Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

CS Basics
13) Structures and Files

E. Benoist & C. Grothoff
Fall Term 2018-19

Computer Science Division

Structures and Files

Enumerations

Enumeration

- **Data type**
  - Its members are constants that are written as identifiers
  - They have signed integer values
  - The constants represent values that can be assigned to corresponding enumeration variables
  - Similar to Java
- **Syntax**

```
enum tag {member1, member2, member3, ...↘
→ , memberN}
```

- **Each member is a constant (signed int)**
  - A variable can have an enum for a type

```
enum tag variable1, variable2, variable3;
variable1 = member1;
variable2 = memberN;
```

Enumeration

- **Example of use**
  - An enumeration of colors
  - The variable is of type `enum color`
  - Its value is an unsigned integer
- **Usage:**

```
enum color {black, white, red, green, blue↘
→, yellow};
enum color col1 = black;
enum color col2 = white;
enum color col3 = red;
printf("Print␣colors␣col1=%d,␣col2=%d,␣↘
→col3=%d\n", col1,col2,col3);
/* Output: Print colors col1=0, col2=1, ↘
→col3=2 */
```


Enumerations (Cont.)

- **One can set the value of one or more members**
  - Example

```
enum color2 {b=-1, w, r, g, bl, y↘
→};
```

  - The value of the variables is the one defined
- **Example**

```
enum color2 {b=-1, w, r, g, bl, y};
enum color2 col4 = b;
enum color2 col5 = w;
enum color2 col6 = r;
printf("Print␣colors␣col4=%d,␣col5=%d,␣↘
→col6=%d\n", col4,col5,col6);
/* Output Print colors col4=-1, col5=0, ↘
→col6=1 */
```


Enumerate

- **Value is an integer**
  - It can be used in a switch
  - Can compare value to members (= constants)

```
enum color2 {b=-1, w, r, g, bl, y};
enum color2 background=w,foreground;
switch(background){
b:
g:
bl:
  foreground=w;
  break;
w:
y:
  foreground=b;
default:
  foreground=bl;
}
printf("Background:%d␣,␣Foreground:␣%d\n",
     background,foreground);
```


Structures and Unions

Structures

Structure

- **Contains different elements**
  - Can differ in type
- **A structure contains members**
  - Can be ordinary variables, pointers, arrays, or other structures
- **Syntax**

```
struct tag {
  member1;
  member2;
  ...
  memberm;
};
```


Typical Structure declaration

- **Bank account**
  - Number
  - Type
  - Name
  - Balance
- **Example**

```
struct account{
  unsigned int acct_no;
  enum { CHECKING, SAVINGS } acct_type;
  char name[80];
  float balance;
};
```


Define a variable in a structure

- **The name "struct account" can be used as a type name**
  - Define a variable

```
/* define a variable myAccount2 of type ↘
→struct account */
struct account myAccount2;

/* Define an array containing 100 struct ↘
→accounts */
struct account myAccountArray[100];
```

- **Variable can also be defined at the construction of the structure**
  - Direct in the end of the definition
  - One can define one or many variables.

```
struct account {
  int acct_no;
  char acct_type;
  char name[80];
  float balance;
} myAccount1, myAccount2;
```

Initialize a variable

- **One can set members of a structure**
  - Syntax: `variable.member = value;`

```
myAccount1.acct_no = 10;
myAccount1.acct_type =SAVINGS;
myAccount1.acct_no = 9;
```

- **One can initialize as structure like an array**
  - One value for each member in a list

```
struct account myAccount3 = {100,SAVINGS,"↘
→Hans␣Muster", 1023.9};
```


Use a structure inside a structure

- **Structure can be used like any other type, even inside another structure**
  - Needs to be used after the definition
  - `struct name` is used as a type

```
struct date{
  int year;
  int month;
  int day;
};
struct birthdate{
 char name[80];
 struct date birth;
};
```


Define and initialize an array of structures

- **Initialize an array, loop**

```
static struct birthdate birthdays[]={
  {"Emmanuel",{1960,10,24}},
  {"Jean",{1940,1,30}},
  {"Paul",{1930,5,10}}
};
int size_birthdays = 3;

printf("Birthdays␣list:\n");
int i;
for(i=0;i< size_birthdays ;i++){
  printf("Birth␣of␣%s␣was␣on␣%d.%d.%d␣\n",↘
  →birthdays[i].name,birthdays[i].birth.↘
  →day,birthdays[i].birth.month,birthdays↘
  →[i].birth.year);
}
```


Define a new type

- **Use the `typedef` instruction**
  - Is used to define a new type:

```
typedef int age;
age i=10; /* i is an int */
```

  - or

```
typedef float height[100];
height myArray; /* is an array of ↘
→100 floats */
```

- **Obscures nature of type, common but not good style!**
  - We define a new type corresponding to the `struct`

```
struct user_structure{
  int user_id;
  char* username;
  char* password;
};
  typedef  struct user_structure user;
```

Define a new type

We can even avoid giving the `struct` a name:

```
typedef struct {
  int user_id;
  char* username;
  char* password;
} user;
```

Also possible when declaring variables or using nested `structs`:

```
struct {
  struct {
    int i;
    int j;
  } a, b;
} var;
use (var.a.i, var.b.j);
```


Pointers on Structures

- **A structure is often large**
  - Not convenient as a function argument: need to be copied
  - Value changed in the function is not changed in the calling code
  - Lot of memory is needed
- **Solution: use pointers**
  - Work like pointers on any type
  - A pointer contains the address of an object in memory
  - The pointed value is accessed using `*`

Pointers on Structures (Example)

- **Example (using the type `user`)**

```
user u1 = {
 .user_id = 1, .username = "bie1"  };
printf("u1: Userid= %d, username=%s\n",
          u1.user_id,u1.username);
user* pu1 = &u1;
printf("pu1: Userid= %d, username=%s\n",
           pu1->user_id,pu1->username);
user* list_of_users[80];
list_of_users[0]=&u1;
printf("lou[0]: Userid= %d, username=%s\n",
       list_of_users[0]->user_id, list_of_users
       →[0]->username);
u1.user_id=2;
printf("lou[0]: Userid= %d, username=%s\n",
        list_of_users[0]->user_id,
        list_of_users[0]->username);
```


Pointers on Structures (Example)

- **To allocate memory inside the heap: use `malloc()`**

```
list_of_users[1]=(user*)malloc(sizeof(user
→));
list_of_users[1]->user_id=3;
list_of_users[1]->username="beo1";
list_of_users[1]->password="bar";
printf("lou[1]: Userid= %d, username=%s
→and password =%s\n",
          list_of_users[1]->user_id,
          list_of_users[1]->username,
          list_of_users[1]->password);
```

Or define an allocation convenience macro:

```
#define my_new(t) (t*) malloc (sizeof (t))
```

Bitfields

- **Two new types**
  - date and person
- **We allocate two persons in the heap**
  - using (person*) malloc(sizeof(person));

```
struct date {
  int year;
  unsigned int month : 4; /* only uses 4 bits */
  unsigned int day : 5;   /* only uses 5 bits */
};
typedef struct{
  char* name;
  char* firstname;
  struct date birth;
  struct date death;
} person;
```


Another exemple of dynamic memory allocation (Cont.)

```
int main(){
  person* emmanuel = my_new (person);
  person* p2 = my_new (person);

  emmanuel->name="Benoist";
  emmanuel->firstname="Emmanuel";
  emmanuel->birth=(date){1960,10,29};
  emmanuel->death=(date){0,0,0};
  p2->name = "Muster";
  p2->firstname = "Hans";
  p2->birth=(struct date){1890,1,1};
  p2->death=(struct date){1918,11,11};
  person* list_of_persons[20];
  list_of_persons[0]=emmanuel;
  list_of_persons[1]=p2;
}
```


Unions

Unions

- **Two or more values "share" the same memory**
  - A union can only store one value at a time
  - Members: share the same memory
  - Used to save space
- **Initialize values**

```
typedef struct
{int r; int g; int b;} color;
union val{
  color col;
  int size;
};
int main(){
  union val array[10];
  array[0].col=(color){.r=255,.g=0,.b=0};
  array[1].size=41;
  array[2].col=(color){255,255,255};
  array[3].size=39;
  array[4].col=(color){0,255,55};
  array[5].size=44;
```

Size of a union

- **All the members are on the same place**
  - Size is the size of the largest member

```
printf("Size␣of␣color:␣%lu\n", sizeof(↘
→color));
printf("Size␣of␣val:␣%lu\n", sizeof(union ↘
→val));
printf("Size␣of␣int:␣%lu\n", sizeof(int));
/* Output
   Size of color: 12
   Size of val: 12
   Size of int: 4
*/
```


Memory is shared

- **Programmer is responsible to know which `union` field is used**
  - The programmer must know what was stored
  - Possible to use `union` to convert between types

```
union { unsigned int i; float f; } u;

u.f = 3.1415;
printf ("Bit␣representation␣Pi:␣%X\n",
        u.i);
u.f = 3.0;
printf ("Bit␣representation␣3.0:␣%X\n",
        u.i);
/* Output:
  Bit representation Pi: 40490E56
  Bit representation 3.0: 40400000 */
```


Often `enum`, `union` and `struct` are combined

```
struct InternetAddress {
  enum { IPv4, IPv6 } address_family;
  union {
    struct {
      uint32_t value;
    } ipv4;
    struct {
      uint32_t value[4];
    } ipv6;
  } address;
};
```


Writing and Reading Files

Writing and Reading Files

- **Read and Write files, like standard input and output**
  - Using functions similar to `scanf` and `printf`
  - For text files
- **Read and Write binary files**
  - Using low level functions
  - `putc()` and `getc()`
  - Read and write one byte at a time

Open and Close a File

- **File are accessed using FILE pointers**
  - Definition

```
FILE * ptvar;
```

  - The structure `FILE` is defined in `<stdio.h>`
  - Is called a *stream pointer* (or simply *stream*)
- **Open a file**
  - Syntax:

```
ptvar = fopen(file-name, file-type);
```


File types

- **File-type must be one of the following strings**
  - `"r"` Opening an existing file "read only"
  - `"w"` Open a new file for writing only. If the file currently exists, it will be truncated and replaced by a new file
  - `"a"` Open an existing file for appending. A new file is created if the file does not exist
  - `"r+"` Open an exisiting file for both reading and writing
  - `"w+"` Open a new file for both reading and writing. If the file exists, it is truncated.
  - `"a+"` Open an existing file for both reading and appending. A new file is created if the file does not exist
- **A data must be closed at the end of program execution**
  - Using the `fclose()` function.
- **Example**

```
FILE *file_pointer;
file_pointer = fopen("output.txt","w");
...
fclose(file_pointer);
```


Write in a text file

- **Open the file in "Write" only mode**
  - Use function `fprintf()` similar to `printf()` with a new parameter: the stream

```
#include <stdio.h>
int main(){
  FILE *file_pointer;
  file_pointer = fopen("output.txt","w");
  puts("Type␣a␣number");
  int nb;
  scanf("␣%d",&nb);
  fprintf(file_pointer,"Hello␣world\n");
  for(int i=0;i<nb;i++)
    fprintf(file_pointer, "%d", i);
  fclose(file_pointer);
}
```

Reading a text file

- **Using `fscanf()` function**
  - Similar to scanf() with a stream

```
FILE *file_pointer;
int n;
char *arrayOfStrings[10];

file_pointer = fopen("output2.txt","r");
puts("Reading strings from the file: output2.txt↘
→");
for (n=0;n<10;n++) {
  arrayOfStrings[n]= malloc (80);
  if (1 != fscanf (file_pointer, "%79s", ↘
  →arrayOfStrings[n]))
    break;
}
printf("List of strings\n");
for(int i=0;i<n;i++)
  printf("%d) %s\n",(i+1),arrayOfStrings[i]);
fclose(file_pointer);
```


Write/Read Binary Data

- **One can write binary data**
  - Put bytes after bytes in the file
  - Use function `putc(charstream)`,
- **One can read binary data**
  - Using the function `char getc(stream)`

```
#include <stdio.h>
int main(){
  FILE *file_pointer;
  file_pointer = fopen("output.bin","w");

  int data[10];
  ... /* initialize the value for data */
  char* data_char = (char*)data;
  for(int i=0;i<nb*sizeof(data);i++){
    putc(data_char[i],file_pointer);
  }
  fclose(file_pointer);
}
```


Write/Read Binary Data Quickly

Use `fwrite` and `fread` to read/write larger chunks:

```
#include <stdio.h>
int main(){
  FILE *fp;
  int out[10];
  int in[3];
  ... /* initialize the value for out */

  fo = fopen("output.bin","w+");
  fwrite (out, 10, sizeof (int), fp);
  fseek (fp, 4 * sizeof (int), SEEK_SET);
  fread (in, 3, sizeof (int), fp);
  /* Now: in[0] == out[4] */
  fclose(fp);
}
```


Accessing directories

```
#include <sys/types.h>
#include <dirent.h>
int main(){
  DIR *dir;
  struct dirent *de;

  dir = opendir ("/proc");
  while (NULL != (de = readdir (dir)))
    printf ("%s\n", de->d_name);
  closedir (dir);
}
```

File metadata

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
int main(int argc, char **argv){
  struct stat s;

  if (0 != stat (argv[1], &s))
    perror ("stat");
  else switch (sb.st_mode & S_IFMT) {
   case S_IFDIR:  printf("directory\n"); break;
   case S_IFIFO:  printf("FIFO/pipe\n"); break;
   case S_IFLNK:  printf("symlink\n");   break;
   case S_IFREG:  printf("file\n");      break;
   case S_IFSOCK: printf("socket\n");    break;
   default:       printf("other\n");     break;
  }
  return 0;
}
```


Conclusion

Conclusion

- **Structures are useful to store information**
  - Store complex data
  - New types
  - Very different from Classes (no methods, no inheritence, no encapsulation, ...)
  - Should often be used as pointers
- **Files**
  - Can be Binary or Text
  - Can be read / write /append, ...
  - Use functions similar to the ones used for stdin and stdout.

Bibliography

- This course corresponds to chapters *11*, *12* and *13* of the course book:
  Schaum's OuTlines, Programming with C (second edition), *Byron Gottfied*, Mc Graw-Hill, 1996