Small is Beautiful! Techniques to minimize memory footprint

Steven Pigeon

Small is Beautiful!
Techniques to minimize memory footprint

Steven Pigeon
Professeur
Département de mathématiques, informatique et génie
steven_pigeon@uqar.ca

Université du Québec à Rimouski

September 17, 2019

But why?

- Embedded systems
- Mobile computing
- No infinite memory!
- Play nicely with the memory hierarchy and I/O

Compact Data Structures

Today, we'll look at two things:

- Pointers: their use and their representations,
- `enum`s and "snug fit" types.

Pointers, pointers, pointers everywhere!

Many data structures are pointer-rich (lists, trees, HATs):

- A non negligible part of memory is composed of pointers.
- Calls to `new`/`delete` are not cheap.
- May cause fragmentation.

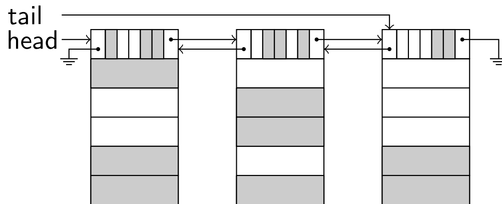
There's a trade-off between how much we'd like to get rid of pointers and

- how much memory is wasted by unused slots;
- how complexity grows by managing space without pointers.

Pointer-Free data structures

- Array Lists (like `std::vector<T>`)

- Paged Lists and Paged Trees

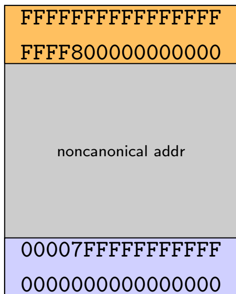


Compressed Pointers

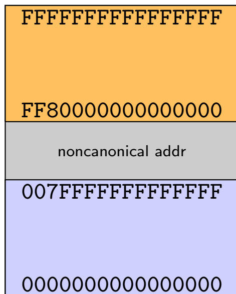
To compress pointers, we'll use two key facts:

- The logical address space is considerably larger than the physically addressable memory,
- Programs usually use only a (very) small portion of their logical address space.

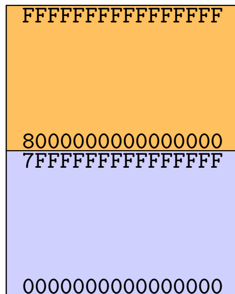
Compressed Pointers: x86_64 Flat Memory Model and Implementations



48 bits Addresses

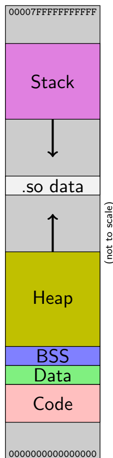


56 bits Addresses



64 bits Addresses

Compressed Pointers: A Program and the x86_64 Flat Memory Model



A typical program has 5–6 regions:

- The code segment,
- The data segment,
- Statically initialized Data (BSS),
- The heap,
- The stack,
- ...possibly shared memory.

Where they are is decided at load time (Address Space Layout Randomization)

Compressed Pointers

Pointers are

- Heap-like (close to the start of the heap)*
- Stack-like (close to the top of the stack)

Since stack and heap are far apart, we can use this to compress pointers differentially.

* Or data-like if we use pointers in the data-segment.

Compressed Pointers: Program Addresses

The Gnu C Library (a.k.a glibc) conveniently provides symbols to locate these regions:

```
extern char // actual type not that important!
    _init, // initialization begins here (before main)
    _start, // program entry point (before main)
    _fini, // final initialization (cleanup)
    etext, // end of 'text' (code)
    __data_start, // beginning of (initialized) data segment
    edata, // end of (initialized) data segment=begin of bss
    __bss_start, // Static default-init. (Block Started by Symbol)
    end; // end of (all) data segments = base of the heap
```

Compressed Pointers: Helpers

The compressed pointers will need an (unsigned) arithmetic type, `uint_pointer_t`, as well as the number of effective address bits (because `uintptr_t` is *optional*!).

```
using uint_pointer_t = uint64_t; // implementation-defined
constexpr int effective_address_bits=47; // implementation-defined
value!
```

Compressed Pointers: Template Class

```
template <typename T, // type to point to
          std::size_t bits=40,
          std::size_t alignment_bits=log2(alignof(T))
          >
class compressed_ptr
  {
    public:
      using uptr_t=typename uint_pointer_t<sizeof(T*)>::type;
    protected:
      static constexpr std::size_t nb_bytes=bytes_from_bits(bits);
      //0x0..07ff..fff+1, implementation-defined
      static constexpr uptr_t top=(uptr_t{1}<<effective_address_bits);
      static constexpr uptr_t direction_bit=(uptr_t{1}<<(bits-1));
      static constexpr uptr_t mask=direction_bit-1;
      uint8_t ptr[nb_bytes]; // compressed ptr
      ...
```

Compressed Pointers: Compressing the pointer

```
void p_compress(T * p)
  {
    uptr_t v{reinterpret_cast<uptr_t>(p)};
    if (v)
      {
        uptr_t bottom=reinterpret_cast<uptr_t>(&end-sizeof(T));
        uptr_t bottom_diff= v-bottom; //udiff(v,bottom);
        uptr_t top_diff = top-v;//udiff(v,top);
        uptr_t diff; // best difference
        uptr_t dbit; // direction bit
        if (bottom_diff < top_diff )
          {
            diff=bottom_diff >> alignment_bits; // heap-like
            dbit=0;
          }
        else
          {
            diff=top_diff >> alignment_bits; // stack-like
            dbit=direction_bit;
          }
        if (diff>mask) throw std::range_error("pointer too big!");
        v=dbit|diff;
      }
    copy<nb_bytes>((char*)&v,(char*)ptr); // assumes little endian
  }
```

Compressed Pointers: Decompressing the pointer

```
T * p_decompress() const
  {
    uptr_t v=0;
    copy<nb_bytes>((char*)ptr,(char*)&v); // assumes little endian
    if (v)
      {
        uptr_t bottom=reinterpret_cast<uptr_t>(&end-sizeof(T));
        if (v & direction_bit)
          v=top-((v & mask)<<alignment_bits); // stack-like
        else
          v=bottom+(v<<alignment_bits); // heap-like

        return reinterpret_cast<T*>(v);
      }
    else return nullptr;
  }
```

Compressed Pointers: Using Compressed Pointers

The class has the needed casts and operators to behave like a normal pointer.

```
for (int i=0;i<10;i++)
  {
    int *t=new int;
    compressed_ptr<int,36> z(t);

    std::cout
        << std::hex << std::setw(16) << t
        << '\t'
        << std::hex << std::setw(16) << z
        << std::endl;

    delete z;
  }
```

Compressed Pointers: Using Compressed Pointers

Compressed pointers...

- Can be tailored depending the available memory,
- Aren't very expensive to (de)compress,
- Have the less strict alignment requirements possible (`uint8_t`),
- Yields pointers 2,3,4 bytes shorter.

They also could be incorporated into `unique_ptr`, `shared_ptr`, ...

Just Enough

We often use larger than necessary types.

One example of this, is **enum**.

- Without underlying type specification, it's **int** (§9.6.1.5)
- With specification, it can be as tight as we want, as long as it allows representation of all enumerators.

```
enum class gizmo:char { rock, paper, scissors };
```

Still wasteful if only a few enumerators, especially with no initializer in the enumerator-definition (§9.6.1.2) as only a few bits would be necessary.

Just Enough and Bit-Fields

What about using bit-fields?

```
class zigzig
  {
    ...

    char choice:2,
         level:3;
         //filler:3

    ...
  };
```

Here, we should use a **constexpr** that finds the number of bits needed to represent the maximum value for an **enum**) and also a template that finds the smallest storage for the bitfield.

Just Enough

```
template <int x> struct __just_enough_uint; // incomplete type
// more types as needed, implementation-defined
template <> struct __just_enough_uint<64> { using type = uint64_t; };
template <> struct __just_enough_uint<32> { using type = uint32_t; };
template <> struct __just_enough_uint<16> { using type = uint16_t; };
template <> struct __just_enough_uint<8>  { using type = uint8_t; };

template <const int x>
using just_enough_uint=
typename
__just_enough_uint<(x>32)?64:((x>16)?32:((x>8)?16:8))>::type;

constexpr std::size_t bits_from_value(std::size_t n)
{ return (n<2)?1:(1+bits_from_value(n/2)); }
```

Just Enough

Template `just_enough_uint<int>`, with `constexpr` function `bits_from_value`, lets us get the smallest integer that accommodates the largest desired value:

```
just_enough_uint<bits_from_value(10000)> x; // likely uint16_t
```

(Smaller integer types will have smaller ("less strict") alignment, so possibly less space lost to alignment.)

Just Enough and Bit-Fields

What about using bit-fields?

```
class zigzig
  {
    ...

    just_enough_uint<5>
         choice:bits_from_value(3),
         level:bits_from_value(4);
    ...
  };
```

...but that's *still* wasteful!

Let's look at `level`. It uses 3 bits, but has only 5 values.

0 000
1 001
2 010
3 011
4 100
5 101 wasted
6 110 wasted
7 111 wasted

On the Efficiency of Bit-Fields

Let's define "Bit Efficiency" as the number of bits needed to encode the value to the number of bits in the bit-field:

$$\frac{\log_2 v}{\log_2 2^n} = \frac{\log_2 v}{n}$$

To maximize efficiency, either we bring v to 2^n or... we choose n so that $2^n \approx v$...

Ideally, we would use *fractions of bits*...

On the Efficiency of Bit-Fields

While we can only allocate entire bits, we should be able to share bits between values, arbitrarily finely.

Say we have 3 fields, with 3, 5, and 11 possible values, a classical bit field would use 2, 3 and 4 bits per field, for a total of 9 bits (fits on `short`).

But if we use fractions of bits, we should be able to use

$$\log_2 3 + \log_2 5 + \log_2 11 = \log_2(3 \cdot 5 \cdot 11) \approx 7.36$$

bits... which now fits on a byte.

Indeed, we have $3 \times 5 \times 11 = 165$ possible combinations!

Fractional bit Arithmetic

If you agree that

$$\texttt{x << 3} = x*2^3 = x*8$$

then you *must* also agree that

$$x*5 = x*2^{\log_2 5} = \texttt{x <<} \log_2 5$$

This will gives us a mean for fractional-bit shifts.

Fractional Bit Arithmetic

The "fractional-bit or" is simply addition, if (and only if) there's no carry.

You will agree that

$$\texttt{x}\,|\,3 = \mathrm{x}+3$$

if (and only if) there's no carry.

We can therefore combine both results:

$$\texttt{x*5+v} = (\texttt{x} \ll \log_2 5)+\mathrm{v}$$

It will "shift" x by $\log_2 5$, just enough to or/add a value v between 0 and 4.

Sub-bitfields

We can generalize the method to encode any number of values.

Let's say again we have fields with 3, 5, and 11 possible values, which we want to set to specific values `v1=2`, `v2=3`, and `v3=9`. We would compute

```
b=((v3)*5+v2)*3+v1; // ((9)*5+3)*3+2=146
```

Extraction, fortunately, isn't very complex:

```
v1=b % 3; // 146%3=2
b/=3;     // 146/3=48
v3=b % 5; // 48%5=3
b/=5;     // 48/5=9
v3=b;     // =9
```

Sub-bitfields: Extracting a single field

Let n_1, n_2, ..., n_b, the number of possible values for each of the b fields. Let

$$p_0 = 1$$

and

$$p_k = \prod_{i=1}^{k-1} n_i$$

The product of all numbers of values that precede the kth field. The value of the kth field is

$$(v \text{ div } p_k) \bmod n_k$$

Sub-bitfields: Setting a single field

To set the kth field with new value c, we compute:

$$(v - (v \bmod p_{k+1})) + c \cdot p_k + (v \bmod p_k)$$

In this formula,

- $(v \bmod p_k)$ is the "masked" lower part, the $k-1$ first fields;
- $(v - (v \bmod p_{k+1}))$ is the "masked" upper part, the value without the k first fields;
- $c \cdot p_k$ is the value c "shifted" in the kth slot.

Sub-bitfields: (Almost) Everything Is Compile-Time

If the numbers of fields and the number of values per field are known at compile-time, we can compute the p_k at compile-time!

We will have `set` and `get` functions that take the field number as a template argument.

Let's say we use a `std::initializer_list<int>` to declare the numbers of values per field (and also the number of fields).

Sub-bitfields: Computing the p_k

```
constexpr int prods(int f,
  const std::initializer_list<int>::const_iterator & t )
    {
      return f ? (*t*prods(f-1,t+1)) : 1;
    }

constexpr int prods(int f,
  const std::initializer_list<int> & l)
    {
      return prods(f,l.begin());
    }
```

Sub-bitfields: Extracting and Setting

```
constexpr int next(int f,
  const std::initializer_list<int>::const_iterator & t )
    {
      return f ? (next(f-1,t+1)) : *t;
    }

constexpr int next(int f,
  const std::initializer_list<int> & l)
    {
      return next(f,l.begin());
    }

template <int f>
  constexpr int t_get(int c,
    const std::initializer_list<int> & n)
      {
        return (c/prods(f,n)) % next(f,n);
      }

template <int f>
  constexpr void t_set(int & c, int v,
    const std::initializer_list<int> & n)
      {
        c=(c-(c % prods(f+1,n))) + (v*prods(f,n)) + (c % prods(f,n));
      }
```

Sub-bitfields: A Complete Declaration

```
class subbitfield_test
  {
    protected:

      static constexpr std::initializer_list<int> ranges{11,3,4,5,12};
      static constexpr int nb_fields=ranges.end()-ranges.begin();
      static constexpr int nb_bits=bits_from_value(prods(ranges));
      using subbit_type = just_enough_uint<nb_bits>;
      static constexpr int nb_stowaway_bits=bits_from_type<subbit_type>::value-nb_bits;

      subbit_type
          sub_bits: nb_bits,
          stowaway: nb_stowaway_bits; // available (boom! if ==0)

    public:

      subbitfield_test(): sub_bits(subbitfield_init({7,1,3,3,9},ranges)),stowaway(0)
        {
          ...
```

~talk()

After this talk:

- We know it's possible to (mostly) dispense with pointers;
- We understand how we can compress pointers;
- We understand how do use sub-bitfields.

Still so much more to do!

✉ steven_pigeon@uqar.ca

@steven_pigeon