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

struct

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

```
1 struct foo1_s
2 {
3   double bar;
4   int baz;
5 };
```

```
1 sizeof(double)          == 8
2 sizeof(int)             == 4
3 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

```
1 union foo1_u
2 {
3   double bar;
4   char baz[8];
5 };
```

```
1 sizeof(double)         == 8
2 sizeof(char[8])        == 8
3 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++

```
1  struct foo1_s
2  {
3    double bar;
4    int baz;
5  };
6
7  typedef struct foo1_s foo1_s;
8
9  struct foo1_s a;
10 foo1_s b;
```

Introduction

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

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

```
1 sizeof(char)           == 1
2 sizeof(int)            == 4
3 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

```
1  struct foo3_s
2  {
3    unsigned int bar  : 8;
4    unsigned int flag : 1;
5    unsigned int      : 23;
6  };
7
8  struct foo3_s a;
9  a.bar = 255;
10 a.flag = 2; /* invalid! */
```

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

Access

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

```
1  struct foo2_s
2  {
3    char bar;
4    int baz;
5  };
6
7  struct foo2_s a;
8  a.bar = 'a';
9
10 struct foo2_s b[1];
11 b->baz = 42;
```

Initialization

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

```
1 struct foo2_s
2 {
3   char bar;
4   int baz;
5 };
6
7 struct foo2_s a = { 'a', 42 };
8 struct foo2_s b = { .baz = 23 };
9 struct foo2_s c = { 0 };
```

Assignments

- Compound literals have the following form:
  `(type){arguments}`

```
 1 struct arg_s
 2 {
 3   int a;
 4   int b;
 5 };
 6
 7 struct arg_s a = { 1, 2 };
 8 struct arg_s b;
 9 b = { 1, 2 }; /* invalid! */
10 b = (struct arg_s){ 1, 2 };
```

Assignments, Function Arguments and Return Values

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

```
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 });
```

Declaration

- Putting `struct` definitions into header files unnecessarily leaks implementation details to users
  - `structs` are often used as opaque data types

Listing 1: `opaque.h`

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

Definition

- The actual `struct` contents and implementation are hidden

Listing 2: opaque.c

```
1  #include "opaque.h"
2
3  struct opaque_s
4  {
5    char* name;
6  };
7
8  char const* opaque_get_name (opaque_s* o)
9  {
10   return o->name;
11 }
```

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

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

```
1 42 0
```

```
1 warning: dereferencing type-punned pointer
    ↪ will break strict-aliasing rules
    ↪ [-Wstrict-aliasing]
```