C++

Professional Programming

Michael Lappenbusch
IT-SPECIALIST APPLICATION DEVELOPMENT

Table of contents

Introduction to C++

1.1 What is C++?

C++ is a general programming language developed by Bjarne Stroustrup in 1983. It is an extension of the C programming language and was originally developed to simplify the programming of large and complex systems. C++ adds a number of advanced features to C, such as object-oriented programming, templates, and exception handling.

C++ is a compiled language, which means that the source code is translated into machine-readable binaries before being run on a computer. This allows C++ to achieve high execution speed, making it a suitable choice for system programming, game development, application development, and many other uses.

C++ has seen a large number of extensions and improvements in recent years, including standardization by the ISO (International Standards Organization) in 1998 and the release of C++11, C++14, and C++17. These standards have made C++ a modern and powerful language that continues to be used in many areas.

1.2 History of C++

C++ was developed at Bell Labs in 1983 by Bjarne Stroustrup, a Danish computer scientist. Stroustrup began developing C++ to simplify the programming of large and complex systems by adding a number of advanced features to the C programming language. These functions included in particular object-oriented programming, templates and exception handling.

The first version of C++ was released in 1983 and was called C with Classes. Over the years, C++ has continued to improve and expand. In 1985 the first public version of C++ was published and in 1989 the first standardization was carried out. In 1998, C++ was standardized by the ISO (International Standards Organization).

The latest release of C++ is C++20 which was released in July 2020, C++20 contains many new features and extensions such as Concepts, Modules, Coroutines and Ranges and others.

C++ has seen a large number of enhancements and improvements over the last few years. These standards have made C++ a modern and powerful language that continues to be used in many areas. C++ is used in systems programming, game development, application development, artificial intelligence, machine learning, and many other fields.

1.3 Comparison with other programming languages

C++ shares some similarities with other programming languages, especially C since it is an extension of C. However, it also has important differences, particularly in terms of support for object-oriented programming, templates, and exception handling.

Comparing C++ to other languages such as Java or C# shows that C++ offers greater power and control as it allows for a lower level of programming, but also has a higher learning curve and error proneness. C++ requires more manual memory management and is not as platform independent as Java or C#.

A comparison with Python shows that C++ has higher performance and speed, but also has a higher learning curve and error-proneness. However, Python is a simpler language and offers higher code readability and maintainability.

A comparison with C# or Java shows that C++ offers better performance and control, but also has a higher learning curve and error-proneness. However, C# and Java are more platform-independent and offer higher code maintainability and security.

It is important to emphasize that the choice of programming language often depends on the needs of the project and the experience of the developer. C++ is a powerful and versatile language used in many fields, but it may not be the best choice for every project.

1.4 Setting up a development environment

To start developing C++ applications, you must first set up a development environment. A development environment consists of a text editor or IDE (Integrated Development Environment) to write the code, a compiler to translate the code into machine-readable binaries, and a debugger to test and debug the code.

One of the easiest ways to set up a development environment for C++ is to use a free and open source IDE like Visual Studio Code or Code::Blocks. These IDEs already contain all the necessary tools for writing, compiling and debugging C++ code.

If you use Visual Studio Code, you must install a compiler and debugger to compile and debug C++ code. The most commonly used compiler for C++ is GCC (GNU Compiler Collection). You can install GCC either from the command line or by adding an extension in Visual Studio Code.

If you use Code::Blocks, it already includes a compiler and debugger. All you have to do is install Code::Blocks on your computer and open it to start developing C++ applications.

There are also many other IDEs and text editors available that are supported, such as Eclipse, NetBeans, Xcode, etc. It is important that you feel comfortable and confident with the development environment in order for it to help you when developing C++ applications can help.

basics of the language

2.1 Data Types and Variables

In C++, data types are how information is stored in the computer's memory. C++ supports a variety of data types, such as integers, floating point, characters, and Boolean values. Each data type has its own limitations and applications.

The most common numeric data types in C++ are int (for integers) and double (for floating point). For example, you can declare an integer like this:

```
int a = 5;
```

In this example, the variable a is declared as int and initialized with the value 5.

Another important data type in C++ is the boolean data type (bool), which can only hold the values true or false. For example, you can declare a boolean value like this:

```
boolean b = true;
```

Another important data type in C++ is the char data type, which can store a single character. For example, you can declare a character like this:

```
char c = 'a';
```

Variables in C++ are placeholders for values that can change over time. A variable must be declared before it can be used, by assigning it a data type and a name. For example, you can declare a variable like this:

```
int x;
```

A variable can also be initialized with an initial value when it is declared. For example, you can initialize a variable like this:

```
int y = 10;
```

It is important to note that each variable has a specific data type and can only store values of that type. Attempting to store a value of a different type in a variable may result in errors. It is also important that the names of the variables are unique and start with a letter or underscore in C++ and consist of letters, underscores and digits.

2.2 Operators and Expressions

Operators in C++ are symbols or keywords used to perform specific operations on variables or values. There are different types of operators in C++, such as arithmetic, comparison, logical, and assignment operators.

The arithmetic operators in C++ allow you to perform basic arithmetic operations such as addition, subtraction, multiplication, and division. For example, one can perform arithmetic operations like this:

```
int a = 5, b = 2;

int c = a + b; // c contains 7

c = a - b; // c contains 3

c = a * b; // c contains 10

c = a / b; // c contains 2
```

Comparison operators in C++ make it possible to determine the relationship between two values by returning the truth or falsity of a statement about the relationship between the values. Examples of comparison operators are < (less than), > (greater than), == (equal), and != (not equal).

For example, one can perform comparison operations like this:

```
int a = 5, b = 2;
boolean c = (a > b); // c contains true
c = (a == b); // c contains false
c = (a != b); // c contains true
```

Logical operators in C++ allow you to combine multiple comparison expressions and return the truth or falsity of a statement about the relationships between the values. Examples of logical operators are && (and), || (or and ! (Not). For example, one can perform logical operations like this:

```
boolean a = true, b = false;
bool c = (a && b); // c contains false
c = (a || b); // c contains true
c = !a; // c contains false
```

Assignment operators in C++ allow you to assign a value to a variable. The simplest assignment operator is the equality (=) operator, which is used to assign a value to a variable. For example, you can do an assignment like this:

```
int a = 5;
a = 10; // a now contains the value 10
```

There are also compound assignment operators that combine arithmetic operations and assignments in one statement, such as +=, -=, *=, and /=. For example, you can do a combined assignment like this:

```
int a = 5;
a += 2; // a now contains the value 7
```

Expressions in C++ are any combination of variables, values, and operators that return a value. For example, an expression might look like this:

```
int a = 5, b = 2;

int c = a + b * 2; // c contains 9
```

There are also expressions that combine multiple operations and perform a more complex logical or arithmetic calculation, such as

```
int a = 5, b = 2, c = 3;

bool result = ( (a > b) || (b <= c) ) && (a != c); // result contains true
```

It is important to note that the order of execution of operations in an expression is determined by the rules of operator precedence. C++ follows the usual rules of mathematics, doing multiplication and division before addition and subtraction. Within an equal priority, the operations are executed from left to right. Parentheses can be used to change the execution order.

It is also important that the data types of the operands in an expression are compatible and that the effects of overflows or underflows in arithmetic operations are taken into account.

Overall, data types, variables, and operators are the basic building blocks for creating C++ programs, and it is important to understand them well and use them properly in order to write C++ code successfully. It is important to understand the different data types and their limitations to ensure that the variables are of the correct data type and that the operators are applied to compatible data types. It's also important to understand the rules of operator precedence and the implications of overflow or underflow in arithmetic operations to avoid code problems.

2.3 Control structures (if, for, while)

Control structures in C++ make it possible to control the flow of a program by executing or skipping certain statements under certain conditions. The most common control structures in C++ are if-else, for loops, and while loops.

The if-else structure in C++ allows certain statements to be executed if a certain condition is true, and other statements to be executed if the condition is false. For example, you can use an if-else structure like this:

```
int a = 5;
if (a > 10) {
// Execute statements if a is greater than 10
} else {
// Execute statements if a is less than or equal to 10
}
```

The for loop in C++ allows specific statements to be executed repeatedly as long as a specific condition is true. A for loop consists of an initialization part, a termination condition part and an increment part. For example, you can use a for loop like this:

```
for (int i = 0; i < 10; i++) {
// Execute statements repeatedly as long as i is less than 10
}
```

The while loop in C++ allows certain statements to be repeated until a certain condition is no longer met. A while loop consists of a condition that is checked before each iteration. For example, you can use a while loop like this:

```
int a = 5;
while (a < 10) {
// Execute statements repeatedly as long as a is less than 10
a++;
}
```

It is important to note that an infinite loop can result when the conditions in a loop are not properly set up or the conditions within the loop are not properly changed. It's also important that the statements within a loop or if-else structure are spelled and executed correctly to ensure that the program works as expected.

It's also important to note that in C++ there is the ability to nest multiple if-else structures or loops within each other, allowing for more complex decisions to be made and more complex flows to be controlled.

Overall, control structures such as if, for, and while are important tools in controlling the flow of a C++ program, and it is important to understand and use them well in order to write C++ code successfully.

2.4 Features

Functions in C++ allow you to create reusable blocks of code that perform specific tasks. A function consists of a function head, which specifies the function name, argument list, and return type, and a function body, which contains the code that the function executes.

The function header of a function in C++ begins with the keyword "void" or the return type, followed by the name of the function and a list of arguments in parentheses. For example, the header of a function might look like this:

```
int add(int a, int b)
```

In this example, "int" is the return type, "add" is the name of the function, and "(int a, int b)" is the argument list, indicating that the function expects two integer arguments.

The function body of a function contains the code that the function executes, and begins with an opening curly brace and ends with a closing curly brace.

For example, the body of a function might look like this:

```
int add(int a, int b) {

return a + b;

}
```

In this example, the only statement within the body of the function is a return expression that returns the sum of a and b.

Functions can be invoked by specifying the function name followed by the argument list in parentheses. For example, a function can be called like this:

```
int result = add(5, 3);
```

Functions can also be overloaded by defining multiple functions with the same name but different argument lists. This makes it possible to provide the same functionality for different types or numbers of arguments.

It's important to note that every function must have a return type, unless it's a void function that doesn't return a value. It is also important to ensure that the arguments passed to a function are compatible with the arguments in the function declaration and that the function is called correctly.

Overall, features in C++ enable code structuring and code reuse, which improves the readability and maintainability of C++ programs. It is important to understand and apply the concepts of function declaration, function call, and function overloading in order to write C++ code successfully.

2.5 Arrays and Pointers

Arrays in C++ allow multiple variables of the same data type to be treated as a single entity. An array consists of a sequence of variables of the same data type that are stored under a common name and can be addressed via an index.

An array can be declared in C++ as follows:

```
int myArray[5];
```

This example declares an array named "myArray" that provides space for 5 integer values. The elements of the array can be addressed via their index, which is in the range from 0 to 4. For example, one can assign and query an element of the array like this:

```
myArray[2] = 10;

int x = myArray[2]; // x now contains the value 10
```

Pointers in C++ make it possible to refer to memory addresses of variables, rather than the value of the variable itself. A pointer is denoted by the "*" keyword and can be used to refer to the memory address of a variable, or to refer to the value of a to change a specific memory address.

A pointer can be declared in C++ as follows:

```
int *p;
```

This example declares a pointer named "p" that points to an integer variable. To set a pointer to a specific variable, one can use the "&" operand, which returns the memory address of a variable. For example, one can use a pointer like this:

```
int x = 5;

p = ?

*p=10; // x now contains the value 10
```

It is important to note that pointers are an advanced concept in C++ and should be used with caution as they can easily lead to errors if used incorrectly. These include the importance of ensuring that a pointer points to a valid memory address and that the data type of the pointer matches the data type of the variable being referenced. It is also important to be careful not to access shared memory or invalid memory, which can lead to unexpected behavior or program crashes.

Overall, arrays and pointers in C++ are powerful tools that allow working with data in an efficient and flexible manner. It is important to have a good understanding of the concepts of arrays, indices, pointers, and memory addresses and to use them correctly in order to write C++ code successfully.

Object Orientation in C++

3.1 Concepts of object-oriented programming

Object-oriented programming (OOP) is a programming paradigm in which a program is modeled as a collection of interacting objects. Every object has a state and a behavior, and the interaction between objects occurs through the calls to methods.

Some of the most important concepts of object-oriented programming are:

Classes: A class is a template for an object and describes its state and behavior. A class contains variables (also known as fields or properties) and methods.

Objects: An object is an instance of a class and has its own state, represented by the values of the object's fields.

Methods: Methods are functions that are part of a class and can be called to control the object's behavior or to change its state.

Inheritance: Inheritance allows a new class to be defined based on an existing class, with the new class automatically inheriting the properties and methods of the original class.

Polymorphism: Polymorphism allows a method or operator to exhibit different behaviors depending on the context in which it is used.

Encapsulation: Encapsulation allows an object's properties and methods to be protected from unauthorized access by declaring them private or protected. This allows the implementation of a class to be changed without affecting classes derived from it.

Abstract classes and virtual methods: Abstract classes cannot be instantiated and serve as a base class for other classes. Virtual methods can be declared in abstract classes or in base classes, and can be overridden in derived classes to provide specific behavior.

Overall, object-oriented programming allows for more natural and logical problem modeling and increases code reusability and maintainability. It is important to have a good understanding of the concepts of classes, objects, methods, inheritance, polymorphism, encapsulation, abstract classes, and virtual methods and to apply them correctly in order to successfully write object-oriented C++ code.

3.2 Classes and Objects

In object-oriented programming, classes are the basis for creating objects. A class is a template for an object and describes its state and behavior. A class contains variables (also known as fields or properties) and methods.

A class is declared in C++ with the keyword "class". A class has a name and can contain fields and methods. For example, a Person class can be declared as follows:

```
class Person {

public:

string name;

int age;

void setName(string newName);

string getName();

};
```

In this example, the Person class contains name and age fields, which are of type string and int, and setName and getName methods.

An object is created by following the class name with the new operator. An object has its own state, represented by the values of the object's fields. For example, an object of class Person can be created as follows:

```
person person1;

person1.name = "Jane";

person1.age = 25;
```

This example creates an object "person1" of class "Person" and initializes its fields "name" and "age" with appropriate values.

Methods can be called to control an object's behavior or to change its state. For example, the setName method can be called to change a person's name:

```
person1.setName("John");
```

Methods can also access and modify or query the object's fields. For example, the getName method can be used to query a person's name:

```
string name = person1.getName();
```

It's important to note that each object has its own state and is independent of other objects of the same class, even if they call the same method or have the same property.

Overall, classes and objects in C++ are the foundation of object-oriented programming and allow problems to be modeled in a natural and logical way. It is important to have a good understanding of the concepts of classes, objects, fields, and methods and to apply them correctly in order to successfully write object-oriented C++ code.

3.3 Constructors and Destructors

In object-oriented programming, constructors and destructors are special methods that are called automatically when an object is created and destroyed, respectively.

A constructor is a method that is called automatically when an object is created. A constructor has the same name as the class and has no return type. There can be multiple constructors in a class that have different arguments. You will be overloaded.

The purpose of a constructor is to set an object's initial state and perform any necessary initializations. For example, a Person class may have a constructor that initializes the object's name and age fields:

```
class Person {

public:

string name;

int age;

Person(string newName, int newAge) {

name = newName;

age = newAge;

}

};
```

In this example, a new object of class "Person" would be created by following the class name with the new operator and calling the constructor with the appropriate arguments:

```
Person person1("John", 25);
```

A destructor is a method that is called automatically when an object is destroyed. A destructor has the same name as the class, starts with a tilde, and has no return type.

The purpose of a destructor is to free or clean up resources used by an object before the object is destroyed. For example, a Person class can have a destructor that prints a message when an object is destroyed:

```
class Person {
public:
string name;
int age;
Person(string newName, int newAge) {
name = newName;
age = newAge;
}
~Person() {
cout<< "Person " << name << " destroyed" << endl;
}
};
```

It's important to note that a destructor is only called when an object was created using the new operator. When an object is automatically created on the stack, the destructor is not called automatically, which can lead to memory leaks.

Overall, constructors and destructors in C++ are useful tools for setting an object's initial state and freeing or cleaning up resources when an object is destroyed. Understanding the concepts of constructors and destructors and using them correctly is important to successfully writing object-oriented C++ code.

3.4 Inheritance and Polymorphism

In object-oriented programming, inheritance and polymorphism are two key concepts that make it possible to increase code reusability and adaptability.

Inheritance allows a new class to be defined based on an existing class, with the new class automatically inheriting the properties and methods of the original class. A derived class is also called a subclass or child class, and the original class is called a base class or parent class.

In C++, inheritance is declared with the ":" keyword. For example, the Student class can be derived from the Person class, inheriting the properties and methods of the Person class:

```
class Person {

public:

string name;

int age;

void setName(string newName);

string getName();

};


class Student : public Person {

public:

string major;

void setMajor(string newMajor);

string getMajor();

};
```

In this example, the Student class inherits the name and age fields and the setName and getName methods from the Person class, in addition to defining its own fields and methods.

Polymorphism can be used. This allows methods and functions that refer to the base class or interface to be called with objects of derived classes without knowing the exact type of the object.

In C++, polymorphism is supported by virtual methods and abstract classes. A virtual method is a method that is declared in a base class and can be overridden in a derived class. An abstract class is a class that has no instances and serves as a base class for other classes.

For example, an abstract class "Shape" can be declared with a virtual method "calculateArea" that can be overridden in derived classes like "Rectangle" and "Circle":

```
class Shape {
public:
virtual double calculateArea() = 0;
};

class Rectangle : public Shape {
public:
double width;
double height;
double calculateArea() {
return width * height;
}
};

class Circle : public Shape {
public:
double radius;
double calculateArea() {
return 3.14 * radius * radius;
}
};
```

In this example, an array of Shape objects can be created and the calculateArea() method can be called on each element of the array without knowing the exact type of the object.

Overall, inheritance enables code reuse, and polymorphism increases code adaptability by allowing a base class or interface to be used as a type for a derived class. It is important to have a good understanding of the concepts of inheritance and polymorphism and to apply them correctly in order to successfully write object-oriented C++ code.

templates and generics

4.1 What are templates?

In C++, templates make it possible to write generic code that can be used for different data types without having to explicitly rewrite the code for each type. Templates are an important part of the C++ language and are commonly used to implement container classes, algorithms, and functions.

Templates are declared with the keyword "template" and the type parameters are given in curly brackets. For example, a function "swap" can be declared as a template that swaps two arguments of the same type:

```
template <typename T>

void swap(T& a, T& b) {

T temp = a;

a = b;

b = temp;

}
```

In this example, the swap function can be used for different data types by passing the type as an argument to the function, eg swap(x, y) for int values or swap(a, b) for string values.

Templates can also be used for classes, such as a "vector" template class that manages a dynamic list of elements of a given type:

```
template <typename T>

class vector {

private:

T* array;

int size;

public:

vector(int s) {

  size = s;

array = new T[size];

}

// other methods, e.g. push_back, pop_back, etc.

};
```

In this example, the vector class can be used for different data types by passing the type as an argument to the class when creating an object, e.g. vector<int> myIntVector(10) for int values or vector<string> myStringVector(20) for string values.

Templates also allow the use of multiple type parameters, e.g. when defining a class that takes both the value type and the comparison type as template arguments:

```
template <typename T, typename Compare = std::less<T>>
class MyClass {
// ...
};
```

It is important to note that templates are only resolved at compile time and that the compiler must create a specific version of the code for each use of the template. This can increase the size of the executable code and increase compile time.

Overall, templates in C++ make it possible to write generic code that can be used for different data types without having to explicitly rewrite the code for each type. It is important to have a good understanding of the concept of templates and to use them correctly in order to successfully write C++ code.

4.2 Use of templates in C++

Templates are a powerful tool in C++ and are used in many areas of programming. Some of the most common uses of templates are:

Container Classes: Templates are often used to implement container classes such as arrays, lists, stacks, queues, etc. that can hold data of the same type, e.g. the Standard Template Library (STL) in C++ contains classes like vector, list, stack and queue, all of which are templates.

Algorithms: Templates make it possible to implement algorithms such as sorting, searching, iterating, etc. for different data types without explicitly rewriting the code for each type. For example, the STL contains algorithms like sort, find, for_each, all of which are templates.

Functions: Templates make it possible to implement functions for different data types without explicitly rewriting the code for each type. Examples of this are functions for exchanging values, comparing values, or performing calculations on values.

4.3 Generics and Standard Template Library (STL)

Generics and the Standard Template Library (STL) are two concepts that are closely related and facilitate the use of templates in C++.

Generics are a concept that allow specific code to be used across a variety of data types without having to explicitly rewrite the code for each type. They make it possible to write code independently of the actual data types and only specify the actual type at runtime or compile time. Generics are supported in many modern programming languages, such as C#, Java, and C++.

The Standard Template Library (STL) is a collection of generic algorithms and container classes included with C++ that enable developers to perform commonly used functionality such as sorting, searching, storing, and manipulating data without having to write code themselves . The STL contains classes like vector, list, stack, queue, map, set, which are all templates and can be used for different data types.

Using generics and the STL allows developers to write code faster and more efficiently while increasing code reusability and customizability. It is important to note that the STL is a massive library and will take time to fully understand and master.

Error handling and debugging

5.1 Exception Handling

In C++ there is the ability to throw and catch exceptions to simplify error handling. An exception is thrown by the throw command and can be caught by a catch block.

An example of using exceptions:

```
#include <iostream>
using namespace std;
int divide(int a, int b) {
if (b == 0) {
throw "Division by zero not allowed";
}
return a / b;
}
int main() {
int x = 5, y = 0;
try {
cout << divide(x, y) << endl;
} catch (const char* message) {
cerr << message << endl;
}

return 0;
}
```

In this example, an exception is thrown if the divisor is zero and the error is caught and thrown by a catch block. It's also possible to use multiple catch blocks to catch different types of exceptions.

It is also possible to create your own exception classes, deriving from the standard std::exception class. This allows additional information to be transmitted along with the exception, such as an error message or error code.

It is important to note that exceptions can affect the execution of the program and should therefore be used with care. It's also important to catch any exception that may be thrown in a specific function or method to avoid unforeseen effects on program flow.

5.2 Assertions and Checks

Assertions and checks are mechanisms used to verify code integrity and detect potential errors early on.

An assertion is a statement that checks a condition and throws an exception if an error occurs. It is used to identify invalid or unexpected states in code. The assert macro in C++ checks the specified condition and, if an error occurs, throws an exception that contains the message "Assertion failed" by default. Example:

```
#include <cassert>

int main() {
int x = 5;
assert(x > 0); // OK, x > 0 is true
assert(x < 0); // throws an exception because x < 0 is false
return 0;
}
```

It is important to note that assertions are typically only used during development and should be disabled in the final version of code as they can affect execution.

Checks, on the other hand, are similar to assertions, but they don't necessarily throw an exception on error. They can be used to catch potential errors and provide more targeted error messages. Example:

```
void processData(int x) {
if (x <= 0) {
std::cerr << "Invalid value for x: " << x << std::endl;
return;
}
// Further processing of x
}
```

It is important to use both assertions and checks carefully to detect and avoid potential errors at an early stage. It's also important to consider the impact of assertions and checks on code performance and usability.

5.3 Debugging Tools and Techniques

Debugging is the process of finding and fixing bugs in a program. There are a variety of tools and techniques that developers can use to find and fix bugs in their code. Some of the most common tools and techniques are:

Debugger: A debugger is a tool that allows developers to track and examine the execution history of their program. It allows developers to step through the code, inspect variable values and set breakpoints to stop the code at certain points. C++ has built-in support for debuggers such as GDB, LLDB, and MSVC Debugger.

Print Statements: A simple but effective troubleshooting technique is to use print statements to monitor the state of the program. This allows developers to check the value of certain variables or the progress of the program without running the code step by step.

Logging: Another technique for debugging is the use of logging libraries, which allow developers to capture and store information about the program's execution. This allows developers to retrospectively investigate errors and identify the root causes of problems.

Profiling: Profiling is a technique that allows developers to measure and optimize the performance of their program. It allows developers to measure the execution time of certain parts of the code and to monitor the program's resource consumption (e.g. memory, CPU).

Memory Debugging: Another important tool is memory debugging, which allows developers to spot bugs in the program's memory usage. It allows developers to identify and fix memory leaks and avoid invalid memory accesses.

It is important to note that no single tool or technique can cover all errors and it is often necessary to combine more than one tool or techniques to successfully find and fix errors. It is also important that the tools and techniques used are well documented and organized to allow for efficient error diagnosis and resolution.

Another important concept in debugging is the use of test cases. By creating test cases that run the program under certain conditions, one can ensure that the program works as expected and errors are caught early. This makes troubleshooting and troubleshooting much easier.

In conclusion, debugging is an important part of the development process and requires the use of tools as well as the application of techniques and knowledge of the code. It also requires patience and perseverance to find and fix bugs successfully.

Advanced Topics

6.1 Multithreading and Parallel Programming

Multithreading and parallel programming refer to techniques that allow multiple tasks to be performed simultaneously. This can significantly improve the performance and execution time of programs.

Multithreading is the process of running multiple threads within a process at the same time. A thread is an independent unit of execution within a process that has its own stack space and program counter. C++ supports multithreading by using the C++11 and later threading library. Example:

```
#include <iostream>

#include <thread>


void print_message() {

std::cout << "Hello from thread" << std::endl;

}


int main() {

std::thread t(print_message);

t.join();

return 0;

}
```

This example creates a thread that runs the print_message function. The join() call waits for the thread to exit before exiting the main program.

Parallel programming refers to techniques that allow multiple processes or threads to run concurrently on multiple processors or cores. This makes it possible to improve the performance of programs using multiple processors or cores. C++ supports parallel programming through the use of libraries such as OpenMP and MPI.

It is important to note that parallel programming requires code to be carefully designed and written to avoid unwanted side effects and to ensure thread synchronization. It is also important to consider the impact of parallel programming on system performance and resource consumption.

6.2 Network Programming

Network programming refers to the development of applications that communicate with each other over a network. This makes it possible to connect applications on different devices or computers and exchange data.

In C++ there are several libraries and frameworks that make it possible to develop network applications. Some of the most common are:

Berkeley Sockets: This is a C-based library that allows developing network applications on the transport layer (TCP/UDP). It provides a lower level of abstraction than higher level libraries and allows it to be more flexible, but it also requires more overhead for managing connections and transferring data.

Boost.Asio: This is a C++-based library built on top of Berkeley Sockets and allows developing network applications to a higher level of abstraction. It makes it easier to manage connections and transfer data, but offers less flexibility than the lower-level Berkeley Sockets.

QT: is a cross-platform GUI framework that also provides network programming capabilities. It allows to easily develop network applications with a graphical user interface.

Network programming requires a basic understanding of network protocols and technologies, such as TCP/IP, HTTP, and FTP. It's also important to consider the security and privacy aspects of network applications to ensure data is transmitted securely and protected.

In conclusion, network programming is an important part of modern software development and allows applications on different devices and computers to connect and exchange data. It requires a basic understanding of network protocols and technologies, as well as consideration of security and privacy issues. There are various libraries and frameworks that allow developing network applications in C++, each with their own strengths and weaknesses. It is important to choose the right library or framework for the needs of the application and to ensure that the code is well structured and organized to enable efficient and reliable network communication.

6.3 Database Access

Database access refers to managing and accessing data in a database from a program. C++ offers several ways to access databases, including:

ODBC (Open Database Connectivity): ODBC is a platform-independent API that makes it possible to access different databases. It provides a lower level of abstraction than higher level libraries and allows it to be more flexible, but it also requires more overhead to manage connections and queries.

SQLite: SQLite is a lightweight database engine written in C and pluggable into C++. It makes it possible to have a local database in an application and does not require any additional servers or configurations.

ORM (Object-Relational Mapping) Libraries: ORM libraries make it possible to use database tables as C++ objects and to write queries in native language. Examples of ORM libraries in C++ are ORMapper, SQLObject and SOCI.

Library for specific DBMS (e.g. MySQL Connector/C++, Pqxx for PostgreSQL): There are also specific libraries for certain database management systems, which allow accessing these databases directly. They often offer higher performance and better integration with the particular DBMS, but they are limited to using that particular DBMS.

It is important to note that using databases requires code to be carefully designed and written to ensure the security and integrity of the data. It is also important to consider the impact of database accesses on system performance and resource consumption.

In conclusion, there are several ways to access databases in C++ including ODBC, SQLite, and ORM libraries. It is important to choose the right method for the needs of the application and to ensure that the code is well structured and organized to enable secure and efficient database access.

6.4 Interoperability with Other Languages

Interoperability refers to the ability of programs to communicate and exchange data with each other, regardless of the programming language used. C++ offers several ways to interoperate with other languages, including:

C interfaces: C++ is closely related to C and many C libraries can be included in C++ programs. This makes it possible to integrate the functionality of C libraries into C++ programs.

C++/CLI (Common Language Infrastructure): C++/CLI is an extension of C++ that allows C++ code to interoperate with .NET languages such as C# and Visual Basic. This makes it possible to integrate the functionality of .NET libraries into C++ programs.

SWIG (Simplified Wrapper and Interface Generator): SWIG is a tool that allows C/C++ code to interoperate with other languages such as Python, Perl, Ruby and Java.

JNI (Java Native Interface): JNI allows Java code to access C/C++ code and vice versa.

It's important to note that interoperability requires code to be carefully designed and written to ensure that data structures and calling conventions are correct. It is also important to consider the impact of interoperability on system performance and resource consumption.

Finally, there are several ways to interoperate with other languages in C++, including C interfaces, C++/CLI, SWIG, and JNI. It is important to choose the right method for the needs of the application and to ensure that the code is well structured and organized to enable successful interoperability.

imprint

Author: Michael Lappenbusch

E-mail: admin@perplex.click

Homepage: https://www.perplex.click

Release year: 2023