CppNorth 2025

Heaps Don't Lie

Guidelines for Memory Allocation in C++

Mathieu Ropert

Heaps Don't Lie

Guidelines for Memory Allocation in C++

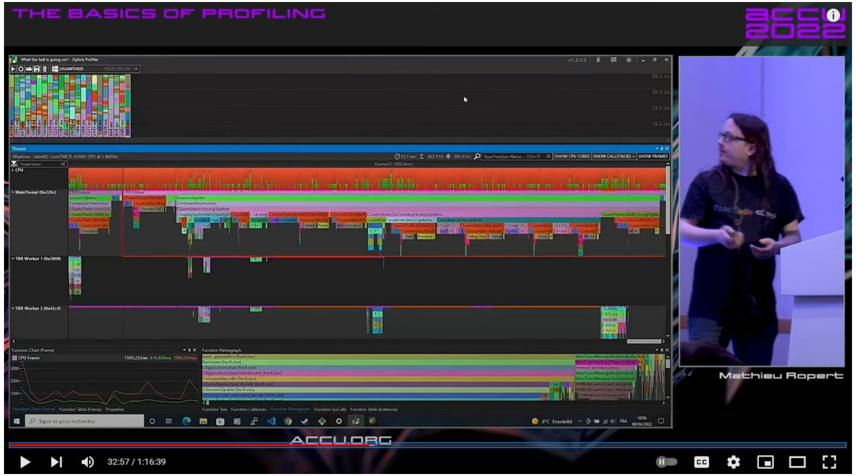
Don’t use `malloc()`

Don't use `malloc()` while the plane is in the air

“

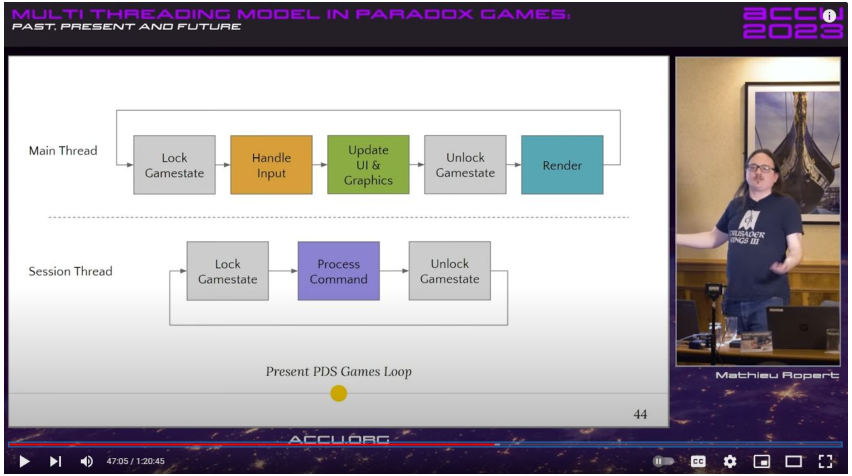
On performance

- Profiling



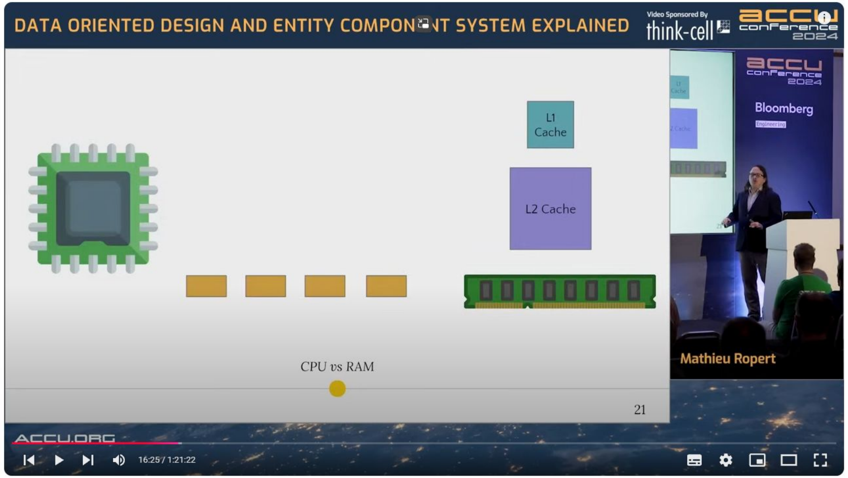
On performance

- Profiling
- Multithreading



On performance

- Profiling
- Multithreading
- Caches & data locality



On performance

- Profiling
- Multithreading
- Caches & data locality
- Allocations

`this`

Hello!

I am **Mathieu Ropert**

I'm a C++ programmer, game developer and meetup organizer. I'm available for consulting and training!

You can reach me at:

- mro@puchiko.net
- @MatRopert   @matropert.bsky.social
- https://mropert.github.io

About this talk

- How does the heap work?
- 10 guidelines for handling memory allocations
- For beginners and experts alike

1

What’s in a malloc()?

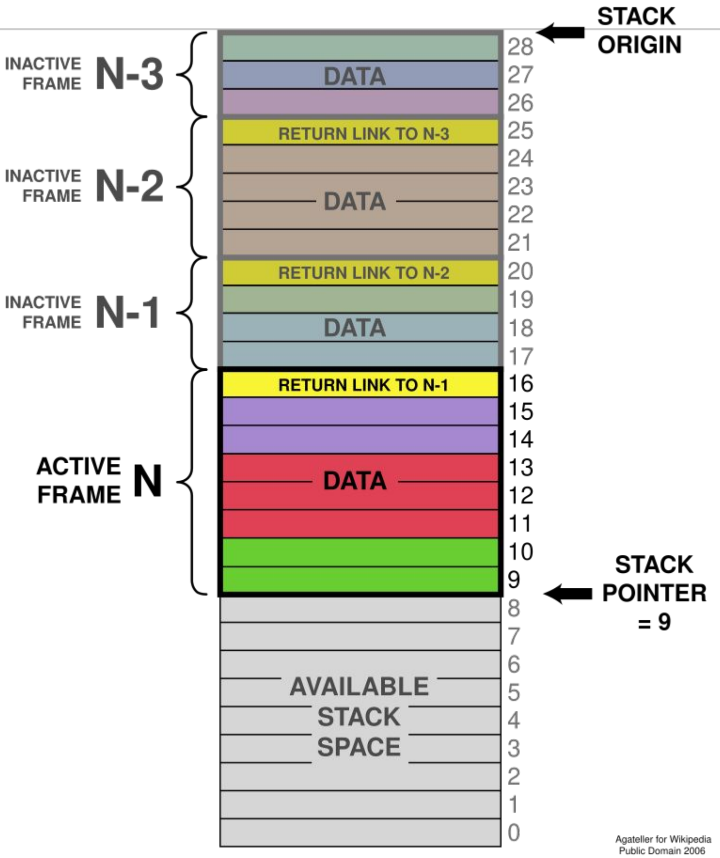
Quite a bit, it turns out

Defining “allocation”

- The act of reserving *uninitialized* bytes
- Not concerned with zeroing or constructing objects in it
- Likewise for deallocation

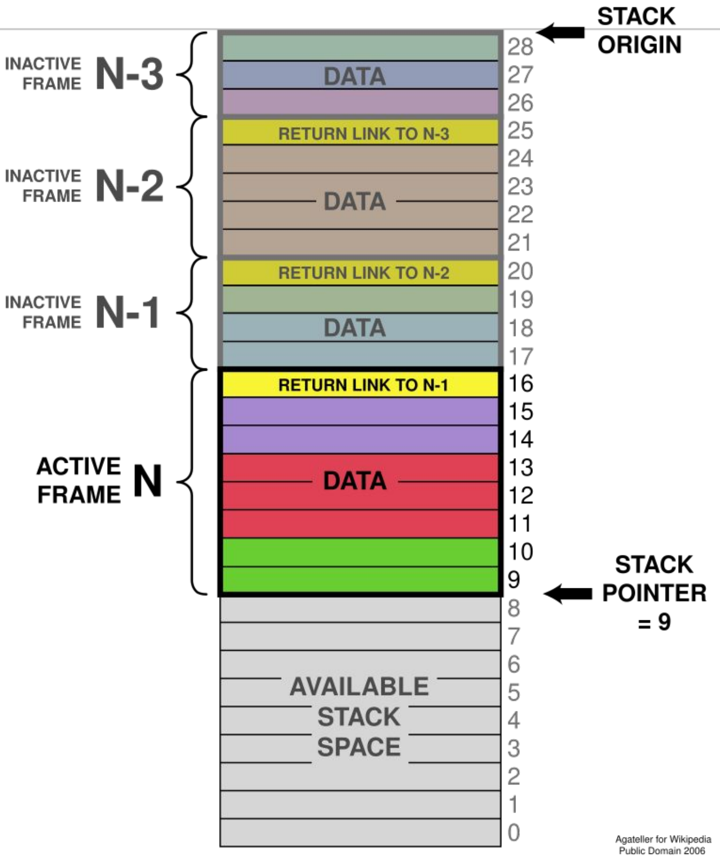
A (stack) frame of reference

- Stack allocation
- Increment stack pointer and you're done
- One register add
- No library or operating system call



Stack memory limitations

- Size of allocation defined at compile time
- Limited in space
- Automatically reclaimed on scope exit



Guideline #1: prefer stack alloc when possible

Enter `malloc()`

- Can allocate arbitrary amounts of memory
- Lives until matching `free()` is called
- Can mimic stack lifetime behaviour with `unique_ptr` and RAII

`malloc()` process

- Try to get a free block of size N from free list

10x

- Grab a free page, initialize it as page of size objects

10x

- Use an OS system call to get more pages

10x

- Swap some physical pages to disk

Guideline #2: malloc() is usually fast, except when it isn't

`malloc()` implementation

- Free blocks and pages are managed by C library implementation
- Virtual memory and paging depend on OS
- First one can be replaced by user-provided implementation

The slide about MSVC

- The default allocator that comes with MSVCRT is not great
- Locks mutex on `free(NULL)` in Debug builds
- Bad performance when allocating/freeing in parallel

The slide about MSVC

ree

Guideline #3: prefer third party heap implementation on Windows

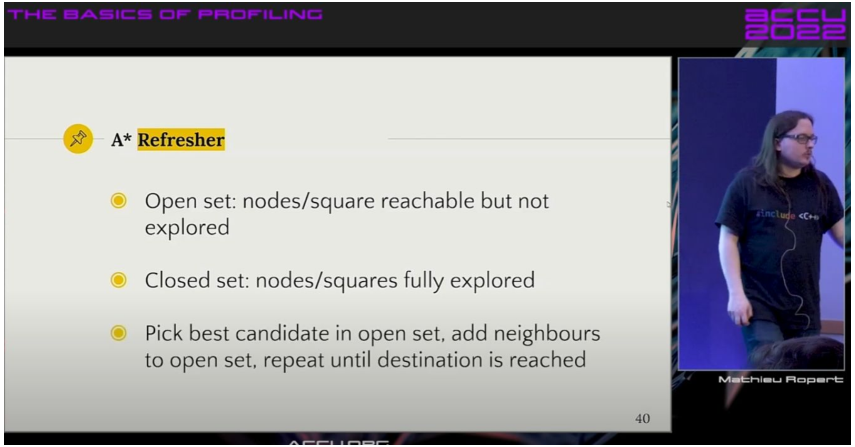
2

Allocation best practices

Experiences collected over the years

The recurring pathfinder test

- Somewhat simple exercise in principle
- Seen dozens of implementations over the years
- Some allocation related mistakes keep repeating



Exploring neighbours

- For a given cell (x, y)
- Check each neighbours
- Filter out/return the ones that are within map bounds and are traversable

Exploring neighbours

```
std::vector<Point> find_neighbours( Point p ) const
{
    std::vector<Point> neighbours;
    if ( p.x_ > 0 ) neighbours.emplace_back( p.x_ - 1, p.y_ );
    if ( p.x_ < size_.x_ - 1 ) neighbours.emplace_back( p.x_ + 1, p.y_ );
    if ( p.y_ > 0 ) neighbours.emplace_back( p.x_, p.y_ - 1 );
    if ( p.y_ < size_.y_ - 1 ) neighbours.emplace_back( p.x_, p.y_ - 1 );
    return neighbours;
}
```

Allocates!
Allocates!
Allocates!

Exploring neighbours

```
std::vector<Point> find_neighbours( Point p ) const
{
    std::vector<Point> neighbours;
    neighbours.reserve( 4 );
    if ( p.x_ > 0 ) neighbours.emplace_back( p.x_ - 1, p.y_ );
    if ( p.x_ < size_.x_ - 1 ) neighbours.emplace_back( p.x_ + 1, p.y_ );
    if ( p.y_ > 0 ) neighbours.emplace_back( p.x_, p.y_ - 1 );
    if ( p.y_ < size_.y_ - 1 ) neighbours.emplace_back( p.x_, p.y_ - 1 );
    return neighbours;
}
```

Allocates!

Guideline #4: reserve final container size when known rather than rely on geometric growth

Abusing reserve()

Allocates!

```
void add_neighbours( const std::vector<Point>& added )
{
    neighbours_.reserve( neighbours_.size() + added.size() );
    std::copy( begin( added ), end( added ),
               std::back_inserter( neighbours_ ) );
}
```


Append Allocs vs Reserve Allocs
Append Allocs
Reserve Allocs
250
200
150
100
50
0
0
200
400
600
800
Size

Guideline #5: avoid using `reserve()` with constant delta in loops

“

Zooming out

```
while ( !search_queue.empty() )
{
    const Point current = search_queue.top();
    search_queue.pop();
    for ( const Point p : find_neighbours( current ) )
    {
        ...
    }
}
```

Allocates!

Exploring neighbours, redux

```
void find_neighbours( Point p, std::vector<Point>& neighbours ) const
{
    if ( p.x_ > 0 ) neighbours.emplace_back( p.x_ - 1, p.y_ );
    if ( p.x_ < size_.x_ - 1 ) neighbours.emplace_back( p.x_ + 1, p.y_ );
    if ( p.y_ > 0 ) neighbours.emplace_back( p.x_, p.y_ - 1 );
    if ( p.y_ < size_.y_ - 1 ) neighbours.emplace_back( p.x_, p.y_ - 1 );
}
```

Outer loop, revised

```
std::vector<Point> neighbours;
while ( !search_queue.empty() )
{
    const Point current = search_queue.top();
    search_queue.pop();
    neighbours.clear();
    find_neighbours( current, neighbours );
    for ( const Point p : neighbours ) { ... }
}
```

Allocates!
(once)

Guideline #6: prefer output parameters to returns for containers

“

Guideline #7: reuse previous allocations when possible

Clearing structures

```
struct Frame {
    std::vector<const Entity*> visible_entities_;
    std::vector<const Entity*> audible_entities_;
    std::string debug_message_;
    ...
    void clear()
    {
        *this = Frame();  Frees!
    }
};
```

Reassigning containers

```
vector& operator=( vector&& other )
{
    clear();
    shrink_to_fit();   Frees!
    swap( other );
    return *this;
};
```

Reassigning containers

```
vector& operator=( vector&& other )
{
    clear();
    if ( !other.empty() )
    {
        shrink_to_fit();
        swap( other );
    }
    return *this;
};
```

Reassigning containers

```
vector& operator=( vector&& other )
{
    clear();
    if ( !other.empty()
        || ( other.empty() && capacity() < other.capacity() ) )
    {
        shrink_to_fit();
        swap( other );
    }
    return *this;
};
```

Clearing structures compared

```
void clear()
{
    *this = Frame();
}
```

```
void clear()
{
    visible_entities_.clear();
    audible_entities_.clear();
    debug_message_.clear();
}
```

Clearing structures compared

```
void render()
{
Oops!   frame_data_ = Frame();
        ...
}
```

```
void render()
{
        frame_data_.clear();
        ...
}
```

Guideline #8: prefer memberwise `.clear()` over assignment to empty struct

“

3

Custom allocators

And other alternatives

General purpose allocator

- Has to be “good enough” for everyone
- Can’t make assumptions about usage patterns
- Compromises have to be made
- What else is there?

What about alloca()?

- Not standard (but very common)
- Dynamic stack allocation
- C API, need wrapper to add RAII back
- Fast

Drawbacks of alloca()

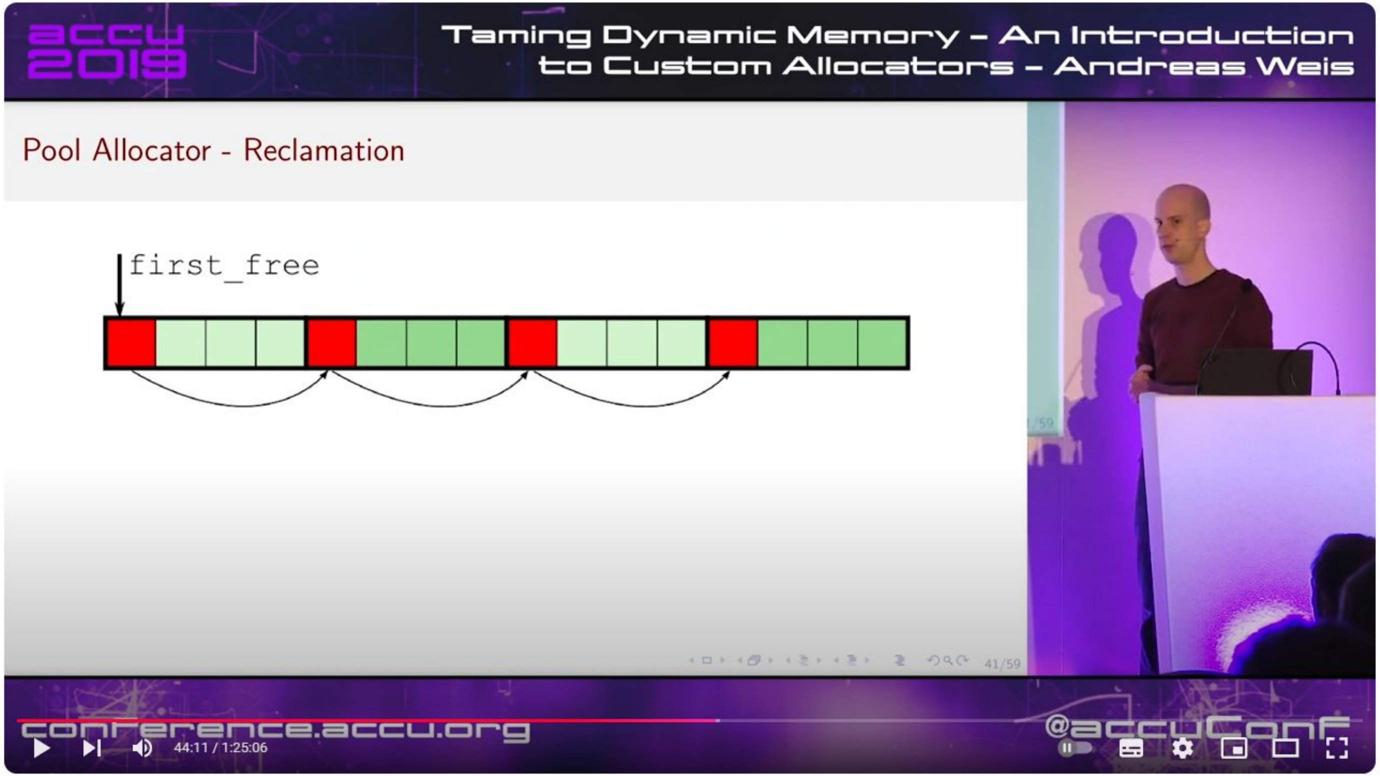
- Bound by stack size
- Limited space
  - Especially on threads
  - Especially on macOS
- Not all threading libraries support setting stack size (standard C++ doesn't)

Guideline #9: avoid using `alloca()` because it's likely to overflow the stack

“

Custom allocators

- Special-purpose allocators
- Found in various libraries and in the standard since C++17
- Drop some malloc() constraints for speed (and/or fragmentation)



Taming Dynamic Memory - An Introduction to Custom Allocators - Andreas Weis - ACCU 2019

Monotonic allocators

- Allocates by increasing a pointer until it runs out
- As fast as stack alloc
- Cannot deallocate one specific object
- Free all or nothing

Simple monotonic allocator

```
struct monotonic_buffer
{
    std::unique_ptr<std::byte[]> buffer_;
    std::byte *current_;

    explicit monotonic_buffer( std::size_t size )
        : buffer_( std::make_unique<std::byte[]>( size ) )
        , current_( buffer_.get() ) { }
    ...
};
```

Simple monotonic allocator

```
void* allocate( std::size_t size )
{
    constexpr auto align_mask = sizeof( std::max_align_t ) - 1;
    const auto alloc_size = ( size + align_mask ) & ~align_mask ;
    void* allocated = current_;
    current_ += alloc_size;
    return allocated;
}
```

Simple monotonic allocator

```
void deallocate( void* ptr ) { }

void deallocate_all()
{
    current_ = buffer_.get();
}
```

Monotonic allocators

- Good for scoped allocations that can be budgeted
- Game levels
- State machines
- Operation, transaction and request processing

Guideline #10: consider monotonic allocators when resources can be scoped and budgeted

“

4

Wrapping up

Would you like to know more?

In conclusion

- Dynamic allocation comes at an unpredictable runtime cost
- Size-up allocations once and reuse containers as much as possible
- Consider monotonic allocators, but beware of going further

Furthermore

Furthermore, I think your build should be destroyed

“

Thanks!

Any questions ?

You can reach me at

- mro@puchiko.net
- @MatRopert
- @matropert.bsky.social
- @mropert
- https://mropert.github.io