单选题

1. 以下哪个选项不全是 C++中的关键字？

- **A. integer, float, double**
- **B. auto, inline, switch**
- **C. class, union, struct**
- **D. virtual, static, namespace**

解析：

- **A. integer, float, double**
  - "integer" 不是 C++的关键字，正确的关键字是 "int"。
  - "float" 和 "double" 是 C++中表示浮点类型的关键字。
  - 因为 "integer" 不是关键字，所以此选项 **不全是** C++关键字。
- **B. auto, inline, switch**
  - "auto" 是 C++中的关键字，用于自动类型推导（C++11 起功能增强）。
  - "inline" 是关键字，用于建议函数内联优化。
  - "switch" 是关键字，用于条件分支语句。
  - 三个都是 C++关键字，此选项正确。
- **C. class, union, struct**
  - "class" 是定义类的关键字。
  - "union" 是定义共用体的关键字。
  - "struct" 是定义结构体的关键字。
  - 三个都是 C++关键字，此选项正确。
- **D. virtual, static, namespace**
  - "virtual" 是关键字，用于虚函数和虚继承。
  - "static" 是关键字，用于定义静态变量或函数。
  - "namespace" 是关键字，用于定义命名空间。
  - 三个都是 C++关键字，此选项正确。

答案：A

原因： A 选项中的 "integer" 不是 C++关键字，而其他选项中的词均为关键字。

2. 下面叙述不正确的是?

- **A. 派生类一般都用公有派生**
- **B. 对基类成员的访问必须是无二义性的**
- **C. 赋值兼容规则也适用于多重继承的组合**
- **D. 基类的公有成员在派生类中仍然是公有的**

解析:

- **A. 派生类一般都用公有派生**
  - 公有派生是 C++继承中最常用的方式，保持基类的公有成员和保护成员的访问权限。
  - 此叙述正确，符合常规使用场景。
- **B. 对基类成员的访问必须是无二义性的**
  - 在继承中，尤其是多重继承时，若多个基类有同名成员，必须通过作用域解析（如 `Base1::member` ）避免二义性。
  - 此叙述正确，是 C++的基本规则。
- **C. 赋值兼容规则也适用于多重继承的组合**
  - 赋值兼容规则指派生类对象可以赋值给基类对象或指针。多重继承中，只要无二义性，这一规则仍然适用。
  - 此叙述正确。
- **D. 基类的公有成员在派生类中仍然是公有的**
  - 这取决于继承方式:
    - **公有派生**：基类的公有成员在派生类中仍为公有。
    - **私有派生**：基类的公有成员在派生类中变为私有。
    - **保护派生**：基类的公有成员在派生类中变为保护。
  - 题目未指定继承方式，但叙述假设所有情况下都成立，显然不正确。例如，在私有派生中，基类的公有成员不再是公有的。

答案：D

原因： D 选项的叙述在私有或保护派生时不成立，因此不正确。

3. 下列关于 new 运算符的描述中，哪个是错误的?

- **A. 它可以用来动态创建对象和对象数组**
- **B. 使用 new 创建的 int 型数组 p[10]，可以用 delete p;来释放空间**
- **C. 使用它创建对象时要调用构造函数**
- **D. new 创建的动态变量的空间是在栈区中分配的**

解析：

- **A. 它可以用来动态创建对象和对象数组**
  - `new` 可以分配单一对象（如 `new int`）或数组（如 `new int[10]`），正确。
- **B. 使用 new 创建的 int 型数组 p[10]，可以用 delete p;来释放空间**
  - 用 `new int[10]` 创建数组时，必须用 `delete[] p;` 释放空间。
  - 用 `delete p;` 只适用于单一对象（如 `new int`），对数组会导致未定义行为。
  - 此叙述错误。
- **C. 使用它创建对象时要调用构造函数**
  - 当用 `new` 创建类对象时，会自动调用构造函数（如 `new MyClass()`），正确。
- **D. new 创建的动态变量的空间是在栈区中分配的**
  - `new` 分配内存是在 **堆区**（heap），而非栈区（stack）。
  - 栈区由编译器自动管理（如局部变量），此叙述错误。

答案：D

原因： D 选项错误，`new` 分配内存是在堆区，而非栈区。注意：B 选项虽有误，但题目问“错误的”，D 更明显且常考。

4. 下面代码编译时不会报错的语句是？

```
const int *p;
int *q;
const int x = 0;
int y;
```

- **(1) p = &x; *p = 1;**
- **(2) q = &x;**
- **(3) p = &y;**
- **(4) p = &x; q = p;**

解析：

- **(1) p = &x; *p = 1;**
  - `p` 是 `const int*`，可以指向 `const int x` 的地址，`p = &x` 正确。
  - 但 `*p = 1` 试图修改 `x` 的值，而 `x` 是 `const`，不可修改，编译错误。
- **(2) q = &x;**
  - `q` 是 `int*`，而 `&x` 是 `const int*` 类型。

- 将 `const int*` 赋值给 `int*` 会丢弃 `const` 限定符，编译错误。

- **(3) p = &y;**
  - `p` 是 `const int*`，指向不可修改的 `int`。
  - `y` 是普通 `int`，`&y` 是 `int*`，可以赋值给 `const int*`（允许非 const 到 const 的转换），正确。
  - 未尝试修改 `*p`，无错误。
- **(4) p = &x; q = p;**
  - `p = &x` 正确，`p` 是 `const int*`，`&x` 类型匹配。
  - `q = p` 将 `const int*` 赋值给 `int*`，丢弃 `const` 限定符，编译错误。

答案：C (3)

原因： 只有 `(3) p = &y;` 编译无错。

5. 下列有关重载函数的说法中正确的是？

- **A. 重载函数必须具有不同的返回值类型**
- **B. 重载函数参数个数必须相同**
- **C. 重载函数必须有不同的形参列表**
- **D. 重载函数名可以不同**

解析：

- **A. 重载函数必须具有不同的返回值类型**
  - 函数重载只依赖形参列表，与返回值类型无关。
  - 错误。
- **B. 重载函数参数个数必须相同**
  - 重载要求形参列表不同，参数个数可以不同。
  - 错误。
- **C. 重载函数必须有不同的形参列表**
  - 重载的定义：函数名相同，形参列表（参数个数、类型或顺序）不同。
  - 正确。
- **D. 重载函数名可以不同**
  - 重载要求函数名相同，否则是不同函数而非重载。
  - 错误。

答案：C

原因： 重载函数的核心是形参列表不同。

6. 创建 D 类对象 d 时，所调用的构造函数及它们的执行顺序是?

```
class A {
    int x;
public:
    A(int i) { x = i; }
};
class B : virtual public A {
    int y;
public:
    B(int i) : A(1) { y = i; }
};
class C : virtual public A {
    int z;
public:
    C(int i) : A(2) { z = i; }
};
class D : public B, public C {
    int m;
public:
    D(int i, int j, int k) : C(j), B(i), A(3) { m = k; }
};
D d(1, 2, 3);
```

- **A. D(), B(), C(), A()**
- **B. D(), C(), B(), A()**
- **C. A(), C(), B(), D()**
- **D. A(), B(), C(), D()**

解析：

- **虚继承特点**：虚基类 `A` 的构造函数由最底层的派生类 `D` 直接调用，优先于其他基类。
- **执行顺序**：
  i. `D` 的构造函数显式调用 `A(3)`，先执行 `A()`。
  ii. 然后调用直接基类 `B` 和 `C` 的构造函数，按声明顺序（`B` 在前）。

    - `B(1)` 调用，但 `A(1)` 被 `D` 的 `A(3)` 覆盖。
    - `C(2)` 调用，同样 `A(2)` 被覆盖。
  iii. 最后执行 `D` 自身的构造函数体。
- **顺序**：`A()，B()，C()，D()`。

答案：D

原因： 虚继承中，虚基类最先构造，之后按基类声明顺序，最后是自身。

7. 语句 myclass obj=10; 会发生什么?

假设 `myclass` 定义了拷贝构造函数、整型参数构造函数和重载赋值运算符。

- **A. 调用拷贝构造函数**
- **B. 调用整型参数的构造函数**
- **C. 调用赋值运算符**
- **D. 引起编译错误**

解析：

- `myclass obj = 10;` 是初始化，等价于 `myclass obj(10);`。
- 若类有 `myclass(int)` 构造函数，则直接调用此构造函数。
- 拷贝构造函数和赋值运算符用于对象之间的操作，此处无涉及。

答案：B

原因： 初始化调用整型参数构造函数。

8. 关于友元，下列说法错误的是?

- **A. 如果类 A 是类 B 的友元，类 B 是类 C 的友元，那么类 A 也是类 C 的友元**
- **B. 如果函数 fun()被说明为类 A 的友元，那么在 fun()中可以访问类 A 的私有成员**
- **C. 友元关系不能被继承**
- **D. 友元是数据保护和数据访问效率之间的一种折衷方案**

解析：

- **A.** 友元关系无传递性，错误。

- **B.** 友元函数可访问私有成员，正确。
- **C.** 友元不能继承，正确。
- **D.** 友元平衡保护与效率，正确。

答案：A

原因： 友元无传递性。

9. 关于 this 指针使用说法正确的是?

- **A. 保证每个对象拥有自己的数据成员，但共享处理这些数据的代码**
- **B. 保证基类私有成员在子类中可以被访问**
- **C. 保证基类保护成员在子类中可以被访问**
- **D. 保证基类公有成员在子类中可以被访问**

解析：

- **A.** `this` 指向当前对象，确保数据独立，代码共享，正确。
- **B, C, D.** `this` 不影响访问权限，错误。

答案：A

原因： `this` 的作用是区分对象实例。

10. 执行 A x(4,5); 后，x.a 和 x.b 的值是?

```
A(int aa, int bb) { a = aa--; b = a * bb; }
```

- **A. 3 和 15**
- **B. 5 和 4**
- **C. 4 和 20**
- **D. 20 和 5**

解析：

- `aa = 4, bb = 5`
- `a = aa--`：先赋值 `a = 4`，然后 `aa = 3`。

- `b = a * bb = 4 * 5 = 20` 。
- 结果： `x.a = 4, x.b = 20` 。

答案：C

原因： 计算符合后置递减逻辑。

11. 下列 func 调用中不正确的是?

```
template<class T> T func(T x, T y) { return x*x + y*y; }
```

- **A. func(3.5, 4.5);**
- **B. func(3, 5);**
- **C. func<double>(3.5, 4.5);**
- **D. func<int>(3.5, 4.5);**

解析：

- **A.** 推导 `T` 为 `double` ，正确。
- **B.** 推导 `T` 为 `int` ，正确。
- **C.** 显式指定 `double` ，参数匹配，正确。
- **D.** 显式指定 `int` ，但参数是 `double` ，类型不匹配，错误。

答案：D

原因： 参数类型与模板类型冲突。

12. 下列有关输入输出（I/O）的说法中正确的是?

- **A. 在 C++中，输入输出是语言定义的成分**
- **B. 在 C++中，输入输出操作不是一种基于字节流的操作**
- **C. 对自定义的类重载插入操作符“<<”和抽取操作符“>>”时不能作为类的成员函数来重载**
- **D. 文件输入操作是指把计算机内存中的数据写入到外存中的文件里**

解析：

- **A.** I/O 通过标准库实现，非语言核心，错误。

- **B.** I/O 基于字节流，错误。
- **C.** `<<` 和 `>>` 需作为全局或友元函数重载，正确。
- **D.** 文件输入是从文件读取到内存，错误。

答案：C

原因： 重载流运算符的限制。

13. 以下哪些语句是没有问题的？

```
class A { public: void f() {} };
class B : public A { public: void g() {} };
A a; B b;
```

- **(1) a.g();**
- **(2) A *p = &b;**
- **(3) b = a;**
- **(4) void func1(A *p); func1(&b);**

解析：

- **(1)** `a` 是 `A` 对象，无 `g()`，错误。
- **(2)** 派生类指针可赋值给基类指针，正确。
- **(3)** 基类不可直接赋值给派生类，错误。
- **(4)** 派生类地址可传给基类指针参数，正确。

答案：C (2, 4)

原因： 继承中的类型兼容性。

14. 类 MyDERIVED 中保护成员个数是?

```
class MyBASE {
    int k;
public:
    void set(int n) { k = n; }
protected:
    int get() const { return k; }
};
class MyDERIVED : protected MyBASE {
    int j;
public:
    void set(int m, int n) { MyBASE::set(m); j = n; }
    int get() const { return MyBASE::get() + j; }
};
```

- **A. 4**
- **B. 3**
- **C. 2**
- **D. 1**

答案：D

原因： 保护成员数量为 1。

15. 运算符的最佳原型是?

实现 `A a, b, c; a = b = c;` 且高效。

- **A. A A::operator=(A a);**
- **B. A A::operator=(const A&);**
- **C. A& A::operator=(A a);**
- **D. A& A::operator=(const A& a);**

解析：

- 支持连等：返回 `A&` 。
- 高效：参数用 `const A&` 避免拷贝。
- **D** 满足两者。

答案：D

原因： 返回引用且参数高效。

程序分析题

16. 填写程序，利用引用类型实现交换两个 int*型指针变量的值（4分）

题目代码：

```
#include <iostream>
using namespace std;
void swap(___(1)___)  // 交换两个int*型指针变量的值
{
    int t;
    _____(2)_____
}
int main()
{
    int a=0, b=1;
    int *p=&a, *q=&b;
    cout << *p << ',' << *q << endl;  // 输出 0,1
    swap(p, q);
    cout << *p << ',' << *q << endl;  // 输出：1,0
    return 0;
}
```

解析：

- **目标**：交换两个 `int*` 指针的值，使 `p` 指向 `b` ， `q` 指向 `a` 。
- **方法**：使用引用传递指针，在函数内直接修改 `p` 和 `q` 的指向。
- **填空**：
  - **(1)**：函数参数需为 `int*` 的引用，写作 `int*& p1, int*& p2` 。
  - **(2)**：交换逻辑需用临时指针保存值，正确写法是 `int* t = p1; p1 = p2; p2 = t;` 。
- **输出验证**：交换后 `*p` 为 1， `*q` 为 0，符合要求。

答案：

- **(1)**： `int*& p1, int*& p2`

- **(2)**： `int* t = p1; p1 = p2; p2 = t;`

说明：

- 使用引用（ `&` ）确保函数修改的是实参指针本身。
- `t` 是 `int*` 类型，用于临时存储指针地址。

17. 填写程序，实现二维数组的求和操作（4 分）

题目代码：

```
int sum(____(1)____, int)  // 2分
{
    int s=0;
    for (int i=0; i<num; i++) s += x[i];
    return s;
}
...
int a[10][5], b[40][20];
...
cout << sum(____(2)____, 10*5);  // 1分
cout << sum(____(3)____, 40*20);  // 1分
```

解析：

- **目标**：计算二维数组所有元素的和。
- **方法**：将二维数组视为连续的一维数组，传递首地址和元素总数。
- **填空**：
  - **(1)**：函数参数为指针和总数，写作 `int* x, int num` 。
  - **(2)**： `a` 的首地址为 `&a[0][0]` ，元素数为 `10*5` 。
  - **(3)**： `b` 的首地址为 `&b[0][0]` ，元素数为 `40*20` 。
- **原理**：C++中二维数组按行优先存储，首地址可作为一维数组处理。

答案：

- **(1)**： `int* x, int num`
- **(2)**： `&a[0][0]`
- **(3)**： `&b[0][0]`

说明：

- `int* x` 接收数组首地址， `num` 指定循环范围。
- `&a[0][0]` 和 `&b[0][0]` 是数组的起始内存位置。

18. 阅读程序回答问题（4 分）

题目代码：

```
class A {
    int x, y;
    char *p;
public:
    A(char *str) {
        x = 0; y = 0;
        p = new char[strlen(str)+1];
        strcpy(p, str);
    }
    ~A() { delete [] p; p = NULL; }
    ...
};
A a1("abcd");
A a2(a1);
```

- **(1) 以上代码存在什么问题？（2 分）**
- **(2) 如何解决？（2 分）**

解析：

- **问题分析**：
  - `A a2(a1)` 调用默认拷贝构造函数，执行浅拷贝。
  - `a1` 和 `a2` 的 `p` 指向同一块动态内存。
  - 析构时， `a1` 和 `a2` 都调用 `delete[] p` ，导致重复释放同一内存，引发未定义行为。
- **解决方法**：
  - 定义深拷贝的拷贝构造函数，为 `a2` 分配新内存并复制 `p` 的内容。

答案：

- **(1)**：浅拷贝导致的重复释放问题。

- **(2)**：实现拷贝构造函数，进行深拷贝。

说明：

- **问题**：浅拷贝只复制指针地址，未复制数据。
- **解决代码示例**：

```
A(const A& other) {
    x = other.x; y = other.y;
    p = new char[strlen(other.p) + 1];
    strcpy(p, other.p);
}
```

19. 阅读下列程序，写出程序具体调用函数（6 分）

题目代码：

```
#include <iostream>
using namespace std;
class A {
public:
    A() { f(); }
    virtual ~A();
    virtual void f();
    void g();
    void h() { f(); g(); }
};
class B : public A {
public:
    ~B();
    void f();
    void g();
};
void main() {
    B b;  // 调用 B::B(), A::A() 和 A::f()
    A *p;
    p = &b;
    p->f();  // 调用 B::f()
    p->A::f();  // 调用 A::f()
    p->g();  // 调用 A::g()
    p->h();  // 调用 _____(1)_____
    p = new B;  // 调用 _____(2)_____
    delete p;  // 调用 _____(3)_____
}
```

解析：

- **(1) p->h()**：
  - `h()` 是 `A` 的非虚函数，调用 `A::h()`。
  - 内部：`f()` 是虚函数，`p` 指向 `B`，调用 `B::f()`；`g()` 非虚，调用 `A::g()`。
  - 结果：`A::h()`（内部调用 `B::f()` 和 `A::g()`）。
- **(2) p = new B**：
  - 创建 `B` 对象，先调用基类 `A::A()`，再调用 `B::B()`。
- **(3) delete p**：

- p 是 A* 类型，指向 B，因 ~A() 是虚函数，调用 B::~B()。

答案：

- **(1)**：A::h()（内部调用 B::f() 和 A::g()）
- **(2)**：A::A() 和 B::B()
- **(3)**：B::~B()

说明：

- 虚函数通过动态绑定调用派生类实现，非虚函数按声明类调用。
- 构造顺序：基类 → 派生类；析构顺序：派生类 → 基类。

20. 填写程序，完成文件输入（4 分）

题目代码：

```
#include __(1)___
#include <iostream>
using namespace std;
struct Student {
    int no;
    char name[10];
    int scores[5];
} s1;
void main() {
    // 以二进制方式输入数据
    ifstream in_file("d:\\students.dat", ____(2)____);
    if(_____(3)____) {
        cerr << "Fail to open file" << endl;
        exit(1);
    }
    in_file.read(_____(4)_____, sizeof(s1));
    in_file.close();
}
```

解析：

- **目标**：从文件读取二进制数据到结构体 s1。
- **填空**：

- **(1)**：文件流需包含 `<fstream>`。
- **(2)**：二进制模式为 `ios::binary`。
- **(3)**：检查文件打开失败，`!in_file.is_open()`。
- **(4)**：`read` 需 `char*` 类型指针，转换 `s1` 地址为 `reinterpret_cast<char*>(&s1)`。

答案：

- **(1)**：`<fstream>`
- **(2)**：`ios::binary`
- **(3)**：`!in_file.is_open()`
- **(4)**：`reinterpret_cast<char*>(&s1)`

说明：

- `<fstream>` 提供 `ifstream` 类。
- 二进制读取需类型转换，确保内存对齐。

21. 完成如下程序（4 分）

题目代码：

```
#include <iostream>
using namespace std;
template <__(1)___>
class Stack {
    T buffer[size];
    int top;
public:
    Stack() { top = 1; }  // 应为-1
    bool push(const T &x) {
        if (top == __(2)___) {
            cout << "Stack is overflow.\n";
            return false;
        } else {
            top++; buffer[top] = x;
            return true;
        }
    }
    bool pop(T &x) {
        if (top == -1) {
            cout << "Stack is empty.\n";
            return false;
        } else {
            _____(3)_____; top--;
            return true;
        }
    }
};
int main() {
    double x;
    Stack<__(4)___> st1;  // 为元素个数为100的double型栈
    st1.push(10.0);
    st1.pop(x);
}
```

解析：

- **目标**：实现模板栈类。
- **填空**：

- **(1)**：模板参数包括类型和大小， `class T, int size` 。
- **(2)**：栈满条件， `size - 1` （索引从 0 到 `size-1` ）。
- **(3)**：出栈赋值， `x = buffer[top]` 。
- **(4)**：实例化为 `double` 类型，100 个元素， `double, 100` 。

- **修正**： `Stack()` 中 `top = 1` 应为 `top = -1` （空栈）。

答案：

- **(1)**： `class T, int size`
- **(2)**： `size - 1`
- **(3)**： `x = buffer[top]`
- **(4)**： `double, 100`

说明：

- 模板参数支持类型和非类型参数。
- 栈操作遵循“后进先出”， `top` 初始化为-1。

简答题

22. 在面向对象程序设计中，如何理解数据的抽象与封装?

- **数据抽象**：
  数据抽象是指将数据的具体实现细节隐藏起来，只向用户提供必要的操作接口。
  在 C++中，通过**类（class）**实现数据抽象，类中包含数据成员和成员函数，用户通过调用成员函数操作数据，而无需了解内部如何实现。
  例如，定义一个 `BankAccount` 类，用户可以用 `deposit()` 存钱， `withdraw()` 取钱，但不需要知道余额是如何存储或计算的。
  核心：关注“做什么”，而不是“怎么做”。
- **封装**：
  封装是数据抽象的具体实现方式，通过访问控制（如 `public` 、 `private` 、 `protected` ）保护数据，防止外部直接访问或修改。
  在 C++中，类的私有成员（如 `private int balance;` ）只能通过公有方法（如 `getBalance()` ）访问，确保数据安全性和实现的可维护性。
  例如，在 `BankAccount` 类中， `balance` 设为 `private` ，外部只能通过方法间接操作。
  核心：隐藏实现细节，提供受控访问。
- **关系与总结**：
  数据抽象定义了接口（做什么），封装保护了实现（怎么做）。两者结合实现了面向对象设计中的模块化和信息隐藏。

23. 拷贝构造函数的作用是什么？何时会调用拷贝构造函数？

- **作用**：
  拷贝构造函数是一个特殊的构造函数，用于**创建一个新对象，并用已有对象的成员变量值初始化新对象**。
  它确保新对象是现有对象的副本，尤其在类中有动态分配资源（如指针）时，需进行**深拷贝**，避免浅拷贝导致的问题（如多个对象共享同一内存）。
  例如：

```
class MyClass {
    int* ptr;
public:
    MyClass(const MyClass& other) {  // 拷贝构造函数
        ptr = new int(*other.ptr);  // 深拷贝
    }
};
```

- **调用时机**：
  i. **对象初始化**：用一个对象初始化另一个对象时，例
  如 `MyClass obj2 = obj1;` 或 `MyClass obj2(obj1);` 。
  ii. **函数参数传递**：对象以值传递方式传入函数时，形参创建副本，例
  如 `void func(MyClass obj); func(obj1);` 。
  iii. **函数返回值**：函数返回对象时，返回值创建副本，例如 `MyClass func() { return obj1; }` （除非有返回值优化）。
- **总结**：
  拷贝构造函数保证对象复制的正确性，尤其在资源管理中至关重要，避免浅拷贝的错误。

24. C++怎样实现消息的动态绑定，请简单说明下实现过程。

- **动态绑定**：
  动态绑定是指在**运行时**（而非编译时）决定调用哪个函数版本，主要用于继承和多态场景。C++通过**虚函数（virtual functions）**实现。
- **实现过程**：
  i. **虚函数表（vtable）**：
  当类中定义虚函数时，编译器为该类生成一个虚函数表，存储所有虚函数的地址。

ii. **虚函数指针（vptr）**：
每个含虚函数的对象在内存中有一个隐藏的虚函数指针，指向其类的 vtable。

iii. **运行时解析**：
通过基类指针或引用调用虚函数时，程序根据对象的 `vptr` 找到 vtable，再从表中获取实际函数地址（通常是派生类的实现）。
例如：

```
class Base {
public:
    virtual void func() { cout << "Base"; }
};
class Derived : public Base {
public:
    void func() override { cout << "Derived"; }
};
Base* p = new Derived();
p->func();  // 输出 "Derived"
```

- **总结**：
动态绑定通过 vtable 和 vptr 实现运行时多态，确保调用正确的函数版本。

25. C++标准模板库（STL）中包含哪几类模板？它们的作用分别是什么？

STL（标准模板库）包含以下几类模板，每类有特定作用：

1. **容器（Containers）**：
   - **作用**：存储和管理数据，提供不同结构。
   - **示例**： `vector` （动态数组）、 `list` （双向链表）、 `map` （键值对）、 `set` （唯一集合）。
2. **迭代器（Iterators）**：
   - **作用**：遍历容器元素，像指针一样访问数据，支持泛型操作。
   - **示例**： `vector<int>::iterator` 用于遍历 `vector` 。
3. **算法（Algorithms）**：
   - **作用**：提供通用数据操作，与容器和迭代器配合使用。
   - **示例**： `sort` （排序）、 `find` （查找）、 `copy` （复制）。
4. **函数对象（Functors）**：
   - **作用**：重载 `operator()` 的类对象，用于自定义算法行为。
   - **示例**： `greater<int>()` 用于降序排序。
5. **适配器（Adapters）**：

- **作用**：修改现有容器接口，提供新功能。
- **示例**： `stack` （栈）、 `queue` （队列）。

6. **分配器（Allocators）**：
   - **作用**：管理内存分配和释放，可自定义内存策略。
   - **示例**： `std::allocator` 是默认分配器。

- **总结**：
  STL 通过这六类模板提供高效、灵活的泛型编程工具，广泛用于数据管理和处理。

程序设计题

26. 编写类 String 的构造函数、析构函数、赋值函数，以及测试的 main 函数（9 分）

题目要求：

已知类 `String` 的原型如下：

```
#include <iostream>
#include <string.h>
class String {
public:
    String(const char *str = NULL);  // 普通构造函数
    String(const String &other);     // 拷贝构造函数
    ~String();                       // 析构函数
    String operator=(const String &other);  // 赋值函数
    void show() { std::cout << m_data << std::endl; }
private:
    char *m_data;  // 用于保存字符串
};
```

需要实现构造函数、析构函数、赋值函数，并编写 `main` 函数进行测试。

解答：

实现代码：

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

class String {
public:
    // 普通构造函数
    String(const char *str = NULL) {
        if (str == NULL) {
            m_data = new char[1];  // 分配1字节给空字符串
            *m_data = '\0';        // 设置为空字符
        } else {
            m_data = new char[strlen(str) + 1];  // 分配空间，含结束符'\0'
            strcpy(m_data, str);                 // 复制字符串
        }
    }

    // 拷贝构造函数
    String(const String &other) {
        m_data = new char[strlen(other.m_data) + 1];  // 深拷贝，分配新内存
        strcpy(m_data, other.m_data);                 // 复制内容
    }

    // 析构函数
    ~String() {
        delete[] m_data;  // 释放动态分配的内存
    }

    // 赋值函数
    String operator=(const String &other) {
        if (this != &other) {  // 防止自赋值
            delete[] m_data;   // 释放旧内存
            m_data = new char[strlen(other.m_data) + 1];  // 分配新内存
            strcpy(m_data, other.m_data);                 // 复制内容
        }
        return *this;  // 返回当前对象，支持连等
    }

    // 显示函数（已提供）
```

```
    void show() { cout << m_data << endl; }

private:
    char *m_data;  // 用于保存字符串
};

// 测试的main函数
int main() {
    String s1("Hello");  // 普通构造函数
    String s2(s1);       // 拷贝构造函数
    String s3;           // 默认构造函数
    s3 = s1;             // 赋值函数
    s1.show();           // 输出 "Hello"
    s2.show();           // 输出 "Hello"
    s3.show();           // 输出 "Hello"
    return 0;
}
```

说明：

- **普通构造函数**：处理空指针和非空字符串，动态分配内存并复制内容。
- **拷贝构造函数**：深拷贝，确保新对象有独立的内存副本。
- **析构函数**：释放动态内存，避免内存泄漏。
- **赋值函数**：防止自赋值，释放旧内存后深拷贝新内容。
- **main 函数**：测试各种构造和赋值情况，验证功能正确性。

27. 定义抽象立体图形类 Geometry，派生出球体、长方体、圆柱体类，并求体积之和（9 分）

题目要求：

- 定义抽象基类 `Geometry` ，包含名称 `name` 和纯虚函数 `getVolume()` 。
- 派生出 `Sphere` （球体）、 `Cuboid` （长方体）、 `Cylinder` （圆柱体）类，实现 `getVolume()` 。
- 在 `main` 函数中用基类指针数组计算体积之和。

解答：

实现代码：

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

#define PI 3.1415926535

// 抽象基类Geometry
class Geometry {
public:
    string name;
    Geometry(string n) : name(n) {}
    virtual double getVolume() = 0;  // 纯虚函数
};

// 球体类
class Sphere : public Geometry {
    double r;  // 半径
public:
    Sphere(double radius) : Geometry("Sphere"), r(radius) {}
    double getVolume() { return (4.0 / 3.0) * PI * r * r * r; }  // 体积公式：4/3πr³
};

// 长方体类
class Cuboid : public Geometry {
    double l, w, h;  // 长、宽、高
public:
    Cuboid(double length, double width, double height)
        : Geometry("Cuboid"), l(length), w(width), h(height) {}
    double getVolume() { return l * w * h; }  // 体积公式：lwh
};

// 圆柱体类
class Cylinder : public Geometry {
    double r, h;  // 底半径、高
public:
    Cylinder(double radius, double height)
        : Geometry("Cylinder"), r(radius), h(height) {}
    double getVolume() { return PI * r * r * h; }  // 体积公式：πr²h
};
```

```
// main函数
int main() {
    Geometry* shapes[3];  // 基类指针数组
    shapes[0] = new Sphere(1.0);       // 半径1的球体
    shapes[1] = new Cuboid(1.0, 2.0, 3.0);  // 长1宽2高3的长方体
    shapes[2] = new Cylinder(1.0, 2.0);     // 底半径1高2的圆柱体

    double totalVolume = 0.0;
    for (int i = 0; i < 3; i++) {
        totalVolume += shapes[i]->getVolume();  // 多态调用
        delete shapes[i];  // 释放内存
    }
    cout << "Total Volume: " << totalVolume << endl;
    return 0;
}
```

说明：

- **抽象基类**： `Geometry` 通过纯虚函数 `getVolume()` 实现抽象。
- **派生类**：实现具体体积计算公式。
- **main 函数**：用 `Geometry*` 数组实现多态，动态分配对象并计算总和，结束后释放内存。
- **输出**：示例中输出球体、长方体、圆柱体的体积和。

28. 编写学生成绩输入输出程序，使用重载操作符>>和<<（10 分）

题目要求：

- 类 `StudentScores` 定义如下：

```
const int MAX_NUM_OF_COURSES = 30;
const int MAX_ID_LEN = 10;
const int MAX_NAME_LEN = 8;
class StudentScores {
public:
    StudentScores() { initialized = false; }
    bool data_is_ok() const { return initialized; }
private:
    int scores[MAX_NUM_OF_COURSES], num_of_courses;
    char id[MAX_ID_LEN + 1], name[MAX_NAME_LEN + 1];
    bool initialized;
    friend istream &operator>>(istream &in, StudentScores &x);
    friend ostream &operator<<(ostream &out, const StudentScores &x);
};
```

- 从键盘输入学生信息（学号、姓名、选课门数、成绩），保存到文件。
- 使用重载 >> 和 << 实现输入输出。

解答：

实现代码：

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

const int MAX_NUM_OF_COURSES = 30;
const int MAX_ID_LEN = 10;
const int MAX_NAME_LEN = 8;

class StudentScores {
public:
    StudentScores() { initialized = false; }
    bool data_is_ok() const { return initialized; }
private:
    int scores[MAX_NUM_OF_COURSES], num_of_courses;
    char id[MAX_ID_LEN + 1], name[MAX_NAME_LEN + 1];
    bool initialized;
    friend istream &operator>>(istream &in, StudentScores &x);
    friend ostream &operator<<(ostream &out, const StudentScores &x);
};

// 重载输入操作符
istream &operator>>(istream &in, StudentScores &x) {
    in >> x.id >> x.name >> x.num_of_courses;
    for (int i = 0; i < x.num_of_courses; i++) {
        in >> x.scores[i];
    }
    x.initialized = true;  // 数据有效
    return in;
}

// 重载输出操作符
ostream &operator<<(ostream &out, const StudentScores &x) {
    if (!x.initialized) return out;  // 未初始化则不输出
    out << x.id << " " << x.name << " " << x.num_of_courses;
    for (int i = 0; i < x.num_of_courses; i++) {
        out << " " << x.scores[i];
    }
    return out;
}
```

```cpp
// main函数
int main() {
    StudentScores student;
    cout << "Enter student data (id name courses scores): ";
    cin >> student;  // 从键盘输入

    // 保存到文件
    ofstream out("students.txt");
    if (!out) {
        cerr << "Failed to open file!" << endl;
        return 1;
    }
    out << student;
    out.close();

    // 从文件读取并显示
    ifstream in("students.txt");
    if (!in) {
        cerr << "Failed to open file!" << endl;
        return 1;
    }
    StudentScores student2;
    in >> student2;
    cout << "Read from file: " << student2 << endl;
    in.close();

    return 0;
}
```

说明：

- **重载>>**：从输入流读取学号、姓名、课程数和成绩，标记数据有效。
- **重载<<**：将学生信息格式化输出到流中。
- **main 函数**：实现从键盘输入到文件保存，再从文件读取并显示的完整流程。
- **文件操作**：使用 `fstream` 确保数据持久化。