CONTENTS

1. Template metaprogramming
2. Variadic template arguments
3. Smart pointers
Template metaprogramming

- Template metaprogramming is a Turing complete language
  - Every intuitive computable number can be computed
    - Meaning: we can basically compute anything
  - Funny implication
    - There cannot be a correct C++ compiler!
- TMP is a bit esoteric
  - Some software companies do not allow it
  - However, there are some users
    - Boost.Hana – your standard library for metaprogramming
- Try to use `constexpr` (since C++11) instead of TMP
  - You will see why that is one the next few slides!
Template metaprogramming prerequisites

- static variables in struct/class
  - Are shared across all variables of that type
  - They belong to the type itself
  - Similar to global variables but with limited scope
- Great news
  - Types can store values
    - And with values we can perform computations
    - So we can perform computations with types
  - Templates are processing types!
  - We just discovered metaprogramming

- TMP uses types to express computations

```cpp
#include <iostream>
struct A {
  // 'value' exists only once-across all // variables of type A
  static const int value = 100;
};
int main() {
  A a, b;
  std::cout << a.value << '
';
  std::cout << b.value << '
';
  std::cout << A::value << '
';
  return 0;
}
```
Template metaprogramming

- Functional language
  - Compute using pattern matching and recursion
- Example: computing the power function

```cpp
#include <iostream>

template<int B, unsigned E>
struct power {
    static const int value = B * power<B, E - 1>::value;
};

template<int B>
struct power<B, 0> { // template specialization on the power template type
    static const int value = 1;
};

int main() {
    const int p = power<2, 10>::value;
    std::cout << p << '
';
    return 0;
}
```
Template metaprogramming

```cpp
#include <iostream>
template<int B, unsigned E>
struct power {
    static const int value = B *
        power<B, E - 1>::value;
};
template<int B>
struct power<B, 0> {
    static const int value = 1;
};
int main() {
    const int p = power<2, 10>::value;
    std::cout << p << '
';
    return 0;
}
```

- In programming using templates
  - Types are used as functions
  - They can get
    1. Types
    2. Constant values
    3. References to functions
      - as input parameters
      - They can store a
        1. type with `typedef`
        2. constant with `enum` or `static const`
  - Template specialization directs control flow (pattern matching and recursion)
- In our example
  - templates get instantiated ...
    - until the base case is reached
Template metaprogramming

```cpp
#include <iostream>

template<int B, unsigned E>
struct power {
    static const int value = B *
        power<B, E - 1>::value;
};

template<int B>
struct power<B, 0> {
    static const int value = 1;
};

int main() {
    const int p = power<2, 10>::value;
    std::cout << p << '
';
    return 0;
}
```

```cpp
#include <iostream>

constexpr int power(int base, unsigned exp) {
    return (exp == 0) ? 1
        : base * power(base, exp - 1);
}

int main {
    constexpr p = power(2, 10);
    std::cout << p << '
';
    return 0;
}
```
Template metaprogramming

- Even data structures can be realized
- Remember the triple type from the exercises
- C++’s tuple data type is implemented using template metaprogramming
- Lists are also possible
Computing Euler’s number at compile time using TMP

- Use this formula for $e$

\[
e = 1 + \frac{1}{1} + \frac{1}{1 \cdot 2} + \frac{1}{1 \cdot 2 \cdot 3} + \frac{1}{1 \cdot 2 \cdot 3 \cdot 4} + \cdots
\]

\[= \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \cdots
\]

\[= \sum_{k=0}^{\infty} \frac{1}{k!}
\]
Computing Euler’s number at compile time (TMP) I

#include <iostream>

template<int N, int D>
struct Frac {
    const static int Num = N;
    const static int Den = D;
};

template<int X, typename F>
struct Mult {
    typedef Frac<X * F::Num, X * F::Den> value;
};

template<int X, int Y>
struct GCD {
    const static int value = GCD<Y, X % Y>::value;
};

template<int X>
struct GCD<X, 0> {
    const static int value = X;
};

template<typename F>
struct Simplify {
    const static int gcd = GCD<F::Num, F::Den>::value;
    typedef Frac<F::Num / gcd, F::Den / gcd> value;
};

template<typename X1, typename Y1>
struct SameBase {
    typedef typename Mult<Y1::Den, X1>::value X;
    typedef typename Mult<X1::Den, Y1>::value Y;
};

template<typename X, typename Y>
struct Sum {
    typedef SameBase<X, Y> B;
    const static int Num = B::X::Num + B::Y::Num;
    const static int Den = B::Y::Den;
    typedef typename Simplify<Frac<Num, Den>>::value value;
};
Computing Euler’s number at compile time (TMP) II

```cpp
template<int N>
struct Fact {
    const static int value = N*Fact<N-1>::value;
};

template<>
struct Fact<0> {
    const static int value = 1;
};

template<int N>
struct E {
    const static int Den = Fact<N>::value;
    typedef Frac<1, Den> term;
    typedef typename E<N-1>::value next_term;
    typedef typename Sum<term, next_term>::value value;
};

template<>
struct E<0> {
    typedef Frac<1, 1> value;
};

int main() {
    typedef E<12>::value X;
    std::cout << "e = " << (1.0 * X::Num / X::Den) << '\n';
    std::cout << "e = " << X::Num << " / " << X::Den << '\n';
    return 0;
}
```

[Example taken from https://monoinfinito.wordpress.com/series/introduction-to-c-template-metaprogramming/]

© Heinz Nixdorf Institut / Fraunhofer IEM
Computing Euler’s number at compile time (**constexpr**) III

- Using the same formula

```cpp
#include <iostream>

constexpr unsigned factorial(unsigned n) {
    return (n == 0) ? 1 : n * factorial(n - 1);
}

constexpr double euler(unsigned n) {
    double e = 1;
    for (unsigned i = 1; i <= n; ++i) {
        e += 1.0 / factorial(i);
    }
    return e;
}

int main() {
    constexpr double e = euler(12);
    std::cout << "Euler's number is: " << e << std::endl;
    return 0;
}
```

- Let’s see what the compiler does

- Compile with:

  ```
  clang++ -std=c++17 -Wall -emit-llvm -S -fno-discard-value-names euler.cpp
  ```

  (obtain LLVM compiler’s internal representation)
Pros and cons using template metaprogramming

- **Pros**
  - Evaluated at compile time
  - Higher abstraction possible

- **Cons**
  - Longer compile times
  - Hard to read / write
  - Functional style does not match C++
  - Not supported by development tools
  - Error messages usually make no sense
  - Heavily overused
  - “No type information”

- Use C++ `constexpr` instead!
- Unless you have good reason to do otherwise
Variadic template arguments

```cpp
#include <iostream>

// With C++14

template<typename T>
T sum(T t) {
    return t;
}

template<typename T, typename... Args>
T sum(T t, Args... args) {
    return t + sum(args...);
}

int main() {
    int res = sum(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
    std::cout << res << '\n';
    return 0;
}

// With C++17

template<typename... Args>
auto sum(Args&&... args) {
    return (args + ...);
}
```

- Using C++17 fold expressions

Compiler can oftentimes deduce template parameter(s)
Variadic template arguments

- Another example: printing everything
- Print arbitrary many arguments of arbitrary type

```cpp
#include <iostream>
#include <string>
template<class T>
void print_everything(T t) {
    std::cout << t << '
';
}

template<class T, class... Args>
void print_everything(T t, Args... args) {
    std::cout << t << ' ';
    print_everything(args...);
}

int main() {
    print_everything("Hello",
                    1,
                    2.333,
                    std::string("World"));
    return 0;
}
```

• Using C++17 fold expressions

```cpp
template<typename... Args>
void print(Args&&... args) {
    ((std::cout << t << ' '), ...);
}
```
Smart pointers

- Remember (raw) pointers
  ```cpp
  int i = 42;
  int *i_ptr = &i;
  ```
  - Pointers are necessary for dynamically memory allocation
    ```cpp
    int *dyn_int = new int;
    delete dyn_int;
    int *dyn_array = new int[12];
    delete[] dyn_array;
    ```
    - What was the problem here?
      - You probably will forget to use `delete / delete[]` at some point
      - Finding memory leaks is expensive
  - Smart pointers (SPs) are safe wrappers for raw pointers
Ownership problematic

```c
matrix *matrix_multiply(matrix *a, matrix *b) {
    matrix *c = new matrix(a->rows(), b->cols());
    // perform the computation c = a * b;
    return c;
}
```

- Problem
  - Who frees matrix \(c\), allocated in \texttt{matrix\_multiply()}?
  - It has to be deleted at some point
- Problem in general: Who is responsible, who owns the resource(s)?
  - Who allocates memory and who frees it after usage?
    1. Caller allocates, caller frees (cf. right)
    2. Callee allocates, caller frees (cf. above)
    3. Callee allocates, callee frees (cf. \texttt{std::string, std::vector})
Smart pointers

- Help with ownership problematic
  - SPs know who owns what resource
- SPs do the clean-up (`delete`) themselves
  - They automatically call the destructor if the managed resource has no owner anymore
    - “Are no longer used by anyone”
  - How?
    - SPs calls `delete` for object pointing-to when their own destructor is called
    - Smart pointer know about ownership!
- That is not a real garbage collector
- It is just reference counting – “The poor man’s garbage collector.”
  - “Only pay for counter-variables and incrementing / decrementing counters”
- By the way: it is possible to leak resources in Java (although it has a garbage collector)
Smart pointers

- Three types of smart pointers exist
  - `std::unique_ptr` // for unique ownership
    - One user at a time
  - `std::shared_ptr` // for shared ownership
    - One or more users at a time
  - `std::weak_ptr` // for non-owned things
    - Does not own, but is allowed to use the underlying object
    - Not commonly used in practice

- SPs are implemented in the STL
- All SPs defined in `<memory>`
  - Use `#include <memory>`
std::unique_ptr

- `std::unique_ptr` behaves like an ordinary pointer
- Example
  ```cpp
  struct Data {
    double x;
    double y;
    Data(double x, double y) : x(x), y(y) {}
  };
  int main() {
    std::unique_ptr<Data> data_ptr(new Data(12.5, 14.8));
    return 0;
  }
  ```
- Note that we do not use `delete` explicitly
std::unique_ptr

- Using a factory function

```cpp
struct Data {
  double x;
  double y;
  Data(double x, double y) : x(x), y(y) {}
};
int main() {
  std::unique_ptr<Data> data_ptr(std::make_unique<Data>(12.5, 14.8));
  return 0;
}
```

- Caution: `std::make_unique()` has been introduced in C++14
  - It has been “kind of” forgotten in C++11
  - In C++11 just use `new`
**std::unique_ptr**

1. How to model a std::unique_ptr?
   - Make it a class providing a pointer to the resource

2. How to ensure data_ptr is the only user?
   - Disallow copying the smart pointer
     ```cpp
     unique_ptr(const unique_ptr& up) = delete;
     unique_ptr& operator= (const unique_ptr& up) = delete;
     ```
   - Now we can only have one user at a time
   - Attempts of copying result in an compiler error

3. How is data_ptr able to delete its resource?
   - It uses the destructor
     ```cpp
     ~unique_ptr() { delete resource; }  
     ```
   - Now the resource is cleaned up for us

4. How to use it elsewhere without copying?
   - Use std::move()
How to use smart pointers and what about dereferencing?

- Operators are overloaded to make the smart pointer behave like a raw pointer
- Dereference and obtain the managed resource
  - `T& operator*()`
- Dereference and access a member of the managed resource
  - `T* operator->()`
This code does not compile

- Why?
  - `std::unique_ptr` cannot be copied
  - Because copying results in more than one user!
- Here we would have two owners
  - `main()`
  - `setZero()`
- Move data instead of copying to have one user at a time
  - `std::move()` `data_ptr` into `setZero()`
  - and back from `setZero()` to `main()`
std::unique_ptr

```cpp
struct Data {
    double x;
    double y;
    Data(double x, double y) : x(x), y(y) {}
};

std::unique_ptr<Data> setZero(std::unique_ptr<Data> d) {
    d->x = 0.0;
    d->y = 0.0;
    return d;
}

int main() {
    std::unique_ptr<Data> data_ptr(new Data(12.5, 14.8));
    std::unique_ptr<Data> zero = setZero(std::move(data_ptr));
    std::cout << zero->x << '
';
    std::cout << zero->y << '
';
    return 0;
}
```

- That works
- Caution:
  - Do not use `data_ptr` after you moved it somewhere else!
    - Undefined behavior
    - Segmentation fault
  - The second `std::move()` is “hidden”
    - `setZero()` moves `d` back to `main()` into the variable `zero`
  - Compiler will complain if you forget `move()`
std::shared_ptr

- Allows multiple owners

```cpp
class Data {
public:
    double x; double y;
    Data(double x, double y) : x(x), y(y) {}
};
```

```cpp
std::shared_ptr<Data> setZero(std::shared_ptr<Data> d) {
    d->x = 0.0;
    d->y = 0.0;
    return d;
}
```

```cpp
int main() {
    std::shared_ptr<Data> data_ptr(new Data(12.5, 14.8));
    std::shared_ptr<Data> zero = setZero(data_ptr);
    std::cout << zero->x << 'n';
    std::cout << zero->y << 'n';
    return 0;
}
```

- Keeps track of its owners using an internal counter
- `setZero()` can now be used without `std::move()`
  - It can be copied
  - We allow more than one user!
### std::shared_ptr

- **Improved example**

```cpp
struct Data {
    double x; double y;
    Data(double x, double y) : x(x), y(y) {}
};

std::shared_ptr<Data> setZero(std::shared_ptr<Data> d) {
    d->x = 0.0;
    d->y = 0.0;
    return d;
}

int main() {
    std::shared_ptr<Data> data_ptr = std::make_shared<Data>(12.5, 14.8);
    std::shared_ptr<Data> zero = setZero(data_ptr);
    std::cout << zero->x << '\n';
    std::cout << zero->y << '\n';
    return 0;
}
```

- **std::make_shared() makes a difference**
  - Performs only one allocation for data and reference counter
  - Data and reference counter sit in one block of memory
  - More efficient because of data locality

---

© Heinz Nixdorf Institut / Fraunhofer IEM
std::shared_ptr

1. How to model a `shared_ptr`?
   - Make it a class providing a pointer to a resource
2. How to store the number of users/references?
   - Store them in a counter
3. How to copy?
   - Just perform a **flat copy** of the handle (do not copy the managed resource)
   - Increment the reference counter on copy
4. When to delete the resource?
   - `~shared_ptr` { `if (--refcounter == 0) delete resource;` }
std::weak_ptr

- Can hold a reference but is not an owner

```cpp
#include <iostream>
#include <memory>

std::weak_ptr<int> wp;

void f() {
    if (std::shared_ptr<int> spt = wp.lock()) {
        std::cout << *spt << '\n';
    } else {
        std::cout << "wp is expired" << '\n';
    }
}

int main() {
    auto sp = std::make_shared<int>(42);
    wp = sp;
    f();
    f();
    return 0;
}
```

- You rarely use it
- A std::weak_ptr must be copied into a std::shared_ptr in order to be used
A note on smart pointers

- Massively reduce probability of introducing memory leaks
- Always prefer using smart pointers when managing resources
- Prefer `std::unique_ptr` over `std::shared_ptr`, if possible
- Custom deleters are possible
- Smart pointers behave like raw pointers
  - Just need a tiny little bit more memory in case of `std::shared_ptr`
- Only fallback to raw pointers …
  - if you cannot afford a few bytes more per variable
  - if your platform does not provide an STL implementation
  - if you implement algorithms
  - if you have another good reason

```
std::unique_ptr

Defined in header `<memory>`

```
```template<
  class T,
  class Deleter = std::default_delete<T>
>
  class unique_ptr;
```
```template<
  class T,
  class Deleter
>
  class unique_ptr<T[], Deleter>;
```
A note on dynamic memory allocation

- If you have to dynamically allocate objects
  - Use smart pointers (`std::unique_ptr`, `std::shared_ptr`)
- If you have to dynamically allocate an array of objects
  - Use `std::vector`

- Do not think there are no exceptions
  - Raw pointers are still needed
    - When implementing algorithms
    - If you are only a user and not an owner of a resource
    - ...
Status Quo

- You know very much about modern C++
  - Probably more than your older professors
- What next?
  - We have to deepen your knowledge
  - There will be a summer exercise sheet with 16 additional points
  - Object oriented programming (OOP)
  - Threads and asynchronous tasks (running computations in parallel)
  - High performance computing (HPC) and what you should know about it
  - Static analysis and job offers
  - Introduction to the final project as well as hacks and miscellaneous
- A nice talk by Bjarne Stroustrup that recaps everything so far and more:
  - [https://www.youtube.com/watch?v=86xWVb4XlyE](https://www.youtube.com/watch?v=86xWVb4XlyE)
Recap

- Template metaprogramming
- Variadic template arguments
- Ownership – who is responsible for clean-up
- Smart pointers
  - `std::unique_ptr`
  - `std::shared_ptr`
  - `std::weak_ptr`
- Status quo
Thank you for your attention
Questions?