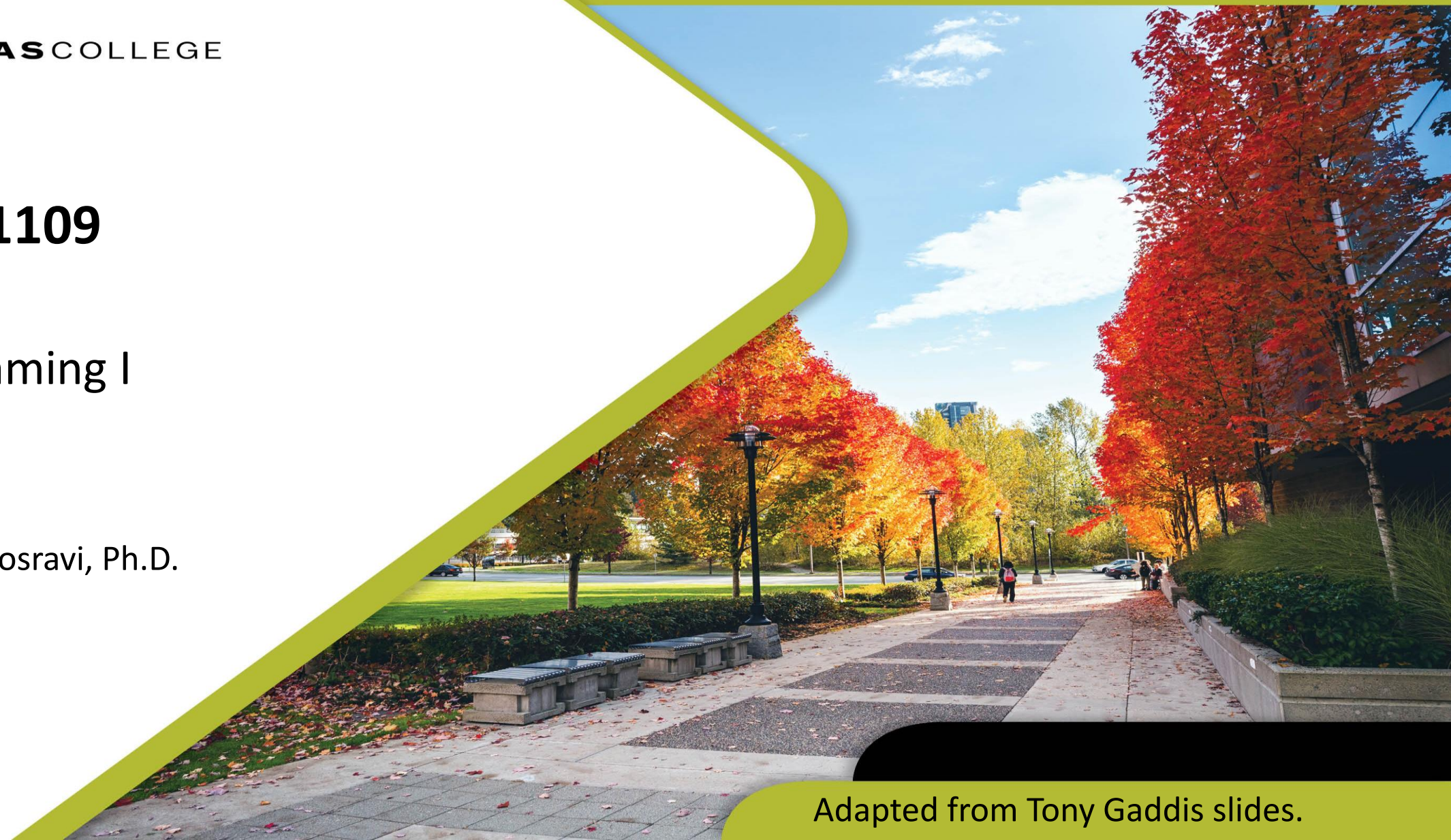
DOUGLAS COLLEGE

CMPT 1109

Programming I

Shahriar Khosravi, Ph.D.

Lecture 10



Adapted from Tony Gaddis slides.

Plan for Today

- Abstract Data Types
- Structures
- Accessing Structure Members
- Initializing a Structure
- Arrays of Structures
- Nested Structures
- Structures as Function Arguments
- Returning a Structure from a Function
- Pointers to Structures
- Enumerated Data Types

DOUGLAS COLLEGE

Abstract Data Types

Abstract Data Types

- **Abstract data types (ADTs)** are data types created by the programmer.
- ADTs have their own range (or domain) of data and their own sets of operations that may be performed on them.
- Users of an abstract data type does not need to know the implementation of the data type, e.g., how the data is stored
- **Abstraction**: a definition that captures general characteristics without details.
  - Example: An abstract triangle is a 3-sided polygon.
  - A specific triangle may be scalene, isosceles, or equilateral.
- **Data Type** defines the values that can be stored in a variable and the operations that can be performed on it.

DOUGLAS COLLEGE

Combining Data into Structures

Combining Data into Structures

- We have so far written programs that keep data in individual variables.
- If we need to group items together, C++ allows us to create arrays.
- However, **all elements in an array must be of the same data type**.
- Sometimes a relationship exists between items of different types.
- For example, a payroll system might keep the variables shown here.
- **Structure**: A C++ construct that allows multiple variables to be grouped together. C++ gives us the ability to package them together into a **structure**.

Variable Definition	Data Held
`int empNumber;`	Employee number
`string name;`	Employee's name
`double hours;`	Hours worked
`double payRate;`	Hourly pay rate
`double grossPay;`	Gross pay

Example `struct` Declaration

- Before a structure can be used, it must be declared.
- The declaration inside curly brackets must have **;** after closing **}**.
- **`struct`** names commonly begin with an uppercase letter.
- Multiple fields of same type can be in comma-separated list.

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

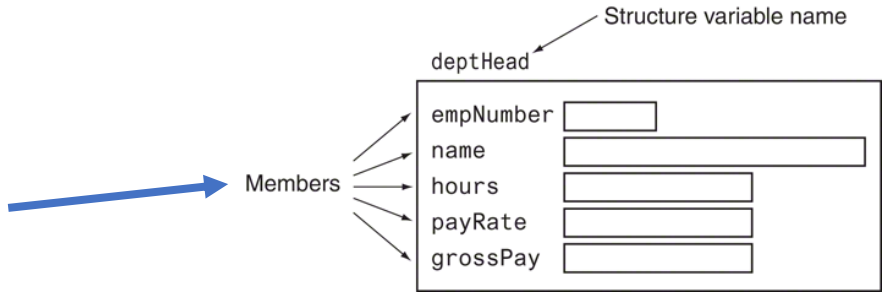
structure tag

structure members

Abstract Data Types

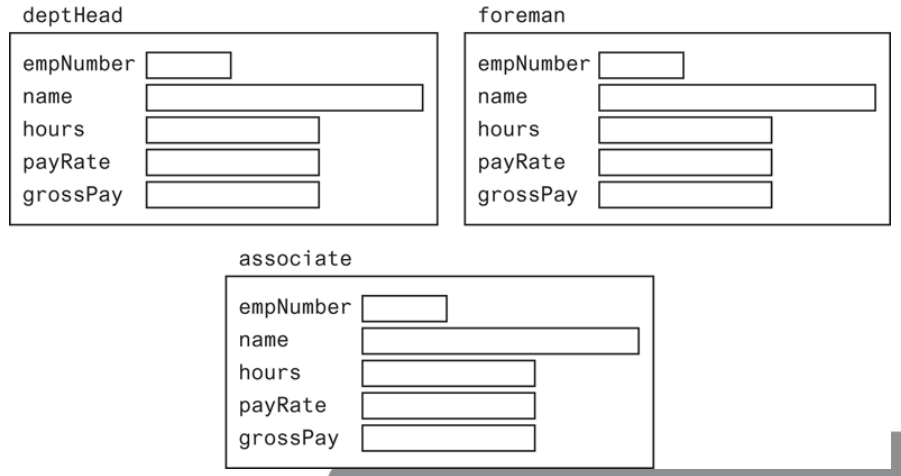
- Note the **`struct`** declaration in our example does **not** define a variable.
  - It simply tells the compiler what a `PayRoll` structure is made of.
  - It creates a new data type named **`PayRoll`**.

```
struct PayRoll
{
    int empNumber;
    string name;
    double hours, payRate, grossPay;
};
```



  - We can define variables of this type with simple definition statements, just as we would with any other data type.

```
PayRoll deptHead, foreman, associate;
```



- Each of the variables defined in this example is a separate **instance** of the PayRoll structure and contains its own members.

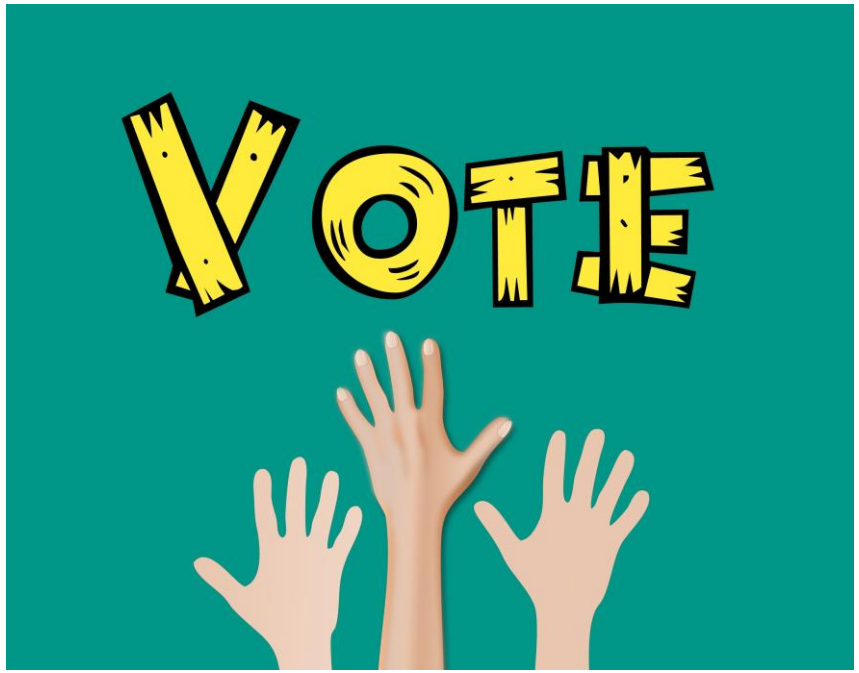
Poll 1 (Extra Credit)

A struct declaration allocates memory for the structure's member variables.

a) True

b) False

Please use the "Poll" window to participate for extra credit! One answer only please!



DOUGLAS COLLEGE

Accessing Structure Members

Accessing Structure Members

- The dot operator (`.`) allows us to access structure members in a program.
- The following statement demonstrates how to access the **`empNumber`** member:

```
deptHead.empNumber = 475;
```

- With the dot operator, we can use member variables just like regular variables.

```
cout << deptHead.empNumber << endl;
cin >> deptHead.grossPay;
```

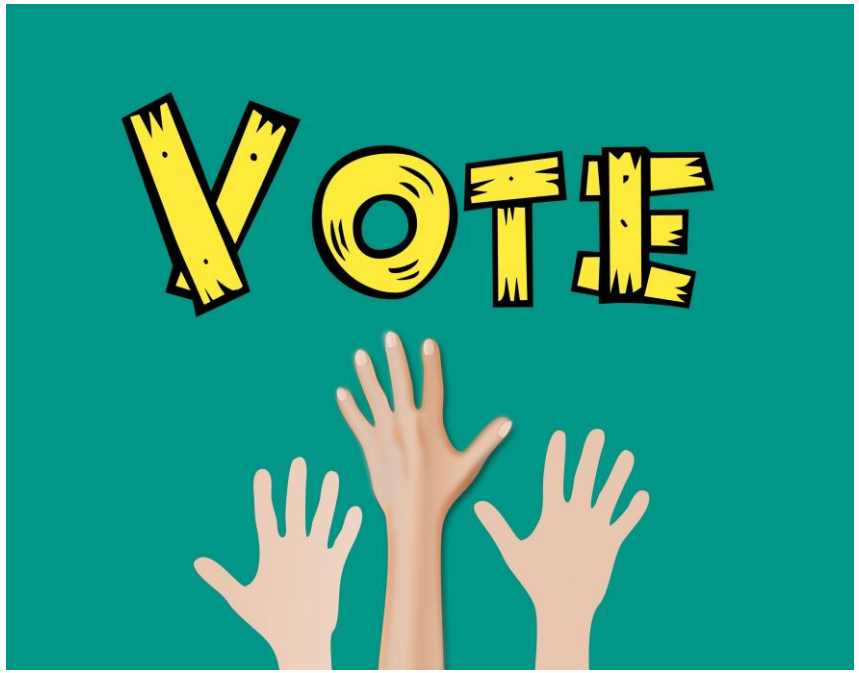
Poll 2 (Extra Credit)

The contents of a structure variable can be displayed by passing the entire variable to `cout`.

a) True

b) False

Please use the "Poll" window to participate for extra credit! One answer only please!



Accessing Structure Members

- The dot operator (`.`) allows us to access structure members in a program.
- The following statement demonstrates how to access the **`empNumber`** member:

```
deptHead.empNumber = 475;
```

- With the dot operator, we can use member variables just like regular variables.

```
cout << deptHead.empNumber << endl;
cin >> deptHead.grossPay;
```

- The contents of a structure variable **cannot** be displayed by passing the entire variable to **`cout`**.

```
cout << employee << endl;  // Will not work!
```

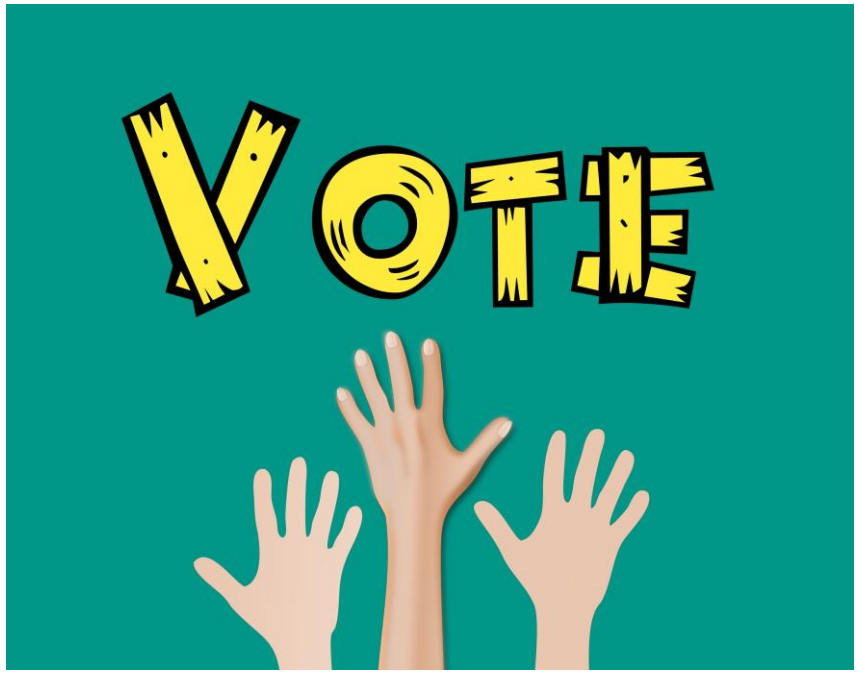
Poll 3 (Extra Credit)

We can perform comparison operations directly on structure variables. For example, assume `circle1` and `circle2` are `Circle` structure variables. The following statement is syntactically legal.

```
if (circle1 == circle2)
```

a) True

b) False



Please use the "Poll" window to participate for extra credit! One answer only please!

Accessing Structure Members

- The dot operator (**.**) allows us to access structure members in a program.
- The following statement demonstrates how to access the **empNumber** member:

```
deptHead.empNumber = 475;
```

- With the dot operator, we can use member variables just like regular variables.

```
cout << deptHead.empNumber << endl;
cin >> deptHead.grossPay;
```

- The contents of a structure variable **cannot** be displayed by passing the entire variable to **cout**.

```
cout << employee << endl;  // Will not work!
```

- We **cannot** perform comparison operations directly on structure variables.

```
if (deptHead1 == deptHead2) // Will not work!
```

DOUGLAS COLLEGE

Initializing a Structure

Initializing a Structure

- The members of a structure variable may be initialized with starting values when the structure variable is defined.
- Assume the following structure declaration exists in a program:

```
struct CityInfo
{
    string cityName;
    string province;
    long population;
    int distance;
};
```

- A variable may then be defined with an initialization list, as shown in the following:

```
CityInfo location = { "Asheville", "BC", 80000, 28 };
```

- We do **not** have to provide initializers for all the members of a structure variable.

```
CityInfo location = { "New Westminster" };
```

- **If we leave a structure member uninitialized, we must leave all the members that follow it uninitialized as well. C++ does not provide a way to skip members in a structure.**

Initializing a Structure

- The members of a structure variable may be initialized with starting values when the structure variable is defined.
- Assume the following structure declaration exists in a program:

```
struct CityInfo
{
    string cityName;
    string province;
    long population;
    int distance;
};
```

Since C++11, we can omit the = sign in the initialization statement.

- A variable may then be defined with an initialization list, as shown in the following:

```
CityInfo location = { "Asheville", "BC", 80000, 28 };
```

- We do **not** have to provide initializers for all the members of a structure variable.

```
CityInfo location = { "New Westminster" };
```

- If we leave a structure member uninitialized, we must leave all the members that follow it uninitialized as well. C++ does not provide a way to skip members in a structure.

Example

```
// This program demonstrates partially initialized
// structure variables.
#include <iostream>
#include <string>
#include <iomanip>
using namespace std;

struct EmployeePay
{
    string name;     // Employee name
    int empNum;          // Employee number
    double payRate;      // Hourly pay rate
    double hours;        // Hours worked
    double grossPay;     // Gross pay
};

int main()
{
    EmployeePay employee1 = { "Doja Cat", 141, 18.75 };
    EmployeePay employee2 = { "Justin Bieber", 142, 17.50 };

    cout << fixed << showpoint << setprecision(2);

    // Calculate pay for employee1
    cout << "Name: " << employee1.name << endl;
    cout << "Employee Number: " << employee1.empNum << endl;
    cout << "Enter the hours worked by this employee: ";
    cin >> employee1.hours;
    employee1.grossPay = employee1.hours * employee1.payRate;
    cout << "Gross Pay: " << employee1.grossPay << endl << endl;

    // Calculate pay for employee2
    cout << "Name: " << employee2.name << endl;
    cout << "Employee Number: " << employee2.empNum << endl;
    cout << "Enter the hours worked by this employee: ";
    cin >> employee2.hours;
    employee2.grossPay = employee2.hours * employee2.payRate;
    cout << "Gross Pay: " << employee2.grossPay << endl;
    return 0;
}
```

DOUGLAS COLLEGE

Arrays of Structures

Arrays of Structures

- An array of structures is defined like any other array. Assume the following structure declaration exists in a program:

```
struct BookInfo
{
    string title;
    string author;
    string publisher;
    double price;
};
```

- The following statement defines an array, **bookList**, that has **20** elements.

```
BookInfo bookList[20];
```

- The following expression refers to the title member of **bookList[5]**:

```
bookList[5].title;
```

- The following loop steps through the array, displaying the data stored in each element:

```
for (int index = 0; index < 20; index++)
        cout << bookList[index].title << endl;
```

Initializing a Structure Array

- To initialize a structure array, simply provide an initialization list for one or more of the elements.
- For example, given the following **`struct`** definition:

```
struct PayInfo
{
    int hours;          // Hours worked
    double payRate;     // Hourly pay rate
};
```

- We could initialize an array of **`PayInfo`** structures like this:

```
const int NUM_WORKERS = 3;      // Number of workers
PayInfo workers[NUM_WORKERS] = {
                               {10, 9.75 },
                               {15, 8.62 },
                               {40, 15.65}
};
```

DOUGLAS COLLEGE

Nested Structures

Nested Structures

- It is possible for a structure variable to be a member of another structure variable. consider the following structure declarations:

```
struct Costs
{
    double wholesale;
    double retail;
};
struct Item
{
    string partNum;
    string description;
    Costs pricing;
};
```

- Assume the variable **widget** is defined as follows:

```
Item widget;
```

- The following statements show examples of accessing members of the **pricing** variable, which is inside **widget**:

```
widget.pricing.wholesale = 100.0;
widget.pricing.retail = 150.0;
```

DOUGLAS COLLEGE

Structures as Function Arguments

Structures as Function Arguments

- Structure variables may be passed as arguments to functions.
- assume the following structure declaration exists in a program:

```
struct Rectangle
{
    double length;
    double width;
    double area;
};
```

- And let us imagine the following function definition exists in the same program:

```
double multiply(double x, double y)
{
    return x * y;
}
```

- Assuming **box** is a variable of the **Rectangle** structure type, the following function call will pass **box.length** into x and **box.width** into **y**. The return value will be stored in **box.area**

```
box.area = multiply(box.length, box.width);
```

Structures as Function Arguments

- Sometimes it is more convenient to pass an entire **`struct`** variable into a function instead of individual members.
- For example, the following function definition uses a **`Rectangle`** structure variable as its input parameter:

```
void showRect(Rectangle r)
{
     cout << r.length << endl;
     cout << r.width << endl;
     cout << r.area << endl;
}
```

- The following function call passes the **box** variable into **`r`**:

```
showRect(box);
```

- If a function is to access the members of the original argument, a reference to the **`struct`** variable may be used as the parameter.

```
showRect(box);

void showRect(Rectangle r)
{
     cout << r.length << endl;
     cout << r.width << endl;
     cout << r.area << endl;
}
```

Example

```
// This program has functions that accept structure variables
// as arguments.
#include <iostream>
#include <string>
#include <iomanip>
using namespace std;

struct InventoryItem
{
    int partNum;                  // Part number
    string description; // Item description
    int onHand;                   // Units on hand
    double price;                 // Unit price
};

void getItem(InventoryItem& p)   // Uses a reference parameter
{
    // Get the part number.
    cout << "Enter the part number: ";
    cin >> p.partNum;

    // Get the part description.
    cout << "Enter the part description: ";
    cin.ignore();  // Ignore the remaining newline character
    getline(cin, p.description);

    // Get the quantity on hand.
    cout << "Enter the quantity on hand: ";
    cin >> p.onHand;

    // Get the unit price.
    cout << "Enter the unit price: ";
    cin >> p.price;
}

void showItem(InventoryItem p)
{
    cout << fixed << showpoint << setprecision(2);
    cout << "Part Number: " << p.partNum << endl;
    cout << "Description: " << p.description << endl;
    cout << "Units On Hand: " << p.onHand << endl;
    cout << "Price: $" << p.price << endl;
}

int main()
{
    InventoryItem part;

    getItem(part);
    showItem(part);
    return 0;
}
```

Constant Reference Parameters

- Sometimes structures can be quite large.
- Passing large structures by value can decrease a program's performance **because a copy of the structure has to be created**.
- **When a structure is passed by reference, however, it isn't copied**. A reference that points to the original argument is passed instead.
- So, it is often preferable to pass large objects such as structures by reference.
- The disadvantage of passing an object by reference is that the function has access to the original argument – it can accidentally change the argument's value.
- This can be prevented by passing the argument as **a constant reference**.

```
void showItem(const InventoryItem& p)
{
    cout << fixed << showpoint << setprecision(2);
    cout << "Part Number: " << p.partNum << endl;
    cout << "Description: " << p.description << endl;
    cout << "Units On Hand: " << p.onHand << endl;
    cout << "Price: $" << p.price << endl;
}
```

DOUGLAS COLLEGE

Returning a Structure from a Function

Returning a Structure

- A function may **`return`** a structure.
- Here, it is necessary for the function to have a local structure variable to hold the member values that are to be returned.
- We already that C++ only allows us to **`return`** a single value from a function.
- Structures provide a way around this limitation!
- Although a structure may have several members, a structure variable is a single value.

```
// This program uses a function to return a structure. This
// is a modification of Program 11-2.
#include <iostream>
#include <iomanip>
#include <cmath>  // For the pow function
using namespace std;

// Constant for Pi.
const double PI = 3.14159;

// Structure declaration
struct Circle
{
    double radius;      // A circle's radius
    double diameter;    // A circle's diameter
    double area;        // A circle's area
};

// Function prototype
Circle getCircle();

int main()
{
    Circle c;      // Define a structure variable

    // Get data about the circle.
    c = getCircle();

    // Display the circle data.
    cout << "The radius and area of the circle are:\n";
    cout << fixed << setprecision(2);
    cout << "Radius: " << c.radius << endl;
    cout << "Area: " << c.area << endl;
    return 0;
}

Circle getCircle()
{
    Circle tempCircle;  // Temporary structure variable

    // Store circle data in the temporary variable.
    cout << "Enter the diameter of a circle: ";
    cin >> tempCircle.diameter;

    // Calculate the circle's radius and area.
    tempCircle.radius = tempCircle.diameter / 2.0;
    tempCircle.area = PI * pow(tempCircle.radius, 2.0);

    // Return the temporary variable.
    return tempCircle;
}
```

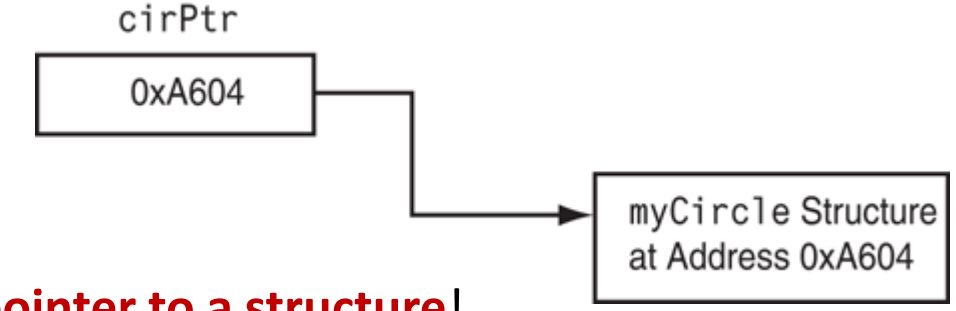
DOUGLAS COLLEGE

Pointers to Structures

Pointers to Structures

- Defining a variable that is a pointer to a structure is as simple as defining any other pointer variable:

```
Circle myCircle = { 10.0, 20.0, 314.159 };
Circle* cirPtr = nullptr;
cirPtr = &myCircle;
```



- **Be extremely careful when dereferencing a pointer to a structure**!

```
*cirPtr.radius = 10; // Illegal!!!
```

- The dot operator has higher precedence than the indirection operator, so the indirection operator tries to dereference **`cirPtr.radius`**, not **`cirPtr`**.
- To dereference the **`cirPtr`** pointer, a set of parentheses must be used.

```
(*cirPtr).radius = 10; // OK
```

- **Due to the awkwardness of this notation, C++ has a special operator for dereferencing pointers to structures**.

The Structure Pointer Operator

- C++ has a special operator for dereferencing pointers to structures.
- It consists of a hyphen (**-**) followed by the greater-than symbol (**>**).
- Consider the statement from the previous example:

```
(*cirPtr).radius = 10; // OK
```

- This statement is exactly equivalent to:

```
cirPtr->radius = 10;
```

- The structure pointer operator takes the place of the dot operator in statements using pointers to structures.
- The operator automatically dereferences the structure pointer on its left. There is no need to enclose the pointer name in parentheses.
- The structure pointer operator is supposed to look like an arrow, thus visually indicating that a “pointer” is being used.

Example

```
// This program demonstrates a function that uses a
// pointer to a structure variable as a parameter.
#include <iostream>
#include <string>
#include <iomanip>
using namespace std;

struct Student
{
    string name;  // Student's name
    int idNum;              // Student ID number
    int creditHours;        // Credit hours enrolled
    double gpa;  // Current GPA
};

void getData(Student*);  // Function prototype

int main()
{
    Student freshman;

    // Get the student data.
    cout << "Enter the following student data:\n";
    getData(&freshman);     // Pass the address of freshman.
    cout << "\nHere is the student data you entered:\n";

    // Now display the data stored in freshman
    cout << setprecision(3);
    cout << "Name: " << freshman.name << endl;
    cout << "ID Number: " << freshman.idNum << endl;
    cout << "Credit Hours: " << freshman.creditHours << endl;
    cout << "GPA: " << freshman.gpa << endl;
    return 0;
}

void getData(Student* s)
{
    // Get the student name.
    cout << "Student name: ";
    getline(cin, s->name);

    // Get the student ID number.
    cout << "Student ID Number: ";
    cin >> s->idNum;

    // Get the credit hours enrolled.
    cout << "Credit Hours Enrolled: ";
    cin >> s->creditHours;

    // Get the GPA.
    cout << "Current GPA: ";
    cin >> s->gpa;
}
```

DOUGLAS COLLEGE

Dynamically Allocating a Structure

Dynamically Allocating a Structure

- We can also use a structure pointer and the new operator to dynamically allocate memory for a structure variable.
- For example, the following code defines a **`Circle`** pointer named **`cirPtr`** and dynamically allocates a **`Circle`** structure.
- Values are then stored in the dynamically allocated structure's members.

```
Circle* cirPtr = nullptr;   // Define a Circle pointer
cirPtr = new Circle;        // Dynamically allocate a Circle structure
cirPtr->radius = 10;        // Store a value in the radius member
cirPtr->diameter = 20;      // Store a value in the diameter member
cirPtr->area = 314.159;     // Store a value in the area member
```

- We can also dynamically allocate an array of structures.

```
Circle* circles = nullptr;
circles = new Circle[5];
for (int count = 0; count < 5; count++)
{
    cout << "Enter the radius for circle "
        << (count + 1) << ": ";
    cin >> circles[count].radius;
}
```

DOUGLAS COLLEGE

Enumerated Data Types

Enumerated Data Types

- An **enumerated data type** is a programmer-defined data type **whose value is restricted to a range of values** known as **enumerators**, which represent integer constants.
- An enumerated type declaration begins with the key word **enum**, followed by the name of the type, followed by a list of identifiers inside braces, and is terminated with a semicolon.
- The example declaration below creates an enumerated data type named **Day**.

```
enum Day { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY };
```

- The identifiers **MONDAY**, **TUESDAY**, **WEDNESDAY**, **THURSDAY**, and **FRIDAY**, which are listed inside the braces, are known as **enumerators**.
- These enumerators represent the values that belong to the **Day** data type.
- Enumerators must be legal C++ identifiers.
- The following statement assigns the value **WEDNESDAY** to the **workDay** variable:

```
Day workDay = WEDNESDAY;
```

Enumerated Data Types

- So, what exactly is an enumerator?
- Think of it as an integer named constant.
- Internally, the compiler assigns integer values to the enumerators, beginning at 0.
- Therefore, the following decleration

```
enum Day { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY };
```

- will be represented in memory by:

```
MONDAY    = 0
TUESDAY   = 1
WEDNESDAY = 2
THURSDAY  = 3
FRIDAY    = 4
```

- Using the Day declaration, the following code

```
cout << MONDAY << ", " << TUESDAY << ", " << WEDNESDAY << ", " << THURSDAY << ", " << FRIDAY << endl;
```

- will produce **0, 1, 2, 3, 4**

Assigning an integer to an enum Variable

- We **cannot** directly assign an **`int`** value to an enum variable. This will **not** work:

```
workDay = 3;// Error!
```

- When assigning a value to an **`enum`** variable, we should use a valid enumerator.
- However, if circumstances require that we store an **`int`** value in an **`enum`** variable, we can do so by casting the integer.

```
workDay = static_cast<Day>(3);
```

- This statement will produce the same results as:

```
Day workDay = THURSDAY;
```

Assigning an Enumerator to an `int` Variable

- Although we cannot directly assign an **`int`** value to an **`enum`** variable, we can directly assign an enumerator to an **`int`** variable.

```
int x;
x = THURSDAY;
cout << x << endl;
```

- When this code runs, it will display 4.

- We can also assign a variable of an enumerated type to an int variable, as shown here:

```
Day workDay = FRIDAY;
int x = workDay;
cout << x << endl;
```

- When this code runs, it will display 3.

Comparing Enumerator Values and Operating on Them

- Enumerator values can be compared using the relational operators.
- The following code will display the message "**Friday is greater than Monday.**":

```
if (FRIDAY > MONDAY)
        cout << "Friday is greater than Monday. \n";
```

- You can also compare enumerator values with integer values.

```
if (MONDAY == 0)
        cout << "Monday is equal to zero.\n";
```

- We **cannot** perform math operations with enum variables.

```
day2 = day1 + 1;  // ERROR! This will not work!
```

- We would need to use an explicit cast to convert the result to a **Day** variable.

```
day2 = static_cast<Day>(day1 + 1);  // This works.
```

Example

```
// This program demonstrates an enumerated data type.
#include <iostream>
#include <iomanip>
using namespace std;

enum Day { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY };

int main()
{
    const int NUM_DAYS = 5;     // The number of days
    double sales[NUM_DAYS];     // To hold sales for each day
    double total = 0.0;         // Accumulator
    int index;                  // Loop counter

    // Get the sales for each day.
    for (index = MONDAY; index <= FRIDAY; index++)
    {
        cout << "Enter the sales for day "
            << index << ": ";
        cin >> sales[index];
    }

    // Calculate the total sales.
    for (index = MONDAY; index <= FRIDAY; index++)
        total += sales[index];

    // Display the total.
    cout << "The total sales are $" << setprecision(2)
        << fixed << total << endl;

    return 0;
}
```

Example

```
// This program demonstrates an enumerated data type.
#include <iostream>
#include <iomanip>
using namespace std;

enum Day { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY };

int main()
{
    const int NUM_DAYS = 5;     // The number of days
    double sales[NUM_DAYS];     // To hold sales for each day
    double total = 0.0;         // Accumulator
    int index;                  // Loop counter

    // Get the sales for each day.
    for (index = MONDAY; index <= FRIDAY; index++)
    {
        cout << "Enter the sales for day "
             << index << ": ";
        cin >> sales[index];
    }

    // Calculate the total sales.
    for (index = MONDAY; index <= FRIDAY; index++)
        total += sales[index];

    // Display the total.
    cout << "The total sales are $" << setprecision(2)
         << fixed << total << endl;

    return 0;
}
```

Program does not define a **Day** variable. So, we do not have to name the data type. This would make the **enum** **anonymous**.

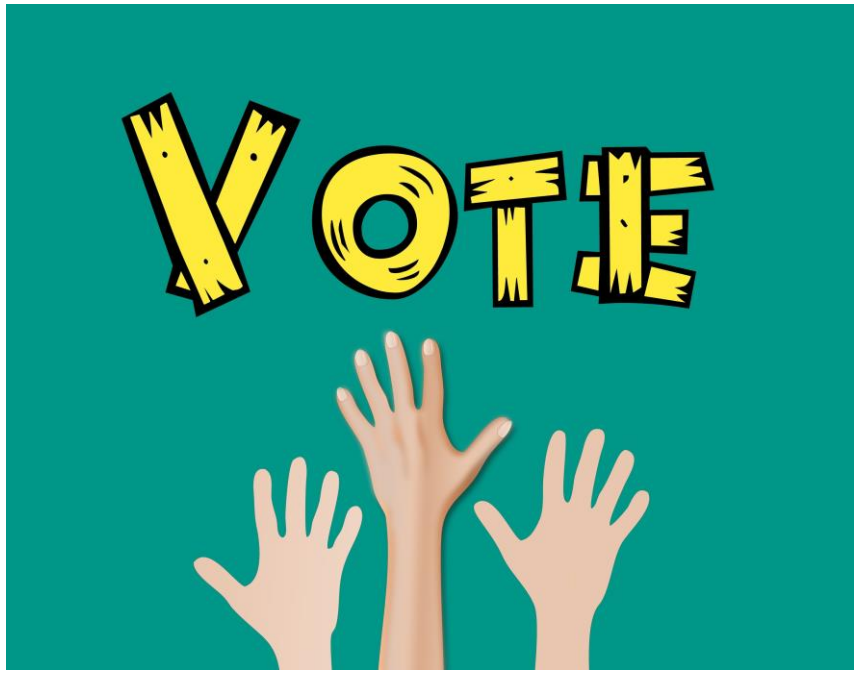
Poll 4 (Extra Credit)

The following expressions are both syntactically legal if Day is an `enum` type.

```
Day workDay = MONDAY;
workDay++;
```

a) True

b) False



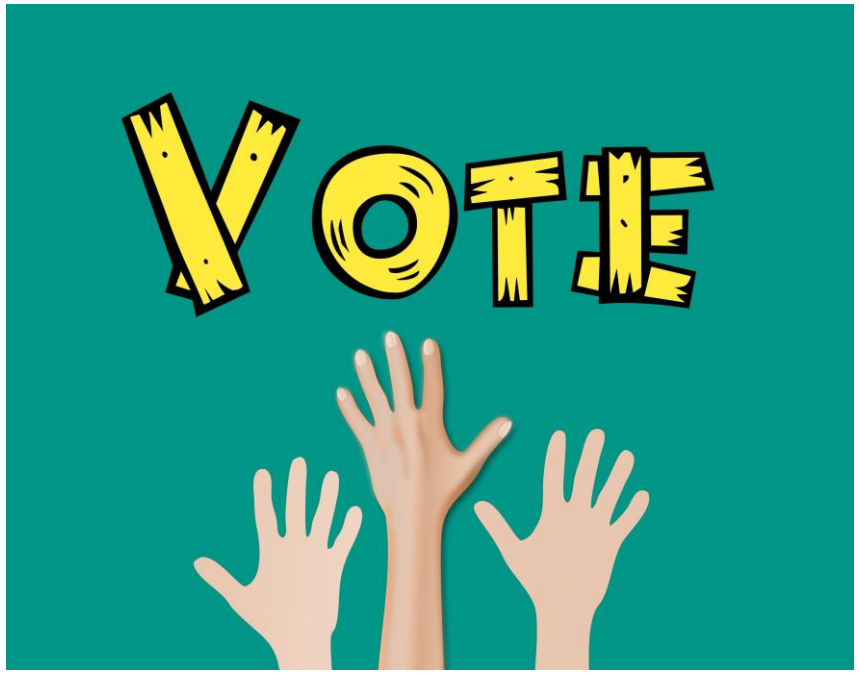
Please use the "Poll" window to participate for extra credit! One answer only please!

Poll 4 (Extra Credit)

The following expressions are both syntactically legal if Day is an `enum` type.

```
Day workDay = MONDAY;
workDay++;
```

This does not work wither because the increment operator does not work with an `enum` variable.



Please use the "Poll" window to participate for extra credit! One answer only please!

Specifying Integer Values for Enumerators

- By default, the enumerators in an enumerated data type are assigned the integer values 0, 1, 2, etc.
- If this is not appropriate, we can specify the values to be assigned:

```
enum Water { FREEZING = 32, BOILING = 212 };
```

- If we leave out the value assignment for one or more of the enumerators, it will be assigned a default value.

```
enum Colors { RED, ORANGE, YELLOW = 9, GREEN, BLUE };
```

- In the above example, the enumerator **RED** will be assigned the value **0**, **ORANGE** will be assigned the value **1**, **YELLOW** will be assigned the value **9**, **GREEN** will be assigned the value **10**, and **BLUE** will be assigned the value **11**.

Enumerator Scope

- Enumerators must be unique within the same scope.
- For example, an error will result if both of the following enumerated types are declared within the same scope.

```
enum Presidents { MCKINLEY, ROOSEVELT, TAFT };
enum VicePresidents { ROOSEVELT, FAIRBANKS, SHERMAN };  // Error!
```

- In the above example, the reason is that **ROOSEVELT** is declared twice.
- The following declarations will also cause an error if they appear within the same scope.

```
enum Status { OFF, ON };
const int OFF = 0;          // Error!
```

Using Strongly Typed enums in C++ 11

- We mentioned earlier that we cannot have multiple enumerators with the same name, within the same scope.
- Since C++ 11, we can use a new type of **enum**, known as a strongly typed **enum** (also known as an **enum** class), to get around this limitation.

```
enum class Presidents { MCKINLEY, ROOSEVELT, TAFT };
enum class VicePresidents { ROOSEVELT, FAIRBANKS, SHERMAN };
```

- Although both **enums** contain the same enumerator (**ROOSEVELT**), these declarations will compile without an error.
- When we use a strongly typed enumerator, we must prefix the enumerator with the name of the **enum**, followed by the **::** operator (i.e., the **scope resolution operator**).

```
Presidents prez = Presidents::ROOSEVELT;
VicePresidents vp = VicePresidents::ROOSEVELT;
```

Using Strongly Typed enums in C++ 11

- If we want to retrieve a strongly typed enumerator's underlying **int** value, we must use a cast operator.

```
int x = static_cast<int>(Presidents::ROOSEVELT);
```

- When we declare a strongly typed **enum**, we can optionally specify any integer data type as the underlying type.
- To do so, we simply add a colon (**:**) after the **enum** name, followed by the desired data type.
- For example, the following statement declares an **enum** that uses the **char** data type for its enumerators:

```
enum class Day : char { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY };
```

In-Class Exercise #1

Complete the following program skeleton.

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

int main()
{
    char place[] = "The Windy City";
    // Complete the program. It should search the array place
    // for the string "Windy" and display the message "Windy
    // found" if it finds the string. Otherwise, it should
    // display the message "Windy not found."
    return 0;
}
```

Thank you.

DOUGLAS COLLEGE

DOUGLAS COLLEGE