Faculty of Computer Science, Dalhousie University *5-Oct-2018*

CSCI 2132 — Software Development

Lecture 14: Software Development Life Cycle

Location: Chemistry 125 Instructor: Vlado Keselj
Time: 12:35 – 13:25

Previous Lecture

- *(Fire Alarm)*
- Integer and floating-point representation
- Character types

Slide notes:

Reading Characters

- scanf can be used to read a character, but it does not ignore white space
- For example, the following statements are not equivalent. (Find an input example.)
```
scanf("%c", &ch);
scanf(" %c", &ch);
```
- `getchar` and `putchar` are used for input and output at the character level; e.g.:
```
int ch = getchar();
putchar(ch);
```
- `int` is used as return to detect EOF

In C, characters are declared to be of the type char. Each char variable is stored in 8 bits (This is different from Java, in which the char type requires 16 bits so that is can store a Unicode).

For example, we can declare and initialize a char variable ch as follows (pay attention to the single quotation marks for char constants):

```
char ch = 'a';
```

C treats characters as small integers, which are ASCII values on most machines. For example, the following piece of code can be used to convert a lowercase letter to uppercase:

```
if ('a' <= ch && ch <= 'z')
  ch = ch - 'a' + 'A';
```

The conversion specification used for char in printf and scanf is `%c`. In scanf, it is important to know that `%c` does not skip white-space characters before reading a character. Therefore, the following two statements are NOT equivalent:

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

C programmers frequently use the getchar function to read a character. For example, the following statement will read a character and assign it to a char variable ch:

```
ch = getchar();
```

It is advisable to use getchar in the above case as getchar is fast and simple. It does not require extra time to do pattern matching as required for scanf.

Let's see an example. We now write a program that converts one line of message to all uppercase characters. This is an example for using loop as well. This program is designed as a UNIX filter, so that it will not prompt for user input. It reads one line from the user, and then prints the result.

```
#include <stdio.h>

int main() {
  int ch;
  while (EOF != (ch = getchar()) && ch != '\n') {
    if ('a' <= ch && ch <= 'z')
      ch = ch - 'a' + 'A';
      putchar(ch);
    }
  putchar('\n');
  return 0;
}
```

One interesting part of this program is the following test condition in the while loop:

```
EOF != (ch = getchar()) && ch != '\n'
```

One question is: are the brackets enclosing ch=getchar() necessary? The answer is yes. This is because (in)equality operator has higher precedence than assignment. Even though it is often sufficient to consult a precedence table when we program, this case is very frequent so we are expected to know this in order to avoid bugs.

Type Conversions

Regarding types, C is more flexible than Java. For example, the following line of code is valid in C:

```
float f = 3.4;
```

However, we know that 3.4 without any suffix is a double constant. This line of code would be invalid in Java. What happens when we write a line of code like this? To answer this, we need learn what implicit conversions do in C.

Implicit conversions are type conversions handled by compilers. That is, you need not specify conversion explicitly in your code. There are two cases in which implicit conversions take place.

First, when the operands in an arithmetic or logical expressions don't have the same type. In this case, operands are promoted to the 'narrowest' type that will safely accommodate the values of the operands. Look at the following example:

```
float f;
double d;
int i;
d = d + f;
f = f + i;
```

In the first arithmetic expression, f is promoted to double. In the second, i is promoted to float.

The other case in which implicit conversion takes place is when the type of the expression on the right side of an assignment does not match the type of the variable on the left side. In this case, the right side is converted to the left side. If you write the following line of code:

```
int i = 8.92;
```

Then the value of i after this line is executed is 8.

Sometimes we might want to explicitly tell the compiler to convert a variable from one type to another. For this we need learn how to use type casting, which tells the compiler to treat a variable as of a different type than its declared type. The syntax is:

```
(type) expression
```

We can make use of this to compute the fractional part of a floating-point value:

```
float f, frac_part;
frac_part = f - (int)f;
```

From this example, we can see that type casting can be used to override the compiler and force it to do conversations that we want. Let us see more examples:

```
float quotient;
int dividend = 5;
int divisor = 4;
quotient = dividend / divisor;
quotient = (float) dividend / divisor;
quotient = (float) (dividend / divisor);
quotient = 1.0f * dividend / divisor;
```

What values would these four statements assign to quotient? Answers: 1, 1.25, 1 and 1.25.

Type Definitions Using typedef

We can use the keyword `typedef` to assign alternative names to existing types. The syntax is

```
typedef typename alternative_name
```

One example:

```
typedef int Bool;
Bool flag;
```

In this example, by defining `Bool` to be an alternative name to `int`, we make it clear what kind of values we store in flag. This makes our program more understandable. In the future, we will learn how to define our own abstract data types, and we often use typedef to define alternative names for them.

The sizeof Operator

The sizes of many C types are implementation-defined. For this reason the C standard operator `sizeof` is very useful, since we can use it to determine home many bytes are required to store a value of a particular type, or how many bytes a variable occupies in memory. The following is the syntax of using the `sizeof` operator:

```
sizeof(type)
```

For example, we can use `sizeof(char)` to get the number of bytes required to store a char variable, which is 1. The value of the expression `sizeof(int)` could be 2, 4, 8 or other value, as the size of `int` is implementation-defined.

The `sizeof` operator can also be applied to variables. Typically, the `sizeof` value is known at the compile time, so a compiler should be able to evaluate it, but with C99 standard and variable-length arrays, which we will introduce shortly, it may be necessary that the `sizeof` is calculated at the run time.

14 Software Development Life Cycle

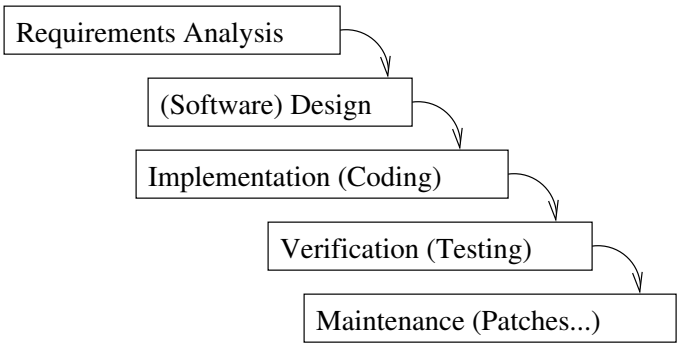
14.1 About Software Development Life Cycle

Software Development Life Cycle (SDLC)

- SDLC is a general term that describes structure imposed on the development of a software product
- Purpose
  - To reduce the risk of missing the deadline
  - To ensure product quality
  - To prevent "scope creep", etc.
- Many models have been proposed to describe SDLC

The Waterfall Model

- A sequential design process



Waterfall: Advantages and Disadvantages

- Advantages
  - Natural and easy to understand
  - Widely used
  - Reinforces notion of "design before coding"

    - Clear milestones
- Disadvantages
    - Often not practical
    - Clients may change requirements
    - Designers may not be aware of implementation difficulties

The Rapid Prototyping Model

1. Gathering preliminary requirements
2. Fast prototyping
3. User evaluation of the prototype
4. Repeat the above steps if necessary
5. Discard the prototype and develop the software using a formal process

Rapid Prototyping: Advantages and Disadvantages

- Advantages
    - Ensure that software product meets client's requirements
    - Reduce time and cost if client requests changes during the process
- Disadvantages
    - Adequate and appropriate user involvement may not always be possible
    - Cost of prototype development

More about Models

- There are many other models
- To be studied in the Software Engineering course
- Choose an appropriate model depending on the particular software to be developed