

# 2020

---

## 填空题

1.若有函数定义为

```
int func(int n) {  
    if(n <= 1)  
        return 1;  
    else  
        return (2 + n * func(n - 1));  
}
```

假设 `m` 为 `int` 类型，则执行语句 `m = func(5);` 后，`m` 的值是 \_\_\_\_\_。

\$\newline\$

## 选择题

1.若线性表最常用的运算是存取第  $\mathrm{i}$  个元素及其前驱的值，则采用 \_\_\_\_\_。

A. 单链表 B. 双链表 C. 顺序表 D. 循环单链表

\$\newline\$

2.若有语句 `int *point, a = 4;` 和 `point = &a;`，下面均代表地址的一组选项是 \_\_\_\_\_。

- A. `a`, `point`, `*&a`
- B. `&(*a)`, `&a`, `*point`
- C. `*(&point)`, `*point`, `&a`
- D. `&a`, `&(*point)`, `point`

\$\newline\$

3.设某非空的单循环链表的头指针（指向第一个结点的指针）为 `Head`，尾指针（指向最后一个结点的指针）为 `Rear`，则下列条件判断结果一定为真的是 \_\_\_\_\_。

- A. `Head == Rear->link`
- B. `Head == Rear->link->link`
- C. `Rear == Head->link`
- D. `Rear == Head->link->link`

\$\newline\$

4.下列语句中，能正确进行字符串赋值的是\_\_\_\_\_。

- A. `char s[10]; s="BUAA";`
- B. `char* sp; *sp="BUAA";`
- C. `char *sp="BUAA";`
- D. `char s[10]; *s="BUAA";`

\$\newline\$

5.若要删除非空线性链表中由 `p` 所指的链结点的直接后继链结点（由 `p->link` 指向），则需依次执行\_\_\_\_\_。

- A. `r=p->link; p->link=r; free(r);`
- B. `r=p->link; p->link=r->link; free(r);`
- C. `r=p->link; p->link=r->link; free(p);`
- D. `p->link=p->link->link; free(p);`

\$\newline\$

6.顺序存储表示中数据元素之间的逻辑关系是由\_\_\_\_\_表示的，链式存储表示中数据元素之间的逻辑关系是由\_\_\_\_\_表示的。

- A. 指针
- B. 逻辑顺序
- C. 存储位置
- D. 问题上下文

\$\newline\$

7.设 `p` 指针指向单链表（单链表长度为 `n`）中的某个结点（`p≠NULL`），若只知道指向该单链表第一个结点的指针和 `p` 指针，则在 `p` 指针所指结点之前和 `p` 指针所指结点之后插入一个结点的时间复杂度分别是\_\_\_\_\_。

- A.  $O(1)$  和  $O(n)$
- B.  $O(1)$  和  $O(1)$
- C.  $O(n)$  和  $O(n)$
- D.  $O(n)$  和  $O(1)$

\$\newline\$

8.设有以下说明语句：

```
struct struType {  
    int a;
```

```
float b;  
} var;
```

则下面叙述中错误的是\_\_\_\_\_。

- A. `struct`是结构类型的关键字
- B. `struct struType`是用户定义的结构类型
- C. `var`是用户定义的结构类型名
- D. `a`和`b`都是结构成员名

\$\newline\$

9.若有以下说明和语句：

```
struct student {  
    int age;  
    int num;  
} std, *p;  
p = &std;
```

则以下对结构变量 `std` 中成员 `age` 的引用方式不正确的是\_\_\_\_\_。

- A. `std.age`
- B. `p->age`
- C. `(*p).age`
- D. `*p.age`

\$\newline\$

## 编程题

### 1. 标识符的识别

#### 问题描述

从控制台读入一行符合 C 语言语法要求的语句，该语句以分号结束。编写程序抽取出语句中的标识符，**重复的标识符只保留一个**，然后按照**字典序由小到大**的顺序输出这些标识符。

规定：

1. 读入的语句中字符个数不超过200，每个标识符的字符个数不超过32，语句中标识符的个数不超过50个，且至少有一个标识符。
2. 读入的语句中**不会出现字符常量**、字符串常量，也没有注释。
3. 读入的语句中**不会出现带后缀的常量类型**：例如：`2.5f`、`15l`、`18L`、`10611`、`5u`等情况。

**\*\*输入形式 \*\***

从控制台输入一行符合C语言语法要求的语句，语句以英文分号结束，末尾有回车符。

**输出形式**

按照字典序由小到大的顺序输出抽取出的标识符，各标识符间以一个空格分隔，最后一个标识符后有无空格均可。

**样例输入**

```
array_sum[i] = array_a[i] +array_a[j]+ _next2[ j ]*getArea( x, y , z) - 100*i;
```

**样例输出**

```
_next2 array_a array_sum getArea i j x y z
```

**样例说明**

从控制台读入的语句中有5个普通的变量：**i**、**j**、**x**、**y** 和 **z**，三个数组名：**array\_sum**、**array\_a** 和 **\_next2**，一个函数调用，其函数名为 **getArea**，将这9个标识符按照字典序由小到大排序输出。

\$\newline\$


2. 空闲空间申请模拟（首次适应）

**问题描述**

在操作系统中，空闲存储空间通常以空闲块链表方式组织，每个块包含块起始位置、块长度及一个指向下一块的指针。空闲块按照**存储位置升序组织**，最后一块指向第一块（构成**循环链表**）。当有空间申请请求时，按照如下原则在空闲块循环链表中寻找并分配合适的空间：

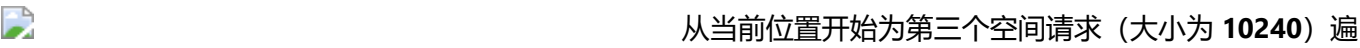
- 1. 从当前位置开始遍历空闲块链表（**初始时从地址最小的第一个空闲块开始**），寻找**第一个大于等于请求空间的空闲块**（即首次适应原则）。
- 2. 如果选择的空闲块恰好与请求的大小相符合，则将它从链表中移除并返回给用户；这时**当前位置变为被移除的空闲块指向的下一空闲块**。
- 3. 如果选择的空闲块大于所申请的空间大小，则将大小合适的空闲块返回给用户，剩下的部分留在空闲块链表中；这时**当前位置仍然为该空闲块**。
- 4. 如果找不到足够大的空闲块，则申请失败；这时**当前位置不变**。

例如：下图示例给出了空闲块链表的初始状态，每个结点表示一个空闲块，结点中上面的数字指空闲块的起始位置，下面的数字指空闲块的长度，**位置和长度都用正整数表示，大小不超过 int 表示范围**。当前位置为最小地址为 1024 的空闲块。

 若有 4 个申请空间请求，申请的空间大小依次为：**1024**、**2560**、**10240** 和 **512**。则从当前位置开始遍历链表，按照上述原则寻找到满足条件的空闲块为地址是 **16384** 的空闲块，其长度正好为 **1024**，所以将其从链表中删除，这时链表状态如下图所示，当前位置变成地址为 **32768** 的空闲块。



从当前位置开始为第二个空间请求（大小为 **2560**）遍历链表，按照上述原则寻找到满足条件的空闲块为当前位置的空闲块，其长度为 **3072**，大于请求的空间大小。于是分配 **2560** 的空间后，该空闲块剩余的长度为 **512**，当前位置不变，链表状态如下图所示：

从当前位置开始为第三个空间请求（大小为 **10240**）遍历链表，遍历一圈后发现找不到足够大的空闲块，则忽略该请求，当前位置不变。下面继续为第四个空间请求（大小为 **512**）遍历链表，按照上述原则寻找到满足条件的空闲块仍然是当前位置的空闲块，其长度等于请求的空间大小，于是将该空闲块删除后，链表状态变为下图所示：

编写程序，模拟上述空闲空间申请。

**输入形式**

先从控制台读入一正整数，表示当前空闲块的个数（大于0且小于等于100）。

然后**按照起始位置由小到大的顺序**分行输入每个空闲块的起始位置和长度，**位置和长度都用正整数表示，大小不超过 int 表示范围，两整数间以一个空格分隔。**

最后在新的一行上依次输入申请空间的大小，以 **-1** 表示结束，各整数间以一个空格分隔，申请请求的个数不超过 100 个。

**输出形式**

按照上述原则模拟完空闲空间申请后，输出当前空闲空间链表状态，即**从当前位置开始**，遍历链表，**分行输出**剩余空闲块的起始位置和长度，位置和长度间以一个空格分隔。若申请完后，链表中没有空闲块，则什么都不输出。

**样例输入**

```
12
1024 512
8192 512
16384 1024
32768 3072
65536 8192
77824 1024
80896 8192
96016 1024
101136 5120
119328 512
134448 1024
142640 3072
1024 2560 10240 512 2048 6400 2560 5600 2000 -1
```

**样例输出**

```
101136 560
119328 512
134448 1024
142640 3072
1024 512
```

|       |      |
|-------|------|
| 8192  | 512  |
| 65536 | 544  |
| 77824 | 1024 |
| 80896 | 1792 |
| 96016 | 1024 |

样例说明

样例输入了 12 个空闲块的信息，形成了如上述第一个图所示的空闲块链表；然后读取了 9 个空间申请请求。为前 4 个请求分配空间后，空闲块链表状态为上述最后一张图所示。满足第五个请求后，地址为 **65536** 的空闲块剩余长度为 **6144**；满足第六个请求后，地址为 **80896** 的空闲块剩余长度为 **1792**；满足第七个请求后，地址为 **101136** 的空闲块剩余长度为 **2560**；满足第八个请求后，地址为 **65536** 的空闲块剩余长度为 **544**；满足第九个请求后，地址为 **101136** 的空闲块剩余长度为 **560**。这时链表中剩余 10 个空闲块，当前位置为地址是 **101136** 的空闲块，从该空闲块开始依次遍历输出所有剩余空闲块的起始位置和长度。