that tight clusters of repeatedly accessed objects are always acted upon by only a single thread at a time, e.g., using local (arena) memory allocators; see *Appendix — Cache lines; L1, L2, and L3 cache; pages; and virtual memory* on page 181.

See Also

- "**alignof**" (§2.1, p. 184) inspects the alignment of a given type.

Appendix

Natural alignment

Many micro-architectures are optimized for working with data that has **natural alignment**; i.e., objects reside on an address boundary that divides their size rounded up to the nearest power of two. ~~With the additional~~ restriction that no padding is allowed between C++ array elements, the alignment requirements of fundamental types are often equal to their respective size on most platforms:

```
char        c;  // size 1;  alignment 1;  boundaries: 0x00, 0x01, 0x02, ...
short       s;  // size 2;  alignment 2:  boundaries: 0x00, 0x02, 0x04, ...
int         i;  // size 4;  alignment 4;  boundaries: 0x00, 0x04, 0x08, ...
float       f;  // size 4;  alignment 4;  boundaries: 0x00, 0x04, 0x08, ...
double      d;  // size 8;  alignment 8;  boundaries: 0x00, 0x08, 0x10, ...
long double l;  // size 16; alignment 16; boundaries: 0x00, 0x10, 0x20, ...
```

The alignment requirement of an array of objects is the same as that of its elements:

```
char  arrC[4];  // size 4; alignment 1
short arrS[4];  // size 8; alignment 2
```

For user-defined types, compilers compute the alignment and add appropriate padding between the data members and after the last one, such that all alignment requirements of the data members are satisfied and no padding would be required should an array of the type be created. Typically, the resulting alignment requirement of a UDT is the same as that of the most strictly aligned nonstatic data member:

```
struct S0
{
    char a;     // size 1; alignment 1
    char b;     // size 1; alignment 1
    int  c;     // size 4; alignment 4
};              // size 8; alignment 4
```