Lecture 1 Notes: Introduction

1 Compiled Languages and C++

1.1 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?

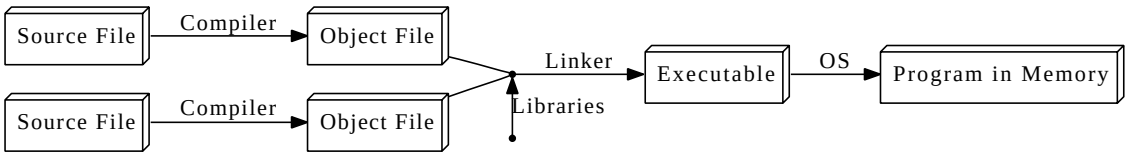
The advantages:

1. **Conciseness:** programming languages allow us to express common sequences of commands more concisely. C++ provides some especially powerful shorthands.
2. **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 Lectures 7-8), which further improves maintainability.
3. **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.

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).

1.2 The Compilation Process

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



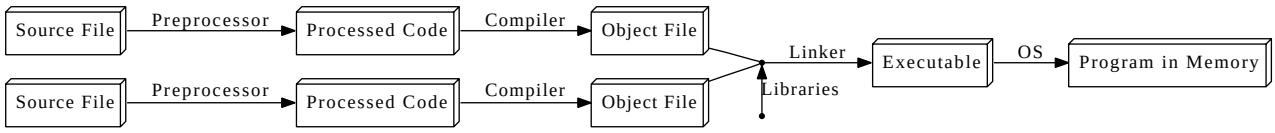
Object files are intermediate files that represent an incomplete copy of the program: each source file only expresses a piece of the program, so when it is compiled into an object file, the object file has some markers indicating which missing pieces it depends on. The linker

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*.

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:



1.3 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`.

2 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++.

2.1 The code

```
1 // A Hello World program
2 #include <iostream>
3
```

```
4 int main() {
5   std::cout << "Hello, world!\n";
6
7   return 0;
8 }
```

2.2 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

2.3 Line-By-Line Explanation

1. `//` 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!

2. 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.

4. `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.

5.
   - `cout <<` : This is the syntax for outputting some piece of text to the screen. We'll discuss how it works in Lecture 9.
   - **Namespaces:** In C++, identifiers can be defined within a context – sort of a directory of names – called a *namespace*. When we want to access an identifier defined in a namespace, we tell the compiler to look for it in that namespace using the *scope resolution operator* (`::`). Here, we're telling the compiler to look for `cout` in the `std` namespace, in which many standard C++ identifiers are defined.

     A cleaner alternative is to add the following line below line 2:

     ```
     using namespace std;
     ```

     This line tells the compiler that it should look in the `std` namespace for any identifier we haven't defined. If we do this, we can omit the `std::` prefix when writing `cout`. This is the recommended practice.
   - **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 `\n` 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:

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

7. `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.

Note that every statement ends with a semicolon (except preprocessor commands and blocks using {}). Forgetting these semicolons is a common mistake among new C++ programmers.

3 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.

3.1 Values and Statements

First, a few definitions:

- 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.

3.2 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.

3.3 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:

Type Names	Description	Size	Range
`char`	Single text character or small integer. Indicated with single quotes (`'a'`, `'3'`).	1 byte	signed: -128 to 127 unsigned: 0 to 255
`int`	Larger integer.	4 bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
`bool`	Boolean (true/false). Indicated with the keywords `true` and `false`.	1 byte	Just `true` (1) or `false` (0).
`double`	"Doubly" precise floating point number.	8 bytes	+/- 1.7e +/- 308 (15 digits)

Notes on this table:

- 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).
- The sizes/ranges for each type are not fully standardized; those shown above are the ones used on most 32-bit computers.

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 in Lecture 5, has the type `char *`.

Variables

A variable is a container (storage area) used to hold data.
Each variable should be given a unique name (identifier).

int a=2;

Here a is the variable name that holds the integer value 2.
The value of a can be changed, hence the name variable.

There are certain rules for naming a variable in C++

1. Can only have alphabets, numbers and underscore.
2. Cannot begin with a number.
3. Cannot begin with an uppercase character.
4. Cannot be a keyword defined in C++ language (like int is a keyword).

Fundamental Data Types in C++

Data types are declarations for variables. This determines the type and size of data associated with variables which is essential to know since different data types occupy different size of memory.

Data Type	Meaning	Size (in Bytes)
int	Integer	4
float	Floating-point	4
double	Double Floating-point	8
char	Character	1
wchar_t	Wide Character	2
bool	Boolean	1
void	Empty	0

1. int

- This data type is used to store integers.
- It occupies 4 bytes in memory.
- It can store values from -2147483648 to 2147483647.

Eg. int age = 18

2. float and double

- Used to store floating-point numbers (decimals and exponentials)
- Size of float is 4 bytes and size of double is 8 bytes.
- Float is used to store upto 7 decimal digits whereas double is used to store upto 15 decimal digits.

Eg. float pi = 3.14
double distance = 24E8 // 24 x 10^8

3. char

- This data type is used to store characters.
- It occupies 1 byte in memory.
- Characters in C++ are enclosed inside single quotes ‘ ‘.
- ASCII code is used to store characters in memory.

Eg. char ch = ‘a’;

ASCII Table

0	?	1	?	2	?	3	?	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	0	49	1	50	2	51	3	52	4	53	5	54	6	55	7
56	8	57	9	58	:	59	;	60	<	61	=	62	>	63	?
64	@	65	A	66	B	67	C	68	D	69	E	70	F	71	G
72	H	73	I	74	J	75	K	76	L	77	M	78	N	79	O
80	P	81	Q	82	R	83	S	84	T	85	U	86	V	87	W
88	X	89	Y	90	Z	91	[	92	\	93	]	94	^	95	_
96	`	97	a	98	b	99	c	100	d	101	e	102	f	103	g
104	h	105	i	106	j	107	k	108	l	109	m	110	n	111	o
112	p	113	q	114	r	115	s	116	t	117	u	118	v	119	w
120	x	121	y	122	z	123	{	124	|	125	}	126	~		

4. bool

- This data type has only 2 values – true and false.
- It occupies 1 byte in memory.
- True is represented as 1 and false as 0.

Eg. bool flag = false

C++ Type Modifiers

Type modifiers are used to modify the fundamental data types.

Data Type	Size (in Bytes)	Meaning
signed int	4	used for integers (equivalent to int)
unsigned int	4	can only store positive integers
short	2	used for small integers (range -32768 to 32767)
long	at least 4	used for large integers (equivalent to long int)
long long int	8	used for very large integers (equivalent to long long int).
unsigned long long	8	used for very large positive integers or 0 (equivalent to unsigned long long int)
long double	~~8~~ 16	used for large floating-point numbers
signed char	1	used for characters (guaranteed range -127 to 127)
unsigned char	1	used for characters (range 0 to 255)

Derived Data Types

These are the data types that are derived from fundamental (or built-in) data types. For example, arrays, pointers, function, reference.

User-Defined Data Types

These are the data types that are defined by user itself.
For example, class, structure, union, enumeration, etc.

We will be studying the derived and user-defined data types in detail in the further video lectures.

4 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.

For example, say we wanted to use the value `4 + 2` multiple times. We might call it `x` and use it as follows:

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5    int x;
6    x = 4 + 2;
7    cout << x / 3 << ' ' << x * 2;
8
9    return 0;
10 }
```

(Note how we can print a sequence of values by "chaining" the `<<` symbol.)

The name of a variable is an identifier token. Identifiers may contain numbers, letters, and underscores (`_`), and may not start with a number.

Line 5 is the *declaration* of the variable `x`. We must tell the compiler what type `x` will be so that it knows how much memory to reserve for it and what kinds of operations may be performed on it.

Line 6 is the *initialization* of `x`, where we specify an initial value for it. This introduces a new operator: `=`, the assignment operator. We can also change the value of `x` later on in the code using this operator.

We could replace lines 5 and 6 with a single statement that does both declaration and initialization:

```
int x = 4 + 2;
```

This form of declaration/initialization is cleaner, so it is to be preferred.

5 Input

Now that we know how to give names to values, we can have the user of the program input values. This is demonstrated in line 6 below:

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5    int x;
6    cin >> x;
7
8    cout << x / 3 << ' ' << x * 2;
9
10   return 0;
11 }
```

Just as `cout <<` is the syntax for outputting values, `cin >>` (line 6) is the syntax for inputting values.

Memory trick: if you have trouble remembering which way the angle brackets go for `cout` and `cin`, think of them as arrows pointing in the direction of data flow. `cin` represents the terminal, with data flowing from it to your variables; `cout` likewise represents the terminal, and your data flows to it.

6 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.

Operators in C++

Operators are nothing but symbols that tell the compiler to perform some specific operations. Operators are of the following types -

1. Arithmetic Operators

Arithmetic operators perform some arithmetic operation on one or two operands. Operators that operate on one operand are called unary arithmetic operators and operators that operate on two operands are called binary arithmetic operators.
+,-,*,/,% are binary operators.
++, -- are unary operators.

Suppose : A=5 and B=10

Operator	Operation	Example
+	Adds two operands	A+B = 15
-	Subtracts right operand from left operand	B-A = 5
*	Multiplies two operands	A*B = 50
/	Divides left operand by right operand	B/A = 2
%	Finds the remainder after integer division	B%A = 0
++	Incrementer	A++ = 6
--	Decrementer	A-- = 4

Pre-incrementer : It increments the value of the operand instantly.
Post-incrementer : It stores the current value of the operand temporarily and only after that statement is completed, the value of the operand is incremented.
Pre-decrementer : It decrements the value of the operand instantly.
Post-decrementer : It stores the current value of the operand temporarily and only after that statement is completed, the value of the operand is decremented.

Example -

```
int a=10;
int b;

b = a++;
cout<<a<<" "<<b<<endl;
```

Output : 11 10

```
int a=10;
int b;

b = ++a;
cout<<a<<" "<<b<<endl;
```

Output : 11 11

2. Relational Operators

Relational operators define the relation between 2 entities.
They give a boolean value as result i.e true or false.

Suppose : A=5 and B=10

Operator	Operation	Example
==	Gives true if two operands are equal	A==B is not true
!=	Gives true if two operands are not equal	A!=B is true
>	Gives true if left operand is more than right operand	A>B is not true
<	Gives true if left operand is less than right operand	A<B is true
>=	Gives true if left operand is more than right operand or equal to it	A>=B is not true
<=	Gives true if left operand is more than right operand or equal to it	A<=B is true

Example -

We need to write a program which prints if a number is more than 10, equal to 10 or less than 10. This could be done using relational operators with if else statements.

```
int n;
cin>>n;

if(n<10){
   cout<<"Less than 10"<<endl;
}
else if(n==10){
   cout<<"Equal to 10"<<endl;
}
else{
   cout<<"More than 10"<<endl;
}
```


3. Logical Operators

Logical operators are used to connect multiple expressions or conditions together.
We have 3 basic logical operators.

Suppose : A=0 and B=1

Operator	Operation	Example
&&	AND operator. Gives true if both operands are non-zero	(A && B) is false
||	OR operator. Gives true if atleast one of the two operands are non-zero.	(A || B) is true
!	NOT operator. Reverse the logical state of operand	!A is true

Example -

If we need to check whether a number is divisible by both 2 and 3, we will use AND operator

(num%2==0) && num(num%3==0)

If this expression gives true value then that means that num is divisible by both 2 and 3.

(num%2==0) || (num%3==0)

If this expression gives true value then that means that num is divisible by 2 or 3 or both.

4. Bitwise Operators

Bitwise operators are the operators that operate on bits and perform bit-by-bit operations.

Suppose : A=5(0101) and B=6(0110)

Operator	Operation	Example
&	Binary AND. Copies a bit to the result if it exists in both operands.	0101 & 0110 0100
|	Binary OR. Copies a bit if it exists in either operand.	0101 | 0110 0111
^	Binary XOR. Copies the bit if it is set in one operand but not both.	0101 ^ 0110 0011
~	Binary Ones Complement. Flips the bit.	~0101 => 1010
<<	Binary Left Shift. The left operand's bits are moved left by the number of places specified by the right operand.	4 (0100) 4<<1 =1000 = 8

>>	Binary Right Shift Operator. The left operand's bits are moved right by the number of places specified by the right operand.	4>>1 =0010 = 2

If shift operator is applied on a number N then,

- N<<a will give a result N*2^a
- N>>a will give a result N/2^a

5. Assignment Operators

Operator	Operation	Example
=	Assigns value of right operand to left operand	A=B will put value of B in A
+=	Adds right operand to the left operand and assigns the result to left operand.	A+=B means A = A+B
-=	Subtracts right operand from the left operand and assigns the result to left operand.	A-=B means A=A-B
*=	Multiplies right operand with the left operand and assigns the result to left operand.	A*=B means A=A*B
/=	Divides left operand with the right operand and assigns the result to left operand.	A/=B means A=A/B

6. Misc Operators

Operator	Operation	Example
sizeof()	Returns the size of variable	If a is integer then sizeof(a) will return 4

Condition? X:Y	Conditional operator. If condition is true, then returns value of X or else value of Y	A+=B means A = A+B
Cast	Casting operators convert one data type to another.	int(4.350) would return 4.
Comma (,)	Comma operator causes a sequence of operations to be performed. The value of the entire comma expression is the value of the last expression of the comma-separated list.	

Precedence of Operators

Category	Operator	Associativity
Postfix	() [] -> . ++ - -	Left to right
Unary	+ - ! ~ ++ - - (type)* & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right

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

Examples –

Ques. Give the output of the following programs.

1.

```
#include<stdio.h>
int main()
{
        int a = 5;
        a = 1, 2, 3;
        printf("%d", a);
        return 0;
}
```

Ans. 1
(Priority for the values assigned to any variable is given from left to right)

2.

```
#include<stdio.h>
int main()
{
        int a;
        a = (1, 2, 3);
        printf("%d", a);
        return 0;
```

```
}
```

Ans. 3
(Priority for the values inside a brackets () assigned to any variable is given from right to left.)

3.

```
#include<stdio.h>
int main()
{
        int x = 2;
        (x & 1) ? printf("true") : printf("false");
        return 0;
}
```

Ans. False
(As & is a unary operator we have to assume all decimal values to binary(0's and 1's)
2_{10} = 00000010_2
Now we go for condition (00000010 & 00000001)
Clearly, condition false as it leads to 0 when multiplied.)

4.

```
#include<stdio.h>
int main()
{
        printf("%d",3 * 2--);
}
```

Ans. 6
(2-- stands meaningless)

5.

```
#include<stdio.h>
```

```
int main()
{
        int i = 10;
        i++;
        i * i;
        printf("%d\n",i);
        return 0;
}
```

Ans. 11
(i++ alone store the result in variable i.)

6.

```
#include<stdio.h>
int main()
{
        int a = 1, b = 3, c;
        c = b <<  a;
        b = c * (b * (++a)--);
        a = a >> b;
        printf("%d",b);
        return 0;
}
```

Ans. 36
(c = 0011 << 1
c = 0110
b = 6* (3*(2)--)
b = 6*6
)