2021 年 C++ 程序设计语言律师等级考试

试题册

考生注意

1. 题目中所有来自 C++ 标准库 (C++ Standard Template Library) 的名称都假设已经引入了对应的头文件

2. 本次考试不考察多线程和网络编程相关内容, 考生答题时可以不考虑因多线程或网络编程导致的问题

3. 本次考试的涉及范围内容止步于 C++ 11, 考生不必考虑 C++ 11 之后的任何用法

4. 所有写在试题册及草稿纸上的答案无效

5. 本场考试为闭卷笔试, 考试时间为 4 个小时

6. 考试开始前考生禁止以任何方式打开试题册

(一)、选择题 : 1 ~ 10 题, 每小题 2 分, 共计 20 分. 每个选择题给出 A、B、C、D、E、F 和 G 七个选项, 每一题至少有一个正确选项, 至多有七个正确选项. 多选、少选或错选均不得分.

1. 本试题考察的内容是 ( )

A、C++ **B**、D **C**、Erlang **D**、Groovy
E、Fortran **F**、Haskell **G**、Lisp

2. 以下 C++ 关键字存在多个用途的是 ( )

A、thread_local **B**、mutable **C**、namespace
D、static **E**、compl **F**、constexpr **G**、extern

3. 设局部变量 p 被声明为 constexpr void *p[] {nullptr, NULL, 0}; 若下列选项中的所有变量和 p 都处于同一个函数的作用范围内, 那么下列声明不会引起编译错误是 ( )

A、decltype(p) p2;
B、decltype((((p)))) p3();
C、static void **&p4(p);
D、extern void *const *p5 {p};
E、union {struct {int *p};} p6;
F、const auto &&p7 = p[0];
G、volatile const void *const &&p8 = {std::move(p[-1])};

4. 设局部变量 fp 被声明为 extern void (*fp[])(); 若下列选项中的所有变量和 fp 都处于同一个函数的作用范围内, 那么下列声明不会引起编译错误是 ( )

A、auto fp2 {[fp]{return fp;}};
B、decltype(fp) fp3();
C、enum class fp4 : bool {fp, i, j};
D、decltype((fp)) &&fp5 = {fp};
E、struct fp6 {constexpr static auto mem_fp = fp};
F、const void **fp7 = fp;
G、class fp8 {void (*(&mem_ref)[])() {fp};} a, b(a);

5. 下列说法正确的是 ( )

A、delete[] 运算符按照从后往前的顺序进行析构的
B、在模板参数列表中, 使用 typename 或 class 是相同的
C、设变量 p 和 q 分别被声明为 auto p = {0}; int q = {0}; 那么 p 和 q 的类型相同
D、可以使用 std::forward_list 作为 std::queue 或者 std::stack 的底层容器
E、命名空间可以仅声明并且可以通过 using namespace 一次性引入多个命名空间
F、explicit 可以对任意构造函数进行标识
G、可以声明一个函数类型的引用

6. 本质与表现是马克思主义哲学理论中一对互不可分的重要概念，它可以帮助我们分析事物的内在特性和外在表现. 所谓本质，即事物的根本属性，而表现则是本质的外在表面特征和外部联系. 对于同一类事物来说，本质和表现存在互异性 : 本质在这一类事物上体现为普遍性，表现则体现为个性. 但从另一方面而言，本质和表现在同一类事物上又是互不可分的，所谓透过现象看本质. 本质决定了表现的形式，而表现又体现出事物内在的本质. 以下 C++ 特性中，能够体现本质与表现这一对概念的是 (　　)

A、使用 Range-for 遍历容器和使用迭代器遍历容器

B、int & 和 int *

C、struct 和 class

D、初始化列表和数组

E、long 和 long long

F、lambda 表达式和重载了函数调用运算符的类

G、constexpr 和 const

7. 下列说法正确的是 (　　)

A、下列代码的输出结果为 BBc

```
template <typename T>
void f(T ...) {std::cout << 'A';}
template <typename ...T>
void f(T ...) {std::cout << "B";}
void f(unsigned) {std::cout << "u";}
void f(int) {std::cout << 'i';}
void f(char) {std::cout << "c";}
auto main() -> int {
    f(1.0);
    f(1.0f, 0.0f);
    char x = 1, y = 2;
    f(x + y);
}
```

B、下列代码无法通过编译

```
struct s1;
struct s2 {
   s2() {std::cout << 'A';}
   friend s2 s1::create_s2();
};
struct s1 {
    s1() {std::cout << 'B';}
    s2 create_s2() {return {};}
};
int main() {
    s1 x;
    s2 y = x.create_s2();
}
```

C、下列代码的输出结果为 001101

```
struct s1 {
    void f() {std::cout << 0;}
    void f() const {std::cout << 1;}
};
struct s2 {
    std::vector<s1> v;
    std::unique_ptr<s1> up;
    const s1 *cp;
    s2() : v(1), up(new s1()), cp(up.get()) {}
};
int main(int argc, char *argv[]) {
    s2 s;
    const s2 &ref {s};
    s.v[0].f();
    s.up->f();
    s.cp->f();
    ref.v[0].f();
    ref.up->f();
    ref.cp->f();
}
```

D、下列代码的输出结果为 1012

```
struct s {
    int i;
    operator bool() const {return i;}
};
class c {
    public:
        c() {std::cout << 1;}
        c(int) {std::cout << 1;}
        c &operator=(const c &) {
            std::cout << 2;
            return *this;
        }
};
int main() {
    s x {0}, y {1};
    std::cout << x + y << (x == y);
    std::map<int, C> m;
    m[42] = C(x.i);
}
```

E、下列代码的输出结果为 ABCDABCd

```
struct A {
    A() {std::cout << 'A';}
    A(const A &) {std::cout << "a";}
};
class B : virtual public A {
    public:
        B() {std::cout << 'B';}
        B(const B &) {std::cout << "b";}
};
struct C : virtual A {
    C() {std::cout << 'C';}
    C(const C &) {std::cout << "c";}
};
struct D : protected B, public C {
    D() {std::cout << 'D';}
    D(const D &) {std::cout << "d";}
};
int main() {
    D d1, d2(d1);
}
```

F、下列代码的输出结果为 1255

```
struct X {
    X() {std::cout << 1;}
    X(X &) {std::cout << 2;}
    X(const X &) {std::cout << 3;}
    X(X &&) {std::cout << 4;}
    ~X() {std::cout << 5;}
};
struct Y {
    mutable X x;
    Y() = default;
    Y(const Y &) = default;
    ~Y() = default;
};
int main() {
    Y x;
    Y y = std::move(x);
}
```

G、下列代码的输出结果为 11010

```
char str[] {"0"};
struct s {
    s() noexcept {*a = '1';}
    ~s() noexcept {*a = '0';}
    operator const char *() const noexcept {return str;}
};
void print(const char *str) {std::cout << str;}
s make() noexcept {return s();}
int main(int argc, char *argv[]) {
    s s1 = make();
    print(s1);
    const char *s2 = make();
    print(s2);
    print(make());
}
```

8. 以下代码中，可能存在未定义行为的是（　）

A、
```
int main() {
    auto integral = std::numeric_limits<unsigned long long>::max();
    ++integral;
}
```

B、
```
char32_t _global {0};
int main(int argc, char *argv[]) {
    std::cout << ::_global << std::endl;
}
```

C、
```
unsigned short x {0xFFFF}, y(x);
auto z = x * y;
```

D、
```
struct s {
    int i : 1;
    double j;
};
int main(int argc, char *argv[]) {
    s x;
    std::cout << (&x.i < reinterpret_cast<int *>(&x.j));
}
```

E、
```
char *str = const_cast<char *>("C++");
str[1] = str[2] = '-';
```

F、
```
struct s {};
bool b = &typeid(s) == &typeid(s);
```

G、
```
int main(int argc, char *argv[]) {
    std::cout << std::boolalpha << (argv[argc] == nullptr);
}
```

其中，函数 main 是程序入口

9. 以下代码中，可能会引起编译错误的有（　）

A、
```
template <typename, typename> struct s1 {};
template <typename T, typename U>
struct s2 : s1<T, U> {
    int i;
    void f() {
        this->f(i);
    }
};
```

其中，类 s2 直到编译为止，从未被使用

B、
```
struct final {};
struct base {
    virtual final f() = 0;
};
struct derived : base {
    virtual auto f() -> final override final {
        return {};
    }
};
```

C、
```
using t = long long;
void f(unsigned long long t);
int main(int argc, char *argv[]) {
    f(0u);
}
```

D、
```
template <typename T>
void call(std::functional<void (T)> f, T value) {
    f(std::move(value));
}
const auto lambda = {[](int x) {}};
int main() {
    call(lambda, 0);
}
```

E、
```
struct s {
    int i;
    s() = default;
};
const s value;
```

F、
```
const void *p = &p;
```

G、
```
class c {
    private:
        int i;
    public:
        explicit c(int i = 0) noexcept : i {i} {}
};
```

10. 下列代码中，最终的输出结果为 11010011 且不存在任何未定义行为的是（　）

A、
```
void f(const std::string &) {std::cout << 0;}
void f(const void *) {std::cout << 1;}
int x;
struct s1 {
    s1() {
        std::cout << 0;
        if(not x++) {
            throw 1;
        }
    }
    ~s1() {std::cout << 1;}
};
struct s2 {
    s1 s;
    s2() {std::cout << 0;}
    ~s2() {std::cout << 1;}
};
void f() {static s2 s;}
int main(int agrc, char *argv[]) {
    const char *str2 = "";
    f("""");
    f(str2);
    try {
        f();
    }catch(...) {
        std::cout << 1;
        f();
    }
}
```

B、
```
struct s {
    s() {
        std::cout << 1;
    }
    s(int) {
        std::cout << 1;
    }
    template <typename I>
    s(I, I) {
        std::cout << 1;
    }
    explicit s(std::initializer_list<int>) {
        std::cout << 0;
    }
};
int x = 1, y = 0, a, b;
int main() {
    a = x, y;
    b = (x, y);
    s s0 {}, f1(1), s1 {1}, f2(1, 1), s2 {1, 0}, s3 {0, 0, 0};
    std::cout << a << b;
}
```

C、
```
template <typename T>
void f(T) {
    static int i {1};
    std::cout << i--;
}
struct s1 {
    s1() {
        std::cout << 1;
    }
    ~s1() {
        std::cout << 0;
    }
};
struct s2 {
    s2(const s1 &) {
        std::cout << 0;
    }
    ~s2() {
        std::cout << 1;
    }
    void f() {
        std::cout << 1;
    }
};
int main(int argc, char *argv[]) {
    f(1);
    f(1u);
    f(1);
    s2 s(s1());
    s.f();
}
```

D、
```
constexpr int f() noexcept {
    return 1;
}
struct base {
    base() {
        f();
    }
    ~base() {
        f();
    }
    virtual void f(bool b = false) {
        std::cout << 1;
        if(b) {
            f();
        }
    }
    void g() {
        f(true);
    }
};
struct derived : base {
    void f(bool = true) {
        std::cout << 0;
    }
};
void f(int, int i = f()) {
    std::cout << i;
}
int main() {
    std::map<bool, std::string> m {
        {1, "hello"}, {2, ", "},
        {3, "world"}, {4, "!"}
    };
    std::cout << m.size();
    derived().g();
    struct {
        int i {};
        int &get() noexcept {
            return this->i;
        }
        void print() const {
            std::cout << this->i;
        }
    } l;
    auto i = l.get();
    ++i;
    l.print();
    std::cout << std::is_same<int, const int>::value;
    f(0);
    f(0);
}
```
其中，std::is_same 是一个接受两个模板参数的类模板．不妨记接受的模板参数为 T 和 U，若 T 和 U 是同一个类型，那么 std::is_same<T, U>::value 的值为 true；否则，std::is_same<T, U>::value 的值为 false

E、
```
template <typename T>
void f_impl(T) {
    std::cout << 1;
}
struct s {};
template <typename T>
void f(T value) {
    f_impl(s());
    f_impl(value);
}
void f_impl(s) {
    std::cout << 0;
}
namespace n1 {
    class c;
    void f(const c &) {
        std::cout << 1;
    }
}
class n1::c {};
namespace n2 {
    void f(const n1::c &) {
        std::cout << 0;
    }
}
int main(int argc, char *argv[]) {
    int x {}, y {}, z = 0;
    std::cout << (++x || ++y && ++z);
    f(s());
    std::cout << x << y << z;
    f(n1::c());
    struct base {
        void f(int) {
            std::cout << 0;
        }
    };
    struct derived : base {
        void f(double) {
            std::cout << 1;
        }
    };
    derived {}.f(0);
}
```

F、
```
auto j = 0;
struct s {
    s() {
        std::cout << 1;
    }
    s(const s &) {
        std::cout << 1;
    }
    const s &operator=(const s &) const {
        std::cout << 0;
        return *this;
    }
};
template <typename T>
void f(T) {
    std::cout << 0;
}
template <>
void f<>(int *) {
    std::cout << 0;
}
template <typename T>
void f(T *) {
    std::cout << 1;
}
int main() {
    s s1;
    s s2 {s1};
    s s3 = s1;
    int *p = nullptr;
    f(p);
    int a = '0';
    const char &b = a;
    std::cout << b;
    ++a;
    std::cout << b;
    int &i = j, j;
    j = 1;
    std::cout << i;
    std::cout << j;
}
```

G、
```
struct s {
    explicit s(int) {std::cout << 0;}
    s(double) {std::cout <<1;}
    static int x;
    static int f() {return 1;}
};
int i;
constexpr int f() noexcept {return 0;}
int s::x = f();
void g() {std::cout << 1;}
template <typename T>
struct base {
    void g() {std::cout << 0;}
};
template <typename T>
struct derived : base<T> {
    void mem_f() {g();}
};
class c {
    public:
        c() {std::cout << 1;}
        ~c() {std::cout << 0;}
};
int main(int argc, char *argv[]) {
    do {
        std::cout << ++i;
        if(i-- < 3) {
            continue;
        }
    }while(false);
    int n = sizeof *new c + 2;
    switch(--------i, n % 2) {
        i = 0;
        case 0:
        do
        {++i;
        case 1:
        ++i;}
        while(--n > 0);
        default:
        break;
        ++++++i;
    }
    std::cout << i;
    s s1(42);
    s s2 = 42;
    std::cout << std::is_same<char, signed char>::value;
    std::cout << std::is_same<char, unsigned char>::value;
    std::cout << s::x;
    derived<int>().mem_f();
}
```

(二)、填空题 : 11 ~ 15 题, 每小题 3 分, 共计 15 分. 对于多个答案的填空题, 全填对得 3 分, 漏填得 1 分, 错填不得分.

11. 设有类

```
class base {
    private:
        int a;
        char b;
        std::string str;
        std::map<std::string, void (*)(int &, char)> m;
        std::vector<std::set<std::bitset<sizeof 0>>> vec;
    private:
        void copy();
        template <typename Iterator>
        void construct_ranges(Iterator, Iterator);
};
```

现在需求变更, 要求类 base 中所有 private 成员都更正为 public 成员, 请在不更改上述代码的情况下添加某些代码以实现该需求, 并且给出注释, 明确被添加的代码应该添加于何处 (不必考虑添加代码之后所导致的任何后果) :

_____.

12. 设有代码

```
void func();
template <class>
struct meta_class;
template <typename T>
using result = typename meta_class<T>::type;
template <typename T>
struct meta_class {
    using type = T (&)[std::is_void<void *const>::value + 42];
};
template <typename R, typename ...Ts>
struct meta_class<R (*)(Ts...) noexcept> {
    using type = R (Ts...);
};
struct base {
    virtual result<int> operator()(result<base (*)() noexcept>,
        const result<decltype(&func) *>, ...) volatile const &&
        noexcept;
};
struct derived : base {
    virtual result<int> operator()(result<base (*)() noexcept>,
        const result<decltype(&func) *>, ...) const volatile &&
        noexcept;
};
```

写出一指针 p 指向类 derived 中重载的函数调用运算符, 不可使用编译器进行推导 (包括但不限于 auto、decltype 以及 template). 其中, std::is_void 是一个接受一个模板参数 T 的类模板, 若 T 为 void 类型, 那么 std::is_void<T>::value 的值为 true; 否则, std::is_void<T>::value 的值为 false. 除此之外, 请移除 p 对应类型中可有可无的 const 或者 volatile 限定, 并将所有类型别名 result<T> 替换为对应的真实类型 :

_____.

13. 写出可能会发生引用折叠的情形 :

_____.

14. 设有一类模板被声明为

```
template <typename Arg, typename F>
class function_wrapper;
```

其作用为对函数进行包装并延迟函数的调用, 以达到懒惰求值的目的. 其中, Arg 表示函数接受的参数对应的类型, F 表示函数的类型. 现要针对函数

```
typename std::add_lvalue_reference<void>::type f(int) noexcept;
```

使用类模板 function_wrapper 进行包装. 在进行包装之前, 若要首先对类模板 function_wrapper 针对函数 f 进行模板实例化, 使得类模板 function_wrapper 针对函数 f 有对应的实体, 则需要写出的声明是 :

_____.

15. 设有函数声明

```
void f(const decltype(nullptr) &);
void f(void *);
void f(const void **&);
void f(volatile void *const *[]);
```

则函数调用 f(NULL); 所选中的函数是 :

_____.

(三)、论述题 : 16 ~ 20 题, 每小题 2 分, 共计 10 分.

16. (1) 简述下列代码可能导致未定义行为的原因, 并且在不更改函数类型的情况下进行改进 :

```
void process(void *, int);
int id_info(int = 0) noexcept;
int main(int argc, char *argv[]) {
    process(new int, id_info(argc));
}
```

(2) 简述 vector<int> {1, 2, 3, 4, 5, 6, 7}[3] 不会导致未定义行为的原因.

17. 简述下列代码并不存在编译错误的原因, 并分析代码最终的输出结果 :

```
int main() {
    std::vector<char> delimiters {",", ";"};
    std::clog << std::hex << delimiters[0] << std::endl;
}
```

18. 下列代码的输出结果并非 -3, 请分析原因并且更改代码使得其输出结果为 -3.

```
int main() {
    int x = 8, y = 6;
    std::cout << -++++x+++++y++;
}
```

19. 设有代码

```
template <typename T, typename U>
void process_impl(T &t, U &u);
template <typename ContainerLHS, typename ContainerRHS>
void process(ContainerLHS &c1, ContainerRHS &c2) {
    assert(v1.size() == v2.size());
    auto it = v1.begin();
    const auto end = v1.end();
    for(auto i = c2.size(); it != end; ++it, static_cast<void>(i)) {
        process_impl(*it, c2[i]);
    }
}
```

分析语句 ++it, static_cast<void>(i), 并说明在各种情形下该代码的必要性.

20. (1) 分析当移动构造函数或者移动赋值运算符仅被定义其中任意一个时, 另外一个会被编译器声明为被删除的函数的原因.

(2) 结合移动语意, 分析 C++ 11 标准废除已经自定义其中任意一个拷贝操作或者析构函数的情况下, 仍然由编译器生成另一个拷贝操作的原因.

(四)、改错题 : 21 ~ 23 题, 每小题 5 分, 共计 15 分. 每个改错题的错误包括但不限于编译错误、未定义行为、错误结果和不正确的实现, 指出可能存在错误的地方并且在不删除代码的情况下更改代码使得其可以正确执行. 初始为 5 分, 漏改一处扣 1 分, 改错一处扣 2 分, 得分不低于 0 分.

21.
```
template <typename T>
class number {
    private:
        T x;
    public:
        constexpr number(T value) : x {value} {}
        T &get() noexcept {
            return this->x;
        }
        //...
};
template <typename T>
inline number<T> operator+(const number<T> &x, const number<T> &y) {
    return number<T>(x.get() + y.get());
}
int main() {
    const int x = 9;
    constexpr auto y = number<decltype(x)>(x) + 42;
}
```

22.
```
int main(int argc, char *argv[]) {
    union {
        private:
            int i;
            double d;
            std::string s;
        public:
            int &get_i() {
                return this->i;
            }
            double &get_d() {
                return this->d;
            }
            std::string &get_s() {
                return this->s;
            }
    } l;
    l.i = 0;
    l.s = "123";
}
```

23.
```
class vector_int {
        private:
            int *p;
            std::size_t n;
        public:
            explicit constexpr vector_int(int n) :
                    p {new int[n] {}}, n {n} {}
            vector_int(const vector_int &rhs) :
                    p {new int[rhs.n] {}}, n {rhs.n} {
                std::memcpy(this->p, rhs.p, n);
            }
            vector_int(vector_int &&) noexcept = default;
            ~vector_int() {
                delete this->p;
            }
            vector_int &operator=(const vector_int &rhs) {
                auto new_p {new int[rhs.n]};
                int i {};
                for(auto elem : rhs.p) {
                    new_p[i++] = elem;
                }
                this->n = rhs.n;
                delete this->p;
            }
            vector_int &operator=(vector_int &&) noexcept = default;
    };
```

㈤、程序设计题 : 24 ~ 28 题, 共计 40 分. 本题中除了有额外的补充说明之外, 要求全程自行设计, 不可使用类似于 C++ 标准库等库中的任何设施.

24. 特性创造题（4 分）: ① ~ ② 题, 每题 2 分. 下列代码中部分特性是 C++ 11 未加入的, 请指出这些特性并介绍这些特性的用处 :

①
```
void f() {
    constexpr auto v = 0xFF'01'AB'CD;
    std::vector vec {v, v, v, v};
    [vec = std::move(vec)] {}();
}
```

②
```
constexpr auto f() noexcept(auto) {
    enum class color {blue, yellow, red};
    using enum color;
    color c {};
    while(c not_eq blue) {
        //...
        if(c == yellow) {
            return;
        }
        //...
    }
}
```

25. 面向对象设计题（6 分）: 交通工具（transportation）是人类（human）智慧（wisdom）的结晶之一. 其总共有几大类 : 自行车（bicycle), 汽车（car), 船（ferry）和飞机（aircraft). 自行车可以骑行（cycle), 轮胎（tire）需要打气（inflate). 通过给自行车进行改装（refit), 就有了电单车, 但它需要充电（charge). 汽车可以驾驶（drive）和加油（oil), 现代某些电动汽车需要充电. 船需要航行（sail）也可以漂浮（float）在水（water）面上, 船上必须具备导航（navigation). 电影（movie）中经常出现的水陆两用车（amphibious vehicle）吸收了船和汽车的共同特点. 自行车、汽车和船都应该配备喇叭（horn). 飞机可以飞行（fly), 飞机上也必须具备导航. 出乎意料的是, 飞机同样可以在水上漂浮. 直升飞机（helicopter）具有飞机没有的特点, 即悬停（hover）在空中, 但是直升飞机通常不能在水上漂浮. 除此之外, 船和飞机都需要加油. 飞船结合了船和飞机的特点, 但是它使用特殊燃料（special fuel）和太阳能（solar energy). 请根据上述描述和下列要求, 设计类 :

①可以使用的 C++ 标准库设施 : std::cout, std::endl

②每一个动作可以使用一个输出语句替换, 不可以出现仅仅声明而不定义的函式

26. 数据结构设计题（6 分）：哈希表（hash table）是 C++ 标准库中 std::unordered_map、std::unordered_multimap、std::unordered_set 和 std::unordered_multiset 的底层数据结构. 哈希表一般通过单链表的数组来设计. 对于暂时拥有 n 个单链表数组的哈希表中，通过一个映射函数 hashing 将给定的参数影射为 [0, n] 范围内的数字，这个数字代表了它应该位于第几个单链表中. 任意一个单链表数组中的单链表还有一个特殊的名字，称为桶（bucket）或者槽（slot）. 要在哈希表中搜索某一个元素，只需要通过映射函数将这个元素影射为索引，然后在对应的单链表中寻找即可，这样的搜索十分高效. 请使用 class 设计一个哈希表，它至少满足元素可以被插入（insert），元素可以被移除（erase），元素可以被放置（emplace），哈希表的初始化（要求哈希表对象的默认初始化拥有编译期计算的能力），判定哈希表是否为空（empty），哈希表的清空（clear），元素的查找（contains），元素总数的统计（size），桶总数的统计（bucket_count）. 除此之外，哈希表应该接受重复元素的插入. 可以使用的 C++ 标准库设施：std::forward_list，但不能声明接受单链表的单链表，即 std::forward_list<std::forward_list<...>>. 其中，映射函数 hashing 被声明为

```
template <typename T>
std::size_t hashing(const T &) noexcept;
```

27. 面向过程设计题（10 分）：① ~ ⑥ 题. 本题要求使用面向过程的程序设计方式进行设计，使用其它设计方式不得分. 其中，除第 ① 小题之外，剩余题目不可以使用递归的方式实现.

①递归设计题（2 分）：分而治之是一种解决问题的有效方案. 其基本思想是当问题足够简单的时候，这个问题可以在瞬间被解决；当问题有一定规模的时候，我们可以通过不断将问题划分为若干个简单子问题来求解. 例如要在若干个元素中寻找最小的元素，当元素总数仅有两个的时候，计算机可以在一瞬间找到那个最小的元素；当元素总数为两个以上的时候，计算机就需要经过若干次比较来选出最小的元素. 现在利用分而治之的思想，我们将元素不断细分，直到要比较的元素只剩下两个为止，从中选出比较小的元素. 这样，我们得到了若干个最小元素的备选. 对于最小元素的备选集合，我们运用同样的方法，直到备选集合中仅剩下两个元素，从中选出最小元素即可. 请依据上述思想，使用递归实现下列函数，其作用是从若干个元素中寻找最大的元素. 其中，函数的声明为：

```
template <typename ...Args>
void find_max(Args &&...);
```

假设类型包 Args 中的任意两个类型都至少可以进行基于小于 "<" 运算符的比较.

②数学设计题（2 分）：对于一个二阶常系数齐次线性递归方程 $f(n) = a_1 f(n-1) + a_2 f(n-2)$，其有两个初始条件 $f(0) = 0,\ f(1) = 1$. 我们通常假设 $f_h(x,n) = A x^n$ 是其解，将 $f_h(x,n)$ 代入递归方程可得到其特征方程 $x^2 = a_1 x + a_2$. 仅考虑实数解 $x = x_1,\ x = x_2$. 若 $x_1 \neq x_2$，则原递归方程的通解为 $f_h(n) = c_1 x_1^n + c_2 x_2^n$；若 $x_1 = x_2$，则原递归方程的通解为 $f_h(n) = c_1 x_1^n + c_2 n x_2^n$. 其中，$c_1, c_2$ 为常数，代入初始条件可以得到具体值. 请设计一程序用于求解二阶常系数齐次线性递归方程.

③STL 设计题（1 分）：某商场有若干个商品（你至少需要添加 10 个商品），这些商品有对应的价格和库存. 商场在进货的时候，首先检查对该商品是否需要进货，然后检查库存，按库存进行补货，且商场不定时引入全新的商品. 当所有商品总库存低于一定值，商场会推出清仓特价，所有商品都能以八五折购入. 商场希望在一次清仓完成之后，按照某一个特定时间内商品的销量进行排序，以商榷是否需要加倍入货. 本题可以任意使用 C++ 标准库的设施，并且任意值可以自由设定（例如库存和价格等）.

④跨文件设计题（1 分）：在 C++ 中，若存在多个变量跨越多个文件，那么这些变量的初始化顺序是未知的，它最终由编译器来决定．例如在 1.cpp 文件中，变量 a 被声明为 int a = 0; 而在 2.cpp 文件中，变量 b 被声明为 int b = 2; 若在 main.cpp 中同时引入变量 a 和 b．此时，变量 a 和 b 在 main.cpp 中仅仅被声明，并不知道它们是否已经被初始化．若在没有被初始化的情况下使用变量 a 和 b 将会导致未定义行为．请设计一方案以解决上述问题，不同文件可以直接使用单独一行的注释 //1.cpp 或者 //2.cpp 的样式进行区分．

⑤代码翻译题（2 分）：编译器在将我们写出的代码转换为机器能够识别的机器码之前，需要对代码进行翻译．首先，对于一个返回类型非 void 的函数，编译器会尝试在原地配置一块内存，然后将这块内存转交给函数初始化之后，函数内部对于该对象的操作可以直接在原地进行，这样避免了返回对象带来的拷贝．这种技术在 C++ 中被称为返回值优化，它可以被用于部分函数．除此之外，编译器在处理类成员函数的时候，也和我们看起来的不太一样．它首先将对象绑定到第一个参数上，然后使用 this 访问成员．如果类中带有虚函数，那么绝大多数编译器会在类的最底部增添一个指向虚函数表的指针．虚函数表中从第一个指针开始，这些指针会指向那些真正会被调用的虚函数．请根据上述信息，翻译下列代码中的函数 f ：

```
class X {
    public:
        X();
        virtual ~X();
        virtual void f();
};
X f() {
    X x;
    X *p = new X();
    x.f();
    p->f();
    delete p;
    return x;
}
```

现假设指向虚函数表的指针名称为 __vptr．虚函数表中的第一个指针指向类 X 的析构函数，第二个指针指向类 X 的虚函数 f

⑥阅读设计题（2 分）：现有一四层楼加装了电梯．不妨设电梯（lift）使用 class 进行了实作，其提供如下操作：上升（up）、下降（down）、停止（stop）、开门（open）、关门（close）、判定某一楼层是否需要停止（stop_at(n)）、获取目前停止的楼层（stop_position）．现在电梯采用的策略是：若目前电梯是上行，那么上行会一直进行到请求电梯的最高楼层，上行过程中在每一个需要请求停止的楼层停止；若目前电梯是下行，那么下行会一直进行到请求电梯的最低楼层，下行过程中在每一个需要请求停止的楼层停止．若在上行或者下行的途中，出现反方向的请求，那么首先处理完当前请求再去处理反方向请求．设计一函数：

```
void run(lift &);
```

来模拟电梯的运行．可以使用的 C++ 标准库设施：std::bitset．

28. 内存控制题（12 分）：① ~ ② 题. 某公司旗下的所有项目都有一个基本要求 : 为了避免内存泄漏，公司规定应该尽量使用 C++ 11 引入的智能指针. 若某一段申请的内存可能会在多个函数下被使用，那么就应该使用智能指针；若要使用内置指针，那么该指针对应的内存仅能在当前函数作用范围之内使用，离开当前函数时，必须释放这段申请的内存. 现该公司某一个项目为了代码的执行效率，在遵守上述前提的基础上，对限于函数内使用的内存全部采用内置指针而不采用智能指针. 另外，由于该项目需要和 C 代码进行交互，因此仅仅使用内置的 ::operator new 和 ::operator delete 来申请内存. 该项目完成之后，需要进行内存泄漏情况的检测，以确保项目交付时不出现内存泄漏的错误. 因此，现在需要手动向代码中添加一个用于检测内存泄漏的类 memory_helper 和相关的代码. 对于类 memory_helper，其使用方式和与其相关的限制如下 :

- 类中的所有成员变量对应类型都是内置类型
- 该类提供了成员函数模板 memory_helper:::allocate 用于申请内存，其接受一个类型为 std::size_t 的参数. 所有原本通过内置的 ::operator new 来完成的内存申请都转交由其来完成. 默认情况下，其返回类型和内置的 ::operator new 的返回类型一致. 除此之外，成员函数模板 memory_helper::allocate 还需要检测类使用者给定的模板参数是否为指针类型. 如果使用者给出的模板参数并非指针类型，应该给出一个说明为 "The template argument must be a type of pointer!" 的编译错误
- 该类提供了成员函数 memory_helper::deallocate 用于释放内存，其接受一个类型为 void * 的参数. 所有通过内置的 ::operator delete 来完成的内存释放都转交由其来完成. 其返回类型和内置的 ::operator delete 的返回类型一致. 除此之外，成员函数 memory_helper::deallocate 还需要检测给定的内存是否经由成员函数模板 memory_helper::allocate 申请. 如果给定的内存并非是由 memory_helper::allocate 申请的，那么应该抛出一说明为 "The given memory is not allocated by memory_helper!" 的异常
- 类的构造函数不应该抛出异常，有编译期计算能力，且交由编译器来完成
- 类的析构函数用于检测是否存在内存泄漏，如果存在内存泄漏，那么抛出异常. 除此之外，析构函数中不能出现任何动态内存申请或者释放的行为
- 在类实现之后，对于需要进行检测的函数，在函数最开始处定义一个 memory_helper 对象. 然后将函数中所有原本使用内置的 ::operator new 和 ::operator delete 进行内存申请或者释放的语句都更改为使用 memory_helper 进行申请或释放. 除此之外，还需要在函数定义处以及函数内部代码之外的地方添加一些代码，就可以做到内存泄漏的检测和泄漏内存的回收，这将最大化地降低了代码的修改. 当出现内存泄漏的情况时，通过 std::cerr 输出 "Memory leak detected!"，并且释放泄漏的内存
- 可以使用的 C++ 标准库设施 : std::runtime_error, std::unique_ptr, std::is_pointer, std::cerr, std::endl
- 泄漏的内存需要手动释放，不可借助 std::unique_ptr
- 提示 : static_assert 是 C++ 11 引入的静态断言，其接受两个函数形式的参数，第一个参数是一个可以转化为 bool 类型的常量表达式；第二个参数是一个字面值字符串. 当第一个参数为 false 时，编译器会抛出编译错误，并且将第二个参数输出作为编译错误提示的一部分

①请根据上述要求实现类 memory_helper（11 分）

②请根据上述要求修改下列代码（1 分）：

```
void f(int size) {
    //...
    auto p {::operator new(size * sizeof(int))};
    //...
    ::operator delete(p);
    //...
}
```

2021 年 C++ 程序设计语言律师等级考试

封页

考生注意

考试开始前考生禁止以任何方式打开试题册