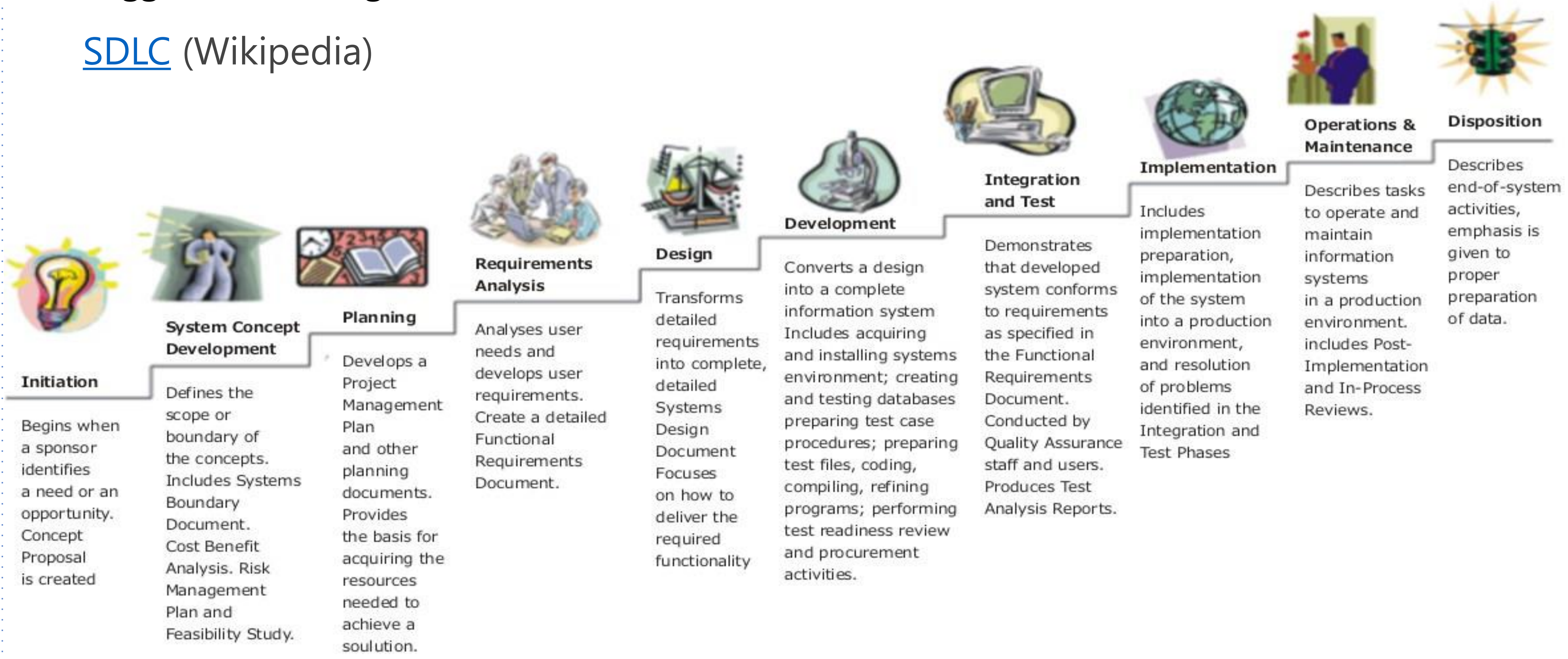
PART 1
BASICS

Systems Development Life Cycle (SDLC) Life-Cycle Phases

Suggested Reading

SDLC (Wikipedia)



Initiation

Begins when a sponsor identifies a need or an opportunity. Concept Proposal is created

System Concept Development

Defines the scope or boundary of the concepts. Includes Systems Boundary Document. Cost Benefit Analysis. Risk Management Plan and Feasibility Study.

Planning

Develops a Project Management Plan and other planning documents. Provides the basis for acquiring the resources needed to achieve a soulution.

Requirements Analysis

Analyses user needs and develops user requirements. Create a detailed Functional Requirements Document.

Design

Transforms detailed requirements into complete, detailed Systems Design Document Focuses on how to deliver the required functionality

Development

Converts a design into a complete information system Includes acquiring and installing systems environment; creating and testing databases preparing test case procedures; preparing test files, coding, compiling, refining programs; performing test readiness review and procurement activities.

Integration and Test

Demonstrates that developed system conforms to requirements as specified in the Functional Requirements Document. Conducted by Quality Assurance staff and users. Produces Test Analysis Reports.

Implementation

Includes implementation preparation, implementation of the system into a production environment, and resolution of problems identified in the Integration and Test Phases

Operations & Maintenance

Describes tasks to operate and maintain information systems in a production environment. includes Post-Implementation and In-Process Reviews.

Disposition

Describes end-of-system activities, emphasis is given to proper preparation of data.

INTRODUCTION

Compiled Languages and C++

Why Use a Language Like C++?

- At its core, a computer is just a processor with some memory, capable of running tiny instructions like "store 5 in memory location 23459."
  - ➢ Why would we express a program as a text file in a programming language, instead of writing processor instructions?
- **C++ is a high-level language**: when you write a program in it, the shorthands are sufficiently expressive that you don't need to worry about the details of processor instructions. C++ does give access to some lower-level functionality than other languages (e.g. memory addresses).

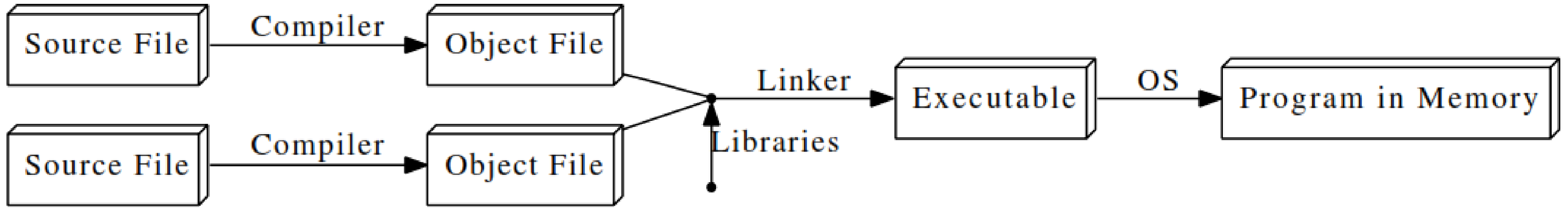
Compiled Languages and C++

The advantages:

- **Conciseness**: programming languages allow us to express common sequences of commands more concisely. C++ provides some especially powerful shorthands.
- **Maintainability**: modifying code is easier when it entails just a few text edits, instead
- of rearranging hundreds of processor instructions. C++ is object oriented (more on that in the next Lectures), which further improves maintainability.
- **Portability**: different processors make different instructions available. Programs written as text can be translated into instructions for many different processors; one of
- C++'s strengths is that it can be used to write programs for nearly any processor.

The compilation process

A program goes from text files (or **source files**) to processor instructions as follows:

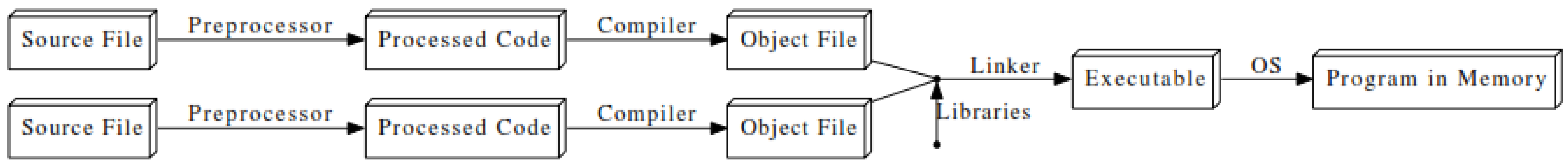


- **Object files** are intermediate files that represent an incomplete copy of the program: each source file only expresses a piece of the program, so when it is compiled into an object file, the object file has some markers indicating which missing pieces it depends on. The linker takes those object files and the compiled libraries of predefined code that they rely on, fills in all the gaps, and spits out the final program, which can then be run by the operating system (OS).

- The **compiler** and **linker** are just regular programs. The step in the compilation process in which the compiler reads the file is called *parsing*.

The compilation process

- **In C++, all these steps are performed ahead of time, before you start running a program. In some languages, they are done during the execution process, which takes time.**
- **This is one of the reasons C++ code runs far faster than code in many more recent languages**.
- C++ actually adds an extra step to the compilation process: the code is run through a **preprocessor**, which applies some modifications to the source code, before being fed to the compiler. Thus, the modified diagram is:

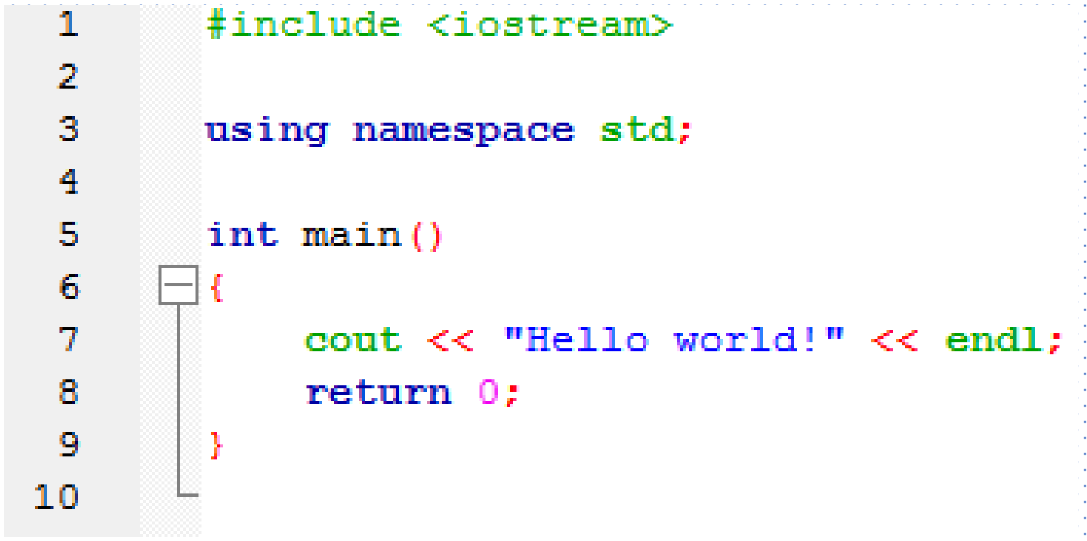


General notes on C++

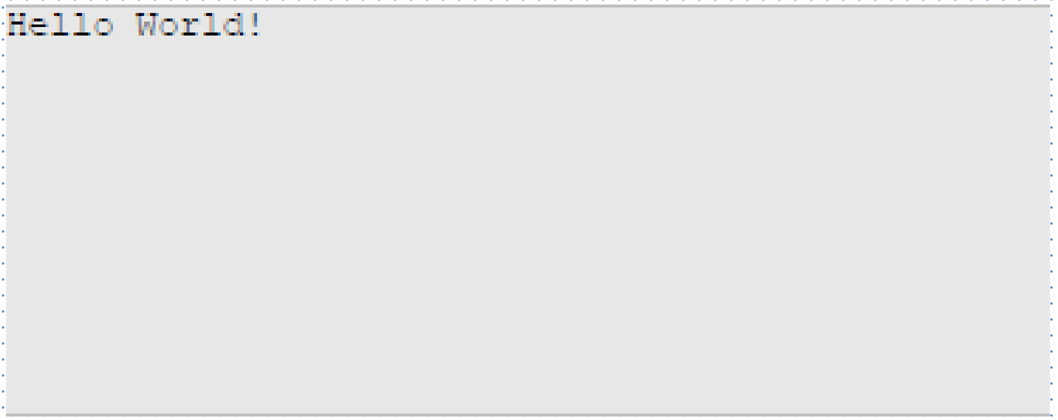
- C++ is immensely popular, particularly for applications that require **speed and/or access** to some low-level features.
- It was created in 1979 by Bjarne Stroustrup, at first as a set of extensions to the C programming language. C++ extends C; our first few lectures will basically be on the C parts of the language.
- Though you can write graphical programs in C++, it is much hairier and less portable than text-based (console) programs. We will be sticking to **console programs** in this course.
- **Everything in C++ is case sensitive:** **someName** is not the same as **SomeName**.

HELLO WORLD

- In the tradition of programmers everywhere, we'll use a "Hello, world!" program as an entry point into the basic features of C++.
- **THE CODE**



```cpp
#include <iostream>

using namespace std;

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

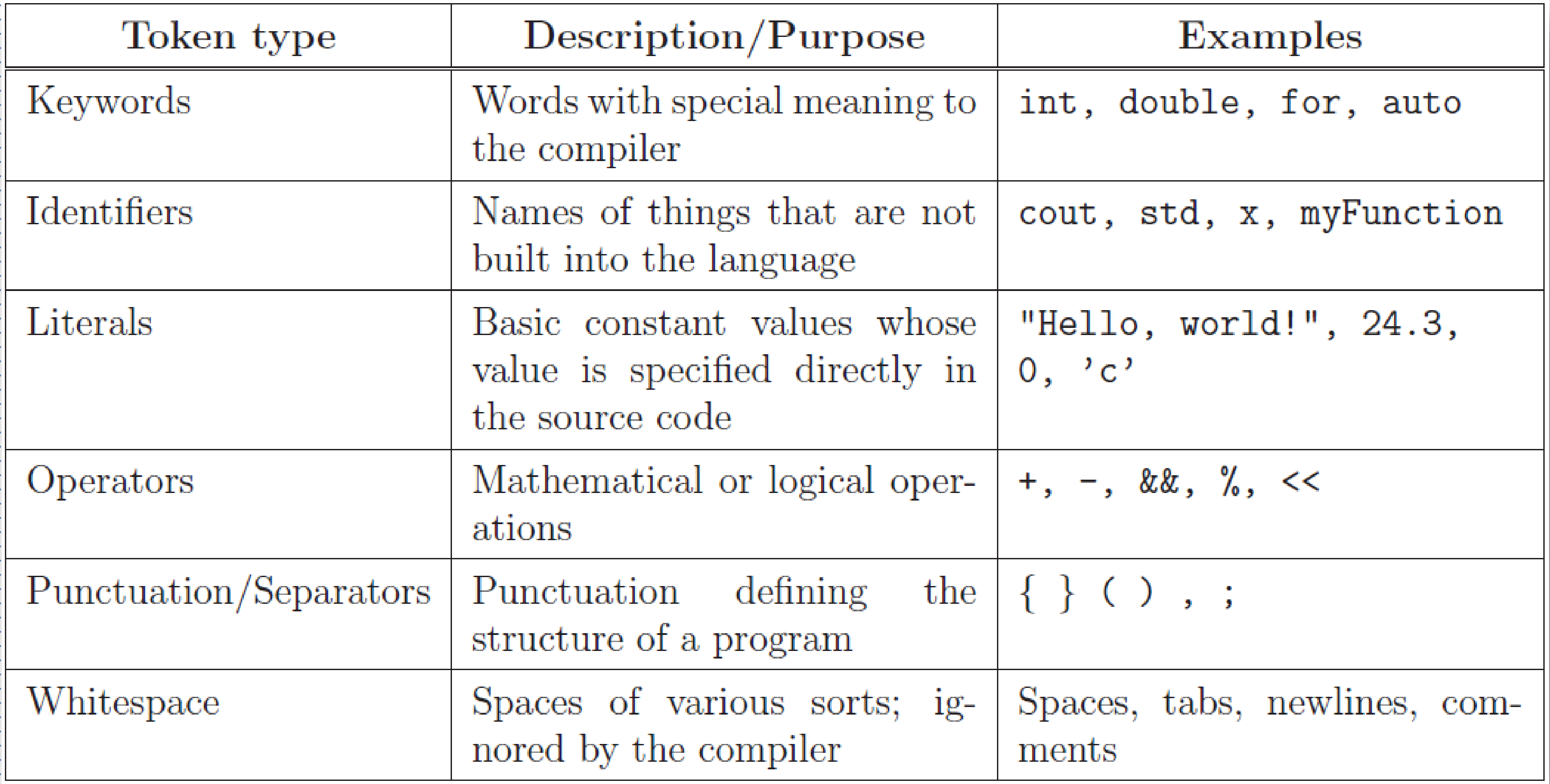


```
Hello World!
```

- The first panel shows the source code for our first program. The second one shows the result of the program once compiled and executed.

HELLO WORLD

- **TOKENS:** *Tokens are the minimals chunk of program that have meaning to the compiler* – the smallest meaningful symbols in the language. Our code displays all 6 kinds of tokens, though the usual use of operators is not present here:



Token type	Description/Purpose	Examples
Keywords	Words with special meaning to the compiler	`int, double, for, auto`
Identifiers	Names of things that are not built into the language	`cout, std, x, myFunction`
Literals	Basic constant values whose value is specified directly in the source code	`"Hello, world!", 24.3, 0, ’c’`
Operators	Mathematical or logical operations	`+, -, &&, %, <<`
Punctuation/Separators	Punctuation defining the structure of a program	`{ } ( ) , ;`
Whitespace	Spaces of various sorts; ignored by the compiler	Spaces, tabs, newlines, comments

HELLO WORLD

LINE BY LINE EXPLANATION:

- // indicates that everything following it until the end of the line is a **comment**: it is ignored by the compiler. Another way to write a comment is to put it between /* and*/ (e.g. x = 1 + /*sneaky comment here*/ 1;). A comment of this form may span multiple lines. Comments exist to explain non-obvious things going on in the code. *Use them: document your code well!*
- Lines beginning with # are **preprocessor commands**, which usually change what code is actually being compiled. **#include** tells the preprocessor to dump in the contents of another file, here the **iostream file**, which defines the procedures for input/output.

HELLO WORLD

LINE BY LINE EXPLANATION:

- **using namespace std;** All the elements of the standard C++ library are declared within what is called a namespace, the namespace with the name *std*. So in order to access its functionality we declare with this expression that we will be using these entities. This line is very frequent in C++ programs that use the standard library.
- **int main() {...}** defines the code that should execute when the program **starts up**. The curly braces represent grouping of multiple commands into a block. *More about this syntax in the next few lectures*.
- **cout <<** : This is the syntax for outputting some piece of text to the screen. *We'll discuss how it works in the next lectures*.

HELLO WORLD

LINE BY LINE EXPLANATION:

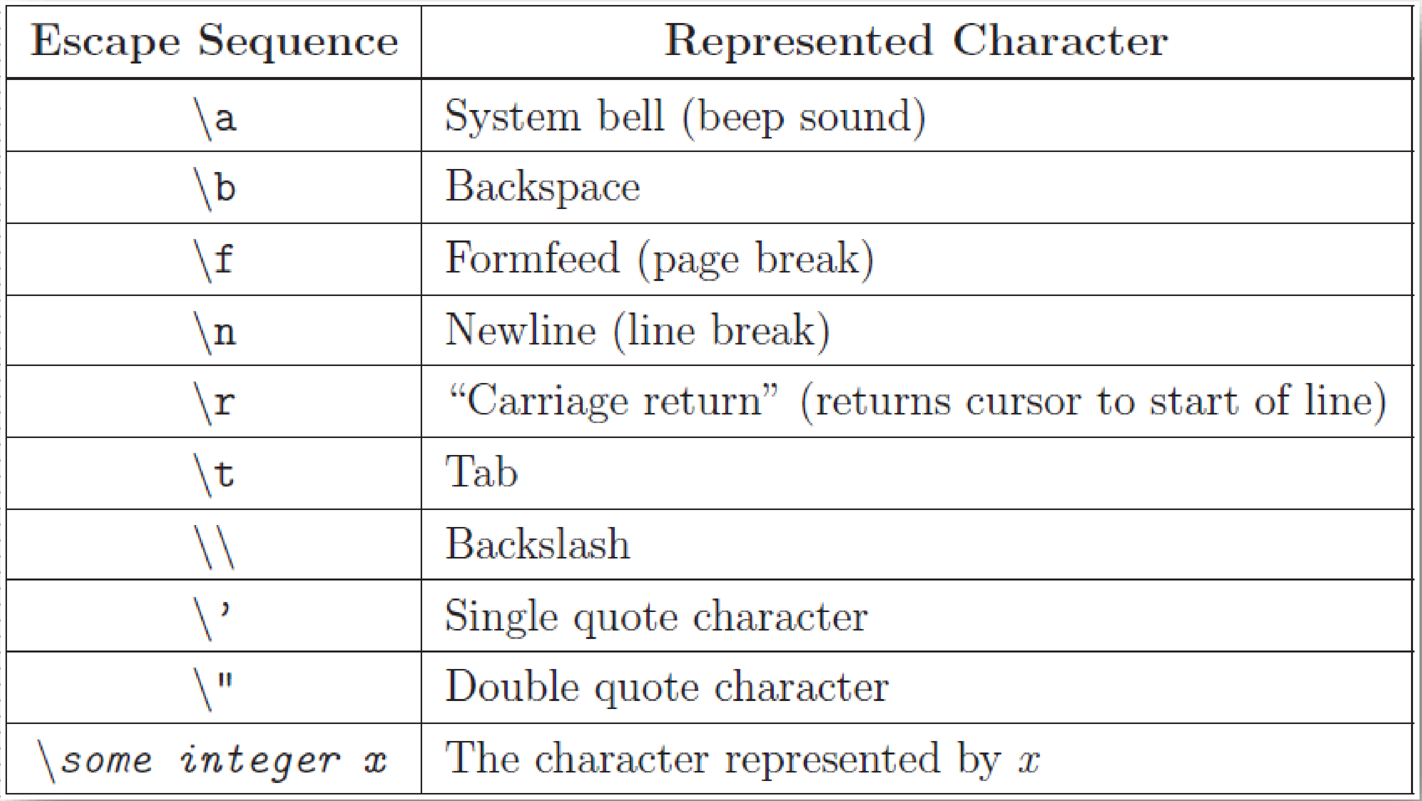
- **Strings**: A sequence of characters such as *Hello, world* is known as a string. A string that is specified explicitly in a program is a string literal.
- **Escape sequences**: The **endl** indicates a newline character. It is an example of an escape sequence – a symbol used to represent a special character in a text literal.
- Here are all the C++ escape sequences which you can include in strings:
- **Note that every statement ends with a semicolon (except preprocessor commands and blocks using {}). This character is used to mark the end of the statement and in fact it must be included at the end of all expression statements in all C++ programs (one of the most common syntax errors is indeed to forget to include some semicolon after a statement).**

HELLO WORLD

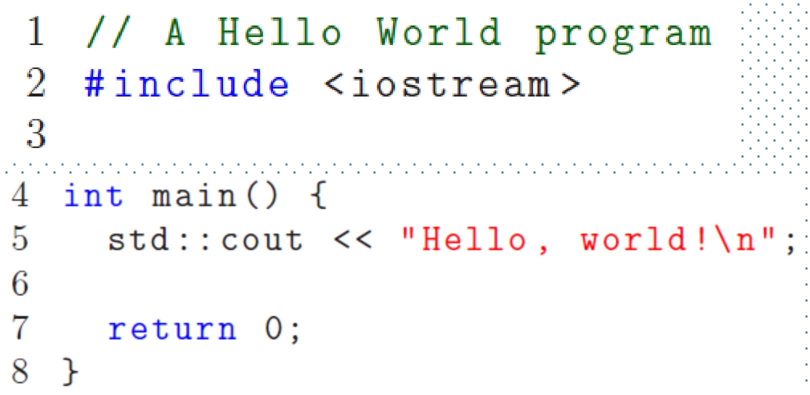
LINE BY LINE EXPLANATION:

- Here are all the C++ escape sequences which you can include in strings:

- **return 0** indicates that the program should tell the operating system it has completed successfully. This syntax will be explained in the context of functions; for now, just include it as the last line in the main block.



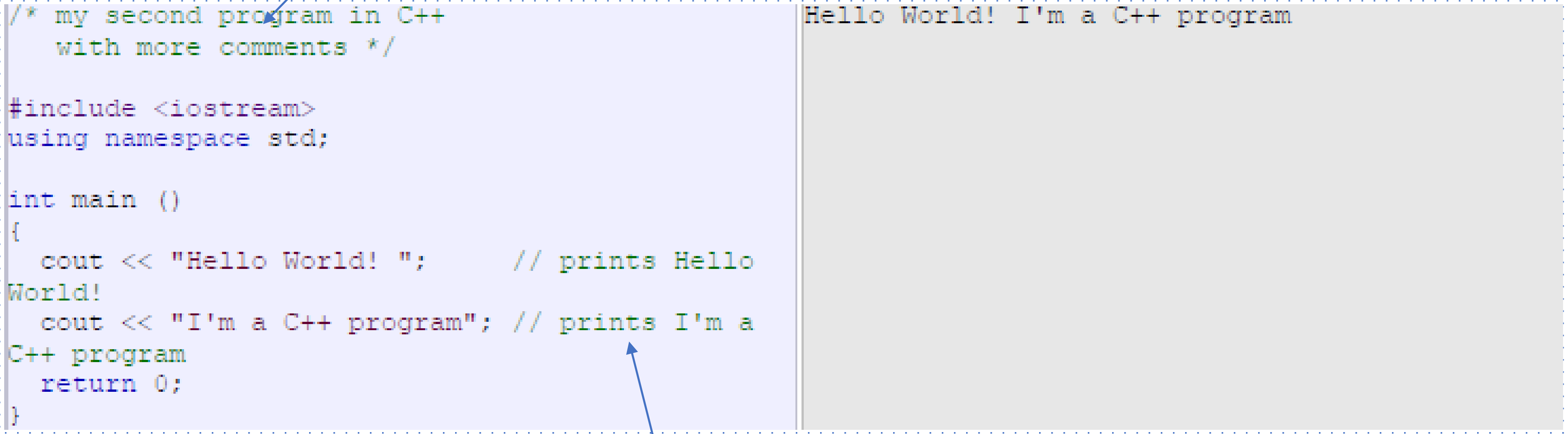
Escape Sequence	Represented Character
`\a`	System bell (beep sound)
`\b`	Backspace
`\f`	Formfeed (page break)
`\n`	Newline (line break)
`\r`	"Carriage return" (returns cursor to start of line)
`\t`	Tab
`\\`	Backslash
`\'`	Single quote character
`\"`	Double quote character
some integer x	The character represented by x



```cpp
// A Hello World program
#include <iostream>

int main() {
  std::cout << "Hello, world!\n";

  return 0;
}
```

HELLO WORLD

Generally used for long comments



```
/* my second program in C++
   with more comments */

#include <iostream>
using namespace std;

int main ()
{
  cout << "Hello World! ";     // prints Hello
World!
  cout << "I'm a C++ program"; // prints I'm a
C++ program
  return 0;
}
```

```
Hello World! I'm a C++ program
```

Generally used for short comments

Basic language features

- So far our program doesn’t do very much. Let’s tweak it in various ways to demonstrate some more interesting constructs.

VALUES AND STATEMENTS

- A **statement** is a unit of code that does something – a basic building block of a program.
- An **expression** is a statement that has a value – for instance, a number, a string, the sum of two numbers, etc. 4 + 2, x - 1, and "Hello, world!\n" are all expressions.
- Not every statement is an expression. It makes no sense to talk about the value of an #include statement, for instance.

Basic language features

OPERATORS

- We can perform arithmetic calculations with operators. Operators act on expressions to form a new expression. For example, we could replace "Hello, world!\n" with (4 + 2) / 3, which would cause the program to print the number 2. In this case, the + operator acts on the expressions 4 and 2 (**its operands**).

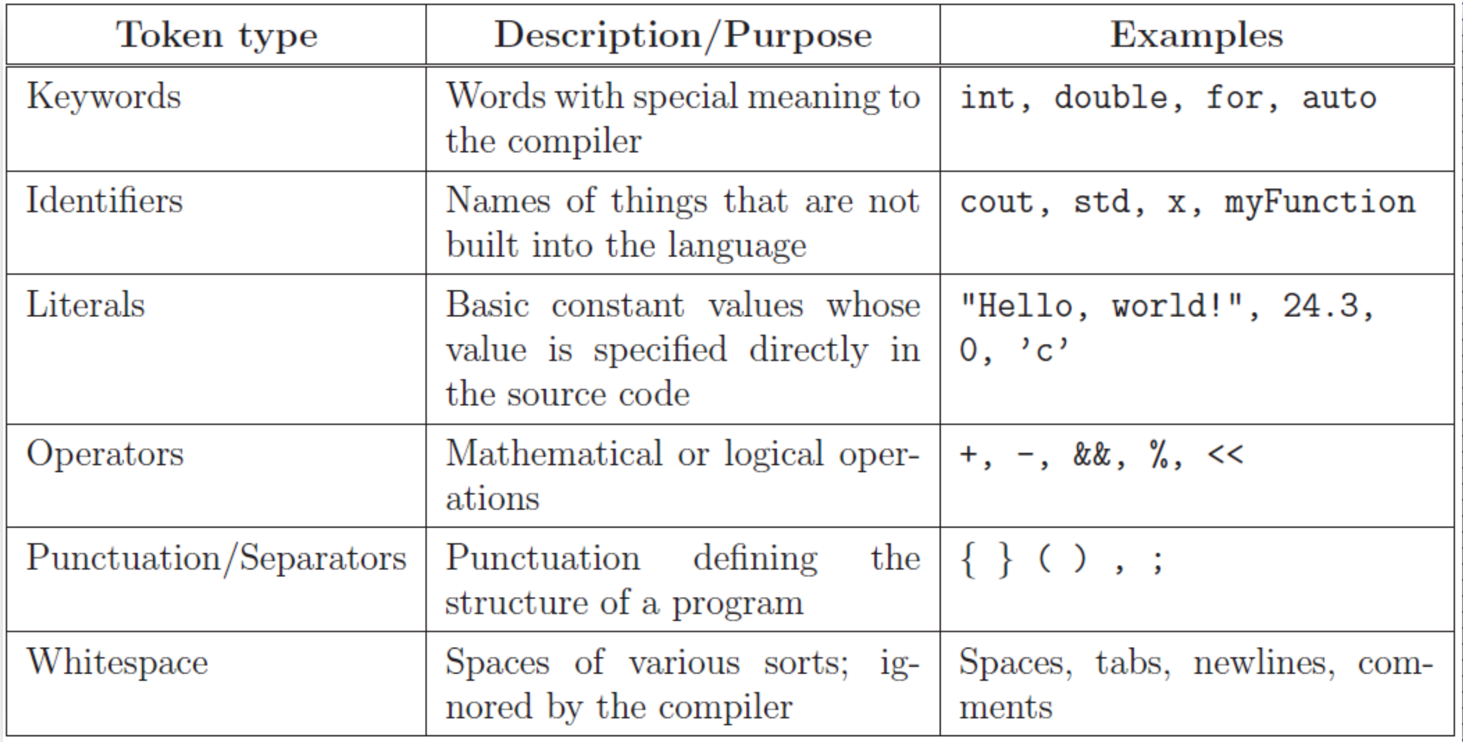
Operator types:

- **Mathematical**: +, -, *, /, and parentheses have their usual mathematical meanings, including using - for negation. % (the modulus operator) takes the remainder of two numbers: 6 % 5 evaluates to 1.
- **Logical**: used for “and,” “or,” and so on. *More on those in the next lecture.*
- **Bitwise**: used to manipulate the binary representations of numbers. We will not focus on these.

Basic language features

DATA TYPES

- Every expression has a type – a formal description of what kind of data its value is. For instance, 0 is an integer, 3.142 is a floating-point (decimal) number, and "Hello, world!\n"is a string value (a sequence of characters).
- Data of different types take a different amounts of memory to store. Here are the built-in datatypes we will use most often:



Token type	Description/Purpose	Examples
Keywords	Words with special meaning to the compiler	`int, double, for, auto`
Identifiers	Names of things that are not built into the language	`cout, std, x, myFunction`
Literals	Basic constant values whose value is specified directly in the source code	`"Hello, world!", 24.3, 0, 'c'`
Operators	Mathematical or logical operations	`+, -, &&, %, <<`
Punctuation/Separators	Punctuation defining the structure of a program	`{ } ( ) , ;`
Whitespace	Spaces of various sorts; ignored by the compiler	Spaces, tabs, newlines, comments

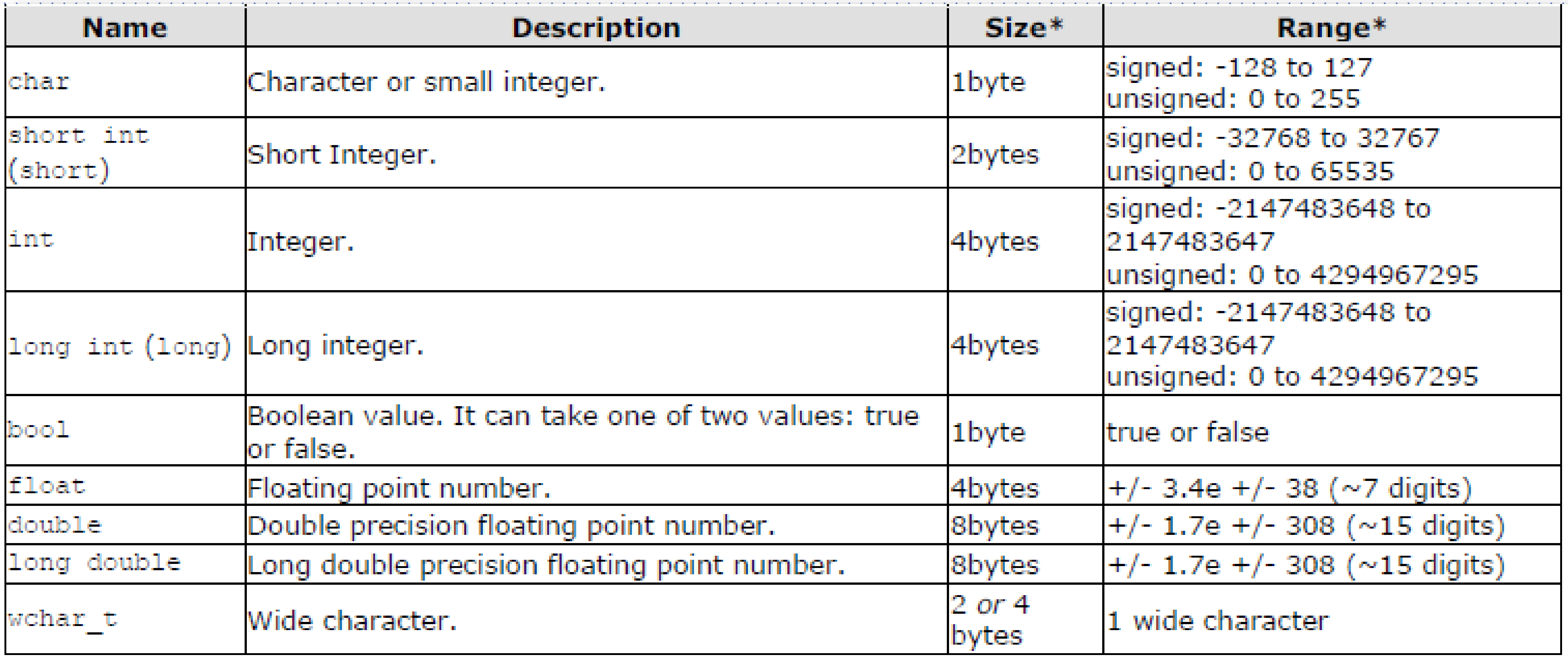
Basic language features

DATA TYPES

- A signed integer is one that can represent a negative number; an unsigned integer will never be interpreted as negative, so it can represent a wider range of positive numbers.
- Most compilers assume signed if unspecified.
- There are actually 3 integer types: short, int, and long, in non-decreasing order of size (int is usually a synonym for one of the other two). You generally don’t need to worry about which kind to use unless you’re worried about memory usage or you’re using really huge numbers. The same goes for the 3 floating point types, float, double, and long double, which are in non-decreasing order of precision (there is usually some imprecision in representing real numbers on a computer).

Basic language features

DATA TYPES



Name	Description	Size*	Range*
`char`	Character or small integer.	1byte	signed: -128 to 127 unsigned: 0 to 255
`short int (short)`	Short Integer.	2bytes	signed: -32768 to 32767 unsigned: 0 to 65535
`int`	Integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
`long int (long)`	Long integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
`bool`	Boolean value. It can take one of two values: true or false.	1byte	true or false
`float`	Floating point number.	4bytes	+/- 3.4e +/- 38 (~7 digits)
`double`	Double precision floating point number.	8bytes	+/- 1.7e +/- 308 (~15 digits)
`long double`	Long double precision floating point number.	8bytes	+/- 1.7e +/- 308 (~15 digits)
`wchar_t`	Wide character.	2 *or* 4 bytes	1 wide character

Basic language features

DATA TYPES

- An operation can only be performed on compatible types. You can add 34 and 3, but you can’t take the remainder of an integer and a floating-point number.
- *An operator also normally produces a value of the same type as its operands; thus, 1 / 4 evaluates to 0 because with two integer operands, / truncates the result to an integer. To get 0.25, you’d need to write something like 1 / 4.0.*
- A text **string**, for reasons we will learn later, has the type **char ***

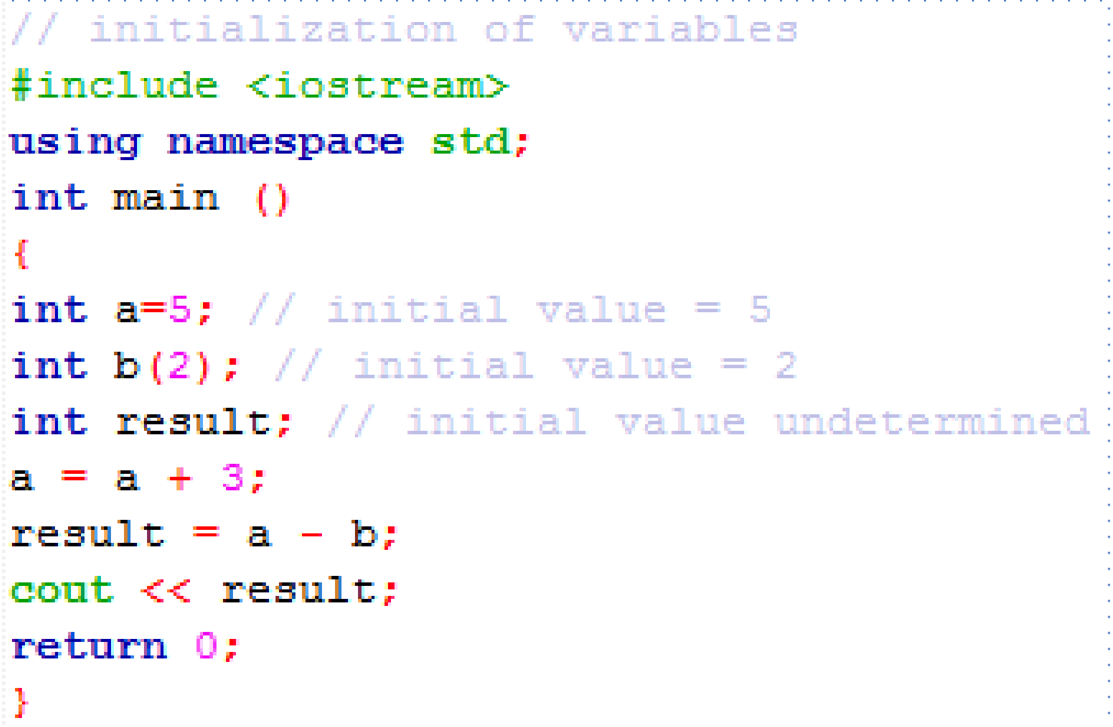
Summary:

- We can distinguish two types of programs: **graphical programs** (**GUI**) and **console programs**. We shall first begin with console programs, since they are simpler.
- A program always contain the function main(): it's the starting point.
- The `cout` directive allows us to paste a text in the console.
- We can add comments to our source code to explain its functioning, they can take the form `//comment` or `/*comment*/.`

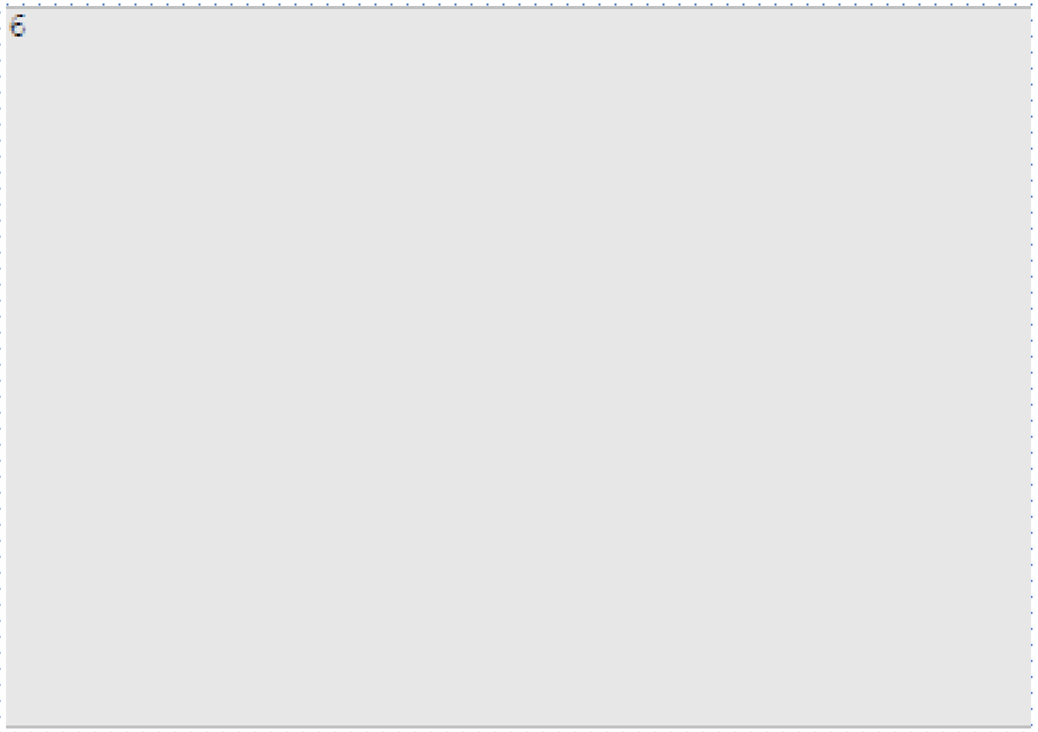
Variables

- We might want to give a value a name so we can refer to it later. We do this using variables.
- A variable is a named location in memory.
- When declaring a regular local variable, its value is by default undetermined. But you may want a variable to store a concrete value at the same moment that it is declared. In order to do that, you can **initialize** the variable. There are **two ways** to do this in C++:
- The first one, known as c-like**, type identifier = initial_value ;**
- The other way to initialize variables, known as constructor initialization, is done by enclosing the initial value between parentheses (()): **type identifier (initial_value) ;**

Variables



```
// initialization of variables
#include <iostream>
using namespace std;
int main ()
{
int a=5; // initial value = 5
int b(2); // initial value = 2
int result; // initial value undetermined
a = a + 3;
result = a - b;
cout << result;
return 0;
}
```

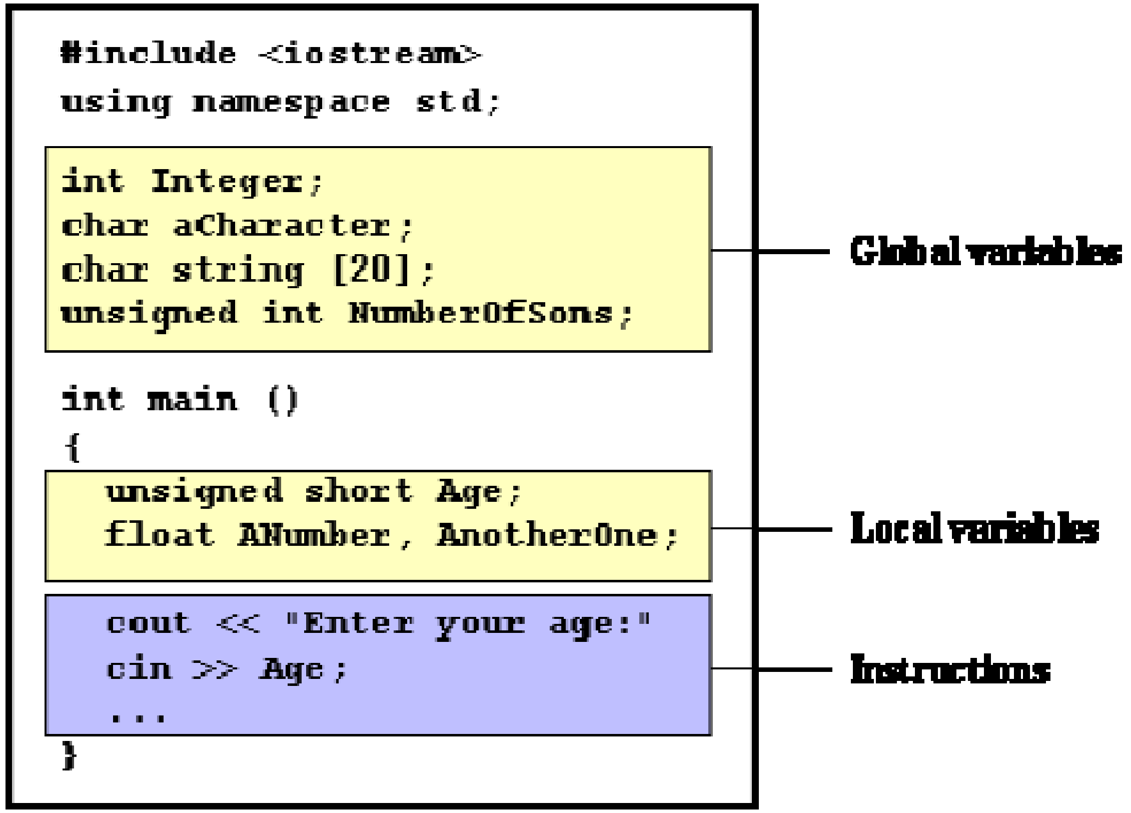


```
6
```

For readability purposes, it is important to use variable names that well describe their content e.g. it is preferable to use a variable name such as `userAge` *rather than using* **myVar** *or* **variable1** *(though for the compiler, it doesn't make any difference but it shall help you and the people working with you in the same program)*

Variables

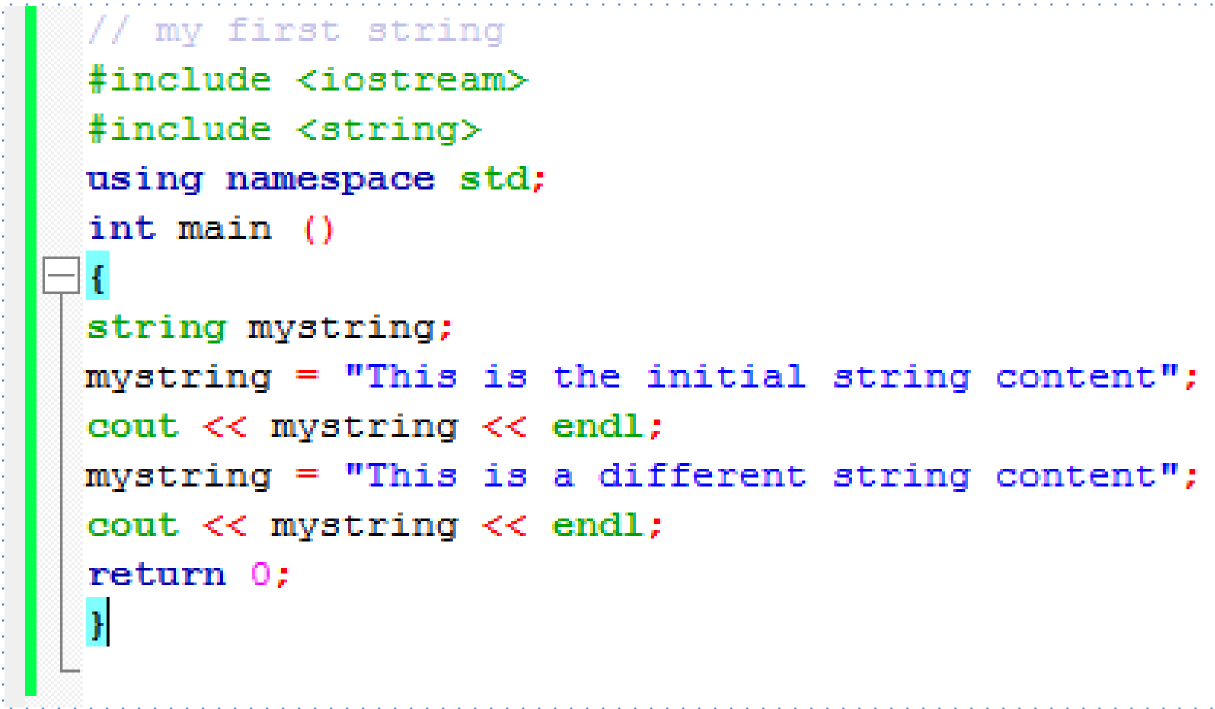
- All the variables that we intend to use in a program must have been declared with its type specifier in an earlier point in the code, like we did in the previous code at the beginning of the body of the function main when we declared that a, b, and result were of type int.
- A variable can be either of **global** or **local** scope. A global variable is a variable declared in the main body of the source code, outside all functions, while a local variable is one declared within the body of a function or a block.



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

int Integer;
char aCharacter;
char string [20];
unsigned int NumberOfSons;

int main ()
{
  unsigned short Age;
  float ANumber, AnotherOne;

  cout << "Enter your age:"
  cin >> Age;
  ...
}
```

Global variables can be referred from anywhere in the code, even inside functions, whenever it is after its declaration.

Variables: case of strings

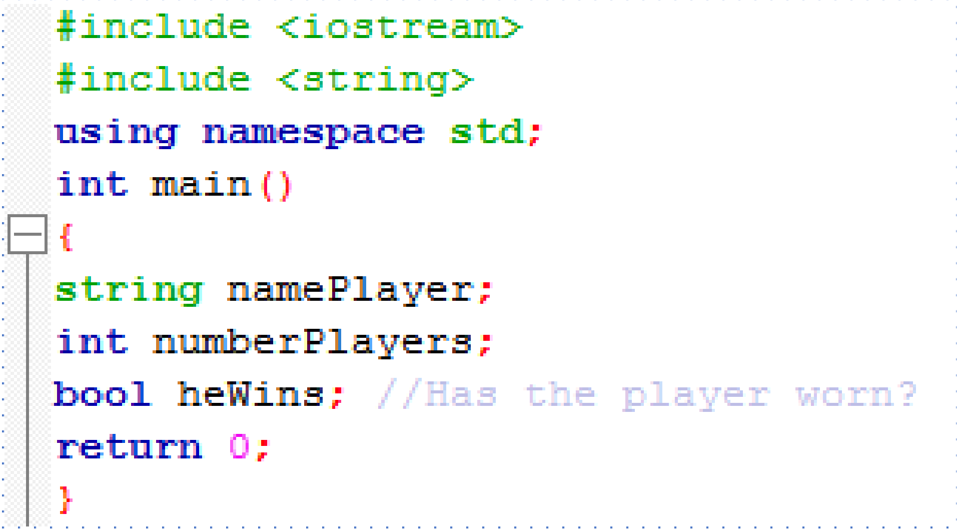


```
// my first string
#include <iostream>
#include <string>
using namespace std;
int main ()
{
string mystring;
mystring = "This is the initial string content";
cout << mystring << endl;
mystring = "This is a different string content";
cout << mystring << endl;
return 0;
}
```

strings can be initialized with any valid string literal just like numerical type variables can be initialized to any valid numerical literal. **Both initialization formats are valid with strings:**

```
string mystring = "This is a string";
string mystring ("This is a string");
```

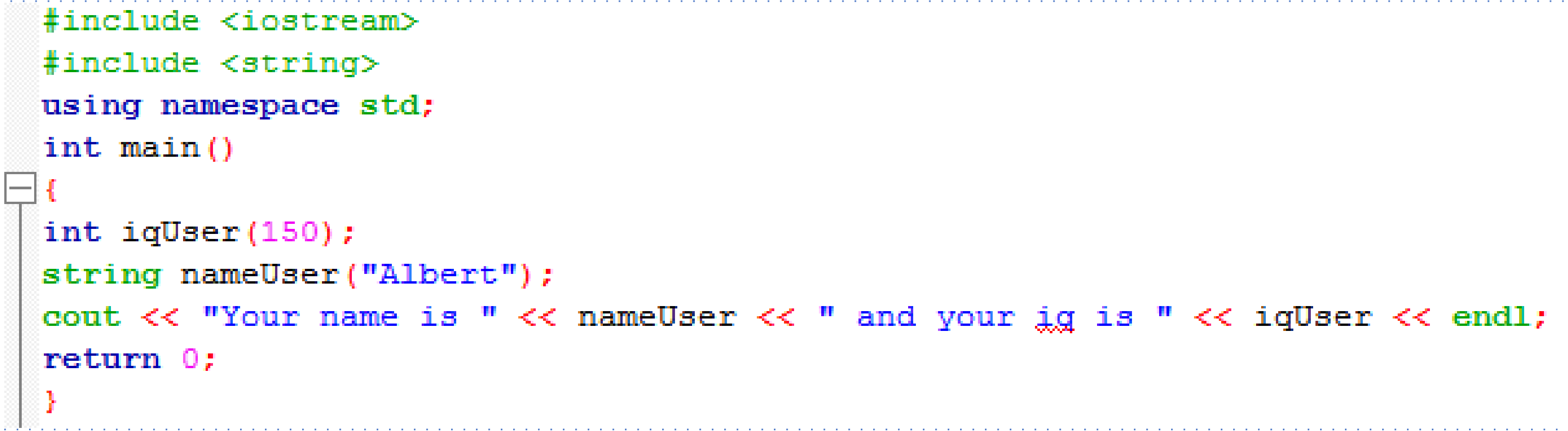
Variables: case of strings



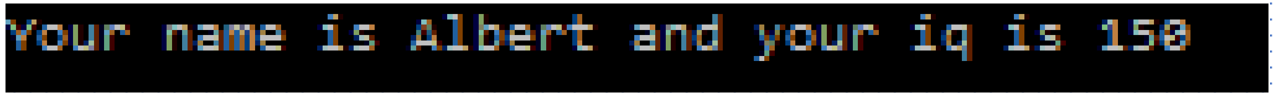
```
#include <iostream>
#include <string>
using namespace std;
int main()
{
string namePlayer;
int numberPlayers;
bool heWins; //Has the player worn?
return 0;
}
```

One can declare strings without initializing. But care must be taken not to write: ~~`namePlayer()`~~ **instead of** `namePlayer`**.**

Variables: case of strings



```cpp
#include <iostream>
#include <string>
using namespace std;
int main()
{
int iqUser(150);
string nameUser("Albert");
cout << "Your name is " << nameUser << " and your iq is " << iqUser << endl;
return 0;
}
```



```
Your name is Albert and your iq is 150
```

Summary:

- A variable is an information stored in memory.
- There exist different types of variables depending on the nature of the stored information: `int`, `char`, `bool`,…
- A variable must be declared before being used. e.g. `int ageUser(16);`
- The value of a variable can be printed out at any time via `cout`.

Constants

Constants are expressions with a fixed value.

Literals are used to express particular values within the source code of a program. We have already used these previously to give concrete values to variables or to express messages we wanted our programs to print out, for example, when we wrote: a = 5; the 5 in this piece of code was a literal constant.

Literal constants can be divided in Integer Numerals, Floating-Point Numerals, Characters, Strings and Boolean Values.

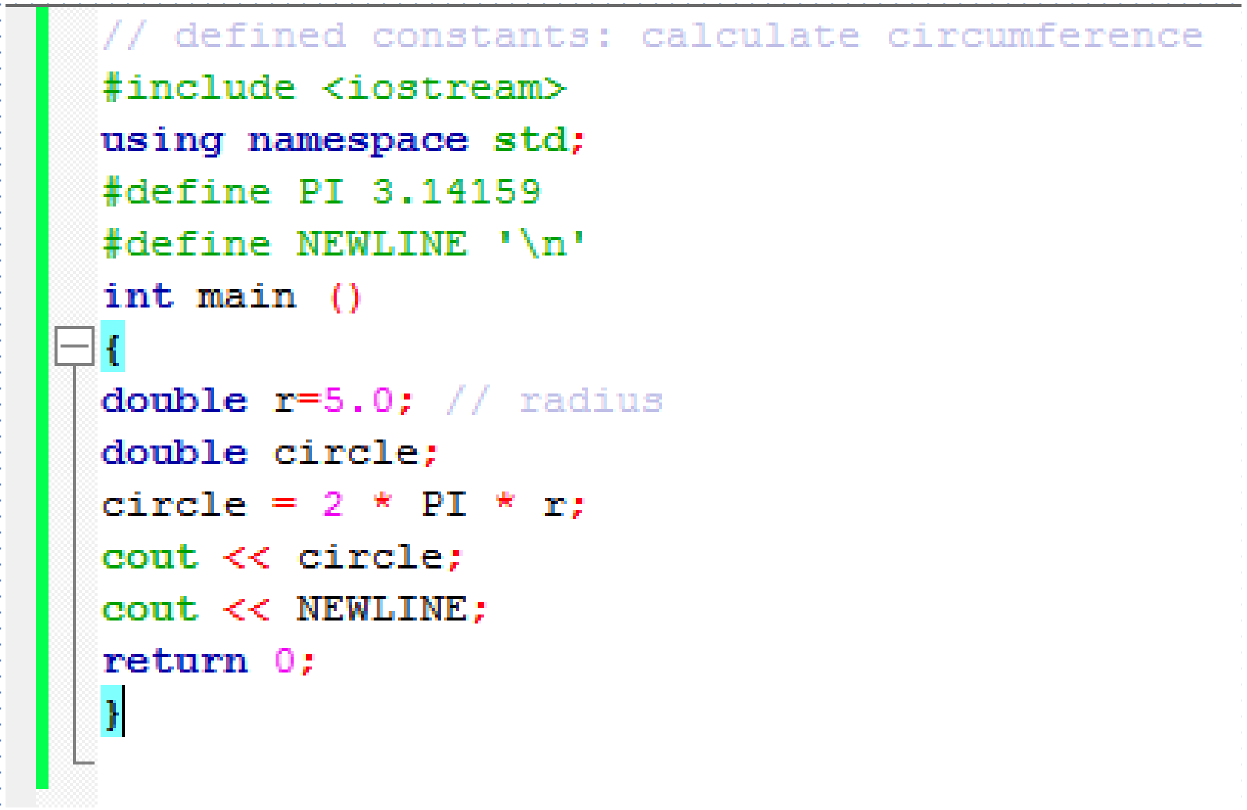
```
75          // decimal
0113        // octal
0x4b        // hexadecimal
75          // int
75u         // unsigned int
75l         // long
75ul        // unsigned long
3.14159     // 3.14159
6.02e23     // 6.02 x 10^23
1.6e-19     // 1.6 x 10^-19
3.0         // 3.0
```

With the const prefix you can declare constants with a specific type in the same way as you would do with a variable:

```
const int pathwidth = 100;
const char tabulator = '\t';
```

Constants

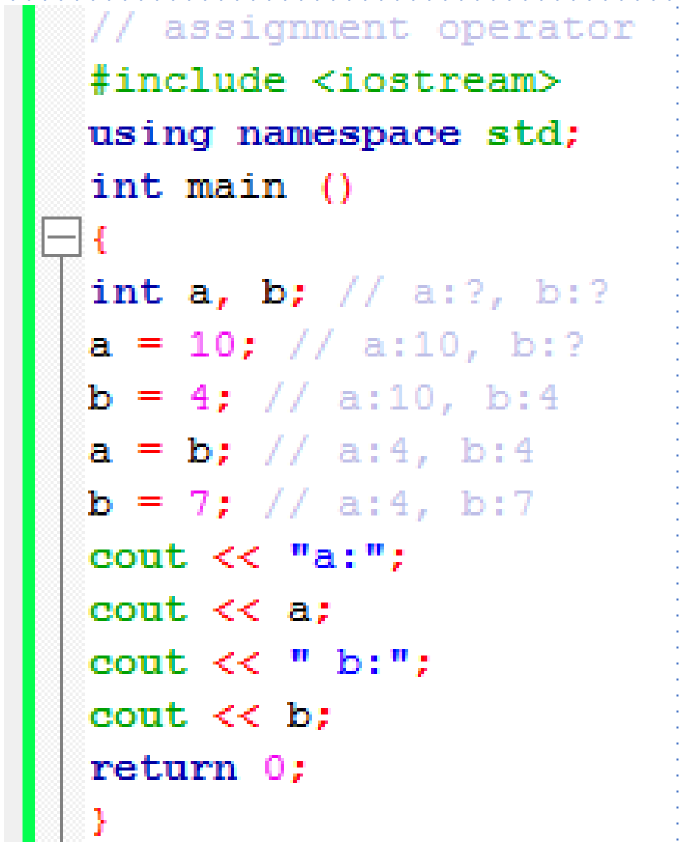
You can define your own names for constants that you use very often without having to resort to memoryconsuming variables, simply by using the #define preprocessor directive. Its format is: **#define identifier value**



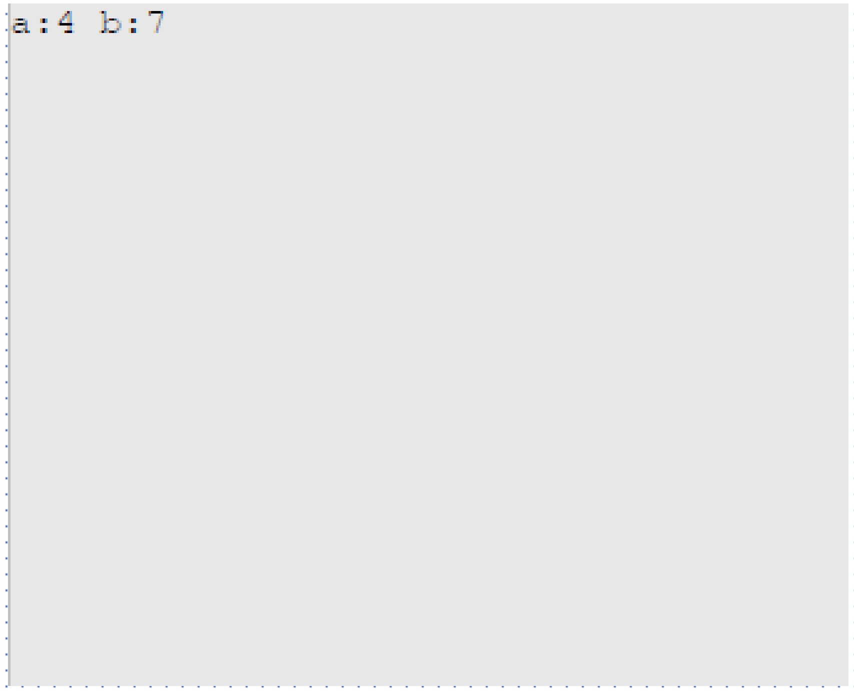
```
// defined constants: calculate circumference
#include <iostream>
using namespace std;
#define PI 3.14159
#define NEWLINE '\n'
int main ()
{
double r=5.0; // radius
double circle;
circle = 2 * PI * r;
cout << circle;
cout << NEWLINE;
return 0;
}
```

The #define directive is not a C++ statement but a directive for the preprocessor; therefore it assumes the entire line as the directive and does not require a semicolon (;) at its end. If you append a semicolon character (;) at the end, it will also be appended in all occurrences within the body of the program that the preprocessor replaces.

Operators



```
// assignment operator
#include <iostream>
using namespace std;
int main ()
{
int a, b; // a:?, b:?
a = 10; // a:10, b:?
b = 4; // a:10, b:4
a = b; // a:4, b:4
b = 7; // a:4, b:7
cout << "a:";
cout << a;
cout << " b:";
cout << b;
return 0;
}
```



```
a:4 b:7
```

Operators

a = 2 + (b = 5);

is equivalent to:

b = 5;a = 2 + b;

that means: first assign 5 to variable b and then assign to a the value 2 plus the result of the previous assignment of b (i.e. 5), leaving a with a final value of 7.

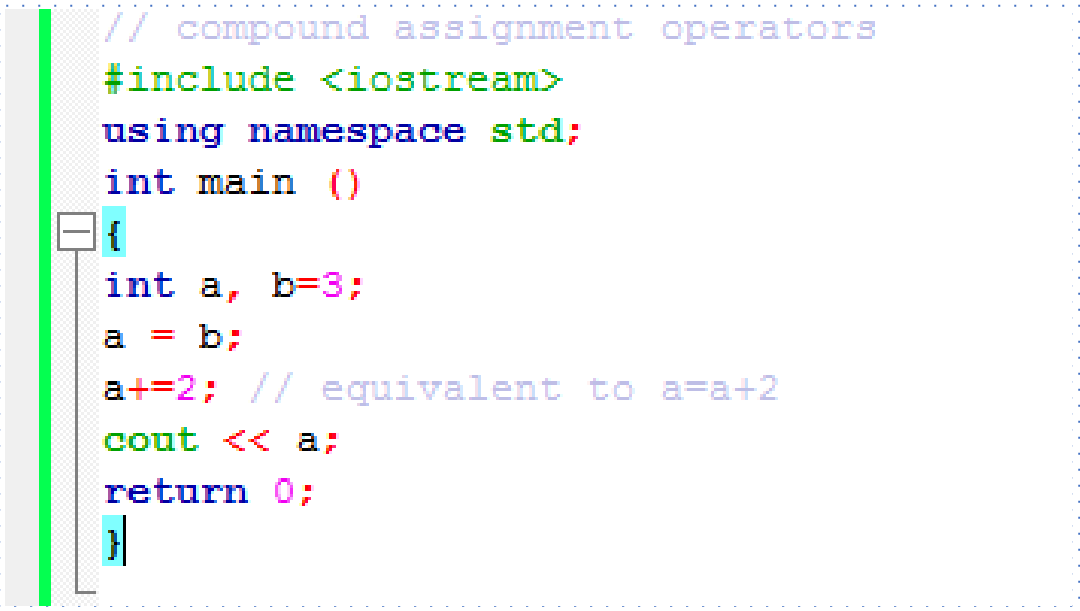
The following expression is also valid in C++:

a = b = c = 5;

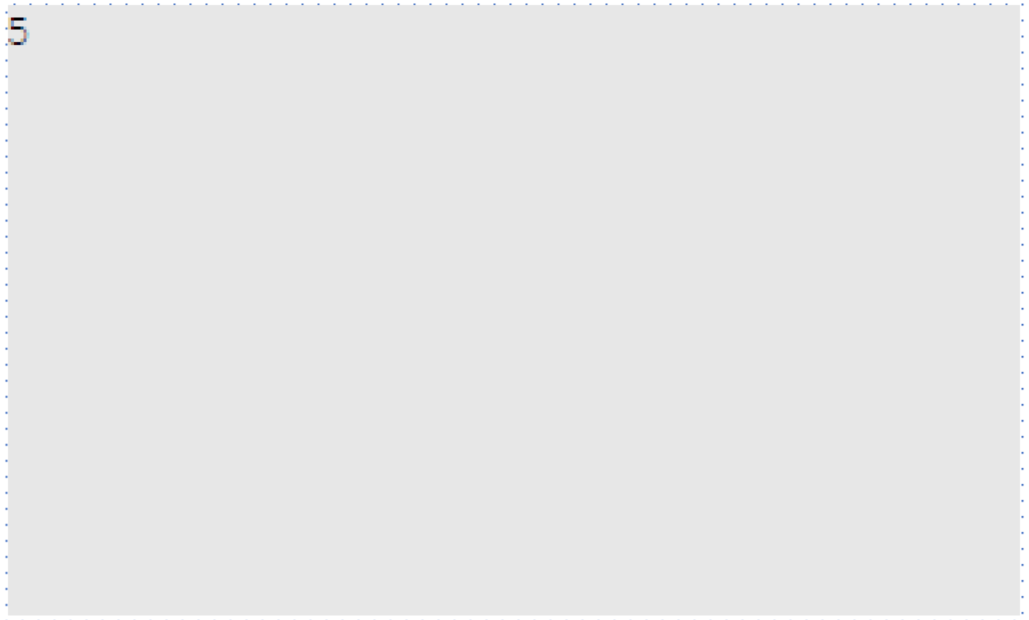
It assigns 5 to the all the three variables: a, b and c.

expression	is equivalent to
`value += increase;`	`value = value + increase;`
`a -= 5;`	`a = a - 5;`
`a /= b;`	`a = a / b;`
`price *= units + 1;`	`price = price * (units + 1);`

Operators



```
// compound assignment operators
#include <iostream>
using namespace std;
int main ()
{
int a, b=3;
a = b;
a+=2; // equivalent to a=a+2
cout << a;
return 0;
}
```



```
5
```

Shortening even more some expressions, the increase operator (++) and the decrease operator (--) increase or
reduce by one the value stored in a variable. They are equivalent to +=1 and to -=1, respectively. Thus:

c++;

c+=1;

c=c+1; are all equivalent in its functionality: the three of them increase by one the value of c.

Operators

In the case that the increase operator is used as a prefix (++a) the value is increased before the result of the expression is evaluated and therefore the increased value is considered in the outer expression; in case that it is used as a suffix (a++) the value stored in a is increased after being evaluated and therefore the value stored before the increase operation is evaluated in the outer expression.

Example 1	Example 2

`B=3;`
`A=++B;`
`// A contains 4, B contains 4`
	`B=3;`
	`A=B++;`
	`// A contains 3, B contains 4`

==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

a	b	a && b
true	true	true
true	false	false
false	true	false
false	false	false

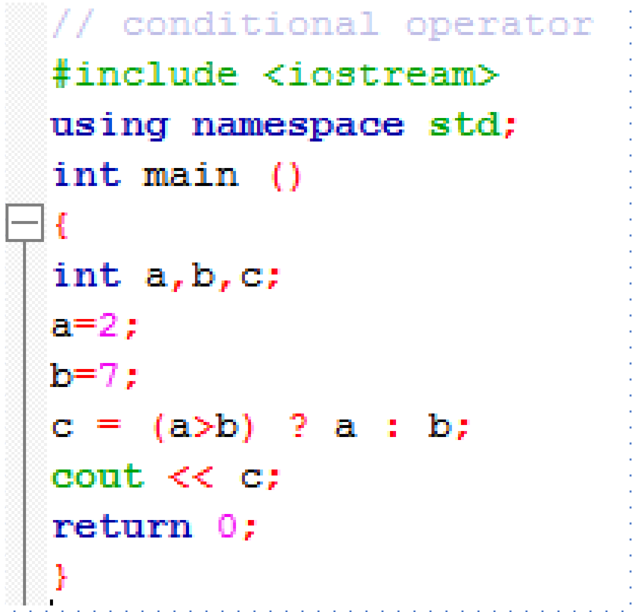
a	b	a || b
true	true	true
true	false	true
false	true	true
false	false	false

Operators

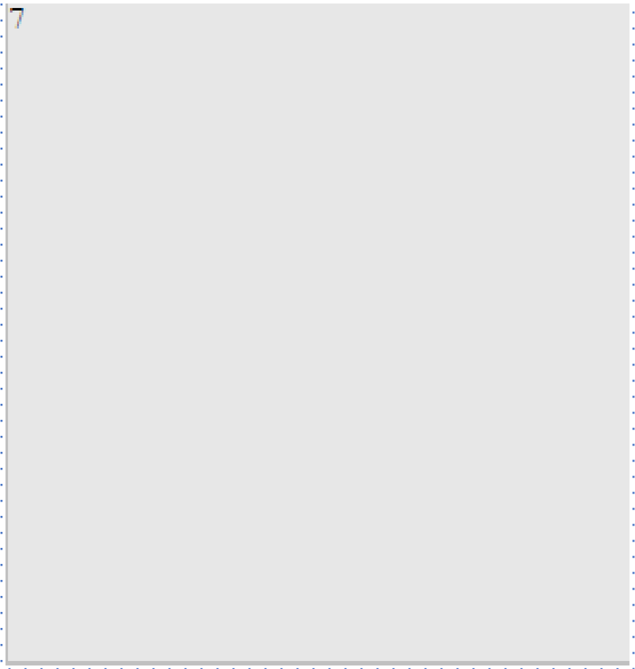
```
( (5 == 5) && (3 > 6) )  // evaluates to false ( true && false ).
( (5 == 5) || (3 > 6) )  // evaluates to true ( true || false ).
```

The conditional operator evaluates an expression returning a value if that expression is true and a different one if the expression is evaluated as false. Its format is: **condition ? result1 : result2**

If condition is true the expression will return result1, if it is not it will return result2.



```
// conditional operator
#include <iostream>
using namespace std;
int main ()
{
int a,b,c;
a=2;
b=7;
c = (a>b) ? a : b;
cout << c;
return 0;
}
```

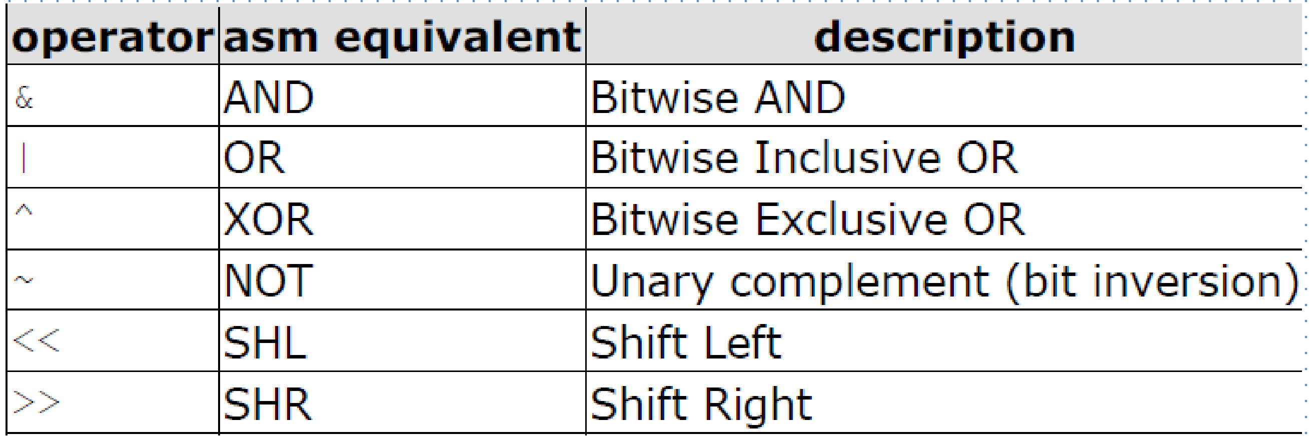


```
7
```

`a = (b=3, b+2);` **returns 5**

Operators

Bitwise operators modify variables considering the bit patterns that represent the values they store.



operator	asm equivalent	description
&	AND	Bitwise AND
|	OR	Bitwise Inclusive OR
^	XOR	Bitwise Exclusive OR
~	NOT	Unary complement (bit inversion)
<<	SHL	Shift Left
>>	SHR	Shift Right

Operators

Type casting operators allow you to convert a datum of a given type to another.

```
int i;
float f = 3.14;
i = (int) f;
```

The previous code converts the float number 3.14 to an integer value (3), the remainder is lost. Here, the typecasting operator was (int). Another way is to:

```
i = int ( f );
```

sizeof(): This operator accepts one parameter, which can be either a type or a variable itself and returns the size in bytes of that type or object:

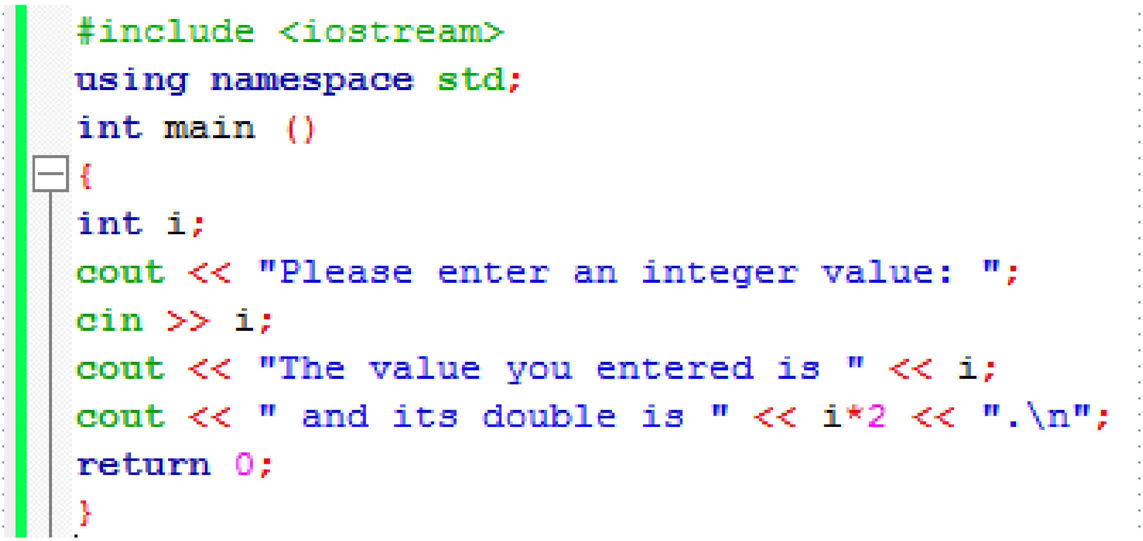
```
a = sizeof (char);
```

This will assign the value 1 to a because char is a one-byte long type.

The value returned by **sizeof** is a constant, so it is always determined before program execution.

Inputs

The **standard input** device is usually the keyboard. Handling the standard input in C++ is done by applying the overloaded operator of extraction (>>) on the cin stream.



```
#include <iostream>
using namespace std;
int main ()
{
int i;
cout << "Please enter an integer value: ";
cin >> i;
cout << "The value you entered is " << i;
cout << " and its double is " << i*2 << ".\n";
return 0;
}
```

You can also use cin to request more than one datum input from the user:

cin >> a >> b;

is equivalent to:

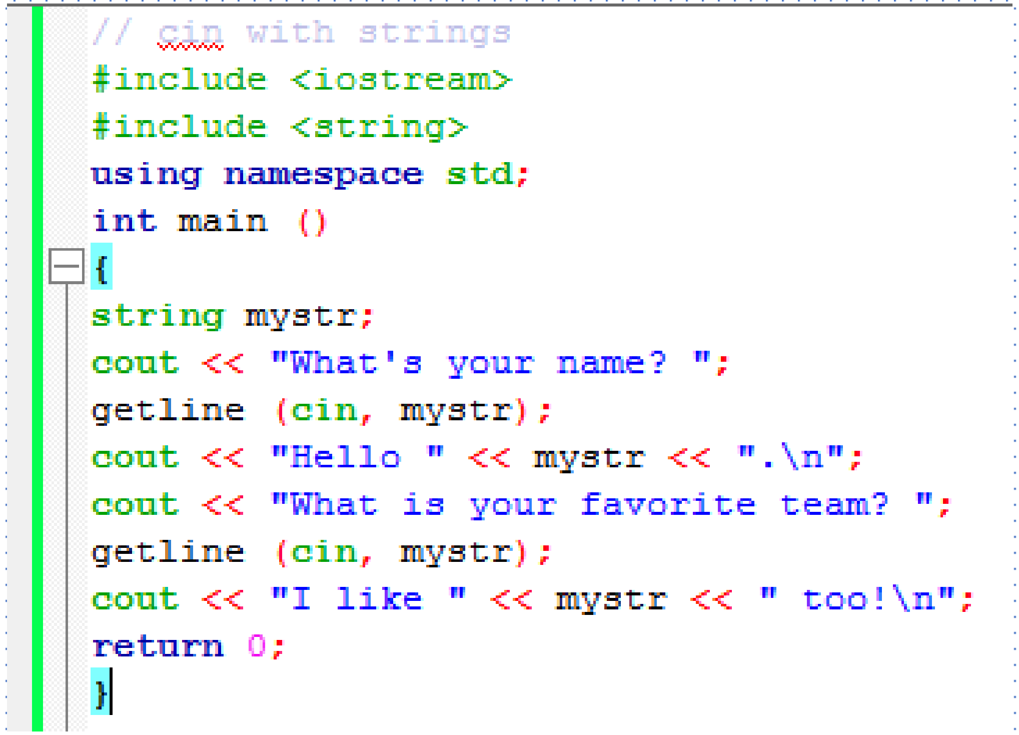
cin >> a;

cin >> b;

In both cases the user must give two data, one for variable a and another one for variable b that may be separated by any valid blank separator: a space, a tab character or a newline.

Inputs

In order to get entire lines, we can use the function **getline**, which is the more recommendable way to get user input with cin:

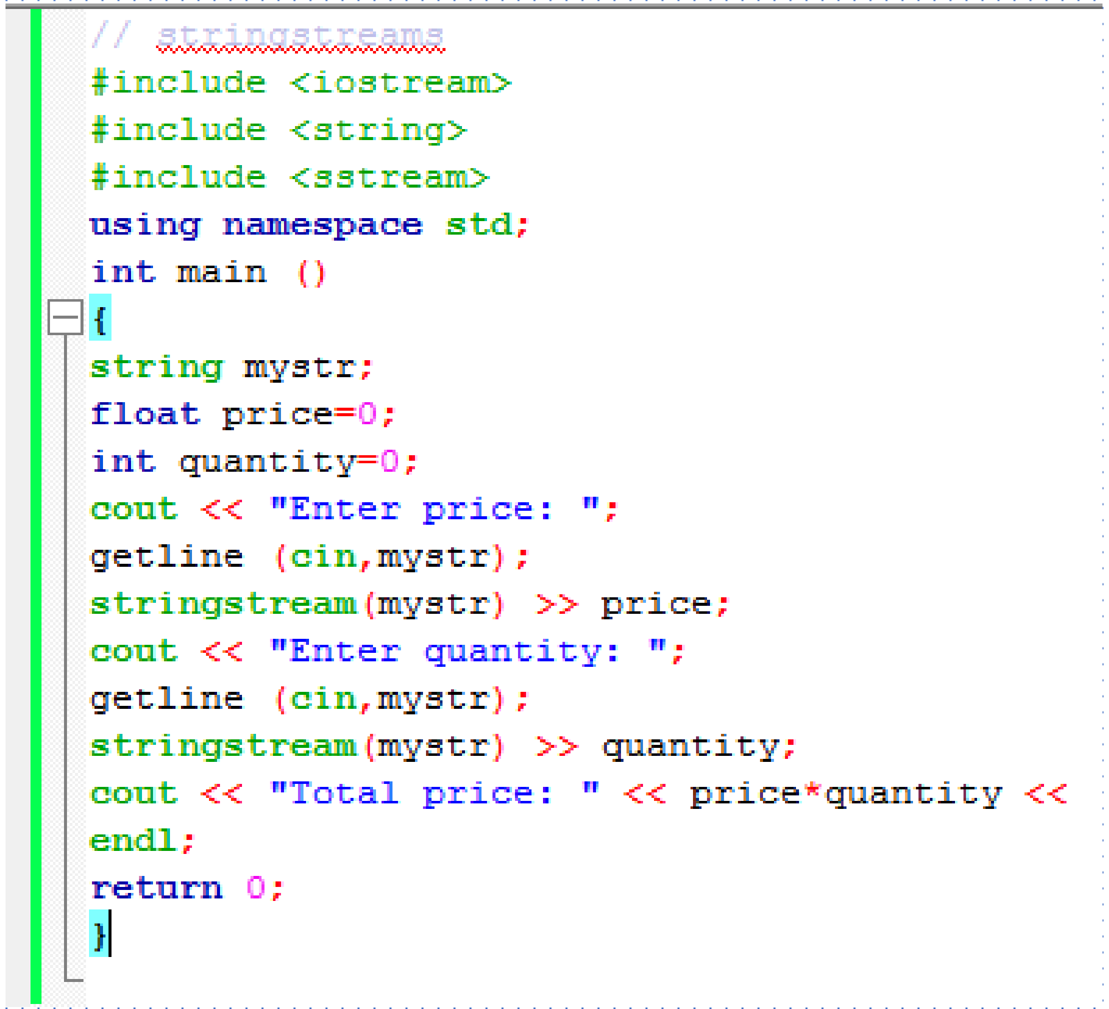


```
// cin with strings
#include <iostream>
#include <string>
using namespace std;
int main ()
{
string mystr;
cout << "What's your name? ";
getline (cin, mystr);
cout << "Hello " << mystr << ".\n";
cout << "What is your favorite team? ";
getline (cin, mystr);
cout << "I like " << mystr << " too!\n";
return 0;
}
```

The standard header file **<sstream>** defines a class called stringstream that allows a string-based object to be treated as a stream. This way we can perform extraction or insertion operations from/to strings, which is especially useful to convert strings to numerical values and vice versa. For example, if we want to extract an integer from a string we can write:

```
string mystr ("1204");
int myint;
stringstream(mystr) >> myint;
```

Inputs



```
// stringstreams
#include <iostream>
#include <string>
#include <sstream>
using namespace std;
int main ()
{
string mystr;
float price=0;
int quantity=0;
cout << "Enter price: ";
getline (cin,mystr);
stringstream(mystr) >> price;
cout << "Enter quantity: ";
getline (cin,mystr);
stringstream(mystr) >> quantity;
cout << "Total price: " << price*quantity <<
endl;
return 0;
}
```

In this example, we acquire numeric values from the standard input indirectly. Instead of extracting numeric values directly from the standard input, we get lines from the standard input (cin) into a string object (mystr), and then we extract the integer values from this string into a variable of type int (quantity).

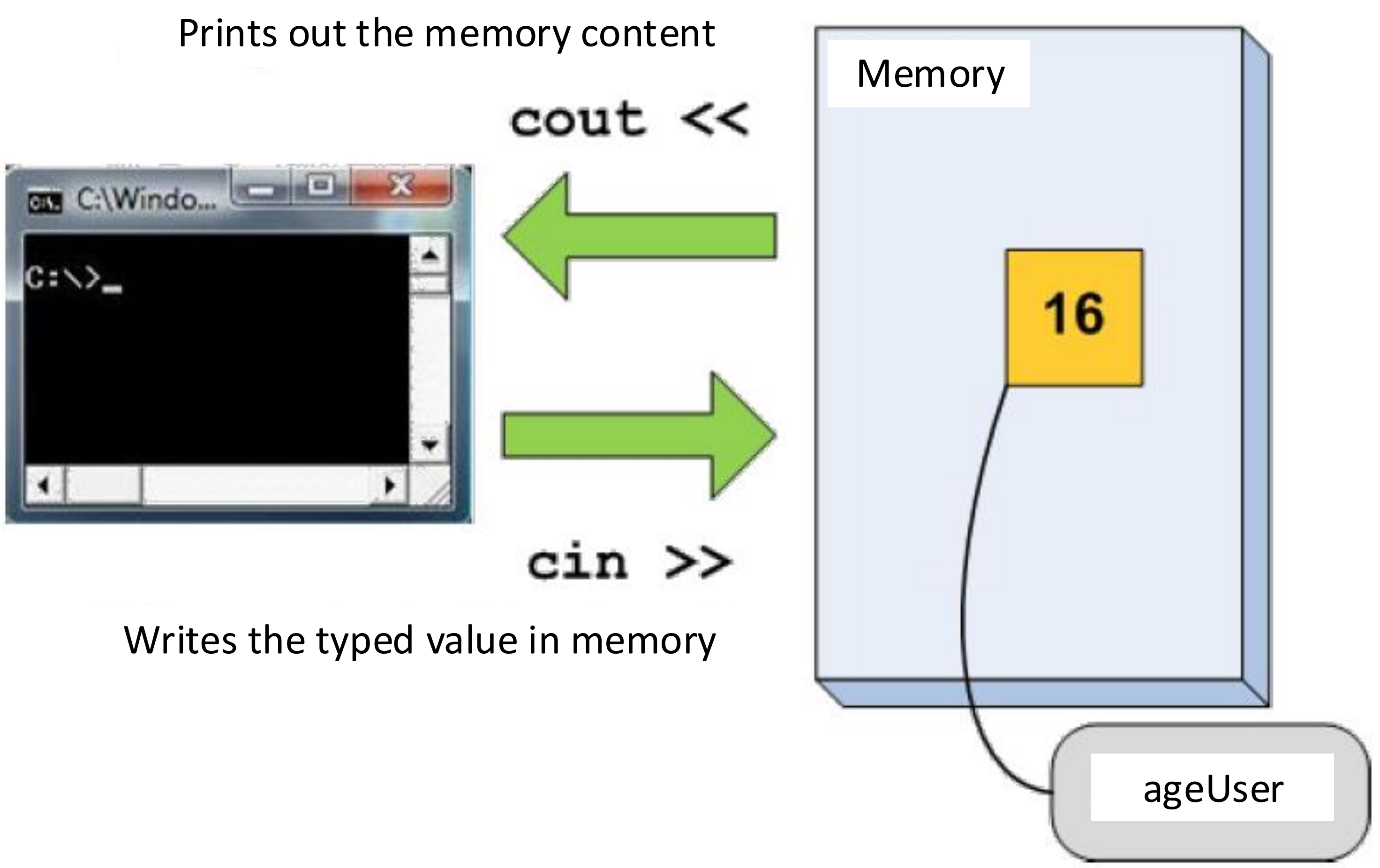
Using this method, instead of direct extractions of integer values, we have more control over what happens with the input of numeric values from the user, since we are separating the process of obtaining input from the user (we now simply ask for lines) with the interpretation of that input.

Therefore, this method is usually preferred to get numerical values from the user in all programs that are intensive in user input.

Debugging

There are two kinds of errors you’ll run into when writing C++ programs**: compilation errors** and **runtime errors**. Compilation errors are problems raised by the compiler, generally resulting from violations of the syntax rules or misuse of types. These are often caused by typos and the like. Runtime errors are problems that you only spot when you run the program: you did specify a legal program, but it doesn’t do what you wanted it to. These are usually more tricky to catch, since the compiler won’t tell you about them.

Summary


Prints out the memory content
cout <<
C:\Windo...
C:\>_
cin >>
Writes the typed value in memory
Memory
16
ageUser

Control of Flow

Motivation

Normally, a program executes statements from first to last. The first statement is executed, then the second, then the third, and so on, until the program reaches its end and terminates. A computer program likely wouldn't be very useful if it ran the same sequence of statements every time it was run. It would be nice to be able to change which statements ran and when, depending on the circumstances. For example, if a program checks a file for the number of times a certain word appears, it should be able to give the correct count no matter what file and word are given to it. Or, a computer game should move the player's character around when the player wants. We need to be able to alter the order in which a program's statements are executed, the ***flow control***.

Control structures

Control structures are portions of program code that contain statements within them and, depending on the circumstances, execute these statements in a certain way. There are typically two kinds: *conditionals* and *loops.*

With the introduction of control structures we are going to have to introduce a new concept: the *compound statement* or *block*. A block is a group of statements which are separated by semicolons (;) like all C++ statements, but grouped together in a block enclosed in braces:

{ }:

{ statement1; statement2; statement3; }

Control structures: if, if-else and else if

Control structures are portions of program code that contain statements within them and, depending on the circumstances, execute these statements in a certain way. There are typically two kinds: *conditionals* and *loops*. With the introduction of control structures we are going to have to introduce a new concept: the *compound statement* or *block*. A block is a group of statements which are separated by semicolons (;) like all C++ statements, but grouped together in a block enclosed in braces: { }:

{ statement1; statement2; statement3; }

```
if(condition)
{
        statement1
        statement2
        …
}
```

```
if(condition)
        statement
```

```
if(condition)
        statementA1
else
        statementB1
```

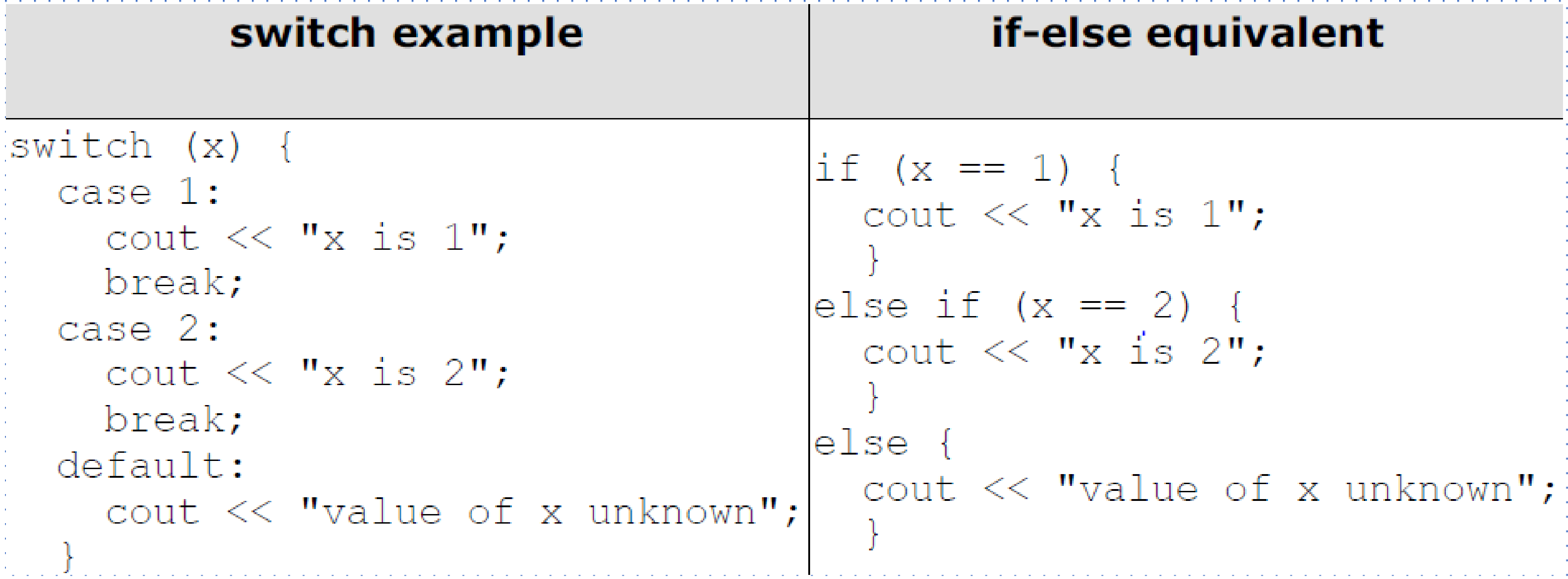
```
if(condition)
{
        statementA1
        statementA2
        …
}
else
{
        statementB1
        statementB2
        …
}
```

```
if(condition1)
{
        statementA1
        statementA2
        …
}
else if(condition2)
{
        statementB1
        statementB2
        …
}
```

Control structures: if, if-else and else if

The condition is some expression whose value is being tested. If the condition resolves to a value of true, then the statements are executed before the program continues on. Otherwise, the statements are ignored. If there is only one statement, the curly braces may be omitted.

The ***switch-case*** is another conditional structure that may or may not execute certain statements. However, the switch-case has peculiar syntax and behavior:



switch example	if-else equivalent

`switch (x) {`
  `case 1:`
    `cout << "x is 1";`
    `break;`
  `case 2:`
    `cout << "x is 2";`
    `break;`
  `default:`
    `cout << "value of x unknown";`
  `}`
	`if (x == 1) {`
	    `cout << "x is 1";`
	    `}`
	`else if (x == 2) {`
	    `cout << "x is 2";`
	    `}`
	`else {`
	    `cout << "value of x unknown";`
	    `}`

Control structures: if, if-else and else if

The switch evaluates expression and, if expression is equal to constant1, then the statements beneath case constant 1: are executed until a break is encountered. If expression is not equal to constant1, then it is compared to constant2. If these are equal, then the statements beneath case constant 2: are executed until a break is encountered. If not, then the same process repeats for each of the constants, in turn. If none of the constants match, then the statements beneath default: are executed.

Due to the peculiar behavior of switch-cases, curly braces are not necessary for cases where there is more than one statement (but they *are* necessary to enclose the entire switch-case). switch-cases generally have if-else equivalents but can often be a cleaner way of expressing the same behavior.

```
switch (x) {
  case 1:
  case 2:
  case 3:
    cout << "x is 1, 2 or 3";
    break;
  default:
    cout << "x is not 1, 2 nor 3";
  }
```

Control structures: if, if-else and else if

The switch evaluates expression and, if expression is equal to constant1, then the statements beneath case constant 1: are executed until a break is encountered. If expression is not equal to constant1, then it is compared to constant2. If these are equal, then the statements beneath case constant 2: are executed until a break is encountered. If not, then the same process repeats for each of the constants, in turn. If none of the constants match, then the statements beneath default: are executed.

Due to the peculiar behavior of switch-cases, curly braces are not necessary for cases where there is more than one statement (but they *are* necessary to enclose the entire switch-case). switch-cases generally have if-else equivalents but can often be a cleaner way of expressing the same behavior.

```
switch (x) {
  case 1:
  case 2:
  case 3:
    cout << "x is 1, 2 or 3";
    break;
  default:
    cout << "x is not 1, 2 nor 3";
  }
```

Notice that switch can only be used to compare an expression against constants. Therefore we cannot put variables as labels (for example `case n:` where `n` is a variable) or ranges (`case (1..3):`) because they are not valid C++ constants.
If you need to check ranges or values that are not constants, use a concatenation of `if` and `else if` statements.

Control structures: Loops

Conditionals execute certain statements if certain conditions are met; loops execute certain statements *while* certain conditions are met. C++ has three kinds of loops: ***while, do-while***, and ***for.***

The *do-while* loop is a variation that guarantees the block of statements will be executed *at least once:*

```
while(condition)
{
        statement1
        statement2
        …
}
```

```
do
{
        statement1
        statement2
        …
}
while(condition);
```

note the semicolon after the `while` condition.

Control structures: Loops

```cpp
// custom countdown using while

#include <iostream>
using namespace std;

int main ()
{
  int n;
  cout << "Enter the starting number > ";
  cin >> n;

  while (n>0) {
    cout << n << ", ";
    --n;
  }

  cout << "FIRE!\n";
  return 0;
}
```

```
Enter the starting number > 8
8, 7, 6, 5, 4, 3, 2, 1, FIRE!
```

Control structures: Loops

```
// number echoer

#include <iostream>
using namespace std;

int main ()
{
  unsigned long n;
  do {
    cout << "Enter number (0 to end): ";
    cin >> n;
    cout << "You entered: " << n << "\n";
  } while (n != 0);
  return 0;
}
```

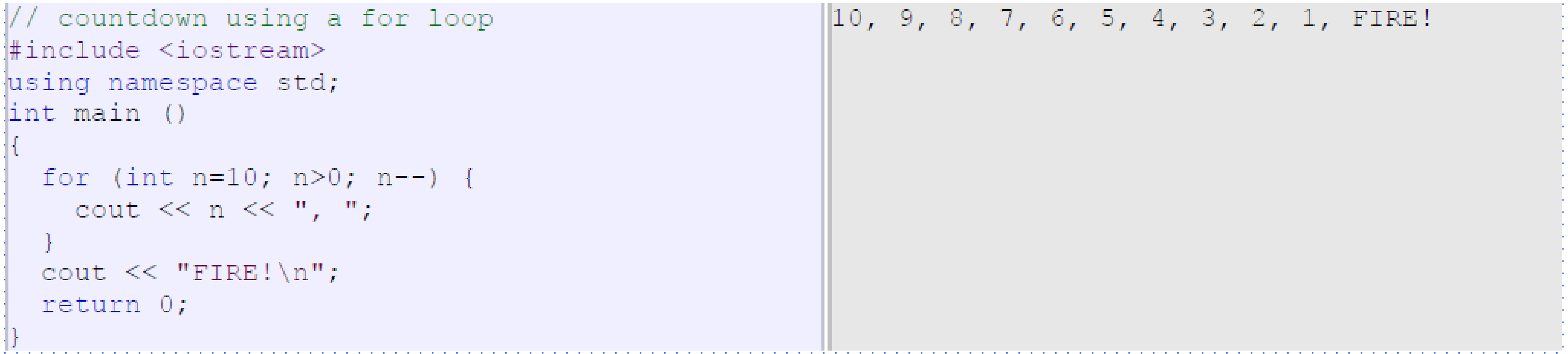
```
Enter number (0 to end): 12345
You entered: 12345
Enter number (0 to end): 160277
You entered: 160277
Enter number (0 to end): 0
You entered: 0
```

Control structures: Loops

The ***for*** loop works like the while loop but with some change in syntax:

```
for(initialization; condition; incrementation)
{
        statement1
        statement2
        ...
}
```

The for loop is designed to allow a counter variable that is initialized at the beginning of the loop and incremented (or decremented) on each iteration of the loop. Curly braces may be omitted if there is only one statement.

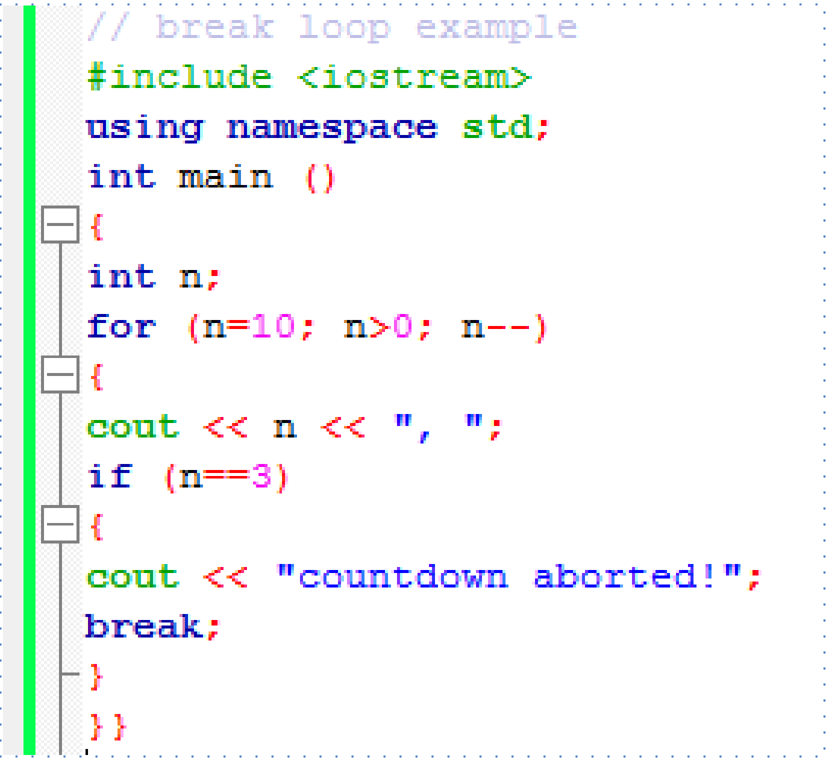


```
// countdown using a for loop
#include <iostream>
using namespace std;
int main ()
{
  for (int n=10; n>0; n--) {
    cout << n << ", ";
  }
  cout << "FIRE!\n";
  return 0;
}
```

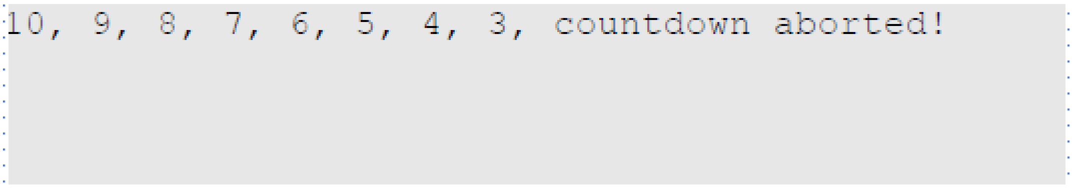
```
10, 9, 8, 7, 6, 5, 4, 3, 2, 1, FIRE!
```

Control structures: Loops

Using **break** we can leave a loop even if the condition for its end is not fulfilled. It can be used to end an infinite loop, or to force it to end before its natural end. For example, we are going to stop the count down before its natural end (maybe because of an engine check failure?):



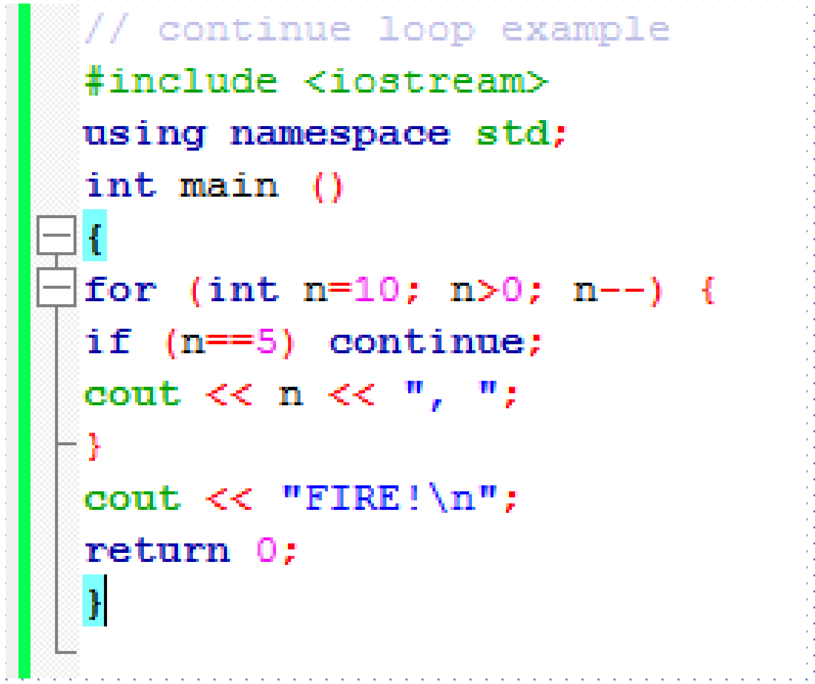
```
// break loop example
#include <iostream>
using namespace std;
int main ()
{
int n;
for (n=10; n>0; n--)
{
cout << n << ", ";
if (n==3)
{
cout << "countdown aborted!";
break;
}
}}
```



```
10, 9, 8, 7, 6, 5, 4, 3, countdown aborted!
```

Control structures: Loops

The **continue** statement causes the program to skip the rest of the loop in the current iteration as if the end of the statement block had been reached, causing it to jump to the start of the following iteration. For example, we are going to skip the number 5 in our countdown:



```
// continue loop example
#include <iostream>
using namespace std;
int main ()
{
for (int n=10; n>0; n--) {
if (n==5) continue;
cout << n << ", ";
}
cout << "FIRE!\n";
return 0;
}
```

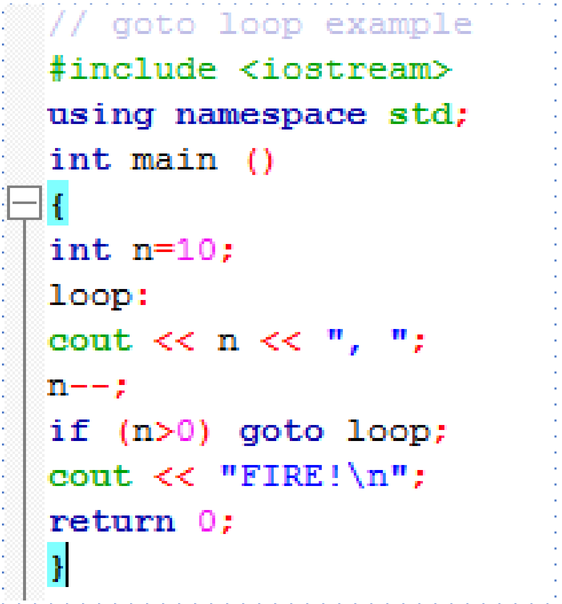
```
10, 9, 8, 7, 6, 4, 3, 2, 1, FIRE!
```

Control structures: Loops

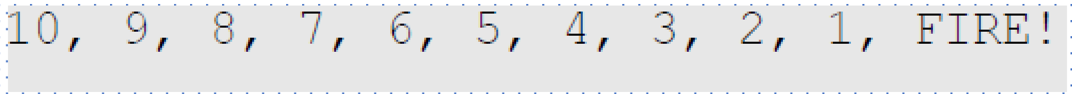
goto allows to make an absolute jump to another point in the program. You should use this feature with caution since its execution causes an unconditional jump ignoring any type of nesting limitations.

The destination point is identified by a label, which is then used as an argument for the goto statement. A label is made of a valid identifier followed by a colon (:).

Generally speaking, this instruction has no concrete use in structured or object oriented programming aside from those that low-level programming fans may find for it. For example, here is our countdown loop using goto:



```
// goto loop example
#include <iostream>
using namespace std;
int main ()
{
int n=10;
loop:
cout << n << ", ";
n--;
if (n>0) goto loop;
cout << "FIRE!\n";
return 0;
}
```



```
10, 9, 8, 7, 6, 5, 4, 3, 2, 1, FIRE!
```

Control structures: Loops

exit is a function defined in the cstdlib library.

The purpose of exit is to terminate the current program with a specific exit code. Its prototype is: **void exit (int exitcode);**

The exit code is used by some operating systems and may be used by calling programs. By convention, an exit

code of 0 means that the program finished normally and any other value means that some error or unexpected results happened.

Exercises

1. Given a list of N integers, find its mean (as a double), maximum value and minimum value. Your program will first ask for N, the number of integers in the list, which the user will input. Then the user will input N more numbers.

2. Write a program to read a number N from the user and then find the first N primes.

3. a) Write a program that loops indefinitely. In each iteration of the loop, read in an integer N (declared as an int) that is inputted by a user, output N/5 if N is nonnegative and divisible by 5, and -1 otherwise. Use the ternary operator (?:) to accomplish this. (Hint: the modulus operator may be useful.)

b) Modify the code from (a) so that if the condition fails, nothing is printed. Use an if and a continue command (instead of the ternary operator) to accomplish this.

c) Modify the code from (b) to let the user break out of the loop by entering -1 or any negative number. Before the program exits, output the string "Goodbye!".

4) Write a program that can do the factorial of n.