BLayout Documentation

API

Types

```
typedef size_t blsize;

struct blayout {
    blsize nmemb;
    blsize size;
    blsize align;
};
```

- `blsize` is the API's size type. It's `size_t` by default. You may change this type by modifying BLayout's header. A `signed` type is also valid. You'd have to change `BL_SIZEMAX` accordingly (see below).
- `blayout` describes a single memory allocation request for an object:
    - `nmemb` is the number of elements (like `calloc()`'s first argument),
    - `size` is the size (in bytes) of each element/type (like `calloc()`'s second argument),
    - `align` is the alignment[1] of the object's type

Constants

```
#define BL_SIZEMAX   SIZE_MAX
#define BL_ALIGNMENT alignof(max_align_t)
```

- `BL_SIZEMAX` is the equivalent to `size_t`'s `SIZE_MAX` and is equal to that by default. You may override this, but the header assumes that it's **greater** than `0`.
- `BL_ALIGNMENT` is never used internally. It's equal to the maximum alignment among C's scalar types. It's provided as a convenience when calling `blcalc()` (see below). This, too, can be overridden.

*Note: To override these, either modify BLayout's header or `#define` them **before** including `blayout.h`.*

Macros

```
#define BL_API    static
#define BL_ASSERT assert
#define BL_INLINE inline
#define BL_DEBUG  0
```

- You may change `BL_API` to `extern` if you want to put the functions in a seperate translation unit. Keep in mind that you'd have to copy-paste the signatures.

[1] **alignment** is *always assumed to be valid*: (1) it denotes *byte* boundaries and (2) is a power of `2`.

- BLayout can use assertions through the `BL_ASSERT` macro to enforce API contracts and prevent footguns. You can override this macro if you use a custom `assert()` function. See `BL_DEBUG` below if you want to disable assertions.
- Every function is `inline` (C99 semantics) through the `BL_INLINE` macro. This is so that you can workaround C's deficiencies, if you so wish.
- `BL_DEBUG` takes three possible values:
  - `0`, where BLayout will use **no** assertions (see above) and, in addition, will take advantage of compiler-specific optimization hints (e.g. `attribute(nonnull(...))`). This is the default.
  - `1`, where BLayout will use **some** assertions **and** optimization hints.
  - `2`, where BLayout will use **all** assertions **and no** optimization hints.

Functions

```
BL_API blsize blcalc(blsize align,
                     ptrdiff_t offs,
                     blsize n,
                     const struct blayout *lays,
                     blsize prev_size);

BL_API void *blnext(void *ptr, blsize curr_size, blsize next_align);
BL_API void *blprev(void *ptr, blsize prev_size, blsize prev_align);

BL_API blsize blsizeof(const struct blayout *l);
```

- `blcalc()` returns the minimum size needed to contiguously allocate multiple objects. The function assumes that all arguments are valid and within bounds. If wrap-around is detected when computing the size, `0` is returned instead, indicating error.
  - `align` is the default alignment your allocator supports. In case you already have an allocated buffer, pass the buffer's alignment. `BL_ALIGNMENT` should work with `malloc()` and with any buffer allocated by it.
  - `offs` is used in case you already have a buffer and want to allocate starting from an offset into that buffer. Pass `0` otherwise.
  - `n` is the number of layouts. Should be **greater** than `0`.
  - `lays` is an array of length `n` containing layouts,
  - `prev_size` is used to chain multiple `blcalc()` calls. When first invoking, `0` must be passed, otherwise the result of the previous `blcalc()` call must be passed, assuming the call succeeded and a **non**-`0` value was returned. `align` and `offs` must not change across any chained calls.
- `blnext()` allocates the next object in a **left-to-right** manner, where:
  - `ptr` is a pointer to the current allocated object. When first invoking, pass a pointer to your buffer. It's assumed to **not** be `NULL` and thus the function doesn't check for this.
  - `curr_size` is the size of the current object (see `blsizeof()`) and is assumed to be valid. When first invoking, pass `0`.
  - `next_align` is the alignment of the next object's type and is assumed to be valid. When first invoking, pass the alignment of the **first** object's type.
- `blprev()` is like `blnext()`, but allocates and returns the previous object, in a **right-to-left** manner. That means you should allocate in **reverse** order, starting with **last** object.
  - `ptr` is a pointer to the current allocated object. When first invoking, pass a pointer to the **end** of your buffer. It's assumed to **not** be `NULL` and thus the function doesn't check for this.
  - `prev_size` is the size of the previous object (see `blsizeof()`) and is assumed to be valid. When first invoking, pass the size of the **last** object.
  - `prev_align` is the alignment of the previous object's type and is assumed to be valid. When first invoking pass the alignment of the **last** object's type.
- `blsizeof()` returns the total size (in bytes) of an object described by its layout. Effectively, it multiplies `blayout.nmemb` with `blayout.size`. It's provided as a convenience.
  - `l` is the pointer to the aforementioned layout.
  1. *Caveat: Padding due to alignment is **not** taken into account.*
  2. *Caveat: Potential integer overflow is **not** checked. The layout is assumed to be correct. `blcalc()` already checks for this.*

Usage

1. `blcalc()` with `blnext()`
2. `blprev()`
3. `blnext()` vs. `blprev()`

`blcalc()` with `blnext()`

```
/*
 * Request:
 *    I. One integer, naturally aligned.
 *   II. Two floats, naturally aligned.
 *
 * The order of the `lays` array is important! `blcalc()` takes the order into
 * account when computing its result. And it does that for the simple reason
 * that the whole point of this header is to allow the programmer to lay out
 * their objects in memory exactly how they want. Hence, we preserve the order,
 * because it might be important, we wouldn't know. If the order is _not_
 * important to _you_, this detail doesn't impair you.
 *
 * This order also defines how `blnext()` (and `blprev()`; see below) should be
 * called. The functions don't check for this, the burden, unfortunately, falls
 * onto the programmer.
 */
const struct blayout lays[] = {{1, sizeof(int),   alignof(int)  },
                               {2, sizeof(float), alignof(float)}};

size_t size = blcalc(BL_ALIGNMENT,  /* Going to use the default alignment. */
                     0,             /* Allocating from the `0`th position. */
                     2,             /* The number of layouts. */
                     lays,          /* The array of layouts describing our objects. */
                     0);            /* We aren't chaining `blcalc()` calls. */
if (size == 0) {
    fprintf(stderr, "blcalc() error\n");
    return 1;
}

void *buf = malloc(size);
if (buf == NULL) {
    fprintf(stderr, "malloc() error\n");
    return 1;
}

/* The first object. */
int *i = blnext(buf,             /* First allocation: pass the buffer. */
                0,               /* First allocation: pass `0`. */
                lays[0].align);  /* Pass the alignment of the first object's type (`int`). */
assert(i != NULL);  /* Guaranteed to _not_ be `NULL`. */

/* Continue with the next object... */
float *f = blnext(i,                    /* Pass the current allocated object. */
                  blsizeof(&lays[0]),   /* Pass the size of the current allocated object. */
                  lays[1].align);       /* Pass the alignment of the next object's type (`float`). */
assert(f != NULL);  /* Likewise. */

/* Use `i` and `f` normally. */
*i = 42;
f[0] = 2.71;
f[1] = 3.14;
printf("*i=%d f[0]=%f f[1]=%f\n", *i, f[0], f[1]);

/* Cleanup. */
free(buf);  /* `i` and `f` are guaranteed to be cleaned up with a _single_ `free()`. */
return 0;

/*
 * Alternatively, if the alignment of the _first_ object's type is
 * _less-or-equal_ to your buffer's alignment, then the pointer to the first
 * allocated object is equivalent to a pointer to the buffer.
 *
 * In our case:
 *
 *  1. The first object's type has alignment `alignof(int)`.
 *  2. `BL_ALIGNMENT` is the default alignment of `malloc()` (which we also
 *     used when calling `blcalc()`).
 *  3. Thus, the buffer returned to us by `malloc()` (`buf`) has that alignment.
 *  4. `malloc()` must, as mandated by the C standard, be able to return a
 *     suitably aligned pointer for _every_ naturally aligned type. That
 *     includes our case: `int`.
 *  5. Since `buf` has alignment `BL_ALIGNMENT` and that alignment _must_ be
 *     suitable for `int`, we can conclude that `BL_ALIGNMENT >= alignof(int)`
 *     or, equivalently, `alignof(int) <= BL_ALIGNMENT`.
 */
```

```
/*
 * Note that any type can be over-aligned. This is fine, because a greater
 * alignment guarantees natural alignment. Or, in the words of the standard[1]:
 *
 *   In general, the concept "correctly aligned" is transitive: if a pointer
 *   to type A is correctly aligned for a pointer to type B, which in turn is
 *   correctly aligned for a pointer to type C, then a pointer to type A is
 *   correctly aligned for a pointer to type C.
 *
 * [1]: <https://port70.net/~nsz/c/c11/n1570.html#note68>
 */
static_assert(alignof(int) <= BL_ALIGNMENT, "incorrect alignment");  /* Just to be sure. */
free(i);  /* Same as `free(buf);` _in our case_. See the above comments why and when this holds
    true. */
return 0;

/*
 * The above observation also implies that we could skip the first call to
 * `blnext()` and allocate the first object (`i`) like so:
 */
int *i = buf;  /* Okay, as long as `alignof(int) <= BL_ALIGNMENT`. */

/* With the rest of the code being identical... */
```

`blprev()`

Usage of `blprev()` is similar to the usage of `blnext()` with these notable differences:

1. You must allocate in **reverse** order.
2. In order to cleanup safely, you **must** retain a pointer to your buffer.

Let's see with a similar example:

```
/*
 * ... Same layouts and boilerplate as the `blnext()` example...
 *
 * Assume `buf` of size `size` has already been allocated as above.
 */
size_t size;
void *buf;

/*
 * Allocate in **reverse** order, starting with the **last** object. `blprev()`
 * works in a _right-to-left_ manner; we have to pass the _end_ of our buffer
 * in the first allocation.
 */
void *end = (char *)buf + size;
float *f = blprev(end,                 /* First allocation: pass the end of the buffer. */
                  blsizeof(&lays[1]),  /* First allocation: pass the size of the last object. */
                  lays[1].align);      /* First allocation: pass the alignment of the last object's
                      type. */
assert(f != NULL);  /* Always true, same as before. */

/* Continue with the previous object... */
int *i = blprev(f,                   /* Pass the current allocated object. */
                blsizeof(&lays[0]),  /* Pass the size of the previous object. */
                lays[0].align);      /* Pass the alignment of the previous object's type. */
assert(i != NULL);  /* Likewise. */

*i = 1337;
f[0] = 1.41;
f[1] = 1.61;
printf("*i=%d f[0]=%f f[1]=%f\n", *i, f[0], f[1]);

/*
 * Cleanup!
 *
 * NOTE: You can _not_ pass `i` or `f` in the case of `blprev()`! You must pass
 *       the pointer returned to you by `malloc()`!
 */
free(buf);
return 0;

//free(i);  /* XXX: Don't do this _ever_! */
//free(f);  /* XXX: Or this either! */
```

`blnext()` vs. `blprev()`

When should you use the one or the other? Given the limitations of `blprev()`, shouldn't you always use `blnext()`? It depends. Firstly, you indeed **can only use one**[2] of the two to allocate from a single memory region. Given that, which to pick?

Let's assume your layouts array is `{{1, 1, 1}, {1, 2, 2}}`. Meaning:

1. One (`1`) type of size `1` (bytes) at an `1`-byte alignment boundary, and
2. One (`1`) type of size `2` (bytes) at an `2`-byte alignment boundary.

Assuming your buffer is `16`-byte aligned, `blcalc()` (correctly) returns `4` for the above array.

If your buffer sits at address `16` then, using `blnext()`, the bytes would be laid out like this:

```
Address | 16 17 18 19
Bytes   | X0    Y0 Y1
```

But, if you were to use `blprev()`, allocating from the end of the buffer, they'd be laid out like this:

```
Address | 16 17 18 19
Bytes   |    X0 Y0 Y1
```

Where `Xi` is the `i`th byte of object `X`, respectively `Y`.

Thus, the two methods give different results. This example also makes clear why you need to retain a pointer to your buffer in the case of `blprev()`: `X0` doesn't sit at the (original) address `16`!

Always use `blnext()`, unless you desire the special properties of `blprev()` and the limitations don't affect you. Here's two scenarios where that could be true:

- You always carry around a pointer to your buffer, so using `blprev()` has no extra burden. *Note that `blprev()` is 2-3 machine instructions shorter than `blnext()`, under `x86_64-sysv` and also depending on compiler and optimization options (Related).*
- You depend on the layout `blprev()` gives. This happens when giving a "header" to a "payload", just like `malloc()` does.

```c
struct header {
    int id;
};

struct payload {
    size_t size;
    unsigned char data[];  /* Flexible array member:
        <https://gustedt.wordpress.com/2011/03/14/flexible-array-member/> */
};

struct payload *new_payload(my_ctx *ctx, size_t payload_size)
{
    const struct blayout lays[] = {
        {1, sizeof(struct header), alignof(struct header)},
        {1, offsetof(struct payload, data) + payload_size, alignof(struct payload)}
    };
    size_t size = blcalc(/* ... */, 0, 2, lays, 0);
    if (size == 0)
        abort();

    void *buf_end = /* ... */;
    struct payload *p = blprev(buf_end, blsizeof(&lays[1]), lays[1].align);
    p->size = payload_size;
    memset(p->data, 0, payload_size);

    struct header *h = blprev(p, blsizeof(&lays[0]), lays[0].align);
    h->id = /* ... */;

    return p;
}

void recycle_payload(my_ctx *ctx, struct payload *p)
{
    struct header *h = blprev(p, sizeof(*h), alignof(struct header));  /* Okay. */
    int id = h->id;
    /* Do something with `id`... */
}
```

[2] Which also means you can't use one function to retrieve objects in reverse order, if you allocated using the other.

The above example **cannot** work with `blnext()`, it **only works with `blprev()`**. Even if you only used `blnext()` to allocate the objects, once you got to retrieve the header in:

```
struct header *h = blprev(p, sizeof(*h), alignof(struct header));  /* I'm sure this is
    perfectly fine, what could possibly go wr */
int id = h->id;  /* Kaboom! */
```

your computer would explode. You can't do this! *Even if it works*, you can't depend on this if you allocate using `blnext()`. Use `blprev()` instead. Remember that the two functions *lay out objects differently*.

LICENSE

```
MIT No Attribution

Copyright 2025 pan <pan_@disroot.org>

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```