Compilers Course
Lecture 14: Data Layout

For every type (primitive or user-declared) the compiler must know:

- Its size in bytes
- Its alignment: if a type has alignment N, then every instance of that type in memory must have an address A = K*N (for some integer K)

Scalar (primitive) types

- char, short, int, long, enumerations, pointers, float, double
- Occupies N=2^K consecutive bytes: [b0,b1,...,bn-1]
  so the size is N; typical values are:
  sizeof(char) == 1
  sizeof(short) == 2
  sizeof(float) == 4
  sizeof(double) == 8
  on 32-bit machines: sizeof(int) == sizeof(pointer) == 4
  on 64-bit machines: sizeof(int) == 4, sizeof(pointer) == 8
  sizeof(long) == sizeof(pointer) except on Win64 where it is sizeof(int)
- The alignment for primitive types is often also N
  misaligned loads/stores may cause exceptions or slow execution

Byte order

- A 32-bit integer requires 4 bytes [b0,b1,b2,b3] in memory
- A value 0x11223344 is usually formatted in one of the following two ways:
  - Little-endian order: [0x11,0x22,0x33,0x44]
  - Big-endian order: [0x44,0x33,0x22,0x11]
- Byte order does not matter as long as integers are accessed using the machine's natural integer-sized load/store instructions, so *compilers usually do not care about byte order*
- Programmers sometimes write sloppy/careless code that is sensitive to byte order, for instance in binary data conversion procedures

Arrays

- Element_type A[N]
- Sequence of N identically-shaped elements:

A[0]	A[1]	…	A[N-1]

- Size = N * size of the element type
- Alignment = alignment of the element type:
  if A[i] is aligned, then so will A[i+1] be

Note that the presence of arrays requires that every single type has a size that is a multiple of its alignment.

Records (structs)

- Struct S { $type_1$ $field_1$; $type_2$ $field_2$; ...; $type_N$ $field_N$; };
- Sequence of N differently-shaped elements:

$field_1$	$field_2$	...	$filed_N$

- The elements are usually stored in the same order as declared
- Alignment = MAX(alignment for any field type)
- Between two fields there may be "internal padding" of unused bytes to ensure alignment of the second field
- After the last field there may be "tail padding" of unused bytes to ensure the size is a multiple of the alignment

Example: `struct S { int i; double d; int j; }`

- Alignment will be 8 because alignof(double) == 8.
- i will be at offset 0
- There will be 4 bytes of internal padding at offset 4
- d will be at offset 8
- j will be at offset 16
- There will be 4 bytes of tail padding at offset 20
- The total size is 24

0:	i	4 bytes
4:	pad	4 bytes
8:	d	8 bytes
16:	j	4 bytes
20:	pad	4 bytes

Unions

- union u { $type_1$ $field_1$; $type_2$ $field_2$; ...; $type_N$ $field_N$; };
- The fields overlap, all fields start at offset 0
- Alignment = MAX(alignment for any field)
- Size = MAX(size of any field) + tail padding to make the size a multiple of the alignment