**Lecture 1:**

* Introduction to the course and overview of the Syllabus.
* Basics of computer software
  + Introduction to the process of software development and the **SDLC** (Software Development Life Cycle)
    - *Requirements analysis* - analysis of a business problem, requirements, possible solutions, costs and benefits.
    - *Design* - choosing a solution, definition of its **architecture** (things difficult to change later), and important elements, business rules, features such as communication with other systems.
    - *Development* - writing the code.
    - *Testing* - validation and verification of the program, checking for errors, checking if the code does what it is supposed to, and does not do what it is supposed not to, checking edge cases.
    - *Deployment* - installing or distributing the program for installation.
    - *Maintenance* - supporting the existing program, providing bug-fixes, training, and assistance.
    - ***Note:****In this course, we focus only on Development and Testing only. The design/specification is provided by the instructor.*
  + A **CPU** (central processing unit, main processor) - an integrated electronic circuit executing the main sequence of instructions in a computer. CPUs are particularly good at executing the following types of operations:
    - Coping values, typically between internal registers *[tiny, fast memory inside a CPU]* or main memory, RAM *[larger, and slower memory external to the CPU]*.
    - Executing mathematical calculations (*arithmetic and logic*), such as add, subtract, multiply, divide, and basic trigonometric operations, just to name a few.
    - Performing comparisons (which are really just mathematical calculations, but due to their special role for conditional jumps we are listing it separately).
    - Performing jumps (in other words just setting the internal Program Counter register to a new value); they can be unconditional ("every time this instruction is executed, jump") or conditional (i.e. "jump if the result of a previous operation, typically a comparison, is zero").
    - Following the fetch-execute cycle ("fetch the next instruction, decode it and load arguments, execute, store the results, move on to the next instruction") [[YouTube video by Tom Scott](https://www.youtube.com/watch?v=Z5JC9Ve1sfI), 10mins]
  + A **computer program** - instructions explaining how to execute steps of a task to a computer, and data needed for these instructions.
    - *Machine code* - a program in a binary form that can be directly executed by a CPU with a specific instruction set architecture [ISA]; such program cannot be executed on a CPU using a different instruction set. The machine code is not readable to a person, and the instructions are just represented by their numeric codes (*opcode*) and values of operands.
    - Source code - a program in a human readable form that cannot be directly executed by a CPU:
      * *Low-level programming language ("close to the machine")*, assembly language (assembler) - a program lists exact instructions as in the machine code in a human readable form, with each instruction represented by a human-friendly name (*mnemonic*, for example *MOV* for move/copy instructions). Such program usually cannot be executed on a CPU using a different instruction set.
      * *High-level programming language ("close to human language")* - a program expressed in a high-level programming language is easier to read for a human, and uses elements of the language closer to the *business domain* than a *specific CPU hardware*. Programs written in high-level programming languages often can be recompiled without any changes for a CPU using a different architecture. The C and C++ programming languages are both considered high-level, although the level of the language is a continuous spectrum, not a binary categorization.

Recommended reading:

* Kim N. King, *"C Programming: A Modern Approach"*, 2nd Edition
  + Chapter 1. Introducing C
* [Fredrick P. Brooks, Jr, *"The Mythical Man-Month. Essays in Software Engineering"*, "Why Do You Do It?"](https://distance.sg.digipen.edu/file.php/3601/References/Fredrick_P_Brooks_Jr_-_The_Mythical_Man-Month_Essays_in_Software_Engineering_-_Why_Do_You_Do_It.pdf)
  + Why is programming fun?
* Please get familiar with [C references](https://distance.sg.digipen.edu/mod/resource/view.php?r=45598) listed in this Moodle page.

**Lecture 2:**

* + Functions
    - A concept of a *function* - a sequence of instructions that
      * executes a certain task,
      * accepts zero (*void*) or more parameters,
      * can produce side effects, such as printing the output to the terminal,
      * returns zero (*void*) or one result value.
    - Functions are a feature of high-level programming languages to group related instructions together
      * They are one of the simplest *abstractions* - a technique where we hide the details [here: instructions] somewhere else [here: in a function's body definition], and later just refer to it by its name [here: in each function call].
    - A program in C must have a function *int main(void)* where the program's execution starts:
      * it executes the logic of the program
      * it accepts no parameters (note: in future we will learn about its variant that does accept special parameters)
      * it can have side effects,
      * it returns a number representing an error code to the operating system where 0 (by convention) indicates a successful execution.
    - Other functions can be declared in a program, but each function must be declared before it is used. They are useful to break long code into smaller, easier to analyze parts.
  + Algorithms
    - A concept of an *algorithm -* a specific set of steps required to execute a task that have the following properties:
      * Includes a start that represent an entry point.
      * Includes one or more ends that represent exit points.
      * Must be specific and clearly defined.
      * Must eventually terminate.
    - Some representations of an algorithm *[think: a pancakes recipe]*
      * Informal description (i.e. plain English description) [[example](https://distance.sg.digipen.edu/file.php/3601/Notes/Lecture_02/Pancakes.txt)].
      * Formal description (i.e. a flow chart diagram) [[example](https://distance.sg.digipen.edu/file.php/3601/Notes/Lecture_02/Pancakes.png)].
  + *Building* executable files in C (often referred to as *compilation*, even though the compilation is only one of the steps in this process).
    - **Preprocessor** - takes in the high-level source code (\*.c) and executes directives starting with *#*, thus expanding the code. The output file is still the valid C source code, but with no preprocessor directives. For example, a line *#include <stdio.h>* gets replaced with the contents of a system header *stdio.h*. Headers contain code stored somewhere else: in a different file, or delivered with a compiler. Some header files may have been written by other developers and shared as the source code; this is the stage where you include them.
    - **Compiler** - takes the C source code with no preprocessor directives, and translates high-level language instructions into low-level language instructions (assembly). Compiler can also do a lot of complex operations such as syntax analysis, conversion of representation, optimization, and so on.
    - **Assembler** - takes the assembly code and translates it the operation mnemonics, addresses, and other syntax elements into numerical codes in a form of machine code, typically compatible only with the current operating system family and the CPU architecture. The output is an object file, which includes incomplete machine code representing the source code.
    - **Linker** - takes one or more object files and combines them, producing an executable file. Some object files may have been written by other developers, but not shared as the source code; this is the stage where you include them.
  + Brief introduction to our environment:
    - **Cygwin** installation ([download](https://www.cygwin.com/) and run installer, choose a mirror server with a good connectivity, make sure to install the "*gcc-g++*" package, complete the installation; restart Windows to ensure the installation path has been included in the [%PATH%](https://en.wikipedia.org/wiki/PATH_(variable)) environment variable).
    - Introduction to **Windows Command Prompt** (cmd.exe). Useful basic commands:
      * *cd* - change directory
        + *cd ..* - go up to a parent directory.
        + *cd \* - go to a drive's root directory.
        + *cd name* - go to a directory named *name.*
      * *mkdir* - create directory
        + *mkdir name* - create a directory named *name*.
      * *del* - delete files and folders (use carefully)
        + *del file* - delete a specific file named *file*.
        + *del \** - delete all files in a folder.
      * *echo* - display some text to the output (a default output is the terminal console)
        + *echo abc* - display text "*abc*".
        + *echo.* - display an empty line (outputs an end-of-line sequence).
        + *echo %errorlevel%* - display a result ("error code") of the last executed program.
      * *cls* - clear the text console window.
      * *exit* - close the text console window.
      * Other notes:
        + A default output is printing text into the text console window (also known as the command prompt, the terminal). A default input is scanning text from the text console window.
        + Output stream can be redirected to a file, for example the following lines writes a new empty line into a file (if the file does not exist, it gets created first):  
          *echo. > file.txt*
    - Cygwin introduces its own Linux/Unix-like command line interface called **Cygwin Terminal**. Once installed, if its path has been included correctly in *%PATH%* environment variable, its commands become also available in Windows Command Prompt, for example:
      * *touch file* - create a file named *file*.
      * *ls* - list current directory (similar to Windows' *cd*).
      * *pwd* - present the path to the current working directory.
      * *clear* - clear the text console window.
    - Once Cygwin has been installed with required packages, typing *gcc* should start the compiler and result in displaying an error "compilation terminated."; this means the compiler works.
  + A minimal valid program [[code](https://distance.sg.digipen.edu/file.php/3601/Notes/Lecture_02/main.c)]:
    - Defines the function *main()* which is the entry point to our program.
    - This function accepts zero parameters from the operating system (*void*).
    - This function returns value of an integer type (to be specific, *int* means signed integer with at least 2 bytes of storage) that is recognized by the operating system as an error code (by convention, 0 means success; see above *echo %errorlevel%*).
    - All the statements within the function's body are surrounded by curly braces (*{ }*).
    - The function's body definition includes a single statement. Individual statements are separated by semicolons (*;*). Our statement *return 0;* does two things: it sets the result of the function to value 0, and terminates the execution of the function, thus returning the control to the code that called the function.
  + A *"Hello World!"* program [[code](https://distance.sg.digipen.edu/file.php/3601/Notes/Lecture_02/hello_world.c)] showing the most basic interaction - output of the text into terminal.
  + A simple demonstration program accepting two numbers, adding them, and displaying their sum. A lot of these elements will be covered in detail in future.
    - Version without comments [[code](https://distance.sg.digipen.edu/file.php/3601/Notes/Lecture_02/sum.c)].
    - Version with comments [[code](https://distance.sg.digipen.edu/file.php/3601/Notes/Lecture_02/sum_commented.c)].
  + Building the code
    - From the source code to the expanded source code (*preprocessor*):  
      *gcc -Wall -Wextra -Wconversion -pedantic -ansi****-E****-o outputFile.i inputFile.c*
    - From the source code to the assembly code (preprocessor + compiler):  
      *gcc -Wall -Wextra -Wconversion -pedantic -ansi****-S****-o outputFile.s inputFile.c*
    - From the source code to the object code (preprocessor + compiler + assember):  
      *gcc -Wall -Wextra -Wconversion -pedantic -ansi****-c****-o outputFile.o inputFile.c*
    - From the source code or the object code to the executable (preprocessor + compiler + assember + linker, or just linker):  
      *gcc -Wall -Wextra -Wconversion -pedantic -ansi -o outputFile.exe inputFile.c other.o*

Recommended reading:

* + Notes by DigiPen's Matthew Mead: [Fundamentals](https://distance.sg.digipen.edu/file.php/3601/Articles/Fundamentals/Fundamentals.html).
* **Lecture 3:**
  + Overview of compilers and interpreters
    - *Compilers* - create an executable program from the source code ahead of execution.
    - *Interpreters* - execute a program by parsing and performing operations from the source code on-the-go in the run-time environment (often referred to as a *"virtual machine"*).
    - *Mixed approach translators* (for instance JIT - "*just-in-time"*compilers) - executing a program by performing right before execution a preliminary compilation (often into an intermediate language) that can be interpreted on-the-go more efficiently.
  + Compilation of the C language
    - Its **source code** contains a directly compilable ***preprocessing units*** (.c) and ***header files*** (.h). Each preprocessing unit can be expanded by a preprocessor to form a separate **translation unit**. Translation units are compiled, assembled and linked into an executable.
    - *Preprocessor* is a simple tool that expands a file by substituting included *directives* with files or fragments defined separately, or by removing portions of the code based on tested conditions.
    - *Compiler* is a complex tool that translates a high-level programming language into a low-level assembly language. This translation often includes the following stages:
      * **Lexical analysis** - processing of a sequence of characters done character-by-character; removing white spaces and comments, and separating groups of characters into tokens.
      * **Syntax analysis** - processing of independent tokens into expressions in a given language grammar that are represented as parse trees. Syntax errors (mismatched brackets or semicolons) are also reported in this step.
      * **Semantic analysis** - finding dependencies between elements of a parse tree and their meaning (detecting unused variables or variables used before their declarations, finding functions with incorrect number of arguments, etc.)
      * **Optimization and code generation** - translating the meaning of the program into a corresponding program in a machine language without changing its behavior. This step is independent from a language, and so multiple compilers in the same compiler collection (like gcc and g++) may reuse this functionality (*compiler's*"*back-end*"; the "*front-end"* processes language-dependent steps).
    - *Assembler* is a program converting low-level assembly into its machine counterpart representation as an object file.
    - *Linker* is a program for merging multiple object files into an executable.
  + Discussion on the syntax of our sample [program](https://distance.sg.digipen.edu/file.php/3601/Notes/Lecture_02/sum.c) from Lecture 2 that adds two numbers.
  + User-defined identifiers (symbols) in the code for names of variables, functions, etc. [[standard c89](http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf), section 6.4.2]:
    - Must not be a reserved [keyword](https://en.cppreference.com/w/c/keyword).
    - Must include 1 or more characters.
    - Must start with a letter or an underscore (*"\_"*).
    - May contain letters, underscores or numbers.
    - Is case-sensitive (character case does matter).
  + Data types in C
    - **Primitive** - introduction to basic arithmetic types [[reference](https://en.cppreference.com/w/c/language/arithmetic_types)].
    - **Complex** - there are other types, but we will discuss them much later.
    - *Note:* the C99 and C11 versions of the C language standards added a few new types; we just mentioned *long long int*.

Recommended reading:

* + [Compilers vs. interpreters](https://distance.sg.digipen.edu/file.php/3601/References/Compilation_Overview.pdf).
  + Notes by DigiPen's Matthew Mead: [Data types](https://distance.sg.digipen.edu/file.php/3601/Articles/DataTypes/DataTypes-1.html)
  + Kim N. King, "C Programming: A Modern Approach", 2nd Edition
    - Chapter 2. C Fundamentals

**Lecture 4:**

* Recap on built-in primitive data types
  + Type names
  + Minimum guaranteed storage sizes
  + Ranges of values
* Literals
  + [character literals](https://en.cppreference.com/w/c/language/character_constant)
    - char (*'A'*) and escape sequences for special characters, for example:
      * '\a' alert (a.k.a. *bell*)
      * '\b' backspace
      * '\r' caret return
      * '\n' new line
      * '\t' tab
      * '\\' backslash
      * '\'' apostrophe
      * '\"' quotation mark
  + [string literals](https://en.cppreference.com/w/c/language/string_literal)
    - [we will discuss the exact type in future] (*"ABC"*), and with escape characters (*"ABC\nD"*)
  + [integer literals](https://en.cppreference.com/w/c/language/integer_constant)
    - int (*1001*)
      * base 10 {decimal} (*1001*); no leading zeros; the digits are from 0..9.
      * base 16 {hexadecimal} (*0x3E9*); character case is insignificant; the digits are from 0..9, A..F.
      * base 8 {octal} (*01751*); with a leading zero; the digits are from 0..7.
    - unsigned int (*1001U*)
    - long int (*1001L*)
    - long long int (*1001LL*)
    - unsigned long long int (*1001ULL*)
  + [floating-point value literals](https://en.cppreference.com/w/c/language/floating_constant)
    - float (*-1.0f*, *1.f*, *.0f*; also in the scientific notation *-1.0e-5f*); single precision.
    - double (*-1.0*, *1.*, *.0*, also in the scientific notation *-1.0e-5*); double precision.
    - long double (*-1.0L*, *1.L*, *.0L*, also in the scientific notation *-1.0e-5L*); extended double precision, rarely used.
* Basic input/output
  + [printf(format, ...)](https://en.cppreference.com/w/c/io/fprintf)
    - Takes in a string of characters (it can include special characters via escape sequences) that represents the *format*, which can include zero or more placeholders; additionally, for each placeholder (except *"%%"*) it accepts a parameter with a value to be used instead of a placeholder.
    - Each placeholder is a conversion specifier: it starts with *'%'*, followed by flags, and terminated with a format specifier letter.
      * Important format specifier letters:
        + % - show a percent sign; does not behave like a placeholder:  
          *printf("%%");  
          /\* outputs: % \*/*
        + *c* - show the char parameter as an ASCII character: *char c = 'A';  
          printf("%c", c);  
          /\* outputs: A \*/*
        + *d* (also *i*) - show a signed decimal number:  
          *int i = 100;  
          printf("%d", i);  
          /\* outputs: 100 \*/*
        + *u* - show an unsigned decimal number:  
          *int i = 100;  
          printf("%d", i);  
          /\* outputs: 100 \*/*
        + *o* - show an octal number (without leading *0*):  
          *int i = 100;  
          printf("%o", i);  
          /\* outputs: 144 \*/*
        + *x* (also *X* for uppercase digits) - show a hexadecimal number (without leading *0x*):  
          *int i = 111;  
          printf("%x", i);  
          /\* outputs: 6f \*/*
        + *f -*show a floating-point value (float or double):  
          *float f = 1.5f;  
          printf("%f", f);  
          /\* outputs: 1.500000 \*/*
        + *s* - show string of ASCII characters: *printf("%s", "ABC");  
          /\* outputs: ABC \*/*
      * Flags preceding a format specifier:
        + *-*(minus)*-*justify the output the the left of the output field.
        + *+*(plus)*-*for non-negative number show explicit *'+'* in front of the number.
        + (space) - for non-negative numbers shown a space in front of the number.
        + *number -*define the minimum width of the output field (larger values can stretch the field)
        + *.number -*define the precision of the number

For integers, fill the value up to this number with leading zeros is not large enough: *int i = 123;  
printf("[%- 8.4d]", i);  
/\* outputs: [ 0123 ] \*/*

For floating-point numbers, show that many decimal places:  
*float f = 1.2f;  
printf("[%- 8.4f]", f);  
/\* outputs: [ 1.20000 ] \*/*

* + [scanf(format, ...)](https://en.cppreference.com/w/c/io/fscanf)
    - Takes in a string of characters (it can include special characters via escape sequences) that represents the *format*, which can include one or more placeholders, other symbols, or space characters (interpreted as one or more *white-spaces*: spaces, tabs, new lines, etc.); additionally, for each placeholder (except the ones including \*, i.e. *"%\*d"*) it accepts a parameter with an address of a variable to be filled in with a captured value.
    - Returns a number of successfully filled in input variables.
    - Each placeholder is a conversion specifier: it starts with *'%'*, followed by flags, and terminated with a format specifier letter.
      * Important format specifier letters:
        + *%* - match a percent sign; does not behave like a placeholder:  
          *scanf("%d%%%d", &i, &j);  
          /\* captures both parameters for: 10%20 \*/*
        + *c* - capture a single character (including a white-space is not preceded by a white-space):  
          *int i;  
          char c;  
          scanf("%d %c", &i, &c);  
          /\* captures both parameters for: 10 X \*/*
        + *i* - capture a signed integer number in any format: decimal, hexadecimal or octal.  
          *int i, j;  
          scanf("%d %i", &i, &j);  
          /\* captures both parameters for: -123 0x123 \*/*
        + *d* - capture a signed decimal integer number.  
          *int i, j;  
          scanf("%d %i", &i, &j);  
          /\* captures both parameters for: -123 0x123 \*/*
        + *u* - capture an unsigned decimal integer number.  
          *unsigned int i;  
          int j;  
          scanf("%u %d", &i, &j);  
          /\* captures both parameters for: 123 -123 \*/*
        + *o* - capture an unsigned octal integer number.  
          *unsigned int u;  
          scanf("%o", &u);  
          /\* captures a parameter with and without the leading 0: 0123 \*/*
        + *x* - capture an unsigned hexadecimal integer number.  
          *unsigned int u;  
          scanf("%x", &u);  
          /\* captures a parameter with and without the leading 0x: 0x123 \*/*
        + *f* - capture a floating-point number.  
          *float f;  
          scanf("%f", &f);  
          /\* captures a parameter in a scientific and non-scientific format: -1.2e-3 \*/*
        + *s* - capture a string of text [we will discuss this in future].
        + *[set]* - capture a string of text with only the characters from the set [we will discuss this in future].
      * Flags preceding a format specifier:
        + \* (asterisk) - suppress capturing of the parameter into a variable, but still require the user to key this part in.
        + *number* - define the maximum number of characters captured for the placeholder.
* Data conversion
  + C is a strongly-typed language and so working with data may require data type conversion.
  + A compiler may generate warnings when the operation could result in a reduction of precision or a change in bit representation.
  + To convert a value without warnings, we can tell a compiler explicitly that such conversion is intended by performing *casting*. To cast a value to a different type you need to precede an expression or a variable with a target type wrapped in parentheses, for example:  
    *int i = 0;  
    float f = (float)(i / 10);  
    /\* performs integral division, then converts the result (int) into float \*/*

Recommended reading:

* [Overview of printf and scanf](https://distance.sg.digipen.edu/file.php/3601/References/Overview_of_printf_and_scanf.pdf)
* Notes by DigiPen's Matthew Mead: [Formatted I/O](https://distance.sg.digipen.edu/file.php/3601/Articles/FormattedIO/FormattedIO-1.html).
* Kim N. King, "C Programming: A Modern Approach", 2nd Edition
  + Chapter 3. Formatted Input/Output

**Lecture 5:**

* **Expression** [[reference](https://en.cppreference.com/w/c/language/expressions)] - a representation of a value
  + Literals (*1*, *2L*, *3.0f*)
  + Variables (*int radius*; *radius*)
  + Constants (*const float PI = 3.14f;* *PI*)
  + Compound expressions - expressions involving **operands**, that is *other expressions*, and **operators**, *symbols representing built-in functions such as addition of two operands*: (*a + b \* c*); the details will be discussed in the next lecture.
* **Statement** [[reference](https://en.cppreference.com/w/c/language/statements)] - a command, a set of instructions, to be executed by a program. A program executes statements top-to-bottom They must terminate with a semicolon (*;*), with few exceptions (i.e. a compound statement). There are different types of statements:
  + **Expression statement** - a statement represented by an expression.
  + **Jump statement** - a statement that cause an unconditional jump to a different instruction in a program. Under-the-hood, jumps are implemented in terms of jumps without additional condition checking necessary.
    - ***break****;*  
      Allowed in *switch* statements and all iteration statements. Unconditionally jumps to the first statement after the current *switch* statement or an iteration statement.
    - ***continue****;*  
      Allowed in all iteration statements. Unconditionally jumps to the beginning of the next iteration, interrupting the current iteration.
    - **goto** *label1*;  
      Unconditionally jumps to a statement prefixed with a code **label** (*label1: statement;*). As this statement makes it difficult to interpret the structure of the program, and allows for difficult to predict behaviours (such as jumping back into a loop without initializing its counter variables), this statement should be avoided.
    - ***return****; /****return****value1;*  
      Interrupts the execution of the current function, and returns the control to the statement following the place that called the current function. If the function returns *void* (nothing) a *return* statement without a value should be used. If a function is expected to return a value of some type, a *return* statement must return a value of this type (or one that the compiler can convert to a desired type).
  + **Selection statement** - a statement that lets the program select a particular path of execution. Under-the-hood, selection statements are implemented in terms of comparisons of a condition expression with 0, and jumps ahead.
    - ***if****(condition)  
      statement;*  
      [[Flow diagram](https://distance.sg.digipen.edu/file.php/3601/Examples/Lecture_06/if.png)] If the *condition* expression evaluates to a non-zero value, the *statement* will be executed; otherwise, the *statement* will be omitted. Multiple *if* statements can be nested.
    - ***if****(condition)  
      statement1;****else*** *statement2;*  
      [[Flow diagram](https://distance.sg.digipen.edu/file.php/3601/Examples/Lecture_06/ifelse.png)] If the *condition* expression evaluates to a non-zero value, the *statement1* will be executed, but not *statement2*; otherwise, the *statement2* will be executed, but not *statement1*. Multiple *if/else* statements can be nested (*if(a) if(b) s1; else s2;*) and chained one after another (*if(a) s1; else if (b) s2; else s3;*); the *else* connects to the most recent *if* statement without *else*.
    - ***switch****(expression)  
      {****case****value1: statement1;****case****value2: statement2;****default****: statement0;  
      }*  
      [[Flow diagram](https://distance.sg.digipen.edu/file.php/3601/Examples/Lecture_06/switch.png)] The *expression* is evaluated (once) into a temporary anonymous *result*. Then the *result* goes through a series of comparisons:
      * If it is equal to *value1*, the *statement1* is executed, and then execution falls through to *statement2* and *statement0*.
      * Otherwise, if it is equal to *value2*, the *statement2* is executed, and then execution falls through to *statement0*.
      * Otherwise, if no other case is met (the default case), the *statement0* is executed alone.
      * Note: The order of cases does matter for the fall-though process. Many compilers protect programmers from the fall-though process, and require a programmer to either end each case with a jump statement, typically break, or explicitly allow fall-though code with an additional compiler flag.
  + **Iteration statement** - a statement that lets the program repeat execution of a statement, facilitating iteration; also known as loops. Under-the-hood, iteration statements are implemented in terms of comparisons of a condition expression with 0, and jumps back.
    - *while (condition)  
      statement;*  
      [[Flow diagram](https://distance.sg.digipen.edu/file.php/3601/Examples/Lecture_06/while.png)] Execute the *statement* as long as the *condition* (re-evaluated each time) is non-zero. As the *condition* is evaluated before the iteration, it is possible that the statement will not be executed at all.
    - *do  
      statement  
      while (condition);*  
      [[Flow diagram](https://distance.sg.digipen.edu/file.php/3601/Examples/Lecture_06/dowhile.png)] Execute the statement as long as the condition (re-evaluated each time) is non-zero. As the condition is evaluated after the iteration, the statement will execute at least once.
    - *for (init, condition, iteration)  
      statement;*  
      [[Flow diagram](https://distance.sg.digipen.edu/file.php/3601/Examples/Lecture_06/for.png)] Executes the *init* expression once, before each iteration checks the *condition* and if it is non-zero the *statement* and *iteration* expression are evaluated. More practical scenario looks like this:  
      */\* for a variable i initialized with 0,  
      as long as it is less then 10,  
      execute the statement  
      and increase i by 1. \*/  
      for (i = 0; i < 10; ++i) statement;*
  + **Compound (block) statement** - a statement that consists of multiple statements, wrapped in curly braces (*{ }*) and used where a single statement is expected.
  + **Empty statement** - a statement with no instructions consisting only of its terminating semicolon and used where a single statement is expected, but there is no statement to provide.

Recommended reading:

* [Edsger Dijkstra: Go To Statement Considered Harmful](https://homepages.cwi.nl/~storm/teaching/reader/Dijkstra68.pdf).
* Kim N. King, "C Programming: A Modern Approach", 2nd Edition
  + Chapter 5. Selection statements
  + Chapter 6. Loops
* Notes by DigiPen's Matthew Mead: [Conditionals](https://distance.sg.digipen.edu/file.php/3601/Articles/Conditionals/Conditionals-1.html).

**Lecture 6:**

* Compound expressions are expressions that require **evaluation**, and given one or more **operands** connected by one or more **operator** symbols, produce the result, and may generate side effects (i.e. print text to the screen).
* Think of operators as built-in functions; for example, imagine that *a + b* means *operator+(a, b)*, while *-x* means *operator-(x)*.
* Classification of expressions:
  + Depending on a number of operands
    - *unary* - 1 single operand preceded or followed by an operator (for example: *-x*, *i++*).
    - *binary* - 2 operands usually with an operator in between (for example: *a + b*, *x \*= y*).
    - *ternary* - 3 operands; there is only one such operator with two symbols in a particular order ?: (for example: *x = cond ? a : b;*)
  + Depending on the precedence - see the [operator precedence table](https://distance.sg.digipen.edu/file.php/3601/References/Operator_Precedence.pdf) for details; (for example: *x = a + b \* c*; multiply first, add second, assign third).
  + Depending on the associativity of operands
    - When there is a group of operands with the same precedence, associativity tells us about the order in which they are evaluated:
      * *Left-to-right,* L-R (for example: *x = a + b+ c;* means *x = (a + b) + c;*)
      * *Right-to-left,* R-L (for example: *x = y = a;* means *x = (y = a);*).
      * Refer to the operator precedence table for details.
    - We can surround individual sub-expressions with brackets to change the order of precedence (for example: x = (a + b) \* c;).
    - Note: the order of precedence does not dictate the order over evaluation. If two separate expressions have the same precedence, a compiler can make a decision which to evaluate first (for example: *x = f1() +f2();* we know that the function call operator takes precedence before addition, but the order of calls to functions *f1()* and *f2()* is dependent on the compiler.
* Overview of operators:
  + Arithmetic and bitwise [[reference](https://en.cppreference.com/w/c/language/operator_arithmetic)] (unary and binary) - basic mathematical operations. Remember that depending on types of operands, they may get translated to different machine code operations. For example:  
    *i = 1 / 2; /\* 0;* integer division produces an integer \*/  
    *f = 1 / 2.0f; /\* 0.5f; float division produces a float value \*/*
  + **Pre/post increment/decrement** [[reference](https://en.cppreference.com/w/c/language/operator_incdec)] - prefix operators (*--i*, *++i*) first increment/decrement the value, then return a new value, while postfix operators increment/decrement the value, but then return an original (non-incremented/non-decremented) value.
  + **Logical** [[reference](https://en.cppreference.com/w/c/language/operator_logical)] - Boolean algebra applied to values considered as true when non-zero, or false when zero.
  + **Assignment and compound assignments** [[reference](https://en.cppreference.com/w/c/language/operator_assignment)]
  + **Comparisons** (we will cover in the next lecture)
  + Advanced
    - [Member access](https://en.cppreference.com/w/c/language/operator_member_access)
    - [Other](https://en.cppreference.com/w/c/language/operator_other)
* Example: [basic calculator](https://distance.sg.digipen.edu/file.php/3601/Examples/Lecture_06/calculator.c).

Recommended reading:

* Kim N. King, "C Programming: A Modern Approach", 2nd Edition
  + Chapter 4. Expressions
* Notes by DigiPen's Matthew Mead: [Expressions](https://distance.sg.digipen.edu/file.php/3601/Articles/Expressions/Expressions-1.html).
* **Lecture 7:**
  + Overview of operators (continued)
    - **Comparisons** [[reference](https://en.cppreference.com/w/c/language/operator_comparison)]
      * Risks of mistaking an assignment ("*=*") operator with check for equality ("*==*") [[example](https://distance.sg.digipen.edu/file.php/3601/Examples/Lecture_07/Comparison.c)]
    - **Short-circuiting** - short circuit evaluation happens with logical operators AND (*&&*) and OR (*||*).
      * Because with a logical AND any *false*value establishes the result as *false*, remaining operands do not have to be evaluated. This approach improves performance, but evades side-effects of evaluation of these operands, which may be undesired. [[example](https://distance.sg.digipen.edu/file.php/3601/Examples/Lecture_07/SpecialAnd.c)]
      * Because with a logical OR any *true*value establishes the result as *true*, remaining operands do not have to be evaluated. This approach improves performance, but evades side-effects of evaluation of these operands, which may be undesired. [[example](https://distance.sg.digipen.edu/file.php/3601/Examples/Lecture_07/SpecialOr.c)]
  + **Functions** - closer look
    - Elements of a function:
      * Return type (or *void* for none)
      * Name (starting with a letter or an underscore, containing a letter, an underscore, or a digit)
      * Zero (*void*) or more parameters
        + Formal parameters are parameter variables declared in a function.
        + Actual parameters are parameter values used in a function call.
      * Zero or more statements surrounded by curly braces (only in definitions, but not declarations; see below)
    - Definitions vs. declarations
      * **Declaration** - tells a compiler that there is a function with given name, specific parameters, and a specific return type. This information is sufficient for the compiler to translate calls to this function into assembly code. Later the linker needs to update the address of this function.
        + All functions must be at least declared (not necessarily defined) before they can be called; generally, C/C++ compilers are top-down compilers.
        + A function can be declared more than once, as long as each declaration is exactly the same.
        + Names of formal parameters in declarations can be omitted (good practice: do not omit them).
      * **Definition** - tells the compiler that there is a function with given name, specific parameters, a specific return type, and specific statements ("fine details"). This information is a full description of a function and it is sufficient for the compiler to translate calls to this function into assembly code, and to translate the statements of the function itself into assembly.
        + A definition also declares a function.
        + A function must be defined exactly once (either in the code being compiled, or in the object code provided to a linker).
      * If a function is *defined* before its use, there is no need for a *declaration*. If a function is *defined* after its use, or *defined* in a separate translation unit and it will be linked from an object file, a *declaration* must be provided before its first use. [[example](https://distance.sg.digipen.edu/file.php/3601/Examples/Lecture_07/DeclarationAndDefinition.c)]

Recommended reading:

* + Kim N. King, "C Programming: A Modern Approach", 2nd Edition
    - Chapter 9.1 Functions, section Defining and Calling Functions

**Lecture 8:**

* + Exercises with expressions and flow control statements
    - Conditions; selection statements - the user enters a positive number [[code](https://distance.sg.digipen.edu/file.php/3601/Examples/Lecture_08/program1input.c)]
    - Loops; iteration statements - the user enters a positive number, and the program displays that many "\*" symbols [[code](https://distance.sg.digipen.edu/file.php/3601/Examples/Lecture_08/program2stars.c)]
    - Nested loops - the user enters a positive number, and the program displays a "Christmas tree" with that many levels, that many rows, and up to that many stars in each row [[code](https://distance.sg.digipen.edu/file.php/3601/Examples/Lecture_08/program3tree.c)]
    - Complex example - the user enters a real number not less than 1, and the program calculates its square root [[code](https://distance.sg.digipen.edu/file.php/3601/Examples/Lecture_08/program4sqrt.c)]
  + Code's visual formatting
    - Using indentation to give the code a familiar structure.
    - Putting opening and closing brackets at the same indentation level as a way of forming blocks of statements easy to assess visually.