Week 2

XMUT-NWEN 241 - 2024 T2

Systems Programming

Mohammad Nekooei

School of Engineering and Computer Science

Victoria University of Wellington

Admin

- Exercise 1 is out
  - Due to 22 September 7:00 pm (China Time)

Content

- C Fundamentals

- Basic I/O

C Fundamentals

Identifiers

- Identifier is used to name **macros**, variables, **functions**, **structs**, **unions**, and other entities in a computer program

- Java and C have similar rules for identifiers, except:
  - In C, $ is not allowed in identifiers (though some compilers allow $)

Rules on Identifiers

- An identifier is a sequence of letters and digits
  - The first character must be a letter
    - The underscore character _ counts as a letter
    - Upper and lower case letters are different

- Identifiers may have any length
  - Usually, only the first 31 characters are significant
  - For macro names, only the first 63 characters are significant

- Reserved keywords cannot be used as identifiers!

Examples

`counter`	Valid: consists of letters
`_Temp_variable_2`	Valid: consists of letters and digits
`1myVariable`	Invalid: first character is not a letter
`$steps`	Invalid: $ is not allowed in C
`continue`	Invalid: reserved word

Reserved Keywords

```
auto
break
case
char
const
continue
default
do
double
else
enum
extern
float
for
goto
if
int
long
register
return
short
signed
sizeof
static
struct
switch
typedef
union
unsigned
void
volatile
while
```

Data Types

- Recall: Java has 8 basic data types which have fixed sizes

Data Type	Size (bytes)
boolean	1
byte	1
char	2
short	2
int	4
long	8
float	4
double	8

Data Types

- C data types:

Data Type	Size (bytes)
~~boolean~~	~~1~~
~~byte~~	~~1~~
char	~~2~~ 1
short (short int)	~~2~~ Machine-dependent
int	~~4~~ Machine-dependent
long (long int)	~~8~~ Machine-dependent
long long (long long int)	Machine-dependent
float	~~4~~ Machine-dependent
double	~~8~~ Machine-dependent
long double	16 Machine-dependent

Integral types

Float types

Data Type Size

- Sizes of different types
  - Use `sizeof()` to find out
  - Some of the types size may vary from machine to machine

- The following rules are always guaranteed:
  - `sizeof(char) == 1`
  - `sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long) <= sizeof(long long)`
  - `sizeof(float) <= sizeof(double) <= sizeof(long double)`

Data Types

- Integral types can either be `signed` or `unsigned`

```
signed int var1;    // Signed integer
unsigned int var2;  // Unsigned integer
int var1;  // If signed or unsigned is not present, default is signed
```

char Data Type

- `unsigned char`: 0 to 255; `signed char`: -128 to 127
- `char` is meant to hold 1 ASCII character
    - see https://www.asciitable.com/

```
|  0 NUL|  1 SOH|  2 STX|  3 ETX|  4 EOT|  5 ENQ|  6 ACK|  7 BEL|
|  8 BS |  9 HT | 10 NL | 11 VT | 12 NP | 13 CR | 14 SO | 15 SI |
| 16 DLE| 17 DC1| 18 DC2| 19 DC3| 20 DC4| 21 NAK| 22 SYN| 23 ETB|
| 24 CAN| 25 EM | 26 SUB| 27 ESC| 28 FS | 29 GS | 30 RS | 31 US |
| 32 SP | 33  ! | 34  " | 35  # | 36  $ | 37  % | 38  & | 39  ' |
| 40  ( | 41  ) | 42  * | 43  + | 44  , | 45  - | 46  . | 47  / |
| 48  0 | 49  1 | 50  2 | 51  3 | 52  4 | 53  5 | 54  6 | 55  7 |
| 56  8 | 57  9 | 58  : | 59  ; | 60  < | 61  = | 62  > | 63  ? |
| 64  @ | 65  A | 66  B | 67  C | 68  D | 69  E | 70  F | 71  G |
| 72  H | 73  I | 74  J | 75  K | 76  L | 77  M | 78  N | 79  O |
| 80  P | 81  Q | 82  R | 83  S | 84  T | 85  U | 86  V | 87  W |
| 88  X | 89  Y | 90  Z | 91  [ | 92  \ | 93  ] | 94  ^ | 95  _ |
| 96  ` | 97  a | 98  b | 99  c |100  d |101  e |102  f |103  g |
|104  h |105  i |106  j |107  k |108  l |109  m |110  n |111  o |
|112  p |113  q |114  r |115  s |116  t |117  u |118  v |119  w |
|120  x |121  y |122  z |123  { |124  | |125  } |126  ~ |127 DEL|
```

Example

01000001

What do you see?

- Interpreted as an integer: 65
- Interpreted as an ASCII character: ‘A’

Variable Declaration

- Similar syntax as Java
- A variable must be declared before it can be used
- A variable may be initialized in its declaration
  - If variable name is followed by an equals sign and an expression, the latter serves as an *initializer*

```
int i = 0, j = 1, k = 2;
char c = 'A';
float f = 1.25;
```

- Possible initializers
  - Constant
  - Expression

Constants and Literals

- Constants are **fixed values** that cannot be changed during a program’s execution
- The fixed values are called **literals**
- Literals
  - Integer
  - Floating Point
  - Character
  - *String*
  - *Enumeration*

Integer Literals

- Used for representing integer-valued constants
  - Can be written in decimal (no prefix), octal (prefix `0`), or hexadecimal (prefix `0x`)
  - Can have suffix that is a combination of `U` (`unsigned`) and `L` (`long`) in any order
    - No suffix means the literal is of type **`int`**

`12345`	Valid
`12345u`	Valid: unsigned
`0xbeef`	Valid: hexadecimal
`081`	Invalid: 8 is not a valid octal digit
`0x123uu`	Invalid: same suffix is repeated

Floating Point Literals

- Used for representing real-valued constants
  - Can be written in decimal form or exponential form
  - Can have suffix `f` (`float`) or `L` (`long double`)
    - No suffix means the literal is of type **`double`**

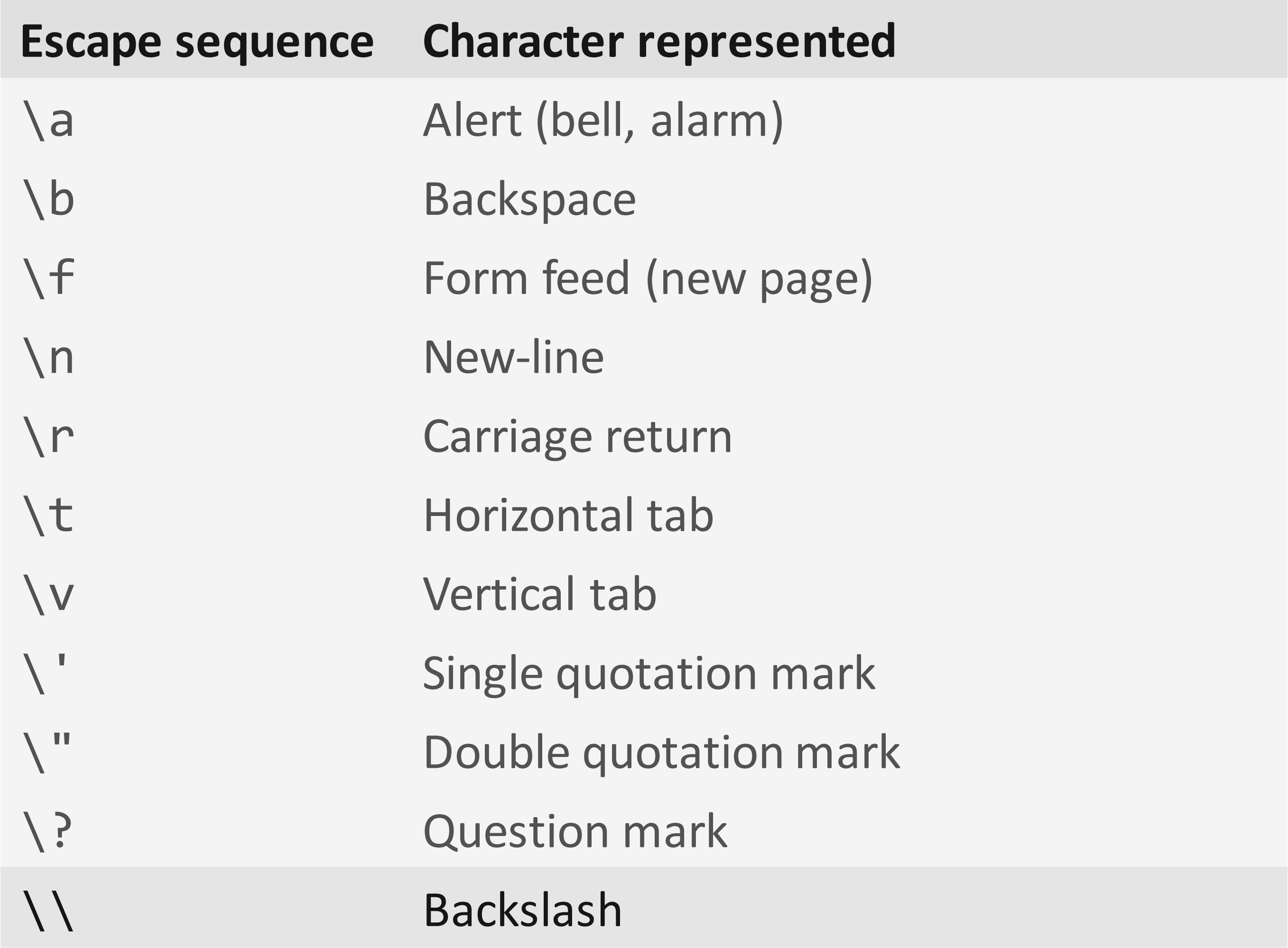
`3.1415`	Valid (decimal form)
`31415e-4`	Valid (exponential form)
`31415e-4L`	Valid: long double
`6.22e`	Invalid: incomplete exponent
`.e23`	Invalid: missing decimal/fraction part

Character Literals

- Used for representing character constants
  - Enclosed in single quotes (')
  - Can be plain (single character) or escape (single character preceded by \)

`'A'`	Valid (plain character)
`'\t'`	Valid (escape character): tab
`'Aa'`	Invalid: multiple characters in single quotes
`'\z'`	Invalid: not a valid escape character

Escape sequences



Escape sequence	Character represented
`\a`	Alert (bell, alarm)
`\b`	Backspace
`\f`	Form feed (new page)
`\n`	New-line
`\r`	Carriage return
`\t`	Horizontal tab
`\v`	Vertical tab
`\'`	Single quotation mark
`\"`	Double quotation mark
`\?`	Question mark
`\\`	Backslash

Declaring Constants

- Constants can be declared using `const` qualifier or `#define` pre-processor

- Such named constants are also called **symbolic constants**

```
const float PI = 3.14;
const int MAX = 12345;
```

```
#define PI 3.14
#define MAX 12345
```

Type Casting

- Type casting is a way to convert a variable from one data type to another data type
- C performs automatic type casting (implicit type conversion)

```
int i = 2;
double d = 2.5;
i = (int)d;      // explicit type casting

i = d;          // d is converted to an int
                // and then assigned to i
```

Operators

- Java and C share many of the built-in operators
  - Arithmetic
  - Assignment
  - Increment/decrement
  - Relational
  - Equality and logical
  - Bitwise
- C specific operators
  - Pointers and reference related operators (*, &, ->)
  - Others (sizeof, scope, casting)

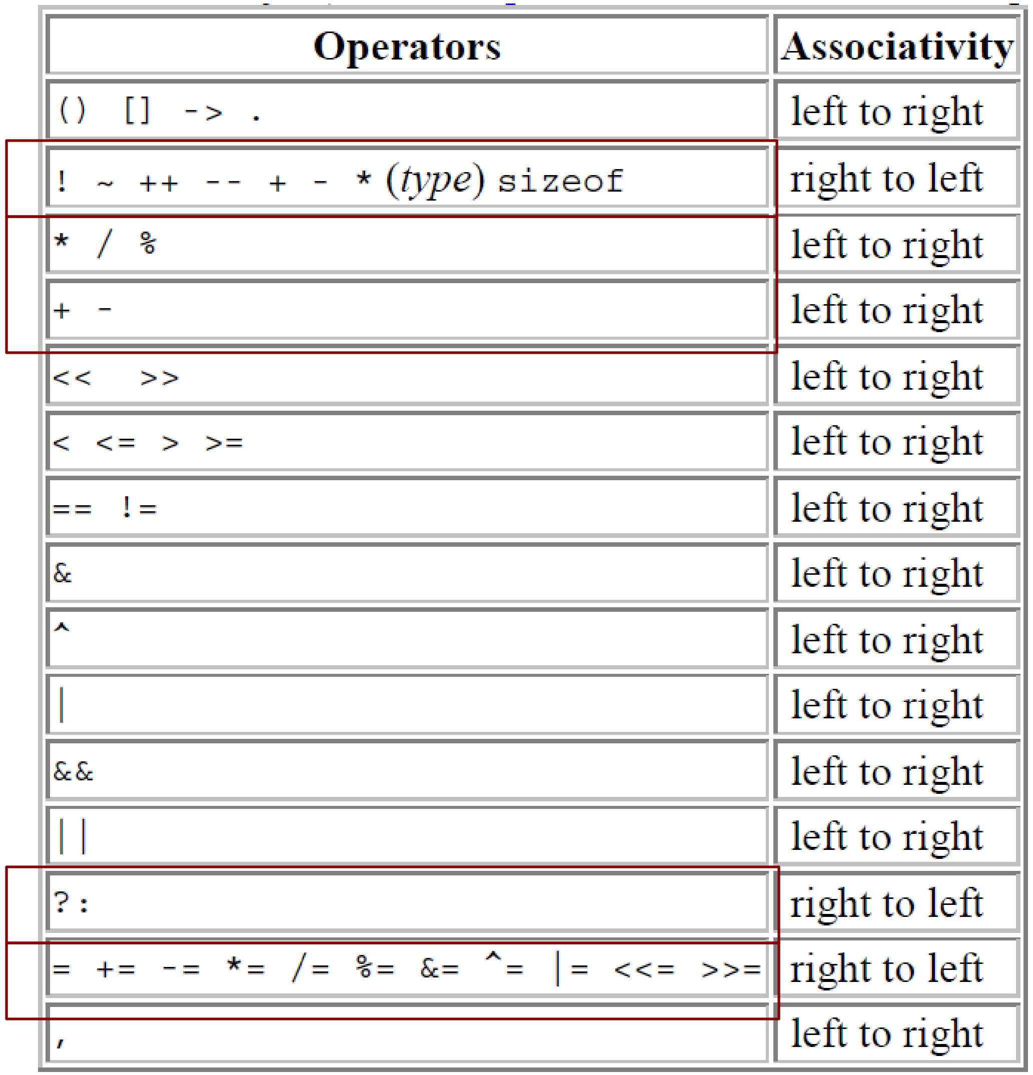
Operator Precedence

- Operator *precedence* determines the sequence in which operators in an expression are evaluated

- *Associativity* determines execution for operators of equal precedence

- Precedence can be overridden by explicit grouping using parenthesis: (and)

Operator Precedence Table (not complete)



Operators	Associativity
`() [] -> .`	left to right
`! ~ ++ -- + - *` (*type*) `sizeof`	right to left
`* / %`	left to right
`+ -`	left to right
`<< >>`	left to right
`< <= > >=`	left to right
`== !=`	left to right
`&`	left to right
`^`	left to right
`|`	left to right
`&&`	left to right
`||`	left to right
`?:`	right to left
`= += -= *= /= %= &= ^= |= <<= >>=`	right to left
`,`	left to right

Unary operators

Arithmetic operators

Ternary operator

Assignment operators

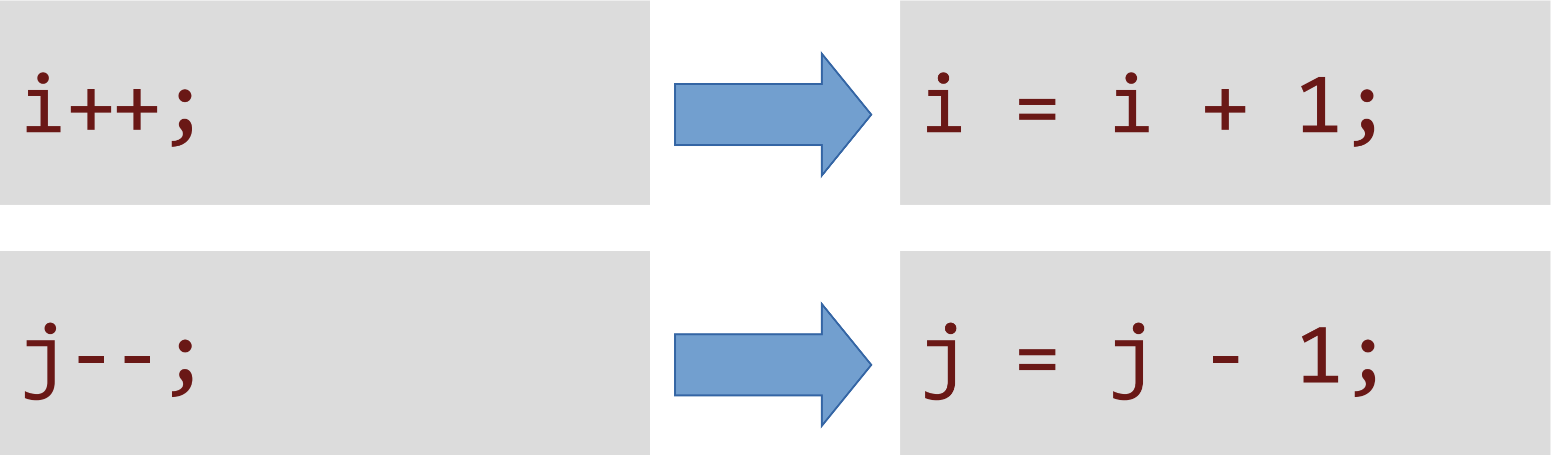
Important Things to Remember

- / denotes integer division if both operands are of integral types
  - 5/2 evaluates to 2 (integer part is used, decimal part is truncated)
- % denotes modulo operation
  - 5%2 evaluates to 1 (the remainder after dividing 5 with 2)

Increment/decrement operators

```
++    --
```

- Increase (++) or decrease (--) variable by 1
- Can be applied to variables, but not constants and ordinary expressions



```
i++;    i = i + 1;
j--;    j = j - 1;
```

- ++ and – are called *unary* operators because they operate on 1 operand

Increment/decrement operators

```
k++;
counter--;
```

Valid if k and counter are variables of basic types

```
777++;
(a + b*c)--;
```

Invalid

++ and -- can be used *postfix* or *prefix*:

```
a = b++;
```

Postfix: use the current value of b in the assignment, then increment b after the assignment

```
a = ++b;
```

Prefix: increment b first, then assign it to a

Increment/decrement operators

```
1.int a, b, c = 0;
2.
3.a = ++c;
4.b = c++;
```

What are the values of a, b and c immediately after line 4?

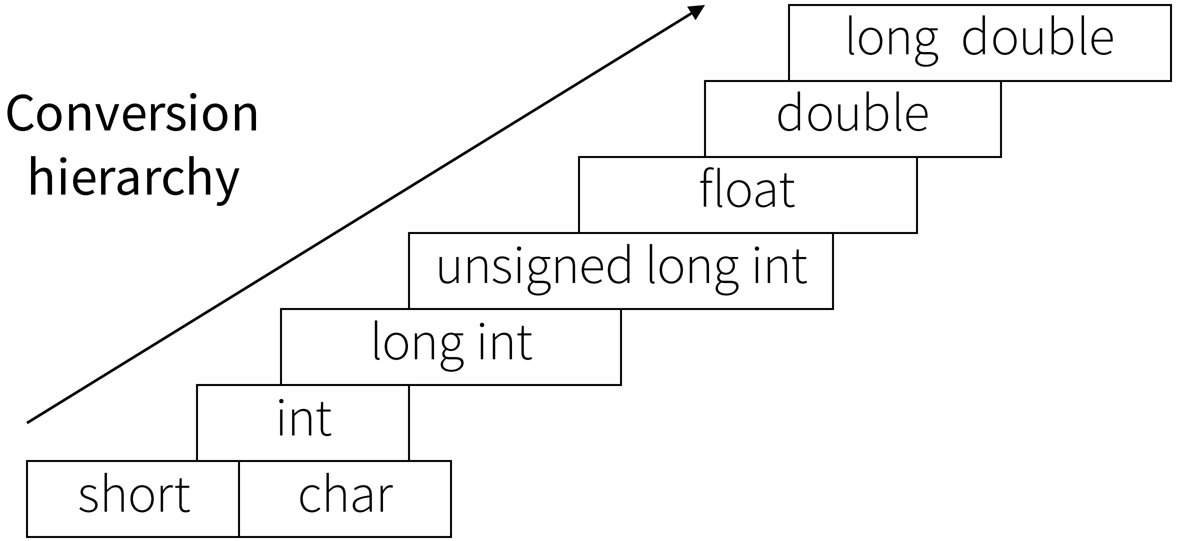
True and false

- Unlike newer programming languages, C doesn’t have native types for Boolean (logical *true* and *false*)
  - Zero (0) is used to denote false
  - Conceptually, one (1) is used to denote true
    - Any non-zero (positive and negative) value is also treated as true

- Relational, equality and logical operations evaluate to either true (1) or false (0)

“Conversion hierarchy”

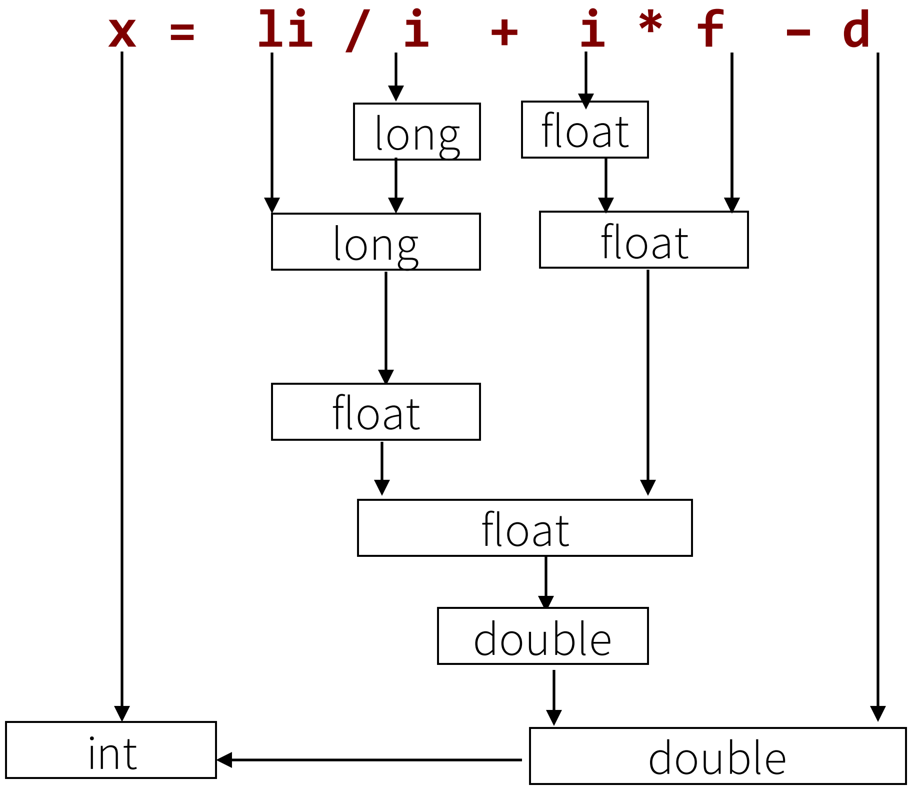
- What happens when operands have different types in an arithmetic expression?
  - **Implicit type conversion is performed:** compiler automatically converts any intermediate values to the proper type so that the expression can be evaluated without losing any significance



Implicit Type Conversion Example

Suppose:

```
int i, x;
float f;
double d;
long int li;
```



The final result of the right hand side expression is converted to the type of the variable on the left of the assignment

Basic I/O

Content

- Basic I/O
- Control flow
  - If-else
  - Else-if
  - Switch
- Iterations
  - While-loop
  - For-loop
  - Do-while-loop
- Same syntax as Java

I/O Using Standard C Library

- Recall: C provides a set of header files (standard C library) that you can use to write your code

C provides a standard library which consists of the following headers:

assert.h
ctype.h
errno.h
float.h
limits.h
locale.h
math.h
setjmp.h
signal.h
stdarg.h
stddef.h
stdio.h
stdlib.h
string.h
time.h

- You don’t have to start from scratch!

I/O Streams

- C provides functions with input and output capability
- From the program's point of view, data input and data output are made possible through files
- Every C program has access to 3 such files: `stdin, stdout, stderr`

File	Description	Remarks
`stdin`	Standard input file	Connected to the keyboard
`stdout`	Standard output file	Connected to the screen
`stderr`	Standard error file	Connected to the screen

I/O Functions

- C input/output functions can be classified into 2 types:
  - Non-formatted input/output
    - `getchar`
    - `putchar`
    - `gets`
    - `puts`
  - Formatted input/output
    - `printf` and its variants
    - `scanf` and its variants

How To Use a Function

- Find its manual or documentation
  - In Linux terminal, use the man command
  - You can also search online
    - This website provides a pretty good documentation for the standard C library: https://www.tutorialspoint.com/c_standard_library/index.htm

- What to look for in the function manual?
  - What the function does
  - What header file(s) to include
  - What are the arguments to the function
  - What is the return type
  - What happens in case of errors

printf()

- printf() writes a string to the standard output stream (stdout)
  - The string is formatted using additional arguments that follow the initial string.
  - %d format specifier to display the value of an integer variable.
  - %c to display character,
  - %f to display float variable,
  - %s to display string variable
  - To generate a newline, we use “\n” in C printf() statement.

```
char ch = ‘A’;
printf("Character is %c \n", ch);
```

Format specifiers in C

Format Specifier	Type
%c	Character
%d	Signed integer
%u	Unsigned int
%e or %E	Scientific notation of floats
%f	Float values
%hi	Signed integer (short)
%ld	Long
%lf	Double
%Lf	Long double
%lli or %lld	Long long
%o	Octal representation
%p	Pointer
%s	String
%x or %X	Hexadecimal representation
%%	Prints % character

`scanf()`

- `scanf()` accepts input from the standard input stream (`stdin`).
  - The format of the expected items is specified, and it returns the number of items successfully scanned
  - The format specifier `%d` is used in `scanf()` statement. So, the value entered is received as an integer and `%s` for string.
  - Ampersand is used before the variable name in `scanf()` statement.

```
char ch;
scanf("%c", &ch);
```

Control flow

Control flow: if-else statement

```
 if (expression){
         statement
 }
```

- If expression evaluates to true, statement is executed

*Recall: true → non-zero;
false → zero

```
if (x != 0.0)
      y /= x;

if (c == ' ') {
      ++blank_counter;
      printf("Found another blank\n");
}
```

```
if (a > b)
      max = a;
else
      max = b;
```

Conditional expression (ternary operator)

$$expr_1 \; ? \; expr_2 \; : \; expr_3$$

- $expr_1$ is evaluated first
- If $expr_1$ evaluates to true, then expression $expr_2$ is evaluated and that is used as the value of the expression
- Otherwise, $expr_3$ is evaluated and that is used as the value of the expression

- Example:

```
z = (a > b) ? a : b;  /* z = max(a, b) */
```

Boolean expressions

What can go in the condition of an **if** statement?

- Boolean expressions:
  - numeric comparisons: (x > 0) (day <= 7),
    (x == y), (day != 7)
  - logical operators: !, &&, || (not, and, or)

    (x > 0 && x < 7)

Writing Boolean expressions

Mostly, boolean expressions are straightforward,

There are just a few traps:

- == is the "equals" operator for simple values,
  = is assignment

  (age == 15) *vs* ~~(age = 15);~~

  - But only use == for numbers (or characters, or references)

Using else-if statement

- Can put another **if** statement in the **else** part:

```
if ( 〈condition1〉 ) {
   〈actions to perform if condition1 is true〉
      :
}
else if ( 〈condition2〉 ) {
   〈actions to perform if condition 2 is true (but not condition 1) 〉
      :
}
else if ( 〈condition3〉 ) {
   〈actions to perform if condition 3 is true (but not conditions 1, 2)〉
      :
}
else {
   〈actions to perform if other conditions are false〉
      :
}
```

Traps with Boolean expressions

- When combining with && and ||, which binds tighter?

  if (x > 5 && y <= z || day == 0 **) { ….**

  - Use (and) whenever you are not sure!

    if ((x > 5 && y <= z **)** || day == 0 **) {** …

    if (x > 5 && **(** y <= z || day == 0 **)) {** …

- The not operator ! goes in front of expressions:

  - **if (** !(x > 5 && y <= z **)** { … **NOT** **if (** (x !> 5 && y !<= z **)**

Example: else-if statement

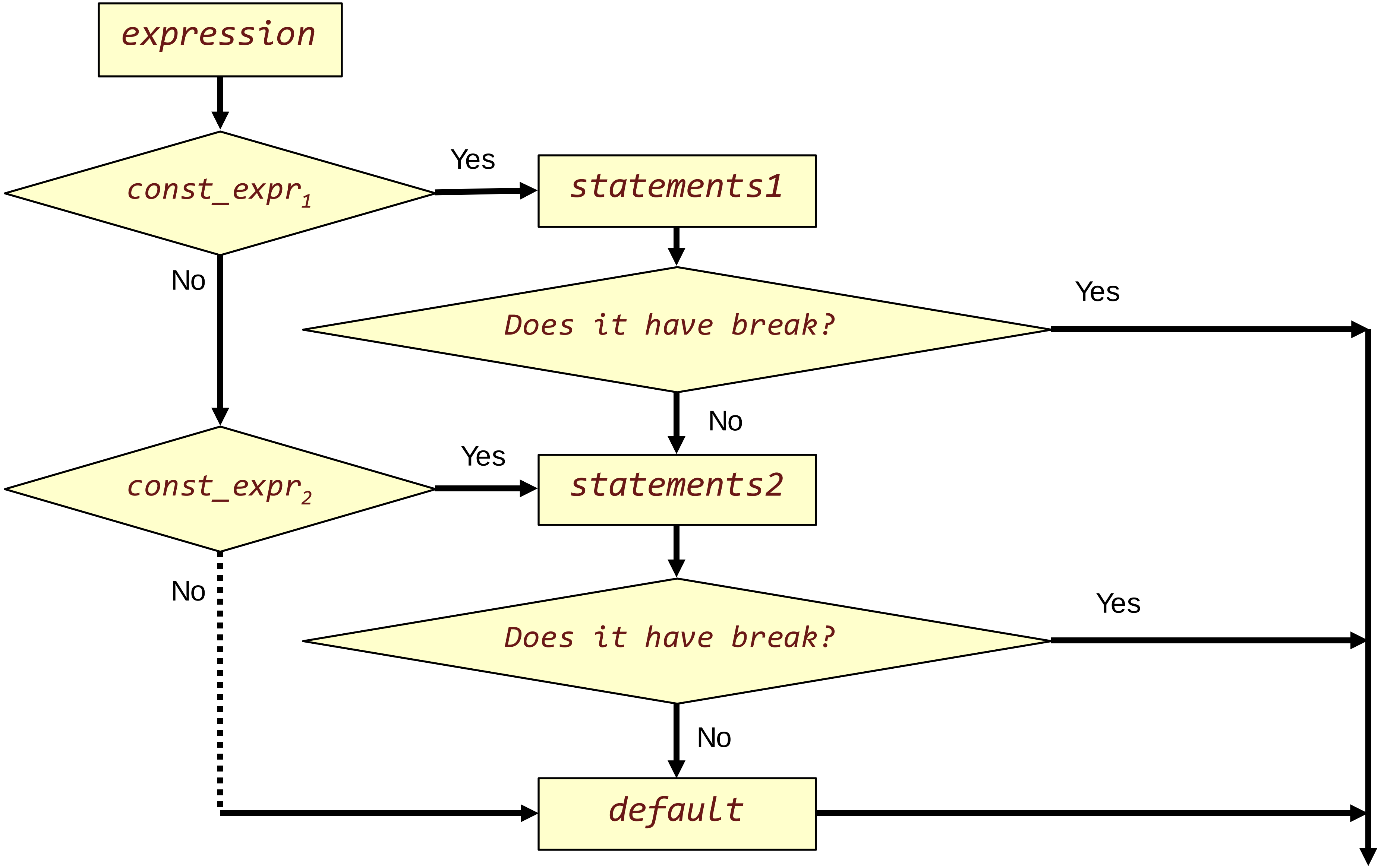
```
if (temp <= 0)
      printf("It's freezing out there.\n");
else if (temp <= 10) {
      too_cold++;
      printf("It's too cold for me.\n");
} else if (temp <= 20)
      printf("It's still cold.\n");
else
      printf("Awesome!\n");
```

Control flow: switch statement

```
switch (expression) {
      case const_expr1:
            statements1
            break;
      case const_expr2:
            statements2
      ...
      case const_exprN:
            statementsN
      default:
            statements
}
```

- The `default` part is optional
- **const_expr**$_1$ to **const_expr**$_N$ must be integer constants or constant expressions
- If `expression` matches **const_expr**$_k$, execution starts at that case
- **`default`** is executed if none of the cases match
- The statements can consist of single or multiple statements statements, or compound statements

Switch statement



Example: switch statement

```
char c = getchar();

switch(c) {
      case 'Y’:
      case 'y':
            printf("You answered yes.\n");
            break;
      case 'N’:
      case 'n':
            printf("You answered no.\n");
            break;
      default:
            printf("What was that?\n");
            break;
}
```

Is this necessary?

Iterations

Iteration: while-loop statement

```
while (expression)
        statement
```

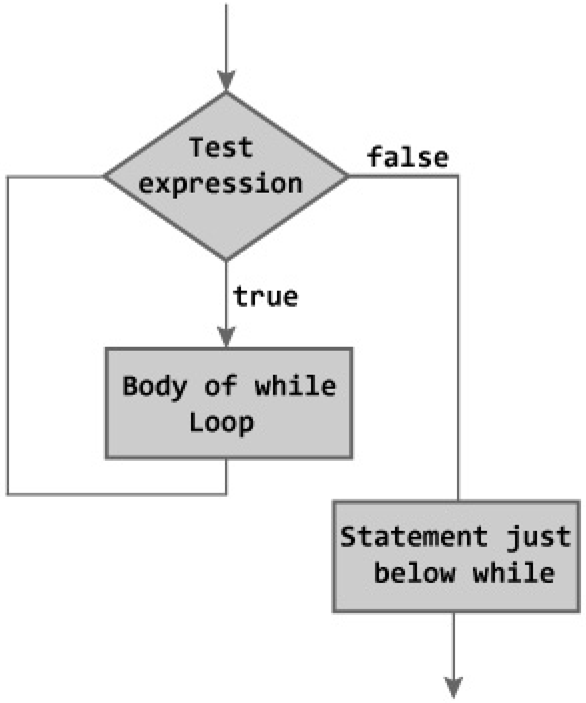
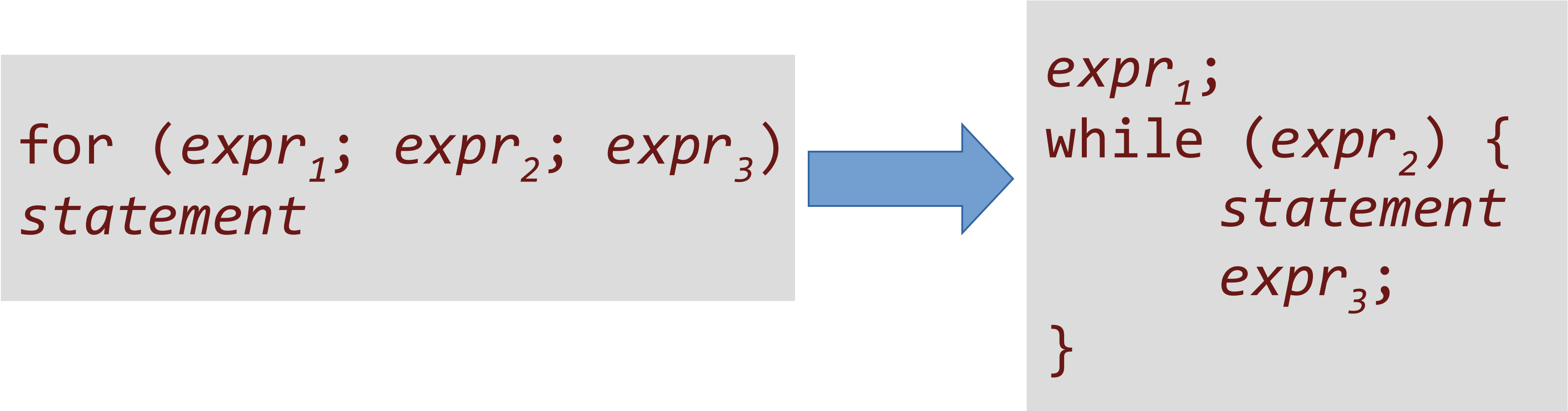


Figure: Flowchart of while Loop

- If `expression` evaluates to true:
  - `statement` is executed
  - `expression` is re-evaluated again
- Cycle continues until `expression` evaluates to false
- `statement` can be single or compound statement

Iteration: for-loop statement



for (*$expr_1$*; *$expr_2$*; *$expr_3$*)
statement

$expr_1$;
while (*$expr_2$*) {
        statement
        $expr_3$;
}

- The expressions are optional
- $expr_1$ and $expr_3$ are usually assignments or function calls
- $expr_2$ is usually a relational expression
  - If **$expr_2$** is missing, it is taken as permanently true

Iteration: do-while-loop statement

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

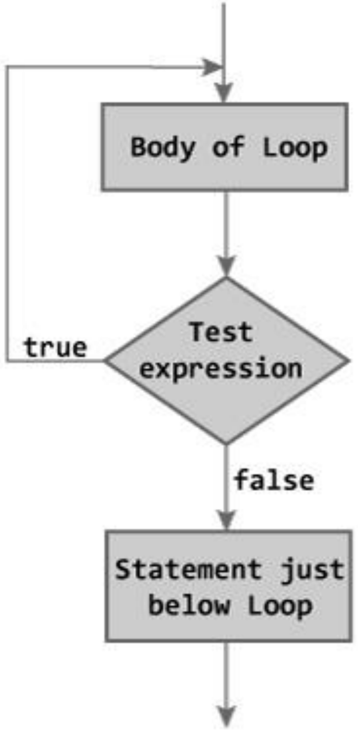


Figure: Flowchart of do...while Loop

- `statement` is executed, then `expression` is evaluated
- If `expression` evaluates to true, `statement` is executed again
- Loop terminates when `expression` evaluates to false

Example: loop statements

Infinite loops:

```
while(1);
```

```
for (;;);
```

```
do {} while(1);
```

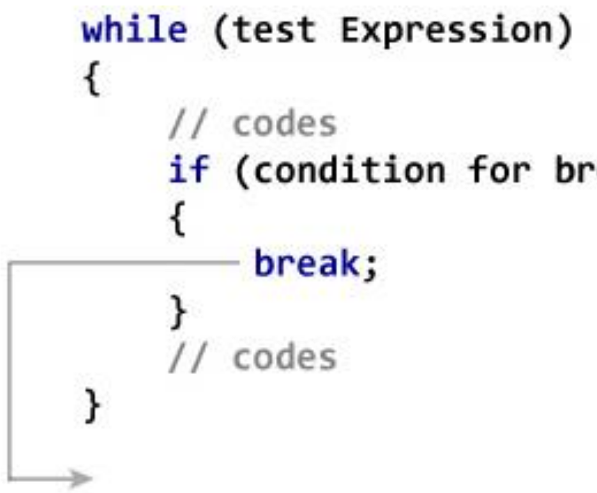
```
int i = 10;
while(i > 0) {
   printf("%d\n", i);
   i--;
}
```

```
for(int i = 10; i > 0; i--) {
   printf("%d\n", i);
}
```

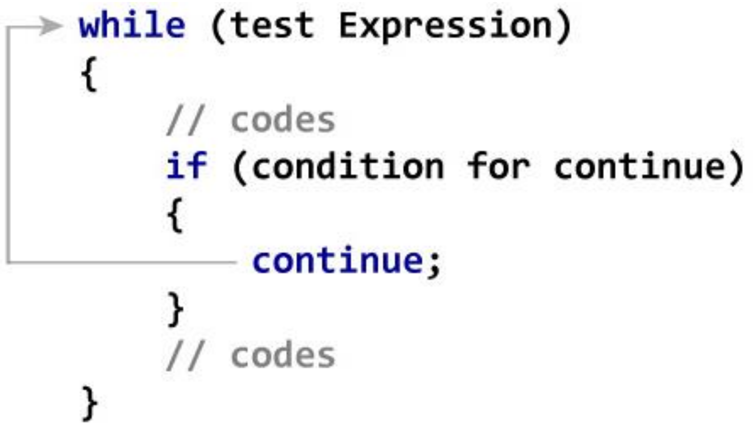
```
do {
   printf("Do you agree with the contract?\n");
   ans = getchar();
} while (ans != 'Y' || ans != 'y');
```

Statements that can alter control flow & loop

- `break`, `return` and `continue`
  - `break`: jumps out of the loop or switch
  - `return`: jumps out of the loop or switch (the loop or switch must be inside a function)
  - `continue`: stops current loop iteration and starts next iteration



```
while (test Expression)
{
    // codes
    if (condition for break)
    {
        break;
    }
    // codes
}
```



```
while (test Expression)
{
    // codes
    if (condition for continue)
    {
        continue;
    }
    // codes
}
```

Differences

Condition in if-else, else-if, while-loop, for-loop and do-while-loop

- In Java, the condition must be an expression that evaluates to boolean
- In C, the condition is an expression that evaluates to any type
  - Considered true if expression evaluates to non-zero value, otherwise false

Break and continue

- In Java, break and continue statements can be labelled or unlabelled
- In C, break and continue statements do not support labels

Example

```
int i = 100;

while (i--) {
   // do stuff
}
```

- Valid in C
- Will generate syntax error in Java
  - Condition inside while-loop should be changed to an expression that will evaluate to boolean type, e.g. `i-- > 0`

Next Lecture

- Function