05 分支与输入处理

根据当前数据选择处理路径，并检查输入是否足以支持判断

本讲学习目标

- 学会用已学条件和 if、if ... else 按当前情况选择处理
  - 区分顺序执行、一次选择与循环重复
- 认识 else if 与 switch 的适用情况，读懂多情况分支的实际路径
- 处理单次字符或整数输入，先确认是否读到值，再判断值能否用于当前任务
- 编写简单分支，用正常值和边界值检查并修正处理错误
- 在已有循环中读懂 continue 与 break 怎样改变后续执行

分支：从固定执行到选择自动化

程序 = 数据对象 + 操作 + 控制

- 分支实现**选择自动化**，使程序根据运行时条件决定执行路径。
- 条件选择使程序针对不同的数据状态执行不同操作，并根据输入情况采取相应处理。

1 条件分支：if 与 if ... else

根据库存追踪本次执行、跳过的语句与共同后续；从实例归纳分支写法，再增加付款要求，独立写出组合条件。

根据库存选择发货或等待

当前库存	固定版：只执行 puts("ship")	按任务应执行
stock=0	发货，不符合规则	等待补货
stock=3	发货，符合规则	发货

- stock 表示当前库存件数；**分支**根据条件结果，选择本次执行的语句。
- 固定版不检查库存；条件版用 stock > 0 选择处理。下面先观察 if 的执行或跳过，再用 else 补足等待处理。

判断 if 执行哪些语句

```
1 int stock = 0;
2 if (stock > 0) {
3     puts("ship");
4 }
5 puts("done");
```

输出词的含义

ship 表示发货；done 表示后续处理

当前值：stock=0

条件是真是假？哪些 puts 执行，哪些跳过？

完成或跳过花括号内的处理后，下一条执行的语句在哪里？

只把 stock 的初值改为 3

重新判断条件真假和语句是否执行，说明哪些结果改变

if：执行或跳过受控语句

当前 stock	stock > 0	puts("ship")	puts("done")
0	假	跳过	执行
3	真	执行	执行

- **受控语句**由 if 条件决定是否执行；本例是包含 puts("ship") 的花括号复合语句。条件为假时跳过它，后面的 puts("done") 仍执行
- 基本形式：if (表达式) 语句；分支选择本次处理，循环决定是否继续重复

判断 if ... else 执行哪个分支

```
1 int stock = 0;                    /* 可改为 3 */
2 if (stock > 0) {
3     puts("ship");
4 } else {
5     puts("wait");
6 }
7 puts("done");
```

新增的输出词

wait 表示等待；ship、done 的含义保持不变

分别代入 stock=0 和 stock=3

条件是真是假？本次执行哪组语句，哪组不执行？

检查结构后的语句

两种情况下，puts("done") 是否执行？

if ... else：两条分支与共同后续

stock	条件结果	执行的处理	未执行的处理	后续
0	假	wait	ship	done
3	真	ship	wait	done

- **执行路径**是程序本次运行实际经过的语句或分支；stock=0：判断 stock > 0 → 执行 puts("wait") → 执行 puts("done")
- 本例的**共同后续**是两种情况都会继续执行的 puts("done")，它位于整个 if ... else 结构之后；同一程序在运行时选择路径，称为**选择自动化**，分支本身不会反复执行。

if ... else 的语法形式

```
1 if (<condition>) {
2     <statements_if>
3 } else {
4     <statements_else>
5 }
```

条件与两组语句

condition 是条件表达式；0 为假，非 0 为真

statements_if 换成真时执行的语句

statements_else 换成假时执行的语句

else 与后续

else 不带条件；没有 else 时，条件为假就到结构后的语句

完成所选处理后，继续执行结构后的语句

占位符与花括号

尖括号内是须替换的占位内容，不能原样编译

花括号组成复合语句；单条受控语句可不加花括号

用条件表达发货规则

- 新规则：有库存且已付款时发货；否则等待
  - stock 仍表示库存件数
  - paid=1 表示已付款；paid=0 表示未付款
- 请写出符合这一规则的条件表达式
- 用写出的条件判断两种情况的真假和应执行的处理
  - stock=3, paid=0：条件是真是假，应发货还是等待?
  - stock=3, paid=1：条件是真是假，应发货还是等待?

库存与付款的组合条件

- “有库存且已付款”的条件是 stock > 0 && paid != 0
  - stock > 0 检查库存，paid != 0 检查付款；两个条件须同时为真
- stock=3, paid=0：库存条件为真，付款条件为假
  - 整体条件为假，执行等待处理
- stock=3, paid=1：两个条件都为真
  - 整体条件为真，执行发货处理
- 将这一条件放入 if 的圆括号内，选择发货或等待

2 else if 的执行顺序与 else 的匹配

多个范围按代码顺序检查；核对首个满足者与端点，再辨认嵌套 if 中 else 的匹配。

判断 else if 执行哪个分支

```
1 if (level >= 80) {
2     puts("high");
3 } else if (level >= 60) {
4     puts("mid");
5 } else if (level >= 40) {
6     puts("basic");
7 } else {
8     puts("low");
9 }
```

已知 level 在 0～100 内

80～100：high；60～79：mid；40～59：basic；0～39：low

四个判断值：level=80、level=60、level=40、level=39

条件按什么顺序检查？本次执行哪个分支？

选中一个分支后，后面的条件还会检查吗？

else if：执行首个满足条件的分支

level	首个为真的条件	实际处理
80	level >= 80	high
60	level >= 60	mid
40	level >= 40	basic
39	三项均为假	low

- 首个条件为真后，不再判断后面的 else if；建议保留花括号，便于辨认范围和以后添加语句
- level=80 的执行路径：判断 level >= 80 → 执行 puts("high") → 到达整个结构之后

嵌套 if 中 else 的匹配

- 嵌套代码须先确定 else 属于哪个 if，再判断执行哪些语句
- A：else 匹配第 2 行 if；B：else 匹配外层 if。缩进不决定匹配

A

```
1 if (stock > 0)
2     if (stock >= 5)
3         puts("enough");
4 else
5     puts("reorder");
6 /* stock: 0→无输出; 2→reorder */
```

B

```
1 if (stock > 0) {
2     if (stock >= 5) {
3         puts("enough");
4     }
5 } else {
6     puts("reorder");
7 }
8 /* stock: 0→reorder; 2→无输出 */
```

3 switch 的匹配与执行

字符命令先确定匹配标签，再沿语句顺序追踪实际出口。

判断 switch 执行哪些语句

```
1 switch (cmd) {
2 case 'a':
3     puts("add");
4     break;
5 case 'd':
6     puts("delete");
7     break;
8 default:
9     puts("unknown");
10 }
11 puts("done");
```

命令含义

'a' 表示添加，'d' 表示删除，其他字符是未知命令

入口与出口

cmd='a' 与 cmd='x' 各从哪个标签进入，从哪里离开？puts("done") 是否执行？

只删除 case 'a' 后的 break

cmd='a' 的输出会怎样改变？

switch 的匹配、执行与 break

```
1 switch (cmd) {
2 case 'a':
3     puts("add");
4     break;
5 case 'd':
6     puts("delete");
7     break;
8 default:
9     puts("unknown");
10 }
11 puts("done");
```

对字符命令程序

cmd='a'：从 case 'a' 进入，输出 add；遇 break 后离开 switch，再输出 done

cmd='x'：从 default 进入，输出 unknown；到结构末尾后输出 done

只删除 case 'a' 后的 break，其余代码与控制值不变

cmd='a' 将依次输出 add、delete、done

进入后不再匹配后面的标签；下一个标签不会自动停止执行，这称为**贯穿**

判断依据

范围条件按 else if 顺序检查；离散整型值（含字符）可用 switch 匹配

switch 的语法形式

```
switch (<integer_expression>) {
case <integer_constant_expression>:
    <statements>
    break;
default:
    <statements>
}
```

- 尖括号内是英文占位名，模板不能原样编译
- integer_expression：整型控制表达式
- integer_constant_expression：整数常量表达式；可有多个 case，各标签值不能重复
- statements：实际处理语句
- default 可省略；break; 按需离开 switch

多个 case 共用一组处理

```
switch (cmd) {
case 'a':
case 'A':
    puts("add");
    break;
case 'd':
    puts("delete");
    break;
default:
    puts("unknown");
}
puts("done");
```

两个标签对应同一组语句

cmd='a' 或 cmd='A'：都输出 add，再遇 break 离开，输出 done

从 case 'a' 进入后，经过 case 'A' 标签继续执行；标签之间没有处理语句

共用处理与P2贯穿的区别

前例删去 break 后执行了其他命令的处理；本页两个命令本就共用处理

这里省略两个标签之间的 break，是有意共用处理

case 值须互不重复

'a' 与 'A' 是不同字符值；同一 switch 不能写两个相同值的 case

4 输入读取与有效性检查

先确认本次是否取得预期数据，再判断已读值是否符合任务规则。

用 getchar() 读取字符并检查 EOF

```c
int ch = getchar();
if (ch != EOF) {
    putchar(ch);
}
```

读取与保存

getchar() 每次读取一个字符；读到字符时返回字符值

用 int ch 保存返回值，以便保存字符值或 EOF

判断与输出

ch != EOF 为真时执行 putchar(ch)，输出读到的字符

换行也是字符；'\n' 表示换行字符

EOF 的含义

EOF 是输入结束或读取错误时的返回值，不是输入流中的普通字符

判断是否读到字符、是否执行输出

```
1 int ch = getchar();
2 if (ch != EOF) {
3     putchar(ch);
4 }
```

ch='A'

本次是否读到字符？ch != EOF 是真还是假？

putchar(ch) 是否执行？若执行，输出什么？

ch='\n'

本次是否读到字符？ch != EOF 是真还是假？

putchar(ch) 是否执行？若执行，输出什么？

ch=EOF

本次是否读到字符？ch != EOF 是真还是假？

putchar(ch) 是否执行？

字符与 EOF 对应的处理

```c
int ch = getchar();
if (ch != EOF) {
    putchar(ch);
}
```

ch='A'

读到字符；ch != EOF 为真，执行 putchar(ch)，输出 A

ch='\n'

读到换行字符；ch != EOF 为真，执行 putchar(ch)，输出换行

ch=EOF

未读到普通字符；ch != EOF 为假，跳过 putchar(ch)

换行属于字符，EOF 表示读取状态；两者不能混为一谈

判断整数是否读到、是否符合范围

```c
int n;
int converted = scanf("%d", &n);
if (converted == 1) {
    if (n >= 0 && n <= 9) {
        puts("in range");
    } else {
        puts("out of range");
    }
} else {
    puts("no integer");
}
```

读取一个整数

%d 读取一项整数；&n 指定保存位置

区分两个值

converted 是成功转换的项数；本例只读一项

n 保存转换成功的整数；任务允许 0～9

判断读取与范围

converted 为 1、0、EOF 时，何时可使用本次 n?

converted=1，分别取 n=0、n=10：读取成功吗？值在范围内吗?

输出词的含义

in range：范围内；out of range：范围外；no integer：未读到整数

整数读取与范围检查的结果

读取情况	本次整数值	读取是否成功	范围判断与输出
converted=1	n=0	是	是 → in range
converted=1	n=10	是	否 → out of range
converted=0	无本次新整数	否，格式未匹配	不判断 → no integer
converted=EOF	无本次新整数	否，输入失败	不判断 → no integer

- 先检查 converted == 1；成立后才检查 n >= 0 && n <= 9，读取成功不等于数值符合规则
- 未取得本次整数时，不把 n 当作新输入；本例只读一次，输出 no integer 后不重试

5 范围分支：编写与边界检查

典型值正确仍可能漏掉端点；依据规则找首次偏差，再独立构造并核查短分支。

检查范围条件是否符合规则

```
1 if (n > 0 && n <= 9) {
2     puts("in range");
3 } else {
4     puts("out of range");
5 }
```

任务规则

已成功取得整数 n；允许 0～9，其他整数在范围外

找到反例

选值比较规则预期、条件真假与实际路径

修改与复验

只改必要的一处；用哪些相邻值复验？

范围边界的遗漏与修正

n	任务规则	原条件 n > 0 && n <= 9	原路径
0	范围内	假	out of range
1	范围内	真	in range
10	范围外	假	out of range

- 任务规则包含下界，原条件的 > 却排除了下界；本例下界为 0。最小修正为 n >= 0 && n <= 9
- 用 -1、0、1、9、10 复验；典型值 1 正确，不足以证明端点 0 正确

按两段范围编写分支

- 已成功取得一个整数 n
  - 0～9：输出 small
  - 10～20：输出 large
  - 其他整数：输出 out of range
- 自行写出条件与处理路径
- 写出 -1、0、9、10、20、21 的预期路径与输出

范围分支的写法与边界核对

```
1 if (n >= 0 && n <= 9) {
2     puts("small");
3 } else if (n >= 10 && n <= 20) {
4     puts("large");
5 } else {
6     puts("out of range");
7 }
```

n	预期输出
-1	out of range
0	small
9	small
10	large
20	large
21	out of range

对照规则、条件和输出，检查两段范围的边界及范围外的值

这是一种可行写法；覆盖相同范围的等价分支也成立，一次正确输出不足以证明边界都正确

6 continue 与 break 的执行效果

追踪 continue 与 break：先找所在结构，再找下一处执行位置。

判断 continue 后的执行位置

```c
for (int i = 1; i <= 4; i++) {
    if (i == 2) {
        continue;
    }
    printf("%d\n", i);
}
puts("done");
```

当前值：i=2

i == 2 是真还是假？本轮会执行 continue 吗？

本轮余下语句

printf 是否执行？说明判断依据

下一处执行位置

下一处是 i++、i <= 4，还是 puts("done")？

continue：跳过本轮余下语句

```c
for (int i = 1; i <= 4; i++) {
    if (i == 2) {
        continue;
    }
    printf("%d\n", i);
}
puts("done");
```

本轮跳过

i=2 时执行 continue，跳过 printf，因此不输出 2

下一处执行位置

先到 for 头部的 i++；更新为 3 后，再判断 i <= 4

更新来自 for 的控制流程，continue 本身不执行 i++

完整输出与作用范围

依次输出 1、3、4；循环结束后再输出 done

continue 跳过本轮余下语句，随后继续循环控制；它不退出循环

判断 break 后的执行位置

```c
int i = 1;
while (i <= 4) {
    if (i == 3) {
        break;
    }
    printf("%d\n", i);
    i++;
}
puts("done");
```

当前值：i=3

i == 3 是真还是假？本轮会执行 break 吗？

本轮余下语句

printf 与 i++ 是否执行？说明判断依据

下一处执行位置

先找最近包围 break 的循环或 switch，再判断它离开哪里

下一处是 while 条件、循环体后续，还是 puts("done")？

break：离开最近包围的结构

```c
int i = 1;
while (i <= 4) {
    if (i == 3) {
        break;
    }
    printf("%d\n", i);
    i++;
}
puts("done");
```

本轮跳过与下一位置

i=3 时执行 break，printf 和 i++ 都不执行

离开 while 后执行 puts("done")，不再回到 while 条件

区分 if 与 while 的作用

if 决定本次是否执行 break；break 离开最近包围它的循环或 switch，本例是 while

完整输出与作用范围

依次输出 1、2、done；离开循环不等于结束程序

最近包围它的是 switch 时，break 从该 switch 之后继续

continue 与 break 的执行区别

语句	被跳过的后续	下一处执行位置	结构是否结束
continue	本轮循环体余下语句	for：轮后更新；while / do while：条件检查	否，继续循环控制
循环中的 break	该循环体内的后续语句	最近循环之后	是，离开该循环
switch 中的 break	该 switch 内的后续语句	最近 switch 之后	是，离开该 switch

- continue 找最近的循环；break 找最近的循环或 switch；再确定被跳过的语句和下一处执行位置
- goto 可跳到同一函数内的标签语句；通常不建议用它组织一般控制流程

本讲总结

- 分支实现选择自动化，运行时条件决定本次处理路径
  - 同一库存程序：0 件时等待，3 件时发货；固定发货不能兼顾两种情况
  - if / if ... else 选路径，else if 取首个满足者，switch 按标签进入
- 读取成功与值符合规则是两项判断
  - getchar 查 EOF，单项整数 scanf 查成功转换数；取得值后再检查范围
- 循环安排重复，分支选择本次处理，二者可以结合
  - continue 跳过本轮余下语句；break 离开最近适用的循环或 switch
- 用正常值和边界值比较规则与实际路径，找出首次不符的位置

本讲内容与教材的对应

《C Primer Plus》第六版	本讲阅读要求
第 7 章 7.1、7.2.3、7.6、7.7 节	重点阅读 if、else if、switch、continue、break 的实际路径
第 7 章 7.2.1 节；第 8 章 8.1、8.3、8.6 节	重点阅读字符输入、EOF、单项 scanf 的读取状态与基本检查
本讲暂不要求	输入缓冲区清理、混合输入、反复提示重输；逻辑运算细则沿用 PF-03