# Computer Programming

## Class

Hung-Yun Hsieh
October 25, 2022

# Computer Programming

## Starting from Structure

# Grouping Data

- Group of data
  - Array groups data of "same" type
    - For example, a group of 10 integers, 20 doubles, …
  - What if we want a group of data of "different" types?
    - For example, a group of integers and doubles

```
A personal record consists of the following fields:

1. Sex (M/F)
2. Age
3. Height
4. Weight
```

☞ How to efficiently handle (access) 50 personal records?

- Using only 1 variable instead of 4 or 200 variables

# Using struct

- Using struct for creating a new data type
  - Used to create a new data type that is a grouping of other data types (same or different)
  - ☞ A compound (derived) data type

```
struct name_of_the_structure
{
    type1 name_of_member1;
    type2 name_of_member2;
    …
};
```

Note the semicolon ; here

A new data type with
1 character
1 integer
2 doubles  to hold the personal record

# `struct` Member Access

```
struct record {
    char sex;
    int  age;
    double height, weight;
} x;
```

- Creating a new data type called "record"

```
struct record
{
      char sex;
      int  age;
      double height, weight;
};
```

Name of the new data type

4 **members** of the new data type

  - Use the dot operator (.) to access individual members given the structure variable

It is wrong to use: **record**.age

```
record x;
x.sex = 'M';
x.age = 20;
x.height = 175.0;
x.weight = 60.0;
```

```
record x = {'M', 20, 175.0, 60.0};
```

# struct Member Access (cont.)

- A different structure called "account"

```
struct account
{
     char id;
     char sex;
     int  age;
};
```

Okay to have the same names for data members as in "record"

- Need a handle to the structure (m or x) to access data members therein

```
account m;
m.id  = 1;
m.sex = x.sex;
m.age = x.age;
```

```
account n;
  n = m;
```

**Member-wise** assignment

# `struct` Example

Note that
`cout << John;`
makes sense only if the programmer specifies how a "record" should be shown (we will discuss this later)

```
#include <iostream>
using namespace std;
struct record {
    char sex;
    int  age;
    double height, weight;
};

int main( )
{
    record John;
    John.sex = 'M';
    John.age = 25;
    John.height = 179.5;
    John.weight = 70;

    record Mary = {'F', 23, 160, 50};

    cout << "John's age is " << John.age << " years old" << endl;
}
```

**Member selection operator:**

1. Dot (.) is preceded by a **struct** variable or reference to a **struct** variable
2. Arrow (->) is preceded by a pointer to a **struct** variable

```
record &rRec = John;
rRec.age = 25;
```

```
record *pRec = &Mary;
pRec->sex = 'F';
pRec->age = 23;
```

compile error:
`cout << John;`

# Data Structure Alignment

Member selection operator (. and ->) has higher precedence than &

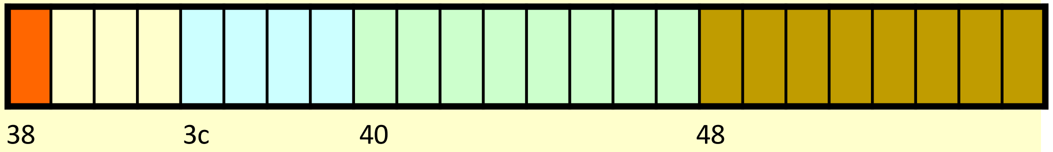
```
#include <iostream>
using namespace std;

struct record {
    char sex;
    int  age;
    double height, weight;
};

int main( )
{
    record John;

    cout << "size of record: " << sizeof(John) << endl;
    cout << "address of sex:   \t " << (void*)&John.sex << endl;
    cout << "address of age:   \t " << &John.age << endl;
    cout << "address of height:\t " << &John.height << endl;
    cout << "address of weight:\t " << &John.weight << endl;
}
```

```
size of record: 24
address of sex:          0x23ff38
address of age:          0x23ff3c
address of height:       0x23ff40
address of weight:       0x23ff48
```



# Data Structure Alignment

C and C++ do not allow the compiler to reorder structure members to save space

- Memory alignment
  - When a modern computer reads from or writes to a memory address, it will do this in word sized chunks
    - A 32-bit system uses a 4 byte chunks
  - ☞ Data alignment and data structure padding
- Typical alignment
  - A `char` is 1-byte aligned
  - A `short` is 2-byte aligned
  - An `int` / `float` is 4-byte aligned
  - A `double` is 8-byte aligned (Windows)
  - A `pointer` is 8-byte aligned (64-bit system)

```
struct account
{
    char id;
    int  age;
    char sex;
};
```

Try swapping age and sex

```
sizeof(account)=12
```

# Rational (1/2)

```
It would be good to use:
c = a + b;
d = a * b;
e = a / b;  (more on this later)
```

```
#include <iostream>
using namespace std;
struct rational {
    int n;
    int d;
};

int main( )
{
    rational a={4, 5}, b={2, 3};
    rational c, d, e;

    c = rplus(a, b);
    d = rmultiply(a, b);
    e = rdivide(a, b);

    cout<<a.n<<"/"<<a.d<<"+"<<b.n<<"/"<<b.d<<"="<<c.n<<"/"<<c.d<<endl;
    cout<<a.n<<"/"<<a.d<<"*"<<b.n<<"/"<<b.d<<"="<<d.n<<"/"<<d.d<<endl;
    cout<<a.n<<"/"<<a.d<<"/"<<b.n<<"/"<<b.d<<"="<<e.n<<"/"<<e.d<<endl;
}
```

```
rational rplus(rational, rational);
rational rmultiply(rational, rational);
rational rdivide(rational, rational);
```

$a = \frac{4}{5}, \quad b = \frac{2}{3}$

Define these functions for rational-specific arithmetic operations

```
4/5+2/3=22/15
```

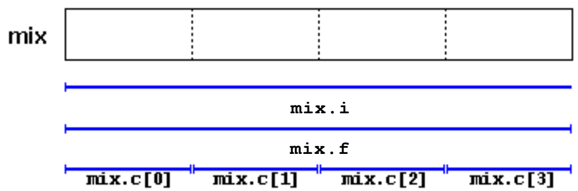
# Rational (2/2)

```
rational rplus(rational x, rational y) {
    rational z;
    z.n = x.n*y.d + x.d*y.n;
    z.d = x.d*y.d;
    return z;
}

rational rmultiply(rational x, rational y) {
    rational z;
    z.n = x.n*y.n;
    z.d = x.d*y.d;
    return z;
}

rational rdivide(rational x, rational y) {
    rational z;
    z.n = x.n*y.d;
    z.d = x.d*y.n;
    return z;
}
```

Can use a user-defined function for calculating GCD for further processing



# enum and union

- Enumerated data types `enum`
  - Enumerated types are types that are defined with a set of *custom identifiers (enumerators)* as possible values

```
enum color {red, green, blue};

color a=red;
if (a==red) a=blue;
```

Each `enum` element is assigned an unsigned integer value internally. If it is not specified otherwise, the first element starts from 0

```
enum color {red=1, green, blue=4};
```

- Union `union`
  - Unions allow one portion of memory to be *accessed as different data types*

```
union mix {int i; float f; unsigned char c[4];} x;

x.f = -1313.3125;
for (int i=3;i>=0;i--) cout << hex << (int)x.c[i];
```

**$-1313.3125 = \text{C4A42A00}_{16}$**

# Defining New Data Type

- Use `struct` to define a new data type
  - Variables can be created (instantiated) based on the new data type (structure)
  - Members can be accessed through the variable name (or pointer/reference to the variable)
  - It is okay to use the same member name across different structures (local scope of the variable name)
  - There might be some operations associated with the new data type that need to be defined in order for the newly created data type to be useful
    - Such operations (functions or operators) would be used only for the new data type (i.e., "data type"-dependent)

    ☞ Defining functions and operators as *part of* the new data type

# Rational – What We Will Learn Later

```
int main( )
{
    rational a, b, c, d, e;

    cout << "Please enter a = "; cin >> a;
    cout << "Please enter b = "; cin >> b;

    c = a + b;
    d = a * b;
    e = a / b;

    cout << a << " + " << b << " = " << c <<endl;
    cout << a << " * " << b << " = " << d <<endl;
    cout << a << " / " << b << " = " << e <<endl;
}
```

This code by itself is **not complete yet** (the rational class needs to be **defined**)

*cf.* syntax on p. 10

```
Please enter a = 2/3
Please enter b = 4/5
2/3 + 4/5 = 22/15
…
```