C++ OOP

Souvik Ghosh

Pointers

Pointers Introduction

Understanding Address

Pointer is a powerful feature of C++ programming that allows us to work with memory addresses.

But before we learn about pointers, let's first learn about addresses.

Suppose we have created a variable like this:

```
int var;
```

Now, a space will be allocated in the computer memory for the `var` variable, and we can access the memory address using `&var`.

Let's see an example.

```
#include <iostream>
using namespace std;
int main() {
int var = 13;
// print the value stored in the variable
cout << "Value of var: " << var << endl;
// print the memory address of the variable
cout << "Address of var: " << &var;
return 0;
}
```

Output

```
Value of var: 13
Address of var: 0x7ffc5ff6f594
```

Here, you can see that printing `&var` gives `0x7ffc5ff6f594`, which is the memory location where 13 is stored.

💡
Note: You may get a different address when you run this code because the output depends on the location where the variable will be stored (which varies from device to device).

Pointer Variables

Now that you know about memory addresses, let's see what role pointers play in all of this.

A pointer is a special symbol that stores the address of a variable rather than a value.

Let's first see how we can create pointers.

Create a Pointer

```
int* pt;
```

Here, `pt` is the pointer variable. Let's compare it with a regular variable:

```
int var;
```

As you can see, we have used `int*` instead of `int` to represent the pointer variable. Note that the `*` here doesn't mean multiplication; it is a part of the syntax used for pointer declaration.

> Note: We can also use the code `int *pt;` to create a pointer variable. However, we recommend you use the `int* pt;` format instead.

Assign Address to a Pointer

When we first declare a pointer, the system assigns a random empty address to it.

We can then change the random address by assigning the address of a variable to a pointer. For example,

```
#include <iostream>
using namespace std;
int main() {
// create a variable
int number = 36;
// create a pointer variable
// a random address is assigned to pt
int* pt;
// assign address of number variable to pointer
pt = &number;
// print the address stored in pt pointer
cout << "Value of pt: " << pt << endl;
// print the address of the number variable
cout << "Address of number: " << &number;
return 0;
}
```

Output

```
Value of pt: 0x7fff50757bec
Address of number: 0x7fff50757bec
```

Here, you can see the value of `pt` (pointer variable) and `&number` (address of `number`) is the same.

This is because we have assigned the address of `number` to the `pt` variable.

```
pt = &number;
```

Here's how this program works:

1. `pt` is a pointer variable, and `number` is a regular variable with a value of 36.

2. The code `pt = &number;` assigns the address of the `number` variable to `pt`.

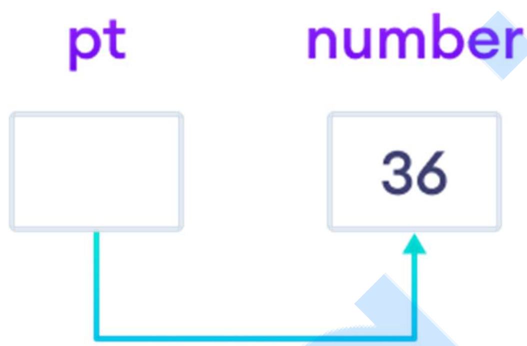

Figure: Assign Address to Pointer

3. Finally, since both `pt` and `&number` point to the address of the `number` variable, we get the same output when we print them.

Get Value Pointed by Pointers

In C++, we can also use pointers to print the value stored in another variable. Let's take an example,

```
int number = 36;
// pt stores the address of number
int* pt = &number;
```

Here, the pointer stores the memory address of the variable `number`. Now, we can also access the value of the `number` variable by using the code `*pt`.

Let's see how.

```
#include <iostream>
using namespace std;
int main() {
// create a variable
int number = 36;
// create a pointer variable to store address of number
int* pt = &number;
// print the memory address
cout << "Address: " << pt << endl;
// print the value of number using pt
cout << "Value: " << *pt;
```

```
    return 0;
}
```

Output

```
Address: 0x7ffee16b0c3c
Value: 36
```

In this program, we have used two `cout` statements:

1. To Print the Memory Address

```
cout << "Address: " << pt << endl;
```

Here, we have used `pt` to indicate the pointer variable.

2. To Print the Value

```
cout << "Value: " << *pt;
```

Here, we have used `*pt` to indicate the value stored in the variable whose address is stored in `pt`.

This process is called dereferencing a pointer in C++. In other words, whenever we use `*pt` to access the variable value, we are dereferencing the `pt` pointer.

Important! Remember the Difference Between pt and *pt!

- `pt` is the pointer variable which gives the memory address.
- `*pt` gives the value of the variable whose address is stored in `pt`.

Common Pointer Mistakes

Many people create pointers like this:

```
int *pt;
```

This is also a valid way to declare a pointer. However, this syntax sometimes creates lots of confusion among beginners.

In this syntax, the `*` symbol is attached to `pt`, so many people think that `*pt` is the pointer variable, which is wrong.

In fact, `pt` is the pointer variable which stores the memory address and `*` is just a part of syntax to create pointers.

And `*pt` denotes the data stored in the address that is pointed by `pt`.

DON'T GET CONFUSED.

Tip: To avoid this confusion, we recommend you use `int* pt`.

Common Mistakes While Working With Pointers

Suppose, we want a pointer `pt` to hold the address of `number`. Then,

```
int number;
int* pt;

// pt is address but number is not
pt = number;    // Error

// &number is address but *pt is not
*pt = &number;    // Error

// both &number and pt are addresses
pt = &number;    // Valid

// both number and *pt are values
*pt = number;    // Valid
```

Pointers & Arrays

Memory Address and Array

In C++, we can use pointers to work with arrays. But before that, let's revise the working of an array.

An array is a collection of multiple data of the same type. For example,

```
#include <iostream>
using namespace std;
int main() {
// array of numbers
int numbers[5] = {1, 2, 3, 4, 5};
// print array
cout << "Array Elements: ";
for (int i = 0; i < 5; ++i) {
cout << numbers[i] << ", ";
}
return 0;
}
// Output:
// Array Elements: 1, 2, 3, 4, 5,
```

Here, we have created an array of numbers. We then used a `for` loop to print the array elements.

Now, let's try to print addresses of array elements.

```
#include <iostream>
using namespace std;
int main() {
// array of numbers
int numbers[5] = {1, 2, 3, 4, 5};
// print the addresses of array elements
for (int i = 0; i < 5; ++i) {
cout << &numbers[i] << endl;
```

```
}
cout << "Address of the array: " << numbers;
return 0;
}
```

Output

```
0x7ffd9ab98350
0x7ffd9ab98354
0x7ffd9ab98358
0x7ffd9ab9835c
0x7ffd9ab98360
Address of the array: 0x7ffd9ab98350
```

Things to notice

1. The difference between the addresses of two array elements is 4.

```
// the last two digits of this address is 50
0x7ffd9ab98350
// the last two digits of the next address is 54
0x7ffd9ab98354
```

This is because our array is of type `int` and the size of `int` is 4 bytes.

Hence, each array element is taking 4 bytes of memory storage.

2. The address of the first array element, 1, and the address of the array, `numbers`, is the same.

This is because the array address always points to the first element of the array.

Figure: Array Address

Note: We have used `numbers` instead of `&numbers` while printing the address of the array. This is because in most contexts, array names decay (get converted) to pointers and we can use pointers to access elements of the array, which we will see next.

Arrays and Pointers

In our last example, we saw that the array name can also decay to a pointer.

Thus, if we have an array `numbers[]`, then the array name `numbers` can be used as a pointer that points to the first element of the array.

For example,

```
#include <iostream>
using namespace std;
int main() {
// array of numbers
int numbers[5] = {1, 2, 3, 4, 5};
// address of first array element
cout << &numbers[0] << endl; // 0x7ffef078f350
// address of first array element
cout << numbers << endl; // 0x7ffef078f350
return 0;
}
```

💡
Remember, `numbers` represents the pointer here.

Now, we can use this pointer to access elements of the array.

Here,

1. `&numbers[0]` is equivalent to `numbers`. Hence, the first element 1 can be accessed using `*numbers`.

Figure: Array and Pointer

2. `&numbers[1]` is equivalent to `numbers + 1` and the second element 2 can be accessed using `*(numbers + 1)`.

3. `&numbers[2]` is equivalent to `numbers + 2` and the third element 3 can be accessed using `*(numbers + 2)`.

And, so on...

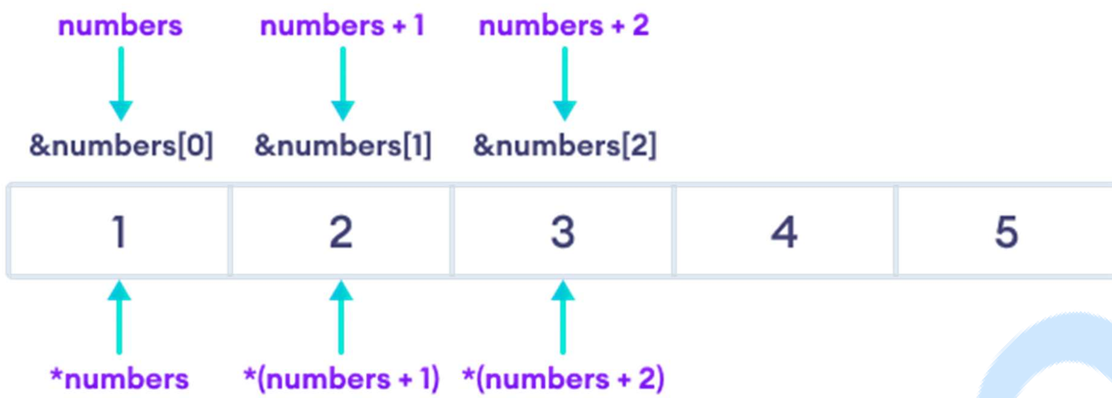

Figure: Array and Pointers

Basically,

- `&numbers[i]` is equivalent to `numbers + i`
- `numbers[i]` is equivalent to `*(numbers + i)`

Now, let's implement this in a working example.

Example: Arrays and Pointers

Here's an example to demonstrate the relationship between pointers and arrays.

```
#include <iostream>
using namespace std;
int main() {
// array of numbers
int numbers[5] = {1, 2, 3};
// print second element using pointer
cout << "Second Élément: " << *(numbers + 1) << endl;
// print last element using pointer
cout << "Last Element: " << *(numbers + 2);
return 0;
}
```

Output

```
Second Element: 2
Last Element: 3
```

As you can see, we have successfully accessed the second and last elements using pointer notation: `*(numbers + 1)` and `*(numbers + 2)` respectively.

We can also change array elements using pointer notation. Let's see an example,

```
#include <iostream>
using namespace std;
int main() {
// array of numbers
```

```
int numbers[3] = {1, 2, 3};
// change second element to 5
*(numbers + 1) = 5;
// change last element to 10
*(numbers + 2) = 10;
// print second element using pointer notation
cout << "Second Element: " << *(numbers + 1) << endl;
// print last element using pointer notation
cout << "Last Element: " << *(numbers + 2);
return 0;
}
```

Output

```
Second Element: 5
Last Element: 10
```

As expected, the values of the second and last elements are changed to 5 and 10, respectively.

Find Largest Array Element Using Pointers

In Learn C++ Basics, we had written a program like the one below to find the largest element of the array.

```
#include <iostream>
using namespace std;
int main() {
// an array of numbers
int numbers[5] = {55, 64, 75, 80, 65};
// assign the first element of the array to the largest variable
int largest = numbers[0];
// iterate each element of the array
// if ith element is greater than largest
// assign that element to largest
for (int i = 1; i < 5; ++i) {
if (largest < numbers[i]) {
largest = numbers[i];
}
}
cout << "Largest: " << largest;
return 0;
}
// Output:
// Largest: 80
```

Now, let's use pointer notation to achieve the same result.

```
#include <iostream>
using namespace std;
int main() {
// an array of numbers
int numbers[5] = {55, 64, 75, 80, 65};
```

```
// assign the first element of the array to the largest variable
int largest = *numbers;
// iterate each element of the array
// if ith element is greater than largest
// assign that element to largest
for (int i = 1; i < 5; ++i) {
if (largest < *(numbers + i)) {
largest = *(numbers + i);
}
}
cout << "Largest: " << largest;
return 0;
}
// Output:
// Largest: 80
```

As you can see, we have successfully found the largest element. Here, we have replaced

- `numbers[0]` with `*numbers` (to indicate the first element)
- `numbers[i]` with `*(numbers + i)` (to indicate the ith element)

Here's how this code works:

i	***(numbers + i)**	**largest < *(numbers +i)**	**largest**
1	64	true	64
2	75	true	75
3	80	true	80
4	65	false	80

Pointers & Functions

Revise Functions

Before we move forward, let's revise the working of a function with an example.

```
#include <iostream>
using namespace std;
// function to add two numbers
int add_numbers(int n1, int n2) {
int sum = n1 + n2;
return sum;
}
int main() {
int number1 = 32;
int number2 = 44;
// function call
int result = add_numbers(number1, number2);
cout << "Result: " << result;
return 0;
}
// Output:
// Result: 76
```

In the above program, we have created the `add_numbers()` function that takes two parameters, `n1` and `n2` and finds their sum. The parameters are like input given to the function while the resulting sum is the function output.

And just like regular variables, it is also possible to pass addresses as arguments to functions. It's because an address is also a value.

Next, we will see an example of passing addresses to a function.

Pointer as Function Argument

Let's start with an example.

```
#include <iostream>
using namespace std;
// function that accepts address as parameter
void change_value(int* n) {
// change value at address to 120
*n = 120;
}
int main() {
int number = 35;
cout << "Number (before): " << number << endl;
// call function with address of number as argument
change_value(&number);
cout << "Number (after): " << number;
return 0;
```

```
}
```

Output

```
Number (before): 35
Number (after): 120
```

In the above example, we have passed the address of the `number` variable during the function call.

```
change_value(&number);
```

This address is now assigned to the `n` pointer (function parameter).

Inside the function, we have assigned `120` to the address pointed by `n`.

```
*n = 120;
```

Now, the value of the `number` variable is also changed to `120` in the `main()` function.

This is because the address in the `n` pointer and that of the `number` variable are the same and we are changing the value at the same address.

Example: Swap Two Numbers

In this example, we will swap two numbers using a function. However, this time we will be using pointers to swap the numbers.

```
#include <iostream>
using namespace std;
// fuction to swap numbers
void swap_numbers(int* n1, int* n2) {
int temp;
// swap values stored in n1 and n2
temp = *n1;
*n1 = *n2;
*n2 = temp;
}
int main() {
int number1 = 34;
int number2 = 57;
// call function by passing address of both variables
swap_numbers(&number1, &number2);
cout << "After Swapping" << endl;
cout << "number1: " << number1 << endl;
cout << "number2: " << number2;
return 0;
}
```

Output

```
After Swapping
number1: 57
number2: 34
```

Here, the `n1` and `n2` pointers in the `swap_numbers()` function take the addresses of `number1` and `number2` variables.

When the values stored in `n1` and `n2` addresses are swapped, the values of `number1` and `number2` are also swapped in the `main()` function.

Return Pointers From a Function

Do you remember this code to return a value from a function?

```
#include <iostream>
using namespace std;
// function to add 10 to a number
int add_ten(int a) {
int sum = a + 10;
return sum;
}
int main() {
int number = 32;
// call function
int result = add_ten(number);
cout << "Result: " << result;
return 0;
}
// Output:
// Result: 42
```

Here, we have returned the `sum` variable after adding 10 to the parameter `a`.

Similarly, we can also return pointers from a function. Let's see an example,

```
#include <iostream>
using namespace std;
// function to add 10 to a number
int* add_ten(int* pt) {
// dereference the pointer
// add 10 to the variable pointed by pointer
*pt = *pt + 10;
// return the pt pointer
return pt;
}
int main() {
int number = 32;
// call add_ten() function
// pass the address of number as parameter
// store the return value in result pointer
int* result = add_ten(&number);
// print the value in number by dereferencing result
cout << "Result: " << *result;
return 0;
}
// Output:
// Result: 42
```

Notice These Things

- `int* add_ten` - `int*` indicates the function returns a pointer
- `add_ten(int* pt)` - the function parameter `pt` is a pointer
- `*pt = *pt + 10` - add 10 to the value pointed by the `pt` pointer
- `return pt` - returns the address pointed by `pt`

Now, let's look at how this program works.

Here, we have passed the address of the `number` variable to the function and stored the return value in the `result` pointer.

```
// function call
int* result = add_ten(&number);
```

Inside the `add_ten()` function,

- the address of `number` is stored in the pointer parameter `pt`,
- 10 is added to the value of `number` by dereferencing `pt`,
- finally, the `pt` pointer (address of `number`) is returned by the function.

```
int* add_ten(int* pt) {
*pt = *pt + 10;
return pt;
}
```

Since the `pt` pointer in `add_ten()` is pointing to the `number` variable, the value of `number` gets changed by the function.

This also means that the `add_ten()` function returns a pointer to the address of the `number` variable, which is stored by the `result` pointer in `main()`.

Thus, `result` is actually a pointer that points to the `number` variable. So when we dereference `result` and print its value, we are actually printing the new value of the `number` variable.

Common Mistake While Returning Pointers

Suppose you want to create a function that adds two numbers and returns a pointer to the memory location that has the sum. In such a case, you might end up writing a program like this:

```
#include <iostream>
using namespace std;
// function that returns address
// of variable that contains sum of numbers
int* add(int n1, int n2) {
// create sum variable inside function
int sum = n1 + n2;
// return address of sum variable
return ∑
}
```

```
int main() {
// call the add() function
// store the return value in sum pointer
int* sum = add(32, 10);
// print the value in sum pointer
cout << "Sum: " << *sum;
return 0;
}
// Output: Segmentation Fault
```

However, when you run this program, you get a Segmentation Fault error.

The compiler gives this error because the `sum` variable inside the `add()` function is only valid inside that function. It gets destroyed once the function call is over and the program control goes back to the `main()` function.

We can fix this by passing a pointer or variable declared in the `main()` function, and then returning that pointer (or the address to the passed variable) from the `add()` function.

Next, we'll implement this solution through an example program.

Example: Fix Mistake While Returning Pointer

This is how we can solve the issue in the previous section while returning a valid pointer/address from the function:

```
#include <iostream>
using namespace std;
// function that returns address
// of variable that contains sum of numbers
// 3rd parameter is a pointer
int* add(int n1, int n2, int* pt) {
// store the result in pt
*pt = n1 + n2;
// return the pt pointer
return pt;
}
int main() {
// create the sum variable
int sum;
// call the add() function
// pass the address of sum variable as 3rd argument
// store the return value in result pointer
int* result = add(32, 10, &sum);
// print the result
cout << "Sum: " << *result;
return 0;
}
// Output:
// Sum: 42
```

Here, we have declared the `sum` variable in `main()` and passed its address to the `add()` function. The `pt` parameter of the function now has the address of `sum`.

The function then returns the address to this `sum` variable, which is stored in the `result` pointer.

This program works because `sum` belongs to the `main()` function. Hence, it is not destroyed when the function call is over.

Revise Pointers

Pointers Summary

Let's revise what we have learned in this chapter:

1. Memory Address

`&number` gives the address of the `number` variable. For example,

```
#include <iostream>
using namespace std;
int main() {
int number = 32;
// print the address of number variable
cout << &number;
return 0;
}
// Output: 0x7ffeb9e74a54
```

2. Pointer Variables

A pointer variable is used to store the memory address of a variable. For example,

```
#include <iostream>
using namespace std;
int main() {
int number = 32;
// pt is a pointer variable that stores
// memory address of number
int* pt = &number;
cout << pt;
return 0;
}
// Output: 0x7ffc89fa4bfc
```

3. Access Value Using Pointers

`*pt` accesses the value pointed by the address stored in the `pt` pointer. This process is known as dereferencing a pointer. For example,

```
#include <iostream>
using namespace std;
int main() {
int number = 32;
// pt is a pointer variable that stores
```

```
// memory address of number
int* pt = &number;
// value stored in the address pointed by pt
cout << *pt;
return 0;
}
// Output: 32
```

We can also use pointers with arrays and functions which we will revise using the following examples.

- Add 10 to each element of the array
- Challenge: Multiply each element of the array by N
- Add two numbers using a function
- Challenge: Divide two numbers using a function

Add 10 to Each Element of the Array

Suppose we have an array with elements: `{8, 7, 21, 13}`. Now, we need to add 10 to each element of the array so our final array looks like this: `{18, 17, 31, 23}`.

Thought Process

First, we need to access each array element using a loop. Inside the loop, we need to add 10 to each element and assign the result to the respective position.

```
#include <iostream>

using namespace std;

int main() {

int numbers[4] = {8, 7, 21, 13};

// loop to access each array element

for (int i = 0; i < 4; ++i) {

// add 10 to the current array element

*(numbers + i) = *(numbers + i) + 10;

}

// print the array

cout << "Array Elements: ";

for (int i = 0; i < 4; ++i) {

cout << *(numbers + i) << ", ";

}

return 0;

}
```

Output

```
Array Elements: 18, 17, 31, 23,
```

In the above example, notice the line:

```
*(numbers + i) = *(numbers + i) + 10;
```

Here, `*(numbers + i)` accesses the array element at position `i`, adds 10 to it and assigns the result to the same position.

Quick Reminder:

- `numbers` - gives the address of first array element
- `*numbers` - gives the value of the first element
- `numbers + 1` - gives the address of the second element
- `*(numbers + 1)` - gives the value of the second element
- ....
- `numbers + i` - gives the address of the ith element
- `*(numbers + i)` - gives the value of the ith element

Add Two Numbers Using Function

Thought Process

Here, we will be using pointers so we need to create a function that accepts pointers as its arguments.

We also want to return a pointer, so we will use a pointer as the return type as well.

Based on what we have learned so far, our function will look like this:

```
int* add_numbers(int* n1, int* n2) {

int* sum = *n1 + *n2;

return sum;

}
```

However, this will give us an unexpected output because we are trying to return the address of the local pointer `sum`.

To avoid this, we need to create `sum` inside the `main()` and pass its address along with the addresses of two other numbers.

Let's implement that.

```
#include <iostream>

using namespace std;

// function to add two numbers

int* add_numbers(int* n1, int* n2, int* sum) {

*sum = *n1 + *n2;

return sum;

}

int main() {

int number1 = 75;

int number2 = 69;

int sum;

// call function with address as parameter

int* result = add_numbers(&number1, &number2, &sum);

cout << "Result: " << *result;

return 0;

}
```

Output

```
Result: 144
```

As you can see, we get the desired output.

OOP (Basics)

Understanding OOP

Object-oriented Programming (OOP)

In this chapter, we will learn about object-oriented programming (OOP) and how to implement it in our code.

Object-oriented programming (OOP) is a popular technique to solve programming problems by creating objects.

Let's try to understand it with an example.

Suppose we need to store the name and the test score of university students. And based on the test score, we need to find if a student passed or failed the examination. Then, the structure of our code would look something like this.

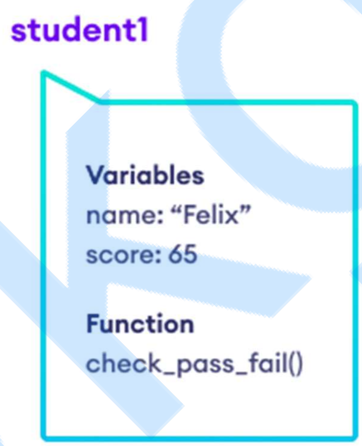

Figure: Code Structure

Now, imagine we have to store the name and the test score of multiple students instead of one student.

If we were to use the same approach, we can use the same `check_pass_fail()` function.

However, we would need to create multiple variables to store the `name` and the `score` for each student. This would make our code less organized and messy.

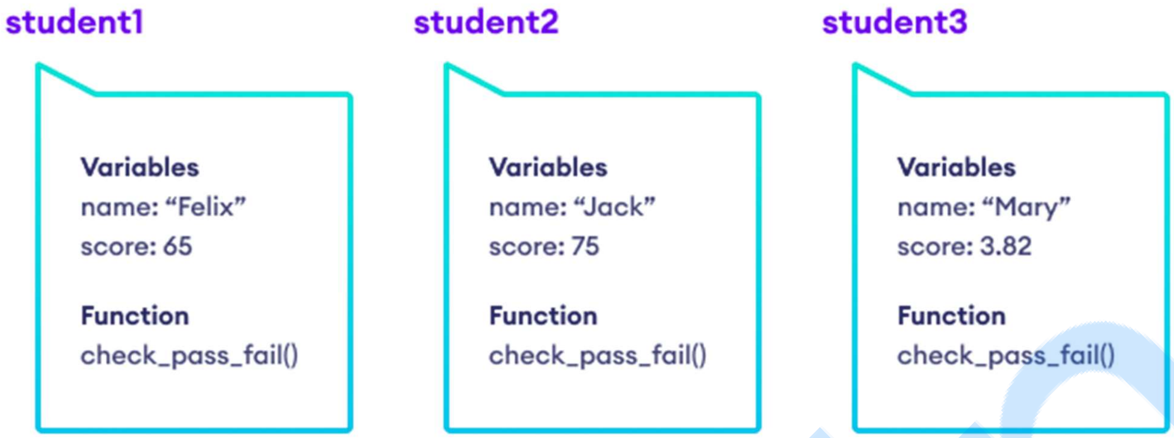

Figure: Code Structure

Since these data and functions are related, it would be better if we could treat them as a single entity. And we can do that by creating objects.

This approach to creating objects to solve problems is known as object-oriented programming.

Next, we will see how we use objects to solve this problem.

Introduction to Classes and Objects

There are two steps involved in creating objects:

1. Define a class
2. Create objects from the class

Define a Class

To solve the problem mentioned on the last page, we will first define a class named Student.

Figure: The Student Class

This `Student` class has two variables `name` and `score`, and a function `check_pass_fail()`.

Think of a class as a blueprint for a house. It contains all the details about the floors, doors, windows, etc. Based on these descriptions, we can build a house. The actual physical house is the object.

Now, let's see how we can create objects.

Creating Objects

Once we define a class, we can create as many objects as we want from the class.

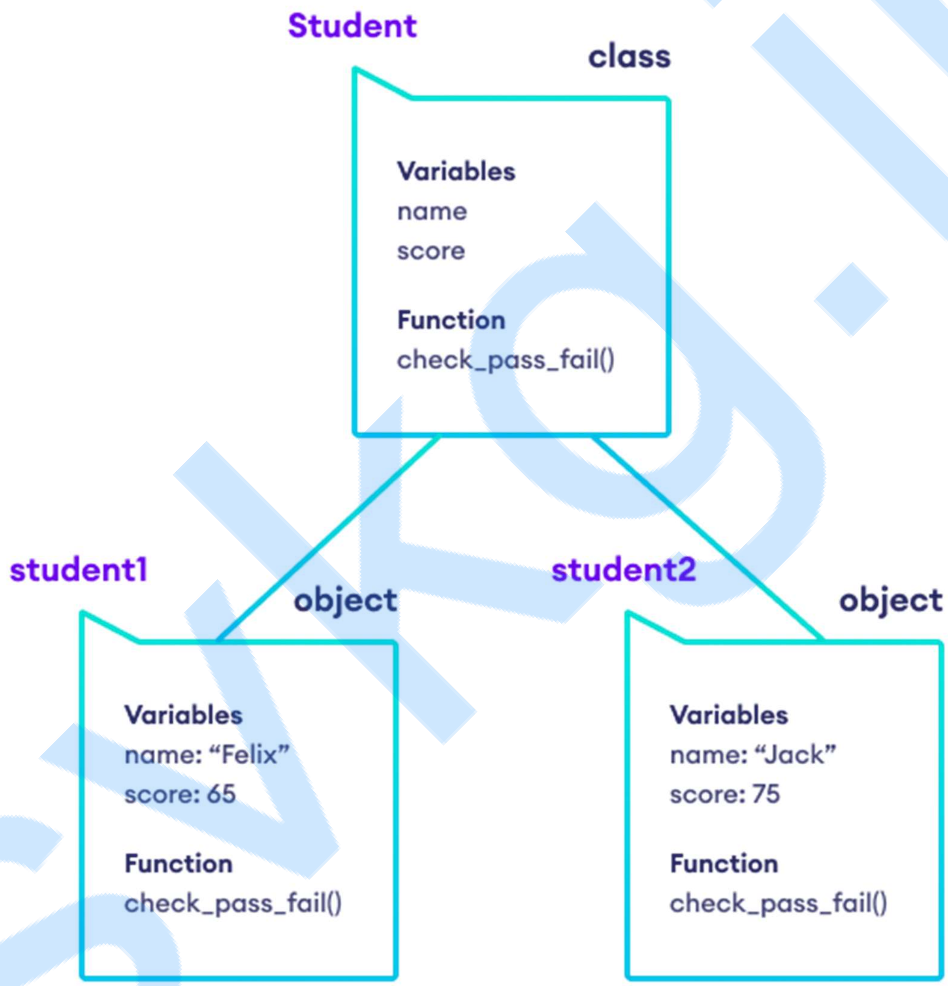

Figure: Classes and Objects

In the image, we have created objects `student1` and `student2` from the `Student` class.

All the objects of this `Student` class will have their own `name` and `score` variables and can use the `check_pass_fail()` function.

Note: The variables and functions of a class are called class members. The variables are called member variables or data members, and the functions are called member functions.

Classes & Objects

Creating a Class

As mentioned before, we need to create a class first before we can create objects from it.

In C++, we use the `class` keyword to create a class. For example,

```
class Car {
  ...
};
```

Here, we have created a class named `Car`.

A class can contain:

- data members - variables/arrays to store data
- member functions - to perform tasks on data members

Note: A class ends with the code `};`. In the past, we have ended loops and functions with the `}` symbol. For classes, however, we need to add a semicolon `;` after the closing brace `}`.

We will gradually add different functions and variables inside a class. But first, let's create objects from the class.

Creating Objects

Here's how we can create objects of a class.

```
// create a class
class Car {
    ...
};

// create object of the Car class
Car car1;
Car car2;
```

Here, `car1` and `car2` are objects of the `Car` class.

Next, we will learn how variables and functions are used with a class.

Add Member Variables

As mentioned earlier, a class contains data members (variables). Let's see how we can add them to the `Car` class.

```
class Car {
public:
// add member variables
int gear = 6;
string brand = "Audi";
};
```

Here, `gear` and `brand` are two data members inside the `Car` class.

Access Member Variables Using Object

Now, we will use an object of the `Car` class to access data members.

```
#include <iostream>
using namespace std;
class Car {
public:
// add data members
int gear = 6;
string brand = "Audi";
};
int main() {
// create object of Car
Car car1;
// access data members using object
cout << "Gear: " << car1.gear << endl;
cout << "Brand: " << car1.brand;
return 0;
}
```

Output

```
Gear: 6
Brand: Audi
```

In the above example, we have created an object named `car1` of the `Car` class. Notice the codes inside the `cout` statements:

```
// access the member variable gear
car1.gear
// access the member variable brand
car1.brand
```

Here, we have used the object along with the `.` dot operator to access the member variables of the class.

Note: In our class, we have directly assigned values to the `gear` and `brand` variables. We are doing this to keep things simple for the moment. But this is not the proper way to use data members in OOP. We will return to this topic later and use member variables correctly.

Next, we will learn to add member functions inside a class.

Adding Member Functions

Now, let's see how we can add member functions to a class.

```
class Car {
public:
// add member function
void check_status(int gear) {
if (gear >= 1) {
cout << "Car is running.";
}
else {
cout << "Car is not running."
}
}
};
```

Here, we have added the `check_status()` member function inside the `Car` class. This function accepts a single parameter `gear` and prints if the car is running or not.

Access Member Function Using Object

```
#include <iostream>
using namespace std;
class Car {
public:
// add member function
void check_status(int gear) {
if (gear >= 1) {
cout << "Car is running." << endl;
}
else {
cout << "Car is not running." << endl;
}
}
};
int main() {
// create object of Car
Car car1;
// access member function
car1.check_status(6);
car1.check_status(0);
```

```
return 0;
}
```

Output

```
Car is running.
Car is not running.
```

Notice the code,

```
car1.check_status(6);
car1.check_status(0);
```

Here, we are using the object `car1` along with the `.` dot operator to call the member function.

Assign Values to Member Variables Using Objects

In our earlier example, we have used member variable like this:

```
class Car {
public:
// member variables
int gear = 6;
string brand = "Audi";
};
```

This is not the proper way to use member variables In the OOP approach.

Instead, we should just declare variables inside the class and assign values using objects.

Let's see an example.

```
#include <iostream>
using namespace std;
class Car {
public:
// member variable
int gear;
string brand;
};
int main() {
// create an object of Car
Car car1;
// assign values to member variable
car1.gear = 6;
car1.brand = "Audi";
// access member variable
cout << "Gear: " << car1.gear << endl;
cout << "Brand: " << car1.brand;
return 0;
}
```

Output

```
Gear: 6
Brand: Audi
```

In the above example, we have created an object of the `Car` class.

```
Car car1;
```

Then, we used the `car1` object with the dot operator `.` to assign values to the variables `gear` and `brand`.

```
car1.gear = 6;
car1.brand = "Audi";
```

Let's see one more example.

Create Multiple Objects

```
#include <iostream>
using namespace std;
class Student {
public:
// data members
string name;
int score;
};
int main() {
// create two Student objects
Student student1, student2;
// initialize member variables of student1
student1.name = "Maria";
student1.score = 56;
// print member variables of student1
cout << "Name: " << student1.name << endl;
cout << "Score: " << student1.score << endl;
// initialize member variables of student2
student2.name = "Johnny";
student2.score = 32;
// print member variables of student2
cout << "Name: " << student2.name << endl;
cout << "Score: " << student2.score;
return 0;
}
```

Output

```
Name: Maria
Score: 56
Name: Johnny
Score: 32
```

In the above example, we have created two objects: `student1` and `student2` from the `Student` class.

```
Student student1, student2;
```

And both the students have their own names and scores.

With this approach, it's now easier to visualize the overall program. That is, `student1` is Maria and her score is 56. Similarly, `student2` is Johnny and his score is 32.

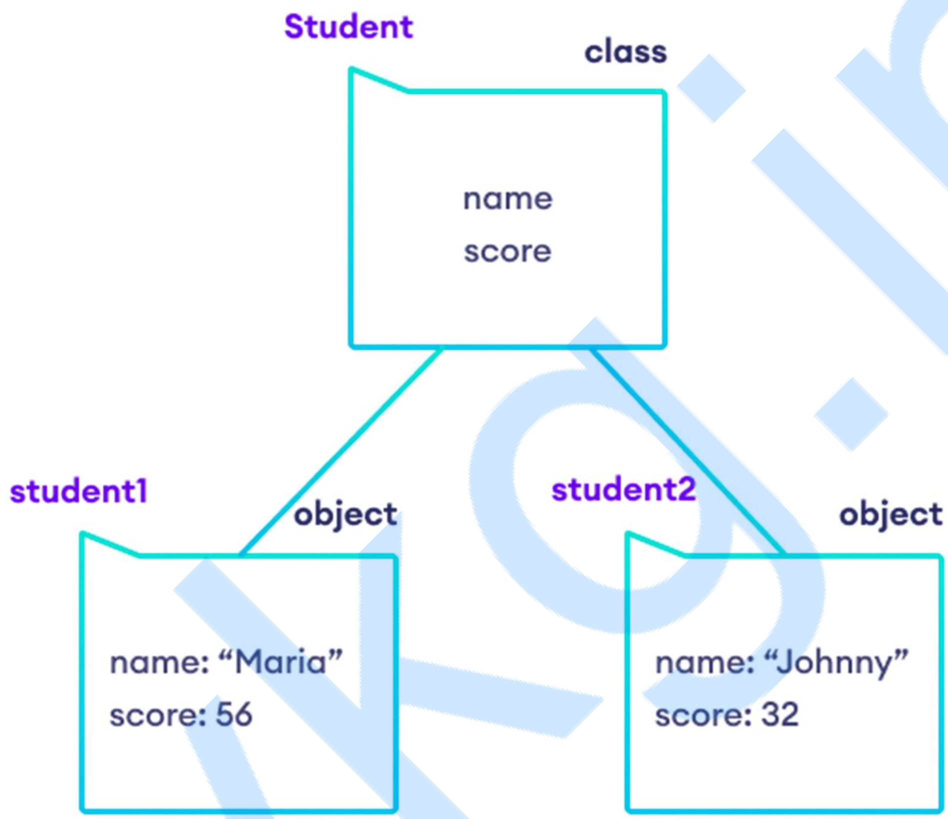

Figure: Create Multiple Objects

Modify Member Variables

We can also change the value of a member variable using objects. For example,

```
#include <iostream>
using namespace std;
class Student {
public:
string name;
};
int main() {
Student student1;
// assign value to name
student1.name = "Rosie";
cout << "Topper: " << student1.name << endl;
```

```
// change the value of name
student1.name = "Smith";
cout << "Topper: " << student1.name;
return 0;
}
```

Output

```
Topper: Rosie
Topper: Smith
```

Here, the initial value of `name` was `"Rosie"`, which we then changed to `"Smith"`.

Multiple Classes

We can also create multiple classes in a single program and access data between one another. For example,

```
#include <iostream>
using namespace std;
class Student {
public:
string name;
};
class Department {
public:
int code;
};
int main() {
// create an object of Student
Student student;
// access data member of student
student.name = "Jackie";
cout << "Student Name: " << student.name << endl;
// create an object of Department
Department department;
// access data member of department
department.code = 32;
cout << "Department Code: " << department.code;
return 0;
}
```

Output

```
Student Name: Jackie
Department Code: 32
```

Here, we have created two classes: `Student` and `Department`.

We will learn more about the uses of multiple classes in later chapters. For now, just remember it is also possible.

Why Objects and Classes?

We could have written all the programs in this lesson without using classes and objects. So you might be wondering where to use classes and objects.

As we have mentioned before, object-oriented programming is an approach we can take to solve problems; it's not mandatory to use classes and objects to solve problems.

So, when should we use classes and objects?

If we are working on a complex problem where variables and functions are related, treating them as a single entity by creating objects makes sense. For example,

Suppose we are working on a racing game.

To solve this problem, we can use objects such as cars, racing tracks, etc. Now, instead of thinking about individual variables and functions, we start thinking about objects and how one object interacts with the other. This helps us to divide a complex problem into smaller sub-problems.

So here's our suggestion:

If you are working on a simple problem, do not use object-oriented programming because you have to write a lot of code.

However, if you are working on a complex problem where many variables and functions are related, creating objects to solve that problem makes sense.

Constructor

C++ Constructors

In C++, a constructor is similar to a member function, but it doesn't have a return type, and it has the same name as the class. For example,

```
class Student {
public:
// constructor
Student() {
...
}
// member function
void check_name() {
...
}
};
```

In the above example, `Student()` is a constructor and `check_name()` is a member function. You can see that the constructor doesn't have a return type, and it has the same name as the class (`Student`).

In C++, the constructor is called automatically when we create an object. Let's see an example,

```
#include <iostream>
using namespace std;
class Student {
public:
// constructor
Student() {
cout << "Calling Constructor";
}
};
int main() {
// create an object
Student student1;
return 0;
}
```

Output

```
Calling Constructor
```

Here, the code `Student student1;` calls the constructor. That's why we get the output.

Types of Constructors

There are broadly two types of constructors in C++. They are

- Default Constructors
- Parameterized Constructors

Let's start with default constructors first.

Default Constructors

In C++, a default constructor is a constructor that has no parameters, and thus takes no arguments. The constructors we've been dealing with so far are all default constructors.

Let's see an example,

```
#include <iostream>

using namespace std;

class Student {

public:

int marks;

// default constructor

Student() {

marks = 0;
```

```
}
};
int main() {
// create an object
Student student1;
// print the value of marks
cout << "Marks: " << student1.marks;
return 0;
}
// Output: Marks: 0
```

Here, the `Student()` constructor doesn't take any argument. Hence, it's a default constructor.

Parameterized Constructors

As mentioned earlier, a parameterized constructor takes in arguments. We use this type of constructor to assign values to member variables for different objects.

Let's explore this with an example.

```
class Car {
public:
int gear;
// parameterized constructor
Car(int gear_no) {
gear = gear_no;
}
};
```

Here, `Car()` is a parameterized constructor that accepts a single parameter, `gear_no`.

Calling Parameterized Constructor

Just like any other constructor, a parameterized constructor is also called while creating objects. However, during the object creation, we pass arguments to the constructor. For example,

```
// call constructor
Car car1(5);
Car car2(6);
```

Here, the value of `gear_no` will be

- 5 for the object `car1`
- 6 for the object `car2`

Let's clarify this by writing a complete program.

```
#include <iostream>
```

```
using namespace std;
class Car {
public:
int gear;
// parameterized constructor to initialize gear
Car(int gear_no) {
gear = gear_no;
}
};
int main() {
// create objects of Car: car1 and car2
// pass 5 and 6 as arguments to constructors
// of car1 and car2 respectively
Car car1(5);
Car car2(6);
// print values of gear for car1 and car2
cout << "Gear for car1: " << car1.gear << endl;
cout << "Gear for car2: " << car2.gear;
return 0;
}
```

Output

```
Gear for car1: 5
Gear for car2: 6
```

In the above example, we have used the parameterized constructor to assign the values of the `gear` data member.

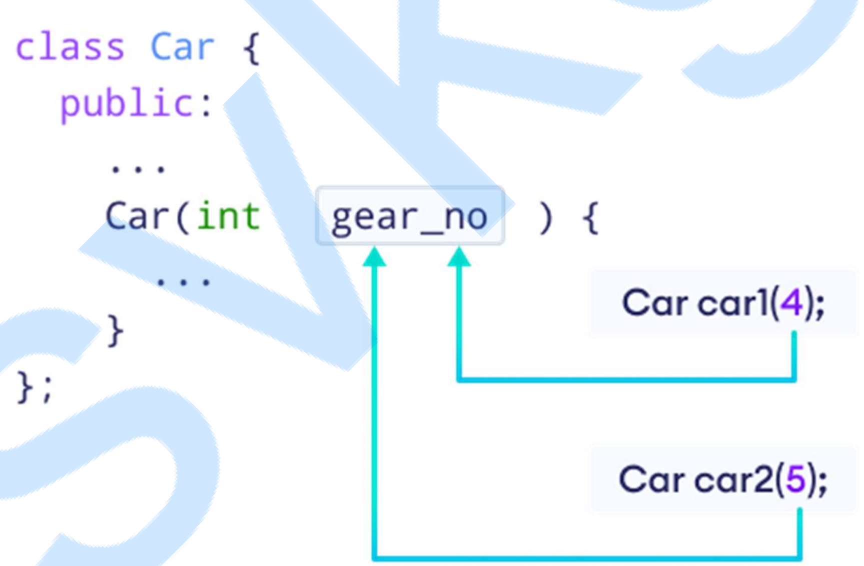

Figure: Passing different arguments to the constructor using different objects

Let's see one more example.

Example: Parameterized Constructor

```
#include <iostream>
using namespace std;
class Student {
public:
string name;
int score;
// parameterized constructor that takes two arguments
Student(string student_name, int student_score) {
name = student_name;
score = student_score;
}
};
int main() {
// create objects of Student
Student student1("Jackie", 76);
Student student2("Maria", 82);
// print data members for student1
cout << "---First Student---" << endl;
cout << "Name: " << student1.name << endl;
cout << "Score: " << student1.score << endl;
// print data members for student2
cout << "---Second Student---" << endl;
cout << "Name: " << student2.name << endl;
cout << "Score: " << student2.score;
return 0;
}
```

Output

```
---First Student---
Name: Jackie
Score: 76
---Second Student---
Name: Maria
Score: 82
```

In the above example, we have used a parameterized constructor to initialize the member variables, `name` and `score`.

```
Student(string student_name, int student_score) {
name = student_name;
score = student_score;
}
```

Here, while creating the objects:

Student student1("Jackie", 76);

- `Jackie` and 76 are assigned to `student_name` and `student_score`, respectively.
- Hence, `student1.name` will become `Jackie` and `student1.score` becomes 76.

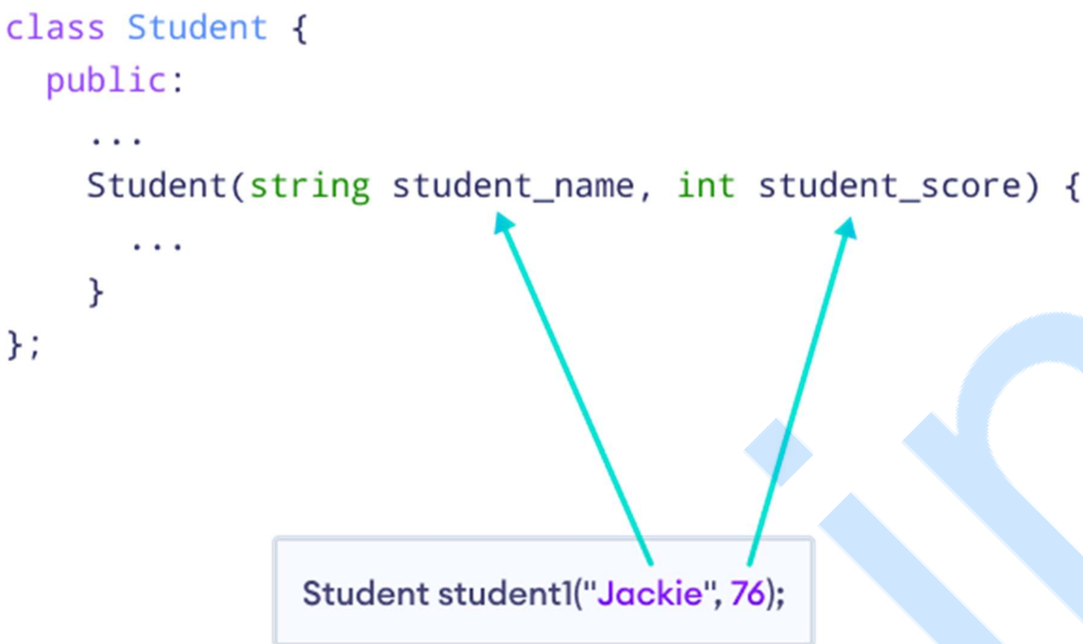

Figure: Multiple Arguments to Parameterized Constructor

Student student2("Maria", 82);

- `Maria` and 82 are assigned to `student_name` and `student_score`, respectively.
- Hence, `student2.name` will become `Maria` and `student2.score` becomes 82.

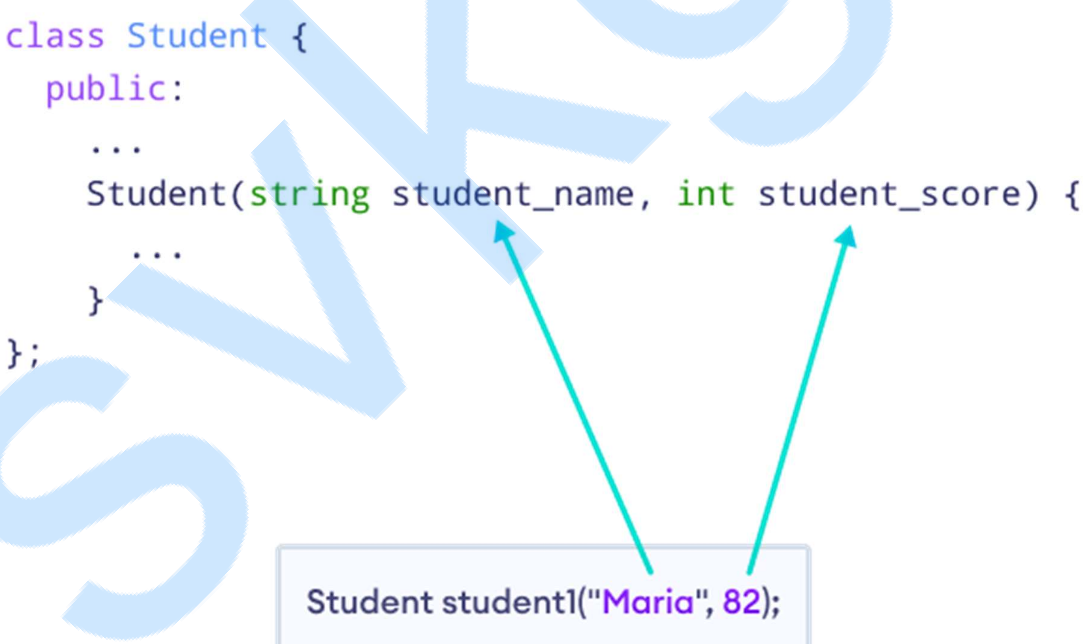

Figure: Multiple Arguments to Parameterized Constructor

Going Forward: Because constructors are executed automatically when we create an object, they are thus excellent tools for initializing member variables. For the rest of this lesson, we will be using constructors almost exclusively for this task.

Constructor Initializer List

In C++ constructors, we can also use an initialization list to initialize member variables. This will make our code look cleaner and more efficient. Let's see an example,

Suppose we are initializing the `name` and `score` variables using a constructor like this:

```
class Student {
    public:

        string name;
        int score;

       // constructor to initialize values
       Student(string sudent_name, int student_score) {
           name = student_name;
           score = student_score;
        }
};
```

Now let's see how we can do this using the initialization list.

```
class Student {
    public:

        string name;
        int score;

        // constructor to initialize values
        Student(string n, int s): name(n), score(s) {}
};
```

You can see our code now looks cleaner. Here,

- `n` and `s` are values passed to the constructor.
- `n` is assigned to the variable `name`.
- `s` is assigned to the variable `score`.

Now, let us put this method into practice with the help of a program.

Example: Constructor Initializer List

```
#include <iostream>
using namespace std;
class Student {
public:
string name;
int score;
// constructor initializer list
Student(string n, int s) : name(n), score(s) {}
};
int main() {
// create objects of Student
Student student1("Jackie", 76);
Student student2("Maria", 82);
// print data members for student1
cout << "---First Student---" << endl;
cout << "Name: " << student1.name << endl;
```

```
cout << "Score: " << student1.score << endl;
// print data members for student2
cout << "---Second Student---" << endl;
cout << "Name: " << student2.name << endl;
cout << "Score: " << student2.score;
return 0;
}
```

Output

```
---First Student---
Name: Jackie
Score: 76
---Second Student---
Name: Maria
Score: 82
```

Here, we have created a constructor named `Student()`. We then used the initializer list to initialize the member variables `name` and `score`.

```
Student(string n, int s) : name(n), score(s) {}
```

In `main()`, we have created two objects - `student1` and `student2` - and passed different arguments for each object.

```
Student student1("Jackie", 76);
Student student2("Maria", 82);
```

As a result, for

- car1 - `name` will be `Jackie` and `score` will be 76
- car2 - `name` will be `Maria` and `score` will be 82

Initializer List: Key Things to Remember

1. An initializer list is more efficient and cleaner. So it is preferred over a normal constructor.

2. Member variables should be initialized in the same order they are declared. For example,

```
class Car {
  public:
    int gear, speed;

    // bad practice
    Car(): speed(200), gear(5) {}
};
```

Here, `gear` is declared first. So it should also be initialized first.

Common Mistake

Calling the parameterized constructor without passing arguments

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

class Car {
    public:
        int gear;

        // constructor with parameter
        Car(int gear_no) {
            gear = gear_no;
        }
};

int main() {

    // error code
    Car car1;

    cout << car1.gear;
    return 0;
}
```

Here, the above program will cause an error. It's because the `Car()` constructor accepts an argument `gear_no`.

However, we are not passing any arguments while creating the object of the `Car` class.

```
// error code
Car car1;
// correct code
Car car1(4);
```

Public and Private Modifiers

Access Modifiers

So far in our example, we have been using the `public` keyword along with our member variables and functions within the class.

```
class Car {
    public:
        // code
};
```

Here, `public` means these data members and functions can be accessed from anywhere in the program. Hence, we were able to access them from the `main()` function.

However, there might be situations where we wouldn't want our data members and functions to be accessed from outside. For this, we use access modifiers in C++.

Access modifiers are used to set the visibility of data members, functions, and even classes. For example, if we don't want our class members to be accessed from outside, we can mark them as private using the `private` access modifier.

```
class Car {
    private:
        // code
};
```

In this lesson, we will learn about two major types of access modifiers in C++.

- `public` - allows access from outside
- `private` - prevents access from outside

There's also a third access modifier - `protected`. But we'll learn about it in a later chapter.

So, let's get started with the `public` modifier.

Public Modifier

As the name suggests, variables and functions declared with the `public` access modifier can be accessed from any class. Let's see an example,

```
#include <iostream>
using namespace std;
class Student {
// public variable
public:
string name;
};
int main() {
// create object of Student
Student student1;
// access the public variable of the Student class
student1.name = "Rosie";
cout << "Student Name: " << student1.name;
return 0;
}
// Output: Student Name: Rosie
```

In the above example, we have used the `public` access modifier with the `name` variable. That's why we are able to assign a new value and access its value from the `main()` function.

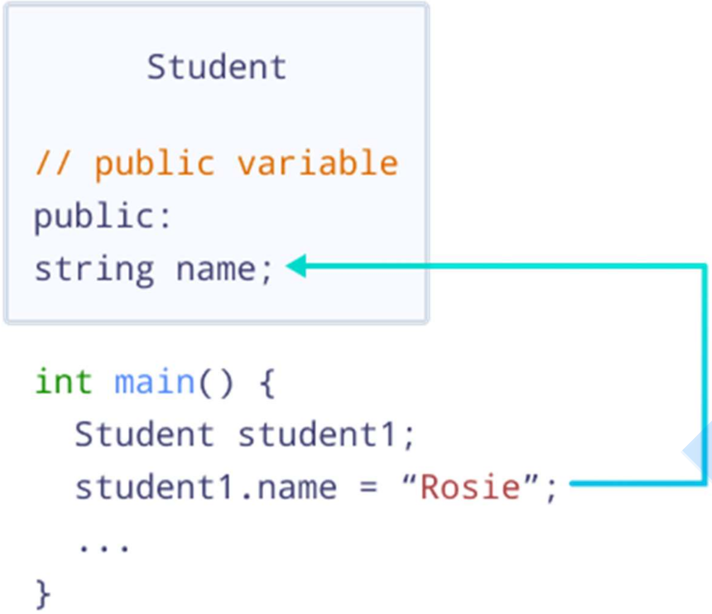

Figure: public Access Modifier

Public Functions

We can also use the `public` access modifier with member functions. Let's see an example.

```
#include <iostream>
using namespace std;
class Student {
// public member function
public:
void display_info() {
cout << "I am a Student";
}
};
int main() {
// create object of Student
Student student1;
// access the public member function
student1.display_info();
return 0;
}
// Output: I am a Student
```

As you can see, we are able to access the public member function of the `Student` class from the `main()` function.

Access Public Members From Another Class

In this example, we will try to access public class members of one class from another class.

```
#include <iostream>
using namespace std;
class Source {
// public data member
public:
double number = 200.56;
};
// class to access public members of Source
class Destination {
// public member function
public:
void access_source() {
// create an object of the Source class
Source src;
// access the member of Source
cout << "Data of Source: " << src.number;
}
};
int main() {
// create an object of Destination
Destination dest;
// call the function of destination
dest.access_source();
return 0;
}
// Output: Data of Source: 200.56
```

In the above example, we have created two classes: `Source` and `Destination`. Here, we are trying to access the public member variable (`number`) of `Source` from `Destination`.

The `access_source()` function of `Destination` first creates an object of the `Source` class and then accesses the data member.

```
void access_source() {
// create an object of the Source class
Source src;
// access the member of Source
cout << "Data of Source: " << src.number;
}
```

It's possible because `number` is a `public` data member inside `Source`.

Private Modifier

As mentioned earlier, if we create a variable with a `private` access modifier, it can't be accessed from outside. Let's see an example.

```
#include <iostream>
using namespace std;
class Student {
// create private variable
private:
string name;
};
int main() {
// create an object of Student
Student student1;
// try to access the private data member
student1.name = "Felix";
cout << "Name: " << student1.name;
return 0;
}
```

When we run this code, we will get an error:

```
std::string Student::name' is private within this context
17 | student1.name = "Felix"
```

Here, you can see that we get an error when we try to access the `private` variable `name` from the `main()` function.

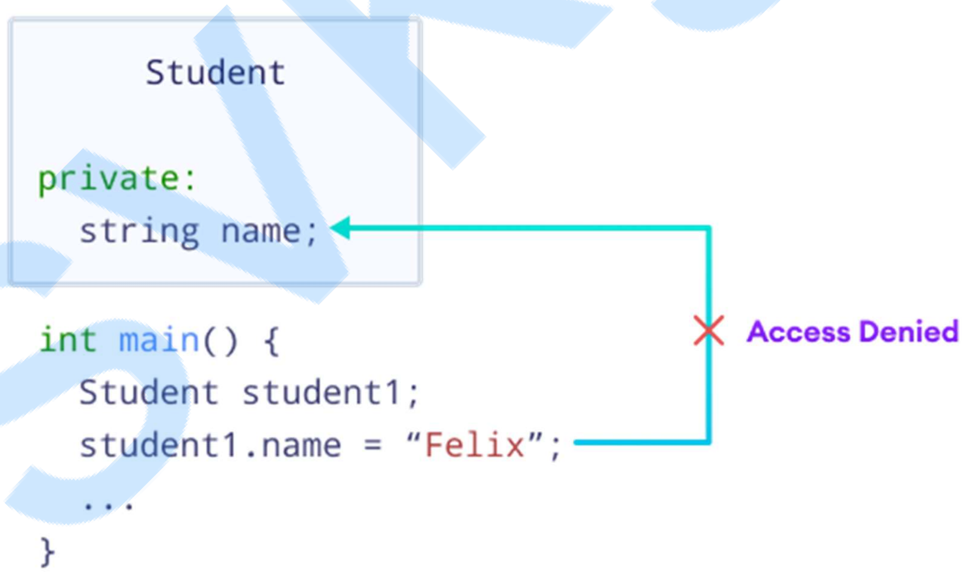

Figure: Private members cannot be accessed from outside the class

Private Functions

Just like a public function, we can also mark our function as private. Let's look at an example.

```
#include <iostream>
using namespace std;
class Student {
// create private function
private:
void display() {
cout << "This is it.";
}
};
int main() {
// create object
Student student;
// error: cannot access private function
student.display();
return 0;
}
```

When we run this code, we will get a familiar error:

```
error: 'void Student::display()' is private within this context
19 | student.display();
| ^
```

Private By Default

In C++, all class members are private by default (unless declared otherwise). So, the code

```
class Student {
private:
string name;
void display_info() {
...
}
};
```

is equivalent to

```
class Student {
string name;
void display_info() {
...
}
};
```

Let's see an example,

```
#include <iostream>
using namespace std;
class Student {
```

```
string name;
void display_info() {
cout << "Name: " << name;
}
};
int main() {
// create an object of Student
Student student1;
// access the private variable
student1.name = "Felix";
// access the private function
student1.display_info();
return 0;
}
```

Error Message

```
error: 'std::string Student::name' is private within this context
   19 |      student1.name = "Felix";

...

error: 'void Student::display_info()' is private within this context
   22 |      student1.display_info();
```

Getter and Setter Functions

We know that a private data member cannot be accessed from outside of a class. However, if we need to access them, we can use getter and setter functions.

- Setter Function - allows us to set the value of data members
- Getter Function - allows us to get the value of data members

Let's see an example.

```
#include <iostream>
using namespace std;
class Student {
private:
string name;
};
int main() {
// create an object of Student
Student student1;
// access the private name
student1.name = "Felix";
cout << "Name: " << student1.name;
return 0;
}
```

We know this code will cause an error because we are trying to directly access the private variable from the `main()` function.

Now let's use the getter and setter functions to access the `name` variable.

```
#include <iostream>
using namespace std;
class Student {
private:
string name;
public:
// setter function
void set_name(string student_name) {
name = student_name;
}
// getter function
string get_name() {
return name;
}
};
int main() {
// create an object of Student
Student student1;
// assign value to name using setter function
student1.set_name("Felix");
// access value of name using getter function
cout << "Name: " << student1.get_name();
return 0;
}
```

Output

```
Name: Felix
```

As you can see, we have successfully assigned a new value and accessed it using the getter and setter functions.

Next, we will see the working of this program.

Working: Getter and Setter Functions

In the last example, we created a class named `Student` with a private variable `name`.

```
private:
    string name;
```

Since it cannot be accessed from outside the class, we have used the public functions `get_name()` and `set_name()` to access them.

1. Setter Function

```
// setter function
void set_name(int student_name) {
    name = student_name;
}
```

Here, `student_name` is the parameter of the setter function `set_name()`. Then, we have assigned the value of this parameter to the private variable `name`.

2. Getter Function

```cpp
// getter function
int get_name() {
    return name;
}
```

Here, we have simply returned the value of the private variable `name`.

Inside the `main()` function, we are able to access and modify the `name` variable using these public functions.

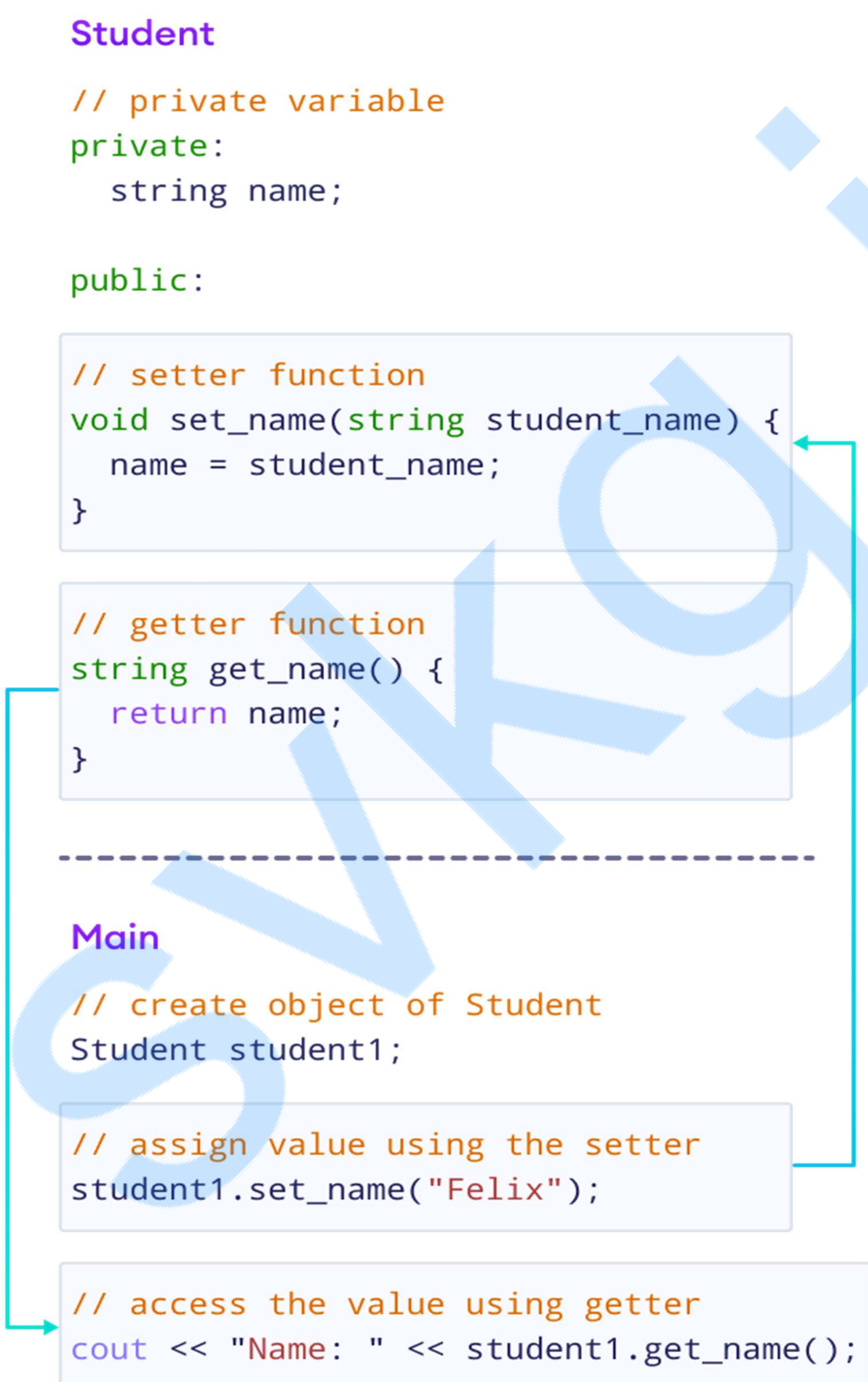

Figure: Getter and Setter functions can access private members

Constructors Should Be Public

We learned about constructors in the previous lesson. We know that constructors are called while creating an object of a class. So we should always make the constructor public.

Otherwise, we won't be able to create an object of the class. For example,

```
#include <iostream>
using namespace std;
class Student {
private:
// private constructor
Student() {
cout << "Private Constructor";
}
};
int main() {
// create an object of the Student class
Student student1;
return 0;
}
```

When we run this code, we will get an error:

```
error: 'Student::Student()' is private within this context
   16 |      Student student1;
```

This is because we have declared our constructor as private, so the compiler is not able to access it from the `main()` function while creating the object.

Hence, we should always make our constructors public.

Revise OOP (Basics)

Understanding OOP

C++ is an object-oriented programming language where we solve complex problems by dividing them into objects.

1. Create a Class

```
class Rectangle {
// code
};
```

Here, `Rectangle` is the name of the class. A class can contain data members such as variables (to store data) and member functions (to perform operations). Collectively, they are known as class members.

```
class Rectangle {
public:
// data members
int length, breadth;
// member function
void calculate_area(){
int area = length * breadth;
cout << "Area: " << area;
}
};
```

2. Create Objects

Here's how we create objects in C++.

```
Rectangle rectangle1;
```

Now we can use the `rectangle1` object to access the class members. For example,

```
#include <iostream>
using namespace std;
class Rectangle {
public:
// data members
int length, breadth;
// member function
void calculate_area(){
int area = length * breadth;
cout << "Area: " << area;
}
};
int main() {
// create object of the Rectangle class
Rectangle rectangle1;
// assign values to length and breadth
```

```
rectangle1.length = 12;
rectangle1.breadth = 5;
// call the member function
rectangle1.calculate_area();
return 0;
}
```

Output

```
Area: 60
```

C++ Constructor

A constructor is similar to a function but it doesn't have a return type. It has the same name as the class. For example,

```
class Rectangle {
public:
// constructor
Rectangle() {
...
}
};
```

Here, `Rectangle()` is a constructor.

Constructors that don't take any argument (such as `Rectangle()`) are known as default constructors.

Parameterized Constructor

A constructor can also accept parameters. For example,

```
#include <iostream>
using namespace std;
class Rectangle {
public:
// member variables
int length, breadth;
// parameterized constructor
// that accepts two parameters
Rectangle(int len, int br) {
length = len;
breadth = br;
}
// member function
void calculate_area() {
int area = length * breadth;
cout << "Area: " << area;
}
};
int main() {
// create object of the Rectangle class
// pass 12 and 5 as arguments to its constructor
```

```
Rectangle rectangle1(12, 5);
// call the member function
rectangle1.calculate_area();
return 0;
}
// Output: Area: 60
```

Here, you can see we have passed two arguments to the constructor while creating the `rectangle1` object.

Public and Private Modifiers

In C++, the `public` and `private` keywords are known as access modifiers.

Public Access Modifier

Class members declared as public can be accessed from outside the class (say, the `main()` function). For example,

```
#include <iostream>
using namespace std;
class Rectangle {
public:
// public member variables
int length, breadth;
// public member function
void calculate_area() {
int area = length * breadth;
cout << "Area: " << area;
}
};
int main() {
// create object of the Rectangle class
Rectangle rectangle1;
// access public member variables
rectangle1.length = 12;
rectangle1.breadth = 5;
// access public member function
rectangle1.calculate_area();
return 0;
}
// Output: Area: 60
```

Private Access Modifier

Class members declared as private cannot be accessed from outside the class. For example,

```
#include <iostream>
using namespace std;
class Rectangle {
private:
// private member variables
int length, breadth;
// private member function
```

```
void calculate_area() {
int area = length * breadth;
cout << "Area: " << area;
}
};
int main() {
// create object of the Rectangle class
Rectangle rectangle1;
// error: cannot access private member variables
rectangle1.length = 12;
rectangle1.breadth = 5;
// error: cannot access private member function
rectangle1.calculate_area();
return 0;
}
```

Getter and Setter Functions

We need to create public getter and setter functions in order to access private members of a class. For example,

```
#include <iostream>
using namespace std;
class Square {
private:
// private variable
int side;
public:
// setter function that assigns
// the value of the s parameter
// to the private variable side
void set_side(int s) {
side = s;
}
// getter function that returns
// the value of the private variable
int get_side() {
return side;
}
};
int main() {
// create object of the Square class
Square square1;
// call setter function
// initialize side to 6
square1.set_side(6);
// call getter function to calculate area
int area = square1.get_side() * square1.get_side();
// print the area
cout << "Area: " << area;
return 0;
}
// Output: Area: 36
```

OOP (Basics) Examples

In this section, we will create examples and solve challenges related to OOP.

Here is a list of programs we will create in this lesson:

- Compute the area of a circle
- Find the average marks of a student
- Challenge: determine pass or fail
- Get and set salary of Employee

Compute the Area of a Circle

In this example, we will first create a class named `Circle`. Inside the class, we will create:

- member variables - `pi` (with value 3.14) and `radius` (no initial value)
- constructor - initialize the value of `radius`
- member function - `calculate_area()` to compute the area of the circle

Note: The area of a circle is given by the formula `pi * radius * radius`.

Source Code

```
#include <iostream>

using namespace std;

class Circle {

public:

double pi = 3.14;

double radius;

// constructor to initialize radius

Circle(double rad): radius(rad) {}

// function to calculate area

double calculate_area() {

return pi * radius * radius;

}

};

int main() {

// create object of Circle

// pass a double value as argument

Circle circle(6.99);
```

```
// call calculate_area() function
cout << "Area: " << circle.calculate_area();
return 0;
}
// Output: Area: 153.421
```

In the above example, we have used the constructor initializer list to initialize the value of the `radius` variable.

```
Circle(double rad): radius(rad) {}
```

While creating the `circle` object, the value 6.99 is passed to the constructor.

```
Circle circle(6.99);
```

We then called the `calculate_area()` function to compute the area of the circle.

Find the Average Marks of a Student

In this example, we will find the average marks of a student using class and object. Here, we will first create a `Student` class.

The class will include

- an integer array named `marks` to store marks of the student
- a constructor to initialize the `marks` array.
- a `calculate_average()` member function to compute the average marks

Source Code

```
#include <iostream>
using namespace std;
class Student {
public:
// create marks array
int marks[4];
// constructor to initialize marks
Student(int mrk[4]) {
for(int i = 0; i < 4; ++ i) {
marks[i] = mrk[i];
}
}
// function to calculate the average
```

```
double calculate_average() {
int sum = 0;
// ranged loop to calculate sum
for(int num : marks) {
sum = sum + num;
}
return sum / 4.0;
}
};
int main() {
// initialize the marks array
int marks[4] = {96, 79, 81, 65};
// create Student object
// pass marks[] array as argument to constructor
Student student(marks);
// find the average marks
// call the calculate_average() function
double average = student.calculate_average();
// print the average marks
cout << average;
return 0;
}
// Output: 80.25
```

In this program, we have created a class named `Student`, which contains an integer array named `marks[]`.

We have then used the `Student()` constructor to initialize `marks[]`.

```
Student(int mrk[4]) {
for(int i = 0; i < 4; ++ i) {
marks[i] = mrk[i];
}
}
```

To calculate the average of the marks, we use the member function `calculate_average()`.

In `main()`, we initialized another `marks[]` array and passed it to the constructor of the `student` object as an argument.

```
// initialize the marks array
int marks[4] = {96, 79, 81, 65};
// create Student object by passing marks[] as argument
Student student(marks);
```

Finally, we have called the `calculate_average()` function, whose return value is stored in the `average` variable.

```
double average = student.calculate_average();
```

Get and Set Salary of Employee

Problem Description

Suppose a company increases the salary of every employee by a certain percentage. Create a program to calculate the salary of employees after the increment.

Thought Process

Here, we need to create an `Employee` class with three variables: `name`, `current_salary`, and `new_salary`. Since the company increases the salary by a certain percentage, we need to make the `new_salary` private, so that it cannot be modified randomly from outside of the class.

We will then use setter and getter functions to increase the salary by the specified percentage and return the increased salary.

Source Code

```
#include <iostream>
using namespace std;
class Employee {
private:
double new_salary;
public:
string name;
double current_salary;
// constructor
Employee(string emp_name, double emp_current_salary) {
name = emp_name;
```

```
current_salary = emp_current_salary;
}
// set new new_salary
void set_salary(double percentage) {
new_salary = current_salary + (percentage / 100.0) * current_salary;
}
// get new_salary
double get_salary() {
return new_salary;
}
};
int main() {
Employee emp1("Felix", 25213.23);
// increase salary by 20%
emp1.set_salary(20.00);
// print employee information
cout << "Name: " << emp1.name << endl;
cout << "New Salary: " << emp1.get_salary() << endl;
Employee emp2("Maria", 8732.32);
// increase salary by 30%
emp2.set_salary(30.00);
// print employee information
cout << "Name: " << emp2.name << endl;
cout << "New Salary: " << emp2.get_salary() << endl;
return 0;
}
```

Output

```
Name: Felix
New Salary: 30255.9
Name: Maria
New Salary: 11352
```

In the above example, we have increased the salaries of `Felix` and `Maria` by 20% and 30%, respectively.

Here, you can see we have declared `new_salary` as private, so it can only be initialized by the `set_salary()` function.

Inheritance

Inheritance

Inheritance Introduction

In the last chapter, we learned about object-oriented programming in C++. Now, let's learn about inheritance, which is a very important concept in OOP.

Let's create a scenario to understand what inheritance is and what problem it solves.

Why Inheritance?

Suppose we need to create a racing game with cars and motorcycles as vehicles.

To solve this problem, we can create two separate classes to handle each of their functionalities.

However, both cars and motorcycles are vehicles and they will share some common variables/arrays and functions.

So instead of creating two independent classes, we can create the Vehicle class that shares the common features of both cars and motorcycles. Then, we can derive the Car class from this Vehicle class.

In doing so, the `Car` class inherits all the variables and functions of the `Vehicle` class. And we can add car-specific features in the `Car` class.

Similarly, we can derive the `Motorcycle` class that inherits from the `Vehicle` class. Again, this `Motorcycle` class gets all vehicle-specific variables and functions from the `Vehicle` class, along with the unique features of motorcycles.

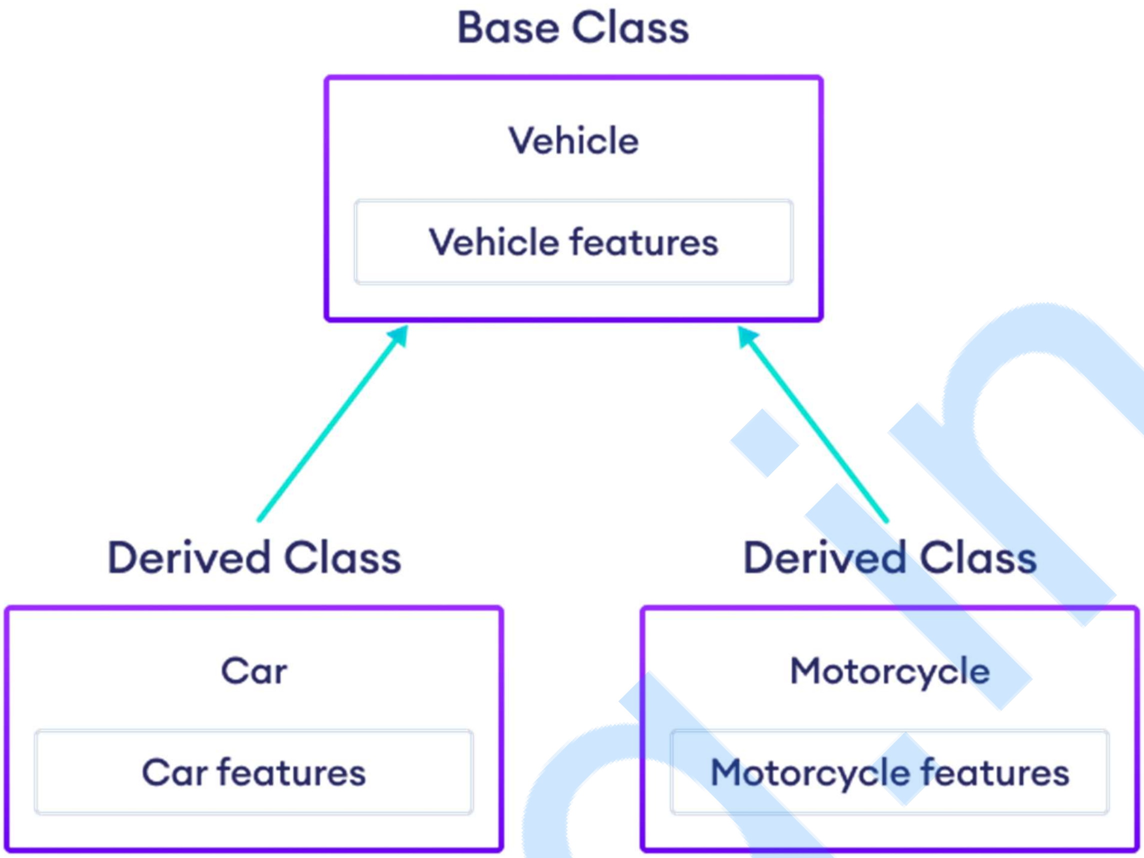

Figure: C++ Inheritance

This is the basic concept of inheritance. Inheritance allows a class (child or derived class) to inherit variables and functions from another class (parent or base class).

In our example, `Vehicle` is the superclass (also known as parent or base class) and `Car` and `Motorcycle` are subclasses (also known as child or derived classes).

Next, we will learn to implement inheritance in C++.

C++ Inheritance

Let's see an example of C++ inheritance.

Let's first create a class named `Animal`.

```
class Animal {
    public:

        void eat() {
            cout << "I can eat" << endl;
        }
};
```

Now, let's derive a class named `Dog` from this class.

```
// base class
class Animal {
    public:

        void eat() {
            cout << "I can eat" << endl;
        }
};

// the Dog class is derived from Animal
class Dog: public Animal {
    public:

        void bark() {
            cout << "I can bark" << endl;
        }
};
```

Here, we have used the code `class Dog: public Animal` to derive the `Dog` class from the `Animal` class. This insures that `Dog` will inherit all the variables and functions of `Animal`.

But what does that mean?

It means that objects of the `Dog` class can not only access variables and functions of the `Dog` class, but they can also access variables and functions of the `Animal` class.

Next, we will create objects of the `Dog` class.

Example: C++ Inheritance

Let's create an object of the `Dog` class and access the functions of `Animal`.

```
#include <iostream>
using namespace std;
// base class
class Animal {
public:
void eat() {
cout << "I can eat" << endl;
}
};
// the Dog class is derived from Animal
```

```
class Dog: public Animal {
public:
void bark() {
cout << "I can bark" << endl;
}
};
int main() {
// create object of Dog
Dog dog1;
// access the bark function of Dog
dog1.bark();
// access the eat() function of Animal
dog1.eat();
return 0;
}
```

Output

```
I can bark
I can eat
```

Here, `dog1` is an object of the `Dog` class. Hence,

- `dog1.bark()` calls the `bark()` function of the `Dog` class.
- `dog1.eat()` calls the `eat()` function of the `Animal` class. This can be done because `Dog` is derived from `Animal`, so the `Dog` class inherits all the variables and functions of `Animal`.

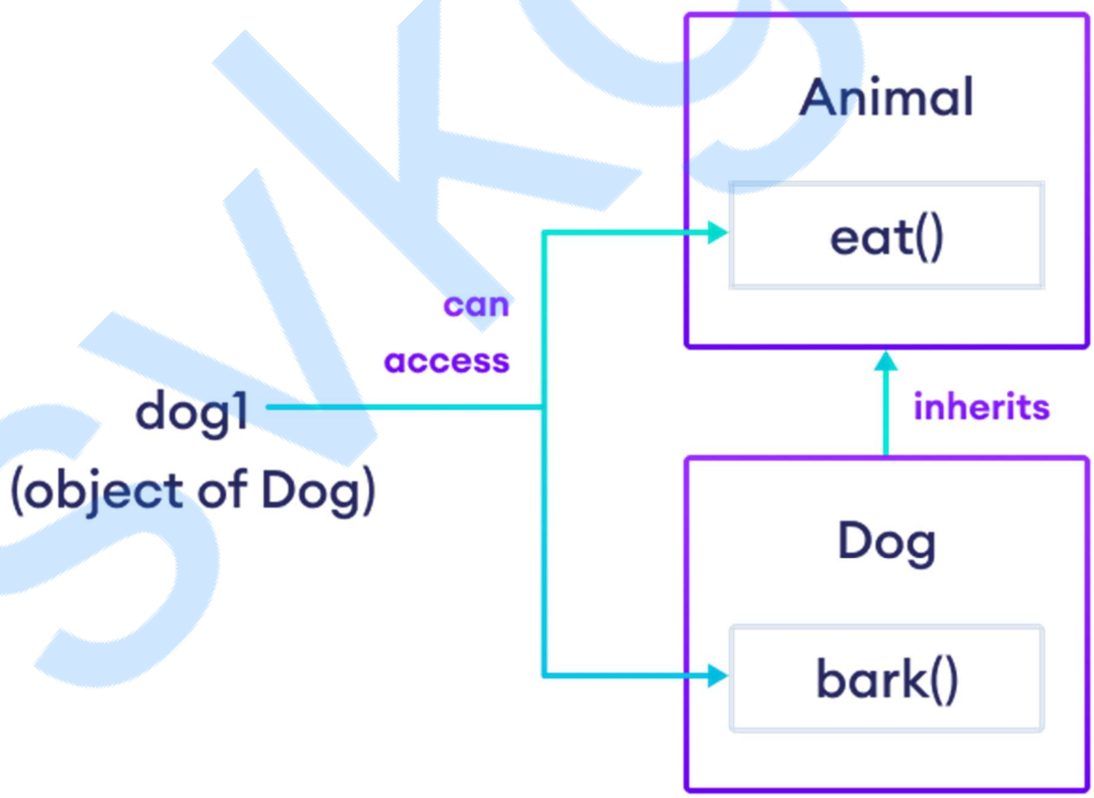

Figure: C++ Inheritance

Note: Objects of `Animal` can only access variables and functions of `Animal`. It's because `Dog` is derived from `Animal` and not the other way around.

Derive Multiple Classes

In C++, we can derive multiple classes from a single class. Let's look at an example.

```
#include <iostream>
using namespace std;
class Animal {
public:
void eat() {
cout << "I can eat" << endl;
}
};
// derive Dog from Animal
class Dog: public Animal {
public:
void bark() {
cout << "I can bark" << endl;
}
};
// derive Cat from Animal
class Cat: public Animal {
public:
void get_grumpy() {
cout << "I am getting grumpy" << endl;
}
};
int main() {
// object of Dog
Dog dog1;
// access member function of Dog class
dog1.bark();
// access member function of Animal class
dog1.eat(); // object of Cat
Cat cat1;
// access member function of Cat class
cat1.get_grumpy();
// access member function of Animal class
cat1.eat();
return 0;
}
```

Output

```
I can bark
I can eat
I am getting grumpy
I can eat
```

As you can see, the objects of the `Cat` class can also access functions of `Animal`. It's because `Cat` is also derived from `Animal`.

In this way, we can derive as many classes as we want from the superclass.

Inherit Class Variables

During inheritance, the child class can also inherit the member variables from the parent class. Let's see an example.

```
#include <iostream>
using namespace std;
class Family {
public:
// member variable of the Family class
string family_name = "Kennedy";
};
// the Person class inherits Family
class Person : public Family {
public:
string personal_name;
// this function uses the member variable of Family
void display_name() {
cout << personal_name << " " << family_name;
}
};
int main() {
// create an object of Person
Person person;
// assign value to the personal_name
person.personal_name = "John";
// call the display_name() function
person.display_name();
return 0;
}
```

Output

```
John Kennedy
```

In the above example, `family_name` is a variable of the `Family` class. However, we are able to use this inside the `Person` class because `Person` inherits the variable from `Family`.

Why Inheritance?

Inheritance allows us to reuse the same code in the base class. This helps save time and reduce bugs.

Tip: We should try to reduce duplicate code as much as possible. It's because if there are duplicate codes and we need to change something, then we have to change every duplicate code. This may result in inflexible code and bugs.

When to Use Inheritance?

While working on large projects, if there exists an is-a relationship between any two objects, we can use inheritance. For example,

- Dog is an Animal
- Car is a Vehicle
- Rectangle is a Polygon
- Triangle is a Polygon

It means,

- Dog can inherit from Animal
- Rectangle and Triangle can inherit from Polygon
- Car can inherit from Vehicle

Function Overloading

Introduction

In the previous lesson, we learned that during inheritance, the child class inherits functions and variables of the parent class. For example,

```
#include <iostream>
using namespace std;
class Animal {
public:
void make_sound() {
cout << "Animal Sound" << endl;
}
};
class Dog: public Animal {};
int main() {
Dog dog;
dog.make_sound();
return 0;
}
// Output: Animal Sound
```

Here, the `Dog` class doesn't have the `make_sound()` function. However, it is able to access it because of inheritance.

Now, suppose if the same function is also present in the Dog class, then what will happen?

Let's try that in our next example.

Function Overriding

```
#include <iostream>
using namespace std;
class Animal {
public:
// make_sound() function of base class
void make_sound() {
cout << "Animal Sound" << endl;
}
};
class Dog: public Animal {
public:
// make_sound() function of derived class
void make_sound() {
cout << "Woof Woof" << endl;
}
};
int main() {
// create object of child class Dog
Dog dog1;
// access function of Dog class
dog1.make_sound();
return 0;
}
// Output: Woof Woof
```

Here, the `make_sound()` function is present in both the `Dog` class and the `Animal` class. However, when we call the function using the `dog1` object, the function of the `Dog` class is executed.

This is because the function in the child class (`Dog`) overrides the same function in the parent class (`Animal`). And this process is known as function overriding.

Note: Function overriding only occurs when the function is called using an object of the derived class. When an object of the base class is used, the function of the base class is called.

Practical Use of Function Overriding

Now that we know about the basics of inheritance and function overriding, let's create a more practical example.

Program Description

In this example, we will create a program to calculate the perimeter of different polygons like triangles and quadrilaterals using inheritance.

- We will first create a `Polygon` class.
- Inside the `Polygon` class, we will create two functions: one to calculate the perimeter and the other to display the info of the polygon.
- Then, we will derive a `Triangle` class and a `Quadrilateral` class from it and add functions specific to these classes inside them.

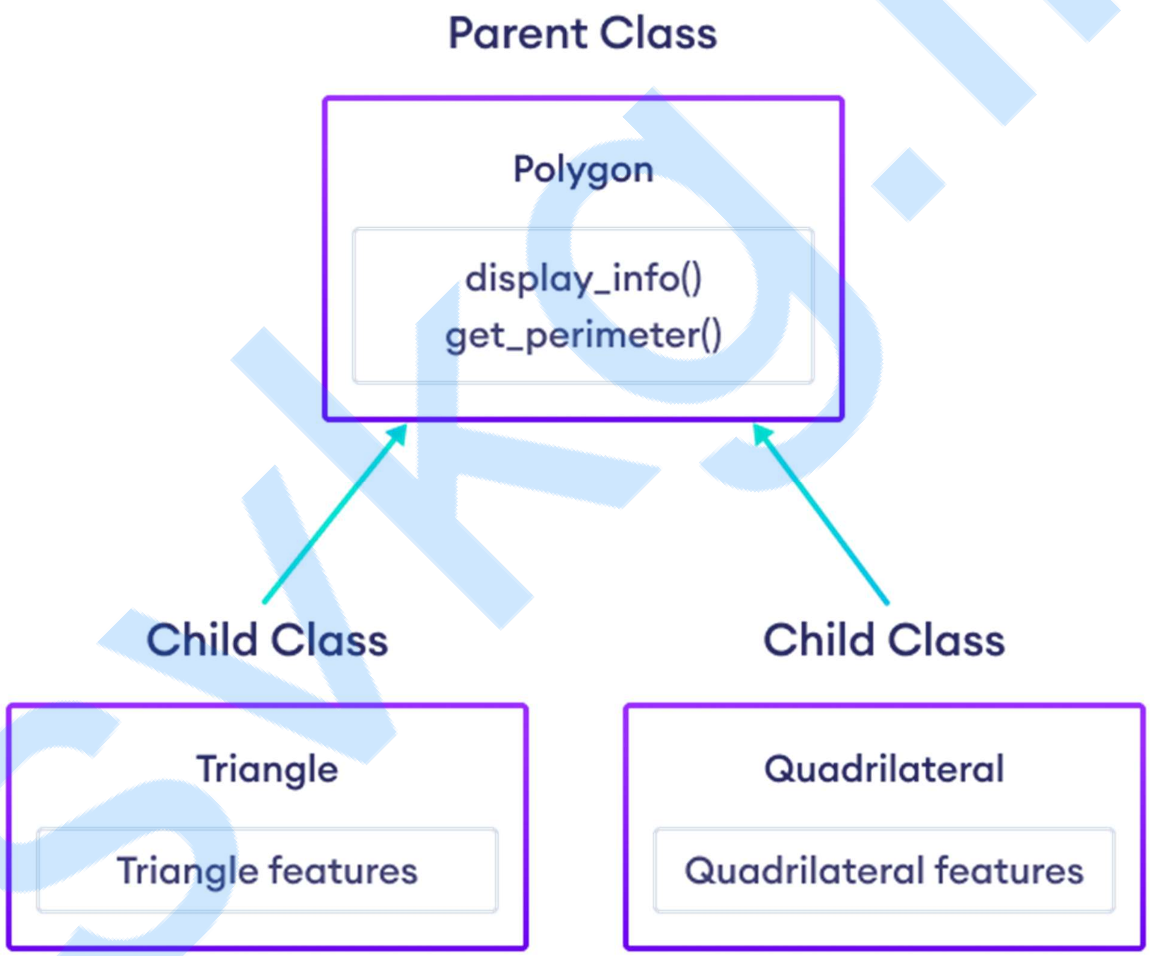

Figure: Practical Use of Function Overriding

Polygon Class

First, we will create the `Polygon` class.

```
#include <iostream>
using namespace std;
class Polygon {
public:
// variable to store total no. of polygon sides
int total_sides;
void display_info() {
cout << "A polygon is a two dimensional shape with straight lines."<< endl;
}
int get_perimeter(int sides[]) {
int perimeter = 0;
// find sum of all sides
for (int i = 0; i < total_sides; ++i) {
perimeter = perimeter + sides[i];
}
return perimeter;
}
};
int main() {
// create array of size 3
int sides[3] = {3, 4, 5};
// create Polygon object
Polygon p1;
// initialize total_sides variable for p1 object
p1.total_sides = 3;
// call the display_info() function
p1.display_info();
// call the get_perimeter() function
// pass the sides array as argument
int perimeter = p1.get_perimeter(sides);
// print the perimeter
cout << "Perimeter: " << perimeter;
return 0;
}
```

Output

```
A polygon is a two dimensional shape with straight lines.
Perimeter: 12
```

Here, the `get_perimeter()` function of the `Polygon` class takes an array of sides as its parameter. Inside the function, it computes the perimeter by adding all the sides (array elements).

```
int get_perimeter(int sides[]) {
int perimeter = 0;
// find sum of all sides
for (int i = 0; i < total_sides; ++i) {
perimeter = perimeter + sides[i];
}
return perimeter;
```

}

Inside this function, the size of the `sides[]` array is given by the `total_sides` variable.

You can see while calling `get_perimeter()`, we are passing the array `{2, 3, 5}` as an argument.

Next, we will inherit the `Triangle` class from `Polygon`.

Inheriting the Triangle Class

```
class Polygon {
    public:

        // variable to store total no. of polygon sides
        int total_sides;

        void display_info() {
            cout << "A polygon is a two dimensional shape with straight lines."<< endl;
        }

        int get_perimeter(int sides[]) {
            int perimeter = 0;

            // find sum of all sides
            for (int i = 0; i < total_sides; ++i) {
                perimeter = perimeter + sides[i];
            }

            return perimeter;
        }
};

class Triangle: public Polygon {
    public:

        // constructor to initialize total_sides
        Triangle() {
            total_sides = 3;
        }

        // function to override display_info() of Polygon
        void display_info() {
            cout << "A triangle is a polygon with 3 sides." << endl;
        }
};
```

Here, we have inherited the `Triangle` class from `Polygon`. We have also removed the code for creating objects.

If you have noticed, both the `Polygon` and `Triangle` classes have the same `display_info()` function.

Since a triangle always has 3 sides, we have used the `Triangle()` constructor to assign a value of 3 to the `total_sides` variable.

Next, we will create an object of the derived class `Triangle`.

Inheriting the Triangle Class (II)

Let's create an object of the `Triangle` class and call the `get_perimeter()` and `display_info()` functions.

```
#include <iostream>
using namespace std;
class Polygon {
public:
// variable to store total no. of polygon sides
int total_sides;
void display_info() {
cout << "A polygon is a two dimensional shape with straight lines."<< endl;
}
int get_perimeter(int sides[]) {
int perimeter = 0;
// find sum of all sides
for (int i = 0; i < total_sides; ++i) {
perimeter = perimeter + sides[i];
}
return perimeter;
}
};
class Triangle: public Polygon {
public:
// constructor to initialize total_sides
Triangle() {
total_sides = 3;
}
void display_info() {
cout << "A triangle is a polygon with 3 sides." << endl;
}
};
int main() {
// create an object of Triangle
Triangle t1;
// array to store sides of triangle
int triangle_sides[3] = {8, 5, 11};
// call display_info() function
t1.display_info(); // call get_perimeter using t1
int perimeter = t1.get_perimeter(triangle_sides);
cout << "Triangle Perimeter: " << perimeter;
return 0;
}
```

Output

```
A triangle is a polygon with 3 sides.
Triangle Perimeter: 24
```

Here, `Triangle t1;` creates an object of the `Triangle` class.

The code `t1.get_perimeter(triangle_sides)` calls the `get_perimeter()` function with the `triangle_sides[]` array as an argument.

Since `get_perimeter()` is not defined in `Triangle`, the `get_perimeter()` function of the `Polygon` class is called.

The code `t1.display_info()` calls the `display_info()` function. However, both the `Polygon` and `Triangle` classes have this function.

So, the function in `Triangle` is called because the function in the child class overrides the function in the parent class (function overriding).

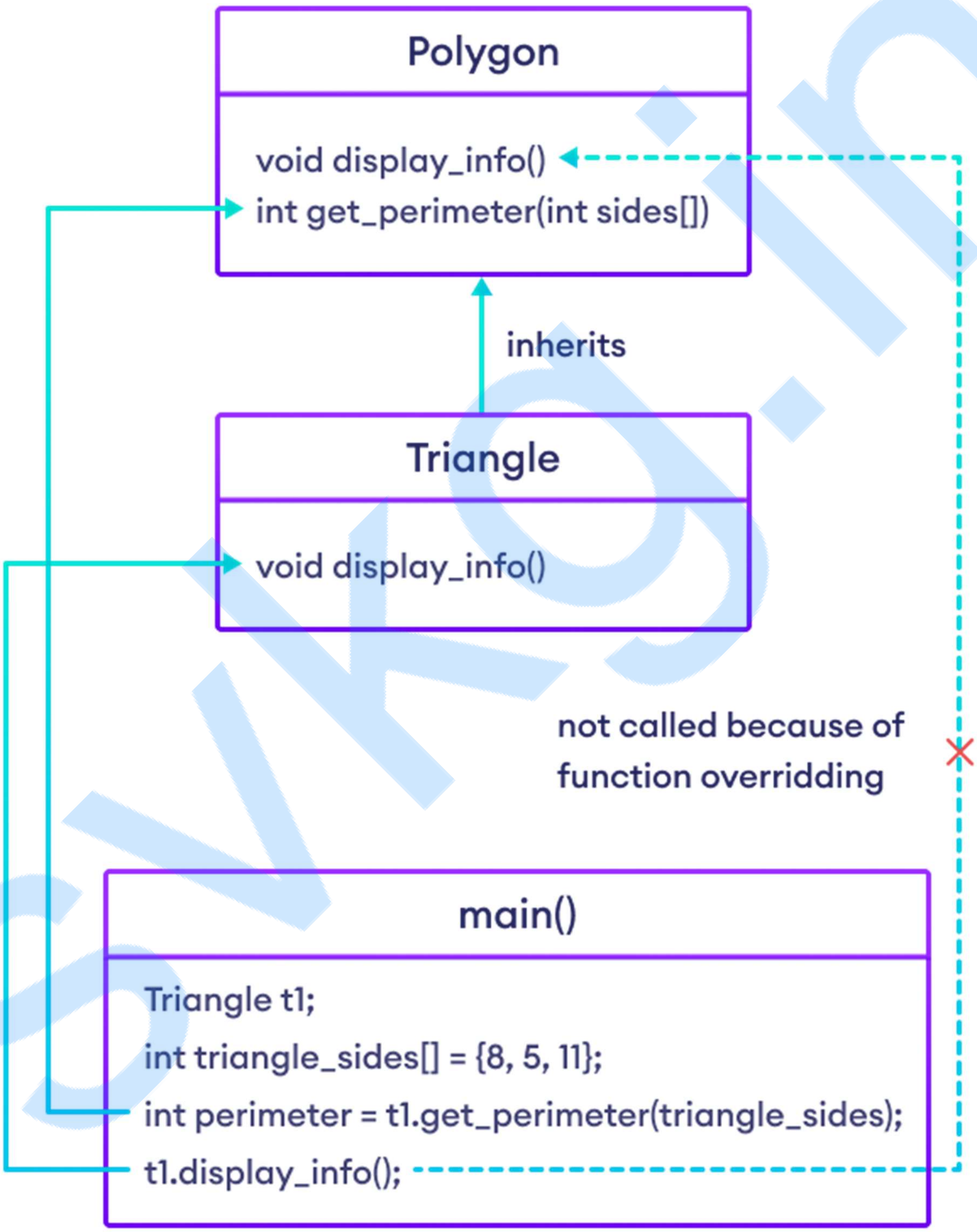

Figure: Function Overriding Example

Access the Parent Class Function

We know that when the same function is present in both the parent and child classes, the function in the child class overrides the function in the parent class.

However, what if we want to access the function of the base class as well?

One way to do that is to create an object of the base class itself and access the function using the object. For example,

```
#include <iostream>
using namespace std;
class Polygon {
public:
void display_info() {
cout << "A polygon is a two dimensional shape with straight lines." << endl;
}
};
class Triangle: public Polygon {
public:
void display_info() {
cout << "A triangle is a polygon with 3 sides." << endl;
}
};
int main() {
// create an object of Polygon
Polygon pol1;
// access the function of the base class
pol1.display_info();
return 0;
}
// Output: A polygon is a two dimensional shape with straight lines.
```

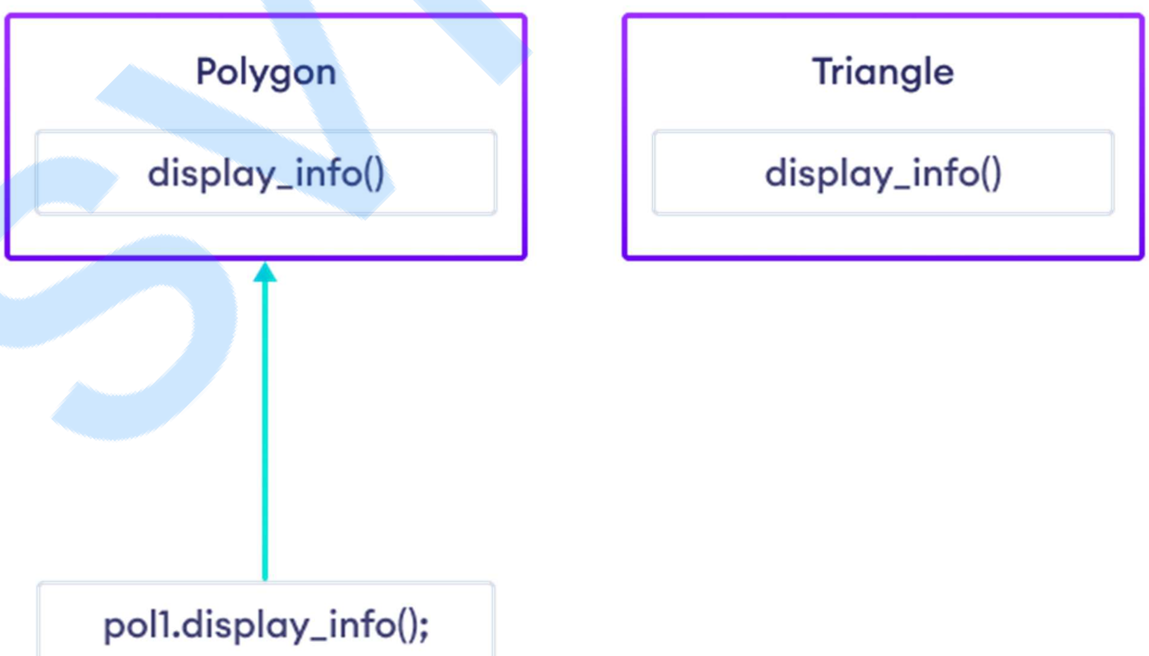

Figure: Access Function of Base Class

Here, we have used the object of the parent class (`Polygon`) to access the overridden function.

But what if we want to access the function of the base class using an object of the derived class itself?

Well, in that case, we can use the scope resolution operator `::`. along with the name of the base class `Polygon`.

There are two ways of using the scope resolution operator:

- inside the derived class
- with the object of the derived class

Let's see how we can use the scope operator inside the derived class.

Scope Resolution Operator

In this example, we will use the `::` operator inside the derived class to access the function of the parent class.

```
#include <iostream>
using namespace std;
class Polygon {
public:
void display_info() {
cout << "A polygon is a two dimensional shape with straight lines." << endl;
}
};
class Triangle : public Polygon {
public:
void display_info() {
// call the function of the base class
Polygon::display_info();
cout << "A triangle is a polygon with 3 sides." << endl;
}
};
int main() {
// create an object of Polygon
Triangle t1;
// access the function of the base class
t1.display_info();
return 0;
}
```

Output

```
A polygon is a two dimensional shape with straight lines.
A triangle is a polygon with 3 sides.
```

Notice the code inside the `display_info()` function of the `Triangle` class.

```
// inside the Triangle class
void display_info() {
Polygon::display_info();
cout << "A triangle is a polygon with 3 sides." << endl;
}
```

Here, `Polygon::display_info()` calls the function of the `Polygon` class.

:: Operator with Child Class Object

As mentioned before, we can also use the `::` operator alongside an object of the child class. However, we still need to specify the name of the parent class. For example,

```
#include <iostream>
using namespace std;
class Polygon {
public:
void display_info() {
cout << "A polygon is a two dimensional shape with straight lines." << endl;
}
};
class Triangle : public Polygon {
public:
void display_info() {
cout << "A triangle is a polygon with 3 sides." << endl;
}
};
int main() {
// create an object of Polygon
Triangle t1;
// access the function of the parent class
t1.Polygon::display_info();
// access the function of the child class
t1.display_info();
return 0;
}
```

Output

```
A polygon is a two dimensional shape with straight lines.
A triangle is a polygon with 3 sides.
```

Notice the following code inside the `main()` function.

```
// access the function of the parent class
t1.Polygon::display_info();
```

Here, the code will access the `display_info()` function of the parent class.

Revision: C++ Inheritance

Revise Inheritance

Let's revise what we have learned in this chapter.

1. C++ Inheritance

We use the `:` operator to inherit one class from another. For example,

```
class Person {
public:
void display() {
cout << "I am a person" << endl;
}
};
class Student: public Person {};
```

During inheritance, the child class inherits all the member variables and functions of the parent class.

```
#include<iostream>
using namespace std;
class Person {
public:
void display() {
cout << "I am a person" << endl;
}
};
class Student : public Person {};
int main() {
Student student1;
student1.display();
return 0;
}
// Output: I am a person
```

2. Function Overriding

If the same function is present in both the parent class and the child class, the function in the child class overrides the same function in the parent class.

```
#include<iostream>
using namespace std;
class Person {
public:
void display() {
cout << "I am a person" << endl;
}
};
```

```
class Student : public Person {
public:
void display() {
cout << "I am a student" << endl;
}
};
int main() {
Student student1;
student1.display();
return 0;
}
// Output: I am a student
```

3. The :: Operator

We can use the `::` operator to access the overridden function of the parent class.

```
#include<iostream>
using namespace std;
class Person {
public:
void display() {
cout << "I am a person" << endl;
}
};
class Student : public Person {
public:
void display() {
cout << "I am a student" << endl;
// access overridden function from inside the function
Person::display();
}
};
int main() {
Student student1;
student1.display();
// access overridden function using object
student1.Person::display();
return 0;
}
// Output:
// I am a student
// I am a person
// I am a person
```

Inheritance Example

Now, we will solve some examples to understand the concept of inheritance more clearly.

- Create a program to implement multilevel inheritance
- Challenge: Hierarchical Inheritance

Let's get started.

Example: Multilevel Inheritance

Suppose we have 3 classes: A, B, and C. In multilevel inheritance, the class B inherits from A and C inherits from B. Here's how the inheritance looks like:

Source Code

```
#include<iostream>

using namespace std;

class A {

public:

void function_A() {

cout << "Function of class A" << endl;

}

};

class B: public A {

public:

void function_B() {

cout << "Function of class B" << endl;

}

};

class C: public B {};

int main() {

// object of the class C

C obj;

// call function of the class B

obj.function_B();

// call function of the class A

obj.function_A();

return 0;

}
```

Output

```
Function of class B
Function of class A
```

Here, when the class `B` inherits `A`, the function of `A` is now inherited to `B`. That's why the object of the class `C` is able to access functions of both classes, `A` and `B`, even though it only inherits `B`.

OOP (Advanced)

Pointer and Object

Introduction

We dealt with the basics of OOP in the previous chapters. In this chapter, we shall deal with some advanced topics in object oriented programming.

We'll begin by exploring the use of pointers with objects. Let's begin!

C++ Pointer to Object

Before exploring the relationship between pointers and objects, Let's first revise the concept of pointers.

```
#include <iostream>
using namespace std;
int main() {
// create a variable
int number = 36;
// create a pointer variable
int* pt;
// assign address of number variable to pointer
pt = &number;
// print the address stored in pt pointer
cout << "Value of pt: " << pt << endl;
// print the address of the number variable
cout << "Address of number: " << &number;
return 0;
}
```

Output

```
Value of pt: 0x7ffeb13ed51c
Address of number: 0x7ffeb13ed51c
```

Here, we have created a pointer type variable `pt` and assigned the address of the `number` variable to it.

```
// integer type pointer
int* pt;
pt = &number;
```

Similarly, we can also create pointers to objects as well. For example, suppose we have a class `Student`. Then,

```
// create object of Student
Student student1;
```

```
// create Student pointer
Student* student_pointer;
// assign address of the student1 object
student_pointer = &student1;
```

Now, if we need to access the `marks` variable of the `student1` object, we can use the arrow operator `->`.

```
// set marks of student to 56
student_pointer->marks = 56;
```

This code is equivalent to

```
student1.marks = 56;
```

Let's look at this with an example.

Example: C++ Pointer to Object

```
#include <iostream>
using namespace std;
class Student {
public:
double marks;
};
int main() {
Student student1;
// create Student pointer
// assign address of student1 object to it
Student* ptr = &student1;
// set marks of student1 to 66.6
student1.marks = 66.6;
// print marks using pointer
cout << ptr->marks << endl;
return 0;
}
// Output: 66.6
```

Virtual Functions

Pointers and Function Overriding

We know that during function overriding, the child class overrides the same function in the parent class.

We can also use pointers to perform function overriding. Let's see an example,

```
#include <iostream>
using namespace std;
class Person {
public:
void display_info() {
cout << "I am a person." << endl;
}
};
class Student: public Person {
public:
void display_info() {
cout << "I am a student." << endl;
}
};
int main() {
Student student1;
// create Student pointer
Student* ptr = &student1;
// override the function of the parent class
ptr->display_info();
return 0;
}
// Output: I am a student.
```

In the above example, you can see that the `display_info()` function is present in both the parent class `Person` and the child class `Student`.

In `main()`, we have created a pointer to the `student1` object. When we call the function using this pointer, the function of the child class is called i.e. function overriding occurs.

However, if we create a pointer of the parent class (`Person`) that points to the address an object of the child class, the function overriding doesn't occur. Let's see an example,

```
#include <iostream>
using namespace std;
class Person {
public:
void display_info() {
cout << "I am a person." << endl;
}
};
class Student : public Person {
public:
void display_info() {
```

```
cout << "I am a student." << endl;
}
};
int main() {
Student student1;
// create Person pointer
// point to Student object
Person* ptr = &student1;
ptr->display_info();
return 0;
}
// Output: I am a person.
```

Notice the code,

```
Person* ptr = &student1;
```

Here, we are creating a pointer of the base class that points to the address of an object of the child class. So, when we call the `display_info()` function using this pointer, the function of the child class should be invoked.

Instead, the function of the parent class is invoked, so we get the output `I am a person.`.

We can solve this problem using the concept of virtual functions.

Reminder: The `->` operator is used to access class members using a pointer. On the other hand, the `.` operator is used to access members using objects.

C++ Virtual Functions

A virtual function is a member function of the parent class that should always be overridden by the child class. We use the `virtual` keyword to declare virtual functions in C++.

Let's see an example.

```
#include <iostream>
using namespace std;
class Person {
public:
virtual void display_info() {
cout << "I am a person." << endl;
}
};
class Student : public Person {
public:
void display_info() {
cout << "I am a student." << endl;
}
};
int main() {
Student student1;
// create Person pointer that points to student object
```

```
Person* ptr = &student1;
ptr->display_info();
return 0;
}
// Output: I am a student.
```

This program is similar to the earlier example. The only difference is that we have changed the normal function to a virtual function in the parent class.

This time, the `Person` type pointer to `student` is invoking the function of the child class, thus overriding the function in the parent class.

As you can see, the use of virtual functions allows us to achieve function overriding using pointers as well.

Pure Virtual Functions

So far, we have been creating functions like this:

```
void display_info() {
    cout << "I am a person" << endl;
}
```

Here, the code inside the curly braces `{}` is the body of the function.

In C++, we can also create functions that don't have a body. These types of functions are known as pure virtual functions.

Similar to virtual functions, the child class must override these functions, and we use the `virtual` keyword to create them. For example,

```
virtual void display_info() = 0;
```

Here, `display_info()` is a pure virtual function and you can see the function doesn't have a body. Instead, it's replaced by `= 0`.

You might be wondering what is the use of functions if they don't have any code inside them.

Well, there are some situations that require us to use pure virtual functions. Before learning about these cases, let's first talk about abstract classes.

Abstract Class

Normally, when we create a class, we can create objects from the class. For example,

```
class Animal {
// class body
};
// object of Animal
Animal obj;
```

Here, we are creating an object named `obj` of the `Animal` class.

In C++, we can also create abstract classes which contain pure virtual functions. For example,

```
// abstract class
class Polygon {
public:
// pure virtual function
virtual void get_area() = 0;
};
```

Here, `Polygon` is an abstract class because it includes the pure virtual function `get_area()`.

Unlike regular classes, we cannot create objects of an abstract class. Let's see what happens when we try to create an object of an abstract class.

```
#include <iostream>
using namespace std;
// abstract class
class Polygon {
public:
// pure virtual function
virtual void get_area() = 0;
};
int main() {
// create object of Polygon
Polygon obj;
return 0;
}
```

When we run this code, we will get an error.

```
error: cannot declare variable 'obj' to be of abstract type 'Polygon'
   14 |     Polygon obj;
      |             ^~~
```

This is because we are trying to create an object of the abstract class, which is not possible in C++.

Functions Inside the Abstract Class

Just like regular classes, an abstract class can have both regular functions and pure virtual functions. For example,

```
// abstract class
class Polygon {
    public:

        // regular function
        void print_sides() {
            cout << "Print sides of Polygon." << endl;
        }

        // pure virtual function
        virtual void get_area() = 0;
};
```

In the above example, we have created an abstract class that has

- a regular function named `print_sides()`
- a pure virtual function named `get_area()`

How to Use Abstract Classes and Pure Virtual Functions?

If we cannot create objects of an abstract class, then you might be wondering how we can access the functions inside it.

The answer is that in C++, we must inherit the abstract class to use it. For example,

```
// abstract class
class Polygon {
    public:

        // regular function
        void print_sides() {
            cout << "Print sides of Polygon." << endl;
        }

        // pure virtual function
        virtual void get_area() = 0;
};

class Rectangle: public Polygon {};
```

Here, the `Rectangle` class is inheriting the abstract class `Polygon`, hence it also inherits both the regular and pure virtual functions.

Now, the subclass must provide the implementation of all pure virtual functions, otherwise the subclass will be treated as an abstract class.

Once we provide the implementation of the pure virtual function, we can create objects of the subclass and access the functions, which we will see next.

Example: Abstract Class

```
#include <iostream>
using namespace std;
// abstract class
class Polygon {
public:
// regular function
void print_sides() {
cout << "Print sides of Polygon." << endl;
}
// pure virtual function
virtual void get_area() = 0;
};
class Rectangle: public Polygon {
public:
// implementation of the pure virtual function
void get_area() {
cout << "Print the area of Rectangle." << endl;
}
};
int main() {
// create object of the child class
Rectangle rectangle1;
// access the regular function of Polygon
rectangle1.print_sides();
// access the implemented pure virtual function
rectangle1.get_area();
return 0;
}
```

Output

```
Print sides of Polygon.
Print the area of Rectangle.
```

In the above example, we have created the `Rectangle` class by inheriting the abstract class `Polygon`.

The `Rectangle` class now inherits both the regular and pure virtual functions, so we must provide the implementation for the pure virtual function `get_area()`.

We then used an object of `Rectangle` to access functions of the abstract class.

Why Abstract Classes?

Suppose there is a function that is common among multiple entities. For example, all polygons have an area, and the function for calculating area can be shared among different types of polygons (rectangle, triangle, etc.).

However, the process of calculating the area of each polygon is different from one another. So, we cannot provide one implementation of calculating area that will work for all the polygons.

Instead, we can create a function without any implementation and all the polygons will provide their own implementation for the function.

For this, we use abstract classes with pure virtual functions and all the polygons implementing the class will provide their own version of the pure virtual function.

Let's see an example.

Example: Practical Use of Abstract Classes

```
#include <iostream>
using namespace std;
// abstract class
class Polygon {
public:
// pure virtual function
virtual double get_area() = 0;
};
class Rectangle: public Polygon {
public:
double length;
double breadth;
// initialize length and breadth
Rectangle(double len, double bread) : length(len), breadth(bread) {}
// implementation of the pure virtual function
double get_area() {
double area = length * breadth;
return area;
}
};
int main() {
// create object of the child class
Rectangle rectangle1(12.5, 8);
// access the implemented pure virtual function
double area = rectangle1.get_area();
cout << "Area of Rectangle: " << area;
return 0;
}
```

Output

```
Area of Rectangle: 100
```

In the above example, we have created an abstract class with a single pure virtual function named `get_area()`.

Here, the `Rectangle` class provides its own implementation of `get_area()` to compute the area of the rectangle.

Similarly, we can also inherit a `Triangle` class from `Polygon` which will provide its own implementation of the pure virtual function.

```
class Triangle : public Polygon {
```

```
public:
double base;
double height;
Triangle(double b, double h) : base(b), height(h) {}
double get_area() {
double area = 0.5 * base * height;
return area;
}
};
```

We can then use an object of the `Triangle` class to compute the area.

Go ahead and complete the code to compute the area of both rectangle and triangle.

Polymorphism

Introduction to Polymorphism

Polymorphism is another important concept in object-oriented programming. It simply means more than one form: the same entity (function or operator) can perform different operations in different scenarios.

Remember the working of the `+` operator? It can be used to perform numeric addition as well as string concatenation.

```
#include <iostream>
using namespace std;
int main() {
// use + to add two numbers
int result = 4 + 8;
cout << "Sum: " << result << endl;
string str1 = "Hello ";
string str2 = "World";
// use + to join two strings
string new_string = str1 + str2;
cout << new_string;
return 0;
}
```

Output

```
Sum: 12
Hello World
```

In the above example, we have used the same `+` operator to perform two different tasks:

- `4 + 8` - adds two numbers
- `str1 + str2` - joins two strings

Here, the `+` operator has two different forms. Thus, it is an example of C++ Polymorphism.

Polymorphism With Function Overriding

In function overriding, the same function is present in both the base class and the derived class.

```
// base class
class Animal {
public:
// make_sound() in the base class
void make_sound() {
cout << "Making animal sound" << endl;
}
};
// derived class
class Dog: public Animal {
public:
// make_sound() in the base class
void make_sound() {
cout << "Woof Woof" << endl;
}
};
```

In this case, we can independently access functions of the base class and derived class by using their respective objects. For example,

```
#include <iostream>
using namespace std;
class Animal {
public:
// make_sound() function of base class
void make_sound() {
cout << "Making animal sound" << endl;
}
};
class Dog: public Animal {
public:
// make_sound() function of derived class
void make_sound() {
cout << "Woof Woof" << endl;
}
};
int main() {
// access function of derived class
Dog dog1;
dog1.make_sound();
// access function of base class
Animal animal1;
animal1.make_sound();
return 0;
}
```

Output

```
Woof Woof
Making animal sound
```

As you can see, we are able to use the same function `make_sound()` to perform two different tasks.

Hence, we can say function overriding helps us achieve polymorphism in C++.

Note: Because Polymorphism includes function overriding, the related concepts of virtual functions and pure virtual functions are also examples of Polymorphism.

Polymorphism With Function Overloading

Let's understand function overloading first.

In C++, two or more functions can have the same name if they have different numbers/types of parameters. Let's see an example.

```
// function with no parameter
void display() {
    ...
}

// function with an integer parameter
void display(int number) {
    ...
}

// function with string parameter
void display(string name) {
    ...
}

// function with two parameters
void display(string name, int age) {
    ...
}
```

Here, we have created 4 functions with the same name `display()`, but different parameters. These functions are called overloaded functions and the process is called function overloading.

From the above explanation, it's clear that there are two ways to perform function overloading.

- With different numbers of parameters
- With different types of parameters

Let's see an example of both.

Overloading With Different Number of Parameters

```
#include <iostream>
using namespace std;
class Addition {
public:
// function with 2 parameters
void add_numbers (int num1, int num2) {
int sum = num1 + num2;
cout << "Sum of 2 digits: " << sum << endl;
}
// function with 3 parameters
void add_numbers(int num1, int num2, int num3) {
int sum = num1 + num2 + num3;
cout << "Sum of 3 digits: " << sum << endl;
}
};
int main() {
// create an object of Addition
Addition addition;
// call function with 2 arguments
addition.add_numbers(3, 5);
// call function with 3 arguments
addition.add_numbers(7, 9, 4);
return 0;
}
```

Output

```
Sum of 2 digits: 8
Sum of 3 digits: 20
```

In the above example, we have overloaded the `add_numbers()` function with 2 and 3 parameters.

Here, based on the number of arguments passed during the function call, the corresponding function is executed.

You can see we are able to use the same function `add_numbers()` for two different tasks. Hence, this helps in achieving Polymorphism.

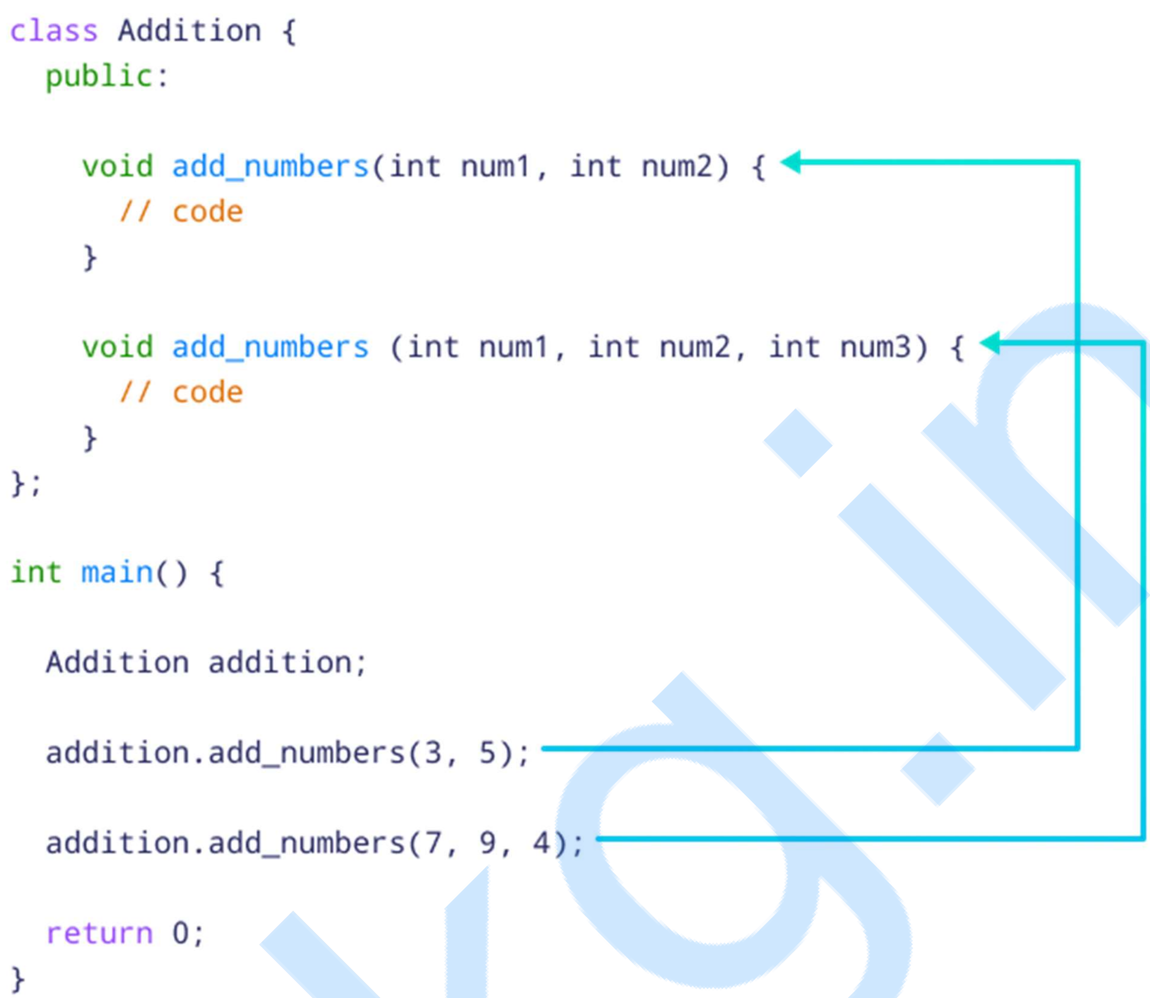

Figure: C++ Function Overloading

Overloading With Different Types of Parameters

Now, let's try function overloading with different parameter types. For example,

```
#include <iostream>
using namespace std;
class Addition {
public:
// function with integer parameters
int add_numbers (int number1, int number2) {
int sum = number1 + number2;
return sum;;
}
// function with double parameters
double add_numbers(double number1, double number2) {
double sum = number1 + number2;
return sum;
}
};
int main() {
```

```
// create an object of Addition
Addition addition;
// call function with integer arguments
int sum1 = addition.add_numbers(12, 9);
cout << "Sum of integers: " << sum1 << endl;
// call function with double arguments
double sum2 = addition.add_numbers(32.9, 43.7);
cout << "Sum of doubles: " << sum2 << endl;
return 0;
}
```

Output

```
Sum of integers: 21
Sum of doubles: 76.6
```

Here, we have overloaded the `add_numbers()` function with `int` and `double` parameters. Now, depending on the types of arguments passed during the function call, the corresponding function is executed.

As you can see, this example also uses the same function for two different purposes. Hence, this example is also an implementation of polymorphism.

Important! Function overloading is only associated with parameters, not their return types. Overloaded functions may have the same or different return types, as long as their parameters are different.

This is Not Function Overloading

Suppose we have two functions with the same name like this:

```
class Addition {
    public:

        // function with the void return type
        void add(int a, int b) {
            cout << a + b;
        }

        // function with the int return type
        int add(int a, int b) {
            return a + b;
        }
};
```

Here, we have two functions with the same name but different return types.

As mentioned earlier, function overloading is only associated with the number and type of parameters, not return types. This is not function overloading because both functions have the same parameters.

Hence, the above code will generate an error.

Remember: For function overloading, functions should have the same name and different parameters (different number of parameters, different types of parameters, or both).

Common Mistakes

Always remember that if we include multiple parameters of different types, we need to supply arguments in the same order while calling the function. Otherwise, it will either throw an error or result in a wrong output.

```
#include <iostream>
using namespace std;
class Multiplication {
public:
void multiply(int num1, int num2) {
int result = num1 * num2;
cout << result << endl;
}
void multiply(double num1, int num2, int num3) {
double result = num1 * num2 * num3;
cout << result << endl;
}
};
int main() {
// create an object of Addition
Multiplication product;
// supplying arguments in the wrong order
product.multiply (5, 6.5, 8);
return 0;
}
```

Expected Output

```
260
```

Actual Output

```
240
```

Here, we have overloaded the `multiply()` function: one function takes two integer parameters while the other takes a `double` parameter followed by two integer parameters.

However, we have called the second function by supplying the `double` argument after an integer argument.

```
// incorrect function call
// result = 5 * 6 * 8 = 240
product (5, 6.5, 8);
// correct function call
// result = 6.5 * 5 * 8 = 260
product (6.5, 5, 8);
```

As a result, we get the output 240 (`5 * 6 * 8`) instead of 260 (`6.5 * 5 * 8`).

Function Overloading Without Class

It's also possible to overload regular C++ functions without using classes. For example,

```
#include <iostream>
using namespace std;
// function with two parameters
void multiply(int number1, int number2) {
int result = number1 * number2;
cout << "Product of 2 numbers: " << result << endl;
}
// function with three parameters
void multiply(int number1, int number2, int number3) {
double result = number1 * number2 * number3;
cout << "Product of 3 numbers: " << result << endl;
}
int main() {
// call function with two arguments
multiply(5, 8);
// call function with three arguments
multiply(3, 6, 10);
return 0;
}
```

Output

```
Product of 2 numbers: 40
Product of 3 numbers: 180
```

As you can see, we have successfully implemented polymorphism without using a class.

However, this does not fall under OOP Polymorphism because we haven't used classes and objects.

Encapsulation

Encapsulation

Encapsulation is another key feature of object-oriented programming. It means bundling variables and functions together inside a class.

Let's understand this with the help of an example.

Suppose we need to compute the area of a rectangle. We know that to compute the area, we need two data (variables) - `length` and `breadth` - and a function - `calculate_area()`.

Hence, we can bundle these variables and the function together inside a single class.

```
class Rectangle {
    public:

        // variables to store data
        int length;
        int breadth;

        // function to calculate area
        int calculate_area() {
            int area = length * breadth;
            return area;
        }
};
```

This is an example of encapsulation.

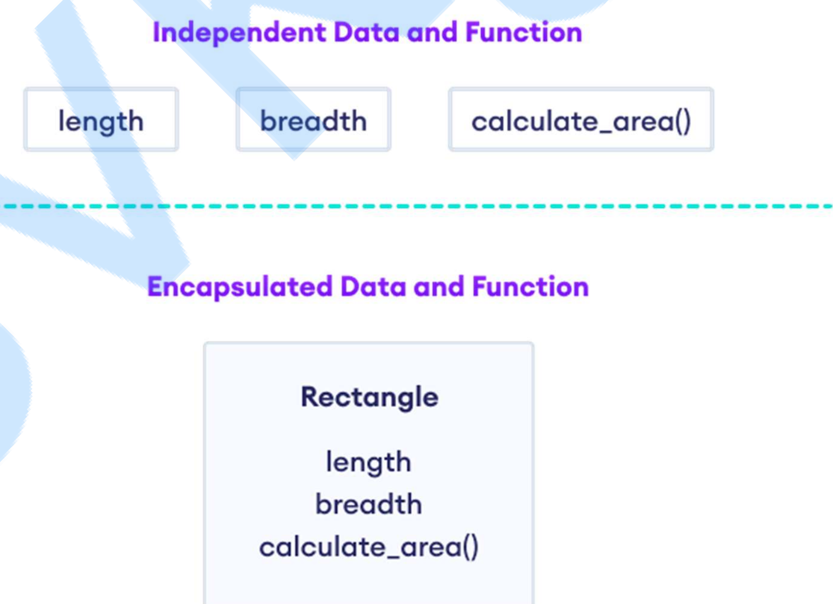

Figure: C++ Encapsulation

With this, we can now keep related variables and functions together, making our code clean and easy to understand.

Example: C++ Encapsulation

```
#include<iostream>
using namespace std;
class Rectangle {
public:
// variables to store data
int length;
int breadth;
// constructor to initialize variables
Rectangle(int l, int b): length(l), breadth(b) {}
// function to calculate area
int calculate_area() {
int area = length * breadth;
return area;
}
};
int main() {
// intialize value of length and breadth
Rectangle rect(12, 9);
// calculate the area
cout << "Area: " << rect.calculate_area();
return 0;
}
// Output:
// Area: 108
```

In the above example, we have created the `Rectangle` class to

- use the `length` and `breadth` variables to store data of a rectangle,
- calculate the area of the rectangle using the `calculate_area()` function.

Inside this class, we have used a constructor to initialize the value of `length` and `breadth`.

```
Rectangle(int l, int b): length(l), breadth(b) {}
```

Once the variables are initialized with the relevant data, we use the function below to calculate the area:

```
int calculate_area() {
int area = length * breadth;
return area;
}
```

Here, you can see the `calculate_area()` function uses `length` and `breadth` variables to compute the area of the rectangle. Both these variables are also present inside the same class.

This is what encapsulation is all about: bundling the related data and function together.

Data Hiding in C++

Data hiding prevents the access of variables and functions of a class from other classes. It is one of the most important benefits of encapsulation.

In our previous example, we can make both `length` and `breadth` variables private.

```
class Rectangle {
private:
// variables to store data of rectangle
int length;
int breadth;
};
```

Now these variables cannot be accessed from outside the class. In order to access these variables, we need to use getter and setter functions.

```
#include <iostream>
using namespace std;
class Rectangle {
private:
// private variables to store data
// the data in these variables is hidden from outside the class
int length;
int breadth;
public:
// function to initialize value of length
void set_length(int len) {
length = len;
}
// function to initialize value of breadth
void set_breadth(int br) {
breadth = br;
}
// function to calculate area
int calculate_area() {
int area = length * breadth;
return area;
}
};
int main() {
// create object
Rectangle rect;
// initialize the value of length and breadth
rect.set_length(12);
rect.set_breadth(9);
// calculate the area
cout << "Area: " << rect.calculate_area();
return 0;
}
```

In the above example, we have used the setter functions:

- `set_length()` - to initialize the value of the private variable `length`
- `set_breadth()` - to initialize the value of the private variable `breadth`

Notes:

- We are not trying to access `length` and `breadth` variables, hence we haven't included the getter functions in our program.
- We can also initialize these variables using a constructor. But it's preferable to use setter functions to initialize private variables.

Here, other classes won't be able to directly access `length` and `breadth`. By making these variables `private`, we have restricted unauthorized access from outside the class.

This is an example of Data Hiding.

Why Encapsulation?

With encapsulation, we can control what types of data our variables will store.

Suppose we want to get an `age` input for the `Person` class. Initially, we can mark `age` as private so that no one can directly modify it from outside the class.

```
class Person {
private:
int age;
};
```

We know the only way to initialize the variable of `age` is by using a setter function.

Inside the function, instead of directly assigning the value, we can use a condition that checks whether `age` is greater than 0 and less than 100.

```
class Person {
private:
int age;
public:
// setter function
void set_age(int person_age) {
if (person_age > 0 && person_age < 100) {
age = person_age;
}
}
};
```

With this, we are now able to control what value the `age` variable stores.

Let's complete the program.

```
#include <iostream>
using namespace std;
class Person {
private:
int age;
public:
// setter function
void set_age(int person_age) {
if (person_age > 0 && person_age < 100) {
age = person_age;
}
else {
cout << "Invalid age" << endl;
// terminate the program
exit(0);
}
}
// getter function
int get_age() {
return age;
}
};
int main() {
// create object of Person
Person person;
int age;
// get input value for age
cout << "Enter your age: ";
cin >> age;
// initialize the value of age
person.set_age(age);
// get value of age
cout << "Age: " << person.get_age();
return 0;
}
```

Sample Output 1

```
Enter your age: 25
Age: 25
```

Sample Output 2

```
Enter your age: 0
Invalid age
```

Here, the program only assigns the value to `age` if it is greater than 0 and less than 100. Otherwise, the program will be terminated after informing the user that the `age` input is invalid.

This way, with the help of encapsulation, we can control our program by not letting users enter an invalid age.

Why Data Hiding?

Not all data inside a class are meant to be universally accessible. It is very important to hide some of the data from other functions and classes in our program.

For instance, consider a class called `Bank_Account` that allows the program to store the bank details of different people. Naturally, many of the details are confidential and should only be accessible to a select few.

But if our program gives public access to these crucial details, then anyone using our program can tamper with sensitive information.

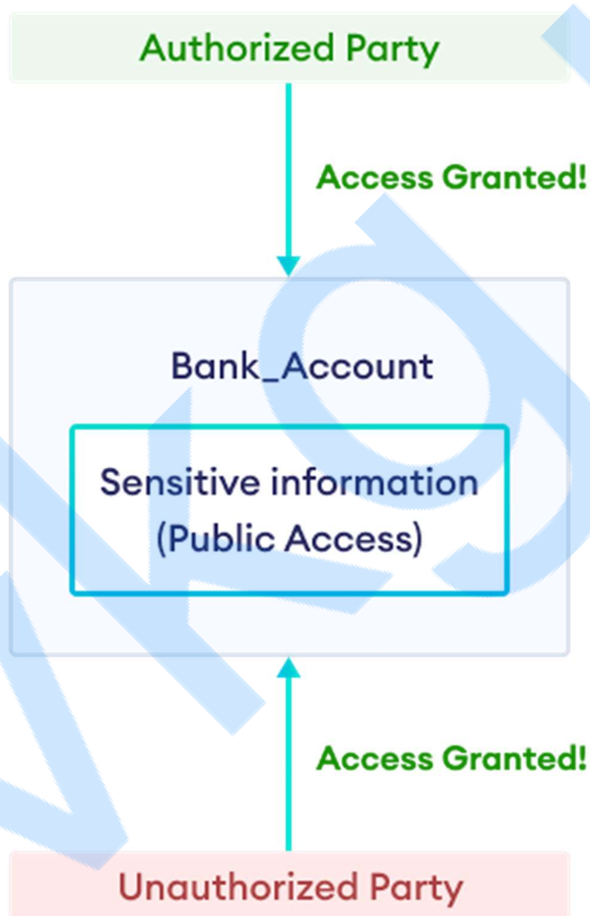

Figure: Public data can be accessed by unauthorized parties

To prevent this, object-oriented programming languages such as C++ have integrated a very crucial feature into their system: data hiding.

Data hiding refers to restricting access to data members of a class. As we have discussed earlier, this is to prevent other functions and classes from tampering with the class data.

That's why it is important to declare sensitive variables `private` so that unauthorized users don't get access to these variables.

Revise OOP (Advanced)

Major OOP Concepts

C++ is an object-oriented programming language, and so far, we have learned about the major concepts of OOP.

- Inheritance
- Abstraction (pure virtual functions and abstract classes)
- Polymorphism
- Encapsulation

We have already covered everything about inheritance in our last chapter.

Now, let's revise the other 3 with the help of the following examples:

- Implement brakes in Motor_Bike class
- Overloading the Payment Process of eCommerce
- Compute the Area of Circle Using Encapsulation

Let's get started.

Implement Brakes in Motor_Bike Class

In this example, we will look into the practical implementation of abstraction using motorbike brakes.

Thought Process

We know that the purpose of a brake is to stop the motorbike. However, the working of brakes is different for different types of bikes.

So, instead of creating separate functions for each type of motor bike, we can simply create a pure virtual function `brake()` to allow for different implementations.

Now, all the other types of bikes will provide a separate implementation of brakes that suits their needs. However, the actual working of the brake will remain the same (to stop the bike).

Source Code

```cpp
#include <iostream>
using namespace std;
// abstract class
class Motor_Bike {
public:
// pure virtual function
virtual void brake() = 0;
};
class Sports_Bike: public Motor_Bike {
public:
// provide an implementation of brake
void brake() {
cout << "Stopping the Sports Bike." << endl;
}
};
class Mountain_Bike: public Motor_Bike {
public:
// provide an implementation of brake
void brake() {
cout << "Stopping the Mountain Bike." << endl;
}
};
int main() {
Sports_Bike s1;
s1.brake();
Mountain_Bike m1;
m1.brake();
return 0;
}
```

Output

```
Stopping the Sports Bike.
Stopping the Mountain Bike.
```

In the above example, we have created an abstract class `Motor_Bike` with a pure virtual function named `brake()`.

Here, `Sports_Bike` and `Mountain_Bike` classes inherit `Motor_Bike` and provide an implementation for the pure virtual function.

Overloading the Payment Function of eCommerce

In this example, we will implement the payment method for an eCommerce business.

Thought Process

We know that every eCommerce business has different payment methods like credit card payment, PayPal payment, and many more.

Each payment method requires different information. For example, we will need a card number, CVV, and expiry date to pay through a credit card. Similarly, to pay through PayPal, we need the PayPal ID.

So let's implement this same logic and overload the payment function with different numbers of parameters.

Source Code

```
#include <iostream>

using namespace std;

class Payment {

public:

// pay through credit card

void make_payment(string card_number, string cvv, string expiry_date) {

cout << "Payment Through Credit Card is Successful." << endl;

}

// pay through PayPal

void make_payment(string id) {

cout << "Payment Through PayPal is Successful." << endl;

}

};

int main() {

Payment pay;

// make the payment through credit card

pay.make_payment("4324 7651 3232 8723", "532", "12/029");
```

```
// make the payment through Paypal
pay.make_payment("8925832997");
return 0;
}
```

Output

```
Payment Through Credit Card is Successful.
Payment Through PayPal is Successful.
```

Here, we have implemented the behavior of polymorphism (through function overloading) in C++.

Note: For simplicity, we are directly providing the values for function arguments. It is good practice to get input for these values.

Compute the Area of Circle Using Encapsulation

In this example, we will compute the area of a circle using the concepts of encapsulation. Here, we will use the following formula.

```
Area of a circle = 3.14 * radius * radius
```

Thought Process

From the formula, we know that we need the radius to calculate the area. Here, we will be using the private variable `radius`, so that it cannot be directly modified from outside of the class.

```
class Area {
private:
double radius;
};
```

We will then use the setter and getter functions to modify and access the value of `radius` from outside.

```
public:
// setter function
void set_radius(double rad) {
// function body
}
double get_radius() {
```

```
// function body
}
```

However, inside the setter function, we will use an `if` condition to prevent `radius` from being negative.

```
void set_radius(double rad) {
if (rad > 0) {
radius = rad;
}
else {
cout << "Error! Radius is negative" << endl;
}
}
```

Now, this code ensures that `radius` can't be negative.

Let's complete this program.

```
#include <iostream>
using namespace std;
class Area {
private:
double radius;
public:
// setter function
void set_radius(double rad) {
if (rad > 0) {
radius = rad;
}
else {
cout << "Error!! Radius is negative" << endl;
}
}
// getter function
double get_radius() {
return radius;
```

```
}
};
int main() {
// get input value for radius
double radius;
cout << "Enter the value of radius: ";
cin >> radius;
Area area;
// set value of radius
area.set_radius(radius);
// access the value of radius and compute the area
double circle_area = 3.14 * area.get_radius() * area.get_radius();
cout << "Area of circle: " << circle_area;
return 0;
}
```

Sample Output 1

```
Enter the value of radius: 12
Area of circle: 452.16
```

Sample Output 2

```
Enter the value of radius: -12
Error!! Radius is negative
Area of circle: 0
```

The program only assigns the input value to `radius` if it is not negative. Otherwise, the default value of 0.0 will be assigned to `radius`.

This way, we can control the program by not letting users enter negative values.

Templates

Introduction to Templates

If you think back on how we use functions and classes, you will realize that we are limited by the data type of the variables and arrays we define inside our function/class. For instance, consider the following class and function:

```
class Sample_Class {
    public:
        int var1;
        double var2;
};
```

```
int sample_function (int num1, int num2) {
    int result = (num1 * num2 * 5) / 11;
    return result;
}
```

From just a glance at the code above, we can easily conclude that

- `Sample_Class` can only work with integer and `double` data, and
- `sample_function()` can only work with integer data.

But what if we could define classes and objects to work with almost any data type?

One option to achieve this is using overloading, but that would require us to define the function multiple times. And we can't overload classes in C++.

What we want is to define the class or function only once, and then let that single class/function work with all sorts of data types.

Fortunately, C++ has provided us with an incredibly useful tool to do just that: templates. This is a powerful feature that allows us to write generic programs i.e. programs that include codes that can work with any data type.

There are two ways we can implement templates:

- Function Templates
- Class Templates

Let's start with function templates.

Note: We can also create templates of structures i.e. `struct`. However, we will not learn about them in this course. Instead, let's just say that `struct` templates are somewhat similar to class templates, and leave it at that.

Function Templates

Function templates are generic functions that can work with multiple data types. For example,

```
template <typename T>
T add(T num1, T num2) {
   return (num1 + num2);
}
```

Here, we have created a function template named `add()`. The template definition consists of the following parts:

- `template` - keyword used to declare a function template
- `typename` - keyword that is part of the function template syntax
- `T` - template argument that represents the data type

Now we can use this function with any type of data.

1. Working with int data

```
// call function template with int data
add<int> (2, 3);
```

Here, the template argument `T` will be `int` and `num1` and `num2` will be 2 and 3 respectively.

2. Working with double data

```
// call function template with double data
add<double> (5.56, 9.34);
```

In this case, the template argument `T` will be `double` and `num1` and `num2` will be 5.56 and 9.34 respectively.

Note: We can also omit the data type while calling a function template. For example, `add(2, 3)` and `add(5.56, 9.34)`. However, it is a good practice to include the data type during the function call.

Example: Function Template

```
#include <iostream>
using namespace std;
template <typename T>
T add(T num1, T num2) {
return num1 + num2;
}
int main() {
// call function template with int data
int result1 = add<int>(2, 3); // call function template with double data
double result2 = add<double>(5.56, 9.34);
```

```
cout << "2 + 3 = " << result1 << endl;
cout << "5.56 + 9.34 = " << result2 << endl;
return 0;
}
```

Output

```
2 + 3 = 5
5.56 + 9.34 = 14.9
```

As you can see, we are able to use the same function to work with both the integer data and `double` data.

Class Templates

Similar to functions, we can also create class templates to work with different types of data. For example,

```
template <class T>
class Number {
    public:
        T var1;
        T var2;
};
```

Notice that we have used the keyword `class` instead of `typename` in the syntax above. We can also use the keyword `typename` instead.

```
template <typename T>
class Number {...};
```

So don't get confused. We will be using class for all our examples.

Now, we can use this class to work with any type of data by creating objects with the appropriate data type. For example,

```
// object that works with integer data
Number<int> integer_object;
```

```
// object that works with double data
Number<double> double_object;
```

Note: Unlike with function templates, we must supply the data type of the parameters when creating objects of class templates.

```
// error: missing template arguments
Number integer_object;
```

Example: Class Templates

```
#include <iostream>
using namespace std;
// class template
template <class T>
class Multiplication {
public:
// variable of type T
T multiplier;
// constructor initializer list
Multiplication(T multi) : multiplier(multi) {}
// function that returns product of
// multiplier variable and the num argument
T multiply(T num) {
return num * multiplier;
}
};
int main() {
// create object with int data
Multiplication<int> num_int(3);
int result1 = num_int.multiply(9);
// create object with double data
Multiplication<double> num_double(5.7);
double result2 = num_double.multiply(13.2);
cout << "Product with int: " << result1 << endl;
cout << "Product with double: " << result2 << endl;
return 0;
}
```

Output

```
Product with int: 27
Product with double: 75.24
```

In this program, we have created a template class:

```
template <class T>
class Multiplication {
...
};
```

The class contains

- `multiplier` - a variable
- `Multiplication()` - constructor to initialize `multiplier`
- `multiply()` - function to calculate product of `multiplier` and its parameter `num`

Let's look at how this program works.

1. Working With int Data

```
Multiplication<int> num_int(3);
```

```
int result1 = num_int.multiply(9);
```

Here, we have created an object of `Multiplication` to work with `int` data. As you can see from the code above:

- `multiplier` is 3
- the argument given to `multiply()` is 9 i.e. `num == 3`
- the return value of `multiply()` is `9 * 3` i.e. 27.

2. Working With double Data

```
Multiplication<double> num_double(5.7);
double result2 = num_double.multiply(13.2);
```

Here, we have created an object of `Multiplication` to work with `double` data. As you can see from the code above:

- `multiplier` is 5.7
- the argument given to `multiply()` is 13.2 i.e. `num == 13.2`
- the return value of `multiply()` is `5.7 * 13.2` i.e. 75.24.

Why Templates?

1. Code Reusability

We can write code that will work with different types of data. For example,

```
int add(int num1, int num2) {
    return num + num2;
}
```

Here, the function only works if we pass `int` data to this. If we want to perform addition of `double` values, we have to create another function.

However, with templates, we can use one function and use it with any type of data.

```
template <typename T>
T add(T num1, T num2) {
    return num1 + num2;
}
```

2. Type Checking

The template parameter, `T`, provides information about the type of data used in the template code. For example,

```
Template_Class<string> obj("Hello");
```

Here, this object will only work with `string` data. Now, if we try to pass a value other than string, we will get an error.

Constructor Overloading

Introduction

C++ allows us to have two or more functions with the same name if they have different parameters (type and number). This process is known as function overloading. For example,

```
#include <iostream>

using namespace std;

class Multiplication {

public:

// function with 2 parameters

void multiply(int num1, int num2) {

int product = num1 * num2;

cout << "Product of 2 numbers: " << product << endl;

}

// function with 3 parameters

void multiply(int num1, int num2, int num3) {

int product = num1 * num2 * num3;

cout << "Product of 3 numbers: " << product << endl;

}

};

int main() {

Multiplication obj;

// call function with 3 arguments

obj.multiply(2, 8, 6);

// call function with 2 arguments

obj.multiply(4, 7);

return 0;

}
```

Output

```
Product of 3 numbers: 96
Product of 2 numbers: 28
```

In the above example, we have overloaded the `multiply()` function to work with different numbers of parameters.

Constructor Overloading

Similarly, we can also overload constructors in C++ to perform different actions based on different parameters.

But first let's revise the different types of constructor available in C++.

C++ Constructors

Basically, a constructor is like a member function of a class that has the same name as the class but no return type. A constructor is automatically called when we create an object of the class. For example,

```
#include <iostream>
using namespace std;
class Sample {
public:
// default constructor with no arguments
Sample() {
cout << "Object created!" << endl;
}
};
int main() {
// create an object of the Sample class
Sample sample1;
return 0;
}
// Output: Object created!
```

Here, `Sample()` is a constructor of the `Sample` class and is called automatically the moment we create the `sample1` object. It is a default constructor since it takes no arguments.

Parameterized Constructor

Constructors can also take parameters. For example,

```
#include <iostream>
using namespace std;
class Sample {
public:
```

```
// constructor with integer parameter
Sample (int num) {
cout << "Constructor Parameter: " << num << endl;
}
};
int main() {
// create object of Sample
// supply 9 as argument to its constructor
Sample sample(9);
return 0;
}
// Output:
// Constructor Parameter: 9
```

Constructor Overloading

Similar to function overloading, overloaded constructors have the same name (name of the class) but different numbers or types of arguments.

Let's see an example.

```
#include <iostream>
using namespace std;
class Sample {
public:
// default constructor with no arguments
Sample() {
cout << "Default constructor!" << endl;
}
// parameterized constructor with an integer argument
Sample (int num) {
cout << "Second Constructor Parameter: " << num << endl;
}
// constructor with 2 parameters
Sample (int num1, double num2) {
cout << "Third Constructor Parameters: ";
cout << num1 << " and " << num2 << endl;
}
};
int main() {
// call the default constructor
Sample sample1;
// call the constructor with a single int argument
Sample sample2(9);
// call the constructor with two arguments
```

```
    Sample sample3(9, 9.5);
    return 0;
    }
```

Output

```
Default constructor!
Second Constructor Parameter: 9
Third Constructor Parameters: 9 and 9.5
```

Here, we have overloaded 3 constructors in the `Sample` class:

- `Sample()` - a default constructor with no parameters
- `Sample(int num)` - a parameterized constructor with an integer parameter
- `Sample(int num1, double num2)` - a parameterized constructor with two parameters: one integer and one `double`.

We can call the desired constructor by supplying the appropriate argument(s) when creating objects of the class. The image below shows how:

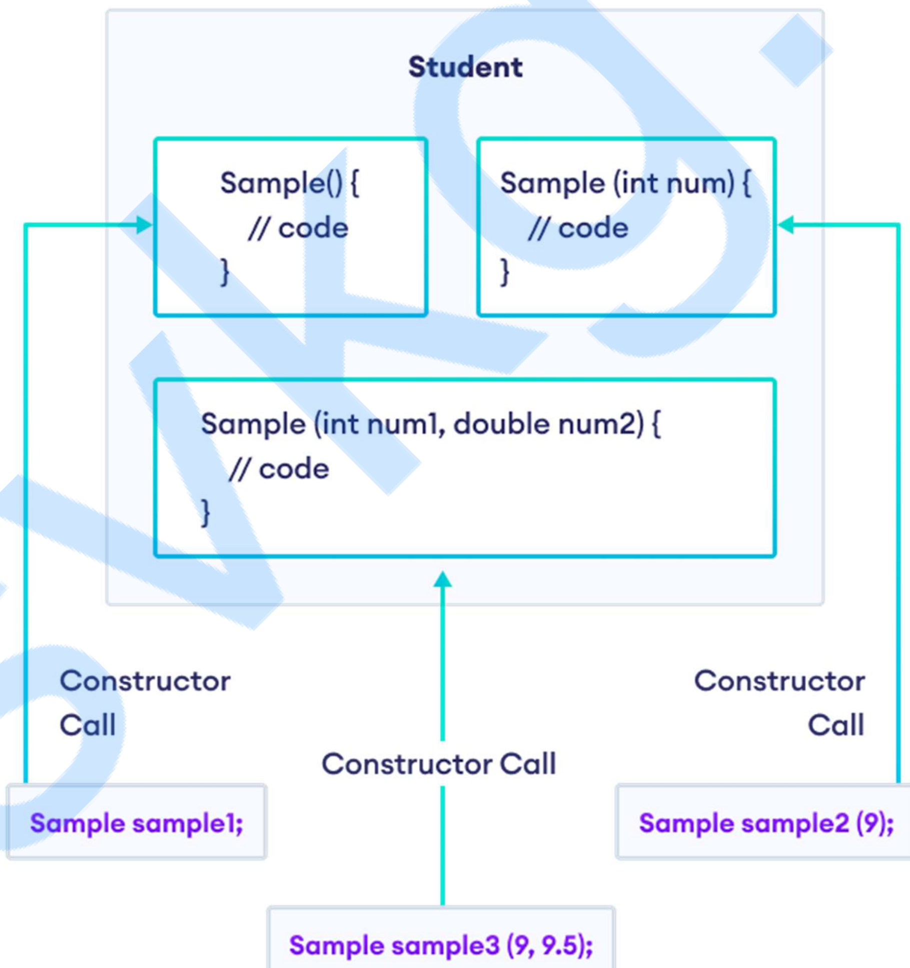

Figure: C++ Constructor Overloading

Example: Constructor Overloading

```
#include <iostream>
using namespace std;
class Person {
private:
int age;
public:
// 1. Constructor with no arguments
Person() {
age = 20;
}
// 2. Constructor with an integer argument
Person(int a) {
age = a;
}
// getter function
int get_age() {
return age;
}
};
int main() {
// call default constructor
Person person1;
// call parameterized constructor
Person person2(45);
cout << "Person1 Age = " << person1.get_age() << endl;
cout << "Person2 Age = " << person2.get_age() << endl;
return 0;
}
```

Output

```
Person1 Age = 20
Person2 Age = 45
```

In this program, we have created a class `Person` that has a single variable `age`.

We have also defined two constructors:

1. Person() Constructor

- default constructor i.e. accepts no argument
- called while creating the object `person1` because we haven't passed any argument
- initializes `age` to 20

2. Person(int a) Constructor

- parameterized constructor with parameter `a`
- called while creating the object `person2` because we have passed 45 as an argument
- initializes `age` to parameter `a` i.e. 45

The function `get_age()` returns the value of `age`, and we use it to print the `age` of `person1` and `person2`.

Why Overload Constructors?

A lot of the times, we may want to initialize objects in different ways. Sometimes, we may want an object to have default values for its member variables.

At other times, we may want to initialize the members with different values. This can easily be achieved through constructor overloading.

So, with constructor overloading, we can make our classes and objects more dynamic and flexible. It can also make our code shorter and look more clean.

Imagine having to assign custom values to different objects. Without constructor overloading, we'd have to either assign the values using the `.` operator:

```
object1.variable1 = value1;
object1.variable2 = value2;
```

```
object2.variable1 = value3;
object2.variable2 = value4;
```

Or we'd have to rely on setter functions to assign those values:

```
object1.set_variable1(value1);
object1.set_variable2(value2);
```

```
object2.set_variable1(value3);
object2.set_variable2(value4);
```

With constructor overloading, we can condense these four lines of codes into two, while also having the freedom to initialize an object with default values:

```
// objects with custom values
Sample_Class object1(value1, value2);
Sample_Class object2(value3, value4);
```

```
// objects with default values
Sample_Class object3, object4;
```

As you can see, this process is far less tedious and is much easier on the eyes. So it is always a good idea to overload constructors if our program demands flexibility with its classes.

C++ Static Keyword

static Keyword

So far, we have been using an object of the class to access variables and functions of a class. For example,

```
#include <iostream>
using namespace std;
class Animal {
public:
void display() {
cout << "I am an animal." << endl;
}
};
int main() {
// object of the Animal class
Animal obj;
// access the function using the object
obj.display();
return 0;
}
```

Output

```
I am an animal.
```

Here, we have used the object `obj` of the `Animal` class to access the member function `display()`.

However, there might be situations where we want to access variables and functions without creating the object. For this,we can use the `static` keyword.

Let's see an example.

Example: static Keyword

```
#include <iostream>
using namespace std;
class Animal {
public:
// static function
static void display() {
cout << "I am an animal." << endl;
}
};
int main() {
// access the function using class
Animal::display();
return 0;
}
// Output: I am an animal.
```

Here, you can see that we are able to directly access the `display()` function using the class name with the scope resolution operator `::`.

```
Animal::display();
```

Notice that we haven't created an object for this purpose. This is possible because we have declared the function as `static`.

Static Member Variables

Unlike static functions, static member variables are declared inside the class and defined outside the class. For example,

```
class Student {

public:

// static variable declaration

static int subject_code;

};

// static variable definition

int Student::subject_code = 13;
```

In the above example, we have created the static variable `subject_code`.

Here, you can see we have declared the static variable inside the class; however, we have provided its definition outside the class.

Access static Variables

Like with static functions, we can use the class name with the scope resolution operator `::` to access static variables. For example,

```
#include <iostream>

using namespace std;

class Student {

public:

// static variable declaration

static int subject_code;

};

// static variable definition and initialization

int Student::subject_code = 13;

int main() {

// access static variable
```

```
cout << Student::subject_code;

return 0;

}

// Output: 13
```

You can see that we have successfully accessed the static variable without creating an object of the class.

Change static Variable

We can also change the value of static variables once it's defined. For example,

```
#include <iostream>
using namespace std;
class Student {
public:
// static variable declaration
static int subject_code;
static string subject;
};
// static variable definition and initialization
int Student::subject_code = 13;
// static variable definition
string Student::subject;
int main() {
// access static variable
cout << "Initial subject_code: " << Student::subject_code << endl;
// change the subject_code variable
Student::subject_code = 15;
cout << "Final subject_code: " << Student::subject_code << endl;
// initialize the subject variable
Student::subject = "Physics";
// print the subject variable
cout << "Subject: " << Student::subject;
return 0;
}
```

Output

```
Initial subject_code: 13
Final subject_code: 15
Subject: Physics
```

In this program, we have created two static variables - `subject_code` and `subject` - inside the `Student` class.

Notice how we have defined the static variables outside the class:

```
// static variable definition and initialization
int Student::subject_code = 13;
// static variable definition
```

```
string Student::subject;
```

Here, we have initialized `subject_code` to 13 but we have not initialized `subject`. This is because we can change and also initialize variables later once they've been defined.

```
// inside the main() function
// change the subject_code variable
Student::subject_code = 15;
// initialize the subject variable
Student::subject = "Physics";
```

The above code:

- changes the value of `subject_code` to 15
- initializes `subject` to `"Physics"`

Why static?

While implementing OOP, we may be faced with situations where all the objects of a class need to share common data. In such cases, we store such data in static variables.

When we declare a static variable, all objects of the class share the same static variable. The static variables and functions belong to the class (rather than objects). And we don't need to create objects of the class to access the static variables and functions.

```
#include <iostream>
using namespace std;
class Company {
public:
static string name;
};
// static variable definition
string Company::name;
int main() {
Company::name = "Programiz";
cout << "Name: " << Company::name;
return 0;
}
```

Figure: Working of static variables

Here, the static variable `name` is common to all objects of the class `Company`.

However, when we declare a non-static variable, all objects will have separate copies of the variable.

```
#include <iostream>
using namespace std;
class Company {
public:
string name;
};
int main() {
Company object1;
Company object2;
object1.name = "Programiz";
object2.name = "Programiz PRO";
cout << "Name for object1: " << object1.name << endl;
cout << "Name for object2: " << object2.name;
return 0;
}
```

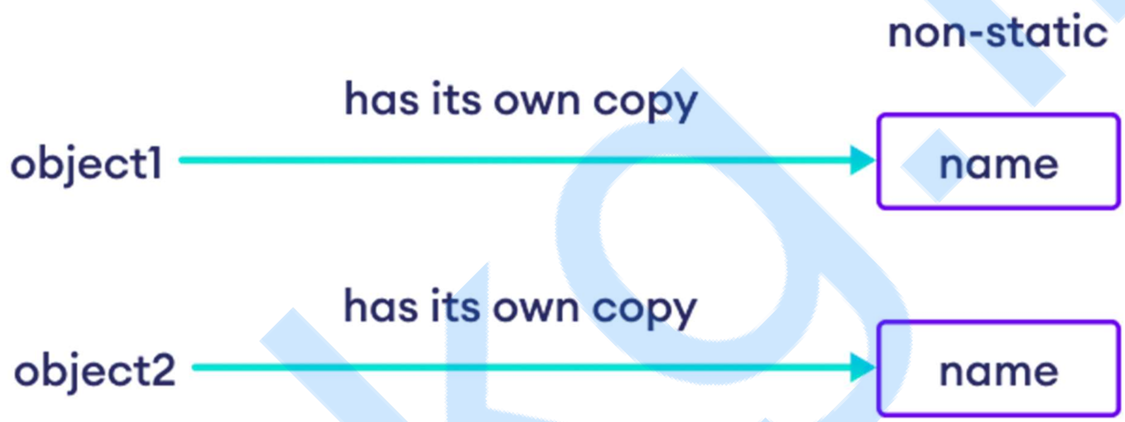

Figure: Working of non-static variables

Here, both `object1` and `object2` will have separate copies of the variable `name`. And they are different from each other.

Example: Practical Use of static

In this example, we will get employee details of a company using static and non-static members.

Thought Process

In this program, we will use OOP to get the names of the employees of a single company. This means that

- the name of the company will be the same for all employees,
- the name of each employee will be different.

Thus, when we create the `Employee` class, we can simply associate the `company_name` variable with the class instead of associating it with individual objects. This means that we need to declare `company_name` as static.

On the other hand, the `employee_name` variable will be non-static because each employee has a different name. Thus, each object will have its own copy of `employee_name`.

In `main()`, we will initialize the static variable. Then, we will create two `Employee` objects, initialize their `employee_name` variables, and then print all the static and non-static members of the objects.

Source Code

```
#include <iostream>

using namespace std;

class Employee {

public:

// static variable

static string company_name;

// non-static variable

string employee_name;

};

// define and initialize static variable

string Employee::company_name = "Microsoft";

int main() {

// create Employee objects

Employee employee1, employee2;

// get user input for employee_name of the objects

cout << "Enter Employee1 Name: ";

getline(cin, employee1.employee_name);

cout << "Enter Employee2 Name: ";

getline(cin, employee2.employee_name);

// print the variables

cout << "-------------------------------------------------------" << endl;

cout << "Company Name: " << Employee::company_name << endl;

cout << "Employee1 Name: " << employee1.employee_name << endl;

cout << "Employee2 Name: " << employee2.employee_name;

return 0;

}
```

Output

```
Enter Employee1 Name: Chris Rock
Enter Employee2 Name: Will Smith
----------------------------------------------------------
Company Name: Microsoft
Employee1 Name: Chris Rock
Employee2 Name: Will Smith
```

As you can see, `company_name` is static, while `employee_name` is not. So, we must define and initialize `company_name` outside the class.

```
// define static variable outside the class

string Employee::company_name = "Microsoft";
```

In `main()`, we have created two objects: `employee1` and `employee2`. We then took user input for the `employee_name` variables of these objects. Finally, we printed both the static and non-static members for these objects.

Static Functions and Non-Static Variables

In the previous challenge, we printed a static variable inside a static function. Now, let's see what happens if we try to print a non-static variable inside a static function.

```
#include <iostream>
using namespace std;
class Company {
public:
// non-static variable
string name = "Programiz";
// create a static function
static void print_name() {
// print non static variable
cout << name;
}
};
int main() {
// call the static function
Company::print_name();
return 0;
}
```

Output

```
error: invalid use of member 'Company::name' in static member function
   13 |                 cout << name;
```

As you can see, we got an error message from the C++ compiler. The compiler tells us that we cannot use non-static variables inside a static function.

So, if we are to make our program work, we'll have to either declare:

- both `name` and `print_name()` as static, or
- both `name` and `print_name()` as non-static, or
- `name` as static and `print_name()` as non-static

Tip: Try rewriting this program using all 3 of the above options and compare the results.

Next, we'll see what happens when we try to access static members using objects.

Note: Non-static functions can use both static and non-static variables.

Static Members and Objects

So far, we have accessed static members using the class name. But we can also access static members using objects. For example,

```
#include <iostream>
using namespace std;
class Square {
public:
static int length;
};
int Square::length = 5;
int main() {
// create Square objects
Square square1, square2;
cout << "Initially," << endl;
// print static variable using the objects
cout << "square 1 length = " << square1.length << endl;
cout << "square 2 length = " << square2.length << endl;
// change length using square1
square1.length = 4;
cout << "\nAfter changing square1 length," << endl;
cout << "square 1 length = " << square1.length << endl;
cout << "square 2 length = " << square2.length << endl;
// change length using square2
square2.length = 7;
cout << "\nAfter changing square2 length," << endl;
cout << "square 1 length = " << square1.length << endl;
cout << "square 2 length = " << square2.length;
return 0;
}
```

Run Code >>

Output

```
Initially,
square 1 length = 5
square 2 length = 5

After changing square1 length,
```

```
square 1 length = 4
square 2 length = 4

After changing square2 length,
square 1 length = 7
square 2 length = 7
```

In this program, we have created two objects `square1` and `square2` from the `Square` class. Then, we used these objects to access the static variable `length`.

Here, we are able to access static variables using objects and both objects give the same value of `length`.

Also, changing the `length` using one object also changes the `length` of the other object. This is because static members are common to all objects, something you already know very well by now.

However, we strongly advise you to not use objects to access static members.

Revision: static Keyword

Let's revise what we have learned in this lesson.

1. static functions

We can access static functions using the class name and the `::` operator. For example,

```
#include <iostream>
using namespace std;
class Square {
public:
static void find_area(int length, int breadth) {
int square = length * breadth;
cout << "Square: " << square;
}
};
int main() {
int length = 12;
int breadth = 8;
// access the static function
Square::find_area(length, breadth);
return 0;
}
// Output: Square: 96
```

2. static Variables

Unlike static functions, a static variable is declared inside the class and defined outside the class. For example,

```
#include <iostream>
using namespace std;
class Company {
public:
// declare a static variable
```

```
static string name;
};
// define the static variable
string Company::name = "Programiz";
int main() {
// access the static variable
cout << "Company name: " << Company::name;
return 0;
}
// Output: Company name: Programiz
```

3. Features of Static and Non-Static Members

- Static class members are associated with the class, so all objects share the same static members.
- Non-static class members are associated with objects, so each object will have its own copy of the non-static member.
- We use the `static` keyword to declare static members.

Protected Access Modifier

Introduction

Previously, we learned about the `public` and `private` access modifiers in the OOP (Basics) chapter. Let's revise what these access modifiers do:

- public members - accessible from outside the class
- private members - not accessible from outside the class

In this lesson, we'll learn about the `protected` access modifier.

C++ Protected Members

Similar to `public` and `private`, we use the `protected` keyword to declare protected class members in C++. For example,

```
class Person {
protected:
int id;
public:
string name;
};
```

Here,

- `id` is protected
- `name` is public

Once we declare a variable/function `protected`, it can be only accessed from that class and its derived classes. If we try to access it from somewhere else, we will get an error. For example,

```
#include <iostream>
using namespace std;
class Person {
protected:
int id;
public:
string name;
};
int main() {
// create an object of Person
Person person;
// access the public variable
person.name = "Jon Snow";
cout << "Name: " << person.name << endl;
// error: protected members cannot be accessed
person.id = 101;
cout << "ID: " << person.id;
return 0;
}
```

When we run this code, we will get an error:

```
error: 'int Person::id' is protected within this context
22 | person.id = 101;
```

This error arises because we are trying to access the protected variable `id` from the `main()` function (outside the `Person` class).

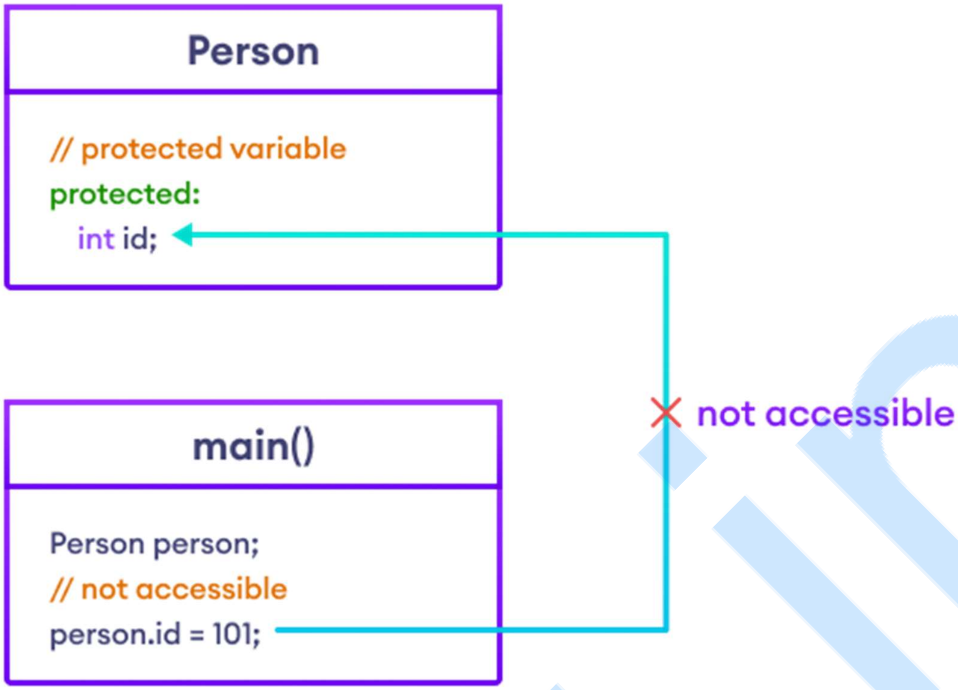

Figure: protected Access Modifier

Access Protected Members

Now let's see how we can access protected class members.

```
#include <iostream>
using namespace std;
class Person {
protected:
int id = 101;
public:
string name;
};
class Student : public Person {
public:
void access_protected() {
// access protected variable
cout << "ID: " << id << endl;
}
};
int main() {
// create an object of Student
Student student;
// access the public variable of the parent class
student.name = "Jon Snow";
cout << "Name: " << student.name << endl;
// call the access_protected() function
student.access_protected();
return 0;
}
```

Output

```
Name: Jon Snow
ID: 101
```

In the above example, we are accessing the protected variable inside the subclass `Student`.

```
void access_protected() {
cout << "ID: " << id;
}
```

This is possible because protected variables can only be accessed by the same class or its subclasses.

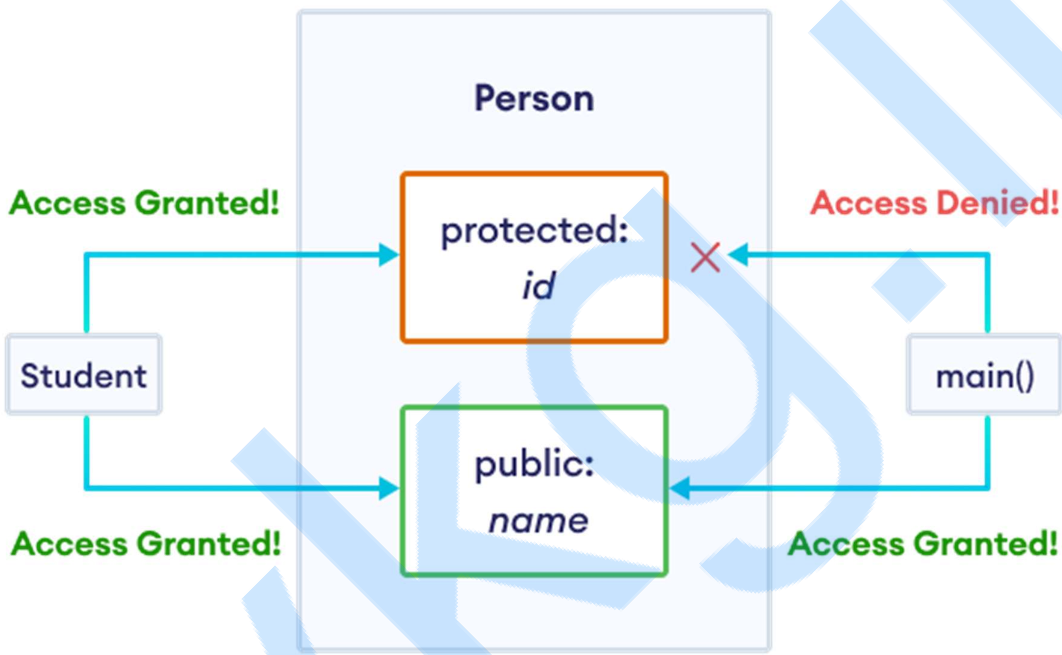

Figure: Protected Access Modifier

In order to access protected members outside the class and its subclasses, we must use public getter and setter functions (either inside the base class or inside the derived class) . Let's see how.

Getter/Setter With protected Modifier

Similar to private variables, we can also access protected members using public getter and setter functions. For example,

```
#include <iostream>
using namespace std;
class Person {
protected:
int id;
public:
string name;
// setter function
```

```
void set_id(int num) {
id = num;
}
// getter function
int get_id() {
return id;
}
};
int main() {
// create an object of Person
Person person;
// access the public variable
person.name = "Jon Snow";
cout << "Name: " << person.name << endl;
// access the protected variable using getter and setter
person.set_id(101);
cout << "ID: " << person.get_id() << endl;
return 0;
}
```

Output

```
Name: Jon Snow
ID: 101
```

In this program, we have created public setter and getter functions in the `Person` class. This allows us to access the protected variable `id` from `main()`.

Inheritance Access Control

Revise Inheritance

We've already learned about inheritance in C++. Here's how we've been implementing inheritance so far,

```
class Person {
    // statements
};

class Student: public Person {
    // statements
};
```

Here, the `Student` class is derived from the `Person` class. As a result,

- `Student` can access all the non-private members of `Person`,
- `Person` cannot access the members of `Student`.

Notice the keyword `public` that is used during inheritance. This indicates that we are performing public inheritance.

Based on the access modifiers (`public`, `private`, and `protected`), we can perform inheritance in 3 different modes:

- Public Inheritance
- Protected Inheritance
- Private Inheritance

Let's start with public inheritance in C++.

Tip: If you are still confused about `public` and `private`, then you ought to revise the OOP (Basics) chapter before proceeding further. And if you're still confused about `protected`, please revisit the previous lesson: protected Access Modifier.

C++ Public Inheritance

In C++ public inheritance,

- public members of the base class are inherited as public in the derived class
- protected members of the base class remain protected in the derived class
- private members of the base class are inaccessible to the derived class

Suppose we have the following classes:

```
class Person {
    private:
        int id;

    protected:
        int marks;

    public:
        string name;
};
class Student: public Person {...};

int main() {...}
```

Here,

- `id` is private, so it cannot be accessed by `Student` and `main()`
- `marks` is protected, so it can be accessed by `Student` but not by `main()`
- `name` is public, so it can be accessed by both `Student` and `main()`

Now let's complete the above example.

Example: C++ Public Inheritance

```
#include <iostream>
using namespace std;
class Person {
private:
int id;
protected:
int marks;
public:
string name;
```

```
};
class Student: public Person {};
int main() {
Student student;
// valid code because name is public
student.name = "Tom Araya";
// error: marks is protected and cannot be accessed
student.marks = 97;
// error: id is private and cannot be accessed
student.id = 101;
return 0;
}
```

In the above example, we have created an object named `student` of the child class. We have then used this object to access the public (`name`), protected (`marks`), and private (`id`) variables of `Person`.

However, when we run this code, we will get an error

```
// error: marks is protected and cannot be accessed
student.marks = 97;
// error: id is private and cannot be accessed
student.id = 101;
```

This is because during public inheritance, the child class inherits variables from the parent class the way they were originally defined. Meaning

- public variable `name` will be inherited as public in `Student`
- protected variable `marks` will be inherited as protected, and
- private variable `id` will be inherited as `private`.

And because of the accessibility (shown in the following table), we get an error.

Accessibility	Private Members	Protected Members	Public Members
Base Class	Yes	Yes	Yes
Derived Class	No	Yes	Yes

We have already discussed how to access the private and protected variables from outside:

- private variables - use public getter and setter functions
- protected variables - access inside the subclass or use public getter and setter functions

We highly recommend you use both of this process and try to solve the issue in the above example.

If you're still not confident about accessing private and protected members, then we suggest you revisit the following before proceeding further:

- Chapter 4: Inheritance
- Lesson: protected Access Modifier

C++ Protected Inheritance

Now that we've learned about public inheritance, it's time to shift our attention to protected inheritance.

In C++ protected inheritance,

- public members of the base class are inherited as protected in the derived class
- protected members of the base class remain protected in the derived class
- private members of the base class are inaccessible to the derived class

Suppose we have the following classes:

```
class Person {
    private:
        int id;

    protected:
        int marks;

    public:
        string name;
};
class Student: protected Person {...};

int main() {...}
```

Here,

- `id` is private, so it cannot be accessed by `Student` and `main()`
- `marks` is inherited as protected, so it can be accessed by `Student` but not by `main()`
- `name` is public in `Person` but inherited by `Student` as protected, so it can be accessed by `Student` but not by `main()`

Let's look at this with an example.

Example: C++ Protected Inheritance

In our Public Inheritance section, we learned that we need to define getter and setter functions in the base class to access its private and protected members.

Now, let's apply the same concept in protected inheritance and see what happens.

```
#include <iostream>
using namespace std;
class Person {
private:
int id;
protected:
int marks;
public:
string name;
// setter function for private variable
void set_id(int num) {
```

```
id = num;
}
// getter function for private variable
int get_id() {
return id;
}
// setter function for protected variable
void set_marks(int num) {
marks = num;
}
// getter function for protected variable
int get_marks() {
return marks;
}
};
class Student : protected Person {};
int main() {
Student student;
// Error: name is inherited as protected
student.name = "Tom Araya";
// Error: set_marks() is inherited as protected
student.set_marks(97);
// Error: set_id() is inherited as protected
student.set_id(101);
// Error: name is inherited as protected
cout << "Name: " << student.name << endl;
// Error: get_id() and get_marks() are inherited as protected
cout << "Id: " << student.get_id() << endl;
cout << "Marks: " << student.get_marks();
return 0;
}
```

In the above example, we have inherited the `Student` class from the `Person` class in protected mode.

In `main()`, we then attempted to access the public members of the `Person` class using an object of the `Student` class.

However, we got multiple errors because the public members of the `Person` class are inherited as protected in the `Student` class.

To get a better idea, here's how all the class members of `Person` are inherited:

- public variable `name` will be inherited as protected in `Student`,
- public functions `set_id()`, `get_id()`, `set_marks()`, and `get_marks()` will be inherited as protected,
- protected variable `marks` will be inherited as protected, and
- private variable `id` will be inherited as `private`.

And because of the accessibility (shown in the following table), we get an error.

Accessibility	Private Members	Protected Members	Public Members
Base Class	Yes	Yes	Yes
Derived Class	No	Yes	Yes (inherited as protected variables)

This means that we cannot access any member of the `Person` class with the way we've written our program.

But there is a solution: we can access `marks` and `name` if we define their getter and setter functions inside the child class `Student` instead of the parent class `Person`.

That way, the public getter and setter functions inside `Student` will remain accessible to the `main()` function, since they are public in `Student`.

On the other hand, the private variable `id` cannot be accessed directly by the `Student` class. There's a roundabout method to do so, but we will not be learning about that right now.

Instead, we'll focus our efforts on accessing only `marks` and `name`. So, let's implement the strategy we just discussed in our next challenge.

C++ Private Inheritance

In C++ private inheritance,

- public members of the base class are inherited as private in the derived class
- protected members of the base class are inherited as private in the derived class
- private members of the base class are inaccessible to the derived class.

Suppose we have the following classes:

```
class Person {
   private:
      int id;

   protected:
      int marks;

   public:
      string name;
};
class Student: private Person {...};

int main() {...}
```

Here,

- `id` is private in `Person` itself, so it cannot be accessed by `Student` and `main()`
- `marks` is inherited as private, so it can be accessed by `Student` but not by `main()`
- `name` is inherited as private, so it can be accessed by `Student` but not by `main()`

Let's look at this with an example.

Example: C++ Private Inheritance

```
#include <iostream>
using namespace std;
class Person {
private:
int id;
protected:
int marks;
public:
string name;
// setter function for private variable
void set_id(int num) {
id = num;
}
// getter function for private variable
int get_id() {
return id;
}
// setter function for protected variable
void set_marks(int num) {
marks = num;
}
// getter function for protected variable
int get_marks() {
return marks;
}
};
class Student: private Person {};
int main() {
Student student;
// Error: name is inherited as private
student.name = "Tom Araya";
// Error: set_marks() is inherited as private
student.set_marks(97);
// Error: set_id() is inherited as private
student.set_id(101);
// print Student information
cout << "Name: " << student.name << endl;
// Error: get_id() and get_marks() are inherited as private
cout << "Id: " << student.get_id() << endl;
cout << "Marks: " << student.get_marks();
return 0;
}
```

Just like in protected inheritance, we are unable to access the `name` and `marks` variables and the getter and setter functions using the `student` object, even though they have been declared as public or protected in the parent class.

This is because they are inherited as private by the `Student` class. So, we can't use a `Student` object to access them in `main()`.

This is shown by the accessibility table below:

Accessibility	Private members	Protected members	Public members
Base Class	Yes	Yes	Yes
Derived Class	No	Yes (inherited as private variables)	Yes (inherited as private variables)

Once again, we need to create public getter and setter functions inside the child class `Student` to access the `name` and `marks` variables of the parent class `Person`.

And like in Protected Inheritance, we won't be accessing the private variable `id`.

We will, however, tell you the steps necessary to access it towards the end of this lesson. But you'll have to write the code yourself :)

Revision

Before we dive into the next lesson, let's first revise the key concepts we learned in this lesson.

1. Inheritance Access Control in C++

In C++, we can derive classes in 3 modes:

- Public Inheritance
- Protected Inheritance
- Private Inheritance

2. Properties of the Different Inheritance Modes

The following code specifies how members of the base class are inherited in the derived classes:

```
class Parent {
    public:
        int x;
    protected:
        int y;
    private:
        int z;
};

// public inheritance
class Public_Child: public Parent {
    // x is public
    // y is protected
    // z is not accessible from Public_Child
};

// protected inheritance
class Protected_Child: protected Parent {
    // x is protected
    // y is protected
    // z is not accessible from Protected_Child
};

// private inheritance
class Private_Child: private Base {
    // x is private
    // y is private
```

```
    // z is not accessible from Private_Child
};
```

3. Access Members of the Base Class (Public Inheritance)

- private members - create public getter and setter functions in the base class to access
- protected members - create public getter and setter functions in either the base class or the derived class
- public members - can be accessed from outside the class

4. Access Members of the Base Class (Protected and Private Inheritance)

- protected and public members - create public getter and setter functions in the derived class
- private members - can't be accessed directly from the derived class

Self-Study: Access Private Members (Optional)

If you have a parent class `Person` with a private variable `id`, how can you access the variable using an object of the child class `Student` using private or protected inheritance?

One answer lies in the procedure outlined below:

1. Inside the Person Class

- Create getter and setter functions `get_id()` and `set_id()`.

2. Inside the Student Class

1. Create an object of the `Person` class named `person`.
2. Create a function `set_student_id()` that accepts an integer parameter `num`.
3. Inside this function, call `set_id()` using the `person` object and pass `num` as an argument to it.
4. Create another function `get_student_id()`.
5. Inside this function, use the code `return person.get_id();` to return the required value.

3. Inside the main() Function

- Use `set_student_id()` and `get_student_id()` to access the private variable using a `Student` object.

This concludes our lesson on Inheritance Access Control. Next, we will look at some additional topics in OOP and Pointers.

Additional Topics

Functions and Objects

Introduction

Let's first revise the working of functions and objects.

1. Revise Function

```
#include <iostream>
using namespace std;
// function that adds two integers and returns the result
int add_numbers(int number1, int number2) {
int sum = number1 + number2;
return sum;
}
int main() {
// call the function
int sum = add_numbers(42, 18);
cout << "Sum = " << sum;
return 0;
}
// Output: Sum = 60
```

2. Revise Objects

```
#include <iostream>
using namespace std;
// create a class
class Addition {
public:
int add_numbers(int number1, int number2) {
int sum = number1 + number2;
return sum;
}
```

```
};
int main() {
// create an object of Addition
Addition addition;
// access the function using an object
int sum = addition.add_numbers(42, 18);
cout << "Sum = " << sum;
return 0;
}
// Output: Sum = 60
```

Functions and Objects

In C++, we can also use functions and objects together i.e. we can

- pass objects as arguments to a function
- return an object from the function.

Pass Objects to Function

Suppose we have a class named `Student`.

```
class Student {
    ...
};
```

Now, we can pass an object of this class to a function.

```
// function that takes a Student object as argument
void display(Student obj_arg) {
    ...
}
```

Here, you can see we have included the `Student` object `obj_arg` as the parameter to the `display()` function.

Now, to call this function, we can create an object of `Student` and pass it during the function call.

```
// create an object of Student
Student obj;
```

```
// call the function by passing the object as argument
display(obj);
```

Here, we have passed the `obj` object of the `Student` class to the `display()` function.

Next, let's look at a working example of this.

Example: Pass Object to Function

```
#include <iostream>
using namespace std;
class Student {
public:
double marks;
// constructor to initialize marks
Student(double m) : marks(m) {}
};
// function that accepts an object as argument
void display_marks(Student obj) {
cout << "Marks = " << obj.marks;
}
int main() {
// create Student object
Student student(88.5);
// call the function
display_marks(student);
return 0;
}
// Output: Marks = 88.5
```

In the above example, we have created a function named `display_marks()`.

```
void display_marks(Student obj) {
cout << "Marks = " obj.marks;
}
```

Here, the function takes an object of the `Student` class as a parameter and prints the `marks` using the object.

Even though the function is declared outside of the `Student` class, we are able to access the `marks` variable from this function.

We can do this because we have passed the object of the `Student` class while calling the function.

```
// create Student object
Student student(88.5);
// call function
display_marks(student);
```

Return Object From a Function

Similarly, we can also return an object from a function. All we have to do is use the class name as the return type. For example,

```
Student add_numbers () {
// function body
return obj;
}
```

Here, the `add_numbers()` function will return an object of the `Student` class. Let's see a complete example of this.

```
#include <iostream>
using namespace std;
class Student {
public:
double marks1, marks2;
};
// function that returns object of Student
Student initialize_object() {
// create Student object
Student student;
// initialize marks1 and marks2
student.marks1 = 96.5;
student.marks2 = 75.0;
// return the object
return student;
}
int main() {
Student obj;
// call function and assign
// the return value to obj
obj = initialize_object();
// print member variables of obj
cout << "Marks 1 = " << obj.marks1 << endl;
cout << "Marks 2 = " << obj.marks2;
return 0;
}
```

Output

```
Marks 1 = 96.5
Marks 2 = 75
```

In this program, we have created a function `initialize_object()` that returns an object of `Student` class.

```
Student initialize_object() {
...
}
```

Inside the function, we have

- created the `Student` object named `student`
- initialized `marks1` and `marks2` variables using the object
- returned the object

We have then called `initialize_object()` from the `main()` function.

```
// call function
obj = initialize_object();
cout << "Marks 1 = " << obj.marks1 << endl;
cout << "Marks 2 = " << obj.marks2;
```

Here, we have stored the returned object in the `obj` object and accessed the `marks1` and `marks2` values associated with the object.

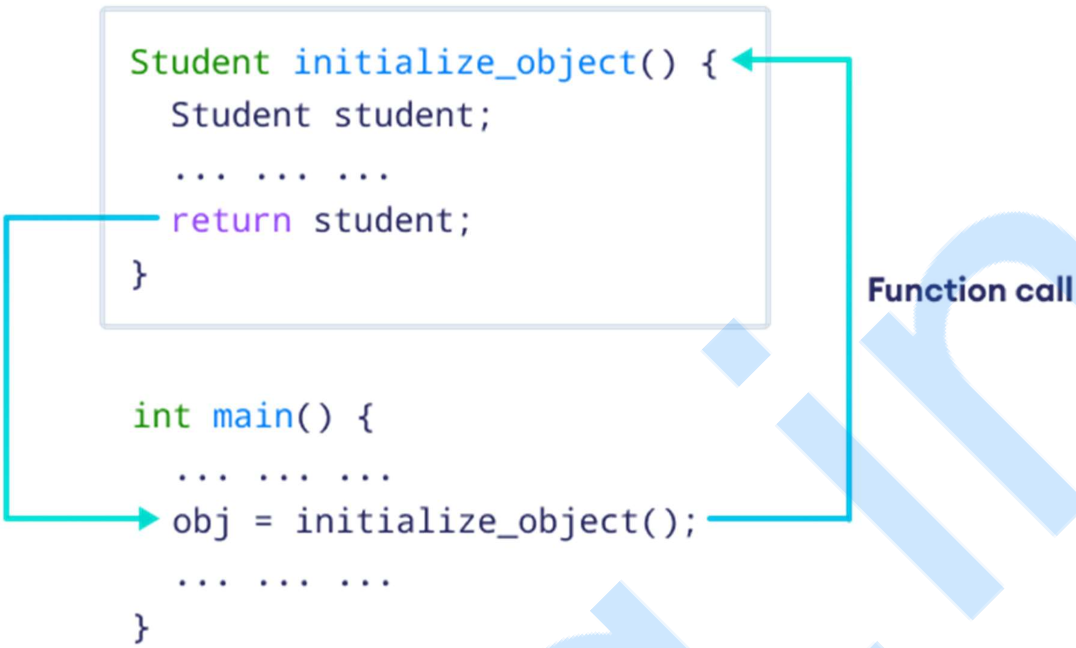

Figure: Return Object From Function

Example: Add Complex Numbers

In this example, we will add two complex numbers using functions and objects. A complex number has the format

```
// format of complex numbers

8 + 2.4i

6 + 4.2i
```

Here, 8 and 6 are real parts and 2.4 and 4.2 are imaginary parts.

While performing addition of two complex numbers, we add real and imaginary parts separately. Hence, the sum of above mentioned numbers will be `14 + 6.6i`.

Source Code

```
#include <iostream>

using namespace std;

class Complex {

public:

// variables to store real and imaginary part

double real, imag;
```

```cpp
// constructor to initialize real and imag
Complex(double r, double i) : real(r), imag(i) {}
};
// function to add complex numbers
// takes two Complex objects as arguments
// returns a Complex object that contains the sum
Complex add_complex(Complex c1, Complex c2) {
// create a new object of Complex
// initial real and imag values are set to 0
Complex result(0, 0);
// add real parts of complex numbers
result.real = c1.real + c2.real;
// add imaginary parts of complex numbers
result.imag = c1.imag + c2.imag;
// return the result
return result;
}
// function to print complex numbers
// takes an object of Complex as the parameter
void print_complex(Complex c1) {
cout << c1.real << " + " << c1.imag << "i" << endl;
}
int main() {
// create objects for two complex numbers
// with real and imaginary values
Complex c1(8, 2.4);
Complex c2(6, 4.2);
// object to store the addition
// initial real and imag values are set to 0
Complex sum(0, 0);
// print complex numbers
cout << "First Complex Number: ";
print_complex(c1);
cout << "Second Complex Number: ";
```

```
print_complex(c2);

// call the add_complex() function

sum = add_complex(c1, c2);

// print the resulting complex

cout << "Resulting Complex Number: ";

print_complex(sum);

return 0;

}
```

Output

```
First Complex Number: 8 + 2.4i
Second Complex Number: 6 + 4.2i
Resulting Complex Number: 14 + 6.6i
```

In the above example, we have created a class named `Complex` with two `double` variables: `real` and `imag`. We have also created a constructor to initialize these variables.

Notice the function,

```
Complex add_complex(Complex c1, Complex c2) {

// create a new object of Complex

// initial real and imag values are set to 0

Complex result(0, 0);

// add real parts of complex numbers

result.real = c1.real + c2.real;

// add imaginary parts of complex numbers

result.imag = c1.imag + c2.imag;

// return the result

return result;

}
```

Here, the function takes two objects (`c1` and `c2`) as parameters and returns an object of the `Complex` class. Inside the function, we have

- created an object `result` with initial values for `real` and `imag` as 0
- performed addition between the `real` parts of `c1` and `c2` objects and assigned the sum to `real` of `result`
- performed addition between the `imag` part of `c1` and `c2` objects and assigned the sum to `imag` of `result`

Important! We highly recommend you try this code again and again. If you can solve this problem without referring to the program we've written above, you will have a better understanding of how to use functions and objects together.

Friend Function and Friend Class

Introduction

In previous lessons, we have learned that `private` and `protected` class members declared cannot be accessed from outside of the class.

```
#include <iostream>
using namespace std;
class Rectangle {
private:
int length, breadth;
public:
// constructor to initialize variables
Rectangle() : length(8), breadth(6) {}
};
int find_area(Rectangle obj) {
// error
int area = obj.length * obj.breadth;
return area;
}
int main() {
Rectangle obj;
// call find_area() by
// passing the object of Rectangle
cout << "Area = " << find_area(obj);
return 0;
}
```

When we run this code, we will get an error because we are trying to access the private variables `length` and `breadth` of the `Rectangle` class from the `find_area()` function.

```
error: 'int Rectangle::length' is private within this context
   15 |      int area = obj.length * obj.breadth;
```

Now, the only way we have accessed private members so far is through getter and setter functions (and sometimes with constructors).

However, there is another way to access private members, known as friend functions and friend classes.

Friend functions and classes are exceptional cases using which we can access all class members from outside of the class, including private and protected members.

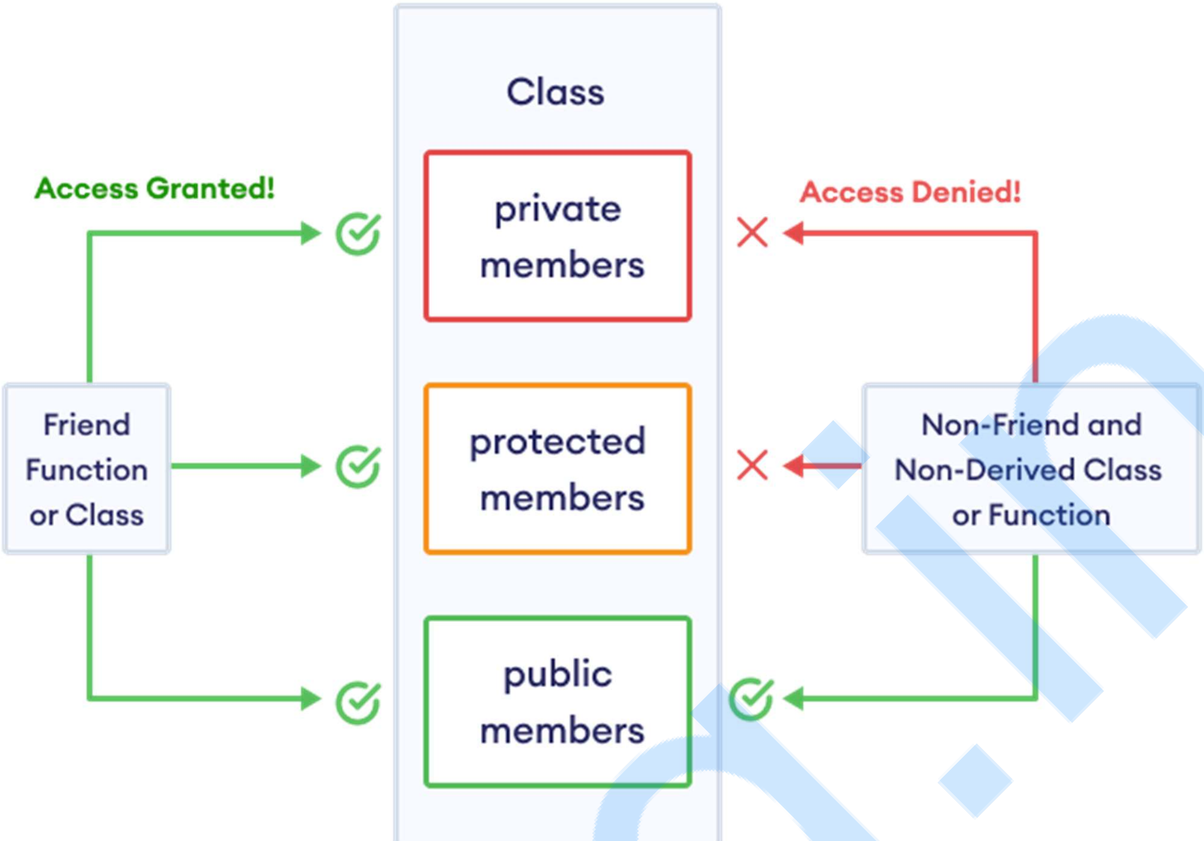

Figure: Friend Function and Classes

Let's start with the friend function first.

C++ Friend Function

As mentioned before, a friend function can access the private and protected members of a class. We use the `friend` keyword to declare a friend function. For example,

```
class Rectangle {
...
// friend function declaration
friend int find_area(Rectangle);
...
};
```

In the above code, we have declared a friend function `find_area()` inside the `Rectangle` class so that it can access all of the class members.

Let's explore further with an example.

```
#include <iostream>
using namespace std;
class Rectangle {
private:
int length, breadth;
```

```
public:
// constructor to initialize variables
Rectangle() : length(8), breadth(6) {}
// friend function declaration
friend int find_area(Rectangle);
};
// friend function definition
int find_area(Rectangle obj) {
// access private members
// from the friend function
int area = obj.length * obj.breadth;
return area;
}
int main() {
Rectangle obj;
// call find_area() by
// passing the object of Rectangle
cout << "Area = " << find_area(obj);
return 0;
}
// Output: Area = 48
```

In the above example, we have created the `Rectangle` class. It consists of two private members: `length` and `breadth`.

Notice that we have declared a friend function inside the `Rectangle` class and its definition is outside the class.

```
class Rectangle {
...
// friend function declaration
friend int find_area(Rectangle);
};
// friend function definition
int find_area(Rectangle obj) {
...
}
```

The function accepts an object of the `Rectangle` class as its parameter.

As you can see, we are able to access the private variables: `length` and `breadth` from the outer function (`find_area()`). It's possible because the outer function `find_area()` is declared as a friend function.

C++ Friend Class

Similar to friend functions, we can also create friend classes. A friend class can access the member variables and member functions of the class it is declared in. For example,

```
#include <iostream>
using namespace std;
class Animal {
private:
```

```
int legs_count;
public:
// constructor to initialize variable
Animal() : legs_count(4) {}
// declare friend class
friend class Dog;
};
// define friend class
class Dog {
public:
void count_legs() {
// create Animal object
Animal animal;
// access private variable of Animal class
cout << "Legs = " << animal.legs_count;
}
};
int main() {
// create object of friend class
Dog dog;
dog.count_legs();
return 0;
}
// Output: Legs = 4
```

Here, the class `Dog` is a friend class of class `Animal`.

```
// inside Animal class
// declare friend class
friend class Dog;
```

That's why we are able to access the private variable `legs_count` from the `Dog` class.

```
// inside Dog class
void leg_count() {
Animal animal;
cout << "Legs = " << animal.legs_count;
}
```

Dynamic Memory Allocation

Why Dynamic Memory Allocation?

Before we learn about dynamic memory allocation, let's first look at some limitations of an array.

```
int marks[10];
```

Here, the size of the array is 10, which is fixed and cannot be changed.

Creating an array of fixed size can lead to two issues:

- If we only need to store the marks of, say, 3 students, then we have wasted memory.
- If we need to store the marks of more than 10 students, we cannot do that.

To solve this issue, the concept of dynamic memory allocation was introduced in C++ programming.

Dynamic memory allocation allows us to allocate memory after we run our program (during run-time).

Now that we know why we need dynamic memory allocation, let's learn about it in further detail.

Dynamic Memory Allocation

Revision: Pointer

Let's first revise the concept of pointers with the help of this example.

```
#include <iostream>
using namespace std;
int main() {
// create a variable
int number = 36;
// create a pointer variable and
// assign the address of number to it
int* ptr = &number;
// print value of number using ptr
cout << *ptr; // 36
return 0;
}
```

Here,

- `&number` - memory address of `number`
- `int* ptr` - pointer variable
- `*ptr` - gives the value pointed by the `ptr` pointer

Now, let's get back to dynamic memory allocation.

Dynamic Memory Allocation

In C++ dynamic memory allocation, we use the following operators alongside a pointer:

- `new` - dynamically allocates memory during run-time
- `delete` - clears the dynamically allocated memory after it is of no use to us

Now, let's learn about these operators in greater detail.

C++ new and delete

We use the `new` operator to dynamically allocate memory in C++. Let's start by dynamically allocate memory to an integer variable:

```
int* ptr = new int;
```

Here, we have dynamically allocated memory to an `int` variable.

To dynamically allocate memory for a `double` variable, we use the following code:

```
double* ptr = new double;
```

Now, if we print the pointer `ptr`, we will get the memory address as output.

```
cout << ptr;    // 0x56425d6e7eb0
```

This is because the `new` operator returns the address of the newly allocated memory location.

Assign Value to Dynamically Allocated Memory

Since we use pointers for dynamic memory allocation, we use the dereference operator `*` to assign value to the allocated memory:

```
int* ptr = new int;
*ptr = 5;
```

Here, `*ptr = 5;` assigns the integer value 5 to the dynamically allocated memory.

Next, we will learn to deallocate the newly allocated memory.

C++ delete Operator

We use the `delete` operator to dynamically deallocate a memory. For example,

```
// allocate memory
int* ptr = new int;
```

```
// deallocate memory
delete ptr;
```

This is important because we need to free the dynamically allocated memory once it is used.

Example: Dynamic Memory Allocation

Let's look at a simple program that dynamically allocates (and then deallocates) memory to a single integer variable.

```
#include <iostream>
using namespace std;
int main() {
// dynamically allocate memory
int* number = new int;
// assign value to the memory
*number = 256;
cout << *number;
// deallocate the memory
delete number;
return 0;
}
// Output: 256
```

In this program, we have used the `new` keyword to dynamically allocate memory to an `int` variable.

```
int* number = new int;
```

After assigning a value to the variable and printing it on the screen, we finally deallocated the memory using `delete`.

```
delete number;
```

Dynamic Memory Allocation: Array

Before we implement dynamic memory allocation for arrays, let's revise the concept of pointers and arrays with the help of this example.

```
#include <iostream>
using namespace std;
int main() {
// integer array
int numbers[] = {1, 2, 3};
// print array elements
for (int i = 0; i < 3; ++i) {
cout << *(numbers + i) << " ";
}
return 0;
}
// Output: 1 2 3
```

Here, `*(numbers + i)` returns the ith element of the array.

Now, let's create a program that stores the marks of `n` students where the value of `n` will be provided dynamically during run time.

```
#include <iostream>
using namespace std;
int main() {
int n;
cout << "Enter the number of students: ";
cin >> n;
// create pointer variable and dynamically allocate
// n number of memory locations to it
int* marks = new int[n];
cout << "Enter marks:";
for (int i = 0; i < n; ++i) {
// store value at the allocated memory using marks pointer
cin >> *(marks + i);
}
cout << "Marks: ";
for (int i = 0; i < n; ++i) {
cout << *(marks + i) << endl;
}
// free the allocated memory
delete[] marks;
return 0;
}
```

Output

```
Enter the number of students: 3
Enter marks:25
68
72
Marks: 25
68
72
```

Notice how we have allocated and deallocated the memories for the array:

```
// allocate memory to array of size n
int* marks = new int[n];
// deallocate the memory in the array
delete[] marks;
```

Remember, when we allocate memory for the array, we need to specify the array size at the end using the `[]` symbol.

Similarly, we need to use `[]` with the `delete` operator to deallocate the array memory.

Reallocate Memory

Sometimes, the dynamically allocated memory is insufficient or more than required.

Unfortunately, C++ does not provide any standard function to reallocate memory. Instead, we can tackle this problem by following the given steps:

1. First, allocate new memory using a different pointer.
2. Then, copy the contents of the original dynamic array to the new array.
3. Deallocate the memory assigned to the original array.
4. Finally, use the new dynamic array after that point.

Let us look at this with an example.

Example: Reallocate Memory

Let's suppose we dynamically created an array of size 4. And we have to append one more element to the array. Let's see an example of how we can do that.

```
#include <iostream>
using namespace std;
int main() {
// dynamically create an array of size 4
int* array1 = new int[4];
cout << "Enter Array Elements: ";
// get input value for array1
for (int i = 0; i < 4; ++i) {
cin >> *(array1 + i);
}
cout << "Array Elements: ";
```

```
// print array elements
for (int i = 0; i < 4; ++i) {
cout << *(array1 + i) << ", ";
}
// create new pointer of size 5
int* array2 = new int[5];
// copy array1 to array2
for (int i = 0; i < 4; ++i) {
// assign ith element of array1
// to ith element of array2
*(array2 + i) = *(array1 + i);
}
// deallocate array1
delete[] array1;
// add 20 as the last element of array2
array2[4] = 20;
// print all elements of array2
cout << "\nNew Array Elements: ";
for (int i = 0; i < 5; ++i) {
cout << *(array2 + i) << ", ";
}
// deallocate array2
delete[] array2;
return 0;
}
```

Output

```
Enter Array Elements: 16
17
18
19
Array Elements: 16, 17, 18, 19,
New Array Elements: 16, 17, 18, 19, 20,
```

Here's how this program works:

1. Create initial array and get user input

First, we have dynamically created the `array1` array and took user inputs for it.

```
int* array1 = new int[4];
for (int i = 0; i < 4; ++i) {
cin >> *(array1 + i);
}
```

Figure: Elements of the Initial Dynamic Array

2. Create replacement array and copy original elements to it

Then, we created `array2` and copied the elements of `array1` to it.

```
int* array2 = new int[5];
for (int i = 0; i < 4; ++i) {
*(array2 + i) = *(array1 + i);
}
```

Figure: Copying Elements of One Array to Another

3. Deallocate the original array and add 20 to the new array

After that, we deallocated the memory allocated to `array1` and added the integer 20 as the 5th element of the array.

```
delete[] array1;
array2[4] = 20;
```

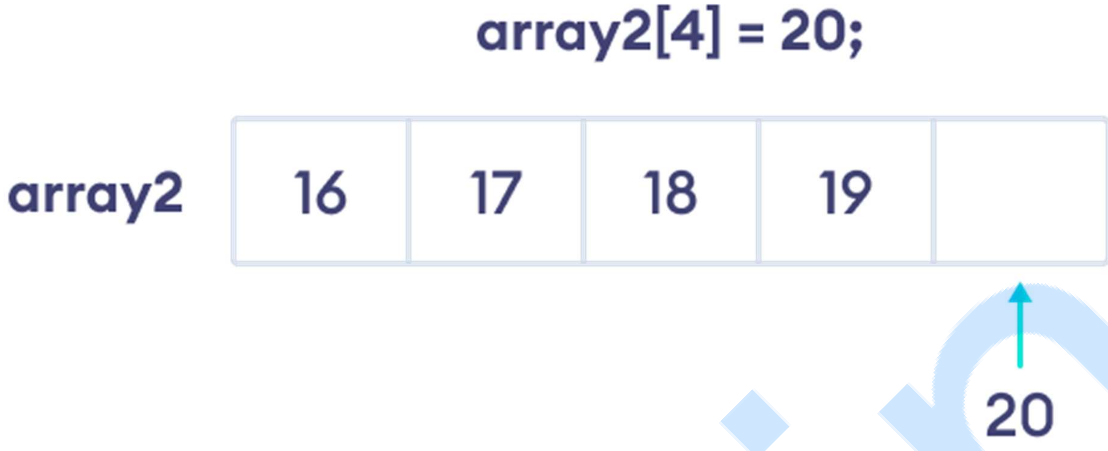

Figure: Adding Integer 20 to Index 4 of array2

4. Print the new array and deallocate the memory

Finally, we printed the new array and deallocated the memory using `delete[]`.

Dynamic Object Creation

We can also use the `new` keyword to dynamically create an object in C++. For example,

```
#include <iostream>
using namespace std;
class Student {
public:
string name;
};
int main() {
// dynamically create Student object
Student* ptr = new Student();
// initialize and print class variable
ptr->name = "Peter Parker";
cout << "Name = " << ptr->name;
// ptr memory is released
delete ptr;
return 0;
}
// Output: Name = Peter Parker
```

In the above example, we have used the `new` keyword to create an object of the `Student` class.

```
// dynamically declare Student object
Student* ptr = new Student();
```

Here, `ptr` is a pointer variable of `Student` type.

We have then used the pointer to initialize and print the class variable.

```
// initialize and print class variable
ptr->name = "Peter Parker";
cout << "Name = " << ptr->name;
```

Remember that the arrow operator `->` is used to access class members using pointers.

Finally, we use the `delete ptr;` code to dynamically remove the object from the computer memory.

Example: C++ Dynamic Object Creation

In this example, we will dynamically create an object to calculate the area and circumference of a circle.

```
#include <iostream>
using namespace std;
class Circle {
private:
double radius;
double pi = 3.14;
public:
// constructor to initialize radius
Circle(double rad) : radius(rad) {}
// function to calculate area of the circle
double calculate_area() {
return pi * radius * radius;
}
// function to calculate circumference of the circle
double calculate_circumference() {
return 2 * pi * radius;
}
};
int main() {
// get user input for radius
double radius;
cout << "Enter radius of circle: ";
cin >> radius;
// dynamically create Circle object
// pass radius as argument to constructor
Circle* circle = new Circle(radius);
// get the area of circle
double area = circle->calculate_area();
// get the circumference of circle
double circumference = circle->calculate_circumference();
// print area and circumference
cout << "Circle Area: " << area << endl;
cout << "Circle Circumference: " << circumference;
// free the computer memory
delete circle;
return 0;
}
```

Output

```
Enter radius of circle: 2.5
Circle Area: 19.625
Circle Circumference: 15.7
```

Notice the following code in the program above.

```
// dynamically create Circle object
// pass radius as argument to constructor
Circle* circle = new Circle(radius);
```

Here, we have passed the variable `radius` as argument to the constructor of the object.

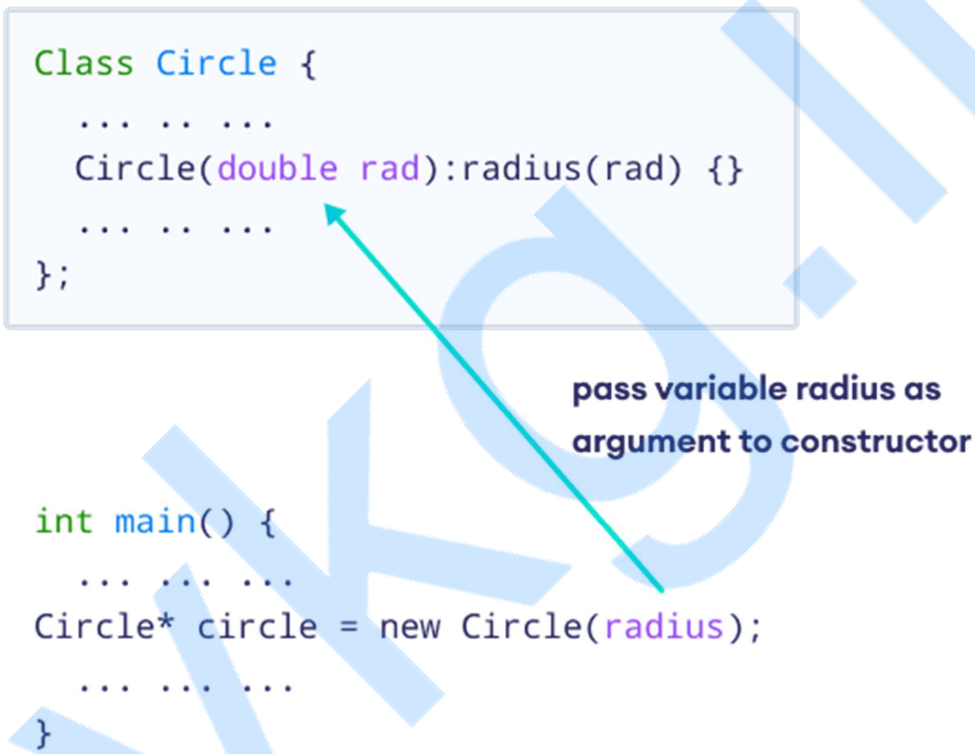

Figure: Pass argument to constructor when creating an object dynamically

Next, we will pass literals as argument to the constructor.

Pass Literal as an Argument

```
#include <iostream>
using namespace std;
class Circle {
private:
double radius;
double pi = 3.14;
public:
// constructor to initialize radius
Circle(double rad): radius(rad) {}
```

```
// function to calculate area of the circle
double calculate_area() {
return pi * radius * radius;
}
// function to calculate circumference of the circle
double calculate_circumference() {
return 2 * pi * radius;
}
};
int main() {
// dynamically create Circle object
// pass 3.6 as argument to constructor
Circle* circle = new Circle(3.6);
// print radius, area and circumference
cout << "Circle Radius: 3.6" << endl;
cout << "Circle Area: " << circle->calculate_area() << endl;
cout << "Circle Circumference: " << circle->calculate_circumference();
// free the computer memory
delete circle;
return 0;
}
```

Output

```
Circle Radius: 3.6
Circle Area: 40.6944
Circle Circumference: 22.608
```

In the above program, notice the following code.

```
Circle* circle = new Circle(3.6);
```

Here, we have passed the literal 3.6 as an argument to the constructor of the object.

With this, we've completed the section on dynamic memory allocation. Let's now learn about some important types of pointers in C++.

But first, do complete the final challenge of this lesson :)

Pointer Types

Introduction

So far, we have learned about the many ways pointers can be used in C++. In this section, we will look at some special types of pointers. They are

- `this` pointer
- void pointers
- dangling pointers

Let's start with 'this' pointer.

Introduction to 'this' Pointer

In C++, we use the `this` keyword to refer to the current object. Let's see what that means.

```
#include <iostream>
using namespace std;
// define the Student class
class Student {
public:
// public string variable to hold the student's name
string name;
// function that displays the student's name
void display_name() {
cout << "Student's name using this: " << this->name << endl;
}
};
int main() {
// create a Student object and set the name variable
Student student;
student.name = "John Doe";
// call the display_name() function
student.display_name();
// print the student's name
cout << "Student's name using object: " << student.name;
return 0;
}
```

Output

```
Student's name using this: John Doe
Student's name using object: John Doe
```

In the above example, you can see both `student.name` and `this->name` give the same result, `John Doe`.

Basically, what happens here is when we call the `display_name()` function using the `student` object, `this` will refer to the current object, which is `student`.

```
void display_name() {
cout << "Student's name using this: " << this->name << endl;
}
```

Hence, we get the output `John Doe` (value of `name` for `student`).

Similarly, if we call the function with another object (let's say `student2`), `this->name` will print the value of `name` for `student2`. For example,

```
#include <iostream>
using namespace std;
// define the Student class
class Student {
public:
// public string variable to hold the student's name
string name;
```

```
// function that displays the student's name
void display_name() {
cout << "Student's name using this: " << this->name << endl;
}
};
int main() {
// create a Student object and set the name variable
Student student;
student.name = "John Doe";
// call the display_name() function
student.display_name();
// create a Student object and set the name variable
Student student2;
student2.name = "Lily Doe";
// call the display_name() function
student2.display_name();
return 0;
}
```

Output

```
Student's name using this: John Doe
Student's name using this: Lily Doe
```

Here, for the function call

- `student.display_name()` - `this` refers to the `student` object
- `student2.display_name()` - `this` refers to the `student2` object

C++ this in Constructor

In C++, we often use the `this` keyword to initialize member variables inside a constructor. For example,

```
#include <iostream>
using namespace std;
// define the Student class
class Student {
public:
// public string variable to hold the student's name
string name;
// public int variable to hold the student's score
int score;
// constructor that initializes the student's name and score
Student(string name, int score) {
this->name = name;
this->score = score;
}
};
int main() {
// create two Student objects and set their name and score variables
Student student1("John Doe", 80);
Student student2("Jane Doe", 90);
// print the student1 names and scores
```

```
cout << "First student: " << endl;
cout << "Name: " << student1.name << endl;
cout << "Score: " << student1.score << endl << endl;
// print the student1 names and scores
cout << "Second student: " << endl;
cout << "Name: " << student2.name << endl;
cout << "Score: " << student2.score << endl;
return 0;
}
```

Output

```
First student:
Name: John Doe
Score: 80

Second student:
Name: Jane Doe
Score: 90
```

In the above example, we have used a constructor to assign the values of variables `name` and `score`.

```
Student(string name, int score) {
this->name = name;
this->score = score;
}
```

Notice that the constructor parameters have the same name as the member variables.

Hence, we have used the `this` pointer to point to the member variables, while the variables without the `this` pointer refer to the constructor parameters.

Since we know that `this` refers to the current object, here's what happens when creating the objects:

Student student1("John Doe", 80);

- `this` will refer to `student1`
- arguments: `John Doe` and 80 will be assigned to `student1.name` and `student1.score`

Student student2("Jane Doe", 90);

- `this` refers to `student2`
- arguments: `Jane Doe` and 90 will be assigned to `student2.name` and `student2.score`

More on 'this' Pointer

In the last example, we created a constructor with the same parameter names as the member variables. Then we referred to the member variables using the `this` pointer.

We could have also used different variables in the parameter name. In this section, we will learn why using the `this` pointer in the constructor is important.

```
class Rectangle {
    public:

        // member variables
        double length;
        double breadth;

        // constructor to initialize variables
        Rectangle(double len, double brth) {
            length = len;
            breadth = brth;
        }
};
```

Here, `len` and `brth` are the parameters of the `Rectangle()` constructor, and they are used to initialize the `length` and `breadth` member variables, respectively.

The variable names `len` and `brth` are not informative. Remember, variable names in any programming language should be as clear and informative as possible.

So, it is preferable to name our constructor parameters as `length` and `breadth`. However, this will create a lot of confusion. For example,

```
// error code
Rectangle(double length, double breadth) {
    length = length;
    breadth = breadth;
}
```

As you can see from the code above, both the member variables and the constructor parameters share the same variable names.

Obviously, this creates a lot of confusion, i.e., we can't tell which is the member variable and which is the constructor parameter.

The C++ compiler will also suffer from the same confusion. So when we run the code, it will not initialize the member variables. Instead, we get unexpected output.

We can solve this problem by using the `this` pointer.

```
// use this pointer inside constructor
Rectangle(double length, double breadth) {
    this->length = length;
    this->breadth = breadth;
}
```

Here,

- `this->length` and `this->breadth` indicate the member variables
- `length` and `breadth` are the constructor parameters

Example: C++ this Pointer

Let us look at the following example to make the concept of `this` pointer clear.

```
#include <iostream>
using namespace std;
class Rectangle {
private:
// member variables
double length;
double breadth;
public:
// constructor to initialize variables
Rectangle(double length, double breadth) {
// this->length and this->breadth are member variables
this->length = length;
this->breadth = breadth;
}
// function to calculate the area of the rectangle
double calculate_area() {
return this->length * this->breadth;
}
};
int main() {
// create Rectangle objects
Rectangle rectangle1(25.5, 16.8);
Rectangle rectangle2(12.0, 8.0);
// call the calculate_area() function of rectangle1
double area1 = rectangle1.calculate_area();
cout << "Rectangle 1 Area = " << area1 << endl;
// call the calculate_area() function of rectangle2
double area2 = rectangle2.calculate_area();
cout << "Rectangle 2 Area = " << area2;
return 0;
}
```

Output

```
Rectangle 1 Area = 428.4
Rectangle 2 Area = 96
```

In this program, we have used the `this` pointer inside the constructor to initialize the member variables, since the constructor parameters and the member variables share the same names.

```
Rectangle(double length, double breadth) {
this->length = length;
this->breadth = breadth;
}
```

We have also used the `this` pointer inside the `calculate_area()` function.

```
double calculate_area() {
return this->length * this->breadth;
}
```

However, it's not necessary to use the `this` pointer inside this function because there are no function parameters or function variables with conflicting names. But there is no harm in using `this` either.

Common Mistakes With this Pointer (I)

1. Not Using this Pointer When it is Required

We've already stated that the C++ compiler will get confused if the function/constructor parameters have the same names as the member variables.

As a result, we get unexpected output. Let's see this with an example.

```
#include <iostream>
using namespace std;
class Person {
public:
string name;
// invalid code
Person(string name) {
name = name;
}
};
int main() {
Person person("M. Bison");
// print name of person
cout << "Hello, " << person.name;
return 0;
}
// Output: Hello,
```

In the above program, our expected output is `Hello, M. Bison`. But we only get `Hello,` instead. This is because our constructor failed to initialize the `name` variable.

2. Using Dot Operator Instead of Arrow Operator

In C++, the `this` keyword is used with the arrow operator `->`, not the dot operator (`.`). It is only in languages like Java that `this` is used with the dot operator. For example,

```
#include <iostream>
using namespace std;
class Person {
public:
string name;
// error: use -> instead of .
Person(string name) {
this.name = name;
}
};
int main() {
Person person("M. Bison");
cout << "Hello, " << person.name;
return 0;
}
```

```
// Output:
// error: request for member 'name' in '(Person*)this', which is of pointer type 'Person*' (maybe you meant to use '->' ?)
```

To access class members,

- we use the dot operator `.` with objects
- the arrow operator `->` with pointers

Since `this` is a pointer, we must use the `->` operator.

Common Mistakes With this Pointer (II)

3. Using this Pointer in Constructor Initializer Lists

Using `this` to access member variables in constructor initializer lists will cause an error.

This is because the initialization list already makes the member variables and the constructor parameters unambiguous.

So, there is no need to remove the ambiguity through the use of `this` pointer.

```
#include <iostream>
using namespace std;
class Person {
public:
string name;
// error: use of this pointer in constructor initializer list
Person(string name): this->name(name) {}
};
int main() {
Person person("Terry Bogard");
// print name of person
cout << "Hello, " << person.name;
return 0;
}
// Output:
// error: expected identifier before 'this'
```

We can fix this error by removing the `this` keyword from the constructor.

```
Person(string name): name(name) {}
```

Now that we've covered the basics of `this` pointer, it's time to shift our attention to void pointers.

But before that, let's solve a coding challenge to test what you've just learned in this section!

C++ Void Pointers

If we don't know the data type of a variable that the pointer points to, it is known as a void pointer. It is also known as pointer to void.

It is a generic pointer that is declared using the `void` keyword. For example,

```
void *ptr;
```

Here, `ptr` is a void pointer. Let us see how we can use this type of pointer:

```
int *ptr;
double number = 9.0;

ptr = &number;   // Error
```

Here, `ptr` is a pointer of `int` type and `number` is a `double` type variable.

Since the code `ptr = &number` tries to assign the address of the `double` type variable to `int` type, we will get an error.

In this case, we can use pointer to void or void pointer.

```
void *ptr;
double number = 9.0;
ptr = &number;   // valid
```

Example: Void Pointer

Let us see an example of a `void` pointer.

```
#include <iostream>
using namespace std;
int main() {
// create void pointer
void* ptr;
double number = 2.3;
// assign double address to void
ptr = &number;
cout << "Address of number: " << &number << endl;
cout << "Address pointed to by ptr: " << ptr;
return 0;
}
```

Output

```
Address of number: 0x7ffd0d6cffc8
Address pointed to by ptr: 0x7ffd0d6cffc8
```

In the above example, we have assigned the address of variable `number` to a `void` pointer `ptr`.

When we print the address of `number` and the value of `ptr`, we get the same output.

Dereferencing a Void Pointer (I)

Let's look at the program we have previously written.

```
#include <iostream>
using namespace std;
int main() {
// create void pointer
void* ptr;
double number = 2.3;
// assign double address to void
ptr = &number;
cout << "Address of number: " << &number << endl;
cout << "Address pointed to by ptr: " << ptr;
return 0;
}
```

Here, the void pointer `ptr` points to a `double` variable `number`. We have then used `ptr` to print the address stored inside of it.

But what if we want to print the value stored in the address that `ptr` points to, i.e., to print the value of the `number` variable using `ptr`?

Normally, we'd dereference the pointer to print the value. But this doesn't work for a void pointer. For example,

```
#include <iostream>
using namespace std;
int main() {
// create void pointer
void* ptr;
double number = 2.3;
// assign double address to void
ptr = &number;
// dereference the void pointer
cout << *ptr;
return 0;
}
```

Output

```
error: 'void*' is not a pointer-to-object type
   15 |      cout << *ptr;
      |               ^~~
```

Here, we get this error message because we haven't converted the void pointer to a concrete data type.

To dereference a void pointer, we first need to cast the `void` pointer to point to the specific type of data that we want to access.

For example, if the void pointer is pointing to an `int` value, we must cast the void pointer to point to an `int` data type. This can be done with the help of type casting.

Next, we will see how we can properly deference the void pointer.

Dereferencing a Void Pointer (II)

The syntax to type cast a void pointer is:

```
*(data_type*)pointer_variable
```

So, we write the following code to dereference the void pointer `ptr` that points to the address of a `double` variable.

```
*(double*)ptr
```

Let's see apply this code inside a program.

```
#include <iostream>
using namespace std;
int main() {
// create void pointer
void* ptr;
double number = 2.3;
// assign double address to void
ptr = &number;
// dereference ptr by type casting
// print the value stored in the address pointed to by ptr
cout << "Value in the address pointed to by ptr: " << *(double*)ptr;
return 0;
}
```

Output

```
Value in the address pointed to by ptr: 2.3
```

Here, we have printed the value of the `number` variable by dereferencing `ptr`.

```
cout << "Value in the address pointed to by ptr: " << *(double*)ptr;
```

Important! It is better to use `static_cast` for type casting and dereferencing void pointers. We will learn about this type of casting in our Learn C++ Beyond Basics course.

For now, just remember this syntax for dereferencing void pointers using `static_cast`.

```
// syntax for dereferencing void pointer
*static_cast<data_type*>(pointer_variable)
// code to dereference ptr in the above program
*static_cast<double*>(ptr)
```

Change Value of a Variable Using Void Pointers

In this example, we will use a `void` pointer to change value of a variable.

```
#include <iostream>
using namespace std;
int main() {
// create integer and character variables
int number = 911;
// create void pointer
// assign address of number to ptr
void* ptr = &number;
// print initial value of number
cout << "Initial value: " << *(int*)ptr << endl;
// increase the value of number by 88
*(int*)ptr = *(int*)ptr + 88;
cout << "Final value: " << *(int*)ptr;
return 0;
}
```

Output

```
Initial value: 911
Final value: 999
```

Here, we have increased the value of the `number` variable by 88.

```
// number = number + 88;
*(int*)ptr = *(int*)ptr + 88;
```

Next, we will use `static_cast` to perform this task.

Change Value Of A Variable Using Static Cast

Let's see how we can use `static_cast` to change value of a variable.

```
#include <iostream>
using namespace std;
int main() {
int number = 911;
void* ptr = &number;
// use static_cast for dereferencing
cout << "Initial value: " << *static_cast<int*>(ptr) << endl;
// increase the value of number by 88
*static_cast<int*>(ptr) = *static_cast<int*>(ptr) + 88;
cout << "Final value: " << *static_cast<int*>(ptr);
return 0;
}
```

Alternatively, you can also use the following codes to change the value of `number` using `ptr`.

```
// each line of code below are equivalent to
// number = number + 88;
// Alternative 1
*(int*)ptr = number + 88;
// Alternative 2
number = *(int*)ptr + 88;
// Alternative 3
*static_cast<int*>(ptr) = number + 88;
// Alternative 4
number = *static_cast<int*>(ptr) + 88;
```

Assign Value of a Void Pointer to Another Pointer (I)

Let's look at the program we wrote in the last section.

```
#include <iostream>
using namespace std;
int main() {
int number = 911;
void* ptr = &number;
// use static_cast for derefercning
cout << "Initial value: " << *static_cast<int*>(ptr) << endl;
// increase the value of number by 88
*static_cast<int*>(ptr) = *static_cast<int*>(ptr) + 88;
cout << "Final value: " << *static_cast<int*>(ptr);
return 0;
}
```

Here, we have used the void pointer `ptr` to change the value of `number` and print it.

But what if we want to assign the address in the void pointer `ptr` to an integer pointer `ptr_int`? We can then simply dereference `ptr_int` to access the value of the `number` variable.

We can do this by writing the following code:

```
// assign address in void pointer to integer pointer
int* ptr_int = (int*)ptr;
// use static_cast for assignment
int* ptr_int = static_cast<int*>(ptr);
```

We can then simply dereference this integer pointer to access the value stored in the address.

```
// access the value stored in number
*ptr_int
```

Assign Value of a Void Pointer to Another Pointer (II)

```
#include <iostream>
using namespace std;
int main() {
int number = 911;
void* ptr = &number;
// assign address stored in ptr to an int pointer
int* ptr_int = static_cast<int*>(ptr);
// print value of number by dereferencing ptr_int
cout << "Initial value: " << *ptr_int << endl;
// increase the value of number by 88
*ptr_int = *ptr_int + 88;
cout << "Final value: " << *ptr_int;
return 0;
}
```

Output

```
Initial value: 911
Final value: 999
```

Dangling Pointers

A dangling pointer is a pointer that is used to point to a non-existing memory location (deallocated memory). For example,

```
int* ptr = new int;
cout << ptr;
```

```
// Output: 0x10319e0
```

Here, `ptr` is a pointer that points to the memory address 0x10319e0.

Suppose we deallocate the memory using the following code:

```
// deallocate memory
delete ptr;
```

Now, the memory no longer exists. However, if we print `ptr`, we still get the memory address.

```
// dangling pointer
cout << ptr;
```

```
// Output: 0x10319e0
```

In this case, `ptr` is now a dangling pointer.

💡

Note: Dangling pointers can create a lot of problems in our program. So, it is best to avoid it.

Avoiding Dangling Pointers

We can avoid dangling pointers by setting the pointer value to `NULL` after deallocating the memory. For example,

```
int* ptr = new int;

// deallocate memory
delete ptr;

// assign NULL to ptr
ptr = NULL;

cout << ptr;  // gives 0
```

Here, after deallocating the pointer, we assign `NULL` to the pointer. Now the pointer doesn't point to any memory address.

Hence, it doesn't become a dangling pointer.

Important! If you're using C++ 11 or above, it's better to use `nullptr` instead of `NULL`. This is because `nullptr` was specifically introduced for pointers, while `NULL` actually represents an integer value. This integer value can cause problems in some situations, which we won't discuss here.