COMSC-110 Chapter 12

Working with database records: structs

Valerie Colber, MBA, PMP, SCPM

Data categories

- Numbers
- Text
- Graphics
- Audio
- Video

Data types

- Integer
- Character
- String
- Double
- Float
- File
- Boolean
- Arrays
- Constants

Data structures (covered in this course)

- Literal values
- Variables
- Arrays
- Structs
- Arrays of structs
- Lists
- Data collections

What Is a Database?

- Database: A collection of related data stored in a manner so it can be retrieved as needed
- Database management system (DBMS): Used to create, maintain, and access computer databases
  - Includes database engine: Part of the program that stores and retrieves the data
  - Various tools: Used to perform various tasks
- A database typically consists of:
  - Tables: Contain fields and records
  - Fields (columns): Single category of data to be stored in a database
  - Records (rows): Collection of related fields in a database

Multi-valued records

- What you need to know about multi-valued records is that the language provides a way to group programmer-defined sets of values into new data types.
- Variables of these data types can store multiple values -- all the values of the fields of a record.

Programmer-Defined Data Types

- Other variables that can store multiple values besides arrays
- Array's store multiple values, but...they must all be of the *same data type*
- suitable for *lists*
- not so suitable for *database records*
- so C++ allows programmer-defined data types to let YOU chose the multiple values' data types

What is a struct in C++?

- A data **structure** is a group of data elements grouped together under one name.
- These data elements, known as members, can have different types and different lengths.

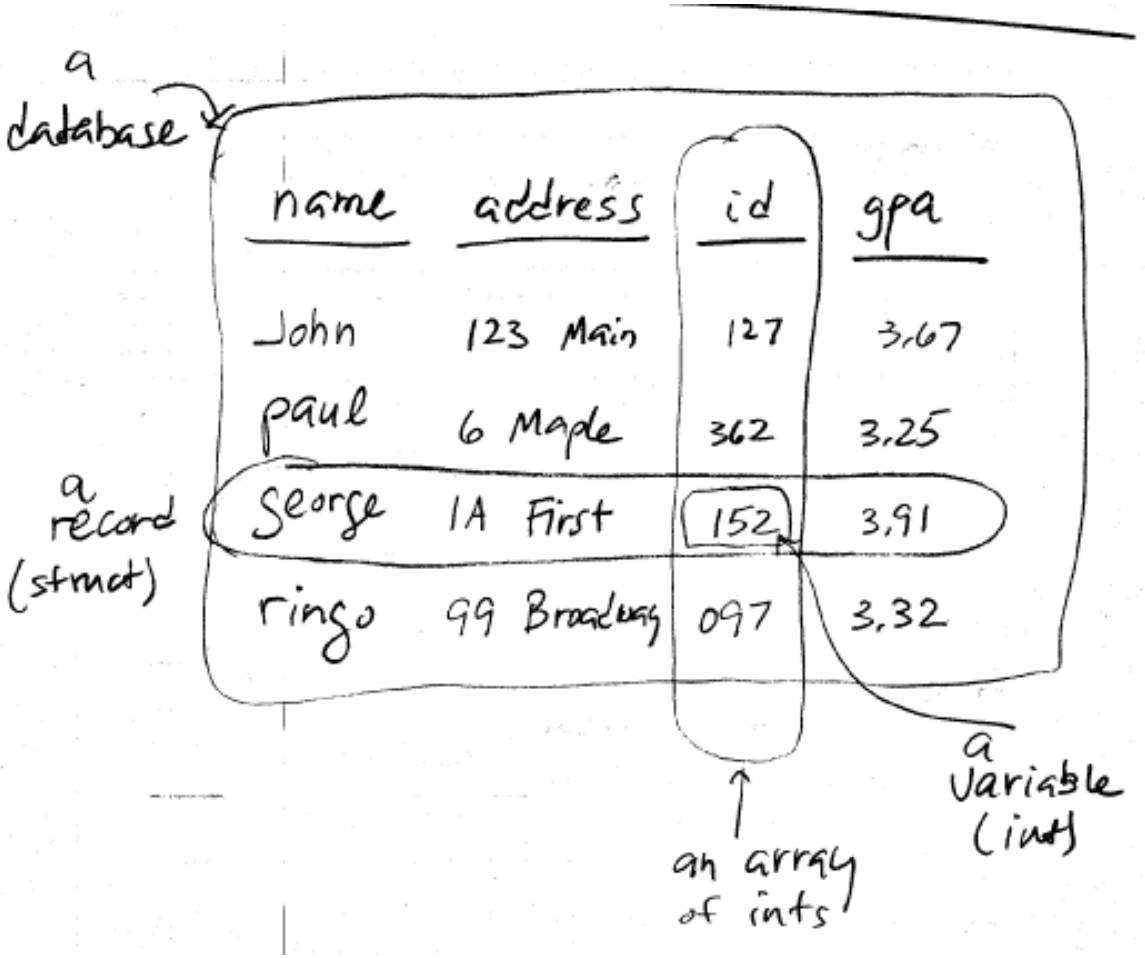
The Application: Records

- programmer-defined data types are for *records*
- YOU specify the "fields" for the "record", and their data types
- example: name as text, gpa as floating point number,
- gender as 1-character text, ID as integer
- 2 ways in C++: structs and classes
- we will study structs

Struct

- So along the same lines as arrays, C++ offers another feature for grouping *unlike* values into the same variable.
- The *struct* enables programmers to do this.
- C++ also has *classes*, which extend the capabilities of structs into "object-oriented programming“ (not covered in this course)

Variables, arrays, records, and data bases



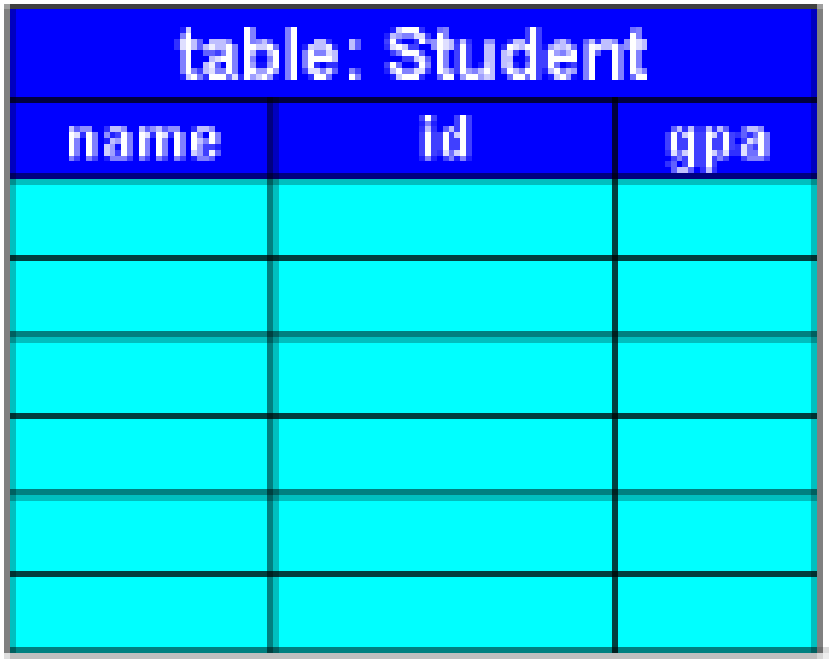
Struct definition/representation

y:

A struct Definition

```
struct Student
{
   string name;
   int id;
   float gpa;
}; // Student
```

Representation in code



Conceptual representation

Struct definition

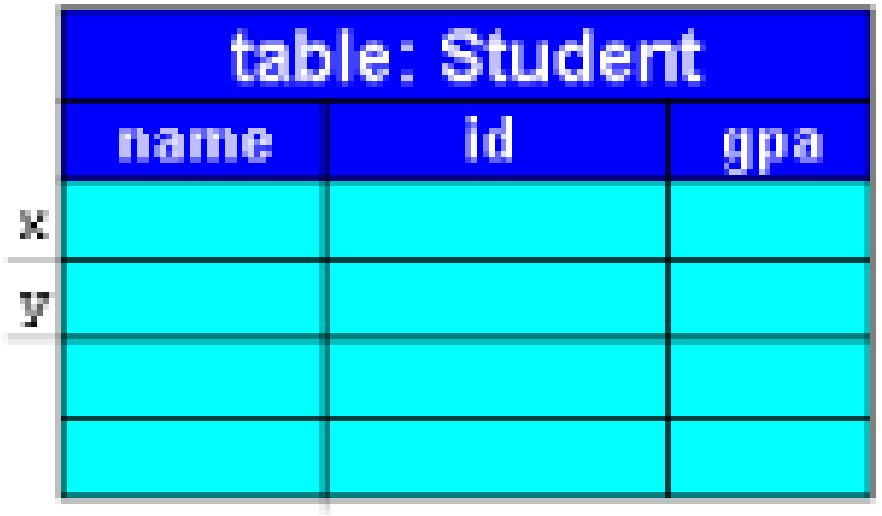
- The **struct** definition represents an empty data table, with each row able to represent a record, and each column representing a field.
- Note that just defining the **struct** is like typing column headings -- data records will be added later.
- Where the definition goes is *below* the includes and *above* the constants (if any) and functions.
- It actually defines a *new data type*, that you can now use in the rest of your program below its definition, so *be careful not to choose an identifier that matches any data type of C++ (like **string**).*

Struct definition

- You can use any previously-defined data type for a field in your record -- even one defined by yourself above.
- But you *cannot* use the struct being defined as a data type for one of its own fields.
- In other words, **struct Student** cannot have a field **Student bestFriend;** in it.
- Note the semicolon after the closing curly brace of the struct definition -- it's required.
- Leaving it off can result in pages of error messages from the compiler -- so don't forget it!

Struct variable definitions

```
Student x;
Student y;
```



Structs versus classes

```
#include <iostream>
#include <string>
using namespace std;

struct Student
{
  int id;
  string name;
  float gpa;
  char gender;
};

int main()
{
  Student x;
  Student y;

  return 0;
}
```

```
#include <iostream>
#include <string>
using namespace std;

class Student
{
  public:
  int id;
  string name;
  float gpa;
  char gender;
};

int main()
{
  Student x;
  Student y;

  return 0;
}
```

Structs versus classes

(Advanced C++ not covered in this course)

- The main purpose of **C++** programming is to add **object** orientation to the C programming language and classes are the central feature of **C++** that supports **object**-oriented programming and are often called **programmer-defined data types.**
- A class is used to specify the form of an object and it combines data representation and methods for manipulating that data into one neat package.
- The data and functions within a class are called members of the class.
- **We will NOT be using classes in this course.**

struct Variables In Functions

```
Student getGeorge()
{
  Student george;
  ...
  return george
} // getGeorge
```

```
void printRecord(Student& x)
{
  ...
} // printRecord
```

```
int main()
{
  ...
  Student s = getGeorge();
  ...
  printRecord(s);
  ...
} // main
```

Struct variables in Functions

- Note the ampersand that is appended to the data type name in the parameter list

  (Student& x)

- This symbol is actually not required, but it is good idea to use it here.
- The ampersand symbol tells the compiler to pass a variable by *reference* that would otherwise be passed by *value*.
- This prevents the making of a copy, and uses the original variable in the function.

Assigning and accessing struct variables

- But there are some things you cannot do with struct variables, such as use them directly in a **cout** statement.
- C++ does *not* know to print each field in order, each separated by a space, when you use a struct variable in a console output statement!
- The language provides the "dot operator", which extracts a field by its identifier.

Assigning and accessing struct variables

```
Student a;
Student b;
```

```
a.name = "George";
a.id = 17890;
a.gpa = 3.67;
```

```
b.name = "John";
b.id = 17970;
b.gpa = 3.53;
```

table: Student

	name	id	gpa
a	George	17890	3.67
b	John	17970	3.53

Read the ID from a file

```
fin >> a.id;
fin.ignore(1000, 10);
```

Output a record

```
cout << "Name=" << a.name;
cout << ", ID=" << a.id;
cout << ", GPA=" << a.gpa;
```

Assigning fields of a record in Student

```
Student x;
Student y;
...
x = y;
```

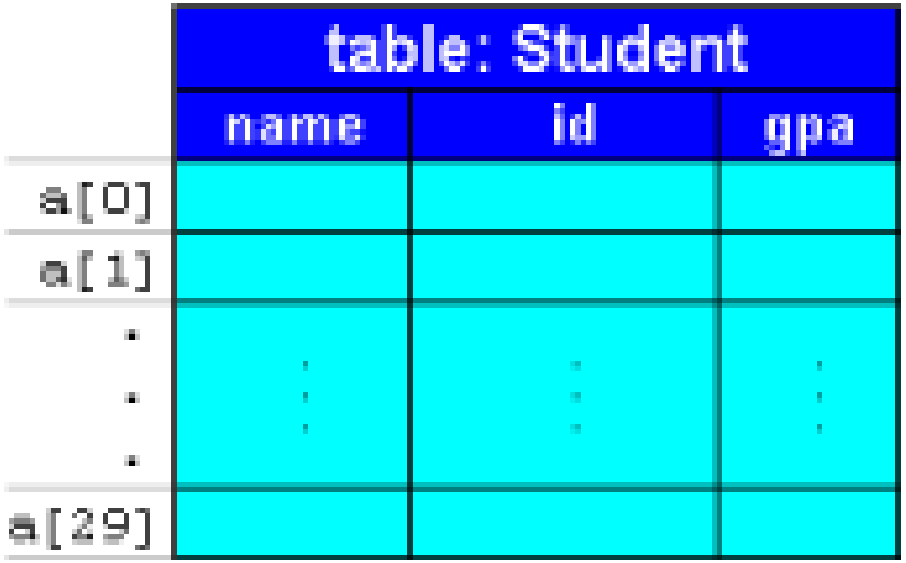
Curly brace initialization

Student george = {"George Washington", 1, 4.0};

- Use curly brace initialization to set the values of a struct variable *when it is declared*.
- Make sure the order of the data values matches the order in which the are listed in the struct definition.
- Also be careful not to type leading zeros for the student ID -- leading zeros represent octal base 8 numbers!
- Also note that the use of strings in curly brace initialization does not work Microsoft Visual C++ 6.0!

Arrays of struct Variables

```
Student a[30];
```



Arrays of struct Variables

- For **Student a[30];** , the way to get to the name of the first student is **a[0].name** .
- The remaining fields are **a[0].id** and **a[0].gpa** .
- The length of the first student's name is **a[0].name.length()** , which looks complicated, but is not so bad when you break it down into parts.
- To set the grade point average of the first student, use **a[0].gpa = 3.67;** .
- To read the first student's ID from a file, use:

```
fin >> a[0].id;
fin.ignore(1000, 10);
```

todTest.cpp

```
#include <iomanip>
#include <iostream>
using namespace std;
struct tod
{
  int hour;
  int minute;
  int second;
}; // tod
void printTod(tod& t)
{
  cout.fill('0');
  cout << setw(2) << t.hour << ':‘ << setw(2) << t.minute << ':‘ << setw(2) <<
t.second << endl;
  cout.fill(' '); // set back to a space
}
int main()
{
  tod noon = {12};
  printTod(noon);
  return 0;
} // main
```

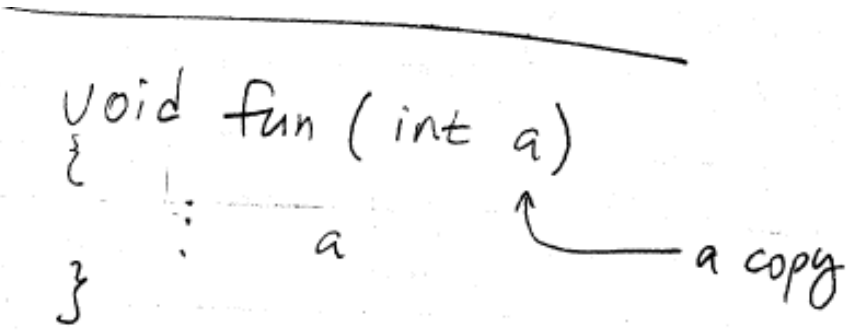
todTest.cpp

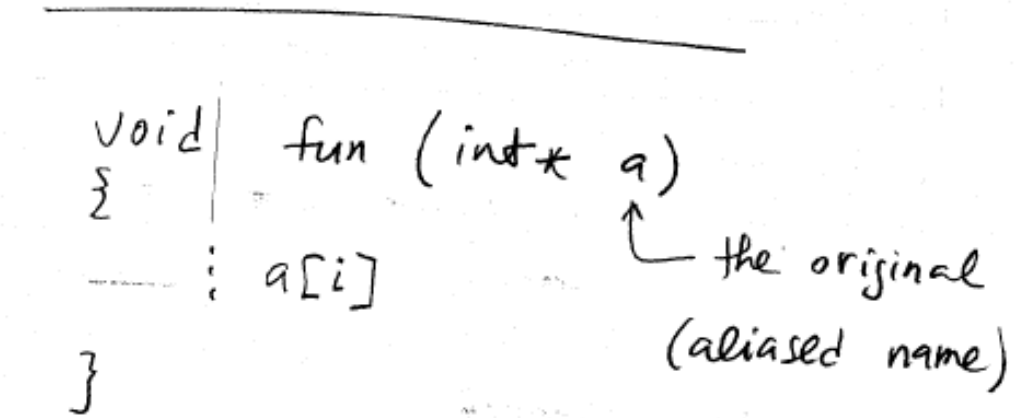
- Note that for the **minute** and **second** data field values are not specified.
- Any fields left out of the curly brace initialization are set to zero -- just as missing elements are in the curly brace initialization of arrays.
- Also note that the function parameter list includes a specification for the struct variable.
- The ampersand symbol **&** in the struct variable specification is *not* a typo -- it tells the program to share the original struct variable with the function instead of sharing a copy.

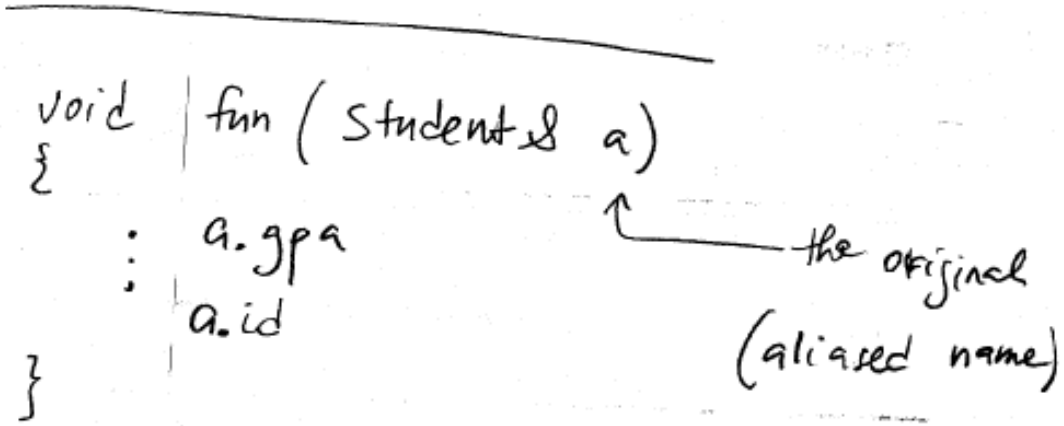
Records in Parameter lists

in parameter list: void print(Student& s)...
note the ampersand
in call:
Student al;
...
print(al);
note: this passes/shares the original -- not a copy!

Parameters


void fun (int a)
{
a
}
a copy


void fun (int* a)
{
a[i]
}
the original
(aliased name)

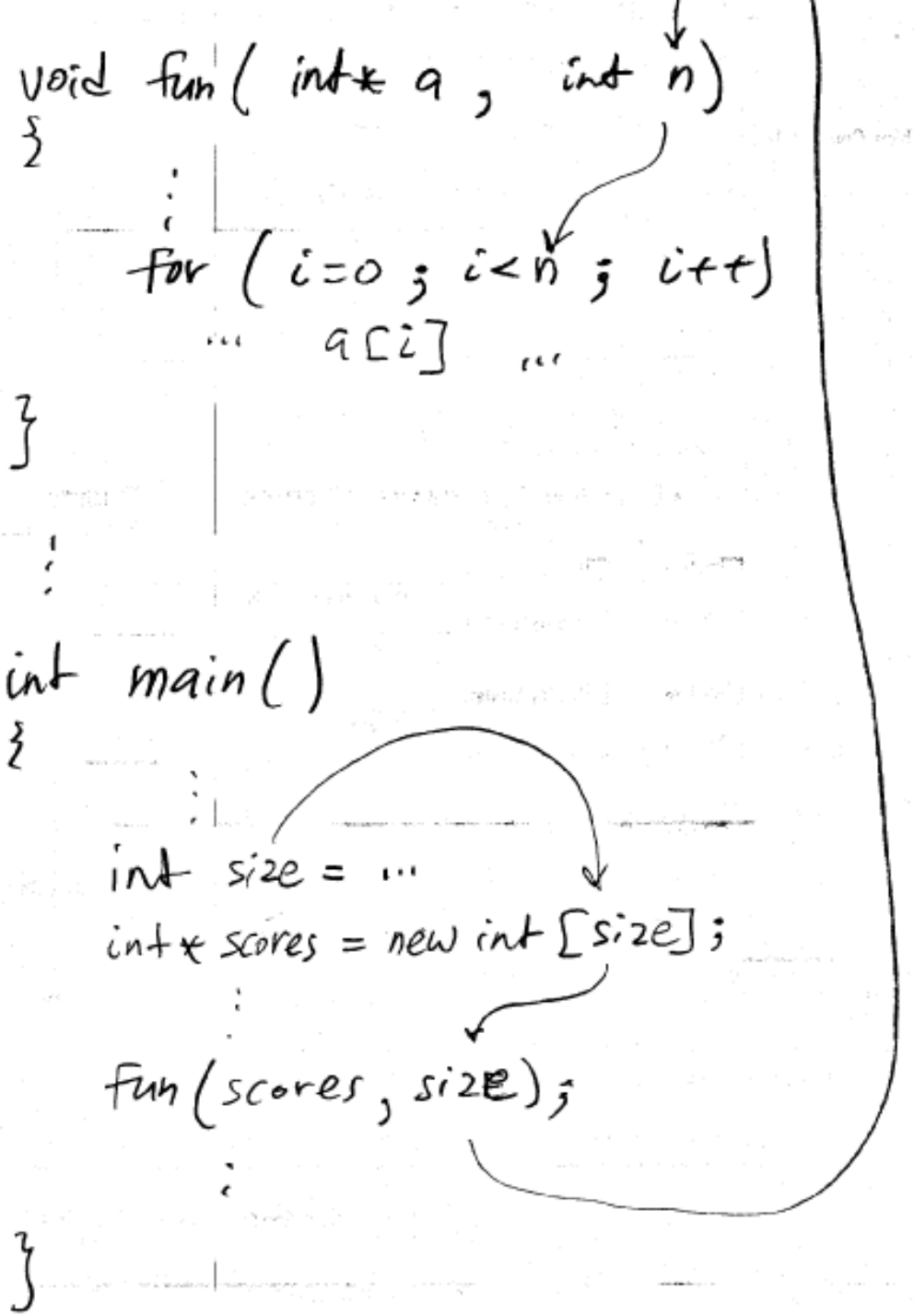

void fun (Student& a)
{
a.gpa
a.id
}
the original
(aliased name)

Array as parameter

problem : array size unknown !

SOLUTION 1 (fixed size arrays only)

```
const int SIZE = 100;        <-- global constant
void fun (int* a)
{
    :
    for (i=0 ; i<SIZE ; i++)
        ... a[i] ...
}
:
int main()
{
    :
    int scores[SIZE];
    :
    fun(scores);
    :
```


void fun(int* a , int n)
{
for (i=0 ; i<n ; i++)
... a[i] ...
}
int main( )
{
int size = ...
int* scores = new int [size];
fun(scores, size);
}

How to share an ENTIRE ARRAY OF RECORDS with a function:

```
struct Student
{
...
};
...
void fun(Student* a) // the * indicates that "a" is an array
{
...
}
...
int main()
{
  Student a[10];
...
  fun(a); // the "a" parameter sends the whole array to the function
...
}
```

How to share a SINGLE RECORDS FROM AN ARRAY OF RECORDS with a function

```
struct Student
{
...
};
...
void fun(Student& x) // the function does not know or care that "x" lives in an array
{
...
}
...
int main()
{
  Student a[10];
...
  fun(a[0]); // the parameter refers to ONE RECORD from the "a" array
...
}
```

References

- INTRODUCTION TO PROGRAMMING USING C++ by Robert Burns
- Understanding Computers Today and Tomorrow by Deborah Morley and Charles Parker