

8. Funktionen Teil I

Prof. Dr. Markus Gross
Informatik I für D-ITET (WS 03/04)

- Grundlagen
- Funktionsprototypen
- Pass by Value
- Pointer Argumente
- Pointer auf Funktionen
- Rekursion



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Copyright: M. Gross, ETHZ, 2003

2



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Definition und Deklaration

- Die Verwendung einer Funktion in C++ erfordert folgende Komponenten
 - ◆ Funktionsprototyp
 - ◆ Funktionsdefinition
 - ◆ Funktionsaufruf
- Unterscheidung zwischen Deklaration und Definition
- Prototypen sind oft in Header Files deklariert
- Prototyp kann auch zu Beginn des Quellfiles und ausserhalb anderer Funktionen stehen
- Wir unterscheiden zwischen
 - ◆ Funktionen ohne Rückgabewert - Typ `void`
 - ◆ Funktionen mit Rückgabewert

Definition

- Allgemeine Form der C++ Funktionsdefinition


```
typeName functionName (argumentList)
{
    statements;
    return value;
}
```
- Rückgabewert muss korrekten Typ besitzen oder umwandelbar sein
- Arrays können nicht direkt übergeben werden
- Structs können direkt übergeben werden
- Bei Uebergabe des Wertes wird eine *temporäre Kopie* der Argumente angelegt
- Im Allgemeinen ineffizient



Beispiel_1: Prototypen

```
// protos.cpp -- use prototypes and function calls
#include <iostream>
using namespace std;

void cheers(int); // prototype: no return value
double cube(double x); // prototype: returns a double

int main(void)
{
    cheers(5); // function call
    cout << "Give me a number: ";
    double side;
    cin >> side;
    double volume = cube(side); // function call
    cout << "A " << side << "-foot cube has a volume of ";
    cout << volume << " cubic feet.\n";
    cheers(cube(2)); // prototype protection at work
    return 0;
}

void cheers(int n)
{
    for (int i = 0; i < n; i++)
        cout << "Cheers! ";
    cout << "\n";
}

double cube(double x)
{
    return x * x * x;
}
```

Prototypen

Zuweisung mit
Funktion

Prototypen

- Prototypen beschreiben die Funktionsschnittstelle
- Compiler benötigt diese Information für diverse Ueberprüfungen
 - ◆ Rückgabetyt
 - ◆ Anzahl Argumente
 - ◆ Typ der Argumente
 - ◆ *Statische Typenprüfung* zur Compilezeit
- Eine Funktion ohne Prototyp muss VOR dem ersten Aufruf in der Quelldatei definiert sein
- Ein Prototyp ist eine Anweisung (endet mit Semikolon)
- Argumentnamen werden nicht benötigt
 - ◆ `double cube(double x);`
 - ◆ `double cube(double);`
- Programmierpraxis: Kopiere den Header als Prototyp

Funktionsargumente

- Argumente können einer Funktion auf zwei Arten übergeben werden
 - ◆ Die Werte der übergebenen Variablen werden kopiert (*call by value*)
 - ◆ Die Adressen der Variablen werden kopiert (*call by reference*)
- Beim Aufruf


```
double volume = cube(side);
```

 wird der Inhalt von `side` in das Argument von `cube` kopiert
 - ◆ Der Inhalt der Variablen der aufrufenden Funktion bleibt erhalten
- Funktions-Argumente werden wie *lokale Variablen* innerhalb der Funktion benutzt
- Ihr Gültigkeitsbereich ist auf die Funktion beschränkt
- Nach Rücksprung erlöschen diese Variablen

Call (Pass) by Value

```
...
double cube(double x);
```

```
int main()
{
```

```
...
double side = 5;
double volume = cube(side);
...
}
```

```
double cube(double x)
{
return x*x*x;
}
```

erstellt eine neue
Variabel `side`
mit dem Wert 5

→ 5 Original-
wert
`side`

→ übergibt den Wert 5
der `cube()` Funktion

erstellt eine neue
Variabel `x`
mit dem Wert 5

→ 5 kopierter
Wert
`x`

Funktionen und Arrays

- Arrays können *nicht by value* übergeben werden
- Ein Array-Argument wird wie folgt definiert


```
int sum_arr(int arr[], int n);
```

 - ◆ Arrayname `arr` ist Pointer auf Anfangsadresse (`&arr[0]`)
 - ◆ `[]` bedeuten, dass die Arraygröße zunächst beliebig sein kann
- Im Funktionskörper kann `arr` wie ein gewöhnliches Array verwendet werden
- Alternativ ist auch die Verwendung eines Pointers als Argument korrekt


```
int sum_arr(int *arr, int n);
```
- Es wird also nur die Anfangsadresse des Arrays übergeben (*call by reference*)

Funktionen und Arrays

- Bei einem *call by reference* arbeitet die Funktion mit den Originaldaten
 - ◆ Vorteil: Geschwindigkeit
 - ◆ Nachteil: Datenkonsistenz
- Anfangsadresse und Grösse können beliebig frei gewählt werden
- Müssen nicht mit Kenngrössen der Originaldaten übereinstimmen
- Es gilt nach wie vor:


```
arr[i] == *(arr+i);
&arr[i] == arr+i;
```



Beispiel_2: Arrays

```
// arrfun2.cpp -- functions with an array argument
#include <iostream>
using namespace std;
const int ArSize = 8;
int sum_arr(int arr[], int n);
int main()
{
    int cookies[ArSize] = {1,2,4,8,16,32,64,128};

    cout << cookies << " = array address, ";

    cout << sizeof cookies << " = sizeof cookies\n";
    int sum = sum_arr(cookies, ArSize);
    cout << "Total cookies eaten: " << sum << "\n";
    sum = sum_arr(cookies, 3); // a lie
    cout << "First three eaters ate " << sum << " cookies.\n";
    sum = sum_arr(cookies + 4, 4); // another lie
    cout << "Last four eaters ate " << sum << " cookies.\n";
    return 0;
}

// return the sum of an integer array
int sum_arr(int arr[], int n)
{
    int total = 0;
    cout << arr << " = arr, ";
    // some systems require a type cast: unsigned (arr)

    cout << sizeof arr << " = sizeof arr\n";
    for (int i = 0; i < n; i++)
        total = total + arr[i];
    return total;
}
```

Anfangsadresse übergeben

Andere Anfangsadresse

Ueberschreibschutz

- *Call by reference* Argumente müssen vor unabsichtlichem Ueberschreiben geschützt werden
 - Kann durch `const` Argument erreicht werden
- ```
void array_show(const double arr[],
 int n);
```
- Funktion kann dann nur lesend auf das Array zugreifen
  - Originaldaten müssen nicht `const` sein, d.h.:
    - ◆ Ist ein Funktionsargument `const`, kann die Funktion sowohl konstante, als auch nichtkonstante Daten verarbeiten
    - ◆ Ist ein Funktionsargument nicht `const`, kann die Funktion nur nichtkonstante Daten verarbeiten



## const Pointers

Kann im Kontext von Pointern auf zwei Arten verwendet werden:

### I. Zeiger auf konstante Variable

```
int age = 39;
```

```
const int *pt = &age;
```

- ◆ Der Wert wird vom Zeiger als konstant angenommen und kann nicht über den Zeiger verändert werden:

```
*pt+=1; // invalid
```

- ◆ Der Wert kann jedoch über die Variable `age` verändert werden

- ◆ Ist die Originalvariable `const`, dann muss es der Zeiger ebenfalls sein

```
const int age = 39;
```

```
const int *cpt = &age;
```

```
int *pt = &age; // invalid
```

## const Pointers

### II. Konstanter Zeiger auf Variable

- ◆ Hierbei soll die Adresse konstant bleiben

```
int age = 39;
int * const pt = &age;
```

- ◆ Der Zeiger wird als konstant angenommen und kann nicht mehr verändert werden `pt=&newage; // invalid`

### III. Kombination von beiden

- ◆ Konstanter Zeiger auf konstante Variable möglich

```
const int age = 39;
const int * const pt = &age;
```

- ◆ Sowohl `pt` als auch `*pt` sind `const`

## Funktionen und Strings

- Strings können in C++ auf 3 Arten repräsentiert werden
  - ◆ Als Array of char
  - ◆ Als String Konstante "C++course";
  - ◆ Als Pointer auf Anfangsadresse des String
- Strings können als Argumente an Funktionen übergeben werden
 

```
char ghost[15] = "galloping";
char * str = "galloping";
int n1 = strlen(ghost);
int n2 = strlen(str);
int n3 = strlen("galloping");
```
- 3 Formen der Uebergabe sind gleichwertig
- Es wird jeweils die *Anfangsadresse des String* übergeben (call by reference)

## Beispiel\_3: String als Argument

```
// strgfun.cpp -- functions with a string argument
#include <iostream>
using namespace std;
int c_in_str(const char * str, char ch);
int main()
{
 char mmm[15] = "minimum"; // string in an array

 char *wail = "ululate"; // wail points to string

 int ms = c_in_str(mmm, 'm');
 int us = c_in_str(wail, 'u');
 cout << ms << " m characters in " << mmm << "\n";
 cout << us << " u characters in " << wail << "\n";
 return 0;
}

// this function counts the number of ch characters
int c_in_str(const char * str, char ch)
{
 int count = 0;
 while (*str) // quit when *str is '\0'
 {
 if (*str == ch)
 count++;
 str++; // move pointer to next char
 }
 return count;
}
```

Arrayname = Anfangsadresse

Pointer=Anfangsadresse

## Beispiel\_4: String als Rückgabewert

```
// strgback.cpp -- a function returning a pointer to char
#include <iostream>
using namespace std;
char * buildstr(char c, int n); // prototype
int main()
{
 int times;
 char ch;

 cout << "Enter a character: ";
 cin >> ch;
 cout << "Enter an integer: ";
 cin >> times;
 char *ps = buildstr(ch, times);
 cout << ps << "\n";
 delete [] ps; // free memory
 ps = buildstr('+', 20); // reuse pointer
 cout << ps << "-DONE-" << ps << "\n";
 delete [] ps; // free memory
 return 0;
}

// builds string made of n c characters
char * buildstr(char c, int n)
{
 char * pstr = new char[n + 1];
 pstr[n] = '\0'; // terminate string
 while (n-- > 0)
 pstr[n] = c; // fill rest of string
 return pstr;
}
```

Anfangsadresse

Speicherfreigabe

Speicherallokation



## Funktionen und Structs

- Structs können in C++ auf 3 Arten übergeben werden
  - ◆ Als Kopie der Originalstruktur (call by value)
  - ◆ Als Pointer auf Originalstruktur (call by reference)
  - ◆ Als *Referenz* auf Originalstruktur (call by reference) - später
- Call by value ist mit Kopieraufwand verbunden
- Programmierregel:
  - ◆ Kleine Strukturen: call by value
  - ◆ Grosse Strukturen: call by reference
- Rückgabetypen können ebenfalls Strukturen sein
- Seien `polar` und `rect` Stukturen
 

```
polar rect_to_polar(rect xy);
```
- Argumenttyp und Rückgabetyt sind Strukturen

## Beispiel\_5: Struct - by Value

```
// structfun.cpp -- exerpt
#include <iostream>
#include <cmath>
using namespace std;

// structure templates
struct Polar
{
 double distance; // distance from origin
 double angle; // direction from origin
};
struct rect
{
 double x; // horizontal distance from origin
 double y; // vertical distance from origin
};

// prototypes
Polar rect_to_polar(rect xypos);
void show_polar(Polar dapos);

// convert rectangular to polar coordinates
Polar rect_to_polar(rect xypos)
{
 Polar answer;
 answer.distance =
 sqrt(xypos.x * xypos.x + xypos.y * xypos.y);
 answer.angle = atan2(xypos.y, xypos.x);
 return answer; // returns a Polar structure
}
```

Strukturdeklarationen

Prototypen

Funktionsdefinition

## Funktionen und Structs

- Call by reference kann über Pointer erfolgen
- Hierbei wird die Adresse der Struktur übergeben
- Gleiche Probleme der Datenkonsistenz wie bei Arrays
- Verwendung des `const` Qualifiers, wo immer möglich
- Rückgabetyt sollte dabei ebenfalls Pointer auf Struktur sein
  - ◆ Effizienz
- Grosse Strukturen: call by reference
- Argumenttypen müssen dabei Pointer auf Strukturen sein
- Parameterübergabe erfolgt durch `&`-Operator

## Beispiel\_6: Struct - by Reference

```
// structptr.cpp -- wie Beispiel 5
// prototypes
void rect_to_polar(const rect * pxy, Polar * pda);
void show_polar (const Polar * pda);

int main()
{
 rect rplace;
 Polar pplace;

 cout << "Enter the x and y values: ";
 while (cin >> rplace.x >> rplace.y)
 {
 rect_to_polar(&rplace, &pplace); // pass addresses
 show_polar(&pplace); // pass address
 cout << "Next two numbers (q to quit): ";
 }
 return 0;
}

// convert rectangular to polar coordinates
void rect_to_polar(const rect * pxy, Polar * pda)
{
 pda->distance =
 sqrt(pxy->x * pxy->x + pxy->y * pxy->y);
 pda->angle = atan2(pxy->y, pxy->x);
}
```

Const pointer Argumente

Adresse übergeben

Zugriff auf Members

## Rekursion

- Funktionen können sich im Prinzip selbst aufrufen
- Diese Fähigkeit heisst *Rekursion*
- Rekursion ist ein mächtiger und tiefgründiger Ansatz zur Lösung vieler algorithmischer Probleme
  - ◆ Beispiel: Suchen oder Sortieren
  - ◆ Divide and Conquer Strategien (siehe Informatik II)
- *Rekursion* darf nicht mit *Iteration* verwechselt werden

```
void recurs(argumentList)
{
 statements1;
 if (test)
 recurs(arguments);
 else
 statements2;
}
```

## Rekursion

- Zur Begrenzung der Rekursionstiefe benötigt man eine Abbruchbedingung
- Ansonsten endlose Rekursion → Abbruch
- Jeder Funktionsaufruf benötigt einen eigenen *Stack Frame*
- Es wird jeweils ein neuer Satz lokaler Variablen angelegt
- Bei zu hoher Rekursionstiefe kann es zu einem *Stack Overflow* kommen

## Beispiel\_7: Rekursion

```
// recur.cpp -- use recursion
#include <iostream>
using namespace std;

void countdown(int n);

int main()
{
 countdown(4); // call the recursive function
 return 0;
}

void countdown(int n)
{
 cout << "Counting down ... " << n << "\n";
 if (n > 0)
 countdown(n-1); // function calls itself
 cout << n << ": Kaboom!\n";
}
```

Aufruf aus Hauptfunktion

Rekursiver Aufruf

## Zeiger auf Funktionen

- Funktionen haben eine Anfangsadresse im Programmspeicher
- Funktionen können andere Funktionen sowohl als Argumente verwenden, als auch als Rückgabewerte
- Hierbei wird grundsätzlich mit Zeigern gearbeitet, welche die Funktionsadresse speichern
- *Function Pointers*
- Erlauben die dynamische Uebergaben von Funktionen zur Laufzeit
- Wir benötigen dazu
  - ◆ Funktionsadresse
  - ◆ Pointer auf Funktionsadresse
  - ◆ Aufruf der Funktion über Pointer

## Funktionsadressen

- Die *Adresse* der Funktion ist durch den *Funktionsnamen* gegeben
- Es sei `think(void)` eine Funktion
 

```
process(think); // Uebergabe der Adresse
thought(think()); // Uebergabe des Rueckgabewertes
```
- Pointer auf Funktion wird wie folgt deklariert
 

```
double (*pf)(int);
```

  - ◆ Pointer zeigt auf Funktion mit einem `int` Argument, welche Rückgabewert vom Typ `double` besitzt
- Nicht verwechseln mit
 

```
double *pf(int);
```

  - ◆ Prototyp einer Funktion mit einem `int` Argument, welche Rückgabewert vom Typ Pointer auf `double` besitzt

## Funktionsadressen

- Pointer kann nun mit Funktionsadresse initialisiert werden
 

```
double pam(int);
double (*pf)(int);
pf = pam;
```

  - ◆ Argumentanzahl, Typen und Rückgabetypen müssen übereinstimmen
- Funktionsaufruf über Pointer ist recht einfach
 

```
double d = pf(8);
```

  - ◆ Wird wie ein Funktionsname verwendet
- Allgemein: Prototypendeklaration ist recht schwierig



## Beispiel\_9: Funktionspointer

```
// fun_ptr.cpp -- pointers to functions-exerpt
#include <iostream>
using namespace std;
double betsy(int);
double pam(int);

// second argument is pointer to a type double function that
// takes a type int argument
void estimate(int lines, double (*pf)(int));

int main()
{
 int code;

 cout << "How many lines of code do you need? ";
 cin >> code;
 cout << "Here's Betsy's estimate:\n";
 estimate(code, betsy);
 cout << "Here's Pam's estimate:\n";
 estimate(code, pam);
 return 0;
}

double betsy(int lns)
{
 return 0.05 * lns;
}

void estimate(int lines, double (*pf)(int))
{
 cout << lines << " lines will take ";
 cout << (*pf)(lines) << " hour(s)\n";
}
```

Argument ist Function  
Pointer

Aufruf mit  
Funktionsnamen