Constructors and destructors

Bachelor of Science - École polytechnique

gael.thomas@inria.fr

Key concepts

- A constructor
  - Is a method of a class that has the name of the class
  - Used to initialize the fields of an object
  - A class can have several constructors with different parameters

- A destructor
  - Is a method that has the name of the class prefixed by a tilde
  - Called when the object is destroyed

Constructors

- When we allocate an object, we often have to
  - Pre-initialize some of the fields at fix values
  - Or to execute some code

```
struct monster_t {
  const char* name;
  int health;
  int experience; // has to be initialized to 0

  void print();
};

int main(int argc, char* argv[]) {
  monster_t monster { "Pikachu", 42, 0 };

  return 0;
}
```

Error-prone

Constructors

- To initialize the fields of an object, use a constructor
  - A method with the name of the class
  - And with initialization parameters

```
struct monster_t {
  const char* name;
  int health;
  int experience;

  monster_t(const char* name, int health);

  void print();
};
```

A constructor

Constructor implementation

- Three part in a constructor
  - A declaration
  - Followed by a set of field initializers that come after a colon
    - Initialize the fields like we initialize an object with braces
  - A body that can contain more complex code

```
monster_t::monster_t(const char* name, int health)
  : name { name },
    health { health },
    experience { 0 } {
  // more complex initialisation code goes here
}
```

name of this

The field name of this is initialized
with the value of the parameter name

Using a constructor

- Using a constructor is transparent
  - Use it exactly as we use a list initializer for the fields when the structure does not have a constructor

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

Call `monster_t::monster_t(const char* name, int health)`
with the parameters `"Pikachu"` and `42`

=> `m.experience` is initialized to `0`

Using a constructor

- As soon as a constructor exists, we have to use it
  - Cannot use `{ "Pikachu", 42, 0 }` anymore

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

Call `monster_t::monster_t(const char* name, int health)` with the parameters `"Pikachu"` and `42`

=> `m.experience` is initialized to `0`

Chained constructors

- We can have several constructors with different parameters
  - And we can chain them

```
struct monster_t {
  const char* name;
  int health;
  int experience;

  monster_t(const char* name, int health);
  monster_t(const char* name, int health, int experience);
};

monster_t::monster_t(const char* name, int health)
  : monster_t(name, health, 0) { } // chained to second constructor

monster_t::monster_t(const char* name, int health, int experience)
  : name { name }, health { health }, experience { experience } { }
```

Default parameters

- We can achieve the same goal with default parameters

```
struct monster_t {
  const char* name;
  int health;
  int experience;

  monster_t(const char* name, int health, int experience = 0);
};

monster_t::monster_t(const char* name, int health, int experience)
  : name { name }, health { health }, experience { experience } {
}
```

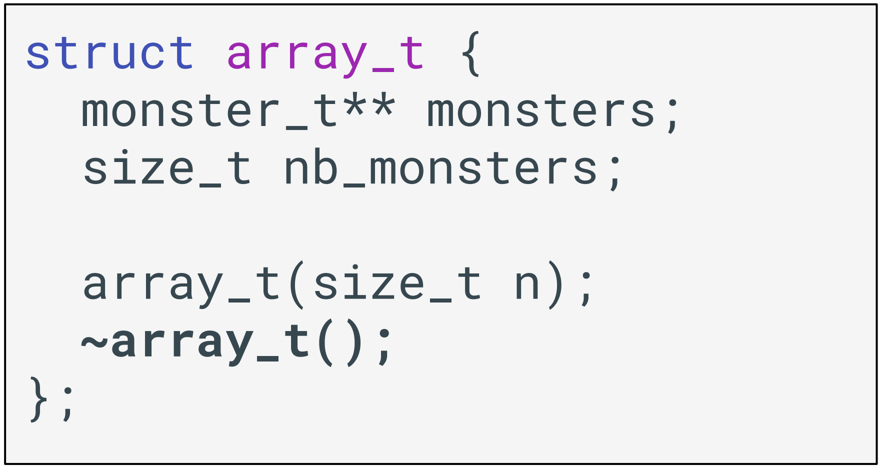
Advanced constructor

- A constructor can execute any operation in its body

```
struct array_t {
  monster_t** monsters;
  size_t nb_monsters;

  array_t(size_t n);
};

array_t::array_t(size_t n) {
  monsters = new monster_t*[n];
  nb_monsters = n;
}

int main(int argc, char* argv[]) {
  array_t array { 78 };
}
```

Destructor

- In this case, the memory has to be freed when the object is destroyed
  - Use a destructor
  - The destructor is a method named with the type prefixed with ~



```
struct array_t {
  monster_t** monsters;
  size_t nb_monsters;

  array_t(size_t n);
  ~array_t();
};
```

Destructor

- Implementation of a destructor: like any other method

```
array_t::array_t(size_t n) {
  monsters = new monster_t*[n];
  nb_monsters = n;
}

array_t::~array_t() {
  delete[] monsters;
}
```

Destructor

- The destructor is called
  - When we call delete
  - Or when a variable is destroyed (e.g., return from a call frame)

```
void test() {
  array_t x { 4 };
  array_t* p = new array_t { 4 };

  delete p; // destructor of p called here

  // destructor of x called when the
  // function returns
}
```

Key concepts

- A constructor
  - Is a method of a class that has the name of the class
  - Used to initialize the fields of an object
  - A class can have several constructors with different parameters

- A destructor
  - Is a method that has the name of the class prefixed by a tilde
  - Called when the object is destroyed