Week 2 Lecture 1

NWEN 241
Systems Programming

Jyoti Sahni

Jyoti.sahni@ecs.vuw.ac.nz

Admin Stuff

- Assignment #1 is out. Visit https://ecs.wgtn.ac.nz/Courses/NWEN241_2024T1/Assignments to download handout and sample test files

- Exercise 1 (2.5% of course marks) is due on Wednesday, 06 Mar, 23:59

- Week 1 Practice Quiz is available in course wiki (Lecture Schedule) for self-assessment of Week 1 topics.

Content

- Literals (continued from previous lecture)
- Operators and expressions
- Functions
- Function-like macros
- Introduction to Arrays

Recap: 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*

Recap: 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
```

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

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)

Unary operators

Arithmetic operators

Ternary operator

Assignment operators

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

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 can only be applied to variables of basic types

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

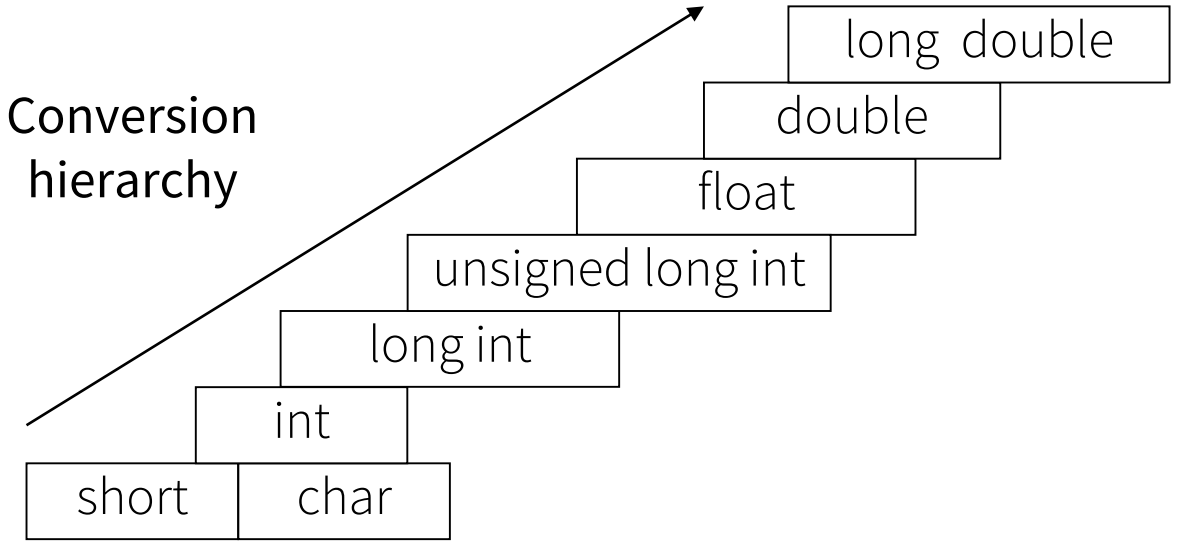
Valid if `k` and `counter` are variables of basic types

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

Invalid

“Conversion hierarchy”

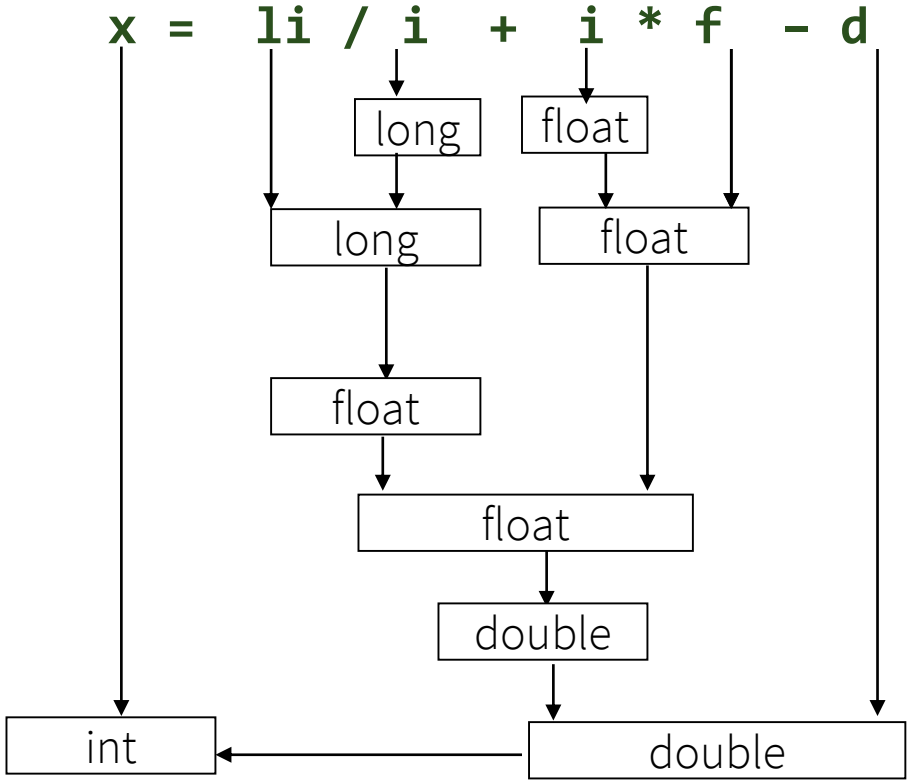
- What happens when operands have different types in an arithmetic expression?
  - **Implicit type conversion is performed:** compiler automatically converts any intermediate values to the proper type so that the expression can be evaluated without losing any significance



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

Control Constructs

- Control flow
  - If-else
  - Else-if
  - Switch
- Loop
  - While-loop
  - For-loop
  - Do-while-loop
- Same syntax as Java

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

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`

Differences

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

```
first:
for (int i = 0; i < 4; i++) {
second:
    for (int j = 0; j < 4; j++) {
        if (i == 1 && j == 1)
            break first;
    }
}
```

- Valid in Java but not in C

Example

```
first:
for (int i = 0; i < 4; i++) {
second:
    for (int j = 0; j < 4; j++) {
        if (i == 1 && j <= 1)
            continue first;
    }
}
```

- Valid in Java but not in C

Functions

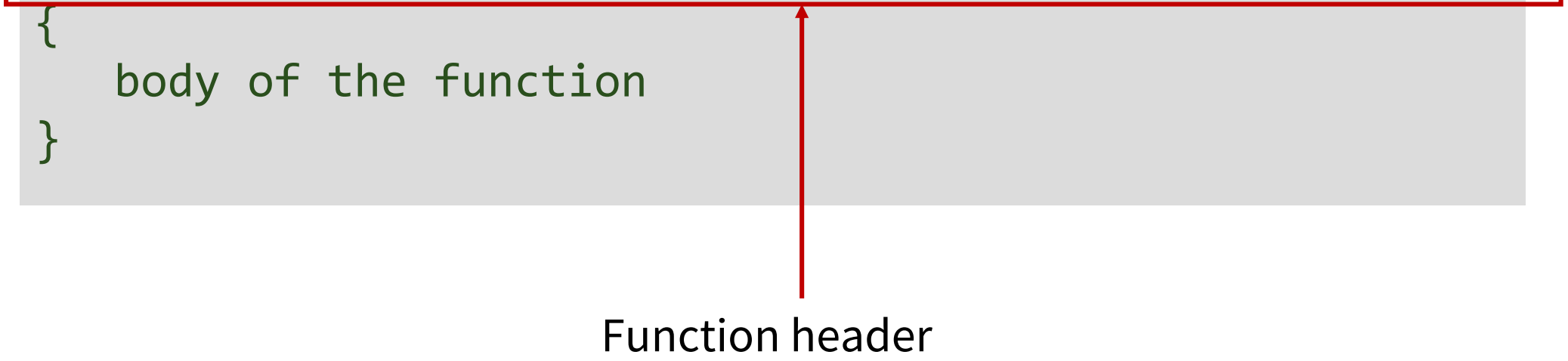
- Unlike Java, C allows functions to exist on their own, i.e., outside any class
  - In C, functions are first-class entities: a C program consists of one or more functions

- A C program must have exactly one `main` function
- Execution begins with the `main` function

Functions

- General form of a C **function definition:**

```
return_type function_name ( parameter_list )
{
   body of the function
}
```

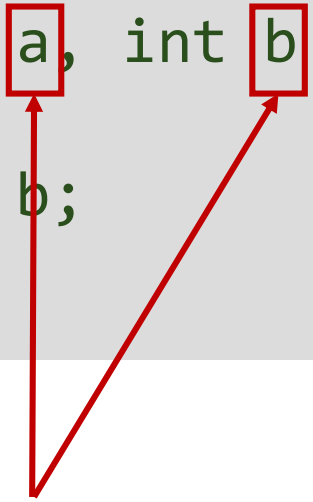


Functions

- Examples

```
void say_hello ( void )
{
   printf("Hello");
}
```

```
int add ( int a, int b )
{
   return a + b;
}
```



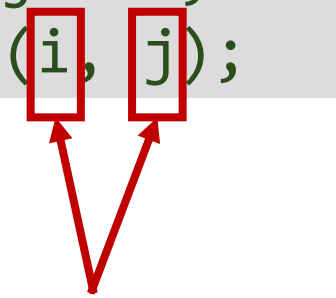
Formal parameters

Invoking Functions

- Example function invocations:

```
say_hello();
```

```
int i = 1, j = 2;
int k = add(i, j);
```



Actual parameters

- Before a function can be invoked, either the **function definition** or **function prototype** should have been declared prior to the invocation

Function Prototype

- A declaration specifying the return type, function name, and list of parameter types

```
return_type function_name ( parameter_types_list );
```

Function Prototype

- Examples

```
void say_hello ( void );
```

```
int add ( int a, int b );
```

- No need to provide identifiers to input parameters, the types of the input parameters are sufficient

```
int add ( int, int );
```

Macro Substitution

- Recall: Can define symbolic constants using #define pre-processor

```
#define PI 3.14
```

`PI` is a macro, every occurrence of `PI` in the program will be replaced by `3.14`

- In general:

```
#define name replacement
```

- Subsequence occurrences of `name` will be replaced by `replacement`

Function-like Macro

- Can *abuse* macro substitution to define **function-like** macros
- To define a function-like macro, just append `()` to the macro name
- Example:

```
#define READ_CHAR()     getchar()
```

- Can be invoked like a regular function:

```
…
int c = READ_CHAR();
…
```

Function-like Macro

- Just like functions, function-like macros can take arguments
  - Insert comma-separated parameter names between (and)
  - Parameter names must be valid identifiers

```
#define MAX(X, Y) ((X) > (Y) ? (X) : (Y))
```

- Invoke just like normal functions

```
z = MAX(1, 3);
```

```
z = ((1)>(3)?(1):(3));
```

This expression evaluates to **3**

Next Lecture

- Function-like macros
- Arrays