What Is A Record?

- Recall:
  - Records: If $I_1, \dots, I_k$ are distinct identifiers, and $T_1, \dots, T_k$ are type expressions, then $\mathbf{RECORD}(I_1\colon\ T_1, \dots, I_k\colon\ T_k)$ is a type expression denoting a record type with k named fields.
- In other words:
  - A record is an aggregation of potentially heterogeneous (sub-)objects into a single unit.
  - The aggregated objects are called the fields of the record, and are referenced by name.

What Is A Record?

- Recall:
  - Records: If $I_1, \ldots, I_k$ are distinct identifiers, and $T_1, \ldots, T_k$ are type expressions, then $\mathbf{RECORD}(I_1\colon T_1, \ldots, I_k\colon T_k)$ is a type expression denoting a record type with k named fields.
- In other words:
  - A record is an aggregation of potentially heterogeneous (sub-)objects into a single unit.
  - The aggregated objects are called the fields of the record, and are referenced by name.
- [C] Record types are called structure types (written `struct`).
  - Can be tagged (`struct T {...} x;`) or untagged (`struct {...} x;`).
  - Often used with typedef (`typedef struct {...} T;`) to produce new type names for structure types.

What Is A Record?

- Recall:
  - Records: If $I_1, \ldots, I_k$ are distinct identifiers, and $T_1, \ldots, T_k$ are type expressions, then $\mathbf{RECORD}(I_1\colon T_1, \ldots, I_k\colon T_k)$ is a type expression denoting a record type with k named fields.
- In other words:
  - A record is an aggregation of potentially heterogeneous (sub-)objects into a single unit.
  - The aggregated objects are called the fields of the record, and are referenced by name.
- [C] Record types are called structure types (written `struct`).
  - Can be tagged (`struct T {...} x;`) or untagged (`struct {...} x;`).
  - Often used with typedef (`typedef struct {...} T;`) to produce new type names for structure types.

```
typedef struct point {
    int x, y;
} Point;

typedef struct {
    struct point start;
    Point end;
} Line;

Point p, *pptr;
Line ln[5];
```

Run-Time Representation of Records

- Assume that we have the following two functions available.
  - $size$: $\text{Type} \rightarrow \textbf{int}$, giving the number of contiguous bytes of memory that a variable of a given type occupies at run-time.
  - $addr$: $\text{Identifier} \rightarrow \textbf{int}$, giving the starting address in memory of this chunk for a given identifier.

Run-Time Representation of Records

- Assume that we have the following two functions available.
  - $size$: Type $\rightarrow$ **int**, giving the number of contiguous bytes of memory that a variable of a given type occupies at run-time.
  - $addr$: Identifier $\rightarrow$ **int**, giving the starting address in memory of this chunk for a given identifier.
- Then, variable x of type T occupies memory bytes $addr(x)$ through $addr(x) + size(x) - 1$.

Run-Time Representation of Records

- Assume that we have the following two functions available.
  - $size$: Type $\rightarrow$ **int**, giving the number of contiguous bytes of memory that a variable of a given type occupies at run-time.
  - $addr$: Identifier $\rightarrow$ **int**, giving the starting address in memory of this chunk for a given identifier.
- Then, variable x of type T occupies memory bytes $addr(x)$ through $addr(x) + size(x) - 1$.
- If T is a record type, the language definition specifies a mapping of each field f (of type τ_f)of the record type to offsets δ_f, such that $addr(x.f) = addr(x) + \delta_f$.

Run-Time Representation of Records

- Assume that we have the following two functions available.
  - $size$: Type $\rightarrow$ **int**, giving the number of contiguous bytes of memory that a variable of a given type occupies at run-time.
  - $addr$: Identifier $\rightarrow$ **int**, giving the starting address in memory of this chunk for a given identifier.
- Then, variable x of type T occupies memory bytes $addr(x)$ through $addr(x) + size(x) - 1$.
- If T is a record type, the language definition specifies a mapping of each field f (of type τ_f)of the record type to offsets δ_f, such that $addr(x.f) = addr(x) + \delta_f$.
- And we will have $size(T) \geq \sum_f size(\tau_f)$.
  - I.e., each field will be contiguous in memory, but there may be gaps ("padding") between successive fields.

Operations on Record Types

- We have a single operation defined on a record type: accessing a field.
- Given a variable x of a record type T, what does $x.f$ mean?
  - Depends on context: rvalue or lvalue.

Operations on Record Types

- We have a single operation defined on a record type: accessing a field.
- Given a variable x of a record type T, what does $x.f$ mean?
  - Depends on context: rvalue or lvalue.

- Case 1: Rvalue (i.e., a reference `...x.f...` within an expression).
  - Returns a value of type τ_f from the bit pattern in memory bytes $addr(x) + \delta(f)$ through $addr(x) + \delta(f) + size(\tau_f) - 1$.

Operations on Record Types

- We have a single operation defined on a record type: accessing a field.
- Given a variable x of a record type T, what does $x.f$ mean?
  - Depends on context: rvalue or lvalue.

- Case 1: Rvalue (i.e., a reference `...x.f...` within an expression).
  - Returns a value of type τ_f from the bit pattern in memory bytes $addr(x) + \delta(f)$ through $addr(x) + \delta(f) + size(\tau_f) - 1$.
- Case 2: Lvalue (i.e., an update `x.f = ...` within an assignment).
  - Memory bytes $addr(x) + \delta(f)$ through $addr(x) + \delta(f) + size(\tau_f) - 1$ with the type-appropriate bit-pattern representation of the rvalue of the assignment statement.

Record Layout and Data Alignment

- Two nagging questions.
  1. Why do we have $size(T) \geq \sum_f size(\tau_f)$ rather than $size(T) = \sum_f size(\tau_f)$?
  2. Why does the C standard insist that "For two structures, corresponding members shall be declared in the same order." [C17 ballot, §6.2.7]?

Record Layout and Data Alignment

- Two nagging questions.
  1. Why do we have $size(T) \geq \sum_f size(\tau_f)$ rather than $size(T) = \sum_f size(\tau_f)$?
  2. Why does the C standard insist that "For two structures, corresponding members shall be declared in the same order." [C17 ballot, §6.2.7]?
- Both questions have the same answer: data alignment.

Record Layout and Data Alignment

- Two nagging questions.
    1. Why do we have $size(T) \geq \sum_f size(\tau_f)$ rather than $size(T) = \sum_f size(\tau_f)$?
    2. Why does the C standard insist that "For two structures, corresponding members shall be declared in the same order." [C17 ballot, §6.2.7]?
- Both questions have the same answer: data alignment.
- What is data alignment?
    - Computers often impose restrictions on the valid/desired placement of data objects in memory (i.e., $addr(x)$) in order to simplify high-performance memory design. These restrictions are called data alignment rules.
    - Such restrictions are not particularly onerous, because they can be easily handled by the compiler and the run-time system.
    - Reality check: The x64 architecture will work correctly regardless of data alignment (except for some SSE instructions), *but performance may suffer*.

Data Alignment in C

- A variable x of a basic data type of size K bytes and located at memory address μ_x is said to be aligned iff $\mu_x = 0 \ \mathbf{mod}\ K$.

Data Alignment in C

- A variable x of a basic data type of size K bytes and located at memory address μ_x is said to be aligned iff $\mu_x = 0 \bmod K$.
- A variable x of a derived data type T located at memory address μ_x iff the following two conditions hold:
  - [The sub-object rule] Every sub-object of this type is (recursively) aligned.
  - [The array rule] Every element of an object y of type “array of T” is aligned.

Data Alignment in C

- A variable x of a basic data type of size K bytes and located at memory address μ_x is said to be aligned iff $\mu_x = 0 \textbf{ mod } K$.
- A variable x of a derived data type T located at memory address μ_x iff the following two conditions hold:
  - [The sub-object rule] Every sub-object of this type is (recursively) aligned.
  - [The array rule] Every element of an object y of type "array of T" is aligned.

```
typedef struct point {
    int x, y;
} Point;

typedef struct {
    struct point start;
    Point end;
} Line;

Line ln[5];
```

Data Alignment in C

- A variable x of a basic data type of size K bytes and located at memory address μ_x is said to be aligned iff $\mu_x = 0 \mathbf{\ mod\ } K$.
- A variable x of a derived data type T located at memory address μ_x iff the following two conditions hold:
  - [The sub-object rule] Every sub-object of this type is (recursively) aligned.
  - [The array rule] Every element of an object y of type "array of T" is aligned.

```
struct S1 {
  int i;
  char c;
  int j;
};
```

```
struct S2 {
  int i;
  int j;
  char c;
};
```

```
typedef struct point {
    int x, y;
} Point;

typedef struct {
    struct point start;
    Point end;
} Line;

Line ln[5];
```

Function-Valued Fields in Records

- Nothing prevents us from adding function pointers as fields in records.

Function-Valued Fields in Records

- Nothing prevents us from adding function pointers as fields in records.

```
typedef struct point {
    int x, y;
    double (*d)(double, double);
} Point;

typedef struct {
    struct point start;
    Point end;
    double (*l)(Point, Point);
} Line;
```

Function-Valued Fields in Records

- Nothing prevents us from adding function pointers as fields in records.

```
typedef struct point {
    int x, y;
    double (*d)(double, double);
} Point;

typedef struct {
    struct point start;
    Point end;
    double (*l)(Point, Point);
} Line;
```

```
#include <math.h>
double L2_dist(double x, double y) {
      return sqrt(x*x+y*y);
}
double L1_dist(double x, double y) {
      return fabs(x) + fabs(y);
}
double line_len(Point p, Point q) {
      return L2_dist(L1_dist(p.x, p.y),
                     L1_dist(q.x, q.y));
}
```

Function-Valued Fields in Records

- Nothing prevents us from adding function pointers as fields in records.

```
typedef struct point {
    int x, y;
    double (*d)(double, double);
} Point;

typedef struct {
    struct point start;
    Point end;
    double (*l)(Point, Point);
} Line;
```

```
#include <math.h>
double L2_dist(double x, double y) {
return sqrt(x*x+y*y);
}
double L1_dist(double x, double y) {
      return fabs(x) + fabs(y);
}
double line_len(Point p, Point q) {
  return L2_dist(L1_dist(p.x, p.y),
                 L1_dist(q.x, q.y));
}
```

```
Point p1, p2;
Line l;
p1.x = 10; p1.y = 20; p1.d = L2_dist;
p2.x = 30; p2.y = 40; p2.d = L1_dist;
l.start = p1; l.end = p2; l.len = line_len;
```

Function-Valued Fields in Records

- Nothing prevents us from adding function pointers as fields in records.
- What is this beginning to resemble?

```
typedef struct point {
    int x, y;
    double (*d)(double, double);
} Point;

typedef struct {
    struct point start;
    Point end;
    double (*l)(Point, Point);
} Line;
```

```
#include <math.h>
double L2_dist(double x, double y) {
return sqrt(x*x+y*y);
}
double L1_dist(double x, double y) {
      return fabs(x) + fabs(y);
}
double line_len(Point p, Point q) {
      return L2_dist(L1_dist(p.x, p.y),
                     L1_dist(q.x, q.y));
}
```

```
Point p1, p2;
Line l;
p1.x = 10; p1.y = 20; p1.d = L2_dist;
p2.x = 30; p2.y = 40; p2.d = L1_dist;
l.start = p1; l.end = p2; l.len = line_len;
```