Compile-Time Dynamic Memory Allocation is Real

Scottish Programming Languages Seminars
Heriot-Watt University

Paul Keir Andrew Gozillon

School of Computing, Engineering & Physical Sciences
University of the West of Scotland, Paisley, UK

October 21st, 2020

Overview

- C++ Dynamic Memory Allocation
- Enhanced Support in C++20 for Constant Evaluation
- Detecting Runtime Memory Corruption using such Features
- Compile-Time Execution of a Metamath Database Verifier
- The C'est library: A C++ Compile-Time Standard Library
- Transience: Crossing the Compile-Time/Runtime Boundary
- Conclusion/Future Work

Background

- C++20's constexpr support can now tackle large/existent programs
- Relevant features are now available in recent Clang and GCC
- Which program properties can be verified using constant evaluation?
- C++ constant evaluation to verify code correctness using EDSLs
- Construct typed objects using established C++ verification libraries

Generalised Constant Expressions

- C++11 introduced generalised constant expressions
- A `constexpr` function in C++11 was a single `return` statement:

```
constexpr int factorial(int n)
{
  return n <= 1 ? 1 : (n * factorial(n - 1));
}
```

Generalised Constant Expressions

- C++11 introduced generalised constant expressions
- A `constexpr` function in C++11 was a single `return` statement:

```
constexpr int factorial(int n)
{
  return n <= 1 ? 1 : (n * factorial(n - 1));
}
```

- Since C++14, multiple statements are permitted:

```
constexpr int factorial(int n)
{
  int r = 1;
  for (; n>1; --n)
    r *= n;
  return r;
}
```

Generalised Constant Expressions

- C++11 introduced generalised constant expressions
- A `constexpr` function in C++11 was a single `return` statement:

```
constexpr int factorial(int n)
{
  return n <= 1 ? 1 : (n * factorial(n - 1));
}
```

- Since C++14, multiple statements are permitted:

```
constexpr int factorial(int n)
{
  int r = 1;
  for (; n>1; --n)
    r *= n;
  return r;
}
```

- C++17 allowed `constexpr` lambda functions
- C++20 still lacks: non-literal types, static storage; virtual base classes; `goto` statement; `throw` statement...

Compile-time or Runtime evaluation?

- Variables declared as `constexpr`;
- Invocations of functions with `constexpr` annotation;
- Can be evaluated at compile-time
  - ..when given constant arguments; and used in a constant expression

Dynamic Memory Allocation in C and C++

```c
#include <stdlib.h>

void c_malloc()
{
  int *p = malloc(9 * sizeof(int)); // space for 9 integers
  free(p);
}
```

- `malloc` returns a `void*`; implicitly converted in C to an `int*`

Dynamic Memory Allocation in C and C++

```c
#include <stdlib.h>

void c_malloc()
{
  int *p = malloc(9 * sizeof(int)); // space for 9 integers
  free(p);
}
```

- `malloc` returns a `void*`; implicitly converted in C to an `int*`

```cpp
#include <cstdlib>

void cpp_malloc()
{
  int *p = static_cast<int*>(std::malloc(9 * sizeof(int)));
  std::free(p);
}
```

- C++ needs a (static) cast in such cases

Dynamic Memory Allocation in C++

- In C++ the `new` and `delete` operators are recommended
- No casts required; a type is a parameter of `operator new`
- `new` both obtains storage, and initialises an object (here an `int`)
  - ...so starting its *lifetime*

```
void cpp_new()
{
  int *p = new int[9];
  delete [] p;
}
```

New-expressions during Constant Evaluation

- C++20 introduces 7 “Relaxations of constexpr restrictions”
  - P1064, P1002, P1327, P1330, P1331, P1668 and P0784
- The last includes discussion on “constexpr new-expressions”
  - P0784 More constexpr containers (Peter Dimov et al.)
- Only *transient* constexpr allocations were adopted

  i.e. Dynamic memory allocations occurring during a constexpr evaluation that are deallocated before that evaluation completes
- Implementations can omit replaceable global allocation function calls
- *Replaceable:* a user-provided non-member function with the same signature, replaces the default version

Separate Allocation and Initialisation

- It can be useful to allocate storage, but not initialise
- Class template `std::allocator` is the default *Allocator*
- Used (by default) by all containers of the C++ Standard Library

```
#include <memory>

void cpp_allocate()
{
  std::allocator<int> a;
  int *p = a.allocate(1);
  a.deallocate(p,1);
}
```

Separate Allocation and Initialisation

- It can be useful to allocate storage, but not initialise
- Class template `std::allocator` is the default *Allocator*
- Used (by default) by all containers of the C++ Standard Library

```
#include <memory>

constexpr void cpp_allocate()
{
  std::allocator<int> a;
  int *p = a.allocate(1);
  a.deallocate(p,1);
}
```

- Much of `std::allocator` is now marked `constexpr` in C++20

Separate Allocation and Initialisation

```
#include <memory>

constexpr void cpp_allocate()
{
  std::allocator<int> a;
  int *p = a.allocate(4);
  p[3] = 42;
  a.deallocate(p,4);
}
```

- Writing to allocated *memory* (via `p[0]` above) is common practice
  - ...but undefined behaviour; no **int** object was *created*
- R. Smith & V. Voutilainen noted this longstanding defect
- P0593 Implicit creation of objects for low-level object manipulation
- This paper defines *implicit lifetime* types; types where:
  1. Creating an instance of the type runs no code
  2. Destroying an instance of the type runs no code
- Such types are (as of P0593) permitted such implicit creation

Separate Allocation and Initialisation

```
#include <memory>

constexpr void cpp_allocate()
{
  std::allocator<int> a;
  int *p = a.allocate(4);
  p[3] = 42;
  a.deallocate(p,4);
}
```

- Writing to allocated *memory* (via `p[0]` above) is common practice
  - ...but undefined behaviour; no `int` object was *created*
- R. Smith & V. Voutilainen noted this longstanding defect
- P0593 Implicit creation of objects for low-level object manipulation
- This paper defines *implicit lifetime* types; types where:
  1. Creating an instance of the type runs no code
  2. Destroying an instance of the type runs no code
- Such types are (as of P0593) permitted such implicit creation

  ...but not during constant evaluation (Section 3.5)

Separate Allocation and Initialisation

- Consequently, the relevant constructor must be called
- C++20's `std::construct_at` provides simplified syntax

```
#include <memory>

constexpr void cpp_allocate()
{
  std::allocator<int> a;
  int *p = a.allocate(4);

    std::construct_at(&p[3]);

  p[3] = 42;
  a.deallocate(p,4);
}
```

Separate Allocation and Initialisation

- Consequently, the relevant constructor must be called
- C++20's `std::construct_at` provides simplified syntax

```
#include <memory>

constexpr void cpp_allocate()
{
  std::allocator<int> a;
  int *p = a.allocate(4);
  if (std::is_constant_evaluated()) { // this if statement is not required
    std::construct_at(&p[3]);
  }
  p[3] = 42;
  a.deallocate(p,4);
}
```

- This requirement during constant evaluation may disappear
- `std::is_constant_evaluated` can document the changes for now

Detecting Memory Corruption (1 of 4)

- Using non-owned memory
- Undefined behaviour; segmentation fault
- The *static* assert exposes a compilation error:

```
note: read of uninitialized object is not allowed in a constant expression
```

```
#include <memory>
#include <cassert>

constexpr void non_owned() {
  int *p;
  *p = 42;
}

void memory_tests() {
  assert((non_owned(),true));
  static_assert((non_owned(),true));
}
```

Detecting Memory Corruption (2 of 4)

- Using memory beyond that allocated (buffer overflow)
- Undefined behaviour; no runtime error on some systems
- The *static* assert exposes a compilation error:

```
note: assignment to dereferenced one-past-the-end pointer is not allowed in a
constant expression
```

```
#include <memory>
#include <cassert>

constexpr void buffer_overflow(int i) {
  int *p = new int[4];
  p[i] = 43;
  delete [] p;
}

void memory_tests() {
  assert((buffer_overflow(4),true));
  static_assert((buffer_overflow(4),true));
}
```

Detecting Memory Corruption (3 of 4)

- Faulty heap memory management
- Undefined behaviour; segmentation fault
- The *static* assert exposes a compilation error:

```
note: read of uninitialized object is not allowed in a constant expression
```

```
#include <memory>
#include <cassert>

constexpr void unallocated() {
  int *p;
  delete p;
}

void memory_tests() {
  assert((unallocated(),true));
  static_assert((unallocated(),true));
}
```

Detecting Memory Corruption (4 of 4)

- Using uninitialised memory; via a non-implicit lifetime type
- Undefined behaviour; a seg fault; the *static* assert exposes 2 errors:

```
note: read of object outside its lifetime is not allowed in a constant expression
note: allocation performed here was not deallocated
```

```
struct Foo {
  constexpr  Foo() : m_p(new int{}) {}
  constexpr ~Foo() { delete m_p; }
  int *m_p;
};

constexpr void uninitialised() {
  std::allocator<Foo> alloc;
  Foo *fp = alloc.allocate(1);
  /*std::construct_at(fp);*/ *fp->m_p = 42; /*std::destroy_at(fp);*/
  alloc.deallocate(fp,1);
}

void memory_tests() {
  assert((uninitialised(),true));
  static_assert((uninitialised(),true));
}
```

C++ Metamath Database Verifier

- Metamath: a tiny language that can express maths theorems
  - ...accompanied by proofs that can be verified
- Over a dozen proof verifiers are listed on the Metamath website
- **checkmm**: A C++ version by Eric Schmidt
  - 1400 lines of C++ in one source file: checkmm.cpp
  - Makes extensive use of the C++ standard library; 14 headers
  - Containers: `queue`, `string`, `set`, `deque`, `vector`, `pair`, `map`
  - The algorithm's library's `set_intersection` and `find`
  - So too streaming IO operations involving `std::cout` and `std::err`
  - ...and assorted standalone functions

A Compile-Time C++ Standard Library?

- The Metamath verifier **checkmm** uses `std::vector` & `std::string`
- Both were adopted as `constexpr` for the C++ standard library
  - P0980 Making `std::string` constexpr (Louis Dionne)
  - P1004 Making `std::vector` constexpr (Louis Dionne)
- *Neither* is yet implemented in GCC's libstdc++ or Clang's libc++
- Even when they are, other `constexpr` containers will be C++23/26
- There is though no *need* to wait for standard adoption
- The C'est library was created for use today

n.b. C'est is not standalone: its containers work with algorithm & numeric libs

A Simple C'est Example

```
#include "cest/iostream.hpp"
#include "cest/string.hpp"
#include "cest/vector.hpp"
#include "cest/deque.hpp"
#include "cest/set.hpp"
#include "cest/algorithm.hpp"
#include "cest/numeric.hpp"

// clang++ -std=c++2a -I include example.cpp

constexpr bool doit()
{
  using namespace cest;

  string str = "Hello";
  vector<int> v{1,2,3};
  deque<int> dq{2,3,4};
  set<int> s;

  set_intersection(dq.begin(), dq.end(), v.begin(),  v.end(), inserter(s, s.end()));
  auto x = accumulate(s.begin(), s.end(), 0);
  cout << str << " World " << x << endl;

  return 5==x;
}

int main(int argc, char *argv[])
{
  static_assert(doit());
  return doit() ? 0 : 1;
}
```

Porting **checkmm** for C++20 Constant Evaluation

Basic Framework

- Use `static_assert` to ensure compile-time evaluation
- For each `static_assert` also test with a runtime `assert`

Debugging

- Constant evaluation control flow is based only on given arguments
- Runtime `assert` (a macro) can provide errors with line number
- A (`constexpr`-illegal) `throw` will also halt compilation
- GCC standard libraries branch with `std::is_constant_evaluated`
  - Can step debug the code, but may need to replace with `true`

Performance (A10-7850K 16GB)

Filename	linecount	Runtime	Compile-time
`demo0.mm`	51	0.004 secs	3 secs
*`peano.mm`	850	0.006 secs	12 secs
*`set.mm`	688882	23 secs	fails after 200 minutes

*Need Clang's `-fconstexpr-steps` switch (max is 2147483647)

Changes Applied to **checkmm**

1. Added the `constexpr` qualifier to all subroutines
2. Changed all functions, to member functions (methods) of a trivial struct (called `checkmm`)
3. Changed the `static` variable, `names`, within the `readtokens` function to a class member of `checkmm`
4. Compile-time file IO is not possible: `readtokens` now accepts a second string parameter, which is used if it isn't empty
5. File-includes within mm database files are not supported when processing at compile-time. A (helpful) error is issued if this occurs
6. Headers included are from the C'est library, so `#include "cest/vector.hpp"` rather than `#include <vector>` etc.
7. A macro system is available where a pair of C++11-style raw string literal delimiters can be placed before and after the contents of a file. A bash script is available to help with this. To use this approach, we set the preprocessor macro `MMFILEPATH` to the script's output, during C++ compilation (e.g. `-DMMFILEPATH=peano.mm.raw`)

Implementation Repositories

- **C'est**: `https://github.com/pkeir/cest`
- **ctcheckmm**: `https://github.com/pkeir/ctcheckmm`

Transient Allocations Only

- Surprisingly the following code will not compile*

```
#include "cest/vector.hpp"

constexpr cest::vector v = {1,2,3,4,5,6,7};
```

Transient Allocations Only

- Surprisingly the following code will not compile*

```
#include "cest/vector.hpp"

constexpr cest::vector v = {1,2,3,4,5,6,7};
```

- P0784 was adopted in C++20 as excluding non-transient allocations
- Memory allocated during constant evaluation must also be de-allocated
- P1974** proposes a new qualifier: `propconst`

* So too the upcoming `std::vector` will not compile

** P1974 Non-transient constexpr allocation using `propconst` (Jeff Snyder, Louis Dionne & Daveed Vandevoorde)

Saving Transient Values via Statically Sized Types

- Data can be moved to a statically allocated type (e.g. `cest::array`)
- A first attempt will work with a known size (here 7)

```
constexpr auto calc_return_vec() {
  cest::vector v = {1,2,3,4,5,6,7};
  cest::transform(begin(v), end(v), begin(v), [](auto &x) { return x*2; });
  return v;
}

template <auto N>
constexpr auto get_result() {
  auto v = calc_return_vec();
  cest::array<decltype(v)::value_type, N> a;
  cest::move(begin(v), end(v), begin(a));
  return a;
}

constexpr cest::array a = get_result<7>();
```

Saving Transient Values via Statically Sized Types

- Data can be moved to a statically allocated type (e.g. `cest::array`)
- A first attempt will work with a known size (here `7`)

```
constexpr auto calc_return_vec() {
  cest::vector v = {1,2,3,4,5,6,7};
  cest::transform(begin(v), end(v), begin(v), [](auto &x) { return x*2; });
  return v;
}

template <auto N>
constexpr auto get_result() {
  auto v = calc_return_vec();
  cest::array<decltype(v)::value_type, N> a;
  cest::move(begin(v), end(v), begin(a));
  return a;
}

constexpr cest::array a = get_result<7>();
```

- A template argument (here `N`) *demands* a constant expression
- Using `v.size()` would fail, as it reads a non-`constexpr` data member

Saving Transient Values via Statically Sized Types (v2)

- Two calls to `calc_return_vec`; size & result; potentially memoised?
- Support for other containers possible, but also watch `propconst`

```
constexpr auto calc_return_vec() {
  cest::vector v = {1,2,3,4,5,6,7};
  cest::transform(begin(v), end(v), begin(v), [](auto &x) { return x*2; });
  return v;
}

constexpr auto get_size() {
  auto v = calc_return_vec();
  return v.size();
}

template <auto N>
constexpr auto get_result() {
  auto v = calc_return_vec();
  cest::array<decltype(v)::value_type, N> a;
  cest::move(begin(v), end(v), begin(a));
  return a;
}

constexpr cest::array a = get_result<get_size()>();
```

Related Work

- Generalized Constant Expressions (Dos Reis, Stroustrup, Maurer)
  - N1521 (2003), N1972/N1980/N2116 (2006), N2235 (2007)

```
constexpr int square(int x) { return x * x; }
float array[square(9)];
```

- Boost Hana (Louis Dionne, 2014)
  - A popular library: types without template metaprogramming

```
auto types = make_tuple(type_c<int*>, type_c<char&>, type_c<std::string*>);
auto ptr_types = filter(types, [](auto a) { return traits::is_pointer(a); });
static_assert(ptr_types == make_tuple(type_c<int*>, type_c<std::string*>));
```

- Circle Language (Sean Baxter)
  - Interpreter supports compile-time execution of all C++ statements

```
int main(int argc, char *argv[]) {
  printf("Hello world\n");
  @meta printf("Hello circle\n");
}
```

- Dependently Typed Languages (Idris, Agda, Coq)
  - Lack termination/totality checking; non-`constexpr` values as types

Future Work

1. A clang-tidy pass which:
   - adds constexpr to select functions/methods which are without; and
   - refactors common deficiencies such as global/static variables
2. A constexpr inferring compiler
3. Add compiler diagnostics to report constexpr-steps of an expression
   - ...so helping constant evaluation performance unit tests for CI
4. Errors often lead to `std::is_constant_evaluated` branches
   - Add a compiler flag to allow `std::is_constant_evaluated` to return true *at runtime*; to help debug constant expressions
5. Create a clang-tidy pass which replaces suitable loops (etc.) with calls to `std::for_each`, `std::transform` etc.
6. Enhance gprof or callgrind/kcachegrind
   - Display call information about constant evaluations

Acknowledgements

UNIVERSITY OF THE WEST of SCOTLAND

for Andrew Gozillon's Ph.D. studentship

For ongoing contributions to software development:

Seyed Hossein HAERI, Université catholique de Louvain

Jules PENUCHOT, Paris-Saclay University