Advanced practical **Programming** for Scientists

Thorsten Koch

Zuse Institute Berlin

TU Berlin

WS2014/15

ex1

	Language	Time			Memory
		wallclock	user	system	max resident
A.I.	C	110	110		3.906.728
A.G.-P.	Java	63	40		1.336.164
J.S.	C	42	41		2.098.488
S.T.	Julia	129	128		4.234.484
S.S.	Java	200	130	72	4.736.448
A.V.H.	Java	516	250	289	3.557.132
A.G.	C++	558	132	221	1.954.276
F.S.	C	57	57		263.636
a	C	37	36		262.644
a2	C	36	36		262.648
a3	C	37	37		262.580
b	C	55	55		263.636
b3	C	111	110		3.906.744
c	C	36	36		2.215.652
only print numbers	*C*	*30*	*29*		

ex 2 Quine

Java (André):

```
class Quine{public static void main(String[] a){String s =
"class Quine{public static void main(String[] a){String s =
%c%s%c;System.out.printf(s,34,s,34);}}";System.out.printf(s,34
,s,34);}}
```

C:

```
main() { char *s="main() { char *s=%c%s%c; printf(s,34,s,34);
}”; printf(s,34,s,34); }
```

C++

```
#include <cstdio> char* o="#include <cstdio>%cchar*
o=%c%s%c%c%cmain(){%cprintf(o,10,34,o,34,59,10,10,10,10);%c}";
main(){ printf(o,10,34,o,34,59,10,10,10,10); }
```

Python:

```
|x = ['print "x =", x', 'for m in x: print m'] print "x =", x
for m in x: print m|
```

ex 2 C++

```
#include <string>
#include <iostream>
using namespace std;
int main(){
string s0 = "#include <string>";
string s1 = "#include <iostream>";
string s2 = "using namespace std";
string s3 = "int main(){";
string s4 = "}";
string s5 = "\n";
string s6 = ";";
string s7 = " = ";
string s8 = " ";
string s9 = "string s";
string s10 = "\"";
string s11 = "\\n";
string s12 = "\\";
string s13 = " cout<<s0<<s5<<s1<<s5<<s5<<s2<<s5<<s5<<s3<<s5;";
string s14 = " cout<<s8<<s9<<0<<s7<<s0<<s10<<s6<<s5<<s8<<s9<<1<<s7<<s10<<s1<<s10<<s6<<s5<<s8<<s9<<2<<s7<<s10<<s2<<s10<<s6<<s5;";
string s15 = " cout<<s8<<s9<<3<<s7<<s10<<s3<<s10<<s6<<s5<<s8<<s9<<4<<s7<<s10<<s4<<s10<<s6<<s5<<s8<<s9<<5<<s7<<s10<<s11<<s10<<s6<<s5;";
string s16 = " cout<<s8<<s9<<6<<s7<<s10<<s6<<s10<<s6<<s5<<s8<<s9<<7<<s7<<s10<<s7<<s10<<s6<<s5<<s8<<s9<<8<<s7<<s10<<s8<<s10<<s6<<s5;";
string s17 = "
cout<<s8<<s9<<9<<s7<<s10<<s9<<s10<<s6<<s5<<s8<<s9<<10<<s7<<s10<<s12<<s10<<s6<<s5<<s8<<s9<<11<<s7<<s10<<s12<<s11<<s10<<s6<<s5;";
string s18 = " cout<<s8<<s9<<12<<s7<<s10<<s12<<s12<<s6<<s5<<s8<<s9<<13<<s7<<s10<<s13<<s10<<s6<<s5<<s8<<s9<<14<<s7<<s10<<s14<<s10<<s6<<s5;";
string s19 = " cout<<s8<<s9<<15<<s7<<s10<<s15<<s10<<s6<<s5<<s8<<s9<<16<<s7<<s10<<s16<<s10<<s6<<s5<<s8<<s9<<17<<s7<<s10<<s17<<s10<<s6<<s5;";
string s20 = " cout<<s8<<s9<<18<<s7<<s10<<s18<<s10<<s6<<s5<<s8<<s9<<19<<s7<<s10<<s19<<s10<<s6<<s5<<s8<<s9<<20<<s7<<s10<<s20<<s10<<s6<<s5;";
string s21 = " cout<<s8<<s9<<21<<s7<<s10<<s20<<s10<<s6<<s5;";
string s22 = " cout<<s13<<s5<<s14<<s5<<s15<<s5<<s16<<s5<<s17<<s5<<s18<<s5<<s19<<s5<<s20<<s5<<s21<<s5<<s22<<s5<<s4<<s5;";
cout<<s0<<s5<<s1<<s5<<s5<<s2<<s5<<s5<<s3<<s5;
cout<<s8<<s9<<0<<s7<<s0<<s10<<s6<<s5<<s8<<s9<<1<<s7<<s10<<s1<<s10<<s6<<s5<<s8<<s9<<2<<s7<<s10<<s2<<s10<<s6<<s5;
cout<<s8<<s9<<3<<s7<<s10<<s3<<s10<<s6<<s5<<s8<<s9<<4<<s7<<s10<<s4<<s10<<s6<<s5<<s8<<s9<<5<<s7<<s10<<s11<<s10<<s6<<s5;
cout<<s8<<s9<<6<<s7<<s10<<s6<<s10<<s6<<s5<<s8<<s9<<7<<s7<<s10<<s7<<s10<<s6<<s5<<s8<<s9<<8<<s7<<s10<<s8<<s10<<s6<<s5;
cout<<s8<<s9<<9<<s7<<s10<<s9<<s10<<s6<<s5<<s8<<s9<<10<<s7<<s10<<s12<<s10<<s6<<s5<<s8<<s9<<11<<s7<<s10<<s12<<s11<<s10<<s6<<s5;
cout<<s8<<s9<<12<<s7<<s10<<s12<<s12<<s6<<s5<<s8<<s9<<13<<s7<<s10<<s13<<s10<<s6<<s5<<s8<<s9<<14<<s7<<s10<<s14<<s10<<s6<<s5;
cout<<s8<<s9<<15<<s7<<s10<<s15<<s10<<s6<<s5<<s8<<s9<<16<<s7<<s10<<s16<<s10<<s6<<s5<<s8<<s9<<17<<s7<<s10<<s17<<s10<<s6<<s5;
cout<<s8<<s9<<18<<s7<<s10<<s18<<s10<<s6<<s5<<s8<<s9<<19<<s7<<s10<<s19<<s10<<s6<<s5<<s8<<s9<<20<<s7<<s10<<s20<<s10<<s6<<s5;
cout<<s8<<s9<<21<<s7<<s10<<s21<<s10<<s6<<s5<<s8<<s9<<22<<s7<<s10<<s22<<s10<<s6<<s5;
cout<<s13<<s5<<s14<<s5<<s15<<s5<<s16<<s5<<s17<<s5<<s18<<s5<<s19<<s5<<s20<<s5<<s21<<s5<<s22<<s5<<s4<<s5; }
```

ex3

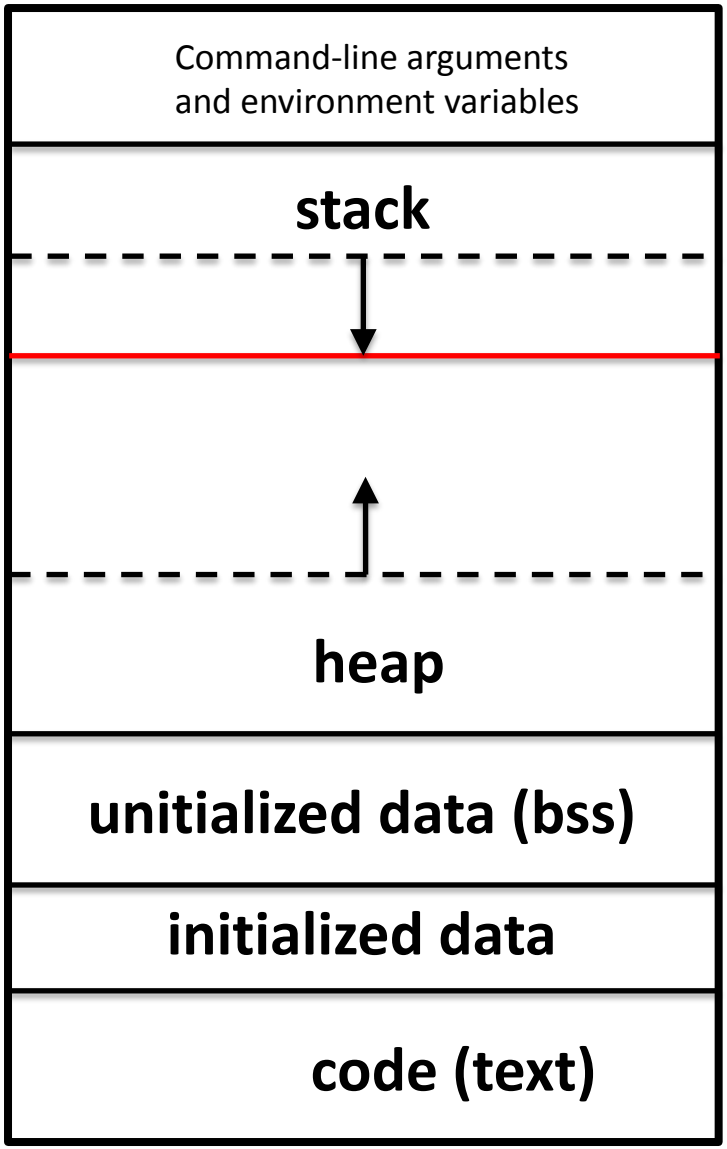
80aa21031baec3d18a2d97f74acd5346a5acf0c2

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



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 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 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

```
#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)
```

```
#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 frame

---

return address (or you never come back)

ptr to previous frame (-fomit-frame-pointer)

arguments (first 6 in registers AMD64)

local variables

```
void f()
{
   int a[1];
   printf("%p", &a[0]);
   printf("%d", a[-15]);
}
```

More details:

http://eli.thegreenplace.net/2011/09/06/stack-frame-layout-on-x86-64/

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
- Use memset/memcpy (string.h)
- memory leaks/overruns
- use after free (linked list)

Virtual memory

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

int main()
{
   int count = 0;

   while(NULL != malloc(1024*1024*1024))
   {
      count++;
      printf("Got %4d GB\n", count);
   }
   printf("Out of memory after %d GB\n", count);
}
```

Virtual memory

```
$ ./a.out
Got     1 GB
Got     2 GB
Got     3 GB
Got     4 GB
Got     5 GB
…
Got 114423 GB
Got 114424 GB
Got 114425 GB
Got 114426 GB
Got 114427 GB
Out of memory after 114427 GB
```

Virtual memory

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
   int  count = 0;
   int* p;

   while(NULL != (p = malloc(1024*1024*1024)))
   {
      for(int i = 0; i < 1024*1024; i++)
         p[i * 1024 / sizeof(*p)] = 5;

      count++;
      printf("Got %4d GB\n", count);
   }
   printf("Out of memory after %d GB\n", count);
}
```

Virtual memory

```
$ ./a.out
Got    1 GB
Got    2 GB
Got    3 GB
Out of memory after 3 GB
```

Memory access

```
#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;
```

Memory access

```
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);
```

Memory access

```
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);
```

Memory access

```
   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
}
```

Member offsets

```
#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 4

Write a program in C, which enumerates all solutions to a binary program of the form:

$$\mathbf{Ax} \leq \mathbf{b},\ \mathbf{x} \in \{0,1\}$$

with **A** being a **m**×**n** matrix, and **b** a **m** vector, with $\mathbf{m,n} \leq \mathbf{32}$.

Example:

$$2\,x_1 + 3\,x_2 + 5\,x_3 - 4\,x_4 \leq 8$$
$$3\,x_1\_1 - 6\,x_2 + 8\,x_4 \leq 10$$
$$x_3 + x_4 \leq 1$$

Exercise 4

The input format for the above would be:

```
4  # columns (variables)
3  # rows (constraints)
2 3 5 4 <= 8
3 6 0 8 <= 10
0 0 1 1 <= 1
```

Everything after an # in a line should be discarded as a comment.

The program should read the data from a file and write out all solution vectors.

Overall Program Design

- Data flow
- Control flow
- Hierarchical dependencies