P3125R0: Pointer Tagging

Date: 2023-03-12
Audience: SG1
Authors: Hana Dusíková (cpp@hanicka.net)

Abstract

This paper proposes a library based design with "magical functions" to store and retrieve information into bits of pointers which are not significant to the pointer's address, and to inform developers about how many such bits are available.

Motivation

Pointer tagging is not allowed in standard C++ as manipulating pointer bits is UB. Because of this limitation, some advanced data structures are not implementable or sub-optimally implementable.

This technique is an existing practice, is widely used in the field, and standardising it would lower the bar for its safe usage among C++'s users.

Existing usage

ARM64 platform is ignoring 8 high-order bits (which can be used sometimes by MTE, Memory Tagging Extension, for safety purposes), CHERI is using fat-pointers of 128 bit size with upper 64 bits used for safety flags and properties. Hash-Array-Mapped-Trie (HAMT) data structure can use pointer tagging to disambiguate node type. LLVM code base is using pointer tagging in the form of `PointerIntPair` templated type.

Functions for manipulating tags

Getting info about the existing environment

```
template <typename Pointee, size_t Alignment = alignof(Pointee)>
constexpr auto tag_bit_mask() noexcept -> uintptr_t;
```

Return `uintptr_t` value with a mask containing usable bits for tagging marked with ones. Some platforms can provide more bits than others, but low-order n-bits (n = log2(alignment)) are available on all modern platforms. The number of bits can also differ based on compiler settings and enabled sanitizers which are using high-order bits.

Type	Alignment	Minimum number of bits guaranteed on modern platforms (log2(alignment))
`uint8_t *`	1	0
`uint16_t *`	2	1
`uint32_t *`	4	2
`uint64_t *`	8	3
overaligned type	16	4

Users can explicitly set the second template argument (Alignment) to get more guaranteed bits if needed for their over-aligned objects.

High-order bits availability

Not every platform or build configuration (sanitizer builds) can guarantee high-order bits availability. Implementation can be pessimistic and not allow it. But in some fields and environments it would be useful to have a large number of bits available. I personally don't have any strong preference but I would like to not close such a possibility without proper discussion in SG1.

Bit mask or number of bits

Another approach would be to return the number of available bits instead of masks. And let tag/untag functions (next paragraphs) allow to automatically reorder them. But this can be done manually with constexpr and will lead to lost ability to control exact bit position and communicate outside of C++ environment.

If we decide to use only low-order bits then such an interface would make sense.

Converting to / from tagged pointer type

```
template <typename T, size_t Alignment = alignof(T)>
constexpr auto tag_pointer(T * original, uintptr_t value) noexcept
-> tagged_pointer<T, Alignment>;

template <typename T, size_t Alignment = alignof(T)>
constexpr auto untag_pointer(tagged_pointer<T, Alignment> ptr) noexcept -> T *;
```

To ensure design safety, it's necessary to convert the original pointer (which can be `void *`) to a tagged pointer which can't be dereferenced or otherwise manipulated and should retain information about the original pointer type. The `void *` pointer type should be supported so that user code can check for stored information and decide to which pointer type to do the later cast (e.g.,: a tree-like data structure where one bit is used to mark leafs vs internal nodes). The `untag_pointer` function is needed to clean and revert back to the original pointer.

`untag_pointer` should be implemented by BITAND masking internal value by result of `tag_bit_mask` function with the same template arguments as `untag_pointer`.

Function `tag_pointer` should have a precondition that set bits are not outside of the mask obtainable via `tag_bit_mask` function as the resulting pointer would be unrecoverable.

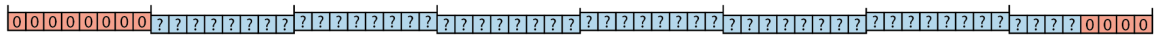
Extracting values from tagged pointer

```
template <typename T, size_t Alignment = alignof(T)>
constexpr auto tag_value(tagged_pointer<T, Alignment> ptr) noexcept -> uintptr_t;
```

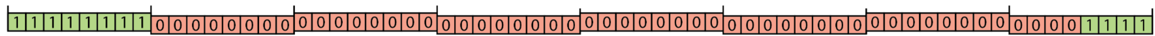
This function takes a tagged pointer and returns the value stored in its unused bits.

Example of pointer, tag value, and tagged pointer

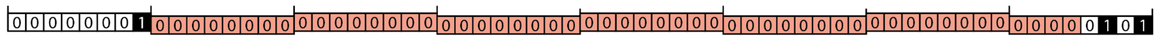
```
Pointer:          0x00?????????????0 (align = 16)
```



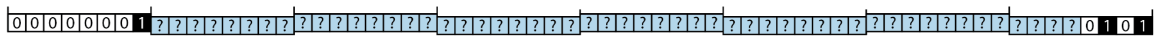
```
Available mask:   0xFF0000000000000F
```



```
Tag value:        0x0100000000000005
```



```
Tagged pointer:   0x01?????????????5
```



Constexpr

Proposed functionality should be constexpr to allow constexpr implementation of advanced data structures (HAMT, trees…) implementation can use hidden storage to store values or allocate a structure with value and original pointer.

Tagged pointer type

There are multiple possibilities of what `tagged_pointer<T, alignment>` should be. The type should be the same size as `void *`, and should be convertible to `void *` (explicitly?) so that it can be passed as a user data in various C style interfaces.

Option 1: void *

`void *` is easily convertible to any other pointer type and can lead to hard-to-diagnose problems (example: conversion of differently aligned pointer to `untag_interface`, value being compared to null, …).

Option 2: original pointer type (T *)

Using the same pointer retains original information about type, but not its alignment. It has similar problems as `void *` and also allows the user to perform pointer arithmetic which shouldn't be allowed on tagged pointers as stored values can contain metadata associated with the target object.

Option 3: special type

A special type avoids the problems with pointer types as stated in previous options, but loses information about the type and alignment.

Option 4: uintptr_t

Same problem as the special type in option 3, but allows user code to observe pointer values which can introduce problems with the memory model as users would be able to construct a pointer out of nothing. It would allow conforming implementation of xor linked list.

Option 5: special templated type (preferred)

This will retain all available information including alignment, original type, and constness. It can be explicitly seen in user code and doesn't need any special behaviour other than to just keep its value inside unobservable. Special type makes implementation for constexpr easier to implement.

Example: Recursing thru a trie-like data structure

```
// take the lowest available bit and use it as a flag
constexpr uintptr_t leaf_node_flag = 1u << countr_zero(tag_bit_mask<const void, 2>());

struct alignas(2) node_t {
 tagged_pointer<const void, 2> subnode[16]; // subnodes
};

struct alignas(2) value_t {
  T value;
}

constexpr auto find_hamt_leaf(const node_t * node, uint32_t hash) -> const value_t * {
  tagged_pointer next = node->subnode[hash & 0b1111u];
  if (tag_value(next) & leaf_node_flag) {
    // we found it!
    return static_cast<const value_t *>(untag_pointer(next));
  } else {
    // tail recursion to next node
    return find_hamt_leaf(static_cast<const node_t *>(untag_pointer(next)), hash >> 4u);
  }
}
```

This example goes through a 16-nary tree and uses 4 bits from the provided hash to find a next node. If there is no collision of hashes a `value_t` node can be next and the pointer contains a tag to store this information. This example is fully constant evaluable as conversion from `tagged_pointer` to original pointer is a constexpr operation and conversion from `void *` to original type at the location is also allowed in constexpr.

Converting bit position in mask to real position

During conversion some people communicate preference to have API “set nth usable bit”. If we allow accessing high-order bits such an operation is not just a right bitshift but a rather complex operation. With proposed mask interface user can emulate such behaviour with constant evaluated value for bit flags:

```
consteval uintptr_t nth_bit_in_mask(uintptr_t mask, unsigned nth) {
    for (; nth > 0; --nth) {
        mask &= mask - 1u;
    }
    return 1ull << countr_zero(mask);
}

// we are considering platform with high-order bits
constexpr uintptr_t flag1 = nth_bit_in_mask(tag_bit_mask<int, 4>(), 0); // low-order
constexpr uintptr_t flag2 = nth_bit_in_mask(tag_bit_mask<int, 4>(), 1); // low-order
constexpr uintptr_t flag3 = nth_bit_in_mask(tag_bit_mask<int, 4>(), 2); // high-order

auto result = tag_pointer(ptr, flag1 | flag3);
```

Special thanks

To Robert C. Seacord, Richard Smith, Chandler Carruth, JF Bastien, and Tony van Eerd.