Objektorientierte Programmierung
(Winter 2010/2011)

Kapitel 9: Funktionen / Prozeduren

- Deklaration von Funktionen/Prozeduren
- Parameter, Übergabemechanismen
- Referenzen
- Globale und Lokale Deklarationen, Gültigkeitsbereiche, Blöcke
- Rekursive Prozeduren

Inhalt

Prozeduren: Motivation (1)

- Programme können lang und kompliziert werden.
- Ein Programm sollte aus kleineren Einheiten aufgebaut sein, die man unabhängig verstehen kann.
- Dazu müssen diese kleineren Einheiten eine gut dokumentierte und möglichst kleine/einfache Schnittstelle zum Rest des Programms haben.

  D.h. die Interaktion mit dem Rest des Programms muss auf wenige, explizit benannte Programmelemente beschränkt sein.
- Prozeduren/Funktionen sind solche Einheiten.

Prozeduren: Motivation (2)

- Oft braucht man in einem Programm an verschiedenen Stellen ein gleiches Stück Programmcode.
- Um die Lesbarkeit & Änderbarkeit des Programms zu verbessern, sollte dieser Programmcode nur einmal aufgeschrieben werden.
- Das ist mit Prozeduren möglich:
  - ◇ Man kann einem Stück Programmcode einen Namen geben, und
  - ◇ es dann an verschiedenen Stellen im Programm aufrufen (ausführen lassen).

Prozeduren: Motivation (3)

- Prozeduren sind ein Mittel der Abstraktion (man interessiert sich nicht mehr für bestimmte Details, vereinfacht die Realität). Hier
  - ◇ spielt es für den Benutzer/Aufrufer keine Rolle, wie die Prozedur ihre Aufgabe löst, sondern nur
  - ◇ was sie genau macht.
- Dadurch, dass man sich Prozeduren definiert, erweitert man gewissermaßen die Sprache:
  - ◇ Man kann sich die Prozeduraufrufe wie neue, mächtigere Befehle vorstellen.

Prozeduren: Motivation (4)

- Stroustrup (The C++ Prog. Lang., 2000):
  - ◇ "The typical way of getting something done in a C++ program is to call a function to do it."
  - ◇ "Defining a function is the way you specify how an operation is to be done."
- Kernighan/Ritchie (The C Prog. Lang., 1978/88):
  - ◇ "A function provides a convenient way to encapsulate some computation, which can then be used without worrying about its implementation."

Prozeduren: Motivation (5)

- In Prozeduren kann man auch selbst wieder Prozeduren aufrufen.
- "Bottom-up" Konstruktion eines Programms: von unten (C++) nach oben (Aufgabe).

  Beginnend mit dem von C++ vorgegebenen Sprachumfang definiert man sich nach und nach mächtigere Prozeduren, die der vorgegebenen Aufgabenstellung immer näher kommen, bis die Prozedur "`main`" sie schließlich löst.
- "Top-down" Konstruktion eines Programms: von oben (Aufgabe) nach unten (C++).

  Man unterteilt die Aufgabenstellung immer weiter in Teilaufgaben, bis diese Teilaufgaben so klein sind, dass sie sich direkt lösen lassen.

Prozeduren: Motivation (6)

- Jensen/Wirth (Pascal User Manual, 1974/1991):
  - ◇ "As we grow in the art of computer programming, we construct programs in a sequence of *refinement steps*. At each step we break our task into a number of subtasks, thereby defining a number of partial programs. To camouflage this structure is undesirable. The concepts of *procedure* and *function* allow you to display the subtasks as explicit subprograms."

Prozeduren: Motivation (7)

- Prozeduren werden selbst wieder zu Klassen, Modulen, und Bibliotheken zusammengefasst.
- Im Rahmen dieser größeren Einheiten werden Prozeduren zum Teil auch in anderen Programmen wiederverwendet:
  - Es muss nicht jeder das Rad neu erfinden.
  - Man kann Zeit und Geld sparen, indem man von Anderen entwickelten Programmcode verwendet.

    Sofern er gut dokumentiert und möglichst fehlerfrei ist.

Prozeduren: Motivation (8)

- Im wesentlichen sind "Prozedur", "Funktion", "Unterprogramm", "Subroutine" synonym.
- In Pascal gibt es Prozeduren und Funktionen:
  - Prozeduren liefern keinen Wert zurück, der Prozeduraufruf ist ein Statement.
  - Funktionen liefern einen Wert und werden in Expressions benutzt. Von Seiteneffekten wird stark abgeraten.
- In C/C++ gibt es nur Funktionen, die den leeren Typ `void` als Ergebnistyp haben können (wenn sie keinen Rückgabewert liefern).

Beispiel, Grundbegriffe (1)

```
#include <iostream>
using namespace std;

int square(int n)
{
        return n * n;
}

int main()
{
        for(int i = 1; i <= 20; i++)
                cout << i << " zum Quadrat ist "
                        << square(i) << "\n";
        return 0;
}
```


Beispiel, Grundbegriffe (2)

- Der Abschnitt:

  ```
  int square(int n)    // Prozedurkopf
  { return n * n; }    // Prozedurrumpf
  ```

  ist die Definition einer Prozedur:
  - Die Prozedur heißt `square`.
  - Sie hat einen Parameter (Eingabewert, Argument) vom Typ `int`, der `n` heißt.
  - Sie liefert einen `int`-Wert.

    Wie bei Variablen steht der Typ (hier Ergebnistyp, Rückgabetyp) vor dem Namen (der Prozedur).
  - Der Ergebniswert berechnet sich als `n * n`.

Beispiel, Grundbegriffe (3)

- Die Definition einer Prozedur alleine tut nichts.

  Der Compiler erzeugt natürlich Maschinencode.

- Erst durch den Aufruf

  `square(i)`

  wird der Rumpf der Prozedur ausgeführt.

  Ein guter Compiler sollte eine Warnung ausgeben, wenn Prozeduren definiert werden, die nicht ausgeführt werden ("toter Code").

- Der Wert von `i`, z.B. `1`, ist der aktuelle Parameter.

  In der Wikipedia steht, dass "aktueller Parameter" eine falsche Übersetzung von "actual parameter" ist. Korrekt wäre "tatsächlicher Parameter". "actual" kann aber auch "gegenwärtig" bedeuten.

Beispiel, Grundbegriffe (4)

- Beim Aufruf findet die Parameterübergabe statt. Dabei wird der formale Parameter `n` an den aktuellen Parameter (z.B. `1`) gebunden.
- Dies wirkt wie eine Zuweisung: `n = 1;`
- Anschließend wird der Rumpf der Prozedur ausgeführt.
- Die Anweisung

  `return n * n;`

  beendet die Ausführung der Prozedur und legt den Ergebniswert (Rückgabewert) fest (im Beispiel: `1`).

Beispiel, Grundbegriffe (5)

- Der Aufruf einer Funktion/Prozedur ist ein Wertausdruck.
  - ◇ Die Eingabewerte (aktuelle Parameter) der Prozedur können mit beliebig komplexen Wertausdrücken berechnet werden.
  - ◇ Der Rückgabewert der Prozedur kann selbst in einem komplexen Wertausdruck weiter verwendet werden.
  - ◇ Im Beispiel ist der Rückgabewert Eingabe des Operators <<.

Beispiel, Grundbegriffe (6)

- Statt "Parameter" kann man auch "Argument" (der Funktion/Prozedur) sagen.

  Je nach Kontext kann "Argument" entweder "aktueller Parameter" oder "formaler Parameter" bedeuten. Man sagt aber nicht "aktuelles/formales Argument".

  Die Unterscheidung "aktuell" vs. "formal" macht man nur, solange man sich mit der Parameterübergabe beschäftigt. Später ergibt sich das immer aus dem Kontext.

- Statt "aktueller Parameter" sagt man auch "Eingabewert" (der Funktion/Prozedur).

  Der mit `return` festgelegte Rückgabewert wäre entsprechend der Ausgabewert der Funktion/Prozedur.

Beispiel, Grundbegriffe (7)

- Eine Prozedur/Funktion kann mehr als einen Parameter haben:

```
double power(double x, int n)
{
        double result = 1.0;
        for(i = 1; i <= n; i++)
                result *= x;
        return result;
}
```

- Aktuelle und formale Parameter werden über ihre Position verknüpft, beim Aufruf `power(2.0, 3)` bekommt `x` (1. Par.) den Wert `2.0` und `n` den Wert `3`.

Beispiel, Grundbegriffe (8)

- Anzahl und Typ der aktuellen Parameter muss zu den deklarierten formalen Parametern passen.
- Zum Beispiel wäre folgender Aufruf falsch:

  `square(2, 4)`

  `square` ist mit nur einem Parameter deklariert.

  Selbstverständlich erkennt der Compiler diesen Fehler und gibt eine entsprechende Fehlermeldung aus.
- Bei der Parameterübergabe finden die gleichen Typ-Umwandlungen wie bei einer Zuweisung statt.

  Diese sind sehr großzügig (z.B. `int` ↔ `float`). Die Übergabe eines Zeigers/Arrays an die Prozedur führt aber zu einer Fehlermeldung.

`void` als Ergebnistyp (1)

- Prozeduren müssen nicht unbedingt einen Wert zurückgeben.
- Dann verwendet man den Ergebnistyp "`void`":

```
void print_hello(const char *name)
{
        cout << "hello, " << name << "!\n";
}
```

- Natürlich kann man den leeren Ergebniswert nicht verwenden:

```
print_hello("world"); // ok
int n = print_hello("Stefan"); // falsch
```


`void` als Ergebnistyp (2)

- Eine Prozedur mit Ergebnistyp `void` braucht keine `return`-Anweisung:
  - ◇ Falls die Ausführung bis zur schließenden Klammer } durchläuft, kehrt sie automatisch zum Aufrufer zurück.
  - ◇ Man darf allerdings `return;` verwenden, um die Ausführung vorzeigt abzubrechen.
- Prozeduren mit einem anderen Ergebnistyp müssen immer mit einer `return ⟨Wert⟩;` Anweisung enden.

Deklaration vs. Definition (1)

- Prozeduren/Funktionen müssen vor ihrem Aufruf deklariert sein, damit der Compiler die Parameterliste beim Aufruf prüfen kann und ggf. Code für Typumwandlungen erzeugen kann.
- Die oben gezeigten Prozedurdeklarationen wirken gleichzeitig als Definition.
- Man kann aber auch getrennt in einer Deklaration nur den Namen der Prozedur, die Liste der Parameter-Typen, und den Ergebnistyp festlegen:

```
double power(double, int);
```


Deklaration vs. Definition (2)

- Es ist zulässig, in der (reinen) Prozedur-Deklaration auch die Namen der Parameter anzugeben:

```
double power(double x, int n);
```

  Sie werden vom Compiler wie Kommentare ignoriert. Für den menschlichen Leser des Programms können sie aber sehr nützlich sein.

- Die (reine) Deklaration einer Prozedur enthält alle Angaben, die der Compiler benötigt, um den Aufruf dieser Prozedur zu übersetzen.

  Die Trennung von Deklaration und Definition ist insofern wichtig, als die Definition bei großen Programmen auch in einer anderen Quelldatei stehen kann, die in einem getrennten Compilerlauf übersetzt wird. Das wird in einem späteren Kapitel genauer besprochen.

Deklaration vs. Definition (3)

- Die (reine) Deklaration enthält nur die Schnittstelle einer Prozedur. Eine Deklaration, die auch Definition ist, darüber hinaus die Implementierung.

  Den Prozedurkopf, der in der Deklaration angegeben wird, nennt man auch den Prototyp der Funktion.

- Syntaktisch steht in der Deklaration mit Definition nach dem Prozedurkopf kein Semikolon.

  Der Prozedurkopf in der Dekaration+Definition ist der (reinen) Deklaration ähnlich, aber Parameternamen müssen angegeben werden.

- Dies zeigt dem Compiler an, dass der Prozedurrumpf (ein "compound statement") folgt.

Deklaration vs. Definition (4)

```
...
int square(int); // reine Deklaration

int main()
{
        for(int i = 1; i <= 20; i++)
                cout << i << " zum Quadrat ist "
                        << square(i) << "\n";
        return 0;
}

int square(int n) // Definition
{
        return n * n;
}
```

Deklaration vs. Definition (5)

- Wie man sieht, kann eine Quelldatei sowohl eine (reine) Deklaration als auch eine Definition der gleichen Prozedur enthalten.
- Natürlich müssen sie in den Parameter-Typen und im Ergebnistyp übereinstimmen.

  Da C++ überladene Prozeduren erlaubt, d.h. mehrere verschiedene Prozeduren mit gleichem Namen, aber unterschiedlicher Parameterliste, wird der Fehler erst beim Linken festgestellt. Überladene Prozeduren werden in einem späteren Kapitel genauer behandelt.
- Die Deklaration muss vor dem Aufruf stehen, die Position der Definition ist egal (sofern sie nicht gleichzeitig als Deklaration dient).

Deklaration vs. Definition (6)

- Bei Variablen waren alle bisher gezeigten Deklarationen auch Definitionen:
  - ◇ Es wurde nicht nur der Name (Bezeichner) mit einem bestimmten Typ eingeführt,
  - ◇ sondern auch Speicherplatz reserviert.
- Im Zusammenhang mit der Aufteilung eines großen Programms auf mehrere Quelldateien (Module) ist es auch möglich, (globale) Variablen nur zu deklarieren, aber nicht zu definieren (s.u.).

Inhalt

Lokale Variablen (1)

- Das syntaktische Konstrukt `{...}` nennt man Block.

  Eigentlich ist es in C++ einfach ein "compound statement". Die Idee der Blockstruktur wurde wohl in Algol 60 eingeführt. Dort ist ein Block die Zusammenfassung von Deklarationen und Statements. Wie oben erläutert, sind Deklarationen in C++ formal auch Statements.
- Die Deklaration einer Variablen ist bis zum Ende des Blockes gültig, in dem sie deklariert wurde.

  Ausnahme sind Deklarationen in Bedingungen oder im `for`-Statement: Sie gelten nur für das jeweils abhängige Statement.
- Insbesondere sind alle in einer Prozedur deklarierten Variablen nur in dieser Prozedur bekannt. Man nennt sie daher lokale Variablen.

Lokale Variablen (2)

```
...
int square(int);

int main()
{
        int i;
        for(i = 1; i <= 20; i++)
                cout << square(i) << "\n";
        return 0;
} // Ende des Gültigkeitsbereiches von i

int square(int n)
{
        return n * i; // Fehler: i hier unbekannt
}
```


Lokale Variablen (3)

- Der Zweck von Prozeduren ist ja, ein in sich geschlossenes Programmstück mit möglichst kleiner Schnittstelle zum Rest des Programms als Einheit zu verpacken.
- Könnte man beliebig auf Variablen fremder Prozeduren zugreifen, wäre die Interaktion zwischen den Prozeduren völlig undurchschaubar.
- Idealerweise läuft jeder Datenaustausch der Prozedur mit dem Rest des Programms nur über Parameter ab.

Globale Variablen (1)

- Es ist möglich, Variablen außerhalb von Funktionen zu deklarieren.
- Solche Variablen sind von der Deklaration bis zum Ende der Quelldatei bekannt.

  Tatsächlich kann man auch von anderen Quelldateien aus auf solche Variablen zugreifen. Das wird in einem anderen Kapitel erläutert.
- Diese Variablen heißen "global".
- Auf globale Variablen können möglicherweise viele Prozeduren zugreifen, ohne sie explizit in der Parameterliste aufzuführen.

Globale Variablen (2)

```
...
int square(int);
int i; // Schlechter Stil!

int main()
{
        for(i = 1; i <= 20; i++) // Schlechter Stil
                cout << square(i) << "\n";
        return 0;
}

int square(int n)
{
        return n * i; // Korrekt, schlechter Stil
}
```

Globale Variablen (3)

- Vorteil der Verwendung von globalen Variablen ist, dass ggf. die Parameterlisten kürzer werden.

  Es gibt Untersuchungen, nach denen man sich die Reihenfolge von mehr als sechs Parametern sehr schlecht merken kann.
  Im Beispiel macht die globale Variable dagegen überhaupt keinen Sinn.

- Nachteil ist, dass die Programme schnell ganz undurchschaubar werden.

  Die Verwendung von globalen Variablen ist verpönt.

- Mit Klassendefinitionen (siehe Kapitel 11) kann die Zugreifbarkeit von "globalen" Variablen auf ein sinnvolles Maß reduziert werden.

Variablen gleichen Namens (1)

```
...

int square(int);

int main()
{
        for(int i = 1; i <= 20; i++)
                cout << square(i) << "\n";
        return 0;
}

int square(int n)
{
        int i = n * n;
        return i;
}
```


Variablen gleichen Namens (2)

- Wenn eine Variable mit Namen X schon deklariert ist, kann man nicht noch eine zweite Variable mit dem gleichen Namen X deklarieren.

  Der Compiler wüßte dann ja nicht mehr, was man meint, wenn man X schreibt.

- Da die Deklaration einer lokalen Variable am Ende der Prozedur wieder vergessen wird, kann man anschließend (in einer anderen Prozedur) eine weitere Variable mit dem gleichen Namen deklarieren.
- Dies sind zwei unterschiedliche Variablen, die nichts miteinander zu tun haben.

Variablen gleichen Namens (3)

- Verschiedene Prozeduren sollten sich möglichst wenig gegenseitig beeinflussen.

  Wenn man eine Prozedur anwendet, soll man nicht gezwungen sein, in den Prozedurrumpf hineinzuschauen. Prozeduren sollten unabhängig von einander entwickelt werden können.

- Deswegen ist es ganz natürlich, dass es keine Namenskonflikte zwischen den innerhalb der Prozeduren deklarierten Variablen gibt.
- Die Parameter werden wie lokale Variablen behandelt: Sie sind auch nur innerhalb der jeweiligen Prozedur bekannt.

Verschattung (1)

- Eine Ausnahme von der Regel "niemals gleichzeitig zwei Variablen mit gleichem Namen" ist:
  - ◇ Man kann eine lokale Variable mit dem gleichen Namen wie eine globale Variable einführen.
  - ◇ Dann steht der Name innerhalb der Prozedur für die lokale Variable. Sie ist "näher" deklariert.
  - ◇ Die globale Variable mit gleichem Namen ist innerhalb der Prozedur nicht zugreifbar: Sie ist durch die lokale Variable "verschattet".

    Offenbar wußte der Programmierer der Prozedur nicht, dass es schon eine globale Variable mit diesem Namen gab. Deswegen schadet es nichts, wenn er auf sie nicht zugreifen kann.

Verschattung (2)

```
...

int square(int);
int n = 20;           // Globale Variable n

int main()
{
        for(int i = 1; i <= n; i++) // glob. Var. n
                cout << square(i) << "\n";
        return 0;
}

int square(int n)     // Parameter n
{
        return n * n; // bezieht sich auf Parameter
}
```


Blockstruktur (1)

- Es gibt nicht nur
  - ◇ globale Variablen (außerhalb von Prozeduren deklariert) und
  - ◇ lokale Variablen und Parameter (innerhalb),

  sondern man kann auch in einer Prozedur Blöcke (Gültigkeitsbereiche) in einander schachteln.
- Die Regel ist immer dieselbe:
  - ◇ Weiter außen deklarierte Variablen gelten auch innen (sofern sie nicht verschattet werden), aber
  - ◇ umgekehrt nicht.

Blockstruktur (2)

```
...

int square(int n)
{
        return n * n;
}

int main()
{
        char newline = '\n';
        for(int i = 1; i <= 20; i++) {
                int s = square(i);
                cout << s << newline;
        } // Ende des Gültigkeitsbereiches von s, i
        return 0;
}
```

Inhalt

1. Motivation, Grundbegriffe, Beispiel
2. Lokale und globale Variablen, Blockstruktur
3. Parameterübergabe, Referenzen
4. Interne Details zu Rücksprungadressen, Stack
5. Rekursive Funktionen
6. Verschiedene fortgeschrittene Konzepte

Parameterübergabe (1)

- Bisher wurden nur Eingabeparameter besprochen, bei denen ein Wert vom Aufrufer in die Prozedur hinein fließt.
- Auch eine Zuweisung an einen Parameter bewirkt nach außen nichts (siehe Beispiel, nächste Folie).
- C benutzt "call by value" zur Parameterübergabe:
  - ◇ Selbst wenn der aktuelle Parameter eine Variable ist, wird der Wert dieser Variable an den formalen Parameter zugewiesen (d.h. kopiert).

    Auf die Variable, die als aktueller Parameter angegeben ist, hat man in der Prozedur keinen Zugriff.

Parameterübergabe (2)

```
...
int square(int n)
{
        int s = n * n;
        n = 1000; // Ändert i nicht.
        return s;
}

int main()
{
        for(int i = 1; i <= 20; i++)
                cout << square(i) << "\n";
        return 0;
}
```


Parameterübergabe (3)

- Oft hat eine Prozedur nur einen Ausgabewert, den kann man dann als Rückgabewert verwenden.
- Bei Bedarf kann man in C aber auch Ausgabe-Parameter deklarieren, indem man die Adresse einer Variablen übergibt (also einen Zeiger/Pointer):
  - ◇ Beim Aufruf muss man mit `&` die Adresse der Variablen bestimmen.

    Dies ist etwas umständlich, aber immerhin sieht man klar, dass die Variable durch den Prozeduraufruf wahrscheinlich geändert wird.
  - ◇ Bei jeder Zuweisung in der Prozedur muss man mit `*` den Pointer dereferenzieren.

Parameterübergabe (4)

```
...

void square(int n, int *s)
{
        *s = n * n;
}

int main()
{
        int result;
        for(int i = 1; i <= 20; i++) {
                square(i, &result);
                cout << result << "\n";
        }
        return 0;
}
```


Parameterübergabe (5)

- Auf diese Art kann man auch Eingabe/Ausgabe-Parameter realisieren:
  - ◇ Man übergibt die Adresse einer bereits initialisierten Variablen.
  - ◇ Die aufgerufene Prozedur kann den Wert dann verändern.
- Damit man sich die Parameterreihenfolge leichter merken kann, sollte man sich an feste Regeln halten, z.B. alle Eingabeparameter vor allen Ausgabeparametern.

Arrays als Parameter (1)

- In C/C++ gibt es eine spezielle Regel für Arrays (Ausnahme):
  - ◇ Arrays werden bei der Parameterübergabe nicht kopiert, sondern es wird die Startadresse übergeben.

    Dies ist immerhin konsistent mit der sonst auch durchgeführten impliziten Umwandlung von einem Array in den entsprechenden Pointer. Die Motivation für diese Regel war aber wohl die Effizienz: Arrays können sehr groß sein, und eine Zuweisung von Arrays erlaubt C ja sonst auch nicht.
- Eine Zuweisung an ein Array-Element ändert also das übergebene Array.

Arrays als Parameter (2)

- Wenn man den Parameter als `const` deklariert, kann das Array in der Prozedur nicht geändert werden.
- Es ist egal, ob man den Parameter als Array oder als Pointer deklariert.

  Kleiner Unterschied: Das Array selbst (nicht sein Inhalt) ist immer konstant. Bei einem Array braucht man die Größe der ersten Dimension nicht anzugeben.
- Die aufgerufene Prozedur erfährt nicht die Größe des Arrays. Man sollte sie ggf. als extra Parameter übergeben.

  Für Strings nicht nötig: Hier gibt es ja die Ende-Markierung `'\0'`.

Arrays als Parameter (3)

- Beispiel:

```
int max(const int a[], int n)
{
        int greatest = a[0];
        for(i = 1; i < n; i++)
                if(a[i] > greatest)
                        greatest = a[i];
        return greatest;
}
```


Referenzen (1)

- Es gibt einige verschiedene Parameter-Übergabe-mechanismen.
- Die häufigsten sind (z.B. in Pascal vorhanden):
  - "Call by Value": Selbst wenn der aktuelle Parameter eine Variable ist, wird ihr Wert übergeben, d.h. in den formalen Parameter kopiert.
  - "Call by Reference" (`var`-Parameter in Pascal): Der aktuelle Parameter muss eine Variable sein, ihre Adresse wird übergeben.

Referenzen (2)

- Vorteil von "call by value":
  - Die Prozedur kann die beim Aufruf angegebene Variable nicht überraschend ändern.

    Wenn man eine Änderungsmöglichkeit geben will, muss man explizit den Adressoperator verwenden.
- Vorteile von "call by reference":
  - Bei großen Werten (mehr als ein `int`, Pointer, etc.) effizienter als "call by value".
  - Für Ausgabeparameter syntaktisch etwas einfacher.

Referenzen (3)

- In C gab es nur "call by value" (mit Ausnahme von Arrays, die "call by reference" übergeben wurden).
- Größere Objekte kann man aber schlecht "call by value" übergeben:
  - Normalerweise könnte der Programmierer selbst mit Zeigern arbeiten.

    In C werden Strukturen (sehr ähnlich zu Objekten) "by value" übergeben. In alten C-Varianten waren solche Parameter verboten.
  - Für die Operator-Syntax funktioniert das aber nicht gut.
- Daher wurden in C++ Referenzen eingeführt.

Referenzen (4)

- Referenzen sind im Prinzip Pointer, die automatisch dereferenziert werden.

  Bei der Initialisierung wird einmal automatisch die Adresse der zugewiesenen Variable bestimmt. Danach sind Änderungen nicht mehr möglich, weil der Pointer ja immer automatisch dereferenziert wird: Auf den Pointer selbst hat man gar keinen Zugriff mehr.

- Beispiel:

```
int i = 1;
int j = 2;
int &r = i; // Entspricht:   int *p = &i;
r = j;      // (*p) = j; Jetzt: i == 2, r == 2
r++;        // (*p)++;   Jetzt: i == 3, r == 3
```


Referenzen (5)

- Mit Parametern eines Referenztyps bekommt man "call by reference" in C++:

```
void set_null(int &n)
{
      n = 0;
}
```

- Beim Programmstück

```
int i = 5;
set_null(i)
cout << i;
```

wird `0` ausgegeben: Der Aufruf von `set_null` setzt die übergebene Variable auf `0`.

Referenzen (6)

- Das ist grundlegend anders als in C: Dort muss der Aufrufer durch Übergabe einer Adresse explizit die Änderung der Variable erlauben.

  Man kann dem Prozeduraufruf in C also ansehen, ob es eine Chance gibt, dass Variablen in dem Aufruf geändert werden. Das gilt in C++ nicht mehr.

- Im Beispiel ist der Name der Prozedur eindeutig. Ansonsten sollte man mit Referenz-Parametern, die geändert werden, sehr vorsichtig sein.

  Die Lesbarkeit des Programms leidet, die Fehlersuche wird erschwert.

Referenzen (7)

- Häufig werden Referenz-Parameter nur verwendet, um größere Objekte ohne Effizienzverlust zu übergeben.
- Man deklariert sie dann als `const`:

```
int square(const int &n)
{
      return n * n;
}
```

- Dies ist nur ein Beispiel, für `int` würde es sich gerade nicht lohnen.

  Klassen und Objekte werden im nächsten Kapitel behandelt.

Referenzen (8)

- Für änderbare Referenz-Parameter muss ein Lvalue übergeben werden, z.B. wäre folgendes falsch:

```
set_null(5); // Falsch.
```

- Bei konstanten Referenz-Parametern ist ein normaler Wert dagegen in Ordnung:

```
int s = square(5);
```

Formal wird der Compiler in diesem Fall eine temporäre Variable anlegen, den Wert 5 dieser Variablen zuweisen, und die Adresse dieser Variablen an die Prozedur übergeben.

Für änderbare Referenzparameter wäre dieses Vorgehen auch möglich, aber dann bewirkt die Zuweisung in der Prozedur nichts: Die temporäre Variable wird nach dem Aufruf wieder gelöscht.

Inhalt

Details: Rückkehr-Adressen (1)

```
...
void print_square(int n)
{
        cout << n;
        cout << " zum Quadrat ist: ";
        cout << n * n;
}
int main()
{
        cout << "Erster Aufruf:\n";
        print_square(15);
        cout << "Zweiter Aufruf:\n";
        print_square(25);
        return 0;
}
```


Details: Rückkehr-Adressen (2)

- Eine Prozedur kann an verschiedenen Stellen in einem Programm aufgerufen werden.
- Sie kehrt jeweils zu der Stelle zurück, von der aus sie aufgerufen wurde (direkt hinter den Aufruf).
- Die CPU hat nur einen "Instruction Pointer" (IP) für die Adresse des aktuell abzuarbeitenden Befehls.
- Damit sie weiß, zu welcher Adresse sie am Ende der Prozedur zurückkehren soll, wird beim Prozeduraufruf die Adresse des nächsten Befehls in einen speziellen Speicherbereich, den "Stack" gelegt.

Details: Rückkehr-Adressen (3)

- Im Beispiel wird beim ersten Aufruf der Prozedur `print_square` in Zeile 12 die Adresse des nächsten Maschinenbefehls nach dem Aufruf (also entsprechend Zeile 13) auf den Stack gelegt.

  Beim zweiten Aufruf (in Zeile 14) wird entsprechend die Adresse des Befehls von Zeile 15 auf den Stack gelegt.

- Am Ende der Prozedur steht ein Befehl, der den Instruktion Pointer in der CPU auf den (obersten) Wert im Stack setzt und diesen Wert vom Stack herunter nimmt ("Rücksprung").

Details: Rückkehr-Adressen (4)

- Nun kann aber auch innerhalb einer Prozedur eine weitere Prozedur aufgerufen werden.
- Deswegen reicht nicht eine einzelne Speicherstelle für die Rücksprungadresse.
- Es entsteht so ein ganzer Stapel (engl. "Stack") von Rücksprungadressen:
  - ◇ Beim Prozeduraufruf wird die Rücksprungadresse auf den Stack gelegt.

    Dies ist der aktuelle Inhalt des Instruction Pointers plus n, wobei n die Länge des Maschinenbefehls für den Prozeduraufruf ist.
  - ◇ Beim Rücksprung wird sie herunter genommen.

Details: Rückkehr-Adressen (5)

```
 (3)  void f(int m)
 (4)  {
 (5)        cout << "f(" << m << ")\n";
 (6)  }
 (7)  void g(int n)
 (8)  {
 (9)        f(n+1);
(10)        f(n+2);
(11)  }
(12)  int main()
(13)  {
(14)        g(10);
(15)        g(20);
(16)        return 0;
(17)  }
```


Details: Rückkehr-Adressen (6)

- Beim ersten Aufruf von g in Zeile 14 wird 15 (bzw. die entsprechende Adresse im Maschinenprogramm) auf den Stack gelegt.
  - ◇ Vor Aufruf von g:

    IP: $\boxed{14}$ Stack:

  - ◇ Nach Aufruf von g (beginnt effektiv in Zeile 9):

    IP: $\boxed{9}$ Stack: 15 $= 14 + 1$

Details: Rückkehr-Adressen (7)

- Innerhalb von `g` wird nun in Zeile 9 die Prozedur `f` aufgerufen.
- Also kommt die Adresse von Zeile 10 (nachfolgender Befehl) auf den Stack, und der Instruction Pointer wird auf die Startadresse von `f` gesetzt (Zeile 5):

IP:	Stack:
5	10
	15

- Nun druckt `f` den Text "`f(11)`".

Details: Rückkehr-Adressen (8)

- Anschließend (Ende von `f`) kehrt die CPU zu der obersten Adresse auf dem Stack zurück (Zeile 10) und nimmt diese Adresse vom Stack:

IP:	Stack:
10	15

- Dann ruft `g` das zweite Mal `f` auf:

IP:	Stack:
5	11
	15

Details: Rückkehr-Adressen (9)

- Bei diesem Aufruf druckt `f` den Text "`f(12)`".
- Anschließend kehrt die CPU zu Zeile 11 zurück (oberste Adresse auf dem Stack):

IP:	Stack:
11	15

- Dann kehrt auch `g` zu Zeile 15 (in `main`) zurück, auch diese Adresse wird vom Stack genommen:

IP:	Stack:
15	

Details: Rückkehr-Adressen (10)

- Anschließend ruft `main` die Prozedur `g` zum zweiten Mal auf:

IP:	Stack:
9	16

- Dies ruft wieder `f` auf:

IP:	Stack:
5	10
	16

- Jetzt wird "`f(21)`" gedruckt. u.s.w.

Aufruf-Schachtelung

- Druckt man am Anfang und am Ende jeder Prozedur einen kurzen Text mit einer öffnenden bzw. schließenden Klammer aus, so erhält man immer eine korrekte Klammerschachtelung:

```
(Anfang von main
    (Anfang von g: n=10
        (Anfang von f: m=11
         Ende von f)
        (Anfang von f: m=12
         Ende von f)
     Ende von g)
    ...
 Ende von main)
```


Inhalt

Rekursive Funktionen (1)

- Eine Funktion kann auch sich selbst aufrufen. Die Funktion und der Aufruf heißen dann "rekursiv".
- Selbstverständlich geht das nicht immer so weiter, sonst erhält man das Gegenstück zu einer Endlosschleife, die Endlosrekursion.
- Endlosrekursionen führen normalerweise schnell zu einem "Stack Overflow".

  Oft ist nur ein relativ kleiner Speicherbereich für den Stack reserviert, z.B. 64 KByte (die Größe kann meist mit einer Option gesteuert werden). Im besseren Fall bemerkt die CPU automatisch den Stack Overflow. Im schlechteren Fall werden einfach Hauptspeicher-Adressen überschrieben.

Rekursive Funktionen (2)

- Beispiel einer rekursiven Funktion ist die Fibonacci-Funktion, für $n \in \mathbb{N}_0$ definiert durch:

$$f(n) := \begin{cases} 1 & \text{für } n = 0,\ n = 1 \\ f(n-1) + f(n-2) & \text{sonst.} \end{cases}$$

  Eine mögliche Veranschaulichung ist, dass $f(n)$ die Anzahl von Kaninchenpärchen zum Zeitpunkt n ist: Man nimmt an, dass keine Kaninchen sterben, deswegen hat man zum Zeitpunkt n mindestens $f(n-1)$ Pärchen. Außerdem haben die zum Zeitpunkt $n-2$ existierenden Kaninchenpärchen jeweils ein Pärchen als Nachwuchs bekommen (man nimmt an, dass sie eine Zeiteinheit brauchen, um geschlechtsreif zu werden).

  Fibonacci-Zahlen werden in der Informatik aber z.B. auch bei der Analyse von AVL-Bäumen benötigt.

Rekursive Funktionen (3)

```
int fib(int n)
{
        if(n < 2)
                return 1;
        else {
                int f1 = fib(n-1);
                int f2 = fib(n-2);
                return f1 + f2;
        }
}
```

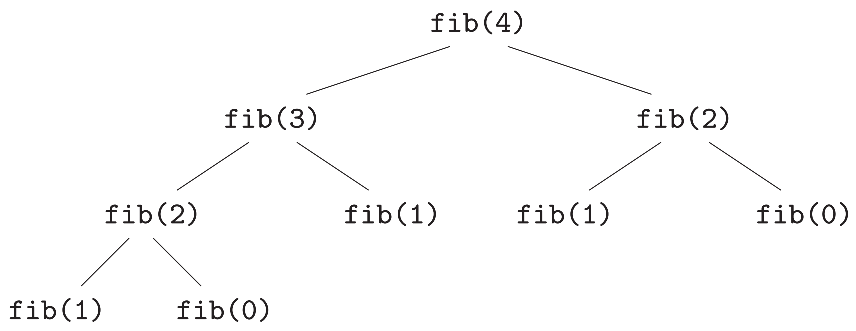
Natürlich hätte man auf die lokalen Variablen `f1` und `f2` verzichten können und einfach "`return fib(n-1) + fib(n-2)`" schreiben. Das wäre normalerweise wohl besser, aber die lokalen Variablen werden in der weiteren Erklärung benötigt.

Rekursive Funktionen (4)

- Die Terminierung ist hier garantiert, weil
  - ◇ der Parameter `n` bei jedem Aufruf um mindestens 1 kleiner wird, und
  - ◇ rekursive Aufrufe nur bei `n` $\geq$ 2 stattfinden.
- Der gleiche Funktionswert wird mehrfach berechnet, man könnte durch eine Umwandlung in eine iterative Berechnung (mit einer Schleife) ohne Doppelung Laufzeit sparen (Übungsaufgabe).

  Das obige Programm hat aber den Vorteil, dass es der mathematischen Definition am nächsten kommt.

Rekursive Funktionen (5)



- Dieser Graph zeigt die Funktionsaufrufe: Z.B. führt der Aufruf `fib(4)` zu den Aufrufen `fib(3)` und `fib(2)`.

Rekursive Funktionen (6)

- Bei rekursiven Prozeduren braucht man gleichzeitig mehrere Kopien der Parameter und der lokalen Variablen:
  - ◇ Sei z.B. der Aufruf für `n` = 2 betrachtet.
  - ◇ Darin wird die Funktion für `n` = 1 aufgerufen.
  - ◇ Wenn dieser Aufruf zurückkehrt, hat `n` wieder den Wert 2 (so dass anschließend `fib(n-2)` zum Aufruf `fib(0)` führt).
  - ◇ Es gibt hier gleichzeitig zwei verschiedene Variablen `n` (eine mit dem Wert 2, eine mit dem Wert 1).

Rekursive Funktionen (7)

- Das gleiche passiert mit den lokalen Variablen `f1` und `f2`:
  - ◇ Sei z.B. der Aufruf `fib(4)` betrachtet.
  - ◇ Er weist `f1` den Wert `3` zu (`fib(3)`).
  - ◇ Anschließend wird `fib(2)` aufgerufen. Dies setzt seine Kopie von `f1` auf `1`.
  - ◇ Wenn der rekursive Aufruf zurückkehrt, hat `f1` wieder den Wert `3`: Es ist eine andere Variable mit gleichem Namen.

Rekursive Funktionen (8)

- Verschiedene Variablen mit gleichem Namen gibt es auch als lokale Variablen in unterschiedlichen Prozeduren (s.o.).
- Die Situation bei der Rekursion ist insofern anders (und komplizierter) als
  - ◇ im Programm syntaktisch nur eine Deklaration steht, und
  - ◇ zur Compile-Zeit nicht bekannt ist, wie viele Kopien der Variablen später zur Laufzeit erforderlich sein werden.

Inhalt

Details: Stackframes (1)

- Der Compiler kann bei rekursiven Funktionen den Variablen also keine festen Adressen zuordnen.

  Die wenigsten Compiler unterscheiden rekursive und nicht-rekursive Funktionen. Alle Funktionen werden als potentiell rekursiv behandelt. In der Sprache Fortran war Rekursion ausgeschlossen, dort können Variablen feste Adressen haben. Bei vielen Prozeduren, die nicht gleichzeitig aktiv sind, wäre das aber eine Speicherplatz-Verschwendung.
- Jeder Prozeduraufruf (Prozedur-Aktivierung, "procedure invocation") benötigt seinen eigenen, frischen Satz von lokalen Variablen.
- Daher bietet es sich an, die lokalen Variablen ebenfalls auf dem Stack abzulegen.

Details: Stackframes (2)

- Typisch ist folgende Reihenfolge:
  - ◇ Zuerst kommen die Werte für die Parameter auf den Stack.

    Und zwar von hinten nach vorne, so dass der erste Parameterwert ganz oben liegt. C erlaubt Prozeduren mit variabler Zahl von Parametern, die aufgerufene Prozedur muss mindestens auf den ersten Parameter zugreifen können.
  - ◇ Dann die Rücksprungadresse.
  - ◇ Anschließend die lokalen Variablen.
- Das ganze heißt Activation Record/Stack Frame.

  Aktivierungs-Records auf dem Stack existieren für alle aufgerufenen, aber noch nicht (durch "`return`", ggf. implizit) beendeten Prozeduren.

Details: Stackframes (3)

1	(Lokale Variable `f1`)	Aktivierungs-Record für `fib(2)`
1	(Lokale Variable `f2`)	
···	(Rücksprung-Adresse)	
2	(Parameter `n`)	
3	(Lokale Variable `f1`)	Aktivierungs-Record für `fib(4)`
–	(Lokale Variable `f2`)	
···	(Rücksprung-Adresse)	
4	(Parameter `n`)	
⋮		

Details: Stackframes (4)

- Bei der Rückkehr aus einer Prozedur
  - ◇ wird der Speicherplatz für die lokalen Variablen auf dem Stack wieder freigegeben,
  - ◇ und die Rückkehradresse vom Stack genommen.
  - ◇ Der Aufrufer entfernt dann noch die Parameterwerte.
- Der Speicherplatz für die lokalen Variablen wird bei der Rückkehr der Prozedur freigegeben, und beim Aufruf der nächsten Prozedur wiederverwendet.

Lebensdauer von Variablen

- Es ist ein Fehler, die Adresse einer lokalen Variablen einer globalen Variablen zuzuweisen.

  Oder über den Funktionswert oder einen Prozedurparameter einer aufrufenden Prozedur zur Verfügung zu stellen.
- Da die Lebensdauer der lokalen Variablen bei der Rückkehr aus ihrer Prozedur endet, zeigen solche Pointer dann auf nicht mehr existierende Variablen.
- Zuweisungen über diese Pointer können andere Variablen oder sogar Rückkehradressen zerstören.

Globale/static Variablen (1)

- Im Unterschied zu lokalen Variablen gilt für globale Variablen:
  - ◇ Von ihnen gibt es jeweils nur ein Exemplar (nicht mehrere Kopien).
  - ◇ Der Compiler kann ihnen eine feste Adresse zuweisen.
  - ◇ Sie existieren für die ganze Programm-Laufzeit.

Globale/static Variablen (2)

- Falls man innerhalb einer Prozedur eine Variable mit den obigen Eigenschaften globaler Variablen haben will, muss man sie mit dem Schlüsselwort "`static`" markieren: `static int n;`
- Auch wenn die Prozedur verlassen wird und später neu aufgerufen, behält `n` seinen Wert.

  Man sollte `n` initialisieren (für ersten Aufruf).
- Auch bei rekursiven Aufrufen gibt es nur eine Variable `n`.
- Man kann die Adresse von `n` weitergeben.

Globale/static Variablen (3)

- Im Unterschied zu globalen Variablen sind `static`-Variablen nur innerhalb ihrer Prozedur zugreifbar,

  Über den Namen. Zugriffe über Zeiger sind natürlich möglich, wenn man die Adresse weitergegeben hat.
- `static`-Variablen werden häufig für initialisierte Arrays benutzt.

  Die Variable kann dann in einem initialisierten Speicherbereich abgelegt werden. Man spart so einige Maschinenbefehle.
- Die normalen lokalen Variablen werden in C auch automatische Variablen genannt.

  Es gibt ein Schlüsselwort "`auto`", das ist aber überflüssig.

Zeiger auf Funktionen (1)

- Es ist auch möglich, eine Funktion (genauer ihre Start-Adresse) in einer Variablen zu speichern.
- Das ist nützlich, wenn ein Programm z.B. viele von Benutzer-Kommandos hat, und zu jedem Kommando eine Funktion, die es ausführt.
- Man kann dann in einer Tabelle (einem Array) Objekte speichern, die jeweils dem Kommandonamen und die auszuführende Funktion enthalten.

  Man könnte auch einen `switch` verwenden, in dem jeder Fall einem Kommando entspricht. Wenn man aber über die Kommandos ohnehin Daten speichern muss, ist es eleganter, dort alles zusammen zu haben.

Zeiger auf Funktionen (2)

- Zeiger auf Funktionen waren in C wichtig, wenn man "objektorientiert" programmieren wollte.
- In C++ sind sie seltener explizit nötig, weil sie automatisch intern z.B. zum Überschreiben von ererbten Methoden in Subklassen verwendet werden (siehe Kapitel 12).

  Auch Templates (siehe Kapitel 15) sind ein neues Sprachkonstrukt in C++, für das man in C wahrscheinlich Funktions-Zeiger verwendet hätte (oder Makros).
- Es ist aber ein wichtiges Konzept, dass man auch Funktionen als Werte behandeln kann.

Zeiger auf Funktionen (3)

- Wie immer deklariert man Variablen, indem man einen Ausdruck angibt, der den Typ auf der linken Seite liefert:

```
void (*f)(int);
```

- Dann ist `f` also ein Zeiger auf eine Funktion, die einen `int`-Parameter hat, und keinen Wert zurückliefert (Rückgabetyp `void`).
- Die in `f` gespeicherte Funktion ruft man dann entsprechend so auf:

```
(*f)(5);
```


Zeiger auf Funktionen (4)

- Man beachte, dass in Deklaration und Aufruf die Klammern bei `(*f)` nötig sind, sonst würde das Ergebnis des Funktionsaufrufs dereferenziert.

  Der Funktionsaufruf hat höhere Priorität als der Dereferenzierungsoperator `*`.
- Seit `g` passend zum Typ von `f` deklariert:

```
void g(int i)
{
    cout << 2 * i << '\n';
}
```


Zeiger auf Funktionen (5)

- Wird der Name einer Funktion nicht in einem Funktionsaufruf verwendet, so steht er automatisch für ihre Adresse.
- Da `g` passenden Typ hat (Rückgabetyp, Parameteranzahl und -Typen sind identisch), kann man sie in der Variable `f` mit folgender Zuweisung speichern:

```
f = g;
```

- Durch den Aufruf

```
(*f)(5);
```

  wird nun `g` ausgeführt (mit Eingabewert 5).