课程目标

- 标准输入输出流
- `C++`字符串
- `C++`动态内存分配
- `C`和`C++`中的结构体
- `C++`中的数据抽象和封装
- `C++`类的定义

课程实验

- `C`语言实现熊和动物的对象并实现吃的动作

课堂引入

- `C++`中的数据抽象和封装是面向对象编程(OOP)的两个核心概念,它们在代码设计和实现中起着重要的作用。
- 通过封装,可以将类的实现细节隐藏起来,使外部无法直接访问对象的内部数据和实现细节。这种隐藏有助于降低系统的复杂性,减少了对外部用户的接口依赖。
- 数据抽象关注的是类对外部世界的抽象表示。通过将类的内部数据和方法进行抽象,只暴露必要的接口,隐藏了不必要的细节,提高了代码的可维护性和安全性。
- 封装将数据和相关操作捆绑在一起,形成一个类,使得代码更具可重用性。其他模块或类只需要使用类的公共接口,而不需要关心具体的实现。数据抽象允许我们定义通用的数据类型和操作,这使得这些抽象可以在不同的上下文中被重复使用。

授课进程

一 标准输入输出流

C++标准输入输出流指的是C++标准库中提供的用于输入输出的流对象,主要包括cin、cout、cerr和clog等。这些流对象都定义在`iostream`头文件中,可使用重定向技术将它们与文件或其他数据源相连,通过重载运算符`>>`和`<<`进行输入输出操作,实现数据的输入输出。例如,可以将cout重定向到文件中,将cin重定向到文件中,从而实现数据的持久化存储和读取。

C++标准输入输出流的成员包括以下几种:

> 1. 标准输入流对象：`cin`是`istream`类实例化的标准输入流对象,用于从标准输入设备(键盘)读取数据。可以通过重载运算符`>>`进行输入操作。
> 2. 标准输出流对象：`cout`是`ostream`类实例化的标准输出流对象,用于向标准输出设备(显示器)输出数据。可以通过重载运算符`<<`进行输出操作。
> 3. 格式化输入函数：`scanf`是`stdio`库中提供的格式化输入函数,用于从标准输入设备(键盘)读取指定格式的数据。
> 4. 格式化输出函数：`printf`是`stdio`库中提供的格式化输出函数,用于向标准输出设备(显示器)输出指定格式的数据。

5. 错误处理函数：`cerr`是`ostream`类实例化的错误流对象,用于输出错误信息。
6. 日志流对象：`clog`是`ostream`类实例化的日志流对象,用于输出日志信息。

除了以上几种成员,C++标准输入输出流还提供了其他一些成员函数和重载运算符,用于更灵活地控制输入输出的格式、缓冲区、精度等。总之,C++标准输入输出流提供了许多强大的功能和底层实现,可以满足大多数应用程序的输入输出需求,同时也提供了灵活性和可扩展性,方便开发人员进行定制和优化。

以下是一些C++标准输入输出流的常用操作：

对于标准输出流`cout`：

1. cout << "Hello, world!"：输出字符串"Hello, world!"。
2. cout << 3.14：输出浮点数3.14。
3. cout << endl：输出换行符并刷新缓冲区。
4. cout.put('a')：输出字符'a'。
5. cout.flush()：刷新缓冲区。

对于标准输入流`cin`：

1. int num; cin >> num：从标准输入读取一个整数并存储到num变量中。
2. double pi; cin >> pi >> pi >> pi：从标准输入读取三个浮点数并存储到pi变量中。
3. cin.get()：读取一个字符并返回。
4. cin.get(25)："读取"25个字符并返回。
5. cin.ignore()：忽略输入流中的一个字符或一行。
6. cin.fail()：检查输入流是否出错。
7. cin.clear()：重置输入流的状态标志。
8. cin.setf(ios::showbase)：设置输入流显示基数。

对于标准错误流`cerr`和标准日志流`clog`,它们的操作类似于`cout`和`cin`,例如：

1. cerr << "This is an error message." << endl：输出错误信息并换行。
2. clog << "This is a log message." << endl：输出日志信息并换行。

需要注意的是,cerr和clog在默认情况下是直接输出到屏幕的,如果需要将它们与文件或其他数据源相连,需要进行重定向操作。

下面是一些C++标准输入输出流的详细介绍和实例：

1. 输入流对象`cin`

`cin`是`istream`类实例化的输入流对象,用于从标准输入设备(键盘)读取数据。可以通过重载运算符`>>`进行输入操作。

[例6] 从标准输入设备读取一个整数

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

int main(int argc, char *argv[]) {
    int num;
    cin >> num; // 从标准输入设备读取一个整数,并将其存储在num变量中
    cout << "You entered: " << num << endl; // 输出读取的整数
    return 0;
}
```

2. 输出流对象`cout`

`cout`是`ostream`类实例化的输出流对象,用于向标准输出设备(显示器)输出数据。可以通过重载运算符<<进行输出操作。

[例7] 向标准输出设备输出一个字符串

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

int main(int argc, char *argv[]) {
    string name = "John";
    cout << "Hello, " << name << "!" << endl;   // 输出字符串"Hello, John!"
    return 0;
}
```

3. 格式化输入函数`scanf`

`scanf`是`stdio`库中提供的格式化输入函数,用于从标准输入设备(键盘)读取指定格式的数据。

[例8] 从标准输入设备读取一个浮点数

```cpp
#include <cstdio>
using namespace std;

int main(int argc, char *argv[]) {
    float num;
    scanf("%f", &num);  // 从标准输入设备读取一个浮点数,并将其存储在num变量中
    cout << "You entered: " << num << endl;     // 输出读取的浮点数
    return 0;
}
```

以下是一个简单的C++标准输入输出流的示例代码

[例9] 从用户读取两个整数,计算它们的和,并将结果输出到屏幕上

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

int main(int argc, char *argv[]) {
    int num1, num2, sum;
    // 从标准输入设备读取两个整数
    cout << "请输入两个整数,用空格隔开: " << endl;
    cin >> num1 >> num2;
    // 计算两个整数的和
```

```
    sum = num1 + num2;
    // 将结果输出到标准输出设备
    cout << "这两个整数的和为：" << sum << endl;

    return 0;
}
```

在这个示例代码中,我们使用了`<iostream>`头文件中的`cout`和`cin`对象。`cout`用于向标准输出设备(通常是显示器)输出数据,而`cin`则用于从标准输入设备(通常是键盘)读取数据。我们还使用了`using namespace std`语句,以便在代码中省略`std::`前缀。

[例10] 从用户读取一系列整数,计算它们的平均值,并将结果输出到屏幕上

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

int main() {
    vector<int> nums;
    int num;

    // 从标准输入设备读取一系列整数
    cout << "请输入一系列整数,用空格隔开：" << endl;

    while (cin >> num) {
        nums.push_back(num);    // 压栈
    }

    // 计算这些整数的平均值
    double sum = 0;
    for (int i = 0; i < nums.size(); i++) {
        sum += nums[i];
    }
    double average = sum / nums.size();

    // 将结果输出到标准输出设备
    cout << "这些整数的平均值为：" << average << endl;
    return 0;
}
```

在这个示例代码中,我们使用了`<vector>`头文件中的`vector`模板类来存储用户输入的一系列整数。我们还使用了循环来遍历这些整数,并计算它们的总和和平均值。最后,我们将结果输出到屏幕上。

二 C++字符串

在C++中提供了一个内建数据类型string,该数据类型可以替代C语言中char数组。需要使用string数据类型时则需要在程序中包含头文件`#include <string>`。string类型处理起来会比较方便,下面我们将介绍该类型主要处理字符串的功能。

函数名称	功能
构造函数	产生或复制字符串
析构函数	销毁字符串
=,assign	赋以新值
swap	交换两个字符串的内容
+=,append(),push_back()	添加字符
insert()	插入字符
erase()	删除字符
clear()	移除全部字符
resize()	改变字符数量
replace()	替换字符
+	串联字符串
= =,! =,<,<=,>,>=,compare()	比较字符串内容
size(),length()	返回字符数量
max_size()	返回字符的最大可能个数
empty()	判断字符串是否为空
capacity()	返回重新分配之前的字符容量
reserve()	保留内存以存储一定数量的字符
[],at()	存取单一字符
>>,getline()	从 stream 中读取某值
<<	将值写入 stream
copy()	将内容复制为一个 C - string
c_str()	将内容以 C - string 形式返回
data()	将内容以字符数组形式返回
substr()	返回子字符串
find()	搜寻某子字符串或字符
begin(),end()	提供正向迭代器支持
rbegin(),rend()	提供逆向迭代器支持
get_allocator()	返回配置器

1. 字符串的遍历

遍历一个字符串中的每个字符,并打印出来。

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

int main(int argc, char *argv[]) {
    string str = "Hello, world!";
    for (auto c : str) {
        cout << c << endl;
    }
    return 0;
}
```

2. 字符串的切割

使用`substr`函数切割字符串,并打印结果。

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

int main(int argc, char *argv[]) {
    string str = "Hello, world!";
    string part1 = str.substr(0, 5);    // 从索引0开始取5个字符
    string part2 = str.substr(7);       // 从索引7开始取到结尾
    cout << "Part1: " << part1 << endl;
    cout << "Part2: " << part2 << endl;
    return 0;
}
```

3. 字符串的比较

比较两个字符串是否相等,并打印结果。

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

int main(int argc, char *argv[]) {
    std::string str1 = "Hello, world!";
    std::string str2 = "Hello, world!";
    if (str1 == str2) {
        std::cout << "Strings are equal." << std::endl;
    }
    else {
        std::cout << "Strings are not equal." << std::endl;
    }
    return 0;
}
```

4. 字符串的替换

使用`replace`函数替换字符串中的某个子串,并打印结果。

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

int main(int argc, char *argv[]) {
    string str = "Hello, world!";
    size_t pos = str.find("world");      // 找到"world"的第一个出现位置
    if (pos == string::npos) {
        cout << "子串不存在" << endl;
        return 0;
    }
    str.replace(pos, 5, "C++");      // 将"world"替换为"C++",注意这里的5表示"world"的
长度
    cout << str << endl;             // 输出结果："Hello, C++!"
    return 0;
}
```

5. 字符串的插入

使用`insert`函数在字符串的特定位置插入另一个字符串,并打印结果。

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

int main(int argc, char *argv[]) {
    string str = "Hello, world!";
    string str_to_insert = "beautiful ";
    str.insert(7, str_to_insert);       // 在索引7的位置插入"beautiful "
    cout << str << endl;                // 输出结果："Hello, beautiful world!"
    return 0;
}
```

6. 字符串的删除

使用`erase`函数删除字符串中的某个子串,并打印结果。

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

int main(int argc, char *argv[]) {
    string str = "Hello, beautiful world!";
    str.erase(7, 6);    // 从索引7开始删除6个字符
    cout << str << endl;    // 输出结果："Hello, ful world!"
    return 0;
}
```

7. 字符串的大小写转换

使用`toupper`和`tolower`函数转换字符串中的字符大小写,并打印结果。

```
#include <iostream>
#include <string>
#include <cctype>   // 需要包含这个头文件来使用toupper和tolower函数
```

```
int main(int argc, char *argv[]) {
    string str = "Hello, world!";
    for (auto &c : str) {
        c = std::toupper(c);    // 将每个字符转换为大写
    }
    cout << str << std::endl;  // 输出结果："HELLO, WORLD!"
    for (char &c : str) {
        c = std::tolower(c);        // 将每个字符转换回小写
    }
    cout << str << std::endl;  // 输出结果："hello, world!"
    return 0;
}
```

三 C++动态内存分配

在C语言中，动态分配和释放内存的函数是malloc、realloc, calloc和free，而在C++语言中，new、new[]、delete和delete[]操作符通常会被用来动态地分配内存和释放内存。

在C++中,动态内存分配是一种在运行时根据需要分配内存的方法。这通常是在使用数组或对象等数据结构时非常有用的,因为这些数据结构的长度通常是未知的或可变的。C++提供了几个关键字来动态地分配内存:

1. `new`:这个操作符用于动态分配一个对象的内存。当使用`new`操作符时,它将在堆上分配内存,并返回指向新分配内存的指针。例如:

```
int *ptr = new int; // 分配一个整数的内存
```

2. `delete`: 这个操作符用于释放通过`new`操作符分配的内存。例如:

```
delete ptr; // 释放之前分配的整数的内存
ptr = nullptr;
```

3. `new[]`: 这个操作符用于动态分配一个对象的数组的内存。例如:

```
int *array = new int[10]; // 分配一个包含10个整数的数组的内存
```

4. `delete[]`: 这个操作符用于释放通过`new[]`操作符分配的内存。例如:

```
delete[] array; // 释放之前分配的整数的数组的内存
```

使用动态内存分配时,必须确保在使用完分配的内存后释放它,否则可能会导致内存泄漏。此外,也可以使用智能指针(`std::unique_ptr`和`std::shared_ptr`)来自动管理动态分配的内存。

C++动态内存分配的基本原理是在运行时根据需要动态地分配内存空间。这种方式相对于静态存储分配更加灵活,因为它允许在程序运行时根据实际输入数据确定所需内存空间的大小,而无法预先确定。

在C++中,动态内存分配通过使用`new`和`delete`关键字来实现。当程序运行到需要一个动态分配的变量或对象时,使用`new`关键字向系统申请取得堆中的一块所需大小的存储空间,用于存储该变量或对象。当不再需要使用这个变量或对象时,使用`delete`关键字释放这块内存空间。

动态内存分配必须在程序运行时进行,因为此时才能知道实际所需内存空间的大小。相对于静态存储分配,动态存储分配更加灵活,但也需要更加谨慎,因为它需要手动分配和释放内存,如果忘记释放分配的内存,就会导致内存泄漏。因此,在使用动态内存分配时,必须确保正确地使用`new`和`delete`关键字来分配和释放内存。

在动态内存分配的过程中可能会导致内存泄漏的问题

内存泄漏是在C++等编程语言中使用动态内存分配时,由于程序员的错误,导致已经分配的内存没有被正确地释放。为了避免内存泄漏,可以采取以下几种方法:

1. 使用智能指针:智能指针是一种能够自动管理内存的工具。例如,当一个`std::unique_ptr`或`std::shared_ptr`指向一个对象时,它会在适当的时候自动删除该对象。这可以防止程序员忘记释放内存,从而避免内存泄漏。

2. 编写清晰的释放逻辑:清晰地定义何时、如何释放内存,并确保在所有可能的执行路径中,这些逻辑都被正确执行。例如,如果你在一个函数中分配了内存,那么在函数结束时,你应该释放这部分内存。

3. 使用RAII原则:资源获取即初始化(Resource Acquisition Is Initialization),这是一种编程技巧,可以让你在创建对象时获取资源,并在对象销毁时释放资源。这可以防止忘记释放内存。

4. 遵循良好的编程习惯:例如,不要在代码中随意使用`new`和`delete`,尽量使用对象生命周期管理,避免一次性申请大量内存等。

new/delete和malloc/free的区别:

在C++中,`new`和`delete`是运算符,而`malloc`和`free`是函数。它们都可以用于动态内存分配和释放,但它们在用法和语义上有一些区别。

1. 构造和析构函数:`new`不仅分配内存,还会调用对象的构造函数来初始化对象。类似地,`delete`不仅会释放内存,还会调用对象的析构函数来清理对象。相比之下,`malloc`和`free`只负责内存的分配和释放,它们不会调用任何函数。

2. 类型安全:`new`和`delete`是类型安全的,它们会根据所分配对象的类型来自动计算所需内存的大小。而`malloc`和`free`需要程序员手动计算所需内存的大小。

3. 异常处理:当`new`无法分配所需的内存时,它会抛出一个异常(通常是`bad_alloc`)。这使得程序员可以捕获这个异常并采取适当的行动。相比之下,如果`malloc`无法分配所需的内存,它只是返回一个空指针。

4. 内存对齐:`new`和`delete`会自动处理对象的内存对齐问题,而`malloc`和`free`则不会。

总的来说,尽管`malloc`和`free`在某些情况下可能更加方便,但使用`new`和`delete`通常更加安全、类型安全和易于管理。在编写C++代码时,推荐优先使用`new`和`delete`。

四 C和C++中结构体

C语言中内存为空结构体分配大小为0,C++中为结构体和类分配大小为 1byte

> 在C语言中,空结构体的大小为0,而在C++语言中,空结构体的大小为1。这是由于C++标准规定:"no object shall have the same address in memory as any other variable",即任何不同的对象不能拥有相同的内存地址。因此,如果空类/结构体的大小为0,那么连续申请多个空类/结构体势必导致多个对象有相同的地址,所以C++中的空类/结构体大小为1。

```
struct People {};
sizeof(People) = 1;
```

✧ C语言中的结构体只涉及到数据结构,而不涉及到算法,也就是说在C中数据结构和算法是分离的。C语言中的结构体只能定义成员变量,不能定义成员函数。然而在C++中既可以定义成员变量又可以定义成员函数,C++中的结构体和类体现了数据结构和算法的结合。

✧ 不过虽然C语言的结构体中不能定义成员函数,但是可以定义函数指针,不过函数指针本质上不是函数而是指针,所以总的来说C语言中的结构体只是一个复杂数据类型,只能定义成员变量,不能定义成员函数,不能用于面向对象编程。

[例1]

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

int mySub(int x, int y) {
    return x - y;
}

typedef struct {
    int (*Add)(int, int);
    int (*Sub)(int, int);
} CTest;

int main() {
    CTest test;
    int retAdd = 0, retSub = 0;
    test.Add = myAdd;
    test.Sub = mySub;
    retAdd = test.Add(10, 5);
    retSub = test.Sub(15, 6);
    printf("retAdd = %d\n", retAdd);
    printf("retSub = %d\n", retSub);

    return 0;
}
```

```
retAdd = 15
retSub = 9
```

[例2]

```c
struct CTest {
    char ch;
    int num;
};

int main(int argc, char *argv[]) {
    CTest test;
    test.num = 1;
    printf("%d", test.num);
}
```

✧ 这样在C语言中是编译不过去的,原因提示未定义标识符CTest。总的来说就是在C语言中结构体变量定义的时候,若为struct结构体名 变量名定义的时候,struct不能省略。但是在C++之中则可以省略struct。

4.1 C++中的结构体与类的区别

4.1.1 类和结构体的相同

结构体中也可以包含函数；也可以定义public、private、protected数据成员；定义了结构体之后,可以用结构体名来创建对象。也就是说在C++当中,结构体中可以有成员变量,可以有成员函数,可以从别的类继承,也可以被别的类继承,可以有虚函数。

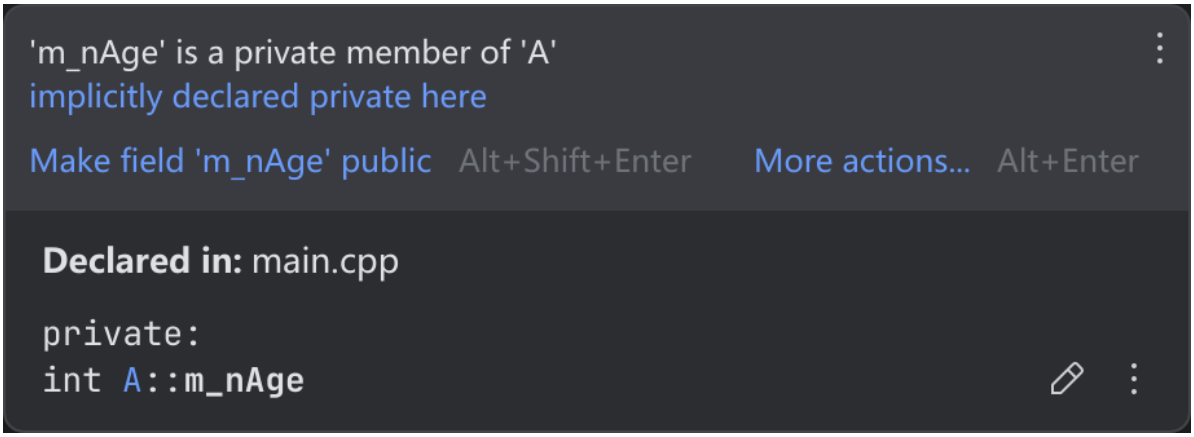
4.1.2 类和结构体的区别

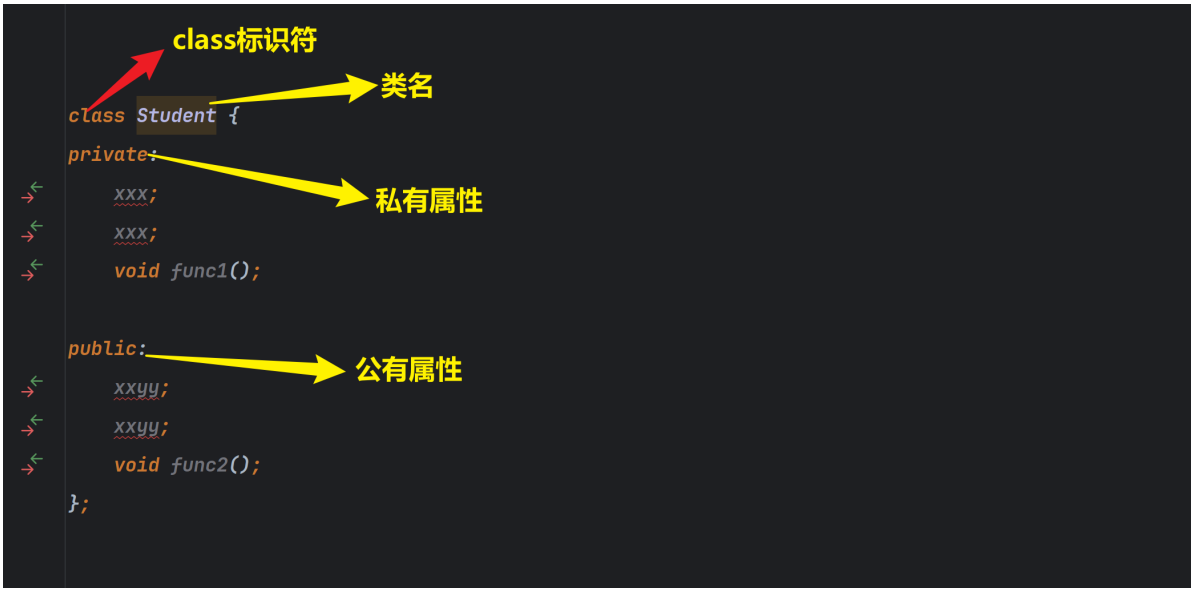
对于成员默认**访问权限**以及**继承方式**,class中默认的是private,而struct中则是public。class还可以用于表示模板类型,struct则不行。

```c
class A {
    int m_nAge;
};

struct B {
    int m_nAge;
};

void test() {
    A a;
    B b;
    a.m_nAge;       // ❎，无法访问私有成员
    b.m_nAge;       // 可正常外部访问
}
```





4.1.3 类/结构体的访问权限

类的访问权限有三种：

> public:向所有区域开放
>
> private:只有在类中才可以访问
>
> protected:子类访问

4.1.4 总结

`class`和`struct`的语法基本相同,从声明到使用都很相似,但是`struct`的约束要比`class`多,理论上`struct`能做到的`class`都能做到,但`class`能做到的`struct`却不一定做的到。

注意：如果一个类的成员函数在类内声明并实现，无论有没有inline关键字，这个函数都被声明为内联函数。

五 数据抽象和封装

数据抽象和封装是面向对象编程的两个重要概念。C++中的数据抽象和封装是通过类来实现的。类是对象的模板,对象是类的实例。

通过定义类和对象,可以将实现细节隐藏起来,只留下公共接口,供程序员使用。每个对象都有自己的数据成员和成员函数。通过将数据成员设置为私有(private),而将公共(public)成员函数设置为访问器(accessor)和修改器(mutator),可以实现对数据的封装。同时,通过定义公共接口,可以实现对数据的抽象。因此,类是实现数据抽象和封装的重要工具。

5.1 抽象

数据抽象是一种将复杂问题简化的方法,它通过隐藏不必要的细节,只暴露必要的接口,使程序员更容易地理解和使用数据和功能。

抽象是通过特定的实例抽取共同特征以后形成概念的过程。一个对象是现实世界中一个实体的抽象,一个类是一组对象的抽象。

5.2 封装

封装是将相关的概念组成一个单元,然后通过一个名称来引用它。面向对象封装是将数据和基于数据的操作封装成一个整体对象,对数据的访问或修改只能通过对象对外提供的接口进行。**在C++中,封装是通过将数据成员设置为私有(private),而将公共(public)成员函数设置为访问器(accessor)和修改器(mutator)来实现的。**这种做法可以防止直接访问或修改数据成员,只能通过公共成员函数来间接操作数据成员。这有助于保护数据的完整性和安全性。

> 1. 把变量(属性)和函数(操作)合成一个整体,封装在一个类中
> 2. 对变量和函数进行访问控制

下面是一个简单的例子,展示了C++中的数据抽象和封装:

[例3]

```
class Circle {
private:
    double m_dRadius = 3.5; // 圆的半径，私有成员
public:
    // 获取半径的函数,公有成员函数
    double getRadius() {
        return m_dRadius;
    }
    // 设置半径的函数,公有成员函数
    void setRadius(double radius) {
        m_dRadius = radius;
    }
    // 计算面积的函数,公有成员函数
    double calcArea() {
        return PI * m_dRadius * m_dRadius;
    }
};

int main() {
    Circle circle;
    cout << "圆的半径为:" << circle.getRadius() << endl;
    circle.setRadius(4.5);
    cout << "圆设置后的半径为:" << circle.getRadius() << endl;
    cout << "圆的面积为:" << circle.calcArea() << endl;

    return 0;
}
```

在这个例子中,Circle类定义了一个私有成员变量radius,表示圆的半径。通过构造函数可以创建Circle对象并设置半径,通过getRadius和setRadius函数可以获取和设置半径的值,通过getArea函数可以计算圆的面积。这些函数都是公共成员函数,可以通过对象来调用。这样就实现了对半径的封装和对圆的面积的计算,同时隐藏了半径的实现细节。

再比如我们要表示熊这个对象,在c语言中,我们可以这么表示:

```
typedef struct {
    char m_strName[64];
    int m_nAge;
    int m_nType;        // 动物种类
} Animal;

typedef struct {
    char m_strName[64];
    int m_nAge;
} Bear;

void BearEat(Bear *bear) {
    printf("%s在吃熊的饭!\n", bear->m_strName);
}

void AnimalEat(Animal *animal) {
    printf("%s在吃动物的饭!\n", animal->m_strName);
}

int main(int argc, char *argv[]) {
    Bear bear;
    strcpy(bear.m_strName, "维尼");
    bear.m_nAge = 30;
    AnimalEat(&bear);   // error: cannot convert 'Bear*' to 'Animal*'
    return 0;
}
```

定义一个结构体用来表示一个对象所包含的属性,函数用来表示一个对象所具有的行为,这样我们就表示出来一个事物,在c语言中,行为和属性是分开的,也就是说吃饭这个属性不属于某类对象,而属于所有的共同的数据,所以不单单是BearEat可以调用Bear数据,AnimalEat也可以调用Bear数据,那么万一调用错误,将会导致问题发生。

从这个案例我们应该可以体会到,属性和行为应该放在一起,一起表示一个具有属性和行为的对象。

假如某对象的某项属性不想被外界获知,比如说女孩的年龄不想被其他人知道,那么年龄这条属性应该作为只有女孩自己知道的属性;

或者女孩的某些行为不想让外界知道,只需要自己知道就可以。那么这种情况下,封装应该再提供一种机制能够给属性和行为的访问权限控制住。所以说封装特性包含两个方面,一个是属性和变量合成一个整体,一个是给属性和函数增加访问权限。

六 C++类的定义

6.1 类的声明

> class Screen;

✧ 在声明之后,定义之前,只知道Screen是一个类名,但不知道包含哪些成员。**只能以有限方式使用它**,不能定义该类型的对象,只能用于定义指向该类型的指针或引用,声明(不是定义)使用该类型作为形参类型或返回类型的函数。

```
class Screen;
void Test1(Screen &a) {};        // OK
void Test1(Screen *a) {};        // OK
void Test1(Screen a) {};         // 错误 创建对象会为这个类分配内存空间。
```

6.2 类名

遵循一般的命名规则;

字母,数字和下划线组合,不要以数字开头。

> **注意**：一般类名以大写字母开头

6.3 类定义

C++类是面向对象编程的基本概念,它是一种用户自定义的数据类型。类可以包含数据成员(变量)和方法(函数),可以用来创建具有相同属性和行为的对象。在创建类的对象之前,必须完整的定义该类,而不只是声明类。所以,**类不能具有自身类型的数据成员**,但可以包含指向本类的指针或引用。

```
class LinkScreen {
public:
    LinkScreen window;      // error
    LinkScreen *next;       // ✅
    LinkScreen *prev;
};
```

6.4 类作用域

✧ 每个类都定义了自己的作用域和唯一的类型。

✧ 类的作用域包括:类的内部(花括号之内), 定义在类外部的成员函数的参数表(小括号之内)和函数体(花括号之内)。

```
class Screen {
    // 类的内部
    char get(index r, index c) const;
    ...
};
```

```
// 类的外部
char Screen::get(index r, index c) const {
    index row = r * width;    // compute the row location
    // offset by c to fetch specified character
    return contents[row + c];
}
```

注意：成员函数的返回类型不一定在类作用域中。可通过类名::来判断是否是类的作用域,::之前不属于类的作用域,::之后属于类的作用域。例如: Screen:: 之前的返回类型就不在类的作用域,Screen:: 之后的函数名开始到函数体都是类的作用域。

6.5 使用类型别名来简化类

✧ 除了定义数据和函数成员之外,类还可以定义自己的局部类型名字。

✧ 使用类型别名能让复杂的类型名字变得简单明了、易于理解和使用,还有助于程序员清楚地知道使用该类型的真实目的。

```
class Screen {
public:
    typedef std::string::size_type index;
    index get_cursor() const;
    int get_cursor() const;
};

// 注意：index前面Screen不能少
Screen::index Screen::get_cursor() const {
    return cursor;
}
```

该函数的返回类型是index,这是在 Screen 类内部定义的一个类型名。在类作用域之外使用,必须用完全限定的类型名Screen::index 来指定所需要的 index 是在类 Screen 中定义的名字。

课堂小结

- 标准输入输出流的使用
- C++中内建字符串的使用
- C++动态内存分配以及与C语言中动态内存分配的不同
- C++对C语言结构体的增强
- 数据的抽象和封装
- C++类的特点

随堂作业

- 定义一个简单的C++类,包含成员变量、成员函数,以及构造函数和析构函数。要求实例化这个类并演示如何使用。