Cairo University, Faculty of Computers and AI

CS213 – 2022 / 2023

Object Oriented Programming

Lecture 2: C++ Structures

By

Dr. Mohammad El-Ramly

http://www.acadox.com/class/64401 PY7OGJ

CS112 – 2021 / 2022 2nd Term

Structured Programming

Lecture 12: C++ Structures

By

Dr. Mohammad El-Ramly

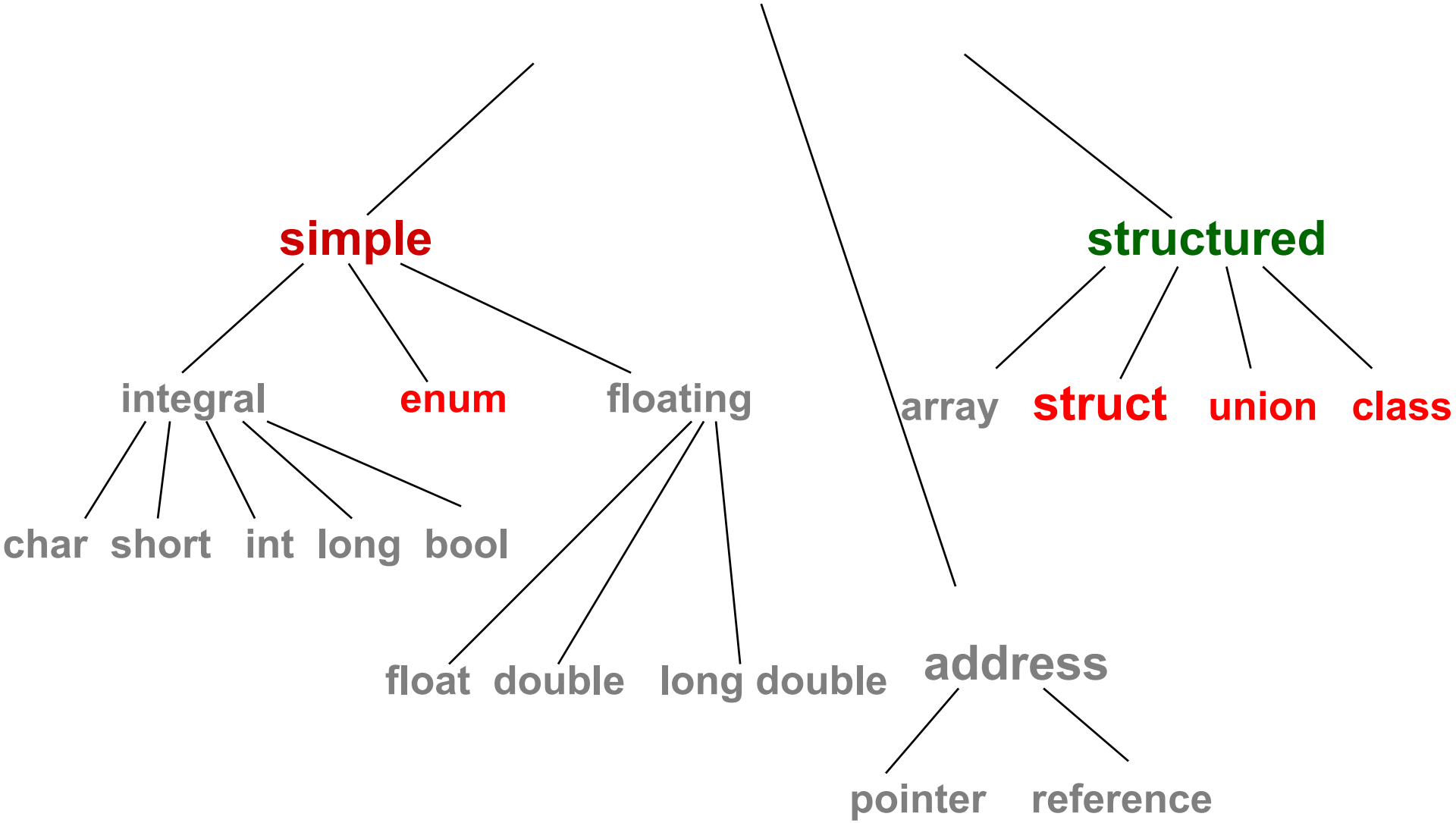
Some slides by Pearson Education

Lecture Objective / Content

1. Structures
2. Arrays of Structures
3. Passing Structures to Functions
4. Nested Structures
5. Pointers to Structures
6. Classes
7. Enumerations
8. Unions

Pb#25, 26, 27

C++ Data Types


simple
structured
integral
enum
floating
array
struct
union
class
char
short
int
long
bool
float
double
long double
address
pointer
reference

More Data Types

- Enumerations are types with restricted set of possible values.
- Pointers are the internal addresses of a value in the memory
- Arrays are ordered collections of data of the same type
- **Records / Structures are collections of data, each consists of items of different types that represent a coherent whole**

1. Structures

- A structure is an aggregation of data items of different types
- It represents a record of related information
- These information have one coherent meaning representing an entity.
- For example, student record, bank account, etc
- By defining a structure we have a new data type to use.

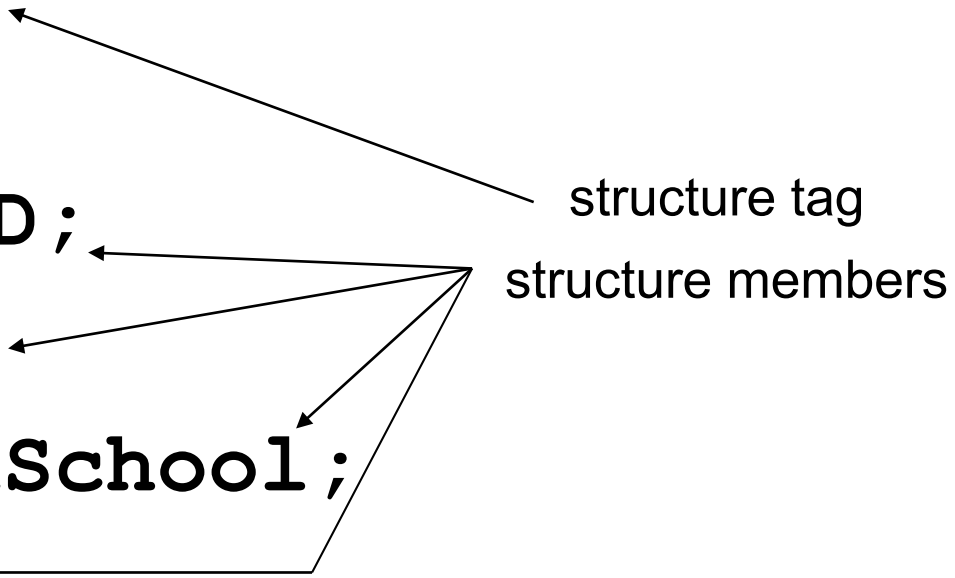
Combining Data into Structures

- Structure: C++ construct that allows multiple related variables to be grouped together
- General Format:

```
struct <structName>
{
  type1 field1;
  type2 field2;
  . . .
};
```

Example `struct` Declaration

```
struct Student
{
   int studentID;
   string name;
    short yearInSchool;
   double gpa;
};
```



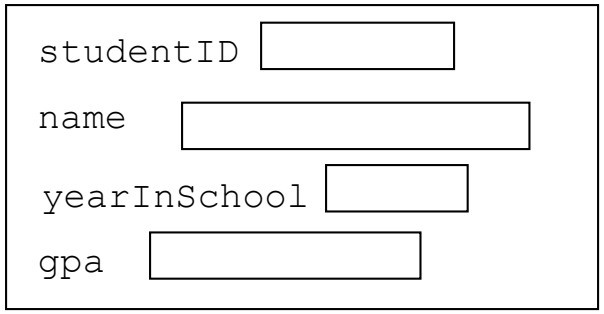
`struct` Declaration Notes

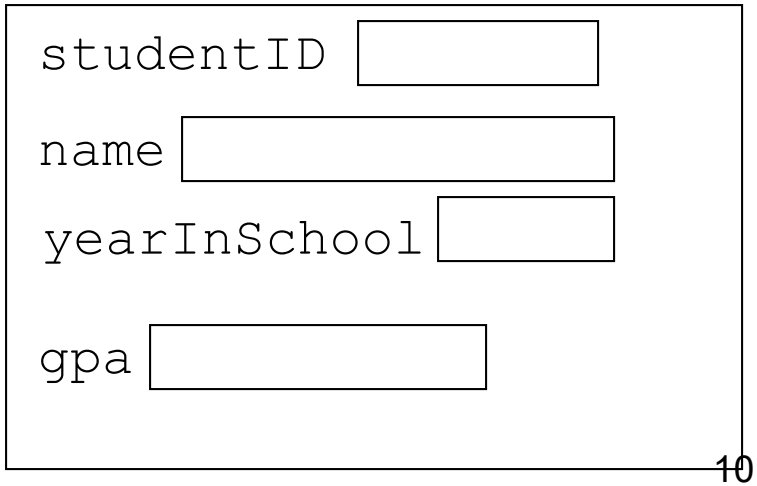
- Must have `;` after closing `}`
- `struct` names commonly begin with uppercase letter
- Multiple fields of same type can be in comma-separated list:

```
string name, address;
```

`struct` Declaration Notes

- **`struct`** declaration does not allocate memory or create variables
- To define variables, use structure tag as type name:
- **`Student stud1;`**
- **`Student stud2;`**





Accessing Structure Members

- Use the dot (`.`) operator to refer to members of **`struct`** variables:

```
cin >> stud1.studentID;

getline(cin, stud1.name);
stud1.gpa = 3.75;
```

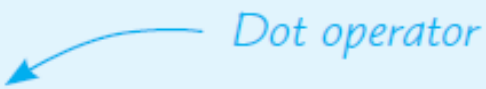
- Member variables can be used in any manner appropriate for their data type

Accessing Structure Members

The Dot Operator

The **dot operator** is used to specify a member variable of a structure variable.

SYNTAX



StructureVariableName.MemberVariableName

```
struct StudentRecord
{
    int studentNumber;
    char grade;
};

int main()
{
    StudentRecord yourRecord;
    yourRecord.studentNumber = 2001;
    yourRecord.grade = 'A';
```

Some writers call the dot operator the *structure member access operator* although we will not use that term.

Pb#25 Employee's Pay `struct`

- Develop a structure to store employee's data in a payroll system like ID, name, hours worked, hourly rate, gross pay.
- Then write a program to take employee's details and then calculate gross pay and then print it.

```
struct Employee {
    int emp_ID;
    string name;
    double hours_worked;
    double hourly_rate;
    double gross_pay;
};
```

Program 11-1

```
1  // This program demonstrates the use of structures.
2  #include <iostream>
3  #include <string>
4  #include <iomanip>
5  using namespace std;
6
7  struct PayRoll
8  {
9     int empNumber;     // Employee number
10    string name;       // Employee's name
11    double hours;      // Hours worked
12    double payRate;    // Hourly payRate
13    double grossPay;   // Gross pay
14 };
15
16 int main()
17 {
18    PayRoll employee; // employee is a PayRoll structure.
19
20    // Get the employee's number.
21    cout << "Enter the employee's number: ";
22    cin >> employee.empNumber;
23
24    // Get the employee's name.
25    cout << "Enter the employee's name: ";
```

```
26      cin.ignore();  // To skip the remaining '\n' character
27      getline(cin, employee.name);
28
29      // Get the hours worked by the employee.
30      cout << "How many hours did the employee work? ";
31      cin >> employee.hours;
32
33      // Get the employee's hourly pay rate.
34      cout << "What is the employee's hourly payRate? ";
35      cin >> employee.payRate;
36
37      // Calculate the employee's gross pay.
38      employee.grossPay = employee.hours * employee.payRate;
39
40      // Display the employee data.
41      cout << "Here is the employee's payroll data:\n";
42      cout << "Name: " << employee.name << endl;
43      cout << "Number: " << employee.empNumber << endl;
44      cout << "Hours worked: " << employee.hours << endl;
45      cout << "Hourly payRate: " << employee.payRate << endl;
46      cout << fixed << showpoint << setprecision(2);
47      cout << "Gross Pay: $" << employee.grossPay << endl;
48      return 0;
49  }
```

Program Output with Example Input Shown in Bold

```
Enter the employee's number: 489 [Enter]
Enter the employee's name: Jill Smith [Enter]
How many hours did the employee work? 40 [Enter]
What is the employee's hourly pay rate? 20 [Enter]
Here is the employee's payroll data:
Name: Jill Smith
Number: 489
Hours worked: 40
Hourly pay rate: 20
Gross pay: $800.00
```

Displaying a `struct` Variable

- To display the contents of a **`struct`** variable, must display each field separately, using the dot operator:

```
cout << stud1; // won't work
cout << stud1.studentID << endl;
cout << stud1.name << endl;
cout << stud1.yearInSchool;
cout << " " << stud1.gpa;
```

- Unless your **overload << operator**

Comparing `struct` Variables

- Cannot compare **`struct`** variables directly:

```
if (stud1 == stud2) // won't work
```

- Instead, must compare on a field basis:

```
if (stud1.studentID ==
                stud2.studentID) ...
```

- Can also overload < and == operators

Initializing a Structure

- **`struct`** variable can be initialized when defined:

```
Student s = {11465, "Joan", 2, 3.75};
```

- Can also be initialized member-by-member after definition:

```
s.name = "Joan";
s.gpa = 3.75;
```

More on Initializing a Structure

- May initialize only some members:

```
Student s = {14579};
```

- Cannot skip over members:

```
Student s = {1234, "John", , 2.83};
 // illegal
```

- Cannot initialize in the structure declaration, since this does not allocate memory

Example `struct` Initialization

```
 8  struct EmployeePay
 9  {
10     string name;          // Employee name
11     int empNum;           // Employee number
12     double payRate;       // Hourly pay rate
13     double hours;         // Hours worked
14     double grossPay;      // Gross pay
15  };

19     EmployeePay employee1 = {"Betty Ross", 141, 18.75};
20     EmployeePay employee2 = {"Jill Sandburg", 142, 17.50};
```

Using a Constructor to Initialize Structure Members

- A constructor is a special function that can be a member of a structure
  - It has the same name as the function
  - It has no return value
  - It is automatically invoked (not explicitly called)
  - It may have default values
- It is written inside the **`struct`** declaration to initialize the structure's data members

A `Struct` with a Constructor

```
struct Dimensions
{
   int length, width, height;

   // Constructor
   Dimensions(int l, int w, int h) {
      length = l; width = w; height = h;
   }
};
```

Default Arguments

```
struct Dimensions
{
   int length, width, height;
   // Constructor
   Dimensions(int l = 1, int w = 1, int h = 1){
      length = l; width = w; height = h;
   }
};
```

Passing Arguments to a Constructor

```
//Create a box with all dimensions given
Dimensions box4(12, 6, 3);

//Create a box using default value 1 for
//height
Dimensions box5(12, 6);

//Create a box using all default values
Dimensions box6;

//Use COPY constructor
Dimensions box7(box4);

//Using ASSIGNMENT initialization
Dimensions box8 = box3;
```

Omit () when no arguments are used

2. Arrays of Structures

- Structures can be defined in arrays
- ```
  const int NUM_STUDENTS = 20;
  Student stuList[NUM_STUDENTS];
  ```
- Individual structures are accessible using subscript notation
- Fields within structures accessible using dot notation:

```
cout << stuList[5].studentID;
```
```

Initialize Arrays of Structures

- ```
  Student stuList[3] =
      {{11465, "Joan", 2, 3.75},
       {10034, "Bilal", 1, 2.75},
       {10174, "Samia", 2, 2.09}};
  ```
  ```

Pb#26 Employees' Pay `struct`

- Repeat Problem #25 but create an array of employees and loop to input their data and then print it with the gross pay for each of them.

Program 11-4

```
1  // This program uses an array of structures.
2  #include <iostream>
3  #include <iomanip>
4  using namespace std;
5
6  struct PayInfo
7  {
8     int hours;          // Hours worked
9     double payRate;     // Hourly pay rate
10 };
11
12 int main()
13 {
14    const int NUM_WORKERS = 3;     // Number of workers
15    PayInfo workers[NUM_WORKERS]; // Array of structures
16    int index;                    // Loop counter
17
```

```
18      // Get employee pay data.
19      cout << "Enter the hours worked by " << NUM_WORKERS
20           << " employees and their hourly rates.\n";
21
22      for (index = 0; index < NUM_WORKERS; index++)
23      {
24         // Get the hours worked by an employee.
25         cout << "Hours worked by employee #" << (index + 1);
26         cout << ": ";
27         cin >> workers[index].hours;
28
29         // Get the employee's hourly pay rate.
30         cout << "Hourly pay rate for employee #";
31         cout << (index + 1) << ": ";
32         cin >> workers[index].payRate;
33         cout << endl;
34      }
35
36      // Display each employee's gross pay.
37      cout << "Here is the gross pay for each employee:\n";
38      cout << fixed << showpoint << setprecision(2);
39      for (index = 0; index < NUM_WORKERS; index++)
40      {
41         double gross;
42         gross = workers[index].hours * workers[index].payRate;
43         cout << "Employee #" << (index + 1);
44         cout << ": $" << gross << endl;
45      }
46      return 0;
47  }
```

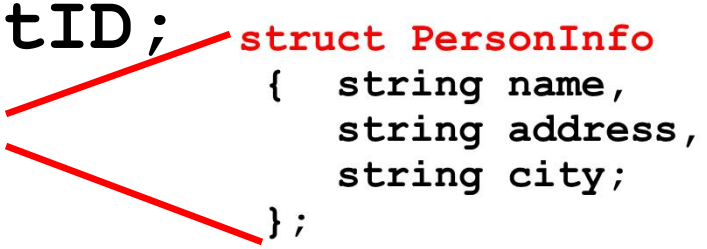
Program Output with Example Input Shown in Bold

```
Enter the hours worked by 3 employees and their hourly rates.
Hours worked by employee #1: 10 [Enter]
Hourly pay rate for employee #1: 9.75 [Enter]

Hours worked by employee #2: 20 [Enter]
Hourly pay rate for employee #2: 10.00 [Enter]

Hours worked by employee #3: 40 [Enter]
Hourly pay rate for employee #3: 20.00 [Enter]

Here is the gross pay for each employee:
Employee #1: $97.50
Employee #2: $200.00
Employee #3: $800.00
```

3. Nested Structures

- A structure can have another structure as a
- member.

```
struct PersonInfo
{   string name,
    string address,
    string city;
};

struct Student
{   int          studentID;
    PersonInfo   pData;
    short        year;
    double       gpa;
};
```



Members of Nested Structures

- Use the dot operator multiple times to access fields of nested structures
- `Student stud;`
- `stud.pData.name = "Nadine";`
- `stud.pData.city = "Samya";`

4. Passing Structures to Functions

- We can pass members of **`struct`** variables to functions

```
computeGPA(stud1.gpa);
```

- Or pass an entire **`struct`** to a functions

```
showData(stud2); // Copied by value
```

- You can use reference parameter & if function needs to modify contents of structure variable

`const` Reference Parameter

- Using a value parameter for structure can slow down a program and waste space
- Using a reference parameter speeds up program, but allows the function to modify data in the structure
- To save space and time, while protecting structure data that should not be changed, use a `const` reference parameter
- `void showData (const Student &s)`

Returning a `struct` from a Function

- Function can return a **`struct`**

```
Student getStuData();   // prototype
s1 = getStuData();      // call
```

- Function must define a local struct variable
  - for internal use
  - to use with **`return`** statement
- Return happens by value. **`struct`** members are copied to the calling function.

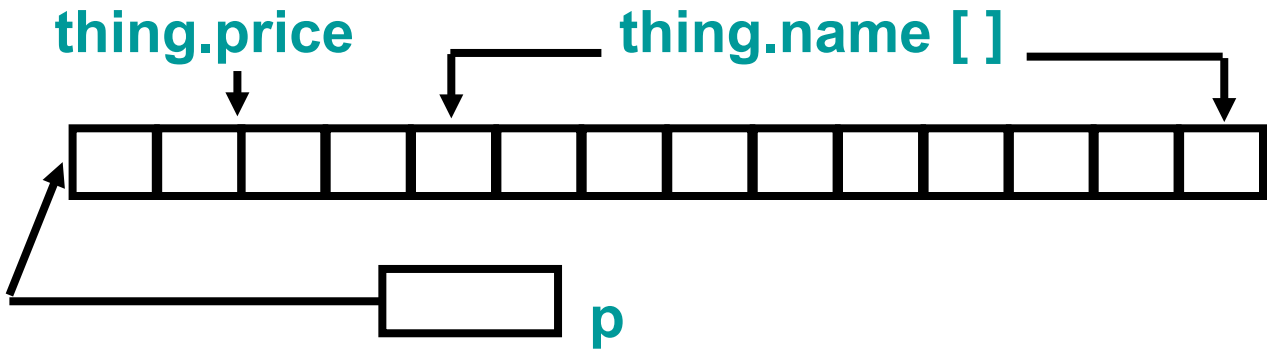
Returning a Structure Example

```
Student getStuData() {
  Student s;     // local variable
  cin >> s.studentID;
  cin.ignore();
  getline(cin, s.pData.name);
  getline(cin, s.pData.address);
  getline(cin, s.pData.city);
  cin >> s.year;
  cin >> s.gpa;
  return s;   // Copy values in s to calling funct
}
```

5. Pointers to Structures

```
struct part {
  float price ;
  char name [10] ;
} ;

struct part *p , thing;
p = &thing;

/* These are equivalent */
thing.price = 50; // dot operator
(*p).price = 50;  // dereference & dot
p -> price = 50;  // arrow operator
```

5. Pointers to Structures



- p is set to point to the first byte of the **`struct`** variable

5. Pointers to Structures

```
struct part * p, *q;
p = new part;
q = new part;
p -> price = 199.99 ;
strncpy( p -> name, "hard disk",9);
(*q) = (*p);
q = p;                  // Memory leak problem
delete(p);
delete(q);              // Dangling pointer pb
```

same address

6. Classes

```
class
struct Dimensions
{
  Public:
  int length, width, height;
  // Constructor
  Dimensions(int l = 1, int w = 1, int h = 1){
     length = l; width = w; height = h;
  }
};
```

6. Classes

- A **`class`** is a generalization of **`struct`**
- **`struct`** members are **`public`** by default
- **`class`** members are **`private`** by default
- Private **`class`** members are NOT accessed directly outside the class. They are accessed through setter and getter functions.
- **`class`** can have data members and ***function*** members called methods.
- An instance of a **`class`** is called object
- You can build a library of your own classes (types)

DayOfYear struct

```
struct DayOfYear {
    int month;
    int day;
};

void output(DayOfYear a_day) {
   cout << "month = " << a_day.month
   << ", day = " <<  a_day.day << endl;
}
```

DayOfYear class

```
class DayOfYear {
   public:
       int month;
       int day;
};

void output(DayOfYear a_day) {
   cout << "month = " << a_day.month
   << ", day = " <<  a_day.day << endl;
}
```

Using DayOfYear

```
int main( ) {
    DayOfYear today;

    cout << "Enter today's date:\n";
    cout << "Enter month as a number: ";
    cin  >> today.month;
    cout << "Enter day of the month: ";
    cin  >> today.day;

    cout << "Today's date is ";
    output(today);
}
```

OOP DayOfYear
Functions added inside class

```
class DayOfYear {
   private:
      int month;
      int day;
   public:
      void output( );  // only prototype
      void setDay (int d){day = d;}
      void setMon (int m){month = m;}
};

void DayOfYear::output( ) { // fun body
   cout << "month = " << month
   << ", day = " << day << endl;
}
```

Using OOP DayOfYear

```
int main( ) {
    DayOfYear today;
    int month, day'

    cout << "Enter today's date:\n";
    cout << "Enter month as a number: ";
    cin  >> month;
    cout << "Enter day of the month: ";
    cin  >> day;
    today.setDay(day); today.setMon(month);
    cout << "Today's date is ";
    today.output( );
}
```

Pb#27 Rational Number Calculator

- Develop a class called rational number and the necessary functions to manipulate it.
- Write a program to demo the use of this class.

Pb#27 Rational Number Calculator

```
class Rational  {    // Goes to header file
 public:
    Rational(int = 0, uint = 1);

    Rational add(const Rational &);
    Rational subtract(const Rational &);
    Rational multiply(const Rational &);
    Rational divide(const Rational &);
    void printRational();
 private:
    int numerator;
    int denominator;
    void reduce();  // fun to reduce rational
};
```

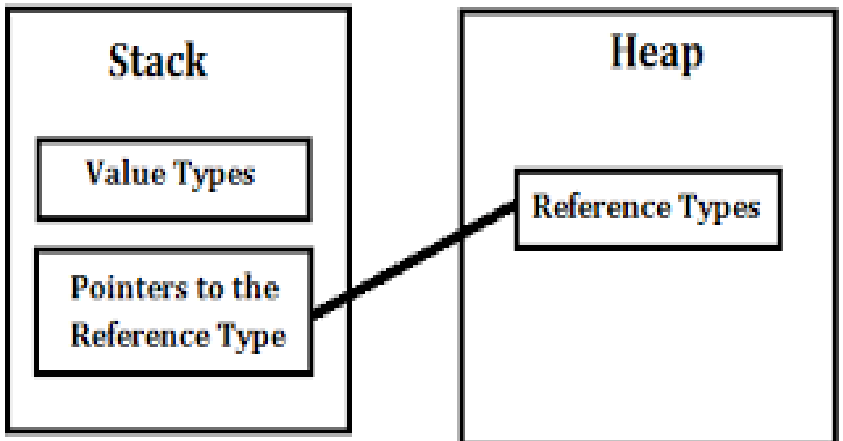
Pb#27 Rational Number Calculator

```
// Example use of Rational class
int main {
    Rational x(-2,6), y(-14,16), z,
    x.printRational ();
    cout << " + ";
    y.printRational ();
    cout << " = " << x.add (y);
};
```

http://courses.washington.edu/css342/zander/css332/

C++ Structures vs Classes

Class	Structure
Members of a class are private by default.	Members of a structure are public by default.
Memory allocation happens on the heap.	Memory allocation happens on a stack.
It is a reference type data type.	It is a value type data type.
It is declared using the **class** keyword.	It is declared using the **struct** keyword.



C++ Structures vs C Structures

C struct is a data holder, C++ struct is more like class

C Structures	C++ Structures
Only data members are allowed, it cannot have member functions.	Can hold both: member functions and data members.
Cannot have static members.	Can have static members.
Cannot have a constructor inside a structure.	Constructor creation is allowed.
Direct Initialization of data members is not possible.	Direct Initialization of data members is possible.
Writing the 'struct' keyword is necessary to declare structure-type variables.	Writing the 'struct' keyword is not necessary to declare structure-type variables.
Do not have access modifiers.	Supports access modifiers.
Only pointers to structs are allowed.	Can have both pointers and references to the struct.

When to use `struct` C++ or `class`

- **Use a structure** if you want to store a record of data and develop some functions to handle it.
- Do not use structure as a class.

- **Use a class** if you want to
  - Develop an app with OOP paradigm
  - If you want to create a new type (Rational, Complex, …) and empower it with some member functions to support using it.

7. Enumerations

- Enumeration, or *enum* for short, is a type whose values are user-defined named constants called *enumerators*.
- There are two kinds of enums: the *unscoped enums* (old fashion) and *scoped enums* (Modern C++).
- Use it when you want to restrict your variable to discrete predefined values.

Unscoped Enumerations

```
enum WeekDay{
     Saturday, Sunday, Monday, Tuesday,
     Wednesday, Thursday, Friday
};

int main()  {
    WeekDay current_day = Friday;
    current_day = Saturday;
    cout << current_day;  // Print 0
}
```

Unscoped Enumerations

```
enum WeekDay{
    Saturday = 1, Sunday, Monday,
    Tuesday, Wednesday, Thursday, Friday
};

int main()  {
   WeekDay current_day = Friday;
   current_day = Tuesday;
   cout << current_day; // Print 4
}
```

Scoped Enumerations

- Unscoped enumerations has two issues
  - They *leak* into an outside scope
  - are implicitly convertible to other int.
  - Cannot use the same name twice.

C++ Scoped Enumerations

```
enum class WeekDay: char{
    Saturday, Sunday, Monday, Tuesday,
    Wednesday, Thursday, Friday
};

int main()  {
    WeekDay day = WeekDay::Friday;
    day = WeekDay::Saturday;
    cout << day;                    // Wrong
}
```

8. Unions

- A **`union`** is a less used facility that defines a variable that can be of one more types.
- Similar to a **`struct`**, but
  - Members share a single memory location, which saves space
  - Only 1 member of the union is used at a time
- Declared using key word **`union`**
- Variables defined and accessed like **`struct`** variables.

Example `union` Declaration

```
union WageInfo
{
   double hourlyRate;
   int    annualSalary;
};
```

union tag

union members

Notice the required ;

```
WageInfo wage;
wage.annualSalary = 111100.0;
cout << wage.hourlyRate << "\n";
cout << wage.annualSalary << "\n";
```

Anonymous Union

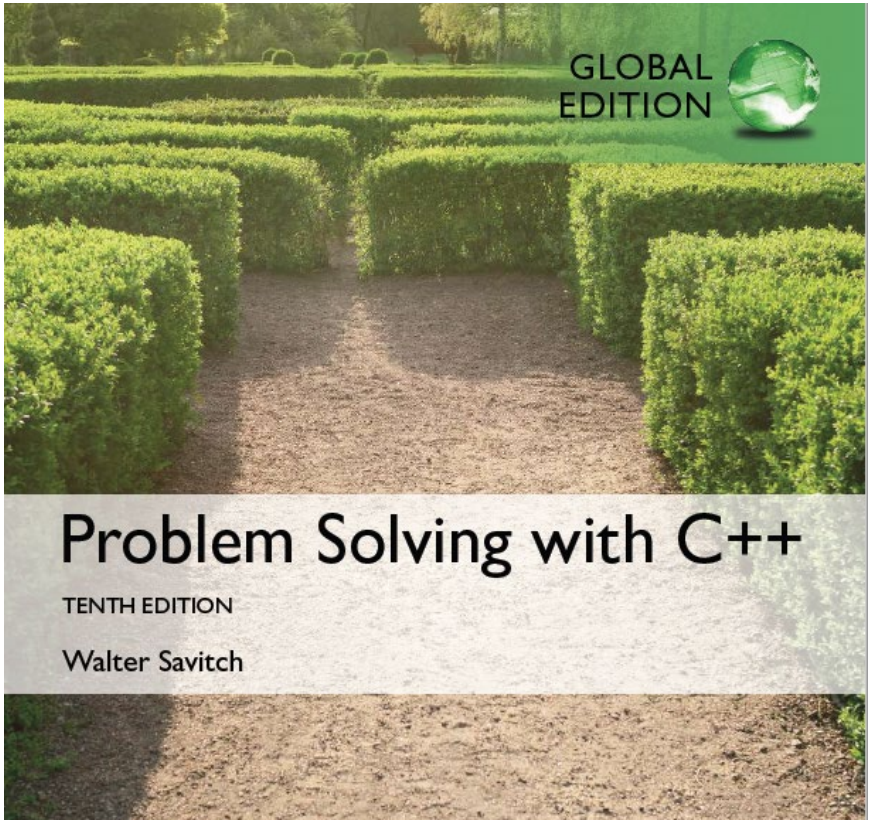
- A **`union`** without a tag:

```
union { ... };
```

- With no tag you cannot create additional union variables of this type later
- Allocates memory at declaration time
- Refer to members directly without dot operator

Readings

- Savitch Chap 10
- Dr Amin notes



- Excellent – Must read
- https://dare2compete.com/blog/difference-between-structure-and-class-in-cpp