# C++ Tutorial 1

**Instructor: Beom Heyn Kim**

beomheynkim@hanyang.ac.kr

Department of Computer Science

## C++ Introduction

- **What is C++?**
  - C++ is a cross-platform language that can be used to create high-performance applications.
  - C++ was developed by Bjarne Stroustrup, as an extension to the C language.
  - C++ gives programmers a high level of control over system resources and memory.
  - The language was updated 4 major times in 2011, 2014, 2017, and 2020 to C++11, C++14, C++17, C++20.

# C++ Introduction

- **Why Use C++**
  - C++ is one of the world's most popular programming languages.
  - C++ can be found in today's operating systems, Graphical User Interfaces, and embedded systems.
  - C++ is an object-oriented programming language which gives a clear structure to programs and allows code to be reused, lowering development costs.
  - C++ is portable and can be used to develop applications that can be adapted to multiple platforms.
  - C++ is fun and easy to learn!
  - As C++ is close to C, C# and Java, it makes it easy for programmers to switch to C++ or vice versa.

# C++ Introduction

- **Difference between C and C++**
  - C++ was developed as an extension of C, and both languages have almost the same syntax.
  - The main difference between C and C++ is that C++ support classes and objects, while C does not.

# Hello World! in C++

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

int main() {
  cout << "Hello World!";
  return 0;
}
```

including <iostream> header file library lets us work with input and output objects such as cout.

use names for objects and variables from the standard library.

cout (pronounced "see-out") is an **object** used together with the *insertion operator* (<<) to output/print text.

```
#include <iostream>

int main() {
  std::cout << "Hello World!";
  return 0;
}
```

The using namespace std line can be omitted and replaced with the **std** keyword, followed by the **::** operator for some objects:

# Hello World! in C++

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

int main() {
// This is a comment
cout << "Hello World! \n"
  cout << "I am learning C++";
  return 0;
}
```

Single-line comments start with two forward slashes (//)

To insert a new line, you can use the \n character

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

int main() {
/* The code below will print the words Hello World!
to the screen, and it is amazing */
cout << "Hello World!" << endl;
  cout << "I am learning C++";
  return 0;
}
```

Multi-line comments start with /* and ends with */.

Another way to insert a new line, is with the endl manipulator

## C++ Variables

- **C++ Variables**
  - Variables are containers for storing data values.
  - In C++, there are different **types** of variables (defined with different keywords), for example:
    - **int** - stores integers (whole numbers), without decimals, such as 123 or -123
    - **double** - stores floating point numbers, with decimals, such as 19.99 or -19.99
    - **char** - stores single characters, such as 'a' or 'B'. Char values are surrounded by single quotes
    - **string** - stores text, such as "Hello World". String values are surrounded by double quotes
    - **bool** - stores values with two states: true or false

# C++ Variables

## Declaring (Creating) Variables

To create a variable, specify the type and assign it a value:

**Syntax**

*type variableName* = *value*;

```
int myNum = 15;
cout << myNum;
```

```
int myNum;
myNum = 15;
cout << myNum;
```

```
int myNum = 15;  // myNum is 15
myNum = 10;  // Now myNum is 10
cout << myNum;  // Outputs 10
```

All C++ **variables** must be **identified** with **unique names**.

These unique names are called **identifiers**.

Identifiers can be short names (like x and y) or more descriptive names (age, sum, totalVolume).

## Declare Many Variables

To declare more than one variable of the **same type**, use a comma-separated list:

```
int x = 5, y = 6, z = 50;
cout << x + y + z;
```

## One Value to Multiple Variables

You can also assign the **same value** to multiple variables in one line:

```
int x, y, z;
x = y = z = 50;
cout << x + y + z;
```

## Constants

When you do not want others (or yourself) to change existing variable values, use the const keyword (this will declare the variable as "constant", which means **unchangeable and read-only**):

```
const int myNum = 15;  // myNum will always be 15
myNum = 10;  // error: assignment of read-only variable 'myNum'
```

# C++ User Input

## C++ User Input

You have already learned that cout is used to output (print) values. Now we will use cin to get user input.

cin is a predefined variable that reads data from the keyboard with the extraction operator (>>).

In the following example, the user can input a number, which is stored in the variable x. Then we print the value of x:

```
int x;
cout << "Type a number: "; // Type a number and press enter
cin >> x; // Get user input from the keyboard
cout << "Your number is: " << x; // Display the input value
```

cout is pronounced "see-out". Used for **output**, and uses the insertion operator (<<)

cin is pronounced "see-in". Used for **input**, and uses the extraction operator (>>)

## Creating a Simple Calculator

In this example, the user must input two numbers. Then we print the sum by calculating (adding) the two numbers:

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

int main() {
  int x, y;
  int sum;
  cout << "Type a number: ";
  cin >> x;
  cout << "Type another number: ";
  cin >> y;
  sum = x + y;
  cout << "Sum is: " << sum;
  return 0;
}
```

# C++ Data Types

## C++ Data Types

A variable in C++ must be a specified data type:

```
int myNum = 5;            // Integer (whole number)
float myFloatNum = 5.99;       // Floating point number
double myDoubleNum = 9.98;   // Floating point number
char myLetter = 'D';              // Character
bool myBoolean = true;          // Boolean
string myText = "Hello";          // String
```

## Basic Data Types

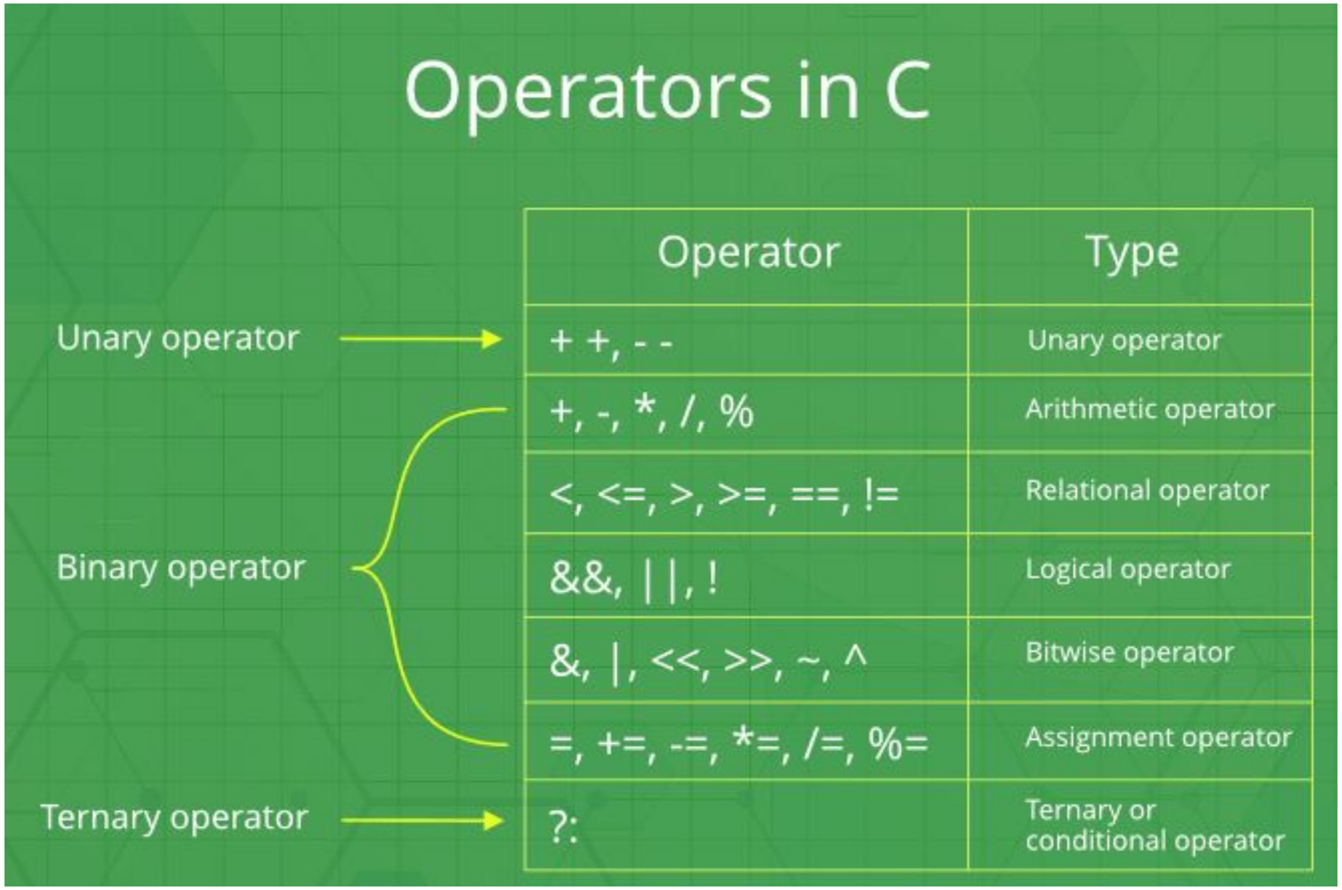
The data type specifies the size and type of information the variable will store:

| Data Type | Size | Description |
|---|---|---|
| boolean | 1 byte | Stores true or false values |
| char | 1 byte | Stores a single character/letter/number, or ASCII values |
| int | 2 or 4 bytes | Stores whole numbers, without decimals |
| float | 4 bytes | Stores fractional numbers, containing one or more decimals. Sufficient for storing 6-7 decimal digits |
| double | 8 bytes | Stores fractional numbers, containing one or more decimals. Sufficient for storing 15 decimal digits |

# C++ Operators

## C++ Operators

Operators are used to perform operations on variables and values.



Operators in C

Unary operator
Binary operator
Ternary operator

| Operator | Type |
|---|---|
| + +, - - | Unary operator |
| +, -, *, /, % | Arithmetic operator |
| <, <=, >, >=, ==, != | Relational operator |
| &&, \|\|, ! | Logical operator |
| &, \|, <<, >>, ~, ^ | Bitwise operator |
| =, +=, -=, *=, /=, %= | Assignment operator |
| ?: | Ternary or conditional operator |

# C++ Operators

## Arithmetic Operators

Arithmetic operators are used to perform common mathematical operations.

```
int sum1 = 100 + 50;          // 150 (100 + 50)
int sum2 = sum1 + 250;       // 400 (150 + 250)
int sum3 = sum2 + sum2;      // 800 (400 + 400)
```

| Operator | Name | Description | Example |
|---|---|---|---|
| + | Addition | Adds together two values | x + y |
| - | Subtraction | Subtracts one value from another | x - y |
| * | Multiplication | Multiplies two values | x * y |
| / | Division | Divides one value by another | x / y |
| % | Modulus | Returns the division remainder | x % y |
| ++ | Increment | Increases the value of a variable by 1 | ++x |
| -- | Decrement | Decreases the value of a variable by 1 | --x |

# C++ Operators

## Assignment Operators

Assignment operators are used to assign values to variables.

```
int x = 10;
x += 5;
```

A list of all assignment operators:

| Operator | Example | Same As |
|---|---|---|
| = | x = 5 | x = 5 |
| += | x += 3 | x = x + 3 |
| -= | x -= 3 | x = x - 3 |
| *= | x *= 3 | x = x * 3 |
| /= | x /= 3 | x = x / 3 |
| %= | x %= 3 | x = x % 3 |
| &= | x &= 3 | x = x & 3 |
| \|= | x \|= 3 | x = x \| 3 |
| ^= | x ^= 3 | x = x ^ 3 |
| >>= | x >>= 3 | x = x >> 3 |
| <<= | x <<= 3 | x = x << 3 |

## C++ Operators

**Assignment Operators Examples**

Try examples below and see their outputs.
Can you explain how you got those outputs?

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

int main() {
  int x = 5;
  x *= 3;
  cout << x;
  return 0;
}
```

```
include <iostream>
using namespace std;

int main() {
  int x = 5;
  x ^= 3;
  cout << x;
  return 0;
}
```

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

int main() {
  int x = 5;
  x >>= 3;
  cout << x;
  return 0;
}
```

# C++ Operators

## Comparison Operators

Comparison operators are used to compare two values (or variables). This is important in programming, because it helps us to find answers and make decisions.

The return value of a comparison is either 1 or 0, which means **true** (1) or **false** (0). These values are known as **Boolean values**, and you will learn more about them later.

```
int x = 5;
int y = 3;
cout << (x > y); // returns 1 (true) because 5 is greater than 3
```

A list of all comparison operators:

| Operator | Name | Example |
|---|---|---|
| == | Equal to | x == y |
| != | Not equal | x != y |
| > | Greater than | x > y |
| < | Less than | x < y |
| >= | Greater than or equal to | x >= y |
| <= | Less than or equal to | x <= y |

## C++ Operators

### Logical Operators

As with comparison operators, you can also test for **true** (1) or **false** (0) values with **logical operators**.

Logical operators are used to determine the logic between variables or values:

| Operator | Name | Description | Example |
|---|---|---|---|
| && | Logical and | Returns true if both statements are true | x < 5 &&  x < 10 |
| \|\| | Logical or | Returns true if one of the statements is true | x < 5 \|\| x < 4 |
| ! | Logical not | Reverse the result, returns false if the result is true | !(x < 5 && x < 10) |

# C++ Strings

## C++ Strings

Strings are used for storing text.

A string variable contains a collection of characters surrounded by double quotes:

```
// Include the string library
#include <string>

// Create a string variable
string greeting = "Hello";
```

## String Concatenation

The + operator can be used between strings to add them together to make a new string. This is called **concatenation**:

```
string firstName = "John ";
string lastName = "Doe";
string fullName = firstName + lastName;
cout << fullName;
```

## Append

A string in C++ is actually an object, which contain functions that can perform certain operations on strings. For example, you can also concatenate strings with the append() function:

```
string firstName = "John ";
string lastName = "Doe";
string fullName = firstName.append(lastName);
cout << fullName;
```

## String Length

To get the length of a string, use the length() function:

```
string txt = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
cout << "The length of the txt string is: " << txt.length();
```

## C++ Strings

### Access Strings

You can access the characters in a string by referring to its index number inside square brackets [].

```
string myString = "Hello";
cout << myString[0];
// Outputs H
```

### Change String Characters

To change the value of a specific character in a string, refer to the index number, and use single quotes:

```
string myString = "Hello";
myString[0] = 'J';
cout << myString;
// Outputs Jello instead of Hello
```

### Backslash Escape Characters

The backslash (\) escape character turns special characters into string characters:

| Escape character | Result | Description |
|---|---|---|
| \' | ' | Single quote |
| \" | " | Double quote |
| \\ | \ | Backslash |
| \n | New Line | n/a |
| \t | Tab | n/a |

# C++ Strings

## User Input Strings

As we saw earlier, it is possible to use the extraction operator >> on cin to display a string entered by a user:

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

int main() {
  string fullName;
  cout << "Type your full name: ";
  cin >> fullName;
  cout << "Your name is: " << fullName;
  return 0;
}
```

Type: John
Then, type: John Doe

What happens and why?

cin considers a space (whitespace, tabs, etc) as a terminating character)

Try the getline() function instead to read a line of text. It takes cin as the first parameter, and the string variable as second:

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

int main() {
  string fullName;
  cout << "Type your full name: ";
  getline (cin, fullName);
  cout << "Your name is: " << fullName;
  return 0;
}
```

# C++ Math

## C++ Math

C++ has many functions that allows you to perform mathematical tasks on numbers.

## Max and min

The max(x,y) and min(x,y) function can be used to find the highest and the lowest value of x and y, respectively:

```
cout << max(5, 10);
cout << min(5, 10);
```

## C++ <cmath> Header

Other functions, such as sqrt (square root), round (rounds a number) and log (natural logarithm), can be found in the <cmath> header file:

```
// Include the cmath library
#include <cmath>

cout << sqrt(64);
cout << round(2.6);
cout << log(2);
```

# C++ Booleans

## Boolean Values

A boolean variable is declared with the bool keyword and can only take the values true or false:

```
bool isCodingFun = true;
bool isFishTasty = false;
cout << isCodingFun;  // Outputs 1 (true)
cout << isFishTasty;  // Outputs 0 (false)
```

## Boolean Expression

A **Boolean expression** returns a boolean value that is either 1 (true) or 0 (false).

This is useful to build logic, and find answers.

You can use a comparison operator, such as the **greater than** (>) operator, to find out if an expression (or variable) is true or false:

```
int x = 10;
int y = 9;
cout << (x > y); // returns 1 (true), because 10 is higher than 9
```

# C++ Conditions

## C++ Conditions and If Statements

C++ has the following conditional statements:

- if to specify a block of code to be executed, if a specified condition is true
- else to specify a block of code to be executed, if the same condition is false
- else if to specify a new condition to test, if the first condition is false
- switch to specify many alternative blocks of code to be executed

```
int time = 22;
if (time < 10) {
  cout << "Good morning.";
} else if (time < 20) {
  cout << "Good day.";
} else {
  cout << "Good evening.";
}
// Outputs "Good evening."
```

```
int time = 20;
string result = (time < 18) ? "Good day." : "Good evening.";
cout << result;
```

```
int day = 4;
switch (day) {
  case 1:
          cout << "Monday";
         break;
  case 2:
         cout << "Tuesday";
         break;
  case 3:
         cout << "Wednesday";
         break;
  case 4:
         cout << "Thursday";
          break;
  case 5:
         cout << "Friday";
          break;
  case 6:
         cout << "Saturday";
          break;
  case 7:
         cout << "Sunday";
          break;
  default:
         cout << "Looking forward to the Weekend";
}
// Outputs "Thursday" (day 4)
```

## C++ While Loop

### C++ Loops

Loops can execute a block of code as long as a specified condition is reached.

Loops are handy because they save time, reduce errors, and they make code more readable.

### C++ While Loop

The while loop loops through a block of code as long as a specified condition is true

```
int i = 0;
while (i < 5) {
  cout << i << "\n";
  i++;
}
```

```
int i = 0;
do {
  cout << i << "\n";
  i++;
}
while (i < 5);
```

**Note:** Do not forget to increase the variable used in the condition, otherwise the loop will never end!

# C++ For Loop

## C++ For Loop

When you know exactly how many times you want to loop through a block of code, use the for loop instead of a while loop:

```
for (int i = 0; i < 5; i++) {
  cout << i << "\n";
}
```

## The foreach Loop

There is also a "**for-each** loop" (introduced in C++ version 11 (2011), which is used exclusively to loop through elements in an array (or other data sets):

```
int myNumbers[5] = {10, 20, 30, 40, 50};
for (int i : myNumbers) {
  cout << i << "\n";
}
```

# C++ Break and Continue

## C++ Break

You have already seen the break statement used in an earlier chapter of this tutorial. It was used to "jump out" of a switch statement.

The break statement can also be used to jump out of a **loop**.

```
for (int i = 0; i < 10; i++) {
  if (i == 4) {
            break;
  }
  cout << i << "\n";
}
```

## C++ Continue

The continue statement breaks one iteration (in the loop), if a specified condition occurs, and continues with the next iteration in the loop.

```
for (int i = 0; i < 10; i++) {
  if (i == 4) {
            continue;
  }
  cout << i << "\n";
}
```

Can you try the example below and then replace while loops with equivalent for loops?

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

int main() {
  int i = 0;
  while (i < 10) {
        cout << i << "\n";
        i++;
        if (i == 4) {
        break;
        }
  }
  i = 0;
  while (i < 10) {
        if (i == 4) {
        i++;
        continue;
        }
        cout << i << "\n";
        i++;
  }
  return 0;
}
```

# C++ Arrays

## C++ Arrays

Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value.

To declare an array

```
string cars[4];
```

```
string cars[4] = {"Volvo", "BMW", "Ford", "Mazda"};
```

```
string cars[] = {"Volvo", "BMW", "Ford", "Mazda"};
```

```
int myNum[3] = {10, 20, 30};
```

## Access the Elements of an Array

You access an array element by referring to the index number inside square brackets [].

```
string cars[4] = {"Volvo", "BMW", "Ford", "Mazda"};
cout << cars[0];
// Outputs Volvo
```

## Change an Array Element

To change the value of a specific element, refer to the index number:

```
string cars[4] = {"Volvo", "BMW", "Ford", "Mazda"};
cars[0] = "Opel";
cout << cars[0];
// Now outputs Opel instead of Volvo
```

## Loop Through an Array

You can loop through the array elements with the for loop.

```
string cars[5] = {"Volvo", "BMW", "Ford", "Mazda", "Tesla"};
for (int i = 0; i < 5; i++) {
  cout << cars[i] << "\n";
}
```

# C++ Arrays

## Get the Size of an Array

To get the size of an array, you can use the sizeof() operator:

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

int main() {
  int myNumbers[5] = {10, 20, 30, 40, 50};
  cout << sizeof(myNumbers);
  return 0;
}
```

Can you try the example above and see what it outputs? Is it what you expected? If not, what do you think has happened??

sizeof() operator returns the size of a type in **bytes**.

**To find out how many elements an array has**, you have to divide the size of the array by the size of the data type it contains:

```
int myNumbers[5] = {10, 20, 30, 40, 50};
int getArrayLength = sizeof(myNumbers) / sizeof(int);
cout << getArrayLength;
```

## Loop Through an Array with sizeof()

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

int main() {
  int myNumbers[5] = {10, 20, 30, 40, 50};
  for (int i = 0; i < sizeof(myNumbers) / sizeof(int); i++) {
        cout << myNumbers[i] << "\n";
  }
  return 0;
}
```

Can you try the example above?
Replace the for loop with a for-each loop and retry.

# C++ Arrays

## Multi-Dimensional Arrays

A multi-dimensional array is an array of arrays.

To declare a multi-dimensional array

```
string letters[2][4];
```

```
string letters[2][4] = {
  { "A", "B", "C", "D" },
  { "E", "F", "G", "H" }
};
```

```
string letters[2][2][2] = {
  {
        { "A", "B" },
         { "C", "D" }
  },
  {
        { "E", "F" },
         { "G", "H" }
  }
};
```

## Loop Through a Multi-Dimensional Array

To loop through a multi-dimensional array, you need one loop for each of the array's dimensions.

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

int main() {
  string letters[2][4] = {
        { "A", "B", "C", "D" },
        { "E", "F", "G", "H" }
  };

  letters[0][0] = "Z";

  for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 4; j++) {
        cout << letters[i][j] << "\n";
        }
  }
  return 0;
}
```

# C++ Structures (Struct)

## C++ Structures

Structures (also called structs) are a way to group several related variables into one place. Each variable in the structure is known as a **member** of the structure.

Unlike an array, a structure can contain many different data types (int, string, bool, etc.).

## Create a Structure

To create a structure, use the struct keyword and declare each of its members inside curly braces.

After the declaration, specify the name of the structure variable (**myStructure** in the example below):

```
struct {           // Structure declaration
  int myNum;              // Member (int variable)
  string myString;   // Member (string variable)
} myStructure;            // Structure variable
```

## Access Structure Members

To access members of a structure, use the dot syntax (.):

```
// Create a structure variable called myStructure
struct {
  int myNum;
  string myString;
} myStructure;

// Assign values to members of myStructure
myStructure.myNum = 1;
myStructure.myString = "Hello World!";

// Print members of myStructure
cout << myStructure.myNum << "\n";
cout << myStructure.myString << "\n";
```

# C++ Structures (Struct)

## One Structure in Multiple Variables

You can use a comma (,) to use one structure in many variables:

```
struct {
  int myNum;
  string myString;
} myStruct1, myStruct2, myStruct3;
// Multiple structure variables separated with commas
```

## Named Structures

By giving a name to the structure, you can treat it as a data type. This means that you can create variables with this structure anywhere in the program at any time.

To create a named structure, put the name of the structure right after the struct keyword:

```
struct myDataType { // This structure is named "myDataType"
  int myNum;
  string myString;
};

myDataType myVar; // To declare a variable of “myDataType”
```

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

// Declare a structure named "car"
struct car {
  string brand;
  string model;
  int year;
};

int main() {
  // Create a car structure and store it in myCar1;
  car myCar1;
  myCar1.brand = "BMW";
  myCar1.model = "X5";
  myCar1.year = 1999;

  // Create another car structure and store it in myCar2;
  car myCar2;
  myCar2.brand = "Ford";
  myCar2.model = "Mustang";
  myCar2.year = 1969;

  // Print the structure members
  cout << myCar1.brand << " " << myCar1.model << " " << myCar1.year << "\n";
  cout << myCar2.brand << " " << myCar2.model << " " << myCar2.year << "\n";

  return 0;
}
```

# C++ References

## Creating References

A reference variable is a "reference" to an existing variable, and it is created with the & operator:

```
string food = "Pizza";  // food variable
string &meal = food;    // reference to food
```

Now, we can use either the variable name food or the reference name meal to refer to the food variable:

```
string food = "Pizza";
string &meal = food;

cout << food << "\n";  // Outputs Pizza
cout << meal << "\n";  // Outputs Pizza
```

## Memory Address

When a variable is created in C++, a memory address is assigned to the variable. And when we assign a value to the variable, it is stored in this memory address.

To access it, use the & operator, and the result will represent where the variable is stored:

```
string food = "Pizza";

cout << &food; // Outputs 0x6dfed4
```

## C++ Pointers

### Creating Pointers

As we saw from the previous slide, we can get the **memory address** of a variable by using the & operator:

```
string food = "Pizza"; // A food variable of type string

cout << food;  // Outputs the value of food (Pizza)
cout << &food; // Outputs the memory address of food (0x6dfed4)
```

A **pointer** however, is a variable that **stores the memory address as its value**.

A pointer variable points to a data type (like int or string) of the same type, and is created with the * operator. The address of the variable you're working with is assigned to the pointer:

```
string food = "Pizza";  // A food variable of type string
string* ptr = &food;  // A pointer variable, with the name ptr, that
stores the address of food

// Output the value of food (Pizza)
cout << food << "\n";

// Output the memory address of food (0x6dfed4)
cout << &food << "\n";

// Output the memory address of food with the pointer (0x6dfed4)
cout << ptr << "\n";
```

### Get Memory Address and Value

We used the pointer variable to get the memory address of a variable (used together with the & **reference** operator). However, you can also use the pointer to get the value of the variable, by using the * operator (the **dereference** operator):

```
string food = "Pizza";  // Variable declaration
string* ptr = &food;  // Pointer declaration

// Reference: Output the memory address of
food with the pointer (0x6dfed4)
cout << ptr << "\n";

// Dereference: Output the value of food with
the pointer (Pizza)
cout << *ptr << "\n";
```

# C++ Pointers

## Modify the Pointer Value

You can also change the pointer's value. But note that this will also change the value of the original variable:

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

int main() {
  string food = "Pizza";
  string* ptr = &food;

  // Output the value of food
  cout << food << "\n";

  // Output the memory address of food
  cout << &food << "\n";

  // Access the memory address of food and output its value
  cout << *ptr << "\n";

  // Change the value of the pointer
  *ptr = "Hamburger";

  // Output the new value of the pointer
  cout << *ptr << "\n";

  // Output the new value of the food variable
  cout << food << "\n";
  return 0;
}
```

## The End