EECS 2031

Software Tools

Module 5 – Introduction to C

YORK UNIVERSITÉ UNIVERSITY

C vs. Java

- Java-like (actually Java has a C-like syntax), some differences
- No garbage collection
- No classes
- No exceptions (try … catch)
- No String type
- Pointers ☺

First C Program (`first.c`)

```
#include <stdio.h>

main() {

   printf("hello, world \n");

}
```

Note: `#include <filename.h>`
replaces the line by the actual file before compilation starts.

Basic I/O

- Every program has a *standard input* and a *standard output.*
- By default, keyboard and monitor, respectively

Input functions	Output functions
`scanf()`	`printf()`
`getchar()`	`putchar()`
`fgets()`	`fputs()`

Output is easy… (`celsius.c`)

- Most of the time, use **`printf()`**
  - Very similar to Java
- See Section 1.2 in the textbook
- Returns the number of characters printed
- Can also use **`putchar()`** for a single character

Input is more complicated

- Several functions for input should **never** be used because they are unsafe
- They are still in the standard library because a lot of code out there uses them
- Avoid using **`gets()`** as well as **`scanf()`** for strings
- Recommended way to read input: **`getchar()`** or **`fgets()`** + **`sscanf()`**

`getchar()`

- To read one character at a time from the *standard input* (the keyboard by default):

```
int getchar(void)
```

- Returns the next input character each time it is called
- Returns EOF when it encounters end of file.
  - EOF input: Ctrl-D (Unix) or Ctrl-Z (Windows).

`getchar()`

- It buffers input characters until a new line or EOF is entered.
  - That is, nothing happens until you hit Return or EOF.
- See **`getchar1.c`** and **`getchar2.c`**
- Take a look at the man page

```
man –S 3 getchar
```

`scanf()`

- `scanf()` can be used for formatted input
- To read an integer:

```
int num;

scanf("%d", &num);
```

- `&num` is a pointer to `num`

scanf()

- To read a char and a float:

```
char c; float f;

scanf("%c %f", &c, &f);
```

- `scanf()` stops when it exhausts its format string, or when some input fails to match

`scanf()`

- Returns the number of successfully matched and assigned input items
- Returns 0 if the input does not match the specification in the format string (i.e., an error).
- On the end of file, EOF is returned.

Line-based I/O

- We'll use **`fgets`** to read a line of input

```
fgets(s, n, stdin);
```

- Reads at most one less than the number of characters specified by **`n`** from the given stream and stores them in the string **`s`**
- If '**`\n`**' is read, it is also stored in **`s`**
- Note: we are not guaranteed a full line!

More examples

- **`getaline.c`**: Use **`getchar()`** to implement a function that reads a line of input
- **`fgetsscanf.c`**: Use **`fgets()`** and **`sscanf()`** to read a line of input

- **`sscanf()`** is similar to **`scanf()`** except it parses a string provided as an argument rather than standard input

C variable names

- Combinations of letters, numbers, and underscore character (_) that
  - do not start with a number
  - are not a keyword

- Upper and lower case letters are distinct (x ≠ X)

C data types

All data types in C are **numeric**

- **char** – commonly used for characters (8 bits)
- **int** – integers (either 16 or 32 bits)
- **long** – integers (64 bits)
- **float** – single precision floating point numbers (4 bytes)
- **double** – double precision floating point numbers (8 bytes)

Qualifiers

- **`unsigned int`**
  - All values are positive
- **`long double`**
  - Even more precision than a double
- **`long long`** is the type that represents the largest integers possible in C
- **`short`** can also be used but is rare

Qualifiers and data sizes

- To get the size of a type, use **`sizeof()`**

```
int_size = sizeof( int );
```

- See **`numeric.c`**

Constants

- Numeric constants
- Character constants
- String constants
- Constant expressions
- Enumeration constants

Integer Constants

- Decimal numbers

  `123487`

- Octal: start with 0 (zero)

  `0654`

- Hexadecimal: starts with 0x or 0X

  `0x4Ab2, 0X1234`

Integer Constants

- long int: suffixed by L or l

  `7L, 106l`

- unsigned int: suffixed by U or u

  `8U,  127u`

  What number is this? `0XFUL`

Floating-point Constants

- Contain a period or scientific notation (or both)

```
15.75    25E-4    -2.5e-3    .001
```

- If there is no suffix, the type is considered double
- Use suffix `F` or `f` for float, `L` or `l` for long

```
100.0F          100.0L
```

Character constants

- The type char is a numeric type of size 8 bits
- Values are typically given in character form between 2 single quotes as in Java

```
char x ='A';
```

- Can also use octal notation for special characters

```
c ='\012'
```

10 in decimal, a new line character

String Constants

- There is no string type in C
- Strings are just arrays of characters
- However, C allows for string constants same as in Java
- `"hello, " " world"` is the same as `"hello, world"`
- Useful for splitting up long strings across several source lines.

Constant Expressions

- Expressions that involve only constants
- Evaluated during compilation

```
#define MAXLINE 1000

char line[MAXLINE+1];
```

Enumeration Constants

```
enum boolean { NO, YES };
```

- The first name in an enum has value 0, the next 1, and so on, unless explicit values are specified.

```
enum colours { black, white,
  red, blue, green };
```

- What is the value of `blue`?

Enumeration Constants

- If not all values are specified, unspecified values continue the progression from the last specified value.

```
enum months { JAN = 1, FEB, MAR,
  APR, MAY, JUN, JUL, AUG, SEP,
  OCT, NOV, DEC };
```

`FEB = 2`, `MAR = 3` etc.

Declarations

- All variables must be declared before use
- A variable may also be initialized in its declaration.

```
char esc = '\\';

int i = 0;

int limit = MAXLINE+1;

float eps = 1.0e-5;
```

Qualifier `const`

- Indicates that the value of a variable will not be changed.
- For an array: the elements will not be altered.

```
const double e = 2.71828182845;

const char msg[] = "Warning: ";
```

Type Conversion

- `float f; int i;`
- What is the type of `f+i` ?
- General rule: convert a “narrower” operand into a “wider” one without losing information.
- So `i` is converted to float before the addition.

More Examples

- What is the type and value of the following expressions?

```
17 / 5
```

```
17.0 / 5
```

```
9 / 2 / 3.0 / 4
```

Type Conversion: More Rules

- Conversions take place **across assignments**; the value of the right side is converted to the type of the left, which is the type of the result.
- Example:

```
int i;

float f = 7, g = 2;

i = f / g;
```

Type Conversion

- Longer integers are converted to shorter ones by dropping the excess high-order bits.

```
int i; char c;

…

c = i;

i = c;
```

The value of `i` may change after these two lines

Casting

- Casting works the same way as in Java

```
int a = 9, b = 2;
double  x;
x = a / b;                  // x is 4.0
x = a / (double) b; // x is 4.5
```

Operators

- Arithmetic operators:

  `+  -  *  /  %`

- Relational operators:

  `>  >=  <  <=  ==  !=`

- Logical operators:

  `!  &&  ||`

- All the same as in Java

Conditions

- 0 is False
- Anything else is True
- If you `#include <stdbool.h>` you can have something like boolean variables

```
bool valid;

valid = false;
```

- Still a numeric value though

Conditions

- Write

```
if (!valid)
```

instead of

```
if (valid == 0)
```

- The following is not a syntax error in C

```
if (i = 0)
```

- The condition will always be false no matter what the value of `i` is

Comparing strings

- Conditions involving strings should use functions from **`string.h`**
- **`s1 == s2`** is only true if both **`s1`** and **`s2`** refer to the same memory position
- **`strcmp(s1,s2)`** returns
  - 0 if the two strings are equal
  - Negative if s1 is lexicographically first
  - Positive if s2 is lexicographically first

Comparing strings

- To check if two strings are equal

```
if (strcmp(s1,s2) == 0)
```

- Other useful string functions
  - **`strlen(s1)`** returns the length of string **`s1`**
  - **`strcat(s1,s2)`** appends string **`s2`** at the end of string **`s1`**
- Strings in C are mutable!

Bitwise Operators

- They work on individual bits
  `&` : Bitwise AND
  `|` : Bitwise OR
  `^` : Bitwise exclusive OR
  `~` : Bitwise complement
- Useful when each bit has a different meaning
- Handy when memory was at a premium
- See **`bitwise.c`**

Bit Shifting

- **`x<<y`** means shift **`x`** to the left **`y`** times.
  - equivalent to multiplication by 2^y
- **`x>>y`** means shift **`x`** to the right **`y`** bits.
  - equivalent to division by 2^y
- Right shifting may have strange behaviour depending on the type of **`x`**
- See **`shifting.c`**

Statements and Blocks

- Statement: followed by a semicolon.
- Block
  - enclosed between { and }
  - syntactically equivalent to a single statement
  - no semicolon after the right brace
- Variables can be declared inside any block

Control Flow Statements

- All are similar to Java
- `if else`
- `switch`
- `while`
- `for`
- `do while`
- `continue` and `break` for loops

`goto`

- In C, it is possible to add a label to a line of code, and then jump to it from any other part of the code
- Code that relies on **`goto`** statements is generally harder to understand and to maintain. So **`goto`** statements should be used rarely, if at all
- **`break`** and **`continue`** should also be used only when necessary