2022 年 C++ 程式设计語言等級考試

試題冊

考生注意

1．題目中所有來自 C++ 標準樣板程式庫（C++ Standard Template Library）的名稱都假設已經引入了對應的標頭檔；

2．本次考試不考察執行緒和網路程式設計相關內容，考生答題時可以不考慮因執行緒或網路程式設計導致的問題；

3．本次考試的涉及範圍內容止步於 C++ 11，考生不必考慮 C++ 11 之後的任何用法；

4．考生需要將答案寫在答題紙上，所有寫在試題冊及草稿紙上的答案無效；

5．本場考試為閉卷筆試，考試時間為 4 個小時；

6．考試開始前考生禁止以任何方式打開試題冊．

(一) 選擇題. 1 ~ 10 題, 每小題 2 分, 共計 20 分. 每個選擇題給出 (**A**), (**B**), (**C**), (**D**), (**E**), (**F**) 和 (**G**) 七個選項, 每一題至少有一個正確選項, 至多有七個正確選項. 多選、少選或錯選均不得分.

1. C++ 的英文全稱為 (　　).
(**A**) CPlusPlus　　(**B**) CPlurPlur　　(**C**) CPlotPlot　　(**D**) CPP
(**E**) CPlumPlum　　(**F**) CPlofPlof　　(**G**) CPackPack

2. 以下 C++ 關鍵字存在省略場景的有 (　　).
(**A**) sizeof　　(**B**) noexcept　　(**C**) class　　(**D**) static
(**E**) typeid　　(**F**) constexpr　　(**G**) delete

3. 設全域名稱空間內有宣告
```
constexpr volatile int **(a)[] {NULL, nullptr, 0}, i {};
```
若下列選項中所有變數都處於同一函式內, 那麼下列宣告不會引起編碼錯誤有 (　　).
(**A**) `decltype(a) a1;`
(**B**) `volatile int *&p1 = **i;`
(**C**) `static volatile auto &&r {(void *)std::move(a[-1]) == i};`
(**D**) `const volatile int ****p2 {new const auto(a)};`
(**E**) `{auto v = {*(*(*(a + 1) + 1) + 1) + i};};`
(**F**) `struct {unsigned b : new auto(0) ? 8 : i = {42};} s = {};`
(**G**) `const volatile int *const *volatile p3 {a[2]};`

4. 設有宣告 `const void (*p1(volatile void (*p2)()))(void (&p3)());` 若該宣告和下列選項中的宣告都處於同一個函式的可視範圍內, 那麼下列宣告不會引起編碼錯誤有 (　　).
(**A**) `auto p2 = {[&]() {return p1 == p3;}};`
(**B**) `decltype(((p1))) p;`
(**C**) `decltype((p1({}))) &&r = {};`
(**D**) `union {const volatile auto p = p1;} u1 {}, u2(u1), u3();`
(**E**) `class c {const void (*p1(volatile void (*p2)()))(void (&p3)()) &;};`
(**F**) `const void (*(*fp)(volatile void (*p3)()))(void (&p2)())(p1);`
(**G**) `auto ap = (decltype(nullptr))p3;`

5. 亞里斯多德在《物理學》中提到 : "一個自然物體的自然本性和真實存在就是它的主要材料(自身中尚未成形的材料)——比如, 床的自然本性是木頭, 雕像的自然本性是青銅. 如果你埋下一張床, 並且如果腐爛的木頭能夠長出幼芽的話, 那麼長出來的東西不會是床, 而是樹木——因為根據技藝規則所作的安排只能是偶然之物, 而實在——事物的真正所是—則是在所有變化中始終持存的東西". 下列選項中, 能夠體現這段話的觀點的有 (　　).
(**A**) 如果有足夠的知識儲備和能力, 那麼我們完全可以用 C++ 來編寫 C++ 編碼器.
(**B**) 不論是什麼型別, 包括內建型別和用戶自訂型別, 它們都是數字.
(**C**) 由於 C++ 是弱型別語言, 所以 `char *` 和 `void *` 非常有用.
(**D**) 雖然 C++ 繼承自 C, 但是 C 和 C++ 並不完全相容.
(**E**) 其實, `std::front_insert_iterator` 和 `std::back_insert_iterator` 都無法窺探到容器內部的樣子.
(**F**) 我們的主要任務其實是提升演算法的效能, 而不是在語言層面過度優化. 通常來說, 一般人能想到的普通優化方法, 其實編碼器也會, 甚至比我們的優化方法更好.
(**G**) `std::vector` 和 `std::deque` 用起來好像沒什麼太大區別.

6. 下列有 11 個說法和集合了任意五個說法的選項. 這些選項中的五個說法裡, 有且唯有一個說法是錯誤的有 (　　).

①在類別之外使用 = default 也會讓該特殊函式帶有內嵌屬性.

②catch 的參數列表中不可以出現右值參考.

③若一個類別物件帶有 const 標識, 那麼 const 只在類別建構完成之後才生效.

④靜態成員變數的型別可以是不完全型別.

⑤若不小心使用 new T [0] 配置了記憶體, 這塊記憶體仍然需要主動進行回收. 但是既可以使用 delete 來回收, 也可以使用 delete [] 來回收.

⑥複製建構子的第一個參數型別可以是類別的參考或者常數參考, 如果有其它參數, 則這些參數必須帶有預設引數.

⑦std::cerr 中的內容需要使用 std::flush 或者 std::endl 來刷新.

⑧所有可以多載的運算子中, 只有 =, [], () 和 -> 的多載必須是類別成員函式. 另外, 運算子多載中不能發明新的運算子.

⑨C++ 標準樣板程式庫中容器的成員函式 shrink_to_fit 只是發送一個請求, 不保證一定會回收配置的記憶體, 但是 reserve 就一定會回收配置的記憶體.

⑩使用 extern template 時, 若編碼器在任意可找到的檔案裡都無法找到對應的具現體, 編碼器並不會因此而擲出編碼錯誤或者連結錯誤.

⑪若某個類別樣板帶有型別參數 T, 那麼可以在類別中將 T 宣告為友誼類別, 即 friend T; 即使 T 是內建型別甚至 void 型別, 也不會產生編碼錯誤.

(A) ①④⑥⑩⑪　　**(B)** ②③④⑥⑧　　**(C)** ③⑤⑧⑨⑩　　**(D)** ②⑤⑥⑧⑪

(E) ①③⑨⑩⑪　　**(F)** ①②⑤⑦⑨　　**(G)** ④⑦⑧⑨⑩

7. 下列斷言中, 正確的有 (　　).

(A) 下列程式碼中, 函式 f 的宣告或者實作中的預設引數需要去掉其中一個, 否則會產生編碼錯誤. 如果將宣告中的預設引數去掉, 那麼最終的輸出結果為 1144005555.

```
struct X;
void f(X *p = nullptr);
struct X {
    X() {std::cout << 1;}
    X(X &) {std::cout << 2;}
    X(const X &) {std::cout << 3;}
    X(X &&) noexcept {std::cout << 4;}
    virtual ~X() {std::cout << 5;}
    virtual void f(int i = 0) {std::cout << 1 << i;}
};
struct Y : X {
    mutable X x;
    Y() = default;
    Y(const Y &) = default;
    ~Y() = default;
    Y &operator=(Y &) = delete;
    Y &operator=(Y &&) noexcept = delete;
    void f(int i = 1) override {std::cout << 0 << i;}
};
void f(X *p = nullptr) {p->f();}
int main() {
    Y x, y = std::move(x);
    X *p = &x;
    f(p);
}
```

(**B**) 下列程式碼中, 需要將 #2, #3, #5 和 #7 對應的整個陳述式移除或者註解, 否則會產生編碼錯誤. 在這四個陳述式被移除之後, 程式的最終輸出為 01100110011000110111.

```
struct d {
    d() {std::cout << 0;}
    d(const d &) {std::cout << 0;}
    d(d &&) noexcept {std::cout << 1;}
    ~d() {std::cout << 1;}
    d &&operator=(d rhs) noexcept {
        std::cout << "01";
        return std::move(rhs);
    }
    void operator()(void *) {std::cout << 0;}
    void operator()(void *) const {std::cout << 1;}
};
struct s {
    std::vector<std::unique_ptr<s, d>> v;
    s() {std::cout << 0;}
    explicit s(int) {
        v.emplace_back(nullptr, d {});
        std::cout << 0;
    }
    explicit s(int, int) {
        d deleter;
        v.emplace_back(this, deleter);
        std::cout << 1;
    }
};
int main(int argc, char *argv[]) {
    s s1(1.2);      // #1
    s s2 = {1};     // #2
    s s3 = {1, 2};      // #3
    s s4();     // #4
    s s5 = 1;       // #5
    const s &s6 {1};        // #6
    const s s7 = static_cast<s>(1);     // #7
    s s8 = static_cast<s>({1, 2});      // #8
    s s9 {1, 2};        //  #9

}
```

(**C**) 下列程式碼的輸出結果為 BA, 但是存在未定行為.

```
template <typename = void> struct s1;
struct s2 {
    s2() {std::cout << 'A';}
    friend s1<s2>;
};
template <typename>
struct s1 {
    s1() {std::cout << 'B';}
    const s2 &create_s2() && {return {};}
};
int main() {
    s1<char> x;
    s2 y = std::move(x).create_s2();
}
```

(**D**) 下列的程式碼中 : s1 的初始化並不會產生編碼錯誤, 因為 a 具有靜態生命週期; s2 的初始化會發生編碼錯誤, 因為 (const int *)&b 不是一個常數表達式; s3 的初始化不會導致編碼錯誤, 但是會產生未定行為; s4 的初始化不會產生編碼錯誤, 因為 d 是一個常數; s5 的初始化會產生編碼錯誤, 雖然 e 具有靜態生命週期, 但是它並不是常數.

```
template <const int *p>
struct s {int var = *p;};
constexpr int a {};
static constexpr char b {};
int c;
s<&a> s1;
s<(const int *)&b> s2;
s<&c> s3;
int main(int argc, char *argv[]) {
    const auto d = 0;
    static int e {};
    s<&d> s4;
    s<&e> s5;
}
```

(**E**) 下列程式碼不存在編碼錯誤, 最終的輸出為 AEFG0Befg1DD.

```
struct A {
    A() {std::cout << 'A';}
    A(const A &) {std::cout << 'B';}
    A(A &&) noexcept {std::cout << 'C';}
    ~A() {std::cout << 'D';}
};
struct B : virtual A {
    B() {std::cout << 'E';}
    B(const B &) {std::cout << 'e';}
};
struct C : virtual A {
    C() {std::cout << 'F';}
    C(const C &) {std::cout << 'f';}
};
struct D : virtual A {
    D() {std::cout << 'G';}
    D(const D &) {std::cout << 'g';}
};
struct E : virtual B, protected C, D, virtual A {
    E() : D(), B(), C(), A() {std::cout << 0;}
    E(const E &) {std::cout << 1;}
};
int main(int argc, char *argv[]) {
    E e1, e2(e1);
    const E &e3 {std::move(e2)};
    auto &&e4 {std::forward<decltype(e3) &&>(e3)};
}
```

(**F**) 若輸入為 "1 2 3 1.2 A" (引號不算作輸入), 那麼下列程式碼的輸出為 123.

```
struct A {
    void f(int i) {std::cout << i;}
};
struct B {
    void f(double i) {std::cout << i;}
};
class C : public A, protected B {};
int main(int argc, char *argv[]) {
    int a;
    while(std::cin >> a) {
        C {}.f(a);
    }
}
```

(**G**) 下列程式碼的輸出結果為 ABAc.

```
template <typename T>
void f(T ...) {std::cout << 'A';}
template <typename ...T, typename ...U>
void f(T ..., U ...) {std::cout << "B";}
void f(unsigned) {std::cout << "u";}
void f(int) {std::cout << 'i';}
void f(char) {std::cout << "c";}
void f(char ...) {std::cout << "s";}
int main(int argc, char *argv[]) {
    f(1ll);
    f('c', L'w', U'0');
    f(1.0, 1.2, 1.3);
    signed char x = 1;
    unsigned char y = 2;
    f(x + y);
}
```

8. 以下程式碼中, 可能或一定存在未定行為的有 (　　).

(**A**)
```
struct A {};
struct B : A {};
struct C : virtual B {
    virtual ~C() = default;
};
int main(int argc, char *argv[]) {
    A *p = new B();
    delete p;
    delete new C;
}
```

(**B**)
```
struct s {
    void *operator new(std::size_t n) noexcept(noexcept(true)) {
        return ::operator new(n);
    }
    void operator delete(void *p)
            noexcept(noexcept(::operator new(0))) {
        ::operator delete(p);
    }
};
```

其中, ::operator new 和 ::operator delete 來自標頭檔 <new>.

(**C**)
```
int *realloc(int *p, int size, int new_size) {
    delete p;
    auto q = new int[new_size] {std::forward<int>(*p)};
    std::memcpy(q, p, sizeof(int) * size);
    return q;
}
void set(int *&p, int size, int pos, int &value) {
    if(pos >= size) {
        p = realloc(p, size, pos + 1);
    }
    p[pos] = value;
}
```
(**D**)
```
const std::string &&str = 0;
```
(**E**)
```
struct base {
    virtual ~base() = default;
};
struct derived : base {};
base *b1 = new base;
base *b2 = new derived();
auto d1 = static_cast<derived *>(b2);
auto d2 = dynamic_cast<derived &>(*b1);
auto d3 = reinterpret_cast<derived *>((char *)b1);
```
其中, d3 不被使用.

(**F**)
```
std::map<std::string, std::ptrdiff_t> m;
m[""] = m.empty();
```
(**G**)
```
struct s {
    void f(std::list<std::shared_ptr<s>> l) {
        l.emplace_back(this);
    }
};
```

9. 以下程式碼中, 只存在兩處地方 (修復了一處, 要求另外一處仍然存在; 修復了一處, 另外一處關聯地不再存在, 這種情況不能算兩處) 可能或一定會引起編碼錯誤的有 (　　).

(**A**)
```
class B;
class A {
public:
    void f(B &);
};
class B {
    int a;
    const int c {};
    friend void A::f(B &b) {
        ++b.a--;
        b.a = std::move(c);
    }
}
```

(**B**)
```
template <typename T>
void destroy(T *p) {
    p->~T();
}
enum {e = -1, f};
void *p = f;
int main() {
    destroy(::p);
}
```

(**C**)
```
void f(int) {}
int main() {
    void f(int (&)[1]);
    int f(void (*)());
    f(1);
    auto i = f(nullptr);
    static auto si = f(NULL);
    struct s {
        static auto si_in_s;
        void f() {
            if(i not_eq 42) {
                throw i;
            }
            if(si not_eq 42) {
                throw si;
            }
        }
    };
    static auto s::si_in_s = si;
}
void f(int (&)[1]) {}
int f(void (*)()) {
    return 42;
}
```

(**D**)
```
struct s {
    bool operator==(int = 0) {
        return false;
    }
    void operator->() {}
};
operator int [42](s) {
    return {};
}
int main(int argc, char **argv) {
    s {}->;
    (void)s {}.operator->();
}
```

(**E**)
```
template <typename T>
struct base : T {
protected:
    int a;
public:
    friend T;
    base() = default;
};
template <typename T>
class derived : base<T> {
private:
    using base<T>::base;
public:
    constexpr derived() : base() {}
};
struct s {
    template <typename T>
    void change(T t) {t.a = 42;};
};
template <typename T>
void f(derived<T> d) {
T t;
t.change(d);
}
auto main() -> int {
    derived<s> d;
    f(d);
}
```

(**F**)
```
struct override;
struct final;
struct base {
    virtual override *f(bool) = 0;
};
struct derived : base {
    virtual auto f(bool b) -> override *override final {
        if(b) {
            try {
                throw static_cast<final *>(nullptr);
            }catch(const void *p) {
                return {};
            }
        }
        return {};
    }
};
```

(**G**)
```
namespace N {
    inline namespace N1 {
        struct S {};
    }
    namespace N1 {
        void f(S) throw(wchar_t) {}
    }
}
void f(N::N1::S, wchar_t) noexcept {}
auto main() -> decltype((0)) {
    N::S s;
    f(s);
}
```

10. 下列程式碼中, 最終的輸出結果為 01000111 (假設程式碼中的記憶體配置操作一定會成功) 且不存在任何未定行為的有 (　　).

(**A**)
```
namespace A {extern "C" int x;}
namespace B {extern "C" int x;}
int A::x = 0;
int f(int a = []() {static int b = 0; return b++;}()) {
    std::cout << a;
    return a;
}
std::vector<int> g(std::vector<int> v) {return v;}
struct derived : std::vector<int> {
    int i;
    derived() = default;
    derived(const derived &) {std::cout << 1;}
} d;
void h(std::initializer_list<derived>) {}
int main()
{
    auto v = g({f(), f()});
    try {
        throw v;
    }catch(std::vector<int> &) {
        std::cout << 0;
    }catch(derived &) {
        std::cout << 1;
    }
    const derived d;
    std::cout << d.i;
    std::cout << B::x;
    A::x=1;
    std::cout << B::x;
    int x {(v[0], v[1])};
    do {
        std::cout << x;
        --x;
        if(x > -1) continue;
    }while(false);
    auto l = {d};
    h(l);
    h(l);
}
```

(**B**)
```
struct A {
    A() {std::cout << 0;}
    void f() {
        static A a;
        std::cout << 1;
    }
};
struct B : public A {void f() {std::cout << 1;}};
void f(A &&a) {a.f();}
int main(int argc, char *argv[]) {
    extern volatile int a;
    auto i {std::numeric_limits<decltype(1ull)>::max()};
    std::cout << (--++(--++i += 1))++;
    std::priority_queue<int> pq1({}, std::vector<int>(1, 2));
    std::priority_queue<int> pq2({}, std::vector<int> {1, 2});
    std::cout << (pq1.size() not_eq pq2.size()) << a;
    f(B {});
    std::cout << [](const std::string &s) {
            return s[s.size()] == '\0';}(*argv);
    std::tuple<int, float, int> t {1, 1, 1};
    std::cout << std::get<int>(t);
}
volatile int a;
```

(**C**)
```
struct A {
    A() {this->g();}
    virtual std::ostream &f(std::ostream &os) const {
        return os << 1;
    }
    virtual void g() {std::cout << 0;}
    void h() {g();}
};
struct B : A {
    std::ostream &f(std::ostream &os) const {return os << 0;}
    void g() {std::cout << 1;}
};
template <typename T>
struct C {
    C(T) {std::cout << 1;}
    explicit C(T, T) {std::cout << 0;}
    explicit C(std::initializer_list<T>) {std::cout << 1;}
    template <typename U>
    operator C<U>() const noexcept {return (1, 2);}
};
std::ostream &operator<<(std::ostream &os, A &a) {
    std::cout << 2["010"];
    return a.f(os);
}
int main(int argc, char *argv[]) {
    B b;
    b.h();
    std::cout << b;
    C<int> c1 = C<char> {1};
    C<int> c2 {1, 2};
    std::list<C<int>> l {1, 2, {}};
}
```

(**D**)
```
struct s1 {
    int x;
    s1() = delete;
    ~s1() {std::cout << 1;}
    static std::unique_ptr<s1> alloc() {
        auto p {reinterpret_cast<s1 *>(operator new(sizeof(s1)))};
        p->x = 2;
        return std::unique_ptr<s1>(p);
    }
};
struct s2 {
    s2(const char *) {std::cout << 0;}
    template <std::size_t N>
    s2(const char (&)[N]) {std::cout << 1;}
    s2(const void *) {std::cout << 1;}
    s2(s1) {std::cout << 0;}
};
auto p = s1::alloc();
void f() {
    if(not ----p->x) {
        throw p->x;
    }
    std::cout << 0;
    static s2 s3("\"""""""\"");
    static s2 s4(*p);
}
int main(int argc, char *argv[]) {
    std::cout << s1 {}.x;
    try {
        f();
        f();
    }catch(int &r) {
        r = -2;
        f();
        std::cout << ++++++p->x;
    }
}
```

(**E**)
```
struct base {
        base() {
            std::cout << 0;
        }
        base(const base &) {std::cout << 1;}
        base(base &&) noexcept {std::cout << 0;}
        base(int i, int j = 1) {std::cout << j - i;}
        base &operator=(const base &) {
            std::cout << 0;
            return *this;
        }
        operator void() {
            std::cout << 1;
        }
        operator const void() {
            std::cout << 0;
        }
        operator bool() {
            return {};
        }
        virtual void f(int i = 0) {
            std::cout << i;
        }
    };
    base operator<<(const base &b, bool &&) {
        return b;
    }
    struct derived : base {
    protected:
        using base::base;
    public:
        derived() : base() {std::cout << 0;}
        derived(derived &d) {}
        derived(int i, int j = 0) : base(j) {}
        void f(int j = 1) {
            std::cout << j;
        }
    };
    int main(int argc, char *argv[]) {
        base b1, b2 = b1, b3();
        static_cast<const void>(b3);
        derived d1, d2 {d1};
        base b4 = std::move(d1) << noexcept(static_cast<char>(b1));
        base b5 = {b1 + d1};
        derived d3({});
    }
```

(**F**)
```
struct base {
    static constexpr const char &c = 0;
    base() {std::cout << 0;}
    base(const base &) {std::cout << 0;}
    base(base &&) noexcept {std::cout << 1;}
    virtual base *clone() const & {return ::new auto(*this);}
    virtual base *clone() && {return new auto(std::move(*this));}
    void f() {std::cout << 0;}
};
struct mid_derived_1 : virtual base {
    mid_derived_1() : base() {std::cout << 1;}
};
struct mid_derived_2 : virtual base {
    mid_derived_2() : base() {std::cout << 0;}
};
struct derived : mid_derived_1, mid_derived_2 {
    derived() : mid_derived_2(), mid_derived_1() {std::cout << 0;}
    virtual derived *clone() const & {return new auto(*this);}
    virtual derived *clone() && {
        return ::new auto(std::move(*this));
    }
};
struct s {
    s() {std::cout << 1;}
    s(const base &) {std::cout << 1;}
    void f() {std::cout << 1;}
};
using vector_type = std::vector<std::shared_ptr<base>> &;
template <typename T>
void add(vector_type v, const T &t) {v.emplace_back(t.clone());}
template <typename T>
void add(vector_type &v, T &&t) {
    v.emplace_back(std::forward<T>(t).clone());
}
int main(int argc, char *argv[]) {
    auto p = std::make_shared<derived>();
    enum E {e = 0xFFFFFFFFFFFFFFFF};
    std::cout << std::is_same<
                 std::underlying_type<E>::type, int
             >::value or std::is_same<
                 std::underlying_type<E>::type, long
             >::value or std::is_signed<
                 decltype(((p->c) + 0))
             >::value;
    std::vector<std::shared_ptr<base>> v;
    v.reserve(42);
    add(v, std::move(*p));
    [p](bool b) {return b ? *p : s {};}(true).f();
}
```

其中, std::is_same 接受兩個樣板型別引數 T 和 U, 只有 T 和 U 是同一個型別時, std::is_same<T, U>::value 的值才是 true. std::underlying_type 接受一個樣板型別引數 T, T 必須是列舉型別, typename std::underlying_type<T>::type 為列舉值對應的真實整型型別. std::is_signed 接受一個樣板型別引數 T, 若且唯若 T 為帶號數型別, std::is_signed<T>::value 的值才為 true.

(**G**)
```
int f() {return 1;}
struct s1 {static int x;
    static int f() {return 0;}
    void f(int &) {std::cout << 0;}
    void f(int &) const {std::cout << 0;}
    void f(int &&) {std::cout << 1;}
    void f(int &&) const {std::cout << 1;}};
int s1::x {f()};
struct s2 {typedef char char8_t;
    std::shared_ptr<s1> p1;
    s1 const *p2;
    s2() : p1 {new s1()}, p2 {p1.get()} {}
    explicit s2(int i) {std::cout << i;}
    s2(float f) {std::cout << f;}
    void f(char uc) {std::cout << uc;}
    void f(unsigned short us) {std::cout << us;}
    void f(std::string s) {std::cout << s;}
    void f(signed char c) {std::cout << c;}
    void f(signed char8_t) {std::cout << 1;}};
struct s3 {
    s3() {std::cout << 0;}
    s3(s3 &&) {std::cout << 1;}};
s3 g() {return {};}
void c(bool b, int i) {std::cout << (b xor i);}
template <typename T, typename U> void f(T &&t, U &&u) {u.p1->f(t);}
template <typename T, typename U>
void g(T &&t, U &&u) {std::move(u).p2->f(std::move(t));}
template <typename T, typename U>
void h(T &&t, U &&u) {std::forward<U>(u).p1->f(std::forward<T>(t));}
template <typename T, typename U>
void i(T &&t, U &&u) {std::move(u).p1->f(std::move(t));}
template <typename T, typename U>
void j(T &&t, U &&u) {std::forward<U>(u).p2->f(t);}
int main(int argc, char *argv[]) {
    s2 s;
    volatile const void *const volatile p = &p;
    c(p, 1);
    s.f("1");
    std::cout << s1::x;
    std::forward_list<char, std::allocator<char>> a {"1", "2"};
    std::cout << *(++++a.cbefore_begin()++);
    auto g0 {g()};
    const auto &r {s};
    auto i = 0;
    do {if(i < 0) {goto L;}
        switch(i ? ----i += 2 : ++++i -= 3) {
            case 0:{s2 s = 1;} g(i, s); h(i, r); break;
            case 1:::i(i, r); j(i, r); f(i, r);
            default: j(42, r); g((decltype((i)))i, r); ::i(42, s);
                break;
        } L : ++i;
        if(i < 3) {if(i > 0) {break;}}else {continue;}
    }while(false);
    s.f('1');
}
```

(二) 填空題. 11 ~ 15 題, 每小題 3 分, 共計 18 分. 對於多個答案的填空題, 錯填或者漏填不得分.

11. 設有程式碼

```
#1
char int;
int = 'a';
#2
```

顯然, 由於 int 是關鍵字, 上述程式碼會導致編碼錯誤. 請在 #1 和 #2 處添加程式碼, 使得上述程式碼在不更改的情況下可以通過編碼, 並且不影響其它程式碼 (第一行寫用於替換 #1 的程式碼, 第二行寫用於替換 #2 的程式碼) :

_____

_____.

12. 設有程式碼

```
struct s {
    template <typename T>
    volatile T *f(T &&) volatile && noexcept;
};
extern template volatile int *
s::f<int>(int &&) volatile && noexcept;
template <typename>
struct transform;
template <class T>
using result_type = typename transform<T>::type;
template <typename T>
struct transform {
    using type = transform<const typename std::add_pointer<T>::type
            (&&)[std::is_array<T>::value + 42]>;
};
template <typename R, typename Class, typename ...Args>
struct transform<R (Class::*)(Args...)> {
    using R_type = typename std::add_pointer<
            typename std::add_lvalue_reference<R>::type>::type;
    using type = result_type<R_type(Args &&...) &&>;
};
struct base {
    inline static result_type<decltype(&s::f<int>)>
    f(result_type<base>, ...);
};
struct derived : base {
    constexpr result_type<int *[42]>
    f(result_type<decltype(&base::f)>) volatile noexcept;
};
```

其中, std::is_array 是接受一個樣板型別引數 T 的類別樣板, 若 T 為陣列型別, 那麼 std::is_array<T>::value 的值為 true; 否則, std::is_array<T>::value 的值為 false. 要求不可使用編碼器進行推導 (包括但不限於 auto, decltype 以及 template), 並且將所有 result_type<T>, std::add_lvalue_reference<T> 和 std::add_pointer<T> 替換為對應的真實型別.

①（1 分）顯然，typename transform<const char &>::type 仍然是一個類別，請寫出該類別中 this 的型別：

_____.

②（2 分）寫出一函式指標 p，指向類別 derived 中的成員函式 f，並移除 p 對應型別中可有可無的 const 或者 volatile 限定：

_____

_____.

13. 現假設有一類別 c，其有一個公用且僅允許右值呼叫的成員函式 f，回傳型別為 void，沒有任何參數. 結合標頭檔 <functional>，寫出類別成員指標轉換為可呼叫物件的方法：

_____.

14. 若某個繼承而來的成員函式（非函式樣板）僅允許左值呼叫，它帶有任何成員函式可能帶有的標識. 請寫出函式名稱之前可能帶有的所有標識和函式名稱之後可能帶有的所有標識（標識僅考慮標準關鍵字和參考標識；第一行寫函式名稱之前的標識，第二行寫函式名稱之後的標識）：

_____

_____.

15. 設有函式宣告

```
struct s;
enum E {};
constexpr void f(void (s::*)());
void f(unsigned short) noexcept;
void f(const float);
void f(const decltype(nullptr) &&);
void f(void *);
void f(volatile unsigned &);
static void f(long long &&);
```

函式呼叫 f(static_cast<E>(0)) 不可能選中的函式有（若存在多個，那麼一行僅寫一個，一行寫多個不給分）：

_____

_____

_____

_____

_____

_____

_____.

(三) 論述題. 16 ~ 20 題, 每小題 2 分, 共計 10 分.

16.
```
/* iterator.hpp */
namespace D {
    namespace aux {
        template <typename T> class forward_list_iterator {#1};
    }
}

/* forward_list.hpp */
#include "iterator.hpp"
namespace D {
    template <typename T>
    class forward_list {
    public:
        using iterator = D::aux::forward_list_iterator<T>;
    };
}
```
為了能夠讓 D::forward_list 存取到 D::aux::forward_list_iterator 的私用成員, 某同學在 #1 中增加了一個友誼宣告 :
```
template <typename T> friend class forward_list;
```
但是在 D::forward_list 存取到 D::aux::forward_list_iterator 的私用成員的時候, 仍然會產生編碼錯誤. 請分析具體的原因並且修改錯誤.

17. ①（1 分）如果移動操作可能擲出例外情況, 那麼我們應該優先選用複製操作. 請分析具體原因.

②（1 分）自訂複製建構子, 複製指派運算子或者解構子的情況下, 編碼器會將未自訂的移動操作設為被刪除的函式; 自訂了移動建構子和移動指派運算子的情況下, 編碼器會將未自訂的複製操作設為被刪除的函式. 一個類別如果沒有某個特殊函式, 那麼衍生類別在未自訂的情況下, 對應的特殊函式會被編碼器設為被刪除的函式. 請分析編碼器這樣做的原因.

18. ①（1 分）請分析下列程式碼從編碼, 連結到運作的具體行為和結果, 並分析產生最後結果的原因.
```
struct B;
struct A {
    A() = default;
    A(const B &) {}
};
struct B {
    operator A() {return {};}
    operator A() const {return {};}
};
void f(const A &) {}
int main (int argc, char *argv[]) {
    B b;
    f(b);
}
```

②（1 分）請分析下列程式碼從編碼，連結到運作的具體行為和結果，並分析產生最後結果的原因．

```
namespace A {
    struct base {};
    void f(base *) {}
}
struct derived : A::base {};
int main() {
    derived d;
    f(&d);
}
```

19. ①（1 分）成員函式 operator new 和 operator delete 是隱含的 static 函式，請分析具體原因．

②（1 分）設有表達式 (void)&a->*f(++i, static_cast<char *>(0) + 42)，請判斷該表達式是否存在錯誤．如果存在錯誤，請修改為正確的表達式．除此之外，請論述各個名稱的含義，寫出一種正確的表達式運作順序．

20. ①（1 分）設有函式宣告 const int * f(void *const) noexcept; 請寫出所有可以產生 f 的可呼叫物件的不同方法．

②（1 分）在全域名稱空間或者名稱空間中宣告不具名等位時，必須為其標識 static 或者放入巢狀的不具名名稱空間中．請分析這條規定的原因．

（四）改錯題．21 ~ 23 題，每小題 5 分，共計 15 分．每個改錯題的錯誤包括但不限於編碼錯誤，未定行為，錯誤結果和不正確的實作，指出可能存在錯誤的地方並且在不刪除任何程式碼的情況下更改程式碼使得其可以正確運作．初始為 5 分，漏改一處扣 1 分，改錯一處扣 0.5 分，每小題得分不低於 0 分．

21.
```
namespace N {extern "C" {int x;}}
namespace M {extern "C" {int x;}}
enum E;
struct s {
    enum E {a, b, c};
    s() = default;
    s(s &) {}
    s(E) {}
};
s operator+(s &a, const s &b) {
    extern int c;
    if(&a < &b or c) {
        throw;
    }
    return s(s {});
}
s s1, s2;
auto s3 = {s1 + s2};
s s4 = {M::x};
volatile s i {};
const s &a {i};
volatile s &b {std::move(const_cast<s<char> &>(a))};
```

22.
```
/* 1.hpp */
namespace {char i1 {'l'};}
static char i2 {'o'};
template <typename T>
void print(const T &) {}
template <typename T>
class s {void f(extern "C" void (*)(extern "C" void (*)()) =
        nullptr);};

/* 1.cpp */
#include "1.hpp"
void print(const char *p) {std::cout << p;}
template <typename T>
void s<T>::f(extern "C" void (*)(extern "C" void (*)())) {
    extern "C" int x;
}
template <>
class s<int> {void g();};
template <>
void s<int>::g() {}
void s<char>::f() {}

/* main.cpp */
class linear_structure {
private:
    int *p;
    std::size_t s;
public:
    linear_structure(std::size_t s) : p {new int[s]}, s {s} {}
    void insert(const int &value) {
        auto new_p = new int[this->s + 1];
        auto p {new_p};
        for(auto c : this->p) {*p++ =c;}
        *p = value;
        delete this->p;
        this->p = new_p;
        ++this->s;
    }
};
int main(int argc, char *argv[]) {
    const auto &s1 {const_cast<const s<unsigned> &>(
            const_cast<const s<unsigned> &&>(s()))};
    decltype(s.f(nullptr)) p;
    (s1.*p(nullptr));
    const char str[] {'h', 'e', 'l', i1, i2};
    print(str);
    linear_structure vector;
    vector.insert(0);
}
```
其中, 主函式最終的輸出結果為 hello.

23.
```
void *operator new(std::size_t n) {return ::operator new(n);}
struct A {
    int size : 4;
private:
    union U {
    private:
        std::string s;
    public:
        std::string &get_s() noexcept {return this->s;}

    } u;
public:
    void *operator new(std::size_t n) noexcept {
        return std::operator new(n);
    }
    void operator delete(void *p) {::operator delete(p);}
    A(int) : u {} {std::vector<int> v(*&size + 42);}
    virtual ~A() {}
    void f() {
        this->s.~std::string();
        new (&s) std::string("final");
        if(s.empty()) {
            throw std::runtime_error("A::u.s is empty!");
        }
    }
};
struct B : virtual A {void f(int) {}};
struct C : virtual A {void f(int *) {}};
struct D : virtual A, B, C {
    using type = A;
    void f(int, int) {}
};
template <typename T>
struct E : D {
    constexpr E() = default;
    using type = T;
    void g() {this->f();}
};
template <typename>
struct traits {};
template <>
struct traits<E<int>::type> {traits() = delete;};
int main(int argc, char *argv[]) {
    D d;
    E<int> e;
    e.f();
    struct F : E<traits<int>> {
        struct G {void h() {e.f(nullptr);}};
    } f;
    A *p = nullptr;
    std::cout << typeid(*p).name();
    B *p2 = dynamic_cast<B *>(p);
    enum class E1 {e = 0xFFFFFFFFFFFFFFFF};
}
```

（五）程式設計題. 24 ~ 28 題，共計 40 分. 除了有額外要求的情形，要求全程自行設計，不可使用類似於 C++ 標準樣板程式庫等中的任何名稱.

24. 特性創造題.（4 分）① ~ ② 題，每題 2 分.

① 根據類別的繼承方式，嘗試設計 Lambda 表達式的繼承.

② 下列程式碼中部分特性是 C++ 11 未引入的，請指出這些特性並介紹這些特性的用處：

```
namespace A::B {
    struct s {};
    constexpr auto f = []<typename T>(T t) {};
}
int main(int argc, char *argv[]) {
    A::B::s s;
    f(s);
}
```

25. 物件導向設計題.（5 分）某上市公司（listed company）下設股東會（shareholders meeting），股東會下設稽核委員會（audit committee，負責監督，成員包括五名監察董事和部分股東），選舉委員會（election commission，負責董事局的選舉活動，成員包含所有董事和部分股東，股東不和稽核委員會中的股東重合）和董事局（board of directors，負責公司管理和經營）. 董事局有一個領頭人為董事長（chairman，主管董事局和公司），一名常務董事（executive director），五名監察董事（supervisory director，參與決策和監督）和若干非常務董事（non-executive director，參與決策）構成. 董事長對公司事務有一票否決權（one-vote veto），常務董事負責公司日常經營（management）和運作（operation），且直接對董事長負責（responsibility）. 常務董事直接管理產品與運營處（product and management department），科技處（science and technology department），行政處（administrative department），財務與審計處（finance and auditing department）和研究與理論中心（research and theory center）. 產品與運營處繼承了銷售與售後科（sales and after-sales office），工程科（engineering office），研發科（research and development office），市場科（marketing office），品質保障科（quality assurance office）和國際事務科（international affairs office）. 科技處繼承了硬件科（hardware office），軟件科（software office），資訊分析科（information analysis office）和研發科. 行政處繼承了人力科（human resources office），公共關係科（public relations office），內部調查科（internal investigation office），客戶關懷科（customer care office），秘書辦公室（secretary office），國際事務科，特別事務科（special affairs office），聯合資訊科，法務辦公室（legal office）和安全保衛科（security office）. 財務與審計處積成了國際事務辦公室和成本控制辦公室（cost control office）. 除此之外，財務審計處還負責薪資發放（payroll），財務審計（financial audit）和原材料和日常用品採購（procurement of raw materials and daily necessities）工作. 研究與理論中心繼承了研發科. 其中，工程科同時受到產品運營處和科技處管理，資料分析科（聯合資訊科）同時受到科技處和行政處管理，國際事務科（國際事務辦公室）同時受到產品與運營處，財務處和行政處管理. 理論上，各科室和各辦公室獨立運作（work independently），但是各處和中心有權直接干預（intervene）各科室和各辦公室的所有運作. 另外，特別事務科和安全保衛科除了受到日常領導之外，直接聽命（order）於董事長. 請根據上述描述，設計各類別. 各科室和辦公室日常工作和活動自訂（工作和活動的設計不少於兩個），函式內可以使用一個輸出陳述式替換，不可以出現僅僅宣告而不實作的函式. 可以使用的 C++ 標準樣板程式庫設施為 std::cout 和 std::endl.

26. 資料結構實作題.（6 分）前向連結串列是程式設計中除了陣列之外，使用最多的線性資料結構，std::forward_list 的底層資料結構便是前向連結串列. 它是由若干個結點構成，每一個結點連接著下一個結點，最後一個結點連結到第一個結點上，形成一個封閉的環狀結構. 每一個結點除了負責連結到下一個結點之外，結點中還儲存了一個元素，例如 1->2->3->4->5->1（5 連結到的 1 是第一個結點中儲存的 1). 若前向連結串列的結點結構為

```
template <typename T>
struct forward_list_node {
    T value;
    forward_list_node *next;
};      // forward_list_node 的結構為 value->..., next 代表了連結符號 ->
```

請根據上面的提示，實作類別（只能向類別中添加權限為 private 的靜態成員函式）：

```
template <typename T>
class forward_list final {
private:
    forward_list_node<T> *first;
    using initializer = std::initializer_list<T>;
public:
    constexpr forward_list() noexcept;
    explicit forward_list(std::size_t n,
            const T &value = {});                              // #1
    forward_list(const T *begin, const T *end);                // #2
    forward_list(initializer init_list);                       // #3
    forward_list(const forward_list &rhs);
    forward_list(forward_list &&rhs) noexcept;
    ~forward_list();
    forward_list &operator=(const forward_list &rhs);
    forward_list &operator=(forward_list &&rhs) noexcept;
    forward_list &operator=(initializer init_list);            // #4
    std::size_t size() const noexcept;                         // #5
    template <typename ...Args>
    void insert(std::size_t pos, std::size_t n,
            Args &&...args);                                   // #6
    void insert(std::size_t pos,
            const T *begin, const T *end);                     // #7
    void insert(std::size_t pos, initializer init_list);       // #8
    void erase(std::size_t pos) noexcept;                      // #9
    template <typename U>
    void remove(const U &value) noexcept;                      // #10
    void merge(forward_list &rhs) noexcept;                    // #11
    void reverse() noexcept;                                   // #12
};
```

其中，#1 表示建構一個具有 n 個結點，各結點元素都為 value 的前向連結串列；#2 表示建構一系列結點，各結點元素按順序和 $[\mathrm{begin}, \mathrm{end})$ 內的元素相同；#3 表示建構一系列結點，各結點元素按順序和初始化列表內的元素相同；#4 表示將前向連結串列中各結點元素按順序修改為和初始化列表內的元素相同；#5 回傳前向連結串列內有多少個結點；#6 表示向第 pos 個位置插入 n 個 value，value 的值由引數包 args 建構；#7 表示向第 pos 個位置插入一系列值，這些值按順序和 $[\mathrm{begin}, \mathrm{end})$ 內的值相同；#8 表示向第 pos 個位置插入一系列元素，這些元素按順序和初始化列表內的元素相同；#9 表示移除第 pos 個位置對應的結點；#10 表示移除前向連結串列內所有元素為 value 的結點；#11 表示合併另一個前向連結串列到 *this；#12 表示反轉前向連結串列（例如 1->2->3->1 反轉為 3->2->1->3）；另外，pos 從 0 開始，forward_list 的樣板參數 T 支援 == 比較操作，且不會擲出例外情況.

27. 過程導向設計題.（11 分）① ~ ⑥ 題. 本題要求使用過程導向的程式設計方式進行實作，使用其它設計方式不得分. 其中，除第 ① 小題之外，剩餘題目不可以使用遞迴的方式實作.

① **遞迴設計題.**（2 分）一個袋子裡面有 2^k 個正方體盒子（$k > 1$），這些盒子的外觀和大小完全相同. 其中有一個盒子的重量非常大，裡面裝的是金塊，比剩餘 $2^k - 1$ 個盒子的重量之和還要大. 有一個類似於天秤的重量比較器，比較器的兩側至多可以放置 2^{k-1} 個盒子. 比較器的功能是計算哪一側的質量較大，但是無法顯示各個盒子具體的重量. 若令型別 box 代表盒子，重量比較器的函式被宣告為多載的 operator+ :

```
bool operator<(const std::tuple<box, box, ...> &a,
               const std::tuple<box, box, ...> &b) noexcept;
```

如果比較器的左側 a 比較重，那麼 a < b 回傳 false; 否則，回傳 true. 函式 split 有能力將 std::tuple<box, box, ...> 從

$$\frac{\texttt{std::tuple\_size<std::tuple<box, box, ...>>::value}}{2}$$

處一分為二（分割處參考尾後疊代器），其宣告如下 :

```
std::pair<std::tuple<box, box, ...>, std::tuple<box, box, ...>>
split(const std::tuple<box, box, ...> &) noexcept;
```

請使用遞迴設計一個函式，至多使用 $k - 1$ 次比較，找到那個裝有金塊的盒子. 其中，函式的宣告如下 :

```
template <typename T>
int find(const T &) noexcept;
```

其中，T 必定是類別樣板 std::tuple 的具現體，其樣板引數為若干個盒子 box，例如 std::tuple <box, box, ...>，回傳金塊在哪一個盒子中. 可以使用的 C++ 標準樣板程式庫設施為 std::tuple, std::tuple_size 和 std::pair.

② **數學設計題.**（3 分）我們用 $\int_a^b \sqrt{f(x)}\mathrm{d}x$ 來表示連續實值函數 $y = \sqrt{f(x)}$ 在區間 $[a, b]$ 上的定積分，其幾何意義為區間 $[a, b]$ 上 $y = \sqrt{f(x)}$ 的曲線和兩個座標軸圍成的封閉圖形（不一定是規則圖形）的面積. 其近似的計算方法是將 $[a, b]$ 等分成 n 份，$\left[a, \frac{1}{n}(b-a)\right]$, $\left[\frac{1}{n}(b-a), \frac{2}{n}(b-a)\right]$, ..., $\left[\frac{n-2}{n}(b-a), \frac{2}{n-1}(b-a)\right]$, $\left[\frac{n-1}{n}(b-a), b\right]$，每一個區間的長度為 $\frac{b-a}{n}$，最後用

$$\int_a^b \sqrt{f(x)}\mathrm{d}x = \sum_{i=1}^{n} \sqrt{f\left(\frac{i}{n}(b-a)\right)} \cdot \frac{b-a}{n}$$

來計算定積分的值. 設函數 $f(x)$ 的函式宣告為

```
long double f(long double x);
```

請設計函式

```
long double definite_integral(double a, double b, unsigned long n);
```

來求定積分 $\int_a^b \sqrt{f(x)}\mathrm{d}x$ 的值. 其中，函式參數中的 a 表示區間左側端點 a 的值，b 表示區間右側端點 b 的值，n 表示將區間等分成 n 份. 另外，$\sqrt{f(x)}$ 的值可以使用遞迴方程式

$$x_m = \begin{cases} \frac{1}{2}\left(1 + f(x)\right) & m = 1 \\ \frac{1}{2}\left(x_m + \frac{f(x)}{x_m}\right) & m > 1 \end{cases}$$

近似求得. 函式 definite_integral 對函式 $\sqrt{f(x)}$ 的精度要求為 $x_{m+1} - x_m < 10^{-8}$.

③ **STL 設計題.**（2 分）某學校針對 2022 級學生（500 人）在新學期開展了十門選修課程，這些課程有固定的課程名稱，選課人數（45 人），候補人數，上課時間，上課地點（普通課室至多容納 50 人），授課教師和課程介紹．候補人數是指在選課人數已滿之後，學生仍然可以選擇該課程，後續若有學生退課，則按順序替補．一個學生至多選擇一門課程和兩門候補課程．學校會進行兩次選課活動．在第一次選課活動中，若學生沒有搶到自己滿意的課程，可以僅選擇兩門候補課程等待候補（不能僅選擇一門候補課程）．若一門課程在第一次選課後的選課人數不超過封頂人數的 20%，那麼學校會決定取消該課程本學期的開課資格．學校會在第二次選課中取消至多兩門選課人數最少的課程，並且開展兩門課程，其中一門課程是候補人數最多的那一門，另一門課程從未取消的課程中隨機選擇一門．這兩門新開的課程雖然課程內容完全相同，但是授課教師和上課地點完全不同．第二次選課活動，第一次未成功選課的學生必須選擇一門課程和至少一門候補課程．在進行兩次選課活動之後，未成功選課的人數至少有 55 人．若未成功選課的人數不多於 60 人，那麼由學校統一安排到大課室（至多容納 60 人），從取消的課程中隨機選擇一門開課．若沒有課程被取消，那麼從十門課程中隨機選擇一門開課．若未成功選課人數大於 60 人，則隨機選擇 60 人按照學校統一安排開課．剩下的人隨機分配到其它課程中．在隨機分配的過程中，要求課程人數不超過普通課室可容納的最多人數．本題可以任意使用所有 C++ 標準樣板程式庫設施，並且任意值都可以自由設定（例如課程名稱和選課人數等）．

⑤ **程式碼翻譯題.**（1 分）Lambda 表達式本質上是一個帶有多載函式呼叫運算子的類別．編碼器在為 Lambda 表達式生成類別的時候，類別名稱為 __lambda_ 加上 Lambda 表達式的名稱．請根據上面的信息，將下列程式碼中的 Lambda 表達式 lambda_1 和 lambda_2 翻譯為類別，並寫出 lambda_2 的呼叫結果：

```
    constexpr bool b {};
    template <typename T>
    void f(const T &);
    int main(int argc, char *argv[]) {
        int a {1};
        static char c {'c'};
        const auto lambda_1 = [=](int a, bool b, char c) noexcept {
            f(a);
            f(b);
            f(c);
            a = 42;
            return static_cast<float>(a);
        }(*&a, b, c);
        auto lambda_2 = [&]() mutable noexcept {
            c = 'C';
            std::cout << c;
            return a + b +c;
        }();
    }
```

⑥ **閱讀設計題.**（3 分）桌子上有若干個相同的大盒子（box), A 同學和 B 同學手上分別持有若干個大小不同的小盒子（object). 隨機從這兩個同學中選擇一個人, 將他們手上的盒子放入大盒子. A 同學的策略是優先選擇空的大盒子, 因為小盒子一定可以被放入空的大盒子中. B 同學的策略是優先選擇排在前面且可以容納小盒子的大盒子, 因為這樣可以讓大盒子的利用率最大化. 然而, 小盒子有個很特殊的特點, 就是一旦某個大盒子沒有足夠空間容納小盒子, 而小盒子又被強行放入, 那麼新放入的小盒子會壓壞掉原本被放置在大盒子中的小盒子, 導致某些小盒子變形. 請設計函式

```
void put_for_A(object &);       // 模擬同學 A
void put_for_B(object &);       // 模擬同學 B
```

來模擬同學 A 和 B 的放置行為, 要求小盒子不能被壓壞. 大盒子類別 box 對外公開的部分為 :

```
class box {
public:
    bool contain(object &o) noexcept;
    void emplace(object &o);
    int object_size() const noexcept;
    int rest() const noexcept;
    constexpr int capacity() const noexcept;
    object &take_out(int k) noexcept;
};
```

其中, contain 用於檢測當前大盒子是否可以容納小盒子 o, emplace 用於將小盒子 o 放入當前大盒子中, object_size 表示當前大盒子中容納了多少個小盒子, rest 用於回傳當前大盒子還剩下多少容量, capacity 用於回傳大盒子的總容量, take_out 表示從大盒子中取出第 k 個小盒子. 小盒子類別 object 對外公開的部分為 :

```
class object {
public:
    int capacity() const noexcept;
};
```

其中, capacity 用於回傳當前小盒子的體積大小, 即放入大盒子之後所佔用的容量. 全域變數 box b[n]; 表示總共有 n 個相同的大盒子. 在 A 同學或者 B 同學放置完成之後, C 同學有 20% 的機率對大盒子中的小盒子進行整理. C 同學的策略是檢查每一個還有容量的非空大盒子 c, 然後從其它還有容量的非空大盒子中找到合適的小盒子將 c 填滿. C 同學的目標是盡量將非空的大盒子利用率最大化, 並留出儘量多的空大盒子. 請在主函式中模擬 100 次 A 同學和 B 同學的放置, 並模擬 C 同學的行為. 可以使用的 C++ 標準樣板程式庫設施為 std::default_random_engine, std::uniform_int_distribution 和 std::time.

28. 記憶體控制題.（12 分）① ~ ② 題. 某伺服器中有一個共享內存的程式 Shared Memory, 該程式負責處理接收到的共享內存區域的存取與寫入操作. 共享內存是指伺服器中的某部分記憶體由各程式共享, 每個程式都可以向 Shared Memory 請求某個共享內存區域, 對該區域進行存取和寫入. 一般來說, Shared Memory 運作得不錯. 但是一旦兩個程式同時對同一區域的共享內存請求寫入, 或者一個程式請求寫入另一個程式恰好又在同時請求存取, 就會造成資料被破壞的嚴重後果. 例如, 兩個程式 A 和 B 同時請求寫入, A 比較快先寫完了 B 才開始寫入, 這個時候 B 會完全將 A 寫入的部分覆蓋掉, 這種情況至少寫入的資料不會被破壞. 如果 A 寫了一部分然後就去運作別的程式碼片段, 這個時候 B 的寫入覆蓋了 A 之前的部分. 當 B 寫入完成之後, A 運作完剛剛的程式碼片段, 從暫停的位置繼續寫入. 最終的資料就由 B 寫入的前半部分和 A 寫入的後半部分組成, 這個時候寫入的資料就會被破壞了. 類似地, 若 A 正在寫入但是還沒寫完, B 要存取, 最終導致 B 得到的資料的前半部分是 A 寫入的新資料, 後半部分是舊的資料. 程式 Shared Memory 必須要避免類似情況的發生, 當 A 正在請求向某個共享內存區域寫入的時候, B 恰好也請求存取或者寫入資料到這塊記憶體, 那麼 Shared Memory 必須等待 A 寫入完成之後, 才能向 B 提供這段記憶體的存取或者寫入權限. 程式 Shared Memory 主要由 shared_memory 類別驅動, 其實作要求和各個成員函式的功能如下 :

- 為了避免頻繁的記憶體配置, 程式 Shared Memory 在啟動時會提前配置 128 塊大小為 128 位元組的共享內存區域. 其中, 共享內存區域 id 從 0 開始. 另外, 由於 Shared Memory 無法判定某塊共享內存區域是否需要回收, 所以回收工作統一在 Shared Memory 程式終結時進行;
- write 負責向編號為 id 的共享內存區域寫入資料, 資料來源於 source, 長度為 size;
- read 負責回傳編號為 id 的共享內存區域, 並提供該共享內存區域的大小;
- get 負責回傳編號為 id 的共享內存區域和該共享內存區域的大小, 該函式會將區域的控制權完全交給呼叫者;
- revert 負責歸還編號為 id 的共享內存區域控制權;
- allocate 負責配置新的共享內存區域, 並回傳這段記憶體和對應的編號, 該成員函式應該有兩個版本, 一個可能會擲出例外情況, 另外一個保證不擲出任何例外情況;
- empty 負責回傳大小為 size 且未使用的共享內存區域和對應的 id, 如果找不到沒有對應大小的共享內存區域, 那麼首先應該尋找是否有大小大於 size 且未使用的共享內存區域並回傳, 找不到時才呼叫成員函式 allocate 配置新的共享內存區域, 當 size 為 0 時, 回傳任意一個未使用過的共享內存區域, 找不到時就呼叫成員函式 allocate 配置大小為 128 位元組的新共享內存區域並回傳;
- 成員函式 set 用於判斷編號為 id 的共享內存區域是否被寫入過;
- 成員函式 lock 用於判斷編號為 id 的共享內存區域是否正在被其它程式寫入或者控制權是否已經交出;
- shared_memory 可以被複製或者移動, 但是共享內存區域不應該發生任何改變, 也不應該在複製或者移動的過程中配置新的共享內存區域;
- 可以使用的 C++ 標準樣板程式庫設施為 std::bad_alloc, std::nothrow_t, std::memcpy, std::memset, std::pair, std::ofstream, std::clog, std::flush;

①（11 分）請根據上述要求實作類別 shared_memory.

②（1 分）設有一函式

```
template <typename O, typename MF, typename ...Args>
void *time_limited_caller(int time, O &&o, MF &&f, Args &&...args);
```

用於計算成員函式的運作時間, 如果超過限定時間 time, 那麼 time_limited_caller 會擲出例外情況, 並且中斷本次函式呼叫. 現有一函式 f 使用了 time_limited_caller :

```
void f(shared_memory &shmem, int id) {
    auto mem = time_limited_caller(shmem, &shared_memory::get, id);
    // ...
}
```

為了避免例外情況導致程式異常被中斷, 必須捕獲 f 擲出的任何例外情況, 並使用 std::cerr 將日誌信息 "時間 函式名稱\n" 輸出到檔案 /var/log/shared_memory/exception.log 中, 最後刷新輸出流緩衝區. 請在不更改函式 f 內部任何程式碼的情況下, 實現該需求. 時間由保證不擲出例外情況的無參數函式 now 提供, 其回傳值支援被 std::basic_ostream 輸出.

2022 年 C++ 程式设计語言等級考試

封頁

考生注意

考試開始前考生禁止以任何方式打開試題冊