NAMESPACES AND FUNCTION OVERLOADING

Name collisions and how to avoid them

Florian Warg, Max Staff

April 27, 2017

NAME COLLISIONS

- a project uses libraries A and B
- A and B offer functions with similar names and arguments

```
/* Library A */
digest hash(block input) { return md5(input); }
/* Library B */
digest hash(block input) { return sha1(input); }
/* User code */
hash(my_message);
```

- the compiler cannot deduce which function you want to call
- this is called a name collision
- it will cause a compiler error

NAMESPACES

- libraries use namespaces to avoid name collisions
- namespaces can contain functions, types and objects
- symbols can be accessed with ns::symbol

```
namespace LibA {
    digest hash(block input);
}
namespace LibB {
    digest hash(block input);
}
/* call hash from Lib A */
LibA::hash(my_message);
/* call hash from Lib B */
LibB::hash(my_message);
```

NESTED NAMESPACES

- namespaces can also contain other namespaces
- this can be used to organize software into packages

```
namespace Lib {
    namespace crypto {
        digest hash(block input) { ... }
    }
}
Lib::crypto::hash(my_message);
```

SPLIT NAMESPACES

- namespaces can be opened and closed at any point
- symbols inside the same namespace are visible to each other

```
namespace A {
    int x;
} /* namespace A */

namespace A {
    int y;
} /* namespace A */
```

IMPORT SYMBOLS

- symbols can be imported into the local namespace
- achieved with a ”using directive”

```
1 #include <string>
2 using std::string;
3 int main() {
4     std::string s1;
5     string s1; // imported std::string as string
6 }
```

IMPORT NAMESPACES

- whole namespaces can be imported, too
- this will import all symbols

```
1 #include <iostream>
2 #include <string>
3 using namespace std;
4 int main() {
5     string name; // std::string
6     cin >> name; // std::cin
7     cout << "Hello, " << name << "\n";
8 }
```

BEST PRACTICE

- global namespace should have as few symbols as possible
- never import symbols in header files
- full symbol name is easier to understand in code reviews
- import single symbols rather than namespaces
- in production code almost always use full names

SCOPES

- each name in C++ has restricted validity (function, namespace)
- a scope is the portion of source code where a certain name (e.g. variable) is present
- begins at declaration
- ends at closing curly brace
- what about custom scopes?

BLOCK SCOPE

```
1  int main() {
2      int a = 0; ++a;
3      {
4          int a = 1;
5          a = 42;
6      }
7      std::cout << a << "\n"; // ?
8  }
9  int b = a; // Error?
```

FUNCTION OVERLOADING

- two functions to achieve the same goal for different arguments
- example: outputting to command line with std::cout
- what about optional parameters with default values?

DEFAULT ARGUMENTS

```
1  float round(float a, float b = 1) {
2      return a - (a % b);
3  }
4
5  round(10.5);
6  round(11.3579, 0.1);
7  round(12345f, 10f);
8
9  // this is incorrect:
10 float round(float b = 1, float a) {
11     return a - (a % b);
12 }
```

DIFFERENT AMOUNT OF ARGUMENTS

```
1  float distance(float a, float b) {
2      return std::sqrt(a*a + b*b);
3  }
4
5  float distance(float a, float b, float c) {
6      return std::sqrt(a*a + b*b + c*c);
7  }
8
9  distance(5, 3);
10 distance(5, 3, 7);
```

DIFFERENT TYPE OF ARGUMENTS

```
1  float round(float a, float b = 1) {
2      return a - (a % b);
3  }
4
5  int round(int a, int b = 1) {
6      return a - (a % b);
7  }
8
9  round(5.7);
10 round(4.222, 0.1);
11 round(355);
12 round(12345, 10);
```

CONCLUSIONS

- in C++ not only the function name is relevant
- compiler also checks arguments and return type

CLASSES

- having functions for own datatype assigned to it would improve readability of code
- what about OOP? About private and public? And all those more or less complicated design patterns?

CLASS DECLARATIONS

```
1  class Rectangle {
2      int width, height;
3  public:
4      int area();
5  }
6
7  int Rectangle::area() {
8      return this->width * height;
9  }
10
11 Rectangle a;
12 a.area();
```

CONSTRUCTORS

```
1  class Rectangle {
2      int width, height;
3  public:
4      Rectangle();
5      int area();
6  }
7
8  Rectangle::Rectangle() {
9      this->width = 0;
10     this->height = 0;
11 }
```

CONSTRUCTORS

```
1  class Rectangle {
2      int width, height;
3  public:
4      Rectangle(int a, int b);
5      int area();
6  }
7
8  Rectangle::Rectangle(int width, int height) {
9      this->width = width;
10     this->height = height;
11 }
```

CONSTRUCTORS

```
1  class Rectangle {
2      int width, height;
3  public:
4      Rectangle(int a, int b);
5      int area();
6  }
7
8  int Rectangle::area() { return this->width * height; }
9
10 Rectangle::Rectangle(int a, int b) :
11     width(a), height(b) {}
12
13 Rectangle a;
14 a.area();
```

RAII

- Constructor gets called on declaration!
- keyword new creates object on heap
  - needs to be freed later with delete!
  - item does not get deleted when going out of scope
- benefits: can’t forget to call constructor
- also we can use destructors now

DESTRUCTORS

```
1  class Rectangle {
2      int width, height;
3      FILE* secretLogForNSA;
4  public:
5      Rectangle();
6      ~Rectangle();
7  }
8
9  Rectangle::Rectangle() {
10     secretLogForNSA = fopen("log.txt", "wa");
11 }
12
13 Rectangle::~Rectangle() {
14     fclose(secretLogForNSA);
15 }
```

INHERITANCE

```
1  class A {
2  public: int data;
3  private: int privateData;
4  }
5
6  class B : public A {
7  public: int metaData;
8  }
9
10 class C : private A {
11 public: int otherData;
12 }
13
14 A a; a.data;
15 B b; b.data;
16 C c; c.data;
```

MULTIPLE INHERITANCES

```
1  class A {
2  public: int data;
3  }
4
5  class B {
6  public: int metaData;
7  }
8
9  class C : private A, public B {
10 public: int otherData;
11 }
```

FRIENDS

```
1  class B;
2
3  class A {
4  friend B;
5  private: int data;
6  }
7
8  class B {
9  private:
10     int metaData;
11     A other;
12 public:
13     int data() { return this->other.data; }
14 }
```

NEW AND DELETE

```
1 class A {
2 private: int data;
3 }
4
5 // put object of type A in heap:
6 A* a = new A();
7 delete a;
```