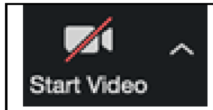
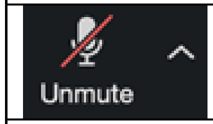
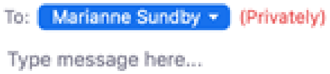
Denne forelesningsøkten vil bli tatt opp og lagt ut i emnet i etterkant.

Hvis du ikke vil være med på opptaket:

	La være å delta med webkameraet ditt.
	La være å delta med mikrofonen din.
 To: Marianne Sundby (Privately) Type message here...	Still spørsmål i Chat i stedet for som lyd. Hvis du ønsker kan spørsmålet også sendes privat til foreleser.

Høyskolen
Kristiania

PG3401 Programmering i C for Linux

Bengt Østby

Recap

Pointers

- Declaration
- Assignment
- Dereference

Recap

Address of a variable

Types of pointers

Dynamic memory operations (malloc, free)

Recap

Type casting

Pointers to pointers

Function arguments

Recap

Pointer problems:

- Dangling reference
- Memory leaks
- Buffer overflow
- Fragmentation

Arrays

Static arrays

accessing arrays

arrays as pointers

dynamic arrays

allocating arrays

Multidimensional arrays

Static Arrays

Size known at compile time

Example :

```
int array[20];
char buffer[100];
float numArray[20];
double _array[100];
```

Or at run time (C11-standard):

```
int iCount = 5;
iCount++;
int iArray[iCount];
```

Accessing arrays

Simple Base+Offset access

Example:

```
int array[10];
array[10] = 100;
int b = array[15];
```

Hva skjer når man gjør dette? (Buffer overflow…)

No safety ensured!

Arrays & Pointers

In C arrays and pointers are equivalent

Example :

```
int array[40]; // “array” is just a pointer
*array ⇔ array[0]
array ⇔ &array[0]
*(array + n) ⇔ array[n]
```

Dynamic arrays

Allocated on **heap**

Size is not required at compile time

Useful when the user is not aware of the size requirements

Always have a limit!

- “A complaining program is better than a crashing one!”
- Because paging is expensive.

Allocating arrays

Dynamic memory must be allocated

C provides memory allocation function:

- malloc – returns a pointer to the memory block requested. argument is size in bytes
- calloc – returns a pointer to the memory block requested. arguments are number of chunks & size of each chunk in bytes
- realloc – returns a pointer to the memory block requested. arguments are old pointer, desired size in bytes. Copies the old data into the start of the new block. How?

Only calloc does the initialization!

Multidimensional arrays

Static arrays can be declared easily :

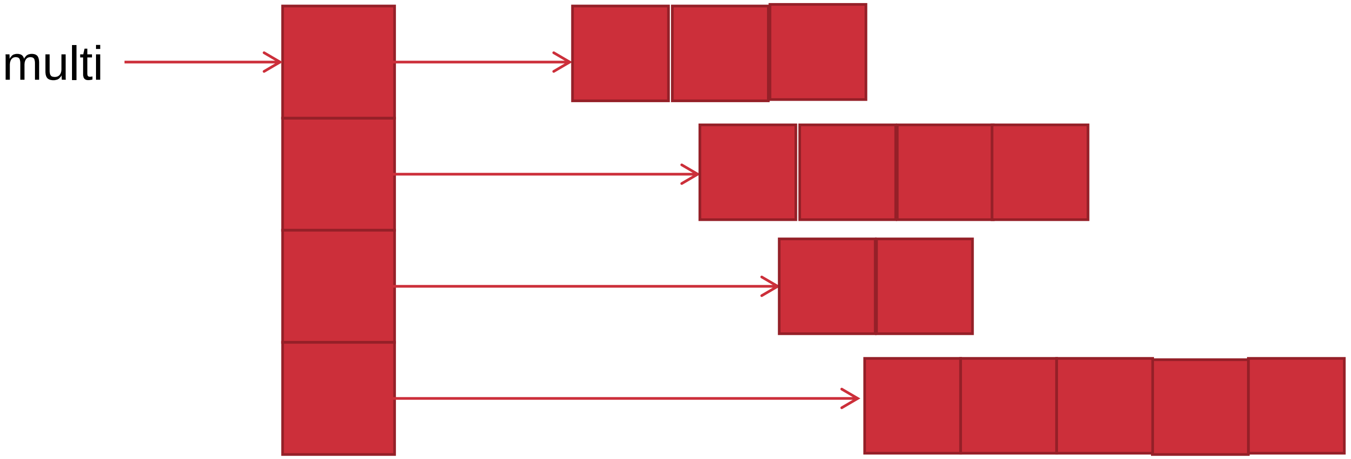
```
int array[20][40];
float _floatArray[100][200][100]
```

Accessing elements :

```
array[10][30] ⇔ array[0][430] ⇔ (*array)[430] ⇔ *(*array+430)
```

Obviously unsafe!

Pointers to pointers



double **multi;
Then what?

Multi-dimensional dynamic arrays

```
int **iArray, i, j, k;
int iRows = 5, iCols = 5;

// Allocate first the rows as an int* array:
iArray = (int **) malloc (iRows * sizeof(int *));
for (i = 0; i < iRows; i++) {
  // For each row, allocate the columns as int arrays:
  iArray[i] = (int *) malloc (iCols * sizeof(int));
}
```

Multi-dimensional dynamic arrays

```
// Accessing the array:
 for (i = 0; i < iRows; i++) {
   for (j = 0; j < iCols; j++) {
     iArray[i][j] = i * j;  // Setting a value
     k = iArray[i][j];      // Getting a value;
     printf ("%2d ", k);
   }
   printf ("\n");
 }
```

Multi-dimensional dynamic arrays

```
// Free'ing the memory used by the array:
 for (i = 0; i < iRows; i++) {
   // For each row, free the columns:
   free(iArray[i]);
 }
 // Then free the row pointer array itself:
 free(iArray);
```

Multi-dimensional dynamic arrays - C11

```
void main (void)
{
  int iRows = 5, iCols = 5;
  int iArray[iRows][iCols], i, j, k;

  for (i = 0; i < iRows; i++) {
    for (j = 0; j < iCols; j++) {
      iArray[i][j] = i * j;  // Setting a value
      k = iArray[i][j];      // Getting a value;
      printf ("%2d ", k);
    }
    printf ("\n");
  }
}
```

Function arguments

Functions in C always take arguments by value!

Hence pass the pointer

Example:

```
SomeFunc(int *arr1, int *arr2, int size);
// SomeFunc(int arr1[], int arr[], int size);

// in main
int a[10], b[23];
SomeFunc(a, b, 100);
```

Details

```
int array[10] ;
  array = something // illegal to do this


int *array;
  array = something // legal but memory leak
                    // if you don’t clear up.
```

Beware of scope

- Do not create dangling pointers!

Strings

Not a native datatype

Array of chars

Always ends with ‘\0’

- Or so the <string.h> assumes

char functions

isalnum(int c)

isalpha(int c)

islower(int c)

isupper(int c)

isdigit(int c)

isxdigit(int c)

isodigit(int c)

isprint(int c)

isgraph(int c)

ispunct(int c)

isspace(int c)

---

tolower(int c)

toupper(int c)

string declaration

```
char *str;
```

```
char *strConst = “hello world”;
```

- If you try to modify it, seg fault is thrown.

```
char strWritable[] = “You can write here”;
```

Always allocate 1 byte extra – why?

Operations

```
size_t strlen(const char *string);
```

- Gives the length of the string. How? Lets write it!

```
char *strcpy(char *dest, const char *src);

char *strncpy(char *dest, const char *src,
              size_t n);

char *strcat(char *dest, const char *src);

char *strncat(char *dest, const char *src,

              size_t n);
```

For all, memory is your problem!

Comparison

```
int strcmp(const char *s1, const char *s2);
```

- Lexicographic comparison
- <0 if s1<s2
- =0 if s1==s2
- >0 if s1>s2

```
int strncmp(const char *s1, const char *s2,
            size_t n);
```

Search

```
char *strchr(const char *str, int c);
```

- Look for the first occurrence in string of a character

```
char *strrchr(const char *str, int c);
```

- Look for the last occurrence in string of a character

```
char *strstr(const char *str,
             const char *substr);
```

- Look for the first occurrence in string of a substring

Span search

size_t strspn(const char *str, const char * set);

- find the longest span of characters from set from the beginning of str

size_t strcspn(const char *str, const char *set);

- find the longest span of characters not from set from the beginning of str

char * strpbrk(const char *str, const char * set);

- find the first occurrence of any of the set in str

char *strtok(char *str, const char * delimiters);

- To tokenize based on delimiters.

And more …

Structures

A heterogeneous collection of data members

Much like classes but :

- All its members are *public*
- Just data members – no methods

Unions are also possible but different from structs.

struct

Structures must be declared before using them.

```
struct EMPLOYEE {
    int iId;
    char szName[20];
    int iSalary;
};
```

Then the declaration of a variable of the employee type.

```
    struct EMPLOYEE e1,e2,e3;
```

Notice the struct keyword again!

struct

You can use typedef

```
typedef struct _EMPLOYEE {
   int iId;
   char szName[20];
   int iSalary;
} EMPLOYEE;
```

Now you can use

```
EMPLOYEE e1,e2,e3;
struct _EMPLOYEE e4, *pe;
```

struct

Or, omit the name in the typedef

```
typedef struct {
        int iId;
        char szName[20];
        int iSalary;
} EMPLOYEE;
```

Now you can use

```
EMPLOYEE e1,e2,e3, *pe;
```

struct

Accessing elements using ‘.’

```
e1.iId = 10;
strcpy(e1.szName ,“penguin”);
e1.iSalary = 42;
```

Size of structs

size of struct is a bit weird!

Example :

```
typedef struct {
        int iId;
        char szName[20];
        int iSalary;
} EMPLOYEE;

EMPLOYEE e1, e2, e3;
```

Demonstration!

Padding…

Hence take care of what and how you use in struct!

Force non-padding - #pragma pack(1) .. #pragma pack()

Assignment & comparison

‘=‘ will do a bitwise copy:

- shallow copy
- take care when you have pointers!
- Easy to create memory leaks.

Comparison is not trivial

‘==‘ is pretty much illegal on the structs

You have to just compare the members individually

Pointers to struct

Example declaration:

```
typedef struct {
        int iId;
        char szName[20];
        int iSalary;
} EMPLOYEE;
EMPLOYEE *pe1,*pe2;
```

Now, they are just pointers like any other pointers

It is a programmers problem to assign memory and properly initialize them.

How do we dereference them?

Memory allocation

```
EMPLOYEE *pStruct;
pStruct = malloc(N*sizeof(EMPLOYEE));
```

This will allocate N structs

Accessing them:

```
pStruct[i].id       // will work
(pStruct+i)->id     // will also work
pStruct->id         // will address the first one.
pStruct++;
pStruct->id         // will address the next one.
```

Initialization

Example:

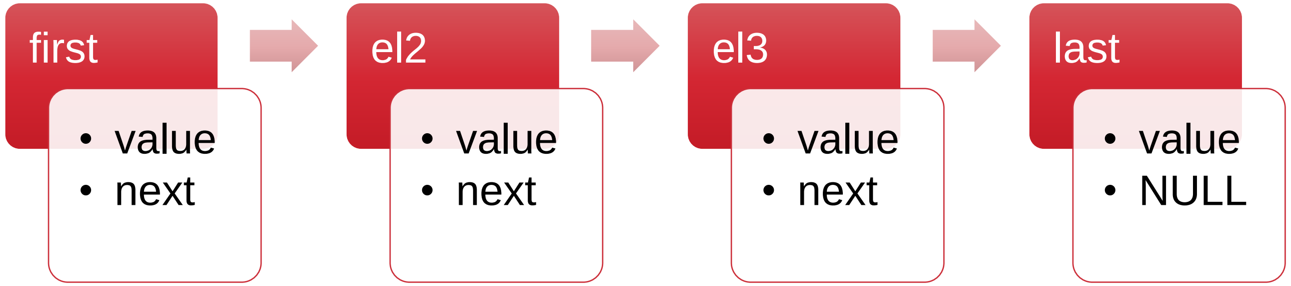
```c
typedef struct{
   double x;
   double y;
} coord;

typedef struct {
   char name[20];
   coord vertices[4];
} rect;

rect r1 = { "first",
   {
            {0,1},
            {0,0},
            {1,0},
            {1,1}
   }
};

rect r2 = { "second", 0,1,0,0,1,0,1,1};
```

Initialization should be proper.

They can be less in number   - rest are 0

Lists

Lists are useful data structures



first
- value
- next

el2
- value
- next

el3
- value
- next

last
- value
- NULL

Lists

A simple idea is to have nodes.

Node :

```
typedef struct NODE {
  struct NODE *next;
  double value;
} NODE;
```

Creation of node-

- allocate the required memory
- initialize with required values

Adding a node –

- at the end
- at the beginning
- somewhere else

Deleting a node –

- by value
- at the end/at the beginning

Flexible array member

```
struct doubleVector {
    unsigned length;
    double array[]; // The flexible array member should be last
};
```

sizeof() returns 0 for that element. Cannot be used in as array members or member of another structure.

Excersies

```
wget http://www.eastwillsecurity.com/pg3401/leksjon5.zip
```