comp10002
Foundations of Algorithms

Semester Two, 2019

More C

Lecture slides prepared by Alistair Moffat

Overview

Chapter 4

Chapter 5

Chapter 6

Chapter 4 – Concepts

the loop
to property want to achieve in the end

Loops should have an invariant that is trivially satisfied by the initialization; is refined at each iteration; and when combined with the negation of the guard, represents the desired outcome.

eg. if $x > 0$

Where a loop has alternative exit conditions, or multiple exit points, the subsequent computation paths will also differ.

Count from zero whenever possible, up to (but not including) `n`.

Loops can iterate over the input, either value by value, or character by character.

- forloop1.c
- daynumber.c
- forloop2.c
- forloop3.c
- savings.c
- daynumber-squash.c
- threen.c
- isprime.c
- readloop1.c
- fortcomm.c

(remove comment)

int i first execution (test guard condition)

②second execution

for (i=1 ; i<=NUM_LINES ; i+=1){

① ② ③ body ④ ⑤ ⑥ ⑦ ⑧

execute the third statement

print(..

}

return 0

to the end.

while enter the number one by one

while (scan("%d, &n")==1)){

}

scanf ("%d", &n) != 1 → to test if ~~the~~ only 1 number input

:

isprime = 1 → assume that the number is prime

for (divisor = 2 ; divisor * divisor <= n ; divisor ++)

clever point

or #define TRUE (1==1)

TRUE 1

or

3 4 5 six 7 8 ⇒ sum 12

not a number , exist and terminate

when no number
product 1
sum 0
max not defined

Chapter 4 – Exercise

or if (isprime){
}

from I=1 to I=1000
100 DO 300 I = 1, 1000
?

fortcomm < fortran.f

do this program on that file

C xxx (comment)

Exercise

Write a nesting of loops that reads numbers from `stdin`, and for each value read, computes and prints the sum of the primes that are less than or equal to it, and then whether or not that sum is itself prime.

Chapter 4 – Summary

not 0 ⇒ true

while (n>1) ⟺ for (; n>1;)

while ( ){ } ⇒ the programme will run forever

Key messages:

- `for` loops and `while` loops *feel* different, but are almost identical
- Loop guards are integer-valued expressions; loop initializers and iterators are expressions
- Loops can iterate over input data through the use of the return value from `scanf`
- Avoid `do` loops
- Use a consistent layout and style to make your programs readable.

Chapter 5 – Concepts

declare function

→ function will return int value

int isprime (int n)

↑ import int

if (isprime(n)) {

↓ return of isprime (n)

↓ is nonzero integer

printf ("the next prime = %d\n", nextprime(n));

return 0;

}

Functions provide abstraction, to complement calculation, selection, and iteration.

Like all variables and constants, functions have a type signature that is declared in advance of their use.

Functions can be separately compiled to make modules and libraries.

There is a wide range of standard C function libraries, for mathematical computation, character processing, string handling, etc.

Chapter 5 – Program examples

```c
/* Show the use of math library functions and constants.
*/
#include <stdio.h>
#include <math.h>

int
main(int argc, char *argv[]) {
        double x;
        printf("Enter a value for x: ");
        scanf("%lf", &x);
        printf("sin(x)  = %.15f\n", sin(x));
        printf("log(x)  = %.15f\n", log(x));
        printf("fabs(x) = %.15f\n", fabs(x));
        printf("sqrt(x) = %.15f\n", sqrt(x));
        printf("M_PI    = %.15f\n", M_PI);
        printf("M_SQRT2 = %.15f\n", M_SQRT2);
        return 0;
}
```

- savingsfunc.c
- isprimefunc.c
- usemathlib.c
- savingsfuncgen.c
- triangle.c
- hanoi.c
- croot.c
- evenodd.c

```c
/* Read a number and determine if it is prime.
*/
#include <stdio.h> → library

int isprime(int n);
int nextprime(int n);

int
main(int argc, char *argv[]) {
        int n;
        printf("Enter a number n: ");
        scanf("%d", &n);
        if (isprime(n)) {
                printf("%d is a prime number\n", n);
        } else {
                printf("%d is not a prime number\n", n);
        }
        printf("The next prime is : %d\n", nextprime(n));
        return 0;
}

/* Determine whether n is prime. */
int              define a function
isprime(int n) {
        int divisor;
        if (n<2) {
                return 0;
        }
        for (divisor=2; divisor*divisor<=n; divisor++) {
                if (n%divisor==0) {
                        /* factor found, so can't be prime */
                        return 0;
                }
        }
        /* no factors, so must be prime */
        return 1;
}

int
nextprime(int n) {                       isprime(n) return 0
        n = n+1;
        while (!isprime(n)) {                  add to test next
                n = n+1;                              number
        }
        return n;
}
```

Chapter 5 – Calling a function

Argument expressions are evaluated in the calling context.

Argument values are copied into local variables in function.

Function executes until `return` or end reached.

Return expression, if any is computed in context of function.

Function exits, all local variables destroyed.

Return value is made available in calling context.

Chapter 5 – Calling a function

```
int
main(int argc, char *argv[]) {
   int n, m, val;
   /* assign values to n and m */
   ...

   val = func(n, m);

   /* now use val */
   ...

   return 0;
}
```

```
int
func(int x, int y) {
   int ans;
   /* compute ans from x and y */
   ...

   return ans;
}
```

Chapter 5 – Parameter passing

Variables (and expressions) are passed as values, copied into local variables.

When pointers are required in a function, they are constructed as pointer expressions, then copied into local variables (Chapter 6).

Arrays are passed as pointers to the first element in the array (Chapter 7); `struct`'s are passed as values, and copied into local variables (Chapter 8).

All non-`static` variables are destroyed when the function returns. Best to avoid `static` variables if possible.

Exercise:

Write a function that takes three `int` arguments and returns the median (middle) one.

Chapter 5 – Summary

Key messages:

- Functions provide a mechanism for abstraction
- The values of arguments to functions are copied in to local variables at the time the function commences
- Changes in the function to arguments do *not* affect variables in the calling context
- Recursion provides another form of iteration.

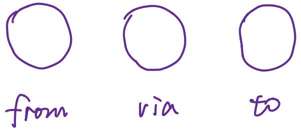
Summary of Chapters 2 to 5

C and Python are alike, because:

- They are imperative languages
- They offer a range of arithmetic and logical operations
- They offer a range of control structures, including selection, iteration, and recursion
- Function arguments are received as initial values of local variables
- Libraries are available for a wide range of other operations.

Summary of Chapters 2 to 5

hanoi



hanoi (char from, char via, char to, int n) ;

C and Python are different, because:

- C program structure is indicated by semicolons and braces, Python program structure by layout
- C integer arithmetic is bounded, and silently overflows
- C does not have an explicit `bool` type, and uses `int` 0 or 1
- C has static typing and requires declarations, Python has dynamic typing
- C is usually compiled, Python is usually interpreted when we use C, use compiler
- Python provides in-built list, set, and dictionary structures, and operations on them
- C provides explicit pointer variables and pointer operations.

Chapter 6 – Concepts

All variables and compound structures are mapped to addresses in memory via execution-time pointer values.

C provides operations that manipulate pointer values, including `&`, `*`, `+`, and (Chapter 8) `->`.

Pointer variables and expressions derive their types from the underlying variables. So `int*` is type "pointer to `int`".

store address in memory, in that address we hope to find integer

Functions that need to alter their arguments must receive pointers; the corresponding call must provide addresses of variables of the same type.

The declaration `void*` allows untyped pointers.

Chapter 6 – Concepts

The scope rules determine which variables can be accessed at each point in a program.

Program data segment
– store global memory

Variables declared in a function are local, or automatic; variables declared outside any function are global.

Stack

– local memory
function call

Argument variables are considered to be local to the function, but can also be shadowed by local variables declared within the function.

Local and global variables can also be declared with the modifier `static`. Static variables are initialized once, and thereafter retain their value through the execution.

Chapter 6 – Program examples

&w

what is the address of memory of variable w

%16p.
right justified
16 places

int x, y, z;
⇓
take 4 bytes increasing

xxxx16
xxxx18
xxxx14

- void.c → global & local variable
- scope1.c
- scope2.c → initialize z=5 outside the main ⇒ global variable. avoid change inside of function call
- scope3.c
- scope4.c static int z=7 ⇒ almost global variable for the only call to this function
- pointer1.c
- pointer2.c local variable → static variable → global variable → error
- pointer3.c
- readnum.c

int *pi (pointer). (4 bytes) pointer store address

eg. double *d (8 bytes)

*pi ⇒ reference, point to the oringial thing of that address.

*pi = *pi + 1 → get the value, assign +1.

pi= pi++

⇓

change the address

Case Study

Write a function that reads integers until it obtains one in the range given by its first two arguments. When a suitable value is read, it stores that value using its third argument, and returns the predefined constant `READ_OK`. If no suitable value is located, the predefined constant `READ_ERROR` should be returned.

- `readnum.c`

Exercise

Write a function that orders its three `int` arguments from smallest to largest.

Chapter 6 – Summary

The scope rules determine which variables can be accessed at each point of a program.

Pointer arguments allow functions to to make changes to variables in the calling environment.

This facility is sometimes called call by reference; the alternative is call by value (which in fact is what C always does).

Pointers provide a mechanism for aliasing. It is a flexibility that is extremely useful, not just in functions, but needs to be treated with respect.

In C, pointers and arrays go hand in hand.