Quick ’n Dirty Coding

A Guide for Beginners

Stacey Pellegrino, M.Sc.

Table of Contents

Introduction

C++, Java and Python are powerful and versatile programming languages used for a wide variety of applications. As general-purpose, Object-Oriented Programming (OOP) languages, they enable developers to create efficient, reliable and secure software.

Object-Orientation is a programming paradigm based on the concept of objects, which are data structures composed of data fields and methods that operate on these data fields. OOP aims to abstract away the intricate details of a system by encapsulating data and behaviour within objects. This encapsulation and abstraction allow programmers to hide complexity and present a simplified view of the code, focusing on objects and their interactions rather than the details of implementation.

OOP abstraction in C++, Java and Python involves creating classes. Consider a general class called `Animal`, which serves as the parent class for `Human`, `Cat` and `Dog`. These three classes are subclasses of `Animal` and can access the methods and fields of the `Animal` class. Each of these classes can then be instantiated to become objects, such as a specific `Human`, `Cat` or `Dog`. This approach allows for the creation of abstractions of real-world entities while providing the flexibility to create specific objects that can be interacted with. More details on OOP concepts like classes, polymorphism and inheritance will be discussed later.

Before starting with programming in C++, is the need to install a compiler, which is software that converts code written in a programming language into a machine-readable form. There are many C++ compilers available, including Microsoft Visual C++ and GNU Compiler Collection (GCC) C++ compiler `g++`. Upon installing a suitable compiler, it is then possible to begin writing and compiling C++ programs. Alternatively, Java requires an installation of JDK (Java Development Kit), which includes the Java compiler `javac` and a JRE (Java Runtime Environment). Python uses an interpreter, which executes the code directly without the need for compilation.

To begin writing programs in C++, Java or Python, it is essential to understand the basics of data types, variables and operators. This also includes learning the syntax of the language, control structures, functions and classes. While the syntax and some concepts may differ among these languages, the foundational ideas of programming are consistent, making the transition between them easier for aspiring developers.

By grasping these fundamentals enables exploration of the more advanced features of C++, Java and Python, therefore leveraging the fullest potential of each language in developing robust applications.

To begin writing programs, requires understanding the basics of data types, variables and operators, also including learning the syntax of the language, control structures, functions and classes.

Data Types

C++ supports a variety of data types, including integers, floating-point numbers, characters and strings. Each data type has different properties and uses. For example, an integer is a whole number, while a floating-point number is a number with a decimal point. A character is an equivalent to a single alphabetic letter or alphanumeric symbol, such as “A" or “B”, while a string is a sequence of characters, such as “Hello world”.

These are represented by the following reserved words:

```
int
float
double
char
string
```

Variables

Variables are the fundamental building blocks of any programming language. They are used to store and manipulate data within your program. Variables in C++ must be declared before they can be used. This means that you must specify the type of data that the variable contains, such as an integer, a float, a character or a string. For example, an integer variable would be declared as:

```
int myVariable;
```

Declarations can also be initialised, such as:

```
int i = 0;
int j = 100;

float k = 0.1;

char c = 'A';

string str = "Hello world”;
```

NOTE: Assignments to type `char` uses single quotes and type `string` uses double quotes.

Operators

C++ has a variety of operators that can be used to manipulate data, such as addition `+`, subtraction `-`, multiplication `*`, and division `/`. Other operators include the modulus operator `%`, which returns the remainder of a division, and the increment and decrement operators `++` and `--`. These operators can be used to modify the value of a variable and are commonly used in loops and other control structures.

Syntax

The syntax of C++ is a powerful and complex language that is closely related to the C programming language. The syntax of C++ is very similar to the syntax of C, but with a few key differences. The basic structure of a C++ program is as follows:

```
#include <iostream>

using namespace std;

int main()
{
  // code goes here
  return 0;
}
```

The first line of the program includes the input/output stream header. This header provides access to the functions and objects necessary to input and output data in C++. The second line declares the standard namespace, which provides access to the standard C++ library and global functions. The third line is the main function, which is the entry point for the program. All of the code for the program should be placed within the main function. The last line of the program returns a value of `0` to indicate that the program has completed successfully.

The following code is essentially the first program, that being “Hello world”, to write and compile accordingly:

```
#include <iostream>

using namespace std;

int main()
{
  cout << "Hello world" << "\n";
  return 0;
}
```

Save this code to a file named `hello.cpp` and compile it with the command:

```
$ g++ -o hello hello.cpp
```

The output (as denoted by the `-o` flag and its argument) is a binary named `hello`. You can invoke the execution of this program after compilation by inputting the following command in the same working directory:

```
$ ./hello
```

The following code is an extension and variant to the first program:

```
#include <iostream>

using namespace std;

int main(int argc, char* argv[])
{
  cout << "Hello " << argv[1] << "\n";
  return 0;
}
```

This program is not as robust as it should be. It requires an argument when invoking the binary program but will create a segmentation fault without any argument!

Therefore, it needs to test a condition to handle the missing argument, yet also too many arguments, which is as follows:

```
#include <iostream>

using namespace std;

int main(int argc, char* argv[])
{
  if (argc == 2)
  {
    cout << "Hello " << argv[1] << "\n";
  }
  else if (argc <= 1)
  {
    cerr << "An argument is required" << "\n";
    return 1;
  }
  else if (argc > 2)
  {
    cerr << "Too many arguments given" << "\n";
    return 1;
  }
  else
  {
    cerr << "An unexpected error occurred" << "\n";
    return 1;

  }

  return 0;
}
```

However, this could be refactored with combining two `if` statements into one as such:

```
  else if (argc <= 1 || argc > 2) { ...
```

NOTE: The logical OR operator is represented as `||` yet the logical AND operator is represented with `&&`.

When the binary executable is run the following output is generated:

```
$ ./hello
An argument is required

$ echo $?
1

$ ./hello John Smith
Too many arguments given

$ echo $?
1

$ ./hello John
Hello John

$ echo $?
0
```

NOTE: The Bash shell variable `$?` stores the last execution return code, with `0` meaning successful execution.

To process more than one argument requires a for loop as an addition to the code, which is as follows:

```
#include <iostream>

using namespace std;

int main(int argc, char* argv[])
{
  if (argc >= 2)
  {
    cout << "Hello";

    for (int i = 1; i < argc; i++)
    {
      cout << " " << argv[i];
    }
    cout << "\n";
  }
  else if (argc <= 1)
  {
    cerr << "One or more arguments is required" << "\n";
    return 1;
  }
  else
  {
    cerr << "An unexpected error occurred" << "\n";
    return 1;
  }

  return 0;
}
```

NOTE: the array `argv[0]` is the name of the binary executed, that being `./hello` in this specific case.

To understand the progression of the C++ code samples above requires further knowledge of control structures and arrays (to be discussed later). So what about Java and Python?

The following code is “Hello world” in Java:

```
class Hello
{
      public static void main(String args[])
      {
            System.out.println("Hello world");
      }
}
```

Save this code to a file named `Hello.java` (notice that the first character is uppercase) and compile it with the command:

```
$ javac Hello.java
```

You can then invoke the execution of this program after compilation into byte-code by inputting the following command in the same working directory:

```
$ java Hello
```

The following code is “Hello world” in Python:

```
#!/usr/bin/python3

print("Hello world");
```

Save this code to a file named `hello.py` and make it executable with the command:

```
$ chmod +x hello.py
```

You can invoke the execution of this program with the Python interpreter by inputting the following command in the same working directory:

```
$ ./hello.py
```

Passing and processing arguments to a C++ program (see previous example above) can be achieved in Java and Python as well.

The following Java program `Args.java` is similar to the previous C++ example:

```
class Args
{
      public static void main(String args[])
      {
            if (args.length >= 1)
            {
                  System.out.print("Hello");
                  for (int index = 0; index < args.length; index++)
                  {
                        System.out.print(" " + args[index]);
                  }
                  System.out.println();
            }
            else if (args.length == 0)
            {
                  System.err.println("One or more arguments is required");
                  System.exit(1);
            }
            else
            {
                  System.err.println("An unexpected error occurred");
                  System.exit(1);
            }
      System.exit(0);

      }
}


$ javac Args.java

$ java Args
One or more arguments is required

$ java Args John
Hello John

$ java Args John Smith
Hello John Smith
```

The following Python program `args.py` is also similar to the original C++ example:

```
#!/usr/bin/python3

import sys

def main(argc, argv):
      if argc >= 2:
            print("Hello", end = '');

            i = 1;

            while i < argc:
                  print(" " + argv[i], end = '');
                  i += 1;
            print();
      elif argc <= 1:
            sys.stderr.write("One or more arguments is required\n")
            sys.exit(1);
      else:
            sys.stderr.write("An unexpected error occurred\n")
            sys.exit(1);

      sys.exit(0);

if __name__ == '__main__':
    main(len(sys.argv), sys.argv);


$ chmod +x args.py

$ ./args.py
```

Control Structures

Control structures are essential for any programming language. They allow the programmer to control the flow of their code and make decisions based on certain conditions. In C++, the primary control structures are the `if` and `else` statement, the `switch` statement, and the loop structures (`while`, `do while`, and `for`). These structures allow the programmer to create branching logic, loop through code multiple times and create complex conditions.

Arrays

Arrays are data structures that store multiple elements of the same type. Arrays are a powerful tool for storing and manipulating data in C++, Java and Python. For easier understanding arrays are likened to a list represented by a contiguous block of memory that stores elements of the same type in a linear fashion.

An array can be declared by specifying the type of data it holds, followed by the array name and the size of the array. For example, to declare an array of 10 integers in C++, the syntax would be:

```
int myArray[10];
```

Here, `myArray` is the name of the array and `10` is the size of the array.

An array can be initialised by assigning values to each element of the array. For example, to initialise the array declared above with the numbers from 0 to 9, the syntax would be:

```
int myArray[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
```

Here, each element in the array is assigned a corresponding value.

To access an element from an array, we need to specify the index of the element. The index of the first element of an array is `0` and the last element is the size of the array minus one. For example, to access the fourth element of the array, the syntax would be:

```
int fourthElement = myArray[3];
```

Here, the fourth element is accessed by specifying its index.

Array elements can be manipulated by using arithmetic and logical operators. For example, to increment the first element of the array by one, the syntax would be:

```
myArray[0]++;
```

Here, the first element is incremented by one.

Functions

Functions are used to encapsulate code, making it easier to read and maintain. Functions allow the programmer to reuse code, reducing the time spent on writing and debugging code. Functions in C++ have a specific syntax, which includes the function name, parameters and the return type. The value returned by the function can be used in other parts of the program. Functions can also be declared to return no value, in which case they are called void functions.

Classes

Classes are the foundation of Object-Oriented Programming (OOP), which is a paradigm that focuses on the relationship between different objects. In C++, Java and Python objects are defined and instantiated from classes. Each object created from a class will have its own set of properties and methods, which can be used to manipulate the object. Effectively each class is a blueprint that can be used to create objects. Like functions, classes are used to encapsulate code. The syntax of classes in C++ is similar to that of functions, but there are some important differences. Classes have member variables (attributes), member functions (methods) and constructors, which are used to create and initialise the object. Classes also have access

modifiers, namely getters and setters, which are used to control the access that other parts of code have to the class members. Furthermore, classes can also be structurally composed of elements that are scoped as `public`, `private` and `protected`. A class diagram design can be done with aspects of UML (Unified Modelling Language) as a means for a diagrammatic blueprint.

In C++, classes are declared using the keyword `class`. For example, a class to represent a 3D shape could be declared as:

```
class Shape
{
  public:
    void calculateVolume();
  private:
    int length;
    int width;
    int height;
};
```

Inheritance

Inheritance allows a derived class to inherit the properties and methods of a base class. For example, a class `Person` could be used as a base class, and then derived classes such as `Student` and `Teacher` could be created which inherit the properties of `Person`. This allows the derived classes to have access to the properties and methods of `Person`, as well as their own properties and methods. Another example could be that of an `Animal` class, with derived classes of species.

A base class called `Animal` (containing generic animal attributes and methods) could be written as follows:

```
class Animal {
public:
    int age;
    int legs;
    string type;

    void move() {
        cout << "The animal is moving" << endl;
    }
};
```

A subclass called `Dog` inherits from `Animal`:

```
class Dog : public Animal {
public:
    string breed;

    void bark() {
        cout << "Woof!" << endl;
    }
};
```

The following demonstrates how to create a `Dog` object and to access both the `Animal` and `Dog` class attributes and methods:

```
Dog dog;
dog.legs = 4;
dog.type = "Mammal";
dog.breed = “Yorkshire Terrier";
dog.age = 3;
dog.move();
dog.bark();
```

This example shows how inheritance can provide additional features to a class without having to rewrite the same code for each class. NOTE: The class variables (attributes) are scoped as public! However, it is typical to have `public` class methods to get/set attributes that are scoped `private` in the class.

Therefore, the classes for `Animal` and `Dog` could be written better and refactored as follows:

```
class Animal {

public:
    int getAge() {
        return age;
    }

    void setAge(int age) {
        this->age = age;
    }

    void setLegs(int l) {
        legs = l;
    }

    int getLegs() {
        return legs;
    }

    void setType(string t) {
        type = t;
    }

    string getType() {
        return type;
    }

    void move() {
        cout << "The animal is moving" << endl;
    }

private:
    int age;
    int legs;
    string type;
};


class Dog : public Animal {
public:
    string getBreed() {
        return breed;
    }

    void setBreed(string breed) {
        this->breed = breed;
    }

    void bark() {
        cout << "Woof!" << endl;
    }
private:
    string breed;
};
```

What follows demonstrates the concept of inheritance in more detail, comparing implementations in C++, Java and Python.

C++ inheritance ALL in one file

The following file is `animal.cpp`:

```
#include <iostream>

using namespace std;

// Base class
class Animal {
public:
      void eat() {
            cout << "I can eat!" << endl;
      }

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



// Derived class
class Dog : public Animal {
public:
      void bark() {
            cout << "I can bark! Woof woof!!" << endl;
      }
};

// Another derived class
class Cat : public Animal {
public:
      void meow() {
            cout << "I can meow! Meow meow!!" << endl;
      }
};

int main() {

    // Create object of the Dog class
    Dog dog;

    // Calling members of the base class
    dog.eat();
    dog.sleep();

    // Calling member of the derived class
    dog.bark();

    // Create object of the Cat class
    Cat cat;

    // Calling members of the base class
    cat.eat();
    cat.sleep();

    // Calling member of the derived class
    cat.meow();

    return 0;
}
```

Compile this code with:

```
$ g++ -o animal animal.cpp
```

Execute this code with:

```
$ ./animal
```

C++ inheritance in multiple files

The following separate header file is `animal.h`:

```
#ifndef ANIMAL_H
#define ANIMAL_H

#include <iostream>

using namespace std;

class Animal {
public:
      Animal();
      void move();
      void eat();
      void sleep();

      void setAge(int);
      int getAge();

      void setLegs(int);
      int getLegs();

      void setBreed(string);
      string getBreed();

private:
      int age;
      int legs;
      string breed;
};

#endif
```

The following separate source file is `animal.cpp`:

```
#include "animal.h"

Animal::Animal() { }

void Animal::move() {
      cout << "The animal is moving" << endl;
}

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

void Animal::sleep() {
      cout << "I can sleep!" << endl;
}

void Animal::setAge(int a) {
      age = a;
}

int Animal::getAge() {
      return age;
}

void Animal::setLegs(int l) {
      legs = l;
}

int Animal::getLegs() {
      return legs;
}

void Animal::setBreed(string t) {
      breed = t;
}

string Animal::getBreed() {
      return breed;
}
```

The following separate header file is `cat.h`:

```
#ifndef CAT_H
#define CAT_H

#include "animal.h"

#include <iostream>

using namespace std;

class Cat : public Animal {
public:
      Cat();
      void meow();
};

#endif
```

The following separate source file is `cat.cpp`:

```
#include "cat.h"

Cat::Cat() { }
void Cat::meow() {
      cout << "Meow!" << endl;
}
```

The following is the separate header file `dog.h`:

```
#ifndef DOG_H
#define DOG_H

#include "animal.h"

#include <iostream>

using namespace std;

class Dog : public Animal {
public:
      Dog();
      void bark();
};

#endif
```

The following is the separate source file `dog.cpp`:

```
#include "dog.h"

Dog::Dog() { }
void Dog::bark() {
      cout << "Bark!" << endl;
}
```

Bringing this all together is the main source file `main.cpp`:

```
#include "dog.h"
#include "cat.h"

#include <iostream>

using namespace std;

int main()
{
      cout << "Dog:" << endl;

      // Create object of the Dog class
      Dog dog;

      // Calling members of the base class
      dog.eat();
      dog.sleep();
```

```
    // Calling member of the derived class
    dog.bark();

    // setter (aka mutator method)
    dog.setAge(7);

    // getter (aka accessor method)
    cout << "Age: " << dog.getAge() << endl;

    // setter (aka mutator method)
    dog.setBreed("Yorkshire Terrier”);

    // getter (aka accessor method)
    cout << "Breed: " << dog.getBreed() << endl;

    // setter (aka mutator method)
    dog.setLegs(4);

    // getter (aka accessor method)
    cout << "Number of legs: " << dog.getLegs() << end;

    cout << endl << "Cat:" << endl;

    // Create object of the Cat class
    Cat cat;

    // Calling members of the base class
    cat.eat();
    cat.sleep();


    // Calling member of the derived class
    cat.meow();

    // setter (aka mutator method)
    cat.setAge(3);

    // getter (aka accessor method)
    cout << "Age: " << cat.getAge() << endl;

    // setter (aka mutator method)
    cat.setBreed(“Persian");

    // getter (aka accessor method)
    cout << "Breed: " << cat.getBreed() << endl;

    // setter (aka mutator method)
    cat.setLegs(4);

    // getter (aka accessor method)
    cout << "Number of legs: " << cat.getLegs() << endl;

    return 0;
}
```

Compile this code with:

```
$ g++ -o animal main.cpp animal.cpp cat.cpp dog.cpp
```

Execute this code with:

```
$ ./animal
Dog:
I can eat!
I can sleep!
Bark!
Age: 7
Breed: Yorkshire Terrier
Number of legs: 4

Cat:
I can eat!
I can sleep!
Meow!
Age: 3
Breed: Persian
Number of legs: 4
```

Java inheritance ALL in one file

The following file is `Main.java`:

```
// Base class
class Animal {
      public void eat() {
            System.out.println("I can eat!");
      }

      public void sleep() {
            System.out.println("I can sleep!");
      }
}

// Derived class
class Dog extends Animal {
      public void bark() {
            System.out.println("I can bark! Woof woof!!");
      }
}

// Another derived class
class Cat extends Animal {
      public void meow() {
            System.out.println("I can meow! Meow meow!!");
      }
}

class Main {
      public static void main(String args[]) {

            // Create object of the Dog class
            Dog dog = new Dog();

            // Calling members of the base class
            dog.eat();
            dog.sleep();

            // Calling member of the derived class
            dog.bark();

            // Create object of the Cat class
            Cat cat = new Cat();

            // Calling members of the base class
            cat.eat();
            cat.sleep();

            // Calling member of the derived class
            cat.meow();
      }
}
```

Compile the code with:

```
$ javac Main.java
```

Execute the code with:

```
$ java Main
```

Java inheritance in multiple files

The following is the separate source file `Animal.java`:

```
class Animal {

      private int age;
      private int legs;
      private String breed;

      public void eat() {
            System.out.println("I can eat!");
      }
```

```
	public void sleep() {
		System.out.println("I can sleep!");
	}

	public void setAge(int a) {
		age = a;
	}

	public int getAge() {
		return age;
	}

	public void setLegs(int l) {
		legs = l;
	}

	public int getLegs() {
		return legs;
	}

	public void setBreed(String type) {
		breed = type;
	}

	public String getBreed() {
		return breed;
	}
}
```

The following is the separate source file `Cat.java`:

```
class Cat extends Animal {
	public void meow() {
		System.out.println("I can meow! Meow meow!!");
	}
}
```

The following is the separate source file `Dog.java`:

```
class Dog extends Animal {
	public void bark() {
		System.out.println("I can bark! Woof woof!!");
	}
}
```

Bringing this all together is the main source file `Main.java`:

```
class Main {
	public static void main(String args[]) {

		// Create object of the Dog class
		Dog dog = new Dog();


		// Calling members of the base class
		dog.eat();
		dog.sleep();

		// Calling member of the derived class
		dog.bark();

		dog.setAge(7);                                    // mutator
		System.out.println("Age: " + dog.getAge());       // accessor

		dog.setBreed("Yorkshire Terrier");                // mutator
		System.out.println("Breed: " + dog.getBreed());   // accessor

		dog.setLegs(4);                                   // mutator
		System.out.println("Legs: " + dog.getLegs());     // accessor

		System.out.println();

		// Create object of the Cat class
		Cat cat = new Cat();
```

```
            // Calling members of the base class
            cat.eat();
            cat.sleep();

            // Calling member of the derived class
            cat.meow();

            cat.setAge(3);                                      // mutator
            System.out.println("Age: " + cat.getAge());         // accessor

            cat.setBreed("Persian");                            // mutator
            System.out.println("Breed: " + cat.getBreed());     // accessor

            cat.setLegs(4);                                     // mutator
            System.out.println("Legs: " + cat.getLegs());       // accessor
      }
}
```

Compile the source with:

```
$ javac Main.java
```

Execute the source with:

```
$ java Main
I can eat!
I can sleep!
I can bark! Woof woof!!
Age: 7
Breed: Yorkshire Terrier
Legs: 4

I can eat!
I can sleep!
I can meow! Meow meow!!
Age: 3
Breed: Persian
Legs: 4
```

Python inheritance ALL in one file

The following file is `animal.py`:

```
#!/usr/bin/python3

class Animal:
      def eat(self):
            print("I can eat!");

      def sleep(self):
            print("I can sleep!");

class Dog(Animal):
      def bark(self):
            print("I can bark! Woof woof!!");

class Cat(Animal):
      def meow(self):
            print("I can meow! Meow meow!!");

dog = Dog();

dog.eat();
dog.sleep();

dog.bark();

cat = Cat();

cat.eat();
cat.sleep();

cat.meow();
```

```
$ chmod +x animal.py

$ ./animal.py
```

Python inheritance in multiple files

The following file is `animal.py`:

```
class Animal:

      def getAge(self):
            return self.__age

      def setAge(self, value):
            self.__age = value

      def getLegs(self):
            return self.__legs

      def setLegs(self, value):
            self.__legs = value


      def getBreed(self):
            return self.__breed

      def setBreed(self, value):
            self.__breed = value

      def eat(self):
            print("I can eat!");

      def sleep(self):
            print("I can sleep!");
```

The following file is `cat.py`:

```
from animal import *

class Cat(Animal):
      def meow(self):
            print("I can meow! Meow meow!!");
```

The following file is `dog.py`:

```
from animal import *

class Dog(Animal):
      def bark(self):
            print("I can bark! Woof woof!!");
```

Bringing this all together is the main source file `main.py`:

```
#!/usr/bin/python3

from cat import *
from dog import *

print("Dog");
dog = Dog();

dog.eat();
dog.sleep();

dog.bark();

dog.setAge(7);
print("Age:", dog.getAge());

dog.setLegs(4);
print("Legs:", dog.getLegs());
```

```
dog.setBreed("Yorkshire Terrier");
print("Breed:", dog.getBreed());

print("\nCat");

cat = Cat();

cat.eat();
cat.sleep();

cat.meow();

cat.setAge(3);
print("Age:", cat.getAge());

cat.setLegs(4);
print("Legs:", cat.getLegs());

cat.setBreed("Persian");
print("Breed:", cat.getBreed());
```

Make the main source file executable with:

```
$ chmod +x main.py
```

Execute the code with:

```
$ ./main.py
```

Polymorphism

Polymorphism allows objects of different types to be treated the same way. For example, a base class `Shape` could be used to define the `area()` function, and then derived classes such as `Rectangle` and `Circle` could each implement their own `area()` function. In this way, the same function can be used to calculate the area of different shapes, and the correct version will be called depending on the type of object being used.

Some examples of polymorphism in C++ include:

- Function overloading is a type of polymorphism in which we can have multiple functions with the same name but different parameters. For example, we can have two `add()` functions, one that takes two integers and another that takes two floats. Both functions will have the same name but different parameters.
- Operator overloading is a type of polymorphism in which we can use the same operator in different ways. For example, we can use the `+` operator to add two integers, two floats or two strings. All of these operations will use the same operator but will have different results.
- Virtual functions are a type of polymorphism in which we can use the same function in different ways. For example, we can have a base class with a virtual function, and then have several derived classes that implement the virtual function in different ways. This allows us to call the same function on objects of different types and get different results.

A detailed example of using C++ polymorphism for shapes:

- Create a base `Shape` class with virtual member functions for drawing, rotating, and changing size.
- Derive classes from the base `Shape` class for each type of shape: `Circle`, `Square`, `Rectangle`, etc.
- Override the virtual functions in each derived class to implement the specific drawing, rotation and size changing behaviour for each shape.

The following C++ code demonstrates a basic implementation of polymorphism:

```
// Base Shape class
class Shape {
public:
    virtual void draw() = 0;
    virtual void rotate() = 0;
    virtual void changeSize() = 0;
};
```

```
    // Circle derived from Shape
    class Circle : public Shape {
    public:
        void draw() override {
            // Draw a circle
        }
        void rotate() override {
            // Rotate the circle
        }
        void changeSize() override {
            // Change size of the circle
        }
    };

    // Square derived from Shape
    class Square : public Shape {
    public:
        void draw() override {
            // Draw a square
        }
        void rotate() override {
            // Rotate the square
        }
        void changeSize() override {
            // Change size of the square
        }
    };

    // Rectangle derived from Shape
    class Rectangle : public Shape {
    public:
        void draw() override {
            // Draw a rectangle
        }
        void rotate() override {
            // Rotate the rectangle
        }
        void changeSize() override {
            // Change size of the rectangle
        }
    };
```

For testing the code:

```
    int main() {

        Circle circle;
        circle.draw();
        circle.rotate();
        circle.changeSize();

        return 0;
    }
```

Aggregation

Aggregation in C++ is a process of combining two different classes into a single class. It allows a class to contain an object of another class as its member. This type of relationship is known as composition or aggregation.

In the following example, the class `Car` is composed of an object of the class `Engine` as its member:

```
class Engine {
public:
    int cylinder_count;
    double capacity;
};

class Car {
public:
    Engine engine;
    string color;
    string brand;
};
```

```
int main() {
    Car car;
    car.engine.cylinder_count = 4;
    car.engine.capacity = 1500;
    car.color = "Red";
    car.brand = "Toyota";
    return 0;
}
```

Once the basics of C++ have been mastered, it follows to explore the more advanced features of the language, such as exceptions, templates and the Standard Template Library. These features allow developers to write more efficient and robust programs.

Exceptions

Exceptions are a powerful feature of the C++ language that allow for the handling of errors in a more structured manner. Exceptions provide a way to capture errors and handle them in a more organised way. They can be used to catch errors before they cause a crash and can be used to provide custom error messages. In addition, exceptions provide a way to transfer control flow to different parts of the program. This allows for complex applications to handle errors in a more efficient and reliable way. The following is an example of this:

```
try
{
    int a = 10;
    int b = 0;
    int c = a / b;
}
catch(const std::exception& e)
{
    std::cerr << "Exception thrown: " << e.what() << '\n';
}
```

Templates

Templates are powerful features of the C++ language that allow for generic programming. They allow for the creation of type-agnostic functions and classes. This means that functions and classes can be written without having to specify the type of data that is being used. This allows for the same code to be reused for different data types, which is especially useful when the same code is needed for multiple data types. In addition, templates can be used to create generic algorithms that can be applied to any data type.

Example 1

```
template <typename T>
class Test
{
public:
    Test(T x)
    {
        data = x;
    }

    T getData()
    {
        return data;
    }

private:
    T data;
};
```

```
int main()
{
    Test<int> t1(12);
    Test<float> t2(5.5);

    cout << t1.getData() << endl;
    cout << t2.getData() << endl;

    return 0;
}
```

Example 2

```
template <class T>
class Pair
{
public:
    Pair(T x, T y)
    {
        first = x;
        second = y;
    }

    T add()
    {
        return (first + second);
    }

private:
    T first, second;
};

int main()
{
    Pair<int> pair1(6, 5);
    Pair<float> pair2(3.5, 5.5);

    cout << pair1.add() << endl;
    cout << pair2.add() << endl;

    return 0;
}
```

Standard Template Library

The STL (Standard Template Library) is a set of template-based containers and algorithms that are part of the C++ language. The containers provide a way to store and manipulate data in a type-agnostic way. This means that the same containers can be used for different data types. The algorithms provide a set of functions to perform tasks, such as sorting and searching, on the data stored in the containers. The STL is a powerful tool that allows for the efficient manipulation of data and is essential for the development of more complex applications.

Examples of STL include `Vector`, `List`, `Map` and `Set`, which can be demonstrated as follows:

```
// 1. Vector:
#include <iostream>
#include <vector>

int main()
{
    // Declaring a vector
    std::vector<int> numbers;

    // Adding elements using push_back()
    numbers.push_back(1);
    numbers.push_back(2);
    numbers.push_back(3);
    numbers.push_back(4);
    numbers.push_back(5);
```

```
    // Accessing elements of vector using at()
    for (int i = 0; i < numbers.size(); i++) {
        std::cout << numbers.at(i) << " ";
    }

    return 0;
}


// 2. List:
#include <iostream>
#include <list>

int main()
{
    // Declaring a list
    std::list<int> numbers;

    // Adding elements using push_back()
    numbers.push_back(1);
    numbers.push_back(2);
    numbers.push_back(3);
    numbers.push_back(4);
    numbers.push_back(5);

    // Accessing elements of list using iterator
    std::list<int>::iterator it;
    for (it = numbers.begin(); it != numbers.end(); it++) {
        std::cout << *it << " ";
    }

    return 0;
}


// 3. Map:
#include <iostream>
#include <map>

int main()
{
    // Declaring a map
    std::map<int, int> numbers;

    // Adding elements using insert()
    numbers.insert(std::pair<int, int>(1, 2));
    numbers.insert(std::pair<int, int>(2, 4));
    numbers.insert(std::pair<int, int>(3, 6));
    numbers.insert(std::pair<int, int>(4, 8));
    numbers.insert(std::pair<int, int>(5, 10));

    // Accessing elements of map using iterator
    std::map<int, int>::iterator it;
    for (it = numbers.begin(); it != numbers.end(); it++) {
        std::cout << it->first << " : " << it->second << std::endl;
    }

    return 0;
}


// 4. Set:
#include <iostream>
#include <set>

int main()
{
    // Declaring a set
    std::set<int> numbers;

    // Adding elements using insert()
    numbers.insert(1);
    numbers.insert(2);
    numbers.insert(3);
    numbers.insert(4);
    numbers.insert(5);

    // Accessing elements of set using iterator
    std::set<int>::iterator it;
```

```
    for (it = numbers.begin(); it != numbers.end(); it++) {
        std::cout << *it << " ";
    }

    return 0;
}
```

Memory management and pointers

C++ also provides access to low-level features such as memory management and pointers, which allow developers to have more control over the performance of their programs. Memory management and pointers are two of the most fundamental and important concepts in C++ programming. They are the cornerstones of the language, and understanding how they work is essential for mastering C++. However, this is not the case for Java and Python, as these approach memory management in a different but ultimately safer way.

Memory management is the process of allocating and managing the memory of a computer. In C++ this is done through the use of variables, dynamic memory allocation and garbage collection. Variables are used to store data and they must be allocated memory in order to be used. This memory can be allocated in either a static or dynamic fashion. Static memory allocation is done at compile time and is the default behaviour of C++. This means that when a variable is declared, it is assigned a fixed amount of memory. Dynamic memory allocation is done at runtime and involves assigning memory to a variable as it is needed. This allows for the efficient use of memory, as the amount of memory allocated is only as much as is needed. Garbage collection is used to free up unused memory. This is done by tracking which variables are being used and then reclaiming memory from variables that are no longer needed.

Pointers are another essential concept in C++ programming. A pointer is a variable that contains the address of another variable. Pointers are used to access the value of a variable or to refer to the variable itself. They are also used to manipulate memory locations directly. In C++ pointers are used to create dynamic data structures such as linked lists and trees. Pointers can also be used to pass parameters to functions and to create references to objects. The following code shows memory management with pointers including passing by reference and value:

```
#include <iostream>

using namespace std;

void passByRef(int *ptr)     // Function to pass a value by reference
{
    (*ptr)++;                // Increment the value of the pointer

    cout << "Value of pointer inside passByRef(): " << *ptr << endl;
}

void passByValue(int num)    // Function to pass a value by value
{
    num++;                   // Increment the value of the parameter

    cout << "Value of parameter inside passByValue(): " << num << endl;
}

int main()
{
    int *ptr;                // Declare a pointer to an integer
    ptr = new int;           // Allocate memory for an integer
    *ptr = 5;                // Assign a value to the pointer

    cout << "Value of \"ptr\": " << ptr << endl;    // Print the address of the pointer "ptr"
    cout << "Value of \"*ptr\": " << *ptr << endl;  // Print value at address of pointer "ptr"
    cout << "Value of \"&ptr\": " << &ptr << endl;  // Print address of the pointer "ptr" itself
                                                    // NOTE: &ptr is called the address operator
    cout << endl;
    cout << "Value of pointer before passByRef(): " << *ptr << endl;

    passByRef(ptr);          // Call passByRef()

    cout << "Value of pointer after passByRef(): " << *ptr << endl;

    int num = 10;            // Declare an integer
    cout << "Value of variable before passByValue(): " << num << endl;

    passByValue(num);        // Call passByValue()
```

```
    cout << "Value of variable after passByValue(): " << num << endl;

    delete ptr;              // Free the memory

    return 0;
}
```

This program has the following output upon execution:

```
Value of "ptr": 0x6000029d4020
Value of "*ptr": 5
Value of "&ptr": 0x7ff7be6aaa50

Value of pointer before passByRef(): 5
Value of pointer inside passByRef(): 6
Value of pointer after passByRef(): 6
Value of variable before passByValue(): 10
Value of parameter inside passByValue(): 11
Value of variable after passByValue(): 10
```

In summary, memory management and pointers are two of the most important concepts in C++ programming by allocating and managing memory. Understanding how they work is essential for mastering the language. With a thorough understanding of these concepts, C++ developers can create powerful and efficient program as a proficient and effective programmer.

Conclusion

This tutorial is designed to provide a comprehensive “quick ’n dirty” introduction to the C++, Java and Python programming languages but with more of a focus on C++ due to its associated complexity and depth required to master it accordingly. With reference to what has been demonstrated it should be apparent that C++, Java and Python are powerful and versatile languages. The C++ code referenced in this document showed many parallel implementations in Java and Python to illustrate the similarities and differences between these languages. Grasping these fundamentals and exploring the more advanced features of C++, Java and Python should enable a greater potential to develop robust applications with each language. Furthermore, the skills developed from this tutorial are very much transferrable, especially with software developed in C++ considering it is used in a variety of applications, from low-level embedded systems to high-performance gaming engines. C++ is regarded as a language that has a high-performance execution of compiled code (only C and assembly language are more efficient, although FORTRAN is better regarding numerics) but C++ has the added capability of OOP. However, worth noting is that Java and Python are regarded as much easier where syntax is concerned, although compromising execution performance in comparison to C++. Nonetheless, both Java and Python do come with extensive libraries, therefore providing powerful alternatives as part of a heterogeneous development toolkit for rapid application development. The skills developed through this tutorial should enable and provide a great foundation for advancing further learning in software development.