COMP6771
Advanced C++ Programming

3.1 - Scopes

What is a Scope?

- The scope of a variable is the part of the program where it is accessible
  - Scope starts at variable definition
  - Scope (usually) ends at the next }
  - You're probably familiar with this even if you've never seen the term
- Define variables as close to first usage as possible
- This may be the opposite of what you've been previously taught.
  - Defining all variables at the top is especially bad in C++

```
#include <iostream>

int i = 1;
int main() {
   std::cout << i << "\n"; // prints 1
   if (i > 0) {
      int i = 2;
      // int &ri = i; // create a reference
      std::cout << i << "\n"; // prints 2
      {
         int i = 3;
         std::cout << i << "\n"; // prints 3
      }
      std::cout << i << "\n"; // prints 2
   }
   std::cout << i << "\n"; // prints 1
}
```

New Scopes

- `if`-statements
- `switch`-statements
- Loops
- Compound statements
  - i.e., Random Braces
- Function bodies
- Class types
- Namespaces
- Global scope

```
int i; // global scope

namespace N { // namespace scope
   int j = i;
}

struct point { // struct scope
   int x;
   int y;
};

int main() {  // new function scope
   if (int k = i; k >= 0) {
      // new if-statement scope
   }
   for (int m = 0; m < N::j; ++m) {
      // new loop scope
   }
}
```

Namespaces

- Used to:
  - Group related names together
  - Modularise code
- Can only be created:
  - At global scope
  - At namespace scope (nested namespace)
  - Entities inside accessed with the scope operator `::`

```
// in std.h
namespace std {
    auto cout = /* definition */;
    char get_char();
}

// in std.cpp
namespace std {
    char get_char() { return 'c'; }
}

// main.cpp
int main() {
    std::cout << std::get_char() << "\n";
}
```

Nested Namespaces

- It is possible to define a namespace within another namespace.
  - The Standard Library does this with `std::chrono` and `std::ranges`
- Prefer top-level and occasionally two-tier namespaces to multi-tier.
- It's okay to own multiple namespaces per project if they logically separate things.
- Nested entities accessed by chaining `::`

```
// in std.h
namespace std {
   auto cout = /* definition */;
   namespace chars {
      char get_char();
   }
}

// in std.cpp
// legal syntax since C++17
namespace std::chars {
   char get_char() { return ‘c’; }
}

// main.cpp
int main() {
   std::cout << std::chars::get_char()<<“\n”;
}
```

Inline Namespaces

- It is possible to define nested namespaces as being `inline`.
- This injects all names inside the nested namespace into the enclosing one.
- Can still access names via the fully-qualified name.
- Useful for symbol versioning:
  - ”Default” version of a type, etc., accessible through the enclosing namespace.
  - Other versions possible through nested namespaces.
  - Change what is the "current" symbol by moving it into the inline namespace.

```
namespace std {
   inline namespace curr {
      class vector {
         /* current interface of vector */
      }
   }
   namespace cpp98 {
      class vector {
         /* old interface of vector */
      }
   }
}

std::vector v1; // default: std::curr::vector
std::curr::vector v2; // same as above
std::cpp98::vector v3; // opt-in to old version
```

Anonymous Namespaces

- In C you had `static` functions that made functions local to a file.
- C++ uses anonymous namespaces to achieve the same effect.
- Functions that you don't want in your public interface should be put into anonymous namespaces.
  - Defining an anonymous namespace in a header file will still make those public though!
- Any names inside an anonymous namespace are injected into the enclosing scope automatically.

```
// in api.h
namespace api {
   char get_char();
}

// in api.cpp
namespace {
   // helper only accessible in this file
   char helper() { return ‘c’; }
}

namespace api {
   char get_char() { return helper(); }
}

// main.cpp
int main() {
   std::cout << api::get_char() << “\n”;
}
```

Namespace Aliases

- Gives a namespace a new name.
- Often good for shortening nested namespaces.
- Aliases should be descriptive.

```
namespace chrono = std::chrono;
namespace views = ranges::views;
```

Namespace `using`-Directives

- It is possible to inject all or some names in a namespace into the enclosing scope using a `using` directive.
- This should be avoided as much as possible to prevent name collisions.
  - **DO NOT** write in header files `using namespace std;`
- Limiting the scope of `using`-directives to inside functions is OK.

```
// potentially catastrophic
// using namespace std;

int main() {
   using std::vector;
   vector<int> v = {1, 2, 3};

   if (v.size()) {
      using namespace std::chrono;
      auto txt = millisecond{500};
   } else {
      // this won't work: different scope
      auto txt2 = millisecond{500};
   }
}
```

Name Lookup

- There are certain complex rules about how overload resolution works that will surprise you.
  - This is called **A**rgument-**D**ependent **L**ookup (ADL) and will not be covered in this course.
- It is best to always fully-qualify your function calls.
  - Even if you're within the same namespace!

```
namespace ex {
    struct empty {};
    void f(empty) { std::cout << “hi\n”; }
}

void f(ex::empty) { std::cout << “hi\n”; }

int main() {
    ex::empty e;

    f(e); // which function is called?
    // due to ADL, both ::f and ex::f are
    // equally callable! This is ambiguous!

    ex::f(e); // OK! Calls ex::f
    ::f(e);   // OK! Calls the global f
}
```

Object Lifecycle in C++

- An object is a piece of memory of a specific type that holds some data.
  - All variables are objects.
  - This does not add overhead because objects are stack-allocated by default.
- Object lifetime starts when it comes in scope.
  - Memory for the object is first allocated.
  - Then it is formally “constructed”.
  - Every type has at least one constructor that says how to initialise it.
- Object lifetime ends when it goes out of scope.
  - The object is formally “destructed”.
  - The memory is then deallocated.
  - Every type has exactly one destructor that knows how to destruct an object.

Object Construction

- We generally use ( ) to call functions, and { } to construct objects.
  - Functions only callable with ( )
  - Construction can use ( ) or { }
- There are cases when which is used matters:
  - Sometimes it is ambiguous between a constructor and an initialiser list.
  - See the most vexing parse
- Fundamental types have constructors too.
  - They do *nothing* by default.
  - Initialisation must be manually performed by the programmer.
  - Especially problematic with pointers.

```
void ex() {
 int i; // default ctor: uninitialised
 int i{}; // initialised with 0.
 int i{4}; // value-initialised to 4
 int i(4); // same as above.

 // is {1, 2} an initialiser list?
 // or a call to one of vector's constructors?
 // this will be interpreted as an init list
 std::vector<int> v{1, 2};

 int *p; // default ctor: uninitialised.
 // p is known as a "wild" pointer
}
```

Object Destruction

- Objects are destructed once they go out of scope.
- An object goes out of scope when:
  - Local variables: the } marking the end of its scope is encountered.
  - `static`/global variables: when the program ends
  - Dynamically-allocated variables: when the programmer calls `free()`/`delete`
- Destruction is deterministic
  - C++ is not garbage-collected.

```
auto v = std::vector<int>{1, 2, 3};

int main() {
    int *i = new int{4};

    {
        std::string str = “hi!”;
    } // str is destructed

    delete i; // the int is destructed.
}
// i would NOT have been destructed
// at the end of main since it is
// dynamically allocated.
// It is our responsibility to delete it.


// v is destructed here at program end
```

Why is the Object Lifecycle Useful?

Can you think of a thing where you always have to remember to do something when you're done?

- What happens if we forget to close a file?
- How easy to spot is the mistake?
- How easy would it be for a compiler to spot this mistake for us?
  - How would it know?
  - Through the object lifecycle rules!

Feedback (stop recording)