CS-202

C++ Structs

C. Papachristos
Autonomous Robots Lab
University of Nevada, Reno

Course Week

Course , Projects , Labs:

Monday	Tuesday	Wednesday	Thursday	Friday
			Lab (9:00-12:50)	
	CLASS		CLASS	
PASS Session	PASS Session	**Project DEADLINE**	NEW Project	

Your 2nd Project will be announced today Thursday 9/7.

1st Project Deadline was this Wednesday 9/6.

- 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*:

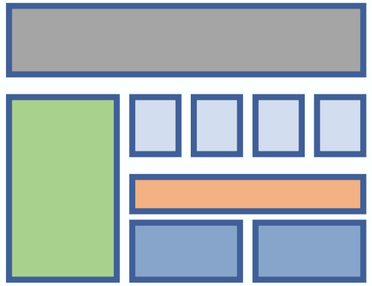
- It can contain only data of the same type.

C++ Structs

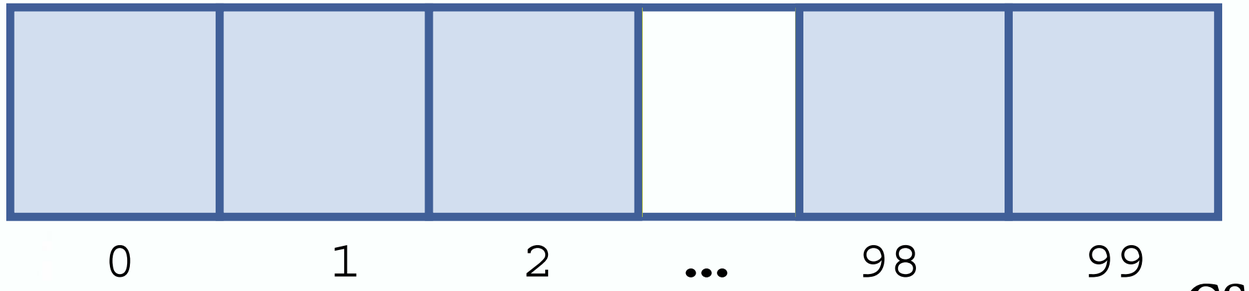
Description

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
- `struct`

C++ Structs

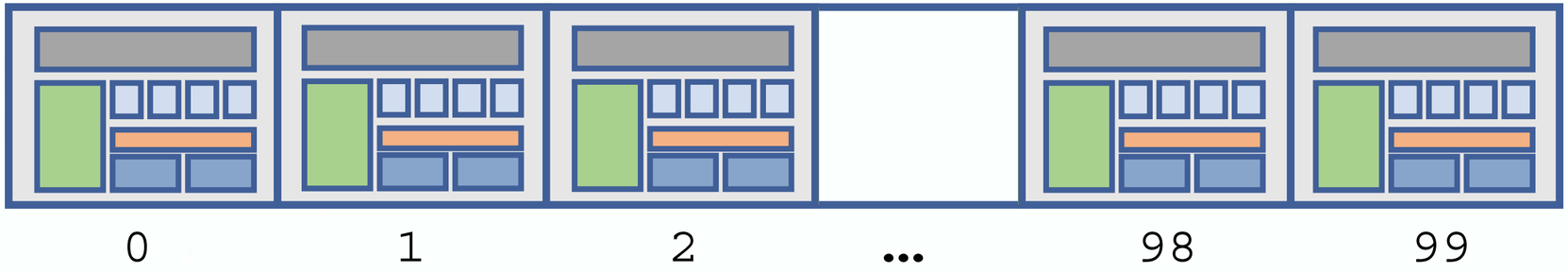
Members

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 CGA;
};
```

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;
};
```

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

```
struct StudentRecord{
   char Name[9];
   int Id;
   char Dept[4];
   char Gender;
};
```

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>;
    ...
} ;
```

Declaration of a new variable of that type:

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

Example:

```
StudentRecord Student1, Student2;
```

```
struct StudentRecord{
    char Name[9];
    int Id;
    char Dept[4];
    char Gender;
};
```

C++ Structs

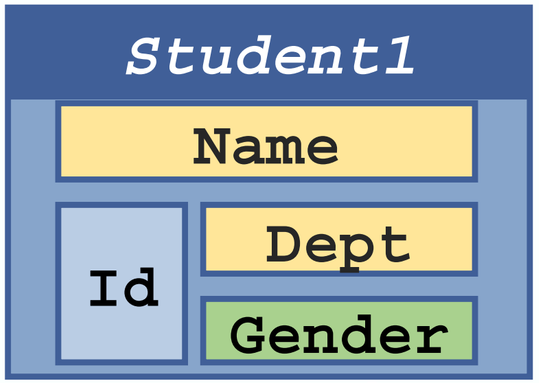
Variable Declaration

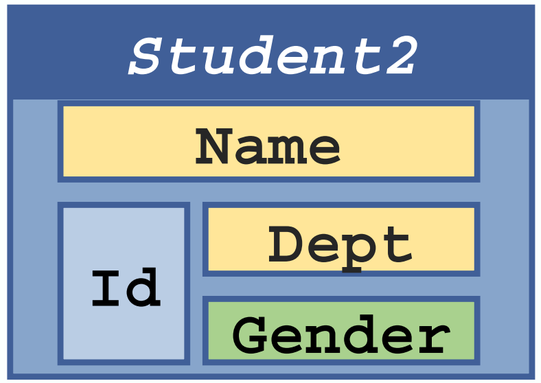
Example:

```
StudentRecord Student1, Student2;
```

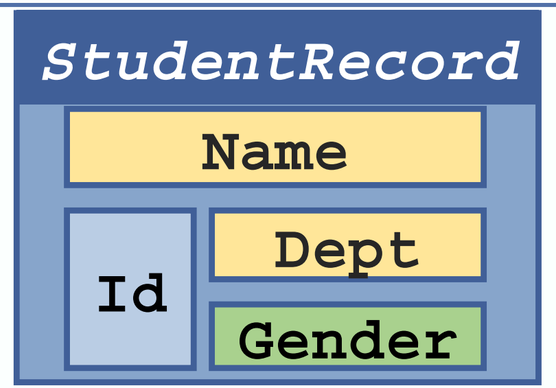
- Both variables of type:

```
(struct)StudentRecord
```






```
struct StudentRecord{
    char Name[9];
    int Id;
    char Dept[4];
    char Gender;
};
```



C++ Structs

Member Access

The Dot (.) Operator:

Used to provide `struct` type member access.

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

Example:

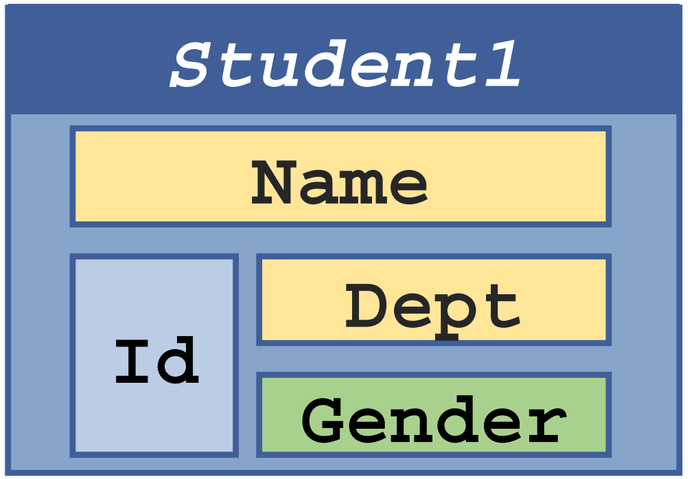
```
Student1.Name
Student1.Id
Student1.Dept
Student1.Gender
```

```
struct StudentRecord{
    char Name[9];
    int Id;
    char Dept[4];
    char Gender;
};
```

Member Access

Example:

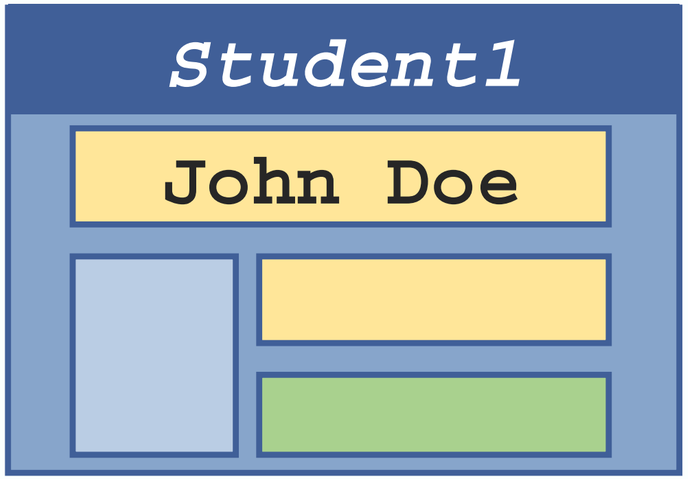
```
strcpy(Student1.Name, "John Doe");
Student1.Id = 123;
strcpy(Student1.Dept, "CSE");
Student1.gender = 'M';
cout << "The student is ";
switch (Student1.gender){
   case 'F': cout << "Ms. "; break;
   case 'M': cout << "Mr. "; break;
}
cout << Student1.Name << endl;
```



Member Access

Example:

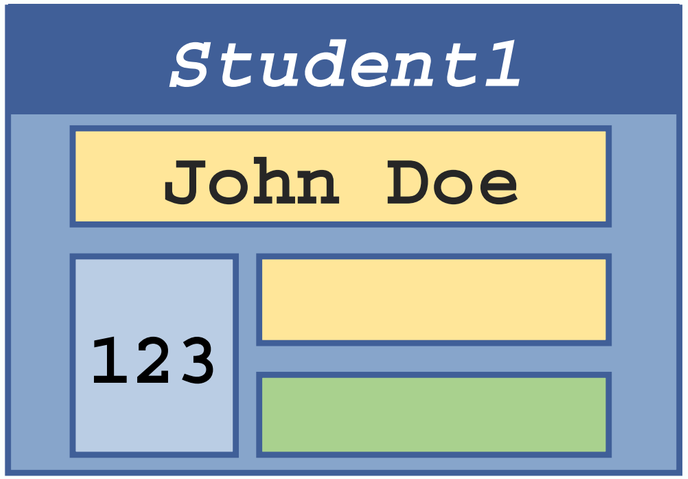
```
strcpy(Student1.Name, "John Doe");
Student1.Id = 123;
strcpy(Student1.Dept, "CSE");
Student1.gender = 'M';
cout << "The student is ";
switch (Student1.gender){
    case 'F': cout << "Ms. "; break;
    case 'M': cout << "Mr. "; break;
}
cout << Student1.Name << endl;
```



Member Access

Example:

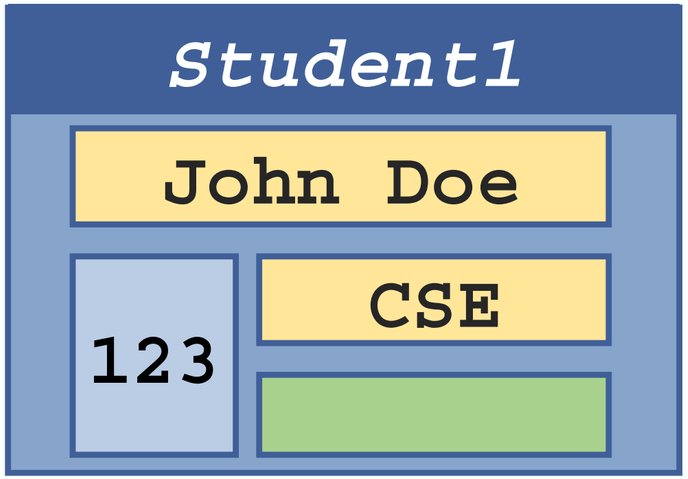
```
strcpy(Student1.Name, "John Doe");
Student1.Id = 123;
strcpy(Student1.Dept, "CSE");
Student1.gender = 'M';
cout << "The student is ";
switch (Student1.gender){
   case 'F': cout << "Ms. "; break;
   case 'M': cout << "Mr. "; break;
}
cout << Student1.Name << endl;
```



Member Access

Example:

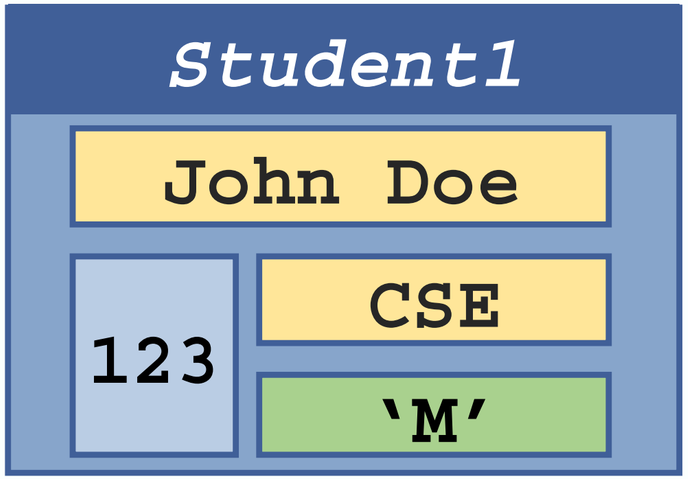
```
strcpy(Student1.Name, "John Doe");
Student1.Id = 123;
strcpy(Student1.Dept, "CSE");
Student1.gender = 'M';
cout << "The student is ";
switch (Student1.gender){
   case 'F': cout << "Ms. "; break;
   case 'M': cout << "Mr. "; break;
}
cout << Student1.Name << endl;
```



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};
```

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;
};
```

- Too reliant on many "semantics"…

```
struct StudentRecord{
    char Name[15];
    int Id;
    char Dept[5];
    char Gender;
};
```

Won't work.

Assignment

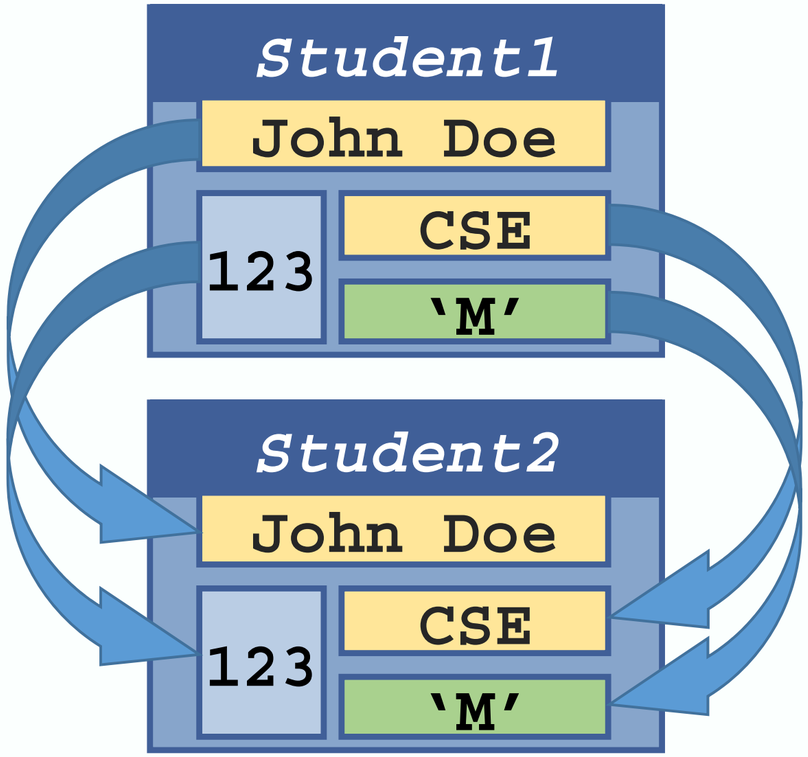
The values contained in one `struct` type variable can be assigned to another variable of the same `struct` type.

- 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`.

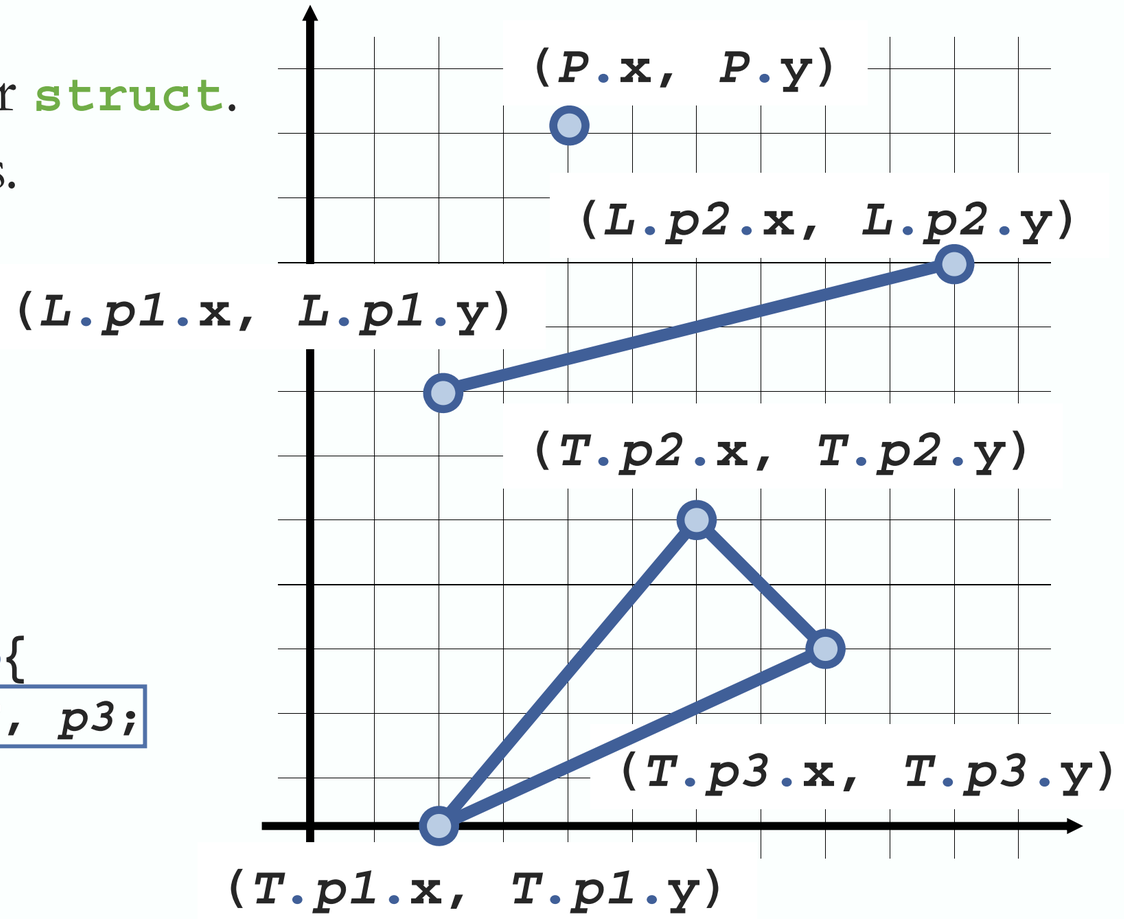
- Program design w.r.t. inherent attributes.

Example:

```
struct point{
    double x, y;
};
point P;
```

```
struct line{
    point p1, p2;
};
line L;
```

```
struct triangle{
    point p1, p2, p3;
};
triangle T;
```



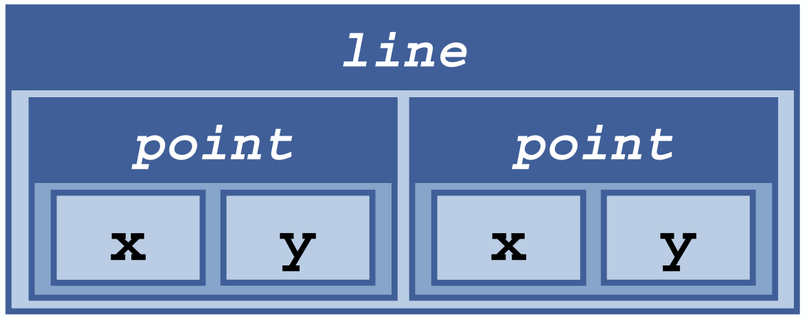
Nested Structures

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

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

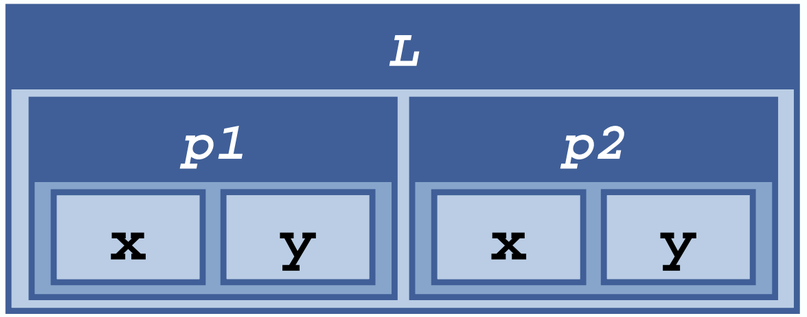
Example:

```
struct line{
   point p1, p2;
};
```



Type Definition

```
line L;
```



Variable Creation

Nested Structures

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

- NOTE: No 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

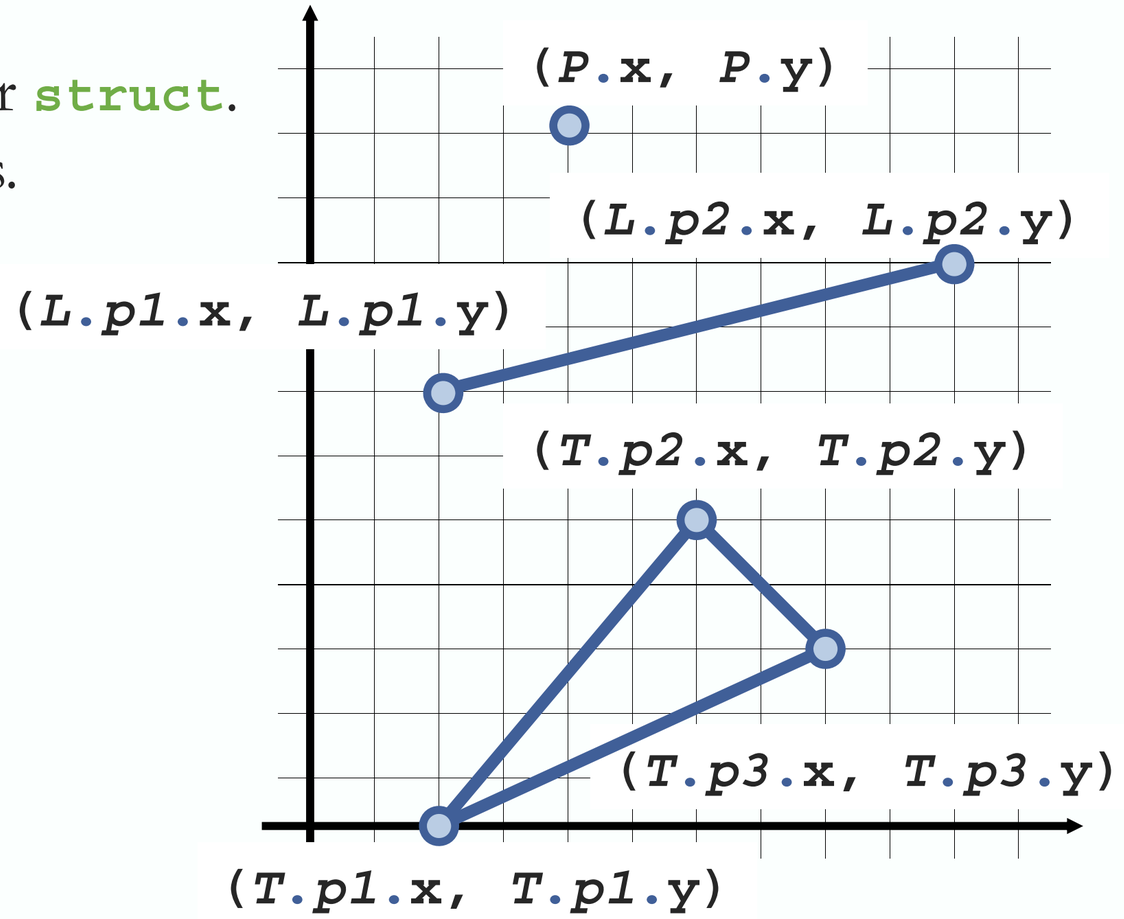
A `struct` type can be a member of another `struct`.

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

Example:

```
point P;
line L;

P.x = 2.00;
P.y = 7.00;
L.p1 = P;

P.x = 9.00;
P.y = 8.00;
L.p2 = P;
```

Data Copying



Nested Structures

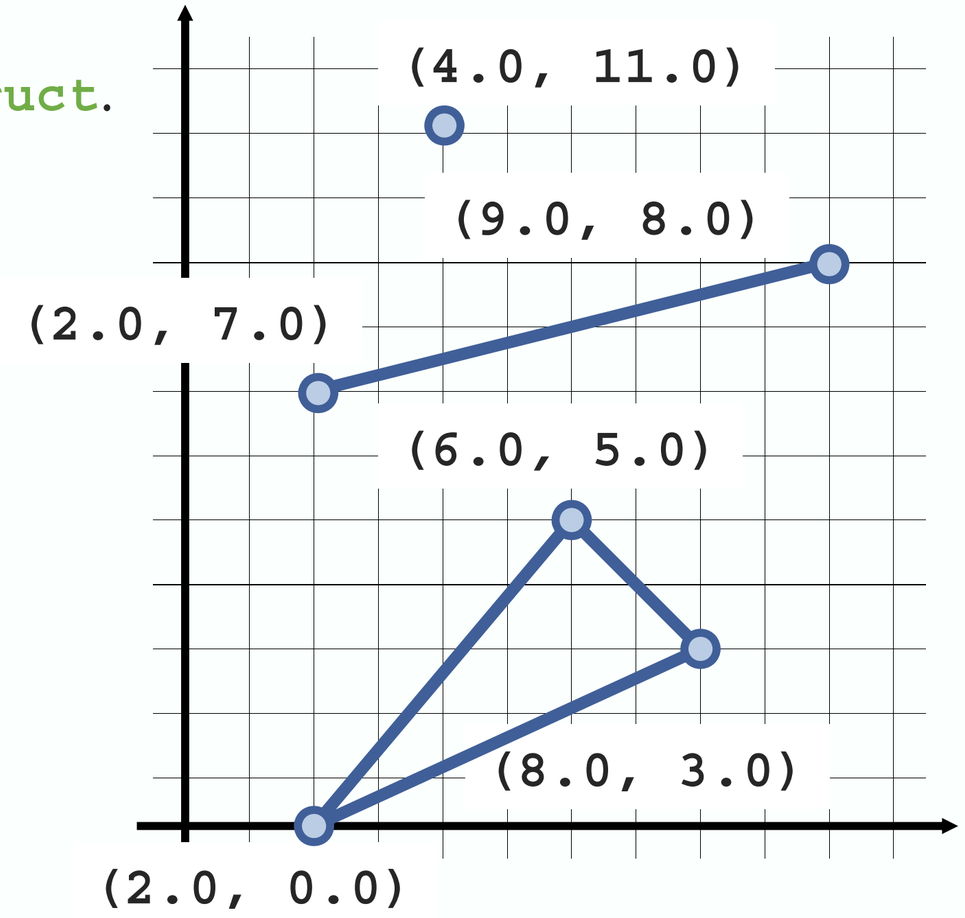
A `struct` type can be a member of another `struct`.

- 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



Nested Structures

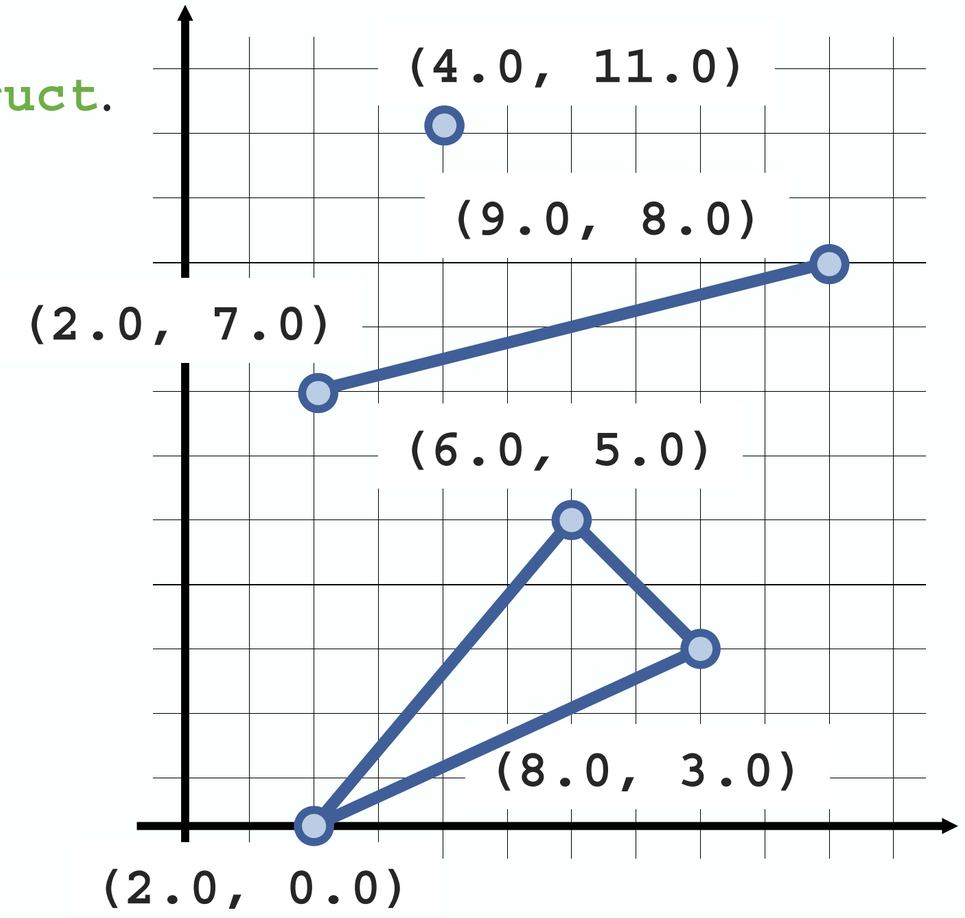
A `struct` type can be a member of another `struct`.

- 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



Nested Structures

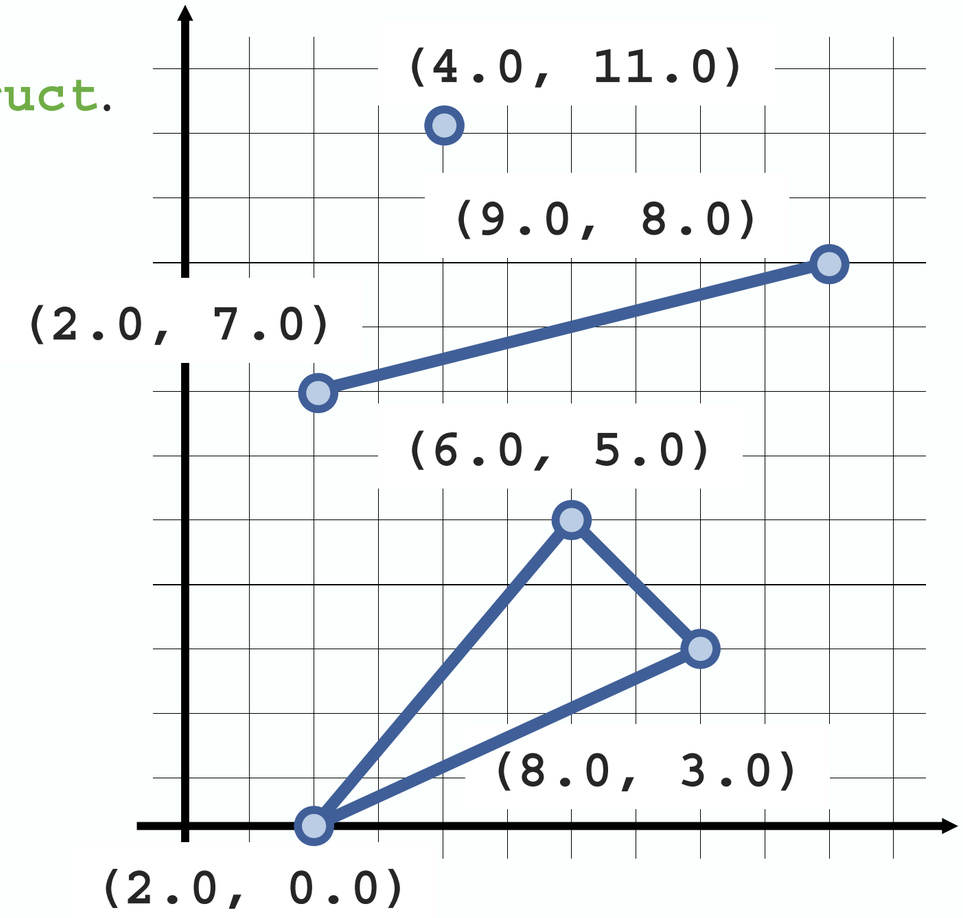
A `struct` type can be a member of another `struct`.

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

Example:

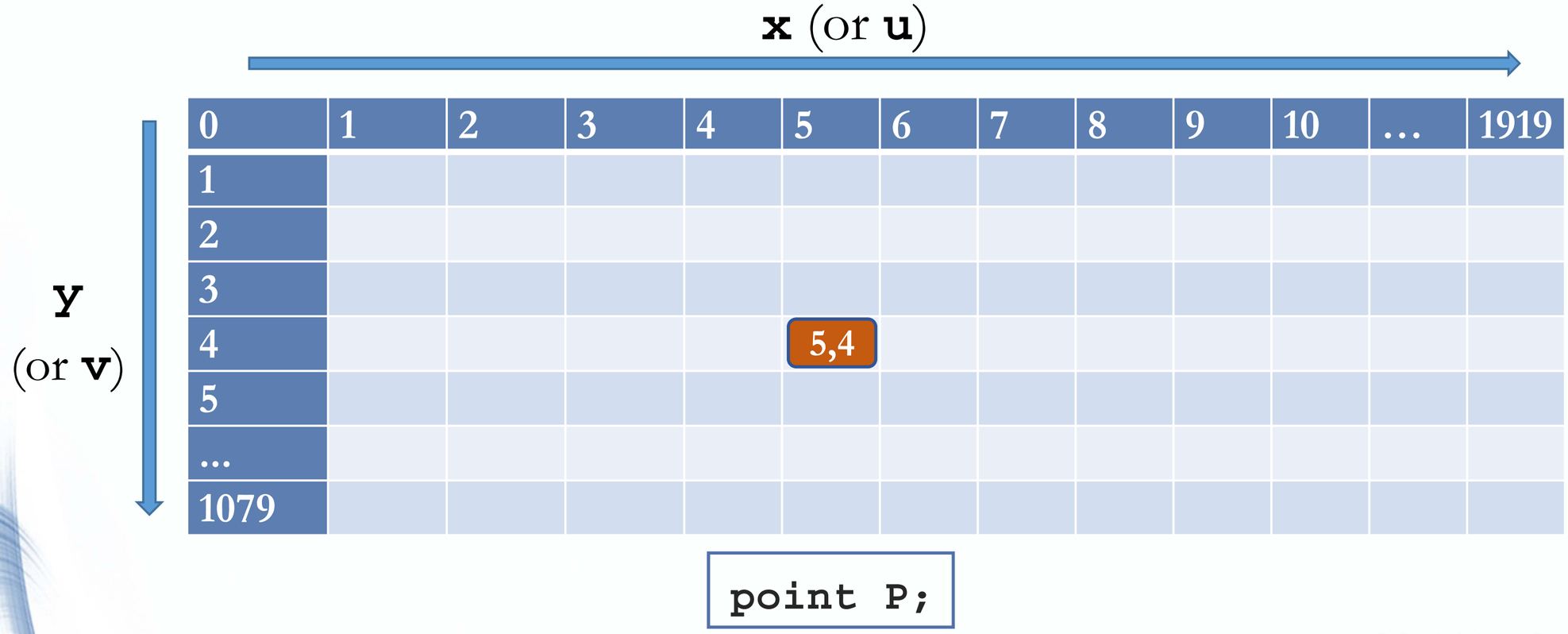
```
point P;
line L;
triangle T;

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



Extended Example

Computer Vision – Graphing shapes on 2D Images: *



Extended Example

```
struct point {int x, y;};  // or u, v in pixel-based notation

void user_input(point& P){ // pass by reference
    // get user input and check that it is on the grid
    do{
    cout << "Enter column (x<" << NUMBER_COLS << ") & row (y<"
        << NUMBER_ROWS <<") of the 1st point: ";
    cin >> P.x >> P.y;
    } while ((P.y<0) || (P.y >= NUMBER_ROWS) ||
        (P.x<0) || (P.x >= NUMBER_COLS));
}
```

Extended Example

```
struct point {int x, y;};  // or u, v in pixel-based notation

// Put a point on the grid
void graph_point(char grid[][NUMBER_COLS], point P){
    grid[P.y][P.x] = '*';
}
```

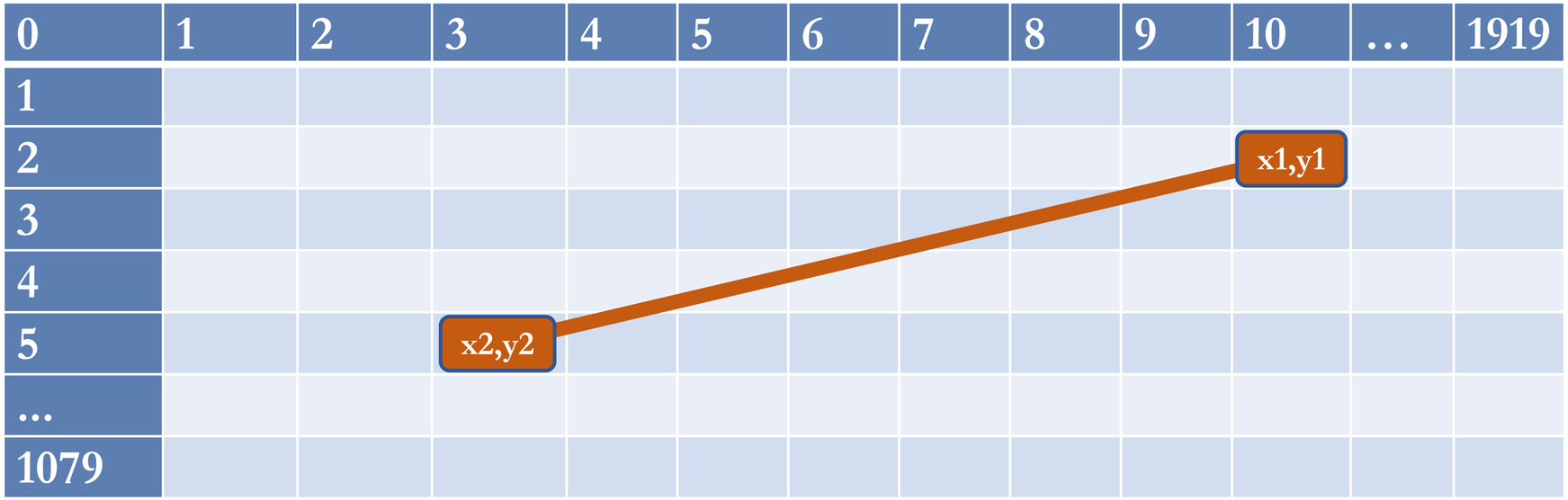
0	1	2	3	4	5	6	7	8	9	10	…	1919
1												
2												
3												
4					★							
5												
…												
1079												

Extended Example

Line equation going through two points (**x1**, **y1**) & (**x2**, **y2**) is represented by:

$$(y-y1)/(x-x1) = (y2-y1)/(x2-x1)$$
$$y = ((y2-y1)/(x2-x1))\,(x-x1) + y1$$

(y2-y1)/(x2-x1) is the "slope"



Extended Example

```
void graph_line(char grid[][NUMBER_COLS], line line1){
    int row, col;
    double rise, run, slope;
    // one point
    if((line1.p1.y==line1.p2.y)&&(line1.p1.x==line2.p2.x))
        grid[line1.p1.y][line1.p1.x] = '*';
    else if(line1.p2.x==line1.p1.x){  // infinite slope
        if (line1.p1.y < line1.p2.y){
            for(row=line1.p1.y; row <= line1.p2.y; row++)
                grid[row][line1.p1.x] = '*';
        }
        else{
            for(row=line1.p1.y; row >= line1.p2.y; row--)
                grid[row][line1.p1.x] = '*';
        }
    }
```

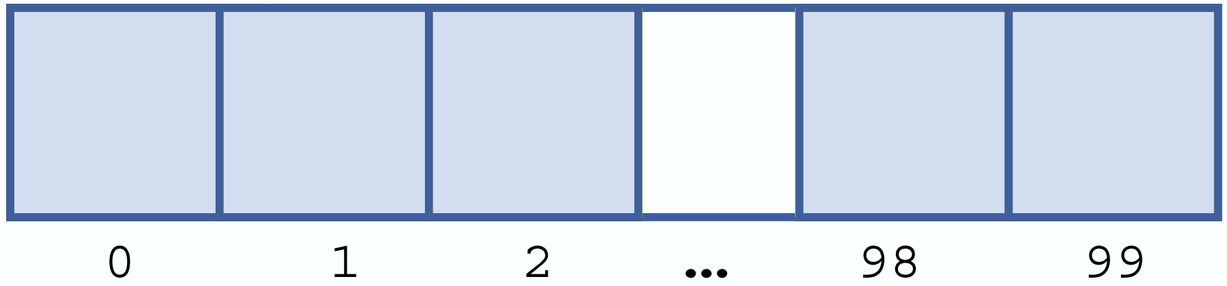
Extended Example

```
        else{
            rise=line1.p2.y-line1.p1.y;  run=line1.p2.x-line1.p1.x;
            slope = (double)rise / run;    // run cannot = 0
            if (run >0){
                for(col = line1.p1.x; col <= line1.p2.x; col++){
                   // line1.p1.y is offset for starting point
                   row=(int)(slope*(col-line1.p1.x)+line1.p1.y);
                   grid[row][col] = '*';
                }
            }
            else{
                for(col=line1.p1.x; col >= line1.p2.x; col--){
                   row=(int)(slope*(col-line1.p1.x)+line1.p1.y);
                   grid[row][col] = '*';
                }
            }
        }
}
```

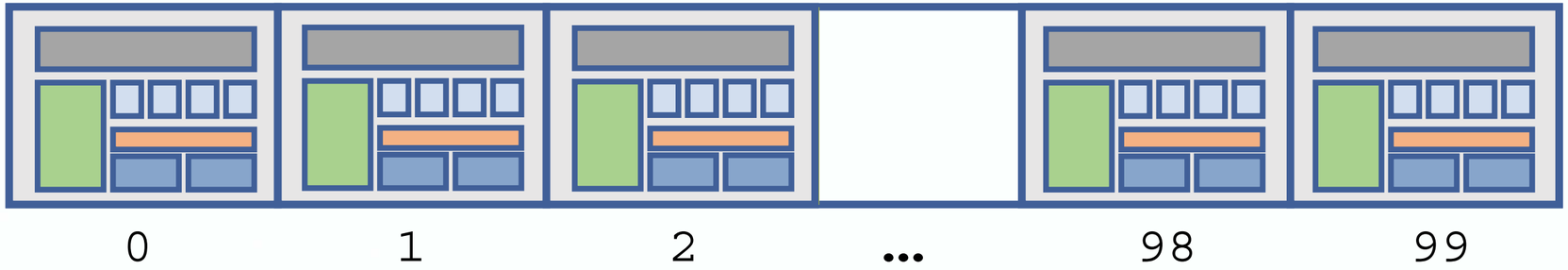
Arrays of Structs

Arrays are *homogeneous* (one data type):

- Regular data type.



- Supported type can be `struct`.

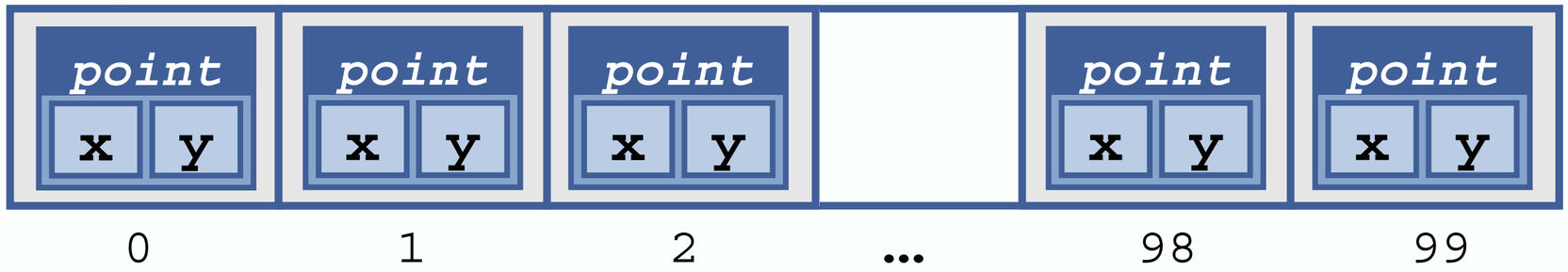


Arrays of Structs

Arrays are *homogeneous* (one data type):

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

point point_array[100];
```

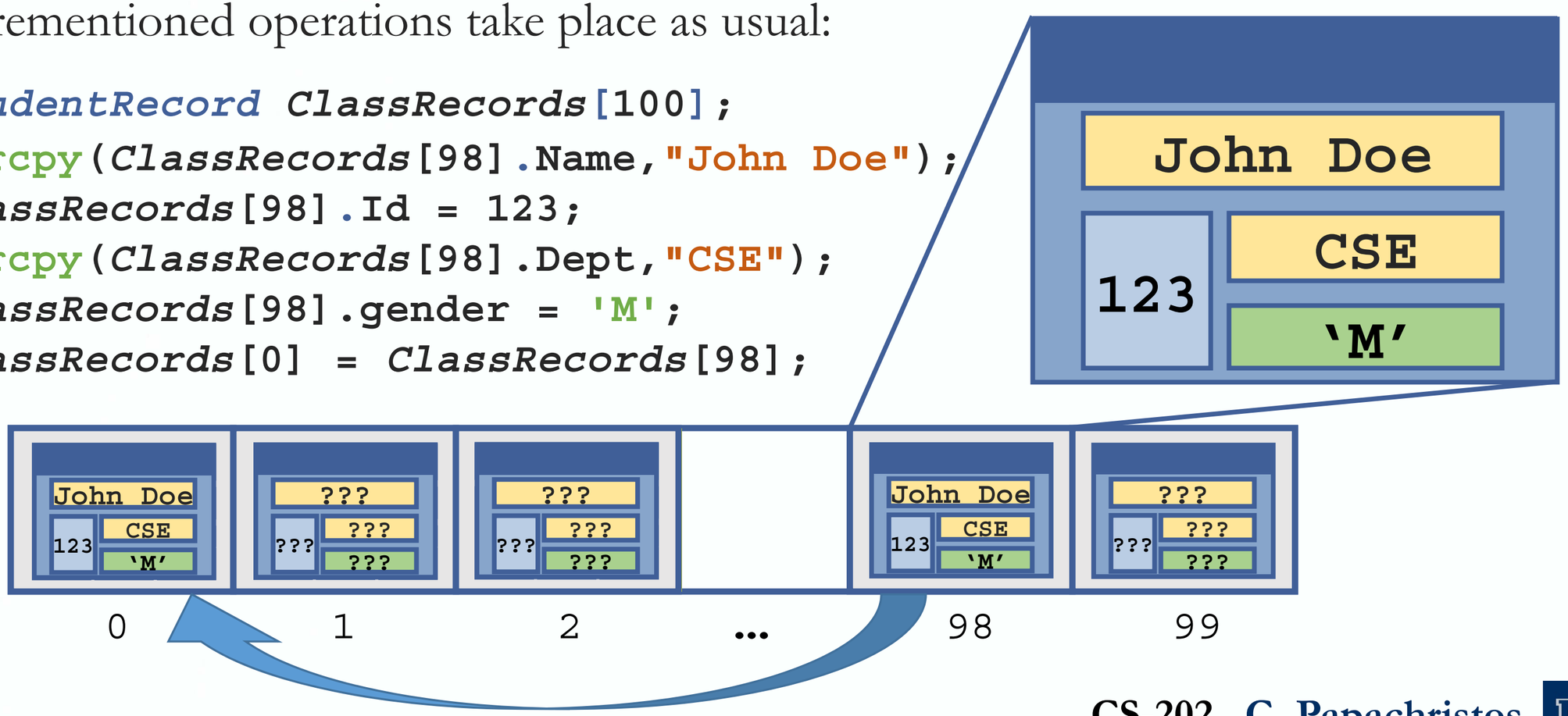
- Supported type can be `struct`.



Arrays of Structs

All aforementioned operations take place as usual:

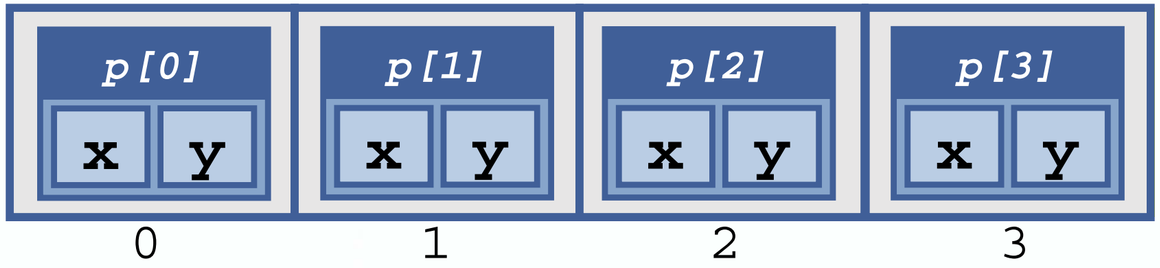
```
StudentRecord ClassRecords[100];
strcpy(ClassRecords[98].Name,"John Doe");
ClassRecords[98].Id = 123;
strcpy(ClassRecords[98].Dept,"CSE");
ClassRecords[98].gender = 'M';
ClassRecords[0] = ClassRecords[98];
```

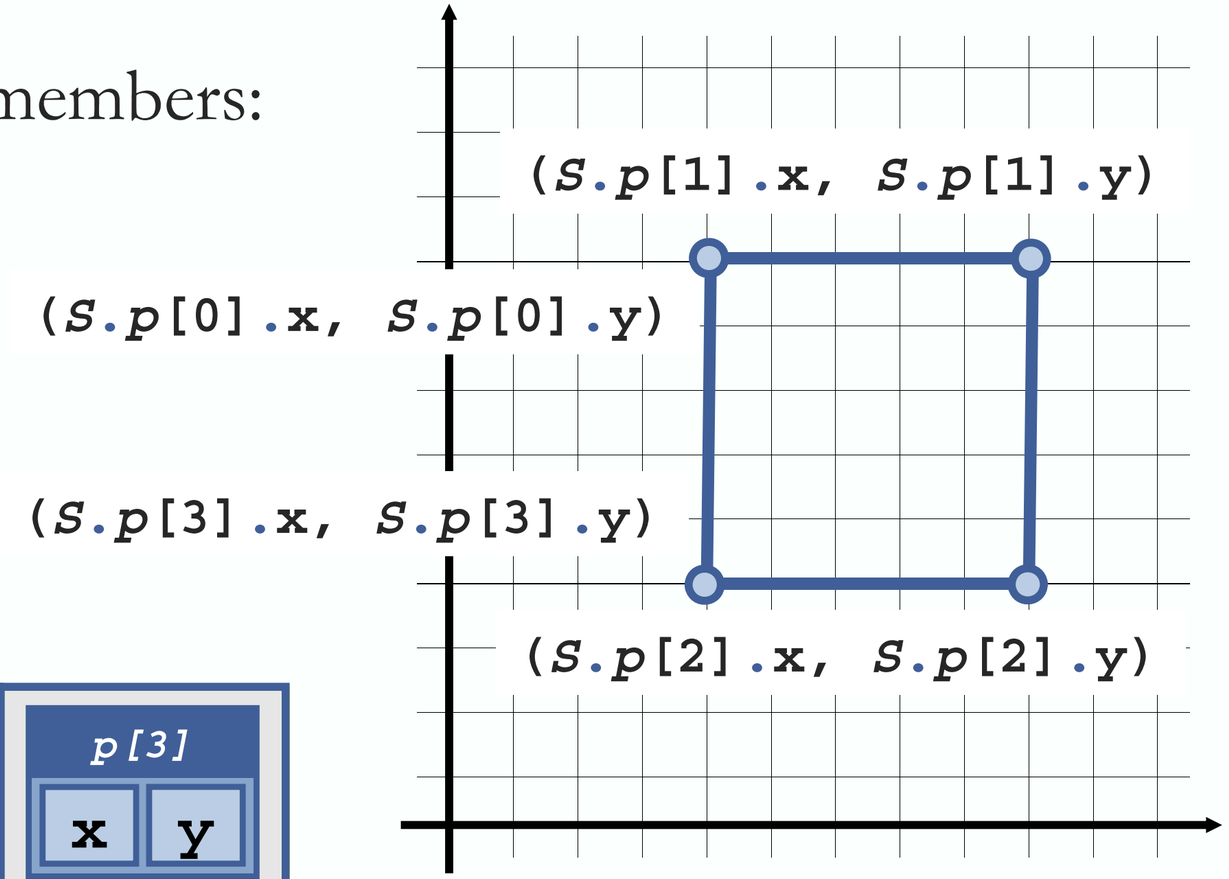


Struct Arrays in Structs

RemeberArrays 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; };
```

- Call-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; };
```

- Call-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; };
```

- Call-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 &`:

- Call-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;
}
```

- Call-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; };
```

- Call-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; };
```

- Struct Array can be Called-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);
```

Modifies `struct` members

C++ Structs

Structs and Functions

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

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

- Struct Array can be Called-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);
```

Parameter similarly as:
`point p_arr[]`

Modifies `struct` members

C++ Structs

Structs and Functions

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

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

- Struct Array can be Called-By-Address

```
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 =`

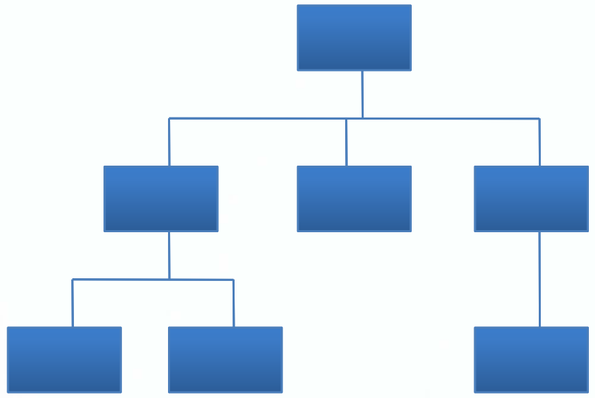
Remember: Procedural *vs* Object-Oriented

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)

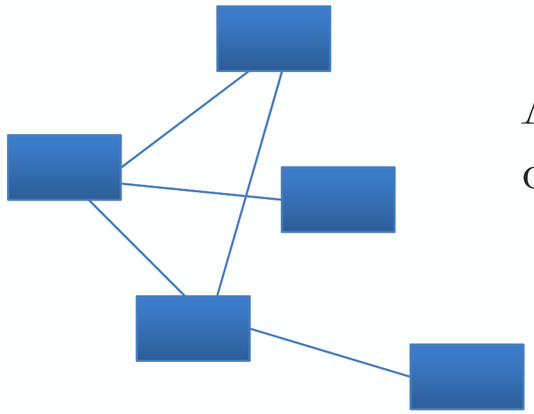
Package-up self-sufficient modular pieces of code.

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

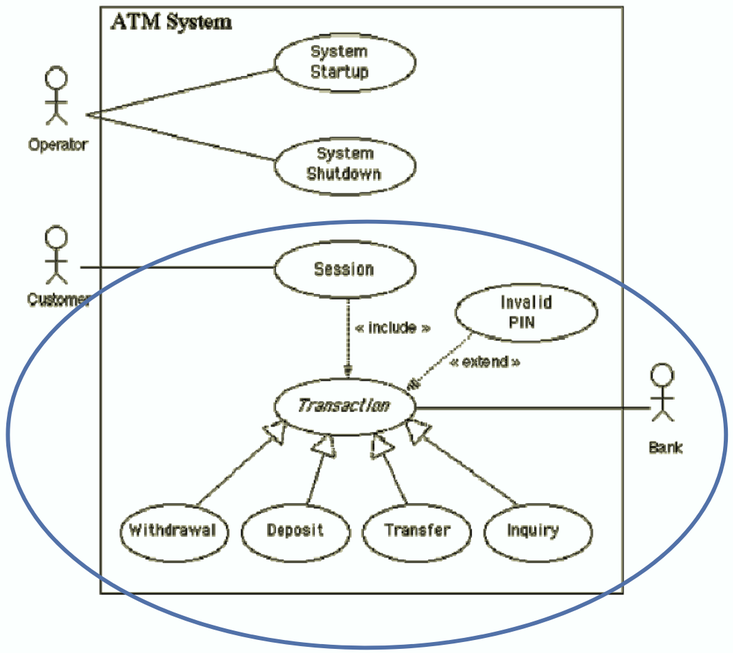


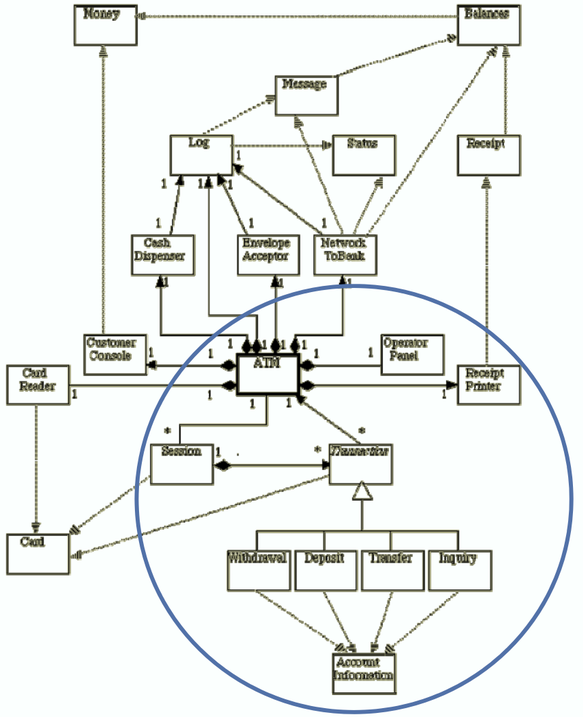
A collection of Objects

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;
};
```

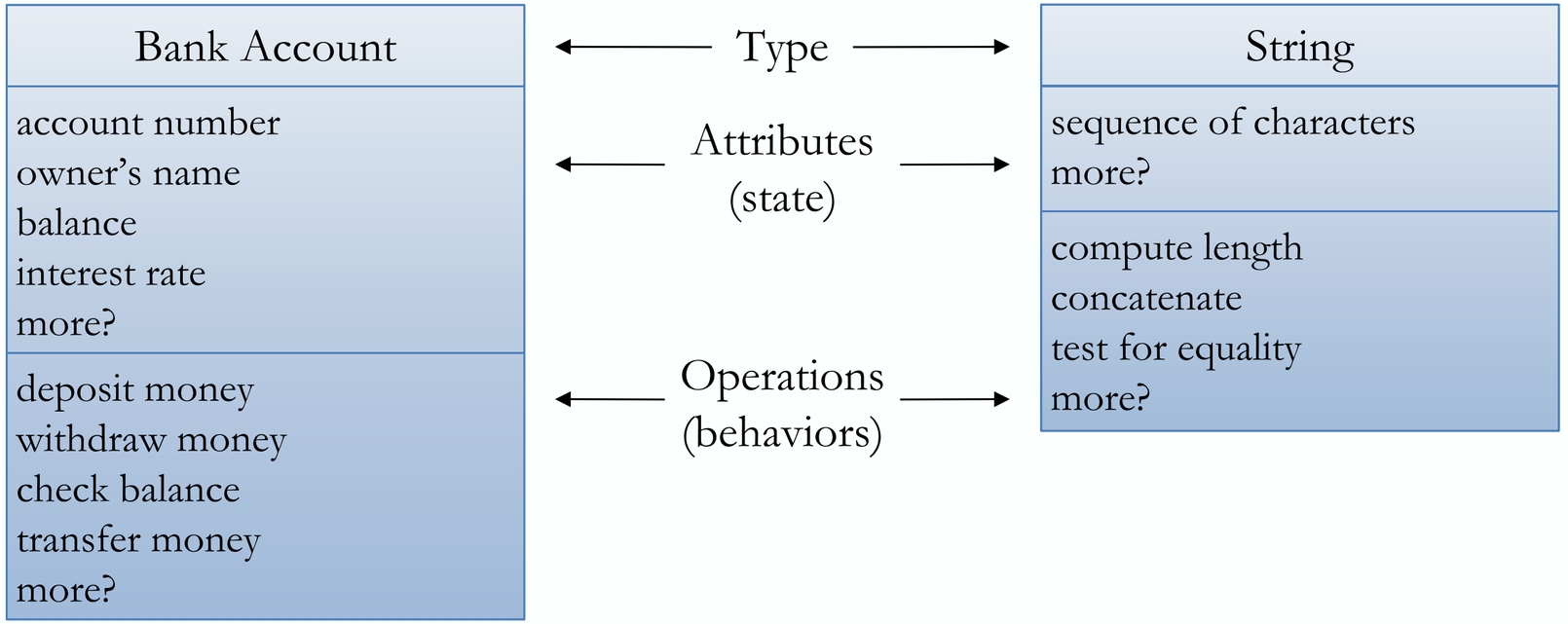




Remember: Classes

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, Book, or Airplane, 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.

(for 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 things that are mostly data-oriented.
- Data-only limitations?
  Data sanity checking.

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.

Structs can have:

- 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)

C++ Structs

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;
        fix_leap_date();
    }
    void fix_leap_date(){
        if (year ... && month ... && day ...){
          day = ...;
        }
    }
};
```

Not a leap year!

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

C++ Structs

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;
        fix_leap_date();
    }
    void fix_leap_date(){
        if (year ... && month ... && day ...){
          day = ...;
        }
    }
};
```

Not a leap year!

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

➢ Constructor call !

C++ Structs

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;
        fix_leap_date();
    }
    void fix_leap_date(){
        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 is_leap_year(){ ... return ... ; }
};

Date bDay(2, 29, 2015);
bool leapYear = bDay.is_leap_year();
```

C++ Structs

Structs in C++

Structs encapsulate related data/behaviours.

Structs can have:

- Member variables
- Methods (i.e. functions)
- Constructors, Destructors, Operators, etc.

Object-Oriented (OO)

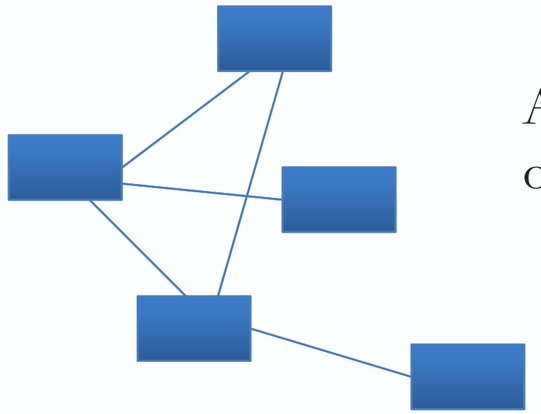
Package-up self-sufficient modular pieces of code.

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

C++ Structs

Structs in C++

Structs encapsulate related data/behaviours.

Structs can have:

- Member variables
- Methods (i.e. functions)
- Constructors, Destructors, Operators, etc.

When to use a `struct`:

- For things that are mostly data-oriented.
- Add Constructors and Operators to work with STL containers/algorithms.

Object-Oriented (OO)

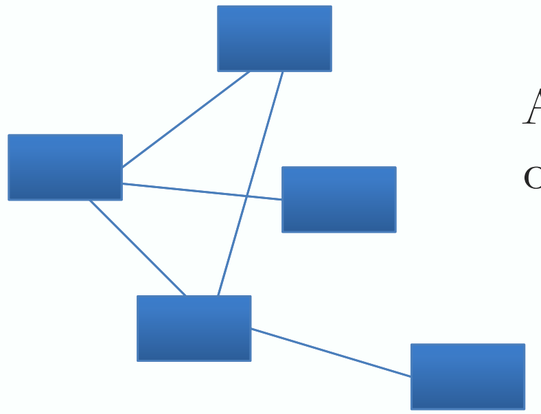
Package-up self-sufficient modular pieces of code.

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

By the way … (OO in C)

Did you know?

```
typedef struct student_t student_t, *student_Pt_t;
struct student_t
{
    char Name[15];
    int Id;
    student_Pt_t EmergencyContact;

    void (*ConstructStudent)();
    student_Pt_t (*AddEmergContact)(student_t *);
};

void create_student() { ... }
student_Pt_t add_emerg_contact(student_t *self) { ... }
```

C++ Structs

By the way … (OO in C)

Did you know?

```
typedef struct student_t student_t, *student_Pt_t;
struct student_t
{
    char Name[15];
    int Id;
    student_Pt_t EmergencyContact;

    void (*ConstructStudent)();
    student_Pt_t (*AddEmergContact)(student_t *);
};

void create_student() { ... }
student_Pt_t add_emerg_contact(student_t *self) { ... }
```

`typedef` required

C++ Structs

By the way … (OO in C)

Did you know?

```
typedef struct student_t student_t, *student_Pt_t;
struct student_t
{
    char Name[15];
    int Id;
    student_Pt_t EmergencyContact;

    void (*ConstructStudent)();
    student_Pt_t (*AddEmergContact)(student_t *);
};

void create_student() { ... }
student_Pt_t add_emerg_contact(student_t *self) { ... }
```

`typedef` required

`void*` (or worse) members required

C++ Structs

By the way … (OO in C)

Did you know?

```
typedef struct student_t student_t, *student_Pt_t;
struct student_t
{
    char Name[15];
    int Id;
    student_Pt_t EmergencyContact;

    void (*ConstructStudent)();
    student_Pt_t (*AddEmergContact)(student_t *);
};

void create_student() { ... }
student_Pt_t add_emerg_contact(student_t *self) { ... }
```

`typedef` required

`void*` (or worse) members required

External Functions required

C++ Structs

By the way … (OO in C)

Thankfully C++ is much more Versatile, Expressive than that !

```
struct student_t
{  ...
    void (*ConstructStudent)();
    student_Pt_t (*AddEmergContact)(student_t *);
};
void create_student() { ... }
student_Pt_t add_emerg_contact(student_t *self) { ... }

int main()
{
   student_t student_a;
   student_a.ConstructStudent = &create_student;
   student_a.ConstructStudent();
}
```

Function Binding required

CS-202

Time for Questions !