Structures
Proseminar “C – Grundlagen und Konzepte”

Michael Kuhn

Research Group Scientific Computing
Department of Informatics
Faculty of Mathematics, Informatics und Natural Sciences
University of Hamburg

2013-05-10
Overview

1. Structures and Unions
2. Alignment, Padding and Bit Fields
3. Access and Initialization
4. Compound Literals
5. Opaque Structures
6. Summary
struct

- structs can contain multiple variables
  - Potentially different data types
- The contained variables are called *members*

```c
struct foo1_s
{
    double bar;
    int baz;
};
```

```c
sizeof(double) == 8
sizeof(int) == 4
sizeof(struct foo1_s) == 12
```
union

- unions look the same as structs
- They only take up the space required by the largest member
  - Only one member is valid at any time

```c
union foo1_u
{
  double bar;
  char baz[8];
};
```

```c
sizeof(double)    == 8
sizeof(char[8])  == 8
sizeof(union foo1_u) == 8
```
typedef

- Writing “struct foo1_s” every time can be cumbersome
- typedef allows defining a new data type
  - Done implicitly in C++

```c
struct foo1_s
{
   double bar;
   int baz;
};

typedef struct foo1_s foo1_s;

struct foo1_s a;
foo1_s b;
```
Introduction

- The compiler aligns `struct` members for optimal access
  - This can lead to padding

```c
struct foo2_s
{
  char bar;
  int baz;
};
```

```c
sizeof(char) == 1
sizeof(int) == 4
sizeof(struct foo2_s) == 8
```

- `baz` starts at a 4 byte boundary, wasting 3 bytes after `bar`
- Best practice: Group members of same type, from largest to smallest
Bit Fields

- Bit fields allow limiting the memory occupied by a member
- Can be used to provide convenient bitwise access

```c
struct foo3_s
{
    unsigned int bar : 8;
    unsigned int flag : 1;
    unsigned int : 23;
};

struct foo3_s a;
a.bar = 255;
a.flag = 2; /* invalid! */
```

```
sizeof(struct foo3_s) == 4
```
Access

- Members can be accessed using . and ->
  - a->b is equal to (*a).b

```c
1 struct foo2_s
2 {
3     char bar;
4     int baz;
5 }

6 struct foo2_s a;
7 a.bar = 'a';

8 struct foo2_s b[1];
9 b->baz = 42;
```
Initialization

- structs can also be directly initialized
  - Unspecified members are initialized to zero

```c
struct foo2_s
{
    char bar;
    int baz;
};

struct foo2_s a = {'a', 42};
struct foo2_s b = {.baz = 23};
struct foo2_s c = {0};
```
## Assignments

- Compound literals have the following form:

\[
\text{(type)\{arguments\}}
\]

```
struct arg_s
{
    int a;
    int b;
};

struct arg_s a = { 1, 2 };    // valid
struct arg_s b;

b = { 1, 2 };  // invalid!

b = (struct arg_s){ 1, 2 };  // valid
```
Assignments, Function Arguments and Return Values

```c
1 struct arg_s* p;
2 p = &{ 1, 2 }; /* invalid! */
3 p = &(struct arg_s){ 1, 2 };
```

```c
1 static
2 struct arg_s
3 foo (struct arg_s a)
4 {
5     return (struct arg_s){ a.a, a.b };
6 }
7
8 foo({ 1, 2 }); /* invalid! */
9 foo((struct arg_s){ 1, 2 });
```
Putting struct definitions into header files unnecessarily leaks implementation details to users
- structs are often used as opaque data types

Listing 1: opaque.h

```c
1 struct opaque_s;
2 typedef struct opaque_s opaque_s;
3 char const* opaque_get_name (opaque_s*);
```
Definition

- The actual struct contents and implementation are hidden

Listing 2: opaque.c

```c
#include "opaque.h"

struct opaque_s
{
    char* name;
};

char const* opaque_get_name (opaque_s* o)
{
    return o->name;
}
```
Summary

- Structures and unions allow grouping of different data types.
- The compiler aligns and pads structures for optimal access, which can lead to wasted memory.
- Structures can be initialized by simply listing values for all members or by using designated initializers.
- Compound literals allow assigning structures directly and passing anonymous structures to functions.
- Opaque structures can be used to separate the interface from the implementation.
### Bonus: Strict Aliasing

- The compiler can assume that two objects of different data types do not reside at the same memory address.
  - That is, they do not *alias* each other.

```c
1  uint16_t a[2] = { 1, 2 };   
2  *(uint32_t*)a = 42;        
3  printf("%d %d\n", a[0], a[1]);
```

```
1  42 0
```

```
1  warning: dereferencing type-punned pointer 

  will break strict-aliasing rules

  [-Wstrict-aliasing]
```