Week 8

XMUT-NWEN 241 - 2024 T2

Systems Programming

Mohammad Nekooei

School of Engineering and Computer Science

Victoria University of Wellington

Content

- User-Defined Types
  - Enumeration
- Derived data types
  - Union

Enumeration types

Background

- Basic data types
  - int : integer ✓
  - char : character ✓
  - float : floating point number ✓
  - double : double-precision floating point number ✓
- Derived data types
  - Arrays ✓
  - Strings ✓
  - Structures ✓
  - Unions
- User defined data types
  - *Enumeration* types

Motivation for Enumeration Type

- Oftentimes, a variable can only take a few possible discrete values

- Macro is often used to define symbolic constants that will represent possible values of the variable

- **Enumeration is a better alternative**

```
#define COLOR_RED     0
#define COLOR_YELLOW  1
#define COLOR_GREEN   2

int main(void)
{
    int color;
    // can either be 0, 1 or 2
    …
    color = COLOR_GREEN;
}
```

Enumeration

- Enumeration is a user-defined data type that is used to assign identifiers to **integral constants**

- Declaration syntax:

```
enum enum_tag {name_0, name_1, ..., name_n} variable_list;
```

  - Defines a new enumerated type
  - Defines symbolic constants that take on integer values from `0` through `n`
    - `name_0` has value `0`, `name_1` has value `1`, and so on

Enumeration

- Enumeration is a user-defined data type that is used to assign identifiers to **integral constants**

- Declaration syntax:

```
enum enum_tag {name_0, name_1, ..., name_n} variable_list;
```

  - *enum_tag* and *variable_list* are optional

Enumeration

As an example, the statement:

```
enum colors { red, yellow, green };
```

- Defines a new enumerated type `enum colors`
- Defines three integer constants: `red` is assigned the value 0, `yellow` is assigned 1 and `green` is assigned 2
- Any variable of `enum colors` type or basic data type can be assigned either `red`, `yellow` or `green`

Enumeration

Unnamed enumeration example:

```
enum { red, yellow, green };
```

- Defines three integer constants: `red` is assigned the value 0, `yellow` is assigned 1 and `green` is assigned 2
- Any variable of basic data type can be assigned either `red`, `yellow` or `green`

Enumeration

- It is possible to override the integer assignment, e.g.

```
enum colors {red = 3, yellow = 2, green = 1};
```

- `typedef` can be used to create an alias for the new type, e.g.

```
typedef enum colors {red = 3, yellow = 2, green = 1} color_t;
```

  - `color_t` is a new type which can be used for declaring variables

Enumeration

- If an identifier is assigned a value and subsequent identifiers are not assigned, the subsequent identifiers continue the progression from the assigned value

```
enum colors { red, yellow = 3, green, blue };
```

`red` is assigned the value 0, `yellow` is assigned 3, `green` is assigned 4, and `blue` is assigned 5.

enum Example (1)

```
#include <stdio.h>

/* Declaration defines new enumerated type and integer constants */
enum colors { red, yellow = 3, green, blue };

int main(void)
{
   /* Declaration defines variables of type enum colors */
   /* Can take values of red, yellow, green or blue */
   enum colors fgcolor = blue, bgcolor = yellow;

   printf ("%d %d\n", fgcolor, bgcolor);
   /* Will print 5 3 */

   return 0;
}
```

enum Example (2)

```c
#include <stdio.h>

/* Declaration defines integer constants */
enum { red, yellow = 3, green, blue };

int main(void)
{
   /* Declaration defines variables of type int */
   /* Can be assigned red, yellow, green or blue */
   int fgcolor = blue, bgcolor = yellow;

   printf ("%d %d\n", fgcolor, bgcolor);
   /* Will print 5 3 */

   return 0;
}
```

Repeated Identifiers

- An identifier in an enumerated type cannot be re-used to declare a new variable or enumeration in the same scope

```
void func(void)
{
   enum colors { red, yellow, black };
   enum rgb { red, green, blue };
   ...
}
```

```
void func(void)
{
   enum colors { red, yellow, black };
   int red;
   ...
}
```

Will not compile due to re-use of identifier `red` in the same scope

Unions

Unions

- A union is like a struct, but the different fields take up the **same** space within memory

- Declaration syntax:

```
union union_tag {
      type1 member1;
      type2 member2;
      ...
} variable_list;
```

  - *union_tag* specifies the name of the union
  - *union_tag* and *variable_list* are optional
  - If *union_tag* is not specified, *variable_list* should be specified; otherwise, there is no way to declare variables using the unnamed union type

Unions

- A union is like a struct, but the different fields take up the **same** space within memory

- Declaration syntax:

```
union union_tag {
     type1 member1;
     type2 member2;
     ...
} variable_list;
```

- Union members can be
  - Basic data types
  - Derived and user-defined types
  - Pointers to basic, derived and user-defined data types
  - Function pointers

Union vs Structure

	Structure	Union
Declaration syntax	Same	
Storage allocation	Allocates storage for all members separately	• Allocates common storage for all its members • Space is allocated to hold the biggest member
Access	All members can be accessed at the same time	Only one member can be "active" at any given time

Union vs Structure: Storage Allocation

```c
struct space {
    int i;
    float f;
    char c[4];
};
```

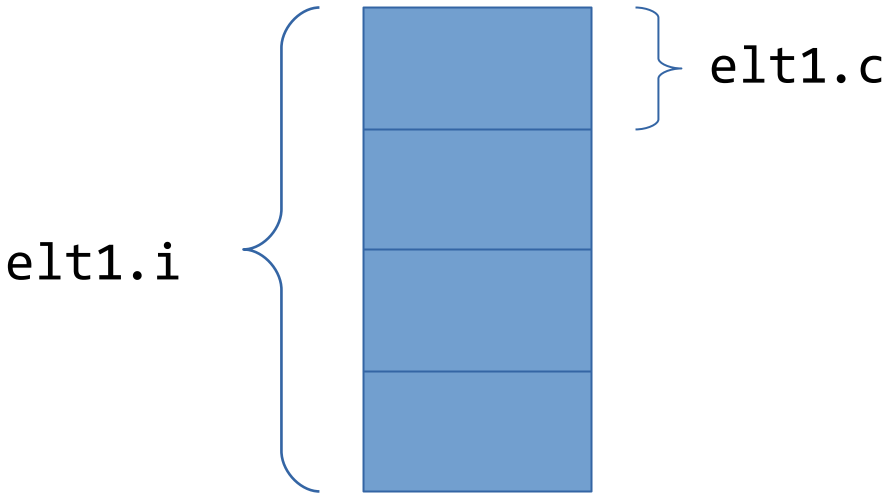
```c
union space {
    int i;
    float f;
    char c[4];
};
```

```
sizeof(struct space) = sizeof(i) + sizeof(f) + sizeof(c)

sizeof(union space) = max(sizeof(i), sizeof(f), sizeof(c))
```

union Example

```
union elt {
   int   i;
   char  c;
} elt1;

elt1.c = 'A';
elt1.i = 300;
```

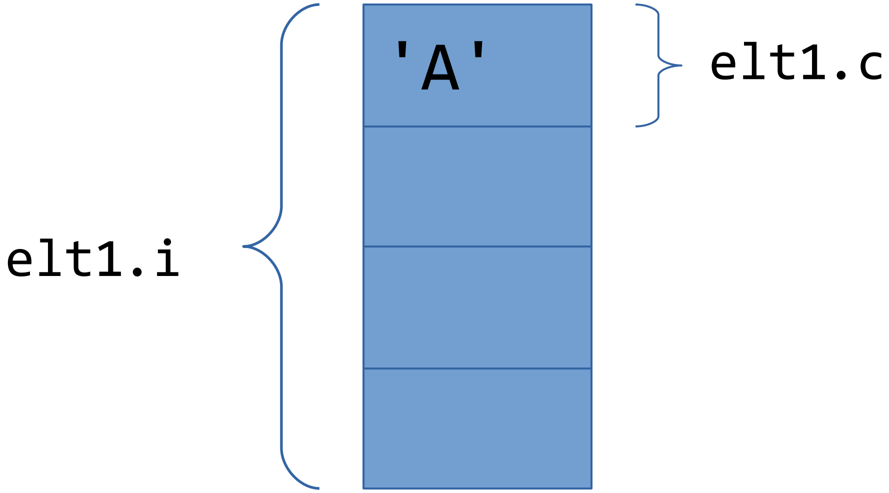
Assuming an `int` takes up 32 bits (4 bytes):



union Example

```
union elt {
   int    i;
   char   c;
} elt1;

elt1.c = 'A';
elt1.i = 300;
```

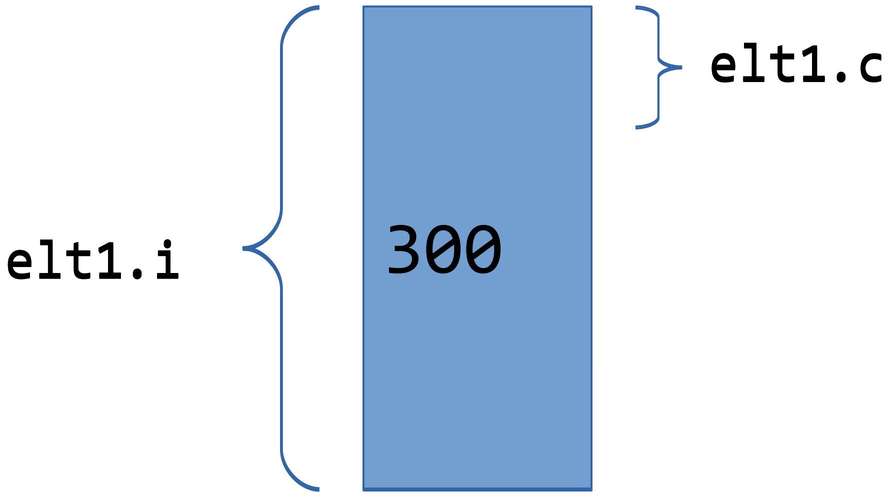
Assuming an `int` takes up 32 bits (4 bytes):



union Example

```
union elt {
    int    i;
    char   c;
} elt1;

elt1.c = 'A';
elt1.i = 300;
```

Assuming an `int` takes up 32 bits (4 bytes):



Review: Strings

- `long int strlen(const char* source);`
  - Calculates the length of a given string, up to the first null character.

- `char* strcpy(char* destination, const char* source);`
  - Copies the source string to the destination character array.

- `int strcmp (const char* str1, const char* str2);`
  - Compares two strings and returns 0 if both strings are identical.

- `char *strcat(char *dest, const char *src);`
  - Concatenates two strings and stores the result in the first argument.

Review: Structures

```
//declare "struct person" type
struct person
{
  char name[100];
  int age
};

// give it an alias person_t
typedef struct person person_t;
```

- Struct is just a collection of variables (which can have different types) under a single name
- You can access members with the ‘.’ operator or through a pointer with the ‘->’ operator
- A struct can be referenced, copied, and assigned to
- The size of a struct is guaranteed to be as large as the sum as the size of its members

Review: * And &

	In Declaration	In Expression
*	**`int *i;`** *Declare* ***`i`*** *as a pointer to* ***`int`***	**`*i`** *Dereference `i` or obtain the value that `i` points to*
&	N/A	**`&i`** *Get the address of `i` (a pointer to `i`)*

Review: Pointers and Arrays

- **Array decays into a pointer: an array is just a fixed pointer**

- You cannot re-assign an array to point to another location

- You can let another pointer point to the array

```
int *p;
```

- p can point to an `int`
- p can point to an array of `int`

Introducing GDB

- GDB: GNU Debugger
- A much better way to debug your programs
  - No need to rely on `printf()` to see the values of the variables
  - You can step through your code
  - You can even change variable values!!!
- You learn more about GDB in Exercise 2 (out on Monday, 21 October)