Week 6 Lecture 1

NWEN 241
Systems Programming

Jyoti Sahni

Jyoti.sahni@ecs.vuw.ac.nz

Announcement

- Mid-Term Test on 19 April, Friday @17:00
  - Covers topics from Week 1 – Week 6(lecture 1 – Enum, Union; file handling will not be asked)
  - Test is 50 minutes long, total of 45 marks
  - True/False, multiple choice and short answer questions

- Room Assignment (based on Last Name or Surname)
  - HMLT205 : (A… - L…)
  - KKLT303: (M… - Z…)

- If you are overseas and want to sit online, email me

- Feedback portal is now open, please share your feedback

Content

User Defined Data Types

File Stream I/O

User Defined Data 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-1} 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)

```
#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

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

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

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

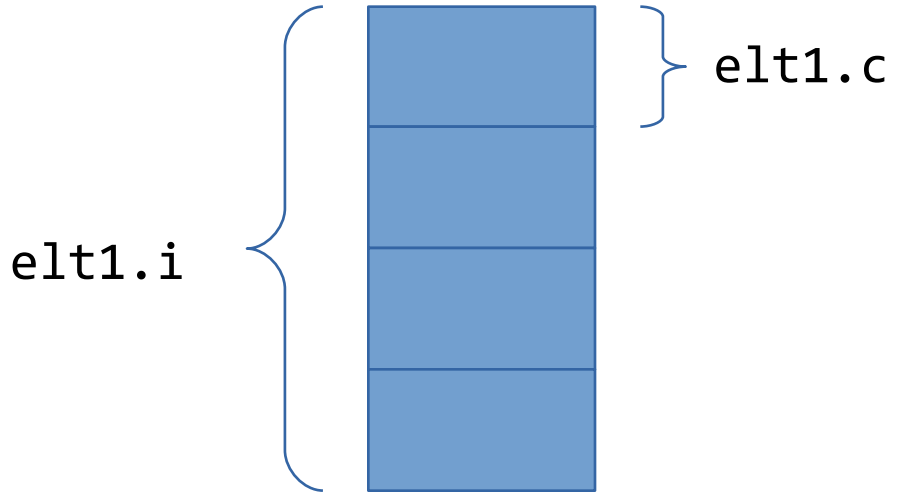
sizeof(union space) = *max*(sizeof(i), sizeof(f), sizeof(c))

union Example

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

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

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

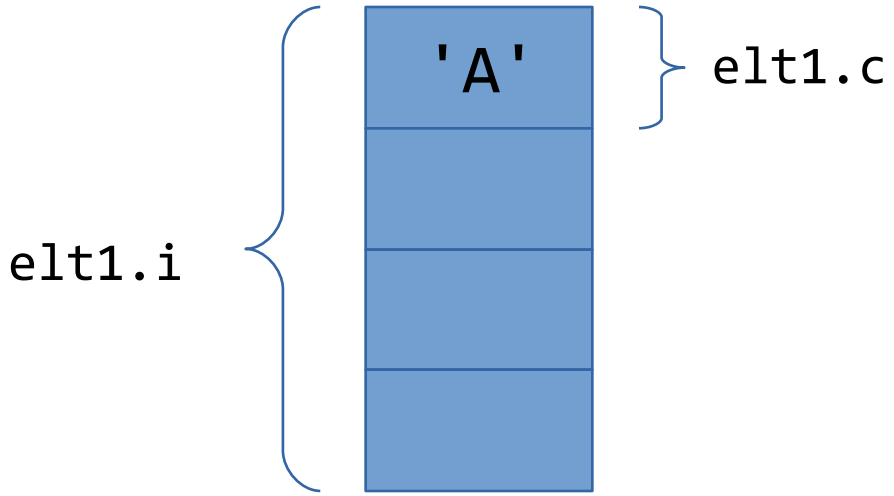


union Example

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

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

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

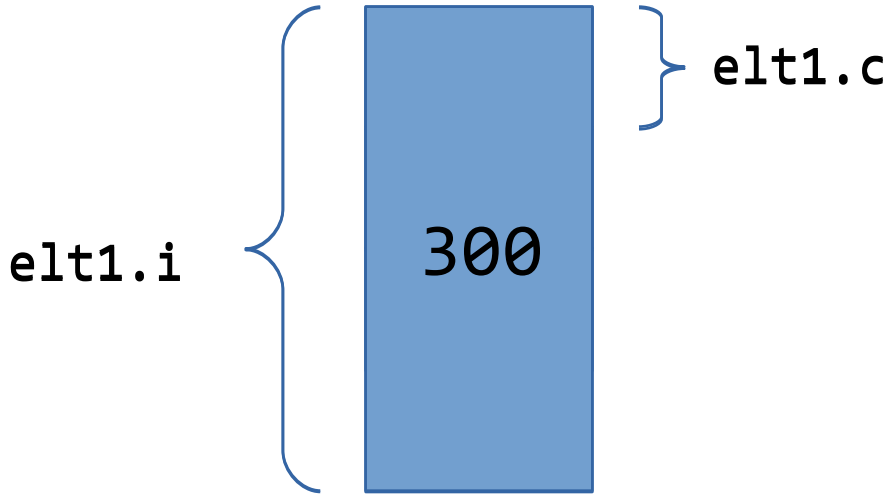


union Example

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

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

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



File Stream I/O

Introduction to File Input / Output

- **I/O** is the process of copying data between main memory and external devices, like terminals (keyboards), disk drives, networks, etc.

- In C, everything is abstracted as a ***file***
  - Each file is simply a sequential stream of bytes
  - C imposes no structure on a file

- **From the program's point of view, data input and data output are made possible through files**

Accessing Files

- **A file must first be opened properly before it can be accessed for reading or writing**

- Opening a file establishes a “communication channel” between the program and the file

File Stream vs File Descriptor

- "Communication channel" can either be a **file stream** or **file descriptor**
- C provides functions for accessing files via file stream or file descriptor

	File Descriptor	File Stream
Content access	*Primitive access*: contents can be accessed as blocks of bytes	*Rich access*: contents can be formatted using format specifiers
Control operations	Allows setting of control parameters	Does not allow
Special I/O modes	Allows special access modes such as non-blocking	Does not allow
Buffering	None	Supports 3 modes of buffering

File Stream vs File Descriptor

- **File streams provide a higher-level interface, layered on top of the primitive file descriptor facilities**

- For special files (e.g. I/O devices and **sockets**), file descriptor is the recommended approach

- For **regular files** (files on disk), file stream is the recommended approach

Stream Buffering

- One of the common pitfalls when dealing with file streams is **buffering**

- More problematic in interactive I/O streams
  - Data written by program to file does not appear immediately
  - Data read by program from file does not appear immediately

Illustration

```
char str[100];
scanf ("%s", str);
```

- If user types the string

```
The quick brown fox
```

- The string `str` will only be assigned `"The"`
- What happens to the rest?

Stream Buffering Modes

Mode	Description
Unbuffered	Characters written to or read from an unbuffered stream are transmitted individually to or from the file as soon as possible.
Line buffered	Characters written to a line buffered stream are transmitted to the file in blocks when a newline character is encountered.
Fully buffered	Characters written to or read from a fully buffered stream are transmitted to or from the file in blocks of arbitrary size.

- Newly opened streams are fully buffered by default, except streams connected to interactive devices which are line buffered
- C provides functions for changing stream buffering mode

Next Lecture

- Continuation of File Stream I/O
- Command Line Arguments