



# The Zen of C++

Chapter 10: Abstract Data Types

Adnan Zejnilovic

# Introduction

- Primitive data types can become too limiting to limiting in modeling real-world problems
- An example of this would be using parallel arrays to model an entity such as person, a student, an employee, a car, etc.
- To better model real-world entities the C++ language provides a mechanism for programmers to create their own datatypes
- Programmer defined data types are commonly referred to as an abstract types(ADT)
- In this chapter we're going to introduce structures
- Structures act as containers for primitive data types in the used to model real-world entities

# Introduction

Structure consists of:

- keyword "struct"
- a tag(new data type),
- open "{" brace
- (mostly) primitive data types
- closed "}" brace followed by
- A semicolon ";"

```
struct CarInfo {  
    string make;  
    string model;  
    int yearMade;  
    double mpg;  
};
```

The diagram illustrates the components of a C++ struct definition. A red box labeled "New Data Type" points to the tag "CarInfo". A red box labeled "Keyword" points to the keyword "struct". A blue bracket groups the member declarations: "string make;", "string model;", "int yearMade;", and "double mpg;". Blue arrows point from the list items to the corresponding parts of the code: "keyword 'struct'" points to "struct", "a tag(new data type)" points to "CarInfo", "open '{' brace" points to the opening brace, "(mostly) primitive data types" points to the member declarations, "closed '}' brace followed by" points to the closing brace, and "A semicolon ';'" points to the semicolon.

# When Does the Memory Get Allocated?

- When a struct is declared, it is important to note that no memory gets allocated at this point
- Memory gets allocated when a variable of this new data type gets instantiated:

```
CarInfo myCar; // at this point memory
                // gets allocated for each
                // member of the struct
```

# Allocating a Variable

The process of allocating a variable of an abstract data type is same as the process for allocating a variable of a primitive data type:

`<data-type> variable name;`

Data Type	Variable Name
int	x;
double	y;
CarInfo	myCar;

# Initialization

- A structure cannot be initialized while it is being declared
- This is due to the fact that no memory is allocated when the structure is declared
- Once a variable of the structure data type is declared, it is possible to initialize individual structure members
- This is accomplished through the dot (.) operator:

```
CarInfo myCar;  
myCar.mpg = 20.54;
```

# How it Works

- No memory has been allocated at this point
- Memory gets allocated when a variable of data type CarInfo is instantiated
  - *Note: in this example, memory for two string objects (make, model), one integer (yearMade) and one double (mpg) is allocated*
- Now it is possible to initialize individual struct members with data

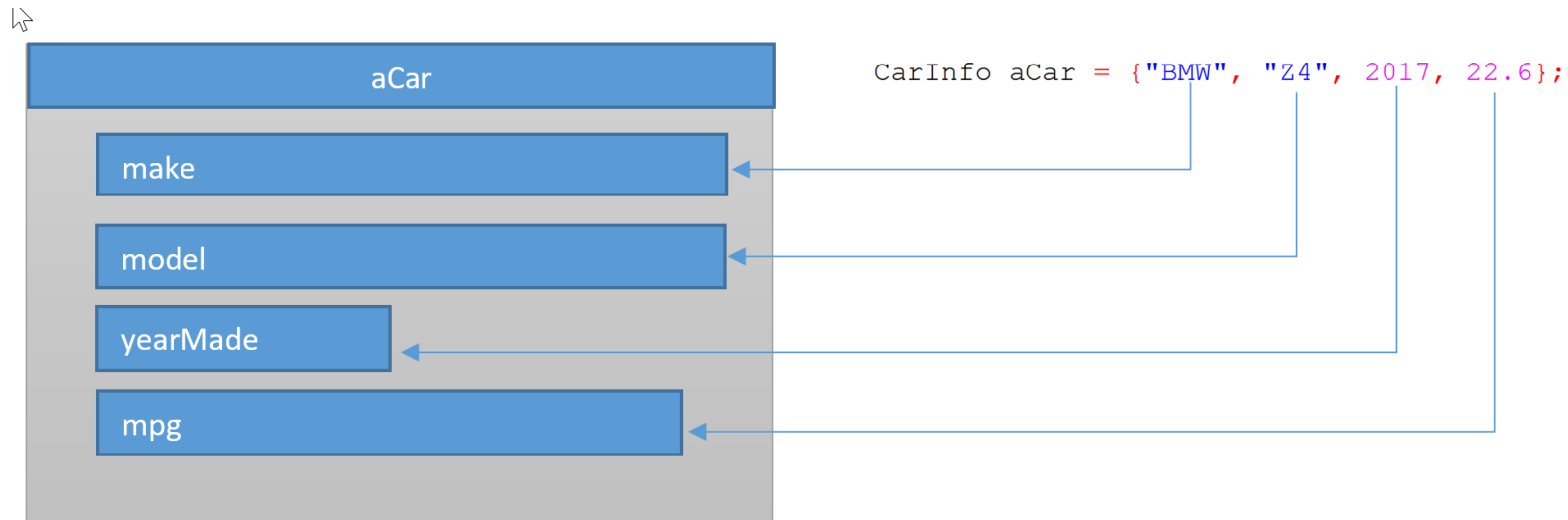
```
struct CarInfo
{
    string make;
    string model;
    int yearMade;
    double mpg;
};

int main()
{
    CarInfo myCar;
    myCar.make = "BMW";
    myCar.model = "M4";
    myCar.yearMade = 2017;
    myCar.mpg = 20.72;

    return 0;
}
```

I

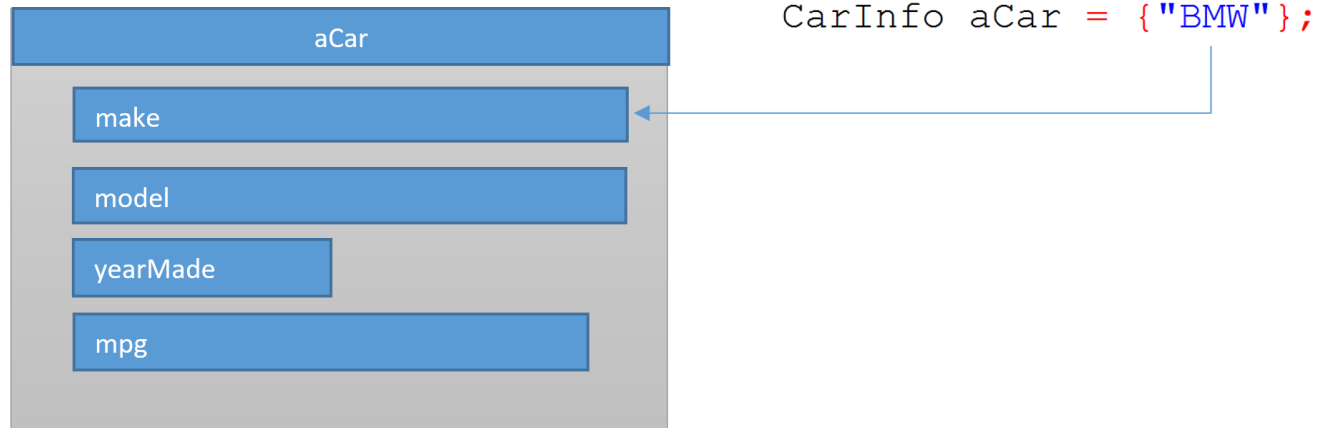
# List Initialization



- C++ provides a convenient way to initialize members of a structure through a initialization list
- The first element in the list is assigned to the first member of the structure, second element in the list is assigned to the second member of the structure and so on
- This is very similar to the mechanism used to list initialize an array

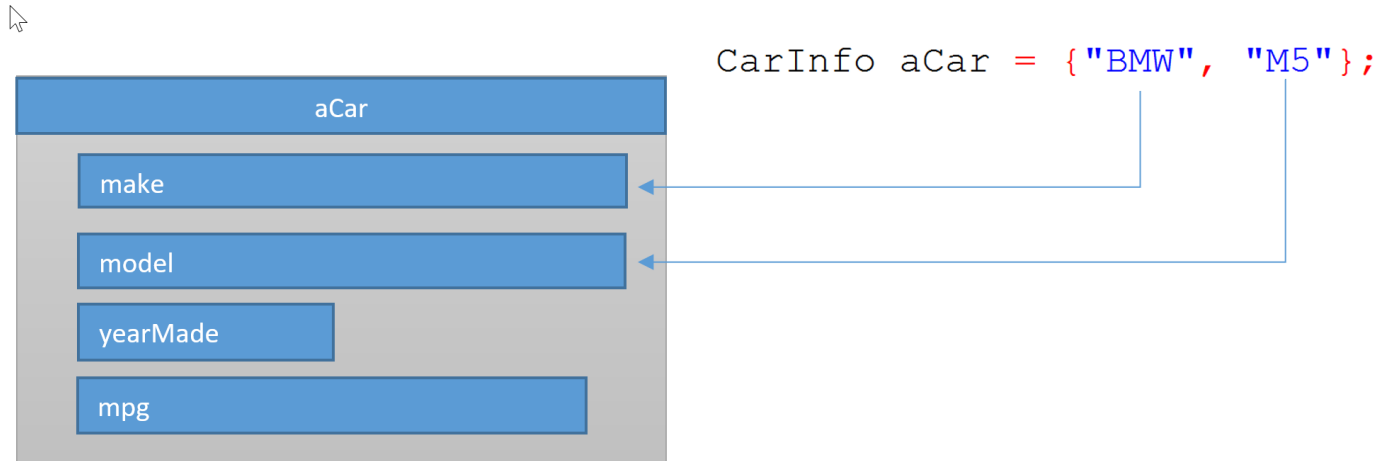


# Partial List Initialization



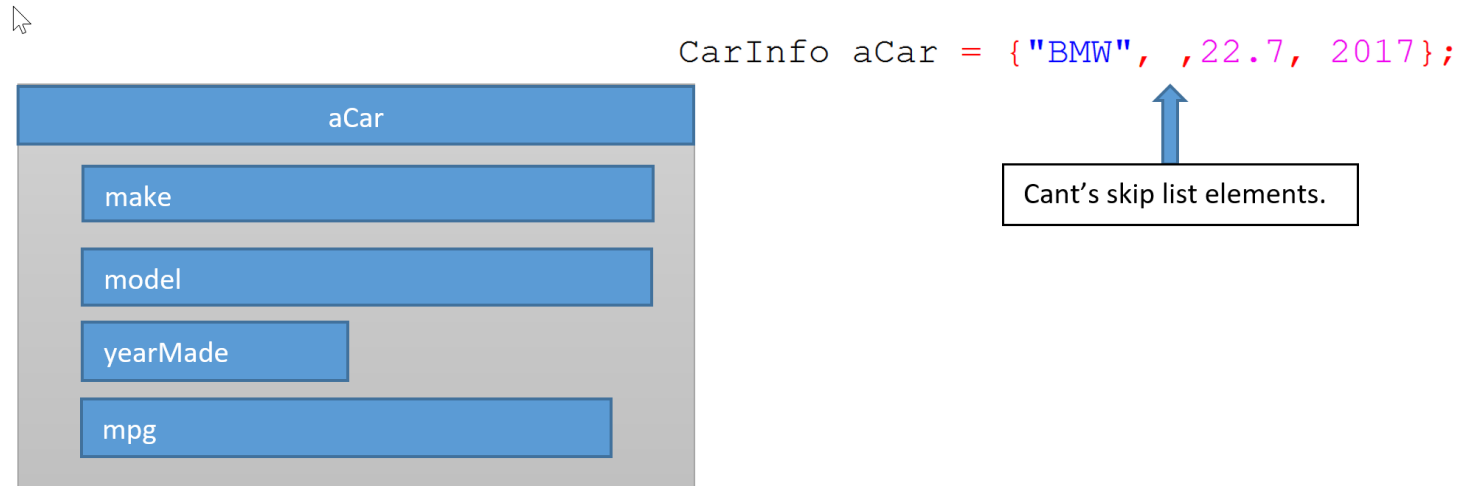
- It is possible to partially initialize structure members using list initialization
- In this example only the first member of the structure (`make`) was initialized with a string value from the list (`"BMW"`)

# Partial List Initialization



- It is possible to partially initialize structure members using list initialization
- In this example, the first member of the structure (`make`), and the second member of the structure (`model`) were initialized with a string values from the list `"BMW"`, `"M5"` respectively

# Partial List Initialization



- Skipping over list elements would result in a syntax error

# Accessing Structure Members

- The C++ dot (".") operator is a simple mechanism.
- You can think of a structure variable is the common variable (the “root”) for the entire container (structure)
- To access individual members of the structure it is necessary to first reference the “root” followed by the dot (".") operator followed by the member variable
- In this example, aCar is the structure variable, and we are accessing the member variable model through the dot (".") operator

```
struct CarInfo
{
    string make;
    string model;
    int    yearMade;
    double mpg;
};

int main()
{
    CarInfo aCar = {"BMW", "M5", 22.7, 2017};

    // accessing a struct member through the dot operator
    cout << "The model of the car is: " << aCar.model << endl;

    return 0;
}
```

# Math Operations on Structure Members

- Any method operations that can be performed on primitive data types can be performed the structure members

```
struct Person
{
    string name;
    int weight; // in lbs
    int height; // in inches
};

int main()
{
    Person student = {"Joe", 230, 75};

    // accessing a struct member through the dot operator
    cout << "Before the he went on a diet, "
         << student.name
         << " was : "
         << student.weight
         << " pounds."
         << endl;

    // Joe lost 10 lbs on the diet
    student.weight = student.weight - 10;

    cout << student.name << " now weighs : " << student.weight << endl;

    return 0;
}
```

Math operation



The output of the program run:

```
Before the he went on a diet, Joe was :230 pounds.
Joe now weighs : 220

Process returned 0 (0x0)   execution time : 0.138 s
Press any key to continue.
```

# Comparing Structures

- Structures have to be compared on a member to member basis

```
1 #include <iostream>
2
3 using namespace std;
4
5 struct Person
6 {
7     string name;
8     int weight; // in lbs
9     int height; // in inches
10 };
11
12 int main()
13 {
14     Person studentA = {"Joe", 230, 75};
15     Person studentB = {"Joe", 185, 68};
16
17     if (studentA == studentB)
18     {
19         cout << "They are equal" << endl;
20     }
21     return 0;
22 }
```

- When comparing the two struct variables, the program will not compile

Logs & others

Code::Blocks Search results Cccc Build log Build messages CppCheck CppCheck messages

File L.. Message

C:\Use... In function 'int main()':

C:\Use... 17 error: no match for 'operator==' (operand typ...

C:\Use... 17 note: candidates are:

C:\Pro... 2.. note: template<class \_StateT> bool std::opera...

C:\Pro... 2.. note: template argument deduction/substitut...

C:\Use... 17 note: 'Person' is not derived from 'const s...

C:\Pro... 2.. note: template<class \_T1, class \_T2> bool std...

C:\Pro... 2.. note: template argument deduction/substitut...

C:\Use... 17 note: 'Person' is not derived from 'const s...

C:\Pro... 2.. note: template<class \_Iterator> bool std::ope...

C:\Pro... 2.. note: template argument deduction/substitut...

# Comparing Structures

- Structures have to be compared on a member to member basis

```
#include <iostream>

using namespace std;

struct Person
{
    string name;
    int weight; // in lbs
    int height; // in inches
};

int main()
{
    Person studentA = {"Joe", 230, 75};
    Person studentB = {"Joe", 185, 68};

    if (studentA.name == studentB.name)
    {
        cout << "They have the same name." << endl;
    }
    return 0;
}
```

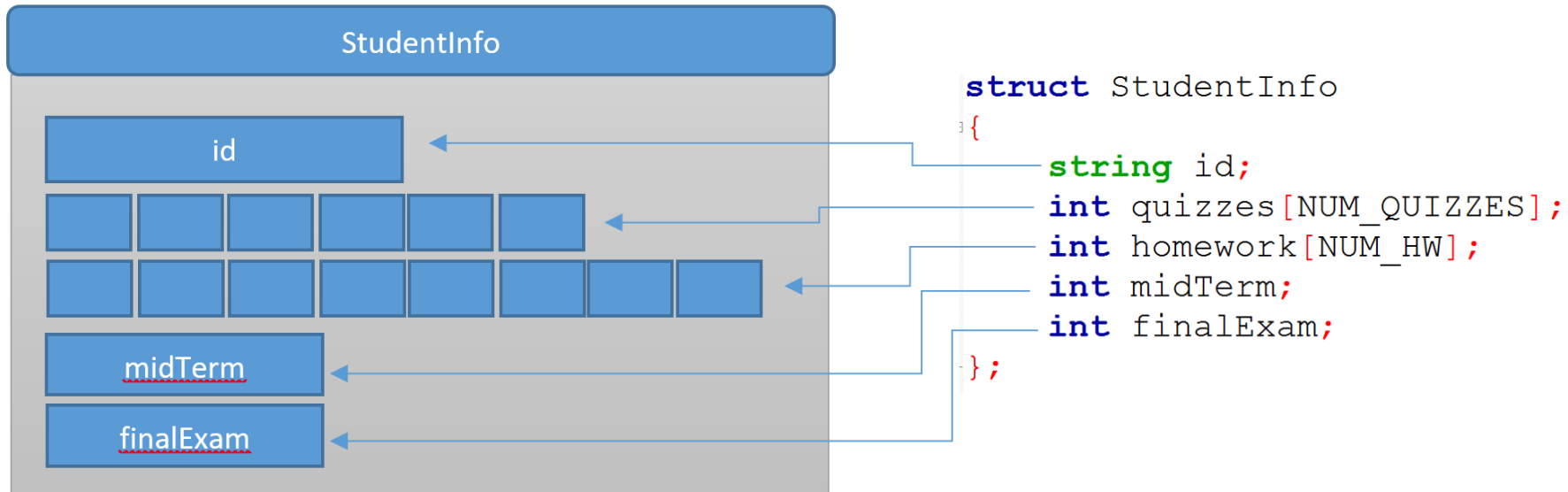
- Member wise comparison
- 

Program output:

```
They have the same name.
```

```
Process returned 0 (0x0)   execution time : 0.118 s
Press any key to continue.
```

# Arrays as Structure Members





# Nested Structures

```
struct CarInfo
{
    string make;
    string model;
    int yearMade;
    double mpg;
    bool manual;
};

struct Address
{
    int number;
    string streetName;
    string city;
    string state;
    int zipCode;
};

struct StudentInfo
{
    string name;
    string major;
    Address homeAddress;
    Address businessAddress;
    CarInfo car;
};
```


- You can nest a structure inside of a structure
- Notice the data type of homeAddress, businessAddress?
- Also, the data type of car?

Use the dit operator to access members of nested structs:

```
StudentInfo myStudent;
myStudent.name = "Eddie Murphy";
myStudent.major = "Psychology";
myStudent.homeAddress.number = 123;
myStudent.homeAddress.streetName = "Comedy street";
myStudent.homeAddress.city = "New York";
myStudent.homeAddress.state = "NY";
myStudent.homeAddress.zipCode = 11123;
myStudent.businessAddress.number = 5000;
myStudent.businessAddress.streetName = "Wilshire Blvd";
myStudent.businessAddress.city = "Los Angeles";
myStudent.businessAddress.state = "CA";
myStudent.businessAddress.zipCode = 90210;
myStudent.car.make = "Aston Martin";
myStudent.car.model = "DB Vantage";
myStudent.car.yearMade = 2017;
```

# Arrays of Structures

```
int    arrayName[10];  
CarInfo myCars[200];
```



- It is possible to declare an array of structs
- The mechanism is the same as for the primitive data types:
  - <data type> arrayName[DIMENSION];
- To access an individual record (struct), you need to use the index:

```
for (int i=0; i<3; i++)  
{  
    cout << "Make: " << myCars[i].make << endl;  
    cout << "Model: " << myCars[i].model << endl;  
    cout << "Year Made: " << myCars[i].yearMade << endl;  
    cout << "MPG: " << myCars[i].mpg << endl;  
}
```

# Arrays of Structures

- Without structures you way to combine attributes of different data type than describe an entity was through the use of parallel arrays
- Parallel arrays are a bit cumbersome because each array can store only elements of the same data type
- Thus, if the entity being modeled has a lot of attributes, that would mean a lot of arrays
- Structures on the other hand are excellent tool to model a record (employee, car, student, etc.)
- If there's a collection of records such as classroom full of students then an array of structures lends itself is a good option

# Initializing an Array of Structures via Initialization List

```
struct CarInfo
{
    string make;
    string model;
    int    yearMade;
    double mpg;
};

int main()
{
    CarInfo myCars[3] = { {"BMW", "M5", 2017, 15.6},
                          {"Acura", "NSX", 2017, 11.2},
                          {"Mini", "Countryman", 2015, 34.2}
    };
}
```

- There is an easy way to quickly initialize an array of structures to an initialization list
- The syntax is similar to the initialization list for an array of primitive data types
- The only exception is that the individual structures inside the initialization list are enclosed in their own sets of braces and separated by commas
- The last structure in the initialization list does not end with a comma

# Passing Structures to Functions

- Is a programmer you will need to make design decisions that may impact the performance of your program
- Structures can be passed to functions either by value, by reference, or as a pointer

# Pass by Value

- When passing a structure by value to a function, the contents of the argument variable are copied to the parameter (local) variable in the function
- In this example, the contents of **mySalsa** (argument) were copied to **aSalsa** (parameter)
- The structure in this example is rather small so making a copy of it (passing it by value) will not affect program performance that much
- However, in some cases if you repetitively pass by value then the program performance may be affected by all copying
- In this case, passing by reference or as a pointer would be a better choice

```
#include <iostream>

using namespace std;

struct SalsaInfo
{
    string name;
    double price;
    int numJars;
};

void displaySalsaInfo(SalsaInfo );

int main()
{
    SalsaInfo mySalsa;

    // populate the record with data
    mySalsa.name = "MILD";
    mySalsa.price = 5.99;
    mySalsa.numJars = 16;

    displaySalsaInfo(mySalsa);

    return 0;
}

void displaySalsaInfo(SalsaInfo aSalsa)
{
    cout << "Name: " << aSalsa.name << endl;
    cout << "Price: " << aSalsa.price << endl;
    cout << "Num Jars: " << aSalsa.numJars << endl;
}
```

# Pass by Reference

- Passing by reference is faster because there is no copying of the data involved
- In essence, you pass the address of the data
- Should you always pass my reference?

```
#include <iostream>

using namespace std;

struct SalsaInfo
{
    string name;
    double price;
    int    numJars;
};

void displaySalsaInfo(SalsaInfo &);

int main()
{
    SalsaInfo mySalsa;

    // populate the record
    mySalsa.name = "MILD";
    mySalsa.price = 5.99;
    mySalsa.numJars = 16;

    displaySalsaInfo(mySalsa);

    return 0;
}

void displaySalsaInfo(SalsaInfo &aSalsa)
{
    cout << "Name: " << aSalsa.name << endl;
    cout << "Price: " << aSalsa.price << endl;
    cout << "Num Jars: " << aSalsa.numJars << endl;
}
```

# Pass by Reference – Possible Problems

```
#include <iostream>

using namespace std;

struct SalsaInfo
{
    string name;
    double price;
    int numJars;
};

void displaySalsaInfo(SalsaInfo &);

int main()
{
    SalsaInfo mySalsa;

    // populate the record
    mySalsa.name = "MILD";
    mySalsa.price = 5.99;
    mySalsa.numJars = 16;

    displaySalsaInfo(mySalsa);

    return 0;
}

void displaySalsaInfo(SalsaInfo &aSalsa)
{
    aSalsa.name = "ZESTY";
    cout << "Name: " << aSalsa.name << endl;
    cout << "Price: " << aSalsa.price << endl;
    cout << "Num Jars: " << aSalsa.numJars << endl;
}
```

```
Name: ZESTY
Price: 5.99
Num Jars: 16
```

```
Process returned 0 (0x0)   execution time : 0.165 s
Press any key to continue.
```

- Passing by reference is faster than passing by value, however it is not without possible pitfalls
- In this example, a line of code in the displaySalsaInfo function can alter the data
- Why?
- Recall that when passing by reference you are working on the original value as there is no copy
- So is there a better choice?



# Pass by Constant Reference

```
#include <iostream>

using namespace std;

struct SalsaInfo
{
    string name;
    double price;
    int numJars;
};

void displaySalsaInfo(const SalsaInfo &);

int main()
{
    SalsaInfo mySalsa;

    // populate the record
    mySalsa.name = "MILD";
    mySalsa.price = 5.99;
    mySalsa.numJars = 16;

    displaySalsaInfo(mySalsa);

    return 0;
}

void displaySalsaInfo(const SalsaInfo &aSalsa)
{
    cout << "Name: " << aSalsa.name << endl;
    cout << "Price: " << aSalsa.price << endl;
    cout << "Num Jars: " << aSalsa.numJars << endl;
}
```

- Passing by reference avoidance unnecessary coping
- Passing by reference is dangerous so if the function does not need to change the data then the data should always be passed by constant reference
- In this example, if you try to change any members of the aSalsa struct, the program will result in a syntax error
- If you want the speed of passing by reference and also “safety” of your data, then pass by constant reference
- Passing by constant reference is indeed “Best of Both Worlds”

# Returning Structures from Functions

- When writing modular code capturing data from the user should be isolated to function
- In this example, we declare a local structure **myCar** inside the **getCarInfo** function, populate it with user data and return it to the caller
- Then in the caller (main), we set it to the local variable in main:

```
myCar = getCarInfo();
```

```
#include <iostream>

using namespace std;

struct CarInfo
{
    string make;
    string model;
    int yearMade;
    double mpg;
};

CarInfo getCarInfo();
void displayCarInfo(const CarInfo &);

int main()
{
    CarInfo myCar;
    myCar = getCarInfo();
    displayCarInfo(myCar);
    return 0;
}

CarInfo getCarInfo()
{
    CarInfo myCar;
    cout << "Make: ";
    getline(cin, myCar.make);

    cout << "Model: ";
    getline(cin, myCar.model);

    cout << "Year Made: ";
    cin >> myCar.yearMade;

    cout << "MPG: ";
    cin >> myCar.mpg;

    return myCar;
}

void displayCarInfo(const CarInfo &aCar)
{
    cout << "Make: " << aCar.make << endl;
    cout << "Model: " << aCar.model << endl;
    cout << "Year Made: " << aCar.yearMade << endl;
    cout << "MPG: " << aCar.mpg << endl;
}
```

# Pointers to Structures

```
#include <iostream>

using namespace std;

struct CarInfo
{
    string make;
    string model;
    int yearMade;
    double mpg;
};

int main()
{
    CarInfo aCar;
    CarInfo* ptrCar; // pointer capable of pointing
                    // to CarInfo data type

    ptrCar = &aCar;

    // populate the structure through the pointer
    ptrCar->make = "BMW";
    ptrCar->model = "M4";
    ptrCar->yearMade = 2017;
    ptrCar->mpg = 20.45;

    // display car
    cout << "Make: " << aCar.make << endl;
    cout << "Model: " << aCar.model << endl;
    cout << "Year: " << aCar.yearMade << endl;
    cout << "MPG: " << aCar.mpg << endl;

    return 0;
}
```

```
Make: BMW
Model: M4
Year: 2017
MPG: 20.45

Process returned 0 (0x0)   execution time : 0.209 s
Press any key to continue.
```

- Similar to declaring the pointer to a primitive data type, programmer can declare a pointer to an abstract data type
- In this example, we declare a pointer to a CarInfo data type
- Note the arrow “->” notation when working with the pointer

# Pointers to Structures as Function Parameters

- Just like references, structures could be used as function parameters to avoid unnecessary copying data
- Even though pointers require little more work than references, it is well worth it because pointers are more flexible than references
- Unlike a pointer, a reference must always be associated with valid memory
- Once reference is defined, the memory it refers to cannot be changed
- References are less flexible than pointers because they're associated with the same variable; there is no separate memory address to store a reference
- In fact, a reference is simply an alias for the variable it refers to
- A pointer is a separate variable (memory location) from the variable it points to

```

#include <iostream>

using namespace std;

struct CarInfo
{
    string make;
    string model;
    int    yearMade;
    double mpg;
};

void getCarInfo(CarInfo *);
void displayCarInfo(const CarInfo *);

int main()
{
    CarInfo myCar;           // allocate a struct of data type CarInfo
    CarInfo* ptrMyCar;       // declare a pointer capable of pointing
                             // to CarInfo
    ptrMyCar = &myCar;      // point to myCar

    getCarInfo(ptrMyCar);
    displayCarInfo(ptrMyCar);
    return 0;
}

void getCarInfo(CarInfo *ptrCar)
{
    cout << "Make: ";
    getline(cin, ptrCar->make);

    cout << "Model: ";
    getline(cin, ptrCar->model);

    cout << "Year Made: ";
    cin >> ptrCar->yearMade;

    cout << "MPG: ";
    cin >> ptrCar->mpg;
}

void displayCarInfo(const CarInfo* ptrCar)
{
    cout << "Make: " << ptrCar->make << endl;
    cout << "Model: " << ptrCar->model << endl;
    cout << "Year Made: " << ptrCar->yearMade << endl;
    cout << "MPG: " << ptrCar->mpg << endl;
}

```

- In this example, the getCarInfo function is passed a pointer of data type CarInfo
- Notice that prior to function call, ptrMyCar was assigned the address of my car
- Next the program calls the getCarInfo function and passes the address (pointer to myCar)
- The function populates the structure with user data
- The pointer to myCar is then passed to displayCarInfo function as a constant pointer to avoid function accidentally overriding any of the information

# Returning Structure Pointers from Functions

```
#include <iostream>

using namespace std;

struct CarInfo
{
    string make;
    string model;
    int yearMade;
    double mpg;
};

CarInfo* getCarInfo();
void displayCarInfo(const CarInfo *);

int main()
{
    CarInfo* ptrMyCar; // declare a pointer capable holding
                       // the address of CarInfo

    ptrMyCar = getCarInfo();
    displayCarInfo(ptrMyCar);
    delete ptrMyCar;
    ptrMyCar = NULL;
    return 0;
}

CarInfo* getCarInfo()
{
    CarInfo *ptrCar = new CarInfo;
    cout << "Make: ";
    getline(cin, ptrCar->make);

    cout << "Model: ";
    getline(cin, ptrCar->model);

    cout << "Year Made: ";
    cin >> ptrCar->yearMade;

    cout << "MPG: ";
    cin >> ptrCar->mpg;

    return ptrCar;
}
```

- In this example, we declare a pointer that is capable of holding the address of the CarInfo data type
- What is different in this example is that in the getCarInfo function we dynamically allocate a block of memory (on the heap)
- We then proceed to populate this block of memory with data.
- Right before the function goes out of scope we return the pointer to this block of memory
- The address to this block is assigned to the ptrMyCar pointer
- And is finally passed to the displayCarInfo function whose body you can see below:

```
void displayCarInfo(const CarInfo* ptrCar)
{
    cout << "Make: " << ptrCar->make << endl;
    cout << "Model: " << ptrCar->model << endl;
    cout << "Year Made: " << ptrCar->yearMade << endl;
    cout << "MPG: " << ptrCar->mpg << endl;
}
```

- Finally we clean up the allocated memory with the aid of the delete operator

# Defensive Programming

- When allocating memory dynamically with the new operator, it is a good practice to test whether the allocation was successful
- Most of the time it will be successful, but to be 100% sure, a good programmer always utilizes defensive programming techniques:

```
CarInfo *ptrCar = new CarInfo;

if (!ptrCar)
{
    cout << "Unable to allocate memory. Exiting....";
    return EXIT_FAILURE;
}
```

# Enumerated Data Types

- Enumerated data type is an abstract data type giving programmers the ability to define their own datatype for a fixed list of values
- Following is the syntax needed to define enumerated data type:

```
enum TypeName { comma separated list of enumerators};
```

- Keyword “enum”
  - followed by the programmer defined data type
  - followed by the opening “{” and
  - a list of comma separated enumerators and
  - finally the closing “}”
  - Followed by a semicolon “;”
- Enumerated data types often simplify source code – making it more readable



# Enumerated Data Types

- Enumerated data types are an abstract data type and they cannot be mixed/confused with integers even though they may look like integers
- The compiler automatically assigns an ordinal number to the members of the list of enumerators
- By default the first member and the list is assigned a value of zero
- The ordinal number of the next element in the list is the previous element +1

```
#include <iostream>

using namespace std;

int main()
{
    enum CarMakers {Ferrari, Lamborghini, McLaren, Porsche};
    CarMakers superCars;

    cout << "Ferrari      : " << Ferrari << endl;
    cout << "Lamborghini: " << Lamborghini << endl;
    cout << "McLaren      : " << McLaren << endl;
    cout << "Porsche      : " << Porsche << endl;
    return 0;
}
```

```
Ferrari      : 0
Lamborghini: 1
McLaren      : 2
Porsche      : 3

Process returned 0 (0x0)   execution time : 0.089 s
Press any key to continue.
```

# Enumerated Data Types

- It is possible to assign a different ordinal number to an enumerator
- Keeping to remember, is that from that point forward enumerators in the list follow the same rule for assigning values to enumerators: previous enumerator +1
- Since we signed Lamborghini ordinal number 100, and next element in the list becomes 101 (McLaren)

```
#include <iostream>

using namespace std;

int main()
{
    enum CarMakers {Ferrari, Lamborghini=100, McLaren, Porsche};
    CarMakers superCars;

    cout << "Ferrari      : " << Ferrari << endl;
    cout << "Lamborghini: " << Lamborghini << endl;
    cout << "McLaren     : " << McLaren << endl;
    cout << "Porsche     : " << Porsche << endl;
    return 0;
}
```

```
Ferrari      : 0
Lamborghini: 100
McLaren     : 101
Porsche     : 102

Process returned 0 (0x0)    execution time : 0.101 s
Press any key to continue.
```

# Enumerated Data Types

```
#include <iostream>

using namespace std;

enum CarMakers {Audi, BMW, Saab, Toyota};

int main()
{
    CarMakers myCar;
    myCar = Saab;

    if (myCar == Audi)
        cout << "I drive an Audi" << endl;
    else if (myCar == BMW)
        cout << "I drive a BMW" << endl;
    else if (myCar == Saab)
        cout << "I drive a Saab" << endl;
    else if (myCar == Toyota)
        cout << "I drive a Toyota" << endl;

    return 0;
}
```

Running the program produces the following output:

```
I drive a Saab

Process returned 0 (0x0)   execution time : 0.115 s
Press any key to continue.
```

# Enumerated Data Types and Integers

- Enumerated datatypes are an abstract data type and they cannot be mixed/confused with integers

- For example, the following statement:

```
myCar = Audi;
```

is correct but the following statement will result in a syntax error:

```
myCar = 100;
```

- The reason for this is that they are two different data types.
- myCar is an enumerated ADT and 100 is an integer
- To correctly assign integer literal 100 to my car the programming needs to cast it to appropriate data type:

```
myCar = static_cast<CarMaker>(100);
```

- However, the other way around it is okay because enumerators are stored in computer memory as integers:

```
int aCar = Audi;
```