预备节 C++基本语法

什么是编程语言

1. **基本定义**：

编程语言是一种形式语言，用于通过特定的语法和语义与计算机进行交互。它允许程序员编写指令以控制计算机的行为，实现数据处理、计算和自动化任务。

2. **分类**：
   - **低级语言**：接近机器语言，能够直接控制硬件。
     - 示例：汇编语言。汇编语言通过助记符和指令直接与CPU交互，例如：

```
MOV AX, 1  ; 将1移动到寄存器AX
ADD AX, 2  ; AX加上2
```

   - **高级语言**：更接近人类自然语言，提供了更抽象的概念。
     - 示例：C++。高级语言代码更易于阅读和维护，例如：

```
int sum(int a, int b) {
    return a + b; // 返回两数之和
}
```

3. **从高级到低级到二进制的过程**：

高级语言代码经过编译器编译后，转换为汇编语言，然后再转换为机器语言（二进制代码），最后由CPU执行。例如：

```
// C++代码
int main() {
    int a = 5;
    int b = 10;
    return a + b;
}
```

- 编译为汇编语言：

```
MOV EAX, 5   ; 将5移动到EAX寄存器
ADD EAX, 10  ; EAX加上10
```

- 编译为二进制（示例）：

```
10111000 00000101 10111000 00001010
```

变量与数据类型

1. 变量的定义与声明：

变量是计算机内存中用于存储数据的命名位置。在C++中，变量需要在使用前声明，声明时需要指定数据类型。

```
int age; // 声明一个整数类型的变量age
age = 25; // 给变量赋值
```

2. 基本数据类型：

- **整数类型：**
  - `int`：标准整数类型，通常占用4字节（32位），可表示范围从-2,147,483,648到2,147,483,647。
  - `short`：短整数类型，通常占用2字节（16位），可表示范围从-32,768到32,767。
  - `long`：长整数类型，通常占用4字节（32位），在某些平台上占用8字节（64位）。
  - `long long`：更长的整数类型，通常占用8字节（64位）。

- **浮点类型**：
  - `float`：单精度浮点数，通常占用4字节，精度约为7位有效数字。
  - `double`：双精度浮点数，通常占用8字节，精度约为15位有效数字。
  - `long double`：扩展精度浮点数，通常占用12或16字节，具体大小依赖于编译器。

- **字符类型**：
  - `char`：用于表示单个字符，通常占用1字节。
  - `wchar_t`：宽字符类型，用于表示更广泛的字符集，如Unicode。

- **布尔类型**：
  - `bool`：用于表示真值，取值为 `true` 或 `false`。

```
int count = 10; // 整数类型
float price = 19.99f; // 单精度浮点数
char initial = 'A'; // 字符类型
bool isAvailable = true; // 布尔类型
```

3. **修饰符**：
   - **无符号类型**：`unsigned`，表示只能存储非负数。

     ```
     unsigned int u = 10; // 无符号整数
     ```

   - **常量与枚举类型**：
     - **常量**：使用 `const` 关键字声明的变量，其值在定义后不可修改。
     - **枚举类型**：定义一组命名的整型常量，提供更具可读性的代码。

     ```
     const int MAX_USERS = 100; // 常量
     enum Color { RED, GREEN, BLUE }; // 枚举类型
     ```

控制结构

1. **条件语句**：
   - **if语句**：根据条件执行代码块。

```
int score = 85;
if (score >= 60) {
    std::cout << "及格" << std::endl;
} else {
    std::cout << "不及格" << std::endl;
}
```

- **switch语句**：用于多条件选择。

```
int day = 3;
switch (day) {
    case 1: std::cout << "星期一"; break;
    case 2: std::cout << "星期二"; break;
    case 3: std::cout << "星期三"; break;
    default: std::cout << "未知"; break;
}
```

2. **循环语句**：
   - **for循环**：用于执行确定次数的循环。

```
for (int i = 0; i < 5; i++) {
    std::cout << "当前计数: " << i << std::endl;
}
```

- **while循环**：用于根据条件执行循环。

```
int i = 0;
while (i < 5) {
    std::cout << "当前计数: " << i << std::endl;
    i++;
}
```

- **do-while循环**：至少执行一次循环体后再判断条件。

```
int j = 0;
do {
    std::cout << "当前计数: " << j << std::endl;
    j++;
} while (j < 5);
```

3. **跳转语句**：
    - **break**：用于跳出循环或switch语句。
    - **continue**：用于跳过当前循环的剩余部分，继续下一次循环。
    - **return**：用于退出函数并返回值。

好的，以下是更详细的3.4部分关于函数与参数的内容：

函数与参数

1. **函数的定义与调用**：

函数是执行特定任务的代码块，它可以接收输入参数并返回结果。函数的使用可以提高代码的可重用性和可读性，使得复杂的程序结构更加清晰。

- **函数的定义：**

函数的定义包括返回类型、函数名称、参数列表（可选）以及函数体。返回类型指示函数返回值的类型，函数名称是标识符，参数列表则包含函数所需的输入变量。

```
int add(int a, int b) {
    return a + b; // 返回两数之和
}
```

- **函数的调用：**

调用函数时，可以将具体的值或变量作为参数传入。调用过程会产生一个新的执行上下文，函数执行完成后将控制权返回到调用者。

```
int main() {
    int sum = add(5, 10); // 调用add函数
    std::cout << "和: " << sum << std::endl; // 输出和
}
```

- **调用栈原理：**

当一个函数被调用时，程序会将当前执行的位置（返回地址）压入调用栈，并为函数的参数和局部变量分配内存。调用栈的结构使得函数可以在执行完后恢复到之前的状态。

例如，当调用 `add(5, 10)` 时，调用栈会进行以下操作：

1. 当前的返回地址被压入栈中。
2. 参数 `5` 和 `10` 被压入栈中。
3. 分配 `a` 和 `b` 的内存空间并初始化。
4. 执行函数体，计算和。
5. 返回结果并将控制权返回到调用位置，恢复之前的状态。

2. **参数传递方式：**

参数传递决定了在函数调用时，如何将参数的值传递给函数。C++支持两种主要的参数传递方式：值传递和引用传递。

- **值传递：**

在值传递中，函数接收参数的副本。这意味着在函数内部对参数的修改不会影响到外部变量。值传递适用于小型数据类型（如基本类型）时，性能较好，但对于大型对象来说，复制成本较高。

```
void modifyValue(int x) {
    x = 20; // 修改了副本
}

int main() {
    int a = 10;
    modifyValue(a); // 调用函数
    std::cout << "a的值: " << a << std::endl; // 输出10，值未改变
}
```

- **引用传递：**

引用传递允许函数直接访问参数的原始变量，而不是其副本。这意味着在函数内部对参数的修改将影响外部变量。引用传递在处理大型对象时特别有用，因为它避免了不必要的复制，提高了性能。

```
void modifyValue(int &x) {
    x = 20; // 修改了原始变量
}

int main() {
    int a = 10;
    modifyValue(a); // 调用函数
    std::cout << "a的值: " << a << std::endl; // 输出20，值已改变
}
```

3. **函数重载与默认参数：**

C++允许使用同一函数名称定义多个函数，只要它们的参数类型或数量不同，这称为函数重载。函数重载能够提高代码的灵活性和可读性。

- **函数重载**：

```
int add(int a, int b) {
    return a + b; // 整数相加
}

double add(double a, double b) {
    return a + b; // 浮点数相加
}
```

在调用时，编译器会根据传入参数的类型和数量选择合适的重载版本。

- **默认参数**：

在函数声明时，可以为参数指定默认值，这样在调用函数时可以省略某些参数。如果省略了，则使用默认值。

```
void printMessage(const std::string &message = "Hello, World!") {
    std::cout << message << std::endl; // 输出消息
}

int main() {
    printMessage(); // 输出默认消息
    printMessage("Hello, C++!"); // 输出自定义消息
}
```

4. **内联函数**：

内联函数使用 `inline` 关键字定义，编译器会在调用处将函数的代码替换到调用位置，而不是通过调用栈进行正常的函数调用。内联函数可以减少函数调用的开销，提高程序性能，但过多的内联函数可能会导致代码膨胀。

```cpp
inline int square(int x) {
    return x * x; // 返回x的平方
}

int main() {
    std::cout << "4的平方是: " << square(4) << std::endl; // 编译时替换为4 * 4
}
```

- **内联函数的使用场景**：

内联函数适合于小型、简单的函数，特别是那些被频繁调用的函数。这样可以避免函数调用带来的开销，提高运行效率。

C++ `class` 简单教程

`class` 是 C++ 中最重要的特性之一，用于实现面向对象编程（OOP）。它将数据和操作这些数据的方法（函数）封装在一起，形成一种数据类型，从而增强代码的可读性、可维护性和复用性。

什么是 `class`？

- `class` 是一种用户定义的数据类型，它包含属性（成员变量）和行为（成员函数/方法）。
- `class` 是 **"对象的蓝图"** 或模板，而对象（object） 是 `class` 的实例。

基本语法

```cpp
class ClassName {
public:
    // 构造函数
    ClassName();

    // 成员变量
    int attribute1;
    double attribute2;
```

```
9
10      // 成员函数（方法）
11      void memberFunction();
12  };
```

- `class` 关键字：用于定义一个新的类。
- `ClassName`：类的名称，通常使用大写字母开头的驼峰式命名。
- `public:`：访问修饰符，表示该部分的成员变量和函数可以在类的外部访问。

示例：创建一个简单的类

假设我们想创建一个类，表示一个简单的学生，包含他的名字和年龄，并提供一个方法来显示信息。

```
1  #include <iostream>
2  #include <string>
3
4  class Student {
5  public:
6      // 成员变量
7      std::string name;
8      int age;
9
10     // 成员函数
11     void displayInfo() {
12         std::cout << "Name: " << name << ", Age: " << age << std::endl;
13     }
14 };
```

- `name` 和 `age`：类的属性（成员变量），用于存储学生的信息。
- `displayInfo()`：类的行为（成员函数），用于显示学生的属性。

如何使用 `class`

你可以通过创建对象来使用 `class`，然后访问成员变量和调用成员函数。

```
int main() {
    // 创建 Student 类的对象
    Student student1;

    // 访问成员变量并赋值
    student1.name = "John";
    student1.age = 20;

    // 调用成员函数
    student1.displayInfo(); // 输出: Name: John, Age: 20

    return 0;
}
```

- **`student1`**：类 `Student` 的对象，包含了 `name` 和 `age` 属性。
- **赋值**：可以通过 `.` 操作符访问对象的成员变量并赋值。
- **调用函数**：可以通过 `.` 操作符调用对象的成员函数。

成员变量

定义：**成员变量**（也叫**数据成员**）是类中的变量，表示对象的**属性**或**状态**。它们存储在类的实例（对象）中，每个对象都有自己独立的成员变量。

访问控制：成员变量可以设置为 public（公有）、private（私有）或 protected（受保护），以控制外部对这些变量的访问权限。

生命周期：成员变量的生命周期与对象相同，当对象被创建时，成员变量被分配内存，当对象被销毁时，成员变量也被销毁。

成员函数

定义：**成员函数**（也叫**方法**）是类中的函数，用来表示对象的**行为**或**功能**。它们可以访问和操作成员变量，定义了类的对象可以执行哪些操作。

访问控制：成员函数同样可以设置为 public、private 或 protected，以控制是否允许从类的外部调用这些函数。

作用：成员函数主要用于对对象的成员变量进行操作，实现对象的行为。

访问修饰符

访问修饰符定义了类的成员的可访问性。C++ 中常用的访问修饰符有：

- `public`：类的成员在类的外部可访问。
- `private`：类的成员在类的外部不可访问，只能在类的内部使用。
- `protected`：与 `private` 类似，但允许子类访问。

默认情况下，`class` 的成员是 `private` 的：

```
class Example {
    int hiddenValue; // 默认 private
public:
    int visibleValue; // 显式 public
};
```

示例：

```
#include <iostream>

class Counter {
private:
    int count; // 私有成员变量

public:
    // 构造函数
    Counter() : count(0) {}

    // 成员函数
    void increment() {
        count++;
    }

    int getCount() const {
        return count;
```

```
    }
};

int main() {
    Counter c;
    c.increment();
    c.increment();
    std::cout << "Count: " << c.getCount() << std::endl; // 输出: Count: 2

    // c.count = 10; // 错误! count 是私有的，不能在类外直接访问

    return 0;
}
```

- `private`： `count` 是私有的，不能在类的外部直接访问。
- `increment()` 和 `getCount()`：是公有的，可以从类的外部调用，提供了一种访问和修改 `count` 的方法。

构造函数与析构函数

- **构造函数**：构造对象时自动调用，用于初始化对象。构造函数的名称与类名相同，没有返回值。
- **析构函数**：对象销毁时自动调用，用于清理资源，名称是 `~ClassName`，没有参数和返回值。

```
class Person {
public:
    std::string name;

    // 构造函数
    Person(std::string n) : name(n) {
        std::cout << "Person " << name << " is created." << std::endl;
    }

    // 析构函数
    ~Person() {
        std::cout << "Person " << name << " is destroyed." << std::endl;
```

```
    }
};

int main() {
    Person p1("Alice");
    // 输出：Person Alice is created.
    // 当 p1 超出作用域时，输出：Person Alice is destroyed.

    return 0;
}
```

- **构造函数**：在 `p1` 创建时，初始化 `name`，并打印出创建信息。
- **析构函数**：在 `p1` 超出作用域时，自动调用析构函数，释放资源。