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
  - Many software companies do not allow it
  - However, there are some users
    - Boost.Hana – Your standard library for metaprogramming
- Try to use \texttt{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

- Great news
  - Types can store values, then
    - 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 in order to express computations

```cpp
#include <iostream>
using namespace std;

struct A {
    // `value` exists only once across all variables of type A
    static const int value = 100;
};

int main() {
    A a, b;
    cout << a.value << 'n';
    cout << b.value << 'n';
    // you do not even need an instance
    cout << A::value << 'n';
    return 0;
}
```
Template metaprogramming

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

```cpp
#include <iostream>
using namespace std;

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;
    cout << p << \n';
    return 0;
}
```
Template metaprogramming

- 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 (recursion)

- In our example
  - templates get instantiated …
    - until the base case is reached

```cpp
#include <iostream>
using namespace std;

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;
    cout << p << '
';
    return 0;
}
```
# Template metaprogramming

```cpp
#include <iostream>
using namespace std;

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;
    cout << p << '
';
    return 0;
}
```

```cpp
#include <iostream>
using namespace std;

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

int main {
    constexpr int p = power(2, 10);
    cout << p << 'n';
    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

```cpp
#include <iostream>
using namespace std;

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;
};
```

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

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<X, Den> term;
    typedef typename E<N-1>::value next_term;
};

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

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

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

YOU ARE A WIZARD, HARRY.
Computing Euler’s number at compile time (**constexpr**) III

- Using the same formula:

```c++
#include <iostream>
using namespace std;

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

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

t void main() {
    constexpr double e = euler(12);
    cout << "Eulers number is: " << e << '\n';
    return 0;
}
```

- Let’s see what the compiler does:

Compile with:

```
clang++ -std=c++17 -Wall -emit-llvm -S euler.cpp
```

(obtain compilers internal representation)

- In addition, use can use imperative programming

```c++
int main() {
    double e = euler(12);
    cout << "Eulers number is: " << e << '\n';
    return 0;
}
```
Pros and cons using template metaprogramming

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

- **Cons**
  - Compile time gets longer
  - 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**

- Example: add function from exercises

```cpp
#include <iostream>
using namespace std;

template<class T>
T add(T t) {
    return t;
}

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

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

- 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>
using namespace std;

template<class T>
void print_everything(T t) {
    cout << t << 'n';
}

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

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

[Have a look at http://eli.thegreenplace.net/2014/variadic-templates-in-c/]
```
Smart pointers

- Remember (raw) pointers
  ```cpp
  int i = 42;
  int *i_ptr = &i;
  ```

- Pointers are necessary for dynamically memory allocation
  ```cpp
  int *dyn_array = new int[12];
  delete[] dyn_array;

  int *dyn_int = new int;
  delete dyn_int;
  ```

- What was the problem here?
  - You probably will forget to use `delete / delete[]` at some point
  - Finding memory leaks can cost days / weeks / $\infty$ amount of time

- Smart pointers (SPs) are safe wrappers for raw pointers
Ownership problematic

```cpp
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 c, allocated in `matrix_multiply()`?
  - It has to be deleted at some point

- Problem in general: Who is responsible, who owns the resources?
  - Who allocates memory and who frees it after usage?
    - Caller allocates, caller frees (cf. right)
    - Callee allocates, caller frees (cf. above)
    - Callee allocates, callee frees (cf. `std::string, std::vector`)

```cpp
void matrix_multiply(matrix* a, matrix* b, matrix* c);
```
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!

- This is not a real garbage collector
- This 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>`
**unique_ptr**

- std::unique_ptr behaves like a usual pointer
- Example

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

```cpp
int main() {
    unique_ptr<Data> data_ptr(new Data(12.5, 14.8));
    return 0;
}
```

- Notice we do not use delete explicitly

**Did it work?**

```
Memcheck, a memory error detector
Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
Command: ./unique
HEAP SUMMARY:	in use at exit: 0 bytes in 0 blocks
total heap usage: 1 allocs, 1 frees, 16 bytes allocated
All heap blocks were freed -- no leaks are possible
ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

**GREAT!**
unique_ptr

- Using the factory function

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

int main() {
    unique_ptr<Data> data_ptr(make_unique<Data>(12.5, 14.8)); // use make_unique
    return 0;
}
```

- Caution: `make_unique()` exists since C++14
  - It has been ‘kind of’ forgotten in C++11
  - In C++11 just use `new`
unique_ptr

1. How to model a unique_ptr?
   - Make it a class providing a pointer to a 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 data_ptr
   - Attempts of copying result in a compiler error

3. How is data_ptr able to delete its resource?
   - Use its 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 about dereferencing?

- Use operator overloading to make your 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?
- `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
- `move()` `data_ptr` into `setZero()`
- and back from `setZero()` to `main()`

---

**Example**

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

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

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

- **Example**

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

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

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

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

- Allows multiple owners
- **Example**

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

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

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

- Keeps track of is owners via an internal counter
- **setZero()** can now be used without **move()**
  - It can be copied
  - We allow more than one user!
- Does it still clean-up?
**shared_ptr**

- **Improved example**

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

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

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

- **make_shared()** makes a difference
  - Does only one allocation for data and reference counter
  - data and reference counter sit in one block of memory
  - More efficient!
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;
   }

- Actual implementation is more advanced

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

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

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

- Can hold a reference but is not an owner

```cpp
#include <iostream>
#include <memory>
using namespace std;

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

int main() {
    auto sp = make_shared<int>(42);
    wp = sp;
    f();
}
```
A note on smart pointers

- Massively reduce probability of introducing memory leaks
- Always prefer using smart pointers when managing resources
- If it makes sense, prefer `std::unique_ptr` over `std::shared_ptr`
- Smart pointers behave like raw pointers
  - Just need a tiny little bit more memory (`std::shared_ptr`)
- Only fallback to raw pointers ...
  - if you cannot afford a few bytes more per variable
  - if your platform does not provide a STL implementation
  - if you implement algorithms
  - if you have another good reason
A note on dynamic memory allocation

- If you have to dynamically allocate objects
  - Use smart pointers
- 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 is 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 (SA) 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
- Smart pointers
  - std::unique_ptr
  - std::shared_ptr
  - std::weak_ptr
- Status quo
Thank you for your attention

Questions?