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

京都先端科学大学

Information Processing 2 06

statements and blocks
conditional statements
loops, `break` and `continue`
labels and `goto`, `switch`

Ian Piumarta

Faculty of Engineering, KUAS

last week: declarations, operators, precedence

declarations and initialisation
operators: arithmetic, relational
increment and decrement operators
bitwise logical operators
assignment operators
conditional expressions
operator precedence and associativity
sequence points during expression evaluation
ambiguous (undefined) expressions

review: integers

integer constants have type `int` by default

to make a constant be	`unsigned`	add ‘`U`’ to the end
	`long`	add ‘`L`’ to the end
	`unsigned long`	add ‘`UL`’ to the end

integers are represented in decimal (base 10) by default

octal, hexadecimal, and binary are useful too:

binary	base 2	`0b101010`
octal	base 8	`052`
decimal	base 10	`42`
hexadecimal	base 16	`0x2A`

review: evaluation order and strings

C does not specify the order of operand evaluation

beware of sequence-dependent (and therefore undefined) expressions:

- `f(i++, i++)`
- `i + i--`
- etc.

a string is an array of `char` that is *terminated* with a nul (character 0)

every string is an array of `char`

an array of `char` is not a string

- unless you ensure it is always terminated with a nul
- leave an additional space for it in an array if you have to
  (the compiler will do this for you if initialise from a string and leave the size blank)

```
char a[6] = "string"; // a is not a string (no space for nul)
char s[7] = "string"; // s is a string (one extra element for nul)
```

test

this 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`

control flow

the order in which computations are performed

default order is linear, top to bottom

control statements modify the flow of control

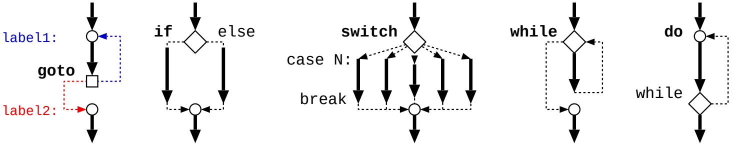
conditional statements: `if`, `else`

jumps: `goto`, `switch`

loops: `for`, `while`, `do`

default control flow

```
a = 3;
b = 4;
h = sqrt(a*a + b*b);
```



statements and blocks

any expression becomes a statement when followed by a semicolon

expression	$\rightarrow$	*statement*
`x = 0`		`x = 0;`
`i++`		`i++;`
`printf(...)`		`printf(...);`

braces { and } group declarations and statements into a *block*

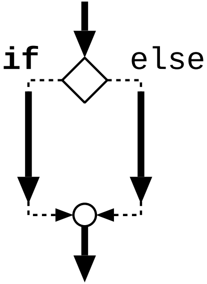
- also called a *compound statement*
- equivalent to a single statement
- no semicolon after the 'closing' brace }

```
if (i < limit - 1) {
  int c = getchar();
  line[i] = c;
  ++i;
}
```

conditional statement: `if` and `else`

used to express decisions (the `else` part is optional)

```
if ( expression )       // 1. the expression is evaluated
     statement1         // 2. if expression is true (non-zero), statement1 is executed
else                    // if the else part is present, then
     statement2         // 3. if expression is false (zero), statement2 is executed
```



the *expression* is tested for its numeric value

coding shortcuts are possible:

`if (expression != 0)` $\leftrightarrow$ `if (expression)`

write the *expression* in the way that is most natural and clear

conditional statement: `if` and ambiguous `else`

nested `if`s with only one `else` could be ambiguous

- rule: an `else` is associated with the closest matching (`else`-less) `if`

```
if (n > 0)
  if (a > b)
    z = a;
  else // a <= b
    z = b;
```

```
if (n > 0)
  if (a > b)
    z = a;
else // n <= 0
  z = b;
```

← *wrong!*

recommended: use braces to make the associated `if` unambiguous

```
if (n > 0) {
  if (a > b)
    z = a;
  else // a <= b
    z = b;
}
```

```
if (n > 0) {
  if (a > b)
    z = a;
}
else // n <= 0
  z = b;
```

labels and goto

a label is a name attached to a statement

- precede the statement with an identifier then a colon ‘:’, e.g., ‘myLabel:’

the goto statement unconditionally changes the flow of control

- use a label to specify where control should go, e.g., ‘goto myLabel;’

```
  int i = 0;
repeat:                       // destination label for ‘goto’
  printf("%d", i);
  i += 1;
  if (i < 10) goto repeat;    // go back to the printf statement above
  printf("\n");
```

replacing goto with a do loop

in the previous example, the ‘loop test’ comes at the *end* of the loop

this pattern of ‘test-and-goto’ can be written as a do loop

```
  int i = 0;
repeat:
  printf("%d", i);
  i += 1;
  if (i < 10) goto repeat;
  printf("\n");
```

$\Longleftrightarrow$

```
int i = 0;
do {
    printf("%d", i);
    i += 1;
} while (i < 10);
printf("\n");
```

output: 123456789

most uses of labels and goto can (and should) be avoided

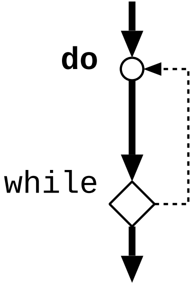
- they can be replaced by loops which are easier to understand and modify

(the various loops were invented to replace the most common patterns of if and goto)

loops: `do`

the `do` loop repeats a statement while a condition is true

- the repeated part of the loop is called the *body*
- the body will always run at least once
- the *condition* for repeating the loop is at the end of the body



`do`
    statement
`while` (*expression*) ;

1. execute *statement*
2. evaluate *expression*
3. if the value of *expression* is true
   - then transfer control back to step 1

replacing `goto` with a `while` loop

the test can also be placed at the *start* of a loop

this pattern of 'test-and-`goto`' can be written as a `while` loop

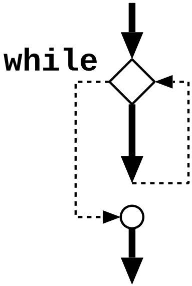
```
int getint(void)
{
  int n = 0;
  int c = getchar();
repeat:
  if (!isdigit(c)) goto done;
  n = n * 10 + c - '0';
  c = getchar();
  goto repeat;
done:
  return n;
}
```

```
int getint(void)
{
  int n = 0;
  int c = getchar();

  while (isdigit(c)) {
    n = n * 10 + c - '0';
    c = getchar();
  }

  return n;
}
```

loops: `while`

the `while` loop is similar to `do` except…

- the test is performed at the start of the body
- the body might be not be run at all



`while` (*expression*)
    statement

1. *expression* is evaluated
2. if the value is non-zero
   - *statement* is executed
   - control goes back to step 1
3. when *expression* becomes zero
   - the loop stops and control resumes after *statement*

processing an array of data almost always looks the same

algorithms that process a sequence of data often use the same common pattern

e.g., a function to sum the integers in an array

```
int sum(int items[], int numItems)
{
    int result = 0;
    int i = 0;                    // set-up a variable controlling the loop
    while (i < numItems) {        // loop condition
        result += items[i];       // loop body
        ++i;                      // update the variable controlling the loop
    }
    return result;
}
```

replacing `while` with a `for` loop

this pattern of ‘set-up, test, loop body, update, repeat’ can be written as a `for` loop

```
int sum(int items[], int numItems)
{
    int result = 0;
    int i = 0;
    while (i < numItems) {
        result += items[i];
        ++i;
    }
    return result;
}
```

```
int sum(int items[], int numItems)
{
    int result = 0;
    for (int i = 0;  i < numItems;  ++i)

        result += items[i];


    return result;
}
```

the variable `i` controls the execution of the loop

- initial value, condition, updated value for next iteration of the loop

the advantage of `for` is that it gathers all these operations on `i` into a single line

loops: `for`

```
for ( expression1 ; expression2 ; expression3 )
    statement
```

is equivalent to

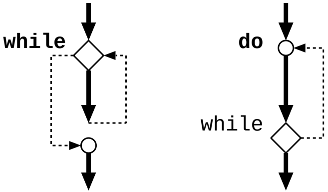
```
expression1 ;
while ( expression2 ) {
    statement
    expression3 ;
}
```

all three *expression*s are optional

- if *expression₁* (setup) is missing, it means ‘do nothing’
- if *expression₂* (test) is missing, it is ‘always true’
- if *expression₃* (update) is missing, it means ‘do nothing’

a popular way to write an *infinite loop* is therefore: `for (;;)` *statement*

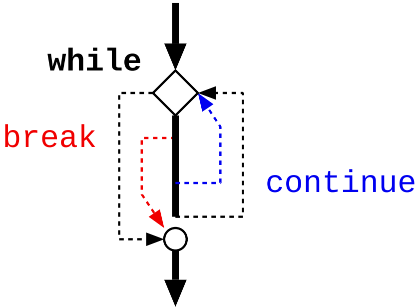
`break` and `continue`

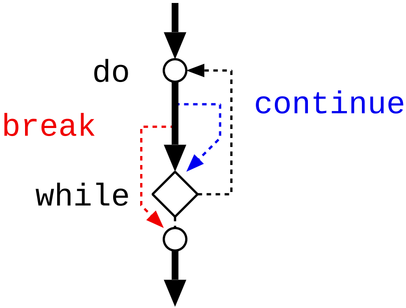


`break` makes a loop terminate immediately

`continue` makes a loop immediately jump back to its ‘test’ step

- in a `while` loop, `continue` jumps *back* to the test at the *start* of the loop
- in a `do` loop, `continue` jumps *forward* to the test at the *end* of the loop





`break` and `continue` facilitate complex loop conditions

`break` and `continue` offer a lot of flexibility to simplify complex loops

```
int getlineNoSpaces(char line[], int limit)
{
  int i = 0;                     // next index to write in line

  while (i < limit - 1) {        // test 1: line is not full
    int c = getchar();
    if (c == EOF)  break;        // test 2: no more input
    if (c == ' ')  continue;     // test 3: ignore spaces
    line[i++] = c;               // body:   store character in array
    if (c == '\n') break;        // test 4: stop at end of line
  }
  line[i] = 0;                   // terminate string

  return i;                      // answer length of string
}
```

(this is *exactly* how I would write this function in my own programs)

break and continue in for loop

```
int i = 0;
while (i < 10) {              // <-+
  printf("%d\n", i);          //   +
  if (i == 5) continue;       // --+ return to test, skipping update
  printf("%d\n", i + 100);
  ++i;                        // update (not run after 'continue')
}
        //          +-----------+
        //          |           |
int i;  //          v           |
for (i = 0;  i < 10;  ++i) { //    |
  printf("%d\n", i);          //    |
  if (i == 5) continue;       // --+ return to update, and then test
  printf("%d\n", i + 100);
}
```

comma operator in `for`

use comma to

- initialise more than variable before the loop
- update more than one variable each time the loop repeats

```
/* reverse a string in place */

void reverse(char s[])
{
    int c, i, j;

    for (i = 0, j = strlen(s) - 1;  i < j;  ++i, --j) {
        c = s[i];
        s[i] = s[j];
        s[j] = c;
    }
}
```

multi-way conditionals using `else if`

common way to write multi-way decision:

```
if ( expression₁ )
        statement₁
else if ( expression₂ )
        statement₂
else if ( expression₃ )
        statement₃
else if ( expression₄ )
        statement₄
else                // optional
        statement   // executed if no expression was true
```

if any $expression_i$ is true

- its associated $statement_i$ is executed
- control passes to the statement following the entire chain of conditionals

sequences of `if`s to test multiple values

often an action depends on which value a variable has out of many possibilities

```
int isspace(int c)
{
  if (c == ' ' ) return 1; // space
  if (c == '\t') return 1; // tab
  if (c == '\n') return 1; // newline
  if (c == '\r') return 1; // carriage return
  return 0;
}
```

note that an `else` between the `if`s is redundant

- the first `if` that is satisfied will `return`, terminating the function

replacing multiple `if`s with `switch`

a sequence of `if`s comparing with constant integers can be replaced with a `switch`

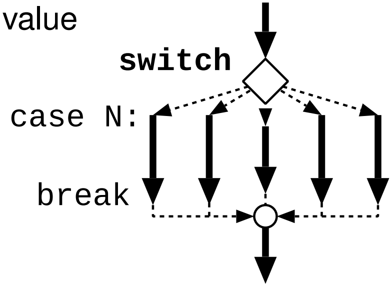
```
int isspace(int c) {

  if (c == ' ' ) return 1;
  if (c == '\t') return 1;
  if (c == '\n') return 1;
  if (c == '\r') return 1;

  return 0;
}
```

```
int isspace(int c) {
  switch(c) {
    case ' ' :  return 1;
    case '\t':  return 1;
    case '\n':  return 1;
    case '\r':  return 1;
  }
  return 0;
}
```

this *only* works when testing *one* `int` variable against *constants*

multi-way jump: `switch`

multi-way `goto` to one of many labels based on an integer's value




```
switch ( expression ) {
  case const-expr1:   statements1
  case const-expr2:   statements2
  default:   default-statements // optional
}
```

each *const-expr* must be a different integer value

the *expression* is evaluated to produce an integer value v

if v matches one of the $\mathit{const\text{-}expr}_i$ `case` labels

- a `goto` is performed to the corresponding `case` label (i.e., to $\mathit{statement}_i$)

if v does not match any $\mathit{const\text{-}expr}_i$ **and** a `default:` label is present

- a `goto` is performed to the `default` label (i.e., to *default-statements*)

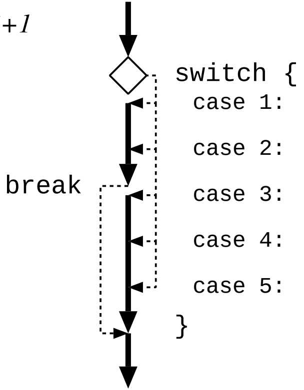
`break` inside a `switch`

`break` causes an immediate exit from the body of the `switch` statement

each `case` is just a destination label for the `switch` to jump to

- control 'falls through' linearly from $statements_i$ to $statements_{i+1}$
- `break` causes a jump to the end of the last $statements_i$

```
for (int i = 0;  i <= 6;  ++i) {
    switch (i) {                        // output printed...
      case 1: printf("1");              // 12
      case 2: printf("2");              // 2
      case 3: printf("3");  break;      // 345
      case 4: printf("4");              // 45
      case 5: printf("5");              // 5
    }
    printf("\n");
}
```



`case` ranges

checking for large sets of value is tedious → use a range: `case` *first* . . . *last* :

```
switch (i) {
  case 'A': case 'B': case 'C': case 'D': case 'E':
  case 'F': case 'G': case 'H': case 'I': case 'J':
  case 'K': case 'L': case 'M': case 'N': case 'O':
  case 'P': case 'Q': case 'R': case 'S': case 'T':
  case 'U': case 'V': case 'W': case 'X': case 'Y':
  case 'Z':
    letter();
    break;
  case '0': case '1': case '2': case '3': case '4':
  case '5': case '6': case '7': case '8': case '9':
    digit();
    break;
}
```

```
switch (i) {
  case 'A' ... 'Z':
    letter();
    break;
  case '0' ... '9':
    digit();
    break;
}
```

counting characters using `switch`

```
int main() {
    int c, digits[10], blanks = 0, others = 0;

    for (int i= 0;  i < 10;  ++i)          // set all digit
        digits[i] = 0;                     // counts to 0

    while (EOF != (c = getchar())) {       // c = each character in file
        switch (c) {
          case '0'...'9':                  // character is digit
              digits[c - '0']++;           // count it
              break;                       // continue with next character
          case ' ': case '\t': case '\n':  // character is blank
              ++blanks;                    // count it
              break;                       // continue with next character
          default:                         // all other characters
              ++others;                    // count it
              break;                       // continue with next character
        }
    }

    for (int i= 0;  i < 10;  ++i)          // print count of each digit
        printf("'%d': %d\n", i, digits[i]);

    printf("blanks: %d\n", blanks);        // print counts of blanks and others
    printf("others: %d\n", others);

    return 0;
}
```

goto is almost never needed but is *essential* when it is needed

code using loops and conditionals is almost always easier to understand than `goto`s

however, a few situations can be made simpler using `goto`

1. breaking out of deeply nested loops (`break` only terminates one level of loop)
2. restarting a loop without performing the test again

```
  for ( ... )
      for ( ... ) {
          if (problem)
              goto error;
      }
  return result;

error:
  recover from error
```

```
while (EOF != (c = getchar())) {
  testChar:
    switch (c) {
        case '0'...'9': {
            int i = c - '0';
            while (isdigit(c = getchar()))
                i = i * 10 + c - '0';
            printf("%d\n", i);
            // continue the loop but avoid calling
            // getchar() again in the test
            goto testChar;
        }
    }
}
```

next week: functions and program structure

function to search file for string
  flowchart, pseudocode, code
factoring the program
function `getline()`
function `strindex(s, t)`
compilation from multiple files
returning non-integers
`atof()`
using header files to ensure function type safety
simple calculator using `atof()`
desk calculator: stack structure

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

exercises

please download assignment from

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

exercises

counting many kinds of character using `switch`

generalising a function by adding parameters

- convert a digit value into a character
- print a number in decimal
- convert a number to a string in decimal
- convert a digit value into a character value
- convert a number to a string in a specified base
- convert a number to a string right-justified, padded with spaces
- convert a number to a string padded with a specified character

replacing many specific functions with a single general function

- rewrite the conversion functions using the one general function

bonus: rewrite `while` and `for` loops using `goto`

exercise notes and preparation

transforming a string while copying it

1. convert single input character into multiple output characters

```
int convert(char to[], char from[])
{
  int t= 0, f= 0, c;
  while ((c = to[t] = from[f])) { // not at end of string
    switch (c) {
      case 'x':  to[t++] = 'E';  to[t++] = 'K';  to[t] = 'S';  break;
      case 'y':  to[t++] = 'W';  to[t++] = 'H';  to[t] = 'Y';  break;
    }
    ++t;
    ++f;
  }
  return t; // to[t] is guaranteed to be the nul terminator
}
```

exercise notes and preparation

2. convert multiple input characters into a single output character

```
int convert(char to[], char from[])
{
  int t= 0, f= 0, c;
  while ((c = to[t] = from[f])) { // not at end of string
    if (c == 'W' && from[f+1] == 'H' && from[f+2] == 'Y') {
      to[t] = 'y';
      f += 2;
    }
    else if (c == 'E' && from[f+1] == 'K' && from[f+2] == 'S') {
      to[t] = 'x';
      f += 2;
    }
    ++t, ++f;
  }
  return t; // to[t] is guaranteed to be the nul terminator
}
```

exercise notes and preparation

3. detecting a range using prefix synyax: "ab*48yz" → "ab45678yz"

```
int convert(char to[], char from[])
{
  int t= 0, f= 0, c, first, last;
  while ((c = to[t] = from[f])) { // not at end of string
    if (c == '*' && (first = from[f+1]) && (last = from[f+2])) {
      while (first <= last)
        to[t++] = first++;  // insert chars from first to last
      f += 3;               // skip *XY
      continue;             // t and f are already correct for next iteration
    }
    ++t, ++f;
  }
  return t; // to[t] is guaranteed to be the nul terminator
}
```

13:00	5	last week: declarations, operators
13:05	5	control flow
13:10	5	statements and blocks
13:15	5	conditional: if and else
13:20	5	ambiguous else
13:25	5	else if
13:30	5	switch
13:35	5	break in switch
13:40	5	case ranges
13:45	10	example: counting characters
13:55	10	while and for
14:05	5	do
14:10	5	break and continue
14:15	5	labels and goto
14:20	15	exercise preparation
14:35	10	*break*
14:45	90	exercises
16:15	00	end