UNSW SYDNEY

MTRN2500

Tutorial 3 – Basics II & Memory

Basics II

UNSW
SYDNEY

The Standard Library

- A collection of tools built into C++
- Accessible in the ‘std::’ namespace
- Include:
  - Basic libraries like strings, tuples, or pairs
  - The standard template library (contains a bunch of data structures that you can use

Enumerations

- Enumerations (enums) are user-defined and contain a set of values
- Enumerations can be unscoped or scoped
  - Unscoped: enum EnumName {enum1, enum2, …};
  - Scoped: enum class/struct EnumName : type {enum1, enum2, …};
- Review lecture for comparison between scoped and unscoped, but we will prefer scoped enums
- Enums could be declared within a class to further restrict their access or denote purpose

Enumerations – Continued

```
1. #include <iostream>

2. enum class Direction : unsigned int {up, down, left, right, forward, backward};
3. enum struct SoccerPosition : char {goalkeeper, defender, midfielder, forward};

4. int main(){
5.     Direction dir {Direction::up};
6.     SoccerPosition pos {SoccerPosition::defender + 1};
7.     // error: no match for ‘operator+’ (operand types are ‘SoccerPosition’ and ‘int’)

8.     int i {static_cast<int>(Direction::up)};
9.     int j {static_cast<int>(SoccerPosition::defender) + 1};
10.    std::cout << i << ' ' << j << '\n'; // 0 2
11.
12.    if (dir == Direction::up) { // make comparison in enumerations
13.        std::cout << “direction is up!\n";
14.    }
15.
16.    return 0;
17.}
```

std::pair

std::pair is a class template that provides a way to store two objects as a single unit (they do not have to have the same type).

- #include <utility>
- std::make_pair
- .first
- .second

std::pair – Continued

```
1. #include <iostream>
2. #include <utility>

3. std::pair<int, std::string> returnIntAndString (int i, std::string s) {
4.     return {i, s};
5.     // return std::make_pair(i, s); //Alternatively
6.     // return std::pair<int, std::string> {i, s}; //Alternatively
7. }

8. int main()
9. {
10.    std::pair<int, std::string> p1 {1, "test1"};
11.    auto p2 {std::make_pair(2, "test2")};
12.    std::cout << p1.first << " " << p1.second << '\n';
13.    std::cout << p2.first << " " << p2.second << '\n';
14.
15.    auto p3 {returnIntAndString (3, "test3")};
16.    std::cout << p3.first << " " << p3.second << '\n’;

17.    return 0;
18.}
```

std::tuple

Class template std::tuple is a fixed-size collection of values (they do not have to have the same type)..

It is a generalization of std::pair.

- #include <tuple>
- std::make_tuple
- std::get<i>()

std::tuple – Continued

```
1. #include <iostream>
2. #include <tuple>

3. std::tuple<int, double, std::string> returnIntDoubleAndString (int i, double d, std::string s) {
4.     return {i, d, s};
5.     // return std::make_tuple(i, d, s); //Alternatively
6.     // return std::tuple<int, double, std::string> {i, d, s}; //Alternatively
7. }

8. int main()
9. {
10.     std::tuple<int, double, std::string> t1 {1, 1.1, "test1"};
11.     auto t2 {std::make_tuple(2, 2.2, "test2")};
12.     std::cout << std::get<0>(t1) << " " << std::get<1>(t1) << " " << std::get<2>(t1) << '\n';
13.
14.     std::cout << std::get<0>(t2) << " " << std::get<1>(t2) << " " << std::get<2>(t2) << '\n';
15.     auto t3 {returnIntDoubleAndString (3, 3.3, "test3")};
16.
17.     std::cout << std::get<0>(t3) << " " << std::get<1>(t3) << " " << std::get<2>(t3) << '\n';
18.
      return 0;
19.}
```

std::string

- This is part of the standard library and replaces c-style char-based strings
- This type acts like other primitives (int, double) enabling standard input and output as well as concatenation
- std::string is a class and thus has a variety of built in functions that can be accessed (std::basic_string - cppreference.com)

std::string – Continued

```
1. #include <iostream>
2. #include <string>

3. int main() {
4.     std::string firstName {};   // initialise with an empty string
5.     std::string lastName {};    // initialise with an empty string
6.
7.     std::cout << "What's your name?\n";
8.     std::cin >> firstName >> lastName;
9.
10.    std::string fullName {};    // initialise with an empty string
11.    fullName = firstName + " " + lastName;

12.    unsigned int lastNameLength {};
13.    lastNameLength = lastName.length();
14.
15.    char firstLetter {lastName[0]};
16.    char lastLetter {lastName[lastNameLength-1]};
17.
18.    return 0;
19.}
```

std::array

- A standard library container for a fixed-size contiguous array
- The size and type must be specified on creation and cannot be changed
- C-style array access and modifiers can be used on std::array
- As it is a class structure has a lot of useful functions that act over it (std::array – cppreference.com)

```
1. std::array<int, 3> myArray {1 2 3};                  // define a 3-element int array
2. std::array<double, 4> myArray2 {1.1 2.2 3.3 4.4};    // define a 4-element double array

3. myArray[0] = 3
4. std::cout << myArray2[0]
```

std::vector

- std::vector is a template class similar to array, however it supports dynamic allocation of data
- Size of the structure can change over time (it can grow when more elements need to be added)
- The type must be specified on creation and cannot be changed
- As it is a class structure has a lot of useful functions that act over it (std::vector – cppreference.com)

Ways to initialise a vector

```
1. #include <iostream>
2. #include <vector>

3. int main() {
4.
5.     std::vector<std::string> v1 {};          //empty vector
6.     std::vector<int> v2 {1, 2, 3};           //uniform initialiser
7.     std::vector<int> v3 {v2};                //uniform initialiser
8.     std::vector<int> v4 (v2);                //copy of vector
9.     std::vector<int> v5 = v2;                //equivalent to v4(v3)
10.    std::vector<double> v6 (3, 4.1);         //v6 has 3 elements with value 4.1
11.    std::vector<double> v7 (3);              //v7 has 3 elements with value 0.0
12.    std::vector<double> v8 {3};              //v8 has 1 elements with value 3.0
13.
14.    std::cout << "v2: " << v2[0] << " " << v2[1] << " " << v2[2] << '\n';
15.    std::cout << "v3: " << v3[0] << " " << v3[1] << " " << v3[2] << '\n';
16.    std::cout << "v4: " << v4[0] << " " << v4[1] << " " << v4[2] << '\n';
17.    std::cout << "v5: " << v5[0] << " " << v5[1] << " " << v5[2] << '\n';
18.    std::cout << "v6: " << v6[0] << " " << v6[1] << " " << v6[2] << '\n';
19.    std::cout << "v7: " << v7[0] << " " << v7[1] << " " << v7[2] << '\n';
20.    std::cout << "v8: " << v8[0] << '\n';

21.    return 0;
22.}
```

Useful vector operations

```
1. std::vector<std::string> v1 {};
2.
3. // empty()
4. bool isVecEmpty {v1.empty()};
5. std::cout << "isVecEmpty: " << isVecEmpty << '\n';
6.
7. // size()
8. std::size_t vecSize {v1.size()};
9. std::cout << "vecSize: " << vecSize << '\n';
10.
11.// push_back()
12.v1.push_back("Monday");
13.vecSize = v1.size();
14.std::cout << "vecSize after one push_back: " << vecSize
15.    << '\n';
16.v1.push_back("Tuesday");
17.v1.push_back("Wednesday");
18.vecSize = v1.size();
19.std::cout << "vecSize after three push_back: " << vecSize
20.    << '\n';
21.
22.
```

```
1. // []
2. std::cout << "v1[2]: " << v1[2] << '\n';

26.// copy
27.std::vector<std::string> v2 {};
28.v2 = v1;
29.for (auto elem : v2) {
30.    std::cout << elem << '\n';
31.}

32.// compare
33.if (v1 != v2) {
34.    std::cout << "v1 is not equal to v2!" << '\n';
35.} else if (v1 == v2) {
36.    std::cout << "v1 is equal to v2!" << '\n';
37.}

38.// pop_back()
39.v1.pop_back();
40.vecSize = v1.size();
41.std::cout << "vecSize after pop_back: " << vecSize
42.    << '\n';
```

std::vector vs. std::array

	std::vector	std::array
Creation	Vector is a sequential container to store elements.	Array is an original data structure based on index concept.
Length	Vector is dynamic in nature so, size increases with insertion of elements.	As array is fixed size, once initialized can't be resized.
Memory	Vector occupies more memory.	Array is memory efficient data structure.
Efficiency	Vector takes more time in accessing elements.	Array access elements in constant time irrespective of their location as elements are arranged in a contiguous memory allocation.

Memory

UNSW
SYDNEY

Pointers

- A pointer is a variable that stores a memory address
- This denotes the location of some piece of data
- Pointer must be associated with a type which may be a built-in type or a user-defined type

```
1. int* iPtr1; //OK, recommended
2. int *iPtr2; //OK
3. int * iPtr3;//OK
```

```
1. int* iPtr; double* dPtr; float* fPtr; char* cPtr; bool* bPtr;

2. class Robot{...};        // define a Robot class
3. Robot* pRobot;            // a pointer of the type Robot class
```

Pointers – Continued

- There are many ways to initialise a pointer

```
1. int num{88};             // An int variable with a value

2. int* pNum1;              // Declare a pointer variable called pNum1
3.                          // pointing to an int (or int pointer)

4. pNum1 = &num;            // Assign the address of the variable num to
5.                          // pointer pNum1

6. int* pNum2{&num};        // Declare another int pointer and init to address
7.                          // of the variable num

8. int* pNum3{nullptr};     // Initialise with null pointer (points to nothing)
```

Accessing data in a pointers

- The dereferencing operator (*) returns the value stored at the address
- Access class members by dereferencing first then using the dot operator
  - The arrow (->) operator does these two steps for simplicity

```
1. int num{88};              // An int variable with a value
2. int* pNum{&num};          // Declare an int pointer and init to address
3.                           // of the variable num
4. std::cout << "pNum: " << pNum << '\n';     // 0x7ffd99b5d1ac
5. std::cout << "*pNum: " << *pNum << '\n';   // 88


6. Circle circle{0.0, 0.0, 3.0};                // declare a circle with radius=3.0
7. std::cout << "Radius of circle: " << circle.getRadius() << '\n';
8.
9. Circle* pCircle{&circle};                    // declare a pointer to circle
10.std::cout << "Radius of circle via pCircle using (*). : " << (*pCircle).getRadius() <<  '\n';
11.std::cout << "Radius of circle via pCircle using -> : "  << pCircle->getRadius() <<  '\n';
```

Allocating memory

- Dynamic memory allocation can be done to create space on the heap
- Memory created on the heap must be manually freed (otherwise a memory leak occurs)
- C++ uses new and delete to dynamically allocate and deallocate memory
  - new[] and delete[] for array
- new (new[]): returns a pointer to the memory allocated
- delete(delete[]): takes a pointer (pointing to the memory allocated via new)

```
1. // dynamic memory allocation
2. Circle* dpCircle{new Circle(1.0, 2.0, 4.0)};        // dynamic memory allocation
3. std::cout << "Radius of circle via dpCircle: " << dpCircle->getRadius() << '\n';
4.
5. // free dynamically allocated memory
6. delete dpCircle;                                    // remember to free memory
```

Pointers and memory practices

- The usage of raw pointers and explicit memory allocation are both generally discouraged – unless explicitly specified
- These can be replaced with references or smart pointers (explored in the following section)
- These will be explored next

References

- References are an “alias” for referring to objects/variables
- It can be used as an alternative to pointers (particularly when passing arguments to functions)
- Denoted by the ampersand (‘&’) symbol
  - Beware of the different meanings of ‘&’ when used to denote the address of a variable

```
1. int& iRef1{num};  //OK, recommended
2. int &iRef2{num};  //OK
3. int & iRef3{num}; //OK
```

References – Continued

- You cannot have an uninitialized reference (it must always have a value)
- It must be initialised as a reference to an existing object
  - The object being referenced cannot be changed
- References can be used to associate with any built-in or user-defined type

```
1.  int num{88};          // An int variable with a value
2.  int& rNum1{num};      // Declare a reference variable called rNum1 referring
3.                        // to an int
4.  int& rNum2;   // error: ‘rNum2’ declared as reference but not initialized

5.  // use rNum1 as if it were num (alias)
6.  std::cout << "rNum1: " << rNum1 << ’\n’;          // 88
7.  std::cout << "&rNum1: " << &rNum1 << ’\n’;                  // 0x7ffcd5a33034
8.  std::cout << "*(&rNum1): " << *(&rNum1) << ’\n’; // 88
```

Pass by value, pass by pointer, pass by reference

```
1.  std::string foo(std::string s) {
2.      s = s + "1";
3.      return s;
4.  }
5.  std::string result{foo(str)};
```



```
1.  std::string foo(str) {
2.      std::string s{str};
3.      s = s + "1";
4.      return s;
5.  }
```

Pass by value:

- s is a local copy of str;
- A cloning operation is needed (not efficient);
- If, for some reason, we make changes to s inside the function, the changes will not reflect on str.

```
1.  std::string foo(std::string* s) {
2.      *s = *s + "1";
3.      return *s;
4.  }
5.  std::string result{foo(&str)};
```



```
1.  std::string foo(&str) {
2.      std::string* s{&str};
3.      *s = *s + "1";
4.      return *s;
5.  }
```

Pass by pointer:

- s is a pointer to str;
- A cloning operation is not needed (efficient);
- If, for some reason, we make changes to s inside the function, the changes will reflect on str.
- Pointer may be null (have to check before using)!

```
1.  std::string foo(std::string& s) {
2.      s = s + "1";
3.      return s;
4.  }
5.  std::string result{foo(str)};
```



```
1.  std::string foo(str) {
2.      std::string& s{str};
3.      s = s + "1";
4.      return s;
5.  }
```

Pass by reference:

- s is a reference to str;
- A cloning operation is not needed (efficient);
- If, for some reason, we make changes to s inside the function, the changes will reflect on str.
- Reference cannot be null (no need to check before using)!

Smart Pointers

- Smart pointers are used as a replacement to raw pointers
- The base interface is similar to that for raw pointers (e.g., dereferencing and assignment)
- Raw pointers have ownership for memory which means it is automatically released when it goes out of scope (no longer needed)
- Helpful in creating memory-leak free programs
- Found within the ‘memory’ library (`#include <memory>`)

Smart Pointers – Continued

- std::unique_ptr
  - smart pointer that retains exclusive ownership of object through pointer
- std::shared_ptr
  - smart pointer that retains shared ownership of object through pointer
- std::weak_ptr
  - smart pointer that holds non-owning reference to object managed by std::shared_ptr

- Prefer the use of std::unique_ptr if there are no other reasons

Smart Pointers Usage

```
1. Circle circle{1.0, 2.0, 3.0};
2. std::cout << "Radius of circle: " << circle.getRadius() << '\n';
3.
4. // pointer (pointing to an object on stack)
5. Circle* pCircle{&circle};
6. std::cout << "Radius of circle via pCircle: " << pCircle->getRadius() << '\n';
7.
8. // dynamic memory allocation
9. Circle* dpCircle{new Circle(1.0, 2.0, 4.0)};       // dynamic memory allocation
10.std::cout << "Radius of circle via dpCircle: " << dpCircle->getRadius() << '\n';
11.
12.// free dynamically allocated memory
13.delete dpCircle;                                    // remember to free memory

14.// smart pointer, no need to manually free
15.std::unique_ptr<Circle> spCircle{std::make_unique<Circle>(Circle(1.0, 2.0, 4.0))};
16.std::cout << "Radius of circle via spCircle: " << spCircle->getRadius() << '\n';
```

Lab tasks

UNSW
SYDNEY