

Penetration Testing Student

Programming in C++

Section 02 | Module 02

© Caendra Inc. 2019
All Rights Reserved

Table of Contents

Module 02 | Programming in C++

2.1	C++ IDE	2.6	Iteration & Conditional Structures
2.2	Structure of C++ Programs	2.7	Pointers
2.3	Variables & Types	2.8	Arrays
2.4	Input / Output	2.9	Functions
2.5	Operators	2.10	Hera Lab: C++-assisted exploitation



Learning Objectives

By the end of this module, you should have a better understanding of:

- C++ IDE
- C++ fundamentals: Variables, functions and basic code constructs



2.1

C++ IDE



2.1 C++ IDE

Welcome to the Programming Section! Every Penetration Tester should have basic programming skills.

In this module, we will cover basic concepts that will help you write code in C++.



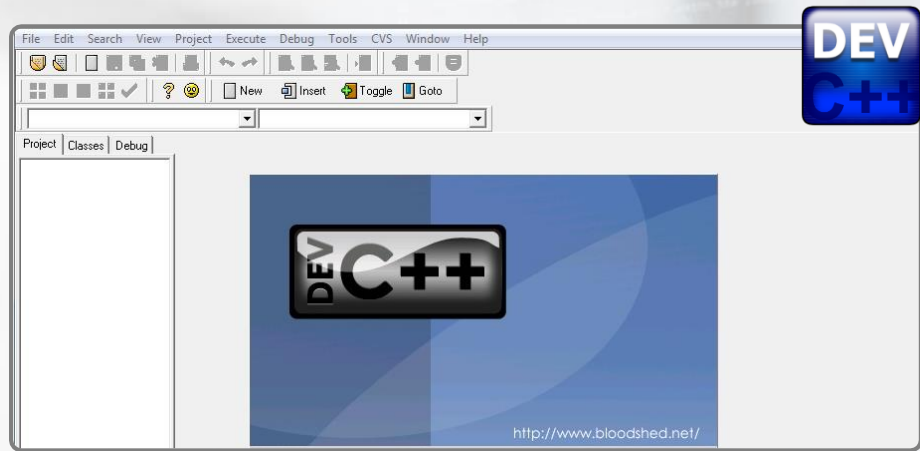
You can find all the C++ code samples used on the **Resources** drop-down menu of this module.

2.1 C++ IDE

Let's start downloading and installing an IDE (Integrated Development Environment) for C++.

We can download Dev-C++ at the following link:

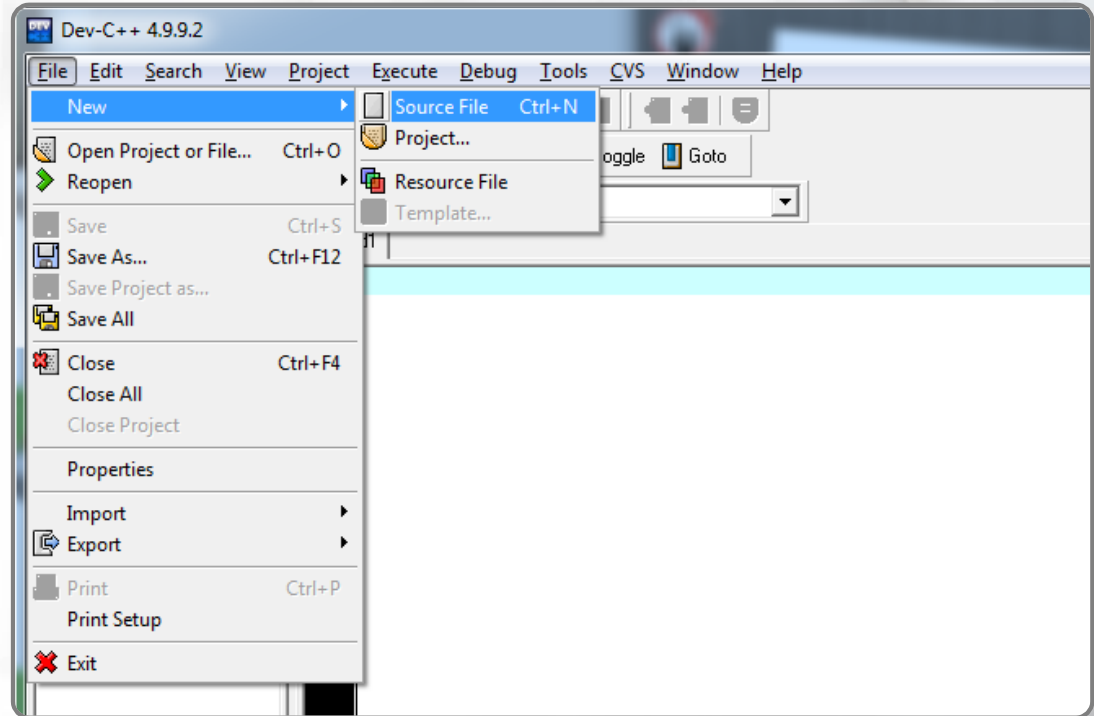
<http://sourceforge.net/projects/orwelldevcpp/>



2.1 C++ IDE

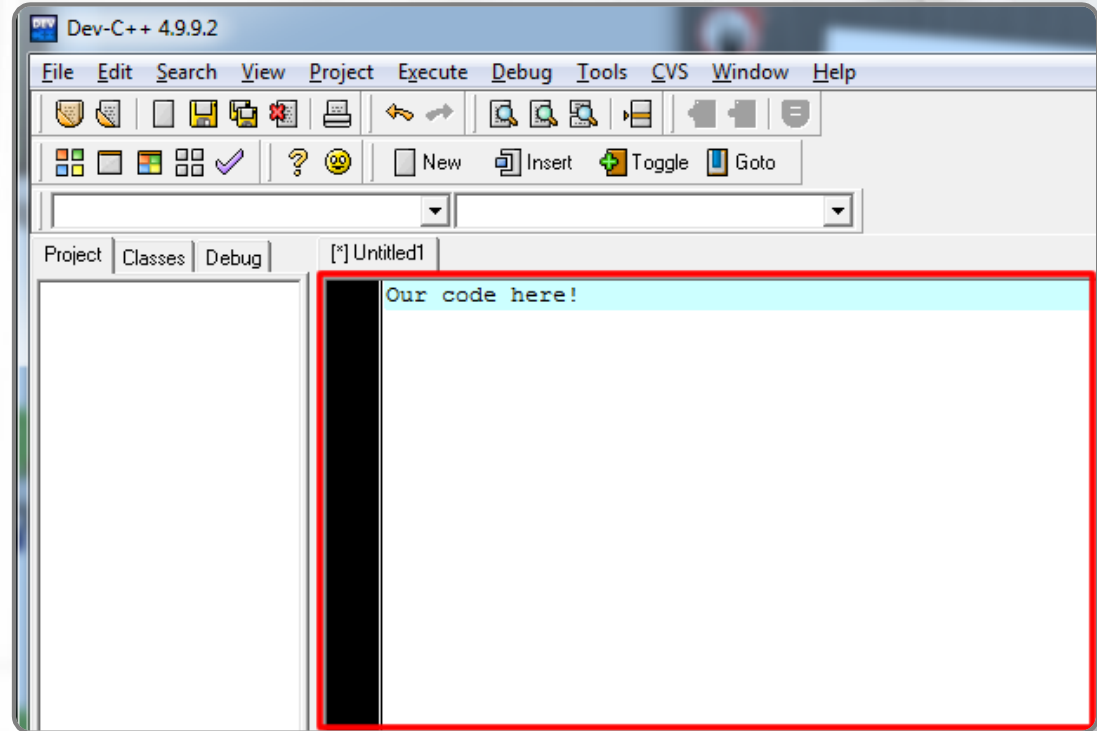
The first step is to create a new file where we will insert our source code.

To do that, click on *File>New>Source File*.



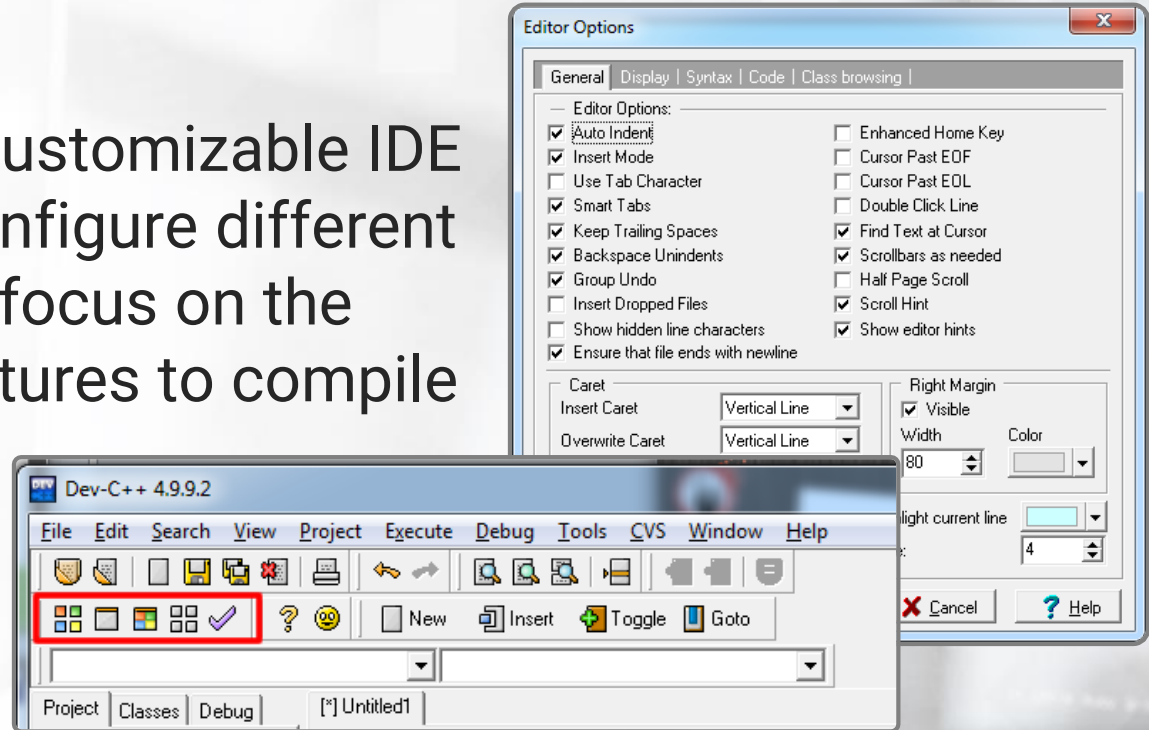
2.1 C++ IDE

Here we see the main panel where we are going to write our source code.

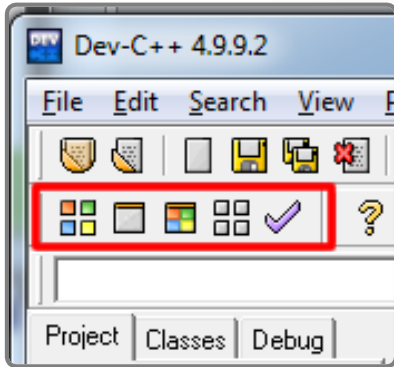






2.1 C++ IDE

Dev-C++ is a highly customizable IDE and allows us to configure different settings. Let's now focus on the most important features to compile our first program.



2.1 C++ IDE



-  Compiles the source code. If the source code compiles successfully, a .exe file will be created.
-  Allows you to run the program you just compiled.
-  Merges the previous two commands; it first compiles your code and then runs it.
-  This button allows you to debug your source code.

Structure of C++ Programs



2.2 Structure of C++ Programs

In this chapter, you will see what a C++ program looks like.

Furthermore, you will learn how to write, compile and execute your first program: **Hello World!**

```
1 // ...
2 #include <bits/stdc++.h>
3 using namespace std;
4
5 // ...
6
7 // ...
8
9 // ...
10
11 // ...
12
13 // ...
14
15 // ...
16
17 // ...
18
19 // ...
20
21 // ...
22
23 // ...
24
25 // ...
26
27 // ...
28
29 // ...
30
31 // ...
32
33 // ...
34
35 // ...
36
37 // ...
38
39 // ...
40
41 // ...
42
43 // ...
44
45 // ...
46
47 // ...
48
49 // ...
50
51 // ...
52
53 // ...
54
55 // ...
56
57 // ...
58
59 // ...
60
61 // ...
62
63 // ...
64
65 // ...
66
67 // ...
68
69 // ...
70
71 // ...
72
73 // ...
74
75 // ...
76
77 // ...
78
79 // ...
80
81 // ...
82
83 // ...
84
85 // ...
86
87 // ...
88
89 // ...
90
91 // ...
92
93 // ...
94
95 // ...
96
97 // ...
98
99 // ...
100
```



2.2 Structure of C++ Programs

Here we see the code of *Hello World*, a super simple program that we will compile to test the configuration of DevC++.

```
  
// This is my first Program in C++  
#include <iostream>  
using namespace std;  
int main ()  
{  
    cout << "Hello World!";  
    return 0;  
}
```

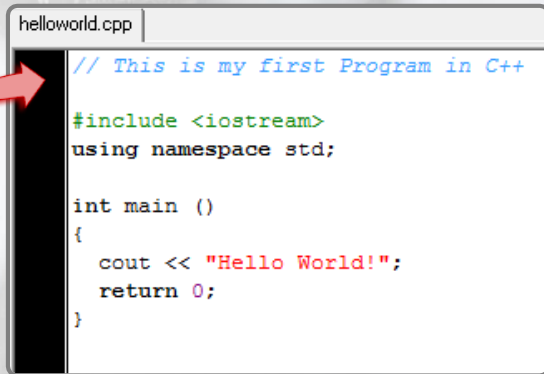
Now, let's go study it line by line.

2.2 Structure of C++ Programs



```
// This is my first program in C++
```

The first line is a **comment**. All the lines starting with double slashes (//) are considered comments and do not have any effect on the program. The compiler will ignore them.

A code editor window titled 'helloworld.cpp' showing C++ code. A red arrow points to the first line of the code, which is a comment. The code includes the <iostream> header, uses the std namespace, and defines a main function that prints 'Hello World!' and returns 0.


```
helloworld.cpp  
// This is my first Program in C++  
  
#include <iostream>  
using namespace std;  
  
int main ()  
{  
    cout << "Hello World!";  
    return 0;  
}
```

2.2 Structure of C++ Programs



```
#include <iostream>
```

All lines starting with the hash (#) character are **directives**. In this example, it instructs the compiler to include the code of the *iostream* library in our program. The *iostream* library provides input and output functionalities. A **library** is a collection of routines that a program can use.



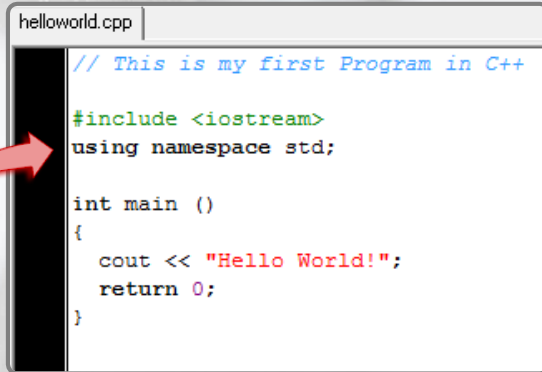
```
helloworld.cpp  
// This is my first Program in C++  
#include <iostream>  
using namespace std;  
  
int main ()  
{  
    cout << "Hello World!";  
    return 0;  
}
```


2.2 Structure of C++ Programs



Using namespace std;

Namespaces are used to group a set of classes, functions etc. under a name. Since all the elements in the standard C++ library (such as *iostream*) are declared within the **std namespace**, we need this command to access its functionalities.

A code editor window titled 'helloworld.cpp' showing C++ code. A red arrow points from the 'using namespace std;' line in the text above to the same line in the code editor.

```
helloworld.cpp
// This is my first Program in C++

#include <iostream>
using namespace std;


int main ()
{
    cout << "Hello World!";
    return 0;
}
```

2.2 Structure of C++ Programs



Using namespace std;

Please note the semicolon (;) at the end of the command; this is part of the syntax, and it is called a **terminator**. It tells the compiler that it has reached the end of a command.



```
helloworld.cpp
// This is my first Program in C++

#include <iostream>
using namespace std;


int main ()
{
    cout << "Hello World!";
    return 0;
}
```

2.2 Structure of C++ Programs



```
int main ()
```

Here is the declaration of the **main function** of our program. The **main** function is **where our program execution starts**. In other words, wherever the main function is declared in our source code, it will be the first code to be executed.




```
helloworld.cpp  
// This is my first Program in C++  
  
#include <iostream>  
using namespace std;  
  
int main ()  
{  
    cout << "Hello World!";  
    return 0;  
}
```

2.2 Structure of C++ Programs

```
</>
{
    instructions here
}
```

The two curly brackets '**{ }**' contain the **body** of the main function. The brackets determine where the main function code starts and ends.



```
helloworld.cpp
// This is my first Program in C++

#include <iostream>
using namespace std;


int main ()
{
    cout << "Hello World!";
    return 0;
}
```

2.2 Structure of C++ Programs



```
cout << "Hello World!";
```

cout is the name of the standard output. Most of the time, the standard output is the console. The **cout <<** statement tells the compiler to put a sequence of characters, 'Hello World!' in our example, onto the standard output stream (the console). In other words, it prints the string 'Hello World!' on the screen.




```
helloworld.cpp  
  
// This is my first Program in C++  
  
#include <iostream>  
using namespace std;  
  
int main ()  
{  
    cout << "Hello World!";  
    return 0;  
}
```

2.2 Structure of C++ Programs



```
return 0;
```

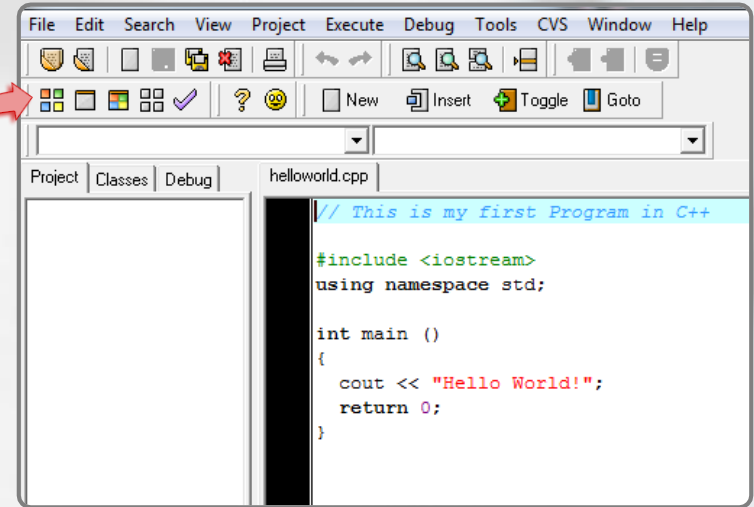
This last statement causes the main function to end. As we will see later in the function section, the **return** statement can have different values. In our case the value is '0', and it means that the program has completed its execution without any errors.



```
helloworld.cpp  
  
// This is my first Program in C++  
  
#include <iostream>  
using namespace std;  
  
int main ()  
{  
    cout << "Hello World!";  
    return 0;  
}
```

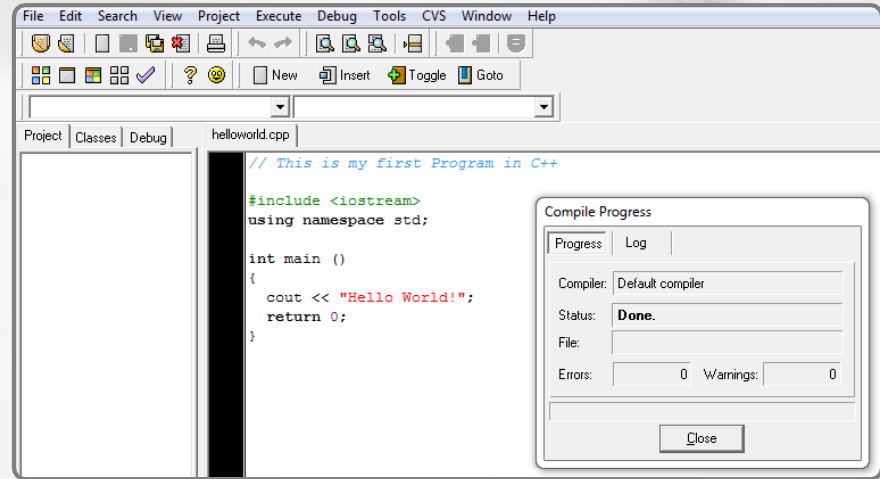
2.2 Structure of C++ Programs

Now that we have written our first program let's try to **compile** it and see what happens. To do that, we will use the button (🗑️) shown before. So now we can start!



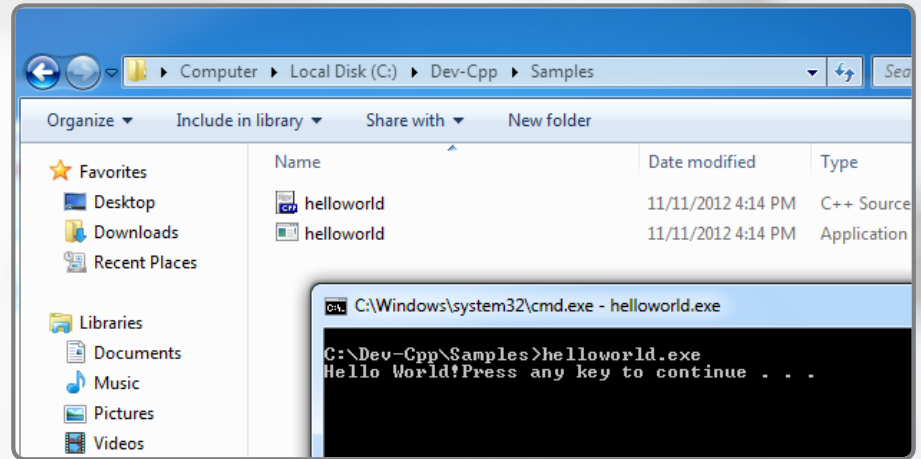
2.2 Structure of C++ Programs

If the program is successfully compiled we should see a window like the following; this window tells us that the compiler is done and that no errors or warnings were generated.



2.2 Structure of C++ Programs

A new file named “helloworld.exe” has been created in the same directory of our source code. Now, we can open a CMD prompt and run it!



2.2 Structure of C++ Programs


We can also use the Dev-C++ button to run our compiled program.

Please note that a program terminates as soon as it completes its operations; this means that running our program will open a terminal window for less than a second.



2.2 Structure of C++ Programs

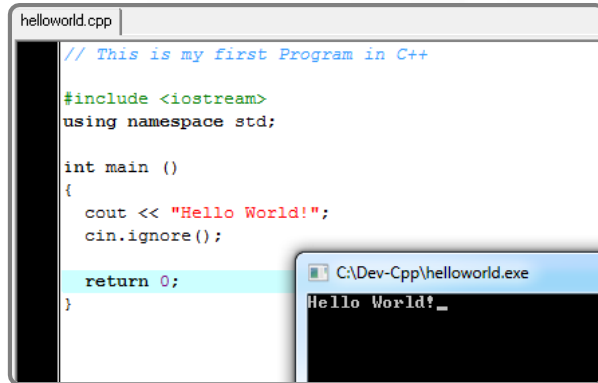
In order to avoid the console closing automatically, we can use different commands.

For example, insert one of the two following commands right before the “**return 0;**” statement and then compile and run () the program within Dev-C++:

- `system("PAUSE");`
- `cin.ignore();`

2.2 Structure of C++ Programs

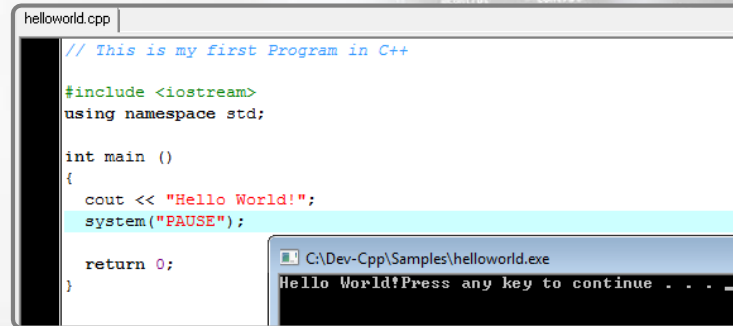
You should now be able to run the program and see the console output.



The screenshot shows a code editor window titled 'helloworld.cpp' with the following C++ code:

```
// This is my first Program in C++  
  
#include <iostream>  
using namespace std;  
  
int main ()  
{  
    cout << "Hello World!";  
    cin.ignore();  
  
    return 0;  
}
```

Below the code editor, a console window titled 'C:\Dev-Cpp\helloworld.exe' displays the output: 'Hello World!_'.



The screenshot shows a code editor window titled 'helloworld.cpp' with the following C++ code:

```
// This is my first Program in C++  
  
#include <iostream>  
using namespace std;  
  
int main ()  
{  
    cout << "Hello World!";  
    system("PAUSE");  
  
    return 0;  
}
```

Below the code editor, a console window titled 'C:\Dev-Cpp\Samples\helloworld.exe' displays the output: 'Hello World!Press any key to continue . . . _'.

Variables & Types



2.3 Variables & Types

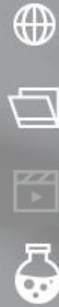
Now that we are more confident about the IDE, let's see how we can define variables with different types.



2.3 Variables & Types

The “*Hello World*” example was very simple.

Variables are portions of memory where values are stored. Each variable is recognizable by a human (the programmer) through a symbolic name (or identifier). In other words, this identifier is the way we can reference the stored value.



2.3 Variables & Types

Since we are going to store these values in the computer's memory, we have to specify the type of data we are going to store in it. For this reason, when we declare a new variable, we have to define its **type**.

Now, let's look at some examples to clarify these concepts.

2.3 Variables & Types

variables_types.cpp

```
#include <iostream>
using namespace std;

int main ()
{
    // Declaration of variables
    int a = 0;
    int b = 2;
    int sum = a + b;

    cout << "The value of variable sum is: " << sum << endl;

    a = 3;

    sum = a + b;

    cout << "The value of variable sum is: " << sum << endl;

    cin.ignore();
    return 0;
}
```

In this example, we declared three variables (**a**, **b**, **sum**) and then we changed their values. The program prints the sum of the variables **a** and **b**.

The first time we print the value of **sum** (through *cout*), it is 2, while the second time it is 5. This happens because we change the value of the variable **a** during the execution of the program.

2.3 Variables & Types

```
variables_types.cpp
#include <iostream>
using namespace std;

int main ()
{
    // Declaration of variables
    int a = 0;
    int b = 2;
    int sum = a + b;

    cout << "The value of variable sum is: " << sum << endl;

    a = 3;

    sum = a + b;

    cout << "The value of variable sum is: " << sum << endl;

    cin.ignore();
    return 0;
}
```

Here we declare the three variables, and we assign a value to each of them.

As you can see in the code, each variable has its type, which is an integer (*int*) in our case. We can also declare and assign the value in the same line.

2.3 Variables & Types

variables_types.cpp

```
#include <iostream>
using namespace std;

int main ()
{
    // Declaration of variables
    int a = 0;
    int b = 2;
    int sum = a + b;

    cout << "The value of variable sum is: " << sum << endl;

    a = 3;

    sum = a + b;

    cout << "The value of variable sum is: " << sum << endl;

    cin.ignore();
    return 0;
}
```

This line prints the value of the sum variable. In this case, it prints the following string:

"The value of variable sum is: 2"

2.3 Variables & Types

variables_types.cpp

```
#include <iostream>
using namespace std;

int main ()
{
    // Declaration of variables
    int a = 0;
    int b = 2;
    int sum = a + b;

    cout << "The value of variable sum is: " << sum << endl;

    a = 3;

    sum = a + b;

    cout << "The value of variable sum is: " << sum << endl;

    cin.ignore();
    return 0;
}
```

Here we assign a new value to the variable **a**.

The previous value (0) is overwritten with the new one (3).

2.3 Variables & Types

variables_types.cpp

```
#include <iostream>
using namespace std;

int main ()
{
    // Declaration of variables
    int a = 0;
    int b = 2;
    int sum = a + b;

    cout << "The value of variable sum is: " << sum << endl;

    a = 3;
    sum = a + b;

    cout << "The value of variable sum is: " << sum << endl;

    cin.ignore();
    return 0;
}
```

We assign a new value to the variable **sum**.

Since the value of the variable **a** is changed, the value of the sum is now 5 (3+2).

2.3 Variables & Types

There are many different data types we can use.

Here is a short list.

short short int	Short integer (2 bytes)
int	Integer (4 bytes)
long long int	Long integer (4 bytes)
bool	Boolean (1 byte)
float	Floating point number (4 bytes)
double	Double precision floating point number (8 bytes)
char	Character (1 byte)

2.3 Variables & Types

Before we see the **Iteration and Conditional Structures** section, there is one last thing you need to know!

Each variable we are going to use must be declared somewhere in the source code. Depending on the position where it is declared it has a different scope: global or local. We'll now explain this concept with an example.

2.3 Variables & Types

```
global.cpp
#include <iostream>
using namespace std;

// Declaration of global variables
int global_variable = 4;

int main ()
{
    cout << "Value of global_variable: " << global_variable << endl;

    // Declaration of local variables

    int global_variable = 2;

    cout << "Value of global_variable: " << global_variable << endl;

    cin.ignore();
    return 0;
}
```

A **global** variable is declared in the body of the source code (it is not in a function) and can be referred from anywhere.

In the first instruction in the **main** function, we can print the value of the variable named **global_variable** and the output is "Value of *global_variable*: 4".

2.3 Variables & Types

```
global.cpp
#include <iostream>
using namespace std;

// Declaration of global variables
int global_variable = 4;

int main ()
{
    cout << "Value of global_variable: " << global_variable << endl;

    // Declaration of local variables
    int global_variable = 2;

    cout << "Value of global_variable: " << global_variable << endl;

    cin.ignore();
    return 0;
}
```

Local variables are variables declared inside a function body or block enclosed in curly brackets “{}” (main function in our case) and their scope is limited to the block where they are declared.

In this case, the second `cout` prints the following string: *Value of global_variable: 2*

2.3 Variables & Types

```
global.cpp
#include <iostream>
using namespace std;

// Declaration of global variables
int global_variable = 4;

int main ()
{
    cout << "Value of global_variable: " << global_variable << endl;

    // Declaration of local variables
    int global_variable = 2;

    cout << "Value of global_variable: " << global_variable << endl;

    cin.ignore();
    return 0;
}
```

Global scope

Local Scope

So the scope of the **global_variable** is different.

If we use **global_variable** in another function, its value will be 4, no matter if we have already executed the instruction *"int global_variable = 2;"*

Input / Output



2.4 Input / Output

In the previous examples, we used a function that allows us to print a message on the screen. We can expand it and see how we can interact with the user and get his input from the keyboard.

Do you remember this line of code?



```
cout << "The value of variable sum is: " << sum << endl;
```

Let's split it and analyze each part.

2.4 Input / Output

The **cout** statement represents the standard output.

Since our default output is the console, **cout** tells our program to print the following code to the console:



```
cout << "The value of variable sum is: " << sum << endl;
```


2.4 Input / Output

The **<<** operator tells the program to insert the next data into the stream. Since we use **cout**, it puts the data in the standard output stream. This operator can be used multiple times, and it is useful when we want to print a combination of strings and variables. In this case, we want the string **“The value of variable sum is:”** followed by the value of the variable **sum**.



```
cout << "The value of variable sum is: " << sum << endl;
```

2.4 Input / Output

endl inserts a '*new line character*' and flushes the buffer; this ensures that the next output prints in the next line.



```
cout << "The value of variable sum is: " << sum << endl;
```



2.4 Input / Output

Similarly, we can use the **cin** function followed by the operator **>>** in order to get the user's input.

In this case, the standard input is the keyboard.



2.4 Input / Output

In this example, we first declare the variable where the user input is stored (**uservalue**), and then we print some messages to the user.

```
inputoutput.cpp
#include <iostream>
using namespace std;

int main()
{
    int uservalue;

    cout << "This program adds 10 to your input." << endl;
    cout << "Please insert your input |";

    cin >> uservalue;

    cout << "The value inserted is " << uservalue;
    cout << " and the new value is " << uservalue + 10 << endl;

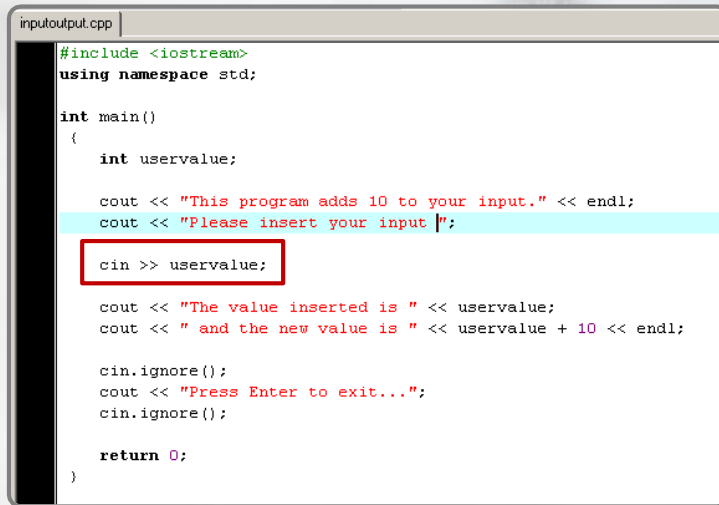
    cin.ignore();
    cout << "Press Enter to exit...";
    cin.ignore();

    return 0;
}
```

2.4 Input / Output

With the **cin >> uservalue;** statement, we instruct the program to get the input from the standard input (the keyboard) and save it in the **uservalue** variable.

As you can see in the next two lines, we first print the value provided by the user and then we print out the value plus 10.



```
inputoutput.cpp
#include <iostream>
using namespace std;

int main()
{
    int uservalue;

    cout << "This program adds 10 to your input." << endl;
    cout << "Please insert your input |";

    cin >> uservalue;

    cout << "The value inserted is " << uservalue;
    cout << " and the new value is " << uservalue + 10 << endl;

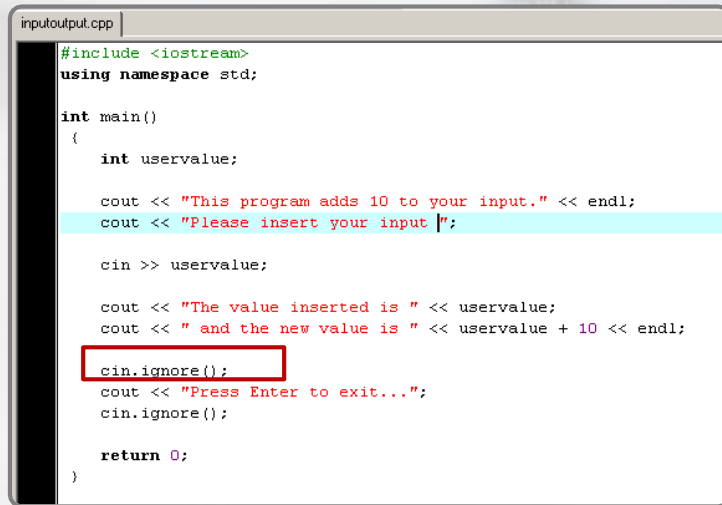
    cin.ignore();
    cout << "Press Enter to exit...";
    cin.ignore();

    return 0;
}
```

2.4 Input / Output

When the user inserts their value and presses enter (return) on the keyboard, the return value goes into the buffer.

When we run the program, in order to keep the console opened, we need to clean this buffer with the first **cin.ignore()** statement and then with the second **cin.ignore()** we can prevent the console from closing.

A screenshot of a code editor window titled 'inputoutput.cpp'. The code is in C++ and demonstrates how to handle input and keep the console open. The code includes the <iostream> header and uses the std namespace. The main function declares an integer 'uservalue'. It then prints a message and prompts the user to insert input. The user's input is stored in 'uservalue'. The program then prints the value and the value plus 10. A red box highlights the first 'cin.ignore()' statement, which is used to clear the input buffer. The program then prints a message to press enter to exit and calls 'cin.ignore()' again to prevent the console from closing. The code is as follows:

```
inputoutput.cpp
#include <iostream>
using namespace std;

int main()
{
    int uservalue;

    cout << "This program adds 10 to your input." << endl;
    cout << "Please insert your input |";

    cin >> uservalue;

    cout << "The value inserted is " << uservalue;
    cout << " and the new value is " << uservalue + 10 << endl;
    cin.ignore();
    cout << "Press Enter to exit...";
    cin.ignore();

    return 0;
}
```

2.4 Input / Output

In other words, the first **cin.ignore()** reads the return value and the second waits for a new input, keeping the console on the screen.

```
inputoutput.cpp
#include <iostream>
using namespace std;

int main()
{
    int uservalue;

    cout << "This program adds 10 to your input." << endl;
    cout << "Please insert your input |";

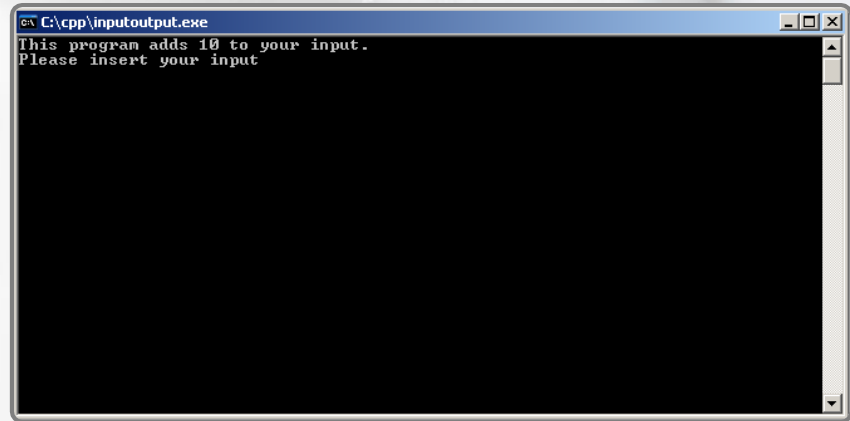
    cin >> uservalue;

    cout << "The value inserted is " << uservalue;
    cout << " and the new value is " << uservalue + 10 << endl;
    cin.ignore();
    cout << "Press Enter to exit...";
    cin.ignore();

    return 0;
}
```

2.4 Input / Output

Here we can see what our program looks like.



```
C:\cpp\inputoutput.exe
This program adds 10 to your input.
Please insert your input
```


Operators



2.5 Operators

In C++ there are four main classes of operators:

Arithmetic

Relational

Logical

Bitwise

2.5 Operators



```
variable_name = expression;
```

The assignment operator can be used within any valid expression, and we can see the general form above.

The target (the left part – variable name) of the assignment must be a variable or a pointer (we will see later what pointers are) and can't be a function or a constant.



2.5 Operators




```
variable_name = expression;
```

In C++ literature, you will see these two terms: **lvalue** and **rvalue**.

- **lvalue** is any label that appears on the left side of an assignment statement; in other words, we can say it is the variable name.
- **rvalue** refers to expressions/value on the right side of an assignment and simply means the value that will be assigned to the variable.

2.5 Operators

In the following example, the first statement assigns the value 10 to the variable named **variable1**.

```
  
variable1 = 10;  
  
b = variable1;
```

The second statement assigns the value contained in **variable1** to variable **b**. At this point, the value of variable **b** will be 10.

2.5 Operators

The following table summarizes the arithmetical operators.

Operator	Action
-	Subtraction
+	Addition
*	Multiplication
/	Division
%	Modulus

2.5 Operators

`a = a + 1;`

is the same as

`++a;`

`a = a - 1;`

is the same as

`--a;`

In addition to the previous operators, C++ includes an **increment** operator (`++`) and a **decrement** operator (`--`) where:

- **++** adds 1 to its operand
- **--** subtracts 1 from its operand

2.5 Operators

```
x = 10;  
y = ++x;
```

Set **y** to 11

```
x = 10;  
y = x++;
```

Set **y** to 10 and **x** to 11

Both the increment and decrement operators may precede (**++x**) or follow (**x++**) the operand. The difference between them is that when the operator precedes its operand, the increment or decrement operation is performed before obtaining the value of the operand, while if the operator follows its operand, the value is obtained before incrementing or decrementing it.

2.5 Operators

Operator	Action
>	Greater than
>=	Greater than or equal
<	Less than
<=	Less than or equal
==	Equal
!=	Not equal

Relational

Logical

Operator	Action
&&	AND
	OR
!	NOT

Relational operators define a relationship between two values.

Logical operators define how previous relationships must be connected.

2.5 Operators

The idea of true and false is the basic concept of relational and logical operators. In C++, true is any value other than zero. False is zero. Expressions that use relational or logical operators return **0** for *false* and **1** for *true*.

For this purpose, in C++ we can use the **bool** data type and the Boolean constants **true** and **false**. So a 0 value automatically converts to false while a non-zero value automatically converts to true.

2.5 Operators

Relational operators are used in order to evaluate a comparison between two expressions. The result is a *Boolean* value.

Operation	Value
(10 > 1)	True
(10 >= 10)	True
(10 < 5)	False
(5 <= 10)	True
(1 == 1)	True
(1 != 1)	False

2.5 Operators

Logical operators define how previous relationships must be connected.

x	y	x && y	x y	!x
0	0	0	0	1
0	1	0	1	1
1	1	1	1	0
1	0	0	1	0

The first line may be read as the following: when **x** is false (0) and **y** is false (0) the result of **x AND y** (**x && y**) is false (0), the result of **x OR y** is false, and the **inverse** of **x** is true

2.5 Operators

The logical operator **!** has only one operand (at its right) and it inverses this value (false if its operand is true, and true if its operand is false). The logical operator **&&** and **||** evaluate two expressions in order to obtain a relational result.

- **&&** (AND) results true if both operands are true and false otherwise
- **||** (OR) results true if either one of its operands is true, false when both are false

2.5 Operators

C++ supports many operations that can be done in assembler, including operations on bits.

Bitwise operations refer to testing, setting or shifting the actual bits in a byte or word.

Operator	Action
&	AND
	OR
^	Exclusive OR (XOR)
~	One's complement (NOT)
>>	Shift right
<<	Shift left

2.5 Operators

The following program executes a bitwise AND, a bitwise OR and then shifts the value of x.

```
#include <iostream>
using namespace std;

int main ()
{
    int x = 206;
    int y = 152;


    int z = x & y;
    cout << "Bitwise AND: " << z << "\n";

    z = x | y;
    cout << "Bitwise OR: " << z << "\n";

    x = x << 1;

    cout << "Left shift 1 bit: " << x << "\n";

    cin.ignore();
    return 0;
}
```



```
C:\Dev-Cpp\Samples\bitwise.exe
Bitwise AND: 136
Bitwise OR: 222
Left shift 1 bit: 412
```

Bitwise AND

11001110	&	206	&
10011000	=	152	=
10001000		136	

Iteration & Conditional Structures



2.6 Iteration & Conditional Structures

Let's see how we can define control structures.

These structures are useful to instruct the program to execute or to repeat a specific operation when some condition is matched.

```
23 # The experiment will result in 2 observations
24 # observations - an array of Observations, in this case
25 # control - the control observation
26
27
28 def initialize(experiment, observations = [], control = nil)
29   @experiment = experiment
30   @observations = observations
31   @control = control
32   freeze
33 end
34
35 # Returns the name of the experiment
36 def experiment_name
37   @experiment.name
38 end
39
40 # Returns the result a match between an experiment
41 def match?
42   @experiment.result == 1
43 end
```



2.6 Iteration & Conditional Structures

A statement is part of our program that can be executed and specifies an action. In this section, we are going to see three main groups of statements.

SELECTION

- if
- switch

ITERATION

- while
- for
- do-while

JUMP

- break
- continue
- goto
- return

2.6 Iteration & Conditional Structures

The general form of the **if** statement is:

```
</>  
if (expression)  
    statement;  
else  
    statement;
```

Where a statement may consist of a single statement, a block of statements (but they **must** be enclosed in curly brackets), or nothing (in case of an empty statement). The **else** clause is optional.

2.6 Iteration & Conditional Structures

```
</>  
if (expression)  
    statement;  
else  
    statement;
```

If the *expression* evaluates to true, the statement or block of statements that form the target of

of **if** is executed; otherwise, the statement or block that is the target of **else** will be executed.

Only the code associated with **if** or the code associated with **else** executes, never both.

2.6 Iteration & Conditional Structures

```
ifelse.cpp
#include <iostream>
using namespace std;

int main ()
{
    int user_value;
    cout << "Insert a number\n";
    cin >> user_value;
    cin.ignore();
    if(user_value < 10)
    {
        cout << "The value is less than 10";
    }else{
        cout << "The value is greater than 10";
    }

    cin.ignore();
    return 0;
}
```

```
C:\cpp\ifelse.exe
Insert a number
5
The value is less than 10
```

```
C:\cpp\ifelse.exe
Insert a number
15
The value is greater than 10_
```

The above program contains an example of an if - else statement. The program simply checks if the number provided is greater or less than 10. Depending on the value, the program will print different messages.

2.6 Iteration & Conditional Structures

A **nested if** is an if contained in the body of a parent if. In a nested if, the else statement refers to the nearest if statement in the same block that is not already associated with another else.



```
if(x)
{
    if(a) statement 1;
    if(b) statement 2; /* this if is associated */
    else statement 3; /* with this else */
}
else statement 4; /* associated with if(x) */
```

2.6 Iteration & Conditional Structures

C++ implements a multiple-branch selection statement, called **switch**, which tests, in order, the value of an expression against a list of values.

Only the block of operations associated with the matching expression is executed. Note that the value must be an integer or a constant.

```
</>  
switch (expression){  
    case constant1:  
        statement sequence  
        break;  
    case constant2:  
        statement sequence  
        break;  
    .  
    :  
    .  
    default  
        statement sequence  
}
```


2.6 Iteration & Conditional Structures

The value of the expression is sequentially tested against the values specified in the **case** statements. When a match is found, the statement block associated is executed until the break statement, or the end of the switch is reached. The **default** statement is executed if no matches are found.

Note that **default** is optional, so if it is not defined, there is no action if all matches fail.



2.6 Iteration & Conditional Structures

The **break** statement is one of C++'s jump statements. You can use it in loops as well as in the **switch** statement.

When break is encountered in a switch, the execution “jumps” to the line of code following the switch statement.

```
</>  
switch (expression) {  
    case constant1:  
        statement sequence  
        break;  
    case constant2:  
        statement sequence  
        break;  
    .  
    .  
    .  
    default  
        statement sequence  
}
```

2.6 Iteration & Conditional Structures

The following example shows a menu selection.

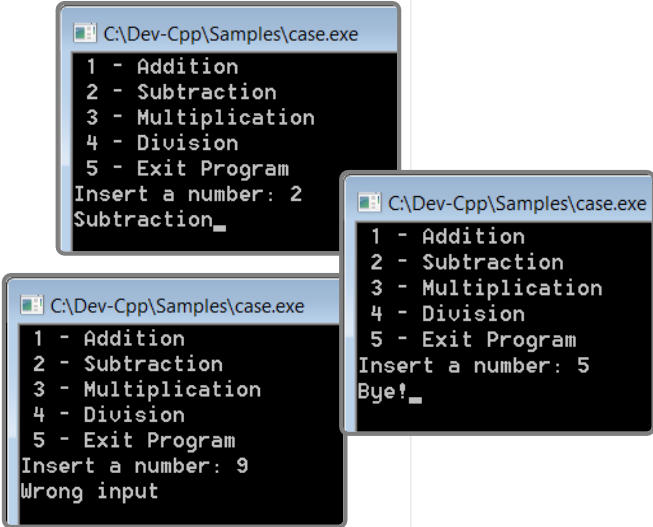
This simple program displays a menu, gets the user input and calls the proper procedures.

```
#include <iostream>
using namespace std;

int main ()
{
    int user_value;
    cout << " 1 - Addition\n 2 - Subtraction\n 3 - Multiplication\n 4 - Division \n 5 - Exit Program\n";
    cout << "Insert a number: ";
    cin >> user_value;
    cin.ignore();

    switch (user_value){
        case 1:
            cout << "Addition";
            break;
        case 2:
            cout << "Subtraction";
            break;
        case 3:
            cout << "Multiplication";
            break;
        case 4:
            cout << "Division";
            break;
        case 5:
            cout << "Bye!";
            break;
        default:
            cout << "Wrong input";
    }

    cin.ignore();
    return 0;
}
```



The image shows three overlapping screenshots of a Windows command prompt window titled "C:\Dev-Cpp\Samples\case.exe". Each screenshot displays the program's menu and the result of a user input. The first screenshot shows the menu and the input "2", resulting in "Subtraction_". The second screenshot shows the menu and the input "9", resulting in "Wrong input". The third screenshot shows the menu and the input "5", resulting in "Bye!".

2.6 Iteration & Conditional Structures

ITERATION

Iteration statements, also called **loops**, allow a set of instructions to be executed repeatedly for a fixed number of times or until a certain condition is reached.

While in for loops the condition is predefined, in do-while loops are open-ended.

2.6 Iteration & Conditional Structures

The general form of a **for** statement is:



```
for(initialization;condition;increment) {  
    statement;  
}
```

Where:

- **initialization** is an assignment statement that sets the starting value of the loop control variable
- **condition** determines when the loop must end
- **increment** defines how the control variable changes for each iteration

2.6 Iteration & Conditional Structures

```
#include <iostream>
using namespace std;

int main ()
{
    int a;
    for (a = 2; a <= 50; a = a+2){
        cout << a << "-";
    }

    cin.ignore();
    return 0;
}
```

Start with $a = 2$

Until a is less or equal to 50

Increment a by 2 in each loop

C:\Dev-Cpp\Samples\for.exe

2-4-6-8-10-12-14-16-18-20-22-24-26-28-30-32-34-36-38-40-42-44-46-48-50-

The **for** loop continues as long as the condition is true. Once the condition fails, the program executes the statement right after the **for**. In the above program, the **for** loop is used to print all the multiples of 2 until 50 is reached.

2.6 Iteration & Conditional Structures

The **for** statement is also used for infinite loops.

Since *initialization*, *condition*, and *increment* of the for loop are not required, we can make an infinite loop by leaving them empty:

```
</>  
for( ; ; ){  
    statement;  
}
```

2.6 Iteration & Conditional Structures

When the conditional expression is left empty, it is processed as true. Note that the **for(;;)** construct can exit from an infinite loop through a **break** statement present anywhere in the body of the loop. The break statement causes the termination of the loop, and the program control resumes from the next instruction following the loop.

You may be wondering though, why one would want to use an infinite loop.

2.6 Iteration & Conditional Structures

```
#include <iostream>
using namespace std;

int main ()
{
    int user_value;
    for (;;) {
        cout << " 1 - Addition\n 2 - Exit Program\n";
        cout << "Insert a number: ";
        cin >> user_value;
        cin.ignore();

        if (user_value == 1) {
            cout << "Your addition source code here\n\n";
        } else if (user_value == 2) {
            cout << "Bye";
            break;
        } else {
            cout << "wrong data\n\n";
        }
    }

    cin.ignore();
    return 0;
}
```



```
C:\Dev-Cpp\Samples\infinite_loop.exe
1 - Addition
2 - Exit Program
Insert a number: 6
Wrong data

1 - Addition
2 - Exit Program
Insert a number: 1
Your addition source code here

1 - Addition
2 - Exit Program
Insert a number: 1
Your addition source code here

1 - Addition
2 - Exit Program
Insert a number: 2
Bye
```

As shown in the code above, an **infinite loop** used with a **break** statement can be useful to keep the console alive until the user chooses to exit. The program terminates only if the user inserts the number 2; otherwise, they will see the menu over and over again.

2.6 Iteration & Conditional Structures

In the same way as with other statements, for loops can be nested. Nested loops are very common in programming since they add power and flexibility to complex algorithms.

Let's take a look at an example of how nested loops can be used.

```
1 def experiment_results(experiment_name):
2     """Returns the results of an experiment"""
3     # Get the experiment results
4     results = get_experiment_results(experiment_name)
5     # Iterate over the results
6     for result in results:
7         # Iterate over the candidates
8         for candidate in result['candidates']:
9             # Evaluate the candidate
10            evaluate_candidate(candidate, result['observations'])
11
12 def freeze():
13     """Freeze the model"""
14     # Freeze the model
15     freeze_model()
16
17 # Main function
18 def main():
19     # Get the experiment name
20     experiment_name = get_experiment_name()
21     # Iterate over the experiment results
22     for result in experiment_results(experiment_name):
23         # Iterate over the candidates
24         for candidate in result['candidates']:
25             # Evaluate the candidate
26             evaluate_candidate(candidate, result['observations'])
27
28 if __name__ == '__main__':
29     main()
```

2.6 Iteration & Conditional Structures

```
#include <iostream>
using namespace std;

int main ()
{
    int base, height;
    cout << "Insert the base length: ";
    cin >> base;
    cout << "Insert the height length: ";
    cin >> height;
    cin.ignore();
    cout << endl;

    for(int i = 0; i < height; i++){
        for(int j = 0; j < base;j++){
            cout << " *";
        }
        cout << endl;
    }
    cin.ignore();
    return 0;
}
```

C:\Dev-Cpp\Samples\square.exe
Insert the base length: 10
Insert the height length: 3

```
* * * * *
* * * * *
* * * * *
```

C:\Dev-Cpp\Samples\square.exe
Insert the base length: 5
Insert the height length: 5

```
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
```

Given two numbers (base and height), let's say we want to draw a rectangle using the char '*'.

What we can do is use nested loops to iterate columns and rows, as we see here in the code on the left.

2.6 Iteration & Conditional Structures

The second loop available in C++ is the **while loop**. Its general form is:



```
while(condition) {  
    statement;  
}
```

Where:

- **statement** is either an empty statement, a single statement, or a block of statements
- **condition** may be any expression, and true is any non-zero value

```
20  
21  
22  
23  
24  
25  
26  
27  
28  
29  
30  
31  
32  
33  
34  
35  
36  
37  
38  
39  
40  
41  
42  
43  
44  
45  
46  
47  
48  
49  
50  
51  
52  
53  
54  
55  
56  
57  
58  
59  
60  
61  
62  
63  
64  
65  
66  
67  
68  
69  
70  
71  
72  
73  
74  
75  
76  
77  
78  
79  
80  
81  
82  
83  
84  
85  
86  
87  
88  
89  
90  
91  
92  
93  
94  
95  
96  
97  
98  
99  
100  
101  
102  
103  
104  
105  
106  
107  
108  
109  
110  
111  
112  
113  
114  
115  
116  
117  
118  
119  
120  
121  
122  
123  
124  
125  
126  
127  
128  
129  
130  
131  
132  
133  
134  
135  
136  
137  
138  
139  
140  
141  
142  
143  
144  
145  
146  
147  
148  
149  
150  
151  
152  
153  
154  
155  
156  
157  
158  
159  
160  
161  
162  
163  
164  
165  
166  
167  
168  
169  
170  
171  
172  
173  
174  
175  
176  
177  
178  
179  
180  
181  
182  
183  
184  
185  
186  
187  
188  
189  
190  
191  
192  
193  
194  
195  
196  
197  
198  
199  
200  
201  
202  
203  
204  
205  
206  
207  
208  
209  
210  
211  
212  
213  
214  
215  
216  
217  
218  
219  
220  
221  
222  
223  
224  
225  
226  
227  
228  
229  
230  
231  
232  
233  
234  
235  
236  
237  
238  
239  
240  
241  
242  
243  
244  
245  
246  
247  
248  
249  
250  
251  
252  
253  
254  
255  
256  
257  
258  
259  
260  
261  
262  
263  
264  
265  
266  
267  
268  
269  
270  
271  
272  
273  
274  
275  
276  
277  
278  
279  
280  
281  
282  
283  
284  
285  
286  
287  
288  
289  
290  
291  
292  
293  
294  
295  
296  
297  
298  
299  
300  
301  
302  
303  
304  
305  
306  
307  
308  
309  
310  
311  
312  
313  
314  
315  
316  
317  
318  
319  
320  
321  
322  
323  
324  
325  
326  
327  
328  
329  
330  
331  
332  
333  
334  
335  
336  
337  
338  
339  
340  
341  
342  
343  
344  
345  
346  
347  
348  
349  
350  
351  
352  
353  
354  
355  
356  
357  
358  
359  
360  
361  
362  
363  
364  
365  
366  
367  
368  
369  
370  
371  
372  
373  
374  
375  
376  
377  
378  
379  
380  
381  
382  
383  
384  
385  
386  
387  
388  
389  
390  
391  
392  
393  
394  
395  
396  
397  
398  
399  
400  
401  
402  
403  
404  
405  
406  
407  
408  
409  
410  
411  
412  
413  
414  
415  
416  
417  
418  
419  
420  
421  
422  
423  
424  
425  
426  
427  
428  
429  
430  
431  
432  
433  
434  
435  
436  
437  
438  
439  
440  
441  
442  
443  
444  
445  
446  
447  
448  
449  
450  
451  
452  
453  
454  
455  
456  
457  
458  
459  
460  
461  
462  
463  
464  
465  
466  
467  
468  
469  
470  
471  
472  
473  
474  
475  
476  
477  
478  
479  
480  
481  
482  
483  
484  
485  
486  
487  
488  
489  
490  
491  
492  
493  
494  
495  
496  
497  
498  
499  
500  
501  
502  
503  
504  
505  
506  
507  
508  
509  
510  
511  
512  
513  
514  
515  
516  
517  
518  
519  
520  
521  
522  
523  
524  
525  
526  
527  
528  
529  
530  
531  
532  
533  
534  
535  
536  
537  
538  
539  
540  
541  
542  
543  
544  
545  
546  
547  
548  
549  
550  
551  
552  
553  
554  
555  
556  
557  
558  
559  
560  
561  
562  
563  
564  
565  
566  
567  
568  
569  
570  
571  
572  
573  
574  
575  
576  
577  
578  
579  
580  
581  
582  
583  
584  
585  
586  
587  
588  
589  
590  
591  
592  
593  
594  
595  
596  
597  
598  
599  
600  
601  
602  
603  
604  
605  
606  
607  
608  
609  
610  
611  
612  
613  
614  
615  
616  
617  
618  
619  
620  
621  
622  
623  
624  
625  
626  
627  
628  
629  
630  
631  
632  
633  
634  
635  
636  
637  
638  
639  
640  
641  
642  
643  
644  
645  
646  
647  
648  
649  
650  
651  
652  
653  
654  
655  
656  
657  
658  
659  
660  
661  
662  
663  
664  
665  
666  
667  
668  
669  
670  
671  
672  
673  
674  
675  
676  
677  
678  
679  
680  
681  
682  
683  
684  
685  
686  
687  
688  
689  
690  
691  
692  
693  
694  
695  
696  
697  
698  
699  
700  
701  
702  
703  
704  
705  
706  
707  
708  
709  
710  
711  
712  
713  
714  
715  
716  
717  
718  
719  
720  
721  
722  
723  
724  
725  
726  
727  
728  
729  
730  
731  
732  
733  
734  
735  
736  
737  
738  
739  
740  
741  
742  
743  
744  
745  
746  
747  
748  
749  
750  
751  
752  
753  
754  
755  
756  
757  
758  
759  
760  
761  
762  
763  
764  
765  
766  
767  
768  
769  
770  
771  
772  
773  
774  
775  
776  
777  
778  
779  
780  
781  
782  
783  
784  
785  
786  
787  
788  
789  
790  
791  
792  
793  
794  
795  
796  
797  
798  
799  
800  
801  
802  
803  
804  
805  
806  
807  
808  
809  
810  
811  
812  
813  
814  
815  
816  
817  
818  
819  
820  
821  
822  
823  
824  
825  
826  
827  
828  
829  
830  
831  
832  
833  
834  
835  
836  
837  
838  
839  
840  
841  
842  
843  
844  
845  
846  
847  
848  
849  
850  
851  
852  
853  
854  
855  
856  
857  
858  
859  
860  
861  
862  
863  
864  
865  
866  
867  
868  
869  
870  
871  
872  
873  
874  
875  
876  
877  
878  
879  
880  
881  
882  
883  
884  
885  
886  
887  
888  
889  
890  
891  
892  
893  
894  
895  
896  
897  
898  
899  
900  
901  
902  
903  
904  
905  
906  
907  
908  
909  
910  
911  
912  
913  
914  
915  
916  
917  
918  
919  
920  
921  
922  
923  
924  
925  
926  
927  
928  
929  
930  
931  
932  
933  
934  
935  
936  
937  
938  
939  
940  
941  
942  
943  
944  
945  
946  
947  
948  
949  
950  
951  
952  
953  
954  
955  
956  
957  
958  
959  
960  
961  
962  
963  
964  
965  
966  
967  
968  
969  
970  
971  
972  
973  
974  
975  
976  
977  
978  
979  
980  
981  
982  
983  
984  
985  
986  
987  
988  
989  
990  
991  
992  
993  
994  
995  
996  
997  
998  
999  
1000  
1001  
1002  
1003  
1004  
1005  
1006  
1007  
1008  
1009  
1010  
1011  
1012  
1013  
1014  
1015  
1016  
1017  
1018  
1019  
1020  
1021  
1022  
1023  
1024  
1025  
1026  
1027  
1028  
1029  
1030  
1031  
1032  
1033  
1034  
1035  
1036  
1037  
1038  
1039  
1040  
1041  
1042  
1043  
1044  
1045  
1046  
1047  
1048  
1049  
1050  
1051  
1052  
1053  
1054  
1055  
1056  
1057  
1058  
1059  
1060  
1061  
1062  
1063  
1064  
1065  
1066  
1067  
1068  
1069  
1070  
1071  
1072  
1073  
1074  
1075  
1076  
1077  
1078  
1079  
1080  
1081  
1082  
1083  
1084  
1085  
1086  
1087  
1088  
1089  
1090  
1091  
1092  
1093  
1094  
1095  
1096  
1097  
1098  
1099  
1100  
1101  
1102  
1103  
1104  
1105  
1106  
1107  
1108  
1109  
1110  
1111  
1112  
1113  
1114  
1115  
1116  
1117  
1118  
1119  
1120  
1121  
1122  
1123  
1124  
1125  
1126  
1127  
1128  
1129  
1130  
1131  
1132  
1133  
1134  
1135  
1136  
1137  
1138  
1139  
1140  
1141  
1142  
1143  
1144  
1145  
1146  
1147  
1148  
1149  
1150  
1151  
1152  
1153  
1154  
1155  
1156  
1157  
1158  
1159  
1160  
1161  
1162  
1163  
1164  
1165  
1166  
1167  
1168  
1169  
1170  
1171  
1172  
1173  
1174  
1175  
1176  
1177  
1178  
1179  
1180  
1181  
1182  
1183  
1184  
1185  
1186  
1187  
1188  
1189  
1190  
1191  
1192  
1193  
1194  
1195  
1196  
1197  
1198  
1199  
1200  
1201  
1202  
1203  
1204  
1205  
1206  
1207  
1208  
1209  
1210  
1211  
1212  
1213  
1214  
1215  
1216  
1217  
1218  
1219  
1220  
1221  
1222  
1223  
1224  
1225  
1226  
1227  
1228  
1229  
1230  
1231  
1232  
1233  
1234  
1235  
1236  
1237  
1238  
1239  
1240  
1241  
1242  
1243  
1244  
1245  
1246  
1247  
1248  
1249  
1250  
1251  
1252  
1253  
1254  
1255  
1256  
1257  
1258  
1259  
1260  
1261  
1262  
1263  
1264  
1265  
1266  
1267  
1268  
1269  
1270  
1271  
1272  
1273  
1274  
1275  
1276  
1277  
1278  
1279  
1280  
1281  
1282  
1283  
1284  
1285  
1286  
1287  
1288  
1289  
1290  
1291  
1292  
1293  
1294  
1295  
1296  
1297  
1298  
1299  
1300  
1301  
1302  
1303  
1304  
1305  
1306  
1307  
1308  
1309  
1310  
1311  
1312  
1313  
1314  
1315  
1316  
1317  
1318  
1319  
1320  
1321  
1322  
1323  
1324  
1325  
1326  
1327  
1328  
1329  
1330  
1331  
1332  
1333  
1334  
1335  
1336  
1337  
1338  
1339  
1340  
1341  
1342  
1343  
1344  
1345  
1346  
1347  
1348  
1349  
1350  
1351  
1352  
1353  
1354  
1355  
1356  
1357  
1358  
1359  
1360  
1361  
1362  
1363  
1364  
1365  
1366  
1367  
1368  
1369  
1370  
1371  
1372  
1373  
1374  
1375  
1376  
1377  
1378  
1379  
1380  
1381  
1382  
1383  
1384  
1385  
1386  
1387  
1388  
1389  
1390  
1391  
1392  
1393  
1394  
1395  
1396  
1397  
1398  
1399  
1400  
1401  
1402  
1403  
1404  
1405  
1406  
1407  
1408  
1409  
1410  
1411  
1412  
1413  
1414  
1415  
1416  
1417  
1418  
1419  
1420  
1421  
1422  
1423  
1424  
1425  
1426  
1427  
1428  
1429  
1430  
1431  
1432  
1433  
1434  
1435  
1436  
1437  
1438  
1439  
1440  
1441  
1442  
1443  
1444  
1445  
1446  
1447  
1448  
1449  
1450  
1451  
1452  
1453  
1454  
1455  
1456  
1457  
1458  
1459  
1460  
1461  
1462  
1463  
1464  
1465  
1466  
1467  
1468  
1469  
1470  
1471  
1472  
1473  
1474  
1475  
1476  
1477  
1478  
1479  
1480  
1481  
1482  
1483  
1484  
1485  
1486  
1487  
1488  
1489  
1490  
1491  
1492  
1493  
1494  
1495  
1496  
1497  
1498  
1499  
1500  
1501  
1502  
1503  
1504  
1505  
1506  
1507  
1508  
1509  
1510  
1511  
1512  
1513  
1514  
1515  
1516  
1517  
1518  
1519  
1520  
1521  
1522  
1523  
1524  
1525  
1526  
1527  
1528  
1529  
1530  
1531  
1532  
1533  
1534  
1535  
1536  
1537  
1538  
1539  
1540  
1541  
1542  
1543  
1544  
1545  
1546  
1547  
1548  
1549  
1550  
1551  
1552  
1553  
1554  
1555  
1556  
1557  
1558  
1559  
1560  
1561  
1562  
1563  
1564  
1565  
1566  
1567  
1568  
1569  
1570  
1571  
1572  
1573  
1574  
1575  
1576  
1577  
1578  
1579  
1580  
1581  
1582  
1583  
1584  
1585  
1586  
1587  
1588  
1589  
1590  
1591  
1592  
1593  
1594  
1595  
1596  
1597  
1598  
1599  
1600  
1601  
1602  
1603  
1604  
1605  
1606  
1607  
1608  
1609  
1610  
1611  
1612  
1613  
1614  
1615  
1616  
1617  
1618  
1619  
1620  
1621  
1622  
1623  
1624  
1625  
1626  
1627  
1628  
1629  
1630  
1631  
1632  
1633  
1634  
1635  
1636  
1637  
1638  
1639  
1640  
1641  
1642  
1643  
1644  
1645  
1646  
1647  
1648  
1649  
1650  
1651  
1652  
1653  
1654  
1655  
1656  
1657  
1658  
1659  
1660  
1661  
1662  
1663  
1664  
1665  
1666  
1667  
1668  
1669  
1670  
1671  
1672  
1673  
1674  
1675  
1676  
1677  
1678  
1679  
1680  
1681  
1682  
1683  
1684  
1685  
1686  
1687  
1688  
1689  
1690  
1691  
1692  
1693  
1694  
1695  
1696  
1697  
1698  
1699  
1700  
1701  
1702  
1703  
1704  
1705  
1706  
1707  
1708  
1709  
1710  
1711  
1712  
1713  
1714  
1715  
1716  
1717  
1718  
1719  
1720  
1721  
1722  
1723  
1724  
1725  
1726  
1727  
1728  
1729  
1730  
1731  
1732  
1733  
1734  
1735  
1736  
1737  
1738  
1739  
1740  
1741  
1742  
1743  
1744  
1745  
1746  
1747  
1748  
1749  
1750  
1751  
1752  
1753  
1754  
1755  
1756  
1757  
1758  
1759  
1760  
1761  
1762  
1763  
1764  
1765  
1766  
1767  
1768  
1769  
1770  
1771  
1772  
1773  
1774  
1775  
1776  
1777  
1778  
1779  
1780  
1781  
1782  
1783  
1784  
1785  
1786  
1787  
1788  
1789  
1790  
1791  
1792  
1793  
1794  
1795  
1796  
1797  
1798  
1799  
1800  
1801  
1802  
1803  
1804  
1805  
1806  
1807  
1808  
1809  
1810  
1811  
1812  
1813  
1814  
1815  
1816  
1817  
1818  
1819  
1820  
1821  
1822  
1823  
1824  
1825  
1826  
1827  
1828  
1829  
1830  
1831  
1832  
1833  
1834  
1835  
1836  
1837  
1838  
1839  
1840  
1841  
1842  
1843  
1844  
1845  
1846  
1847  
1848  
1849  
1850  
1851  
1852  
1853  
1854  
1855  
1856  
1857  
1858  
1859  
1860  
1861  
1862  
1863  
1864  
1865  
1866  
1867  
1868  
1869  
1870  
1871  
1872  
1873  
1874  
1875  
1876  
1877  
1878  
1879  
1880  
1881  
1882  
1883  
1884  
1885  
1886  
1887  
1888  
1889  
1890  
1891  
1892  
1893  
1894  
1895  
1896  
1897  
1898  
1899  
1900  
1901  
1902  
1903  
1904  
1905  
1906  
1907  
1908  
1909  
1910  
1911  
1912  
1913  
1914  
1915  
1916  
1917  
1918  
1919  
1920  
1921  
1922  
1923  
1924  
1925  
1926  
1927  
1928  
1929  
1930  
1931  
1932  
1933  
1934  
1935  
1936  
1937  
1938  
1939  
1940  
1941  
1942  
1943  
1944  
1945  
1946  
1947  
1948  
1949  
1950  
1951  
1952  
1953  
1954  
1955  
1956  
1957  
1958  
1959  
1960  
1961  
1962  
1963  
1964  
1965  
1966  
1967  
1968  
1969  
1970  
1971  
1972  
1973  
1974  
1975  
1976  
1977  
1978  
1979  
1980  
1981  
1982  
1983  
1984  
1985  
1986  
1987  
1988  
1989  
1990  
1991  
1992  
1993  
1994  
1995  
1996  
1997  
1998  
1999  
2000  
2001  
2002  
2003  
2004  
2005  
2006  
2007  
2008  
2009  
2010  
2011  
2012  
2013  
2014  
2015  
2016  
2017  
2018  
2019  
2020  
2021  
2022  
2023  
2024  
2025  
2026  
2027  
2028  
2029  
2030  
2031  
2032  
2033  
2034  
2035  
2036  
2037  
2038  
2039  
2040  
2041  
2042  
2043  
2044  
2045  
2046  
2047  
2048  
2049  
2050  
2051  
2052  
2053  
2054  
2055  
2056  
2057  
2058  
2059  
2060  
2061  
2062  
2063  
2064  
2065  
2066  
2067  
2068  
2069  
2070  
2071  
2072  
2073  
2074  
2075  
2076  
2077  
2078  
2079  
2080  
2081  
2082  
2083  
2084  
2085  
2086  
2087  
2088  
2089  
2090  
2091  
2092  
2093  
2094  
2095  
2096  
2097  
2098  
2099  
2100  
2101  
2102  
2103  
2104  
2105  
2106  
2107  
2108  
2109  
2110  
2111  
2112  
2113  
2114  
2115  
2116  
2117  
2118  
2119  
2120  
2121  
2122  
2123  
2124  
2125  
2126  
2127  
2128  
2129  
2130  
2131  
2132  
2133  
2134  
2135  
2136  
2137  
2138  
2139  
2140  
2141  
2142  
2143  
2144  
2145  
2146  
2147  
2148  
2149  
2150  
2151  
2152  
2153  
2154  
2155  
2156  
2157  
2158  
2159  
2160  
2161  
2162  
2163  
2164  
2165  
2166  
2167  
2168  
2169  
2170  
2171  
2172  
2173  
2174  
2175  
2176  
2177  
2178  
2179  
2180  
2181  
2182  
2183  
2184  
2185  
2186  
2187  
2188  
2189  
2190  
2191  
2192  
2193  
2194  
2195  
2196  
2197  
2198  
2199  
2
```

2.6 Iteration & Conditional Structures

The loop continues while the condition evaluates to true.



```
while(condition) {  
    statement;  
}
```

When the condition evaluates to false, the program control goes to the line of code right after the loop.

2.6 Iteration & Conditional Structures


Similar to the previous example (infinite for loop), this program will continue until the user inserts 3 (the condition **(user_value != 3)** becomes **false**).

```
#include <iostream>
using namespace std;

int main ()
{
    int user_value = 0;
    while(user_value != 3)
    {
        cout << " 1 - Addition\n";
        cout << " 2 - Subtraction\n";
        cout << " 3 - Exit Program\n";
        cout << "Insert a number: ";
        cin >> user_value;
        cin.ignore();

        switch (user_value){
            case 1:
                cout << "Addition code\n";
                break;
            case 2:
                cout << "Subtraction code\n";
                break;
        }

        cout << "Bye";
        cin.ignore();
        return 0;
    }
}
```



```
C:\Dev-Cpp\Samples\while.exe
1 - Addition
2 - Subtraction
3 - Exit Program
Insert a number: 1
Addition code
1 - Addition
2 - Subtraction
3 - Exit Program
Insert a number: 2
Subtraction code
1 - Addition
2 - Subtraction
3 - Exit Program
Insert a number: 3
Bye_
```

2.6 Iteration & Conditional Structures

Since the **user_value** is set to 0, in the first iteration the condition is evaluated to true, and the loop begins.


Each time we insert a value the condition is tested again. Once we insert 3, the condition becomes false, and the loop terminates.

```
#include <iostream>
using namespace std;

int main ()
{
    int user_value = 0;
    while(user_value != 3)
    {
        cout << " 1 - Addition\n";
        cout << " 2 - Subtraction\n";
        cout << " 3 - Exit Program\n";
        cout << "Insert a number: ";
        cin >> user_value;
        cin.ignore();

        switch (user_value){
            case 1:
                cout << "Addition code\n";
                break;
            case 2:
                cout << "Subtraction code\n";
                break;
        }

        cout << "Bye";
        cin.ignore();
        return 0;
    }
}
```



```
C:\Dev-Cpp\Samples\while.exe
1 - Addition
2 - Subtraction
3 - Exit Program
Insert a number: 1
Addition code
1 - Addition
2 - Subtraction
3 - Exit Program
Insert a number: 2
Subtraction code
1 - Addition
2 - Subtraction
3 - Exit Program
Insert a number: 3
Bye_
```

2.6 Iteration & Conditional Structures

Unlike for and while loops, which test the condition at the beginning of the loop, the **do-while** loop checks its condition at the end of the loop; a do-while loop always executes at least once.

The general form of the do-while loop is:

```
</>
do{
    statement;
}while(condition);
```


2.6 Iteration & Conditional Structures

The **do-while** loop iterates until the condition evaluates to false.

The program on the left will first get the user input, and then it will stop only when the condition becomes false.

```
#include <iostream>
using namespace std;

int main ()
{
    int user_value = 0;
    while(user_value != 3)
    {
        cout << " 1 - Addition\n";
        cout << " 2 - Subtraction\n";
        cout << " 3 - Exit Program\n";
        cout << "Insert a number: ";
        cin >> user_value;
        cin.ignore();

        switch (user_value){
            case 1:
                cout << "Addition code\n";
                break;
            case 2:
                cout << "Subtraction code\n";
                break;
        }

        cout << "Bye";
        cin.ignore();
        return 0;
    }
}
```



```
C:\Dev-Cpp\Samples\while.exe
1 - Addition
2 - Subtraction
3 - Exit Program
Insert a number: 1
Addition code
1 - Addition
2 - Subtraction
3 - Exit Program
Insert a number: 2
Subtraction code
1 - Addition
2 - Subtraction
3 - Exit Program
Insert a number: 3
Bye_
```


2.6 Iteration & Conditional Structures

You have all the skills needed to write a program that prints out a simple Xmas tree. Given a number, write a program that will display a triangle made of * chars, which has as many lines as the number provided.

Let's type in 5; the program should print something like this:

```

*
***
*****
*****
*****
*****
*****
*****
```

2.6 Iteration & Conditional Structures

There are many different ways to do it. This code is one of them.

```
#include <iostream>
using namespace std;
int main()
{
    int i, j, lines;
    cout << "Please enter a number " << endl;
    cin >> lines;
    cin.ignore();

    for (j=1; j<=lines * 2; j=j+2)
    {
        for (i=j; i<=lines*2-2; i=i+2)
        {
            cout << " ";
        }
        for (i=1; i<=j; i++)
        {
            cout << "*";
        }
        cout << endl;
    }

    cin.ignore();
}
```

[illegible]

2.6 Iteration & Conditional Structures

JUMP STATEMENT

C++ has four statements that can change the normal execution flow: **return**, **goto**, **break** and **continue**.

While **return** and **goto** are mostly used anywhere in your program, **break** and **continue** statements are often used in conjunction with any of the loop statements seen before.

2.6 Iteration & Conditional Structures

The return statement is used to return from a function. It may or may not have a value associated with it.

The general form of a **return** statement is:



```
return expression;
```

```
21 def _init__(self, name, location, candidates):
22     self.name = name
23     self.location = location
24     self.candidates = candidates
25     self.observations = []
26     self.control = None
27
28 def _init_obs(self, experiment, observations = [], control = None):
29     @experiment = experiment
30     @observations = observations
31     @control = control
32     @candidates = candidates
33     evaluate_candidates
34
35 freeze
36
37 # Main: the experiment's context
38 def context:
39     experiment.context
40
41 # Init: the name of the experiment
42 def experiment_name:
43     experiment.name
44
45 # Eval: the result a match between all
46 def match:
47     result = 1
48     return result
```

2.6 Iteration & Conditional Structures



```
return expression;
```

The **expression** must be used only if the function has a returning value. In this case, the value of the expression will become the return value of the function and can be associated with a variable.

We can use as many **return** statements as we like within a function. However, the function stops executing as soon as it encounters the first return.



2.6 Iteration & Conditional Structures


The main concern about the **goto** is that it makes the programs unreadable. It can be used to jump to a specific statement, such as jumping out of a set of deeply nested loops.

Its general form is:

```
</>  
goto label;  
...  
...  
label:
```

it makes the jump to a specific of deeply nested

2.6 Iteration & Conditional Structures

```
 goto label;  
...  
...  
label:
```

The **goto** statement requires a label. A **label** is an identifier followed by a colon.

Note that the label must be in the same block of statements as the **goto** that uses it, so we cannot jump between functions.

```
23  
24  
25  
26  
27  
28  
29  
30  
31  
32  
33  
34  
35  
36  
37  
38  
39  
40  
41  
42  
43  
44  
45  
46  
47  
48  
49  
50  
51  
52  
53  
54  
55  
56  
57  
58  
59  
60  
61  
62  
63  
64  
65  
66  
67  
68  
69  
70  
71  
72  
73  
74  
75  
76  
77  
78  
79  
80  
81  
82  
83  
84  
85  
86  
87  
88  
89  
90  
91  
92  
93  
94  
95  
96  
97  
98  
99  
100  
101  
102  
103  
104  
105  
106  
107  
108  
109  
110  
111  
112  
113  
114  
115  
116  
117  
118  
119  
120  
121  
122  
123  
124  
125  
126  
127  
128  
129  
130  
131  
132  
133  
134  
135  
136  
137  
138  
139  
140  
141  
142  
143  
144  
145  
146  
147  
148  
149  
150  
151  
152  
153  
154  
155  
156  
157  
158  
159  
160  
161  
162  
163  
164  
165  
166  
167  
168  
169  
170  
171  
172  
173  
174  
175  
176  
177  
178  
179  
180  
181  
182  
183  
184  
185  
186  
187  
188  
189  
190  
191  
192  
193  
194  
195  
196  
197  
198  
199  
200  
201  
202  
203  
204  
205  
206  
207  
208  
209  
210  
211  
212  
213  
214  
215  
216  
217  
218  
219  
220  
221  
222  
223  
224  
225  
226  
227  
228  
229  
230  
231  
232  
233  
234  
235  
236  
237  
238  
239  
240  
241  
242  
243  
244  
245  
246  
247  
248  
249  
250  
251  
252  
253  
254  
255  
256  
257  
258  
259  
260  
261  
262  
263  
264  
265  
266  
267  
268  
269  
270  
271  
272  
273  
274  
275  
276  
277  
278  
279  
280  
281  
282  
283  
284  
285  
286  
287  
288  
289  
290  
291  
292  
293  
294  
295  
296  
297  
298  
299  
300  
301  
302  
303  
304  
305  
306  
307  
308  
309  
310  
311  
312  
313  
314  
315  
316  
317  
318  
319  
320  
321  
322  
323  
324  
325  
326  
327  
328  
329  
330  
331  
332  
333  
334  
335  
336  
337  
338  
339  
340  
341  
342  
343  
344  
345  
346  
347  
348  
349  
350  
351  
352  
353  
354  
355  
356  
357  
358  
359  
360  
361  
362  
363  
364  
365  
366  
367  
368  
369  
370  
371  
372  
373  
374  
375  
376  
377  
378  
379  
380  
381  
382  
383  
384  
385  
386  
387  
388  
389  
390  
391  
392  
393  
394  
395  
396  
397  
398  
399  
400  
401  
402  
403  
404  
405  
406  
407  
408  
409  
410  
411  
412  
413  
414  
415  
416  
417  
418  
419  
420  
421  
422  
423  
424  
425  
426  
427  
428  
429  
430  
431  
432  
433  
434  
435  
436  
437  
438  
439  
440  
441  
442  
443  
444  
445  
446  
447  
448  
449  
450  
451  
452  
453  
454  
455  
456  
457  
458  
459  
460  
461  
462  
463  
464  
465  
466  
467  
468  
469  
470  
471  
472  
473  
474  
475  
476  
477  
478  
479  
480  
481  
482  
483  
484  
485  
486  
487  
488  
489  
490  
491  
492  
493  
494  
495  
496  
497  
498  
499  
500  
501  
502  
503  
504  
505  
506  
507  
508  
509  
510  
511  
512  
513  
514  
515  
516  
517  
518  
519  
520  
521  
522  
523  
524  
525  
526  
527  
528  
529  
530  
531  
532  
533  
534  
535  
536  
537  
538  
539  
540  
541  
542  
543  
544  
545  
546  
547  
548  
549  
550  
551  
552  
553  
554  
555  
556  
557  
558  
559  
560  
561  
562  
563  
564  
565  
566  
567  
568  
569  
570  
571  
572  
573  
574  
575  
576  
577  
578  
579  
580  
581  
582  
583  
584  
585  
586  
587  
588  
589  
590  
591  
592  
593  
594  
595  
596  
597  
598  
599  
600  
601  
602  
603  
604  
605  
606  
607  
608  
609  
610  
611  
612  
613  
614  
615  
616  
617  
618  
619  
620  
621  
622  
623  
624  
625  
626  
627  
628  
629  
630  
631  
632  
633  
634  
635  
636  
637  
638  
639  
640  
641  
642  
643  
644  
645  
646  
647  
648  
649  
650  
651  
652  
653  
654  
655  
656  
657  
658  
659  
660  
661  
662  
663  
664  
665  
666  
667  
668  
669  
670  
671  
672  
673  
674  
675  
676  
677  
678  
679  
680  
681  
682  
683  
684  
685  
686  
687  
688  
689  
690  
691  
692  
693  
694  
695  
696  
697  
698  
699  
700  
701  
702  
703  
704  
705  
706  
707  
708  
709  
710  
711  
712  
713  
714  
715  
716  
717  
718  
719  
720  
721  
722  
723  
724  
725  
726  
727  
728  
729  
730  
731  
732  
733  
734  
735  
736  
737  
738  
739  
740  
741  
742  
743  
744  
745  
746  
747  
748  
749  
750  
751  
752  
753  
754  
755  
756  
757  
758  
759  
760  
761  
762  
763  
764  
765  
766  
767  
768  
769  
770  
771  
772  
773  
774  
775  
776  
777  
778  
779  
780  
781  
782  
783  
784  
785  
786  
787  
788  
789  
790  
791  
792  
793  
794  
795  
796  
797  
798  
799  
800  
801  
802  
803  
804  
805  
806  
807  
808  
809  
810  
811  
812  
813  
814  
815  
816  
817  
818  
819  
820  
821  
822  
823  
824  
825  
826  
827  
828  
829  
830  
831  
832  
833  
834  
835  
836  
837  
838  
839  
840  
841  
842  
843  
844  
845  
846  
847  
848  
849  
850  
851  
852  
853  
854  
855  
856  
857  
858  
859  
860  
861  
862  
863  
864  
865  
866  
867  
868  
869  
870  
871  
872  
873  
874  
875  
876  
877  
878  
879  
880  
881  
882  
883  
884  
885  
886  
887  
888  
889  
890  
891  
892  
893  
894  
895  
896  
897  
898  
899  
900  
901  
902  
903  
904  
905  
906  
907  
908  
909  
910  
911  
912  
913  
914  
915  
916  
917  
918  
919  
920  
921  
922  
923  
924  
925  
926  
927  
928  
929  
930  
931  
932  
933  
934  
935  
936  
937  
938  
939  
940  
941  
942  
943  
944  
945  
946  
947  
948  
949  
950  
951  
952  
953  
954  
955  
956  
957  
958  
959  
960  
961  
962  
963  
964  
965  
966  
967  
968  
969  
970  
971  
972  
973  
974  
975  
976  
977  
978  
979  
980  
981  
982  
983  
984  
985  
986  
987  
988  
989  
990  
991  
992  
993  
994  
995  
996  
997  
998  
999  
1000  
1001  
1002  
1003  
1004  
1005  
1006  
1007  
1008  
1009  
1010  
1011  
1012  
1013  
1014  
1015  
1016  
1017  
1018  
1019  
1020  
1021  
1022  
1023  
1024  
1025  
1026  
1027  
1028  
1029  
1030  
1031  
1032  
1033  
1034  
1035  
1036  
1037  
1038  
1039  
1040  
1041  
1042  
1043  
1044  
1045  
1046  
1047  
1048  
1049  
1050  
1051  
1052  
1053  
1054  
1055  
1056  
1057  
1058  
1059  
1060  
1061  
1062  
1063  
1064  
1065  
1066  
1067  
1068  
1069  
1070  
1071  
1072  
1073  
1074  
1075  
1076  
1077  
1078  
1079  
1080  
1081  
1082  
1083  
1084  
1085  
1086  
1087  
1088  
1089  
1090  
1091  
1092  
1093  
1094  
1095  
1096  
1097  
1098  
1099  
1100  
1101  
1102  
1103  
1104  
1105  
1106  
1107  
1108  
1109  
1110  
1111  
1112  
1113  
1114  
1115  
1116  
1117  
1118  
1119  
1120  
1121  
1122  
1123  
1124  
1125  
1126  
1127  
1128  
1129  
1130  
1131  
1132  
1133  
1134  
1135  
1136  
1137  
1138  
1139  
1140  
1141  
1142  
1143  
1144  
1145  
1146  
1147  
1148  
1149  
1150  
1151  
1152  
1153  
1154  
1155  
1156  
1157  
1158  
1159  
1160  
1161  
1162  
1163  
1164  
1165  
1166  
1167  
1168  
1169  
1170  
1171  
1172  
1173  
1174  
1175  
1176  
1177  
1178  
1179  
1180  
1181  
1182  
1183  
1184  
1185  
1186  
1187  
1188  
1189  
1190  
1191  
1192  
1193  
1194  
1195  
1196  
1197  
1198  
1199  
1200  
1201  
1202  
1203  
1204  
1205  
1206  
1207  
1208  
1209  
1210  
1211  
1212  
1213  
1214  
1215  
1216  
1217  
1218  
1219  
1220  
1221  
1222  
1223  
1224  
1225  
1226  
1227  
1228  
1229  
1230  
1231  
1232  
1233  
1234  
1235  
1236  
1237  
1238  
1239  
1240  
1241  
1242  
1243  
1244  
1245  
1246  
1247  
1248  
1249  
1250  
1251  
1252  
1253  
1254  
1255  
1256  
1257  
1258  
1259  
1260  
1261  
1262  
1263  
1264  
1265  
1266  
1267  
1268  
1269  
1270  
1271  
1272  
1273  
1274  
1275  
1276  
1277  
1278  
1279  
1280  
1281  
1282  
1283  
1284  
1285  
1286  
1287  
1288  
1289  
1290  
1291  
1292  
1293  
1294  
1295  
1296  
1297  
1298  
1299  
1300  
1301  
1302  
1303  
1304  
1305  
1306  
1307  
1308  
1309  
1310  
1311  
1312  
1313  
1314  
1315  
1316  
1317  
1318  
1319  
1320  
1321  
1322  
1323  
1324  
1325  
1326  
1327  
1328  
1329  
1330  
1331  
1332  
1333  
1334  
1335  
1336  
1337  
1338  
1339  
1340  
1341  
1342  
1343  
1344  
1345  
1346  
1347  
1348  
1349  
1350  
1351  
1352  
1353  
1354  
1355  
1356  
1357  
1358  
1359  
1360  
1361  
1362  
1363  
1364  
1365  
1366  
1367  
1368  
1369  
1370  
1371  
1372  
1373  
1374  
1375  
1376  
1377  
1378  
1379  
1380  
1381  
1382  
1383  
1384  
1385  
1386  
1387  
1388  
1389  
1390  
1391  
1392  
1393  
1394  
1395  
1396  
1397  
1398  
1399  
1400  
1401  
1402  
1403  
1404  
1405  
1406  
1407  
1408  
1409  
1410  
1411  
1412  
1413  
1414  
1415  
1416  
1417  
1418  
1419  
1420  
1421  
1422  
1423  
1424  
1425  
1426  
1427  
1428  
1429  
1430  
1431  
1432  
1433  
1434  
1435  
1436  
1437  
1438  
1439  
1440  
1441  
1442  
1443  
1444  
1445  
1446  
1447  
1448  
1449  
1450  
1451  
1452  
1453  
1454  
1455  
1456  
1457  
1458  
1459  
1460  
1461  
1462  
1463  
1464  
1465  
1466  
1467  
1468  
1469  
1470  
1471  
1472  
1473  
1474  
1475  
1476  
1477  
1478  
1479  
1480  
1481  
1482  
1483  
1484  
1485  
1486  
1487  
1488  
1489  
1490  
1491  
1492  
1493  
1494  
1495  
1496  
1497  
1498  
1499  
1500  
1501  
1502  
1503  
1504  
1505  
1506  
1507  
1508  
1509  
1510  
1511  
1512  
1513  
1514  
1515  
1516  
1517  
1518  
1519  
1520  
1521  
1522  
1523  
1524  
1525  
1526  
1527  
1528  
1529  
1530  
1531  
1532  
1533  
1534  
1535  
1536  
1537  
1538  
1539  
1540  
1541  
1542  
1543  
1544  
1545  
1546  
1547  
1548  
1549  
1550  
1551  
1552  
1553  
1554  
1555  
1556  
1557  
1558  
1559  
1560  
1561  
1562  
1563  
1564  
1565  
1566  
1567  
1568  
1569  
1570  
1571  
1572  
1573  
1574  
1575  
1576  
1577  
1578  
1579  
1580  
1581  
1582  
1583  
1584  
1585  
1586  
1587  
1588  
1589  
1590  
1591  
1592  
1593  
1594  
1595  
1596  
1597  
1598  
1599  
1600  
1601  
1602  
1603  
1604  
1605  
1606  
1607  
1608  
1609  
1610  
1611  
1612  
1613  
1614  
1615  
1616  
1617  
1618  
1619  
1620  
1621  
1622  
1623  
1624  
1625  
1626  
1627  
1628  
1629  
1630  
1631  
1632  
1633  
1634  
1635  
1636  
1637  
1638  
1639  
1640  
1641  
1642  
1643  
1644  
1645  
1646  
1647  
1648  
1649  
1650  
1651  
1652  
1653  
1654  
1655  
1656  
1657  
1658  
1659  
1660  
1661  
1662  
1663  
1664  
1665  
1666  
1667  
1668  
1669  
1670  
1671  
1672  
1673  
1674  
1675  
1676  
1677  
1678  
1679  
1680  
1681  
1682  
1683  
1684  
1685  
1686  
1687  
1688  
1689  
1690  
1691  
1692  
1693  
1694  
1695  
1696  
1697  
1698  
1699  
1700  
1701  
1702  
1703  
1704  
1705  
1706  
1707  
1708  
1709  
1710  
1711  
1712  
1713  
1714  
1715  
1716  
1717  
1718  
1719  
1720  
1721  
1722  
1723  
1724  
1725  
1726  
1727  
1728  
1729  
1730  
1731  
1732  
1733  
1734  
1735  
1736  
1737  
1738  
1739  
1740  
1741  
1742  
1743  
1744  
1745  
1746  
1747  
1748  
1749  
1750  
1751  
1752  
1753  
1754  
1755  
1756  
1757  
1758  
1759  
1760  
1761  
1762  
1763  
1764  
1765  
1766  
1767  
1768  
1769  
1770  
1771  
1772  
1773  
1774  
1775  
1776  
1777  
1778  
1779  
1780  
1781  
1782  
1783  
1784  
1785  
1786  
1787  
1788  
1789  
1790  
1791  
1792  
1793  
1794  
1795  
1796  
1797  
1798  
1799  
1800  
1801  
1802  
1803  
1804  
1805  
1806  
1807  
1808  
1809  
1810  
1811  
1812  
1813  
1814  
1815  
1816  
1817  
1818  
1819  
1820  
1821  
1822  
1823  
1824  
1825  
1826  
1827  
1828  
1829  
1830  
1831  
1832  
1833  
1834  
1835  
1836  
1837  
1838  
1839  
1840  
1841  
1842  
1843  
1844  
1845  
1846  
1847  
1848  
1849  
1850  
1851  
1852  
1853  
1854  
1855  
1856  
1857  
1858  
1859  
1860  
1861  
1862  
1863  
1864  
1865  
1866  
1867  
1868  
1869  
1870  
1871  
1872  
1873  
1874  
1875  
1876  
1877  
1878  
1879  
1880  
1881  
1882  
1883  
1884  
1885  
1886  
1887  
1888  
1889  
1890  
1891  
1892  
1893  
1894  
1895  
1896  
1897  
1898  
1899  
1900  
1901  
1902  
1903  
1904  
1905  
1906  
1907  
1908  
1909  
1910  
1911  
1912  
1913  
1914  
1915  
1916  
1917  
1918  
1919  
1920  
1921  
1922  
1923  
1924  
1925  
1926  
1927  
1928  
1929  
1930  
1931  
1932  
1933  
1934  
1935  
1936  
1937  
1938  
1939  
1940  
1941  
1942  
1943  
1944  
1945  
1946  
1947  
1948  
1949  
1950  
1951  
1952  
1953  
1954  
1955  
1956  
1957  
1958  
1959  
1960  
1961  
1962  
1963  
1964  
1965  
1966  
1967  
1968  
1969  
1970  
1971  
1972  
1973  
1974  
1975  
1976  
1977  
1978  
1979  
1980  
1981  
1982  
1983  
1984  
1985  
1986  
1987  
1988  
1989  
1990  
1991  
1992  
1993  
1994  
1995  
1996  
1997  
1998  
1999  
2000  
2001  
2002  
2003  
2004  
2005  
2006  
2007  
2008  
2009  
2010  
2011  
2012  
2013  
2014  
2015  
2016  
2017  
2018  
2019  
2020  
2021  
2022  
2023  
2024  
2025  
2026  
2027  
2028  
2029  
2030  
2031  
2032  
2033  
2034  
2035  
2036  
2037  
2038  
2039  
2040  
2041  
2042  
2043  
2044  
2045  
2046  
2047  
2048  
2049  
2050  
2051  
2052  
2053  
2054  
2055  
2056  
2057  
2058  
2059  
2060  
2061  
2062  
2063  
2064  
2065  
2066  
2067  
2068  
2069  
2070  
2071  
2072  
2073  
2074  
2075  
2076  
2077  
2078  
2079  
2080  
2081  
2082  
2083  
2084  
2085  
2086  
2087  
2088  
2089  
2090  
2091  
2092  
2093  
2094  
2095  
2096  
2097  
2098  
2099  
2100  
2101  
2102  
2103  
2104  
2105  
2106  
2107  
2108  
2109  
2110  
2111  
2112  
2113  
2114  
2115  
2116  
2117  
2118  
2119  
2120  
2121  
2122  
2123  
2124  
2125  
2126  
2127  
2128  
2129  
2130  
2131  
2132  
2133  
2134  
2135  
2136  
2137  
2138  
2139  
2140  
2141  
2142  
2143  
2144  
2145  
2146  
2147  
2148  
2149  
2150  
2151  
2152  
2153  
2154  
2155  
2156  
2157  
2158  
2159  
2160  
2161  
2162  
2163  
2164  
2165  
2166  
2167  
2168  
2169  
2170  
2171  
2172  
2173  
2174  
2175  
2176  
2177  
2178  
2179  
2180  
2181  
2182  
2183  
2184  
2185  
2186  
2187  
2188  
2189  
2190  
2191  
2192  
2193  
2194  
2195  
2196  
2197  
2198  
2199  
2200  
2201  
2202  
2203  
2204  
2205  
2206  
2207  
2208  
2209  
2210  
2211  
2212  
2213  
2214  
2215  
2216  
2217  
22
```


2.6 Iteration & Conditional Structures

The **break** statement has two uses:

- Terminates a case in the switch statement
- Forces the termination of a loop, bypassing the normal loop conditional test.

When we use the break statement within a loop, the loop terminates, and the program control resumes at the statement after the loop.

```
27 def skillize(experiment, observations = [], control = null)
28   # Skillize the experiment's control
29   @experiment = experiment
30   @observations = observations
31   @control = control
32   @candidates = observations -> control
33   evaluate_candidates
34
35   freeze
36   end
37
38 # P4040: the experiment's control
39 def control
40   # Skillize the experiment's control
41   @experiment = experiment
42   @observations = observations
43   @control = control
44   @candidates = observations -> control
45   evaluate_candidates
46
47   freeze
48   end
49
50 def match
51   # Skillize the experiment's control
52   @experiment = experiment
53   @observations = observations
54   @control = control
55   @candidates = observations -> control
56   evaluate_candidates
57
58   freeze
59   end
60
61 # P4040: the experiment's control
62 def control
63   # Skillize the experiment's control
64   @experiment = experiment
65   @observations = observations
66   @control = control
67   @candidates = observations -> control
68   evaluate_candidates
69
70   freeze
71   end
```



2.6 Iteration & Conditional Structures


This program prints numbers from 0 to 10.

Even if the loop should continue until x is 100, it terminates because the **break** causes the loop to terminate, overriding the conditional test $x < 100$.

```
#include <iostream>
using namespace std;

int main()
{
    for (int x = 0; x <= 100; x++)
    {
        cout << x << ", ";
        if (x == 10) break;
    }

    cin.ignore();
}
```



C:\Dev-Cpp\Samples\break.exe
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, _

2.6 Iteration & Conditional Structures

The **continue** statement works similarly to the break statement. Instead of forcing termination, it forces the code to continue to the next iteration of a loop, skipping any code in between.

So, in a *for loop*, **continue** causes an increment of the control variable and a new iteration.

```
23 # the experiment's context
24 def context(self):
25     """Returns the experiment's context"""
26     return self._context
27
28 # initialize experiment, observations = list of candidates
29 @experiment
30 @observations
31 @control
32 @candidates
33 evaluate_candidates
34
35 freeze
36
37 # the experiment's context
38 def context(self):
39     return self._context
40
41 # the experiment's name
42 @experiment_name
43 @experiment_name
44
45 # the result of the match between the candidates
46 def match(self):
47     """Returns the result of the match between the candidates"""
48     return self._match
49
50 # the result of the match between the candidates
51 @result
52 @result
53
54 # the result of the match between the candidates
55 @result
56 @result
57
58 # the result of the match between the candidates
59 @result
60 @result
61
62 # the result of the match between the candidates
63 @result
64 @result
65
66 # the result of the match between the candidates
67 @result
68 @result
69
70 # the result of the match between the candidates
71 @result
72 @result
73
74 # the result of the match between the candidates
75 @result
76 @result
77
78 # the result of the match between the candidates
79 @result
80 @result
81
82 # the result of the match between the candidates
83 @result
84 @result
85
86 # the result of the match between the candidates
87 @result
88 @result
89
90 # the result of the match between the candidates
91 @result
92 @result
93
94 # the result of the match between the candidates
95 @result
96 @result
97
98 # the result of the match between the candidates
99 @result
100 @result
```

2.6 Iteration & Conditional Structures

This program checks how many numbers are odd and how many are even. As you can see, if the number is even (**if(user_value % 2 == 0)**) the program encounters the **continue** statement. In this case, the odd variable will not be incremented.

```
#include <iostream>
using namespace std;

int main ()
{
    int user_value;
    int even = 0;
    int odd = 0;
    for (int i = 0; i < 5; i++){
        cout << "Insert a number: ";
        cin >> user_value;
        cin.ignore();

        if(user_value % 2 == 0){
            ++even;
            continue;
        }
        ++odd;
    }

    cout << endl << "even: " << even << " - odd: " << odd ;

    cin.ignore();
    return 0;
}
```

C:\Dev-Cpp\Samples\continue.exe

```
Insert a number: 1
Insert a number: 2
Insert a number: 3
Insert a number: 4
Insert a number: 5
```

even: 2 - odd: 3_

Pointers



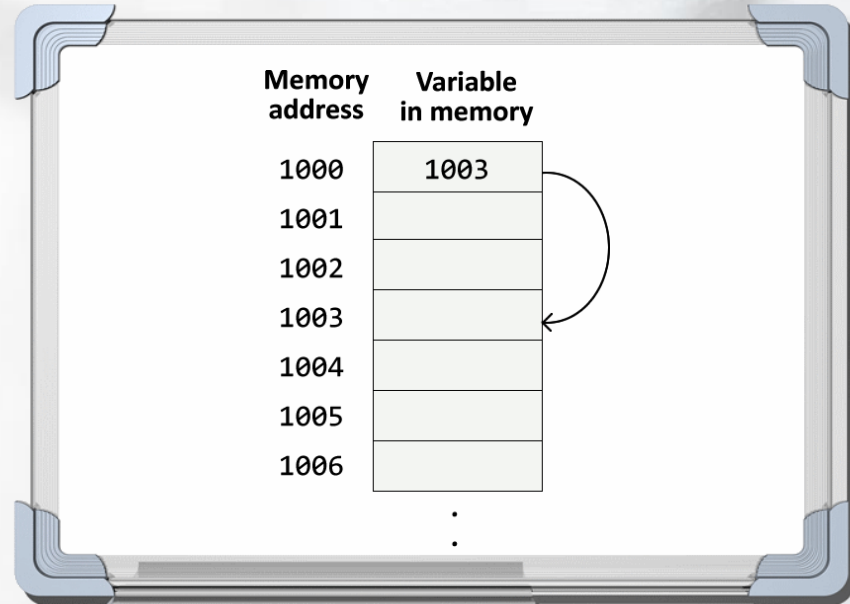
2.7 Pointers

A pointer is a variable that holds a memory address. This address is the location of another object in memory.

For example, if one variable (a) contains the address of another variable (b), a is said to point to b.

2.7 Pointers

The following image shows the situation where one variable points to another.



2.7 Pointers

If a variable is a pointer, it must be declared in a different way. We will write an `*` and the variable name. The general form is:

```
</>  
type *name;
```


Where:

- **type** is the base type of the pointer (int, char...);
- **name** is the name (identifier) of the pointer variable

Type defines the type of variable the pointer can point to.

2.7 Pointers


There are two special pointer operators: `*` and `&`. The `&` returns the memory address of the variable. For example:

```
 x = &y;
```

Put the memory address of the variable **y** into **x**. This address is the computer's internal location of the variable. It is not the value of `y` but its address. In other words, the `&` operator returns “the address of”. Therefore, the above statement means “**x** holds the **address** of **y**”.

2.7 Pointers

The second pointer operator, `*`, is the complement of `&`. It returns the value located at the address of the following operator. For example:



```
x = *y;
```

places the value in memory pointed by **y**, into **x**. So if **y** contains the memory address of another variable, let us say **counter**, **x** will have the value of **counter**.

```
24 with_reader: read_experiment()
25
26 # Optional: create a log object
27
28 # Experiment: the experiment will result in 10
29 # observations - an array of Observations, in memory
30 # control: the control observation
31
32
33 def initialize(experiment, observations = [], control = null)
34   @experiment = experiment
35   @observations = observations
36   @control = control
37   @candidates = observations + [control]
38   evaluate_candidates
39
40   freeze
41 end
42
43 # Public: the experiment's name
44 def name
45   @name
46 end
47
48 # Public: was the result a match between an
49 def match?
50   @match
51 end
52
53 # Private: result of the experiment
54 def result
55   @result
56 end
57
58 # Private: result of the experiment
59 def result
60   @result
61 end
```

2.7 Pointers

This program shows how the two pointer operators work. In the indicated statement, **p1** points to the memory address of the variable **x**.

```
#include <iostream>
using namespace std;

int main ()
{
    int x = 10;
    int y = 0;
    int *p1, *p2;

    p1 = &x;
    p2 = p1;
    y = *p2;
    *p2 = 5;

    cout << "p1 : " << p1 << "\np2 : " << p2 << endl;
    cout << "\ny : " << y << "\nx : " << x << endl;

    cin.ignore();
    return 0;
}
```

2.7 Pointers

With this statement, **p2** points to the same memory address of **p1**, meaning that **p2** now points to the memory address of **x**.

```
#include <iostream>
using namespace std;

int main ()
{
    int x = 10;
    int y = 0;
    int *p1, *p2;

    p1 = &x;
    p2 = p1;
    y = *p2;
    *p2 = 5;

    cout << "p1 : " << p1 << "\np2 : " << p2 << endl;
    cout << "\ny : " << y << "\nx : " << x << endl;

    cin.ignore();
    return 0;
}
```

2.7 Pointers

The next statement assigns to variable **y**, the value located at the memory address pointed by **p2**. In other words, **y** now contains the value of the variable **x**.

```
#include <iostream>
using namespace std;

int main ()
{
    int x = 10;
    int y = 0;
    int *p1, *p2;

    p1 = &x;
    p2 = p1;
    y = *p2;
    *p2 = 5;

    cout << "p1 : " << p1 << "\np2 : " << p2 << endl;
    cout << "\ny : " << y << "\nx : " << x << endl;

    cin.ignore();
    return 0;
}
```

2.7 Pointers

With this last statement, we assign 5 to the value located at the memory address pointed by **p2**. Remember that **p2** was pointing to the memory address of **x**, so the value of **x** is now 5.

```
#include <iostream>
using namespace std;

int main ()
{
    int x = 10;
    int y = 0;
    int *p1, *p2;

    p1 = &x;
    p2 = p1;
    y = *p2;
    *p2 = 5;

    cout << "p1 : " << p1 << "\np2 : " << p2 << endl;
    cout << "\ny : " << y << "\nx : " << x << endl;

    cin.ignore();
    return 0;
}
```

C:\Dev-Cpp\Samples\pointers.exe

p1 : 0x28ff44
p2 : 0x28ff44

y : 10
x : 5

Arrays



2.8 Arrays

An array is a collection of variables of the same type. A specific element in an array is accessed by an index.



2.8 Arrays

An array may have several dimensions. The general form is:

```
</>  
type var_name[size];
```

Where:

- **type** declares the type of the array (the type of each element in the array)
- **size** defines the length of the array (how many elements the array can contain)

All arrays have 0 as an index of the first element. Therefore, if we declare an array of 10 elements its index goes from [0] to [9].

2.8 Arrays

The program creates an integer array of 20 elements. The **for** loop places numbers from 0 to 19 inside the array.

Remember that an array is accessed by indexing the array name. This is done by placing the index of the element we want to access within square brackets.

```
#include <iostream>
using namespace std;

int main ()
{
    int x[20];
    int i;

    for (i=0; i<20; ++i){
        x[i] = i;
    }

    for (i=0; i<20; ++i){
        cout << x[i] << ", ";
    }

    cin.ignore();
    return 0;
}
```

Array declaration

Element accessed by indexing the array name

Print array elements with index from 0 to 20

Functions




2.9 Functions

Functions are blocks of statements defined under a name. In other words, it is a group of statements that get executed when this name is called in the program.

Functions perform a given operation and often return a result.

2.9 Functions


The general form of a function is:

```
type function_name(parameter1, parameter2,...){  
    statements;  
}
```

Where:

- **type** specifies the type of data that the function returns
- **function_name** is the identifier used to call that function
- **parameters** is a comma-separated list of variables and their associated types. Those variables receive the values when the function is called.

2.9 Functions

```
type function_name(){  
    statements;  
}
```

Note that functions may have no parameters, but they still require parentheses '()'.

2.9 Functions

Here is an example of a function that takes two numbers from the user, sums them and returns the result of the operation.

```
#include <iostream>
using namespace std;

int sum (int x, int y){
    int z;
    z = x + y;
    return(z);
}

int main()
{
    int a, b, result;
    cout << "Please enter two numbers: " << endl;
    cin >> a;
    cin.ignore();
    cin >> b;
    cin.ignore();

    result = sum(a,b);

    cout << "The result of " << a << "+" << b << " is " << result;

    cin.ignore();
}
```

Function definition

Function call

2.9 Functions

```
#include <iostream>
using namespace std;

int sum (int x, int y){
    int z;
    z = x + y;
    return(z);
}
```

Formal parameters

Since this function uses two arguments, we have to declare two variables (**int x, int y**) that will accept the values from the caller. These variables are called formal parameters of the function. They are like any other local variable inside the function, and they are declared when the function is called and destroyed when the function returns.

2.9 Functions

Let's see how this program works step by step.

As usual, the first instruction to be executed is the one within the main function.

```
#include <iostream>
using namespace std;

int sum (int x, int y){

    int z;
    z = x + y;
    return(z);

}

int main()
{
    int a, b, result;
    cout << "Please enter two numbers: " << endl;
    cin >> a;
    cin.ignore();
    cin >> b;
    cin.ignore();

    result = sum(a,b);

    cout << "The result of " << a << "+" << b << " is " << result;

    cin.ignore();
}
```


2.9 Functions

The program will then ask the user to insert two values: **a** and **b**.

Let's say that we insert the following values: 5 and 3.

```
#include <iostream>
using namespace std;

int sum (int x, int y){

    int z;
    z = x + y;
    return(z);

}

int main()
{
    int a, b, result;
    cout << "Please enter two numbers: " << endl;
    cin >> a;
    cin.ignore();
    cin >> b;
    cin.ignore();

    result = sum(a,b);

    cout << "The result of " << a << "+" << b << " is " << result;

    cin.ignore();
}
```

a = 5;
b = 3;

2.9 Functions

Here the function **sum** is called, and the control is lost by main, and it is passed to **sum**.

The value of both arguments in the caller function (**a, b**) are copied into the local variables (**int x, int y**) of the function **sum**.

```
#include <iostream>
using namespace std;

int sum (int x, int y){

    int z;
    z = x + y;
    return(z);

}

int main()
{
    int a, b, result;
    cout << "Please enter two numbers: " << endl;
    cin >> a;
    cin.ignore();
    cin >> b;
    cin.ignore();

    result = sum(a,b);

    cout << "The result of " << a << "+" << b << " is " << result;
    cin.ignore();
}
```

The diagram illustrates the flow of execution. It starts with a red box containing the assignment `a = 5; b = 3;`. Two arrows point from this box to a second red box containing the function call `sum(a , b)`. From there, two arrows point to a third red box containing the function signature `sum(int x, int y)`, indicating the transition to the function's local scope.

2.9 Functions

The execution now occurs in the **sum** function.

Here a new variable is declared and the operation **$z = x + y$** is executed.

Since the actual value of **x** and **y** are 5 and 3, **z** will be 8 (5+3).


```
#include <iostream>
using namespace std;

int sum (int x, int y){
    int z;
    z = x + y;
    return(z);
}

int main()
{
    int a, b, result;
    cout << "Please enter two numbers: " << endl;
    cin >> a;
    cin.ignore();
    cin >> b;
    cin.ignore();

    result = sum(a,b);

    cout << "The result of " << a << "+" << b << " is " << result;
    cin.ignore();
}
```



$x = 5; y = 3;$

$z = x + y;$

$z = 5 + 3;$

$z = 8;$

2.9 Functions

The return statement finalizes the function and returns the control back to the caller function (in this case **main**).

The program then resumes execution at the line of code following the function call.


```
#include <iostream>
using namespace std;

int sum (int x, int y){
    int z;
    z = x + y;
    return(z);
}

int main()
{
    int a, b, result;
    cout << "Please enter two numbers: " << endl;
    cin >> a;
    cin.ignore();
    cin >> b;
    cin.ignore();

    result = sum(a,b);

    cout << "The result of " << a << "+" << b << " is " << result;
    cin.ignore();
}
```



The diagram illustrates the execution flow. A large white arrow points from the `return(z);` statement in the `sum` function to a red box containing `z = 8;`. Below this, another red box contains `return(z);`, and a third red box contains `result = 8`. This sequence shows how the value returned by the function is assigned to a local variable and then passed back to the caller.

2.9 Functions

Since the function returned a value (**z**), this will be copied in the variable **result**: the value of the **result** will then be 8.


```
#include <iostream>
using namespace std;

int sum (int x, int y){
    int z;
    z = x + y;
    return(z);
}

int main()
{
    int a, b, result;
    cout << "Please enter two numbers: " << endl;
    cin >> a;
    cin.ignore();
    cin >> b;
    cin.ignore();

    result = sum(a,b);

    cout << "The result of " << a << "+" << b << " is " << result;
    cin.ignore();
}
```



The diagram illustrates the return value of the `sum` function being passed to the `result` variable in the `main` function. A large red arrow points from the `return(z);` statement in the `sum` function to the `result = 8;` assignment in the `main` function. The `result` variable is shown in a red box, and the value `8` is shown in a red box.

z = 8;

return(z);

result = 8

2.9 Functions

This instruction prints the values of the variables. As we can see from the console the result is 8.

```
#include <iostream>
using namespace std;

int sum (int x, int y){

    int z;
    z = x + y;
    return(z);

}

int main()
{
    int a, b, result;
    cout << "Please enter two numbers: " << endl;
    cin >> a;
    cin.ignore();
    cin >> b;
    cin.ignore();

    result = sum(a,b);

    cout << "The result of " << a << "+" << b << " is " << result;

    cin.ignore();
}
```

C:\Dev-Cpp\Samples\function_sum.exe

Please enter two numbers:
5
3
The result of 5+3 is 8

2.9 Functions

In almost any programming language there are two ways in which we can pass arguments to a function.

By value

By reference

2.9 Functions

The first method, **call by value**, copies the value of an argument into a parameter. In this case, changes made to the parameter do not affect the argument.

By default, C++ uses **call by value**; this means that the code in the function does not alter the arguments used by the caller.

2.9 Functions

```
#include <iostream>
using namespace std;

int sum (int x){

    x = 10 + x;
    return(x);

}

int main()
{
    int x = 5;

    cout << "The result of 10 + " << x << " is " << sum(x) << endl;
    cout << "Value of x is still " << x;

    cin.ignore();
}
```

C:\Dev-Cpp\Samples\by_value.exe
The result of 10 + 5 is 15
Value of x is still 5_

Similar to the previous example, in this program, the value of the argument **sum()**, 5, is copied into the parameter **x** (within **sum()**).

When the assignment **x = 10 + x** takes place, only the local variable **x** (within the function) is modified. The variable **x** in the **main** function used to call **sum()** still holds the value of 5.

2.9 Functions

The second method, **call by reference**, passes arguments in a different way. With this method, the **address** of an argument (not the value) is copied into the parameter. Inside the function, the address is used to access the actual argument used in the call, so changes made to the parameter **affect** the argument.

We can create a call by reference by passing a pointer to an argument instead of the argument itself.

2.9 Functions

In the declaration of the function, the type of each parameter is followed by an ampersand sign that specifies that their corresponding arguments are to be passed by reference; this means that we are passing the variable itself and not its value.

```
</>
void swap(int& x, int& y)
{
    int temp;

    temp = *x;
    *x = *y;
    *y = temp;
}
```

2.9 Functions

The **swap** function exchanges the values of the two variables **i** and **j** because we pass the variables and not just their values.

Any modification to local variables in the swap function will have an affect on the variables passed as argument (**&i** and **&j**).

```
#include <iostream>
using namespace std;
void swap(int& x, int& y)
{
    int temp;

    temp = x;
    x = y;
    y = temp;
}

int main ()
{
    int i, j;

    i = 5;
    j = 10;
    cout << "Before swap i is: " << i << " and j is: " << j << endl;

    swap(i, j);

    cout << "After swap i is: " << i << " and j is: " << j;

    cin.ignore();
    return 0;
}
```

C:\Dev-Cpp\Samples\by_reference.exe
Before swap i is: 5 and j is: 10
After swap i is: 10 and j is: 5

2.10

Hera Lab

C++-assisted exploitation



2.10 Lab – C++-assisted exploitation

Let's try to use C++ in order to create simple tools that could be used during your penetration testing activities.

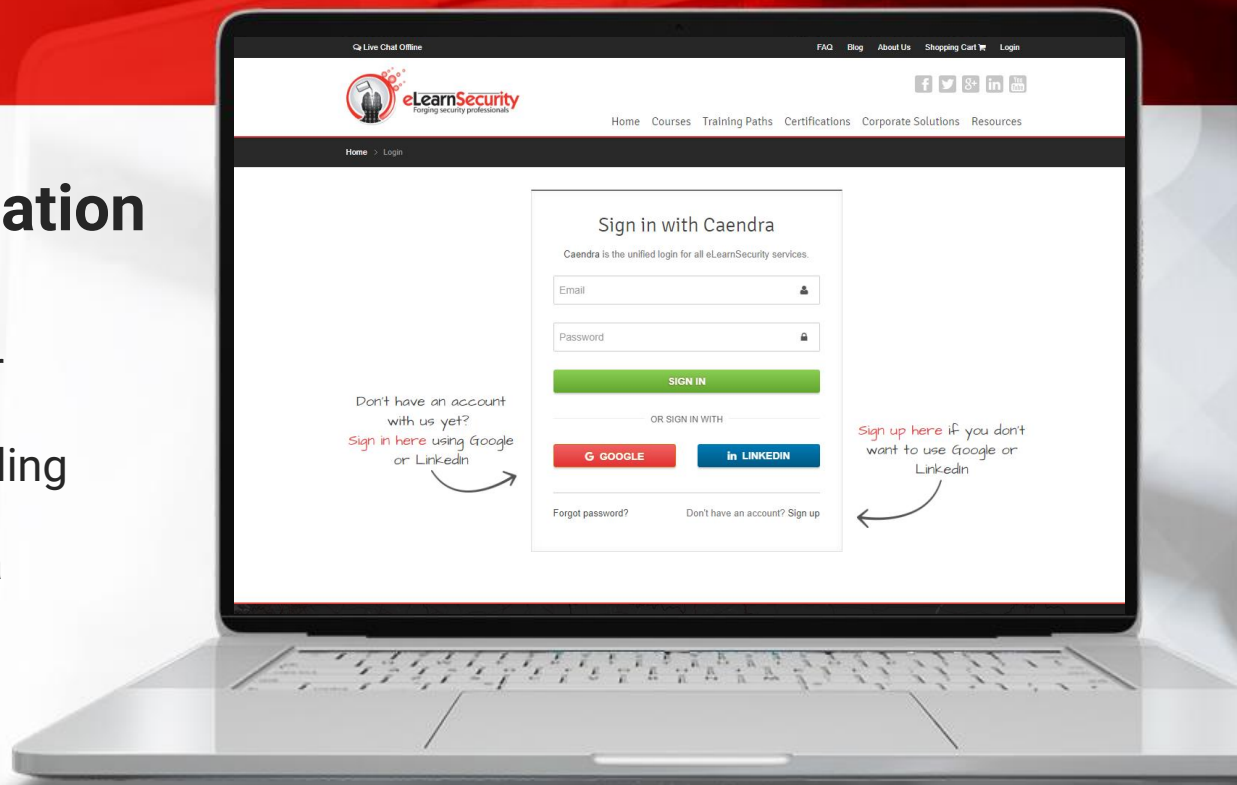


2.10 Lab – C++-assisted exploitation

C++-assisted exploitation

In the lab, you will:

- Create a simple keylogger program
- Create a simple data stealing program
- Exfiltrate stolen data via a network connection



**Labs are only available in Full or Elite Editions of the course. To upgrade, click [HERE](#). To access, go to the course in your members area and click the labs drop-down in the appropriate module line or to the virtual labs tabs on the left navigation.*



References



References

This concludes our basic C++ tutorial. If you want to dig deeper in this programming language, here are some references that you can use:

[C++ tutorial](http://www.cplusplus.com/doc/tutorial/)

<http://www.cplusplus.com/doc/tutorial/>

[The C++ Programming Language \(3rd Edition\)](http://www.amazon.com/The-Programming-Language-3rd-Edition/dp/0201889544)

<http://www.amazon.com/The-Programming-Language-3rd-Edition/dp/0201889544>

[Sams Teach Yourself C++ in One Hour a Day](http://www.amazon.com/Sams-Teach-Yourself-One-Hour/dp/0672335670/)

<http://www.amazon.com/Sams-Teach-Yourself-One-Hour/dp/0672335670/>





C++-assisted exploitation

In the lab, you will:

- Create a simple keylogger program
- Create a simple data stealing program
- Exfiltrate stolen data via a network connection



**Labs are only available in Full or Elite Editions of the course. To upgrade, click [HERE](#). To access, go to the course in your members area and click the labs drop-down in the appropriate module line or to the virtual labs tabs on the left navigation.*