University of Regina, CS 115, Winter 2025

CS 115

Dr. Joseph Eremondi, adapted from Dr. Shakil Khan, Dr. Philip Fong, and Dr. Howard Ham

Last updated: January 9, 2025

1 Introduction

1.1 Welcome to CS 115

1. What is this course all about?

   (a) Objectives

   - Various concepts of object oriented programming. Topics include: two-dimensional arrays, records, data abstraction, classes, composition and inheritance, type systems, subtyping, dynamic binding, polymorphism, pointers/references, dynamic memory management, and searching and sorting algorithms. Along the way, we may also discuss software engineering concepts, including comprehensibility, correctness, efficiency, and refactoring.

2. What is this course all about?

   (a) Major theme: Types

   - This course gives you your first real look at *types* in programming
     - How to define your own types
     - How to use types to structure programs
   - More than just `int` and `bool`

3. What is this course all about?

   (a) Major theme: Software Engineering

- CS 110 is how to write code
- CS 115 is how to write **good** code
- i.e. How to write code that is easy to:
  - Understand
  - Maintain
  - Re-use
  - Extend
- *Intentionally limiting* the ways we are allowed to use parts of our program
  - Abstraction is at the heart of computer science

4. Tentative Outline

   - Review
   - 2D arrays, records (structs), abstract data types
   - Searching and sorting
   - Object Oriented Design
     - Classes and constructors
     - Overloading and coercion
     - Composition and inheritance
   - Pointers and dynamic memory management
   - Program organization

5. Learning Objectives

   (a) By the end of this course, you should be understand
   - Data abstraction, encapsulation, polymorphism
   - Solving real world problems with object-oriented principles
   - Using composite types and abstract types to solve real-world problems
   - Approaches to searching and sorting
   - C++ OOP features
     - Operator overloading, type coercions, constructors, destructors
   - Pointers, dynamic memory management, and linked data structures
   - Modular programming with inheritance and composition

6. Grading
    - 20% assignments
    - 15% labs
    - 20% midterm
    - 45% final
        - Must pass final to pass the class
7. Midterm
    - In-class, Friday March 7
        - Might be split into two rooms
8. Labs
    - See `https://www.labs.cs.uregina.ca/115/`
9. Assignments
    - 4 Assignments
    - Submitted on URCourses
    - Sample based marking
        - You submit the whole thing, we mark part of it
10. Office Hours
    - Tuesdays 11:00-12:00
    - Thursdays 10:30-11:30
    - In RIC 317
        - Take the elevator then go across the bridge
        - or, take the stairs by the vending machines
11. Course Communication
    - In lectures
    - Announcements on URCourses
    - Course email on URCourses
        - For privately contacting instructor
        - Save for things that need to be private

        * Personal circumstances
        * Assignment solutions

12. Course Discussion Forum
    - Ask questions on URCourses!
        - Can be asked anonymously
        - If you're wondering, then other students probably are too
        - Don't post partial or complete assignment solutions on the forum
    - Good for
        - Clarification on assignments
        - Understanding course material
        - General curiosity/information beyond the lectures
13. Academic Honesty
    (a) Students are expected to complete assignments *independently*
        - No sharing solutions
        - No copying from the internet
        - No using ChatGPT, Copilot, Claude, or any other Generative AI tool.

    (b) You need to pass the final, so set yourself up for success and do the assignments
14. Other Logistics
    - Attendance expected
        - You're responsible for anything you miss
    - There are detailed course notes on URCourses
        - by Howard Hamilton and Phillip Fong
        - Excellent **free** resource
    - Lectures are a great time to **ask questions**

2 Review

2.1 Basic program structure, local/global variables, value passing semantics, strings, program dev. process

1. Hello world!

```cpp
#include <iostream>
using namespace std;

int main( ){
  cout << "Hello, World!" << endl;
  return 0;
}
```

- 4 types of control structures:
  - sequences (see above)
  - conditionals
  - loops
  - function invocations

2. Functional abstraction

   (a) Example

   ```cpp
   // Declaration of the triple function
   int triple(int x);

   int main( ){
     int answer;
     answer = triple(5);
     cout << answer << endl;
     cout << triple(2) << endl;
     return 0;
   }

   // Definition of the triple function
   int triple(int x) {
     return 3 * x;
   }
   ```

   (b) Declaration vs. Definition

   - Must declare functions before referencing them
   - use function prototype /header

- OR declare before 1st use
- Scope of a function = file scope
- Can a function call itself?!

3. Local and global variables and constants

   (a) Example

   ```
   // Declaration of a global variable
   int g;

   // Declaration of a global constant
   const int THREE = 3;

   int main( ){
     const int LOC = 29;
     int loc = LOC;
     g = 42;
     cout << g << endl;
     tripleGlobal();
     cout << g << endl;
     return 0;
   }
   ```

   (b) ctd.

   ```
   void tripleGlobal( ){
     // The local var loc is not acc.
     // The global var g is accessible
     g = THREE * g;
   }
   ```

   - Use "extern" to access global variables declared in other files

4. Conditionals (if-then-else branching)

```
int max(int a, int b){
  if (a >= b)
    return a;
  else
    return b;
}
```

```
int main( ){
  cout << max(-1, 2) << endl;
  cout << max(1, -2) << endl;
  return 0;
}
```

5. Conditionals (ternary operator cond ? b1 : b2)

   - Compare the following:

```
int max(int a, int b){
  if (a >= b)
    return a;
  else
    return b;
}

int max(int a, int b) {
  return (a >= b) ? a : b;
}
```

6. Conditionals (nesting)

   - Can be nested:

```
int inRange(int num, int low, int high) {
  if(num>=low)
    if(num<=high)
      return 1;
  return 0;
}
```

   - Note: could have used a compound conditional statement instead

7. Conditionals (else-if and switch cases)

    - Can have multiple branches:

```
int sign(int a){
  if (a > 0)
    return 1;
  else if (a < 0)
    return -1;
  else
    return 0;
}
```

8. Conditionals (else-if and switch cases)

    - Switch cases?

```
switch (month){
 case 1: case 2: case 3: case 4:
   cout << "Winter";
   break;
 case 5: case 6: case 7: case 8:
   cout << "Spring";
   break;
 case 9: case 10: case 11: case 12:
   cout << "Fall";
   break;
 default:
   cout << "Error, universe broken";
 }
```

9. Repetition structures (loops)

    - Want to compute:
    - $f(n) = 1 + 2 + 3 + \ldots + n$

```
unsigned int triangular(unsigned int n){
  unsigned int result = 0;
  for (unsigned int i = 1; i <= n; i++){
    result += i;
  }
  return result;
}
```

- Order of execution?
- Can have an empty body!

10. Repetition structures (loops)

```
const unsigned int BASE = 10;

unsigned int sumOfDigits(unsigned int m){
  unsigned int sum = 0;
  while (m != 0) {
    unsigned int digit;
    digit = m % BASE;
    sum = sum + digit;
    m = m / BASE;
  }
  return sum;
}
```

- More explicit than for loops
- Do-while: like while, but executes at least once
- Loops can be nested

11. Value passing semantics

- Call by value (arguments evaluated)

```
void doubleV(int a){
  a = a*2;
}

int main( ){
  int a = 2;
  doubleV(a+a);
  cout << a << endl;

  return 0;
}
```

12. Value passing semantics

    - Call by reference (can only send vars)

```
void doubleR(int &a){
  a = a*2;
}

int main() {
  int a = 4;
  doubleR(a);
  cout << a << endl;

  return 0;
}
```

13. Value passing semantics

    - Call by address (arguments evaluated)
      - We'll see more of this later
      - Have to explicitly get dereference
        - i.e. get value from the address

```
void doubleP(int *a){
  *a = (*a)*2;
}

int main( ){
  int a = 4;
  doubleP(&a);
  cout << a << endl;

  return 0;
}
```

14. Side effects

- Effects of a function other than the generation of a value to be returned
  - those that persist
- e.g., printing stuff using cout, changing a global variable, changing a local variable via call by reference/pointer, etc.

15. Strings

- Overloading `+` and `[]` operators
  - C++ libraries provide string facilities

```
#include <string>

int main( ){
  string h = "hello";
  string w = "world";
  string msg = h + ' ' + w;
  cout << msg << endl;
  return 0;
}
string s = "hello world";
for (int i = 0; i < s.length(); i++)
  cout << s[i] << endl;
```

16. Strings

- Characters are integer values

```
char charToUpper(char c){
  if ('a' <= c && c <= 'z')
    return c - 'a' + 'A';
  else
    return c;
}
```

17. Strings

- Passing by reference: faster than pass-by-value for large strings
- Not safe: modifying the passed string also modifies the original one
- Solution: pass by constant reference

```
string capitalize(const string &s);
```

- Occasionally, you may want to return a value by constant reference (meh!)

  ```
  const string &chooseFirst(const string &s1, const string &s2) {
     if (s1 < s2)
       return s1;
     else
       return s2;
   }
  ```

18. Code as Communication

    - Passing by constant reference doesn't add any power to the language
      - We can do *less* things with a const reference
    - This is **good**
    - Code communicates an intention
      - "This function shouldn't change this string"
    - Compiler *checks* this intention
      - Gives you an error if you violate it

19. Strings

    (a) Example

    - Function returning with non-constant reference

      ```
      string &chooseFirst(string &s1, string &s2)
      {
        if (s1 < s2)
          return s1;
        else
          return s2;
      }
      ```

```
int main(){
  string s1 ; "ABC";
  string s2 = "XYZ";
  chooseFirst(s1,s2) = "PQR"
  cout << s1;
  return 0;
}
```

(b) Ctd

- chooseFirst( ) returns reference to lexicographically smaller string
- main( ) prints PQR! since s1=PQR!

20. Modular vs. Application programs (115 vs. 110)

- Top-down design
  - repeatedly decomposing a complicated problem into smaller, easier subproblems
  - each can be implemented independently
  - e.g., decomposing a function into many smaller ones
- Alternative is bottom-up approach
  - building reusable tools
  - then using those tools to build even powerful tools
  - eventually solve original problem
- Reuse
  - reduces the overhead of solving a problem over and over again,
  - saves us from redoing testing and documentation for similar code
  - Easier to understand code
  - Code structured into modules
    - separates interface from implementation

21. Standard input and output

- Can redirect standard input and output from and to files resp.
- `myProg < inFile > outFile`

- Can pipe the standard output of a program to the standard input of another
- `myProg1 | myProg2`
- See notes for how
- `getline(cin, <string>)` and `cin.get(<char>)` can be used to read input from a file

22. Misc

    - Separate (unrelated) functions in different files; compile separately using -c command, and link together
      - `g++ -c main.cpp`
      - `g++ -c my_util.cpp`
      - `g++ -o prog.out main.o my_util.o`
    - Collect all function prototypes together in a header file and include it in main.cpp

```
#include "my_util.h"
#pragma once preprocessor
```

23. Misc

    - Assertions (debugging aid)

```
#include <cassert>
...
assert (n>0); //prog. Terminates if not
```

3 Arrays

3.1 One, two, and multi-dimensional arrays

1. Motivation

    - Print 1000 numbers in reverse order

```
int value0;
int value1;
int value2;
// ...
int value999;

cin >> value0;
cin >> value1;
// ...
cin >> value999;

cout << value999 << endl;
cout << value998 << endl;
// ...
cout << value0 << endl;
```

2. Motivation (cont'd)

   - How about 1000000 numbers?
   - Tedious, not scalable, and error prone
   - Solution: use aggregate data type
     - homogenous components
     - indexing support
     - constant time access
     - random access

```
int a[120000];     // Array declaration

for (int i = 0; i < 120000; i++)
  cin >> a[i];     // Array access
for (int i = 119999; i >= 0; i--)
  cout << a[i] << endl;
```

3. Array Operations

   - Call the things we store in the array *elements*
   - Get the ith element's value: `array[i]`
   - Set the ith element: `array[i] = someValue;`

4. Simple arrays

   ```
   const int N = 120000;
   int a[N];     // Array declaration

   for (int i = 0; i < N; i++)
     cin >> a[i];     // Array access
   for (int i = N-1; i >= 0; i--)
     cout << a[i] << endl;
   ```

   - Array size must be a constant expression
   - Easy to change size: just update N (the rest of the program remains intact)

5. Passing arrays as arguments

   ```
   int sumArray(int a[], unsigned int n) // Array argument
   {
     int sum = 0;
     for (int i = 0; i < n; i++)
       sum += a[i];
     return sum;
   }

   int main()
   {
     // Array initialization
     int a[] = { 3, 24, -88, 17, -1 };
     cout << sumArray(a, 5) << endl;
   }
   ```

   - Array size can be left unspecified in array initialization syntax

6. Passing arrays as arguments

   - Array arguments are always automatically passed by reference
   - no special notation is require

```
// int sumArray(int& a[], unsigned int n) - INCORRECT
int sumArray(int a[], unsigned int n)    // CORRECT
{
  ...
    }
```

- Works for arrays of all sizes (size is passed as a separate argument)
- Interface not safe: can modify the content of A

7. A Safer Interface

```
int sumArray(int a[], unsigned int n)
// not safe, sumArray can modify A!
```

- Use the following instead:

```
int sumArray(const int a[], unsigned int n)
```

- How to figure out array size when passing n if the size was left unspecified when declaring it?
- use sizeof function:

```
int a[] = {1,2,6,3,8};
int x = sumArray(a, sizeof(a) / sizeof(int));
```

8. Play time

- Check if integer array sorted

```
bool arrayIsSorted(const int a[], unsigned int n){
  for (int i = 0; i < n-1; i++){
    if (a[i] > a[i+1])
      return false;
  }
  return true;
}
```

9. Play time

- Reversing items in integer array

```
void swap(int &a, int &b) {
  int tmp = a;
  a = b;
  b = tmp;
}
// below a[] is not a constant as want to produce side-effect
void reverseArray(int a[], unsigned int n) {
  for (int i = 0; i < n/2; i++)
    swap(a[i], a[n - i - 1]);
}
```

10. Processing subarrays

    - Compute the sum of an array segment

```
// pos   : index of the first component in the subarray
// count: total number of components in the subarray
int sumSubarray(const int a[],
                unsigned int pos,
                unsigned int count){
  int sum = 0;
  for (int i = pos; i < pos + count; i++)
    sum += a[i];

  return sum;
}
```

11. Processing subarrays

    - Another way to do the same thing

```
// begin: index of first component in the subarray
// end   : index of the last component in the subarray
int sumSubarray(const int a[],
                unsigned int begin,
                unsigned int end){
  assert(begin <= end);
  int sum = 0;
  for (int i = begin; i <= end; i++)
    sum += a[i];

  return sum;
}
```

12. Subtleties

    - C++ does not check if array indices are within bound
    - it's your responsibility
    - Array Copying

    ```
    a = b // invalid
    ```

    - copy cell by cell:

    ```
    a[6]=b[9] // works!
    ```

13. Subtleties

    - Array Comparison

    ```
    if(a == b) // invalid
    ```

    - compare each pair of cells at a time
    - No need to return array as function output, uses call by reference anyway!

14. Prof's Aside

    - C++ arrays are *unsafe*
    - This is *terrible* language design
      - Billions of dollars and many security incidents caused by unsafe memory access
      - Error cost outweighs performance cost of checking array bounds
      - Most checks can be optimized out by the compiler
    - C++ will never change

    - Backwards compatibility
    - `std::array` is safe but isn't the default
  - Languages like Rust make sure that these errors are *impossible*
    - Unless you explicitly disable safety

15. Example

```
#include <iostream>
using namespace std;
int main(){
  char passwd[8] = "secret";
  char username[8] = "bob101";
  string toPrint = "";
  // Oops reading past end of array!
  for (int i = 0; i < 16; i++){
    toPrint += username[i];
  }
  cout << toPrint << endl;
}
```

```
bob101secret
```

3.2 Two Dimensional Arrays

1. Motivation

   - Want to store quantity of different products sold in a store
   - but for multiple locations/regions
   - Conceptually can store as a matrix, where rows represent different locations and columns represent different products
   - `sales[2][1]` are the total number of items sold for location 2 and product 1
   - recall item n is the (n+1)-th item
     - index starts from 0!

2. Declaration and Access

```
const unsigned int NUM_OF_REGIONS = 4;
const unsigned int NUM_OF_PRODUCTS = 3;
```

```
unsigned int sales[NUM_OF_REGIONS][NUM_OF_PRODUCTS];
```

- To access sales figure for first product in second region, use:

```
sales[1][0] // recall, indices start from 0
```

- e.g., want to set sales figure for first product in second region to 500

```
sales[1][0] = 500;
```

3. Populating and Accessing

```
// Read input stream
for (unsigned int region = 0; region < NUM_OF_REGIONS; region++)
  for (unsigned int product = 0; product < NUM_OF_PRODUCTS; product++)
    cin >> sales[region][product];

// total sales for a particular product (product 0)
unsigned int total_sales = 0;

for (unsigned int region = 0; region < NUM_OF_REGIONS; region++)
  // add up sales from all regions for product 0
  total_sales += sales[region][0];
```

- Can you compute total sales from region 1?

4. Passing 2D Arrays

```
unsigned int sumProductSales(
        unsigned int sales[NUM_OF_REGIONS][NUM_OF_PRODUCTS],
        unsigned int product)
{
  unsigned int total_sales = 0;
  for (unsigned int region = 0; region < NUM_OF_REGIONS; region++)
    total_sales += sales[region][product];
  return total_sales;
}
```

- Can you implement a safer interface?
- As usual, can leave size of first dimension unspecified, e.g. `int F(int arr[ ][SIZE])`
- but not the second one (why?)

5. Making things more modular

   - So we can change internal representation without changing interface

```
// Implement a function that returns
// the value of one element from the sales array
unsigned int getSales(
        const unsigned int sales[NUM_OF_REGIONS][NUM_OF_PRODUCTS],
        unsigned int r, unsigned int p){
  return sales[r][p];
}
// Implement a function that sets the value
// of one element from the sales array
void setSales(unsigned int sales[NUM_OF_REGIONS][NUM_OF_PRODUCTS],
              unsigned int r, unsigned int p, unsigned int v){
  sales[r][p] = v;
}
```

6. Using typedef

```
// too lazy to write long types? Use typedef instead!

typedef unsigned int Sales[NUM_OF_REGIONS][NUM_OF_PRODUCTS];

unsigned int sumSales(const Sales sales){
  ...
    }
```

7. Simulating Two-dimensional Arrays by One-dimensional Ones

```
unsigned int sales[NUM_OF_REGIONS][NUM_OF_PRODUCTS];
```

   - versus

```
unsigned int _sales[NUM_OF_REGIONS * NUM_OF_PRODUCTS];
```

   - Issue: how to map between these two?
     - row-major vs. column-major order
     - e.g. `sales[i][j]`
       - same as `_sales[i * NUM_OF_PRODUCTS + j]` in row-major
   - Now you know why the size of the 2nd dimension can't be left unspecified!
     - Can you write the formula for column-major order?

8. Using Row-Major Order

```
unsigned int totalSales = 0;

for (unsigned int region = 0; region < NUM_OF_REGIONS; region++)
  for (unsigned int product = 0;
       product < NUM_OF_PRODUCTS;
       product++){
    totalSales += _sales[region * NUM_OF_PRODUCTS + product];
  }
```

   - This is why we need to know the size of the second dimension
     - To calculate offset

9. Multi-dimensional Arrays

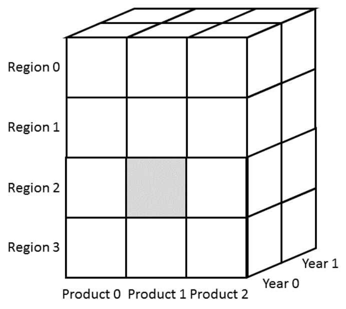
```
const unsigned int NUM_YEARS = 2;
const unsigned int NUM_REGIONS = 4;
const unsigned int NUM_PRODUCTS = 3;
```

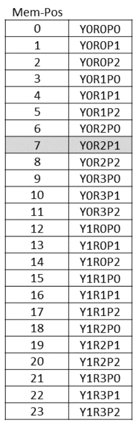
```c
typedef unsigned int Sales[NUM_YEARS][NUM_REGIONS][NUM_PRODUCTS];

unsigned int total_sales = 0;
for (unsigned int year = 0; year < NUM_YEARS; year++)
  for (unsigned int region = 0; region < NUM_REGIONS; region++)
    for (unsigned int product = 0; product < NUM_PRODUCTS; product++)
      total_sales += sales[year][region][product];
```

10. Simulating 3d with 1d

    - `Sales[year][region][product]`
    - vs `_Sales[(year * NUM_REGS * NUM_PRODS) + (region * NUM_-OF_PRODS) + product]`





Mem-Pos	
0	Y0R0P0
1	Y0R0P1
2	Y0R0P2
3	Y0R1P0
4	Y0R1P1
5	Y0R1P2
6	Y0R2P0
7	Y0R2P1
8	Y0R2P2
9	Y0R3P0
10	Y0R3P1
11	Y0R3P2
12	Y1R0P0
13	Y1R0P1
14	Y1R0P2
15	Y1R1P0
16	Y1R1P1
17	Y1R1P2
18	Y1R2P0
19	Y1R2P1
20	Y1R2P2
21	Y1R3P0
22	Y1R3P1
23	Y1R3P2

11. Simulating Multi-dimensional Arrays

    - In general for a d-dimensional array with dimensions S_1, S_2, . . . , S_d, the element at `Item[n_1][n_2]...[n_d]` can be represented as a single dimensional array with the following index

```
_Item[n_d + S_d * (n_{d-1} + S_{d-1}
   * (n_{d-2} + S_{d-2} * (...+S_2*n_1) ... ))]
```

4 Records

4.1 Structs and unions

1. Motivation

   (a) Catalog

   - E.g. Catalog information in a library
   - Data in collection is heterogenous

Title	string
Author	string
Publisher	string
Year	unsigned int
Call Number	string
Price	double

   (b) Soln

   - Solution using arrays:

   ```
   string titles[N];
   string authors[N];
   string publishers[N];
   unsigned int publishingYears[N];
   string callNumbers[N];
   double Price[N];
   ```

   - Poor choice of interface!
   - (many arguments to pass for functions)

2. Use a record instead!

   (a) Col 1

   - Data can be heterogenous
   - Define:

   ```
   struct CatalogEntry {
     string title;
     string author;
     string publisher;
     unsigned int publishingYear;
     string callNumber;
   };
   ```

(b) Col 2

  - Only 1 argument needs to be passed
  - Declare:

```
struct CatalogEntry c;
// or, equivalently this:
CatalogEntry c;
```

  - Initialize:

```
c.title = "Peter Pan";
c.author = "J. M. Barrie";
c.publisher = "Scribner";
c.publishingYear = 1980;
c.callNumber = "B2754 1980";
```

3. Initializing a Record

   - As with arrays

```
CatalogEntry c = {"Peter Pan",
                  "J. M. Barrie",
                  "Scribner",
                  1980,
                  "B2754 1980"};
```

4. Copying a Record

```
// initialization list
CatalogEntry c = { ... };

// initialization by copying
CatalogEntry c1 = c;

// default initialization
CatalogEntry c2;
// assignment operator
c2 = c;
```

5. Functions operating on records

```
void printCatalogEntry(CatalogEntry c){
  cout << "Title: " << c.title << endl;
  cout << "Author: " << c.author << endl;
  cout << "Publisher: " << c.publisher << endl;
  cout << "Publishing Year: " << c.publishingYear << endl;
  cout << "Call Number: " << c.callNumber << endl;
}
```

- As usual, by default arguments are passed by value (call by value)

6. Passing References

   - For efficiency, call by reference is also supported

```
void printCatalogEntry(const CatalogEntry &c){
  cout << "Title: " << c.title << endl;
  cout << "Author: " << c.author << endl;
  cout << "Publisher: " << c.publisher << endl;
  cout << "Publishing Year: " << c.publishingYear << endl;
  cout << "Call Number: " << c.callNumber << endl;
}
```

7. Equality checking

   - Not supported by default

```
if (c1 == c2)   // invalid
```

   - As in the case for arrays, must do this each field at a time

```
bool CatalogEntryEquals(const CatalogEntry &c1, const CatalogEntry &c2) {
  return c1.title == c2.title && c1.author == c2.author &&
         c1.publisher == c2.publisher &&
         c1.publishingYear == c2.publishingYear &&
         c1.callNumber == c2.callNumber;
}
```

8. Complex record data structures

    - Arrays of records

```
CatalogEntry A[3];
CatalogEntry A[] = {{"Peter Pan",
                     "J. M. Barrie",
                     "Scribner",
                     1980,
                     "B2754 1980"},
                    {"C++ Primer",
                     "Stanley B. Lippman",
                     "Addison-Wesley",
                     1998,
                     "QA 76.73 C15 L57 1998"},
                    {"Anatomy of LISP",
                     "John Allen",
                     "McGraw-Hill",
                     1978,
                     "QA 76.73 L23A44"}};
```

9. Practise!

    - See the very first announcement in UR Courses
    - Try the exercises there
        - declare a C++ struct to represent a point in the Cartesian coordinate system
        - declare a C++ struct to represent a hexagon
        - declare a C++ struct to represent a circle

10. Arrays inside of records

    - Can put arrays as fields of records

```
const int MAX_NAMES = 100;

struct FullName {
  string name_component[MAX_NAMES];
  int name_count;
};
```

11. Multi-Dimensional Arrays in Records

```
const int SCREEN_HEIGHT = 768, SCREEN_WIDTH = 1024;
struct Screen{
  char screen_array[SCREEN_HEIGHT][SCREEN_WIDTH];
};

...

Screen my_screen;
for (int i = 0; i < SCREEN_HEIGHT; i++){
  my_screen.screen_array[i][0] = '*';
 }
```

12. Mix and Match

    (a) Col1

    ```
    struct str1 {
      int a[2];
      int b;
    };

    void func1(str1 A[ ]){
      A[0].a[0] = 10;
      A[0].a[1] = 20;
      A[0].b = 30;
    }

    int main( ) {
      str1 A[ ] = {{{1,0},2}, {{3,0},4},{{0,0},9}};
      func1(A);

      std::cout << A[0].b<<"\n";
      std::cout << A[0].a[1]<<"\n";
    }
    ```

    (b) Col2

    - What will the ouput be?

13. Enumerations

(a) Col1

```
enum day {
  Friday = 99,//
  Saturday,//
  Sunday = 90,//
  ...,
  Thursday //
};

day d;
d = Thursday;

if (d == Saturday || d == Sunday)
  cout << "Enjoy the weekend!" ;

cout << d+1 ;
```

(b) Col2

- User-defined data type that consists of integral constants
- What will the output be?

14. Variant records

(a) Col1

- Called `union` in C++
- Multiple component fields can be defined
- At most one field can be in use at one time (fields share the same memory)

(b) Col2

```
union Coordinates {
  int a,
      double b,
      char c
};

Coordinates x;

x.a = 5;
```

```
cout << x.a; // works, prints 5

x.b = 416.905; // destroys the value of x.a
x.c = 'p';     // destroys the value of x.a and x.b
cout << x.a;   // invalid!
cout << x.b;   // invalid!
cout << x.c;   // works, prints p
```

15. Example

    (a) Col1

    ```
    enum CatalogEntryType {
      BookEntry, //
      DVDEntry //
    };

    struct BookSpecificInfo {
      unsigned int pages;
    };
    ```

    (b) Col2

    ```
    struct DVDSpecificInfo {
      unsigned int discs;
      unsigned int minutes;
    };

    union CatalogEntryVariantPart {
      BookSpecificInfo book;
      DVDSpecificInfo dvd;
    };
    ```

16. Example (cont'd)

    ```
    struct CatalogEntry {
      string title;
      string author;
      string publisher;
      unsigned int publishingYear;
      string callNumber;
    ```

```
  CatalogEntryType tag;
  CatalogEntryVariantPart variant;
};
```

17. Example (cont'd)

```
void printCatalogEntry(const CatalogEntry& c) {
  cout << "Title: " << c.title << endl;
  ...
    cout << "Call Number: " << c.callNumber << endl;
  switch (c.tag) {
  case BookEntry:
    cout << "Pages: " << c.variant.book.pages << endl;
    break;
  case DVDEntry:
    cout << "Discs: " << c.variant.dvd.discs << endl;
    cout << "Minutes: " << c.variant.dvd.minutes << endl;
    break;
  }
}
```

18. Prof's Aside

    - C++ unions are unsafe
        - Without the tag, there's no way to know which type a union contains
        - C++ doesn't require the tag to be there
            - You have to make sure it's there
            - You have to make sure the tag actually matches the data
    - Other languages have safe combinations of tags and unions
        - `enum` in Rust and Swift
        - Sealed Classes in Java/Kotlin
        - Algebraic datatypes in functional languages (CS 350)

19. Anonymous declaration of records and variant-records

- Earlier:

```
union CatalogEntryVariantPart {
  BookSpecificInfo book;
  DVDSpecificInfo dvd;
};
```

- Could have actually declared them in-line:

```
union CatalogEntryVariantPart {
  struct BookSpecificInfo { unsigned int pages; } book;
  struct DVDSpecificInfo { unsigned int discs, minutes; } dvd;
};
```

20. Anonymous declaration of records and variant-records

    - Can also anonymize:

```
union CatalogEntryVariantPart {
  struct { unsigned int pages; } book;
  struct { unsigned int discs, minutes; } dvd;
};
```

21. Anonymous declaration of records and variant-records

    - In fact, we could have done the same with the union

```
struct CatalogEntry {
  string title;
  string author;
  string publisher;
  unsigned int publishingYear;
  string callNumber;
  CatalogEntryType tag;
  union {
    struct { unsigned int pages; } book;
    struct { unsigned int discs, minutes; } dvd;
  } variant;
};
```