struct types,

Programação (L.EIC009)

Eduardo R. B. Marques, DCC/FCUP

Outline

- `struct` types
  - declaration
  - struct variables and initialization
  - field access
  - use of references vs call-by-value/return-by-value
  - const references
  - memory layout, padding issues

Declaration

A `struct` type is a type for a sequence of member fields, each with his own type.

General declaration:

```
struct name_of_struct_type {
  type_1 member_1;
  type_2 member_2;
  ...
  type_n member_n;
};
```

If declared in this manner, `name_of_struct_type` can be used as a type for variables:

```
name_of_struct_type var;
```

Example

A type for a 2d-coordinate with `double` values for `x` and `y`:

```
struct point2d {
  double x;
  double y;
};
```

- Type name: `point2d`
- Member fields: `x` and `y`, both of type `double`

Another example

Representation of the time of day:

```
struct time_of_day {
  unsigned char h;
  unsigned char m;
};
```

Interval with an associated `int` id, and `start` and `end` times:

```
struct interval {
  int id;
  time_of_day start;
  time_of_day end;
};
```

Initialisation

```
// no initialisation, fields contain arbitrary values
point2d p1;
interval il1;
// initialisation (according to member order)
point2d p2 { 1.2, 3.4 };
interval il2 { 1, { 17, 30 }, {18, 30} };
// initialisation by copy
point2d p3 = p2;
interval il3 { il2 }; //  alternative syntax
```

Field access

For a variable `v` of `struct` type `s` that has a member field `mf`, we can write `v.mf` to access field `mf` in `v`.

Examples:

```
point2d a { 1.5, 2.5};
point2d b { a.y, a.x }; // --> { 2.5, 1.5}
a.x = b.x + b.y; // a.x is assigned 4 = 2.5 + 1.5

interval il { 1, { 17, 30 }, {18, 30} };
cout << (int) il.start.h << ':' << (int) il.start.m
     << " --> "
     << (int) il.end.h << ':' << (int) il.end.m
     << "\n"; // --> 17:30 --> 18:30
```

Functions

`struct` parameters can be used in functions.

```
point2d sum(point2d a, point2d b) {
  point2d r;
  r.x = a.x + b.x;
  r.y = a.y + b.y;
  return r;
}
. . .
point2d r, a = ... , b = ...;
r = sum(a, b);
```

Alternative formulation:

```
point2d sum(point2d a, point2d b) {
  return { a.x + b.x, a.y + b.y };
}
```

Functions (cont.)

Passing `struct` arguments by value or returning `struct` values often results in overhead due to value copying, unless the `struct` type at stake has a small size.

For efficiency, when the size of the `struct` type exceeds the computer's word size (8 bytes in 64-bit machines) it is more efficient to have `struct` parameters declared as references. In particular, `const` references can be used for read-only parameters.

```
void sum(const point2d& a, const point2d& b, point2d& r) {
  r.x = a.x + b.x;
  r.y = a.y + b.y;
}
. . .
point2d r, a = ... , b = ...;
sum(a, b, r);
```

const references

You can not modify the data associated to a `const` reference!

```
void sum(const point2d& a, const point2d& b, point2d& r) {
    a.x += b.x; // Not allowed - cannot write to a!
    b.y += a.y; // Not allowed - cannot write to b!
    r.x = a.x;
    r.y = b.y;
}
```

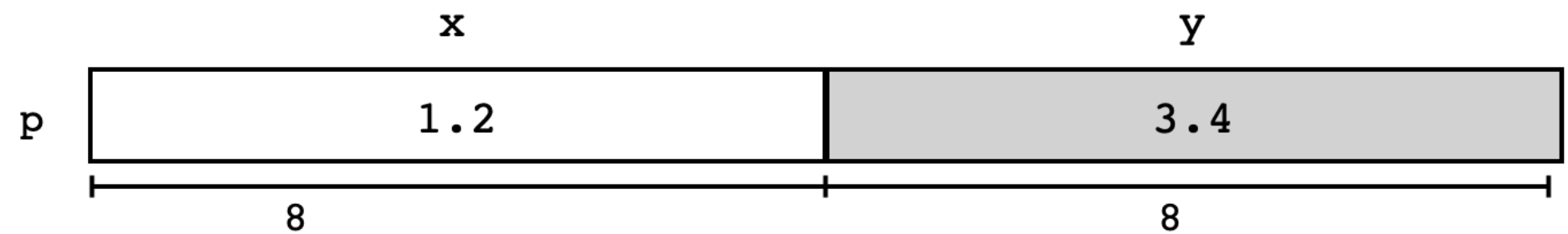
```
error: cannot assign to variable 'a' with const-qualified
       type 'const point2d &'
error: cannot assign to variable 'b' with const-qualified
       type 'const point2d &'
```

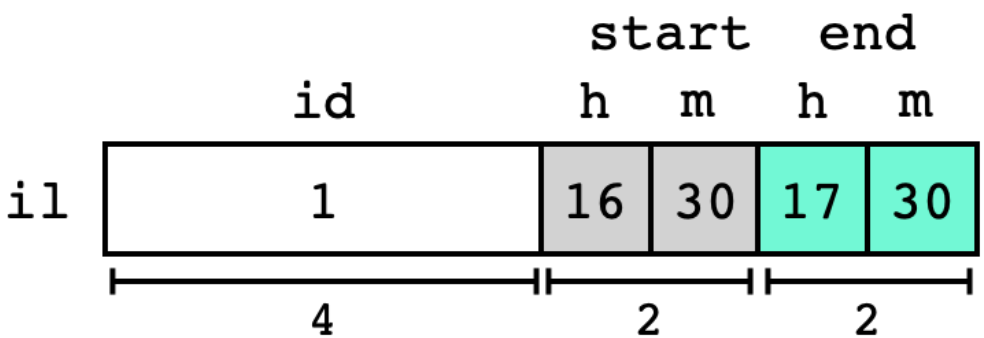
Memory representation

Struct values are stored in memory as a sequence of member field values.

```
point2d p { 1.2, 3.4 };
interval il { 1, { 17, 30 }, {18, 30} };
cout << sizeof(p) << ' ' << sizeof(il) << '\n';
```

```
16 8
```





Memory representation (cont.)

The memory size of a `struct` value can however be higher than the sum of the sizes of member fields due to memory alignment requirements.

```
struct a {
  int i;    // 4
  short s1; // 2
  short s2; // 2
};
struct b {
  short s1; // 2
  int i;    // 4
  short s2; // 2
};
cout << sizeof(a) << ' ' << sizeof(b) << '\n';
```

```
8 12
```

Why does this happen?

Memory representation (cont.)

In the case of

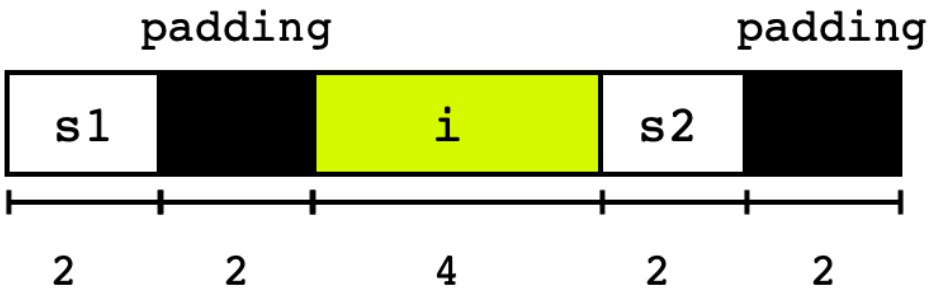
```
struct b {
  short s1; // 2
            // 2 bytes of padding
  int i;    // 4
  short s2; // 2
            // 2 bytes of padding
};
```

field `i` is required to be aligned with an address that is a multiple of 4, while fields `s1` and `s2` just require addresses aligned with a multiple of 2.

An unused memory fragment of 2 bytes, called **padding**, before or after `s1` is used by the compiler to enforce proper memory alignment of `i`. A padding of 2 bytes is also required before or after `s2` to ensure that a `b` value is also aligned with a memory address that is a multiple of 4.

Memory representation (cont.)

```
struct b {
  short s1; // 2
            // 2 bytes of padding
  int i;    // 4
  short s2; // 2
            // 2 bytes of padding
};
```



Further reference:

- About Data Alignment (Microsoft Developer Network)
- cppreference.com - Object - see “Alignment” section