EEE243 – Applied Computer Programming

Enumerations, Type Definitions and Structures

ROYAL MILITARY COLLEGE OF CANADA
ELECTRICAL & COMPUTER ENGINEERING

GÉNIE ÉLECTRIQUE
ET GÉNIE INFORMATIQUE
COLLÈGE MILITAIRE ROYAL DU CANADA

Outline

1. Enumerated Types
2. Type Definition
3. Structures

Enumerated Types

Sometimes you need many related defined constants

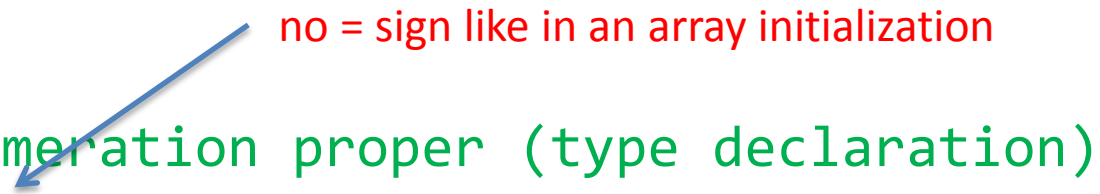
```
#define POWER_SUPPLY_STATUS_UNKNOWN 0
#define POWER_SUPPLY_STATUS_CHARGING 1
#define POWER_SUPPLY_STATUS_DISCHARGING 2
#define POWER_SUPPLY_STATUS_NOT_CHARGING 3
#define POWER_SUPPLY_STATUS_FULL 4
```

Enumerated Types

- The enumerate type let you invent your own data type and specify what values it can take
- Helps to make listings more readable
- The enumerated type, `enum` builds on top of the `int` type
- In an enumerated type, each integer value is associated with an identifier called an ***enumeration constant***

Enumerated Types

There are two basic ways of declaring an enumerated type, but we only teach the preferred method:




```
//The enumeration proper (type declaration)
enum Tag {enumeration constants};

//Declaring a variable for the enum
enum Tag variable_identifier;
enum {enumeration constants} variable_identifier;
```

Enumerated Types - Example

```
// type declaration
enum Months {JAN, FEB, MAR, APR,
             MAY, JUN, JUL, AUG,
             SEP, OCT, NOV, DEC};

// variable declaration
enum Months birth_month;

// declaration with initialization
enum Months graduation_month = MAY;
```

Enumerated Types

- Note that because in C data structures are *zero-based*, the value of the first element in the enumeration is `0` so `JAN == 0`
- But we can specify values explicitly:

```
enum Months {JAN = 1, FEB = 2,…};
```

- If I only specify the first value then the compiler fills in the next values by adding one to each new item.

```
enum Months {JAN = 1, FEB, MAR, APR,…};
```

TRY OUT THIS: enum MONTHS {JAN=1, FEB, MAR= -10, APR, MAY, JUN=1};

Enumerated Types

Two words of caution about enumerated types:

1. C allows two enumeration constants to have the same value. This is a really dumb idea.

```
enum Sizes {SMALL = 1,
            BIG = 2,
            VERY_BIG = 2}; // DON'T!
```

2. There is no range checking on the values assigned to variables:

```
enum Sizes my_size= 4; //just a compiler
                       //warning
```

Advantages of Enumerations

- are enumerations used in real programs?
  - **heck yes**
- for example, from the Linux kernel (`include/linux/power_supply.h`)

```
enum power_supply_type {
        POWER_SUPPLY_TYPE_UNKNOWN = 0,
        POWER_SUPPLY_TYPE_BATTERY,
        POWER_SUPPLY_TYPE_UPS,
        POWER_SUPPLY_TYPE_MAINS,
        POWER_SUPPLY_TYPE_USB,
        POWER_SUPPLY_TYPE_USB_DCP,
        POWER_SUPPLY_TYPE_USB_CDP,
        POWER_SUPPLY_TYPE_USB_ACA,
        POWER_SUPPLY_TYPE_USB_TYPE_C,
        POWER_SUPPLY_TYPE_USB_PD,
        POWER_SUPPLY_TYPE_USB_PD_DRP,
};
```

Advantages of Enumerations

```
enum power_supply_type {
        POWER_SUPPLY_TYPE_UNKNOWN = 0,
        POWER_SUPPLY_TYPE_BATTERY,
        POWER_SUPPLY_TYPE_UPS,
        POWER_SUPPLY_TYPE_MAINS,
        POWER_SUPPLY_TYPE_USB,
        POWER_SUPPLY_TYPE_USB_DCP,
        POWER_SUPPLY_TYPE_USB_CDP,
        POWER_SUPPLY_TYPE_USB_ACA,
        POWER_SUPPLY_TYPE_USB_TYPE_C,
        POWER_SUPPLY_TYPE_USB_PD,
        POWER_SUPPLY_TYPE_USB_PD_DRP,
};
```

is roughly equivalent to

```
#define POWER_SUPPLY_TYPE_UNKNOWN 0
#define POWER_SUPPLY_TYPE_BATTERY 1
#define POWER_SUPPLY_TYPE_UPS 2

  …
```

but imagine this…

```
enum power_supply_property {
/* Properties of type `int' */
POWER_SUPPLY_PROP_STATUS =POWER_SUPPLY_PROP_CHARGE_TYPE,
POWER_SUPPLY_PROP_HEALTH,
POWER_SUPPLY_PROP_PRESENT,
POWER_SUPPLY_PROP_ONLINE,
POWER_SUPPLY_PROP_AUTHENTIC,
POWER_SUPPLY_PROP_TECHNOLOGY,
POWER_SUPPLY_PROP_CYCLE_COUNT,
POWER_SUPPLY_PROP_VOLTAGE_MAX,
POWER_SUPPLY_PROP_VOLTAGE_MIN,
POWER_SUPPLY_PROP_VOLTAGE_MAX_DESIGN,
POWER_SUPPLY_PROP_VOLTAGE_MIN_DESIGN,
POWER_SUPPLY_PROP_VOLTAGE_NOW,
POWER_SUPPLY_PROP_VOLTAGE_AVG,
POWER_SUPPLY_PROP_VOLTAGE_OCV,
POWER_SUPPLY_PROP_CURRENT_MAX,
POWER_SUPPLY_PROP_CURRENT_NOW,
POWER_SUPPLY_PROP_CURRENT_AVG,
POWER_SUPPLY_PROP_POWER_NOW,
POWER_SUPPLY_PROP_POWER_AVG,
POWER_SUPPLY_PROP_CHARGE_FULL_DESIGN,
POWER_SUPPLY_PROP_CHARGE_EMPTY_DESIGN,
POWER_SUPPLY_PROP_CHARGE_FULL,
POWER_SUPPLY_PROP_CHARGE_EMPTY,
POWER_SUPPLY_PROP_CHARGE_NOW,
POWER_SUPPLY_PROP_CHARGE_AVG,
POWER_SUPPLY_PROP_CHARGE_COUNTER,
POWER_SUPPLY_PROP_CONSTANT_CHARGE_CURRENT,
POWER_SUPPLY_PROP_CONSTANT_CHARGE_CURRENT_MAX,
POWER_SUPPLY_PROP_CONSTANT_CHARGE_VOLTAGE,
POWER_SUPPLY_PROP_CONSTANT_CHARGE_VOLTAGE_MAX,
POWER_SUPPLY_PROP_CHARGE_CONTROL_LIMIT,
POWER_SUPPLY_PROP_CHARGE_CONTROL_LIMIT_MAX,
POWER_SUPPLY_PROP_INPUT_CURRENT_LIMIT,
POWER_SUPPLY_PROP_ENERGY_FULL_DESIGN,
POWER_SUPPLY_PROP_ENERGY_EMPTY_DESIGN,
POWER_SUPPLY_PROP_ENERGY_FULL,
POWER_SUPPLY_PROP_ENERGY_EMPTY,
POWER_SUPPLY_PROP_ENERGY_NOW,
POWER_SUPPLY_PROP_ENERGY_AVG,
POWER_SUPPLY_PROP_CAPACITY, /* in percents! */
POWER_SUPPLY_PROP_CAPACITY_ALERT_MIN, /* in percents! */
POWER_SUPPLY_PROP_CAPACITY_ALERT_MAX, /* in percents! */
POWER_SUPPLY_PROP_CAPACITY_LEVEL,
POWER_SUPPLY_PROP_TEMP,
POWER_SUPPLY_PROP_TEMP_MAX,
POWER_SUPPLY_PROP_TEMP_MIN,
POWER_SUPPLY_PROP_TEMP_ALERT_MIN,
POWER_SUPPLY_PROP_TEMP_ALERT_MAX,
POWER_SUPPLY_PROP_TEMP_AMBIENT,
POWER_SUPPLY_PROP_TEMP_AMBIENT_ALERT_MIN,
POWER_SUPPLY_PROP_TEMP_AMBIENT_ALERT_MAX,
POWER_SUPPLY_PROP_TIME_TO_EMPTY_NOW,
POWER_SUPPLY_PROP_TIME_TO_EMPTY_AVG,
POWER_SUPPLY_PROP_TIME_TO_FULL_NOW,
POWER_SUPPLY_PROP_TIME_TO_FULL_AVG,
POWER_SUPPLY_PROP_TYPE, /* use power_supply.type instead */
POWER_SUPPLY_PROP_SCOPE,
POWER_SUPPLY_PROP_CHARGE_TERM_CURRENT,
/* Properties of type `const char *' */
POWER_SUPPLY_PROP_MODEL_NAME,
POWER_SUPPLY_PROP_MANUFACTURER,
POWER_SUPPLY_PROP_SERIAL_NUMBER,
};
```

Advantages of Enumerations

1. values assigned automatically by the compiler
2. easy to read
3. easy to maintain
4. available in the debugger (as opposed to defined constants)

Type Definition

In order to declare a variable based on an enumeration, you always have to write `enum EnumName var_name`.

```
enum Months birth_month;
```

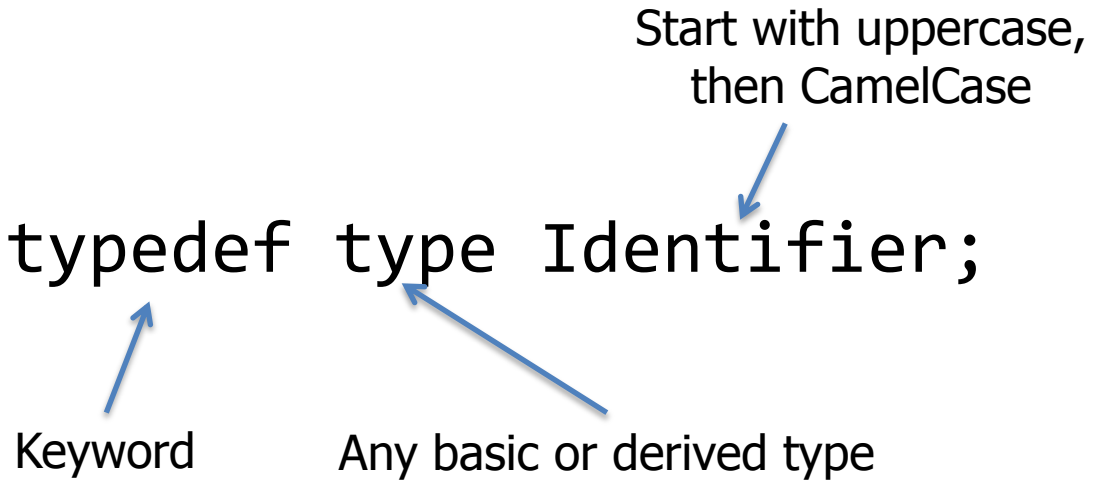
This can be annoying in some cases (not always). There must be a better way!

Type Definition

- Type definitions (`typedef`) are central to creating new data types in C
- With `typedef` you can create a new type from any other type
- We will see later in this lecture how `typedef` helps us to define complex data types
- It (typedef) redefines the existing variable type

Type Definition

- You define a new type with the `typedef` definition format:



Type Definition

- You could define a new type from the type `int` by doing the following:

```
typedef unsigned int Pounds;
```

- Then the following two declarations would be equivalent:

```
unsigned int number=180;
Pounds number=180;
```

- The idea is to create synonyms for the same type to make it more readable.

Type Definition

- Some programmers use `typedef` to define a new type called STRING from the derived type pointer to character (`char*`)

```
typedef char* String;
…
String my_string;//a pointer to char
```

- Equivalent to:

```
char *my_string;
```

Type Definition

- The previous two examples are simple, but they show the built-in flexibility to define new types in the C language
- In the syntax of the language, you can use the name of a defined type wherever you can use the name of a standard type:
  - Declaration of variables
  - Declaration of a function type
  - Declaration of parameters
  - Casting operations,…

Enumerated Types Definition

- Every time you *declare* a new variable of your enumerated type, you must use the `enum` keyword:

```
enum Month birth_month;
```

- We learned earlier today that we can rename a type to something else using `typedef`

Enumerated Types Definition

- Here we simply combine both the `typedef` and the `enum` keywords:

```
typedef          type                Identifier;
typedef enum {RED, BLUE, YELLOW} PrimeColor;
PrimeColor first_color = RED;
PrimeColor second_color = BLUE;
```

- Compare to

```
enum PrimeColor {RED, BLUE, YELLOW};
enum PrimeColor first_color = RED;
enum PrimeColor second_color = BLUE;
```

Structures

- I want to write a program that manages information about students such as :
  - First name
  - Last name
  - College Number
  - Average
- Since arrays can only contain one type, I would have to have multiple arrays.
- It would get messy!

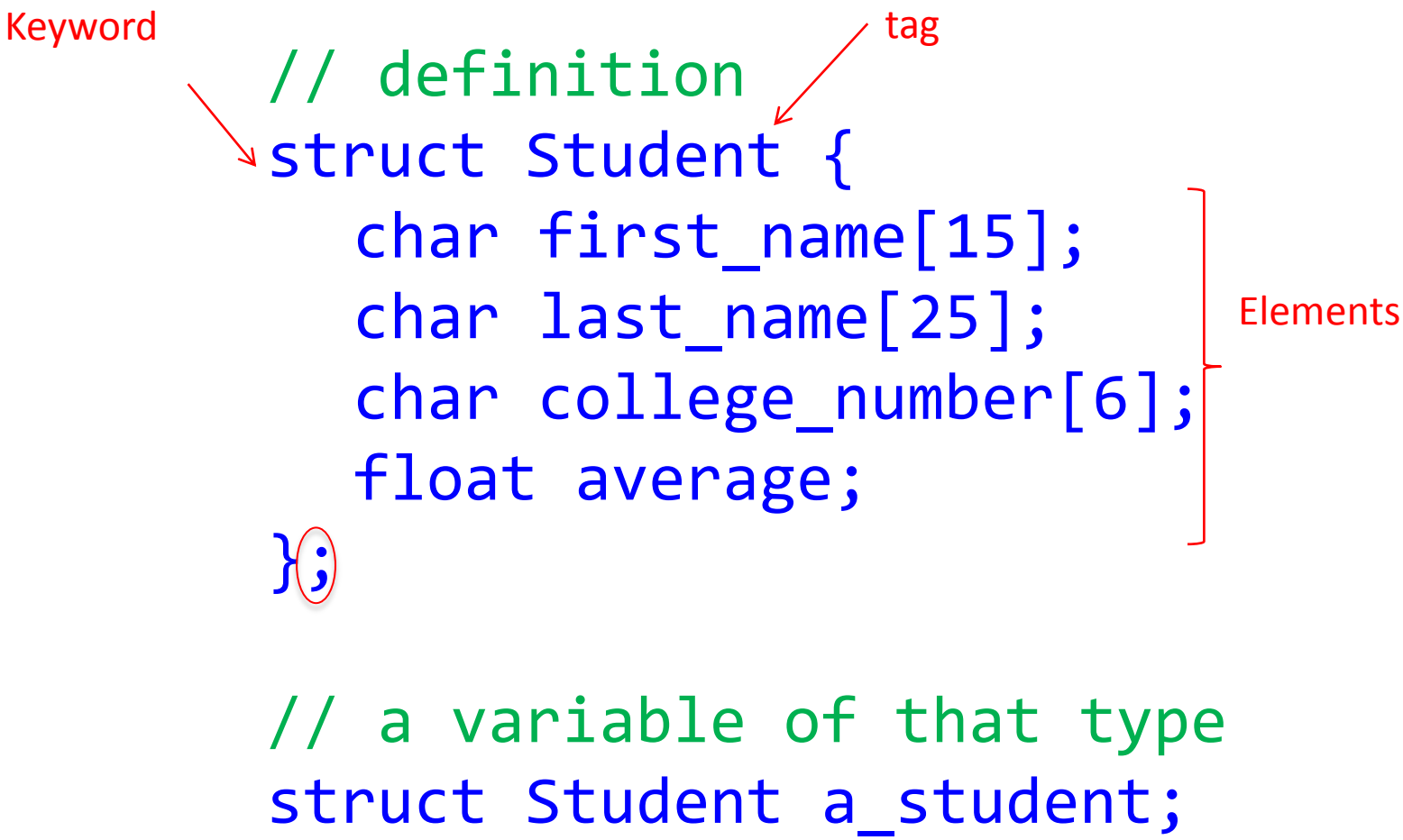
Structures

- If we only used standard types in our programs, they would be very large, and difficult to maintain.
- A structure is a data type whose format is defined by the programmer
- A *structure* is a collection of *related elements*, (called *fields*), possibly of different types having a *single name*

Structures

- Going back to the student example
- We can create a *structure* to store the information together and use an array of structures
- There are three ways to declare a structure in C, but we'll only use two of them:
  - Tagged Structure
  - Type Declaration with `typedef`

Tagged Structures



typedef Structures

- typedef may be used to avoid repeating the struct keyword with each declaration:

```
// definition
typedef struct {
   char first_name[15];
   char last_name[25];
   char college_number[6];
   float average;
} Student;                  // ← note name here

// a variable of that type
Student a_student;
```

Structure - Initialization

- Structures can be initialised similarly to arrays. If we take the `typedef Student` and the variable `a_student` from the previous slide:

```
Student a_student = {"Joe",
  "Shmoe","45239",78.3};
```

As with arrays, this form of initialization works only at declaration, not later.

Example

- Define a `typedef` structure for a car, include
  - the *manufacturer*,
  - *model name*,
  - *transmission type* (manual or automatic),
  - *number of doors* (2 , 3 , 4 or 5),
  - *colour*, (Green, Beige, Grey, Black)
  - *year*,
  - *engine size* (in # cylinders)

- The types from last class can be used in the structure

Example

- Using the type just declared, declare a VehicleType variable for

  Manufacturer Chrysler

  Model Cordoba

  Year 1978

  Doors 2

  Cylinders 8

  Colour black

  Transmission manual

Solution

```
typedef enum {MANUAL, AUTOMATIC}
  Transmission ;

typedef enum { RED, WHITE, YELLOW,
  GREEN, BEIGE, GREY, BLACK} Colour;
```

Solution (.cont)

```
typedef struct {
    char manufacturer[25];
    char model[25];
    int year;
    int num_doors;
    Transmission trans_type;
    Colour car_colour;
    int num_cylinders;
} Car;
```

Solution (cont.)

```
Car slys_first_car = {
     "Chrysler",
     "Cordoba",
     1978,
     2,
     MANUAL,
     BLACK,
     8
};
```

Structures and Fields

- Structures are constructed with fields. Everywhere you can use a variable you can use a structure field
- Each field can be accessed individually with the *structure member operator*
  - (the period) .

A string cannot be copied like other variables using the assignment operator.

```
strcpy(a_student.first_name, "Bob");
a_student.college_number = "98876";
a_student.average = 73.2;
```

Structures Operators, =

- Structures are entities that can be treated as a whole, but ONLY during an assignment operation:

```
Student f_student= {"Jane", "Doe",
  {'2','3','4','9','8'}, 33.2};
a_student = f_student;
```

- You cannot compare two structures:

```
if (a_student == f_student) {
  printf("Same as with array, not equal");
}
```

Structures and fields

- to compare structures, compare each field:

```
// true if all fields are equal
bool compare_students(Student st1, Student st2) {
   return (
     !strcmp (st1.first_name, st2.first_name) &&
     !strcmp (st1.last_name, st2.last_name) &&
     !strcmp (st1.college_number, st2.college_number)
   &&
     (fabs(st1.average - st2.average) < 0.0001)
   );
}
```

Structures and Pointers

- Like any other type in C, pointers can point to structures. The pointer points to the first byte of the structure.

- You can also use pointers to access fields:

```
Student *p_student = &a_student;
a_student.average = 89.5;
(*p_student).average = 92.7;
```

  - Result is the same; we **need** the brackets around the dereferencing due to precedence

Structures and Pointers

- Fortunately C provides another operator that allows us to dereference the pointer and access the field at the same time; the ***structure selection*** operator:

```
pStudent->average= 92.3;
```

same as

```
(*pStudent).average = 92.7;
```

Dynamic memory allocation

- Recall that we can allocate memory dynamically
- You can also allocate memory for any types including `typedef`'d structures:

```
typedef struct {
  char first_name[15];
  char last_name[25];
} Name;

Name *p_name = NULL;
p_name = (Name*)malloc (sizeof(Name));
```

Naming Convention

- **Variables:** lower-case, words separated with underscores, normally nouns. e.g. `student_name`
- **Struct, Enum and Typedef:** capitalized camel case, normally nouns. E.g. `StudentRecord`

Exercise

- What is printed in the following code?

```
char *p_char;
int *p_int;
Student a_student ={"Jane", "Doe", "23498", 33.2};
Student *p_student = &a_student;

p_char = &a_student.college_number;

printf("%s \'s student number is %s",
   (*p_student).first_name, *p_char);

printf("%s \'s student number is %s",
   p_student->first_name, a_student.college_number);
```

Exercise

- If you declare a variable that is a type-defined structure what operator do you use to access its fields?
- If you have a pointer that points to a structure, what operator do you use to access each field?
- Can you assign a complete structure to another?
- Can you compare structures with == ?

Questions?