

Informatik I – Übung 12

Pascal Schärli

pascscha@student.ethz.ch

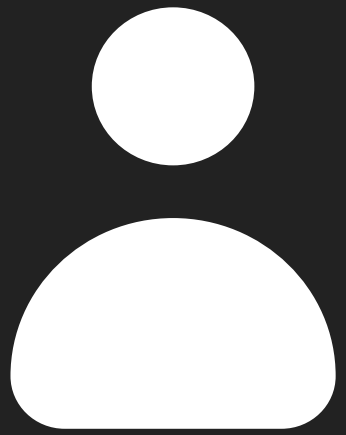
24.05.2019

ETH zürich

Was gibts heute?

- Self-Asessment
- Best-Of Vorlesung
  - Memory Management
- Vorbesprechung
- Prüfungstipps

Self-Assesment



Pascal Schärli *24.05.2019*

ETH zürich

1 Pointers

```
int a = 4;
int& b = a;
int* c = &a;
```

Ausdruck	Ungültig	4	Adresse von a	Adresse von c
`std::cout << b;`		✓		
`std::cout << *b;`	✓			
`std::cout << &b;`			✓	
`std::cout << c;`			✓	
`std::cout << *c;`		✓		
`std::cout << &c;`				✓

2 Iteratoren

```
// PRE:  The interval [begin, end) is valid
//        and not empty
// POST: Returns the maximum value of the
//         interval
```

```
int f(const int* begin, const int* end) {
    int val1 = *begin;
    if (++begin != end) {
        const int val2 = f(begin, end);
        if (val2 > val1) {
            val1 = val2;
        }
    }
    return val1;
}

int g(const int* begin, const int* end) {
    int val = *begin;
    for (const int* it=++begin; it != end; ++it) {
        if (*it > val) {
            val = *it;
        }
    }
    return val;
}
```

- f und g haben dieselbe Vorbedingung
- f und g haben dieselbe Nachbedingung

2 Iteratoren

```
int f(const int* begin, const int* end) {
    int val1 = *begin;
    if (++begin != end) {
        const int val2 = f(begin, end);
        if (val2 > val1) {
            val1 = val2;
        }
    }
    return val1;
}
```

```
{
    const int my_size = 7;
    int* my_values = new int[my_size]{ 3, 8, 1,
                                       12, 20, 2, 6 };
    int result = f(my_values, my_values+my_size);
}
```

- Was ist der Wert der Variable *result* nach dem Aufruf? 20
- Wie oft wird f insgesamt aufgerufen? 7

3 Felder und Arrays

```
int* a = new int[8]{3, 1, -3, -2, 10, 0, 0, 0};
```

3	1	-3	-2	10	0	0	0

```
a[7] = 5;
*(a + 3) = 4;
```

3	1	-3	**4**	10	0	0	**5**

```
int* p = a + 4;
*(a + a[3] + *(p - 2)) = *(p + 3) - 9;
```

3	**-4**	-3	4	10	0	0	5

4 Structs und Operatoren

```cpp
#include<iostream>

struct vector_2 { double x; double y; };

// POST: the scalar product of v and w is returned
double operator* (vector_2 v, vector_2 w)
{
    return v.x * w.x + v.y * w.y;
}

// POST: the scaled vector lambda * v is returned
vector_2 operator* (double lambda, vector_2 v) {
    vector_2 result;
    result.x = lambda * v.x;
    result.y = lambda * v.y;
    return result;
}

int main() {
    vector_2 v = {1.0, 2.0}; vector_2 w = {3.0, 4.0};
    std::cout << v * w << std::endl; // 11
    vector_2 u = 5.0 * v;
    std::cout << '(' << u.x << ", " << u.y << ')' << "\n"; // (5, 10)
    return 0;
}
```

5 Dynamische Datentypen

```
1  class list {
2  private :
3      list_node* first_node ;
4  public :
5      ...
6      // PRE: * this is not empty
7      // POST: the key of the last element in * this is returned
8      int last () const {
9          assert (first_node != nullptr);
10         const list_node* p = first_node;
11         while ( p->next != nullptr)
12             p = p->next;
13         return p->key;
14     }
15
16     // PRE: * this contains at least two elements
17     // POST: the second element is removed from * this
18     void remove_second ()
19     {
20         assert ( first_node != nullptr );
21         assert ( first_node->next != nullptr );
22         list_node* p = first_node->next;
23         first_node->next = first_node->next->next;
24         delete p;
25     }
26 };
```



Vorlesung

New & Delete

- Mit dem Keyword *new* kann man Pointer erstellen, welche auf Daten zeigen die niemals out of scope gehen.
- Im Gegenzug muss man selber die Daten mit dem Keyword *delete* löschen falls man diese nicht mehr braucht.
- Jedes *new* braucht ein *delete*

New & Delete

```cpp
#include <iostream>

int* create_value(){
    int* x = new int(5);
    return x;
}

int main(){
    int* ptr = create_value();

    std::cout << *ptr << "\n";

    delete ptr;

    return 0;
}
```



```
5
```

```
0 1 2 3 4
```



```cpp
#include <iostream>

int* create_array(){
    int* a = new int[5];

    for(unsigned int i = 0; i < 5; i++){
        a[i] = i;
    }

    return a;
}

int main(){
    int* a = create_array();

    std::cout << *a << " ";
    std::cout << *(a + 1) << " ";
    std::cout << *(a + 2) << " ";
    std::cout << *(a + 3) << " ";
    std::cout << *(a + 4) << " ";

    delete[] a;

    return 0;
}
```

New & Delete

- Innerhalb von Klassen können auch Elemente mit *new* Erstellt werden.
- Diese müssen spätestens im destruktor (~Vec) gelöscht werden

```
1  #include <iostream>
2
3  class A {
4      int* array;
5  public:
6      A(unsigned int size) {
7          std::cout << "create\n";
8          array = new int[size];
9      }
10
11     ~A() {
12         std::cout << "delete\n";
13         delete[] array;
14     }
15 };
16
17 int main(){
18     A a(5);
19     a.array[1] = 0;
20     return 0;
21 }
```

Array wurde mit *new* erstellt

Array muss im destructor gelöscht werden.

"create"
"delete" (out of scope)

Memory Management

```
class Cell {
  int value;
public:

  Cell(): value(0) {
    std::cerr << "default constructor" << std::endl;
  }

  Cell(int val): value(val) {
    std::cerr << "overloaded constructor: value=" << value << std::endl;
  }
};
```

```
void step1() {
  Cell demo(1);
  Cell* demo_ptr = new Cell;
}
```

Was ist der Output von *step1*?

```
default constructor
default constructor
```



```
overloaded constructor: value=1
overloaded constructor: value=0
```



```
overloaded constructor: value=1
default constructor
```



```
default constructor
overloaded constructor: value=0
```



Memory Management

```
class Cell {
  int value;
public:

  Cell(): value(0) {
    std::cerr << "default constructor" << std::endl;
  }

  Cell(int val): value(val) {
    std::cerr << "overloaded constructor: value=" << value << std::endl;
  }
};
```

```
void step1() {
  Cell demo(1);
  Cell* demo_ptr = new Cell;
}
```



```
overloaded constructor: value=1
default constructor
```

Memory Management

```
class Cell {
  int value;
public:

  Cell(): value(0) {
  //  std::cerr << "default constructor" << std::endl;
  }

  Cell(int val): value(val) {
  //  std::cerr << "overloaded constructor: value=" << value << std::endl;
  }

  ~Cell() {
    std::cerr << "destructor: value=" << value << std::endl;
  }
};
```

```
void step1() {
  Cell demo(1);
  Cell* demo_ptr = new Cell;
}
```

Was ist der Output von *step1*?

```
destructor value=1
destructor value=0
```



```
destructor value=0
destructor value=1
```



```
destructor value=1
```



```
destructor value=0
```



Memory Management

Die mit *new* erstellte Klasse geht nicht out of scope und wird daher nicht gelöscht.

-> Memory Leak!

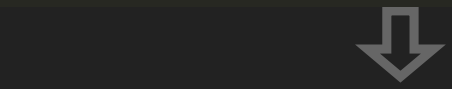
```
class Cell {
  int value;
public:

  Cell(): value(0) {
  //  std::cerr << "default constructor" << std::endl;
  }

  Cell(int val): value(val) {
  //  std::cerr << "overloaded constructor: value=" << value << std::endl;
  }

  ~Cell() {
    std::cerr << "destructor: value=" << value << std::endl;
  }
};
```

```
void step1() {
  Cell demo(1);
  Cell* demo_ptr = new Cell;
}
```



```
destructor value=1
```

Memory Management

```
class Cell {
  int value;
public:

/* constructor, destructor */

Cell(const Cell& other) {
  value = other.value;
  std::cerr << "copy constructor: value=" << value << std::endl;
}

Cell& operator= (const Cell& other) {
  value = other.value;
  std::cerr << "assignment operator: value=" << value << std::endl;
  return *this;
}

};
```

```
void step2() {
  Cell demo(3);
  Cell demo2 = demo;
  demo = demo2;
}
```

Was ist der Output von *step2*?

```
copy constructor: value=3
copy constructor: value=3
```

```
copy constructor: value=3
assignment operator: value=3
```

```
assignment operator: value=3
copy constructor: value=3
```

```
assignment operator: value=3
assignment operator: value=3
```

Memory Management

- Wenn eine neue Klasse erstellt wird, bedeutet "=" den Copy constructor
- Wenn die Klasse bereits existiert, ist es der Assignment operator

```
class Cell {
  int value;
public:

/* constructor, destructor */

Cell(const Cell& other) { // COPY CONSTRUCTOR
  value = other.value;
  std::cerr << "copy constructor: value=" << value << std::endl;
}

Cell& operator= (const Cell& other) { // ASSIGNMENT OPERATOR
  value = other.value;
  std::cerr << "assignment operator: value=" << value << std::endl;
  return *this;
}

};
```

```
void step2() {
  Cell demo(3);
  Cell demo2 = demo;
  demo = demo2;
}
```



```
copy constructor: value=3
assignment operator: value=3
```

Welches Programm ist Fehlerfrei?

```
class A{
    int array[5];
public:
    A(){
        for(unsigned int i = 0; i < 5; i++){
            array[i] = i;
        }
    }

    ~A(){
        delete array;
    }
};
```



```
class B {
    int array[5];
public:
    B(){
        for(unsigned int i = 0; i < 5; i++){
            array[i] = i;
        }
    }
};
```

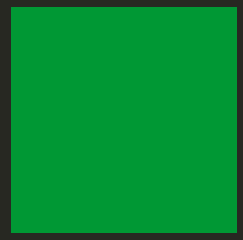


```
class C {
    int* array;
public:
    C(){
        array = new int[5];
        for(unsigned int i = 0; i < 5; i++){
            array[i] = i;
        }
    }

    ~C(){
        delete array;
    }
};
```

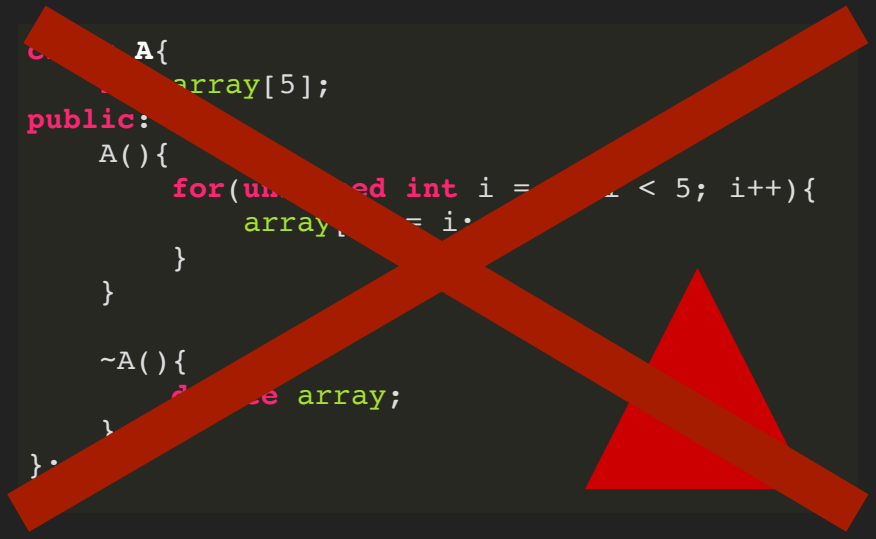


```
class D {
    int* array;
public:
    D(){
        array = new int[5];
        for(unsigned int i = 0; i < 5; i++){
            array[i] = i;
        }
    }
};
```



Welches Programm ist Fehlerfrei?

Variablen welche nicht mit *new* erstellt wurden müssen und dürfen nicht mit delete gelöscht werden.



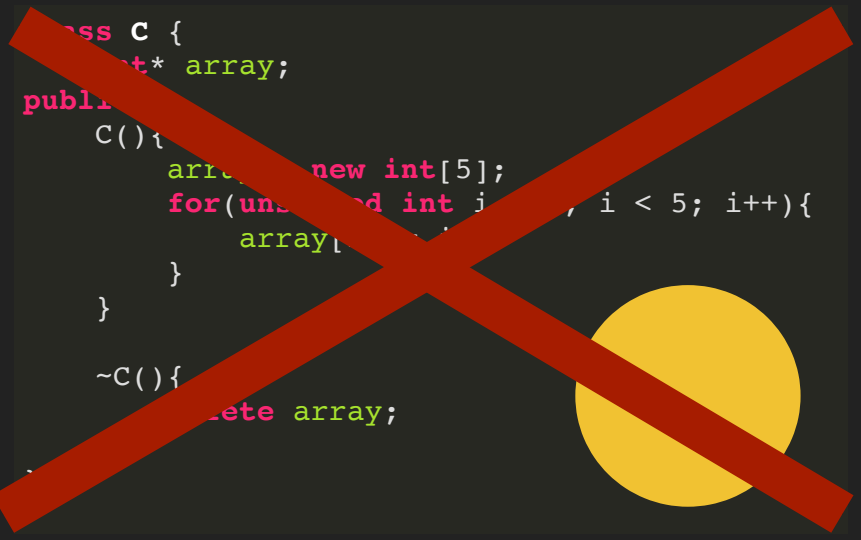
```
class A{
    int array[5];
public:
    A(){
        for(unsigned int i = 0; i < 5; i++){
            array[i] = i;
        }
    }

};
```



Welches Programm ist Fehlerfrei?

Arrays müssen mit delete[] gelöscht werden, sonst wird nur das erste Element gelöscht



```
class C {
    int* array;
public:
    C(){
        array = new int[5];
        for(unsigned int i = 0; i < 5; i++){
            array[i] = i;
        }
    }

    ~C(){
        delete[] array;
    }
};
```



Welches Programm ist Fehlerfrei?

Arrays müssen mit delete[] gelöscht werden, sonst wird nur das erste Element gelöscht

```
class D {
    int* array;
public:
    D(){
        array = new int[5];
        for(unsigned int i = 0; i < 5; i++){
            array[i] = i;
        }
    }

    ~D(){
        delete[] array;
    }
};
```

Welches Programm ist Fehlerfrei?

Variablen welche nicht mit *new* erstellt wurden müssen und dürfen nicht mit delete gelöscht werden.

```
class B {
    int array[5];
public:
    B(){
        for(unsigned int i = 0; i < 5; i++){
            array[i] = i;
        }
    }
};
```



Welche Zeile ist falsch?

```
1  class Arr5 {
2  public:
3      int* array;
4
5      Arr5(){
6          array = new int[5];
7      }
8
9      ~Arr5(){
10         delete[] array;
11     }
12 };
13
14 int main(){
15     Arr5 a = Arr5();
16     delete[] a.array;
17     return 0;
18 }
```



array = new int[5];

delete[] array;

Arr5 a = Arr5();

delete[] a.array;

Welche Zeile ist falsch?

```
1  class Arr5 {
2  public:
3      int* array;
4
5      Arr5(){
6          array = new int[5];
7      }
8
9      ~Arr5(){
10         delete[] array;
11     }
12 };
13
14 int main(){
15     Arr5 a = Arr5();
16     delete[] a.array;
17     return 0;
18 }
```

- Die Funktion main darf delete nicht auf a.array anwenden, da dieser von der Klasse Arr5 managed wird.
- Sobald nachher~Arr5() aufgerufen wird, gibt es einen Segmentation fault, da die daten bereits gelöscht wurden.
- ~Arr5() wird aufgerufen wenn a out of scope geht (am Ende von main)

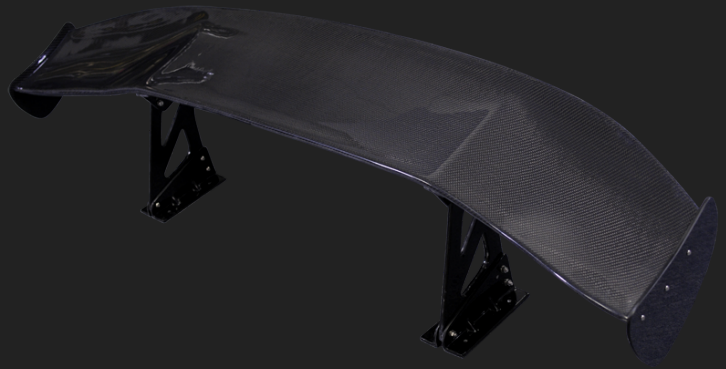
Memory Management II

Wir haben dieselbe Klasse wie zuvor, nur dass "*value*" jetzt ein Pointer ist.

Implementiert im File **cell.cpp**
Einen *destructor*, einen *constructor* und einen *assignment operator* sodass keine memory leaks auftreten

```
1  class Cell {
2    int* value;
3  public:
4    Cell();
5    Cell(int val);
6    ~Cell();
7    Cell(const Cell& other);
8    Cell& operator= (const Cell& other);
9    int get_value();
10   void set_value(int new_value);
11 };
```

Vorbesprechung



1: Operator delete

- Ihr bekommt verschiedene Programme, welche entweder Errors oder Memory Leaks haben
- Findet bei jedem Programm den Fehler.
- Falls es einen Error gibt, schreibt in welcher Zeile dieser vorkommen wird.

Tipps:

- **Jedes** Programm hat irgend einen Fehler
- Wenn ihr Probleme habt, schaut euch unsere Kahoot Aufgaben nochmals an.

2: Array-based Vector

- Gegeben ist eine Implementation von einem Array-based Vector.
- Dieser Vector wird mit einer fixen Grösse initialisiert und speichert seine Elemente in einem Array.
- Schreibt einen
  1. Copy constructor.
  2. Assignment operator.
  3. Destructor.

Tipps:

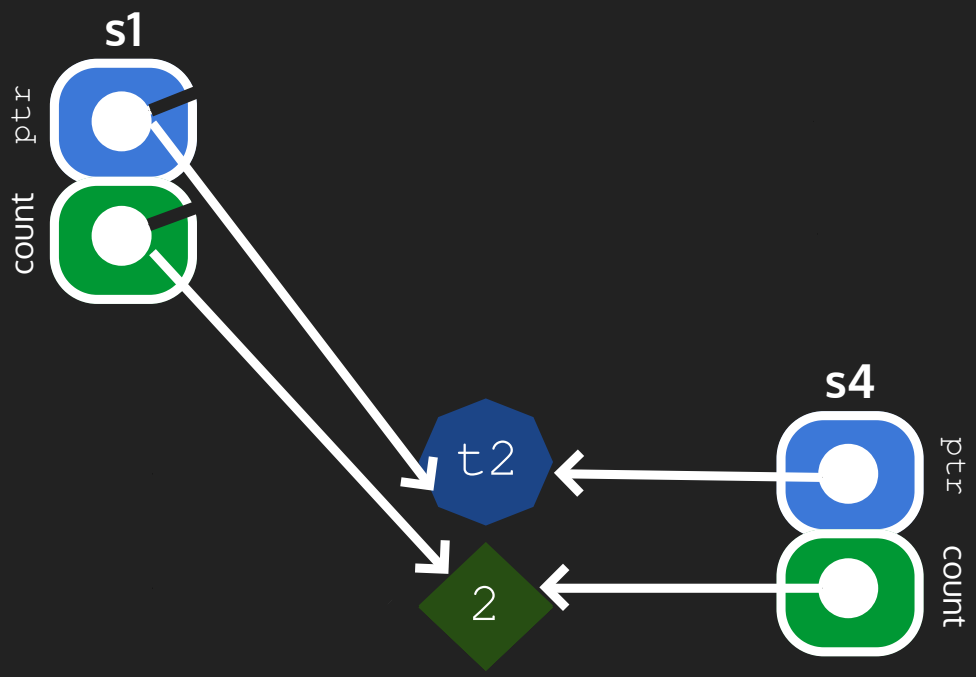
- Sehr ähnlich wie in der Aufgabe *Memory Management*
- Verwendet den Copy constructor bei eurer Implementation vom Assignment operator

3: Smart Pointers

```
1  tracked* t1 = new tracked;
2  tracked* t2 = new tracked;
3
4  Smart s1 = Smart(t1);
5  Smart* s2 = new Smart(t2);
6  Smart s4;
7  {
8    Smart s3 = s1;
9
10   s1 = *s2;
11   delete s2;
12 }
13 s4 = s1;
```

```
13: Smart& Smart::operator=
(const Smart& src) {}
```

s4 zeigt nun auf dasselbe Objekt wie s3.
Der neue Counter wird erhöht.
Da kein alter Pointer existiert, darf dieser auch nicht verkleinert werden.



Prüfungstipps



Prüfungstipps

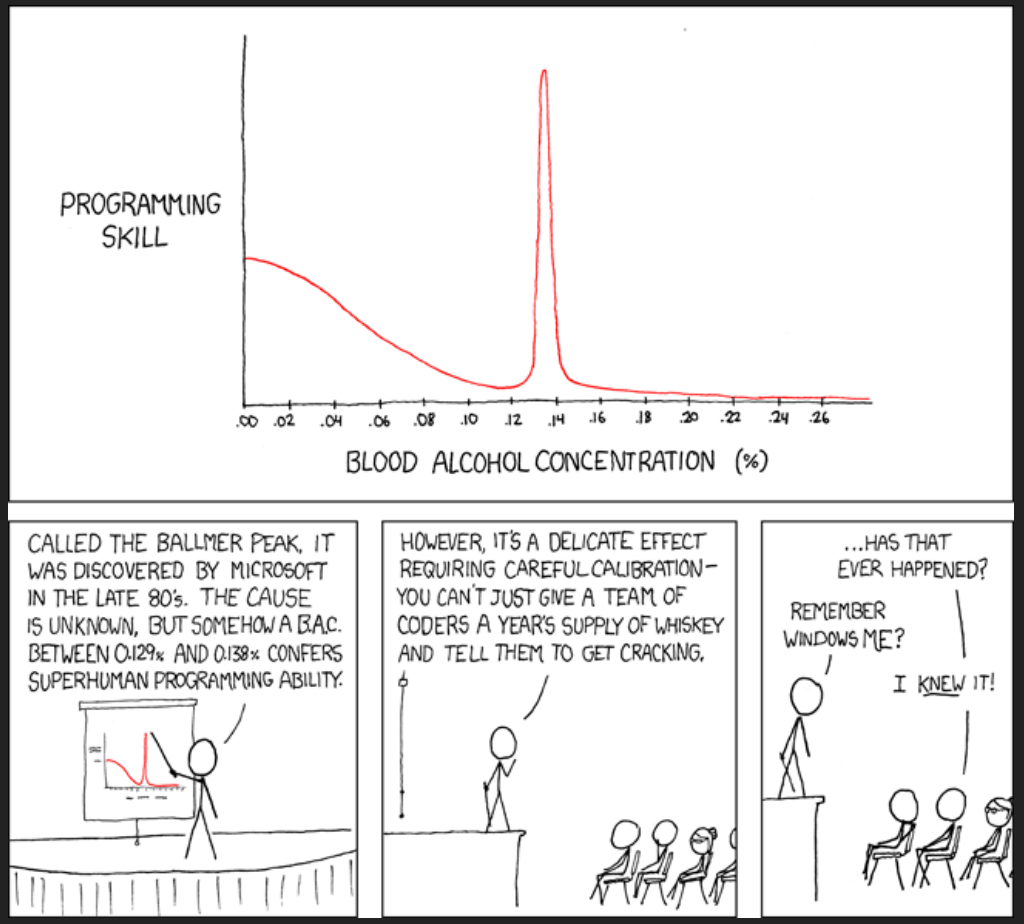
Generell

- Macht euch einen Lernplan für die ganze Lernphase.
- Löst nochmals viele Serien
- Zu wenig lernen ist schlecht, zu viel aber genau so
  -> Gönnt euch Freizeit

Info I

- Es gibt viele Alte Prüfungen, löst so viele wie möglich.
- Der Prüfungsstil war bis jetzt gleich wie bei Prof. Friedrich
- Viele Aufgaben sind jedes Jahr gleich
- Programmiert in eurer Freizeit

Viel Spass!


PROGRAMMING SKILL
.00 .02 .04 .06 .08 .10 .12 .14 .16 .18 .20 .22 .24 .26
BLOOD ALCOHOL CONCENTRATION (%)
CALLED THE BALLMER PEAK. IT WAS DISCOVERED BY MICROSOFT IN THE LATE 80's. THE CAUSE IS UNKNOWN, BUT SOMEHOW A B.A.C. BETWEEN 0.129% AND 0.138% CONFERS SUPERHUMAN PROGRAMMING ABILITY.
HOWEVER, IT'S A DELICATE EFFECT REQUIRING CAREFUL CALIBRATION— YOU CAN'T JUST GIVE A TEAM OF CODERS A YEAR'S SUPPLY OF WHISKEY AND TELL THEM TO GET CRACKING.
...HAS THAT EVER HAPPENED?
REMEMBER WINDOWS ME?
I KNEW IT!