# Tutorial #2 Sequence AND Compilation Process

**Tasks set in tutorials should be finished in your own time!**

**Note – copying and pasting code from Word doesn’t always work as Word uses lots of odd special characters you cannot see, but the compiler can see them! If you start getting strange errors after pasting in code from Word, delete it and try writing it in by hand.**

## Compilation Process

Although C++ programs may appear complex they are written in a human-readable form (with a well-defined syntax). Before a computer can execute such programs, the C++ code must be translated into machine code, something the computer can understand.

This process involves a few stages:

* **Pre-processing**: The **pre-processor** filters the C++ source code before it is compiled. It processes include-files, conditional compilation instructions and expands macros. Pre-processor directives begin with hash sign (#).
* **Compiling**: The **compiler** takes the output of the pre-processor and checks that the source code is grammatically correct (follows the rules of C++) and makes (some) sense. If the compilation is successful it generates assembly code. The assembler then takes this assembly code and produces the equivalent machine code, also known as object code.
* **Linking**: In the last stage the **linker** takes the object code (from one or more object files) and combine them to create the executable code.

Errors can occur at various stages of this process,. They are categorised as compiler, linking, logical or runtime errors depending on the nature of the problem.

1. Put the following program in a new Visual Studio project, let’s say **FoP Project Age**.
2. Store the following source code in a new source file, let’s say **Tut02\_CalculateAge.cpp**..

#include <iostream>

using namespace std;

int main()

{

int birthYear;

cout << "Enter your birth year: ";

cin >> birthYear;

cout << "This year you are (or will be) " << 1901 - birthYear << endl;

system("pause");

return 0;

}

1. Your tutor will show you how to look at some of the files created by the system during the compilation process, there’s a bug in the code above:

* assembly code by right-clicking in the **Editor** pane before the break point for that source file and select "**Go to Disassembly**" option
* object code in the object file called **calculateAge.obj** file stored in the **Debug** folder
* executable code in the **FopProjectAge.exe** executable file stored in the **Debug** folder.

## ERRORS

**Compiler Messages (Warnings and Errors)**

When the compiler is compiling code (i.e., converting it into instructions the machine understands), it will report problems that it finds in the code. A compilation error message often helps programmers debugging the source code for possible errors.

* **Compiler Warnings** - A compiler warning (labelled warning) indicates something in the code is probably not quite right, but not something that will prevent the code from being compiled. Always remember to fix the warnings (even if they don't stop compilation they often lead to other problems that will not be so easy to find).
* **Compiler Errors** - A compiler error indicates misuse of the programming language (syntax) (e.g., missing bracket, wrong operator, misspelt identifier, etc.). The compiler has identified something wrong that must be fixed before the code can be compiled.

Compiler messages usually list the file and line number where it has detected a problem. However, most often errors occur before that point. Also, note that some compilers may choose to call something an error while others may just call it a warning or not complain at all.

Compiler errors prevent the program from being compiled (no **.obj** generated).

**Linker Errors**

Linking is when all the compiled pieces of a program and the libraries it uses (e.g., for cin) are put together from obj files to form an executable. A linker error means that although the code can compile, some function or library that is needed cannot be found. It may be that the library that contain that function has not been included in the code, but most often it’s the name of the function that is simply misspelt or the signature of the function definition, declaration or call do not match each other.

Linking errors prevent an executable from being generated (no **.exe** generated).

**Run-Time Errors**

Run-time errors only occur when a program is executed (and thus, the program must have compiled and linked without errors). There are two main types of run-time errors:

* **Fatal Errors** - A fatal error crashes the executable (e.g., the program divided by zero, the program tries to access memory that your program is not allowed to use or that doesn't exist in the computer, often occur due to improper use of arrays or pointers).
* **Logic Errors** - A logic error simply stops the program from working as expected (e.g., a variable has not been initialised properly, an infinite loop is created because a variable(s) used in the condition of that loop is not updated properly).

To help find the cause of a run-time error it is recommended to narrow down the location of the error and circumstances in which it occurs and to get more information about what is happening in the program (design and execution). This can be done with or without a debugging utility, but it requires a methodical approach to programming.

1. For each of the following simple programs, find the errors produced, identify their type (compiler, linking, logical or runtime error), explain and fix them.

int main()

{

srand(static\_cast<int>(time(0)));

cout << "Roll dice! Value: " << (rand()% 6) + 1 << endl;

system("pause");

return 0;

}

#include <iostream>

using namespace std;

int main()

{

cout << "Enter your birth year: ";

cin >> birthYear

cout << "This year you are (or will be) << 2018 - birthYear << endl;

system("pause");

return 0;

}

#include <iostream>

using namespace std;

int main();

{

cout << "Anyone there? " << endl;

system("pause");

return 0;

#include <iostream>

using namespace std;

int Main()

{

int value;

cout << "Roll dice! Value: " << (rand()%6)+1 << endl;

system("pause");

return 0;

}

#include <iostream>

using namespace std;

int main()

{

int tokens, redTokens;

cout << "Enter number of tokens in bag: ";

cin >> tokens;

cout << "Enter number of red tokens: ";

cin >> redTokens;

cout << "There are " << (redTokens / tokens) \* 100 << "% red tokens in bag.";

cout << endl;

system("pause");

return 0;

}

#include <iostream>

using namespace std;

int main()

{

char letter;

cout << "Enter letter: ";

cin >> letter;

cout << "Uppercase of " << letter << " is " << toupper(letter) << endl;

system("pause");

return 0;

}

#include <iostream>

using namespace std;

int main()

{

char letter;

cout << "Enter a letter: ";

cin >> letter;

cout << "The ASCII number for ", letter, " is ", static\_cast<int>(letter);

cout << endl;

system("pause");

return 0;

}

#include <iostream>

using namespace std;

int main()

{

int dice;

cout << "Enter a dice value: ";

cin >> dice;

if (1 <= dice <= 6)

cout << "Dice face is : " << dice << '\n';

else

cout << "ERROR: Invalid dice face!" << '\n';

system("pause");

return 0;

}

1. ***(In your own time)*** Deliberately introduce errors (one at a time) in the program to get used to the error messages that this cause. You can also look on the internet for details of these errors (just search indicating the error number). You are probably not yet in a position to understand them all, but it is a process you need to get familiar with.

## SEQUENCES

A **sequence** controls the order in which statements are executed.

* In pseudocode, a sequence is represented one instruction per line and each instruction in the sequence should start at the same level of indentation.
* In flowchart, one box for each instruction shown successfully along the flow of control arrow.
* In C++, each instruction is separated by the semicolon operator (;). By convention the instructions in a sequence are also expected to appear at the same level of indentation.

In C++ instructions can be grouped into a single instruction using a pair of curly brackets ({});

1. A time given in seconds is to be converted in digital format.   
   The program asks for and reads in the time given as seconds.   
   It then calculates the matching number of hours, minutes and seconds and displays the time in a digital clock format (e.g., 3723 seconds shows as 01:02:03).  
   *NOTE: Assume that the time given is valid (i.e. represents a time under 24h - possibly 0 seconds).*

convertTimeinHMSintoSeconds

local timeInSeconds, hours, minutes, seconds

output("Enter the time in seconds: ")

input(timeinseconds)

set hours to timeInSeconds div 3600

set secondsLeft to timeInSeconds mod 3600

set minutes to secondsLeft div 60

set seconds to secondsLeft mod 60

output("The time is: ", hours, ":", minutes, ":", seconds)

1. Write the program in C++.
2. Search on the web for information on output manipulators fill and setwto find out how to format the output so that your program produces times as shown below.

|  |
| --- |
| Enter the time in seconds : 3732  The time is : 01:02:12 |

1. Look at the problem specification given below. The pseudocode solution given is incomplete, but you now know enough now to tackle this new problem.

### Specification:

The program should compute the annual salary and monthly salary for an employee in December.

In this company the annual salary is calculated from the salary point for an employee. Salary points are positive numbers from 1 to 24. Each salary point is worth £2000.00 per annum. In December, in addition to their usual monthly salary, employees are also given a Christmas bonus equivalent to 2.5% of their annual salary.

The program lets the user enter the salary point for the employee and displays the total annual salary (incl. the Christmas bonus) and the salary this employee will receive in December, which also includes the Christmas bonus.

*Assume that the user enters a whole number representing a valid salary point (i.e., no need to validate this).*

Look at the following partial pseudocode

computeDecemberSalary

*//declare local variable and constants here*

output("Enter the salary point: ")

input(salaryPoint) //read in the salary point

*//place missing code here*

output("Annual salary is: £", annualSalary) //output annual salary (incl. bonus)

output("December monthly salary is: £", decemberSalary ) //output December salary

The interaction with the program is expected to produce the following behaviour (with user's input shown in bold):

|  |
| --- |
| Enter the salary point: **6**  Annual salary is: £12300.00  December monthly salary is: £1300.00 |

### Instructions:

* 1. *Use the sequence construct (even if you know how to: NO need for procedures, selection or repetition).*
  2. *Except for adding missing instructions, keep the overall structure and sequence of instructions in the top algorithm given above unchanged.*
  3. *Declare the variables used in the program and any additional variables and named constants you may need.*
  4. *Keep the identifiers for named data items given in this program unchanged in your solution and choose equally meaningful identifiers for any additional named data items you may need.*
  5. *Use comments to give information about your program (incl. the type of each variable and named constants used) so as to make your solution easy to understand.*
  6. *Format the output as shown above.*

1. So, let’s think before we start. Identify the various steps needed for your solution.   
   Remember? As if you were describing your method to an attentive (but maybe not too bright) friend over the phone…
2. Write your solution in pseudocode or flowchart (whichever works best for you, for now – you will know both notations eventually!)
3. Test your solution on paper. Identify which data values you will test for the salary point, and for each value explain why you selected it, what outputs do you expect and whether it is what your program produces.
4. Code this in C++.
5. Once you have an executable test it again using the same values. If you detect a problem, make any necessary amendment(s) and test again!

## Procedures

**A** **procedure** is a sub-program that can be called when needed. When a procedure is called the calling program is suspended, the statements the procedure contains are executed, then, when it finishes, the calling program resumes its execution.  
A procedure may in turn involve other procedures

When using procedures it is important to specify the **scope** of the data items (variables or named constants). When they are used only in one procedure they should be declared as local. When they need to be communicated across several procedures they will have - for now - to be declared as global (we don’t like globals).

* **Global** data items are active for the duration of the program execution and visible and accessible from any procedures (including in the top algorithm).
* **Local** data items are defined to a given procedure (including the top algorithm): they are only active for the duration of the execution of that procedure and only visible and accessible from within that procedure.

1. Look at the following pseudocode example which describes the sequence of sub-tasks to be executed, in that order, to convert imperial measurements in the metric system.

convertLengthFromImperialToMetric

call readInImperialMeasurement

call convertFeetAndInchesIntoMetres

call showMetricMeasurement

The above **convertLengthFromImperialToMetric** algorithm is appropriate as a top-level algorithm for converting a given length from the imperial to the metric system. However in order to be useful, these high level tasks need to be *refined* into more specific instructions. In other words the top algorithm will needs to be broken down into **procedures**, each of them describing in more detail parts of the action(s) to be performed, e.g.,

proc readInImperialMeasurement

output("Enter the number of feet: ")

input(feet)

output("Enter the number of inches: ")

input(inches)

endproc

proc convertFeetAndInchesIntoMetres

set lengthInInches to ((feet \* inchesPerFoot) + inches)

set lengthInMs to ((lengthInInches \* cmsPerInch) / 100)

endproc

proc showMetricMeasurement

output("The length is: ", lengthInMs, " metres")

endproc

1. Identify the procedure definition (or declaration), the procedure heading, and the procedure call of the **readInImperialMeasurement** procedure.
2. Identify the data items used in this program. For each of them decide its scope (global or local) and show how they should be declared. Use constant when appropriate and indicate with a comment their recommended data type.
3. Draw the flowchart corresponding to this program.
4. A procedure may in turn involve other procedures. The **convertFeetAndInchesIntoMetres** procedure could be further refined as follows.

proc convertFeetAndInchesIntoMetres

call calculateTotalInches

call convertlInchesToMetres

endproc

Write the definition for both of the new used here. If necessary amend the data items declarations.

## (Simple) Functions

Pseudocode procedures are implemented in C++ as **functions**.  
There are various ways in which such functions receive information from and send information back to other parts of the program. To start with we are only going to consider simple functions which do not take any parameter and do not return any value: If they need to communicate data with the rest of the program they will do so through global variables (nasty).

In C++,

* a (simple) **function definition** defines what the function does when it is called, including which data items are created during the function execution and destroyed when it finishes. Its heading consists of the keyword void, the function name and a set of empty round brackets () (the call operator).   
  The function body is placed between curly brackets, e.g.,

Function heading

void function\_name()

{

Function body including local data declarations and statements

//code here

}

* a **function call** uses the function name followed by the function call operator (), e.g.,

function\_name()

* a **function prototype** is similar to the heading of the function declaration. It gives a forward reference to a function definition that appears later on in the program, e.g.,

void function\_name();

In C++, each **global** variable is declared at the top of the program, before the main function, while each **local** variable is declared before it is used, in the function that needs it.

Whenever possible

* Use local named data items (rather than global data items).
* Declare named data items as constant using the const keyword.

This will help with testing your code later on.

1. Look at the following C++ code for the program presented earlier, written as a sequence of instructions placed in the **main** function:

#include <iostream> //for cin >> and cout <<

#include <iomanip> //for i/o manipulators

using namespace std;

// implements convertLengthFromImperialToMetric program

int main()

{

cout << "Enter the number of feet: ";

cin >> feet;

cout << "Enter the number of inches: ";

cin >> inches;

//get the measure in inches

lengthInInches = (feet \* inchesPerFoot) + inches;

//convert it to metres

lengthInMs = (lengthInInches \* cmsPerInch) / 100;

cout.setf(ios::fixed);

cout << setprecision(2);

cout << "The length is: " << lengthInMs << " metres\n" << endl;

system("pause"); //to hold the output screen

return 0;

}

1. Declare variables and constants needed for this program to work? For each of them select the data type carefully.
2. Make the necessary changes so that, as in the pseudocode developed earlier, it uses three functions that handle different sub-tasks:

readInImperialMeasurement

convertFeetAndInchesIntoMetres

showMetricMeasurement

1. Add two new functions to be called from the convertFeetAndInchesIntoMetres function, making any necessary changes to the data items used in the program

calculateTotalInches

convertlInchesToMetres

## Procedure dependency diagrams

**Procedure Dependency Diagrams** (PDD) simply show static relationships that exist between procedures/functions used in the same program.

* A box in this diagram represents either the top-level algorithm or one of the sub procedures.
* An arrow on the diagram indicates a dependency link between two procedures. There are as many arrows coming out of a procedure as sub-procedures it uses.

A PDD shows the **static relationships between procedures** (and top level algorithm). It also reflects **top-down refinement approach** used to design that program and those procedures. When a procedure is used in several places in a program, several arrows point to the same box in the matching diagram. That is OK, but there should be NO cycles in this diagram (two functions calling each other).

A PDD will related directly to the use of prototypes in the associated C++ program.

1. Produce the PDD for the code developed in previously.
2. List the main advantages for using procedures/function.

## Your turn…

1. Look at the following C++ code:

#include <iostream> //for cin >> and cout <<

using namespace std;

int n1, n2, temp;

void swapTwoValues() {

temp = n1;

n1 = n2;

n2 = temp;

}

int main() {

//read in two values in n1 and n2

cout << "\nEnter integer for n1: ";

cin >> n1;

cout << "\nEnter integer for n2: ";

cin >> n2;

//display the values read in

cout << "\nBefore swap: n1 is " << n1 << " and n2 is "<< n2;

//exchange their values

swapTwoValues();

//display their new values

cout << "\nAfter swap: n1 is " << n1 << " and n2 is "<< n2 << "\n";

system("pause"); //to hold the output screen

return 0;

}

1. There are three global variables used in this program. What is their type?   
   For each of them decide which scope they should have. Check which one(s) could be made constant and indicate how.
2. Make the necessary changes in the program to allow you to place the main function at the beginning of the program.
3. Check that this program works properly (assuming that the user enters whole numbers - integers).
4. What changes are needed in this program to make it work for floating point numbers, characters, strings?
5. Is it possible to have two different functions in the same program, one working for int and the other for char for example?

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1. **ASSESSED** **HOMEWORK** Look at the C++ program for the annual salary problem mentioned earlier. I hope you had a go without looking at this solution! ☺ :   #include <iostream>  #include <iomanip>  using namespace std;  int main() //computeDecemberSalary  {  const float scalePointValue(2000.0f);  const float bonusRate(2.5f);  //read in the salary point  cout << "Enter the salary point: ";  int salaryPoint;  cin >> salaryPoint;  //produce annual base salary (without bonus)  const float annualBaseSalary(salaryPoint \* scalePointValue);  //produce Christmas bonus  const float christmasBonus((annualBaseSalary \* bonusRate) / 100.0f);  //produce final annual salary (incl. bonus)  const float annualSalary(annualBaseSalary + christmasBonus);  //produce monthly salary  const float monthlySalary(annualBaseSalary / 12);  //produce December salary (incl. bonus)  const float decemberSalary(monthlySalary + christmasBonus);  const unsigned char poundSign(156);  //output annual salary  cout.setf(ios::fixed);  cout << setprecision(2);  cout << "Annual salary is: " << poundSign << annualSalary;  //output December salary  cout << "December monthly salary is: " << poundSign <<decemberSalary;  cout << endl;  system("pause");  return 0;  }   1. Check that it works as expected and compare it to your own solution. Which one is better and why? Maybe they are equally good. 2. Rewrite this program using the following functions to handle separate tasks as shown in the pseudocode design given below.   computeDecemberSalary  output("Enter the salary point: ")  input(salaryPoint) //read in the salary point  call calculateAnnualBaseSalary //work out annual salary (no bonus)  call calculateChristmasBonus //work out Christmas bonus  call calculateAnnualSalary //work out final annual salary  call calculateDecemberSalary //work out December salary  output("Annual salary is: £", annualSalary) //output annual salary  output("December monthly salary is: £", decemberSalary ) //output December salary   1. Get a copy of the DESIGN DOC TEMPLATE and fill it in (d+e+g below go in it) 2. Produce the flowchart for this program (along with the usual spec, pseudocode). 3. Produce the Procedure Dependency Diagram (PDD) for this program. 4. Code it in C++, using functions and paying attention to the type and scope of the data items you use, declaring them as const whenever possible. 5. Design a set of test data for this program and test it. You do not have to check the type of the input (i.e. you can assume that the user will only enter an integer value in the required range). Put your result s in a table as follows:  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | Input **salaryPoint** | Rationale | Output  **annualSalary** | | Output  **decemberSalary** | | | Expected | Observed | Expected | Observed | | 6 | Integer value, in range | 12300.00 |  | 1300.00 | ? | | ? etc. |  |  |  |  |  |   Competencies: 1.1,1.2.1,1.2.3,1.4,1.6.1,1.6.2,1.6.3,2.7.1 |