Advanced practical **Programming** for Scientists

Thorsten Koch

Zuse Institute Berlin

TU Berlin

SS2017

The Zen of Python, by Tim Peters (part 1)

- Beautiful is better than ugly.
- Explicit is better than implicit.
- Simple is better than complex.
- Complex is better than complicated.
- Flat is better than nested.
- Sparse is better than dense.
- Readability counts.
- Special cases aren't special enough to break the rules.
- Although practicality beats purity.
- Errors should never pass silently.
- Unless explicitly silenced.
- In the face of ambiguity, refuse the temptation to guess.

Ex1 again

- Remember:
  store the data and compute the geometric mean on this stored data.
- If it is not obvious how to compile your program, add a REAME file or a comment at the beginning
- It should run as
  `ex1 filenname`
- If you need to start something (python, python3, ...) provide an executable script named `ex1` which calls your program, e.g.

  ```
  #/bin/bash
  python3 ex1.py $1
  ```
- Compare the number of valid values.
  If you have a lower number, you are missing something.
  If you have a higher number, send me the wrong line I am missing.
  File: ex1-100.dat with 100001235 lines
  Valid values Loc0: 50004466 with GeoMean: 36.781736
  Valid values Loc1: 49994581 with GeoMean: 36.782583

Exercise 1: File Format (more detail)

Each line should consists of

- a sequence-number,
- a location (1 or 2), and
- a floating point value > 0.

Empty lines are allowed.

Comments can start a ”#”.

Anything including and after “#” on a line should be ignored.

The fields are separated by “;”.

White space before and after the “;” is allowed.

Extra non white space characters after the value field are not allowed.

Be aware that “NaN” could be considered a valid floating value. (-Inf?)

Exercise 2: Reading XML

We provided some documentation and and an example file.

Write a program **ex2** that reads in the

measured-1.0.0.2017-02-03.b0050c5c8deb1db59c7b2644414b079d.xml

And writes CSV data in format:

YYYY-MM-DD; HH; amountOfPower-Value

You will need the reading part later again.

Try to validate the XML file against the provided schema

The filename to be converted should be taken from the command line.

The output should be to stdout, any errors to stderr.

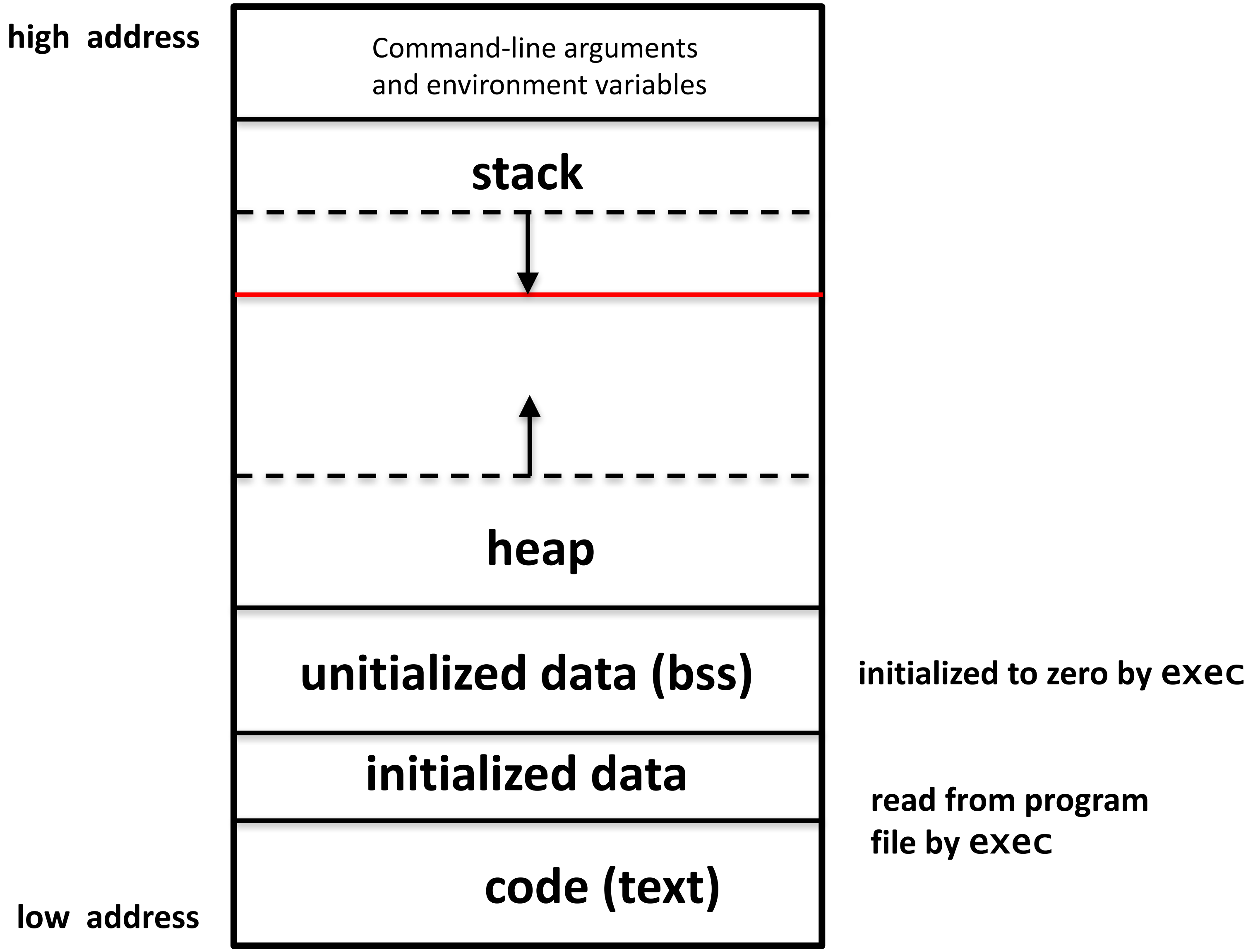
You may use whatever library to parse the XML file.

Eval order: What is the output?

```
#include <stdio.h>

int f1(void) { puts("f1"); return 2; }
int f2(void) { puts("f2"); return 3; }
int f3(void) { puts("f3"); return 5; }
int f4(void) { puts("f4"); return 7; }

int ff(int a, int b, int c) {
   puts("ff");
   return a + b + c;
}

int main() {
   printf("x=%d\n", ff(f1(), f2() * f3(), f4()));
}
```

```
f4  f2  f3  f1  ff  x=24
```

Memory layout of C programs


high address
Command-line arguments and environment variables
stack
heap
unitialized data (bss)
initialized data
code (text)
low address
initialized to zero by exec
read from program file by exec

From http://www.geeksforgeeks.org/memory-layout-of-c-program/

Code Segment

A code segment , also known as a text segment, is one of the sections of a program in an object file or in memory, which contains executable instructions.

As a memory region, a text segment may be placed below the heap or stack in order to prevent heaps and stack overflows from overwriting it.

Usually, the text segment is sharable so that only a single copy needs to be in memory for frequently executed programs, such as text editors, the C compiler, the shells, and so on. Also, the text segment is often read-only, to prevent a program from accidentally modifying its instructions.

Initialized Data Segment

Initialized data segment, usually called simply the Data Segment. A data segment is a portion of virtual address space of a program, which contains the global variables and static variables that are initialized by the programmer.

Note that, data segment is not read-only, since the values of the variables can be altered at run time.

This segment can be further classified into initialized read-only area and initialized read-write area.

For instance the global string defined by `char s[] = “hello world”` in C and a C statement like `int debug=1` outside the main (i.e. global) would be stored in initialized read-write area. And a global C statement like `const char* string = “hello world”` makes the string literal “hello world” to be stored in initialized read-only area and the character pointer variable string in initialized read-write area.

Ex: `static int i = 10` will be stored in data segment and
global `int i = 10` will also be stored in data segment

Uninitialized Data Segment

Uninitialized data segment, often called the “bss” segment. Data in this segment is initialized by the kernel to arithmetic 0 before the program starts executing.

uninitialized data starts at the end of the data segment and contains all global variables and static variables that are initialized to zero or do not have explicit initialization in source code.

For instance a variable declared `static int i;` would be contained in the BSS segment.
For instance a global variable declared `int j;` would be contained in the BSS segment.

Stack

The stack area traditionally adjoined the heap area and grew the opposite direction; when the stack pointer met the heap pointer, free memory was exhausted. (With modern large address spaces and virtual memory techniques they may be placed almost anywhere, but they still typically grow opposite directions.)

The stack area contains the program stack, a LIFO structure, typically located in the higher parts of memory. On the standard PC x86 computer architecture it grows toward address zero; on some other architectures it grows the opposite direction. A “stack pointer” register tracks the top of the stack; it is adjusted each time a value is “pushed” onto the stack. The set of values pushed for one function call is termed a “stack frame”; A stack frame consists at minimum of a return address.

Stack, where automatic variables are stored, along with information that is saved each time a function is called. Each time a function is called, the address of where to return to and certain information about the caller’s environment, such as some of the machine registers, are saved on the stack. The newly called function then allocates room on the stack for its automatic and temporary variables. This is how recursive functions in C can work. **Each time a recursive function calls itself, a new stack frame is used, so one set of variables doesn’t interfere with the variables from another instance of the function.**

Heap

Heap is the segment where dynamic memory allocation usually takes place.

The heap area begins at the end of the BSS segment and grows to larger addresses from there.

The Heap area is managed by malloc(), realloc(), and free(),
which may use the brk() and sbrk() system calls to adjust its size.

(note that the use of brk/sbrk and a single “heap area” is not required to fulfill the contract of malloc/realloc/free; they may also be implemented using mmap to reserve potentially non-contiguous regions of virtual memory into the process’ virtual address space).

The Heap area is shared by all shared libraries and dynamically loaded modules in a process.

The size(1) command reports the sizes (in bytes) of the text, data, and bss segments. (for more details please refer man page of size(1), objdump(1))

```
#include <stdio.h>

int main(void)
{
    return 0;
}

$ gcc memory-layout.c -o memory-layout
$ size memory-layout

text        data        bss        dec        hex     filename
960         248          8        1216        4c0    memory-layout
```

added one global variable in program, now check the size of bss

```
#include <stdio.h>

int global; /* Uninitialized variable stored in bss*/

int main(void)
{
    return 0;
}
```

```
text       data        bss        dec        hex    filename
 960        248         12       1220        4c4    memory-layout
```

Let us add one static variable which is also stored in bss.

```
#include <stdio.h>

int global; /* Uninitialized variable stored in bss*/

int main(void)
{
    static int i; /* Uninitialized static variable stored in bss */
    return 0;
}
```

```
text       data        bss       dec        hex    filename
 960        248         16      1224        4c8    memory-layout
```

Let us initialize the static variable which will then be stored in the Data Segment (DS)

```
#include <stdio.h>

int global; /* Uninitialized variable stored in bss*/

int main(void)
{
    static int i = 100; /* Initialized static variable stored in DS*/
    return 0;
}
```

```
text        data        bss        dec        hex     filename
960          252         12       1224        4c8     memory-layout
```

Let us initialize the global variable which will then be stored in the Data Segment (DS)

```
#include <stdio.h>

int global = 10; /* initialized global variable stored in DS*/

int main(void)
{
    static int i = 100; /* Initialized static variable stored in DS*/
    return 0;
}
```

```
text        data        bss        dec        hex    filename
960          256          8       1224        4c8    memory-layout
```

```
$ ulimit -d -m -n -s -u
data seg size            (kbytes, -d) unlimited
max memory size          (kbytes, -m) unlimited
open files                       (-n) 1024
stack size               (kbytes, -s) 8192
max user processes               (-u) 15789
```

Stack

```c
#include <stdio.h>

void f(int i)
{
   int a[100];

   a[1] = i + 1;

   if (i % 1000 == 0) {
      printf("%d ", i / 1000);
      fflush(stdout);
   }
   f(a[1]);
}
int main() { f(1); }
```

```
$ ./a.out
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
18 Segmentation fault (core dumped)

$ ulimit -s 16384
$ ./a.out
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
18 19 20 21 22 23 24 25 26 27 28 29 30 31
32 33 34 35 36 37 Segmentation fault (core
dumped)
```

```c
#include <stdio.h>

void f(int i)
{
   int a[2];

   a[1] = i + 1;

   if (i % 1000 == 0) {
      printf("%d ", i / 1000);
      fflush(stdout);
   }
   f(a[1]);
}
int main() { f(1); }
```

```
$ ./a.out
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39
40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57
58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75
76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93
94 95 96 97 98 99 100 101 102 103 104 105 106 107 108
109 110 111 112 113 114 115 116 117 118 119 120 121 122
123 124 125 126 127 128 129 130 Segmentation fault
(core dumped)
```

```c
#include <stdio.h>
#include <stdlib.h>

void f(int i)
{
   int* a = malloc(100 * sizeof(*a));

   a[1] = i + 1;

   if (i % 1000 == 0) {
      printf("%d ", i / 1000);
      fflush(stdout);
   }
   f(a[1]);

   free(a);
}
int main() { f(1); }
```

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40
41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77
78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110
111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 Segmentation fault (core dumped)
```

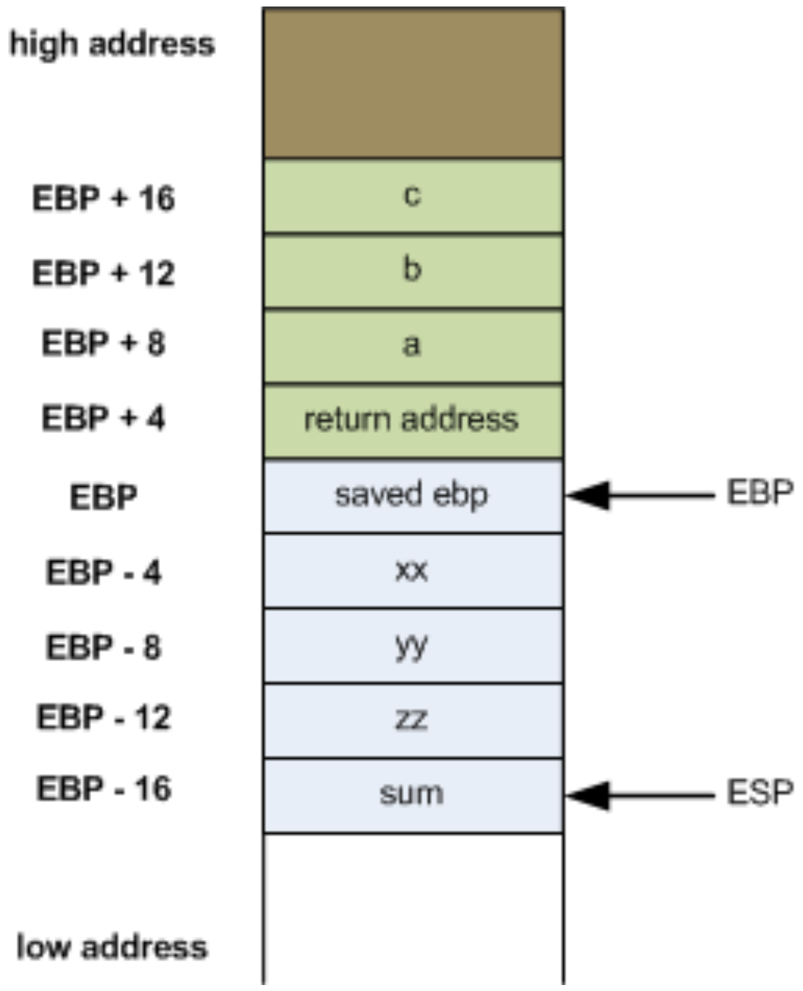
```
$ valgrind ./a.out
==6373== Memcheck, a memory error detector
==6373==
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34
35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65
66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96
97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120
121 122 123 124 125 126 127 128 129 130
==6373== Stack overflow in thread 1: can't grow stack to 0xffe801fd0
==6373==
==6373== Process terminating with default action of signal 11 (SIGSEGV)
==6373==  The main thread stack size used in this run was 8388608.
==6373== Stack overflow in thread 1: can't grow stack to 0xffe801fb8
==6373==
==6373== Process terminating with default action of signal 11 (SIGSEGV)
==6373==
==6373== HEAP SUMMARY:
==6373==     in use at exit: 523,728,000 bytes in 130,932 blocks
==6373==   total heap usage: 130,932 allocs, 0 frees, 523,728,000 bytes allocated
==6373==
==6373== LEAK SUMMARY:
==6373==    still reachable: 523,728,000 bytes in 130,932 blocks
```

```
#include <stdio.h>
#include <alloca.h>

void f(int i)
{
   int* a = alloca(100 * sizeof(*a));

   a[1] = i + 1;

   if (i % 1000 == 0) {
      printf("%d ", i / 1000);
      fflush(stdout);
   }
   f(a[1]);
}
int main() { f(1); }
```

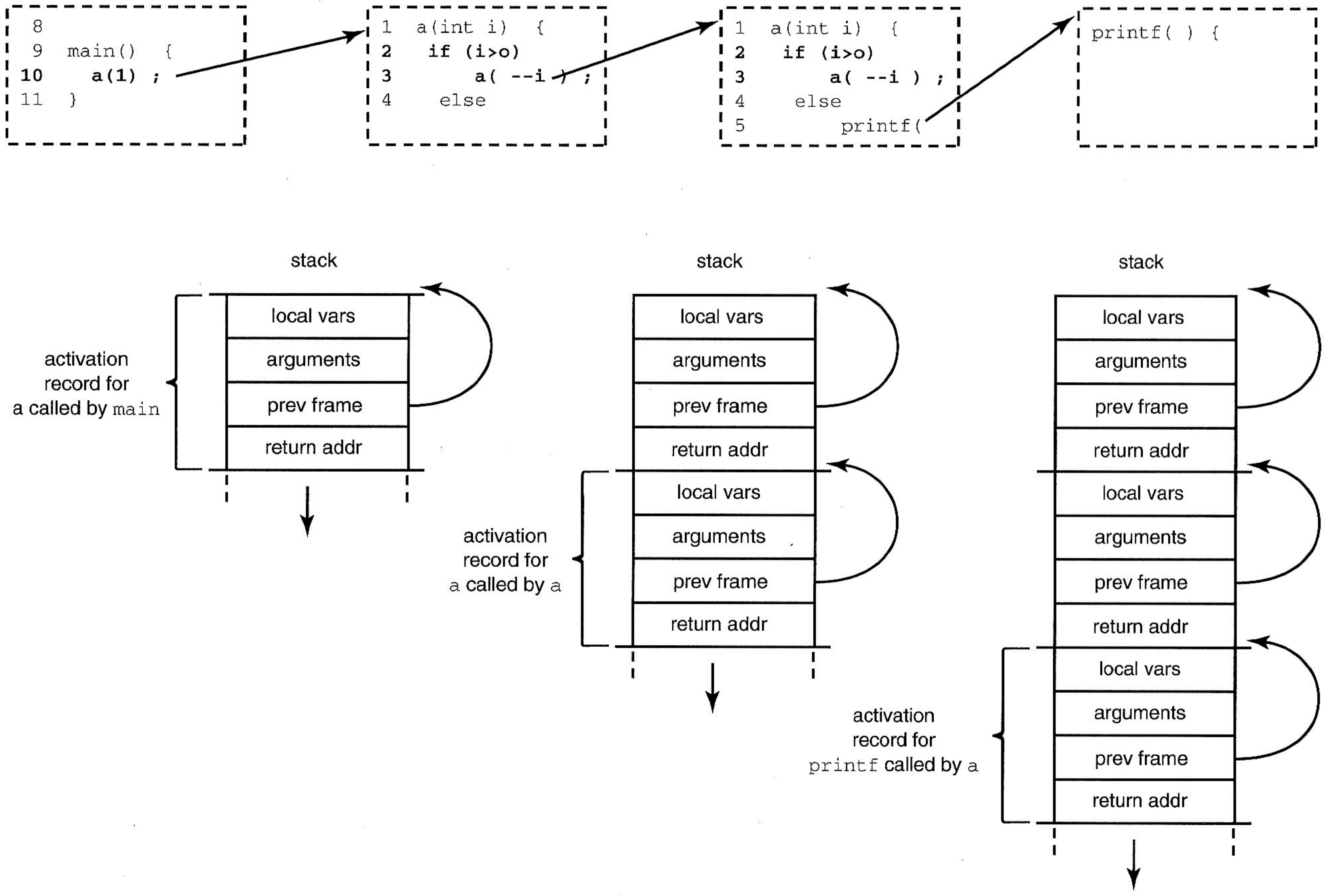
```
$ ./a.out
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
Segmentation fault (core dumped)
```

Stack Pointer

```
int foobar(int a, int b, int c)
{
  int xx = a + 2;
  int yy = b + 3;
  int zz = c + 4;
  int sum = xx + yy + zz;
  return xx * yy * zz + sum;
}
int main()
{
  return foobar(77, 88, 99);
}
```



Stack Frame



Page 150, Van der Linden, Expert C Programing, Prentice Hall, 1994

Memory allocation

```
#include <stdlib.h>

void *malloc(size_t size);
void free(void *ptr);
void *calloc(size_t nmemb, size_t size);
void *realloc(void *ptr, size_t size);
```

Example:

```
double x = malloc(1000 * sizeof(*x))
double x = calloc(1000, sizeof(*x))
double x = realloc(x, 10000 * sizeof(*x))
free(x)
```

Note: size_t is unsigned.

Note: free() should be on the same level than malloc()

Memory allocation

malloc/calloc can always return NULL.

- Must check.
- Happens too late.
- Hard to test (ulimit)
- Hard to recover
- Replace by cover functions
- With possible guards
- Performance variation
- Memory pool for many identical small calls
- memory leaks/overruns
- use after free (linked list)

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define SIZE 100000000

#define GET_SEC(a, b)  ((b - a) / (double)CLOCKS_PER_SEC)

int main(int argc, char** argv)
{
   clock_t start;
   double* x = malloc(SIZE * sizeof(*x));
   double  sum;

   for(int i = 0; i < SIZE; i++)
      x[i] = i + 1.0;
```

```
sum = 0.0; start = clock();

/* In order 1 */
for(int i = 0; i < SIZE; i++)
   sum += x[i];

printf("IO time=%.3f sum=%.1f\n", GET_SEC(start, clock()), sum);


sum = 0.0; start = clock();

/* In order 2 */
for(int k = 0; k < 100; k++)
   for(int i = 0; i < SIZE / 100; i++)
      sum += x[k * (SIZE / 100) + i];

printf("O1 time=%.3f sum=%.1f\n", GET_SEC(start, clock()), sum);
```

```
sum = 0.0; start = clock();

for(int i = 0; i < SIZE / 100; i++)
   for(int k = 0; k < 100; k++)
      sum += x[k * (SIZE / 100) + i];

printf("OH time=%.3f sum=%.1f\n", GET_SEC(start, clock()), sum);


sum = 0.0; start = clock();

for(int i = 0; i < SIZE / 1000; i++)
   for(int k = 0; k < 1000; k++)
      sum += x[k * (SIZE / 1000) + i];

printf("OT time=%.3f sum=%.1f\n", GET_SEC(start, clock()), sum);
```

```
   sum = 0.0; start = clock();

   for(int i = 0; i < SIZE / 10000; i++)
      for(int k = 0; k < 10000; k++)
         sum += x[k * (SIZE / 10000) + i];

   printf("Ot time=%.3f sum=%.1f\n", GET_SEC(start, clock()), sum);


   sum = 0.0; start = clock();

   for(int i = 0; i < SIZE / 100000; i++)
      for(int k = 0; k < 100000; k++)
         sum += x[k * (SIZE / 100000) + i];

   printf("Oh time=%.3f sum=%.1f\n", GET_SEC(start, clock()), sum);
}
```

Array access times

Offset	**base**	**-O**	**-O2**	**-O3**
in order	0.36	0.34	0.13	0.13
offset 1	0.35	0.34	0.13	0.13
offset 100	1.41	1.34	1.12	0.70
offset 1000	1.74	1.64	1.68	0.86
offset 10000	2.34	2.25	2.29	1.16
offset 100000	2.73	2.18	2.02	1.03
Ratio	*7.8*	*6.6*	*17.6*	*8.9*

Struct sizes

```
#include <stdio.h>

struct { int    i; char   c; double d; char   x; } icdx;
struct { char   c; double d; char   x; int    i; } cdxi;
struct { char   c; char   x; int    i; double d; } cxid;
struct { char   x; double d; char   c; double y; } xdcy;
struct { double d; double y; char   c; char   x; } dycx;

int main()
{
   printf("icdx=%lu\n", sizeof(icdx));   24
   printf("cdxi=%lu\n", sizeof(cdxi));   24
   printf("cxid=%lu\n", sizeof(cxid));   16
   printf("xdcy=%lu\n", sizeof(xdcy));   32
   printf("dycx=%lu\n", sizeof(dycx));   24
}
```

```
#include <stdio.h>
#include <stddef.h>

struct cxid { char   c; char   x; int    i; double d; };

int main()
{
   printf("c=%lu\n", offsetof(struct cxid, c));
   printf("x=%lu\n", offsetof(struct cxid, x));
   printf("i=%lu\n", offsetof(struct cxid, i));
   printf("d=%lu\n", offsetof(struct cxid, d));
}
```

```
c=0
x=1
i=4
d=8
```

union

```
#include <stdio.h>
#include <stddef.h>

typedef union { char c; int i; double d; short s[4]; } CIDS;

static CIDS cids = { .d = 12.4 };

int main()
{

   printf("size cids=%lu\n", sizeof(cids));
   printf("offset d=%lu\n", offsetof(CIDS, d));
   printf("d=%f\n", cids.d);
   printf("c=%d\n", cids.c);
   printf("i=%d\n", cids.i);
}
```

```
size cids=8
offset d=0
d=12.400000
c=-51
i=-858993459
```

Exercise 3: What to do

Write a program in C or your favorite compiled language, which takes no input and produces a copy of its own source code as its only output.

The program should be as short as possible (not important) and have at least one character (because there are languages where the empty program is a valid program).

The standard terms for these programs in the computability theory and computer science literature are “self-replicating programs”, “self-reproducing programs”, and “self-copying programs”.

Otherwise it is called a **Quine**.

Please, given the info above it is easy enough to look this up in the Internet. The purpose of this exercise is that you try it yourself.

We will discuss the most interesting ones in the lecture.

Exercise DEADLINES

Wednesday May 17

Ex 1, 2, 3

Lecture 12.05: Digital Future

Please register yourself via

https://science-match.tagesspiegel.de/the-digital-future-may-2017

Voucher/VIP Code: *future17-1*