Structs and Arrays of Structs

Mark Sheldon
Tufts University
Email: msheldon@cs.tufts.edu
Web: http://www.cs.tufts.edu/~msheldon

Noah Mendelsohn
Tufts University
Email: noah@cs.tufts.edu
Web: http://www.cs.tufts.edu/~noah

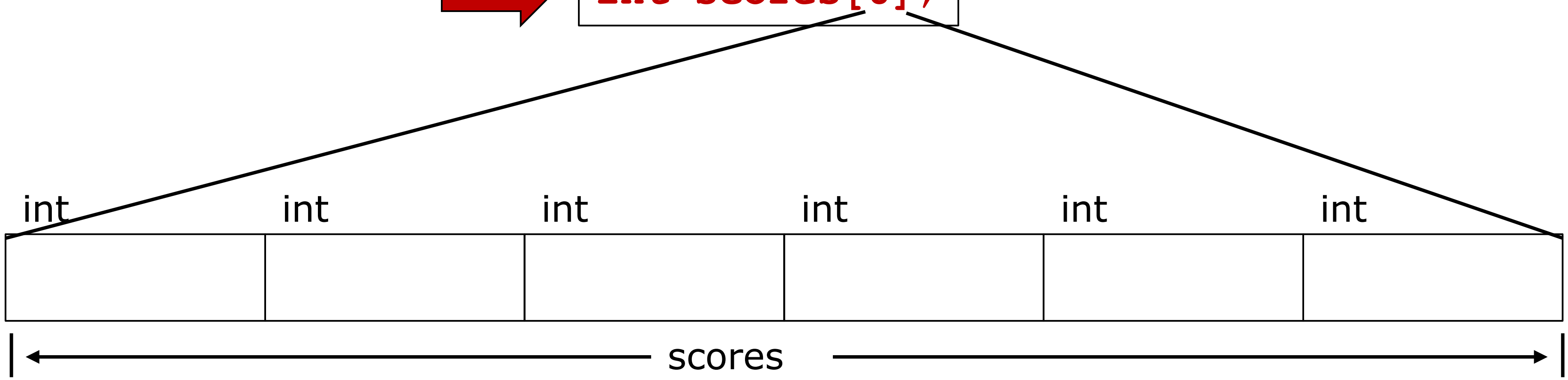
Goals for this session

- **Briefly review: arrays and loops**
- **Briefly show: string subscripting and length**
- **Detailed exploration of structs**
  - Why structs?
  - Initializing structs
  - Functions that process structs; functions that return structs
- **Arrays of structs…looping through arrays of structs**
- **If there's time: loops and recursion compared**

Review: Arrays

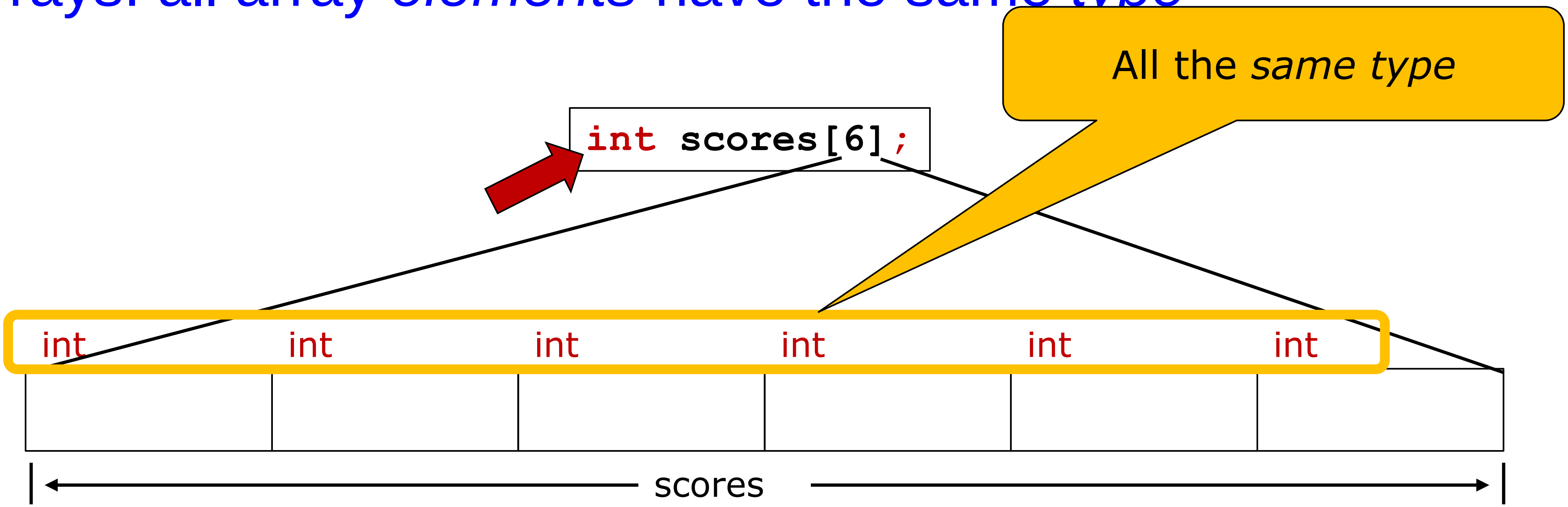
Declaring an arrays: one array can hold lots of values

```
int scores[6];
```



Space reserved in the computer's memory for <u>6 integers</u>, all in the array named "scores".

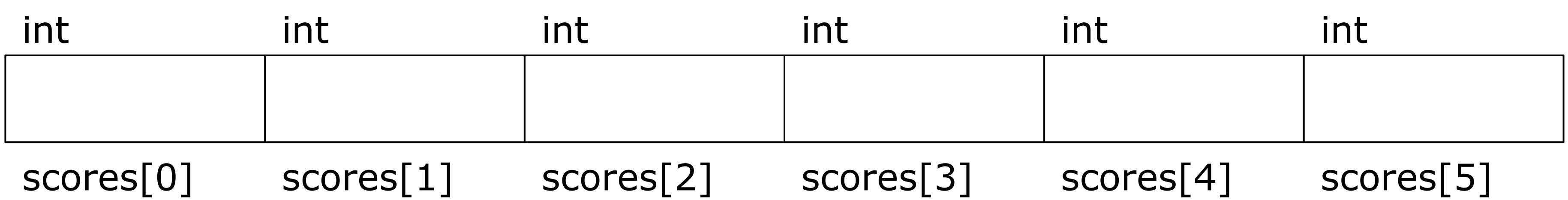
Arrays: all array *elements* have the same *type*



Space reserved in the computer's memory for 6 integers, all of type "int"

Using *subscripts* to address individual array elements

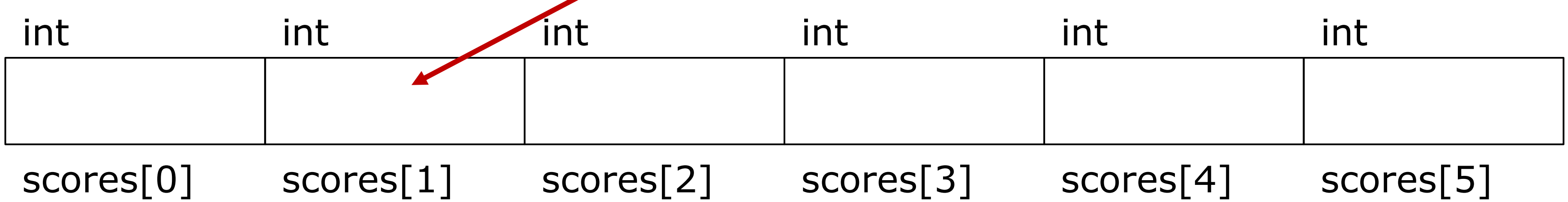
```
int scores[6];
scores[1] =  3;
scores[5] = 12;
```



Space reserved in the computer's memory for the whole array!

Using *subscripts* to address individual array elements

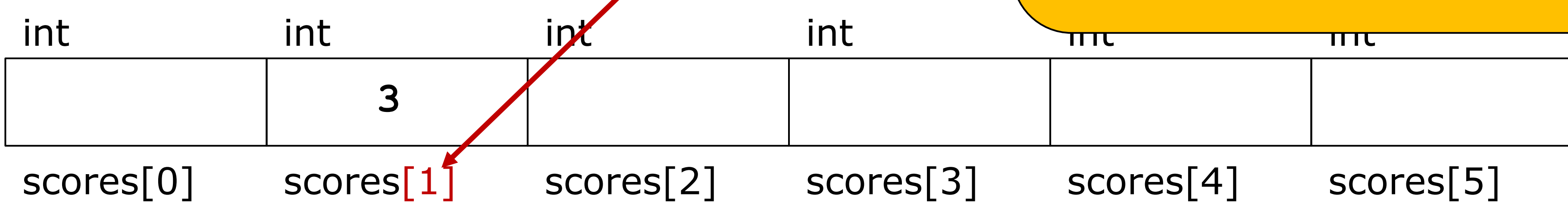
```
int scores[6];
scores[1] =  3;
scores[5] = 12;
```



Using *subscripts* to address individual array elements

```
int scores[6];
scores[1]
scores[5] = 12;
```

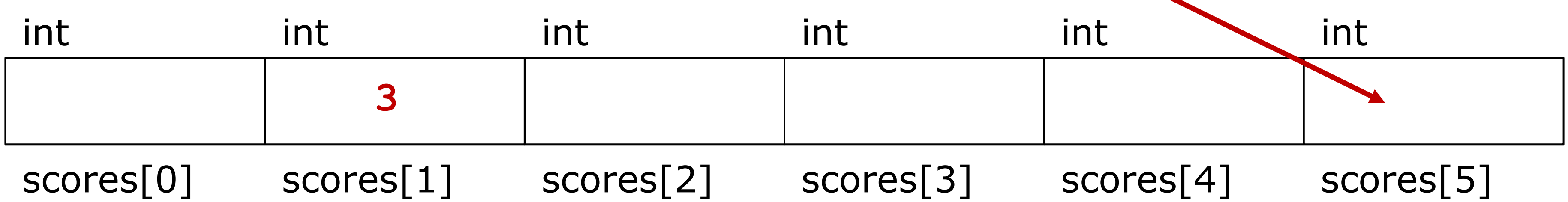
The value in the brackets is called a *subscript* or an *index*……it identifies the particular element in the array



We can *subscript* the array <u>to choose one member</u>…
that member is itself a variable of type int!

Arrays: but how can we use these many variables?

```
int scores[6];
scores[1] =  3;
scores[5] = 12;
```

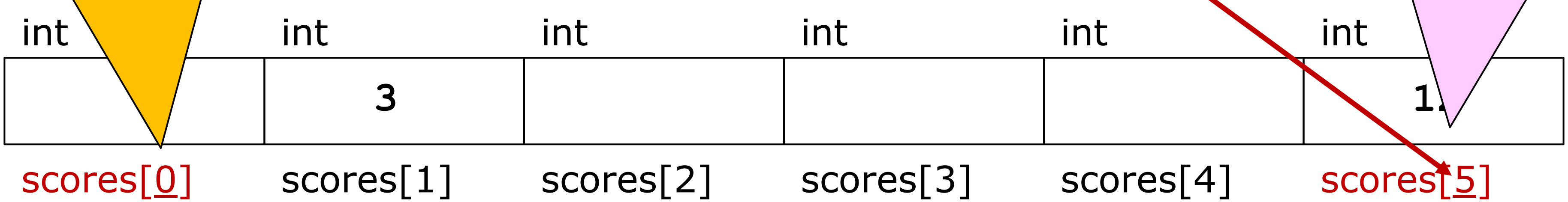


Arrays: first element is always [0] … last is [size-1]

First element is *always* at [0]
…
not [1]!!

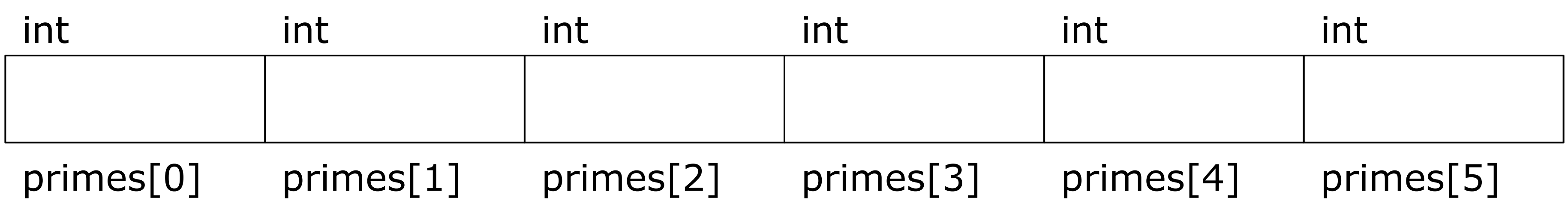
```
int scores[6];
scores[1] =  3;
scores[5] = 12;
```

Therefore last index is always number_of_elements - 1



Array initialization

```
int primes[6] = {2, 3, 5, 7, 11, 13};
```



Review: Loops

The three important features of a typical loop

```
// Writes 10, 9, 8 ..... Blastoff!
// Author: Noah Mendelsohn

#include <iostream>
using namespace std;

int main()
{
  int i = 10;

  while (i >= 1) {
          cout << i << endl;
          i = i - 1;
  }

  cout << "Blastoff!!" << endl;
  return 0;
}
```

Initialize

Termination conditional

Update loop variable

Getting started with loops: the `while` statement

These three things are so common when creating loops that C++ gives us convenient statement that does them all together…the `for` loop!

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

int main()
{
   int i = 10;

   while (i >= 1) {
            cout << i << endl;
            i = i - 1;
   }

   cout << "Blastoff!!" << endl;
   return 0;
}
```

Initialize

Termination conditional

Update loop variable

Blastoff re-implemented with a `for` loop

```
//  Writes 10, 9, 8 ..... Blastoff!!

#include <iostream>
using namespace std;

int main()
{
  for (int i = 10; i >= 1; i--) {
        cout << i << endl;
  }
  cout << "Blastoff!!" << endl;
  return 0;
}
```

All in one statement!

For loop: initialization

```
//  Writes 10, 9, 8 ..... Blastoff!!

#include <iostream>
using namespace std;

int main()
{
  for (int i = 10; i >= 1; i--) {
          cout << i <<
  }
  cout << "Blastoff!!" << endl;
  return 0;
}
```

Declare variable i and initialize

This clause runs once before the loop starts and never again.

For loop: conditional test

```
//  Writes 10, 9, 8 ..... Blastoff!!

#include <iostream>
using namespace std;

int main()
{
  for (int i = 10; i >= 1; i--) {
          cout << i << endl;
  }
  cout << "Blastoff!!" << endl;
  return 0;
}
```

Termination conditional

Just like the while loop: this is checked before each time around. If this is false, jump immediately to the end of the loop.

For loop: update

```
//  Writes 10, 9, 8 ..... Blastoff!!

#include <iostream>
using namespace std;

int main()
{
  for (int i = 10; i >= 1; i--) {
          cout << i << endl;
  }
  cout << "Blastoff!!" << endl;
  return 0;
}
```

Update loop variable:
i-- is subtracting 1 from i each time

This runs *after* each time around the loop body.

Blastoff re-implemented with a `for` loop

```
//  Writes 10, 9, 8 ..... Blastoff!!

#include <iostream>
using namespace std;

int main()
{
  for (int i = 10; i >= 1; i--) {
          cout << i << endl;
  }
  cout << "Blastoff!!" << endl;
  return 0;
}
```

A `for` loop like this is *defined* to do ***exactly*** the same as the `while` loop we studied earlier!!!

By The Way…
strings act a lot like arrays of characters

Two interesting features of strings

```
string s = "HELLO";
```

- `s[0]` is `'H'` ← of type char (not string!)
- `s[1]` is `'E'`
- `s[2]` is `'L'`
- `s[3]` is `'L'`
- `s[4]` is `'O'`

- `s.length()` ← returns the length of s in charcters (5)

Strings are not actually arrays, but you can subscript them as if they were

String demonstrations

```
string s = "HELLO";

// if you put .length() after a string variable you get
// the length of the string
cout << "The length of string " << s
     << " is " << s.length() << endl;

// You can access the characters in a string
// using array-like subscripting

char second_char = s[1];   // remember first index is 0
cout << "The second character in " << s
     << " is " << second_char << endl;

// And you can loop through all of them
cout << "All the characters in " << s << " are: ";
for (unsigned int i=0; i<s.length(); i++) {
  cout << s[i] << " ";    // print the next char and a blank
}
cout << endl;
return 0;
```

Prints:

```
The length of string HELLO is 5
```

```
// if you put .length() after a string variable you get
// the length of the string
cout << "The length of string " << s
     << " is " << s.length() << endl;

// You can access the characters in a string
// using array-like subscripting

char second_char = s[1];   // remember first index is 0
cout << "The second character in " << s
     << " is " << second_char << endl;

// And you can loop through all of them
cout << "All the characters in " << s << " are: ";
for (unsigned int i=0; i<s.length(); i++) {
  cout << s[i] << " ";    // print the next char and a blank
}
cout << endl;
return 0;
```

Prints:

```
The length of string HELLO is 5
The second character in HELLO is E
```

```
// if you put .length() after a string variable you get
// the length of the string
cout << "The length of string " << s
     << " is " << s.length() << endl;

// You can access the characters in a string
// using array-like subscripting

char second_char = s[1];   // remember first index is 0
cout << "The second character in " << s
     << " is " << second_char << endl;

// And you can loop through all of them
cout << "All the characters in " << s << " are: ";
for (unsigned int i=0; i<s.length(); i++) {
  cout << s[i] << " ";    // print the next char and a blank
}
cout << endl;
return 0;
```

Prints:

```
The length of string HELLO is 5
The second character in HELLO is E
All the charcters in HELLO are H E L L O
```

```
// if you put .length() after a string variable you get
// the length of the string
cout << "The length of string " << s
     << " is " << s.length() << endl;

// You can access the characters in a string
// using array-like subscripting

char second_char = s[1];   // remember first index is 0
cout << "The second character in " << s
     << " is " << second_char << endl;

// And you can loop through all of them
cout << "All the characters in " << s << " are: ";
for (unsigned int i=0; i < s.length(); i++) {
  cout << s[i] << " ";    // print the next char and a blank
}
cout << endl;
return 0;
```

Structs
The Big Picture

Highlights to watch for

- **Structs will allow us to *model* real-world & abstract things**
  - Real world: people, cars, baseball bats, books, windows (on a building), windows (on a computer screen)
  - Abstract: shapes, lists, election polls,
- **New: we will be defining our own new types!**

The power of defining your own types

```
int count;
float weight;
string family_name;
int lots_of_numbers[100];
```

The power of defining your own types

You define your own *variable names*

```
int count;
float weight;
string family_name;
int lots_of_numbers[100];
```

The power of defining your own types

Types like `int` are built into C++

You define your own *variable names*

```
int count;
float weight;
string family_name;
int lots_of_numbers[100];
```

The power of defining your own types

What if we could define our own types…

```
int count;
float weight;
string family_name;
int lots_of_numbers[100];

Car my_ford;
Car marks_mercedes;
Student bob;
Student cs11_students[270];
Lecture_Hall Pearson;
```

…and use them just like the built in ones?

Even make arrays of the new types.

The power of defining your own types

Note that most of our example types model real world objects (cars, people, etc.)!!

```
Car my_ford;
Car marks_mercedes;
Student bob;
Student cs11_students[270];
Lecture_Hall Pearson;
```

The power of defining your own types

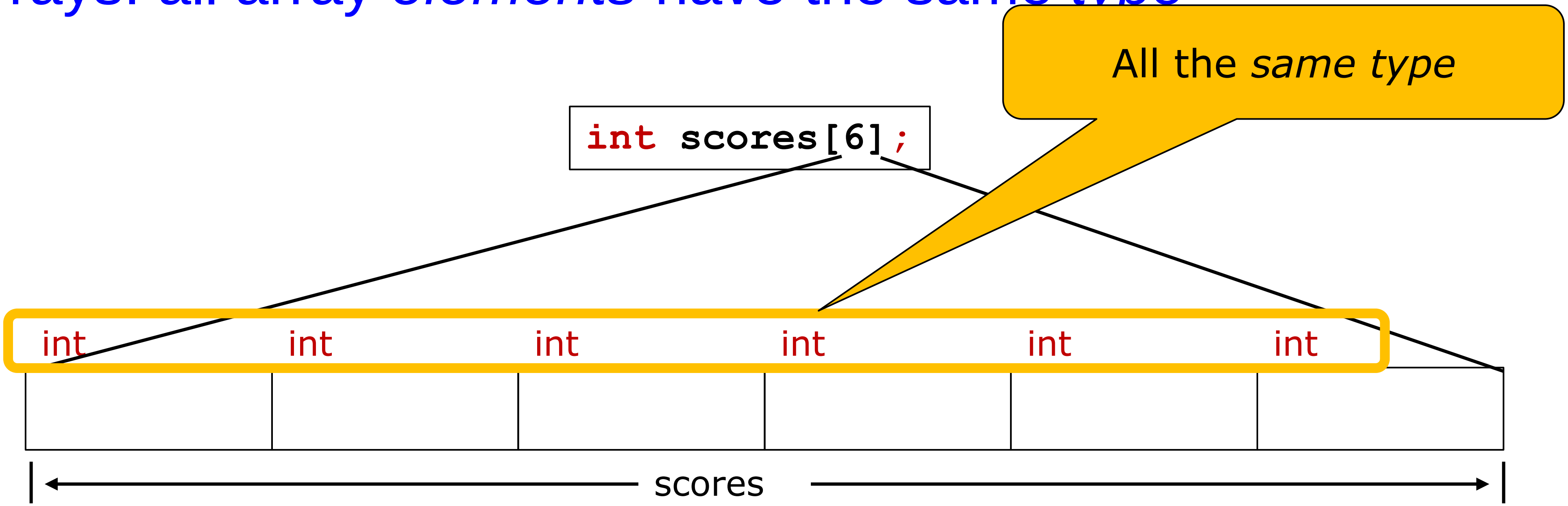
In a quite deep and interesting way, we have extended C++ to write programs about cars, students, and lecture halls!

```
Car my_ford;
Car marks_mercedes;
Student bob;
Student cs11_students[270];
Lecture_Hall Pearson;
```

Introducing C++ Structs

Why Not Just Use Arrays

Arrays: all array *elements* have the same *type*



Space reserved in the computer's memory for 6 integers, all of type "int"

Array element names are index numbers: `0` → `(n-1)`

```
int scores[6];
scores[1] =  3;
scores[5] = 12;
```

int	int	int	int	int	int
scores[0]	scores[1]	scores[2]	scores[3]	scores[4]	scores[5]

Space reserved in the computer's memory for the whole array!

Pros and cons of arrays

- **Array strengths**
  - Lots of data in one declaration
  - Indexing makes accessing easy
  - *You can use variables or expressions to compute the subscript at run time* (e.g. `array[i+2]`)
  - *Therefore, you can loop through array elements!*
- **Array limitations**
  - All elements have same type
  - No named elements
  - Passed by reference
  - Can't return an array from a function

Introducing structs – features to watch for

- **Structs have *named fields* (called members)**
- **Fields can have *different types***
- **You will *define your own,reusable, struct types* for the struct *as a whole***
- **You can *use that same struct type to declare lots of similar struct variables* (each of which has lots of fields!)**
- **You *cannot* do the equivalent of** `arr[i+2]`**, i.e. choosing a field at runtime**

Introducing C++ Structs

Basics of Defining and Using Structs

Structs: you *define your own structured type*

Until now, all the types we've used (e.g. int, float string) have been built into C++.

Here we define our own new type!

It's a struct named "Car_Part"

```
struct Car_Part {
          int part_number;
          string description;
          int quantity;
          float price;
};
```

```
int counter;            // declare an int named counter
Car_part horn;          // declare a Car_Part named horn
Car_part headlight;     // declare a Car_Part named headlight
```

Structs: you *define your own structured type*

Until now, all the types we've used (e.g. int, float string) have been built into C++.

Here we define our own new type!

It's a struct named "Car_Part"

```
struct Car_Part {
        int part_number;
        string description;
        int quantity;
        float price;
};
```

We can declare variables using our new type!

```
int counter;            // declare an int named counter
Car_part horn;          // declare a Car_Part named horn
Car_part headlight;     // declare a Car_Part named headlight
```

Structs: you *define your own structured type*

Until now, all the types we've used (e.g. int, float string) have been built into C++.

Here we define our own new type!

It's a struct named "Car_Part"

Each Car_Part has all of these fields!

```
struct Car_Part {
          int part_number;
          string description;
          int quantity;
          float price;
};
```

```
int counter;                   // declare an int named counter
Car_part horn;                 // declare a Car_Part named horn
Car_part headlight;            // declare a Car_Part named headlight
```

Another Big Leap In Abstraction

- With functions, we extended our machine to know how to do new types of operations
- With structs, we give the machine new abstractions for data types like Car Part
- Together.. the combination is powerful…e.g, later we will define functions that work on our new data types (e.g. print data about a Car_Part)

```
struct Car_Part {
        int part_number;
        string description;
        int quantity;
        float price;
};
```

```
int counter;            // declare an int named counter
Car_part horn;          // declare a Car_Part named horn
Car_part headlight;     // declare a Car_Part named headlight
```

Structs: you define your own structured type

When we actually declare the variable, we get space in memory for all fields

```
struct Car_Part {
        int part_number;
        string description;
        int quantity;
        float price;
};

Car_Part horn;
```

Defining the struct type declares what a Car_Part *will look like when we use it*

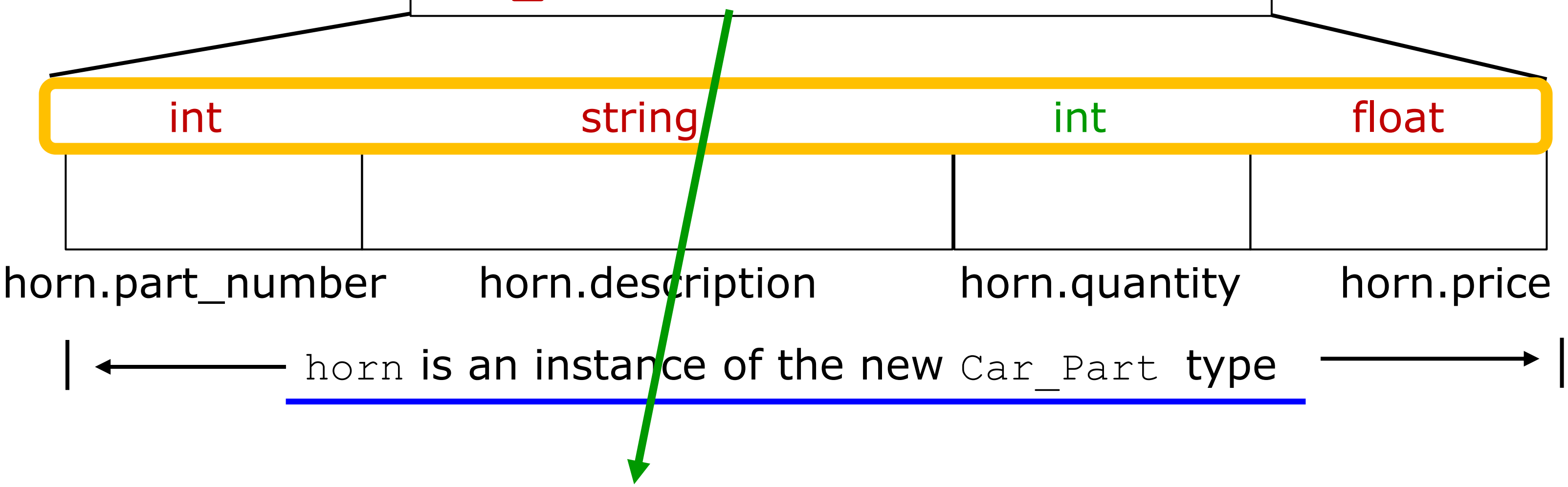
int	string	int	float
horn.part_number	horn.description	horn.quantity	horn.price

`horn` is an *instance* of the new `Car_Part` type

Referencing members of structs

```
struct Car_Part {
        int part_number;
        string description;
        int quantity;
        float price;
};

Car_Part horn;
```

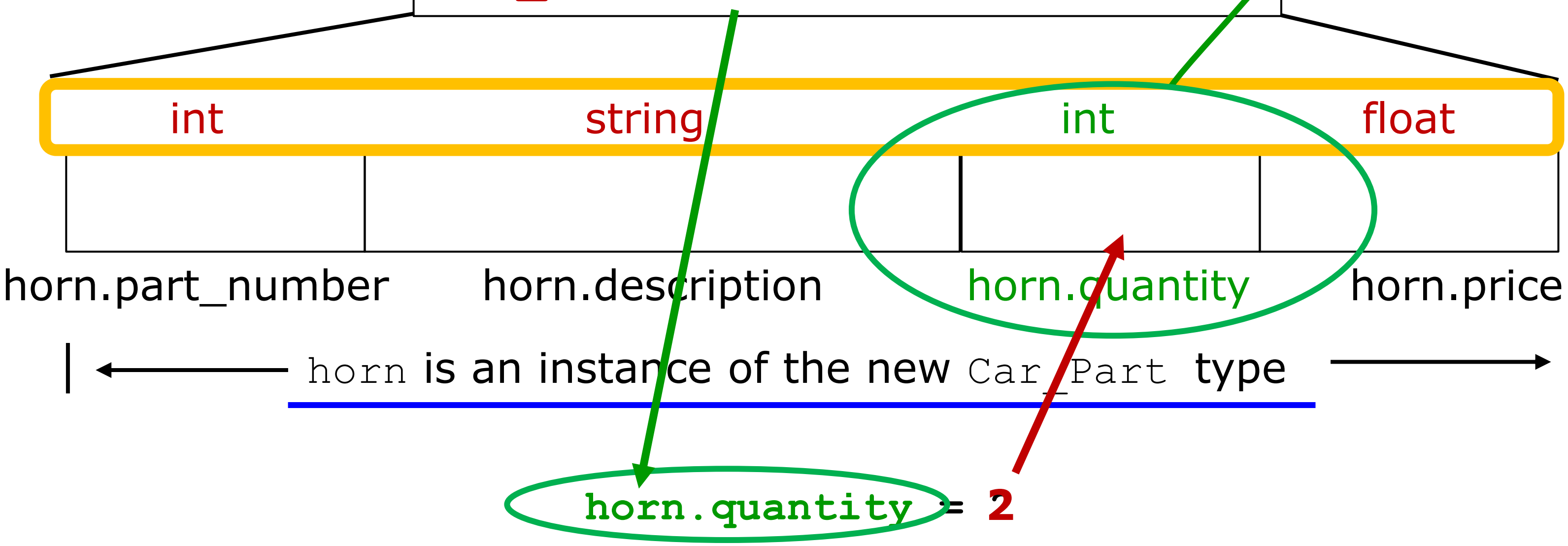



```
horn.quantity = 2
```

Referencing members of structs

```
struct Car_Part {
        int part_number;
        string description;
        int quantity;
        float price;
};

Car_Part horn;
```



Programming Interlude
Using the Same Struct Type
For Two Instance Variables

Using a struct in a program

```
struct Car_Part {
  int part_number;
  string description;
  int quantity;
  float price;
};

// Declare two variables...each is of type Car_Part
Car_Part tires;
Car_Part battery;

// Set the data members (variables) in the tires struct
tires.part_number = 34523;
tires.description = "Car tire";
tires.quantity = 4;
tires.price = 125.33;

// Set the data members (variables) in the battery struct
battery.part_number = 24563;
battery.description = "12 Volt Battery";
battery.quantity = 1;
battery.price = 147.20;

// Print the tires description, quantity and total price

cout << "Tires info: description=" << tires.description
     <<   " Quantity=" << tires.quantity
     << " total cost=$" << tires.quantity * tires.price
     << endl;
```

Using a struct in a program

```
struct Car_Part {
  int part_number;
  string description;
  int quantity;
  float price;
};

// Declare two variables...each is of type Car_Part
Car_Part tires;
Car_Part battery;

// Set the data members (variables) in the tires struct
tires.part_number = 34523;
tires.description = "Car tire";
tires.quantity = 4;
tires.price = 125.33;

// Set the data members (variables) in the battery struct
battery.part_number = 24563;
battery.description = "12 Volt Battery";
battery.quantity = 1;
battery.price = 147.20;

// Print the tires description, quantity and total price

cout << "Tires info: description=" << tires.description
     <<   " Quantity=" << tires.quantity
     << " total cost=$" << tires.quantity * tires.price
     << endl;
```

Using a struct in a program

```
struct Car_Part {
  int part_number;
  string description;
  int quantity;
  float price;
};

// Declare two variables...each is of type Car_Part
Car_Part tires;
Car_Part battery;

// Set the data members (variables) in the tires struct
tires.part_number = 34523;
tires.description = "Car tire";
tires.quantity = 4;
tires.price = 125.33;

// Set the data members (variables) in the battery struct
battery.part_number = 24563;
battery.description = "12 Volt Battery";
battery.quantity = 1;
battery.price = 147.20;

// Print the tires description, quantity and total price

cout << "Tires info: description=" << tires.description
     <<   " Quantity=" << tires.quantity
     << " total cost=$" << tires.quantity * tires.price
     << endl;
```

Using a struct in a program

```
struct Car_Part {
  int part_number;
  string description;
  int quantity;
  float price;
};

// Declare two variables...each is of type Car_Part
Car_Part tires;
Car_Part battery;

// Set the data members (variables) in the tires struct
tires.part_number = 34523;
tires.description = "Car tire";
tires.quantity = 4;
tires.price = 125.33;

// Set the data members (variables) in the battery struct
battery.part_number = 24563;
battery.description = "12 Volt Battery";
battery.quantity = 1;
battery.price = 147.20;

// Print the tires description, quantity and total price

cout << "Tires info: description=" << tires.description
     <<   " Quantity=" << tires.quantity
     << " total cost=$" << tires.quantity * tires.price
     << endl;
```

Using a struct in a program

```
struct Car_Part {
  int part_number;
  string description;
  int quantity;
  float price;
};

// Declare two variables...each is of type Car_Part
Car_Part tires;
Car_Part battery;

// Set the data members (variables) in the tires struct
tires.part_number = 34523;
tires.description = "Car tire";
tires.quantity = 4;
tires.price = 125.33;

// Set the data members (variables) in the battery struct
battery.part_number = 24563;
battery.description = "12 Volt Battery";
battery.quantity = 1;
battery.price = 147.20;

// Print the tires description, quantity and total price

cout << "Tires info: description=" << tires.description
     <<   " Quantity=" << tires.quantity
     << " total cost=$" << tires.quantity * tires.price
     << endl;
```

Using a struct in a program

```
struct Car_Part {
  int part_number;
  string description;
  int quantity;
  float price;
};

// Declare two variables...each is of type Car_Part
Car_Part tires;
Car_Part battery;

// Set the data members (variables) in the tires struct
tires.part_number = 34523;
tires.description = "Car tire";
tires.quantity = 4;
tires.price = 125.33;

// Set the data members (variables) in the battery struct
battery.part_number = 24563;
battery.description = "12 Volt Battery";
battery.quantity = 1;
battery.price = 147.20;

// Print the tires description, quantity and total price

cout << "Tires info: description=" << tires.description
     <<   " Quantity=" << tires.quantity
     << " total cost=$" << tires.quantity * tires.price
     << endl;
```

```
Prints:
Tires info: description=Car tire
Quantity=4 total cost=$501.32
```

Initializing Struct Values

Initializers for structs

Review: initializing an int

```
int max_lines = 30;
```

Review: initializing structs

```
struct Car_Part {
        int part_number;
        string description;
        int quantity;
        float price;
};

 // Declare and initialize two variables...each is of type Car_Part
 Car_Part tires = {34523, "Car tire", 4, 125.33};
 Car_Part battery = {24653, "12 Volt Battery", 1, 147.20};
```

Initializers for structs

Review: initializing an int

```
int max_lines = 30;
```

Review: initializing structs

```
struct Car_Part {
        int part_number;
        string description;
        int quantity;
        float price;
};

 // Declare and initialize two variables.. each is of type Car_Pa
 Car_Part tires = {34523, "Car tire", 4, 125.33};
 Car_Part battery = {24653, "12 Volt Battery", 1, 147.20};
```

Initializer sets same values as these assignment statements…

```
// Set the data members (variables) in the tires struct
tires.part_number = 34523;
tires.description = "Car tire";
tires.quantity = 4;
tires.price = 125.33;
```

Arrays of Structs

Review: declaring and initializing arrays

```
int arr[5];                          // Five integers

int arr[5] = {3, 6, 9, 12, 15};      // initialized
```

We are about to do the same thing for structs:

```
Car_Part all_parts[5];               // Five Car_Parts

Car_Part all_parts[5] = {...see next slide...};
                                     // initialized
```

Arrays of structs – a classic combination!

```
// Assume struct Car_Part same as before
const int NUM_PARTS = 4;

float total_cost(Car_Part cp) {
        return cp.quantity * cp.price;
}

void print_part(Car_Part cp)
{
        cout << "Car Part: Description=" << cp.description <<
                " Quantity=" << cp.quantity
             << " total cost=$" << total_cost(cp) << endl;
}
int main()
{

        // Declare an array of parts
        // Each entry in the array is of type Car_Part
        Car_Part all_parts[NUM_PARTS] = {
                {34523, "Car tire", 4, 99.95},
                {24653, "12 Volt Battery", 1, 147.20},
                {3412, "Lug nuts", 24, 2.25},
                {98765, "Fuzzy dice", 1, 12.50},
        };

        // Use a loop to print all the parts
        for (int i = 0; i < NUM_PARTS; i++) {
                print_part(all_parts[i]);
        }

        return 0;
}
```

Arrays of structs – a classic combination!

```
// Assume struct Car_Part same as before
const int NUM_PARTS = 4;

float total_cost(Car_Part cp) {
        return cp.quantity * cp.price;
}

void print_part(Car_Part cp)
{
        cout << "Car Part: Description=" << cp.
                " Quantity=" << cp.quantity
             << " total cost=$" << total_cost(c
}
int main()
{

        // Declare an array of parts
        // Each entry in the array is of type Car_Part
        Car_Part all_parts[NUM_PARTS] = {
                {34523, "Car tire", 4, 99.95},
                {24653, "12 Volt Battery", 1, 147.20},
                {3412, "Lug nuts", 24, 2.25},
                {98765, "Fuzzy dice", 1, 12.50},
        };

        // Use a loop to print all the parts
        for (int i = 0; i < NUM_PARTS; i++) {
                print_part(all_parts[i]);
        }

        return 0;
}
```

Declare and initialize an array of four items…each is a Car_Part.

Arrays of structs – a classic combination!

```
// Assume struct Car_Part same as before
const int NUM_PARTS = 4;

float total_cost(Car_Part cp) {
        return cp.quantity * cp.price;
}

void print_part(Car_Part cp)
{
        cout << "Car Part: Description=" << cp.description <<
                " Quantity=" << cp.quantity
             << " total cost=$" << total_cost(cp) << endl;
}
int main()
{

        // Declare an array of parts
        // Each entry in the array is of type Car_Part
        Car_Part all_parts[NUM_PARTS] = {
                {34523, "Car tire", 4, 99.95},
                {24653, "12 Volt Battery", 1, 147.20},
                {3412, "Lug nuts", 24, 2.25},
                {98765, "Fuzzy dice", 1, 12.50},
        };

        // Use a loop to print all the parts
        for (int i = 0; i < NUM_PARTS; i++) {
                print_part(all_parts[i]);
        }

        return 0;
}
```

Loop through all the `all_parts` array…

…same as for any other array!

Arrays of structs – a classic combination!

```
// Assume struct Car_Part same as before
const int NUM_PARTS = 4;

float total_cost(Car_Part cp) {
        return cp.quantity * cp.price;
}

void print_part(Car_Part cp)
{
        cout << "Car Part: Description=" << cp.description <<
                " Quantity=" << cp.quantity
             << " total cost=$" << total_cost(cp) << endl;
}
int main()
{

        // Declare an array of parts
        // Each entry in the array is of type Ca
        Car_Part all_parts[NUM_PARTS] = {
                {34523, "Car tire", 4, 99.95},
                {24653, "12 Volt Battery", 1, 14
                {3412, "Lug nuts", 24, 2.25},
                {98765, "Fuzzy dice", 1, 12.50}
        };

        // Use a loop to print all the parts
        for (int i = 0; i < NUM_PARTS; i++) {
                print_part(all_parts[i]);
        }

        return 0;
}
```

For each part in the `all_parts` array…

…call print_part.

Prints:

```
Car Part: Description=Car tire Quantity=4 total cost=$399.8
Car Part: Description=12 Volt Battery Quantity=1 total cost=$147.2
Car Part: Description=Lug nuts Quantity=24 total cost=$54
Car Part: Description=Fuzzy dice Quantity=1 total cost=$12.5
```

```
void print_part(Car_Part cp)
{
        cout << "Car Part: Description=" << cp.description <<
                " Quantity=" << cp.quantity
             << " total cost=$" << total_cost(cp) << endl;
}
int main()
{

        // Declare an array of parts
        // Each entry in the array is of type Car_Part
        Car_Part all_parts[NUM_PARTS] = {
                {34523, "Car tire", 4, 99.95},
                {24653, "12 Volt Battery", 1, 147.20},
                {3412, "Lug nuts", 24, 2.25},
                {98765, "Fuzzy dice", 1, 12.50},
        };

        // Use a loop to print all the parts
        for (int i = 0; i < NUM_PARTS; i++) {
                print_part(all_parts[i]);
        }

        return 0;
}
```

Functions Returning Structs

Function that returns a struct

Function returns *copy* of Car_Part with highest total cost

```
//              most_expensive
//
//   Returns a copy of the data for the part with highest total co
//
//   Note that return type of the function is Car_P
//   (yes, you can return a struct!)
//
//   Also: this function ass          _PARTS >= 1. A better implementation
//   would check
//

Car_Part most_expensive(const int nparts, Car_Part parts_to_search[])
{
        Car_Part highest_so_far = parts_to_search[0]; // start with first one

        // If any of the others is more expensive, switch to that one
        // Note that loop starts with i==1, because parts_to_search[0]
        // was already handled above.  (Most loops through arrays start with
        // i=0.)
        for (int i = 1; i < nparts; i++)
        {
                if (total_cost(parts_to_search[i]) > total_cost(highest_so_far)) {
                        highest_so_far = parts_to_search[i];
                }
        }

        // We've searched them all, so highest_so_far is now the highest of all
        return highest_so_far;
}
```

Function that returns a struct

Prints:

```
MOST EXPENSIVE
--------------
Car Part: Description=Car tire Quantity=4 total cost=$399.8
```

```
Car_Part most_expensive(const int nparts, Car_Part parts_to_search[])
{
        Car_Part highest_so_far = parts_to_search[0]; // start with first one

        // If any of the others is more expensive, switch to that
        // Note that loop starts with i==1, because parts_to_sear
        // was already handled above.  (Most loops through arrays
        // i=0.)
        for (int i = 1; i < nparts; i++)
        {
                if (total_cost(parts_to_search[i]) > total_cost(h
                        highest_so_far = parts_to_search[i];
                }
        }

        // We've searched them all, so highest_so_far is n   e highest of all
        return highest_so_far;
}
        cout << "MOST EXPENSIVE" << endl << "--------------" << endl;
        print_part(most_expensive(NUM_PARTS, all_parts));
```

Invoke `most_expensive`…

…pass that part to `print_part`

Function that returns a struct

```
//              most_expensive
//
//    Returns a copy of the data for the part with highest total_cost
//
//    Note that return type of the function is Car_Part
//    (yes, you can return a struct!)
//
//    Also: this function assumes NUM_PARTS >= 1. A better implementation
//    would check
//

Car_Part most_expensive(const int nparts, Car_Part parts_to_search[])
{
        Car_Part highest_so_far = parts_to_search[0]; // start with first one

        // If any of the others is more expensive, switch to that one
        // Note that loop starts with i==1, because parts_to_search[0]
        // was already handled above.  (Most loops through arrays start with
        // i=0.)
        for (int i = 1; i < nparts; i++)
        {
                if (total_cost(parts_to_search[i]) > total_cost(highest_so_far)) {
                        highest_so_far = parts_to_search[i];
                }
        }

        // We've searched them all, so highest_so_far is now the highest of all
        return highest_so_far;
}
```

```
cout << "MOST EXPENSIVE" << endl << "--------------" << endl;
print_part(most_expensive(NUM_PARTS, all_parts));
```

Prints:

```
Car Part: Description=Car tire Quantity=4 total cost=$399.8
Car Part: Description=12 Volt Battery Quantity=1 total cost=$147.2
Car Part: Description=Lug nuts Quantity=24 total cost=$54
Car Part: Description=Fuzzy dice Quantity=1 total cost=$12.5
```

```
void print_part(Car_Part cp)
{
        cout << "Car Part: Description=" << cp.description <<
                " Quantity=" << cp.quantity
             << " total cost=$" << total_cost(cp) << endl;
}
int main()
{

        // Declare an array of parts
        // Each entry in the array is of type Car_Part
        Car_Part all_parts[NUM_PARTS] = {
                {34523, "Car tire", 4, 99.95},
                {24653, "12 Volt Battery", 1, 147.20},
                {3412, "Lug nuts", 24, 2.25},
                {98765, "Fuzzy dice", 1, 12.50},
        };

        // Use a loop to print all the parts
        for (int i = 0; i < NUM_PARTS; i++) {
                print_part(all_parts[i]);
        }

        return 0;
}
```

Comparing Arrays and Structs

Key features compared

	Arrays	Structs
Member types	All the same	Can be different
Addressing	Numeric subscript: `arr[3]`	Dot notation: `student.name`
Computed names	Yes: `arr[i+2]`	No
Loop through members	Yes: `while(i-- > 0) cout << arr[i];`	No (but you can make arrays of structs, and loop through those!)
Pass as function argument	Yes, by reference	Yes, by value (copied)
Return from function	No	Yes, by value (copied)
Defines new named type	No*	Yes (e.g. `struct Car_Part`)

* There are ways to name array types, but for now the answer is "No"

You can combine arrays and structs

- **Arrays within structs:**

```
// declare new type: struct Parent
struct Person {
  string mothers_name;
  int ages_of_children[10]; // Array within struct
}
```

You can combine arrays and structs

- **Arrays within structs:**

```
// declare new type: struct Parent
struct Person {
  string mothers_name;
  int ages_of_children[10]; // Array within struct
}
Person bob;          // One variable of type struct Person
bob.mothers_name = "Mary";
bob.ages_of_children[0] = 8; //first child's age
bob.ages_of_children[1] = 3; //second child's age
```

- **Arrays of structs (very common):**

```
Person lots_of_people[100]; // array of structs
// Fill in some data for 3rd person in lots_of_people array
lots_of_people[2].mothers_name = "Mary";
lots_of_people[2].ages_of_children[0] = 8; age
lots_of_people[2].ages_of_children[1] = 3; child's age
```

Wrapup

Summary

- **We've now learned about two types for collections of data:**
  - Arrays: all elements have the same type, integer indices
  - Structs: named elements, each with different types
- **We have learned to combine them to make very powerful collections**
- **We have used functions with struct type arguments and you can return structs too**
- ***Structs extend our programming languages to directly model real world abstractions like people, business records, car parts, etc.***