2021 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. 本試題考察的內容是 (　　).
(**A**) C++　　(**B**) D　　(**C**) Erlang　　(**D**) Groovy
(**E**) Fortran　　(**F**) Haskell　　(**G**) Lisp

2. 以下 C++ 關鍵字存在多個用途的有 (　　).
(**A**) thread_local　　(**B**) mutable　　(**C**) namespace
(**D**) static　　(**E**) compl　　(**F**) constexpr　　(**G**) extern

3. 設局域變數 p 被宣告為 constexpr void *p[] {nullptr, NULL, 0}; 若下列選項中的所有變數和 p 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**) 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. 設局域變數 fp 被宣告為 extern void (*fp[])(); 若下列選項中的所有變數和 fp 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 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. 下列說法正確的有 (　　).
(**A**) delete[] 運算子按照從後往前的順序進行解構的.
(**B**) 在樣板參數列表中, 使用 typename 或 class 是相同的.
(**C**) 設變數 p 和 q 分別被宣告為 auto p = {0}; int q = {0}; 那麼 p 和 q 的型別相同.
(**D**) 可以使用 std::forward_list 作為 std::queue 或者 std::stack 的底層容器.
(**E**) 名稱空間可以僅宣告並且可以通過 using namespace 一次性引入多個名稱空間.
(**F**) explicit 可以對任意建構子進行標識.
(**G**) 可以宣告一個函式型別的參考.

6. 本質與表現可以幫助我們分析事物的內在特性和外在表現. 所謂本質, 即事物的根本屬性, 而表現則是本質的外在表面特徵和外部聯繫. 對於同一類事物來說, 本質和表現存在互異性 : 本質在這一類事物上體現為普遍性, 表現則體現為個性. 但從另一方面而言, 本質和表現在同一類事物上又是互不可分的, 所謂透過現象看本質就是這個道理. 本質決定了表現的形式, 而表現又體現出事物內在的本質. 以下 C++ 特性中, 能夠體現本質與表現這一對概念的有 (　　).

(**A**) 使用 Range-for 尋訪容器和使用疊代器尋訪容器.

(**B**) int & 和 int *.

(**C**) struct 和 class.

(**D**) 初始化列表和陣列.

(**E**) long 和 long long.

(**F**) lambda 表達式和多載了函式呼叫運算子的類別.

(**G**) constexpr 和 const.

7. 下列斷言中, 正確的有 (　　).

(**A**) 下列程式碼的輸出結果為 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**) 下列程式碼無法通過編碼器編碼.

```
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**) 下列程式碼的輸出結果為 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**) 下列程式碼的輸出結果為 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**) 下列程式碼的輸出結果為 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**) 下列程式碼的輸出結果為 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**) 下列程式碼的輸出結果為 11010.

```
char str[] {"0"};
struct s {
    s() noexcept {*str = '1';}
    ~s() noexcept {*str = '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. 以下程式碼中, 可能或一定存在未定行為的有 (　　).

(**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, 函式 main 是程式入口.

9. 以下程式碼中, 可能或一定會引起編碼錯誤的有 (　　).

(**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, 類別 s2 直到編碼為止, 從未被使用.

(**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. 下列程式碼中, 最終的輸出結果為 11010011 且不存在任何未定行為的有 (　　).

(**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 是一個接受兩個樣板引數的類別樣板. 不妨記接受的樣板引數為 T 和 U, 若 T 和 U 是同一個型別, 那麼 std::is_same<T, U>::value 的值為 true; 否則, std::is_same<T, U>::value 的值為 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. 設有類別

```
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);
};
```

現在需求變更, 要求類別 base 中所有 private 成員都更正為 public 成員. 請在不更改上述程式碼和不計後果的情況下給出解決方案, 並且明確解決方案應該添加於何處 :

_____.

12. 設有程式碼

```
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;
};
```

寫出一指標 p 指向類別 derived 中多載的函式呼叫運算子, 不可使用編碼器进行推導 (包括但不限於 auto, decltype 以及 template). 其中, std::is_void 是接受一個樣板型別引數 T 的類別樣板, 若 T 為 void 系列型別, 那麼 std::is_void<T>::value 的值為 true; 否則, std::is_void<T>::value 為 false. 除此之外, 請移除 p 對應型別中可有可無的 const 或者 volatile 限定, 並將所有 result<T> 替換為對應的真實型別 :

_____.

13. 寫出可能會發生參考折疊的情形 **:**

_____.

14. 設有一類別樣板被宣告為

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

其作用為對函式進行包裝並延遲函式的呼叫, 以達到懶惰呼叫的目的. 其中, Arg 表示函式接受的引數對應型別, F 表示函式的型別. 現要針對函式

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

使用類別樣板 function_wrapper 進行包裝. 在進行包裝之前, 若要首先對類別樣板 function_wrapper 針對函式 f 進行樣板具現化, 使得類別樣板 function_wrapper 針對函式 f 有對應的具現體, 則需要寫出的宣告式是 (將 std::add_lvalue_reference 替換為真實的型別) **:**

_____.

15. 設有函式宣告

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

函式呼叫 f(NULL) 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. ①（1 分）簡述下列程式碼可能導致未定行為的原因, 並且在不更改函式型別的情況下進行改進 :

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

②（1 分）簡述陣列注標表達式 vector<int> {1, 2, 3, 4, 5, 6, 7}[3] 不會導致未定行為的原因.

17. 簡述下列程式碼並不存在編碼錯誤的原因, 並分析程式碼最終的輸出結果 :

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

18. 下列程式碼的輸出結果並非 −3, 請分析原因並且更改程式碼使得其輸出結果為 −3.

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

19. 設有程式碼

```
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]);
    }
}
```

分析陳述式 ++it, static_cast<void>(++i), 並說明在各種情形下該程式碼的必要性.

20. ①（1 分）分析當移動建構子或者移動指派運算子僅被實作其中任意一個時, 另外一個會被編碼器宣告為被刪除的函式的原因.

②（1 分）結合移動語意, 分析 C++ 11 標準遺棄 "已經自訂其中任意一個複製操作或者解構子的情況下, 仍然由編碼器生成另一個複製操作" 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 2 分, 每小題得分不低於 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;
};
```

(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. 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.

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

②constexpr 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. 物件導向設計題. (6 分) 交通工具 (transportation) 是人類 (human) 智慧 (wisdom) 的結晶之一. 其總共有幾大類 : 自行車 (bicycle), 汽車 (car), 船 (ferry) 和飛機 (aircraft). 自行車可以騎行 (cycle), 輪胎 (tire) 需要打氣 (inflate). 通過給自行車進行改裝 (refit), 就有了電單車 (motorcycle), 但它需要充電 (charge). 汽車可以駕駛 (drive) 但是需要加油 (oil). 現代某些電動汽車不需要加油但是需要充電. 船需要航行 (sail) 也可以漂浮 (float) 在水 (water) 面上, 船上必須具備導航 (navigation). 電影 (movie) 中經常出現的水陸兩用車 (amphibious vehicle) 吸收了船和汽車的共同特點. 自行車、汽車和船都應該配備喇叭 (horn). 飛機可以飛行 (fly), 飛機上也必須具備導航. 為了安全起見, 飛機同樣具備在水上漂浮的能力 (ability). 直升飛機 (helicopter) 具有飛機沒有的特點, 即懸停 (hover) 在空中, 但是直升飛機通常不能在水上漂浮. 除此之外, 船和飛機都需要加油. 飛船結合了船和飛機的特點, 但是它使用特殊燃料 (special fuel) 和太陽能 (solar energy). 請根據上述描述和下列要求, 設計類別 :

①可以使用的 C++ 標準樣板程式庫設施為 std::cout 和 std::endl;

②每一個動作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.

26. 資料結構實作題.（6 分）雜湊表（hash table）是 C++ 標準樣板程式庫中 std::unordered_map, std::unordered_multimap, std::unordered_set 和 std::unordered_multiset 的底層資料結構. 雜湊表一般通過前向連結串列的陣列來實作. 對於暫時擁有 n 個前向連結串列陣列的雜湊表中, 通過一個影射函式 hashing 將給定的引數影射為 [0, n] 範圍內的數字, 這個數字代表了它應該位於第幾個前向連結串列中. 任意一個前向連結串列陣列中的前向連結串列還有一個特殊的名字, 稱為桶（bucket）或者槽（slot）. 要在雜湊表中搜尋某一個元素, 只需要通過影射函式將這個元素影射為索引, 然後在對應位置的前向連結串列中搜尋即可. 這樣的搜尋十分高效. 請使用 class 實作一個雜湊表, 它至少滿足元素可以被插入（insert）, 元素可以被移除（erase）, 元素可以被放置（emplace）, 雜湊表的初始化（要求雜湊表物件的預設初始化擁有編碼期計算的能力）, 判定雜湊表是否為空（empty）, 雜湊表的清空（clear）, 元素的搜尋（contains）, 元素總數的統計（size）, 桶總數的統計（bucket_count）. 除此之外, 雜湊表應該接受重複元素的插入. 可以使用的 C++ 標準樣板程式庫設施為 std::forward_list, 但不能宣告接受前向連結串列的前向連結串列, 即 std::forward_list<std::forward_list<...>>. 其中, 影射函式 hashing 被宣告為

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

27. 過程導向設計題.（10 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）分而治之是一種解決問題的有效方案. 其基本思想是當問題足夠簡單的時候, 這個問題可以在瞬間被解決; 當問題有一定規模的時候, 我們可以通過不斷將問題劃分為若干個簡單子問題來求解. 例如要在若干個元素中尋找最小的元素, 當元素總數僅有兩個的時候, 程式可以在一瞬間找到那個最小的元素; 否則需要比較若干次. 現在利用分而治之的思想將元素不斷分組, 直到每一組要比較的元素只剩下兩個為止, 從中選出比較小的元素. 這樣, 我們得到了若干個最小元素的備選. 對於最小元素的備選集合, 我們運用同樣的方法, 直到備選集合中僅剩下兩個元素, 從中選出最小元素即可. 請依據上述思想, 使用遞迴實作下列函式, 其作用是從若干個元素中尋找最大的元素. 其中, 函式的宣告為 :

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

假設型別包 Args 中的任意兩個型別都至少可以進行基於小於運算子 "<" 的比較.

② **數學設計題.**（2 分）對於一個二階常係數齊次線性遞迴方程式 $f(n) = a_1 f(n-1) + a_2 f(n-2)$, 其有兩個初始條件 $f(0) = 0, f(1) = 1$. 我們通常假設 $f_h(x,n) = A x^n$ 是其解, 將 $f_h(x,n)$ 代入遞迴方程式可得到其特徵方程式 $x^2 = a_1 x + a_2$. 僅考慮實數解 $x = x_1, x = x_2$. 若 $x_1 \neq x_2$, 則原遞迴方程式的通解為 $f_h(n) = c_1 x_1^n + c_2 x_2^n$; 若 $x_1 = x_2$, 則原遞迴方程式的通解為 $f_h(n) = c_1 x_1^n + c_2 n x_2^n$. 其中, c_1, c_2 為常數, 代入初始條件可以得到具體值. 請設計一程式用於求解二階常係數齊次線性遞迴方程式.

③ **STL 設計題.**（1 分）某商場有若干個商品（至少需要添加 10 個商品）, 這些商品有對應的價格和庫存. 商場在進貨的時候, 首先檢查對該商品是否需要進貨, 然後檢查庫存, 按庫存進行補貨, 且商場不定時引入全新的商品. 當所有商品總庫存低於一定值, 商場會推出清倉特價, 所有商品都能以八五折購入. 商場希望在一次清倉完成之後, 按照某一個特定時間內商品的銷量進行排序, 以商榷是否需要加倍入貨. 本題可以任意使用所有 C++ 標準樣板程式庫的設施, 並且任意值都可以自由設定（例如庫存和價格等）.

④ **跨檔案設計題.**（1 分）在 C++ 中，若存在多個變數跨越多個檔案，那麼這些變數的初始化順序是未知的，它最終由編碼器來決定. 例如在 1.cpp 檔案中，變數 a 被宣告為 int a = 0; 而在 2.cpp 檔案中，變數 b 被宣告為 int b = 2; 若在 main.cpp 中同時引入變數 a 和 b. 此時，變數 a 和 b 在 main.cpp 中僅僅被宣告，並不知道它們是否已經被初始化. 若在沒有被初始化的情況下使用變數 a 和 b 將會導致未定行為. 請設計一方案以解決上述問題，不同檔案可以直接使用單獨一行的註解的樣式進行區分（類似於 //1.cpp, //2.cpp 或者 //main.cpp).

⑤ **程式碼翻譯題.**（2 分）編碼器在將我們寫出的程式碼轉換為機器能夠識別的機器碼之前，需要對程式碼進行翻譯. 首先，對於一個回傳型別非 void 的函式，編碼器會嘗試在原地配置一塊記憶體，然後將這塊記憶體轉交給函式初始化之後，函式內部對於該物件的操作可以直接在原地進行，這樣避免了回傳物件帶來的複製. 這種技術在 C++ 中被稱為回傳值優化技術，它可以被用於部分函式. 除此之外，編碼器在處理類別成員函式的時候，也和我們看起來的不太一樣. 它首先將物件以引數的形式繫結到第一個參數上，然後使用 this 訪問成員. 如果類別中帶有虛擬函式，那麼絕大多數編碼器會在類別的最底部增添一個指向虛擬函式表格的指標. 虛擬函式表格中從第一個指標開始，這些指標會指向那些真正會被呼叫的虛擬函式. 現假設指向虛擬函式表格的指標名稱為 __vptr. 虛擬函式表格中的第一個指標指向類別 X 的解構子，第二個指標指向類別 X 的虛擬成員函式 f. 請根據上述信息，翻譯下列程式碼中的函式 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;
}
```

⑥ **閱讀設計題.**（2 分）現有一四層樓加裝了電梯. 不妨設電梯（lift）使用 class 進行了實作，其提供如下操作 : 上升（up)，下降（down)，停止（stop)，開門（open)，關門（close)，判定某一樓層是否需要停止（stop_at(n)）和獲取目前停止的樓層（stop_position). 現在電梯採用的策略是 : 若目前電梯是上行，那麼上行會一直進行到請求電梯的最高樓層，上行過程中在每一個需要請求停止的樓層停止; 若目前電梯是下行，那麼下行會一直進行到請求電梯的最低樓層，下行過程中在每一個需要請求停止的樓層停止. 若在上行或者下行的途中，出現反方向的請求，那麼首先處理完當前請求再去處理反方向請求. 設計一函式 :

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

來模擬電梯的運作，並且簡單說明模擬電梯運作在作業系統中的某些應用. 可以使用的 C++ 標準樣板程式庫设施為 std::bitset.

28. 記憶體控制題.（12 分）① ~ ② 題. 某公司旗下的所有項目都有一個基本要求 : 為了避免記憶體流失, 公司規定應該儘量使用 C++ 11 引入的智慧指標, 即 std::shared_ptr, std::unique_ptr 和 std::weak_ptr. 該公司規定 : 若某一段配置的記憶體可能會在多個函式下被使用, 那麼就應該使用智慧指標; 若要使用內建指標, 那麼該指標對應的記憶體僅能在當前函式可視範圍之內使用, 離開當前函式前必須回收這段配置的記憶體. 現該公司某一個項目為了程式碼的運作效率, 在遵守上述前提的基礎上, 對限於函式內使用的記憶體全部採用內建指標而不採用智慧指標. 另外, 由於該項目需要和 C 程式碼進行交互, 因此僅僅使用內建的 ::operator new 和 ::operator delete 來配置記憶體. 該項目完成之後, 需要進行記憶體流失情況的檢測, 以確保項目交付時不出現記憶體流失的錯誤. 因此, 現在需要手動向程式碼中添加一個用於檢測記憶體流失的類別 memory_helper 和相關的程式碼. 對於類別 memory_helper, 其使用方式和與其相關的限制如下 :

- 類別中的所有成員變數對應型別都是內建型別;
- 該類別提供了成員函式樣板 memory_helper:::allocate 用於配置記憶體, 其接受一個型別為 std::size_t 的引數. 所有原本通過內建的 ::operator new 來完成的記憶體配置都轉交由其來完成. 預設情況下, 其回傳型別和內建的 ::operator new 的回傳型別一致. 除此之外, 成員函式樣板 memory_helper::allocate 還需要檢測類別使用者給定的樣板引數是否為指標型別. 如果使用者給出的樣板引數並非指標型別, 應該給出一個說明為 "The template argument must be a type of pointer!" 的編碼錯誤;
- 該類別提供了成員函式 memory_helper::deallocate 用於回收記憶體, 其接受一個型別為 void * 的引數. 所有通過內建的 ::operator delete 來完成的記憶體回收都轉交由其來完成. 其回傳型別和內建的 ::operator delete 的回傳型別一致. 除此之外, 成員函式 memory_helper::deallocate 還需要檢測給定的記憶體是否經由成員函式樣板 memory_helper::allocate 配置. 如果給定的記憶體並非是由 memory_helper::allocate 配置的, 那麼應該擲出一說明為 "The given memory is not allocated by memory_helper!" 的例外情況;
- 類別的建構子不應該擲出例外情況, 有編碼期計算能力, 且交由編碼器來完成;
- 類別的解構子用於檢測是否存在記憶體流失, 如果存在記憶體流失, 那麼擲出例外情況. 除此之外, 解構子中不能出現任何動態記憶體配置或者回收的行為;
- 在類別實作完成之後, 對於任意需要進行檢測的函式, 在函式最開始處宣告一個 memory_helper 物件. 然後將函式中所有原本使用內建的 ::operator new 和 ::operator delete 進行記憶體配置或者回收的陳述式都更改為使用 memory_helper 進行配置或者回收. 除此之外, 還需要在函式定義處以及函式內部程式碼之外的地方添加一些程式碼, 就可以做到記憶體流失的檢測和流失記憶體的回收, 這將最大化地降低程式碼的修改幅度. 當出現記憶體流失的情況時, 通過 std::cerr 輸出 "Memory leak detected!", 並且回收流失的記憶體;
- 可以使用的 C++ 標準樣板程式庫設施為 std::runtime_error, std::unique_ptr, std::is_pointer, std::cerr 和 std::endl;
- 流失的記憶體需要手動回收, 不可借助 std::unique_ptr;
- 提示 : static_assert 是 C++ 11 引入的靜態斷言, 其接受兩個函式形式的引數, 第一個引數是一個可以轉型為 bool 型別的常數表達式, 第二個引數是一個字面值字串. 當第一個引數為 false 時, 編碼器會擲出編碼錯誤, 並且將第二個引數輸出作為編碼錯誤提示的一部分.

①（11 分）請根據上述要求實作類別 memory_helper.

②（1 分）請根據上述要求修改下列程式碼 :

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

2021 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