CS-202

C++ Structs

C. Papachristos
Autonomous Robots Lab
University of Nevada, Reno

Course Week

Course , Projects , Labs:

Monday	Tuesday	Wednesday	Thursday	Friday		Sunday
			Lab (8 Sections)			
	CLASS		CLASS			
PASS Session	PASS Session	**Project DEADLINE**	NEW Project	PASS Session		PASS Session

Your 2nd Project will be announced today Thursday 1/31.

1st Project Deadline was this Wednesday 1/30.

- NO Project accepted past the 24-hrs delayed extension (@ 20% grade penalty).
- Send what you have in time!

Today's Topics

C++ Structs

- C (basic) Structs
- C++ Context
- Struct vs Class

Structs and Arrays

Structs and Functions

C++ Structs

Description

A "Structure" is a collection of related data items, possibly of different types.

- A structure type in C++ is called `struct`.

A `struct` is *heterogeneous*:

- It can be composed of data of different types.

vs

An array is *homogeneous*:

- It can contain only data of the same type.

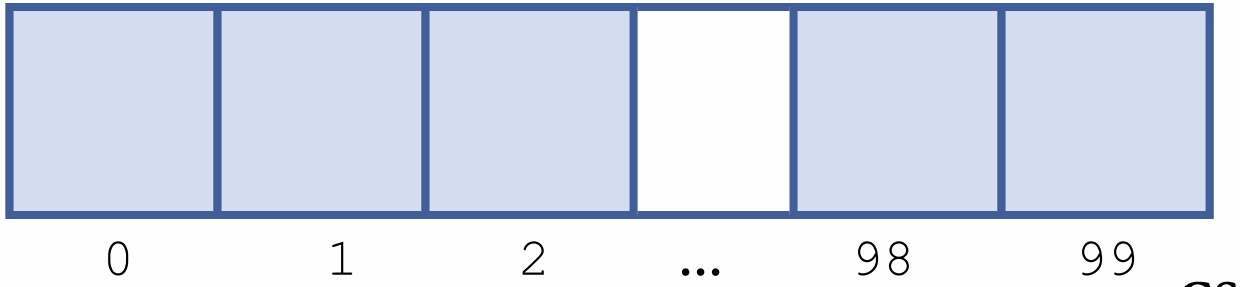
C++ Structs

Description

A "Structure" is a collection of related data items, possibly of different types.

A `struct` is *heterogeneous* in that it can be composed of data of different types.

In contrast, an array is *homogeneous* since it can contain only data of the same type.



C++ Structs

Description

Structures are used to hold data that *belong* together.

Examples:

- Student record: student id, name, major, gender, start year, …
- Bank account: account number, name, currency, balance, …
- Address book contact: name, address, telephone number, …

In database applications, structures are called records.

Members

Struct Members (or Fields):

- Individual components of a `struct` type.

Versatility:

Struct Members can be of different types:

- Simple
- Array
- Another `struct`

C++ Structs

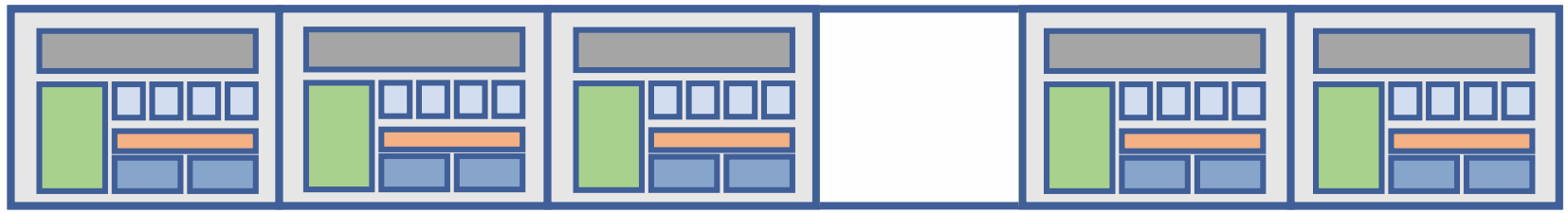
Members

Naming – Resolution:

- A `struct` is named as a whole.
- Individual Struct Members are named using *field identifiers*.

Versatility:

Complex data structures can be formed by defining arrays of `struct`s.



C++ Structs

Type Declaration

```
struct <struct-type> {
    <type> <identifier_list>;
    <type> <identifier_list>;
    ...
} ;
```

- Type Name is up to you to declare!
- Members in Brackets
- Semicolon

Example:

```
struct Date {
    int day;
    int month;
    int year;
} ;
```

or

```
struct Date {
    int day, month, year;
    int hours, minutes, seconds;
    long microseconds;
} ;
```

C++ Structs

Type Declaration

Examples:

```
struct StudentInfo {
    int id;
    int age;
    char gender;
    double gpa;
};
```

The ***StudentInfo*** structure has 4 members of different types.

```
struct StudentGrade {
    char name[9];
    char course[9];
    int lab[5];
    int homework[3];
    int exam[2];
};
```

The ***StudentGrade*** structure has 5 members of different array types.

C++ Structs

Type Declaration

Examples:

```
struct BankAccount {
    char name[15];
    int acountNo[10];
    double balance;
    Date birthday;
};
```

```
struct StudentRecord {
    char name[9];
    int id;
    char dept[4];
    char gender;
};
```

The `BankAccount` structure has simple, array and `struct` types as members.

The `StudentRecord` structure has 4 members.

C++ Structs

Variable Declaration

Declaration of a variable of `struct` type:

- NOTE: Type declaration must come first.

```
struct <struct-type> {
    <type> <identifier_list>;
    ...
} ;
```

```
struct StudentRecord {
    char name[9];
    int id;
    char dept[4];
    char gender;
};
```

Declaration of a new variable of that type:

```
<struct-type> <identifier_list>;
```

Example:

```
StudentRecord student1, student2;
```

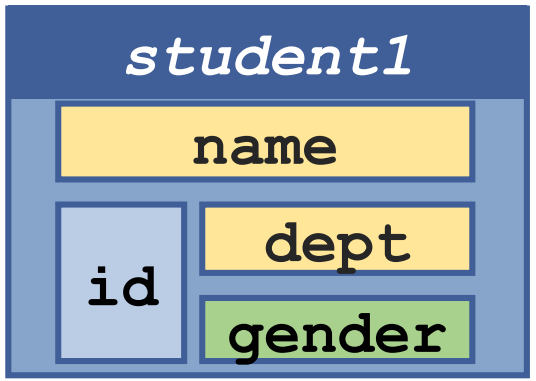
C++ Structs

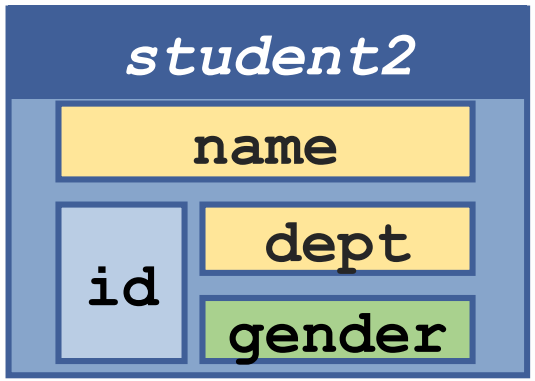
Variable Declaration

Example:

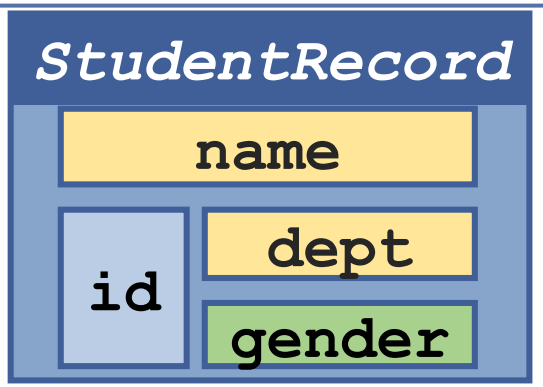
```
StudentRecord student1, student2;
```

- Both variables of type:
  `(struct) StudentRecord`






```
struct StudentRecord {
    char name[9];
    int id;
    char dept[4];
    char gender;
};
```



C++ Structs

Member Access

The Dot (.) Pointer-to-Member Operator:

Used to provide `struct` type member access.

`<struct-variable>.<member_name>;`

Example:

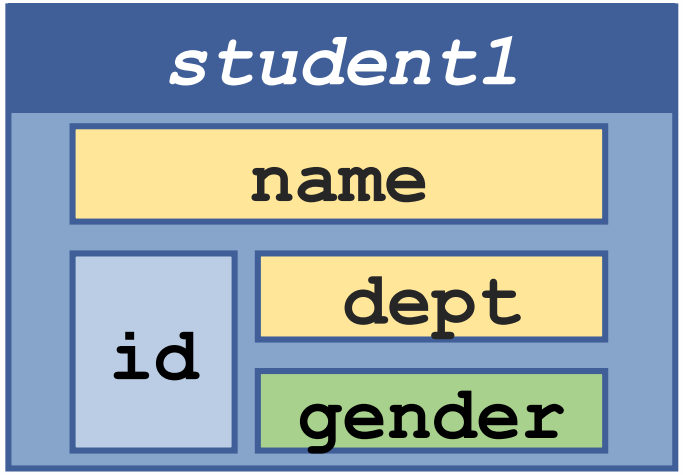
`student1.name`
`student1.id`
`student1.dept`
`student1.gender`

```
struct StudentRecord {
    char name[9];
    int id;
    char dept[4];
    char gender;
};
```

Member Access

Example:

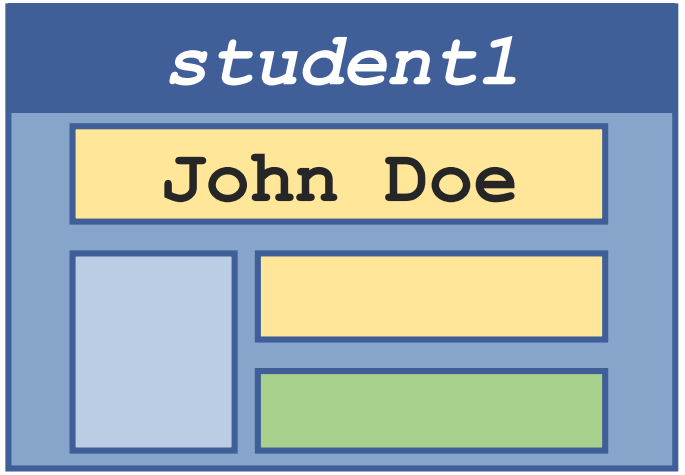
```
strcpy(student1.name, "John Doe");
student1.id = 123;
strcpy(student1.dept, "CSE");
student1.gender = 'M';
cout << "The student is ";
switch (student1.gender){
    case 'F': cout << "Ms. "; break;
    case 'M': cout << "Mr. "; break;
}
cout << student1.name << endl;
```



Member Access

Example:

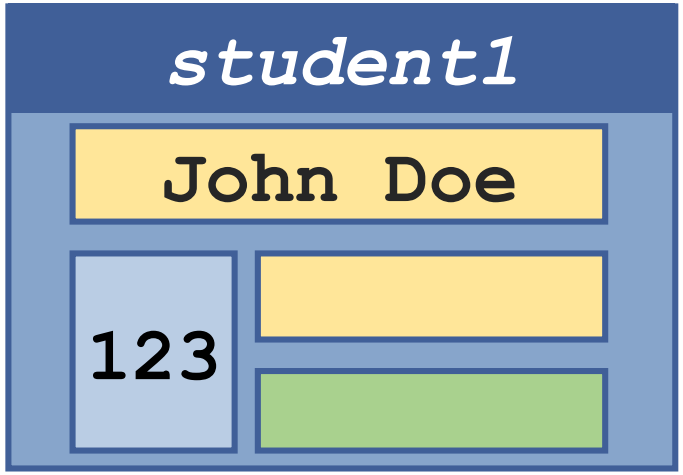
```
strcpy(student1.name, "John Doe");
student1.id = 123;
strcpy(student1.dept, "CSE");
student1.gender = 'M';
cout << "The student is ";
switch (student1.gender){
   case 'F': cout << "Ms. "; break;
   case 'M': cout << "Mr. "; break;
}
cout << student1.name << endl;
```



Member Access

Example:

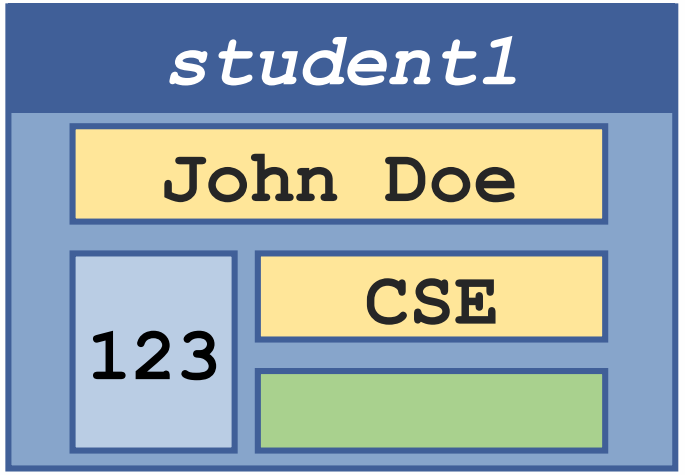
```
strcpy(student1.name, "John Doe");
student1.id = 123;
strcpy(student1.dept, "CSE");
student1.gender = 'M';
cout << "The student is ";
switch (student1.gender){
   case 'F': cout << "Ms. "; break;
   case 'M': cout << "Mr. "; break;
}
cout << student1.name << endl;
```



Member Access

Example:

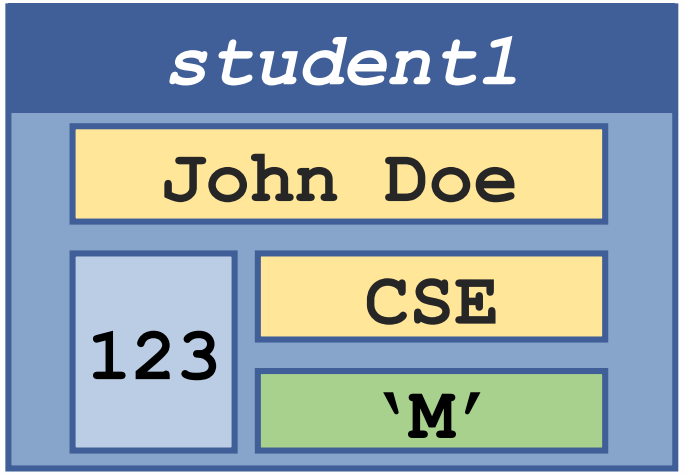
```
strcpy(student1.name, "John Doe");
student1.id = 123;
strcpy(student1.dept, "CSE");
student1.gender = 'M';
cout << "The student is ";
switch (student1.gender){
   case 'F': cout << "Ms. "; break;
   case 'M': cout << "Mr. "; break;
}
cout << student1.name << endl;
```



Member Access

Example:

```
strcpy(student1.name, "John Doe");
student1.id = 123;
strcpy(student1.dept, "CSE");
student1.gender = 'M';
cout << "The student is ";
switch (student1.gender){
   case 'F': cout << "Ms. "; break;
   case 'M': cout << "Mr. "; break;
}
cout << student1.name << endl;
```



Member Access

Example:

```
strcpy(student1.name, "John Doe");
student1.id = 123;
strcpy(student1.dept, "CSE");
student1.gender = 'M';
cout << "The student is ";
switch (student1.gender){
    case 'F': cout << "Ms. "; break;
    case 'M': cout << "Mr. "; break;
}
cout << student1.name << endl;
```

Output:

```
The student is Mr. John Doe
```

C++ Structs

Initialization

```
StudentRecord student1 = {"John Doe", 123, "CSE", 'M'};
```

- Heavily depends on `struct` type definition. Compromised maintainability.
- Might break (type mismatch).
- Might work but mess up (wrong value assignment).

```
struct StudentRecord{
    char name[9];
    int id;
    char dept[4];
    char gender;
};
```

C99 Inline initialization list with designators (*NOT supported in C++*):

```
StudentRecord student1 = {.name="John Doe",.Id=123,.dept="CSE",.gender='M'};
StudentRecord student1 = {.name="John Doe",123,"CSE",'M'};
StudentRecord student1 = {.dept="CSE",'M',.name="John Doe",.id=123};
```

(*Note*: C++20 reintroduces the concept, under stricter rules …)

C++ Structs

Initialization

```
StudentRecord student1 = {"John Doe", 123, "CSE", 'M'};
```

- Heavily depends on `struct` type definition. Compromised maintainability.
- Might break (type mismatch).
- Might work but mess up (wrong value assignment altogether or wrong semantics).

```
struct StudentRecord{
    char name[9];
    int id;
    char dept[4];
    char gender;
};
```

- Too reliant on many "semantics"…

Potential Problems due to change in `struct` declaration!

- C++ static array type-checking by their size yields compile-time

```
error: initializer-string for array of chars is too long [-fpermissive]
```

- In C allocated array memory bounds may be overwritten …

```
struct StudentRecord{
    char name[5];
    int id;
    char dept[3];
    char gender;
};
```

Assignment

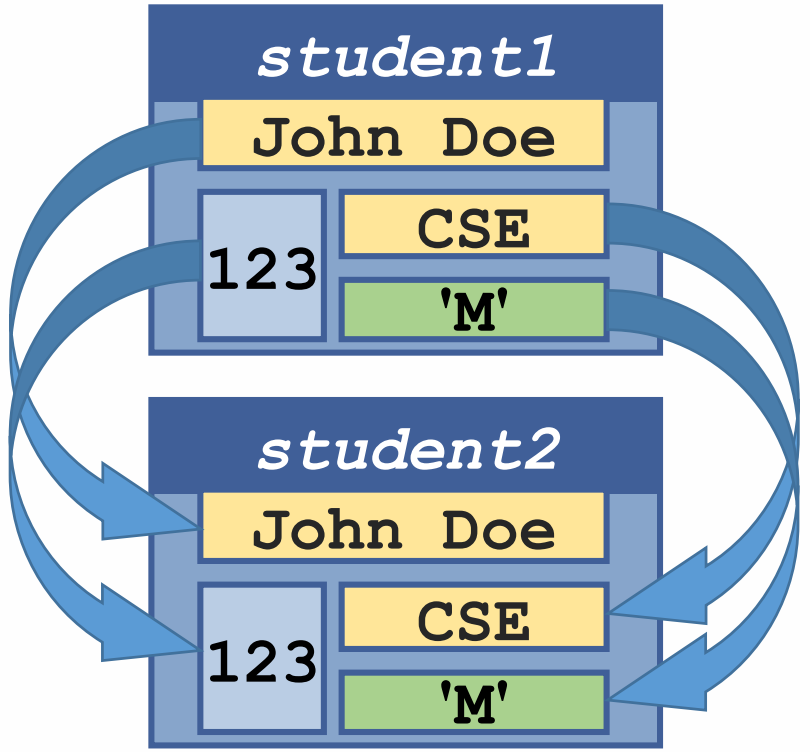
The values contained in one `struct` type variable can be assigned to another variable of the same `struct` type.

- This involves *Data Copy* operations.

Example:

```
strcpy(student1.name, "John Doe");
student1.id = 123;
strcpy(student1.dept, "CSE");
student1.gender = 'M';
```

```
StudentRecord student2 = student1;
```



Nested Structures

A `struct` type can be a member of another `struct`.

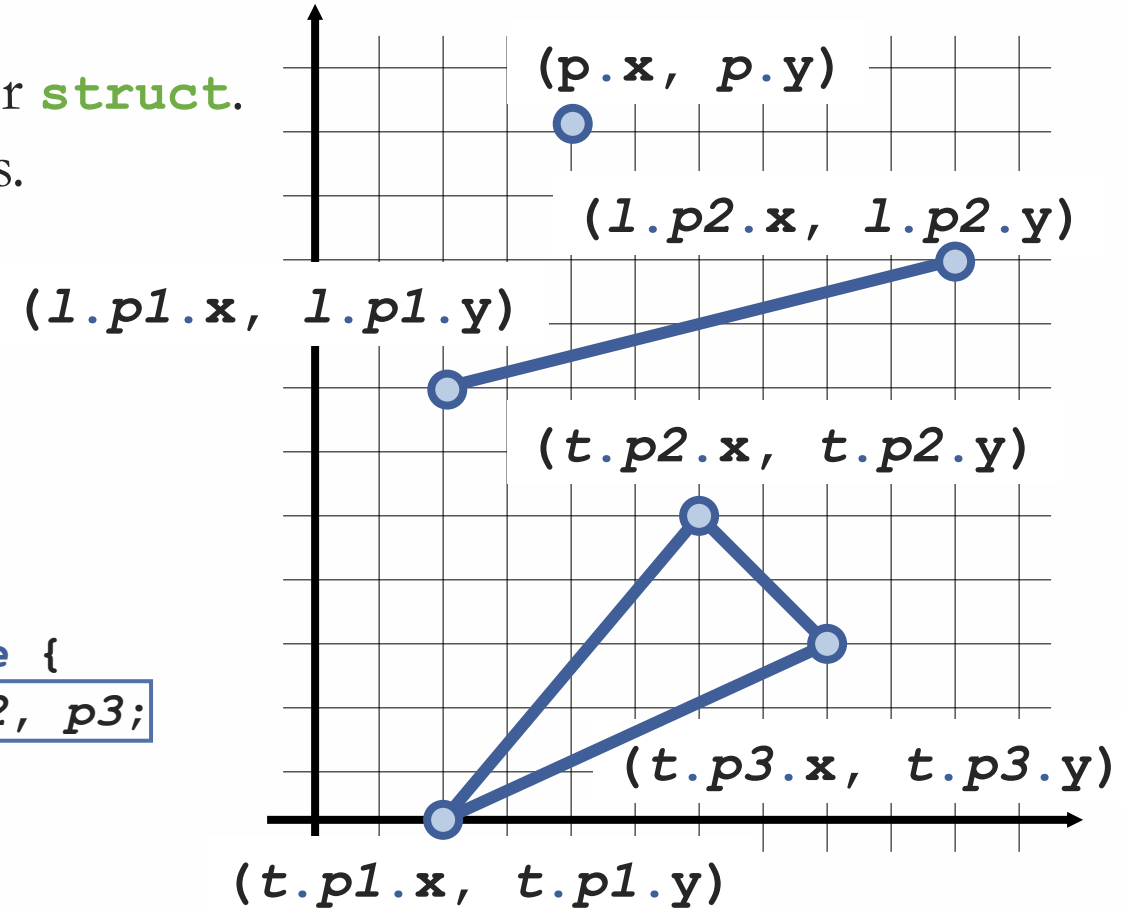
- Program design w.r.t. inherent attributes.

Example:

```
struct Point {
    double x, y;
};
point p;
```

```
struct Line {
    Point p1, p2;
};
Line l;
```

```
struct Triangle {
    Point p1, p2, p3;
};
Triangle t;
```



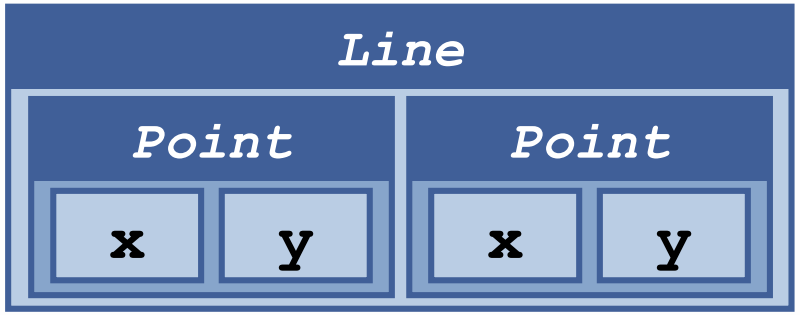
Nested Structures

A `struct` type can be a member of another `struct`.

- Program design w.r.t. inherent attributes.

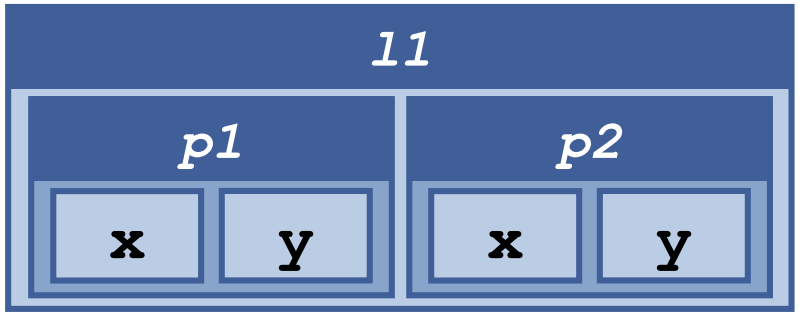
Example:

```
struct Line {
   Point p1, p2;
};
```



Type Definition

```
Line l1;
```



Variable Creation

Nested Structures

A `struct` type can be a member of another `struct`.

- NOTE: Cannot have recursion here !

Example:

```
struct StudentRecord {
   char name[15];
   int id;
   char dept[5];
   char gender;
   StudentRecord emergContact;
};
```

NO

Pointer of self-referencing type is allowed

```
struct StudentRecord {
   char name[15];
   int id;
   char dept[5];
   char gender;
   StudentRecord * emergContact;
};
```

YES !

Nested Structures

A `struct` type can be a member of another `struct`.

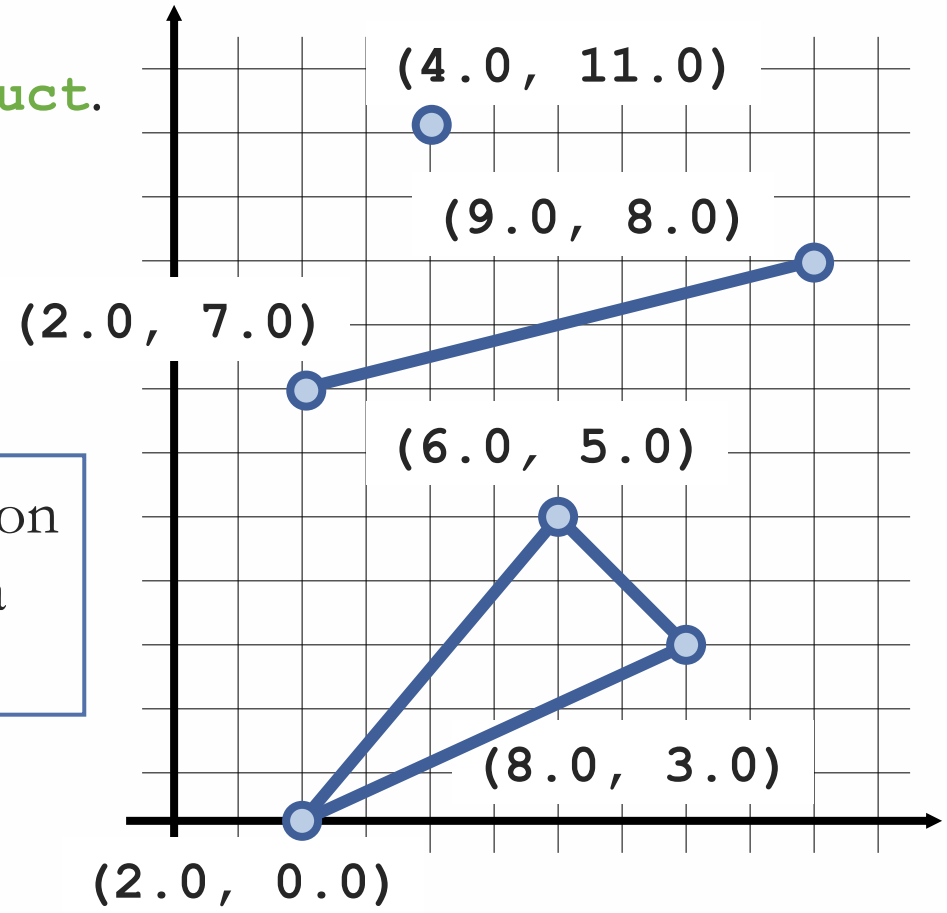
- Program design w.r.t. inherent attributes.

Example:

```
Point p;
Line l;
Triangle t;

p.x = 4.0;
p.y = 11.0;
```

Literals-based Initialization of every primitive data member



Nested Structures

A `struct` type can be a member of another `struct`.

- Program design w.r.t. inherent attributes.

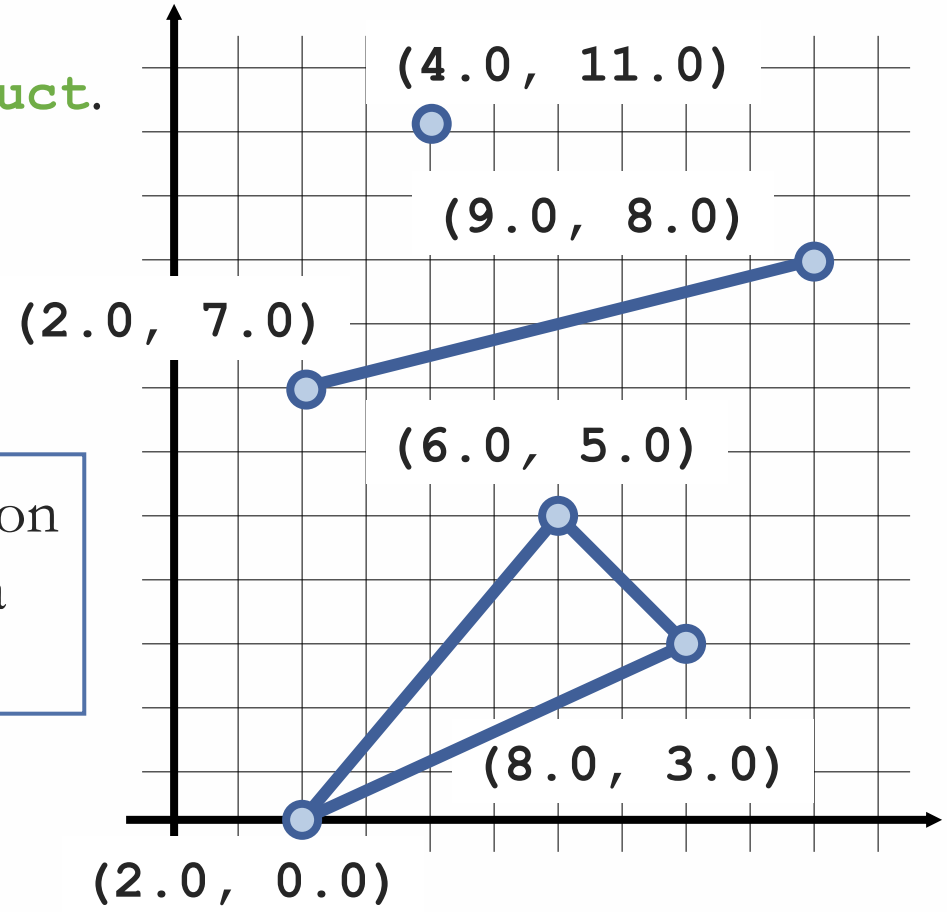
Example:

```
Point p;
Line l;
Triangle t;

l.p1.x = 2.0;
l.p1.y = 7.0;

l.p2.x = 9.0;
l.p2.y = 8.0;
```

Literals-based Initialization of every primitive data member



Nested Structures

A `struct` type can be a member of another `struct`.

- Program design w.r.t. inherent attributes.

Example:

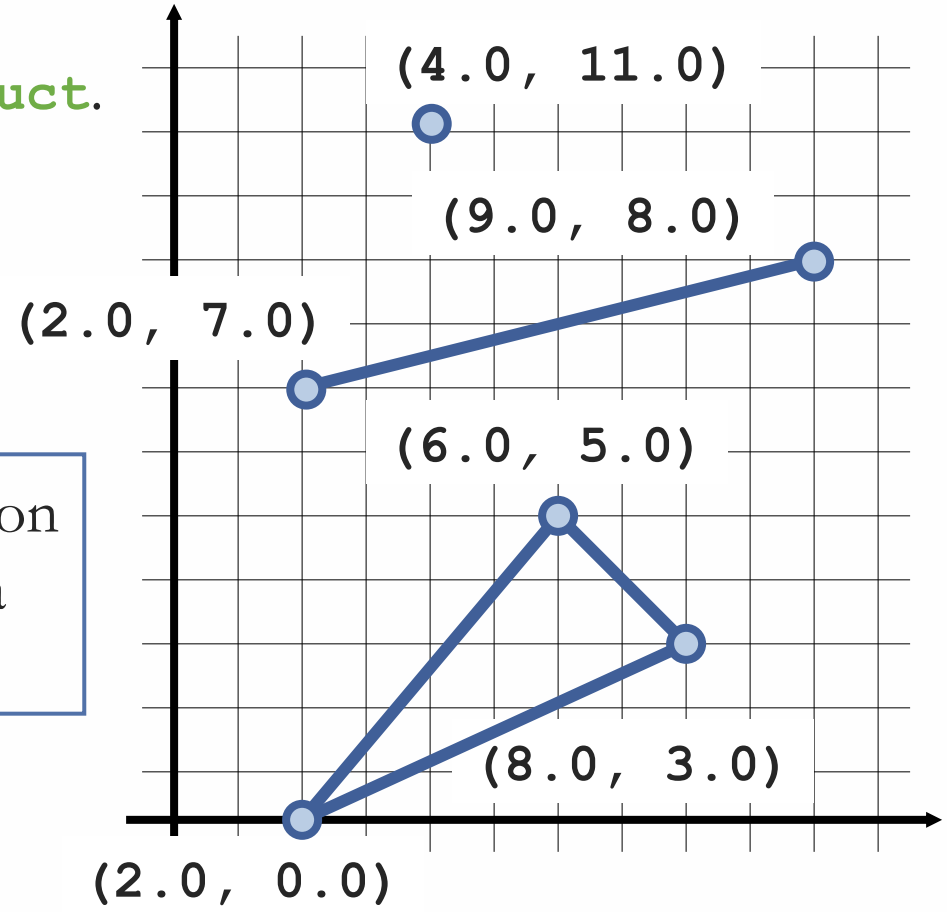
```
Point p;
Line l;
Triangle t;

t.p1.x = 6.0;
t.p1.y = 5.0;

t.p2.x = 8.0;
t.p2.y = 3.0;

t.p3.x = 2.0;
t.p3.y = 0.0;
```

Literals-based Initialization of every primitive data member



Nested Structures

A `struct` type can be a member of another `struct`.

- Program design w.r.t. inherent attributes.

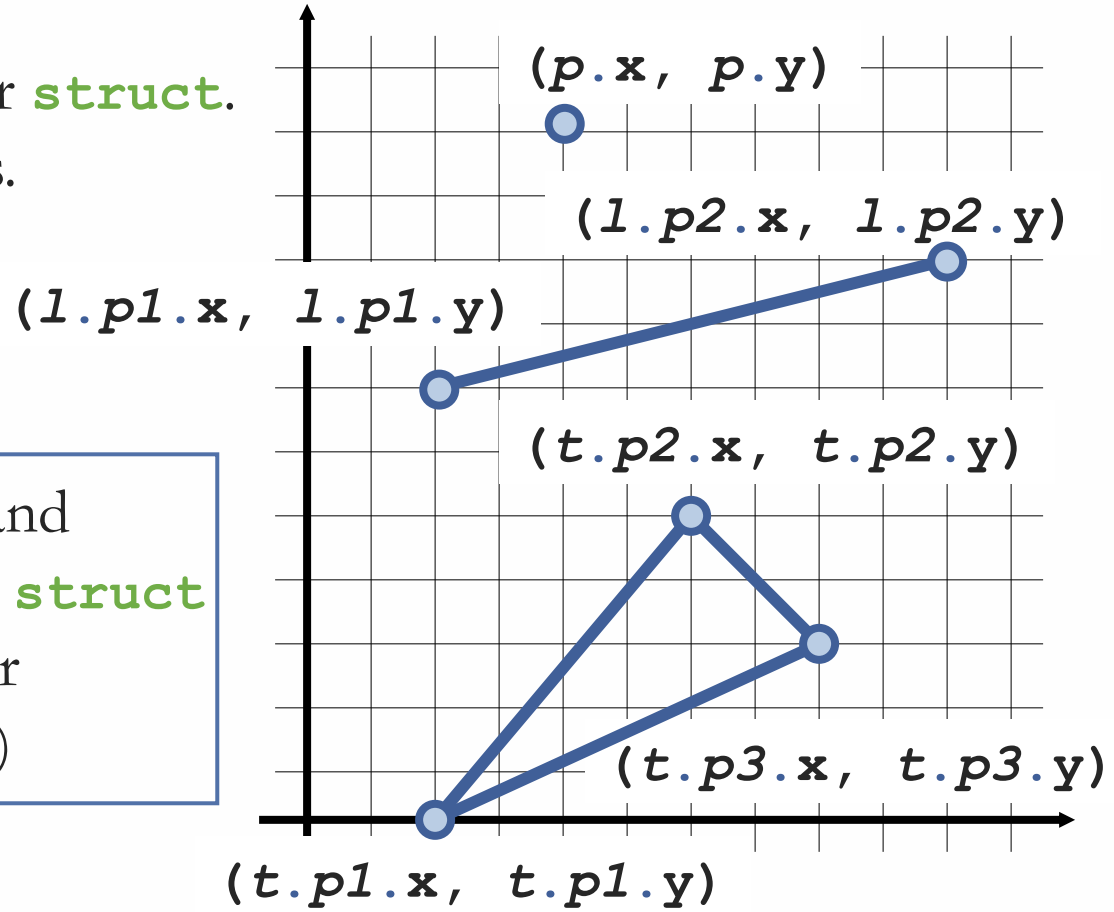
Example:

```
Point p;
Line l;

p.x = 2.00;
p.y = 7.00;
l.p1 = p;

p.x = 9.00;
p.y = 8.00;
l.p2 = p;
```

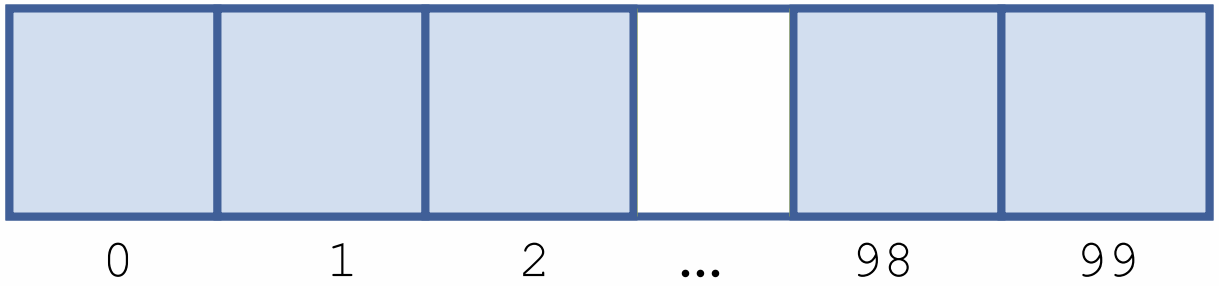
Modification and Assignment to each `struct` data member (Data-Copy)



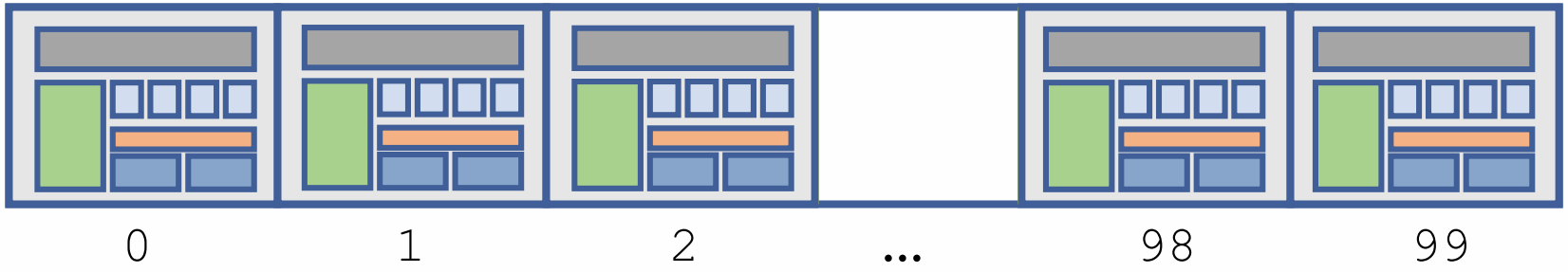
Arrays of Structs

Arrays are *homogeneous* (one data type):

- Regular data type.



- Supported type can be `struct`.



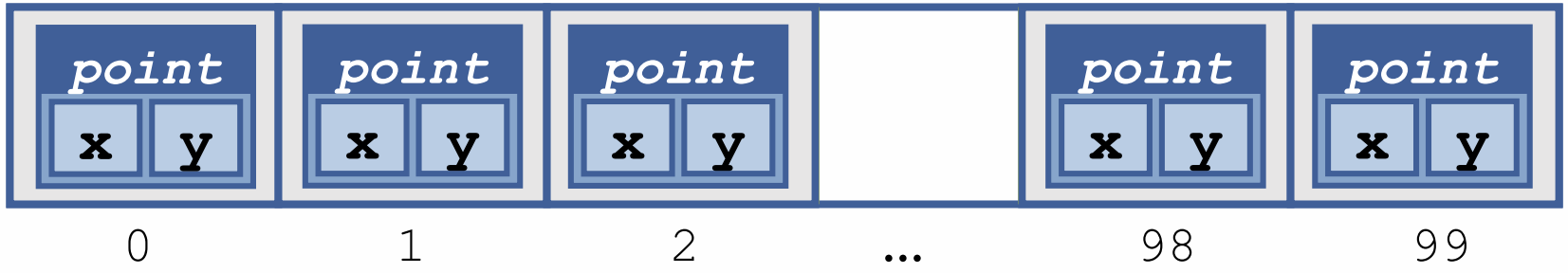
Arrays of Structs

Arrays are *homogeneous* (one data type):

```
struct Point {
   double x, y;
};

Point point_array[100];
```

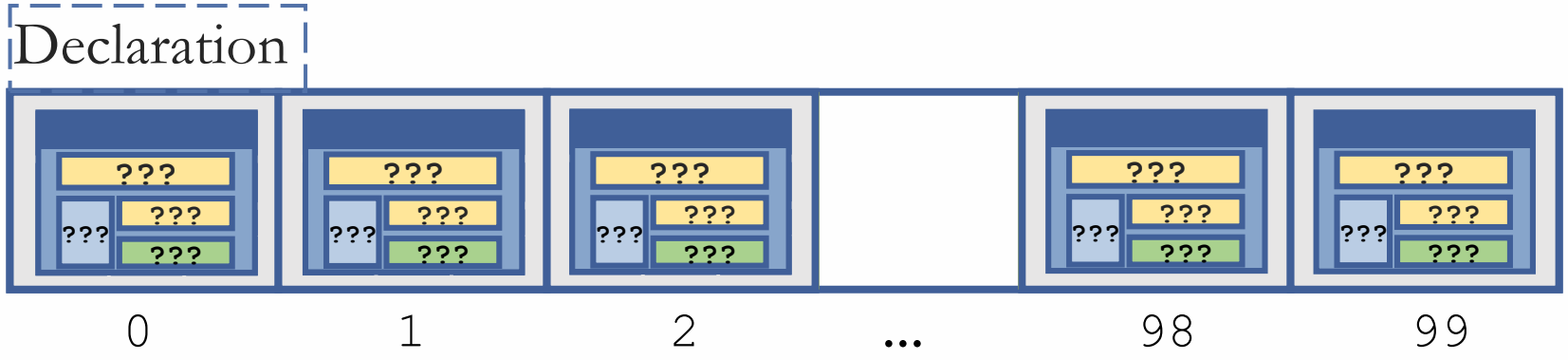
- Supported type can be `struct`.



Arrays of Structs

All aforementioned operations take place as usual:

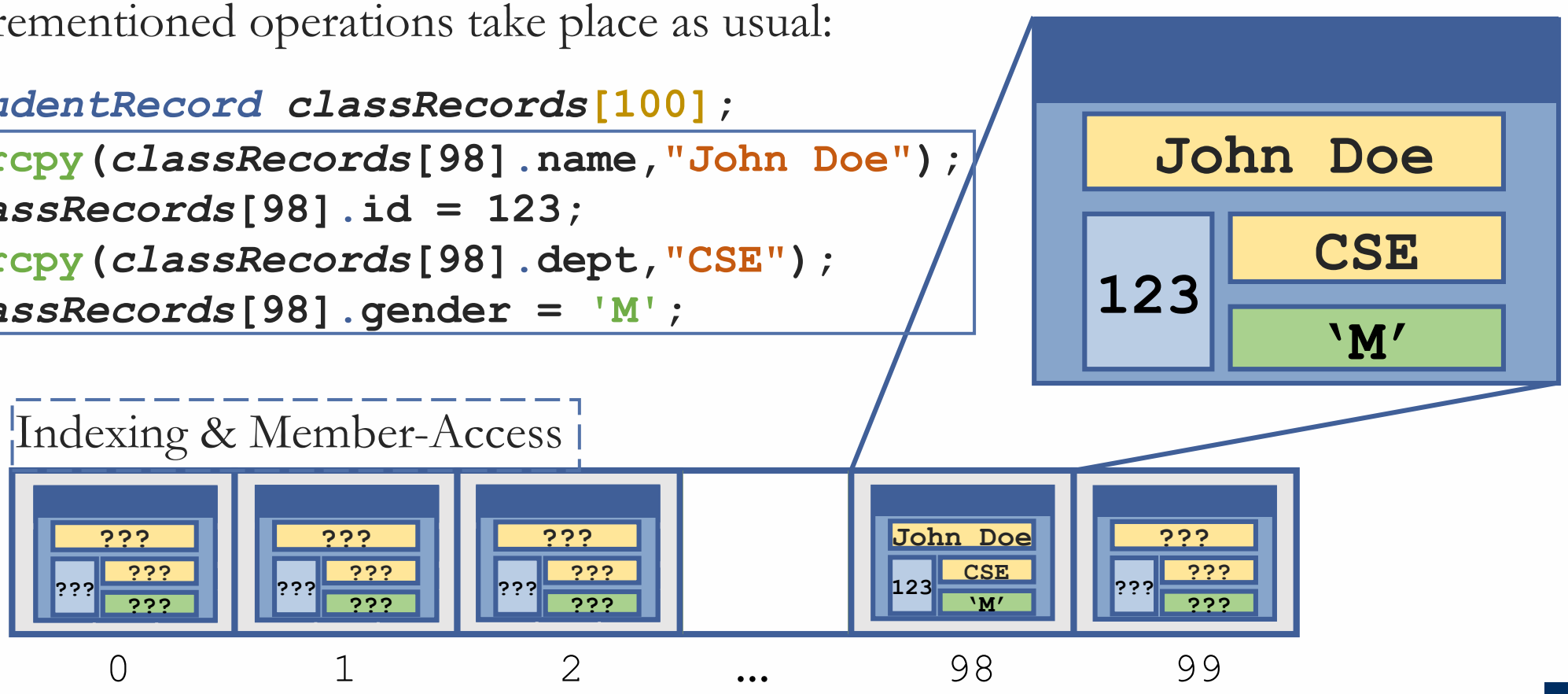
```
StudentRecord classRecords[100];
```



Arrays of Structs

All aforementioned operations take place as usual:

```
StudentRecord classRecords[100];
strcpy(classRecords[98].name,"John Doe");
classRecords[98].id = 123;
strcpy(classRecords[98].dept,"CSE");
classRecords[98].gender = 'M';
```

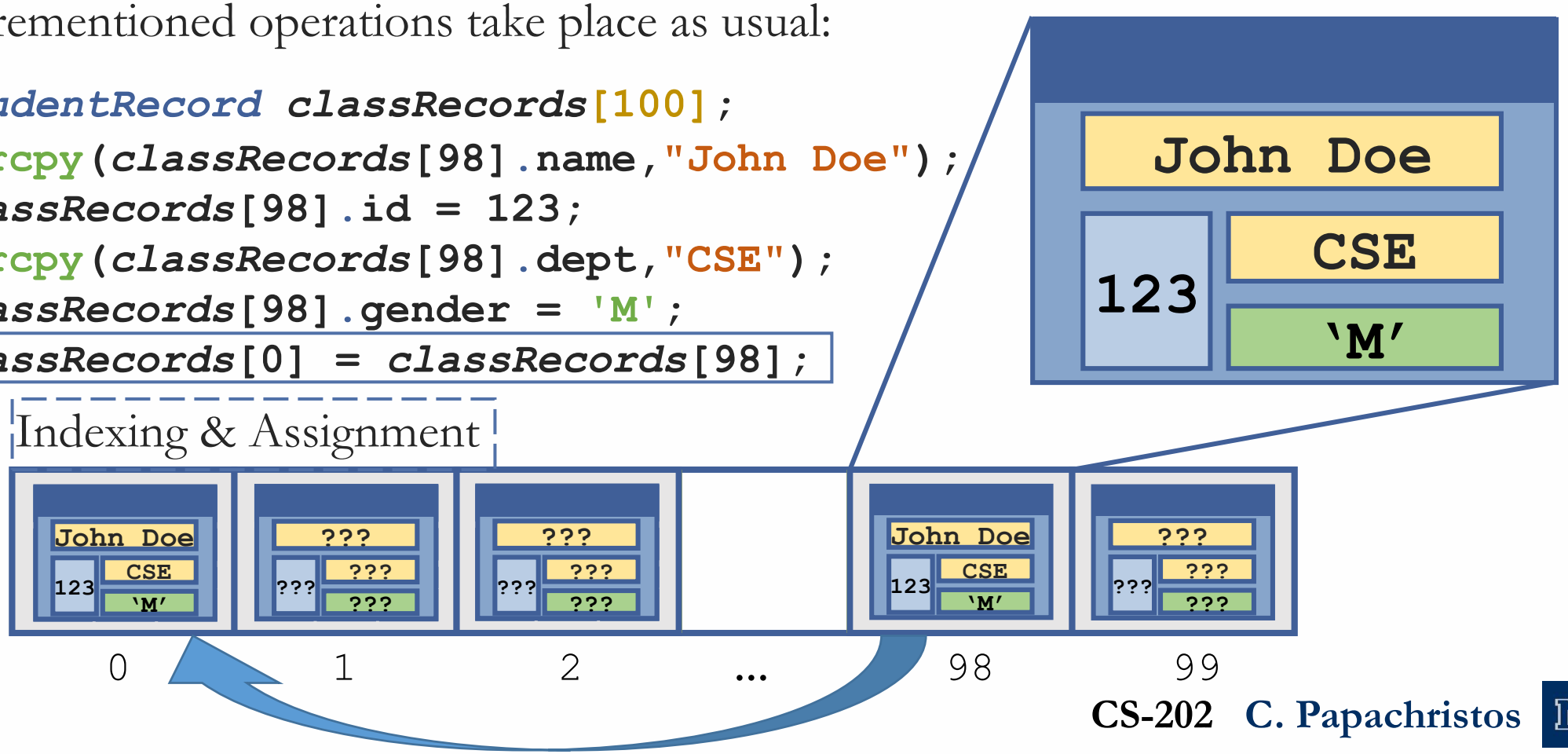


Arrays of Structs

All aforementioned operations take place as usual:

```
StudentRecord classRecords[100];
strcpy(classRecords[98].name,"John Doe");
classRecords[98].id = 123;
strcpy(classRecords[98].dept,"CSE");
classRecords[98].gender = 'M';
classRecords[0] = classRecords[98];
```

Indexing & Assignment

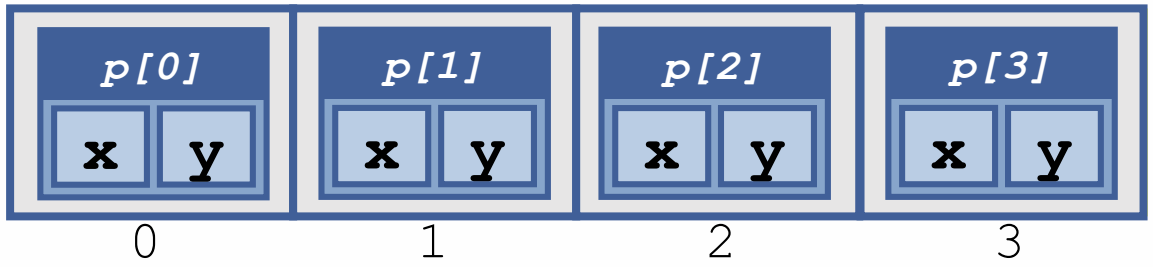


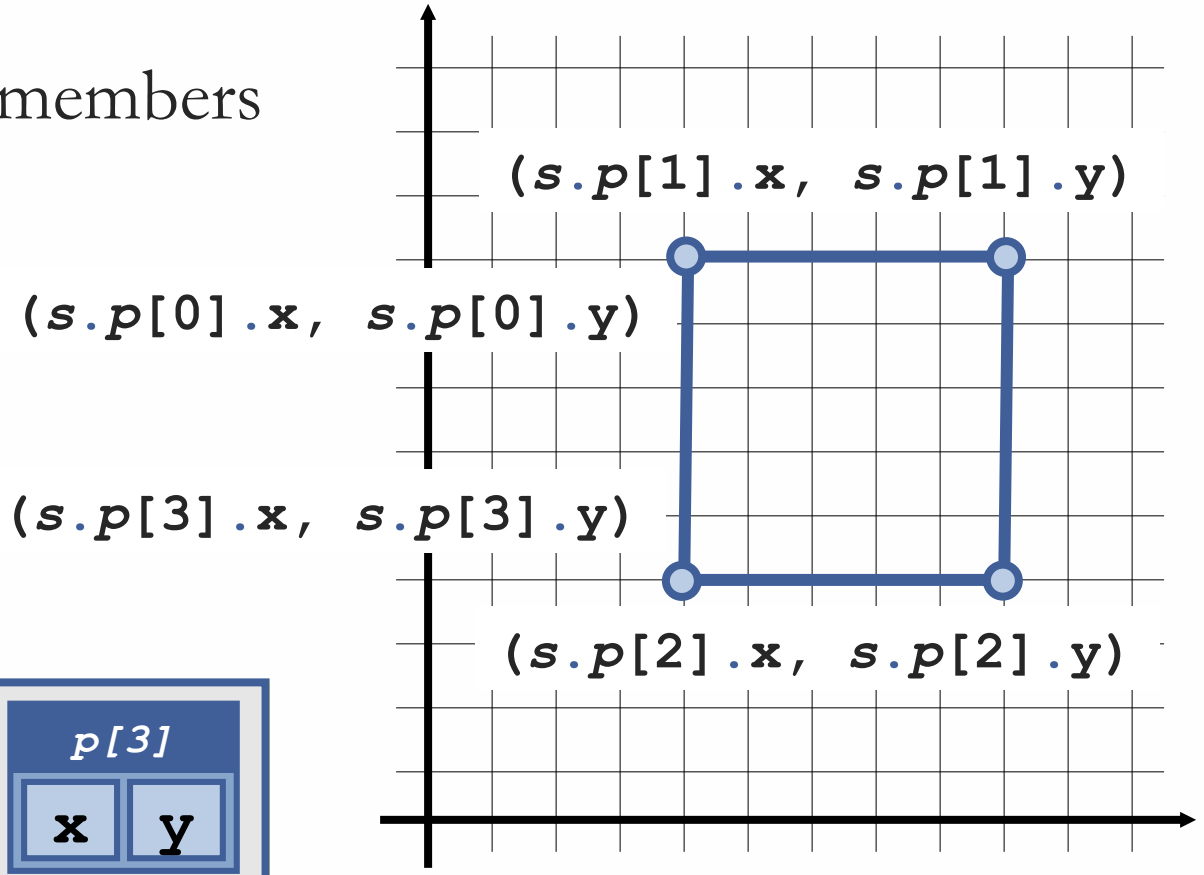
Struct Arrays in Structs

Remember: Arrays can be `struct` members

```
struct Point {
    double x, y;
};

struct Square {
    Point p[4];
};
Square s;
```





Structs and Functions

Supported type for Function Parameters can be **struct**:

```
struct Point{ double x, y; };  // need declaration before any mention
                               // of Point can be made in the program
```

- Pass-By-Value

```
double points_distance(Point p1, Point p2){
   return sqrt((p1.x-p2.x)*(p1.x-p2.x)+(p1.y-p2.y)*(p1.y-p2.y);
}

Point p1, p2;
double p12_distance = points_distance(p1, p2);
```

Data-Copy operation

Structs and Functions

Supported type for Function Parameters can be `struct &`:

```
struct Point{ double x, y; };  // need declaration before any mention
                               // of Point can be made in the program
```

- Pass-by-Reference

```
void shift_point_upright(Point & p){
    p.x += 1.0;
    p.y -= 1.0;
}

Point p;
shift_point_upright(p);
```

No *Data-Copy*

Modifies `struct` members

Structs and Functions

Supported type for Function Parameters can be `struct const &`:

```
struct Point{ double x, y; };  // need declaration before any mention
                               // of Point can be made in the program
```

- Pass-by-`const`-Reference

```
bool is_point_inbounds(const Point & p){
   return p.x>=0 && p.x<NUM_COLS && p.y>=0 && p.x<NUM_ROWS;
}

Point p;
bool p_inbounds = is_point_inbounds(p);
```

No *Data-Copy*

Structs and Functions

Supported type for Function Parameters can be `struct &`:

- Pass-by-Reference

```
void set_point_inbounds(Point & p){
    if (p.x<0) p.x=0; else if (p.x>=NUM_COLS) p.x=NUM_COLS-1;
    if (p.y<0) p.y=0; else if (p.y>=NUM_ROWS) p.y=NUM_ROWS-1;
}
```

- Pass-by-`const`-Reference

```
void set_point_inbounds(const Point & p){
    if (p.x<0) p.x=0; else if (p.x>=NUM_COLS) p.x=NUM_COLS-1;
    if (p.y<0) p.y=0; else if (p.y>=NUM_ROWS) p.y=NUM_ROWS-1;
}
```

`const`-Reference will not allow mutation

Structs and Functions

Supported type for Function Parameters can be `struct *`:

```
struct Point{ double x, y; };  // need declaration before any mention
                               // of Point can be made in the program
```

- Pass-by-Address

```
void shift_point_upright(Point * p){
    (*p).x += 1.0;
    (*p).y -= 1.0;
}

Point p;
Point * p_Pt = &p;
shift_point_upright(p_Pt);
```

Dereferencing to access Value-Pointed-By

Modifies `struct` members

Structs and Functions

Supported type for Function Parameters can be `struct []/*`:

```
struct Point{ double x, y; };  // need declaration before any mention
                               // of Point can be made in the program
```

- `struct` Array can be Passed-by-Address

```
void shift_points_downleft(Point * p_arr, int sz){
    for (int i=0; i<sz; ++i){
        p_arr[i].x -= 1.0;
        p_arr[i].y += 1.0;
    }
}

Point points_array[100];
shift_points_downleft(points_array, 100);
```

Parameter similarly as:
`Point p_arr[]`

Modifies `struct` members

Structs and Functions

Supported **return** type for Functions can be **struct**:

```
struct Point{ double x, y; };  // need declaration before any mention
                               // of Point can be made in the program
```

- **return** type can be **struct** Value

```
Point mirror_point(const Point & p_in){
    Point p_out;
    p_out.x = -p_in.x;
    p_out.y = -p_in.y;
    return p_out;
}

Point p1;
Point p1_mirrored = mirror_point(p1);
```

Local variable
Point p_out

Lifetime?

Data-Copy (assignment)
Point p1_mirrored =

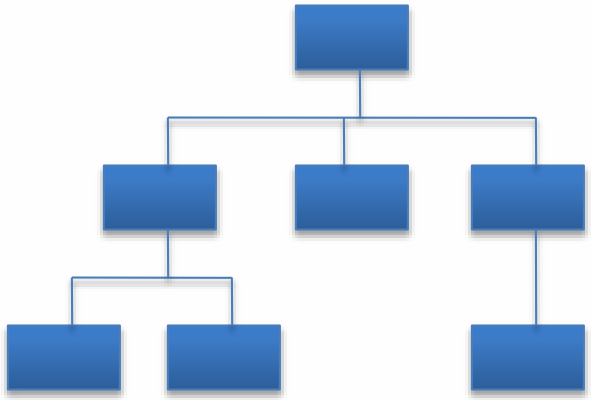
Remember: Procedural *vs* Object-Oriented

Procedural

Focused on the question: "What should the program do next?" Structure program by:

- Splitting into sets of tasks and subtasks.
- Make functions for tasks.
- Perform them in sequence (computer).

Large amount of data and/or tasks makes projects/programs unmaintainable.



A hierarchy of functions

Object-Oriented (OO)

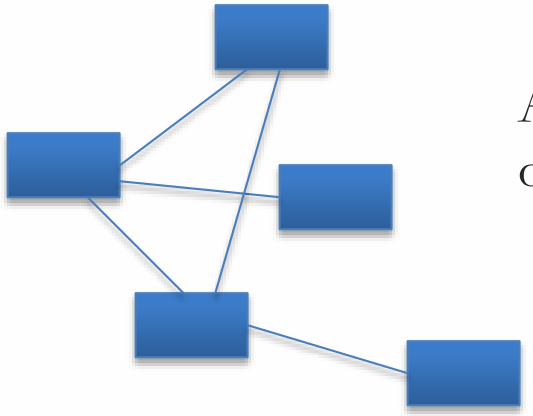
Package-up self-sufficient modular pieces of code.

The world is made up of interacting objects.

Pack away details into boxes (objects) keep them in mind in their abstract form.

Focus on (numerous) interactions.

- Encapsulation
- Inheritance
- Polymorphism



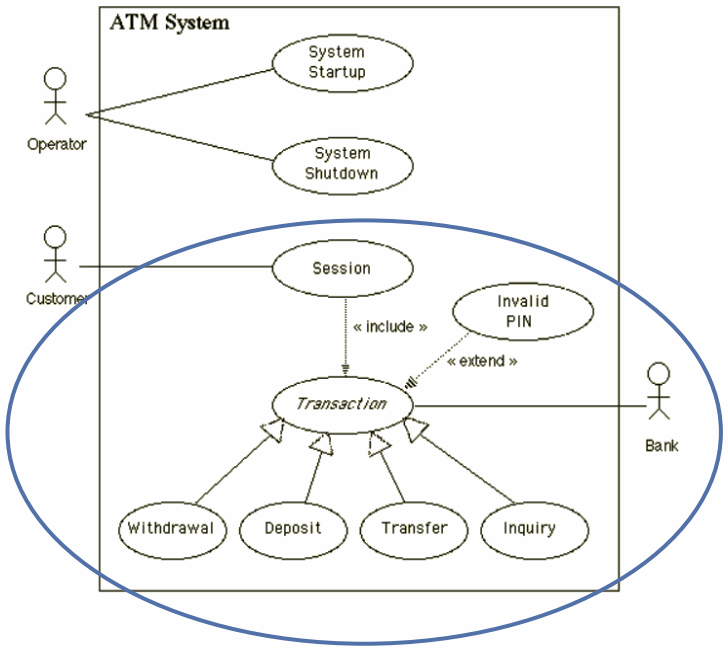
A collection of Objects

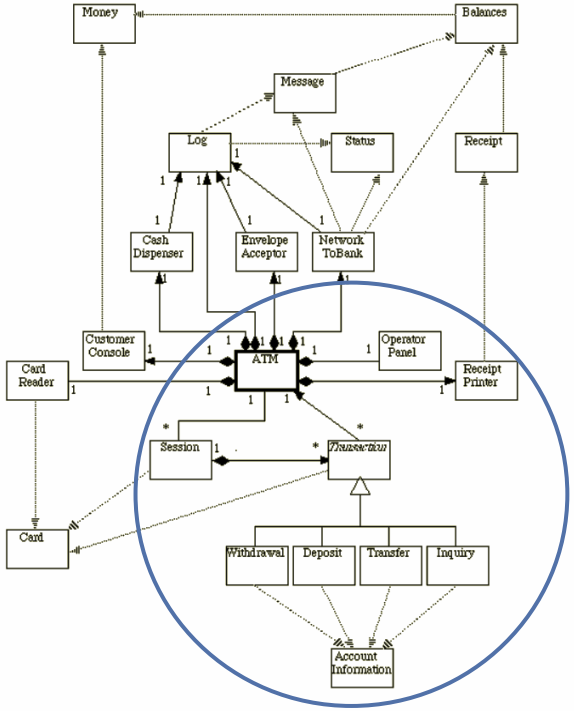
Remember: Procedural *vs* Object-Oriented

The ATM Machine paradigm

```
struct Date {
    int day;
    int month;
    int year;
};

struct BankAccount {
    char name[15];
    int acountNo[10];
    double balance;
    Date birthday;
};
```

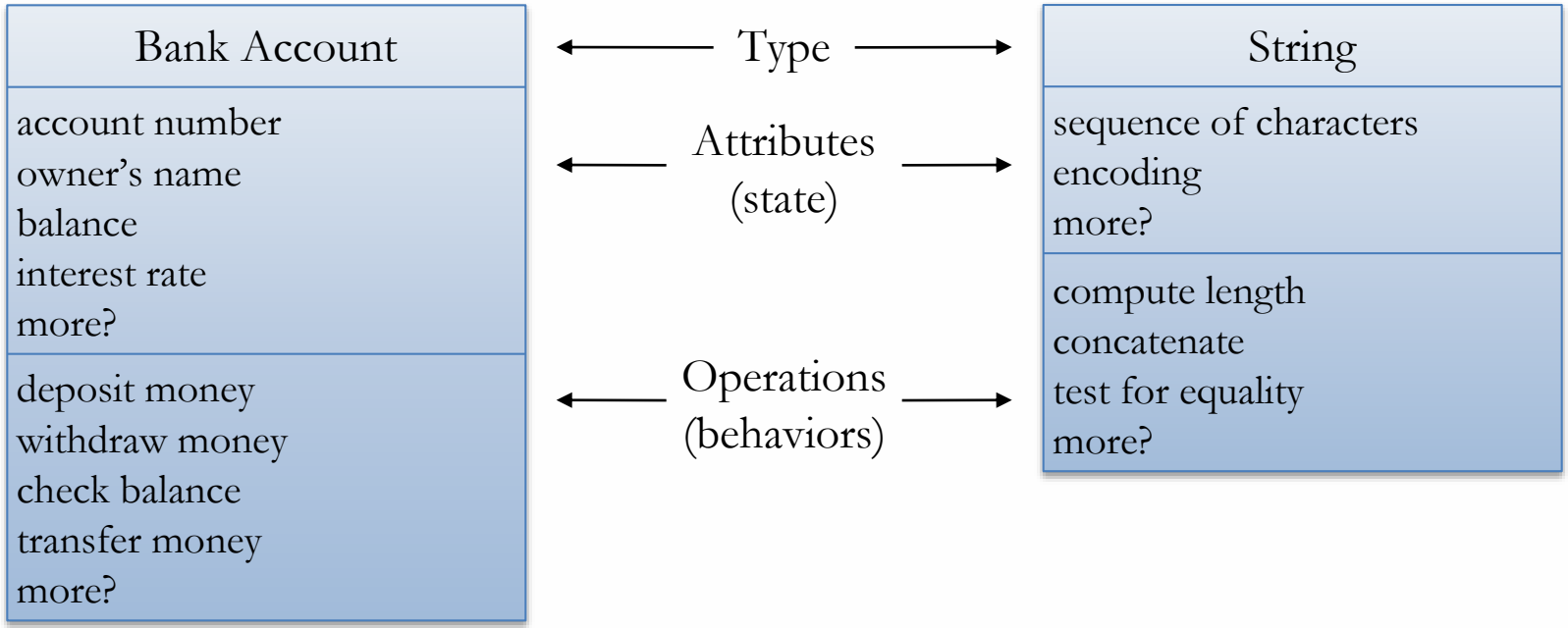




Remember: Classes

Class

C++ Classes are very similar to C Structs, in that they both include user-defined sets of data items, which collectively describe some entity such as a Student, a Book, an Airplane, or a data construct such as a String, a ComplexNumber, etc…



C++ Structs

Structs in C++

`structs` encapsulate related data.

- Member variables maintain each object's state.
- All member "parts" *by default* are `public`ly accessible.

(later: Class members *by default* are `private` – internally accessible for a specific Object from own methods, i.e. functions)

When to use a `struct` (for now) :

- For things that are mostly data-oriented.

C++ Structs

Structs in C++

structs encapsulate related data.

- Member variables maintain each object's state.
- All member "parts" *by default* are **public**ly accessible.

(later: Class members *by default* are **private** – internally accessible for a specific Object from own methods, i.e. functions)

When to use a **struct** (for now) :

- For things that are mostly data-oriented.
- Are there data-only limitations?
  Data sanity checking might be necessary!

Not a leap year!

```
struct Date {
    int month;
    int day;
    int year;
};
Date bDay{2, 29, 2015};
```

C++ Structs

Structs in C++

`structs` can have methods (i.e. functions).

- Actually in C++ `struct` and Class are very similar.
- Default access level (`public` *vs* `private`) is the difference of significance from what we know so far.

`structs` can have: (– *Note*: like Classes do)

- Member variables
- Methods (i.e. Functions)
- Constructors, Destructors, etc. (more on these later)
- `public`, `private`, and `protected` attributes (more on these later)
- `virtual` functions (more on these later)

Struct Methods / Constructors / etc. in C++

```
struct Date {
    int month, day, year;

    Date(int m_month, int m_day, int m_year) :
      month(m_month), day(m_day), year(m_year){
        if (month<=0) month = 1;
        if (day<=0) day = 1;
        if (year<1985) year = 1985;
        fixLeapDate();
    }
    void fixLeapDate(){
        if (year ... && month ... && day ...){
           day = ...;
        }
    }
};
```

Not a leap year!

```
Date bday(2, 29, 2015);
```

Struct Methods / Constructors / etc. in C++

```
struct Date {
    int month, day, year;
    Date(int m_month, int m_day, int m_year) :
      month(m_month), day(m_day), year(m_year){
        if (month<=0) month = 1;
        if (day<=0) day = 1;
        if (year<1985) year = 1985;
        fixLeapDate();
    }
    void fixLeapDate(){
        if (year ... && month ... && day ...){
           day = ...;
        }
    }
};
```

Not a leap year!

```
Date bday(2, 29, 2015);
```

➢ Constructor call !

Struct Methods / Constructors / etc. in C++

```
struct Date {
    int month, day, year;
    Date(int m_month, int m_day, int m_year) :
      month(m_month), day(m_day), year(m_year){
        if (month<=0) month = 1;
        if (day<=0) day = 1;
        if (year<1985) year = 1985;
        fixLeapDate();
    }
    void fixLeapDate(){
        if (year ... && month ... && day ...){
           day = ...;
        }
    }
};
```

Not a leap year!

```
Date bday(2, 29, 2015);
```

- Constructor call !

- Perform series of checks.
- Calls internal method on itself.

Struct Methods / Constructors / etc. in C++

Calling Member Methods (i.e. Member Functions)

- Member Access Operator (.) - Just like accessing a member.

```
struct Date {
   int month, day, year;

   Date(int m_month, int m_day, int m_year) :
     month(m_month), day(m_day), year(m_year){ ... }
   bool isLeapYear(){ ... return ... ; }
};

Date bday(2, 29, 2015);
bool leap_year = bDay.isLeapYear();
```

CS-202

Time for Questions !