1 - C++ Basics

Statements and Structure

- Statements : a type of instruction that causes a program to perform some action (usually ends with ;) . Some include, declaration, jump, expression, compound, selections, iteration and try blocks.
- Function : collection of statements that execute sequentially.
- Execution starts with the first statement inside the main function.
- Library File : a collection of precompiled code that has been “packaged up” for reuse in other programs. The C++ Standard Library is a library that ships with C++ that contains additional functionality to use in your programs

Comments

- **Comments** are readable notes put in by the programmer for code articulation.
- **Single-line** comments begins with //
  - Use tabs to align comments so that they are neater
  - Or just place comment on top of statements

```
1 std::cout << "Hello world!\n"; // std::cout lives in the iostream library
2 std::cout << "It is very nice to meet you!\n"; // these comments make the code hard to read
3 std::cout << "Yeah!\n"; // especially when lines are different lengths
```

Having comments to the right of a line can make both the code and the comment hard to read, particularly if the line is long. If the lines are fairly short, the comments can simply be aligned (usually to a tab stop), like so:

```
1 std::cout << "Hello world!\n";                // std::cout lives in the iostream library
2 std::cout << "It is very nice to meet you!\n"; // this is much easier to read
3 std::cout << "Yeah!\n";                       // don't you think so?
```

- **Multi-line** comments are denoted by /* and */

```
/* This is a multi-line comment.
 * the matching asterisks to the left
 * can make this easier to read
 */
```

- Don't use multi-line comments inside other multi-line comments. Wrapping single-line comments inside a multi-line comment is okay.
- Bad and Good comments

Bad comment:

```
// Calculate the cost of the items
cost = quantity * 2 * storePrice;
```

Reason: We can see that this is a cost calculation, but why is quantity multiplied by 2?

Good comment:

```
// We need to multiply quantity by 2 here because they are bought in pairs
cost = quantity * 2 * storePrice;
```

Reason: Now we know why this formula makes sense.

Variables

- Direct access memory not allowed in C++, instead we work with objects which is a region of storage that has a value.
- **Definitions** used to create variables.
- Let's say that variable x is instantiated at memory location 140. Whenever the program then uses variable x, it will access the value in memory location 140. An instantiated object is sometimes also called an instance.
- A **variable** is a named region of memory.
- An **identifier** is the name that a variable is accessed by
- **Data type** (types) tells compiler what type of value does the compiler store - number, letter...
- C++ also allows you to create your own user-defined types.

Assignments and Initializations

- Copy initialization (**CI**): `int a = 10;`
- Direct initialization (**DI**): `int a (10);`
- DI can perform better than CI as it helps boost performance.
- Since DI can't be used everywhere, Uniform Initializations (**UI**) are used: `int a {10};`
- UI disallows narrowing conversions (low data loss), i.e error thrown here: `int a{4.3};`
- UI is what is preferred.
- Doing something like this `int a, b=10;` leaves a uninitialized and the program could crash as a result.

Iostream, cout, cin and endl

- Iostream is a part of the C++ library that deals with basic input/output.
- cout : is used to output a value : `std::cout << x;`
- cin : is used to take in an input from the user : `std::cin >> x;`
- endl : allows printing stuff on different lines, **'\n'** is **preferred** over endl
- Best practice is to **always initialized** variables.

Uninitialized variables and undefined behaviour

- If a variable is not initialized with any value, it's assigned **garbage value.**
  - Compiler assigns some unused memory to the variable and the variable assumes the value in the memory
  - Always initialize your variables" **best practice**.
- Undefined behaviour is result of executing code whose behavior is not well defined by the C++ language (leaving variables uninitialized could cause this).

Keywords and naming identifiers

alignas (C++11)	enum	return
alignof (C++11)	explicit	short
and	export	signed
and_eq	extern	sizeof
asm	false	static
auto	float	static_assert (C++11)
bitand	for	static_cast
bitor	friend	struct
bool	goto	switch
break	if	template
case	inline	this
catch	int	thread_local (C++11)
char	long	throw
char16_t (C++11)	mutable	true
char32_t (C++11)	namespace	try
class	new	typedef
compl	noexcept (C++11)	typeid
const	not	typename
constexpr (C++11)	not_eq	union
const_cast	nullptr (C++11)	unsigned
continue	operator	using
decltype (C++11)	or	virtual
default	or_eq	void
delete	private	volatile
do	protected	wchar_t
double	public	while
dynamic_cast	register	xor
else	reinterpret_cast	xor_eq

- The name of a variable (or function, type, or other kind of item) is called an Identifier.
- The following are a few rules to be kept in mind while naming identifiers:
  - The identifier can not be a keyword. Keywords are reserved.
  - The identifier can only be composed of letters (lower or upper case), numbers, and the underscore character. That means the name can not contain symbols (except the underscore) nor whitespace (spaces or tabs).
  - The identifier must begin with a letter (lower or upper case) or an underscore. It can not start with a number.
  - C++ is case sensitive, and thus distinguishes between lower and upper case letters. `nvalue` is different than `nValue` is different than `NVALUE`.

Introduction to literals and operators

- A **literal** is a fixed value that has been directly inserted into the source code.
- An **operation** is a mathematical calculation involving zero or more input values (called **operands**) that produces a new value.
  - The specific operation to be performed is denoted by a construct called an **operator**
  - Urnary operator acts only on one operand, **Binary** on two, **tertiary** on three...

Introduction to expressions

- A combination of literals, variables and explicit function calls that produces a single output value. (x=5,
- Expressions do not end in a semicolon, and cannot be compiled by themselves.
  - Eg: there's no ; after 2 + 3 `int x{ 2 + 3 };`
- An expression statement is a statement that consists of an expression followed by a semicolon.
  - We can take any expression (such as x = 5), and turn it into an expression statement (such as x = 5;) that will compile.
- **Statements** are used when we want the program to perform an action. **Expressions** are used when we want the program to calculate a value.

First program

```
# include <iostream>


int main(){

    int a{0};
    std::cout << "Enter your number : ";
    std::cin >> a; //integer that'll take in input value
```

```
    int b{0};
    std::cout << "Enter another number : ";
    std::cin >> b;

    std::cout << "Two's multiple of " << a << " is " << 2*a <<"\n";
    std::cout << "Three's multiple of " << a << " is " << 3*a <<"\n";
    std::cout << a << " + " << b << " = " <<a+b << "\n";
    std::cout << a << " - " << b << " = " <<a-b << "\n";
    return 0;
}
```

SUMMARY