Structs in C++: An Introduction

- In C++, a struct is a user-defined composite data type that groups together variables of different data types under a single name.
- They are primarily used to organize related data and create custom composite data types.
- Struct stands for structure, i.e. user-defined data structure.

Here's why they are needed and how they are stored in memory.

Motivation for Structs:

1. **Data Organization**: Structs provide a way to group related data together. For instance, if you need to represent information about a person, you can create a struct that includes variables for their name, age, and address.

2. **Passing Data**: Structs make it easy to pass multiple pieces of data as a single unit to functions. This is particularly useful when working with functions that require several input parameters.

3. **Memory Efficiency**: Unlike classes (we will get to those), which come with additional features like encapsulation, structs are typically more memory-efficient because they do not have member functions and do not impose access control like private and public sections (again we will get to those).

Memory Storage:

Structs store their members in memory sequentially, one after the other. The memory allocated for a struct is equal to the sum of the memory allocated for each of its members. The order of the members is the order in which they are declared.

Examples

Example 1: Creating a Simple Struct

```
#include <iostream>
// Create a Point struct
struct Point {
    int x;                      // 4 bytes
    int y;                      // 4 bytes
};                              // total 8 bytes

int main() {
    Point p1;                   // Declare an instance of the Point struct
    p1.x = 5;                   // Assign 5 to x
    p1.y = 10;                  // Assign 10 to y

    int sum = p1.x + p1.y;      // add x and y together and store in sum

    std::cout << sum << "\n";   // print out sum

    return 0;
}
```

In this example, we've defined a `Point` struct with two integer members, `x` and `y`. In the `main` function, we declare an instance of the `Point` struct and access its members to perform operations.

Each `int` takes up 4 bytes, so the `Point` struct takes up a total of 8 bytes.

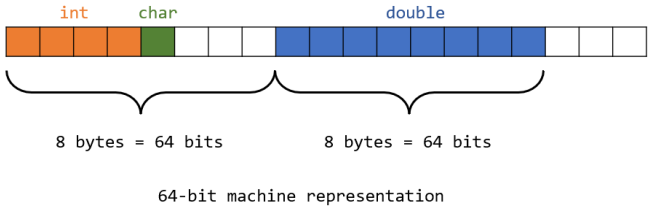
Example 2: Struct with mixed data types

In this example we create a struct with 3 different data types.

```
struct Product {
    int stockQuantity;  // An integer for the quantity in stock
    char productCode;   // A single character for the product code
    double price;       // A double for the price of the product
};

// actual size in memory is 16 bytes on a 64 bit system because of data alignment
(padding)
```

The total size of these data types is 13 bytes and they are stored sequentially. However they may not be stored contiguously because of data alignment.

On a 64-bit machine, the memory layout of this struct might look like this:



64-bit machine representation

In this representation:

- The `integer` member is of type `int`, which typically uses 4 bytes (on most systems).
- The `character` member is of type `char`, which usually uses 1 byte for storing a single character.
- The `floatingPoint` member is of type `double`, which is typically 8 bytes on most systems.

This visual representation shows a simplified layout of the `Data` struct in memory, with members allocated sequentially. Keep in mind that the actual memory layout can be affected by padding and alignment considerations imposed by the compiler for efficient memory access.

Here the `int` is 4-byte aligned, the `char` is 1-byte aligned and the `double` is aligned 8-byte aligned.

Therefore there is 3 bytes of padding to align the `double`.

https://en.wikipedia.org/wiki/Data_structure_alignment

Example 3: Struct Array

In this example we will create a `Book` struct and then create an array of `Book` structs. We can then loop over these printing out each book.

```
#include <iostream>
#include <string>

// Create a Book struct
struct Book {
    std::string title;
    std::string author;
    int year;
};

int main() {
    Book library[3];  // An array of Book structs

    // Assign values to the struct stored at index 0
    library[0] = {"The Great Gatsby", "F. Scott Fitzgerald", 1925};
    // Assign values to the struct stored at index 1
    library[1] = {"To Kill a Mockingbird", "Harper Lee", 1960};
    // Assign values to the struct stored at index 2
    library[2] = {"1984", "George Orwell", 1949};

    // loop over the array and print out the values for each Book in the array
    for (int i = 0; i < 3; i++) {
        std::cout << "Title: " << library[i].title << ", Author: " <<
library[i].author << ", Year: " << library[i].year << "\n";
    }

    return 0;
}
```

This demonstrates how structs can be used to create custom data structures that can be stored in arrays or other data collections.

NOTE: A `string` is a dynamic, heap-allocated data structure. It is not stored directly within the struct. Instead, a std::string object contains a pointer to the dynamically allocated memory that holds the actual string data.

Pointers are explained in the other half of the lecture notes.

If you are interested then please come ask me or look it up.

Struct as a Function Parameter

Structs can be passed to functions, this allows us not to pass all the variables individually!

Structs can be very large and sometimes we don't want to make additional copies in memory. Why would you if you were just trying to read the data!

NOTE: The following takes about passing by reference and value, this is common for any data type. Here were talk about it using structs.

Example 4: Pass by Reference (Read-only)

In the following example we pass the struct as a reference using the `&` operator. This means we are actually giving the function access to the original struct in memory.

We also use the keyword const to make it read-only, we cannot make changes to the struct in the function.

Key-Point:

- Use when you want to just read the struct.
- This means you don't create another copy in memory.
- The function `printStudentInfo` takes in a readonly reference of a struct.

```
#include <iostream>

struct Student {
    std::string name;
    int age;
};

// We pass the struct by reference (&) and make it readonly (const)
// WE HAVE NOT MADE A COPY IN MEMORY!
void printStudentInfo(const Student &student) {
    std::cout << "Name: " << student.name << ", Age: " << student.age << "\n";
}

int main() {
    Student s1;
    s1.name = "Alice";
    s1.age = 20;

    printStudentInfo(s1);   // prints Name: Alice, Age: 20

    return 0;
}
```

Example 5: Pass by Reference

Key-Point:

- Use when you want to update the original struct instance.
- This means you don't create another copy in memory.
- The function `incrementAge` takes in a reference of a struct.

```
#include <iostream>

struct Student {
    std::string name;
    int age;
};

// We pass the struct by reference (&) and make it readonly (const)
void printStudentInfo(const Student &student) {
    std::cout << "Name: " << student.name << ", Age: " << student.age << "\n";
}

// We pass the struct by reference (no copy), we can update this!
// It updates the orginal instance s1
void incrementAge(Student &student) {
    student.age = 21;
}

int main() {
    Student s1;
    s1.name = "Alice";
    s1.age = 20;

    printStudentInfo(s1);       // prints Name: Alice, Age: 20
    incrementAge(s1);           // add 1 to age and update s1 (pass by reference)
    printStudentInfo(s1);       // prints Name: Alice, Age: 21

    return 0;
}
```

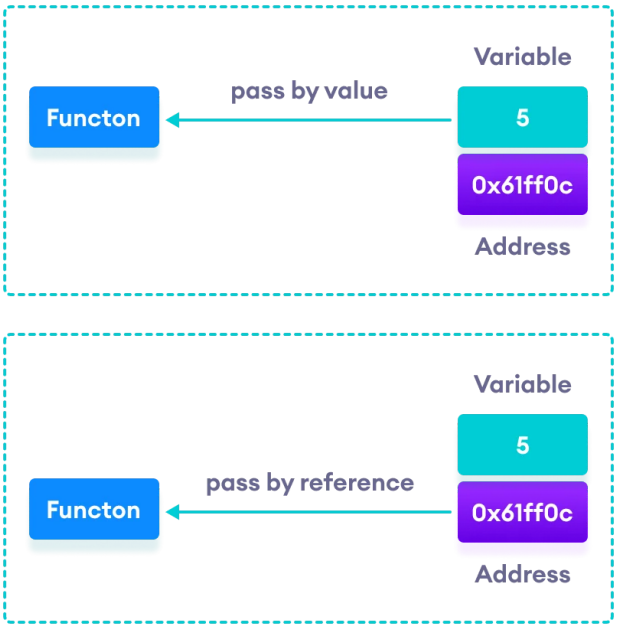
Example 6: Pass by Value

Key-Point:

- Use when you want a copy of the struct to work on.
- This means you create a copy in memory.
- The function `makeOld` creates a copy of the struct.

```
#include <iostream>
#include <string>

struct Student {
    std::string name;
    int age;
};

// We pass the struct by reference (&) and make it readonly (const)
void printStudentInfo(const Student &student) {
    std::cout << "Name: " << student.name << ", Age: " << student.age << "\n";
}

// We pass the struct by reference, we can update this!
// It updates the orginal instance s1
void incrementAge(Student &student) {
    student.age = 21;
}

// We pass the struct by value, we create a copy.
// Orginal instance s1 is not updated.
void makeOld(Student student) {
    student.age = 89;
}

int main() {
    Student s1;
    s1.name = "Alice";
    s1.age = 20;

    printStudentInfo(s1);       // prints Name: Alice, Age: 20
    incrementAge(s1);           // add 1 to age and update s1 (pass by reference)
    printStudentInfo(s1);       // prints Name: Alice, Age: 21
    makeOld(s1);                // pass a copy of s1, update age (pass by value)
    // prints Name: Alice, Age: 21 - The call to makeOld did not change s1!
    printStudentInfo(s1);

    return 0;
}
```

The image below shows a pictorial representation of pass by reference and pass by value using an `int`.



C++ Pass by Value vs. Pass by Reference [Programiz]

Summary of Structs

- In C++, a struct is a user-defined composite data type that groups together variables of different data types under a single name.
- They are primarily used for organizing related data and creating custom composite data types.
- Structs allow multiple pieces of data to be passed as a single unit to functions. This simplifies function calls, especially when dealing with functions requiring several input parameters.
- Structs store their members in memory sequentially, one after the other. The memory allocated for a struct is the sum of the memory allocated for each of its members.
- The order in which members are declared determines their order in memory.