

**C++** 六级

2024 年 09 ⽉

**1** 单选题 (每题 **2** 分,共 **30** 分)

**第 1 题** 以下（ ）没有涉及 C++语⾔的⾯向对象特性⽀持。  **A.** C++ 中构造一个 class 或 struct

 **B.** C++ 中调⽤ printf 函数

口 **C.** C++ 中调⽤⽤户定义的类成员函数

口 **D.** C++ 中构造来源于同一基类的多个派⽣类

**第 2 题** 关于以下C++代码，（ ）⾏代码会引起编译错误。

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

#include <iostream> using namespace std;

class Base { private:

int a; protected:

int b; public:

int c;

Base() : a(1), b(2), c(3) {} };

class Derived : public Base { public:

void show() {

cout << a << endl; // Line 1

cout << b << endl; // Line 2

cout << c << endl; // Line 3 }

};

口 **A.** Line 1 口 **B.** Line 2

口 **C.** Line 3

口 **D.** 没有编译错误

**第 3 题** 有6个元素 ，按照 6,5,4,3,2, 1 的顺序进⼊栈S ，下列（ ）的出栈序列是不能出现的（ ）。 口 **A.** 5,4,3,6, 1,2

口 **B.** 4,5,3, 1,2,6

口 **C.** 3,4,6,5,2, 1

口 **D.** 2,3,4, 1,5,6

**第 4 题** 采⽤如下代码实现检查输⼊的字符串括号是否匹配 ，横线上应填⼊的代码为（ ）。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | #include <iostream> #include <stack>  #include <string>  using namespace std;  bool is\_valid(string s) {  stack<char> st;  char top;  for (char& ch : s) {  if (ch == '( ' || ch == '{ ' || ch == '[') { st.push(ch); // 左括号入栈  }  else  {  if (st.empty())  return false;  ———————————————————————— // 在此处填入代码  if ((ch == ') ' && top != '(') || (ch == '} ' && top != '{') || (ch == '] ' && top != '[')) { return false;  }  }  }  return st.empty(); // 栈为空则说明所有括号匹配成功  } |

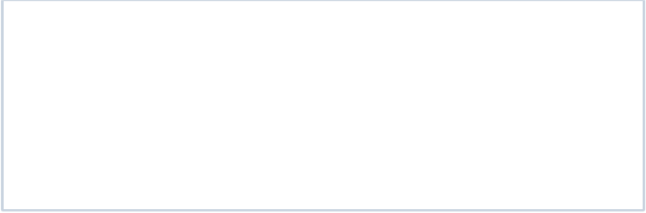
 **A.** top = st.top(); st.pop();

 **B.** st.pop(); top = st.top();

 **C.** st.pop(); top = st.front();

 **D.** top = st.front(); st.pop();

**第 5 题** 下⾯代码判断队列的第一个元素是否等于a ，并删除该元素 ，横向上应填写（ ）。

1

2

3

4

5

6

7

8

9

10

11

#include <iostream> #include <queue>

using namespace std;

bool is\_front\_equal(std::queue<int>& q, int a) { bool is\_equal = false;

if (!q.empty()) {

———————————————————————— // 在此处填入代码

}

return is\_equal; }

口 **A.** is\_equal = (q.front() == a);

口 **B.** is\_equal = (q.front() == a); q.pop();

 **C.** q.pop();is\_equal=(q.front()==a);

口 **D.** q.pop(); is\_equal = (q.top() == a);

**第 6 题** 假设字母表 {a,b,c,d,e} 在字符串出现的频率分别为 10%， 15% ，30%， 16% ，29% 。若使⽤哈夫曼编码⽅ 式对字母进⾏⼆进制编码 ，则字符 abcdef 分别对应的一组哈夫曼编码的长度分别为（ ）。

口 **A.** 4, 4, 1, 3, 2

口 **B.** 3, 3, 2, 2, 2

口 **C.** 3, 3, 1, 2, 1

口 **D.** 4, 4, 1, 2, 2

**第 7 题** 以下C++代码实现n位的格雷码 ，则横线上应填写（ ）。

#include <iostream> #include <vector>

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

#include <string>

using namespace std;

// 生成 n 位的格雷码

vector<string> generate\_graycode(int n) {

vector<string> graycode\_list;

if (n <= 0) {

return graycode\_list; }

// 初始1位格雷码

graycode\_list.push\_back("0");

graycode\_list.push\_back("1");

// 迭代生成 n 位的格雷码

for (int i = 2; i <= n; i++) {

int current\_size = graycode\_list.size();

for (int j = current\_size - 1; j >= 0; j--) {

graycode\_list.push\_back("1" + graycode\_list[j]);

}

for (int j = 0; j < current\_size; j++) {

25

26

27

28

29

30

}

}

return graycode\_list; }

// 在此处填入代码

 **A.** graycode\_list.push\_back("0" + graycode\_list[j]);

 **B.** graycode\_list[j] = "0" + graycode\_list[j];

 **C.** graycode\_list.push\_back("1" + graycode\_list[j]);

 **D.** graycode\_list[j] = "1" + graycode\_list[j];

**第 8 题** 给定一棵⼆叉树 ，其前序遍历结果为：ABDECFG, 中序遍历结果为：DEBACFG ，则这棵树的正确后序遍历 结果是（ ）。

口 **A.** EDBGFCA

 **B.** EDGBFCA

口 **C.** DEBGFCA

口 **D.** DBEGFCA

**第 9 题** 一棵有n个结点的完全⼆叉树⽤数组进⾏存储与表⽰， 已知根结点存储在数组的第1个位置 。若存储在数组第 9个位置的结点存在兄弟结点和两个⼦结点 ，则它的兄弟结点和右⼦结点的位置分别是（ ）。

口 **A.** 8, 18

口 **B.** 10, 18

口 **C.** 8, 19

口 **D.** 10, 19

**第 10 题** ⼆叉树的深度定义为从根结点到叶结点的最长路径上的结点数 ，则以下基于⼆叉树的深度优先搜索实现的 深度计算函数中横线上应填写（ ）。

// 定义二叉树的结点结构

1

2

3

4

5

6

7

8

struct tree\_node { int val;

tree\_node\* left;

tree\_node\* right;

tree\_node(int x) : val(x), left(nullptr), right(nullptr) {} };

9

// 计算二叉树的深度

10

11

12

13

14

15

16

17

int max\_depth(tree\_node\* root) { if (root == nullptr) {

return 0; // 如果根结点为空 ，则深度为 0

}

int left\_depth = max\_depth(root->left);

int right\_depth = max\_depth(root->right);

18

19

// 在此处填入代码

————————————————————————

20

}

口 **A.** return left\_depth + right\_depth;

口 **B.** return max(left\_depth, right\_depth);

 **C.** return max(left\_depth, right\_depth) + 1;

 **D.** return left\_depth + right\_depth + 1;

**第 11 题** 上一题的⼆叉树深度计算还可以采⽤⼆叉树的⼴度优先搜索来实现 。以下基于⼆叉树的⼴度优先搜索实现 的深度计算函数中横线上应填写（ ）。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | #include <queue>  int max\_depth\_bfs(tree\_node\* root) { if (root == nullptr) {  return 0; // 如果树为空 ，深度为 0  }  queue <tree\_node\*> q;  q.push(root);  int depth = 0;  // 使用队列进行层序遍历  while (!q.empty()) {  ———————————————————————— // 在此处填入代码  for (int i = 0; i < level\_size; ++i) {  tree\_node\* node = q.front();  q.pop();  if (node->left) {  q.push(node->left); }  if (node->right) {  q.push(node->right); }  }  }  return depth; } |

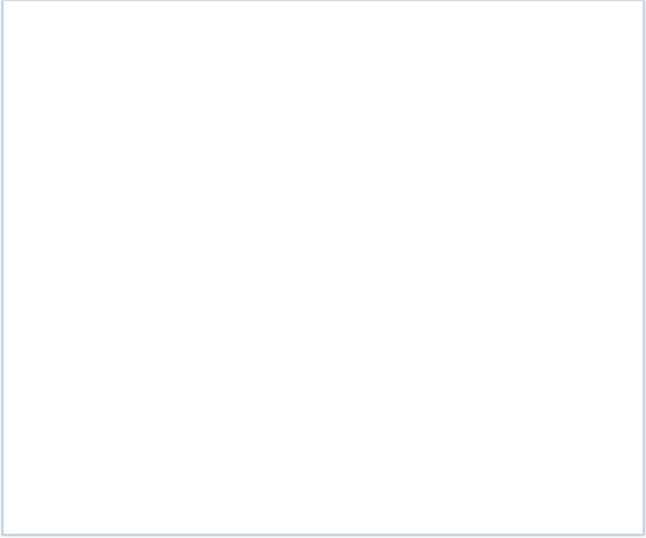
口 **A.** int level\_size = q.size(); depth++;

口 **B.** int level\_size = 2; depth++;

 **C.** int level\_size = q.size(); depth += level\_size;

 **D.** int level\_size = 2; depth += level\_size;

**第 12 题** ⼆叉搜索树中的每个结点 ，其左⼦树的所有结点值都⼩于该结点值 ，右⼦树的所有结点值都⼤于该结点 值 。以下代码对给定的整数数组(假设数组中没有数值相等的元素) ，构造一个对应的⼆叉搜索树 ，横线上应填写 ( ) :

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

// 定义二叉树的结点结构

struct tree\_node { int val;

tree\_node\* left;

tree\_node\* right;

tree\_node(int x) : val(x), left(nullptr), right(nullptr) {} };

// 插入结点到二叉搜索树中

tree\_node\* insert(tree\_node\* root, int val) { if (root == nullptr) {

return new tree\_node(val); }

———————————————————————— // 在此处填入代码

return root; }

// 根据给定数组构造二叉搜索树

tree\_node\* constructBST(const int arr[], int size) { tree\_node\* root = nullptr;

for (int i = 0; i < size; ++i) {

root = insert(root, arr[i]); }

return root; }

口 **A.**



1

2

3

4



if (val < root->val)

root->left = insert(root->left, val);

else

root->right = insert(root->right, val);

口 **B.**

1

2

3

4

|  |
| --- |
| if (val > root->val)  root->left = insert(root->left, val);  else  root->right = insert(root->right, val); |

口 **C.**

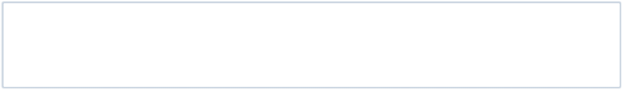


1

2

3

4



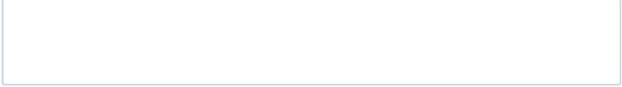
if (val < root->val)

root->left = insert(root, val);

else

root->right = insert(root, val);

口 **D.**

1

2

3

4

if (val > root->val)

root->left = insert(root, val);

else

root->right = insert(root, val);

**第 13 题**

对上题中的⼆叉搜素树 ，当输⼊数组为5 , 3 , 7, 2 , 4, 6, 8]时 ，构建⼆叉搜索树 ，并采⽤如下代码实现的遍历⽅式 ，得到 的输出是（ ）。

1

2

3

4

5

6

7

8

9

10

11

12

13

#include <iostream> using namespace std;

// 遍历二叉搜索树 ，输出结点值

void traversal(tree\_node\* root) { if (root == nullptr) {

return; }

traversal(root->left);

cout << root->val << " ";

traversal(root->right); }

口 **A.** 5 3 7 2 4 6 8

口 **B.** 2 3 45 6 78

口 **C.** 2 43 6 8 7 5

口 **D.** 2 43 5 6 7 8

**第 14 题** 动态规划通常⽤于解决（ ）。

口 **A.** ⽆法分解的问题

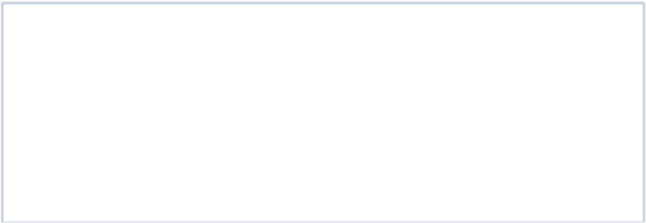
口 **B.** 可以分解成相互依赖的⼦问题的问题 口 **C.** 可以通过贪⼼算法解决的问题

口 **D.** 只能通过递归解决的问题

**第 15 题** 阅读以下⽤动态规划解决的0-1背包问题的函数 ，假设背包的容量w是10kg ，假设输⼊4个物品的重量

ueights分别为1 , 3, 4, 6

（单位为kg） ，每个物品对应的价值values分别为20, 30, 50, 60 ，则函数的输出为（ ）。

#include <iostream> #include <vector>

1

2

3

4

5

6

7

using namespace std;

// 0/1背包问题

int knapsack(int W, const vector<int>& weights, const vector<int>& values, int n) { vector<vector<int>> dp(n + 1, vector<int>(W + 1, 0));

8

for (int i = 1; i <= n; ++i) {

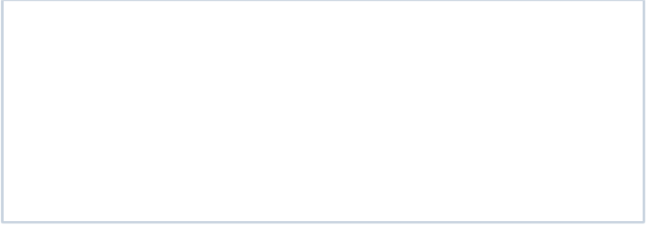
9

10

11

for (int w = 0; w <= W; ++w) {

if (weights[i - 1] <= w) {

12

13

14

15

16

17

18

19

20

21

22

dp[i][w] = max(dp[i - 1][w], dp[i - 1][w - weights[i - 1]] + values[i - 1]);

}

else

{

dp[i][w] = dp[i - 1][w];

}

}

}

return dp[n][W]; }

口 **A.** 90

口 **B.** 100

口 **C.** 110

口 **D.** 140

**2** 判断题 (每题 **2** 分,共 **20** 分)

**第 1 题** C++ 、Python和JAVA等都是⾯向对象的编程语⾔ 。

**第 2 题** 在C++中 ，类的静态成员变量只能被该类对象的成员函数访问。

**第 3 题** 栈是一种线性结构 ，可通过数组或链表来实现 。⼆者相⽐ ，数组实现占⽤的内存较少 ，链表实现的⼊队和出 队操作的时间复杂度较低。

**第 4 题** 运⾏以下C++代码 ，屏幕将输出“derived class”。

#include <iostream> using namespace std;

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

class base { public:

virtual void show() {

cout << "base class" << endl; }

};

class derived : public base { public:

void show() override {

cout << "derived class" << endl; }

};

int main() {

base\* b;

derived d;

b = &d;

22

23

24

b->show();

return 0; }

**第 5 题** 如下列代码所⽰的基类（base）及其派⽣类（derived） ，则⽣成一个派⽣类的对象时 ，只调⽤派⽣类的构造 函数。

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

#include <iostream> using namespace std;

class base { public:

base() {

cout << "base constructor" << endl; }

~base() {

cout << "base destructor" << endl; }

};

class derived : public base { public:

derived() {

cout << "derived constructor" << endl; }

~derived() {

cout << "derived destructor" << endl; }

};

**第 6 题** 哈夫曼编码本质上是一种贪⼼策略。

**第 7 题** 如果根结点的深度记为1 ，则一棵恰有2024个叶结点的⼆叉树的深度最少是12 。 **第 8 题** 在⾮递归实现的树的⼴度优先搜索中 ，通常使⽤栈来辅助实现。

**第 9 题** 状态转移⽅程是动态规划的核⼼ ，可以通过递推⽅式表⽰问题状态的变化。

**第 10 题** 应⽤动态规划算法时 ，识别并存储重叠⼦问题的解是必须的。