

C 语言和程序设计思想 I

郑艺林

计算机科学与工程系

2016 年 11 月 25 日

课程内容 I

1 C 控制语句：分支和跳转

- 逻辑运算符
- if/else 语句
- 多层嵌套 if/else 语句
- switch 语句
- 跳转神器 continue 和 break
- 打死不用的 goto
- 避免 goto
- 三目运算符 ?:
- *ctype.h 头文件

2 字符的输入和输出

- getchar() 和 putchar()
- *缓冲区与非缓冲区



课程内容 II

- 终止键盘输入
- 重定向
- 用户界面的创建
- 输入确认
- 菜单浏览

3 函数

- 函数原型、定义
- 头文件
- 函数参数
- 函数返回值
- 函数调用
- 一元运算符 & *
- 指针初步



课程内容 III

■ 递归

4 作业

■ 作业



课程内容

1 C 控制语句：分支和跳转

- 逻辑运算符
- if/else 语句
- 多层嵌套 if/else 语句
- switch 语句
- 跳转神器 continue 和 break
- 打死不用的 goto
- 避免 goto
- 三目运算符 ?:
- *ctype.h 头文件

2 字符的输入和输出



逻辑运算符

逻辑运算符用以连接逻辑命题以形成更复杂的逻辑语句

逻辑运算符

逻辑运算符用以连接逻辑命题以形成更复杂的逻辑语句

■ &&

逻辑运算符

逻辑运算符用以连接逻辑命题以形成更复杂的逻辑语句

■ &&

■ ||

■ ?



逻辑运算符

逻辑运算符用以连接逻辑命题以形成更复杂的逻辑语句

■ &&

■ ||

■ ?

■ 优先级：|| < && < !



逻辑对应关系

运算符	含义
&&	与
	或
!	非



真值判断

逻辑运算符 < 关系运算符



真值判断

逻辑运算符 < 关系运算符

■ $2 < 3$



真值判断

逻辑运算符 < 关系运算符

■ $!2 < 3$

■ $1 + 2 < 4 || 2^3 == 8$



离散数学之逻辑符号

或、且、非

离散数学之逻辑符号

或、且、非

■ V



南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

离散数学之逻辑符号

或、且、非

■ \vee

■ \wedge

■ \neg



if/else 语句

■ if 语法结构

```
if(expression)
    statements
```

if/else 语句

■ if 语法结构

```
if(expression)
    statements
```

■ if/else 语法结构 I

```
if(expression)
    statements
else
    statements
```



if/else 语句

■ if 语法结构

```
if(expression)
    statements
```

■ if/else 语法结构 I

```
if(expression)
    statements
else
    statements
```

■ if/else 语法结构 II

```
if(expression)
    statement
else if(expression)
    statements
else
    statements // no necessary
```



多个并列 if 和 if/else 区别

■ 多个并列 if

```
if(expression)
    statements
if(expression)
    statements
if(expression)
    statements
...
```



多个并列 if 和 if/else 区别

■ 多个并列 if

```
if(expression)
    statements
if(expression)
    statements
if(expression)
    statements
...
```

多个 if 语句有多重选择的作用，覆盖范围可以不完全；
if-else 可以覆盖全部情况。



多层嵌套 if/else 语句

■ 语法结构 I

```
if(expression)
    statements
    if(expression)
        statements
        if(expression)
            statements
            . . .
```



多层嵌套 if/else 语句

■ 语法结构 I

```
if(expression)
    statements
    if(expression)
        statements
        if(expression)
            statements
        ...
```

■ 语法结构 II

```
if(expression)
    statements
else if(expression)
    statements
    else if(expression)
        statements
    ...
```



switch 语句

switch 语句结构提供多重选择

switch 语句

switch 语句结构提供多重选择

■ 语法结构

```
switch(integer expression)
{
    case constan1:
        statements    //optinal
    case constan2:
        statements    //optinal
    default:
        statements    //optinal
}
```



标签

integer expression 和 case 标签都是整型值

标签

integer expression 和 case 标签都是整型值

■ ‘a’



标签

integer expression 和 case 标签都是整型值

- 'a'
- 6



多重标签

case 标签可以是多重的

多重标签

case 标签可以是多重的

■ e. g.

```
switch('c')
{
    case 'a':
    case 'A':
        statements
    case 'b':
    case 'B':
        statements
    case 'c':
        statements
    default:
        statements
}
```



continue

continue 可用于循环结构，跳过当次迭代的剩余部分

continue

continue 可用于循环结构，跳过当次迭代的剩余部分

```
■ int i = 0, sum = 0;
  while(i<10)
  {
      sum += i;
      if(i++ % 2 != 0)
          continue;
      i++;
  }
  printf("%d\n", sum);
```



break

break 可用于循环可跳出当前循环层



break

break 可用于循环可跳出当前循环层

■ e. g.

```
int factors = 1, i;
for(i = 1; i < 10; i++)
{
    factors *= i;
    if(factors > 250)
        break;
}
```



switch 中的 break

break 是 switch 中不可缺少的部分，用于停止遍历



switch 中的 break

break 是 switch 中不可缺少的部分，用于停止遍历

■ e. g.

```
switch('c')
{
    case 'a':
    case 'A':
        statements
        break;
    case 'b':
    case 'B':
        statements
        break;
    case 'c':
        statements
        break;
    default:
        statements
}
```



被废的 goto

用于跳转到某一个行代码

被废的 goto

用于跳转到某一个行代码

- 包含 goto 和一个标签名称

```
goto part2;  
part2: printf("Hello Latex!");
```



使用其他结构代替 goto

C 没有它会更好



使用其他结构代替 goto

C 没有它会更好

■ 选择

```
if(size > 12)
    goto a;
goto b;
a: cost = cost * 1.05;
    flag = 2;
b: bill = cost * flag;
```



使用其他结构代替 goto

C 没有它会更好

■ 选择

```
if(size > 12)
    goto a;
goto b;
a: cost = cost * 1.05;
    flag = 2;
b: bill = cost * flag;
```

■ if 代替



使用其他结构代替 goto

C 没有它会更好

■ 选择

```
if(size > 12)
    goto a;
goto b;
a: cost = cost * 1.05;
    flag = 2;
b: bill = cost * flag;
```

■ if 代替

```
if(size > 12)
{
    cost = cost * 1.05;
    flag = 2;
}
bill = cost * flag;
```



使用其他结构代替 goto

使用其他结构代替 goto

■ 不确定循环 - while

使用其他结构代替 goto

- 不确定循环 - while
- 跳到末尾开始下一轮循环 - continue



使用其他结构代替 goto

- 不确定循环 - while
- 跳到末尾开始下一轮循环 - continue
- 跳出循环 - break



使用其他结构代替 goto

- 不确定循环 - while
- 跳到末尾开始下一轮循环 - continue
- 跳出循环 - break

注意：千万不要随意跳到程序的其他任何部分！禁用 goto
!!!

三目运算符 ?:

需要三个操作数，实现二选一的功能



三目运算符 ?:

需要三个操作数，实现二选一的功能

■ e. g.

```
int a = 10 > 11 ? 10 : 11;
```



三目运算符 ?:

需要三个操作数，实现二选一的功能

■ e. g.

```
int a = 10 > 11 ? 10 : 11;
```

相当于一个 if-else 语句组合



字符判断函数

字符判断函数

■ isalnum() - 字母或数字

字符判断函数

- isalnum() - 字母或数字
- isalpha() - 字母

字符判断函数

- isalnum() - 字母或数字
- isalpha() - 字母
- isdigit() - 阿拉伯数字



字符判断函数

- isalnum() - 字母或数字
- isalpha() - 字母
- isdigit() - 阿拉伯数字
- islower() - 小写字母



字符判断函数

- isalnum() - 字母或数字
- isalpha() - 字母
- isdigit() - 阿拉伯数字
- islower() - 小写字母
- isupper() - 大写字母

符合判断条件，返回真值。

字符映射函数

包含一系列分析字符的函数

字符映射函数

包含一系列分析字符的函数

■ `tolower()` - 大写转小写

字符映射函数

包含一系列分析字符的函数

- tolower() - 大写转小写
- toupper() - 小写转大写

字符映射函数

包含一系列分析字符的函数

■ tolower() - 大写转小写

■ toupper() - 小写转大写

不符合转化条件，则返回原值

课程内容

1 C 控制语句：分支和跳转

2 字符的输入和输出

- `getchar()` 和 `putchar()`
- *缓冲区与非缓冲区
- 终止键盘输入
- 重定向
- 用户界面的创建
- 输入确认
- 菜单浏览

3 函数



getchar () 和 putchar ()

使用头文件 <stdio.h>



getchar () 和 putchar ()

使用头文件 <stdio.h>

- `getchar()` - 读取一个字符



使用示例

■ e. g.

```
#include <stdio.h>
int main(void)
{
    char ch;

    while ((ch = getchar()) != '#')
        putchar(ch);
    return 0;
}
```



使用示例

■ e. g.

```
#include <stdio.h>
int main(void)
{
    char ch;

    while ((ch = getchar()) != '#')
        putchar(ch);
    return 0;
}
```

它们并不是真正的函数，而是定义为预处理器宏，详见 16 章。

缓冲区

缓冲区

■ 延迟回显

缓冲区

- 延迟回显
- 字符被存储在临时存储区域

缓冲区

- 延迟回显
- 字符被存储在临时存储区域
- 按下回车/Enter 键使数据对程序可用



缓冲的优点

缓冲的优点

- 存储一块一起发送耗时少

缓冲的优点

- 存储一块一起发送耗时少
- 可以修正输入错误

缓冲的优点

- 存储一块一起发送耗时少
- 可以修正输入错误
- 使用回车/Enter 确认输入

缓冲的分类

- ### ■ 完全缓冲 - 缓冲区满时被清空



缓冲的分类

- 完全缓冲 - 缓冲区满时被清空
- 行缓冲 - 遇到换行字符时清空



非缓冲区

一些交互性程序需要非缓冲输入

文件、流和键盘输入

文件是一块存储信息的存储器区域

文件、流和键盘输入

文件是一块存储信息的存储器区域

- 读写、打开/关闭文件的库函数

文件、流和键盘输入

文件是一块存储信息的存储器区域

- 读写、打开/关闭文件的库函数
- I/O 包

文件、流和键盘输入

文件是一块存储信息的存储器区域

- 读写、打开/关闭文件的库函数

- I/O 包

- 低级 I/O 包 - 操作系统基本文件工具



文件、流和键盘输入

文件是一块存储信息的存储器区域

- 读写、打开/关闭文件的库函数

- I/O 包

- 低级 I/O 包 - 操作系统基本文件工具

- 标准 I/O 包 - 函数的标准模型和标准集



文件、流和键盘输入

文件是一块存储信息的存储器区域

- 读写、打开/关闭文件的库函数

- I/O 包

- 低级 I/O 包 - 操作系统基本文件工具

- 标准 I/O 包 - 函数的标准模型和标准集



文件、流和键盘输入

文件是一块存储信息的存储器区域

- 读写、打开/关闭文件的库函数

- I/O 包

- 低级 I/O 包 - 操作系统基本文件工具

- 标准 I/O 包 - 函数的标准模型和标准集

使用标准 I/O 包可以屏蔽掉不用操作系统带来的差异



文件、流和键盘输入

C 处理流而不是文件

文件、流和键盘输入

C 处理流而不是文件

- 流是一个理想的数据流，实际的输入和输出映射到该数据流



文件、流和键盘输入

C 处理流而不是文件

- 流是一个理想的数据流，实际的输入和输出映射到该数据流
- e. g.



文件、流和键盘输入

C 处理流而不是文件

- 流是一个理想的数据流，实际的输入和输出映射到该数据流
- e. g.
 - 打开文件就是将流与文件相关联，通过流进行读写



文件、流和键盘输入

C 处理流而不是文件

- 流是一个理想的数据流，实际的输入和输出映射到该数据流
- e. g.
 - 打开文件就是将流与文件相关联，通过流进行读写
 - 流可以重定向



标准 I/O 包函数

标准 I/O 函数提供多个函数

标准 I/O 包函数

标准 I/O 函数提供多个函数

- `getchar()`

标准 I/O 包函数

标准 I/O 函数提供多个函数

- `getchar()`
- `putchar()`



标准 I/O 包函数

标准 I/O 函数提供多个函数

- `getchar()`
- `putchar()`
- `printf()`



标准 I/O 包函数

标准 I/O 函数提供多个函数

- `getchar()`
- `putchar()`
- `printf()`
- `scanf()`

标准 I/O 包函数

标准 I/O 函数提供多个函数

- `getchar()`
- `putchar()`
- `printf()`
- `scanf()`



标准 I/O 包函数

标准 I/O 函数提供多个函数

- `getchar()`
- `putchar()`
- `printf()`
- `scanf()`

`stdin` 流和 `stdout` 流



文件结尾

检测文件结尾的方式两种：

文件结尾

检测文件结尾的方式两种：

- 放置特殊字符 e.g. Ctrl + Z

文件结尾

检测文件结尾的方式两种：

- 放置特殊字符 e.g. Ctrl + Z
- 存储文件大小信息 e.g. Unix

文件结尾

检测文件结尾的方式两种：

- 放置特殊字符 e.g. Ctrl + Z
- 存储文件大小信息 e.g. Unix

文件结尾

检测文件结尾的方式两种：

- 放置特殊字符 e.g. Ctrl + Z
- 存储文件大小信息 e.g. Unix

C 让函数到达文件结尾时返回特殊值 EOF (End of File)



EOF

EOF

■ `getchar()` 和 `scanf()` 都返回 EOF

EOF

- `getchar()` 和 `scanf()` 都返回 EOF
- `define EOF -1`

EOF

- `getchar()` 和 `scanf()` 都返回 EOF
- `define EOF -1`

EOF

■ `getchar()` 和 `scanf()` 都返回 EOF

■ `define EOF -1`

e. g. `while((ch = getchar()) != EOF)`



示例代码

```
■ #include <stdio.h>
   int main(void)
   {
       int ch;
       while((ch = getchar()) != EOF)
           putchar(ch);
       return 0;
   }
```

注意 EOF 已经在 `stdio.h` 中定义，不必再定义
EOF 的实际值也不需要操心，可直接使用符号



重定向

重新指定数据的流向



重定向

重新指定数据的流向

■ 输入重定向

重定向

重新指定数据的流向

- 输入重定向
- 输出重定向

重定向

重新指定数据的流向

- 输入重定向
- 输出重定向
- 组合重定向

示例程序

编写并保存以下程序

```
■      #include <stdio.h>
      int main(void)
      {
          int ch;
          while((ch = getchar()) != EOF)
              putchar(ch);
          return 0;
      }
```



输入重定向

输入重定向

■ DOS 和 Unix/Linux 输入重定向运算符 <

输入重定向

- DOS 和 Unix/Linux 输入重定向运算符 <
- e. g. `echo_eof < words`



输出重定向

■ DOS 和 Unix/Linux 输出重定向运算符 >



输出重定向

- DOS 和 Unix/Linux 输出重定向运算符 >
- e. g. `echo_eof > words`

组合重定向

组合重定向

- e. g. `echo_eof > mywords < savewords`
`echo_eof < savewords > mywords`

组合重定向

- e. g. `echo_eof > mywords < savewords`
`echo_eof < savewords > mywords`

组合重定向

- e. g. `echo_eof > mywords < savewords`
`echo_eof < savewords > mywords`

注意：

输入不能来自多个文件

输出不能定向到多个文件

不能同时输入和输出到同一个文件

不能用于一个文件到另一个文件的连接

混合输入数字和字符

- 注意 `getchar()` 和 `scanf()` 的作用细节



预见输入错误



预见输入错误

■ 预见并提前处理可能的输入错误

预见输入错误

- 预见并提前处理可能的输入错误
- 给予必要的输入提示或条件



使用菜单说明程序功能



使用菜单说明程序功能

■ 引导用户使用

使用菜单说明程序功能

- 引导用户使用
- 调理更清晰，执行更流畅

课程内容

1 C 控制语句：分支和跳转

2 字符的输入和输出

3 函数

- 函数原型、定义
- 头文件
- 函数参数
- 函数返回值
- 函数调用
- 一元运算符 & *
- 指针初步
- 递归



函数

函数是用于完成特定任务的程序代码的自包含单元



函数

函数是用于完成特定任务的程序代码的自包含单元

■ 函数的功能

函数

函数是用于完成特定任务的程序代码的自包含单元

- 函数的功能

- 执行某些动作



函数

函数是用于完成特定任务的程序代码的自包含单元

■ 函数的功能

- 执行某些动作
- 返回需要的值



函数

函数是用于完成特定任务的程序代码的自包含单元

■ 函数的功能

- 执行某些动作
- 返回需要的值

■ 使用函数的好处



函数

函数是用于完成特定任务的程序代码的自包含单元

■ 函数的功能

- 执行某些动作
- 返回需要的值

■ 使用函数的好处

- 省略重复代码



函数

函数是用于完成特定任务的程序代码的自包含单元

■ 函数的功能

- 执行某些动作
- 返回需要的值

■ 使用函数的好处

- 省略重复代码
- 模块化程序



函数

函数是用于完成特定任务的程序代码的自包含单元

■ 函数的功能

- 执行某些动作
- 返回需要的值

■ 使用函数的好处

- 省略重复代码
- 模块化程序
- 便于修改、完善和维护



函数原型

函数的声明语法结构

函数原型

函数的声明语法结构

■ 函数原型

返回值 函数名(传入参数/void);

函数原型

函数的声明语法结构

■ 函数原型

返回值 函数名(传入参数/void);

- e.g. `int Add(int num1, int num2);`
`int Add(int, int);`



函数原型

函数的声明语法结构

■ 函数原型

返回值 函数名(传入参数/void);

- e.g. `int Add(int num1, int num2);`
`int Add(int, int);`



函数原型

函数的声明语法结构

■ 函数原型

返回值 函数名(传入参数/void);

- e.g. `int Add(int num1, int num2);`
`int Add(int, int);`

函数原型是必要的，告知编译器函数的类型



函数定义

函数定义

■ 函数定义

返回值 函数名(传入参数/void)

```
{
    statements;
}
```

■ e. g.

```
int Add(int num1,int num2)
{
    int sum = num1 + num2;
    return sum;
}
```



函数定义

■ 函数定义

返回值 函数名(传入参数/void)

```
{  
    statements;  
}
```

■ e. g.

```
int Add(int num1,int num2)  
{  
    int sum = num1 + num2;  
    return sum;  
}
```

函数定义确切指定了函数的具体功能



头文件

文件以 .h 结尾的文件是头文件

头文件

文件以 .h 结尾的文件是头文件

- 函数原型和常量经常在头文件中定义

头文件

文件以 .h 结尾的文件是头文件

- 函数原型和常量经常在头文件中定义
- 头文件中不包含函数的定义
函数的具体定义在另一个库函数文件中



函数参数

形式参数/形式参量

函数参数

形式参数/形式参量

- 形参是局部变量，作用域仅在函数体内



函数参数

形式参数/形式参量

- 形参是局部变量，作用域仅在函数体内
- 调用函数时形参会被赋值



函数参数

形式参数/形式参量

- 形参是局部变量，作用域仅在函数体内
- 调用函数时形参会被赋值



函数参数

形式参数/形式参量

- 形参是局部变量，作用域仅在函数体内
- 调用函数时形参会被赋值

带参数的函数使函数利于重复使用



函数的返回值

函数执行结束可以有返回值也可以没有

函数的返回值

函数执行结束可以有返回值也可以没有

- 基本的数据类型可以作为函数返回值，必须有 `return`



函数的返回值

函数执行结束可以有返回值也可以没有

- 基本的数据类型可以作为函数返回值，必须有 `return`
- 无返回值则使用关键词 `void`



return 的作用

return 具有多项作用

return 的作用

return 具有多项作用

- 返回函数返回值



return 的作用

return 具有多项作用

- 返回函数返回值
- 终止执行函数 - 只能用于 void 类型函数中



函数间的通信

调用带有参数的函数需要使用实际参数

函数间的通信

调用带有参数的函数需要使用实际参数

- 实际参数对形式参数进行赋值



函数间的通信

调用带有参数的函数需要使用实际参数

- 实际参数对形式参数进行赋值
- 类型必须对应



函数间的通信

调用带有参数的函数需要使用实际参数

- 实际参数对形式参数进行赋值
- 类型必须对应
- e. g. `Add(1, 2);`



函数间的通信

调用带有参数的函数需要使用实际参数

- 实际参数对形式参数进行赋值
- 类型必须对应
- e. g. `Add(1, 2);`



函数间的通信

调用带有参数的函数需要使用实际参数

- 实际参数对形式参数进行赋值
- 类型必须对应
- e. g. `Add(1, 2);`

注意不对应的类型赋值导致的问题



地址运算符 &

地址运算符的作用是取地址

地址运算符 &

地址运算符的作用是取地址

■ e.g. `int num; int * p; p = #`



地址运算符 &

地址运算符的作用是取地址

- e.g. `int num; int * p; p = #`
- 输出地址使用格式说明符 `%p`



间接运算符 *

定义指针变量时使用间接运算符 `*`，也叫取值运算符

间接运算符 *

定义指针变量时使用间接运算符 `*`，也叫取值运算符

■ e. g. `p = # *p ?`

间接运算符 *

定义指针变量时使用间接运算符 `*`，也叫取值运算符

■ e. g. `p = # *p ?`

间接运算符 *

定义指针变量时使用间接运算符 `*`，也叫取值运算符

■ e. g. `p = # *p ?`

`*p = 10;`

指针

指针是其值为地址的变量类型

指针

指针是其值为地址的变量类型

- 创建指针变量需要声明类型



指针

指针是其值为地址的变量类型

- 创建指针变量需要声明类型
- * 表明该类型是一个指针类型

一个错误示例

一个错误示例

■ 一个交换数值的程序

```
#include <stdio.h>
void Swap(int a, int b);
int main(void)
{
    int a = 10, b = 20;
    Swap(a,b);
    printf("a = %d\nb = %d\n",a, b);
    return 0;
}

void Swap(int a, int b)
{
    int temp = a;
    a = b;
    b = temp;
}
```



一个错误示例

■ 一个交换数值的程序

```
#include <stdio.h>
void Swap(int a, int b);
int main(void)
{
    int a = 10, b = 20;
    Swap(a,b);
    printf("a = %d\nb = %d\n",a, b);
    return 0;
}
void Swap(int a, int b)
{
    int temp = a;
    a = b;
    b = temp;
}
```

这是错误示例!!!



指针的一个使用示例

正确的程序如下

指针的一个使用示例

正确的程序如下

```
■ #include <stdio.h>
void Swap(int * a, int * b);
int main(void)
{
    int a = 10, b = 20;
    Swap(&a,&b);
    printf("a = %d\nb = %d\n",a, b);
    return 0;
}
void Swap(int * a, int * b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}
```



递归

函数调用本身的过程称为递归



递归

函数调用本身的过程称为递归

■ 使用示例

```
int Factor(int n)
{
    if( n == 0)
    {
        return 1;
    }
    return n*Factor(n-1);
}
```



递归

函数调用本身的过程称为递归

■ 使用示例

```
int Factor(int n)
{
    if( n == 0)
    {
        return 1;
    }
    return n*Factor(n-1);
}
```

思考一下



程序分析



程序分析

■ 执行过程

```
Factor(n)
    Factor(n-1)
        Factor(n-2)
            ...
                Factor(base case)
                return value
            ...
        return value to Factor(n-2)
    return value to Factor(n-1)
return value to Factor(n)
```



递归基本原理

递归有如下几点原理

递归基本原理

递归有如下几点原理

- 每一级函数调用都有自己的变量



递归基本原理

递归有如下几点原理

- 每一级函数调用都有自己的变量
- 代码不会被复制



递归基本原理

递归有如下几点原理

- 每一级函数调用都有自己的变量
- 代码不会被复制
- 每次函数调用都会有一次返回



递归基本原理

递归有如下几点原理

- 每一级函数调用都有自己的变量
- 代码不会被复制
- 每次函数调用都会有一次返回
- 位于递归调用前的语句和各级被调函数的执行顺序相同



递归基本原理

递归有如下几点原理

- 每一级函数调用都有自己的变量
- 代码不会被复制
- 每次函数调用都会有一次返回
- 位于递归调用前的语句和各级被调函数的执行顺序相同
- 位于递归调用后的语句和各个被调函数的执行顺序相反



递归基本原理

递归有如下几点原理

- 每一级函数调用都有自己的变量
- 代码不会被复制
- 每次函数调用都会有一次返回
- 位于递归调用前的语句和各级被调函数的执行顺序相同
- 位于递归调用后的语句和各个被调函数的执行顺序相反
- 递归函数中必须包含可以终止递归调用的语句



递归的应用

递归的应用广泛

递归的应用

递归的应用广泛

- 尾递归最简单

递归的应用

递归的应用广泛

- 尾递归最简单
- 递归可以用以反向计算 e. g. 十进制转化二进制输出



递归的优缺点

递归的优缺点

■ 递归优点



递归的优缺点

■ 递归优点

- 为某些问题提供了简单的解决方法



递归的优缺点

■ 递归优点

- 为某些问题提供了简单的解决方法
- 可简化代码



递归的优缺点

■ 递归优点

- 为某些问题提供了简单的解决方法
- 可简化代码

■ 递归缺点



递归的优缺点

■ 递归优点

- 为某些问题提供了简单的解决方法
- 可简化代码

■ 递归缺点

- 内存资源消耗大



递归的优缺点

■ 递归优点

- 为某些问题提供了简单的解决方法
- 可简化代码

■ 递归缺点

- 内存资源消耗大



递归的优缺点

■ 递归优点

- 为某些问题提供了简单的解决方法
- 可简化代码

■ 递归缺点

- 内存资源消耗大

谨慎使用递归！！！！



课程内容

1 C 控制语句：分支和跳转

2 字符的输入和输出

3 函数

4 作业

■ 作业



作业

- 第七章 7.11-1、2、3、6、7；7.12-3、5、8、11
- 第八章 8.10-2、4；8.11-2、8
- 第九章 9.10-2、7、8、9；9.11-2、8、10

