Topics

- Linux programming environment (2h)
- Introduction to C programming (12h)
  1. Getting started with C Programming
  2. Variables, Data-types, Operators and Control Flow
  3. Functions and Libraries
  4. Arrays and Pointers
  5. User defined datatype and data structure
  6. Input and Output
- Basic system programming in Linux (10h)
Overview

1. User defined datatype
   - Structure
   - Structure and Function
   - Structure and Array
   - Pointers to Structures
   - Nested Structures
   - Typedef and enum
   - Unions
   - Bit-fields

2. Data structure
   - Memory allocation
   - Concatenated List
Definition of a **struct**

- **Array**: $N$ items all having the **same type**.
- **struct**: a collection of possibly **different types** grouped together under a single name.
  - made out of primitive data type variables
  - can contain arrays
  - can contain other structs

**General syntax for a struct declaration:**

```c
struct <StructName> {
    <type name1>;
    <type name2>;
    ...
};
```
struct cat {
    int age;     // cat's age
    double weight; // weight (kg)
    double food[7]; // food during last week (kg)
}; /* notice the ' ; ' at the end */
Features and Usage

- **`struct`** defines a new **datatype**
- The variables declared within a structure are called its **members**
- An optional **TagLabel** may be assigned:
  ```c
  struct {...} TagLabel;
  ```
- Variables can be declared like any other built in data-type
  ```c
  struct <StructName> <MyfavoriteName>;
  ```
- To (read/write) access a member of **struct**, use the following **syntax**
  ```c
  <MyfavoriteName>..<MemberName>
  ```
Parametric Init

Example

```c
#include <stdio.h>

struct cat {
    int age;    // cat's age
    double weight;   // weight (kg)
    double food[7]; // food during last week (kg)
};

int main() {
    struct cat felix;  // Declaration
    struct cat luna;

    felix.age = 14;    // Assignment
    felix.weight = 4.5;
    luna.age = felix.age - 5;
    luna.food[0] = 0.5;
    luna.weight = 5.3;

    if (luna.weight > 5)
        printf("Luna eats too much!\n");

    return 0;
}
```
To initialize a `struct`, values can be assigned to its members in the same order they appear in the declaration of the `struct`:

```c
struct cat {
    int age;       // cat’s age
    double weight; // weight (kg)
    double food[7]; // food during last week (kg)
};

int main(){
    struct cat felix = {14, 4.5, {0.2, 0.1, 0.3, 0.3, 0.2, 0.5, 0.3}};
    /* do something */
    return 0;
}
```
Passing structs to functions

Here’s how you can pass structs as argument to a function

```c
struct cat {
    int age;       // cat’s age
    double weight; // weight (kg)
    double food[7]; // food during last week (kg)
};
void display(struct cat mycat);

int main() {
    struct cat felix;  /* Declaration */
    felix.age = 14;    /* Assignment */
    felix.weight = 4.5;
    display(felix);    /* call function */
    return 0;
}

void display(struct cat mycat) {   /* STRUCT
    as argument */
    printf("Displaying info
");
    printf("Age: %d
", mycat.age);
    printf("Weight: %.2f
", mycat.weight);
}
```

Output

Displaying info
Age: 14
Weight: 4.50
Returning structs from a function

Here’s how you can return a structure from a function

```c
struct cat {
    int age;       // cat's age
    double weight; // weight (kg)
};

struct cat setInformation();

int main(){
    struct cat felix; /* Declaration */
    felix = setInformation();
    printf("Displaying info\n");
    printf("Age: %d\n", felix.age);
    printf("Weight: %.2lf\n", felix.weight);
    return 0;
}

struct cat setInformation() { /* returning a STRUCT */
    struct cat mycat;
    printf("Age: "); scanf("%d", &mycat.age);
    printf("Weight: "); scanf("%lf", &mycat.weight);
    return mycat;
}
```

Output

<table>
<thead>
<tr>
<th>Age</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>4.5</td>
</tr>
</tbody>
</table>

Displaying info

Age: 14
Weight: 4.50
Passing a struct as a parameter

Let us assume to have a `struct cat` named `felix`:

```c
struct cat felix;
```

Passing a `struct` as a parameter works like passing a variable:

- If you want to pass a `copy` of variable in `felix` to the frame of the function, pass the value of `struct`:
  ```c
  check_weight(felix);
  ```

- If you want to pass a `reference` to the variables in `felix` (changes visible also when function returns), pass the memory’s address of `struct`:
  ```c
  update_weight(&felix, new_weight);
  ```
Access to \textbf{struct}

Summary:

- To access variables in a \textbf{struct} \texttt{s}, when we have the \textit{value} of the struct
  \begin{verbatim}
  struct name_struct s
  s.name_var
  \end{verbatim}

- To access variables in a \textbf{struct} \texttt{s}, when we have the \textit{address} of the struct
  \begin{verbatim}
  struct name_struct* s
  s->name_var
  \end{verbatim}
Passing structs by reference

Example

```c
struct cat {
    int age;          // cat's age
    double weight;   // weight (kg)
};

void update_weight(struct cat *felix, double new_weight);

int main(){
    struct cat felix; /* Declaration */
    felix.weight = 4.5; /* Assignment */

    printf("Current Weight: %.2lf \n", felix.weight);
    update_weight(&felix, 5.2); // pass a reference
    printf("Updated Weight: %.2lf ", felix.weight);
    return 0;
}

void update_weight(struct cat *felix, double new_weight) {
    felix->weight = new_weight; /* modifies the struct */
}
```

Output

```
Current  
Weight:  4.50
Updated
Weight:  5.20
```
A `struct` can contain an array, accessible like a primitive data.
struct and arrays

Example

```c
struct cat {
    int age; // cat's age
    double weight; // weight (kg)
    double food[7]; // food during the last week (kg)
};

int main() {
    struct cat felix;
    int i;
    double sum_food = 0;

    felix.age = 14;
    felix.food[0] = 0.50;
    felix.food[1] = 0.22;
    felix.food[2] = 0.56;

    for (i = 0; i < 3; i++)
        sum_food += felix.food[i];
    printf("In three days, Felix ate %.2lf kg.", sum_food);
    return 0;
}
```

Output

In three days, Felix ate 1.28 kg.
Consider writing a program that counts the number of each cat.

We need an array of char strings to hold the names, and array of integer for the counts.

One possibility is to use two parallel arrays

```c
char *name[NumCats];
int count[NumCats];
```

But the very fact that the arrays are parallel suggests a different organization, an array of structures. Each keyword is a pair:

```c
char *name;
int count;
```
Arrays of Structures (2)

- The structure declaration

```c
struct cat {
    char *name;
    int count;
} cats[NumCats];
```

- declares a structure type `cat`, and there is an array of pairs.
Arrays of Structures (3)

- This could also be written as

```c
struct cat {
    char *name;
    int count;
};

struct cat cats[NumCats];
```
### Arrays of Structures

#### Example

```c
#include <stdio.h>
#define NumCats 2
struct cat {
    char *name;
    int count;
} cats[NumCats];

int main() {
    cats[0].name = "Felix"; // declare & init
    cats[0].count = 31;
    cats[1].name = "Luna";
    cats[1].count = 12;

    for (int i=0; i<NumCats; i++)
        printf("%d %s \n", cats[i].count, cats[i].name);

    return 0;
}
```

#### Output

```
31 Felix
12 Luna
```
Can we allocate memory dynamically to `struct` types?

We have already learned that a pointer is a variable which points to the address of another variable of any data type.

Similarly, we can have a **pointer to structures**, where a pointer variable can point to the address of a structure variable.

To get started, let us **use pointers to (array of) struct** having fixed size.
Let us revisit above cat counting program, this time using pointers instead of array indices.

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

struct cat {
    char *name;
    int count;
};

int main() {
    struct cat mycat, *catPtr; // a struct and a pointer
    catPtr = &mycat; // Ptr points to struct

    catPtr->name = "Felix"; // assign values
    catPtr->count = 31;
    printf("%d %s \n", catPtr->count, catPtr->name);
    return 0;
}
```
Pointers to Structure

Note

- `catPtr->name` is equivalent to `(*catPtr).name`
- `catPtr->count` is equivalent to `(*catPtr).count`
Pointers to Structure

For a fixed number of cats,

```c
#include <stdio.h>
#include <string.h>
#define NumCats 2
struct cat {
    char *name;
    int count;
};
int main() {
    struct cat mycat[NumCats], *catPtr; // array of struct + Ptr
    catPtr = mycat; // Ptr points to first element

    catPtr[0].name = "Felix";
    catPtr[0].count = 31;
    printf("catPtr[0] %d %s \n", catPtr[0].count, catPtr[0].name);
    return 0;
}
```
Nested **struct** (1)

- A **struct** can contain another **struct**, but there are rules.
- The size of the **struct** has to be finite, constant, and calculable at compilation time, therefore they cannot be recursive!
- A struct $S_2$ can contain a **struct** $S_1$ only if $S_1$ is defined before in the program.
- In this way, the nesting of **struct** forms a **tree**.
The `struct nation` has a variable of type `struct city`. The `struct city` cannot have a variable `struct nation`, otherwise it would not be possible to determine the space occupied by the `struct`. 
Nested struct (3)

```c
struct city {
    int citizens;
    struct nation n;
};
struct nation {
    struct city capital;
    double surface;
    int time_zone;
};
```

sizeof(struct city) = 4 + sizeof(struct nation)
sizeof(struct nation) = 4 + 8 + sizeof(struct city)

Note

This `struct` would form a loop rather than a tree. Hence, the compiler gives an error.
Self-referential `struct`

However, there are cases when structs have to logically recursively call themselves (recursive nesting). **Example:** we want to represent a family tree:

```c
struct person {
    struct person mother;
    struct person father;
    int birth_year;
};
```

Memory size would tend to infinity. What to do? Pointers to `struct` have always the same size (e.g., 8 bytes in 64 bit). Solution:

```c
struct person {
    struct person* mother;  // pointer to struct
    struct person* father;
    int birth_year;
};
```
Example

```c
int main() {

    // init
    struct person me;
    me.mother = NULL;
    me.father = NULL;
    me.birth_year = 2000;

    struct person dad;
    dad.mother = NULL;
    dad.father = NULL;
    dad.birth_year = 1971;

    struct person mum;
    mum.mother = NULL;
    mum.father = NULL;
    mum.birth_year = 1976;

    me.father = &dad;    // tree linking
    me.mother = &mum;

    // Print data
    printf("me: %d\n", me.birth_year);
    printf("dad: %d\n", me.father->birth_year);
    printf("mum: %d\n", me.mother->birth_year);
    return 0;
}
```

Terminal

```
terminal
me: 2000
dad: 1971
mum: 1976
```
Size of a struct

- The size of a struct is greater than or equal to the sum of the sizes of its members.

- Byte alignment
  ```c
  struct cat {
    int age;
    /* padding for alignment requirements */
    double weight;
    double food[7];
  };
  ```

- Size of a struct is the sum of the sizes of its primitive data **PLUS possibly padding** to align with the "natural" address boundaries.
Padding

- Why doing that? Unaligned memory access is slower on architectures that allow it (like x86 and amd64), and is explicitly prohibited on strict alignment architectures like SPARC.
- The structure **padding is automatically done by the compiler** to make sure all its members are byte aligned.
- A `char` of 1 byte can be allocated anywhere in memory.
- An `int` of 4 bytes must start at a 4-byte boundary.
- In our case,
  
  $$\text{sizeof(struct cat)} == \text{sizeof(int)} [4 \text{ Bytes}] + \text{PADDING} [4 \text{ Bytes}] + \text{sizeof(double)} [8 \text{ Bytes}] + 7*\text{sizeof(double)} [7*8 \text{ Bytes}] = 72 \text{ Bytes}.$$
Packing

- Packing, on the other hand prevents compiler from doing padding.
- Alignment can be explicitly requested by the compiler extension `__attribute__((__packed__))`

```c
struct __attribute__((__packed__)) cat {
    int age;    // 4 Bytes
    double weight;    // 8 Bytes
    double food[7];    // 7*8 = 56 Bytes
};
```

- Now, the `struct cat` occupies only 68 Bytes of memory.
The keyword `typedef` can be used to define an *alias* for types i.e., an alternative names for an existing data type.

**Example:**

```
typedef int age_t;
```

defines a new type `age_t`, equivalent to type `int`. Now, we can:

```
age_t age_mario = 67;
```

Equivalent to

```
int age_mario = 67;
```
**typedef and struct**

Use `typedef` with structure to define a new data type and then use that data type to define structure variables directly (analogous for `enum` as we will see shortly).

**Example:**
```c
struct student_struct {
    int StudentID;
    int birth_year;
};
typedef struct student_struct student;

int main() {
    student giorgio; /* we save the long declaration */
    giorgio.StudentID = 666;
}
```
It is also possible to define a struct directly with a `typedef`.

```c
typedef struct {
    int StudentID;
    int birth_year;
} student;
```

This is the best way.
Enumeration is a **user defined datatype** in C language.

It is used to **assign names to enumeration constants** (type `int`) making a program easy to read and maintain.

Keyword `enum` is used to declare an enumeration.

**Syntax**

```c
enum <EnumName> {const1, const2, ....... };
```

The value of the first enumerator is zero by default.
**enum**

**Example**

**Definition:**

```c
enum color_cat {
    red,
    black,
    white
};
```

**Function** `main`:

```c
enum color_cat color;
color = red;
switch (color) {
    case red:
        /* do sth. */
        break;
    case black:
        /* do sth. else */
        break;
    case white:
        /* do sth. else */
        break;
}
```
Like in `struct`, it is possible to define `enum` and its `typedef` in one statement as follows:

**Definition:**

```c
typedef enum {
    red,
    black,
    white
} color_cat;
```

**Function `main`:**

```c
color_cat color;
color = red;
/* print enum values */
printf("color = %d \t%d \t%d \n", red, black, white);
```

**Terminal**

```
color = 0 1 2
```
To assign other enumeration constants,

**Definition:**

```c
typedef enum {
    red=10,
    black,
    white=4
} color_cat;
```

**Function main:**

```c
color_cat color;  
printf("color = %d\t%d\t%d\t\n",red,black,white);
```

**Terminal**

```
color = 10 11 4
```
enum within struct

To access the enum types without a scope, declare them so:

```c
typedef enum {
    red,
    black,
    white
} color_cat;

typedef struct {
    color_cat color; /* enum call */
    int age;
    double weight;
} cat;

void main(){
    cat felix;
    felix.color = red;
    if (felix.color != black)
        printf("We wanted a black cat!");
}
```

Output

We wanted a black cat!
A union is **one variable** that may hold objects of **different types** (and hence, sizes) in the **same memory location**.

- Any built-in or user defined data types can be used inside a union
- Useful when you want to store a data that could have several data types.

```c
union <Tag> {
    <type1> <variable1>;
    <type2> <variable2>;
    ...
    <typeN> <variableN>;
} <UnionTag>;
```

- The UnionTag is optional.
union (cont’d)

The size of the union variable is equal to the **size of its largest element plus padding**.

```c
union foo {
    int i; // 4 Bytes, 4-Byte alignment
    float f; // 8 Bytes, 4-Byte alignment
    char str[10]; // 10*1 Byte, no alignment
};
int main() {
    union foo data;
    printf( "Memory size: %zu\n", sizeof(data));
    return 0;
}
```

**Output**

Memory size: 12
union

Example

```c
int main() {
    union foo data;
    data.i = 10;
    data.f = 3.141529;
    strcpy(data.str, "Teams");
    printf("data.i : %d\n", data.i);
    printf("data.f : %.3f\n", data.f);
    printf("data.str : %s\n", data.str);
    return 0;
}
```

What result do you expect?
union

Example (cont’d)

Output

data.i : 1835099476
data.f : 43597890872730174000000000000.00000
data.str : Teams

Note

the int and float values of the union became corrupted because the final str value assigned to the variable has occupied the memory location.
Bit-fields

- A **bit-field** is a set of adjacent bits within a single `WORD`.
- One common use is a **set of single-bit flags** in applications like compiler symbol tables.
- Syntax

  ```c
  struct {
      type <StructName> : <Width> ;
  };
  ```

- The `<type>` determines how a bit-field’s value is interpreted. The type may be `int`, `signed int`, or `unsigned int`.
- The `<Width>` determines the number of `bits` in the bit-field. Its max. width is given by `type`. 
Size of Bit-fields

- The **memory size** of a bit-field is determined by the sum of the bits for each variable **plus padding** to be aligned with a 4-Byte boundary of **int**.
- The idea is to use memory efficiently when we know that the value of a field or group of fields will never exceed a limit.

```c
struct flag {
    unsigned int error_code: 8;  // 8 bits
    unsigned int sector: 5;      // 5 bits
    unsigned int command: 5;     // 5 bits
};
```

- The memory size is 4 Bytes (= 32 bits) according to **unsigned int**.
Bit-fields

Some bit field members are stored left to right others are stored right to left in memory.

If bit fields too large, next bit field may be stored consecutively in memory

If portability of code is a premium you can use bit shifting and masking to achieve the same results

Array of bit-fields is not allowed.
So far, we have assigned a fixed amount of memory to the array of `struct` e.g., `NumCats=2` elements.

What to do when the exact number of cats is unknown or the number becomes larger than two???

Alternative approach: Allocate memory dynamically to Pointers of struct
Dynamical memory allocation

Syntax

**New memory allocation**
\[
<\text{MyfavoriteName}> = \text{malloc}(<\text{Size}> * \text{sizeof}(\text{struct}<\text{StructName}>));
\]

**New memory allocation + init to NULL**
\[
<\text{MyfavoriteName}> = \text{calloc}(<\text{Size}>,\text{sizeof}(\text{struct}<\text{StructName}>));
\]

**Memory reallocation** (Elements are not deleted)
\[
<\text{MyfavoriteName}> = \text{realloc}(<\text{MyfavoriteName}>,<\text{NewSize}>*\text{sizeof}(\text{struct}<\text{StructName}>));
\]

**Remedy against memory leaks**
\[
\text{free}(<\text{MyfavoriteName}>)
\]
Dynamical memory allocation

Example

```c
struct student {
    int birth_year;
    int StudID;
};

int main() {
    struct student* giovanni; /* one struct */
giovanni = malloc(sizeof(struct student));
giovanni->birth_year = 1960;

    struct student* student_inf; /* array of structs */
student_inf = calloc(10, sizeof(struct student));
student_inf[0].StudID = 431950;
student_inf[0].birth_year = 1991;
/* your favorite program */

    free(giovanni);
    free(student_inf);
```
Let us see now a powerful data structure, the **Concatenated List**.

A list is made out of nodes:

1. **Data**;
2. **Reference to the next node**.

Rule: last node has, as reference to next, **NULL**.
Concatenated lists - properties

The list, like an array, keeps the data sorted, but:

3. The deletion of an element does not require the rebuilding of the whole array.
Definition of `node` incorporated as `head`:

```c
struct node
{
    int data;
    struct node *next;
};

struct node* head; // starting node of list
```
void append(int num)  // add to end of list
{
    struct node *temp,*last;
    // create a new temp node
    temp=(struct node *)malloc(sizeof(struct node));  // create temp node
    temp->data=num;  // set temp value
    temp->next=NULL;  // set as end of list

    last = head;
    if (last==NULL)  // no head node exists yet
        head=temp;
    else  // find end of current list and add node
    {
        while (last->next != NULL)
            last=last->next;  // goto next node

        last->next = temp;  // set temp as new last node
    }
}
Which operations are necessary to delete the next of node $p$?

- Get the address $s$ of next of $p$.
- Get the address $s'$ of next of $s$.
- Assign $s'$ to the value of the next of $p$.
- Free memory pointed by $s$. 
Concatenated list - delete a node

```c
int delete(int num)
{
    struct node *temp, *next;
    temp=head; // start at first node

    if (num == 0) { //head to be deleted
        head=temp->next;
        return 0;
    }

    for (int i = 0; temp!=NULL && i<num-1; i++) //loop until previous node
        temp = temp->next;

    /* Node temp->next is the node to be deleted,
    Store pointer to the next of node to be deleted */
    next = temp->next->next;

    free(temp->next); //delete current node

    temp->next = next; //unlink the deleted node
}
```
Consider the following declaration

```c
struct addr {
    char village[10];
    char street[30];
    int ID ;
};

struct {
    char name[30];
    int gender;
    struct addr locate;
} student , *sptr = &student ;
```

Then `*(sptr -> name +2)` can be used instead of

1. `student.name +2`
2. `sptr -> (name +2 )`
3. `*((*sptr).name + 2 )`