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

京都先端科学大学

Information Processing 2 03

functions
arguments, call by value
character arrays

Ian Piumarta

Faculty of Engineering, KUAS

last week: character I/O and arrays

1.3 The `for` Statement
1.4 Symbolic Constants
1.5 Character Input and Output
1.6 Arrays

`for` statement and relationship to `while`
symbolic constants `getchar`, `putchar`, simple file copying
character counting
`if` statement
line counting
logical operators `&&` `||`
word counting
arrays, counting occurrences of each digit

review: variables and types

all variables must be given an explicit *type*

`char`	integer *just* big enough to store any character
`int`	a much bigger integer, for normal everyday use
`float`	floating-point (‘real’, sort of) value

variables can only store values of the appropriate type

variables *must* be initialised, otherwise they contain unpredictable junk

```
int n;      // bad: n is a random integer!
int n = 0; // good: we are certain that n contains 0
```

review: arrays

an array holds many values, all of the same type

```
int a[10];  // an array holding 10 integer values
```

values in arrays are accessed by their numeric `index`

```
a[3] = 6;
a[4] = 7;
a[5] = a[3] * a[4];  // a[5] is now 42
```

array indexes start from 0 (not 1)

valid indexes for an array of size N are from 0 to $N - 1$

- index N is just as illegal as index -1

(unlike Python, negative array indexes are illegal in C)

review: printing things

the function `printf()` prints things on the output (usually a terminal window)

the first argument is a *format* string, which is printed as-is, except for...

conversion specifiers '%*x*' inside the string cause data to be printed

successive data value come from successive additional arguments to `printf`

since everything is typed, you have to tell `printf` the type of every value printed

`%i` or `%d`	print a `int` in decimal
`%f`	print a `float` in decimal
`%c`	print a single character
`%s`	print a string

review: formatting printed things

a conversion specifier can say how to justify and pad the printed value

an integer after the % specifies a *field width*

`%6i`	print an `int` as 6 characters, right-justified
`%6f`	print a `float` as 6 characters, right-justified
`%.4f`	print a `float` with 4 decimal places in the fraction
`%9.4f`	print a `float` as 9 characters, right justified, with 4 decimal places

a negative field width causes the output to be left-justified within the field

review: characters and strings

characters are written inside single quotes: `'A'`, `'B'` `'C'`, etc.

characters are *integer* values, e.g: `'0'` = 48, `'A'` = 65, `'D'` = 68, etc.

the function `getchar()` reads a character from the input (usually keyboard)

the integer `EOF` indicates 'end of file'; its value is *outside* the range of character values

strings are written inside double quotes, `"like this"`

within characters and strings backslash is used to write special chaacters

`\n`	represents the newline character
`\t`	represents the tab character
`\'`	represents a single quote character
`\"`	represents a double quote character
`\\`	represents a single backslash character

review: boolean values and loops

in C, the integer 0 is *false* and any other value is *true*

```
while (condition) {
    ...
    loop body statements
    ...
}

for (setup; condition; update) {
    loop body statements
}
```

review: conditionals

```
if (condition) {
  statements for condition true
}
```

```
if (condition) {
  statements for condition true
}
else {
  statements for condition false
}
```

conditions can be inverted	`!x`	false if `x` is true, true if `x` is false
combined with 'or'	`x || y`	either `x` or `y` is true
combined with 'and'	`x && y`	both `x` and `y` are true

this week is a multiple of 3

small test!

this week: functions, line input, character arrays

1.7 Functions
1.8 Arguments-Call by Value
1.9 Character Arrays
1.10 External Variables and Scope

functions, `return`, call-by-value
character arrays
`getchars` and character array `copy`
print longest line

functions

functions let you

- encapsulate some computation
- give it a friendly name
- parameterise it (provide inputs to the computation)
- return a result (provide an output from the computation)

all function definitions follow the same pattern:

```
return-type function-name ( parameter-declarations )
{
    local variable declarations
    statements
}
```

functions

here is a program that calculates and prints 2^{10}

```
#include <stdio.h>

int main()
{
    int base   = 2;              // base and n are the inputs
    int n      = 10;             //   to the 'power' algorithm

    int result = 1;              // for n == 0, the result is 1

    while (n > 0) {
        result = result * base;  // multiply result by base, n times
        --n;
    }                            // result contains output from the algorithm

    printf("%d\n", result);

    return 0;
}
```

let's move the computation into a function that *returns* the result

functions

```
int power()
{
    int base   = 2;
    int n      = 10;

    int result = 1;

    while (n > 0) {
        result = result * base;
        --n;
    }

    return result;
}

int main()
{
    int result = power();

    printf("%d\n", result);

    return 0;
}
```

return-type function-name ()
{
  local variable declarations

  statements
}

`return` *expression* ;

lets you return a result from the function

the caller can then use your function in an expression

now let's make `base` and `n` be *parameters* of the `power` function

functions

```c
#include <stdio.h>

int power(int base, int n)
{
    int result = 1;

    while (n > 0) {
        result = result * base;
        --n;
    }

    return result;
}

int main()
{
    int result = power(2, 10);

    printf("%d\n", result);

    return 0;
}
```

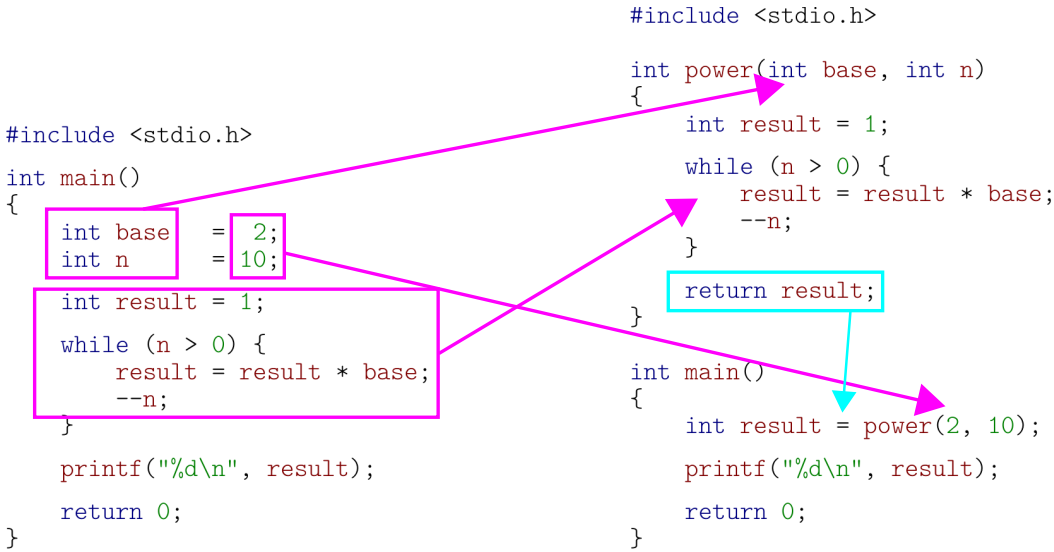
return-type function-name (*parameter-declarations*)
{
local variable declarations

statements
}

parameters behave just like local variables inside the function

parameters `base` and `n` are initialised from the arguments `2` and `10` when the function `power(2, 10)` is called

functions



caller, callee, and return values

if the `main()` function calls `printf()` then:

- `main` is the *caller*, initiating the function call
- `printf` is the *callee*, the function implementing the called function

the callee should `return` a value of the appropriate type to the caller

- the value returned can be used in an expression, or assigned to a variable

a function whose return type is `void` does not return any value to the caller

```
void printInteger(int n)
{
  printf("%d\n", n);
}
```

the compiler will complain if the caller tries to use the (non-existent) return value

arguments and parameters

arguments are the values supplied to a function when it is called

- in '`power(2, 10)`' the arguments are `2` and `10`

parameters are the variables inside the function that receive the argument values

- in '`power(int base, int n) {...}`' the parameters are `base` and `n`

when you call

`power(`x`, `y`)`

you can imagine

`base = `x`; n = `y`;`

is executed before the body of `power`

arguments, call by value

```
int power(int base, int n)
{
    int result = 1;

    while (n > 0) {
        result = result * base;
        --n;
    }

    return result;
}
```

variables defined inside a function are *local* to that function

- the variable `result` in `power` only exists inside the function
- it is not the same variable as `result` inside the `main()` function
- you can rename either of them and the program will still work perfectly

parameter variables are really local variables (and exist only inside the function)

- they are initialised at the time the function is called
- their initial values are the actual arguments supplied by the caller
- `power(2, 10)` initialises `base` to `2` and `n` to `10` inside the `power` function

call by value

the callee (function that is called) cannot directly alter the caller's arguments

- the callee function only has access to parameter variables
- parameter variables are *local* variables containing *copies* of the argument values

```c
int main()
{
  int a = 1, b = 2;

  printf("main: %d %d\n", a, b);
  swap(a, b);
  printf("main: %d %d\n", a, b);

  return 0;
}
```

```c
void swap(int a, int b)
{
  int aa = a;
  a = b;
  b = aa;

  printf("swap: %d %d\n", a, b);
}
```

note that `swap` is declared '`void`' to indicate that it does not return any value

arrays

arrays contain a sequence of values of the same type

declaration similar to a variable but you specify how many values the array can store

```
int variable = 42;  // a single integer variable
int array[32];      // an array of 32 integer variables
```

accessing array elements looks exactly the same as (e.g.) Python lists

```
array[0] = 42;        // store 42 in the first element
array[1] = array[0];  // copy first element to second
```

note how declarations are written in C:

- write an *expression* showing how to *access* a *value*, e.g., '`array[`n`]`'
- write the *type* of the value that would be accessed in front of the expression, e.g., '`int array[`n`];`'
- this convention is consistent for all declarations, no matter how complex

array arguments

when an array is used as an argument, a *copy* of the *location* of the array is passed

- this is the *address* in memory where the contents are stored
- the function can modify elements of the *original* array using subscripts (indexing)

```
int main() {
  int a[2] = { 1, 2 };

  printf("main: %d %d\n", a[0], a[1]);
  swap(a);
  printf("main: %d %d\n", a[0], b[1]);

  return 0;
}
```

```
void swap(int a[2]) // or: a[]
{
  int aa = a[0];
  a[0] = a[1];
  a[1] = aa;
}
```

note: the parameter `a` is the *location* of the array

- the elements are not copied, therefore
- it makes no difference to the function how many elements actually exist
- you can omit the number of elements, but you *must not* access illegal indexes

array results

you cannot declare a function to return an array as its result

in a few weeks we will see why this is not a limitation at all

characters

C really only has two kinds of data, integers and floats

characters are represented as integers

- almost always using the ASCII encoding
- on Linux (WSL) or Mac, type this into a terminal: `man ascii`

e.g., the character ‘A’ has integer value 65

you can write it as `65`, or as `'A'` which means *the same thing*

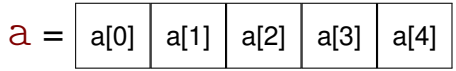
you can also write ‘strings’ like this: `"hello, world"`

- but this is really just an *array* of characters

character arrays

character arrays store characters

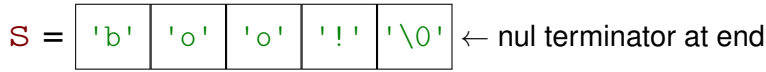
```
char a[5];  // an array containing 5 characters
```



a string is a kind of character array

- it contains one character per element
- the last element **must** be a 'nul' character with value 0 (often written `'\0'`)
- the nul character *terminates* the string (indicates the end of the string)
- a character array can be initialised with a string (including nul, if there is space)

```
char s[5] = "boo!";  // a holds a string of 4 characters
```



printing a character array

the `printf` format specification '`%s`' prints a string

- the next argument must be a string
- in other words, an array of characters *including* a nul terminator

```
#include <stdio.h>

int main()
{
  char s[5] = "boo!"; // s MUST be larger than the number of chars in the string
  printf("%s\n", s);  // otherwise the final nul character will not be present
  return 0;
}
```

mini project: printing the longest line in a file

let's practice character arrays by writing a program to print the longest input line

the algorithm will be:

```
while ( there is another line )
    if ( it is longer than the previous longest )
        save (copy) it as the longest line
        remember its length
print the longest line
```

to make this work let's create two useful functions:

1. `getchars()` will read a line from the input (as a string) into a character array
2. `copy()` will copy one character array into another

reading a string from the input

let's write `getchars` by first writing a program that reads a line of input

- integer constant `LINEMAX` will limit the longest line (including terminating nul)
- `char` array `line` will contain up to `LINEMAX` characters
- we will read input and store it into `line` until newline or `EOF` is read
- if a newline is read then it will be added to `line`

```
let line be an array of LINEMAX characters
let i be 0 (the index of the next character to add to line)
while ( next character is not newline or EOF )
    if ( there is room for more characters in s )
        store character at s[i]
        increment i
if ( character was newline )
    store character at s[i]
    increment i
store nul terminator character '\0' at s[i]
```

reading a string from the input

```c
#include <stdio.h>

#define LINEMAX 1000

int main()
{
    char line[LINEMAX];             // let line be an array of LINEMAX characters
    int c, i;                       // let i be 0 (index of next char to add to line)

    i = 0;
    while ((c = getchar()) != EOF && c != '\n') {  // next char is not EOF or newline
        if (i < LINEMAX - 2) {                     // there is enough room in line
            line[i] = c;                           // append char to line
            ++i;
        }
    }
    if (c == '\n') {                               // char is newline
        line[i] = c;                               // append char to line
        ++i;
    }
    line[i] = '\0';                                // store terminating nul char in line

    printf("%d characters: %s", i, line);

    return 0;
}
```

`getchars` function for reading a string from the input

let's convert the previous program into a parameterised function

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

where

- `line` is an array of `char`s to store the next input line
- `limit` is the number of elements in `line`
- we guarantee to add a terminating 'nul' character to `line`
- the result of `getchars` is the number of characters stored in `line`
  - *excluding* the final 'nul' character

and test it with

```
char line[LINEMAX];
int n = getchars(line, LINEMAX);
printf("%d characters: %s\n", n, line);
```

`getchars` function for reading a string from the input

```
int getchars(char line[], int limit)
{
    int c, i = 0;

    while ((c = getchar()) != EOF && c != '\n') {
        if (i < limit - 2) {
            line[i] = c;
            ++i;
        }
    }

    if (c == '\n') {
        line[i] = c;
        ++i;
    }

    line[i] = '\0';

    return i;
}
```

copying a string from one array to another

let's again write this by first writing a program to demonstrate the behaviour

```
let from be a character array containing a string
let to be a character array with the same number of elements as from
let i be 0 (the index of the next character to copy)

while ( copy from[i] to to[i] and character is not nul '\0' )
    increment i

print s
print t
```

copying a string from one array to another

```c
#include <stdio.h>

#define LINEMAX 1000

int main()
{
    char from[LINEMAX] = "hello, string";
    char   to[LINEMAX] = "-------------";
    int  i;

    printf("%s\n", from);
    printf("%s\n", to);

    i = 0;
    while ((to[i] = from[i]) != '\0')
        ++i;

    printf("%s\n", from);
    printf("%s\n", to);

    return 0;
}
```

copy function for moving a string from one array to another

```
void copy(char to[], char from[])
{
  int i = 0;

  while ((to[i] = from[i]) != '\0')
    ++i;
}

int main()
{
  char s[32] = "hello world";
  char t[32] = "-----------";

  copy(t, s);

  printf("%s\n", s);
  printf("%s\n", t);

  return 0;
}
```

printing the longest line in a file

```
int main()
{
  char line[LINEMAX];      // current input line
  char longest[LINEMAX];   // longest saved line
  int  len;                // length of current line
  int  max = 0;            // length of longest saved line

  while ((len = getchars(line, LINEMAX)) > 0) {     // while there is another line
    if (len > max) {                                //    if it is longer than previous longest
      copy(longest, line);                          //       save (copy) it to longest line
      max = len;                                    //       remember its length
    }
  }

  if (max > 0)                                      // print the longest line
    printf("%s", longest);

  return 0;
}
```

next week: types and expressions; declarations and initialisation

variable name rules
`char int float double short long`
integer constants, characters, ASCII, `'0'` vs. `0`
escape sequences, constant expressions
string constants, array of characters, `'x'` vs `"x"`
the `strlen()` function, `<string.h>`
enumeration constants, `enum`
advantages of `enum` over `#define`
declarations and initialisation
operators: arithmetic, relational
increment and decrement operators
assignment operators
type conversions and cast
`lower()`

Chapter 2. Types, Operators, and Expressions
2.1 Variable Names
2.2 Data Types and Sizes
2.3 Constants
2.4 Declarations
2.5 Arithmetic Operators
2.6 Relational and Logical Operators
2.7 Type Conversions
2.8 Increment and Decrement Operators
2.9 Bitwise Operators
2.10 Assignment Operators and Expressions
2.11 Conditional Expressions
2.12 Precedence and Order of Evaluation

exercises

please download assignment from

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

exercises

powers of floating-point numbers (using $x^y = e^{y \log_e x}$)

table of prime numbers

counting prime numbers

nested loops: multiplication table

printing long lines

reversing the characters in an array

bonus: trimming blanks from the start and end of lines

bonus: copying strings and using small input buffers