KUAS
KYOTO UNIVERSITY OF ADVANCED SCIENCE
京都発世界人財

京都先端科学大学

Information Processing 2 07

functions
return values
example: summing numbers
project: calculator

Ian Piumarta

Faculty of Engineering, KUAS

last week: control flow

semicolons and expression statements
`if`, `else`, `else if`; binary search
`switch`, `break`; count digits, whitespace, others
`while` and `for`; infinite loops
nested loops; sorting, sieve for primes
`reverse()` using comma operator
`do` loop
`break` and `continue`
simplifying loop conditions with `break`
labels and `goto`

Chapter 3. Control Flow
3.1 Statements and Blocks
3.2 If-Else
3.3 Else-If
3.4 Switch
3.5 Loops – While and For
3.6 Loops – Do-while
3.7 Break and Continue
3.8 Goto and Labels

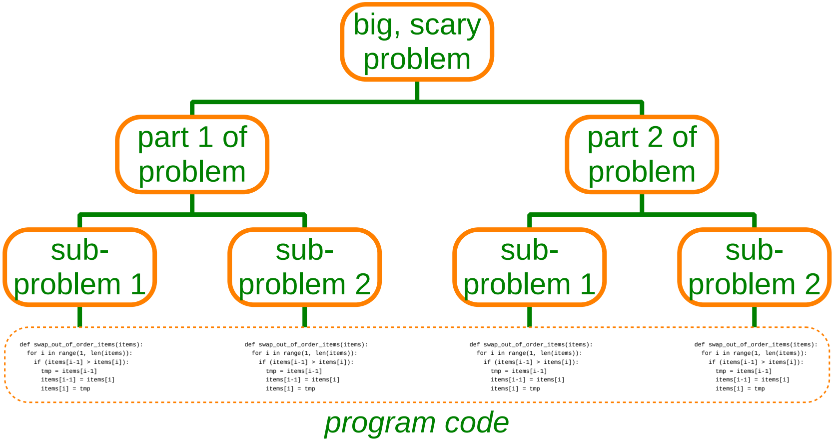
this week: functions and program structure

function to search file for string
- flowchart, pseudocode, code

factoring the program
returning non-integers
running total of floating-point numbers
stack data structure
Reverse Polish Notiation (RPN)
mini-project: RPN desk calculator

Chapter 4. Functions and Program Structure
4.1 Basics of Functions
4.2 Functions Returning Non-integers
4.3 External Variables

algorithm design: hierarchical decomposition



example: summing numbers read from the input

functions are ‘units of algorithm’ in a program

while we can **read another line** of text
  convert the line to a `double`
  add the result to a running total

print the total

```
while (getchars(line, sizeof(line)) >= 0) {
  double number = atod(line);
  total += number;
}
printf("%g\n", total);
```

`int getchars(char s[], int limit)`

read a line of text into `s`, without the final newline, and return the number of characters read or -1 at the end of input

`double atod(char s[])`

convert the text representation of a floating point number in `s` into its value as a `double`

getchars and atod

```
int getchars(char string[], int limit)
{
    int c, n = 0;
    while (n < limit - 1 && (c = getchar()) != EOF && c != '\n')
        string[n++] = c;
    if (c == EOF && n == 0) return -1;
    string[n] = '\0';
    return n;
}

double atod(char s[])
{
  int i = 0;
  double value = 0.0;
  while (isdigit(s[i])) value = value * 10.0 + s[i++] - '0';
  return value;
}
```

function definitions

return-type *function-name* (*parameter declarations*)
{
  declarations and statements
}

the *return-type* must be present

- if the function returns no value, use `void` as the return type

the *parameter declarations* can be

- a list of parameter declarations
  - when called, the arguments are converted to the parameter types
- the word `void` which means the functions take no arguments
- an empty list, which turns off all argument checking
  - you can pass any arguments you like, they will not be checked or converted

the *declarations and statements* can contain `return` statements

- if the *return-type* is not `void` then a compatible value must be provided

function prototypes and implementations

return-type function-name (*parameter declarations*) ← function *prototype*
{
  declarations and statements
} ← function implementation

the function *prototype* describes how the function is used

- the number and types of parameters that it requires
- the type of any result it returns

this does two things

1. it allows the function to be **implemented** properly
   - parameters are available within the body of the function
   - return values can be checked and converted to the correct return type
2. it allows the function to be **called** properly
   - number and types of arguments
   - type of returned result

function declarations

the function prototype can be used alone to declare a function without defining it

- the parameter types and result type are then known
- the compiler understands to call the function correctly

```
int i, atoi(char s[]); // declare int variable and function
double atod(char s[]); // declare double function
```

the parameter names can be omitted

- this is never ambiguous, even if sometimes it looks confusing

```
int i, atoi(char []), getchars(char [], int);
double atod(char []);
```

pro-tip: include parameter names, even in a function declaration

- use names that are descriptive, to ‘document’ parameter meanings

```
int getchars(char text[], int sizeOfTextInBytes);
```

forward declarations: using functions before defining them

after a function is declared, it can be used as if it were defined

declarations for `printf`, `getchar` and similar functions are in `stdio.h`

- after `#include <stdio.h>` they can be used safely, without warnings

you can declare your own functions and then use them before defining them

```
int getchars(char s[], int limit); // forward declaration

int main()
{
  while (getchars(line, LINEMAX) >= 0) { ... }
}

int getchars(char s[], int limit)  // definition
{
  ...
}
```

program structure (very generally)

```
#include <...>                     // included files with declarations
int x, y, f(void), g(int);        // declarations
int z = 42;                        // definitions
int main() { return 0; }           // main can be anywhere
int f(void) { return 123; }       // order of definitions not important
int g(int i) { return i + 1; } // if all functions declared at start
```

project: a ‘reverse Polish’ calculator

we are used to arithmetic using *infix* notation

- the operators are placed **between** the operands
- e.g: `1 + 2` $\Rightarrow$ 3
- e.g: `1 + 2 * 3 + 4` $\Rightarrow$ 11

reverse Polish uses *postfix* notation

- the operators are placed **after** the operands
- e.g: `1 2 +` $\Rightarrow$ 3
- e.g: `1 2 3 * + 4 +` $\Rightarrow$ 11

two steps:

1. make a simple ‘adding machine’ that sums a sequence of numbers
2. convert it into a reverse Polish calculator

[3. convert it into an infix calculator, later]

first step: a simple adding machine

let the total be `0.0` ← `int getnum()`
while a number is read successfully
    add the number to the running total ← `+=`
print the total ← `printf()`

two sub-steps:

1. write the function `int getnum()` as a reusable 'sub-algorithm'
    - return `-1` at `EOF`, or any other value to indicate success and
    - store the number that was read in the global variable `number`
2. write the main algorithm as the `main()` function, using `getnum()`

writing the function `int getnum()`

let `number` be `0.0`

let `c` be the next input character

while `c` is a blank (space or tab) read another character into `c`

while `c` is a digit:

    let `number` be `number * 10.0 + c - '0'`

    read another character into `c`

if `c` is `EOF` then return `-1`

return 0

useful character tests

```
#include <ctype.h>

int c;

  if (isblank(c)) ...
  if (isdigit(c)) ...
  if (isspace(c)) ...
  if (isalpha(c)) ...
  if (isalnum(c)) ...
```

→ c *is horizontal space or tab*

→ c *is decimal digit '0' … '9'*

→ c *is blank or newline*

→ c *is letter 'a' … 'z', 'A' … 'Z'*

→ c *is letter or digit*

adding machine main loop

let `sum` be `0.0`
while a number can be read ← `getnum()`
    add it to the `sum`
print the current value of `sum` ← `printf("\t%g\n", sum)`

project design: reading operators/numbers

to evaluate '`1 2 3 * + 4 +`' we need two new things:

1. a function to recognise either a number or an operator: `+`, `*`, etc.
2. a *data structure* to store intermediate results
3. a way to detect the end of line (end of an expression), as well as end of file

recognising numbers and operators: `int getop()` is like `getnum()`

- skip spaces and tabs
- if the current character `c` is `EOF`, return `-1`
- if the character is newline, return `0`
- if the character is not a digit, return the character (assume it is an operator)
- otherwise `c` begins a number, so
  - convert as many characters as possible into a floating-point `number`
  - return a special constant, e.g., `'0'` (`48`), to indicate a number was read

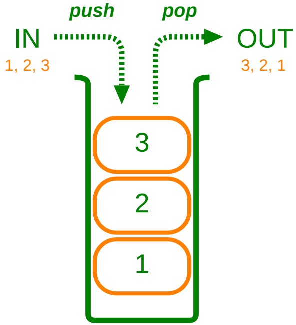
project design: data structure for evaluation

requirements to evaluate '`1 2 3 * + 4 +`':

1. save numbers one at a time
2. recall the most recent two, to perform an operation between them
3. save the result of the operation as the new most-recent number

a *stack* implements the required behaviour with these two operations:

- `push(n)` places `n` on the top of the stack
- `pop()` removes the number at the top of the stack



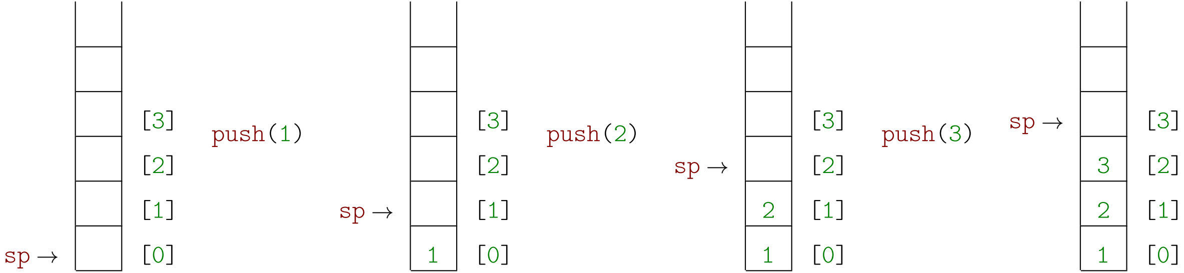
'last-in, first-out' or 'LIFO' behaviour

project design: stack implemented using array

stack contents kept in an array: `double stack[STACKMAX];`

‘stack pointer’ records the index of the next space in the stack: `int sp = 0;`

to `push(d)` use ‘`stack[sp++] = d`’ and to `pop()` use ‘`stack[--sp]`’



project design: data structure for evaluation

input	*stack*	*action*
`1`	`1`	push the operand `1`
`2`	`1 2`	push the operand `2`
`3`	`1 2 3`	push the operand `3`
`*`	`1 6`	pop the two most recent operands, multiply, push result
`+`	`7`	pop the two most recent operands, add, push result
`4`	`7 4`	push the operand `4`
`+`	`11`	pop the two most recent operands, add, push result

the functions `push()` and `pop()` are the *interface* to the stack data structure

- sometimes called an Application Programming Interface (API)

project design: calculator algorithm

forever:

- read an operator or `number` into `op` ← `op = getop()`
- if `op` is `EOF`, stop the program
- else if `op` is `'0'`, `push(number)` onto the stack
- else if `op` is `'+'`,
  - `pop` the top two numbers of the stack ← `r = pop(), l = pop()`
  - add them ← `l += r`
  - `push` the result back on the stack ← `push(l)`
- else if `op` is `0`,
  - `pop` and print each item on the stack ← `printf("\t%g\n", pop())`

of course, this is best done with a `switch` statement

project design: improvements

handle decimal point and fractional part of numbers

handle other arithmetic operators

handle stack underflow and overflow

do not discard the character after a floating-point number

next week: program structure, multiple source files

split calculator into several files
declarations in header files
`static` variables
block structure
defining variables within blocks
variable initialisation
recursive functions
`qsort()`, `swap()`
`#include`, `#define`, `#undef`, conditional compilation
[scope, storage classes, external, static, header files]

mini-project: make a Reverse-Polish Notation calculator

download the instructions from

MS Team “Information Processing 2”, “General” channel, “File” tab

exercises

write and test the function `getnum()` [2 points]

write and test the `getop()` function [2 points]

use an array to implement a stack of `double`s [2 points]

combine `getop()` and the stack into a calculator for additions [2 points]

extend the calculator with other operators [1 point]

read fractional part of floating-point numbers [1 point]

bonus: fix several limitations of the calculator to make it robust [+1 point]

13:00	5	last week: control flow
13:05	5	this week: functions, program structure
13:10	5	hierarchical decomposition
13:15	5	example: match lines
13:20	5	getchars function
13:25	5	strindex function
13:30	5	match lines program
13:35	5	function definitions
13:40	5	prototypes and implementations
13:45	5	function declarations
13:50	5	forward declarations
13:55	5	program structure
14:00	5	project: rpn calculator
14:05	5	adding machine
14:10	5	atof
14:15	5	character tests
14:20	10	adding machine main loop
14:30	10	*break*
14:40	5	project: read operators/numbers
14:45	5	data structure: stack
14:50	5	calculator algorithm
14:55	5	next week: multiple source files
15:00	70	exercises
16:10	00	end