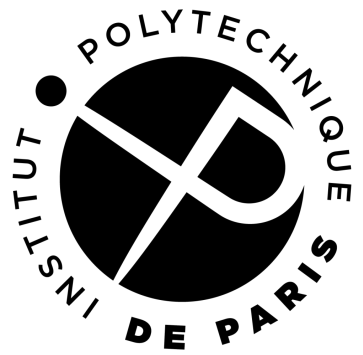


Templates

Bachelor of Science - École polytechnique

gael.thomas@inria.fr

Key concepts

- A template entity is an entity parametrized by parameters
  - The entity can be a class or a method
  - The parameter can be a class or literal

```
template <class T, size_t n>
struct array_t { T elements[n]; };

void f() {
  array_t<int, 4> a;
}
```

- `using` can be used to give a new name to a type
  - Can itself be a template

- `typename` tells the compiler that a syntactic element is a type

Specific versus generic code

- Problem: in our data structures, the type is fixed
  - For example, the army of monsters contains monsters

```
struct army_t {
  monster_t** array;
  size_t top;
  size_t max;

  monster_t* at(size_t i) {
    return array[i];
  }

  void append(monster_t* monster) {
    ...
  }
};
```

Specific versus generic code

- To make the code more generic, we can use `void*`

```
struct army_t {
  void** array;
  size_t top;
  size_t max;

  void* at(size_t i) {
    return array[i];
  }
};
```

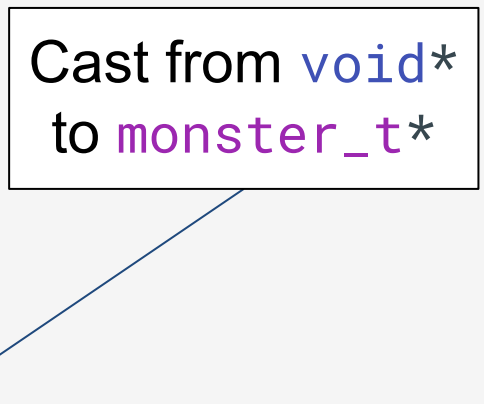
Specific versus generic code

- However, requires a cast to access the elements

```
struct army_t {
  void** array;
  size_t top;
  size_t max;

  void* at(size_t i) {
    return array[i];
  }
};

int main(int arg, char* argv[]) {
  army_t army;
  ...
  monster_t* m = static_cast<monster_t*>(army.at(0));
  return 0;
}
```



Specific versus generic code

- And a cast is error prone!

```
struct army_t {
  void** array;
  size_t top;
  size_t max;

  void* at(size_t i) {
    return array[i];
  }
};

int main(int arg, char* argv[]) {
  army_t army;
  army.append(new complex_t { 1, 2 });
  monster_t* m = static_cast<monster_t*>(army.at(0));
  return 0;
}
```

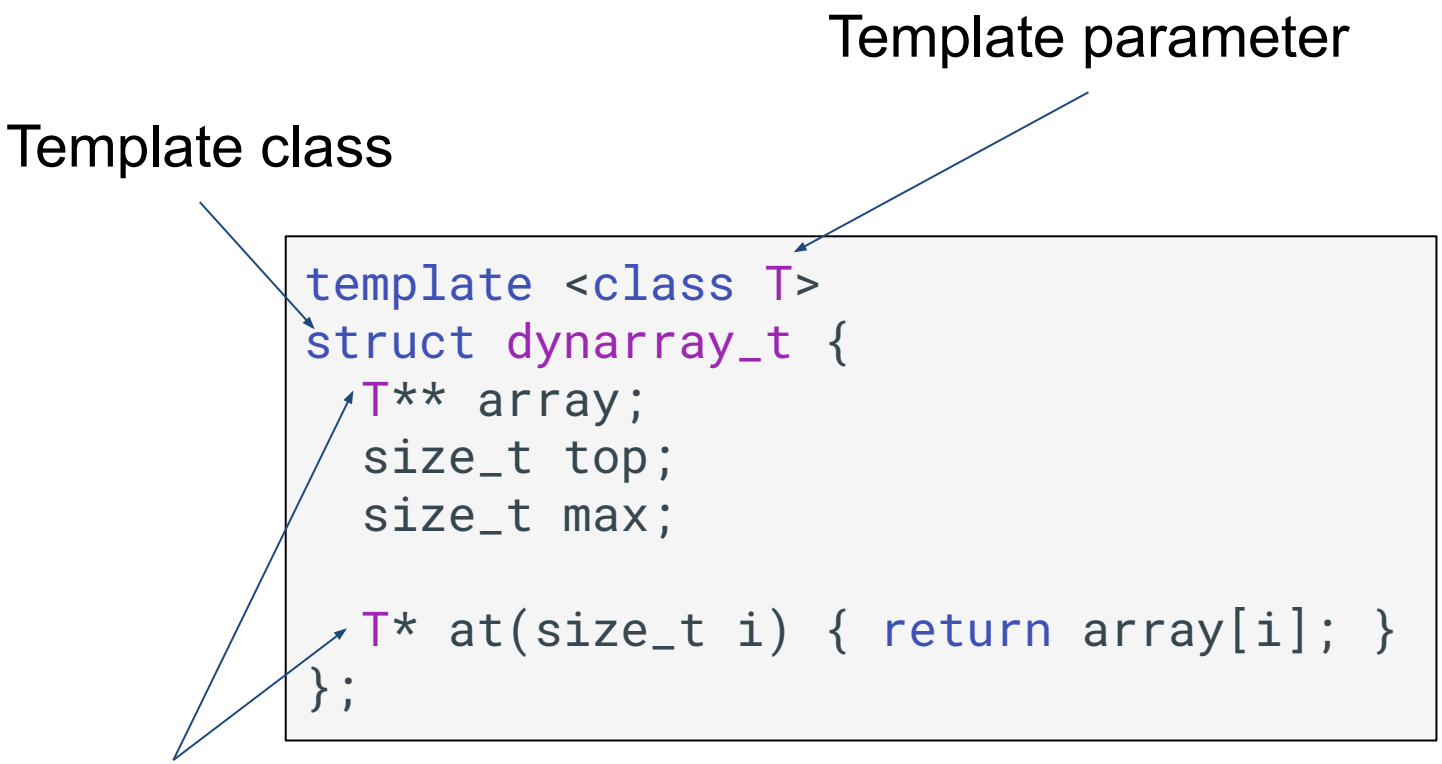
The element is not a `monster_t`!

Specific versus generic code

- Inheritance is not a solution
  - The army is defined by considering a parent class
  - And we could store children class
  - However, this would also lead to an error-prone downcast

Template

- A template is an entity parametrized by one or more template parameters



Using a template

- To use a template, give the type at declaration

```
template <class T>
struct dynarray_t {
  T** array;
  size_t top;
  size_t max;

  T* at(size_t i) { return array[i]; }
};

int main(int arg, char* argv[]) {
  dynarray_t<monster_t> army; // dynamic array of monster_t
  dynarray_t<int> tab;        // dynamic array of int
  ...
  monster_t* m = army.at(0);
}
```

Using a template

- No errors because of incorrect casts

```
template <class T>
struct dynarray_t {
  T** array;
  size_t top;
  size_t max;

  T* at(size_t i) { return array[i]; }
};

int main(int arg, char* argv[]) {
  dynarray_t<monster_t> army; // dynamic array of monster_t
  dynarray_t<int> tab;        // dynamic array of int
  // army.append(new complex_t { 1, 3 }) is an error
  monster_t* m = army.at(0);
}
```

Template and method implementation

- If the method is not implemented in the class, recall the template parameter

```
template <class T>
struct army_t {
  T** array;
  size_t top;
  size_t max;

  T* at(size_t i);
};

template <class T>
T* army_t<T>::at(size_t i) {
  return array[i];
}
```

Recall the template

And that `army_t` takes this template as parameter

Template method

- A template entity can be
  - A template class
  - Or a template method

```
template <class T>
T max(const T a, const T b) {
  return a < b ? b : a;
}

int main(int arg, char* argv[]) {
  std::cout << max(1, 3) << std::endl;       // T is int
  std::cout << max(1.12, 3.13) << std::endl; // T is double

  return 0;
}
```

Template parameters

- A template parameter can be
  - A class
  - A literal

```
template <class T, size_t n>
struct array_t {
  T elements[n];
};

int main(int arg, char* argv[]) {
  array_t<int, 4> a0;      // array_t of 4 integer
  array_t<double, 12> a1; // array_t of 12 double
  ...
}
```

Template parameters

- A template parameter can have a default value

```
template <class T = monster_t*, size_t n = 4>
struct array_t {
  T elements[n];
};

int main(int arg, char* argv[]) {
  array_t<> a0;         // array_t of 4 monster_t*
  array_t<int> a1;      // array_t of 4 int
  array_t<int, 12> a2; // array_t of 12 double
  ...
}
```

Template implementation

- A template implementation can use fields or methods of the template parameters

```
template <class T>
struct array_t {
  T v;
  void print() {
    v.print();
  }
};
```

```
struct monster_t {
  std::string name;
  void print() { std::cout << name; }
};

void f() {
  array_t<monster_t> a;
  a.print();
}
```

```
void f() {
  array_t<int> b;
  //b.print(); int does not have a
} //           print method
```

using

- `using` can be used to create a new name for a type
  - The new type can itself be a template

```
template <class T, size_t n>
struct array_t {
  T elements[n];
};

template <size_t n>
using array_of_int_t = array_t<int, n>;

using four_int_t = array_of_int_t<4>;

int main(int arg, char* argv[]) {
  array_of_int_t<4> a; // array of 4 int
  four_int_t b;        // array of 4 int
  ...
}
```

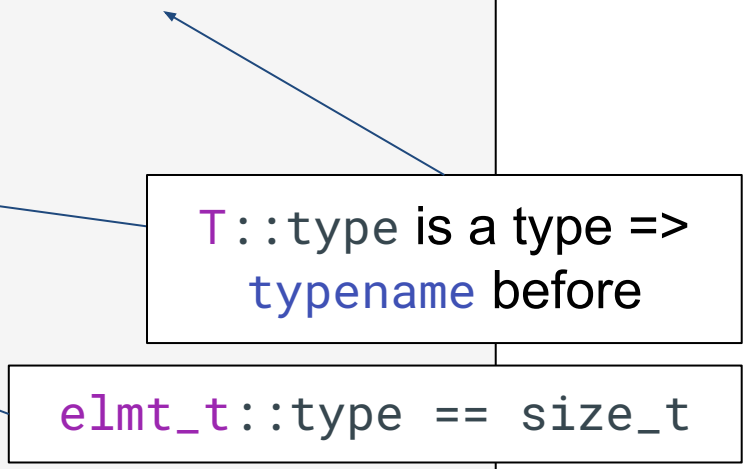
typename

- `typename` is used to tell the compiler that an syntactic element is a type

```
template<class T, typename T::type n = 4>
struct array_t {
  T elements[n];
  typename T::type size;
};

struct elmt_t {
  using type = size_t;
};

int main(int argc, char* argv[]) {
  array_t<elmt_t> a;
  return 0;
}
```



Key concepts

- A template entity is an entity parametrized by parameters
  - The entity can be a class or a method
  - The parameter can be a class or literal

```
template <class T, size_t n>
struct array_t { T elements[n]; };

void f() {
  array_t<int, 4> a;
}
```

- `using` can be used to give a new name to a type
  - Can itself be a template

- `typename` tells the compiler that a syntactic element is a type