Pointer auf Funktionen

- Eine Funktion hat eine Adresse im Memory
- Die Deklaration eines Funktionspointers braucht zusätzliche Informationen wie z.B.
  - Anzahl Parameter
  - Parametertyp
  - Return Typ

  ```
  int (*pfunction) (int);
  ```
- Die Deklaration deklariert einen Funktionspointer mit dem Namen "`pfunction`" mit einem Parameter vom Typ `int` und einem Return Typ vom Typ `int`

Pointer auf Funktionen: Beispiele

```
int sum(int a, int b);     // Calculates a+b
int (*pfun)(int, int) = sum;
int result = pfun(55, 65);
..
int product(int a, int b); // Calculates a*b
pfun = product;
Result = pfun(6, 15);
```

Pointer auf Funktionen: Beispiele

```c
#include <stdio.h>

// Function prototypes
int sum(int, int);
int product(int, int);
int difference(int, int);
int main(void)
{
  int a = 10; // Initial value for a
  int b = 5; // Initial value for b
  int result = 0; // Storage for results
  int (*pfun)(int, int); // Function pointer declaration

  pfun = sum; // Points to function sum()
  result = pfun(a, b); // Call sum() through pointer
  printf("pfun = sum result = %2d\n", result);

  pfun = product; // Points to function product()
  result = pfun(a, b); // Call product() through pointer
  printf("pfun = product result = %2d\n", result);

  pfun = difference; // Points to function difference()
  result = pfun(a, b); // Call difference() through pointer
  printf("pfun = difference result = %2d\n", result);

  return 0;
}
```

Array mit Pointer auf Funktionen

```
int (*pfunctions[10]) (int);
```

```
// Function prototypes
int sum(int, int);
int product(int, int);
int difference(int, int);

int main(void)
{
  int a = 10;                          // Initial value for a
  int b = 5;                           // Initial value for b
  int result = 0;                      // Storage for results
  int (*pfun[3])(int, int);            // Function pointer array declaration

  // Initialize pointers
  pfun[0] = sum;
  pfun[1] = product;
  pfun[2] = difference;

  // Execute each function pointed to
  for(int i = 0 ; i < 3 ; ++i)
  {
    result = pfun[i](a, b);            // Call the function through a pointer
    printf("result = %2d\n", result);  // Display the result
  }
}

int sum(int x, int y)
{
```

Funktionsparameter als Funktionspointer

```c
// Function prototypes
int sum(int,int);
int product(int,int);
int any_function(int(*pfun)(int, int), int x, int y);

int main(void)
{
  int a = 10;                       // Initial value for a
  int b = 5;                        // Initial value for b
  int result = 0;                   // Storage for results
  int (*pf)(int, int) = sum;        // Pointer to function

  // Passing a pointer to a function
  result = any_function(pf, a, b);
  printf("result = %2d\n", result );

  // Passing the address of a function
  result = any_function(product,a, b);
  printf("result = %2d\n", result );
}

// Definition of a function to call a function
int any_function(int(*pfun)(int, int), int x, int y)
{
  return pfun(x, y);
}
```

static Variablen in Funktionen

```c
// Function prototypes
void test1(void);
void test2(void);

int main(void)
{
  for(int i = 0 ; i < 5 ; ++i)
  {
    test1();
    test2();
  }
  return 0;
}

// Function test1 with an automatic variable
void test1(void)
{
  int count = 0;
  printf("test1   count = %d\n", ++count );
}

// Function test2 with a static variable
void test2(void)
{
  static int count = 0;
  printf("test2   count = %d\n", ++count );
}
```

```
test1 count = 1
test2 count = 1
test1 count = 1
test2 count = 2
test1 count = 1
test2 count = 3
test1 count = 1
test2 count = 4
test1 count = 1
test2 count = 5
```

Rekursion

- Eine Funktion ruft sich selber auf

```
void looper(void)
{
  printf(“Looper function called.\n”);
  Looper();
}
```

Führt zu einer undefinierter Anzahl von Printouts

Rekursion (2) – Beispiel: Grösster gemeinsamer Teiler

- *Rekursiv*:

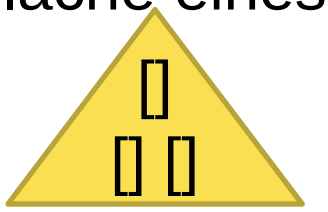
```
int greatest_common_divisor(int x, int y)
{
  return y == 0 ? X : greatest_common_divisor(y, x % y);
}
```

- *Iterativ*:

```
int greatest_common_divisor(int x, int y)
{
  while (true)
  {
    if (y == 0) break;
    int remainder = x % y;
    x = y;
    y = remainder;
  }
  return x;
}
```

Rekursion (3) – Vor und -Nachteile

- *Nachteile*
  - Weniger effizient
  - Gefahr von Stack Overflows
  - Funktionsparameter, Variablen und Return Adressen werden auf dem Stack gespeichert
- *Vorteile*:
  - Rekursive Funktionen sind in der Regel kürzer und eleganter
  - Es gibt Aglorithmen wie z.B. die Berechnung der Permutation die nur sehr schwer iterativ zu programmiern sind.
- Probiere es aus: Berechne die Fläche eines Dreiecks anhand einer gegebenen Breite.

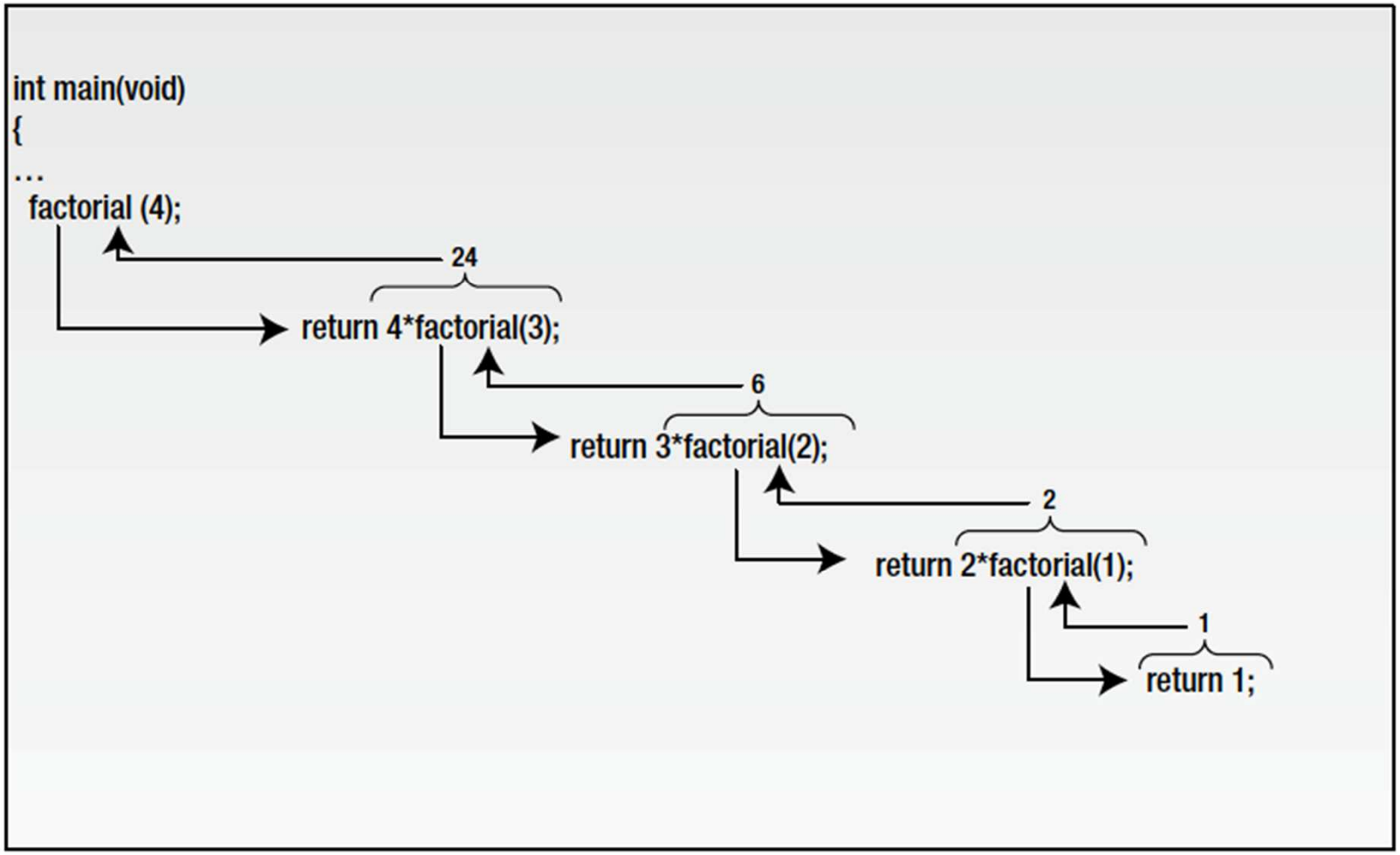


Breite = 2
Fläche = 3

Rekursion (4) – Beispiel: Fakultät (n!)

```c
#include <stdio.h>

unsigned long long factorial(unsigned long long);

int main()
{
    unsigned long long number = 0LL;
    printf("Enter an integer value: ");
    scanf("%llu", &number);
    printf("The factorial of %llu is %llu\n", number, factorial(number));
}

unsigned long long factorial(unsigned long long n)
{
    if (n < 2LL)
        return n;

    return n*factorial(n-1LL);
}
```

Rekursion (5) – Beispiel: Fakultät (n!)



Aufgabe: Othello (Spiel)

- Mache dich mit dem Spiel vertraut: https://de.wikipedia.org/wiki/Othello_(Spiel)
- *Analyse*
  - Bestimme die Programmstruktur
  - Definiere die Funktionen (Name, Parameter, Rückgabewerte)
  - Erstelle ein Flussdiagramm
- *Design*
  - Erstelle den Code
  - Dokumentiere den Code