CHALMERS
UNIVERSITY OF TECHNOLOGY

Machine-Oriented Programming

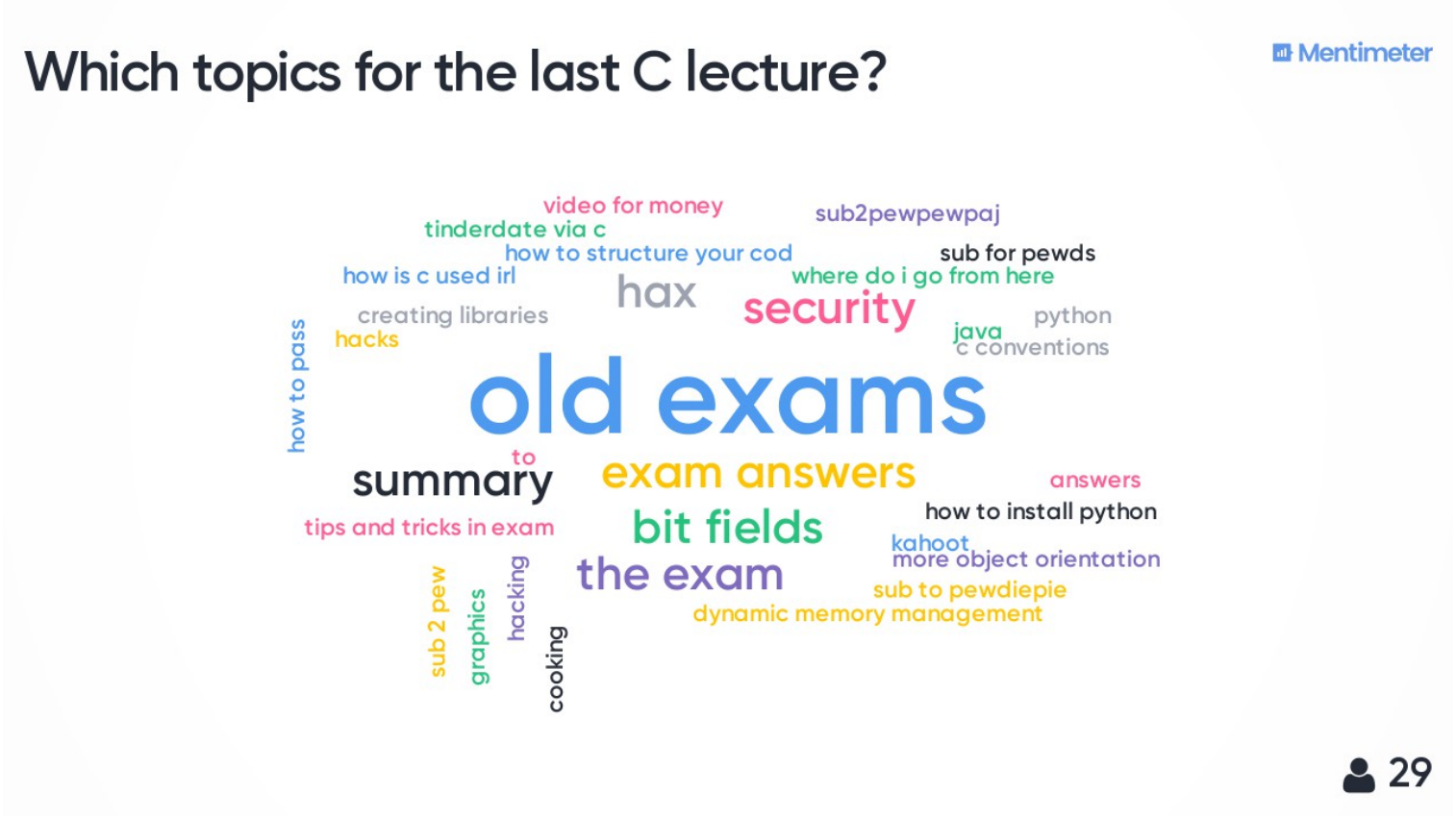
C-Programming

Pedro Moura Trancoso
ppedro@chalmers.se

Vector addition and unions

```
#include <stdio.h>

typedef union data_t {
   unsigned int compound;
   unsigned char element[4];
} data_t;

int main() {
   data_t a, b, c;

   a.element[0] = 0; a.element[1] = 1; a.element[2] = 2; a.element[3] = 3;
   b.element[0] = 4; b.element[1] = 5; b.element[2] = 6; b.element[3] = 7;

   c.compound = a.compound + b.compound;

   printf("a = %2d %2d %2d %2d\n", a.element[0], a.element[1], a.element[2], a.element[3]);
   printf("b = %2d %2d %2d %2d\n", b.element[0], b.element[1], b.element[2], b.element[3]);
   printf("c = %2d %2d %2d %2d\n", c.element[0], c.element[1], c.element[2], c.element[3]);
}
```

Your choice!


Which topics for the last C lecture?
Mentimeter
video for money
sub2pewpewpaj
tinderdate via c
how to structure your cod
sub for pewds
how is c used irl
hax
where do i go from here
security
creating libraries
python
hacks
java
c conventions
how to pass
old exams
to
summary
exam answers
answers
how to install python
tips and tricks in exam
bit fields
kahoot
more object orientation
the exam
sub to pewdiepie
dynamic memory management
sub 2 pew
graphics
hacking
cooking
29

Extra Material

- Big Endian vs Little Endian
- Constant and Inlining
- Constants and Macros with #define & #pragma
- Command line arguments to main()
- Object-oriented in C
- Recursive function
- Linked lists
- Storage, packing, padding
- List of Don’t s

Endian

- Big endian / Little endian

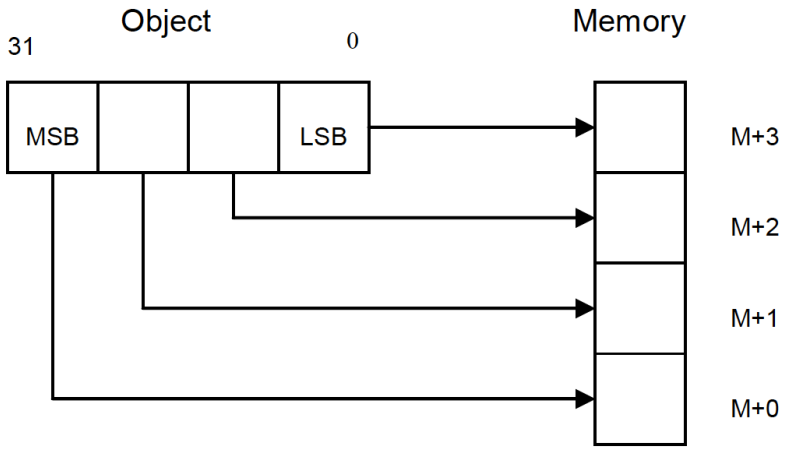


Figure 1, Memory layout of big-endian data object

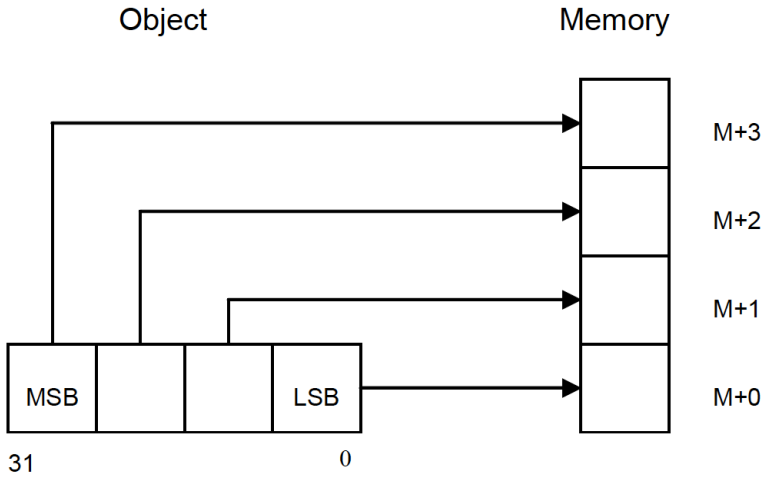


Figure 2, Memory layout of little-endian data object

We use little endian. ARM Cortex-m4 can handle both via settings.

Union + Endian…

```
#include <stdio.h>

union {
   int a;
   char c[4];
} x;

int main() {
   x.a = 256;
   printf("%d: %d %d %d %d\n", x.a, x.c[0], x.c[1], x.c[2], x.c[3]);
}
```

```
256:  0x 00 00 01 00
In Memory (little endian):
     00
     00
     01
     00
```

```
256: 0 1 0 0
```

In big endian?

Some more qualifiers

- const
- inline

```
const double pi = 3.14159265359; // the variable's value can not change

x = 90 * pi / 180; // can be calculated already at compile-time

// now the pointer can not change it's value to another address
volatile unsigned char * const inport = (unsigned char*) 0x40021010;
```

```
static inline int square(int x)
{
    return x * x;
}

int main()
{
    int a = square(5);
}
```

foo () can not be inline (< gcc 2010)

```c
// main.c
#include <stdio.h>
#include "foo.h"

int main()
{
     printf("x is %d", foo(0));
     return 0;
}
```

```c
// foo.h
inline int foo(int x);
```

```c
// foo.c
#include <stdlib.h>

inline int foo(int x)
{
     if( x == 0 ){
          int x = 4;
          return x;
     }

     return x;
}
```

The reason is that when the compiler compiles main.c, the definition of foo () is not available - just the declaration. (The declaration is in foo.h. The definition is in foo.c.)

In fact, often leads to compilation errors.

Now can foo() be inline

```c
// main.c
#include <stdio.h>
#include "foo.h"

int main()
{
     printf("x is %d", foo(0));
     return 0;
}
```

c-file
includes header-file

```c
// foo.h
static inline int foo(int x)
{
     if( x == 0 ){
          int x = 4;
          return x;
     }

     return x;
}
```

header-file

`static` is often necessary

Now main.c sees the whole definition of foo() .

BUT HEADER FILES SHOULD NEVER HAVE CODE!!!

Constants or Macros with #define

```
#define MAX 1000

int arrayOfIn[MAX];

int main() {
   for( int i = 0; i < MAX; i++ )
      arrayOfInt = …
}
```

```
#define _max_(_x_,_a_,_b_) { if(_a_ > _b_) \
                               _x_ = _a_;  \
                             else          \
                               _x_ = _b_; }

int main() {
   int a = 10;
   int b = 20;
   int r;

   _max_(r,a,b);

}
```

#define from compiler command line

```
#include <stdio.h>

int main() {
  int x = 10;

#ifdef DEBUG
  printf(”In main, before calling foo() x=%d\n”, x);
#endif

  foo(x);

#ifdef DEBUG
  printf(”In main, after calling foo() x=%d\n”, x);
#endif

}
```

```
$gcc –DDEBUG hwfile.c –o hwfile
```

```
$gcc hwfile.c –o hwfile
```

#define from compiler command line

```
#include <stdio.h>

#ifdef DEBUG
#define DEBUG1(_msg_,_p1_) { printf(_msg_,_p1_); }
#else
#define DEBUG1(_msg_,_p1_) { }
#endif

int main() {
  int x = 10;

  DEBUG1(”In main, before calling foo() x=%d\n”, x);
  foo(x);
  DEBUG1(”In main, after calling foo() x=%d\n”, x);
}
```

```
$gcc –DDEBUG hwfile.c –o hwfile
```

```
$gcc hwfile.c –o hwfile
```

#pragma

```
#include <stdio.h>

int main() {
  …
  #pragma omp parallel for
  for( int i = 0; i < MAX; i++ )
    X[i] = A[i] + B[i];
  …
}
```

You could also define your own directives!

Command line arguments

- **Addition program from command line**

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

int main(int argc, char *argv[]) {
  int a, b;

  if( argc < 3 ) {
     printf("Usage; %s <num> <num>\n", argv[0]);
     exit(-1);
  }
  a = atoi(argv[1]);
  b = atoi(argv[2]);
  printf("Result = %d\n", a+b);
}
```

```
$./add 2 3
Result = 5
$
```

Object-oriented

- Has been developed continuously since the 50's.
- Huge topic. You are taught in the course Object Orientation (and further courses ...)
- Should be your first pick to design code. (However, there are other good alternatives.)
- For us here and now: a way to structure the functions along with associated data.
- You want local functions that are together with the data in the respective structure. The concept is called **class**. A class can contain both **members/variables** (fields) and **methods** (functions). C does not have classes, so we can simulate this using structs with function pointers as the methods.

For example, we would like to do this :

```
typedef struct {
    int   a;        // a member
    void inc() { // method that increments a
        a++;
    }
} MyClass;

MyClass v = {0}; // initialize variable
v.inc();         // increment variable
```

Not in C!

However, in C, we can do this :

```
typedef struct tMyClass{
    int   a;
    void (*inc) (struct tMyClass* this);
} MyClass;

void incr(MyClass* this)
{
    this->a++;
}

MyClass v = {0, incr};
v.inc(&v);
```

Special attention!

Object-oriented with C

Clarification: This is equivalent ...

```
typedef struct tMyClass {
    int      a;
    void (*inc) (struct tMyClass* this);
} MyClass;
```

tMyClass used to enter a valid parameter type for **this**, since MyClass is undefined until the last line.

... to this:

```
struct MyClass;
typedef struct {
    int      a;
    void (*inc) (MyClass* this);
} MyClass;
```

Tells the compiler that MyClass is a struct because it is not defined before the last line ...

...but is needed here

Both ways are likely OK.

Object-oriented with C

How the method call .inc(…) works in C:

```
// MyClass is a so called object.
// v1, v2 are two instances of the object.
MyClass v1 = {0, incr}, v2 = {1, incr};
// Here we call method inc() for v1 and v2.
v1.inc(&v1);
v2.inc(&v2);
```

```
// incr() is a regular function (which
function pointer v1.inc and v2.inc point to).
void incr(MyClass* this)
{
    this->a++;
}
```

These initiations make v1.a = 0 and v2.a = 1, and v1.inc and v2.inc point to the incr() function. Remember, incr is simply a symbol for the memory address where incr() is located, and v1.inc and v2.inc are pointer variables that point to that address.

Here are the calls to functions which function pointers v1.inc and v2.inc point to, and using &v1 and &v2 as input parameters. So v1.inc(&v1) results in calling incr(&v1) and v2.inc(&v2) results in calling incr(&v2).

The call incr(&v1) means that input parameter this equals &v1 (i.e. The address to v1).
Thus this->a++ is the same as (&v1)->a++ (which is equivalent to v1.a++)…
… which is exactly what we want to happen when we do v1.inc(&v1);
The same applies to v2.inc(&v2); I.e. it increments v2.a.

Object-oriented in C: Example

Examples of homework:

```c
typedef struct tGameObject{
    // members
    GfxObject   gfxObj;
    vec2f       pos;
    float       speed;
    // methods (i.e. function pointers)
    void (*update) (struct tGameObject* this);
} GameObject;

// update should point to any of these functions:
void updateShip (GameObject* this)
{
    this->pos += …
    …
}
void updateAlien(GameObject* this)
{
    this->pos += …
    …
}
```

```c
GameObject ship, alien;

GameObject* objs[] = {&ship, &alien};

void main()
{
    // initialize the function pointer for ship and
    // alien to the right function (initialize even
    // other struct members)
    ship.update = updateShip;
    alien.update = updateAlien;
    …
    // update all objects
    for(int i=0; i<2; i++)
        objs[i]->update(objs[i]);

    …
}
```

This results in the call of ship.update(&ship) and alien.update(&alien), which due to the update-function-pointer results in calling: updateShip(&ship) and updateAlien(&alien)

Object-oriented in C: Example

Example 34 in “Arbetsboken”:

```
typedef struct tObj {
      PGEOMETRY geo;
      int    dirx,diry;
      int    posx,posy;
      void (* draw ) (struct tObj *);
      void (* move ) (struct tObj *);
      void (* set_speed ) (struct tObj *, int, int);
} OBJECT, *POBJECT;
```

```
// store all objects in global array
OBJECT* obj[] = {&ball, &player};
…
// In some function:
//  - move and draw all objects
for(int i=0; i<2; i++)
{
    obj[i]->move(obj[i]);
    obj[i]->draw(obj[i]);
};
```

```
OBJECT ball =
{
    &ball_geometry,        // geometry for a ball
    0,0,                   // move direction (x,y)
    1,1,                   // position (x,y)
    draw_object,           // draw method
    move_object,           // move method
    set_object_speed       // set-speed method
};
```

```
OBJECT player =
{
    &player_geo,           // geometry for a ball
    0,0,                   // move direction (x,y)
    10,10,                 // position (x,y)
    draw_player,           // draw method
    move_player,           // move method
    set_player_speed       // set-speed method
};
```

Structs with function pointers:

revision of class methods

```
In C:
typedef struct tCourse {
      …
      void (*addStudent)(struct tCourse* crs,
                         char* name);
      …
} Course;

void funcAddStudent(Course* crs, char* name) // some C function
{
   …
}

void main()
{
      Course mop;
      mop.addStudents = funcAddStudent; // set the function pointer to our desired function
      …
      mop.addStudent(&mop, "Per");      // call addStudent() like a class method
      …
}
```

Like a class method!

- **but** needs 4 bytes for the function pointer in the struct. Java/C++ store all the class methods in one separate “ghost” struct.

Structs with function pointers:

revision of class methods

```
In C:
typedef struct tCourse {
      char* name;
      float credits;
      int   numStudents;
      char* students[100];
      void (*addStudent)(struct tCourse* this, char* name);
} Course;

void funcAddStudent(Course* this, char* name) {
   this->students[this->numStudents++] = name;
}

void main() {
      Course mop;
      mop.name    = "Maskinorienterad Programmering";
      mop.credits = 7.5f;
      mop.numStudents = 0;
      mop.addStudent = funcAddStudent; // set the function pointer to
      mop.addStudent(&mop, "Per");
}
```

In Java:

```
public class Course {
  String name;
  float credits;
  int numStudents;
  String[] students;
  void addStudent(String name) {
    students[numStudents++]=name;
  }
}

Course mop = new Course();
mop.name = …
mop.credits = 7.5;
mop.addStudent(”Per”);
```

OK, but we could as well have been able to call funcAddStudent(…) here, So what's the point of going through pmop->addStudent(...)?

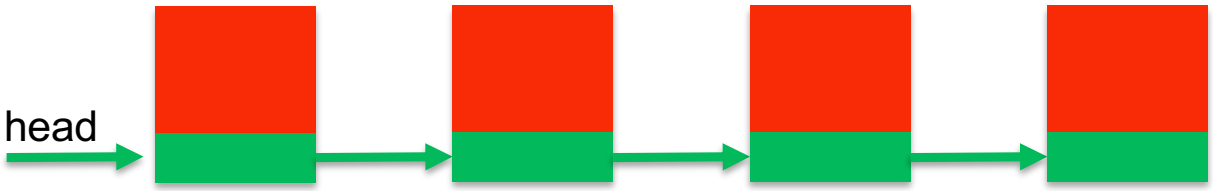
Recursion = ”Function in a function”?

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

char *str="Machine Oriented Programming";

void printReverse( char *str ) {
  int i;
  for( i = strlen(str)-1; i >=0; i-- )
    printf("%c", str[i]);
}

int main() {
  printReverse( str );
  printf("\n");
}
```

```
#include <stdio.h>

char *str="Machine Oriented Programming";

void printReverse( char *str ) {
  if( *str ) {
    printReverse( str+1 );
    printf("%c", *str);
  }
}

int main() {
  printReverse( str );
  printf("\n");
}
```

```
C18GLMM:mop ppedro$ ./reverse2
gnimmargorP detneirO enihcaM
```

Linked Lists




```
typedef struct node_t {
  char first_name[100];
  char last_name[100];
  int  id_number;
  struct node_t* next;
} node_t;
```

```
int find_in_list(int num) {…}
int insert_in_list(node_t* node) {}
int remove_from_list(int num) {}
```

What if we have one list per course?

```
int find_in_list(node_t* head, int num) {…}
node_t* insert_in_list(node_t* head, node_t* node) {}
node_t* remove_from_list(node_t* head, int num) {}
```

Memory Leaks

- A memory leak occurs when we do not free a memory item that we have dynamically allocated (with malloc()).
- Memory leaks can cause system crash if it runs out of memory.
- Memory leakage disappears when program terminates. (or it should…)

Why and How?

Find a memory leak

- You can use a memory analyzer such as DrMemory (http://www.drmemory.org/) or Valgrind (http://valgrind.org)
- DrMemory replaces the default library, and analyzes calls to malloc() and free().

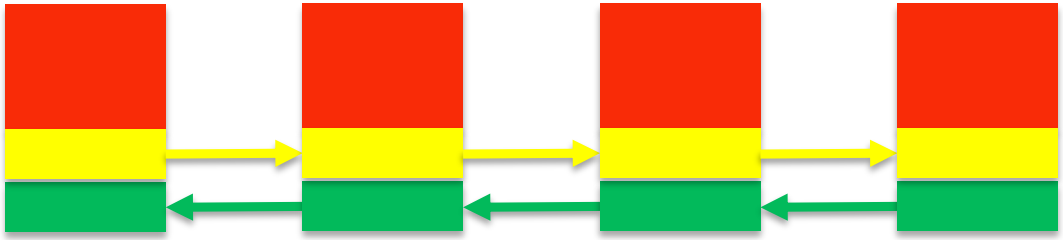
- It also finds accesses to uninitialized memory

DrMemory Example

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

void foo() {
  int *y;
  y = (int*) malloc( sizeof(int)*10 );
  //free(y);
}

int main() {
  char *x = "This is a long string";
  foo();
  foo();
  foo();
  foo();
}
```

```
$ gcc -m32 -g -fno-inline -fno-omit-frame-pointer t1.c -o t1
$ bin/drmemory -- ./t1
~~Dr.M~~ Error #3: POSSIBLE LEAK 40 direct bytes 0x005701c8-
0x005701f0 + 0 indirect bytes
~~Dr.M~~ # 0 replace_malloc
~~Dr.M~~ # 1 foo                                [/Users/t1.c:6]
~~Dr.M~~ # 2 main                               [/Users/t1.c:17]
~~Dr.M~~
~~Dr.M~~ ERRORS FOUND:
~~Dr.M~~  0 unique, 0 total unaddressable access(es)
~~Dr.M~~  0 unique, 0 total uninitialized access(es)
~~Dr.M~~  0 unique, 0 total invalid heap argument(s)
~~Dr.M~~  0 unique, 0 total warning(s)
~~Dr.M~~  0 unique, 0 total,   0 byte(s) of leak(s)
~~Dr.M~~  4 unique, 4 total, 160 byte(s) of possible leak(s)
~~Dr.M~~ ERRORS IGNORED:
~~Dr.M~~  0 unique, 0 total,   0 byte(s) of still-reachable
allocation(s)
```

Doubly Linked Lists

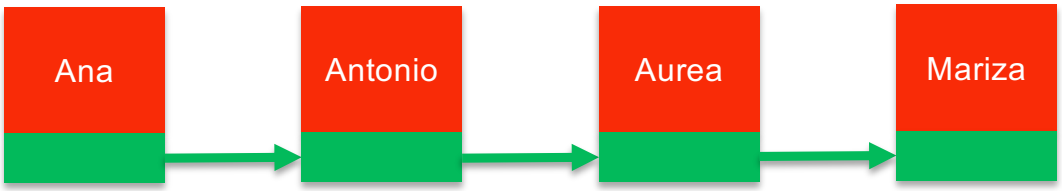


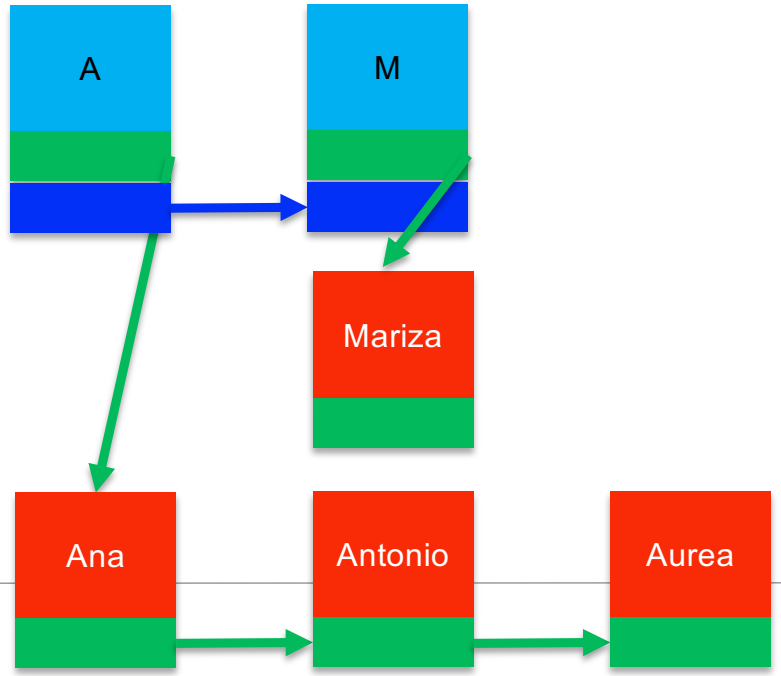
```
typedef struct node_t {
  char firstName[100];
  char lastName[100];
  int  idnumber;
  struct node_t* next;
  struct node_t* prev;
} node_t;
```

```
int findInList(int num) {…}
int insertInList(node_t* node) {}
int removeFromList(int num) {}
```

Linked Lists of Linked LIsts

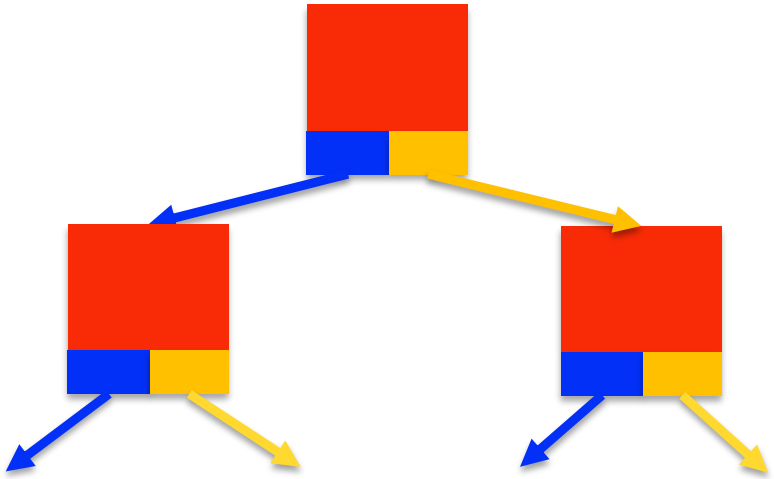
Problem: How to organize a structure for handling different people (e.g. in a Database for the students of MOP!)



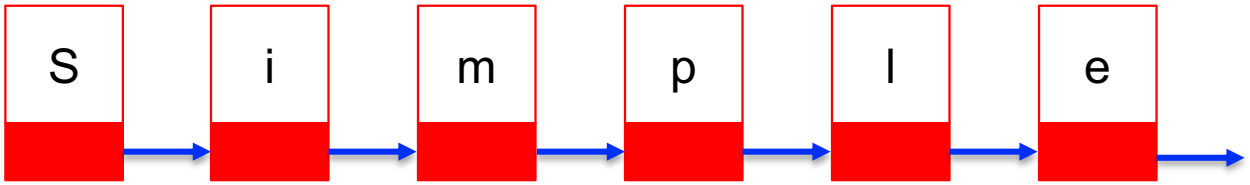


Trees and Graphs

More efficient structures



Example: Simple Text Editor



Structs: Packing and Padding

```
#include <stdio.h>

typedef struct info_t {
  char name1[3];
  int  number1;
  char name2[2];
  int  number2;
} info_t;

int main() {
  info_t x[2];

  printf("sizeof(struct)=%d\n", sizeof(info_t));
  printf("sizeof(elements)=%d\n",
sizeof(x[0].name1)+sizeof(x[0].number1)+sizeof(x[0].name2)+sizeof(x[0].number2));
}
```

```
typedef struct __attribute__((__packed__)) info_t {
  char name1[3];
  int  number1;
  char name2[2];
  int  number2;
} info_t;
```

16 & 13!

DONTs: Out-of-bound Access

```
#include <stdio.h>

int array1[] = { 0,1,2,3,4,5,6,7,8,9 };
int array2[] = { 10,11,12,13,14,15,16,17,18,19 };
int array3[] = { 20,21,22,23,24,25,26,27,28,29 };


int main(int argc, char *argv[]) {
   for(int i=0; i<30; i++)
      printf( "i=%d array1[i]=%d\n", i, array1[i]);

}
```

```
i=0 array1[i]=0
i=1 array1[i]=1
i=2 array1[i]=2
i=3 array1[i]=3
i=4 array1[i]=4
i=5 array1[i]=5
i=6 array1[i]=6
i=7 array1[i]=7
i=8 array1[i]=8
i=9 array1[i]=9
i=10 array1[i]=0
i=11 array1[i]=0
i=12 array1[i]=10
i=13 array1[i]=11
i=14 array1[i]=12
i=15 array1[i]=13
i=16 array1[i]=14
i=17 array1[i]=15
i=18 array1[i]=16
i=19 array1[i]=17
i=20 array1[i]=18
i=21 array1[i]=19
i=22 array1[i]=0
i=23 array1[i]=0
i=24 array1[i]=20
i=25 array1[i]=21
i=26 array1[i]=22
i=27 array1[i]=23
i=28 array1[i]=24
i=29 array1[i]=25
```

DONTs: Access to static local variables

```
#include <stdio.h>

void foo() {
  static int a = 0;
  a++;
  printf("In foo: a=%d\n", a);
}

int main(int argc, char *argv[]) {
   foo();
   foo();
   foo();
   foo();
}
```

```
#include <stdio.h>

int* foo() {
  static int a = 0;
  a++;
  printf("In foo: a=%d\n", a);
  return &a;
}

int main(int argc, char *argv[]) {
   int* x;
   x = foo();
   *x = 0;
   x = foo();
   *x = 0;
   x = foo();
   *x = 0;
   x = foo();
}
```

DONTs: Stack change return address

```
#include <stdio.h>

void foo() {
  unsigned int a = 1;
  unsigned int b = 2;
  unsigned int *x = &b;
  printf("In foo: *x=%u\n", *x); x++;
  printf("In foo: *x=%u\n", *x); x++;
  printf("In foo: *x=%u\n", *x);
  *x+=4;
}

int main(int argc, char *argv[]) {
   int x = 0;
   foo();
   x+=10;
   x+=10;
   printf("x = %d\n", x);
}
```

DONTs: Don’t write code any more!!!

iPad 06:24 72%

New A.I. application can write its own code - Futurity

4 hours ago

Computer scientists have created a deep-learning, software-coding application that can help human programmers navigate the growing multitude of often-undocumented application programming interfaces, or APIs.

Designing applications that can program computers is a long-sought grail of the branch of computer science called artificial intelligence (AI). The new application, called Bayou, came out of an initiative aimed at extracting knowledge from online source code repositories like GitHub. Users can try it out at askbayou.com.

> “The days when a programmer could write code from scratch are long gone.”

bayou http://askbayou.com

Department of Computer Science,
Rice University, USA.

What would you like to see in a MOP course?

Old exams…

Exam 2018-01-09

Uppgift 4 (6p)

Konstruera en C-funktion som undersöker en parameter med avseende på antalet 1-ställda bitar.

Funktionen deklareras:

```
int   bitcheck( unsigned int *pp, int * num );
```

`pp` är en pekare till det värde som ska undersökas

`num` är en pekare till en plats för returvärde, dvs. antalet 1-ställda bitar hos parametern

Funktionen ska returnera 1 om antalet ettor hos parametern är udda, annars ska funktionsvärdet vara 0.

Uppgift 4:

```
int bitcheck( unsigned int *pp, int *num)
{
  int   retval = 0;

  while(*pp)
  {
    if( *pp & 1 )
      retval++;
    *pp >>= 1;
  }
  *num = retval;
  return retval & 1;
}
```

Exam 2018-03-12

Uppgift 4 (6p)

Konstruera en C-funktion som undersöker antalet 1-ställda bitar hos en parameter.

Funktionen deklareras:

```
int   oddbit(int a, unsigned int * num );
```

`a` är det värde som ska undersökas, `num` är en pekare till en plats för ett resultat, dvs. antalet 1-ställda bitar hos parametern. Funktionen ska returnera 1 om antalet ettor hos `a` är udda, annars ska funktionsvärdet vara 0. Funktionen ska vara portabel, dvs. du kan inte göra antaganden om storleken av en `int`.

Uppgift 4:

```
int oddbit (int a, int *num)
{
  int  numbits = 0;

  while(a)
  {
    if( a & 1 )
      numbits ++;
    (unsigned int) a >>= 1;
  }
  *num = numbits;
  return *num & 1;
}
```

Exam 2018-04-04

Uppgift 4 (6p)

Konstruera en C-funktion som undersöker antalet 0-ställda bitar hos en parameter.

Funktionen deklareras:

```
int   oddbit(int a, unsigned int * num );
```

`a` är det värde som ska undersökas, `num` är en pekare till en plats för ett resultat, dvs. antalet 0-ställda bitar hos parametern. Funktionen ska returnera 1 om antalet 0-ställda bitar hos `a` är udda, annars ska funktionsvärdet vara 0. Funktionen ska vara portabel, dvs. du får inte göra antaganden om den exakta (maskinberoende) storleken av en `int`.

Uppgift 4:

```
int oddbit (int a, int *num)
{
  int   numbits = 0;
  /* Enklast att räkna antalet ett-ställda bitar, resten,
    dvs. sizeof(int)*8 - antal ettställda bitar ger resultatet */
  while(a)
  {
    if( a & 1 )
      numbits ++;
    (( unsigned int) a) >>= 1;
  }
  *num = (sizeof(int)*8) - numbits;
  return *num & 1;
}
```

Exam 2018-06-02

Uppgift 3 (8p)

För ett fält med tecken (`char`) kan man definiera en operation *rotation vänster* på fältet som en operation som placerar fältets första tecken på den sista platsen i fältet, det andra tecknet på den första platsen, det tredje tecknet på den andra platsen o.s.v.
Skriv en funktion `rotateleft` som roterar ett fält med tecken ett godtyckligt antal steg åt vänster. Funktionen skall ha tre parametrar, en pekare till fältet som skall roteras s
respektive antalet rotationssteg. Parametrarna ska ges i denna o
inte ha sidoeffekter.
Lösningen får inte använda anrop till standardfunktioner från bib
utan måste implementeras med hjälp av pekartyper.

Uppgift 3:

```
void rotateleft(char *f, int l, int steps) {
  int i
  char *current, *next, save;

  for (;steps > 0; steps--) {
    current  = next = f;
    next++;
    i = l;
    save = *current;   /* kommer att bli det sista elementet */
    while( i>1 )       /* för alla element utom det sista... */
    {
      *current++ = *next++;   /* skifta element vänster */
      i--;
    }
    *current = save;   /* sista element på plats */
  }
}
```

Exam 2019-03-18

Uppgift 3 (6p)

Funktionen:
`void int_concat (int *i1, const int * s2, unsigned int n, unsigned int pos);`
konkatenerar (lägger till) ett fält med `n` heltal från `i2` till postition `pos` i fältet `i1`.
Implementera funktionen `int_concat` med användning av pekare, du får inte använda indexering. Din lösning får heller inte använda någon standardfunktion.

Uppgift 3:

```
void int_concat (int *i1, const int *s2, unsigned int n, unsigned int pos)
{
  int *insert = i1 + pos; int *i2 = s2;

  while ( n-- )
  {
    *insert++ = *i2++;
  }
}
```