2020 年全國 C++ 等級考試

標準答案

(一)、選擇題

1	2	3	4	5	6	7	8
A	ABCDF	BCD	BCF	A	ACEF	BDEG	ABCDE

(二)、填空題

9. decltype(i2 + 0) i3;
10. const void *volatile (Foo::*p)(int &, char, ...) const volatile && noexcept {&Foo::f};
11. 被 constexpr 標識的函式、實作在類別內部的函式、lambda 表達式
12. template <typename T, size_t N>
    auto f(T (&arr)[N]) -> decltype(arr[0]);
13. using std::operator>>;
14. void func(unsigned short *);、void func(unsigned); 與 void func(long long);

(三)、改錯題 (修改的地方使用粗體紅色標出)

15.
```
struct Foo {
  int a {0};
  int override {1};
  constexpr static int c {3};
  bool operator==(const Foo &rhs) {
        return this->a == rhs.a and this->override == rhs.override;
  }
  Foo() noexcept = default;
  Foo(const Foo &) = delete;
  Foo(Foo &&rhs) noexcept {
        this->a = rhs.a;
        this->override = rhs.override;
  }
  virtual ~Foo() = default;
};
struct Bar : Foo {
  std::string str;
  Bar(const std::string str = "0") : str(str) {}
};
int main(int argc, char *argv[]) {
  Bar b1("hello");
  Bar b2 = static_cast<Bar &&>(b1);
}
```
16.
```
template <typename T>
struct add_lvalue_reference {
        using type = T &;
};
template <>
struct add_lvalue_reference<void> {
        using type = void;
};
template <typename T, typename ...Args>
struct add_lvalue_reference<T (Args...)> {
        using type = T (Args...);
};
template <typename T, typename ...Args>
```

```
    struct add_lvalue_reference<T (Args..., ...)> {
          using type = T (Args..., ...);
    };
17. void f(char &) noexcept;
    int main(int argc, char *argv[]) {
     decltype(f) *fp;
     fp = f;
     char c = 0;
     fp(c);
     auto *p = new int;
     const int *cp = p;
     //const_cast<int *>(cp) = p;
     struct Foo {
          int *p;
          struct Bar {
               int i = 0;
               decltype(f) *fp = f;
               bool operator()(char c) noexcept {
                    if(this->i == c) {
                         return false;
                    }
                    fp(c);
                    return true;
               }
          };
     };
    }
18. long long fac(int n, int r = 1) {
     if(n == 1) {
          return r;
     }
     return fac(n - 1, r * n);
    }
    long double cal(int n) {
     bool symbol = false;
     long double r = 1.0l;
     long double t = 1.0l;
     for(int d = 2; d <= n; ++d) {
          t *= (2 * d - 2) * (2 * d - 1);
          r += [=]() mutable -> long double {
               if(symbol) {
                    return 1 / t;
               }
               return -(1 / t);
          }();
          symbol = !symbol;
     }
    }
```

(四)、論述題

19. 從 C++ 層面來說, 所有 I/O 物件對應的複製建構子和複製指派運算子都被標識 private 的, 因此在外界無法進行複製; 從流本身的層面來說, 複製一個流本身沒有任何意義

20. 對於物件的移動, 它通常不額外配置記憶體, 只是簡單地 "竊取" 物件中本身包含地信息. 在移動的過程中, 如果擲出例外情況, 那麼就會出現舊的物件被改變了, 而新的物件由沒有被建構

完成，那麼此時很可能導致兩個物件都無法使用的情況，甚至更嚴重的，還會出現資源流失的情況．對於程式庫來說，如果某個物件是可移動的且移動操作保證不擲出任何例外情況，那麼從一定程度上保障了例外安全性，還方便了程式庫或編碼器對程式碼的優化
21．被移動之後的物件只能保證其可解構，對於類別使用者來說，使用者不可能面面俱到，了解每一個類別的移動建構子或著移動指派運算子是如何工作的．從這個方面來說，類別使用者對移動之後的物件是什麼樣的，處於什麼狀態一無所知．使用這樣未知的物件容易發生未知的錯誤
22．std::list 與 std::forward_list 提供的疊代器並非隨機訪問疊代器，而 C++ 標準樣板程式庫中的 std::sort 函式要求給定的疊代器是一個隨機訪問疊代器．從這兩個類別本身的設計來說，它們內部雖然可以存取元素，但是元素的排列在記憶體中並非順序的，分別給它們設計自己的排序函式可以針對這兩個資料結構進行特別的優化
23．要求疊代器而非要求容器可以做到演算法與資料結構相隔離，從而使得演算法的設計者無需考慮容器的內部結構，只需考慮疊代器滿足的操作．當用戶自己設計容器的時候，若演算法不要求疊代器而要求容器，那麼不能保證用戶自己的容器可以用於每一個 C++ 標準樣板程式庫的演算法；但若要求疊代器，那麼只要用戶設計符合標準的疊代器，那麼就可以讓 C++ 標準樣板程式庫中的演算法用於自己設計的容器
24．
```
const Foo (*p)(Bar &, Baz *) noexcept = ::operator==;

    auto lambda {[](Bar &lhs, Baz *rhs) mutable noexcept -> const Foo {
      return lhs == rhs;
    }}

    struct operator_call_overloading {
      const Foo operator()(Bar &lhs, Baz *rhs) noexcept {
            return lhs == rhs;
      }
    };

#include <functional>
using namespace std;
using namespace std::placeholders;
    auto callable_obj {std::bind(::operator==, __1, _2)};

#include <functional>
using namespace std;
      function<const Foo(Bar &, Baz *) noexcept> f = ::operator==;
```

(五)、程式設計
25．①特性 : 使用 default 替換已經給定的預設引數
好處 : 當某些函式的參數順序不合理時，出現前幾個參數本來無需給出由預設引數填充，而後幾個參數需要自己給定時，再去查看函式的宣告取得預設引數將顯得有些麻煩，此時可以直接使用 default 避免
②特性 : 在 if 陳述式內宣告變數
好處 : 當有些變數僅在 if 陳述式內被用到的時候，提前宣告可能會污染外層的可視範圍．這個特性可以將這些變數的可視範圍限定在 if 陳述式之內
③特性 : 由編碼器推斷 lambda 的參數型別和回傳型別；當 lambda 處於類別內部的時候，允許 lambda 捕獲 this
好處 : 前半個特性使得泛型 lambda 稱為可能，從而有時可以避免 template 污染外圍的可視範圍；後半個特性允許 lambda 內對類別內部的物件進行操作
26．

```
template <typename T>
class stack {
    private:
        T *arr;
        T *cursor;
        T *end;
    public:
        stack(decltype(sizeof 0) size = 64) try : arr {new
T[size] {}}, cursor {this->arr}, end {this->arr + size} {}
catch(...) {
            delete[] this->arr;
            throw;
        }
        stack(const stack &rhs) try : arr {new T[rhs.end -
rhs.arr] {}}, cursor {this->arr}, end {this->arr + (rhs.end -
rhs.arr)} {
            for(auto cursor {rhs.arr}; cursor not_eq
rhs.cursor;) {
                *this->cursor++ = *cursor++;
            }
        }catch(...) {
            delete[] this->arr;
            throw;
        }
        stack(stack &&rhs) noexcept : arr {rhs.arr}, cursor
{rhs.arr}, end {rhs.end} {
            rhs.arr = rhs.cursor = nullptr;
        }
        ~stack() {
            delete[] this->arr;
        }
    public:
        stack &operator=(const stack &rhs) {
            const auto size {rhs.end - rhs.begin};
            T *new_arr {new T[size]};
            auto new_arr_cursor {new_arr};
            try {
                for(auto cursor {rhs.arr}; cursor not_eq
rhs.cursor;) {
                    *new_arr_cursor++ = *cursor++;
                }
                delete[] this->arr;
            }catch(...) {
                delete[] new_arr;
                throw;
            }
            this->arr = new_arr;
            this->cursor = new_arr_cursor;
            this->end = new_arr + size;
            return *this;
        }
```

```
        stack &operator=(stack &&rhs) noexcept {
            if(this == &rhs) {
                return *this;
            }
            delete[] this->arr;
            this->arr = rhs.arr;
            this->cursor = rhs.cursor;
            this->end = rhs.end;
            rhs.arr = nullptr;
            return *this;
        }
    public:
        void push(const T &value) {
            if(this->cursor == this->arr) {
                throw "The stack is full!";
            }
            *this->cursor++ = value;
        }
        void pop() {
            if(this->cursor == this->arr) {
                throw "The stack is empty";
            }
            --this->cursor;
        }
        T &top() noexcept {
            return *this->cursor;
        }
        const T &top() const noexcept {
            return *this->cursor;
        }
        void clear() noexcept {
            this->cursor = this->arr;
        }
        bool empty() const noexcept {
            return this->arr == this->cursor;
        }
};
```

27.

```
#include <iostream>
using namespace std;
class creature {
    friend class nature;
    protected:
        int lifetime;
        int height;
        int weight;
    public:
        creature(int lifetime, int height, int weight)
noexcept : lifetime {lifetime}, height {height}, weight
{weight} {}
        creature(const creature &) noexcept = default;
```

```
        creature(creature &&rhs) noexcept : lifetime
{rhs.lifetime}, height {rhs.height}, weight {rhs.weight} {}
        virtual ~creature() noexcept = default;
    public:
        creature &operator=(const creature &) = delete;
        creature &operator=(creature &&) noexcept = delete;
    public:
        virtual void breathe() const & noexcept = 0;
};
class primate : public creature {
    friend class nature;
    protected:
        struct primate_eyes {};
        struct primate_nose {};
        struct primate_mouse {};
        struct primate_ears {};
        struct primate_hand {};
        struct primate_foot {};
        struct primate_tail {};
        struct primate_hair {};
    protected:
        primate_eyes eyes;
        primate_nose nose;
        primate_mouse mouse;
        primate_ears ears;
        primate_hand hand;
        primate_foot foot;
    public:
        using creature::creature;
    public:
        virtual void crawl() const & noexcept = 0;
};
class monkey : public primate {
    friend class nature;
    protected:
        primate_tail tail;
        primate_hair hair;
    public:
        using primate::primate;
    public:
        void breathe() const & noexcept override {
            cout << "breathe" << endl;
        }
        void crawl() const & noexcept override {
            cout << "crawl" << endl;
        }
        void jump() const & noexcept {
            cout << "jump" << endl;
        }
};
class acient_ape : public monkey {
```

```
    friend class nature;
    private:
        void degeneration() noexcept {
            cout << "degeneration" << endl;
        }
    public:
        acient_ape(int lifetime, int height, int weight) :
monkey(lifetime, height, weight) {
            this->degeneration();
        }
        acient_ape(const acient_ape &) noexcept = default;
        acient_ape(acient_ape &&) noexcept = default;
        ~acient_ape() override = default;
};
class ape : virtual public acient_ape {
    friend class nature;
    public:
        using acient_ape::acient_ape;
};
class primitive : virtual public acient_ape {
    friend class nature;
    public:
        using acient_ape::acient_ape;
    public:
        void walk() const & noexcept {
            cout << "walk" << endl;
        }
        void run() const & noexcept {
            cout << "run" << endl;
        }
};
class human final : public ape, public primitive {
    friend class nature;
    friend class society;
    private:
        string name;
    public:
        human(int lifetime, int height, int weight, const
string &name) : acient_ape(lifetime, height, weight),
ape(lifetime, height, weight), primitive(lifetime, height,
weight), name {name} {

        }
        human(const human &) = delete;
        human(human &&) = delete;
        ~human() noexcept = default;
    public:
        void think() const & noexcept {
            cout << "human think" << endl;
        }
```

```
        void speak(const string &statement) const & noexcept
    {
            cout << statement << endl;
        }
        template <typename Thing>
        Thing create() const & {
            return Thing();
        }
    };
```

28.

```
    #include <iostream>
    #include <vector>
    using namespace std;
    auto move_count {0ull};
    inline void move(vector<int> &from, vector<int> &to, char
    char_from, char char_to) {
        auto value {from.back()};
        from.pop_back();
        to.push_back(value);
        ++move_count;
        cout << "Move " << char_from << " To " << char_to <<
    endl;
    }
    void hanoi_aux(typename vector<int>::size_type size,
    vector<int> &X, vector<int> &Y, vector<int> &Z, char char_X =
    'X', char char_Y = 'Y', char char_Z = 'Z') {
        if(size == 0) {
            return;
        }
        hanoi_aux(size - 1, X, Z, Y, char_X, char_Z, char_Y);
        move(X, Z, char_X, char_Z);
        hanoi_aux(size - 1, Y, Z, X, char_Y, char_Z, char_X);
    }
    inline void hanoi(vector<int> &X, vector<int> &Y, vector<int>
    &Z) {
        hanoi_aux(X.size(), X, Y, Z);
    }
    vector<int> init_X(int n) {
        vector<int> vec;
        while(n > 0) {
            vec.push_back(n--);
        }
        return vec;
    }
    int main() {
        auto X {init_X(10)};
        vector<int> Y {}, Z {};
        hanoi(X, Y, Z);
        cout << "Move Times : " << move_count << endl;
        for(auto c : Z) {
            cout << c << "\t";
```

```
        }
        cout << endl;
    }
```

29. ①

```
    #include <iostream>
    using namespace std;
    template <typename T, size_t N>
    void in(int (&arr)[N], int index, T &&value) {
        arr[index] = forward<T>(value);
    }
    template <typename T, typename ...Args, size_t N>
    void in(int (&arr)[N], int index, T &&value, Args &&...args)
    {
        arr[index] = forward<T>(value);
        in(arr, index + 1, forward<Args>(args)...);
    }
    template <typename ...Args>
    void f(Args &&...args) {
        int arr[sizeof...(Args)] {};
        in(arr, 0, forward<Args>(args)...);
        for(auto i {0}; i < sizeof...(Args); ++i) {
            for(auto j {0}; j < sizeof...(Args) - i - 1; ++j) {
                if(arr[j] > arr[j + 1]) {
                    auto temp {arr[j]};
                    arr[j] = arr[j + 1];
                    arr[j + 1] = temp;
                }
            }
        }
        for(auto c : arr) {
            cout << c << "\t";
        }
        cout << endl;
    }
```

②

```
    #include <cmath>
    using namespace std;
    void derivative(double &a, double &n) {
        a = n-- * a;
    }
    double derivative(double &a, double &n, double x) {
        derivative(a, n);
        return a * pow(x, n);
    }
```

③

```
    #include <cmath>
    using namespace std;
    void integral(double &a, double &n) {
        a /= n++;
    }
```

```
double integral(double &a, double &n, double begin, double
end) {
    integral(a, n);
    return a * pow(begin, n) - a * pow(end, n);
}
```

④

```
#include <vector>
#include <stack>
using namespace std;
vector<int> unfold(const generalized_list<int> &glist)
noexcept {
    vector<int> vec {};
    stack<generalized_list<int>> s {};
    for(auto i {glist.size() - 1}; i >= 0; --i) {
        s.push(glist[i]);
    }
    while(!s.empty()) {
        if(s.top().empty()) {
            s.pop();
            continue;
        }
        if(s.top().is_glist(0)) {
            for(auto i {s.top().size() - 1}; i >= 0; --i) {
                s.push(s.top()[i]);
            }
            continue;
        }
        vec.push_back(s.top());
        s.pop();
    }
    return vec;
}
```

30.

```
#include <memory>

template <typename T, typename Deleter =
std::default_delete<T>>
struct unique_ptr_helper {
  protected:
      std::unique_ptr<T, Deleter> *ptr;
      T *temp_ptr;
  public:
      constexpr unique_ptr_helper() noexcept = default;
      explicit unique_ptr_helper(std::unique_ptr<T, Deleter>
&ptr) noexcept : ptr {&ptr}, temp_ptr {ptr.release()} {}
      unique_ptr_helper(const unique_ptr_helper &) = delete;
      unique_ptr_helper(unique_ptr_helper &&rhs) noexcept :
ptr {std::exchange(rhs.ptr, nullptr)} {}
      virtual ~unique_ptr_helper() noexcept {
          if(this->ptr) {
              this->ptr->reset(this->temp_ptr);
```

```
          }
  }
public:
      unique_ptr_helper &operator=(const unique_ptr_helper &)
& = delete;
      unique_ptr_helper &operator=(unique_ptr_helper &&) &
noexcept = delete;
      explicit virtual operator T **() && noexcept {
          return &this->temp_ptr;
      }
};

template <typename T>
unique_ptr_helper<T> unique_ptr_transfer(std::unique_ptr<T>
&p) {
  return unique_ptr_helper<T>(p);
}
```