Lecture 1: Old and New Friends

Review, Deepening & Structure

Lecture Overview

- **Review Programming from TIC1001**

- **Deepening on learned topics**

- **C/C++ - Structure:**
  - Motivation
  - Syntax and Semantic
  - Examples

Learned from history to craft the future

REVIEW TIC1001

C or C++: I’m confused!

- The **<u>codes are in C++</u>**
  - Minor changes will make it C compatible

- We do use **C style output** for simplicity:

```
printf("x:%d y:%d\n", var1, var2);
```

⇕

```
cout << "x:" << var1 << " y:" << var2 << endl;
```

- If you use any C style input / output:
  - `#include <cstdio>` OR
  - `#include <stdio.h>`

Palindrome: **Let's Review**

- A word is a palindrome if it reads the same forward and backward.
  - Examples: **NOON**, **RADAR, LEVEL, ROTATOR**

- Write a program to ask user for a **single word** **W**, then report whether the word **W** is a palindrome

- Focus on how to:
  - Design, Implement, Test

Palindrome: **Scratchpad (blank)**

Program Execution Flow

CONTROL STATEMENTS

Selection Statements **[For Reading]**

```
if (a > b) {
    ...
} else {
    ...
}
```

- `if-else` statement
- Valid conditions:
  - Comparison
  - Integer values (0 = false, others = true)

```
switch (a) {
   case 1:
       ...
       break;
   case 2:
   case 3:
       ...
   default:
}
```

- `switch-case` statement
- Variables in **`switch( )`** must be integer type (or can be converted to integer)
- `break` : stop the fall through execution
- `default` : catch all unmatched cases

Repetition Statements **[For Reading]**

```
while (a > b) {
    ... //body
}
```

```
do {
    ... //body
} while (a > b);
```

- Valid conditions:
  - Comparison
  - Integer values (0 = false, others = true)
- `while` : check condition before executing body
- `do-while`: execute body before condition checking

```
for (A; B; C) {
    ... //body
}
```

- `A` : initialization (e.g. `i = 0`)
- `B` : condition (e.g. `i < 10`)
- `C` : update (e.g. `i++`)
- Any of the above can be empty
- Execution order:
  - `A, B, body, C, B, body, C` …

Storing information

VARIABLE DECLARATION

Simple Data Types

SYNTAX

```
data_type variable_name;
```

`int`	Integer data, e.g. 123, -789, etc
`char`	Character data, e.g. 'a', '#', ' ', etc
`float` `double`	Floating point data, e.g. 3.14, -0.01, etc
`bool`	Boolean data ❑ Can have the value **`true`** or **`false`** only ❑ Improve readability ➔ reduce human error

Array

- A collection of **homogeneous** data
  - Data of the same type

SYNTAX

```
data_type variable_name[size];
```

```
int iA[10];
```

```
iA[0] = 123;
```

Store value into 1st element

```
iA[9] = 456;
```

Store value into last, 10th element

```
iA[1] = iA[0] + iA[9];
```

Store and read values

Example Usage

Writing essay is more than just knowing vocab and grammar

BEYOND SYNTAX & SEMANTIC

Key Skills

- **Program execution**:
  - Understand the "memory snapshot" during execution
- **Program development**:
  - Top-down and modular
  - Incremental: Code ➔ Compile ➔ Test
- **Program maintainability**:
  - Consistent style (indentation, naming)

Let's go further with what we have

DEEPENING

Nesting Control Statements

- The strength (and the difficulty) of programming is that each control statement is simple on their own:
  - But you can mix / combine them in many ways!

- Try this:
  - A C++ function contains a number of **statements**
  - A C++ statement can be an **assignment, if-else, while, for, etc**
  - A **for** statement can contains a number of **statements**
  - **…..?**

Nested-Loop: **Problem 1**

- Asterisks table:
  - Ask the user for **R** and **C**
  - Print a table with **R rows** and **C columns filled with "*"**

Sample Run:

```
3  5      // 3 rows 5 columns
* * * * *
* * * * *
* * * * *
```

Problem: Top-down approach

```
1. Read R and C from user
2. For rCount = 1 to R
```

```
int R, C, rCount, cCount;

cin >> R >> C;

for ( rCount = 1; rCount <= R; rCount = rCount + 1 ){




}
```

Nested-Loop: Problem 2

- Problem:
  - Ask the user for **R** and **C**
  - Print a multiplication table with **R rows** and **C columns**

Sample Run:

```
3  5              // 3 rows 5 columns
1 2 3 4 5
2 4 6 8 10
3 6 9 12 15
```

Problem: **Top-down approach**

1. Read ***R*** and ***C*** from user
2.

```
int R, C, rCount, cCount;

cin >> R >> C;
```

Problem: **Top-down approach**

```
int R, C, rCount, cCount;

cin >> R >> C;

for ( rCount = 1; rCount <= R; rCount = rCount + 1 ){
    printOneRow( rCount, C );
}
```

```
void printOneRow( int multiplier, int N )
//Print one row with N elements of the form:
// 1xmultiplier 2xmultiplier ………… NxMultiplier
{






}
```

Let's bring order to chaos

STRUCTURE

Structure: **Overview**

- We use only **built-in data types** up till now:
  - int, double, char etc
- However, many entities consist of **multiple pieces of information**, e.g.:
  - **Fraction:** a numerator and denominator
  - **Complex number:** a real and imaginary part
  - **Student information**:
    - Name, age, gender, matriculation number, etc
- It is hard to maintain these information as separate variables
  - **C++ provides `structure`** to define a logical container with multiple data inside

Structure: **Defining a new structure**

SYNTAX

Definition:

```
   struct struct_name {
        datatype fieldname1;     //one or more fields
        [ datatype fieldname2; ]
   };
```

EXAMPLE

```
   struct Fraction {
        int num;                //numerator
        int den;                //denominator
   };
```

- Behavior:
  - This declares a **new type of structure**
    - i.e. a new **data type**
  - **No actual variable is allocated!**

Structure: **Defining a new structure**

- Characteristics of structure:
  - All fields should describe a **common entity**
  - Fields can be of different of data type
    - `int, double, char`
    - array!
    - structure!
  - Structure stores **heterogeneous data** (data can potentially be different types)
    - As oppose to array which stores **homogeneous data** (data of the same type)

Structure: **Declaring a structure variable**

- Once a structure data type has been defined, actual structure variable can be now be declared

SYNTAX

Declaration:

```
   struct_name identifier;

OR
   struct_name identifier = init_values;
```

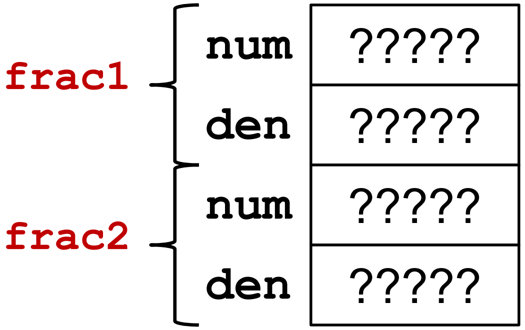
EXAMPLE

$$\underbrace{\texttt{Fraction}}_{\text{data type}}\ \underbrace{\texttt{myFraction;}}_{\text{variable}}$$

Structure Variable: **Memory Snapshot**

```
struct Fraction {
   int num;
   int den;
};
```

```
Fraction frac1, frac2;
```

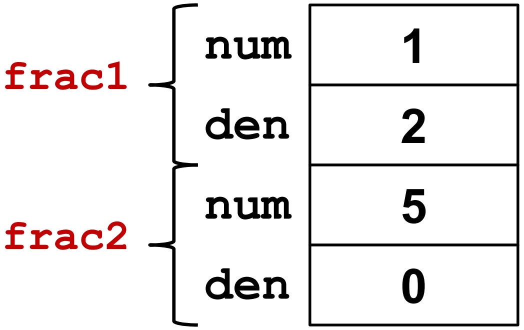


- **Behavior:**
  - ❑ Structure variable contains multiple fields as defined in the structure
    - Each structure variable has an **independent set of the fields**

Structure Variable: **Initialization**

- The **initialization list** for structure variable allows you to give initial values to the fields
  - ❑ Similar to array initialization list, missing values will be taken as zero automatically

```
Fraction frac1 = { 1, 2 };
Fraction frac2 = { 5 };
```



frac1	num	1
	den	2
frac2	num	5
	den	0

- Note:
  - The ordering of the values matches the ordering of the field
  - Use matching type of value for each field

Structure Variable: **Usage**

- To access a field in a structure variable:

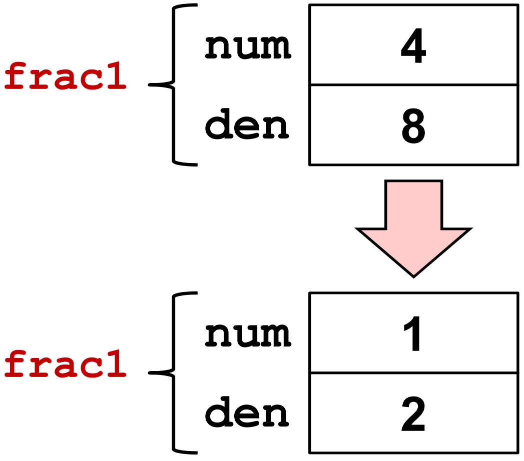
SYNTAX

```
structure_var.fieldname     //note the "."
```

- The "." is known as the **accessor operator**
- A field behaves in the same way as a normal variable with the same datatype

```
Fraction frac1 = { 4, 8 };
int common;

common = GCD( frac1.num, frac1.den );
frac1.num = frac1.num / common;
frac1.den = frac1.den / common;
```



Note: int *GCD*(int, int) returns the **greatest common divisor** of two integers

Structure Variable: **An example**

```
....... // Header not shown

struct Fraction {
   int num;
   int den;
};

int main()
{
   Fraction frac1 = { 0 };
   int common;

   cout << "Numerator and denominator: ";
   cin >> frac1.num >> frac1.den;

   common = GCD( frac1.num, frac1.den );
   frac1.num = frac1.num / common;
   frac1.den = frac1.den / common;

   printf("Simplified: %d / %d\n", frac1.num, frac1.den );

   return 0;
}
```

Structure definition should be placed **at the top of the program**

Initialize both fields to zero

Read value into structure fields.

Manipulate the fields.

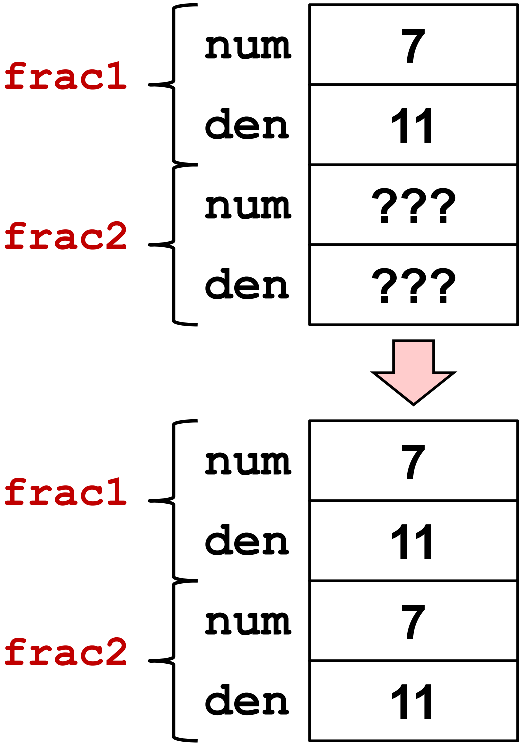
For clarity, the code for ***GCD()*** is omitted.

Structure Variable: **Assignment**

- Assignment between structure variables is allowed:
  - Fields of the source structure will **overwrites all fields of the** target structure

```
Fraction frac1 = { 7, 11 }, frac2;

frac2 = frac1;
```



frac1 num 7
den 11

frac2 num ???
den ???

frac1 num 7
den 11

frac2 num 7
den 11

Structure: **As function parameter**

```
void printFraction( Fraction inFrac );
```

Function Prototype

```
void printFraction( Fraction inFrac )
{
    printf( "%d / %d", inFrac.num, inFrac.den );
}
```

Function Definition

```
int main()
{
   Fraction myFraction = { 123, 456 };

   printFraction( myFraction );

   return 0;
}
```

Function Call

- Structure variable **is passed by value**
  - An **independent copy of the actual argument will be made**

Structure: **Pass by address**

- Structure variable **can be passed by address:**
  - To avoid memory and time wastage in duplicating the structure content
  - To allow a function to modify the actual argument

```
void printFraction( Fraction *fptr )
{
    cout << (*fptr).num << "/" << (*fptr).den;
}
```

Note the "*"

Note how to dereference a structure pointer and access a field

```
int main()
{
   Fraction myFraction = { 123, 456 };

   printFraction( &myFraction );

   return 0;
}
```

Note the "&"

Structure: **Pass by address**

- As dereferencing a structure pointer and accessing a field is very common:
  - C++ provides a **shortcut** notation for this usage
  - The "->" is known as **indirect field selector**

```
void printFraction( Fraction *fptr )
{
    cout << (*fptr).num << "/" << (*fptr).den;
}
```

```
void printFraction( Fraction *fptr )
{
    cout << fptr->num << "/" << fptr->den;
}
```

Equivalent Code by using indirect field selector

Pass by Address: **Example**

```
void readFraction( Fraction *fptr );
```

Function Prototype

```
void readFraction( Fraction *fptr )
{
    int n, d;

    cin >> n >> d;
    fptr->num = n;   // (*fptr).num = n;
    fptr->den = d;   // (*fptr).den = d;
}
```

Function Definition

Challenge
Can you write this function **without** using any local variables?

```
int main()
{
   Fraction myFraction = { 0 };

   readFraction( &myFraction );

   return 0;
}
```

Function Call

Question
Will I get the updated *`myFraction`* in `main()` after function call?

Structure: **Pass by reference**

- In C++, structure variables are commonly **passed by reference:**
  - Essentially providing an **alias** to the original argument
  - Allow a function to modify the actual argument and reduce the "syntax baggage"

```
void printFraction( Fraction& fref )
{
    cout << fref.num << "/" << fref.den;
}
```

Note the "&"

Notice the function can just use "fref" normally without any additional syntax

```
int main()
{
   Fraction myFraction = { 123, 456 };

   printFraction( myFraction );

   return 0;
}
```

Note that there is no additional syntax when a structure is passed by reference

Pass by Reference: **Example**

```
void readFraction( Fraction& fref );
```

Function Prototype

```
void readFraction( Fraction& fref )
{
    int n, d;

    cin >> n >> d;
    fref.num = n;
    fref.den = d;
}
```

Function Definition

Challenge
Same idea: Can be rewritten without local variables

```
int main()
{
   Fraction myFraction = { 0 };

   readFraction( myFraction );

   return 0;
}
```

Function Call

Question
Will I get the updated *`myFraction`* in `main()` after function call?

Structure: **Combining with other data types**

- Structure and array can be "combined" to meet more complicated needs

- For example, it is easy to imagine we may need:
  - Array of structures:
    - Array of fractions, array of students etc
  - Structure with array as field:
    - Student's name is a string (char array)
  - Structure with structure as field:
    - A **line** in a 2D plane can be defined with **two points** (X_1, Y_1) and (X_2, Y_2)

- It is important to understand the basic behavior then apply it to different combinations

Structure: **Array of structures**

- Given the fraction structure:

```
struct Fraction {
    int num;
    int den;
};
```

- Declare an array of **5 fractions**

- Read the **5 fractions** from user

- Swap the 0th and 3rd fractions in the array
  - **Hard mode:** Sort the fractions in ascending order ☺

Structure: **With array as a field**

- Given:

```
struct Triangle {
    int X[3];
    int Y[3];
};
```

A triangle has 3 points; each with (X, Y) coordinate

1. Declare and initialize a `triangle` structure with the following information:
   - The three points are (1, 2), (5, 5), (-1, -3)
2. How do we check whether any point lies on the X-axis or Y-axis?
3. How do we translate (move) the triangle horizontally by 3 units?

Structure: **With structure as a field**

- Given the following structure

```
struct Point {
   int X, Y;
};
```

Represent a point in 2D plane

- Define a new structure **Line**:
  - Consists of two point structure as fields

- Write functions to:
  - Read the (x,y) for a **single point**
  - Read the two points for a **Line**

- **Challenge:**
  - Read two lines and check whether **they intersect?**

Summary

Review	Control Statements: - Selection (if-else, switch-case) - Repetition (while, for) Variable Declarations: - Simple data type - Array
Deepening	nested control flow statements
C++ Elements	Data Type: - Structure - Defining structure - Declaring structure variable - Structure variable usage - Structure as function parameter