Chapter 1: Concepts in Computer Programming and the C Program Structure

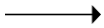
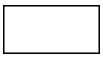
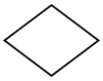
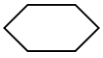
Lesson 1: Problem solving fundamentals

Learning Outcomes

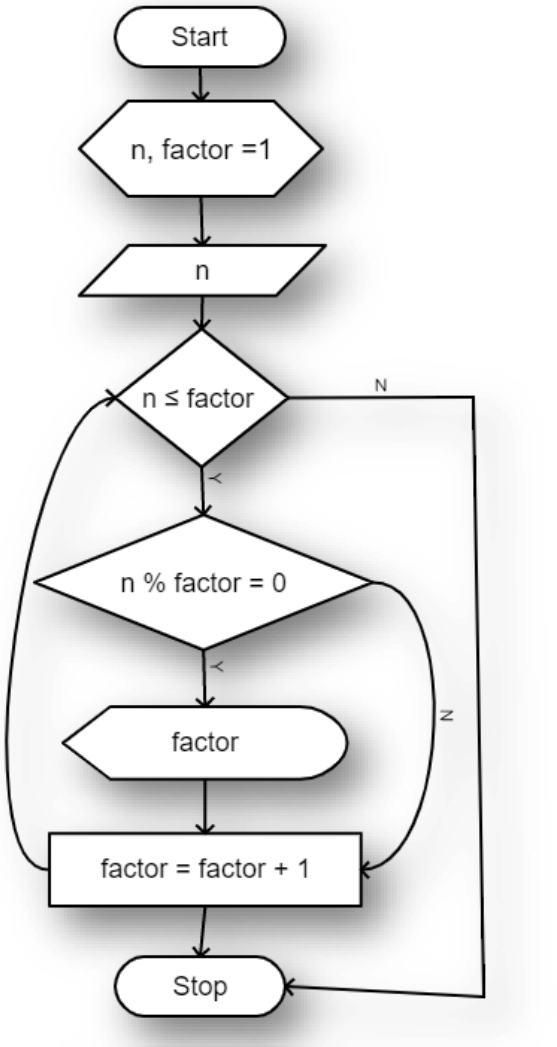
- Analyze the logical flow for solving a particular problem
- Implement flowchart and pseudocode to solve a given problem.

Flowcharts

Flowchart - pictorial representation of an ordered step-by-step solution to a problem. It consists of arrows that represent the direction of process flow and of boxes and other symbols that represent actions.

Shape	Name	Description
	Flow Line	The arrow would direct the process, it must begin from one symbol and then terminate to another
	Terminal	This would contain the start or end of the program.
	Process	Used to show that a process is performed. E.g. z = x + y
	Decision	Commonly answerable by yes or no, true or false. The arrow that would indicate the direction must always be labeled if it is the true path or the false path.
	Input / Output	This involves the data that is received by the user or the data that is produced by the program
	Preparation	This shows the operation that is required to declare and initialize a data prior to a process or condition.
	Display	Indicates a process step where information is displayed to a person (e.g., PC user, machine operator).

Example: Displaying the multiples of input n



% - Modulo (get remainder)

Pseudocode

An English-like language that you can use to state your solution with more precision than you can in plain English but with less precision than is required using a formal programming language. In general it does not obey any rules of any particular language; there is no systematic standard form however writers borrow style and syntax from control structures from some conventional programming language. Below are some statements types and operations used.

Statement Type	Statements
Conditional	If
Iterative	for, while
User Input	input
Terminal Output	display

Type of operation	Symbol
Assignment	← or :=
Comparison	=, ≠, <, >, ≤, ≥
Arithmetic	+, −, ×, /, mod

Example: Displaying 1 up to 10 centimeter into inches conversion table

Pseudocode (using while)

cm:=1
while cm ≤ 10
  display cm *2.54
  cm=cm+1

Lesson 2: Why learn C as your first programming language?

Lesson Summary

Learning Outcomes

- Explain the importance of C language
- Examine the MinGW C compiler and Code::Blocks IDE for actual coding
- Inspect the program development lifecycle process

Discussion

History

The Combined Programming Language (CPL) was then created out of Algol 60 in 1963. In 1967, it evolved into Basic CPL (BCPL), which was itself, the base for B in 1969. C was the direct successor of B in 1971, a stripped down version of BCPL, created by Ken Thompson at Bell Labs that was also a compiled language used in early internal versions of the UNIX operating system. Thompson and Dennis Ritchie, also working at Bell Labs, improved B and called the result NB. Further extensions to NB created its logical successor, C. Most of UNIX was rewritten in NB, and then C, which resulted in a more portable operating system.

The portability of UNIX was the main reason for the initial popularity of both UNIX and C.
Rather than creating a new operating system for each new machine, system programmers could simply write the few system-dependent parts required for the machine, and then write a C compiler for the new system. Since most of the system utilities were thus written in C, it simply made sense to also write new utilities in C.

The American National Standards Institute began work on standardizing the C language in 1983, and completed the standard in 1989. The standard, ANSI X3.159-1989 Programming Language C″, served as the basis for all implementations of C compilers. The standards were later updated in 1990 and 1999, allowing for features that were either in common use, or were appearing in C++.

Why not use Assembly?

While assembly language can provide speed and maximum control of the program, C provides portability. Different processors are programmed using different Assembly languages and having to choose and learn only one of them is too arbitrary. In fact, one of the main strengths of C is that it combines universality and portability across various computer architectures while retaining most of the control

of the hardware provided by assembly language. Assembly, while extremely powerful, is simply too difficult to program large applications and hard to read or interpret in a logical way. C is a compiled language, which creates fast and efficient executable files. It is also a small "what you see is all you get" language: a C statement corresponds to at most a handful of assembly statements, everything else is provided by library functions.

Why not use other high level programming language?

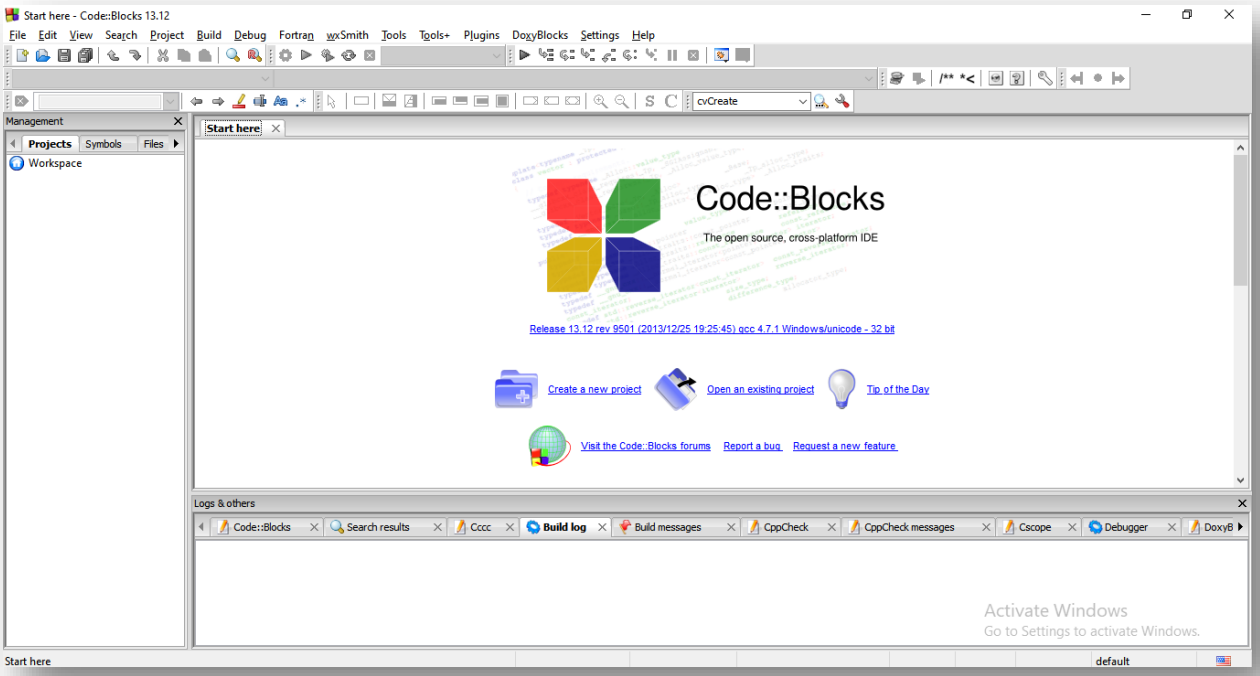
The primary design of C is to produce portable code while maintaining performance and minimizing footprint (CPU time[18], memory[19] usage, disk I/O, etc.). This is useful for operating systems, embedded systems[21] or other programs where performance matters a lot ("high-level" interface would affect performance). With C it's relatively easy to keep a mental picture of what a given line really does, because most of the things are written explicitly in the code. C has a big codebase for low level applications. It is the "native"language of UNIX, which makes it flexible and portable. It is a stable and mature language which is unlikely to disappear for a long time and has been ported to most, if not all, platforms.

One powerful reason is memory allocation. Unlike most programming languages, C allows the programmer to write directly to memory. Key constructs in C such as structs, pointers and arrays are designed to structure and manipulate memory in an efficient, machine independent fashion. In particular, C gives control over the memory layout of data structures. Moreover dynamic memory allocation is under the control of the programmer (which also means that memory deallocation has to be done by the programmer). Languages like Java and Perl shield the programmer from having to manage most details of memory allocation and pointers (except for memory leaks and some other forms of excess memory usage). This can be useful since dealing with memory allocation when building a high-level program is a highly error-prone process. However, when dealing with low-level code such as the part of the OS that controls a device, C provides a uniform, clean interface. These capabilities just do not exist in most other languages.

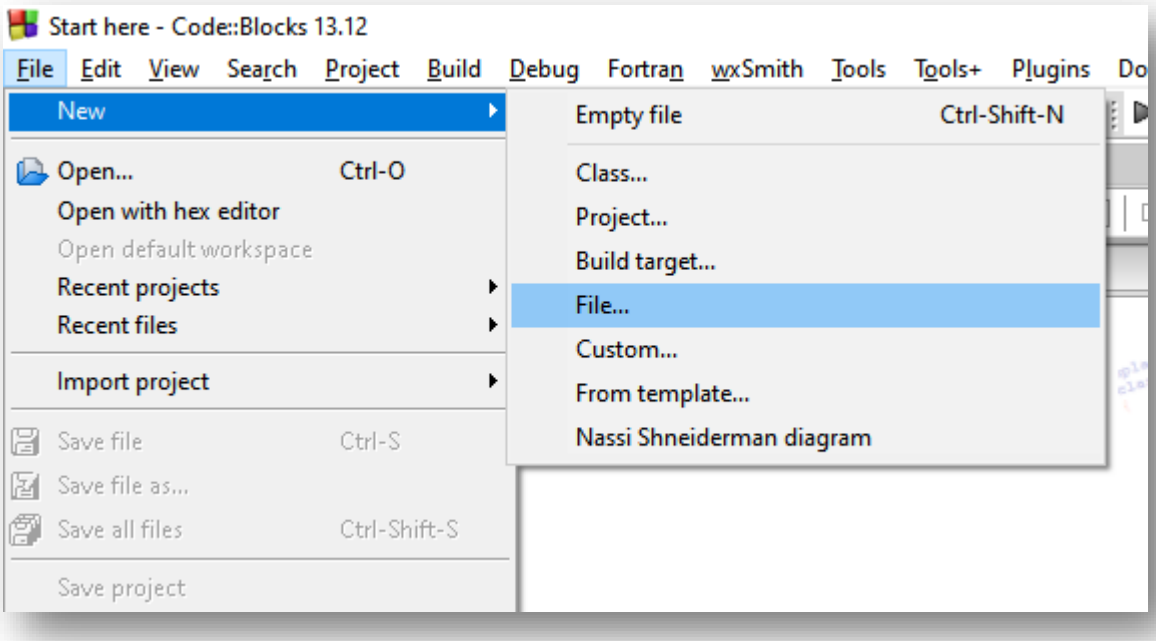
While Perl, PHP, Python and Ruby may be powerful and support many features not provided by default in C, they are not normally implemented in their own language. Rather, most such languages initially relied on being written in C (or another high-performance programming language), and would require their implementation be ported to a new platform before they can be used.

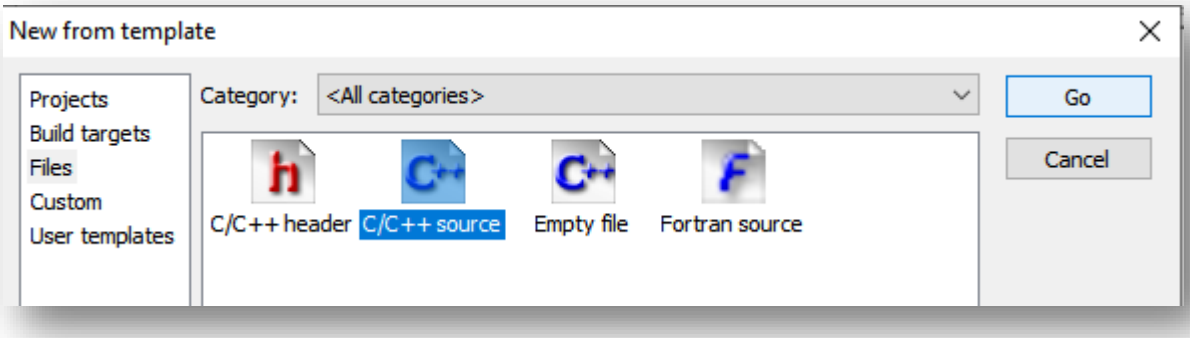
Getting started (MinGW compiler in Code::Blocks IDE)

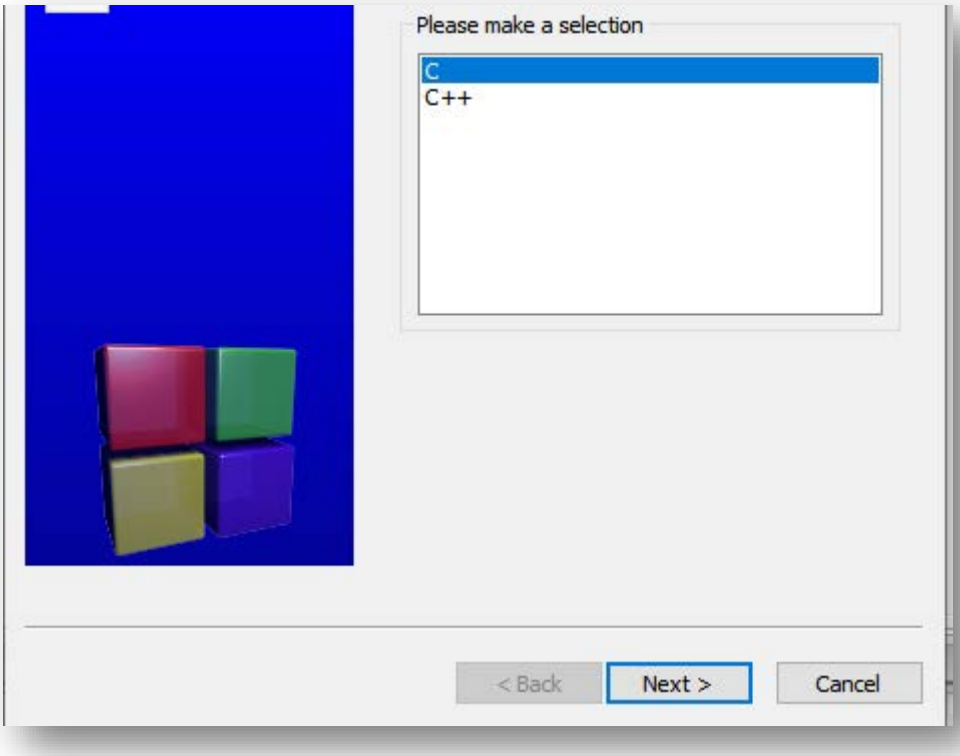
1. Download the Code::Blocks 20.03 http://www.codeblocks.org/downloads/binaries. Select codeblocks-20.03mingw-setup.exe (for windows). This installer would include the Integrated Development Environment (IDE) together with the C MinGW compiler.
2. Install the executable module by opening it and following the on screen instructions.
3. Open the installed app which will greet you with the splash screen and user interface as shown below (note: the screen shot below is the older version but newer version would generally look the same).

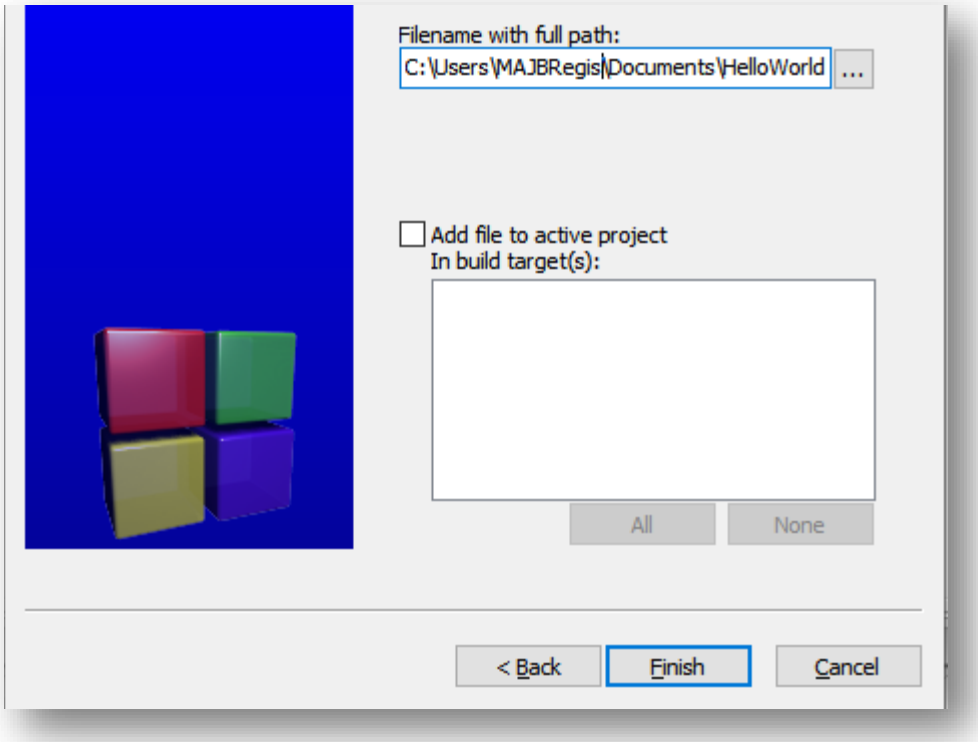


4. To write your code go to File-New-File. Then click C/C++ Source. Click go, click C, then click next. Provide the filename and the path of the source code. Then click finish

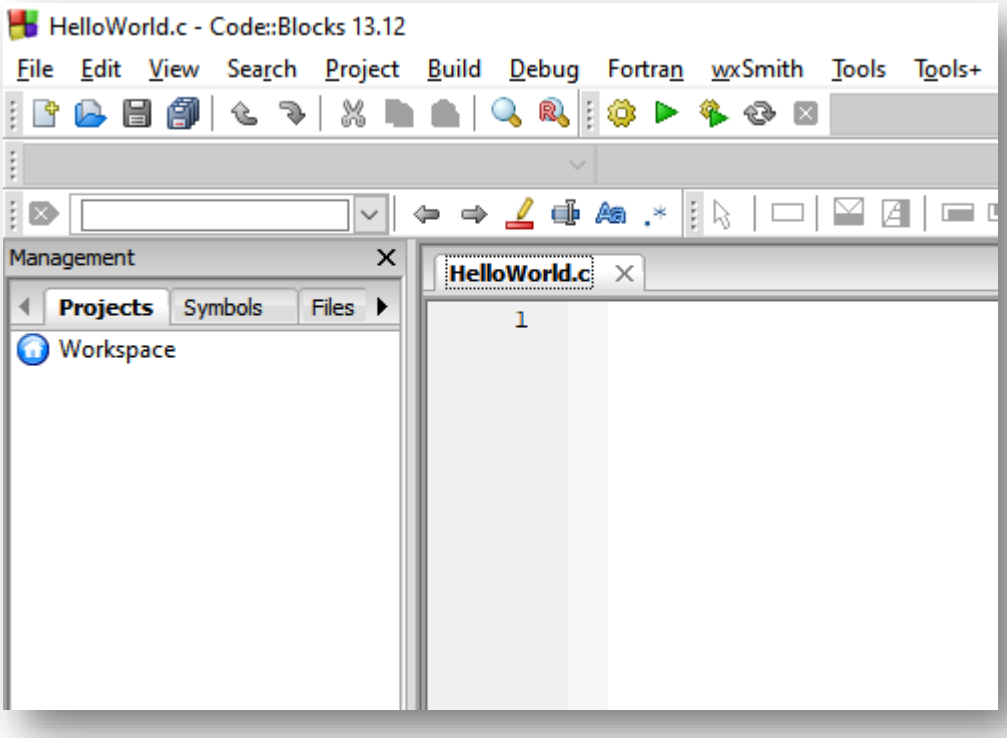



New from template
Projects
Build targets
Files
Custom
User templates
Category:
<All categories>
Go
Cancel
C/C++ header
C/C++ source
Empty file
Fortran source


Please make a selection
C
C++
< Back
Next >
Cancel

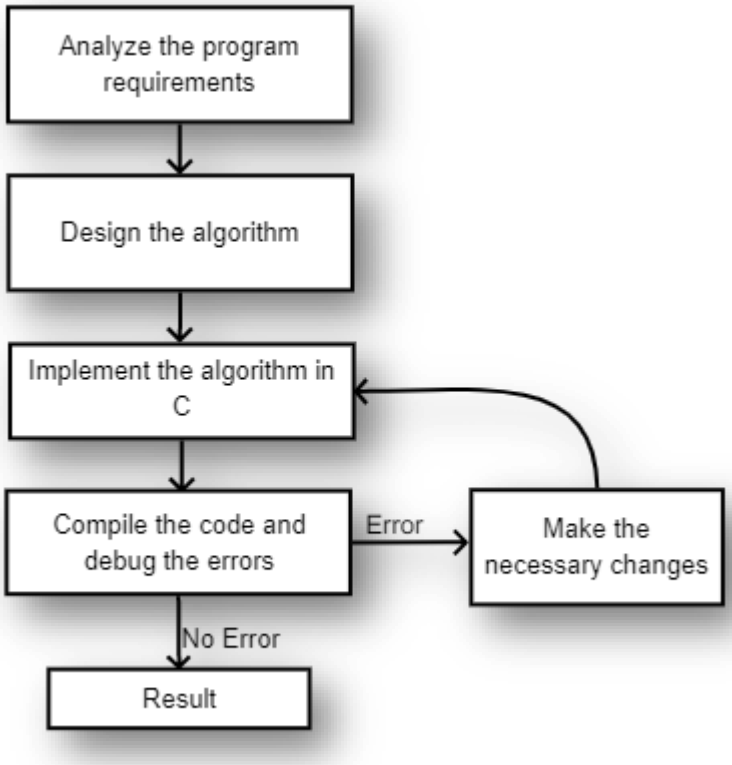


5. You may write your code on the text space provided.



Program development Lifecycle

Programming is basically systematic problem solving, so the program lifecycle as shown below is a guide on how to solve a particular problem.



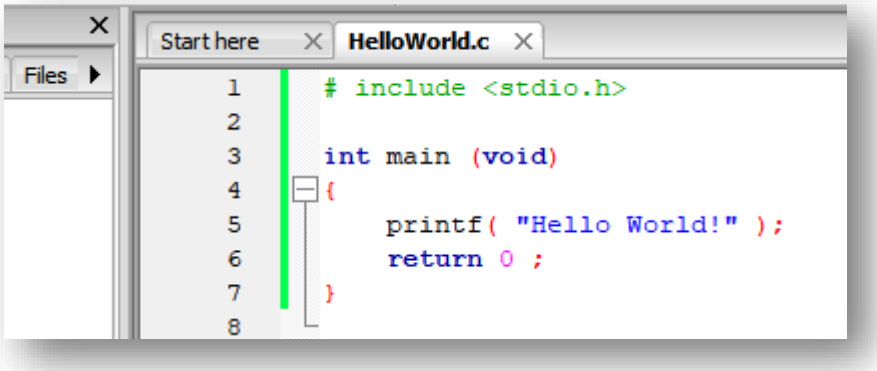
Lesson 3: “Hello World” your first C program

Learning Outcomes

- Write, compile and execute “Hello World” C code
- Analyze the content of the “Hello World” C code.

Source code writing and analysis

After creating the “HelloWorld.c” file write the “Hello World” source code similar to what is shown below.

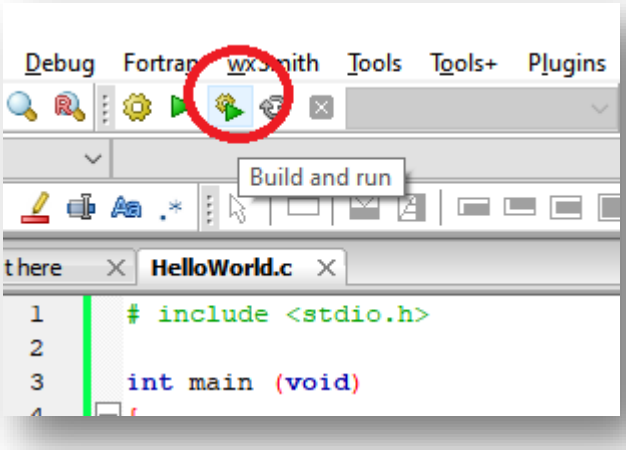


Source code components

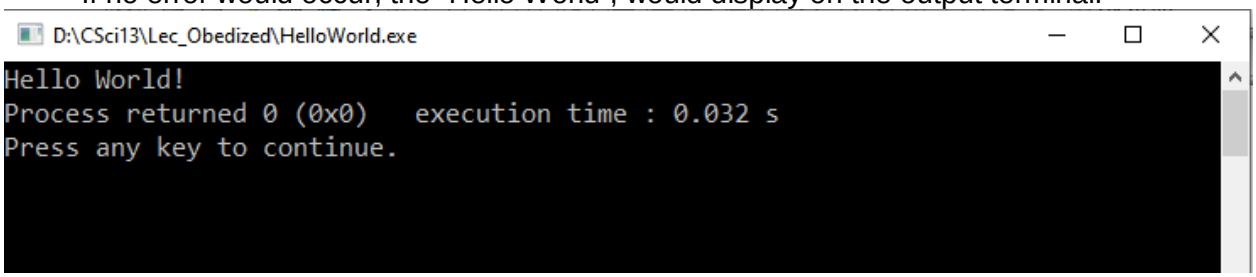
- #include <stdio.h> this is the header file that would allow us to use standard input and output functions. Most of your source code on the latter part of the course would include header files for specific support of specialized functions.
- int main (void) this is the function that would be executed first, sort of a gateway function of your source code. Note: all components that belong to function main is bounded by a pair of curly braces.
- printf("Hello World"); the output function to display the “Hello World” text string on your output terminal.
- return 0; - the return value of function main after execution.

Compiling and Executing

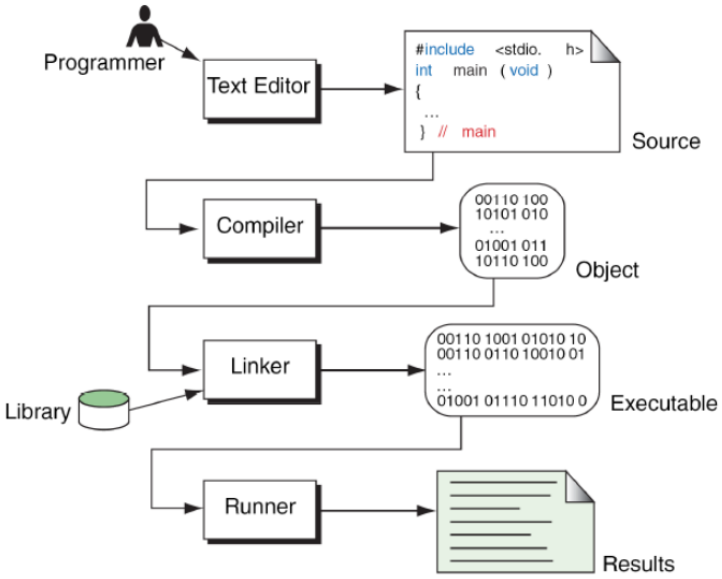
To compile and execute the code, click the gear and play icon on the menu bar as shown below.



If no error would occur, the “Hello World”, would display on the output terminal.



The entire compilation process follow first convert your source code into an object code then a linker would require connect to the necessary libraries for your object code. Then finally the binary executable module would now be created for you to run on your machine.


Programmer
Text Editor
#include <stdio. h>
int main (void)
{
...
} // main
Source
Compiler
00110 100
10101 010
...
01001 011
10110 100
Object
Linker
Library
00110 1001 01010 10
00110 0110 10010 01
...
...
01001 01110 11010 0
Executable
Runner
Results

Lesson 4: Preliminaries in a C code and Coding Style

Learning Outcomes

- To implement the proper grouping of statements in a C code using block structures.
- To use input and output statement in C code
- To explore line breaks, indentions and white spaces to format a C code properly.

Blocks structure

A block consists of executable **statements**. One way to put it is that statements are the text the compiler will attempt to turn into executable instructions, and the whitespace that surrounds them.

```
printf("Hello World");
```

You might have noticed the semicolon at the end of the statement. Statements in C always end with a semicolon (;) character. Leaving off the semicolon is a common mistake that a lot of people make, beginners and experts alike!

In C, blocks begin with an opening brace "{" and end with a closing brace "}". Blocks can contain other blocks which can contain their own blocks, and so on.

Input and Output Statements

The `printf` function allows you to send output to terminal output (standard terminal). The usual for of the output statement is to display a string of characters bounded by double quotations. For example:

```
printf("Hello World");
```

Another is to use a placeholder (or variable format specifier) that would be replaced by the actual variable when the printf statement is executed. For example:

```
int foo = 1;
printf("Hello World: %d",foo);
```

The primitive C data types (to be discussed in lesson 3) can be printed with printf by using different placeholders:

- int (integer) %d
- float (floating point) %f
- char (character) %c

The scanf function allows you to accept input from a keyboard (standard in). The function would still need a placeholder similar to printf but instead of the variable name immediately after a quotation mark it would require you to have the ampersand (&) symbol before the variable name. For example:

```
int fooBar = 0;
scanf("%d",&fooBar);
```

Although the scanf function can perform simple input, it is generally unreliable because it does not handle human errors very well.

White spaces

This would refers to the tab, space and newline characters that separate the text characters that make up the source code. It's hard to appreciate whitespace until it's gone. The code below would all look the same on a C compiler.

```
printf("Hello world");
return 0;
```

which is also the same as

```
printf (
"Hello world") ;
return 0;
```

The compiler simply ignores most whitespace (except when it separates e.g. return from 0). However, it is common practice to use spaces (or tabs) to organize source code for human readability.

Line breaks and Indentions

The addition of white space inside your code is arguably the most important part of good code structure. Effective use of white space can create a visual scale of how your code flows, which can be very important when returning to your code when you want to maintain it. However without using line breaks your code would be barely readable, take for example:

```
#include <stdio.h> int main (void){printf("Hello World!"); return
0 ;}
```

Rather than putting everything on one line, it is much more readable to break up long lines so that each statement and declaration goes on its own line.

```
#include <stdio.h>
 int main (void)
{
printf("Hello World!");
return 0 ;
}
```

Although adding simple line breaks between key blocks of code can make code easier to read, it provides no information about the block structure of the program. Using the tab key can be very helpful now: indentation visually separates paths of execution by moving their starting points to a new column in the line.

```
# include <stdio.h>

int main (void)
{
    printf( "Hello World!" );
    return 0 ;
}
```

As you can see the code inside a new block is indented by one tab. This would make your code visually appealing and easier to follow the appropriate hierarchy.

Comments

Comments in code can be useful for a variety of purposes. They provide the easiest way to set off specific parts of code (and their purpose. Having good comments throughout your code will make it much easier to remember what specific parts of your code do. Single line comments is made by placing // before a specified text.

```
// This is a single line comment
```

Single-line comments are most useful for simple 'side' notes that explain what certain parts of the code do. The best places to put these comments are next to variable declarations, and next to pieces of code that may need explanation. On the other hand multi-line comments is made by bounding multiple lines of text in /* */.

```
/*
This
is
a
multi-line
comment
*/
```

Multi-line comments are most useful for long explanations of code. They can be used as copyright/licensing notices, and they can also be used to explain the purpose of a block of code. This can be useful for two reasons: They make your functions easier to understand, and they make it easier to spot errors in code. If you know what a block is supposed to do, then it is much easier to find the piece of code that is responsible if an error occurs.

Lesson 5: Variables and Operators

Learning Outcomes

- Declare, initialize and assign values to variables in C
- Manipulate the values of variables using arithmetic operators
- Utilize built-in math functions for certain problems

Declaring, initializing and Assigning

Variables are simply names used to refer to some location in memory – a location that holds a value with which we are working. Declaring variables is the way in which a C program shows the number of variables it needs, what they are going to be named, and how much memory they will need. Since a variable is placeholder of value to initialize it you would use the assignment (=) operator to do so. For example:

```
int var = 10 // assign var integer with a value of 10
```

All variables in C must be declared first before initialized or used. The comma operator is used when including multiple variable with similar datatypes declaration in one statement. For example

```
int var1, var2,var3,var4;
```

Naming variables

The Variable names in C can be composed of letters (upper and lower case) and digits and underscore. Variables may begin with letters and underscore except digits.

Example:

```
int foo, _foo, f00, //legal declarations
int 1foo, 0f00; //illegal declarations
```

The basic datatypes

There are 3 standard variable datatypes in C:

- Integer: int
- Floating point: float
- Character: char

The int is a 4-byte (32 bit) integer value meaning it would not accept any decimal place. A float is a 4-byte floating point value meaning it would support decimal place. A char is a 1-byte single character (ASCII characters).

There are a number of derivative types:

- double (8-byte floating point value)
- short (2-byte integer)
- unsigned short or unsigned int (positive integers, no sign bit)

Examples:

```
int integerVar = 10;
float floatVar = 10.12;
char charVar = 'A';
short shortIntVar = 12;
double doubleVar = 101.13;
```

Arithmetic operators

The following operators exist :

- + (addition),
- - (subtraction),
- * (multiplication),
- / (division), and
- % (modulus); returns the remainder of a division (e.g. 5 % 2 = 1).

These are all infix operators meaning they must be between to operands. For example:

```
int a, b;
a + b; //legal
+ab//illegal
```

Assignment, increment and decrement operators

The = (assignment operator), would store value of the right hand operand to the memory location of the left hand operand (also known as lvalue). The other assignment operators are used in conjunction with arithmetic operators such as the *=, /=, %=, +=, -=. Usage example are as follows:

```
int x=10,y=2;

x*=y;//x=x*y, after evaluation x will be 20
x/=y; // x which is 20 will be divided by 2, x will be 10;
x+=y; // x which is 10 will be added with 2, x will now be 12
x-=y; // x which is 12 will be subtracted with 2, x will now be 10
x%=0; // x which is 10 will be modulo with 2, x will now be 0
```

Other variation of assignment operator with a fixed increment and decrement by one is the post fix ++ or – respectively

```
int c= 43;
c++;//c = c + 1 after evaluation c will be 44
c--;//c=c-1 after evaluation c will return to 43
```

Relational equality and operators

The relational binary operators < (less than), > (greater than), <= (less than or equal), and >= (greater than or equal) operators return a value of 1 if the result of the operation is true, 0 if false. The equality operators == (equals) and != (not equals) are similar to relational operators however their precedence is lower.

```
int u=3,w=4;
u>w; //evaluates as false
u<=w; //evaluates as true
u==w; //evaluates as false
```

Logical Operators

The logical operators are && (and), and || (or). The table below describes the operation of && and ||

Right Operand	Left Operand	Right Operand && Left Operand	Right Operand || Left Operand
TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE
FALSE	TRUE	FALSE	TRUE
FALSE	FALSE	FALSE	FALSE

The && operator has higher precedence than the || operator.

```
int j=1,k=2;
j<k || j==k;//evaluated as true
j>k && j!=k;//evaluated as false
j<k && k>j; //evaluated as true
```

Shift operators

Shift functions to move to the shift left or right for each binary digit of a particular variable. The << operator shift binary representation of a digit to the left while the >> operator shift it to the right. For example

```
int m = 2;
m=m>>1; //000000010 in binary, after shift right once -> 00000001
m=2
m=m<<1; // 00000010 in binary, after shift left once -> 00000100
```

Operator precedence

Category	Operator	Associativity
Inc/Dec	++ --	Left to right
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right

Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %=>>= <<= &= ^= |=	Right to left
Comma	,	Left to right

Normally, precedence would be the priority which operation would be evaluated first, however when the operators are of the same precedence the associativity (which operand comes first from either left or right) would be prioritized. For example:

```
int f=3, g=9;

f = f*g+2/1-g%2;
/*
f = (f*g)+2/1-g%2 // f*g evaluated result: 27
f = 27+(2/1)-g%2 // 2/1 evaluated result: 1
f = 27+2-(g%2) // g%2 evaluated result: 1
f=(27+2)-1 // 27+2 evaluated result: 29
f=(29-1) //29-1 evaluated result: 28
f=28
*/
```

When evaluating mixed datatypes for instance float and int, the int would implicitly be converted to float. For example:

```
int h=1;

float i=5.5, i_result=0;
i_result = h*5 + i;

/*
i_result = (h*5)+ i //h*5 evaluated result 5
i_result=(5*i) //5*i evaluated result 10.5
i_result=10.5
*/
```

Some math procedures

Name	Description of Computation	Argument	Return value
abs	The absolute value of the argument	double/integer	same as argument
exp	The value of e(2.71828) raised to the power of the argument	double/integer	double
log	The logarithm (to the base e) of the argument	double/integer	double
sqrt	The positive square root of the argument	double/integer (positive)	double
pow	pow(x, y) raises x to y	double	double
atan	The arc tangent of the argument	double/integer	double (radians)
cos	The cosine of the argument	double/integer	double (radians)
sin	The sine of the argument	double/integer	double (radians)

To use the math operators, you need to include the <math.h> library on the header. The argument of the function would now act as the input of and the return value as the result. For example to translate the equation below into a C code.

$$\mathrm{x} = 2\sin\frac{1}{2}(w+b)\cos\frac{1}{2}(w-b)$$

Given that w = 5, b=10 and are in radians;

```
#include <math.h>
#include <stdio.h>
int main()
{
      float x=0, w=5,b=10;
      x = 2*sin((w+b)/2)*cos((w-b)/2);
      printf ("Result: %f",x);
}
```