionen mit Rückgabetyp void
  - Funktionsparameter werden als Wertetyp übergeben

  → sind nur innerhalb der Funktion gültig

baum.c

Hauptprogramm

- Hauptprogramm in C – Funktion main()
- Deklaration von main:
  - int main (int argc, char * argv[]);
  - int main (int argc, char ** argv);
- Die Parameter
  - argc – Zahl der Argumente (inklusive Programmnamen)
  - argv – Vektor von Zeichenketten (die Argumente)
  - Rückgabewert – int
- Definition von main:
  - main () {} - Rückgabewert int wird implizit definiert, Parameter werden nicht benutzt
  - int main (void) {} – Parameter sollen nicht benutzt werden
  - int main (int argc, char * argv[]) {} – vollständige Definition
- Definition muss einer vorangegangenen Deklaration entsprechen
  - Compiler überprüft _nicht_ ob Deklarationen und Definitionen in verschiedenen Übersetzungseinheiten zueinander passen → Präprozessor

Funktionale Abstraktion

- Funktionen: Unterprogramme, die Werte liefern
  - Verwendbar als Ausdrücke
  - möglichst keine Seiteneffekte
- return-Anweisung bestimmt Funktionsergebnis

```
int ggt( int x, int y) {
    while (x != 0 && y != 0) {
        if (x > y)
            x = x%y;
        else
            y = y%x;
    }
    return x+y;
}
```

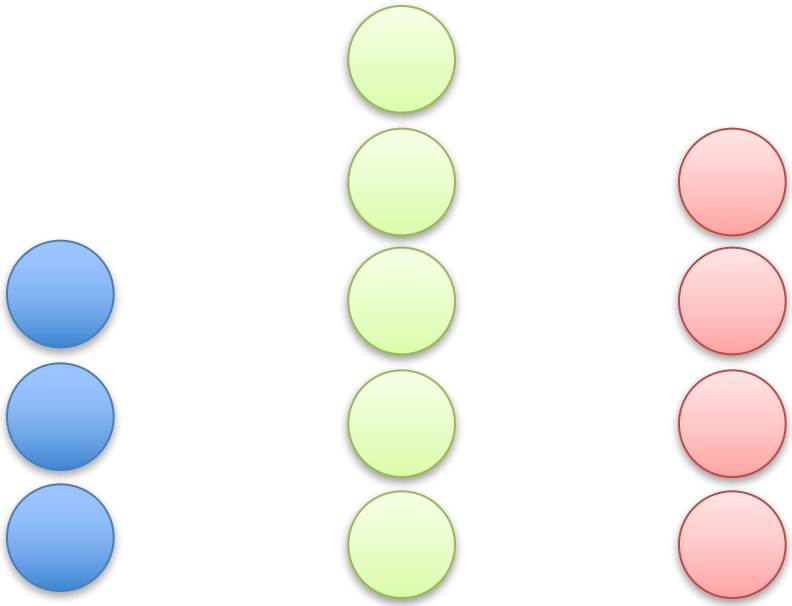
ggt.c

Top-Down-Entwurf

- Annahme: Spezifikation gegeben
  - Ansonsten: Zunächst Problemanalyse
- Beginn der Entwicklung: Programmgerüst
  - Verlagerung von Teilfunktionen in leere Unterprogramme
- Prozedurrümpfe (stubs): abstrakte Operationen
  - C: {}
  - void anmelden(char * vorname, char * nachname) {}
  - int anmelden(char * vorname, char * nachname) { return 0; }
- Schrittweises Ausfüllen des Gerüsts: Verfeinerung
  - iterativer Prozess: in jedem Schritt werden u.U. neue Prozedurrümpfe eingeführt

Top-Down-Entwurf - Beispiel

- NIM-Spiel (Nim Game): Das Spiel beginnt mit drei Reihen von Steinen. Zwei Spieler ziehen abwechselnd. Ein Zug besteht darin, eine Reihe auszuwählen und aus dieser Reihe beliebig viele – jedoch mindestens einen – Stein wegzunehmen. Wer den letzten Stein nimmt, gewinnt.
- Aufgabe: Gesucht ist ein Programm, das dieses Spiel spielt



Programmgerüst (v1)

```
init();
fertig = 0;
spieler = 1;
while ( ! fertig ) {
    zeige_spiel();
    mache_zug();
    if ( spiel_ende())
        fertig = 1;
    else
        spielerwechsel();
}
gratuliere_dem_sieger();
```

Programmgerüst (v2)

```
static int reihe1 = 0, reihe2 = 0, reihe3 = 0;
void init() {
    reihe1 = 3; reihe2 = 5; reihe3 = 4;
}
void zeige_spiel() {}
void mache_zug() {}
int spiel_ende() { return 1; }
void spielerwechsel() {}
void gratuliere_dem_sieger() {}

/* weiter wie gehabt... */
init();
...
```

Top-Down-Entwurf - Verfeinerung

- Aufgabenstellung unterspezifiziert;
  deshalb mehrere mögliche Verfeinerungen
  1. Vervollständigung der Routinen, so dass ein Demospiel vorführbar ist
  2. Einbau einer Strategie, um den bestmöglichen Zug zu ermitteln
  3. Erweiterung der Nutzerinteraktion
     - Initiale Abfrage der Anzahl der Spielsteine
     - Integration des Nutzers als einer der Spieler
  4. graphische Ausgabe statt textueller

Top-Down-Entwurf: Vervollständigung zu Demo- Programm

```
static int reihe1 = reihe2 = reihe3 = 0;  static int zugnummer = 0;
void init() {
      reihe1 = 3; reihe2 = 5; reihe3 = 4;
      zugnummer = 0;
}
void zeige_spiel() {
       /* Ausgabe des Spielfelds auf die Standardausgabe */
       printf("Zugnummer %d: Reihe 1: %d, Reihe 2: %d, Reihe 3: %d\n",
                    zugnummer, reihe1, reihe2, reihe3 );
}
int spiel_ende() {
       /* Das Spiel ist zu Ende wenn es keine Steine mehr gibt */
       return reihe1+reihe2+reihe3 == 0;
}
```

...weiter geht's..

```
static int spieler = 0;

void spieler_wechsel() {
        spieler = 3-spieler;
}

void gratuliere_dem_sieger() {
        /* gewonnen hat, wer den letzten Zug gemacht hat; */
        /* dieser Wert ist noch in Variable spieler gespeichert */
        printf("Gewonnen hat der Spieler %d \t", spieler );
        printf("Herzlichen Glückwunsch\n" );
}
```

NIM-Spiel: Strategie

- Spieler sollte versuchen, dem Gegner ein Spielfeld vorzulegen, bei dem reihe1 xor reihe2 xor reihe3 == 0
  - Nach gegnerischem Zug kann dann nicht mehr reihe1 xor reihe2 xor reihe3 == 0 gelten
  - Spieler kann nach eigenem Zug diesen Zustand immer wieder herstellen
  - Nach letztem Zug gilt reihe1 xor reihe2 xor reihe3 == 0
- NIM-Summe (xor-Summe):
  - Binäre digitale Summe der Größen der Reihen
  - Ohne Übertrag

Binär	Dezimal	
011_2	3_{10}	Reihe 1
101_2	5_{10}	Reihe 2
100_2	4_{10}	Reihe 3
010_2	2_{10}	Resultat

Die NIM-Summe der Reihen 1, 2, 3 ist $3 \oplus 4 \oplus 5 = 2$

Idealer Spielzug

```
void mache_zug() {
        zugnummer += 1;
        if (reihe1 > (reihe2 ^ reihe3))
                reihe1 = reihe2 ^ reihe3;
        else if (reihe2 > (reihe1 ^ reihe3))
                reihe2 = reihe1 ^ reihe3;
        else if (reihe3 > (reihe1 ^ reihe2))
                reihe3 = reihe1 ^ reihe2;
        else
                verlegenheitszug();
}
```

nim.c

Datenfluss zwischen Haupt- und Unterprogramm

- Änderung von globalen Variablen: Seiteneffekt
  - Variablenänderung nicht in Funktionsergebnis sichtbar
  - Abhängigkeiten zwischen Prozeduren werden unüberschaubar
- Kapselung: Zahl der Prozeduren, die auf gemeinsame Variablen zugreifen, sollte klein sein
  - information hiding
- im Beispiel: eine Prozedur

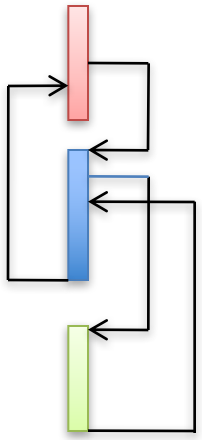
  void nimm_von_reihe(int reihe, int zahl); ...
  - würde das eigentliche Wegnehmen von der Strategie trennen
  - Objektorientierte Programmierung: Kapselung von Werten in Objekten
- im Beispiel: „Spielfeld“ als Klasse, mit Lese- und Schreiboperationen für die Reihen (Alternativ: „Spielstand“)

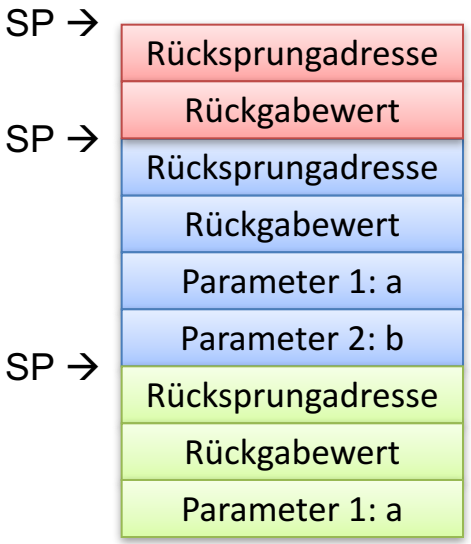
Prinzip Funktionsaufruf

- Compiler muss Sprung realisieren
  - Raum für Rückgabewert, Parameter und lokale Variablen schaffen
  - Rücksprungadresse merken
  - Speichern der Daten in CPU-Registern oder auf Stack

Beispiel:
```
int f( int a, int b ) { return a + g(b); }
int g( int a ) { return a * a; }

main() { return f( 2, 3); }
```





Und nun:
```
main() { return f( 2, g(3)); }
```

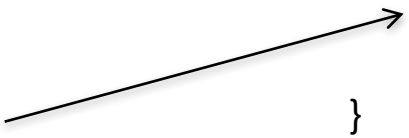
Parameterübergabe

- Wer verwaltet den Stack?
  - Aufrufer – C, C++
    - Nachteilig: Code-Replikation
    - Funktion kann mit variabler Parameterliste gerufen werden
    - Funktion kann Parameter ignorieren
    - Funktion kann Parameter beliebig interpretieren – gefährlich, flexibel
  - Aufgerufener – Pascal, Modula
    - Vorteilhaft: Kompakterer Code
    - Klare Regeln beim Umgang mit Parametern
- Parametertypen
  - Werte-Parameter (call-by-value) – einziger Weg in C
  - Referenz-Parameter (call-by-reference) – VAR n Integer in Pascal
    - Tatsächlicher Parameter muss formalem Parameter vom Typ her entsprechen
    - Problematisch bei Nebeläufigkeit (threads)
  - Call-by-copy-restore (call-by-value-return – FORTRAN)
    - Aufrufer erhält temporäre Kopie der Variablen, Wert wird nach Aufruf zurückkopiert

Referenzparameter simulieren

```
(* Uebergabe der Variablen X als
     Referenzparameter in PASCAL *)
PROGRAM Demo(input,output);
     PROCEDURE Increment (VAR N: Integer);
     BEGIN
       N:=N+1;
     END;
     VAR X: integer;
BEGIN
     Write('Bitte X eingeben'); ReadLn(X);
     Increment(X);
     Write('Der Nachfolger von X ist: ');
     WriteLn(X);
END.
```

```
/* Uebergabe der Adresse der Variablen X
     als Wert-Parameter in C */
# include <stdio.h>
void Increment(int * np)
{
     *np = *np + 1;
}
static int x;
void main() {
     char buf[40];
     printf("Bitte X eingeben:");
     fgets( buf, sizeof(buf), stdin );
     x = atoi( buf );
     Increment( &x );
     printf("Nachfolger ist %d\n", x );
}
```



Übergeben Adresse der Variablen

Rückgabewerte

- Funktionen können alle integralen Datentypen zurückgeben
  - char, int, long, float, double
  - void für Prozeduren (kein Rückgabewert)
- Funktionsdefinition gibt Typ des Rückgabewertes an
  - **double atof(char s[]);** - Deklaration ohne Parameterliste möglich; fehleranfällig
  - Aufrufer muss Typ des Rückgabewertes kennen → **Deklaration** nötig
  - Deklarationen können in header-Dateien zusammengefasst und in die benötigten Übersetzungseinheiten eingeschlossen werden
  - cc –MM file.c → erkennt Abhängigkeiten und generiert Eingabe für make
- Falle:
  - Aufruf ohne Deklaration → C deklariert implizit int als Rückgabewert
  - Bsp: double sum += atof(line);

    → hier würde Rückgabewert von atof() als int behandelt, für die Berechnung nach double konvertiert; ohne Compiler-Warnung!!!

return-Anweisung

- Syntax: return *Ausdruck*;
  - Ausdruck wird vor dem Rücksprung in den Rückgabetyp konvertiert
  - Womöglich explizite Typumwandlung nötig

Beispiel:

```
/* atoi: convert string to integer using atof */
int atoi( char s[] ) {
        double atof( char s[] ); /* Deklaration */
        return (int) atof( s );
}
```

Externe Variablen

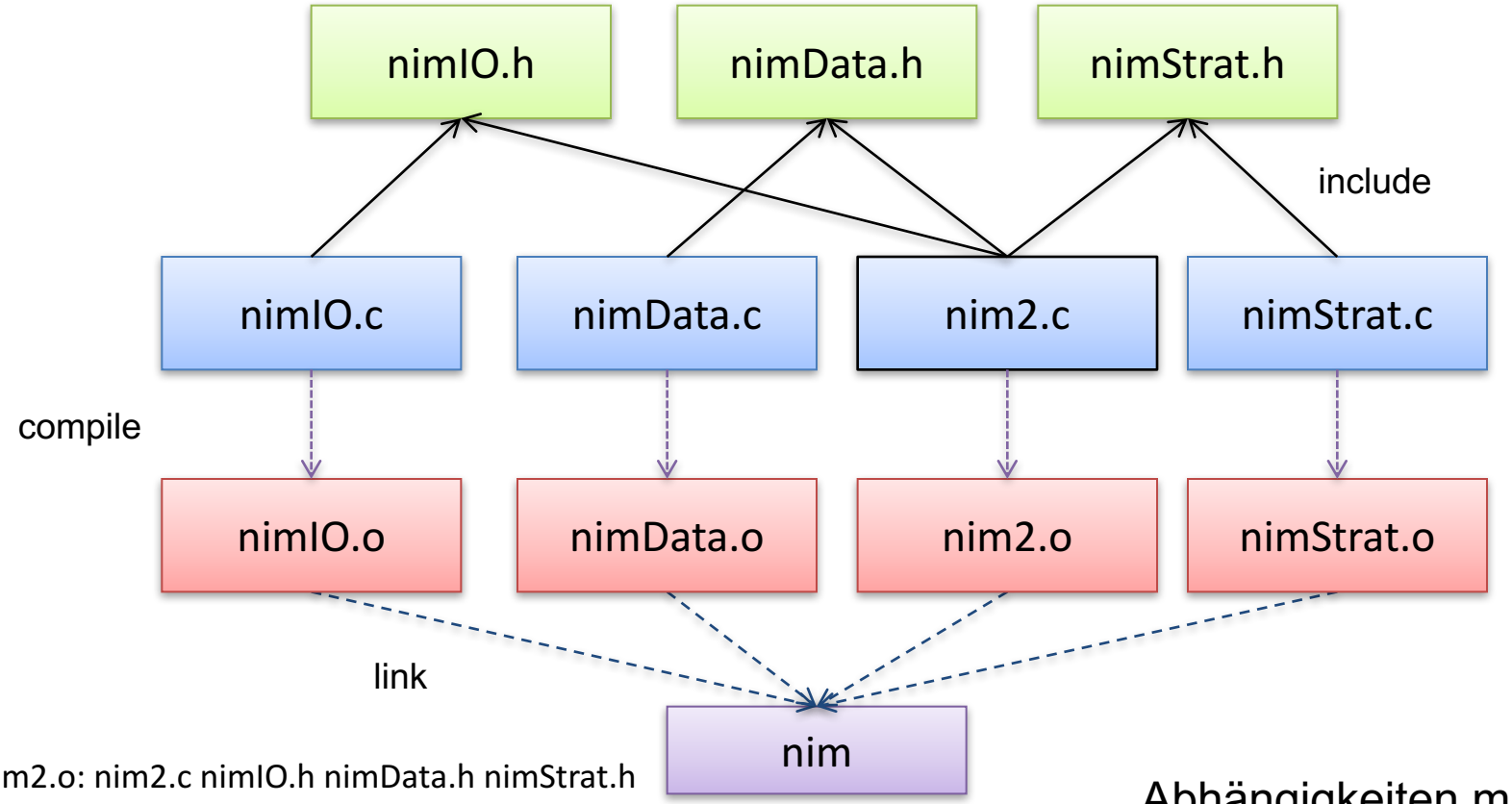
- C-Programm beschreibt eine Menge externer Objekte
  - Funktionen und Variablen
  - Externe Variablen werden außerhalb von Funktionen definiert
    - Für viele Funktionen sichtbar
    - Auch für andere Übersetzungseinheiten sichtbar (Linker-Symbole)
  - Gegenteil von extern ist **static** (Sichtbarkeit auf Datei beschränkt)
- Externe Objekte haben einen gemeinsamen Namensraum
  - Über Übersetzungseinheiten hinweg – (mindestens 6 signifikante Zeichen...)
  - Dürfen nur einmal definiert werden (aber häufiger deklariert)
  - Deklaration mit Schlüsselwort extern, Bsp.: extern int nextSymbol;
- Informationsaustausch über externe Variablen
  - Lebensdauer entspricht Laufzeit des Programms
  - Für alle Funktionen im Programm sichtbar
    - Können aber durch lokale Variablen überdeckt werden
- Schlüsselwort static begrenzt Sichtbarkeit auf Übersetzungseinheit
  - static int stack[], sp = 0;

Sichtbarkeitsregeln

- Scope – Sichtbarkeitsbereich für einen Namen
- Automatische (lokale) Variable in einer Funktion:
  - Scope == von Definition bis Ende des Funktionsblocks
  - Funktionsparameter sind ebenfalls lokale Variablen
- Externe Variable
  - Scope == von Definition bis zum Ende der Übersetzungseinheit
  - Definition kann Initialisierung beinhalten
  - Feldgrößen müssen bei Definition angegeben werden; optional bei Deklaration
- Benutzung vor Definition
  - Variablen in anderen Übersetzungseinheiten
  - Im Quelltext vor der Definition
  - extern <type> <variable>
- Nur eine Definition; aber viele extern Deklarationen möglich

nimData.h

NIM-Spiel : Refaktorisierung



nim2.o: nim2.c nimIO.h nimData.h nimStrat.h
nimIO.o: nimIO.c nimIO.h nimData.h
nimData.o: nimData.c nimData.h
nimStrat.o: nimStrat.c nimStrat.h nimData.h

Abhängigkeiten mit cc –MM <source.c> generieren

Header-Dateien

- Deklarationen sollen zentralisiert sein
  - Übersetzungseinheiten sollen „automatisch“ die richtigen Deklarationen sehen
  - Deklaration und Implementierung sollen getrennt werden
  - → header-Dateien (Endung .h)
- C-Präprozessor
  - # include-Direktive – Einschluss einer Datei
  - Header-Dateien enthalten Deklarationen von Variablen und Funktionen
    - Keine Definitionen, keine Implementationen
  - Typ**definitionen** (später) sind problematisch
    - Präprozessor muss so gesteuert werden, dass header-Datei genau einmal eingeschlossen wird
- Tradeoff
  - Funktionale Dekomposition → viele Module + Header-Dateien (information hiding) versus ease-of-use (eine zentrale header-Datei)

Initialisierung

- Externe und statische Variablen werden implizit auf 0 initialisiert
  - Explizite Initialisierung kann bei Definition angegeben werden
  - Durch konstanten Ausdruck (Berechnung zur Compile-Zeit)
- Automatische (lokale) und Register-Variablen werden implizit nicht initialisiert
  - Explizite Initialisierung bei Definition möglich
  - Durch beliebigen Ausdruck; Berechnung bei Block-Eintritt
- Initialisierung skalarer Variablen

```
int x = 1; char squote = '\'';
long day = 1000L * 60L * 60L * 24L;  /* milliseconds per day */
```

- Array kann durch Initialisierer in geschweiften Klammern bei Definition belegt werden

```
int days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
```

  - Compiler berechnet benötigte Array-Größe
  - Fehlen Elemente, so werden die korrespondierenden Felder auf 0 initialisiert
- Spezialfall character array (string)
  - char pattern[] = "anna"; äquivalent zu char pattern[] = { 'a', 'n', 'n', 'a', '\0' };

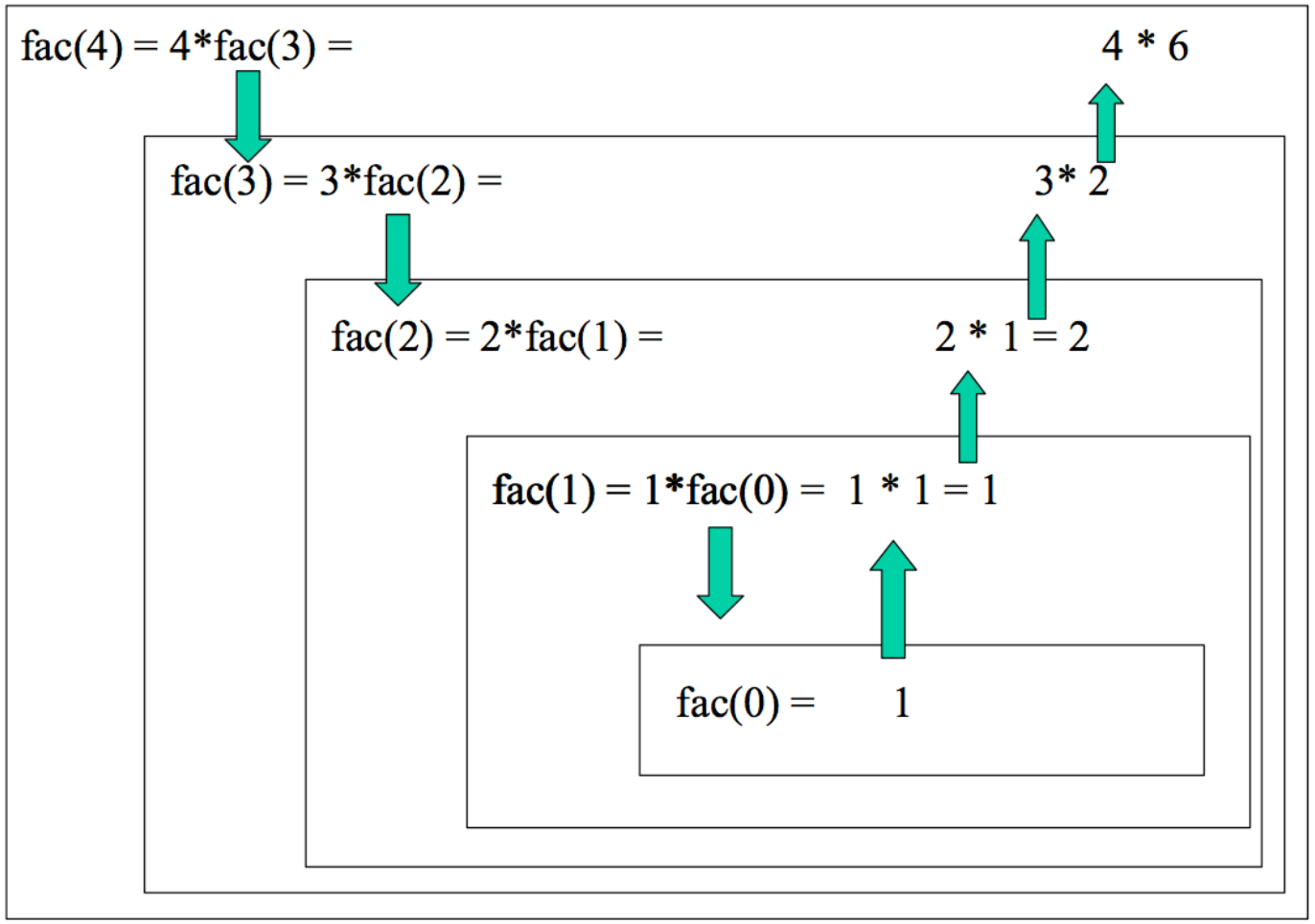
Rekursion

- Beschreibung einer mathematischen Funktion oft durch Fallunterscheidung; ein Fall bezieht sich wieder auf die Funktion
  - Beispiel: Fakultät fac(n) = n!
  - fac(0) = 1
  - fac(n) = n * fac(n-1), falls n > 0

- Ableitung einer Berechnungsvorschrift:
  sequentielles Durchtesten der Fälle

```
int fac(int n) {
        if (n == 0)
                return 1;
        if (n > 0)   /* Test auf n>0 ist hier redundant */
        return n * fac(n-1);
```

Verarbeitung der Rekursion



Formulierung von Rekursion

- Zerlegung des Problems in Teilprobleme:
  - ein Teilproblem ist „gleichartig“ dem Originalproblem
- Beispiel: fac(n-1) ist „im Prinzip“ genauso wie fac(n)
- Teilproblem muss „leichter“ lösbar sein als Originalproblem
  - anderes Teilproblem kombiniert die Lösung des ersten Teilproblems mit weiterem Rechenschritt zur Gesamtlösung:
- Beispiel: fac(n) = n*fac(n-1) (Multiplikation der Teillösung mit n)
- Rekursionsabbruch: „einfachstes“ Teilproblem wird nicht weiter zerlegt; Lösung wird „direkt“ bestimmt
- Teile-und-herrsche-Prinzip
  - Vereinfachung des Problems durch Zerlegung
  - engl.: divide-and-conquer
  - lat.: divide-et-impera
- Historisch falsch: Teile werden nicht gegeneinander ausgespielt

Rekursive Prozeduren

- Umkehrung der Berechnungsreihenfolge durch Rekursion
- Beispiel: Erzeuge Binärdarstellung einer Zahl
  - 1. Versuch: Ziffern werden in falscher Reihenfolge ausgegeben

```
while (n > 0) {
        printf(”%d”, n%2); n /= 2;
}
```

  - Rekursive Lösung: Gib zuerst die höherwertigen Bits aus, danach das letzte:

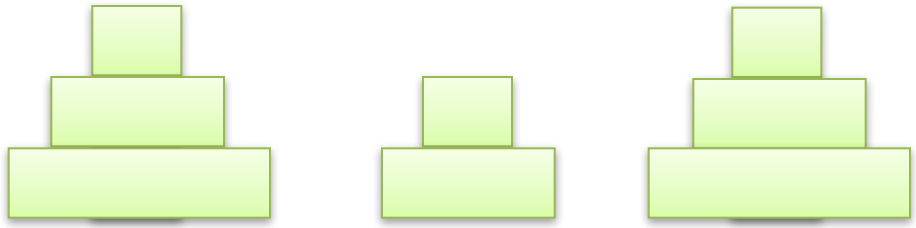
```
void writeBin(int n) {
        if (n < 2)
                printf(”%d”, n);
        else {
                writeBin( n/2 );
                printf(”%d”, n%2);
}
```

Rekursion in C

- C-Funktionen können rekursiv benutzt werden
  - Direkt oder indirekt; (anders als in FORTRAN)
- Bei rekursivem Aufruf:
  - Jede Instanz erhält einen frischen Satz automatischer (lokaler) Variablen
  - Speicher wird auf dem Stack alloziert
- Code häufig kompakter
  - Besser verständlich
  - Keine Ersparnis an Speicherplatz
  - Kein Gewinn an Laufzeit

Türme von Hanoi

- Auf einem Stapel liegen N Scheiben verschiedener Durchmesser; der Durchmesser nimmt von unten nach oben schrittweise ab.
- Der Turm steht auf einem Platz A und soll zu einem Platz C bewegt werden, wobei ein Platz B als Zwischenlager benutzt werden kann.
- Dabei müssen 2 Regeln eingehalten werden:
  - Es darf immer nur eine Scheibe bewegt werden
  - Es darf nie eine größere auf einer kleineren Scheibe liegen



Türme von Hanoi (contd.)

- Lösungsstrategie: induktive Lösung
  - Verschieben einer Scheibe: Scheibe von Platz 1 (z.B. A) auf Platz 2 (z.B. C)
  - Verschieben von K Scheiben: Verschiebe K-1 Scheiben von Platz 1 auf Hilfsplatz H (etwa: B), verschiebe K-te Scheibe von Platz 1 auf Platz 2, verschiebe K-1 Scheiben von H auf Platz 2
- Rekursive Sicht:
  - Angenommen, wir können bereits K-1 Scheiben verschieben, dann wissen wir auch, wie wir K Scheiben verschieben
  - Wir wissen, wie wir eine Scheibe verschieben (Rekursionsende)
- Problem: Keine feste Zuordnung von symbolischen Plätzen (1, 2, H) zu tatsächlichen Plätzen (A, B, C)
- Lösung: symbolische Plätze sind Variablen/Parameter, tatsächliche Plätze die Werte von dieser Variablen

Türme von Hanoi (contd.)

```
void ziehe_scheibe(int nummer, char * von, char * nach) {
        printf("Scheibe %d wird von %s nach %s verschoben\n",
                nummer, von, nach);
}

void hanoi(int N, char * platz1, char * hilfsplatz, char * platz2) {
        if (N == 1)
                ziehe_scheibe(N, platz1, platz2);
        else {
                hanoi(N-1, platz1, platz2, hilfsplatz);
                ziehe_scheibe(N, platz1, platz2);
                hanoi(N-1, hilfsplatz, platz1, platz2);
        }
}
...
hanoi( 4, ”A”, ”B”, ”C” );
```

hanoi.c

Backtracking

- Weitere Verwendung rekursiver Prozeduren: Spielstrategien
  - Annahme: Spiel mit vollständiger Information (alle Konsequenzen eines Zugs sind vorhersehbar), z.B. Schach, Go, ...
- Idee: „In Gedanken“ wird das Spiel zu Ende gespielt und versucht, jeweils den optimalen Zug zu ziehen
- im aktuellen Spielstand werden „in Gedanken“ der Reihe nach alle möglichen Züge ausprobiert
- mehrere mögliche Ergebnisse:
  - egal welchen Zug man spielt, man gewinnt immer
- Ergebnis: Man wird gewinnen, ein beliebiger Zug ist geeignet
  - egal welchen Zug man spielt, man verliert immer
- Ergebnis: entsprechend
  - Bei manchen Zügen wird man gewinnen, bei manchen verlieren
- Ergebnis: man wähle einen Zug, bei dem man gewinnen wird, und verbuche das als „man wird gewinnen“
- Algorithmus sucht der Reihe nach alle Varianten ab, bis er einen Zug gefunden hat, der zum Sieg führen wird
  - anderenfalls nimmt man den Zug „in Gedanken“ zurück, und probiert einen anderen: **Backtracking**

Wechselseitige Rekursion

- in der Definition der Funktion f wird die Funktion g aufgerufen, und in der Definition von g wird f aufgerufen
  - Konsequenz: Verwendung im Programm textuell vor Definition
    - Python, Java: Reihenfolge der Definitionen irrelevant
    - C, Pascal: Vorwärtsdeklarationen
- Beispiel:

```
int ungerade(int n);
int gerade(int n) {
        if (n == 0) return 1;
        return ungerade(n-1);
}
int ungerade(int n) {
        if (n == 0) return 0;
        return gerade(n-1);
}
```

Allgemeine Rekursion

- Definition der Funktion greift mehrfach auf dieselbe Funktion zurück
- Beispiel: Fibonacci-Funktion
  - Modell zur Populationsentwicklung z.B. bei Kaninchen

$$fib(n) = \begin{cases} 1, & \text{falls } n \leq 1 \\ fib(n-1) + fib(n-2), & \text{sonst.} \end{cases}$$

Endrekursion

- Eine Funktionsdefinition ist endrekursiv (*tail recursive*), falls sie die Form hat

$$f(x) = \begin{cases} g(x), & falls\ P(x) \\ f(r(x)), & sonst. \end{cases}$$

- Beispiel:

$$gerade_1(n) = \begin{cases} (n = 0), & \text{falls } n \leq 0 \\ gerade_1(n-2), & \text{sonst.} \end{cases}$$

Endrekursion (contd.)

- Falls dem rekursiven Aufruf noch eine Berechnung folgt, liegt keine Endrekursion vor:

$$gerade_2(n) = \begin{cases} (n = 0), & \text{falls } n < 1 \\ \neg gerade_2(n-1), & \text{sonst.} \end{cases}$$

- Bei Endrekursion kann der ursprüngliche Aufruf gänzlich ersetzt werden:
  - gerade1(4) = gerade1(2) = gerade1(0) = (0=0) = True
  - aber: gerade2(4) = ¬gerade2(3) = ¬ ¬ gerade2(2) = ¬ ¬ ¬ gerade2(1) = ¬ ¬ ¬ ¬ gerade2(0) = ¬ ¬ ¬ ¬ ¬ ¬ True = ... = True
- Endrekursive Definition kann leicht in iterative Definition übertragen werden
  - In manchen Programmiersprache (z.B. LISP, Scheme) passiert das automatisch

Lineare Rekursion

- Verallgemeinerung der Endrekursion
  - für h(x,y)=y ergibt sich Endrekursion

$$f(x) = \begin{cases} g(x), & falls\ P(x) \\ h(x, f(r(x))), & sonst \end{cases}$$

- f(x) = h(x, f(r(x)))
  = h(x, h(r(x), f(r^2(x))))
  = h(x, h(r(x), h(r^2(x), f(r^3(x)))))
  = ...
  = h(x, h(r(x), h(r^2(x), h(..., h(r^{k-1}(x), g(r^k(x)))...))))

  wobei k kleinste nat. Z. mit P(r^k(x)) = 1

Lineare Rekursion (contd.)

- im Allgemeinen Berechnung nur durch Stack s möglich
  - Berechnung von $r^n(x)$, Speichern auf Stack, Auslesen in umgekehrter Reihenfolge (LIFO: Last-In-First-Out)

```
while ( ! P(x)) {
    push(x, s);
    x = r(x);
}
f = g(x)
while ( ! empty(s)) {
    x = top(s);
    pop(s);
    f = h(x,f);
}
return f;
```

Lineare Rekursion (contd.)

- Lineare Rekursion lässt sich u.U. in Endrekursion umformulieren
  - Für assoziative Operationen mit Linkseinheit (siehe Gumm, Sommer, S. 159)
  - Beispiel: Fakultät fac(n)=n!

$$fac(n) = \begin{cases} 1, & falls\ n = 0 \\ n \times fac(n-1), & sonst \end{cases}$$

  - Ersetzbar durch fac(n) = facAux(n, 1)

$$facAux(n,a) = \begin{cases} a, & \text{falls } n = 0 \\ facAux(n-1, n \times a), & \text{sonst.} \end{cases}$$

  - zusätzlicher Parameter übernimmt Rolle des *Akkumulators*

Transformation des Algorithmus zur Berechnung von Fibonacci-Zahlen

- Akkumulatorverfahren u.U. erweiterbar auf allgemeine Rekursion
- Fibonacci berechnet viele Teilergebnisse andauernd
  - Idee: Speichern die letzten Werte von fib(k-2) und fib(k-1)
  - Beispiel: fib(n) = fibAux(n,1,1) mit

```
int fibAux(int n, int acc1, int acc2) {
    if (n == 0) return acc1;
    return fibAux(n-1, acc2, acc1+acc2);
}
```

  - Iterative Definition: Auflösen der Endrekursion:

```
int fib(int n) {
    int acc1 = 1, acc2 = 1, tmp;
    while (n > 0)
        n = n-1; tmp = acc1;
        acc1 = acc2; acc2 = tmp+acc2;
    return acc1 ;
}
```

Der C-Präprozessor

- Erste Stufe der Übersetzung eines C-Programms
  - Textuelle Ersetzung
  - # include – Einschluss von Deklarationen (header-Dateien)
    - # include ”filename” – Suche beginnt im aktuellen Verzeichnis
    - # include <filename> – Suche erfolgt entlang des definierten Include-Pfades
    - Compiler-Option –I <include-pfad>
  - # include wirkt rekursiv
    - Compiler-Option –E gibt Quellprogramm nach Auflösung aller # include und # define-Direktiven aus
    - Compiler-Option –MM gibt Abhängigkeiten zu # include-Dateien in einem für make geeigneten Format aus
  - # define – Definition von Makros
    - Makros bewirken Textersetzung

Makro-Substitution

- Makro-Definition:
  - # define *name Ersetzungstext*
  - Von nun an werden alle Vorkommen von name ersetzt
    - Ersetzungstext reicht bis Zeilenende
    - Zeilenende kann mit \ maskiert werden
  - Ersetzungen finden nur für Token statt, nicht in Zeichenketten
    - # define YES 1
    - Keine Ersetzung in printf(”YES”); oder in YESMAN
  - Name kann mit beliebigem Ersetzungstext definiert werden
    - # define forever for (; ;) /* Endlosschleife */
    - # define BEGIN {
    - # define END }
  - Makros können Argumente besitzen
    - # define max(A, B) ((A) > (B) ? (A) : (B))
    - Wird als inline-Code ersetzt (kein Funktionsaufruf)
    - X = max(p+q, r+s); → x = ((p+q) > (r+s) ? (p+q) : (r+s));
    - Polymorphie: max-Makro funktioniert für alle integralen Typen

Makro-Expansion

- Inline-Code-Ersetzung
  - Variablen werden mehrfach ausgewertet
  - Vorsicht bei Seiteneffekten
    - max (i++, j++) → ((i++) > (j++) ? (i++) : (j++)) – unerwartet und falsch
- Klammern sind wichtig
  - Operator-Vorrangregeln werden u.U. verletzt
    - # define square(x) x * x → square (x+1) → x+1 * x+1   (....== 2x+1)
- Formale Parameter werden in Zeichenketten ersetzt
  - #ausdruck → „stringify“ – ausdruck wird in Zeichenkette umgewandelt
  - Bsp. Debug-Ausgabe:
    - # define dprint(expr)   printf(#expr ” =%g\n”, expr)
    - dprint (x/y); → printf(”x/y” ” = %g\n”, x/y);
    - Zeichenkettenverkettung → printf(„x/y = %g\n“, x/y);
- Makros können mit # undef ungültig gemacht werden

##-Operator

- Verkettung von aktuellen Argumenten während Makroexpansion
  - Benachbarte Parameter werden durch aktuelle Argumente ersetzt
  - Umgebende Leerzeichen werden entfernt
  - Resultat wird neu gescannt (weiterverarbeitet)
- Beispiel:
  - # define paste(front, back) front ## back
  - paste(name, 1) → name1
  - paste(Otto, Lilienthal) → OttoLilienthal

  - Makro swap(t, x, y), das zwei Argumente vom Type t austauscht?
  - #define swap(t,x,y) do{t z=x;x=y;y=z}while(0)
    - Geht gut sofern für t ein Zuweisungsoperator definiert ist

swap.c

Bedingte Verarbeitung

- Steuerung des Präprozessors über bedingte Direktiven
  - # if – Direktive
    - Evaluiert einen konstanten Integer-Ausdruck
      - Ohne sizeof, casts, oder enum-Konstanten
  - Ist der Ausdruck != 0, so werden folgende Zeilen bis # endif, # elif, # else verarbeitet (eingeschlossen)
  - # defined(name)
    - Ergibt 1 wenn name definiert ist, 0 sonst
- Beispiel:
  - Einmaliger Einschluss von header-Dateien

```
# if !defined (HDR)
# define HDR
    /* Inhalt der header-Datei */
# endif
```

- Spezialform:
  - # ifdef, # ifndef

Steuerung der Übersetzung

- Portable Programme erhalten plattformspezifischen Code

```
# if SYSTEM == SYSV
    # define HDR ”sysv.h”
# elif SYSTEM == BSD
    # define HDR ”bsd.h”
# else
    # define HDR ”default.h”
# endif
# include HDR
```

- C-Compiler definiert plattformspezifische Namen
  - # ifdef WIN32, # ifdef GCC ...
- Portabilität zwischen Programmiersprachen
  - C++ verwendet „name mangling“ um Parameter-typen in Linkersymbole zu codieren
  - Deklaration extern ”C” {} verhindert dies

```
# ifdef __cplusplus
extern ”C” {
# endif
int myFunction (int arg );
# ifdef __cplusplus
}
# endif
```

Vorherdefinierte Makros

- vordefinierte Makros:
  - __LINE__
  - __FILE__
  - __DATE__
  - __TIME__
  - __STDC__ (1)
  - __STDC_VERSION__ (199901L)

- weitere vordefinierte Makros: implementation-defined

Zusammenfassung

- Grundlagen
  - Unterprogramme und prozedurale Abstraktion
- Prinzip Funktionsaufruf
  - Parameterübergabe, Rücksprungadresse, verschachtelter Aufruf
  - Stack, call-by-value, call-by-ref, call-by-copy
- Rückgabewerte
  - int ist Standard, andere Typen müssen passend deklariert werden
- externe Variablen
  - Funktionen und Variablen sind weithin sichtbar;
  - Deklaration extern führt Namen ein; keine Definition
- Scope - header files und Übersetzungseinheiten
- Initialisierung
  - Globale Variablen werden automatisch initialisiert (auf 0), lokale nicht !!!
- Rekursion
  - Mächtiges Konzept zur kompakten Darstellung von Algorithmen (Funktionen)
- C Präprozessor