C++ advanced level:

Memory Address

In the example from the previous page, the & operator was used to create a reference variable. But it can also be used to get the memory address of a variable; which is the location of where the variable is stored on the computer.

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

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

Example

```
string food = "Pizza";

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

Creating References

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

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

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

Example

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

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

Creating Pointers

You learned from the previous chapter, that we can get the **memory address** of a variable by using the & operator:

Example

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

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

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

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

Example

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

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

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

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

Example explained

Create a pointer variable with the name ptr, that **points to** a string variable, by using the asterisk sign * (string* ptr). Note that the type of the pointer has to match the type of the variable you're working with.

Use the & operator to store the memory address of the variable called food, and assign it to the pointer.

Now, ptr holds the value of food's memory address.

```
string* mystring; // Preferred
string *mystring;
string * mystring;
```

C++ Functions

A function is a block of code which only runs when it is called.

You can pass data, known as parameters, into a function.

Functions are used to perform certain actions, and they are important for reusing code: Define the code once, and use it many times.

Create a Function

C++ provides some pre-defined functions, such as main(), which is used to execute code. But you can also create your own functions to perform certain actions.

To create (often referred to as *declare*) a function, specify the name of the function, followed by parentheses **()**:

Syntax

```
void myFunction() {
 // code to be executed
}
```

Example Explained

- myFunction() is the name of the function
- void means that the function does not have a return value. You will learn more about return values later in the next chapter
- inside the function (the body), add code that defines what the function should do

Call a Function

Declared functions are not executed immediately. They are "saved for later use", and will be executed later, when they are called.

To call a function, write the function's name followed by two parentheses () and a semicolon ;

In the following example, myFunction() is used to print a text (the action), when it is called:

Example

Inside main, call myFunction():

```
// Create a function
void myFunction() {
  cout << "I just got executed!";
}

int main() {
  myFunction(); // call the function
  return 0;
}

// Outputs "I just got executed!"
```

A function can be called multiple times:

Example

```
void myFunction() {
  cout << "I just got executed!\n";
}

int main() {
  myFunction();
  myFunction();
  myFunction();
  return 0;
}

// I just got executed!
// I just got executed!
// I just got executed!
```

C++ Function Parameters

Information can be passed to functions as a parameter. Parameters act as variables inside the function.

Parameters are specified after the function name, inside the parentheses. You can add as many parameters as you want, just separate them with a comma:

Syntax

```
void functionName(parameter1, parameter2, parameter3) {
  // code to be executed
}
```

The following example has a function that takes a string called **fname** as parameter. When the function is called, we pass along a first name, which is used inside the function to print the full name:

Example

```
void myFunction(string fname) {
  cout << fname << " Refsnes\n";
}

int main() {
  myFunction("Liam");
  myFunction("Jenny");
  myFunction("Anja");
  return 0;
}

// Liam Refsnes
// Jenny Refsnes
// Anja Refsnes
```

Default Parameter Value

You can also use a default parameter value, by using the equals sign (=).

If we call the function without an argument, it uses the default value ("Norway"):

Example

```
void myFunction(string country = "Norway") {
  cout << country << "\n";
}

int main() {
  myFunction("Sweden");
  myFunction("India");
  myFunction();
  myFunction("USA");
  return 0;
}

// Sweden
// India
// Norway
// USA
```

Multiple Parameters

Inside the function, you can add as many parameters as you want:

Example

```
void myFunction(string fname, int age) {
  cout << fname << " Refsnes. " << age << " years old. \n";
}

int main() {
  myFunction("Liam", 3);
  myFunction("Jenny", 14);
  myFunction("Anja", 30);
  return 0;
}

// Liam Refsnes. 3 years old.
// Jenny Refsnes. 14 years old.
// Anja Refsnes. 30 years old.
```

Function Overloading

With **function overloading**, multiple functions can have the same name with different parameters:

Example

```
int myFunction(int x)
float myFunction(float x)
double myFunction(double x, double y)
```

Consider the following example, which have two functions that add numbers of different type:

Example

```
int plusFuncInt(int x, int y) {
  return x + y;
}

double plusFuncDouble(double x, double y) {
  return x + y;
}

int main() {
  int myNum1 = plusFuncInt(8, 5);
  double myNum2 = plusFuncDouble(4.3, 6.26);
  cout << "Int: " << myNum1 << "\n";
  cout << "Double: " << myNum2;
  return 0;
}
```

Instead of defining two functions that should do the same thing, it is better to overload one.

In the example below, we overload the plusFunc function to work for both int and double:

Example

```
int plusFunc(int x, int y) {
  return x + y;
}

double plusFunc(double x, double y) {
  return x + y;
}

int main() {
  int myNum1 = plusFunc(8, 5);
  double myNum2 = plusFunc(4.3, 6.26);
  cout << "Int: " << myNum1 << "\n";
  cout << "Double: " << myNum2;
  return 0;
}
```

C++ Classes/Objects

C++ is an object-oriented programming language.

Everything in C++ is associated with classes and objects, along with its attributes and methods. For example: in real life, a car is an **object**. The car has **attributes**, such as weight and color, and **methods**, such as drive and brake.

Attributes and methods are basically **variables** and **functions** that belongs to the class. These are often referred to as "class members".

A class is a user-defined data type that we can use in our program, and it works as an object constructor, or a "blueprint" for creating objects.

Create a Class

To create a class, use the class keyword:

Example

Create a class called "MyClass":

```
class MyClass {       // The class
  public:             // Access specifier
    int myNum;        // Attribute (int variable)
    string myString;  // Attribute (string variable)
};
```

Create an Object

In C++, an object is created from a class. We have already created the class named MyClass, so now we can use this to create objects.

To create an object of MyClass, specify the class name, followed by the object name.

To access the class attributes (myNum and myString), use the dot syntax (.) on the object:

Example

Create an object called "myObj" and access the attributes:

```
class MyClass { // The class
 public: // Access specifier
  int myNum; // Attribute (int variable)
 string myString; // Attribute (string variable)
};

int main() {
 MyClass myObj; // Create an object of MyClass

 // Access attributes and set values
 myObj.myNum = 15;
 myObj.myString = "Some text";

 // Print attribute values
 cout << myObj.myNum << "\n";
 cout << myObj.myString;
 return 0;
}
```

Multiple Objects

You can create multiple objects of one class:

Example

```
// Create a Car class with some attributes
class Car {
 public:
 string brand;
 string model;
  int year;
};

int main() {
 // Create an object of Car
 Car carObj1;
 carObj1.brand = "BMW";
```

```
  carObj1.model = "X5";
  carObj1.year = 1999;

  // Create another object of Car
  Car carObj2;
  carObj2.brand = "Ford";
  carObj2.model = "Mustang";
  carObj2.year = 1969;

  // Print attribute values
  cout << carObj1.brand << " " << carObj1.model << " " << carObj1.year << "\n";
  cout << carObj2.brand << " " << carObj2.model << " " << carObj2.year << "\n";
  return 0;
}
```

Constructors

A constructor in C++ is a **special method** that is automatically called when an object of a class is created.

To create a constructor, use the same name as the class, followed by parentheses ():

Example

```
class MyClass {     // The class
  public:           // Access specifier
    MyClass() {     // Constructor
      cout << "Hello World!";
    }
};

int main() {
  MyClass myObj;    // Create an object of MyClass (this will call the constructor)
  return 0;
}
```

Constructor Parameters

Constructors can also take parameters (just like regular functions), which can be useful for setting initial values for attributes.

The following class have brand, model and year attributes, and a constructor with different parameters. Inside the constructor we set the attributes equal to the constructor parameters (brand=x, etc). When we call the constructor (by creating an object of the class), we pass parameters to the constructor, which will set the value of the corresponding attributes to the same:

Example

```
class Car { // The class
 public: // Access specifier
  string brand; // Attribute
  string model; // Attribute
  int year; // Attribute
  Car(string x, string y, int z) { // Constructor with parameters
   brand = x;
   model = y;
   year = z;
  }
};

int main() {
 // Create Car objects and call the constructor with different values
 Car carObj1("BMW", "X5", 1999);
 Car carObj2("Ford", "Mustang", 1969);

 // Print values
 cout << carObj1.brand << " " << carObj1.model << " " << carObj1.year << "\n";
 cout << carObj2.brand << " " << carObj2.model << " " << carObj2.year << "\n";
 return 0;
}
```

Just like functions, constructors can also be defined outside the class. First, declare the constructor inside the class, and then define it outside of the class by specifying the name of the class, followed by the scope resolution :: operator, followed by the name of the constructor (which is the same as the class):

Example

```
class Car { // The class
 public: // Access specifier
  string brand; // Attribute
  string model; // Attribute
  int year;    // Attribute
    Car(string x, string y, int z); // Constructor declaration
};

// Constructor definition outside the class
Car::Car(string x, string y, int z) {
  brand = x;
  model = y;
  year = z;
}

int main() {
  // Create Car objects and call the constructor with different values
  Car carObj1("BMW", "X5", 1999);
  Car carObj2("Ford", "Mustang", 1969);

  // Print values
  cout << carObj1.brand << " " << carObj1.model << " " << carObj1.year << "\n";
  cout << carObj2.brand << " " << carObj2.model << " " << carObj2.year << "\n";
  return 0;
}
```

Encapsulation

The meaning of **Encapsulation**, is to make sure that "sensitive" data is hidden from users. To achieve this, you must declare class variables/attributes as private (cannot be accessed from outside the class). If you want others to read or modify the value of a private member, you can provide public **get** and **set** methods.

Access Private Members

To access a private attribute, use public "get" and "set" methods:

Example

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

class Employee {
  private:
```

```
  // Private attribute
  int salary;

 public:
  // Setter
  void setSalary(int s) {
    salary = s;
  }
  // Getter
  int getSalary() {
    return salary;
  }
};

int main() {
  Employee myObj;
  myObj.setSalary(50000);
  cout << myObj.getSalary();
  return 0;
}
```

Example explained

The salary attribute is private, which have restricted access.

The public setSalary() method takes a parameter (s) and assigns it to the salary attribute (salary = s).

The public getSalary() method returns the value of the private salary attribute.

Inside main(), we create an object of the Employee class. Now we can use the setSalary() method to set the value of the private attribute to 50000. Then we call the getSalary() method on the object to return the value.

Why Encapsulation?

- It is considered good practice to declare your class attributes as private (as often as you can). Encapsulation ensures better control of your data, because you (or others) can change one part of the code without affecting other parts
- Increased security of data

Inheritance

In C++, it is possible to inherit attributes and methods from one class to another. We group the "inheritance concept" into two categories:

- **derived class** (child) - the class that inherits from another class
- **base class** (parent) - the class being inherited from

To inherit from a class, use the : symbol.

In the example below, the Car class (child) inherits the attributes and methods from the Vehicle class (parent):

Example

```
// Base class
class Vehicle {
 public:
  string brand = "Ford";
  void honk() {
   cout << "Tuut, tuut! \n" ;
  }
};

// Derived class
class Car: public Vehicle {
 public:
  string model = "Mustang";
};

int main() {
 Car myCar;
 myCar.honk();
 cout << myCar.brand + " " + myCar.model;
 return 0;
}
```

Polymorphism

Polymorphism means "many forms", and it occurs when we have many classes that are related to each other by inheritance.

Like we specified in the previous chapter; **Inheritance** lets us inherit attributes and methods from another class. **Polymorphism** uses those methods to perform different tasks. This allows us to perform a single action in different ways.

For example, think of a base class called Animal that has a method called animalSound(). Derived classes of Animals could be Pigs, Cats, Dogs, Birds - And they also have their own implementation of an animal sound (the pig oinks, and the cat meows, etc.):

Example

```
//                              Base                              class
class Animal                                                          {
  public:
    void animalSound()                                                {
          cout      << "The     animal     makes     a     sound     \n";
                                                                      }
};

//                             Derived                            class
class Pig                        : public Animal                      {
  public:
    void animalSound()                                                {
           cout      << "The     pig     says:     wee     wee     \n";
                                                                      }
};

//                             Derived                            class
class Dog : public Animal                                             {
  public:
    void animalSound() {
          cout      << "The     dog     says:     bow     wow     \n";
                                                                      }
};
```

Now we can create Pig and Dog objects and override the animalSound() method:

Example

```
//                              Base                              class
class Animal                                                          {
  public:
    void animalSound()                                                {
          cout      << "The     animal     makes     a     sound     \n";
                                                                      }
};
```

```
// Derived class
class Pig : public Animal {
  public:
    void animalSound() {
      cout << "The pig says: wee wee \n";
    }
};

// Derived class
class Dog : public Animal {
  public:
    void animalSound() {
      cout << "The dog says: bow wow \n";
    }
};

int main() {
  Animal myAnimal;
  Pig myPig;
  Dog myDog;

  myAnimal.animalSound();
  myPig.animalSound();
  myDog.animalSound();
  return 0;
}
```

C++ Exceptions

When executing C++ code, different errors can occur: coding errors made by the programmer, errors due to wrong input, or other unforeseeable things.

When an error occurs, C++ will normally stop and generate an error message. The technical term for this is: C++ will throw an **exception** (throw an error).

C++ try and catch

Exception handling in C++ consist of three keywords: try, throw and catch:

The try statement allows you to define a block of code to be tested for errors while it is being executed.

The throw keyword throws an exception when a problem is detected, which lets us create a custom error.

The catch statement allows you to define a block of code to be executed, if an error occurs in the try block.

The try and catch keywords come in pairs:

Example

```
try {
 // Block of code to try
 throw exception; // Throw an exception when a problem arise
}
catch () {
 // Block of code to handle errors
}
```

Consider the following example:

Example

```
try {
 int age = 15;
 if (age >= 18) {
  cout << "Access granted - you are old enough.";
 } else {
  throw (age);
 }
}
catch (int myNum) {
   cout << "Access denied - You must be at least 18 years old.\n";
   cout << "Age is: " << myNum;
}
```

Example explained

We use the try block to test some code: If the age variable is less than 18, we will throw an exception, and handle it in our catch block.

In the catch block, we catch the error and do something about it. The catch statement takes a **parameter**: in our example we use an int variable (myNum) (because we are throwing an exception of int type in the try block (age)), to output the value of age.

If no error occurs (e.g. if age is 20 instead of 15, meaning it will be be greater than 18), the catch block is skipped:

Example

```
int age = 20;
```

You can also use the throw keyword to output a reference number, like a custom error number/code for organizing purposes:

Example

```
try {
 int age = 15;
 if (age >= 18) {
  cout << "Access granted - you are old enough.";
 } else {
  throw 505;
 }
}
catch (int myNum) {
 cout << "Access denied - You must be at least 18 years old.\n";
 cout << "Error number: " << myNum;
}
```

Handle Any Type of Exceptions (...)

If you do not know the throw **type** used in the try block, you can use the "three dots" syntax (...) inside the catch block, which will handle any type of exception:

Example

```
try {
 int age = 15;
 if (age >= 18) {
  cout << "Access granted - you are old enough.";
 } else {
  throw 505;
                                                                                          }
}
catch (...)                                                                               {
    cout    << "Access    denied    -    You    must    be    at    least    18    years    old.\n";
}
```