CHALMERS
UNIVERSITY OF TECHNOLOGY

Machine-Oriented Programming

C-Programming part 4

Pedro Trancoso
ppedro@chalmers.se

Original slides by Ulf Assarsson

Objectives

- Topics
  - Sets, bit management: enum, union, little/big endian
  - “Visibility”/scope: extern, static
  - Pragma directives: #if, #ifdef, #include, #define
  - Dynamic memory allocation: malloc and free
  - File operations: FILE, fopen, fprintf, fgets
- Related to:
  - “Arbetsbok” section 5 graphics display
- Homework: v4

Type Union

Union allows to store different data types in the same memory location. Same syntax as struct

```
union opt_name {
      int a;
      char b;
      float c;
 } x;
 x.a = 4;
 x.b = 'i';
 x.c = 3.0;
```

x.a = 1077936128

```
typedef union {
      float v[2];
      struct {float x,y;};
} Vec2f;

Vec2f pos;
pos.v[0] = 1; pos.v[1] = 2;
pos.x = 1; pos.y = 2;
```

&x.a == &x.b == &x.c
a, b and c share the same memory address. That is the same memory address can be accessed in three different ways via three different variable names.

Example: pos.v[0] and pos.x are the same. “pos.x” clearly shows that it is the x-coordinate we address. “pos.v[i]” is useful if we want to write a loop over the x and y coordinates.
For example:

```
Vec2f addVec(Vec2f a, Vec2f b)
{
   for(i=0; i<2; i++)
      a.v[i] += b.v[i];
   return a;
}
```

Endian

- Big endian / Little endian

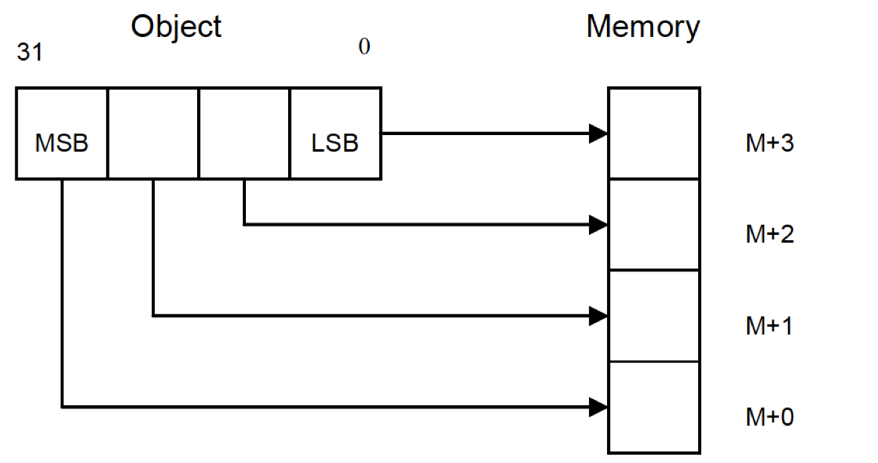


Figure 1, Memory layout of big-endian data object

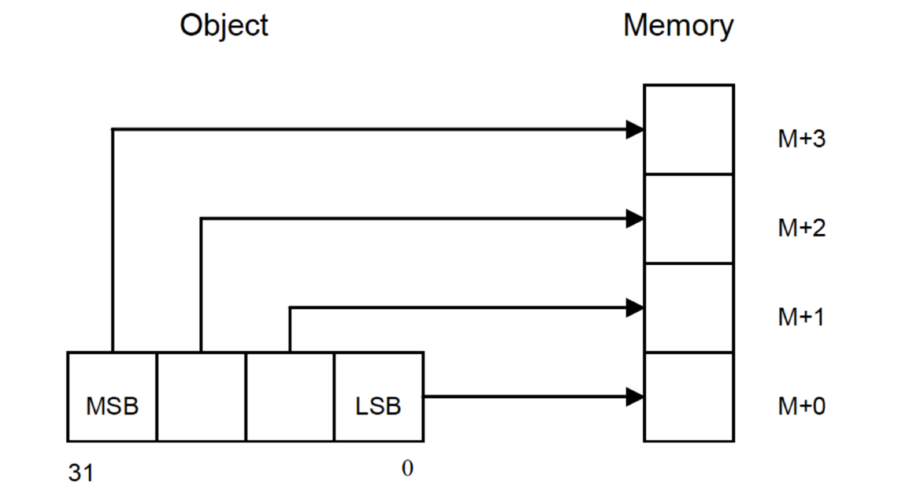


Figure 2, Memory layout of little-endian data object

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?
```

Byte-addressing with unions

For GPIO ports

```
// GPIO
typedef struct _gpio {
   uint32_t    moder;
   uint32_t    otyper;
   uint32_t    ospeedr;
   uint32_t    pupdr;
   union {
      uint32_t    idr;
      struct {
         byte    idrLow;
         byte    idrHigh;
         short  reserved;
      };
   };
   union {
      uint32_t    odr;
      struct {
         byte    odrLow;
         byte    odrHigh;
      };
   };
} GPIO;
#define GPIO_D (*((volatile GPIO*) 0x40020c00))
#define GPIO_E (*((volatile GPIO*) 0x40021000))
```

GPIO Input Data Register (IDR)

offset	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	mnemonic
0x10	RESERVERADE																r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	IDR
																	0x11								0x10								

Bits 16 to 31 are not used and will hold its RESET value, that is 0

Now idrHigh can be addressed with:

```
byte c = GPIO_E.idrHigh;
```

Instead of with:

```
byte c = *((byte*)&(GPIO_E.idr) + 1));
```

As for example:

```
GPIO_E.odrLow &= ( ~B_SELECT & ~x);
```

Instead of:

```
*((byte*)&(GPIO_E.odr)) &=(~B_SELECT & ~x);
```

Enumerations: enum

```
enum type_name { value1, value2,..., valueN }; //type_name optional. By default, value1 = 0, value2 = value1 + 1, etc.

enum type_name { value1 = 0, value2, value3 = 0, value4 }; //However, with gcc values: 0, 1, 0, 1 – not intuitive.

enum day {monday=1, tuesday, wednesday, thursday, friday, saturday, sunday};
enum day today; // day becomes a char, short or int
today=wednesday;
printf("%d:th day",today+1); // output: "4:th day"

typedef enum { false, true } bool;
bool ok = true;
--------------------------------------------------------------------------------
#define B_E         0x40        // using enums
#define B_RST       0x20
#define B_CS2       0x10
#define B_CS1       8
#define B_SELECT    4
#define B_RW        2
#define B_RS        1
```

Can be replaced with:

```
enum {B_RS=1, B_RW=2, B_SELECT=4, B_CS1=8, B_CS2=0x10, B_RST=0x20, B_E=0x40};
```

Enum

```
#include <stdio.h>

typedef enum { one=1, two, three, four, five=10, six, seven, eight, nine, ten } numbers;

int main() {
  printf("two=%d eight=%d\n", two, eight );
}
```

```
two=2 eight=13
```

```
#include <stdio.h>

typedef enum { monday, tuesday, wednesday, thursday, friday, saturday, sunday } daysofweek;
char *daysname[] = { "Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun" };

int main() {
  for(int i = monday; i <= wednesday; i++)
    printf("Day = %s\n", daysname[i]);
}
```

```
Day = Mon
Day = Tue
Day = Wed
```

Extern

extern in C means that the symbol is coming from another file.

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

extern int g_var;
void myFunc();

void main()
{
    g_var = 5;
    myFunc();
}
```

Just declaration – without definition.
g_var defined later – for example in another .c-file

extern not necessary for functions, the prototype means: the code exists at link-time.

```
// myFunc.c
int g_var;
void myFunc()
{
    printf("hej");
};
```

Extern

Note: If you code a larger program for lab5, you may not want all global items/variables declared in the main file. Then you need to use external.

`extern` in C means that the symbol comes from another file.

```
// main.c
#include "fractalmountain.h”
…

POBJECT objects[] = {&player, &fmountain};
unsigned int nObjects = 2;

void main()
{
   …
}
```

```
// fractalmountain.h
#ifndef FRACTALMOUNTAIN_H
#define FRACTALMOUNTAIN_H

#include "object.h"


extern OBJECT fmountain;

#endif //FRACTALMOUNTAIN_H
```

```
// fractalmountain.c
#include "fractalmountain.h"

OBJECT fmountain = {
  0,              // geometry - no
  0,0,            // direction vector
  0,0,            // initial startposition
  drawMountain, // draw method
  0,              // clear_object – unused
  moveMountain, // move method
  set_object_speed // set-speed method
};
```

extern = declare without defining
(without extern we would create a **new** variable.)

Extern: C function called from C++ code

```
extern ”C” int foo( char, float );


int main() {
  char c = ’a’;
  float f = 3.1415;
  int r;

  r = foo( c, f );
  …
}
```

C: visibility for declarations

All declarations (variables, functions) and expressions like typedefs + defines are visible first below them - not above them. Source files are processed from top to bottom.

Example:

```
void fkn1(short param)
{
     …
     if(…) {

        return fkn2('a'); // fkn2 is here unknowned for the C-compiler so it results in a compilation error
     }
     …
     return;
}

void fkn2(char c)
{
     …
     return;
}
```

C: visibility for declarations

All declarations (variables, functions) and expressions like typedefs + defines are visible first below them - not above them. Source files are processed from top to bottom.
Example:

```
void fkn2(char c);         // Fix to make the declaration of fkn2 already known here.
void fkn2(char c);         // We may have the declaration how many times we want
void fkn1(short param)
{
     …
     if(…) {

        return fkn2('a’); // now fkn2 is known here
     }
     …
     return;
}

void fkn2(char c)          // The definition of fkn2
{
     …
     return;
}
```

C: visibility for declarations

All declarations (variables, functions) and expressions like typedefs + defines are visible first below them - not above them. Source files are processed from top to bottom.
Example:

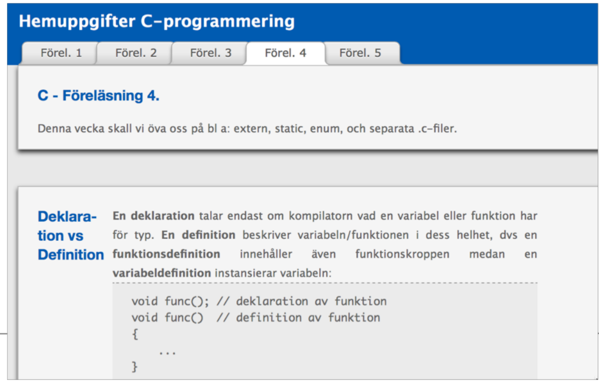
```
void fkn2(char c);         // Fix to make the declaration of fkn2 already known here.
                           // We may have the declaration how many times we want
void fkn1(short param)
{
    void fkn2(char c);      // We can even have it here instead
     if(…) {
        void fkn2(char c);   // or here!
        return fkn2('a'); // now fkn2 is known here
     }
     …
     return;
}

void fkn2(char c)          // The definition of fkn2
{
     …
     return;
}
```

static

static in C can do two things:

1. Give visibility only in its own and inner scopes
   - For example, symbols (variables / features) can not be seen outside its .c file.
2. A static variable is in the data segment as opposed to volatile on the stack
   - Allows you to create local variables that retain their value between the function calls

See also homework assignment 4



Static variant 1

```
static int var;
```

The variable *var* can not be seen from another file, regardless of whether you use `extern`.

In ”Java terminology” this is close to ”private” but the scope is the file.

Static variant 2

```c
#include <stdio.h>

void testFkt()
{
     int var1 = 0;
     static int var2 = 0;
     var1++;
     var2++;
     printf("var1: %i, var2: %i \n", var1, var2);
}

int main()
{
     testFkt();
     testFkt();
     testFkt();
}
```

Output:

```
var1: 1, var2: 1
var1: 1, var2: 2
var1: 1, var2: 3
```

var2 is initialized to zero only the first time we call the function, but then retains its value between calls.

var2 is a **global variable** that has only local visibility in its scope – in this case testFkt().

Static variant 2 – simple example

```
#include <stdio.h>

void testFkt()
{
     static int bFirstTime = 1;
     if(bFirstTime) {
          // For example allocate memory on the heap
          // or calculate something.
          bFirstTime = 0;
     }
     … Do the main calculations/work.
}

int main()
{
     testFkt();
}
```

NOTE – The code is not thread-safe!

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!!!

Compiler Pragma Directives

- #include
- #if, #ifdef, #ifndef, #endif
- #define
- #pragma

Preprocessor directives: #if, #ifdef, #ifndef

Preprocessor conditional inclusion

```
#define X  1      // syntax: #define [identifier name] [value], where [value] is optional

#if X == 0        // syntax: #if <value>, where 0=false and !0==true. Integer arithmetic OK.
    …             // any C-code
#elif X-1 == 1    // elif means else if
    …
#else
    …
#endif

#if 0 // good for temporarily commenting out large blocks of code.
 …    // any C-code
#endif

#define HW_DEBUG
…
#ifdef HW_DEBUG
    … // any C-code, for example:
    #undef SIMULATOR
#endif
```

```
void delay_500ns(void)
{
#ifndef SIMULATOR
      delay_250ns();
      delay_250ns();
#endif
}
```

Include guards

- Include guards are used to automatically avoid to include a header file (.h-file) more than once per .c-file. This may happen if a .c-file includes several .h-files which in turn include the same .h-file

Include guards

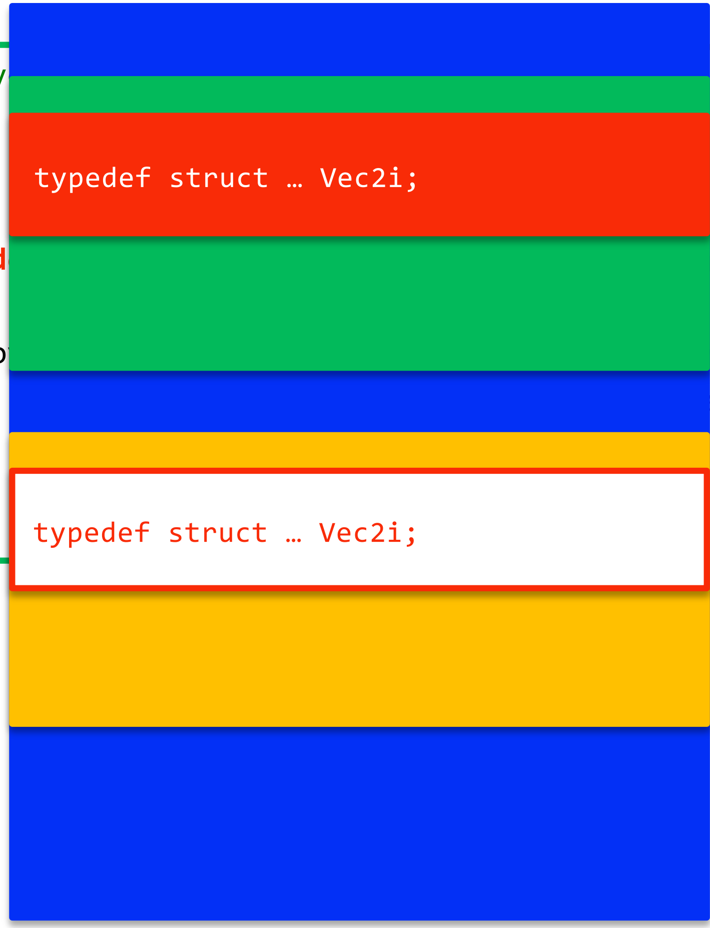
```
// main.c
#include "player.h"
#include "enemies.h"

void main()
{
    …
}
```

c-file

```
// vecmath.h



typedef struct {
    union {
        int v[2];
        struct {int x, y;};
    };
} Vec2i;

Vec2i add2i(Vec2i a, Vec2i b);
char isEqual(Vec2i a, Vec2i b);
```

```
// play

#includ

void mo
```

```
.h"

t i,
c2i v);
```



Include guards

```c
// main.c
#include "player.h"
#include "enemies.h"

void main()
{
    …
}
```

c-file

```c
// vecmath.h
#ifndef VECMATH_H
#define VECMATH_H

typedef struct {
   union {
      int v[2];
      struct {int x, y;};
   };
} Vec2i;

Vec2i add2i(Vec2i a, Vec2i b);
char isEqual(Vec2i a, Vec2i b);

#endif //VECMATH_H
```

```c
// player.h
#ifndef PLAYER_H
#define PLAYER_H

#include "vecmath.h"

void movePlayer(Vec2i v);


#endif //PLAYER_H
```

```c
// enemies.h
#ifndef ENEMIES_H
#define ENEMIES_H

#include "vecmath.h"

void moveEnemy(int i,
               Vec2i v);

#endif //ENEMIES_H
```

We should have include-guards on all .h files - although in this example is not required for player.h as well as enemies.h

Without including guards, vecmath.h is included twice for main.c. The second time, the compiler will complain that `Vec2i` is already defined.

.c-files are compiled separately. The preprocessor and compiler are executed individually per .c-file.

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!

Dynamic Memory Allocation

- `malloc()`   Allocate memory
- `free()`   De-allocate (free) memory

What happens if we do not call free?

Do you need to use some sort of free also in Java? Why?

Function prototypes available in:

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

Dynamic Memory Allocation

```c
#include <stdlib.h>

char s1[] = "This is a long string. It is even more than one sentence.";

int main()
{
    char* p;

    // allocate memory dynamically
    p = (char*)malloc(sizeof(s1));

    // make something with the memory that we have reserved

    // free the memory space
    free(p);
    return 0;
}
```

Can you guess the prototype for malloc?

After malloc ALWAYS do:

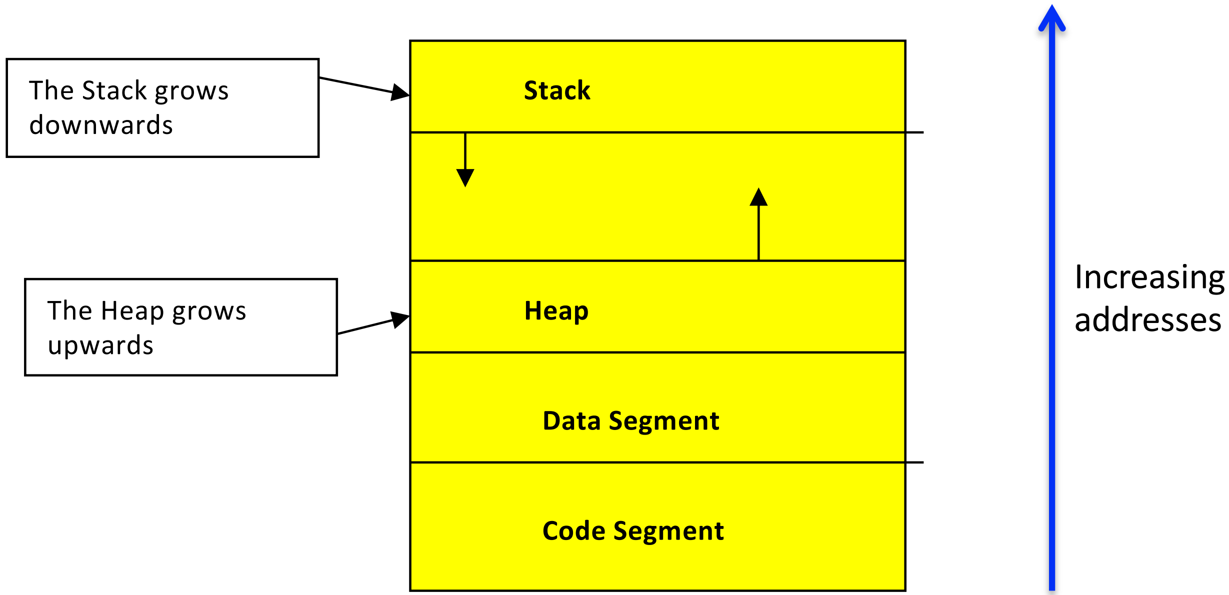
```c
if( !p ) {
   printf(”ERROR: Could not allocate p\n”);
   exit(-1);
}
```

Number of bytes we want to allocate

OUT OF MEMORY
virtual address space is divided into pages swapped to the secondary storage (e.g. hard disk). Even this may end -> crash.

Address Space of a Program

- All programs as they execute, have some associated memory, which is typically divided into:
  - Code Segment
  - Data Segment (Holds Global Data)
  - Stack (where the local variables and other temporary information is stored)
  - Heap



Dynamic Memory Allocation Example

```
// Copy from s1 to the allocated memory pointed by p.
#include <stdio.h>
#include <stdlib.h>

char s1[] = "This is a long string. It is even more than one sentence.";
int main()
{
        char* p;
        int i;
        // allocate memory dynamically
        p = (char*)malloc(sizeof(s1));

        // make something with the memory that is allocated
        for( i=0; i<sizeof(s1); i++)
                *(p+i) = s1[i];
        printf("%s", p);

        // free memory
        free(p);

        return 0;
}
```

Do I need to do something between allocation and use?

What happens if I do *(p+1000) = 1024

Why does free(p) not need information about its size?

Dynamic Memory Allocation – other types

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

typedef struct {
  char *name;
  int   number;
} xyz;

int main() {
  int    *x;
  char   *y;
  double *z;
  xyz    *u;

  //allocate 100 int, char, double and xyz

}
```

```
x = (int*) malloc( sizeof(int)*100 );
if( !x ) exit(-1);
```

```
y = (char*) malloc( sizeof(char)*100 );
if( !y ) exit(-1);
```

```
z = (double*) malloc( sizeof(double)*100 );
if( !z ) exit(-1);
```

```
u = (xyz*) malloc( sizeof(xyz)*100 );
if( !u ) exit(-1);
```

Other dynamic memory allocation functions…

```
ptr = (int*) calloc(n, sizeof(int));
```

```
ptr = (int*) malloc(n1 * sizeof(int));
. . .
ptr = realloc(ptr, n2 * sizeof(int));
```

realloc()

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

int main() {
    int *ptr, i , n1, r
    printf("Enter size
    scanf("%d", &n1)
    ptr = (int*) malloc
    printf("Addresses c
    printf("%u - %u\n",
    printf("\nEnter new
    scanf("%d", &n2);
    ptr = realloc(ptr,
    printf("Addresses c
    printf("%u - %u\n",
    return 0;
}
```

```
Enter size of array: 10
Addresses of previously allocated memory: 1599078400 - 1599078436
Enter new size of array: 20
Addresses of newly allocated memory: 1599078400 - 1599078476

Enter size of array: 10000
Addresses of previously allocated memory: 2877292544 - 2877332540
Enter new size of array: 40000
Addresses of newly allocated memory: 84406272 - 84566268

Enter size of array: 10000
Addresses of previously allocated memory: 3313500160 - 3313540156
Enter new size of array: 2000
Addresses of newly allocated memory: 3313500160 - 3313508156
```

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)
```

File operations

```c
#include <stdio.h>

#define MAX 100

int main(void) {
  FILE *fp;
  char *filename = "x1.c";
  char str[MAX];
  int linenum = 0;

  fp = fopen(filename, "r");
  if( fp == NULL ) {
    printf( "ERROR: Could not find file %s\n", filename );
  }
  while(!feof(fp)) {
    fgets( str, MAX, fp );
    printf("%d: %s", linenum++, str);
  }
  printf("\n");
  fclose(fp);
}
```

Next C Lecture – Your pick!

- Command line parameters with main input parameters
- Packing strings efficiently
- Linked lists, doubly linked lists, trees, etc.
- Any other wish list items?