Imperative programming

8. Struct, union, enum

Zoltán Porkoláb

Enum

- Lots of programming languages has enumerated type

  E = { C1, C2, …, Cn }
- Named constants
- Enum types in languages
  - Pascal, ADA: enumerated type
  - Golang iota
  - C, C++, C#, Java enum
  - C++: enum class

Enum in C

- Named constants of values of the underlying types
- The underlying type is implementation defined (integer-kind) type
- Automatic conversion to and from the implementation type
- Names should be globally unique, values do not need to be unique

```
enum Color { Yellow, Green, Blue, Red, Magenta }; // 0,1,2,3,4
enum Shape { Square, Rectangle, Circle, Count }; // 0,1,2,3
enum En { A, B = 4, C, D = 2, E, F, G }; // 0,4,5,2,3,4,5
enum Shade { Black = 0, Grey = 0x7f, White = 0xff };

void f(void)
{
  enum Shape s = Rectangle;  // 1
  if ( s < Count-1)
    ++s;
  printf("%d\n", s);  // 2
}
```

Enum in C

- Named constants of values of the underlying types
- The underlying type is implementation defined (integer-kind) type
- Automatic conversion to and from the implementation type
- Names should be globally unique, values do not need to be unique

```
enum Shape {
    Square, Rectangle, Circle, Count // 0,1,2,3
};
typedef enum Shape shape_t;

void f(void)
{
  shape_t s = Rectangle;  // 1
  if ( s < Count-1)
    ++s;
  printf("%d\n", s);  /
```

Enum in C

- Named constants of values of the underlying types
- The underlying type is implementation defined (integer-kind) type
- Automatic conversion to and from the implementation type
- Names should be globally unique, values do not need to be unique

```
typedef enum Shape {
    Square, Rectangle, Circle, Count // 0,1,2,3
}
shape_t;

void f(void)
{
  shape_t s = Rectangle;  // 1
  if ( s < Count-1)
    ++s;
  printf("%d\n", s);  // 2
}
```

Struct

- Almost all modern languages has a datatype for TUPLE

  R = T1 x T2 x … x Tn

- A struct groups one or more fields of possibly different types
- A struct defines a composite data type. Other languages call this as:
  - Struct (Algol 68, C++, C#, ...)
  - Record (Pascal, ADA, ...)
  - Class (Java and other OOP languages)

Struct in C

- The keyword **struct** is part of the type name.
- A struct has **members** or **fields.**
- The sizeof(struct Date) >= Σ sizeof(fields)   (gaps could be between fields)

```
struct Date;  // forward declaration

struct Date   // struct definition
{
  int year;
  int month;
  int day;
};

struct Date  today;                    // today is a Date variable
struct Date  xmas = { 2024, 12, 24 }; // aggregate initializ.
struct Date *ptr = &today;             // ptr is pointer to Date
struct Date  next(struct Date d);      // func. with Date parameter
                                       // returning Date
```

Struct

- A struct defines a tuple of types. The keyword **struct** is part of the type name.
- A struct has **members** or **fields.**
- The sizeof(struct Date) >= Σ sizeof(fields)   (gaps could be between fields)

```
struct Date   // struct definition
{
  int year;
  int month;
  int day;
};
typedef struct Date Date_t;  // Date_t is alias of struct Date

Date_t  today;                        // today is a Date variable
Date_t  xmas = { 2024, 12, 24 };  // aggregate initialization
Date_t *ptr = &today;                 // ptr is pointer to Date
Date_t  next(struct Date d);          // func. with Date parameter
                                      // returning Date
```

Struct

- A struct defines a tuple of types. The keyword **struct** is part of the type name.
- A struct has **members** or **fields.**
- The sizeof(struct Date) >= Σ sizeof(fields)   (gaps could be between fields)

```
typedef struct Date   // struct definition + typedef
{
  int year;
  int month;
  int day;
}
Date_t;           // Date_t is alias of struct Date

Date_t  today;                    // today is a Date variable
Date_t  xmas = { 2024, 12, 24 };  // aggregate initialization
Date_t *ptr = &today;             // ptr is pointer to Date
Date_t  next(struct Date d);      // func. with Date parameter
                                  // returning Date
```

Struct

- A struct may be unnamed
  - If we define variable(s)
  - If it is part of an outer struct or union

```
struct Date   // unnamed (anonymous) struct
{
  int year;
  int month;
  int day;
} today, xmas;  // today and xmas are Date variables

xmas.year  = 2024;
xmas.month = 12;
xmas.day   = 24;

today = xmas;
```

Struct

- A struct may be unnamed
  - If we define variable(s)
  - If it is part of an outer struct or union

```
struct Invoice
{
  long  sum; // never use floating point when rounding is bad
  struct D
  {
    int year;
    int month;
    int day;
  } d;
};

struct Invoice a1;
a1.sum = 123456;
a1.d.year = 2024;  // year in the struct D inside Invoice
```

Struct

- A struct may be unnamed
  - If we define variable(s)
  - If it is part of an outer struct or union

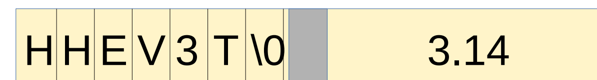
```
struct Invoice
{
  long  sum; // never use floating point when rounding is bad
  struct
  {
    int year;
    int month;
    int day;
  };         // unnamed struct
};

struct Invoice a1;
a1.sum = 123456;
a1.year = 2024;  // year in the unnamed struct inside Invoice
```

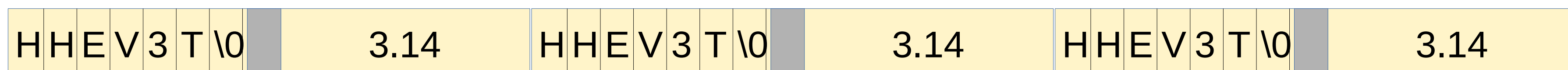
Struct

```
struct Date
{
  int year;
  int month;
  int day;
};

struct CreditAvg
{
  char   neptun_id[7];
  double average;
};

struct Date today;
struct Date xmas;

struct CreditAvg cr;
struct CreditAvg students[20];
```

Struct

```
struct Date
{
  int year;
  int month;
  int day;
};

struct CreditAvg
{
  char   neptun_id[7];
  double average;
};

struct Date today;
struct Date xmas;

struct CreditAvg cr;
struct CreditAvg students[20];
```





Struct

```
struct Date
{
  int year;
  int month;
  int day;
};

struct CreditAvg
{
  char   neptun_id[7];
  double average;
};

struct Date today;
struct Date xmas;

struct CreditAvg cr;
struct CreditAvg students[20];
```

GAP

2024	12	24

H	H	E	V	3	T	\0		3.14

H	H	E	V	3	T	\0		3.14	H	H	E	V	3	T	\0		3.14	H	H	E	V	3	T	\0		3.14

Initialization of struct

- A struct can be initialized with { … }
- If the list is shorter, the remaining elements will initialize to zero
- **Designators** can be used to initialize a field
- Following fields are initialized in order

```
typedef struct Date   // struct definition
{
  int year;
  int month;
  int day;
} date_t;

void f(void)
{
  date_t xmas = { 2024, 12, 24 }; // aggregate initialization
  date_t ymas = { 2024, 12 };                  // ymas.day = 0
  date_t zmas = { .month=12, 24, .year=2025 }; // zmas.day = 24
}
```

Operation on struct

- A struct can be assigned, passed as parameter, and return from function
- Address operator **&**
- Field selector with **.** or **->**
- **sizeof** operator

```
struct Date   // struct definition
{
  int year;
  int month;
  int day;
};
void f(void)
{
  struct Date xmas = { 2024, 12, 24 };
  struct Date *ptr = &xmas;  // pointer to struct Date
  xmas.year = 2025;          // year is int
  (*ptr).month = 1;          // month is int
  ptr->day = 1;              // day is int
}
```

Operation on struct

- A struct can be assigned, passed as parameter, and return from function
- Address operator **&**
- Field selector with **.** or **->**
- **sizeof** operator

```
struct Date    // struct definition
{
  int year;
  int month;
  int day;
};
void f(void)
{
  struct Date xmas = { 2024, 12, 24 };
  struct Date ymas;

  ymas = xmas;     // copy sizeof(struct Date) bytes
}
```

Forward declaration

- Self containment is not valid
- Pointer to the same struct is allowed
- Pointer to other types allowed only after forward declaration

```
struct Person
{
  char           name[20];
  struct Date    birthday;
  struct Person *father;
  struct Person *mother;
  struct Person *children[10];
};
```

Forward declaration

- Self containment is not valid
- Pointer to the same struct is allowed
- Pointer to other types allowed only after forward declaration

```
typedef struct Person
{
  char            name[20];
  struct Date     birthday;
  struct Person *father;
  struct Person *mother;
  struct Person *children[10];
}
person_t;
```

Forward declaration

- Self containment is not valid
- Pointer to the same struct is allowed
- Pointer to other types allowed only after forward declaration

```
typedef struct Person
{
  char            name[20];
  struct Date     birthday;
  person_t       *father;   // error, person_t is unknown here
  person_t       *mother;   // error, person_t is unknown here
  person_t       *children[10];  // error, person_t is unknown
}
person_t;
```

Forward declaration

```
struct Person
{
  char           name[20];
  struct Date    birthday;
  struct Person *father;
  struct Person *mother;
  struct Person *children[10];
};


struct Staffmember
{
  struct Person   pers;
  struct Manager *boss;
};

struct Manager
{
  struct Person       pers;
  struct Manager     *boss;
  struct Staffmember *staff[20];
};
```

Forward declaration

```
struct Person
{
  char           name[20];
  struct Date    birthday;
  struct Person *father;
  struct Person *mother;
  struct Person *children[10];
};


struct Staffmember
{
  struct Person   pers;
  struct Manager *boss;   // error, struct Manager unknown
};

struct Manager
{
  struct Person         pers;
  struct Manager       *boss;
  struct Staffmember *staff[20];
};
```

Forward declaration

```
struct Person
{
  char           name[20];
  struct Date    birthday;
  struct Person *father;
  struct Person *mother;
  struct Person *children[10];
};

struct Manager;  // forward declaration of Manager

struct Staffmember
{
  struct Person   pers;
  struct Manager *boss;   // ok, struct Manager forward declared
};

struct Manager
{
  struct Person       pers;
  struct Manager     *boss;
  struct Staffmember *staff[20];
};
```

Flexible array in struct

- The last field of struct can be an incomplete array
- The flexible array is not part of sizeof
- Assignment (using sizeof) will not copy the flexible array

```
typedef struct Student   // struct definition
{
  char neptun[7];
  int  semester;
  int  marks[];  // incomplete array type, should be the last
} student_t;     // marks[] does not count in sizeof

void f(char *code, int sem, int n, student_t *sp2)
{
  student_t *sp = malloc(sizeof(student_t)+n*sizeof(int));
  strcpy( sp->neptun, code);
  sp->semester = sem;
  sp->marks[0] = 5; /* ... */  sp->marks[n-1] = 2;
  *sp2 = *sp;
  free(sp);
}
```

Flexible array in struct

- The last field of struct can be an incomplete array
- The flexible array is not part of sizeof
- Assignment (using sizeof) will not copy the flexible array

```
typedef struct Student   // struct definition
{
  char neptun[7];
  int  semester;
  int  marks[];  // incomplete array type, should be the last
} student_t;     // marks[] does not count in sizeof

void f(char *code, int sem, int n, student_t *sp2)
{
  student_t *sp = malloc(sizeof(student_t)+n*sizeof(int));
  strcpy( sp->neptun, code);   // security issue
  sp->semester = sem;
  sp->marks[0] = 5; /* ... */  sp->marks[n-1] = 2;
  *sp2 = *sp;
  free(sp);
}
```

Flexible array in struct

- The last field of struct can be an incomplete array
- The flexible array is not part of sizeof
- Assignment (using sizeof) will not copy the flexible array

```c
typedef struct Student   // struct definition
{
  char neptun[7];
  int  semester;
  int  marks[];  // incomplete array type, should be the last
} student_t;     // marks[] does not count in sizeof

void f(char *code, int sem, int n, student_t *sp2)
{
  student_t *sp = malloc(sizeof(student_t)+n*sizeof(int));
  strncpy( sp->neptun, code, 7); sp->neptun[6] = '\0';
  sp->semester = sem;
  sp->marks[0] = 5; /* ... */  sp->marks[n-1] = 2;
  *sp2 = *sp;
  free(sp);
}
```

Flexible array in struct

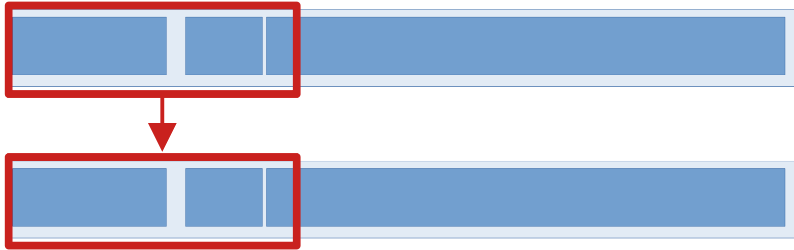
- The last field of struct can be an incomplete array
- The flexible array is not part of sizeof
- Assignment (using sizeof) will not copy the flexible array

```
typedef struct Student   // struct definition
{
  char neptun[7];
  int  semester;
  int  marks[];  // incomplete array type, should be the last
} student_t;     // marks[] does not count in sizeof

void f(char *code, int sem, int n, student_t *sp2)
{
  student_t *sp = malloc(sizeof(student_t)+n*sizeof(int));
  strncpy( sp->neptun, code, 7); sp->neptun[6] = '\0';
  sp->semester = sem;
  sp->marks[0] = 5; /* ... */  sp->marks[n-1] = 2;
  *sp2 = *sp;          // likely wrong!
  free(sp);
}
```

Flexible array in struct

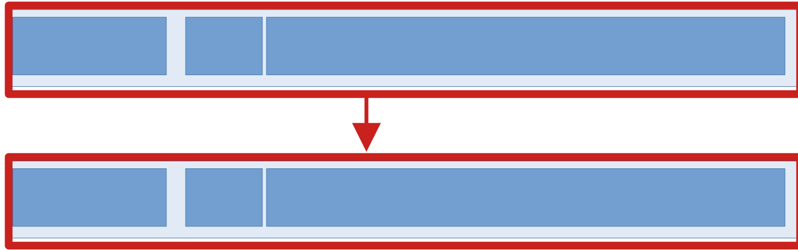
```
typedef struct Student   // struct definition
{
  char neptun[7];
  int  semester;
  int  marks[];  // incomplete array type, should be the last
} student_t;     // marks[] does not count in sizeof

void f(char *code, int sem, int n, student_t *sp2)
{
  student_t *sp = malloc(sizeof(student_t)+n*sizeof(int));
  strncpy( sp->neptun, code, 7); sp->neptun[6] = '\0';
  sp->semester = sem;
  sp->marks[0] = 5; /* ... */  sp->marks[n-1] = 2;
  *sp2 = *sp;          // likely wrong!
  free(sp);
}
```

Flexible array in struct



```
typedef struct Student    // struct definition
{
  char neptun[7];
  int  semester;
  int  marks[];  // incomplete array type, should be the last
} student_t;     // marks[] does not count in sizeof

void f(char *code, int sem, int n, student_t *sp2)
{
  student_t *sp = malloc(sizeof(student_t)+n*sizeof(int));
  strncpy( sp->neptun, code, 7); sp->neptun[6] = '\0';
  sp->semester = sem;
  sp->marks[0] = 5; /* ... */  sp->marks[n-1] = 2;
  *sp2 = *sp;          // likely wrong!
  free(sp);
}
```

Flexible array in struct



```c
typedef struct Student   // struct definition
{
  char neptun[7];
  int  semester;
  int  marks[];  // incomplete array type, should be the last
} student_t;     // marks[] does not count in sizeof

void f(char *code, int sem, int n, student_t *sp2)
{
  student_t *sp = malloc(sizeof(student_t)+n*sizeof(int));
  strncpy( sp->neptun, code, 7); sp->neptun[6] = '\0';
  sp->semester = sem;
  sp->marks[0] = 5; /* ... */  sp->marks[n-1] = 2;
  memcpy( sp2, sp, sizeof(student_t)+n*sizeof(int));
  free(sp);
}
```

Union

- Many (but not all) modern languages has a datatype for UNION

  $U = T1 \cup T2 \cup \ldots \cup Tn$

- A union stores one of the member types at every time == active type
- Tagged union
  - The variable knows that which is the active type
  - Algol68: united mode, Pascal, ADA, Modula-2: variant record
  - Scala: case class, Rust: enum, Swift
  - C++ std::variant
- Non-tagged union
  - Active type should be administered outside of union
  - C, C++: union

Union in C

- Object of union can contain one of the members in every time
- Only that member can be read which was written last (active type)
- The sizeof of the union is equal or greater to the largest member (padding)
- Non-tagged union (the union does not know which is the active member)

```
union Employee
{
  Manager      man;
  Staffmember  stm;
};

void f(void)
{
  union Employee e;
  if ( isManager() )
  {
    struct Manager m = /* ... */ ;
    e.man = m;  // Manager is the active type
  }
}
```

Tagged union in C

- Tagged union can be simulated in C with struct of enum and union

```
enum Shape_tag { square_tag, rectangle_tag, circle_tag };

struct Shape
{
  enum Shape_tag tag;
  union U
  {
    struct Square    s;
    struct Rectangle r;
    struct Circle    c;
  } u;
};

void print_shape(struct Shape x)
{
  switch(x.tag)
  {
  default          : printf("Unknown type"); break;
  case    square_tag: print_square(x.u.s);    break;
  case rectangle_tag: print_rectangle(x.u.r); break;
  case    circle_tag: print_circle(x.u.c);    break;
  }
}
```

Tagged union in C

- Tagged union can be simulated in C with struct of enum and union

```
enum Shape_tag { square_tag, rectangle_tag, circle_tag };

struct Shape
{
  enum Shape_tag tag;
  union              // unnamed (anonymous) union
  {
    struct Square    s;
    struct Rectangle r;
    struct Circle    c;
  };
};

void print_shape(struct Shape x)
{
  switch(x.tag)
  {
  default          : printf("Unknown type"); break;
  case    square_tag: print_square(x.s);      break;
  case rectangle_tag: print_rectangle(x.r);   break;
  case    circle_tag: print_circle(x.c);      break;
  }
}
```

Type punning

- Type punning is when we interpret the bitstring of type A as type B
- It is not a real conversion, it is the reinterpretation of the bits
- The only valid situation to read not the active type

```
union C
{
  double d;
  char   c[sizeof(double)];
};

int main(void)
{
  union C x;
  x.d = 3.14;
  for (int i = 0; i < sizeof(double); ++i)
    printf("%02x ", x.c[i]);
  putchar('\n'); // 1F FFFFFF85 FFFFFFEB 51 FFFFFFB8 1E 09 40
  return 0;
}
```

Type punning

- Type punning is when we interpret the bitstring of type A as type B
- It is not a real conversion, it is the reinterpretation of the bits
- The only valid situation to read not the active type

```
union C
{
  double d;
  unsigned char c[sizeof(double)];
};

int main(void)
{
  union C x;
  x.d = 3.14;
  for (int i = 0; i < sizeof(double); ++i)
    printf("%02x ", x.c[i]);
  putchar('\n'); // 1F 85 EB 51 B8 1E 09 40
  return 0;
}
```