

Instance method, new and delete

Bachelor of Science - École polytechnique

gael.thomas@inria.fr

Key concepts

- A data structure or an array is an **object**
 - The type of an object is called its **class**
 - The object `o` is **instance** of `C` \Leftrightarrow the class of the object `o` is `C`
- An **instance method** is a function defined inside a structure
 - It receives a **this** parameter named the pointer to the **receiver**
 - **this** can be omitted when we access a field of the object
- We can allocate and free an object
 - With **new** and **delete** for a data structure
 - With **new[]** and **delete[]** for an array

The C++ language

- C++ is another language based on C
 - Most of the C constructs exist in C++
 - But is not a superset: some C constructs do not exist in C++
- C++ extends the C language with new **object** abstractions
 - Better code reuse and structure
 - Allow the developer to write more generic code
- In this course, we study the **c++20** standard
 - Compile with `g++ -std=c++20`

The object abstraction

- The C++ language is based on the object abstraction
 - An object is a **data structure**
 - that can have associated **methods**
- A **method** is a **function** that acts on an object
- Main advantages:
 - Links a data structure with the code that manipulates it
 - Make the code clearer and simpler

The object abstraction

- With this definition, data structures and arrays are **objects**

```
int tab[42]; // the 42 elements of the array is an object
struct monster_t m1; // m1 is an object
struct monster_t* m2 // *m2 is an object
    = (struct monster_t*)malloc(...);
```

The object abstraction

- With this definition, data structures and arrays are **objects**

```
int tab[42]; // the 42 elements of the array is an object
struct monster_t m1; // m1 is an object
struct monster_t* m2 // *m2 is an object
    = (struct monster_t*)malloc(...);
```

- An **object** has a **type**, and we call this type its **class**
 - The class of `tab` is `int[]`
 - The classes of `m1` and `*m2` are `monster_t`
- If an object `o` has the class `C`, we say that `o` is an instance of `C`
 - `tab` is an instance of `int[]`
 - `m1` and `*m2` are instances of `monster_t`

Instance method

■ In C

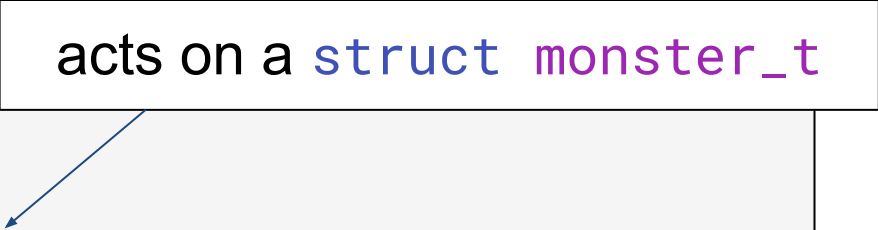
- We define a data structure
- And often a function that acts on the data structure

```
// header file
struct monster_t {
    const char* name;
    int health;
};

extern void print_monster(struct monster_t* m);

// source file
void print_monster(struct monster_t* m) {
    printf("(%s, %d)\n", m->name, m->health);
}
```

acts on a `struct monster_t`



Instance method

■ In C

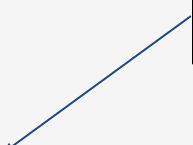
- We define a data structure
- And often a function that acts on the data structure

```
// header file
struct monster_t {
    const char* name;
    int health;
};
```

```
extern void print_monster(struct monster_t* m);
```

```
// source file
void print_monster(struct monster_t* m) {
    printf("(%s, %d)\n", m->name, m->health);
}
```

But naming a function
print_monster because it
acts on a monster is only a
convention



Instance method

■ In C

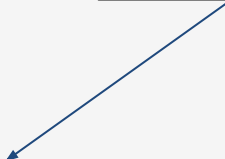
- We define a data structure
- And often a function that acts on the data structure

```
// header file
struct monster_t {
    const char* name;
    int health;
};
```

```
extern void print_monster(struct monster_t* m);
```

```
// source file
void print_monster(struct monster_t* m) {
    printf("(%s, %d)\n", m->name, m->health);
}
```

Add having a parameter with the type `struct monster_t*` seems obvious



Instance method

- C++ introduces instance methods
 - Move `print_monster` inside the structure declaration
 - Which adds it an implicit parameter with the type `monster_t*` named `this`

```
// header file
struct monster_t {
    const char* name;
    int health;

    void print();
};
```

```
// source file
void monster_t::print() {
    printf("(%s, %d)\n", this->name, this->health);
}
```

`this` is now an implicit parameter
with the type `struct monster_t*`



Instance method

- C++ introduces instance methods
 - Move `print_monster` inside the structure declaration
 - Which adds it an implicit parameter with the type `monster_t*` named `this`

```
// header file
struct monster_t {
    const char* name;
    int health;
```

```
    void print();
};
```

```
// source file
```

```
void monster_t::print() {
    printf("(%s, %d)\n", name, health);
}
```

`this` can be omitted



Instance method

- C++ introduces instance methods
 - Move `print_monster` inside the structure declaration
 - Which adds it an implicit parameter with the type `monster_t*` named `this`

```
// header file
struct monster_t {
    const char* name;
    int health;

    void print();
};
```

`print` is the `print` method that
belongs to `monster_t`
=> no need to name it
`print_monster`

```
// source file
void monster_t::print() {
    printf("(%s, %d)\n", name, health);
}
```

Instance method

- C++ introduces instance methods
 - Move `print_monster` inside the structure declaration
 - Which adds it an implicit parameter with the type `monster_t*` named `this`

```
// header file
struct monster_t {
    const char* name;
    int health;

    void print();
};
```

```
// source file
void monster_t::print() {
    printf("(%s, %d)\n", name, health);
}
```

We now say that `print` is an
instance method
of the class `monster_t`

Using an instance method

- Call an instance method with `var.f()`

```
int main(int argc, char* argv[]) {  
    struct monster_t m = { "Pikachu", 42 };  
  
    m.print(); // this in monster_t::print points to m  
               // like a call to monster_t::print(&m);  
    return 0;  
}
```

We say that the object `m` is the **receiver** of the method call
=> `this` is a pointer to the **receiver**

Code simplification

- In C++, we can also get rid of `struct` when we use the type `monster_t`

```
int main(int argc, char* argv[]) {  
    monster_t m = { "Pikachu", 42 };  
  
    m.print(); // this in monster_t::print points to m  
               // like a call to monster_t::print(&m);  
    return 0;  
}
```

Code simplification

- In C++, the `=` is also useless: initialize the fields of `m` with the parameters between the braces without `=`

```
int main(int argc, char* argv[]) {  
    monster_t m { "Pikachu", 42 };  
  
    m.print(); // this in monster_t::print points to m  
               // like a call to monster_t::print(&m);  
    return 0;  
}
```


The `new` keyword

- Allocating and initializing a data structure remains painful
 - `malloc` takes the allocated size as argument
 - Its result has to be casted into a `monster_t`
 - And the fields have to be initialized manually

```
int main(int argc, char* argv[]) {  
    monster_t* m = (monster_t*)malloc(sizeof(*m));  
  
    m->name = "Pikachu";  
    m->health = 42;  
  
    m->print();  
  
    ...  
}
```

The `new` keyword

- `new`: simplifies the allocation code
 - Allocates the data structure without explicitly giving its size
 - And initializes the fields in the same statement

```
int main(int argc, char* argv[]) {  
    monster_t* m = new monster_t { "Pikachu", 42 };  
  
    m->print();  
  
    ...  
}
```

The `delete` keyword

- Use `delete` instead of `free` to free a data structure allocated with `new`

```
int main(int argc, char* argv[]) {  
    monster_t* m = new monster_t { "Pikachu", 42 };  
  
    m->print();  
  
    delete m;  
    ...  
}
```

Dynamically allocated arrays

- Similarly, allocate / free an array with `new[]` / `delete[]`
 - With an explicit size in `new`

```
int main(int argc, char* argv[]) {  
    monster_t* m = new monster_t[2];  
  
    m[0] = { "Pikachu", 42 };  
    m[1] = { "Blastoise", 83 };  
  
    delete[] m;  
  
    return 0;  
}
```

Dynamically allocated arrays

- Similarly, allocate / free an array with `new[]` / `delete[]`
 - With an explicit size in `new`
 - Or with an implicit size because of the initializer

```
int main(int argc, char* argv[]) {  
    monster_t* m = new monster_t[] {  
        { "Pikachu", 42 },  
        { "Blastoise", 83 }  
    };  
  
    delete[] m;  
  
    return 0;  
}
```

Key concepts

- A data structure or an array is an **object**
 - The type of an object is called its **class**
 - The object `o` is **instance** of `C` \Leftrightarrow the class of the object `o` is `C`
- An **instance method** is a function defined inside a structure
 - It receives a **this** parameter named the pointer to the **receiver**
 - **this** can be omitted when we access a field of the object
- We can allocate and free an object
 - With **new** and **delete** for a data structure
 - With **new[]** and **delete[]** for an array