Transient Allocation

Tools and Libraries for Compile-time Software Engineering
HiPEAC Conference 2024
Munich, Germany

Paul Keir [1]
Joel FALCOU [2]

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

[2]Le Laboratoire Interdisciplinaire des Sciences du Numérique (LISN)
Université Paris-Saclay, Paris, France

January 17th, 2024

Dynamic Memory Allocation in Constant Evaluation

- Since C++20, `new` and `delete` operators can execute within constexpr functions
- A `constexpr` function needn't run at compile time (good)
- When required, constant evaluation can be demanded
- e.g. `static_assert`; `constexpr` decl'n; NTTP argument
- Our tests are always paired:
  - runtime tests: `assert`
  - compile-time: `static_assert`

```
#include <cassert>

constexpr bool all_ok()
{
  int *p = new int{42};
  bool b = 42 == *p;
  delete p;

  return b;
}

int main()
{
         assert(all_ok());
  static_assert(all_ok());

  return 0;
}
```

Transient Allocations Only

- Allocations must be free'd before constant evaluation concludes
- Without the call to `delete` on the right: compilation error
  - ...due to the `static_assert`
- Reminiscent of a Valgrind memory leak check

```
#include <cassert>

constexpr bool all_ok()
{
  int *p = new int{42};
  bool b = 42 == *p;
  // delete p;

  return b;
}

int main()
{
         assert(all_ok());
  static_assert(all_ok());

  return 0;
}
```

Beware of Non-Transient Allocation (1 of 2)

- Since C++20 `std::vector` and `std::string` are now "constexpr"
  - ...after Louis Dionne's P0980 & P1004 proposals
- Such containers can now be used within `constexpr` functions
- But there are subtle limits on how they may be used
- All 6 declarations below fail, as memory is used "non-transiently"
  - i.e. memory allocated during constant evaluation isn't free'd

```
#include <vector>
#include <string>

constexpr double *g_p = new double{0.577215};
constexpr std::vector g_v{0,1,1,2,3,5,8};
constexpr std::string g_str = "rosebud";

int main()
{
  constexpr double *p = new double{0.577215};
  constexpr std::vector v{0,1,1,2,3,5,8};
  constexpr std::string str = "rosebud";
  return 0;
}
```

Beware of Non-Transient Allocation (2 of 2)

- So how can `constexpr` standard C++ containers be used?

Beware of Non-Transient Allocation (2 of 2)

- So how can `constexpr` standard C++ containers be used?

```
#include <vector>
#include <string>

constexpr bool string_vector_ok()
{
  std::vector v{0,1,1,2,3,5,8};
  std::string str = "rosebud";
  return v[6] == 8 && str[0] == 'r';
}

int main()
{
         assert(string_vector_ok());
  static_assert(string_vector_ok());
  return 0;
}
```

Beware of Non-Transient Allocation (2 of 2)

- So how can `constexpr` standard C++ containers be used?

```
#include <vector>
#include <string>

constexpr bool string_vector_ok()
{
  std::vector v{0,1,1,2,3,5,8};
  std::string str = "rosebud";
  return v[6] == 8 && str[0] == 'r';
}

int main()
{
         assert(string_vector_ok());
  static_assert(string_vector_ok());
  return 0;
}
```

- Can we really not get `constexpr` result data at runtime? (Hmm...)

Saving Transient Values via Statically Sized Types

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

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

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

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

Saving Transient Values via Statically Sized Types

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

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

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

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

- A template argument (here `N`) *demands* a constant expression
- Using `v.size()` would fail: `get_result` could be called at runtime

Saving Transient Values via Statically Sized Types (v2)

- Two calls to `calc_return_vec`; size & result; potentially memoised?
- Other containers can be handled similarly

```
constexpr auto calc_return_vec() {
  std::vector v = {1,2,3,4,5,6,7};
  std::transform(v.begin(), v.end(), v.begin(), [](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();
  std::array<decltype(v)::value_type, N> a;
  std::move(v.begin(), v.end(), a.begin());
  return a;
}

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

Acknowledgements