**STRUCTURES**

* 2nd aggregate data type: struct
* Recall: aggregate meaning “grouping”
  + Recall array: collection of values of same type
  + Structure: collection of values of different types
* Treated as a single item, like arrays
* Major difference: Must first “define” struct
  + Prior to declaring any variables
* Define struct globally (typically)
* No memory is allocated
  + Just a “placeholder” for what our struct will “look like”
* Definition:

struct CDAccountV1 🡪 Name of new struct “type” (Cash divided account)

{ (Vadeli hesap)

double balance; 🡪 Member names (bakiye, faiz oranı, vade)

double interestRate;

int term;

};

* With structure type defined, now declare variables of this new type:

CDAccountV1 account, a;

* + Just like declaring simple types
  + Variable *account* now of type CDAccountV1
  + It contains “member values”
    - Each of the struct “parts”
* Different structs can have same name member variables, no conflicts

cout << account; 🡪 COMPILER ERROR!

a == account 🡪 COMPILER ERROR!

a = account; 🡪 NO PROBLEM (finds the starting address of a and account, and does a

memcopy between those addresses)

sizeof(account); 🡪 NO PROBLEM

CDAccountV1 \*ap = &a; 🡪 NO PROBLEM

ap -> balance = 2100.00 🡪 NO PROBLEM

*//Program to demonstrate the CDAccountV1 structure type.*

#include <iostream>

**using namespace** std;

*//Structure for a bank certificate of deposit:*

**struct** CDAccountV1{

**double** balance;

**double** interestRate;

**int** term; //monts until maturity

};

**void** getData(CDAccountV1& theAccount);

*//Postcondition: theAccount.balance, theAccount.interestRate, and*

*//theAccount.term have been given values that the user entered at the keyboard.*

void printData(**const** CDAccountV1& account);

**int** main(){

CDAccountV1 account;

getData(account);

printData(account);

return 0;

}

**void** printData(**const** CDAccountV1& account){

**double** rateFraction, interest;

rateFraction = account.interestRate/100.0;

interest = account.balance \* (rateFraction\*(account.term/12.0));

account.balance = account.balance + interest;

cout.setf(ios::fixed);

cout.setf(ios::showpoint);

cout.precision(2);

cout << **“When your CD matures in “**

<< account.term << “ months, \n”

<< “it will have a balance of $”

<< account.balance << endl;

}

**void** getData(CDAccountV1& theAccount){

cout << “Enter account balance: $;

cin >> theAccount.balance;

cout << “Enter account of interest rate: “;

cin >> theAccount.interestRate;

cout << “Enter the number of months until maturity: “;

cin >> theAccount.term;

}

**struct** CDAccountV1{

**double** balance;

**double** interestRate;

**int** term;

} a1 , a2;

*If I do this, now I have 2 global struct variables. If you just use “;” that means that I am just defining the struct, I am not declaring any variables.*

*This is one of the most important differences between JAVA and C++. In JAVA, they don’t use “;”.*

**struct** CDAccountV1{

**double** balance = 0.0;

**double** interestRate;

**int** term;

} a1 , a2;

*In C++11 you can initialize struct members.*

**Structures as Function Arguments**

* Passed like any simple data type
  + Pass-by-value void f(CDAccountV1 a);
  + Pass-by-reference void g(const CDAccountV1& a);
  + Or combination void h(CDAccountV1\* ap);
* Can also be returned by function
  + Return-type is structure type
  + Return statement in function definition sends structure variable back to caller

!!!We can’t dereference an integer. 🡪 int i; We can’t do “\*i”.

* We can initialize the struct at declaration:

struct Date

{

int month;

int day;

int year;

};

Date dueDate = {12, 31, 2003};

* Declaration provides initial data to all 3 member variables.

**CLASSES**

* Similar to structures
  + Adds member FUNCTIONS
  + Not just member data
* Don’t occupy any memory itself
* Integral to object-oriented programming
  + Focus on objects
    - Object: Contains data and operations
    - In C++, variables of class type are objects

struct Rational{

int a;

int b;

void print();

};

Rational r1 = {1, 2};

r1.print(); 🡪 1/2 (we didn’t give parameters, a and b are inside the r1)

r3 = r1.add(r2); 🡪 If I have other member functions, it would be possible to write this.

r1 is an object, not a variable

**Class Definitions**

Defined similar to structures:

class DayOfYear 🡪 name of new class type

{

public:

void output(); 🡪 member function

int month;

int day;

};

Notice only member function’s prototype

* Function’s implementation is elsewhere

**Declaring Objects**

* Declared same as all variables
  + Predefined types, structure types
* Example:

DayOfYear today, birthday;

* + *Declares 2 objects of class type DayOfYear*
* Objects include:
  + Data
    - Members month, day
  + Operations (member functions)
    - output()

**Class Member Access**

* Members accessed same as structures
* Example:

today.month

today.day

* + And to access member function:

today.output(); 🡪 Invokes member function

**Class Member Functions**

* Must define or “implement” class member functions
* Like other function definitions
  + Can be after main() definition
  + Must specify class:

void DayOfYear::output()

{…}

* + - :: is scope resolution operator
    - Instructs compiler “what class” member is from
    - Item before :: called type qualifier

#include <iostream>

using namespace std;

class DayOfYear

{

public:

int month;

int day;

void output();

};

int main(){

DayOfYear today, birthday;

//size of today is size of 2 integers: month and day

//size of birthday is size of 2 integers: month and day

//function is same for today and birthday

cout << “Enter today’s date:\n”;

cout << “Enter month as a number: ”;

cin >> today.month;

cout << “Enter the day of the month: “;

cin >> today.day;

cout << “Enter your birthday:\n”;

cout << “Enter month as a number: “;

cin >> birthday.month;

cout << “Enter the day of the month: “;

cin >> birthday.day;

cout << “Today’s date is “;

today.output();

cout << endl;

cout << “Your birthday is “;

birthday.output();

cout << endl;

if (today.month == birthday.month && today.day == birthday.day)

cout << “Happy Birthday!\n”;

else

cout << “Happy Unbirthday!\n”;

return 0;

}

*//Uses iostream:*

void DayOfYear::output()

{

switch (month)

{

case 1:

cout << “January “; break;

case 2:

cout << “February “; break;

case 3:

cout << “March “; break;

case 4:

cout << “April “; break;

case 5:

cout << “May “; break;

case 6:

cout << “June “; break;

case 7:

cout << “July “; break;

case 8:

cout << “August “; break;

case 9:

cout << “September “; break;

case 10:

cout << “October “; break;

case 11:

cout << “November “; break;

case 12:

cout << “December “; break;

default:

cout << “Error in DayOfYear::output. Contact software vendor.”;

}

cout << day;

*//Month and day come inside today or birthday*

}

Usually in OOP, when you call functions, those are mostly function of the classes and you don’t see many parameters. Because all the data they need to work on are already inside the classes object.

**Dot and Scope Resolution Operator**

* Used to specify “of what thing” they are members
* Dot operator:
  + Specifies member of particular object
* Scope resolution operator:
  + Specifies what class the function definition comes from

**A Class’s Place**

* Class is full-fledged type!
  + Just like data types int, double, etc.
* Can have variables of a class type
  + We simply call them “objects”
* Can have parameters of a class type
  + Pass-by-value
  + Pass-by-reference
* Can use class type like any other type

**Encapsulation**

* Putting all the data and related function inside the same capsul
* Any data type includes
  + Data (range of data)
  + Operations (that can be performed on data)
* Example:
  + *int* data type has:
    - Data: -2147483648 to 2147483647 (for 32 bit int)
    - Operations: +, -, \*, /, %, logical, etc.
* Same with classes
  + But we specify data, and the operations to be allowed on our data.

**Abstract Data Types**

* “Abstract”
  + Programmers don’t know details
* Abbreviated “ADT”
  + Collection of data values together with set of basic operations defined for the values
* ADT’s often “language-independent”
  + We implement ADT’s in C++ with classes
    - C++ class “defines” the ADT
  + Other languages implement ADT’s as well

If 2 classes are not related, they cannot share the same function.

**More Encapsulation**

* Encapsulation
  + Means “bringing together as one”
* Declare a class 🡪 get an object
* Object is “encapsulation” of
  + Data values
  + Operations on the data (member functions)

**Principles of OOP**

* Information Hiding (how the functions work, what kind of data we have in a class)
  + Details of how operations work not known to “user” of class
* Data Abstraction
  + Details of how data is manipulated withing ADT/class not known to user
* Encapsulation
  + Bring together data and operations, but keep “details” hidden

**Public and Private Members**

* Data in class almost always designated private in definition!
  + Upholds principles of OOP
  + Hide data from user
  + Allow manipulation only via operations
    - Which are member functions
* Public items (usually member functions) are “user-accessible”

![Graphical user interface, text, application

Description automatically generated]()

We can have a parameter for member function.

class DayOfYear

{

public:

int month;

int day;

void output();

bool equals(const DayOfYear &other);

bool lessThan(const DayOfYear &other);

DayOfYear nextDay();

};

int main(){

…

if (today.equals(birthday))

cout << “Happy Birthday!\n”;

DayOfYear next = today.nextDay();

…

}

bool DayOfYear::equals(const DayOfYear &other){

//if (month == other.month && day == other.day)

/\* I have access to private data in here but I am not using the

private data directly, I am using the public interface \*/

/\* I am hiding information from myself too bc if I later change

type of month and date, I want my internal code to not

affected that much. \*/

if (getMonth() == other.getMonth() && getDay() == other.getDay)

return true;

return false;

}

DayOfYear DayOfYear::nextDay(){

DayOfYear next;

int mv[] = {0, 31, 28, 30, 31, 30, …}

next.setDate((day+1) % mv[month],

month + (day+1)/30);

return next;

}

class DayOfYear

{

public:

void input(); *MUTATOR*

void output(); *ACCESSOR*

bool equals(const DayOfYear &other); *ACCESSOR*

DayOfYear nextDay(); *ACCESSOR (bc it’s not modifying MY object)*

int getDay();

int getMonth();

//these are getters, they simply return the day and month value

void setDate(int, int);

//this is setter, simply for assign value

/\*in this way, we can check the validity of inputs in this function, not let the user the assign invalid values.\*/

private:

int month;

int day;

//now objects have no direct access to month and day

//nobody can have access to these 2 variables month and day

//so now users can’t access today.day or today.month

/\*month and day can be used freely inside your class, inside

implementations of any of the public functions above.\*/

};

ALWAYS TRY TO USE SETTERS AND GETTERS IF THEY ARE APPROPRIATE.

DayOfYear today; 🡪 Object today can only access public members

**Public and Private Style**

* Can mix & match public & private
  + You can write public stuff first and then you can write private stuffs then you can again write public stuffs
* You can put functions and data to both public and private
* More typically place public first
  + Allows easy viewing of portions that can be USED by programmers using the class
  + Private data is “hidden”, so irrelevant to users
* Outside of class definition, cannot change (or even access) private data
* Class definitions are usually kept in header file.

TRY TO GIVE CUSTOMER AS LITTLE PRILIVAGE AS POSSIBLE.

**Accessor and Mutator Functions**

* Object needs to “do something” with its data
* Call accessor member functions (they just access the data, they don’t modify them - GETTERS)
  + Allow object to read data
  + Also called “get member functions”
  + Simple retrieval of member data
* Mutator member functions (they modify the object data, object itself - SETTERS)
  + Allow object to change data
  + Manipulated based on application

*TRY TO WRITE MORE ACCESSOR FUNCTIONS THAN MUTATOR FUNCTIONS.*

**Separate Interface and Implementation**

* User of class need not see details of how class is implemented
  + Principle of OOP 🡪 ENCAPSULATION
* User only needs “rules”
  + Called “interface” for the class (public section of the class)
    - In C++ 🡪 public member functions and associated comments
* Implementation of class hidden
  + Member function definitions elsewhere
  + User need not see them

**Structures versus Classes**

* Structures
  + Typically all members public by default
  + No member functions
* Classes
  + Typically all data members private by default
  + Interface member functions public
* Technically, same
  + Perceptionally, very different mechanism

**Thinking Objects**

* Focus for programming changes
  + Before (in C) 🡪 algorithms center stage, we write functions
  + OOP 🡪 data is focus
* Algorithms still exist
  + They simply focus on their data
  + Are “made” to “fit” the data
* Designing software solution
  + Define variety of objects and how they interact

#include <iostream>

#include <cstdling>

**using** **namespace** std;

**class** DayOfYear{

**public**:

**void** input(); ***//MUTATOR***

**void** output(); ***//ACCESSOR***

**void** set(**int** newMonth, **int** newDay); ***//MUTATOR***

*//Precondition: newMonth and newDay form a possible date.*

**void** set(**int** newMonth); ***//MUTATOR***

*//Precondition: 1 <= newMonth <= 12*

*//Postcondition: The date is set to the first day of the given month*

*//set function is overloaded!!!*

**int** getMonthNumber(); *//Returns 1 for January, 2 for February, etc.* ***//ACCESSOR***

**int** getDay(); ***//ACCESSOR***

**private**:

**int** month;

**int** day;

**};**

**int** main(){

DayOfYear today, bachBirthday;

cout << “Enter today’s date:\n”;

today.input();

cout << “Today’s date is “;

today.output();

cout << endl;

bachBirthday.set(3, 21);

cout << “J. S. Bach’s birthday is “;

bachBirthday.output();

cout <<endl;

**if** (today.getMonthNumber() == bachBirthday.getMonthNumber() &&

today.getDay() == bachBirthday.getDay())

cout << “Happy Birthday Johann Sebastian!\n”;

**else**

cout << “Happy Unbirthday Johann Sebastian!\n”;

**return** 0;

**}**

*//Uses iostream and cstdlib:*

**void** DayOfYear::set(**int** newMonth, **int** newDay{

**if** ((newMonth >= 1) && (newMonth <=12))

month = newMonth;

**else**{

cout << “Illegal month value! Program aborted.\n”;

exit(1);

}

**if** ((newDay >= 1) && (newDay <= 31))

day = newDay;

**else**{

cout << “Illegal day value! Program aborted.\n” ;

exit(1);

}

}

*//Uses iostream and cstdlib:*

**void** DayOfYear::set(**int** newMonth){

set(newMonth, 1);

}

**int** DayOfYear::getMonthNumber(){

**return** month;

**}**

**int** DayOfYear::getDay(){

**return** day;

**}**

*//Uses iostream and cstdlib:*

**void** DayOfYear::input(){

**int** m, d;

cout << “Enter the month as a number: ”;

cin >> m;

cout << “Enter the day of the month: ”;

cin >> d;

set(m, d);

}

void DayOfYear::output()

{

switch (month)

{

case 1:

cout << “January “; break;

case 2:

cout << “February “; break;

case 3:

cout << “March “; break;

case 4:

cout << “April “; break;

case 5:

cout << “May “; break;

case 6:

cout << “June “; break;

case 7:

cout << “July “; break;

case 8:

cout << “August “; break;

case 9:

cout << “September “; break;

case 10:

cout << “October “; break;

case 11:

cout << “November “; break;

case 12:

cout << “December “; break;

default:

cout << “Error in DayOfYear::output. Contact software vendor.”;

}

cout << day;

}