Ch8: namespace, enum, struct, Class Templates

namespace
OOOO

enum
OOOOOOOOOOO

struct
OOOOOOO

Class Templates
OOOOOO

namespace

User-Defined Namespaces

A naming collision occurs when two identical identifiers are introduced into the same scope, and the compiler cannot disambiguate which one to use. We can resolve it using **User-defined Namespaces**.

foo.cpp

```
// This doSomething() adds the value of its parameters
int doSomething(int x, int y) {
    return x + y;
}
```

goo.cpp

```
// This doSomething() subtracts the value of its parameters
int doSomething(int x, int y) {
    return x - y;
}
```

main.cpp

```
#include <iostream>
int doSomething(int x, int y); // forward declaration for doSomething

int main() {
    std::cout << doSomething(4, 3) << '\n';
}
```

Linker Error: Two different functions with the same name and parameters into the same scope (global scope).

One way to avoid collisions is to put your functions into your own namespaces.

namespace
○●○○

enum
○○○○○○○○○○

struct
○○○○○○○

Class Templates
○○○○○○

User-Defined Namespaces

C++ allows us to define our own namespaces via the namespace keyword. Namespace identifiers are typically non-capitalized. We can access a namespace with the scope resolution operator (::).

foo.cpp

```
namespace foo { // define a namespace named foo
    // This doSomething() belongs to namespace foo
    int doSomething(int x, int y) {
        return x + y;
    }
}
```

goo.cpp

```
namespace goo {  // define a namespace named goo
    // This doSomething() belongs to namespace goo
    int doSomething(int x, int y) {
        return x - y;
    }
}
```

▶

dec.h

```
namespace foo {
    int doSomething(int x, int y);
}
namespace goo {
    int doSomething(int x, int y);
}
```

main.cpp

```
#include <iostream>
#include "dec.h"
int doSomething(int x, int y) {   // This doSomething() belongs to global namespace
    return x * y;
}
int main() {
    std::cout << foo::doSomething(4, 3) << '\n'; // use the doSomething() that exists in namespace foo
    std::cout << goo::doSomething(4, 3) << '\n'; // use the doSomething() that exists in namespace goo
    std::cout << doSomething(4, 3);  // call doSomething(4, 3) in global namespace, same as calling ::doSomething(4, 3)
}
```

namespace
○○○●○

enum
○○○○○○○○○○○

struct
○○○○○○○

Class Templates
○○○○○○

Multiple Namespace Blocks Are Allowed

- It is possible to declare namespace blocks in multiple locations (either across multiple files, or multiple places within the same file). All declarations within the namespace are considered part of the namespace.

- The standard library makes extensive use of this feature, as each standard library header file contains its declarations inside a namespace std block contained within that header file. Otherwise, the entire standard library would have to be defined in a single header file!

namespace
○○○●

enum
○○○○○○○○○○

struct
○○○○○○○

Class Templates
○○○○○○

Stony Brook
University

Identifier Resolution from within a Namespace

If an identifier inside a namespace is used and no scope resolution is provided, the compiler will first try to find a matching declaration in that same namespace. If no matching identifier is found, the compiler will then check the containing namespace (in this case, the global namespace) to see if a match is found.

```cpp
#include <iostream>
void print() { // this print lives in the global namespace
  std::cout << " there\n";
}
namespace foo {
  void print() { // this print lives in the foo namespace
    std::cout << "Hello";
  }
  void printHelloThere() {
    print(); // calls print() in foo namespace
    ::print(); // expilictly calls print() in global namespace
  }
}

int main() {
  foo::printHelloThere();
}
```

▶

• When you write a library or code that you want to distribute to others, always place your code inside a namespace.

namespace
○○○○

enum
○○○○○○○○○○

struct
○○○○○○○

Class Templates
○○○○○○

Enumerations

Enumeration or Enumerated Type

An **Enumeration** or **Enumerated Type** is a user-defined data type to define a set of possible values as symbolic constants (called enumerators), which makes a program easy to read and maintain. C++ supports two kinds of enumerations: unscoped enumerations and scoped enumerations.

Ex: Assigning names to the integral constants

```
// 0 = red , 1 = green, 2 = blue
int main() {
  int appleColor{ 0 }; // my apple is red
  int shirtColor{ 1 }; // my shirt is green
}
```

(this isn't intuitive)

```
const int red{ 0 };
const int green{ 1 };
const int blue{ 2 };

int main() {
    int appleColor{ red };
    int shirtColor{ green };
}
```

(a bit better for reading, however, we can do even better!)

```
// no error for this syntactically valid,
// semantically meaningless expression:
int eyeColor{ 8 };
```

Using Unscopped or Scoped Enumerations

(preferred method)

Unscoped Enumerations

The keyword enum is used to define an unscoped enumeration.

```
enum EnumName {
    enumerator1,
    enumerator2,
    …
};
```

Declaration and initialization:

```
EnumName varName {enumerator};
```

or

```
EnumName varName {EnumName::enumerator};
```

```cpp
enum Color { // Here are the enumerators
    // These symbolic constants define all the values this type can hold
    // Each enumerator is separated by a comma,
    red,  // comma is optional but recommended
    green,
    blue,
}; // the enum definition must end with a semicolon

int main() {
    // Declaration and initialization of variables of an enumerated type
    Color appleColor { red };   // my apple is red
    Color shirtColor {Color::green }; // my shirt is green
    // Color socks { white }; // error: white is not an enumerator of Color
    // Color hat { 2 };     // error: 2 is not an enumerator of Color
}
```

▶

Best practice: Name your enumerated types starting with a capital letter (like other user-defined data types) and name your enumerators starting with a lowercase letter.

Unscoped Enumerations & Integral Values

Unscoped enumerations will <u>implicitly convert to integral values</u> (this is similar to the case with chars).

Each enumerator is automatically assigned an integer value based on its position in the enumerator list. By default, the first enumerator is assigned the integral value 0, and each subsequent enumerator has a value one greater than the previous enumerator.

- When an enumerated type is used in a function call or with an operator, the compiler will first try to find a function or operator that matches the enumerated type. If it cannot find a match, it will implicitly convert an enumerator to its corresponding integer value.

```cpp
#include <iostream>
enum Color {
    black, // assigned 0
    red, // assigned 1
    blue, // assigned 2
    green, // assigned 3
    white, // assigned 4
    cyan, // assigned 5
    yellow, // assigned 6
    magenta, // assigned 7
};
int main() {
    Color shirt{ blue }; // This actually stores the integral value 2
    std::cout << "Your shirt is " << shirt; // Prints 2
}
```
▶

Example: Printing Enumerator Names

```cpp
#include <iostream>
enum Color {
    black,
    red,
    blue,
};
std::string getColor(Color color) {
    switch (color) {
      case black:
          return "black";
      case red:
          return "red";
      case blue:
          return "blue";
      default:
          return "???";
    }
}
int main() {
    Color shirt{ blue };
    std::cout << "Your shirt is " << getColor(shirt) << '\n';
}
```

▶

namespace
OOOO

enum
OOOO●OOOOO

struct
OOOOOOO

Class Templates
OOOOOO

Example

```cpp
#include <iostream>
enum SortOrder {
  alphabetical,
  alphabeticalReverse,
  numerical,
};
void sortData(SortOrder order) {
  if (order == alphabetical)
    std::cout << "Forwards alphabetical order" << '\n';
  else if (order == alphabeticalReverse)
    std::cout << "Backwards alphabetical order" << '\n';
  else if (order == numerical)
    std::cout << "Numerical order" << '\n';
}
int main() {
  sortData(numerical);
}
```
▶

Equality operators (== , !=) can be used to test whether an enumeration has the value of a particular enumerator or not.

```cpp
#include <iostream>
enum SortOrder {
  alphabetical,
  alphabeticalReverse,
  numerical,
};
void sortData(SortOrder order) {
  switch (order) {
    case alphabetical:
      std::cout << "Forwards alphabetical order" << '\n';
      break;
    case alphabeticalReverse:
      std::cout << "Backwards alphabetical order" << '\n';
      break;
    case numerical:
      std::cout << "Numerical order" << '\n';
      break;
  }
}
int main() {
  sortData(numerical);
}
```
▶

Integer to Unscoped Enumerator Conversion

While the compiler will implicitly convert unscoped enumerators to an integer, it **will not** implicitly convert an integer to an unscoped enumerator.

```cpp
#include <iostream>
enum Pet {
    cat, // assigned 0
    dog, // assigned 1
    pig, // assigned 2
    whale, // assigned 3
};
int main() {
    Pet pet1 { 2 }; // Error: integer value 2 won't implicitly convert to a Pet
    pet1 = 3;     // Error: integer value 3 won't implicitly convert to a Pet

    Pet pet2 { static_cast<Pet>(2) }; // convert integer 2 to a Pet
    pet2 = static_cast<Pet>(3);     // convert integer 3 to a Pet

    std::cout << "Enter a pet (0=cat, 1=dog, 2=pig, 3=whale): ";
    Pet pet3 {};
    std::cin >> pet3; // Error: std::cin doesn't know how to input a Pet

    int input{};
    std::cin >> input; // input an integer
    Pet pet4{ static_cast<Pet>(input) }; // static_cast our integer to a Pet
}
```

Scope of Unscoped Enumerations

- Unscoped enumerations put their enumerators into the same scope as the enumeration definition itself.
- An enumerator name cannot be used in multiple enumerations within the same scope

```cpp
enum RGBColor { // this enum is defined in the global namespace
    red,    // all the enumeration names are also put into the global scope
    green,
    blue,
};

enum PrimaryColors {
    red, // Error: naming collision with the above red
    blue, // Error: naming collision with the above blue
    yellow,
};

int main() {
    RGBColor myColor1 { red };
    PrimaryColors myColor2 { yellow };
}
```

Scoped Enumeration (enum class)

Scoped Enumerations work similarly to unscoped enumerations but have two primary differences: They are **strongly typed** (they won't implicitly convert to integers) and **strongly scoped** (the enumerators are only placed into the scope region of the enumeration).

The keyword enum class is used to define a scoped enumeration. →

```
enum class EnumName {
    enumerator1,
    enumerator2,
    ...
};
```

Declaration and initialization: `EnumName varName {EnumName::enumerator};`

```cpp
#include <iostream>
enum class Color {
    red, // assigned 0
    blue, // assigned 1
};
int main() {
    Color color{blue};   // Error
    Color color{ Color::blue }; // Okay
    std::cout << Color::red << '\n'; // Error: std::cout doesn't know how to print this (will not implicitly convert to int)
    std::cout << static_cast<int>(color) << '\n'; // will print 1
}
```

Scoped Enumeration (enum class)

```cpp
#include <iostream>
enum class Pet {
    cat, // assigned 0
    dog, // assigned 1
    pig, // assigned 2
};
int main() {
    std::cout << "Enter a pet (0=cat, 1=dog, 2=pig): ";
    int input{};
    std::cin >> input; // input an integer
    Pet pet1{ static_cast<Pet>(input) }; // static_cast our integer to a Pet
    Pet pet2{input}; // As of C++17, you can directly initialize a scoped enumeration using an integral value
}
```
▶

Equality operators (== , !=) can be used to compare enumerators from within the same scoped enumeration (since they are of the same type).

```cpp
#include <iostream>
enum class Color {red, blue};
int main() {
    Color shirt { Color::red };
    if (shirt == Color::red) // this Color to Color comparison is okay
        std::cout << "The shirt is red!\n";
    else if (shirt == Color::blue)
        std::cout << "The shirt is blue!\n";
}
```
▶

std:array and Scoped/Unscoped enums

```cpp
#include <iostream>
#include <array>

enum StudentNames {
    kenny, // 0
    kyle, // 1
    stan, // 2
    butters, // 3
    cartman, // 4
    max_students // 5: used for the array length
};

int main() {
    std::array<int, max_students> testScores{};
    testScores[stan] = 76;
    for (auto score : testScores)
        std::cout << score << ' ';
}
```
▶

Unscoped enums

```cpp
#include <iostream>
#include <array>

enum class StudentNames {
    kenny, // 0
    kyle, // 1
    stan, // 2
    butters, // 3
    cartman, // 4
    max_students // 5: used for the array length
};

int main() {
    std::array<int, static_cast<int>(StudentNames::max_students)> testScores{};
    testScores[static_cast<int>(StudentNames::stan)] = 76;
    for (auto score : testScores)
        std::cout << score << ' ';
}
```
▶

Scoped enums

- Despite the benefits that scoped enumerations offer, unscoped enumerations are still commonly used in C++ because there are situations where we desire the implicit conversion to int (doing lots of static_casting get annoying).

namespace
0000

enum
0000000000

struct
0000000

Class Templates
000000

struct

namespace
○○○○

enum
○○○○○○○○○○

struct
●○○○○○○

Class Templates
○○○○○○

Defining structs and struct Objects

There are many instances in programming where we need more than one variable in order to represent something of interest. C++ comes with two user-defined data type that allows us to bundle multiple variables together into a single type: struct (short for structure) and class.

struct StructureName {
// member declarations.
member_type1 member_name1;
member_type2 member_name2;
member_type3 member_name3;
...
};

data members

StructureName objectName;
StructureName objectName
...

struct Employee {
　int id;
　int age;
　double wage;
};

Employee joe;
Employee frank;

When an Employee object is created, and the 3 data members within are created in sequential order.

Accessing Data Members

To access a specific data member, we use the member selection operator "." between the struct object name and the member's name and work with them just like normal variables.

```cpp
#include <iostream>
struct Employee {
    int id {}; // value initialization (zero initialization)
    int age {};
    double wage {};
};
```

```cpp
int main() {
    Employee joe;
    joe.id = 14;
    joe.age = 32;
    joe.wage = 60000.0;
    Employee frank;
    frank.id = 15;
    frank.age = 28;
    frank.wage = 45000.0;
    int totalAge { joe.age + frank.age };
    if (joe.wage > frank.wage)
        std::cout << "Joe makes more than Frank\n";
    else if (joe.wage < frank.wage)
        std::cout << "Joe makes less than Frank\n";
    else
        std::cout << "Joe and Frank make the same amount\n";
    frank.wage += 5000.0; // Frank got a promotion
    ++joe.age; // use pre-increment to increment Joe's age by 1
}
```

▶

namespace
○○○○

enum
○○○○○○○○○○

struct
○○●○○○○

Class Templates
○○○○○○

Initialization of a struct

Memberwise initialization where each member in the struct is initialized <u>in the order of declaration</u> is acceptable since C++14:

```cpp
struct Employee {
    int id {};
    int age {};
    double wage {};
};
int main() {
    Employee joe {2, 28, 45000.0};  // list initialization using braced list (preferred)
    Employee frank = {1, 32, 60000.0}; // copy-list initialization using braced list
    Employee alex {2, 28}; // alex.wage will be value-initialized (here 0.0)
    joe = {joe.id, 30, joe.wage}; // or simply joe.age = 30 for updating a value
}
```
▶

- If an object is initialized but the number of initialization values is fewer than the number of members, then all remaining members will be value-initialized.

- Variables of a struct type can be const, and just like all const variables, they must be initialized.

```cpp
struct Rectangle {
    double length {};
    double width {};
};
int main() {
    const Rectangle unit { 1.0, 1.0 };
    const Rectangle zero { }; // value-initialize all members
}
```
▶

namespace
OOOO

enum
OOOOOOOOOO

struct
OOOO●OOO

Class Templates
OOOOOO

Stony Brook University

Default Member Initialization

When we define a struct (or class) type, we can provide a default initialization value for each member as part of the type definition. These default member initialization values will be used if the user does not provide an explicit initialization value when instantiating an object.

```cpp
#include<iostream>
struct MyNumbers {
    int x;        // no initialization value (bad)
    int y {};     // value-initialized
    int z { 2 }; // explicit default value
};

int main() {
    MyNumbers s1; // s1.x is uninitialized, s1.y is 0, and s1.z is 2
    MyNumbers s2 {}; // value initialize s2.x, use default values for s2.y and s2.z (prefer s2 over s1)
    MyNumbers s3 {5, 6, 7}; // use explicit initializers for s3.x, s3.y, and s3.z (no default values are used)
    MyNumbers s4{1, 2}; // use explicit initializers for s4.x, s4.y, and default values for s4.z
}
```
▶

Best practice: Provide a default value for all members. This ensures that your members will be initialized even if the variable definition does not include an initializer list.

Passing structs to a Function

A big advantage of using structs over individual variables is that we can pass the entire struct object (rather than individual members) to a function by <u>only one parameter</u>. Structs are generally <u>passed by (const) reference</u> to avoid making copies.

```cpp
#include <iostream>
struct Employee {
    int id {};
    int age {};
    double wage {};
};
void printEmployee(const Employee& employee) { // note pass by reference
    std::cout << "ID:   " << employee.id << '\n';
    std::cout << "Age:  " << employee.age << '\n';
    std::cout << "Wage: " << employee.wage << '\n';
}
int main() {
    Employee joe { 14, 32, 24.15 };
    Employee frank { 15, 28, 18.27 };
    printEmployee(joe);
    std::cout << '\n';
    printEmployee(frank);
}
```

▶

Returning structs from a Function

One common way is to return <u>several values</u> from a function is to return a struct.

```cpp
#include <iostream>
struct Point3D {
    double x {};
    double y {};
    double z {};
};
Point3D add(const Point3D& p1, const Point3D& p2) {
    Point3D newPoint{p1.x+p2.x, p1.y+p2.y, p1.z+p2.z};
    return newPoint;
}
int main() {
    Point3D p1{1,2,3};
    Point3D p2{4,5,6};

    Point3D p3;
    p3 = add(p1,p2);

    std::cout << "p3: [" <<p3.x <<","<<p3.y<<","<<p3.z<<"]"<< std::endl;
}
```

▶

namespace
○○○○

enum
○○○○○○○○○○

struct
○○○○○○●

Class Templates
○○○○○○

structs with Program-Defined Members

structs (and classes) can have members that are other program-defined types.

```cpp
#include <iostream>

struct Employee {
    int id {};
    int age {};
    double wage {};
};

struct Company {
    int numberOfEmployees {};
    Employee CEO {};
};

int main() {
    Company myCompany{ 7, { 1, 32, 55000.0 } };
    std::cout << myCompany.CEO.wage; // CEO's wage
}
```
▶

(struct in the global scope)

```cpp
#include <iostream>

struct Company {
    struct Employee {
        int id{};
        int age{};
        double wage{};
    };

    int numberOfEmployees{};
    Employee CEO{};
};

int main() {
    Company myCompany{ 7, { 1, 32, 55000.0 } };
    std::cout << myCompany.CEO.wage; // CEO's wage
}
```
▶

(struct nested inside another struct)

namespace
○○○○

enum
○○○○○○○○○○

struct
○○○○○○○

Class Templates
○○○○○○

Class Templates

namespace
○○○○

enum
○○○○○○○○○○

struct
○○○○○○○

Class Templates
●○○○○○

Stony Brook
University

Class Templates

Similar to function template, a **class template** is a template definition for instantiating struct and class types.

```cpp
#include <iostream>
template <typename T>
struct Pair {
    T first{};
    T second{};
};
int main() {
    Pair<int> p1{ 5, 6 };      // instantiates Pair<int> and creates object p1
    std::cout << p1.first << ' ' << p1.second << '\n';

    Pair<double> p2{ 1.2, 3.4 }; // instantiates Pair<double> and creates object p2
    std::cout << p2.first << ' ' << p2.second << '\n';
}
```
▶

Class templates can have some members using a template type and other members using a normal (non-template) type:

```cpp
template <typename T>
struct Foo {
    T first{};    // first will have whatever type T is replaced with
    int second{};  // second will always have type int, regardless of what type T is
};
```

namespace
○○○○

enum
○○○○○○○○○○

struct
○○○○○○○

Class Templates
○●○○○○

Example

A program that uses a struct of two data members and determine the maximum of pairs of values.

```cpp
#include <iostream>
template <typename T>
struct Pair {
    T first{};
    T second{};
};

template <typename T>
T max(const Pair<T>& p) {
    return (p.first < p.second ? p.second : p.first);
}

int main() {
    Pair<int> p1{5, 6};
    std::cout << max(p1) << " is larger\n"; // call to max<int>(p1)

    Pair<double> p2{1.2, 3.4};
    std::cout << max(p2) << " is larger\n"; // call to max<double>(p2)
}
```
▶

Note: Unlike functions, structs cannot be overloaded.

namespace
0000

enum
0000000000

struct
0000000

Class Templates
0 0 ● 0 0 0

Class Templates with Multiple Template Types

```cpp
#include <iostream>

template <typename T, typename U>
struct Pair {
    T first{};
    U second{};
};


template <typename T, typename U>
void print(const Pair<T, U>& p) {
    std::cout << '[' << p.first << ", " << p.second << ']';
}


int main() {
    Pair<int, double> p1{ 1, 2.3 }; // a pair holding int, double
    Pair<double, int> p2{ 4.5, 6 }; // a pair holding double, int
    Pair<int, int> p3{ 7, 8 };     // a pair holding two ints

    print(p2); // calls function print<double, int>(double, int)
}
```
▶

Class Template Argument Deduction

Starting in C++20, when instantiating an object from a class template, the compiler can deduce the template types from the types of the object's initializer.

```cpp
#include <iostream>
template <typename T, typename U>
struct Pair {
    T first{};
    U second{};
};
template <typename T, typename U>
void print(const Pair<T, U>& p) {
    std::cout << '[' << p.first << ", " << p.second << ']';
}

int main() {
    Pair p1{ 1, 2.3 }; // instantiates Pair<int, double> (C++20)
    Pair p2{ 4.5, 6 }; // instantiates Pair<double, int> (C++20)
    Pair p3{ 7, 8 }; // instantiates Pair<int, int> (C++20)

    print(p2); // calls function print<double, int>(double, int)
}                                                        ▶
```

namespace
○○○○

enum
○○○○○○○○○○

struct
○○○○○○○

Class Templates
○○○○●○

Using Class Templates in Multiple Files

Just like function templates, class templates are typically defined in **header files** so they can be included into any code file that needs them.

main.cpp

```
#include <iostream>
#include " template.h"

int main() {
    Pair<double> p1 { 3.4, 5.6 };
    std::cout << max(p1) << " is larger\n";
}
```

template.h

```
template <typename T>
struct Pair {
    T first{};
    T second{};
};


template <typename T>
T max(Pair<T> p) {
    return (p.first < p.second ? p.second : p.first);
}
```

▶

Type Template Parameters with Default Values

Just like function parameters can have default arguments, template parameters can be given <u>default types</u>. These will be used when the template parameter is not explicitly specified and cannot be deduced.

```cpp
template <typename T=int, typename U=int> // default T and U to type int
struct Pair {
    T first{};
    U second{};
};

int main() {
    Pair<int, int> p1{ 1, 2 }; // explicitly specify class template Pair<int, int> (C++11 onward)
    Pair p2{ 1, 2 };           // deduces Pair<int, int> from the initializers (C++20)
    Pair p3;                   // uses default Pair<int, int>
}                                                                                ▶
```

Note: Although class templates has been demonstrated on structs for simplicity, everything here applies equally well to classes.