第一章:C++的初步知识

从C到C++

结构化程序设计,指的是

> 1. 顺序结构
> 2. 迭代结构
> 3. 选择结构

C语言主要是$\blue{面向过程}$的

最简单的C++程序

函数调用时,参数传递的形式有

> 1. 简单变量
> 2. 指针变量
> 3. 数组名
> 4. 数组元素

C++对C的扩充

对于任意程序,可以没有输入,但不能没有输出

C++的输入输出

我们用cout来进行输出,可用<<来将其插入到输出流cout中,其运算结合方向为$\blue{自左向右}$

用const定义常变量

const定义常变量的方法:

> const<数据类型><变量名>=值

如:

> const double PI =3.14

常变量具有变量的属性，有数据类型，占用存储单元，有地址，
可以用指针指向它，只是值固定，不能改变。

函数原型声明

对于函数的声明.下面写法等价

```
int max(int x, int y );
int max(int ,int );
```

函数的重载

两个以上的函数，取$\blue{同一名字}$，只要使用不同类型的参数或参数个数不同，编译器就知道在什么情况下该调用哪个函数，这就叫函数重载。
例子如下:

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

// 函数重载示例：计算两个数的和
int add(int a, int b) {
    return a + b;
}

// 函数重载示例：计算三个数的和
int add(int a, int b, int c) {
    return a + b + c;
}

// 函数重载示例：计算两个浮点数的和
float add(float a, float b) {
    return a + b;
}

int main() {
    std::cout << "Sum of 2 and 3: " << add(2, 3) << std::endl; // 调用第一个函数
    std::cout << "Sum of 2, 3, and 4: " << add(2, 3, 4) << std::endl; // 调用第二个函数
    std::cout << "Sum of 2.5 and 3.5: " << add(2.5, 3.5) << std::endl; // 调用第三个函数

    return 0;
}
```

上述代码就对add进行了重载

函数模板

函数模板是建立一个通用函数，其函数类型和形参类型
不具体指定，用一个虚拟的类型来代表，这个通用函
数就称为函数模板。

下面就是一个函数模板的例子:

```
#include <iostream>

// 函数模板定义
template <typename T>
void swap(T& a, T& b) {
    T temp = a;
    a = b;
    b = temp;
}
```

```
int main() {
    int x = 5;
    int y = 10;
    std::cout << "Before swap: x = " << x << ", y = " << y << std::endl;

    swap(x, y); // 调用模板函数，传入两个int类型的变量
    std::cout << "After swap: x = " << x << ", y = " << y << std::endl;

    double fx = 5.5;
    double fy = 10.5;
    std::cout << "Before swap: fx = " << fx << ", fy = " << fy << std::endl;

    swap(fx, fy); // 调用模板函数，传入两个double类型的变量
    std::cout << "After swap: fx = " << fx << ", fy = " << fy << std::endl;

    char cx = 'A';
    char cy = 'B';
    std::cout << "Before swap: cx = " << cx << ", cy = " << cy << std::endl;

    swap(cx, cy); // 调用模板函数，传入两个char类型的变量
    std::cout << "After swap: cx = " << cx << ", cy = " << cy << std::endl;

    return 0;
}
```

> 函数模板比函数重载更方便，程序更简洁。但函数模板只适用于函数的参数个数相同而类型不同，且函数体相同的情况，如果参数个数不同，则不能用函数模板。

带有默认参数的函数

例子如下

```
#include <iostream>
#include <string>

class Person {
private:
    std::string name;
    int age;
    std::string country;

public:
    // 带有默认参数的构造函数
    Person(const std::string& name = "Unknown", int age = 0, const std::string&
country = "Unknown Country")
        : name(name), age(age), country(country) {}

    void display() const {
        std::cout << "Name: " << name << "\n"
                  << "Age: " << age << "\n"
                  << "Country: " << country << std::endl;
    }
};
```

```
int main() {
    // 使用默认参数创建对象
    Person person1;
    person1.display();

    // 只提供名字参数
    Person person2("Alice");
    person2.display();

    // 提供名字和年龄参数
    Person person3("Bob", 30);
    person3.display();

    // 提供所有参数
    Person person4("Charlie", 25, "USA");
    person4.display();

    return 0;
}
```

使用带有默认值参数的函数时要注意:

> 1. 如果函数的定义在函数调用之前，则应在函数定义中给出默认值。
> 如果函数的定义在函数调用之后，则在函数调用之前需要有函数原型声明，此时必须在函数声明时给出默认值，在函数定义时可以不给出默认值。
> 2. 一个函数不能既作为重载函数，又作为有默认参数的函数。因为当调用函数时，如果少写一个参数，系统无法判定是利用重载所数还是利用默认参数的函数:会出现二义性，系统无法执行。

变量的引用

```
int a = 3;
int &b = a;//声明b是一个整型变量的引用，它被初始化为a
```

我们称b为a的一个引用,其关键字为

> type &<引用名>=<变量名>

$\blue{注意}$:在定义引用时，马上就要对它进行初始化，不能定义完后再赋值,上述结果如果改为

```
int a = 3;
int &b ;
b = a;
```

就会报错

同时,由于b是a的引用,我们对b进行修改,a的值也会改变

> 关于引用的说明

> 1. 引用并不是一种独立的数据类型,在定义时必须说明其代表的是哪个对象

> 2. 引用与其所代表的变量共享同一内存单元(有同样的地址,占据相同的内存空间)

> 3. 对引用的初始化，可以是一个变量名，也可以用另一个引用
> 如:

```
int a = 3;
int &b = a;//声明b是整型变量a的别名
int &c = b;//声明c是整型引用b的别名
```

> 4. 引用在初始化后不能再被重新声明为另一变量的别名

```
int a=3,b=4:
int &c = a;//声明c是整型变量a的别名
c = &b;//企图使c改变成为整型变量b的别名，错误
int &c = b;//企图重新声明c为整型变量b的别名，错误
```

下述代码就是函数引用的一个例子:

```
#include <iostream>

// 交换函数，使用引用作为参数
void swap(int& a, int& b) {
    int temp = a;
    a = b;
    b = temp;
}

int main() {
    int x = 5;
    int y = 10;

    std::cout << "Before swap: x = " << x << ", y = " << y << std::endl;

    swap(x, y); // 调用swap函数，传入x和y的引用

    std::cout << "After swap: x = " << x << ", y = " << y << std::endl;

    return 0;
}
```

> 引用和指针的异同:

- 不必在swap函数中设立指针变量，指针变量要另外开辟内存单元，
  其内容是地址。而引用不是一个独立的变量，不单独占内存单元。
- 在main函数中调用swap函数时，实参不必在变量名前加&以表示地
  址。
- 使用指针变量时，为了表示指针变量所指向的变量，必须使用指
  针运算符，*而使用引用时，引用就代表该变量，不必使用指针运
  算符*。用引用能完成的工作，用指针也能完成。
  。但用引用比用指
  针直观、方便，容易理解

> 对于引用的进一步说明

1. 不能建立void类型的引用
2. 不能引用数组
3. 可以将变量的引用的地址赋给一个指针

```
int a = 3;
int &b = a;
int *p = &b; //指针变量p指向变量a的引用b，相当于指向a，合法
相当于 int *p = &a;
```

4. 可以建立指针变量的引用

```
int i = 5; //定义整型变量i，初值为5
int* p = & i;//定义指针变量p指向i
int* &pt = p;/pt是一个指向整型变量的指针变量的引用，
初始化为p
&pt表示pt是一个变量的引用，它代表一个int*类型的数据对
象(即指针变量)，如果输出*pt的值，就是*p的值5
```

5. 可以用const对引用加以限定,不允许改变引用的值
6. 可以对常量或用表达式对引用进行初始化,但此时必须用const作声明,如

```
#include <iostream>

int main() {
    int value = 10;
    const int& refValue = value; // 常量引用

    std::cout << "原始值: " << value << std::endl;
    std::cout << "引用值: " << refValue << std::endl;

    // 下面的语句将导致编译错误，因为refValue是常量引用
    // refValue = 20;
    // 但是可以这样使用
    /// value = 20;
    return 0;
}
```

内置函数

内联函数,其目的是在编译时将函数的代码直接插入到每个调用该函数的地方，以减少函数调用的开销。这通常用于小型函数，以提高程序的执行效率

例子如下:

```
#include <stdio.h>

// 内联函数的声明和定义
inline int max(int a, int b) {
    return (a > b) ? a : b;
}

int main() {
```

```
    int x = 5;
    int y = 10;

    // 调用内联函数
    int result = max(x, y);
    printf("The maximum of %d and %d is %d\n", x, y, result);

    return 0;
}
```

> 注意事项

- 内联函数的定义必须在调用之前：内联函数的定义必须在第一次被调用之前可见，这意味着内联函数通常在头文件中定义。
- 编译器的决策：即使函数被声明为内联，编译器也可能因为各种原因（如函数体太大）而决定不内联该函数。
- 适用于小型函数：内联函数通常用于小型、简单的函数，因为大型函数的内联可能会导致代码膨胀，从而增加内存使用量，反而降低性能。
- 不能包含复杂的控制流语句：内联函数中不应该包含复杂的控制流语句，如switch、while、for循环等，因为这些语句可能会使内联过程变得复杂，导致编译器放弃内联。

作用域运算符

作用域运算符（也称为范围解析运算符）用于指定名称在特定的作用域或命名空间中。它由两个冒号组成（::）。这个运算符可以用来访问类的成员、全局变量、函数或者命名空间中的元素。

我们以全局变量和局部变量为例

```
#include <iostream>

// 全局变量
int globalVar = 10;

int main() {

    int globalVar = 20;

    // 在main函数中，访问的是局部变量
    std::cout << "Inside main: " << globalVar << std::endl;

    // 在main函数中，访问的仍然是全局变量
    std::cout << "After function call in main: " << globalVar << std::endl;

    return 0;
}
```

结果将是:

```
Inside main: 20
After function call in main: 20
```

此时若在变量前面加::,则会输出全局变量

```cpp
#include <iostream>

// 全局变量
int globalVar = 10;

int main() {

    int globalVar = 20;

    // 在main函数中，访问的是局部变量
    std::cout << "Inside main: " << globalVar << std::endl;

    // 在main函数中，访问的仍然是全局变量
    std::cout << "After function call in main: " <<::globalVar << std::endl;

    return 0;
}
```

结果将是:

```
Inside main: 20
After function call in main: 10
```