Object-Oriented Programming

Scope

Michael L. Collard, Ph.D.

Department of Computer Science, The University of Akron

Variable

A variable is an abstract storage location paired with an associated symbolic name, which contains some known or unknown quantity of information referred to as a value

- *storage location* is different than *symbolic name*

- *name binding* Associating a *symbolic name* with a *storage location*

Scope

The scope of a name binding is the portion of source code in which a binding of a name with an entity applies

- Where in the program the name can be used to refer to the entity

- In other parts of the program, the name may refer to another location, i.e., it is not the same variable

- Scope is the visibility of the *name binding*

- *lexical scope* or *static scope* portion of the source code, i.e., the area of text

Scope Levels

- Global Scope
- Module/Class Scope
- File Scope
- Function Scope
- Block Scope
- Expression Scope

Expression Scope

```
int copyFile(std::string_view inputFilename,
             std::string_view outputFilename);

int copyFile(std::string_view input,
             std::string_view output) {
    // ...
}
```

- Name binding is limited to the expression
- Most common occurrence is in *function declarations* or *function prototypes*, known as *function protocol scope*
- In a function declaration, the scope of parameter names ends right before the semicolon
- Parameter names must be unique only in the parameter list
- Parameter names are a form of documentation
- May be more expressive in *function declaration* than in *function definition*

Block Scope

```
{
    int n = 5;
    assert(n == 5);

    {
        int n = 6;
        assert(n == 6);
    }
    assert(n == 5);
}
```

- Name binding is limited to the block statement of the declaration

- An *automatic variable* or *local variable* is allocated where it is declared and deallocated automatically when the program leaves the variable's scope

- *visibility* of a variable may be hidden by *shadowing*

Function Scope

```
bool palindrome(const char* p, const char* q) {

    --q;
    while (p < q) {
        if (*p++ != *q--)
            return false;
    }

    return true;
}
```

- Name binding is limited to the block of the function

- Similar to *block scope* but for functions

- How *local variables* for a function are created

File Scope

```
const int MAX_STUDENTS = 30;
```

- Name binding is available in the file of the declaration
- Available where declared until the end of the file
- Created when the program starts and destroyed when it ends
- In essence, *global* to the file
- Order of construction is the order of declaration
- Most use is for constants needed in the rest of the file
- Use of the specifier `static` or an *anonymous namespace*
- Mostly seen in C/C++

Proper File Scope Variables

```cpp
// fileScope.cpp
int MAX_STUDENTS = 30;
```

```cpp
// fileStatic.cpp
static int MAX_STUDENTS = 30;
```

```cpp
// fileNamespace.cpp
namespace {
    int MAX_STUDENTS = 30;
}
```

- File scope variables **may be accessible by other files in the program**

- To ensure scope is limited to the file with the declaration, use either the specifier `static` or an *anonymous namespace*

- Prefer an *anonymous namespace*

C++ Function Naming Tools

Description	Command
Compile to Assembly	`g++ -S overloading.cpp`
Compile to x86 Assembly on Apple Silicon Mac	`g++ -target x86_64 -S overloading.cpp`
List symbols in object file	`nm overloading`
Demangle a symbol	`c++filt __Z6actioni`
Demangle all symbols in object file	`nm --demangle overloading`

nm Symbol Types

Symbol	Definition	Description
U	Undefined	External storage, found during linkage
D	Data section	Stored locally, available globally
d	Data section local	Stored locally, available locally only
T	Text (code) section	
S	Another section	Uninitialized or zero-initialized data section for small objects

```
$ cat fileScope.cpp
int MAX_STUDENTS = 30;

void helper() {}

void process() {
    double grades[MAX_STUDENTS];

    helper();
}
```

```
$ nm fileScope.o
0000000000000094 D _MAX_STUDENTS
0000000000000000 T __Z6helperv
0000000000000004 T __Z7processv
                 U ___chkstk_darwin
                 U ___stack_chk_fail
                 U ___stack_chk_guard
0000000000000000 t ltmp0
0000000000000094 d ltmp1
0000000000000098 s ltmp2

$ nm --demangle fileScope.o
0000000000000094 D _MAX_STUDENTS
0000000000000000 T helper()
0000000000000004 T process()
                 U ___chkstk_darwin
                 U ___stack_chk_fail
                 U ___stack_chk_guard
0000000000000000 t ltmp0
0000000000000094 d ltmp1
0000000000000098 s ltmp2
```

```
$ cat fileStatic.cpp
static int MAX_STUDENTS = 30;

static void helper() {}

void process() {
    double grades[MAX_STUDENTS];

    helper();
}
```

```
$ nm fileStatic.o
0000000000000000 T __Z7processv
0000000000000094 d __ZL12MAX_STUDENTS
0000000000000090 t __ZL6helperv
                 U ___chkstk_darwin
                 U ___stack_chk_fail
                 U ___stack_chk_guard
0000000000000000 t ltmp0
0000000000000094 d ltmp1
0000000000000098 s ltmp2

$ nm --demangle fileStatic.o
0000000000000000 T process()
0000000000000094 d MAX_STUDENTS
0000000000000090 t helper()
                 U ___chkstk_darwin
                 U ___stack_chk_fail
                 U ___stack_chk_guard
0000000000000000 t ltmp0
0000000000000094 d ltmp1
0000000000000098 s ltmp2
```

```
$ cat fileNamespace.cpp
namespace {
    int MAX_STUDENTS = 30;
}

namespace {
    void helper() {}
}

void process() {
    double grades[MAX_STUDENTS];

    helper();
}
```

```
$ nm fileNamespace.o
0000000000000000 T __Z7processv
00000000000000a4 d __ZN12_GLOBAL__N_112MAX_STUDENTSE
00000000000000a0 t __ZN12_GLOBAL__N_16helperEv
                 U ___chkstk_darwin
                 U ___stack_chk_fail
                 U ___stack_chk_guard
0000000000000000 t ltmp0
00000000000000a4 d ltmp1
00000000000000a8 s ltmp2

$ nm --demangle fileNamespace.o
0000000000000000 T process()
0000000000000094 d (anonymous namespace)::MAX_STUDENTS
0000000000000090 t (anonymous namespace)::helper()
                 U ___chkstk_darwin
                 U ___stack_chk_fail
                 U ___stack_chk_guard
0000000000000000 t ltmp0
0000000000000094 d ltmp1
0000000000000098 s ltmp2
```

Module/Class Scope

```
class YAMLParser {
public:

    // parse the next key
    void parseKey(std::string& name);

private:
    std::string buffer;
    std::string::const_iterator curPos;
};

// parse the next key
void YAMLParser::parseKey(std::string& name) {

    // ...
    name.assign(curPos, endName);
    // ...
}
```

- Name binding is limited to the *class definition* and *method definition*

- In non-OOP languages, this is accomplished by *modules*

- In the *class definition*, names must be *declared* before they are used

- *method definitions* have access to all of the *fields* (*data members*) of their class, i.e., the fields are in *scope*

Global Scope

```cpp
// module.cpp
int callCount = 0;

void process() {

    // ...

    ++callCount;
}

// main.cpp
#include "module.hpp"
#include <cassert>
extern int callCount;

int main() {

    assert(callCount == 0);

    process();
    assert(callCount == 1);

    process();
    assert(callCount == 2);

    return 0;
}
```

- Name binding is available to more than one source-code file of a program
- Called *global variables*, as with *File Scope*
- File with storage location declared normally
- File using it externally uses the `extern` specifier
- Often, the entire program has access to the variable
- Created when the program starts and destroyed when it ends

Global Scope

```cpp
// module.cpp
int callCount = 0;

void process() {

    // ...

    ++callCount;
}

// main.cpp
#include "module.hpp"
#include <cassert>
extern int callCount;

int main() {

    assert(callCount == 0);

    process();
    assert(callCount == 1);

    process();
    assert(callCount == 2);

    return 0;
}
```

```
$ nm module.o
0000000000000000 T __Z7processv
0000000000000038 S _callCount
...
$ nm main.o
                 U __Z7processv
                 U ___assert_rtn
                 U _callCount
0000000000000000 T _main
...
```

Global Scope Problems

```cpp
// create.cpp
#include "create.hpp"
#include <string>
#include <map>

namespace {
    std::map<int, std::string> data;
}

bool create(int n) {
    data.insert(std::make_pair(n, "Some Label"));

    return true;
}
```

```cpp
// main program
#include "create.hpp"

namespace {
    const auto result =  create(1);
}

int main() {

    bool result = create(1);

    return 0;
}
```

- All global variables are created before entering the `main()` function
- However, the order of creation between different translation units is not specified by the C++ Standard
- This code *segfaults* when run
- Known as the *static initialization order fiasco*

Scope & Design

A good way to improve a design is to minimize scope whenever possible

- Choose a *local variable* over a *parameter* or a *field*
- Declare the variable in the innermost block possible
- If possible, declare local variables where they first receive a value
- Replace all global variables with class members
- In general, reduce the amount of code that has access to a variable

Choose a *local variable* over a *parameter* or *field*

```
char delim;
auto simpleString = removeQuotes("'abc'", delim);

std::string_view removeQuotes(std::string_view s, char delim) {

    // must have at least two characters (opening and closing quotes)
    if (s.size() < 2)
        return ""sv;

    // check the first character for valid quote
    delim = s.front();
    if (delim != '"' && delim != '\'')
        return ""sv;

    // the string must also end with the same quote character
    if (s.back() != delim)
        return ""sv;

    // return the substring that excludes the opening and closing quotes
    return s.substr(1, s.size() - 2);
}
```

```
auto simpleString = removeQuotes("'abc'");

std::string_view removeQuotes(std::string_view s) {

    // must have at least two characters (opening and closing quotes)
    if (s.size() < 2)
        return ""sv;

    // check the first character for valid quote
    char delim = s.front();
    if (delim != '"' && delim != '\'')
        return ""sv;

    // the string must also end with the same quote character
    if (s.back() != delim)
        return ""sv;

    // return the substring that excludes the opening and closing quotes
    return s.substr(1, s.size() - 2);
}
```

Declare the variable in the innermost block possible

```cpp
int total = 0;
int n;
while (total < 1000) {
    std::cin >> n;
    total += n;
}
```

```cpp
int total = 0;
while (total < 1000) {
    int n;
    std::cin >> n;
    total += n;
}
```

Declare the variable in the innermost block possible

```
int middlePoint;
if (digits) {
    middlePoint = std::stoi(value) / 2;
    std::cout << middlePoint << '\n';
} else {
    middlePoint = value.size() / 2;
    std::cout << value.substr(0, middlePoint) << '\n';
}
```

```
if (digits) {
    int middlePoint = std::stoi(value) / 2;
    std::cout << middlePoint << '\n';
} else {
    int middlePoint = value.size() / 2;
    std::cout << value.substr(0, middlePoint) << '\n';
}
```

Scenario: Name Interface Concepts

Concepts	Attributes	Notes
First name	`value`	
Middle name	`value`	
Last name	`value`	

Interface

```
std::string name;
// ...
int nameEndPosition;
getFirstName(name, nameEndPosition);
std::string_view firstName = name.substr(0, nameEndPosition);
// ...

// extract the first name out of the full name
void getFirstName(std::string_view fullName, int& nameEndPosition) {

    nameEndPosition = fullName.find(',');
    if (nameEndPosition == std::string::npos) {
        nameEndPosition = fullName.size();
        return;
    }
}
```

```
std::string name;
// ...
std::string firstName;
getFirstName(name, firstName);
// ...


// extract the first name out of the full name
void getFirstName(std::string_view fullName, std::string& firstName) {

    const auto nameEndPosition = fullName.find(',');
    if (nameEndPosition == std::string::npos) {
        firstName = fullName;
        return;
    }

    firstName = fullName.substr(0, nameEndPosition);
}
```