Quick Revision & Download C++ Basics Slides (Lec 1-14)

Special class

Jay Bansal • Apr 18, 2021

Getting Started with C++

Origins of C++

Ravindrababu Ravula
Jay Bansal

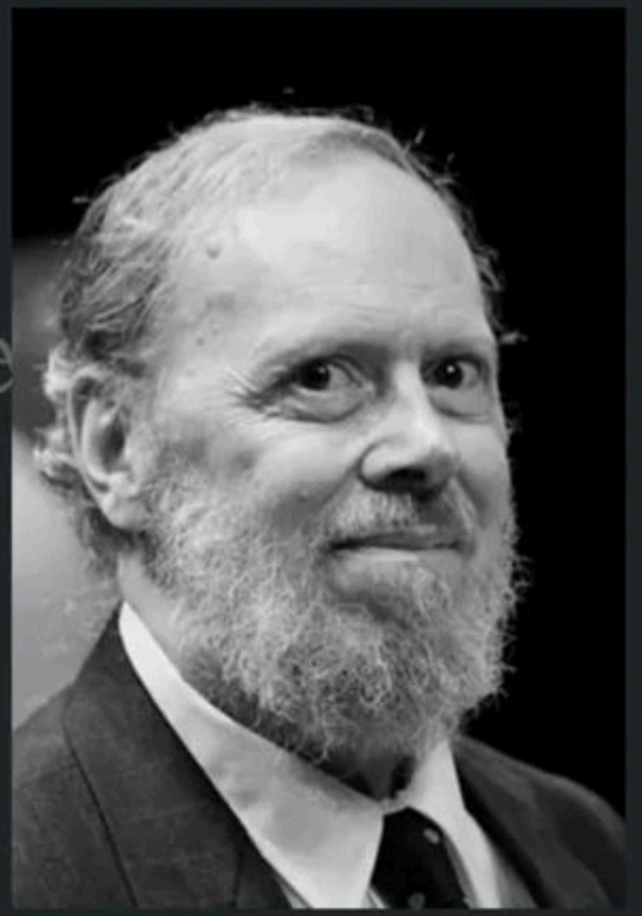
unacademy.com/@ravula
unacademy.com/@jay.bansal

The C programming language came out of Bell Labs in the early 1970s.

The C programming language was devised in the early 1970s as a system implementation language for the nascent Unix operating system.

It is a general purpose high level language

History of C language



Dennis Ritchie

Origins of C++

C++ Development started in 1979

Bjarne Stroustrup worked on Simula (language used for simulation work)

It was the first language to support object-oriented programming (OOP)

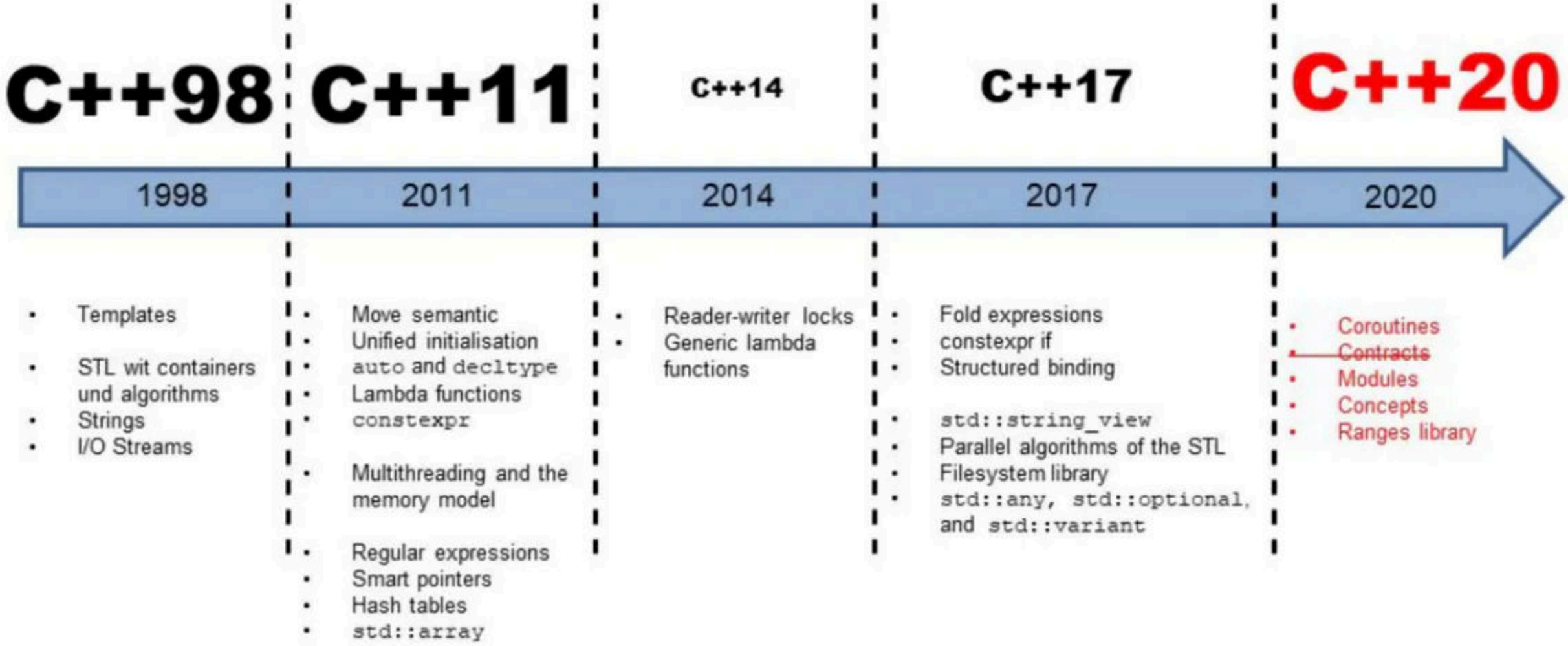
He identified that these OOP features can be included in Software Development, but Simula can't be used because it is too slow

After that, he worked on C language and added OOP features, initially called "C with Classes"

Features were added so as to not affect the basic features of C language

Bjarne Stroustrup

Complete info can be found at: http://www.cplusplus.com/info/history/



Source: https://www.modernescpp.com/index.php/c-20-an-overview

Getting Started with C++

Why C++?

Ravindrababu Ravula
Jay Bansal

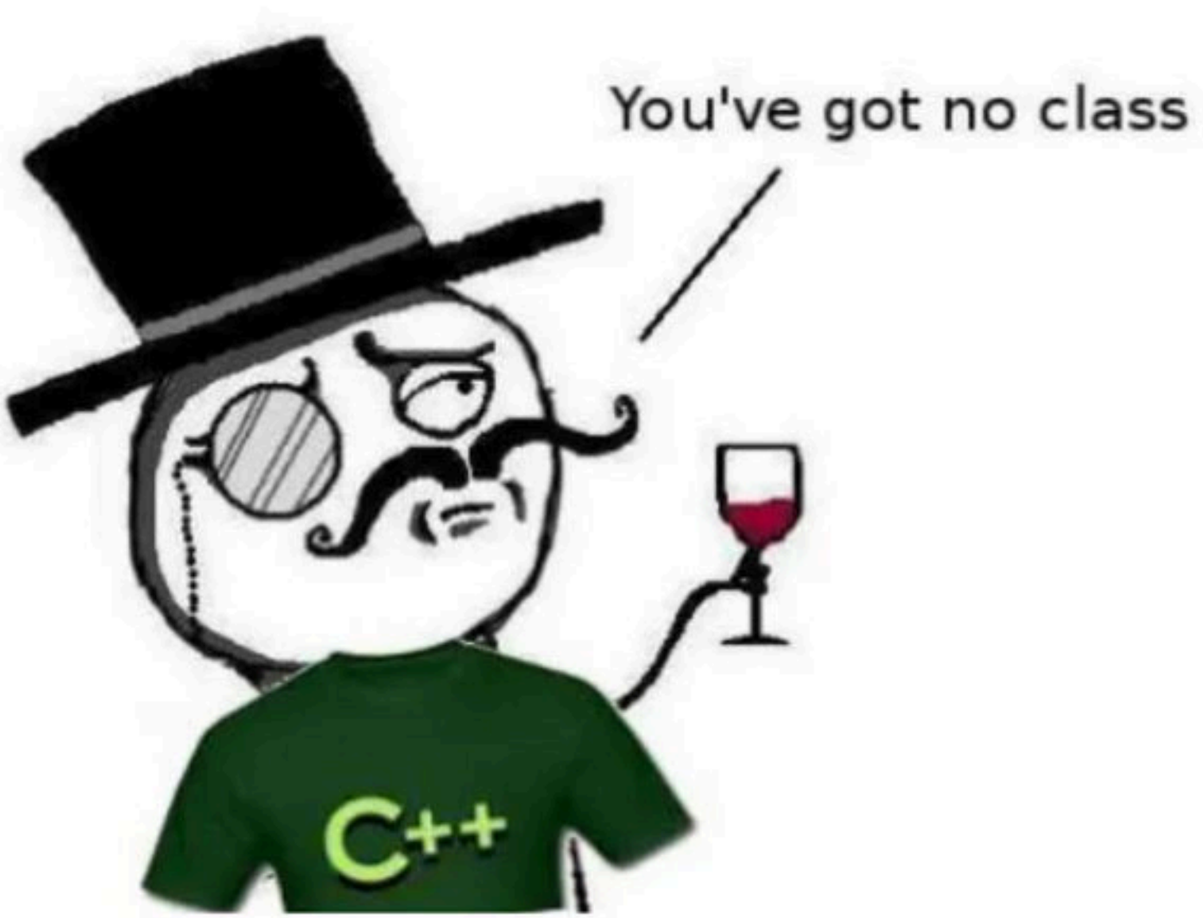
unacademy.com/@ravula
unacademy.com/@jay.bansal

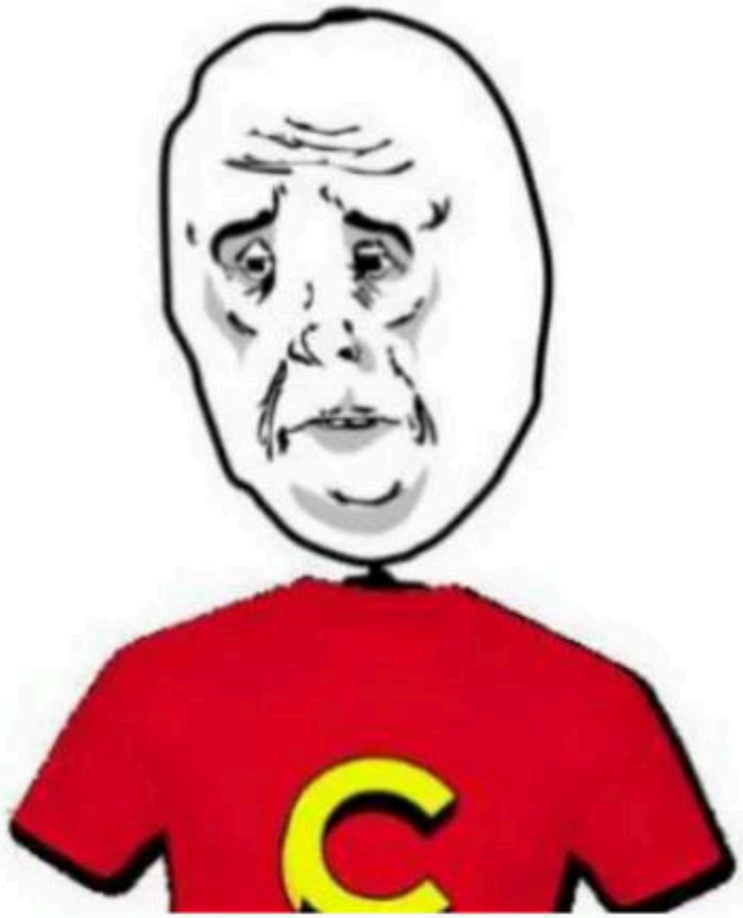
Benefits of C++

- Highly portable language and is a language of choice for multi-device, multi-platform app development.
- Object-oriented programming language and includes classes, inheritance, polymorphism, data abstraction and encapsulation.
- Has a rich function library.
- Allows exception handling, function and operator overloading
- C++ is a powerful, efficient and fast language. Wide range of applications – GUI applications, 3D graphics for games and real-time mathematical simulations.

Features included in C++ language

- Classes, Derived Classes
- Strong Typing
- Inlining
- Default Arguments
- Virtual Functions
- Function name and operator overloading
- Polymorphism





Source: https://devrant.com/

C++ for Competitive Programming

- C++ is very fast and evolving language and has many standards such as C++11, 14, 17. C++ 20 has many new features which solve problems easily such as 3-way comparison operator, Coroutines, modules, etc. and many new header files.

- **STL**: Standard Template Library, a collection of C++ templates to help programmers handle basic data structures and functions such as lists, stacks, arrays, etc. It has container classes, algorithms, iterators, etc. helpful for Competitive Programming

- Ease of Learning, concise in comparison to Java (Python is more concise and is also becoming popular)

- **Fast**: C++ is not only fast but also efficient. And with respect to Java & Python it's much faster

pidigits

source	secs	mem	gz	busy	cpu load
Java	0.79	35,568	764	0.84	99% 3% 3% 1%
C++ g++	0.60	4,944	986	2.38	100% 100% 98% 100%

fannkuch-redux

source	secs	mem	gz	busy	cpu load
Java	11.00	34,104	1282	43.42	99% 98% 98% 99%
C++ g++	8.08	1,900	980	31.45	98% 100% 98% 93%

mandelbrot

source	secs	mem	gz	busy	cpu load
C++ g++	**0.84**	34,604	3542	3.28	98% 99% 98% 95%
Python 3	172.58	12,216	688	689.62	100% 100% 100% 100%

n-body

source	secs	mem	gz	busy	cpu load
C++ g++	**4.09**	1,800	1808	4.13	100% 0% 0% 0%
Python 3	586.17	8,012	1196	589.84	0% 0% 0% 100%

Source: https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/java.html

Getting Started with C++

Setting up C++ environment

Ravindrababu Ravula
Jay Bansal

unacademy.com/@ravula
unacademy.com/@jay.bansal

Linux

1. Install Compiler

   ```
   sudo apt-get install g++
   ```

   Check:

   ```
   g++ --version
   ```

2. Install Editor

   Download Sublime text: https://www.sublimetext.com/3

   Or any other editor...

3. You can use IDE (such as Dev-C++, CLion, Eclipse, etc.) or online platforms such as https://www.codechef.com/ide

How to run C++ program

1. Write your program in a text file and save it with any file name and .cpp extension

2. Now you have to open the Linux terminal and move to the directory where you have saved your file. Run the below command to **compile** your file:

   ```
   g++ file_name.cpp -o object_file_name
   ```

   - file_name.cpp: name of your source code file
   - object_file_name: can be any name of your choice, will be assigned to the executable file which is created by the compiler after compilation

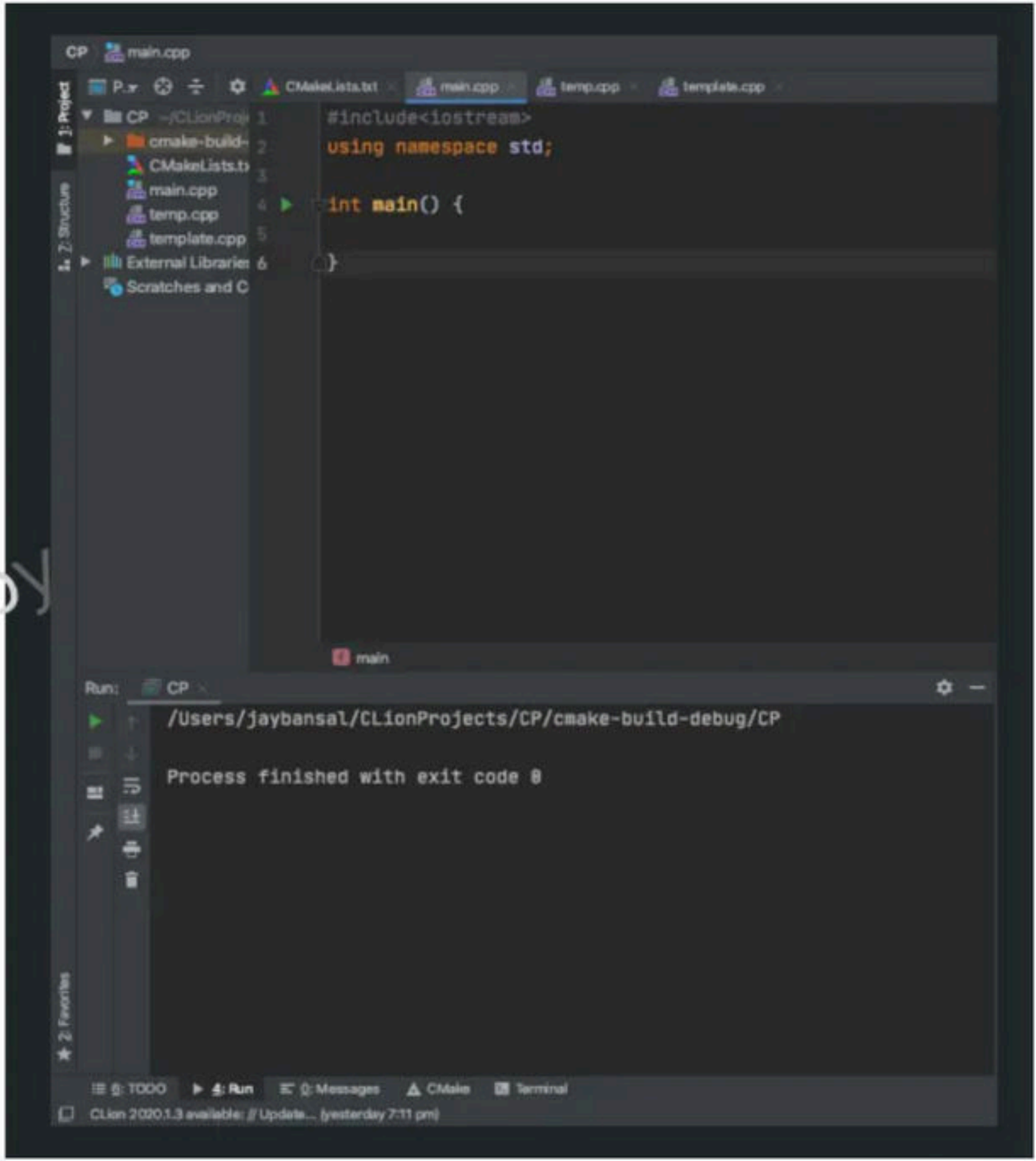
3. Now to run your program you have to run the below command:

   ```
   ./object_file_name
   ```

IDE (Integrated development environment)

Or you can use IDE like VS Code, CLion, Eclipse which gives us:

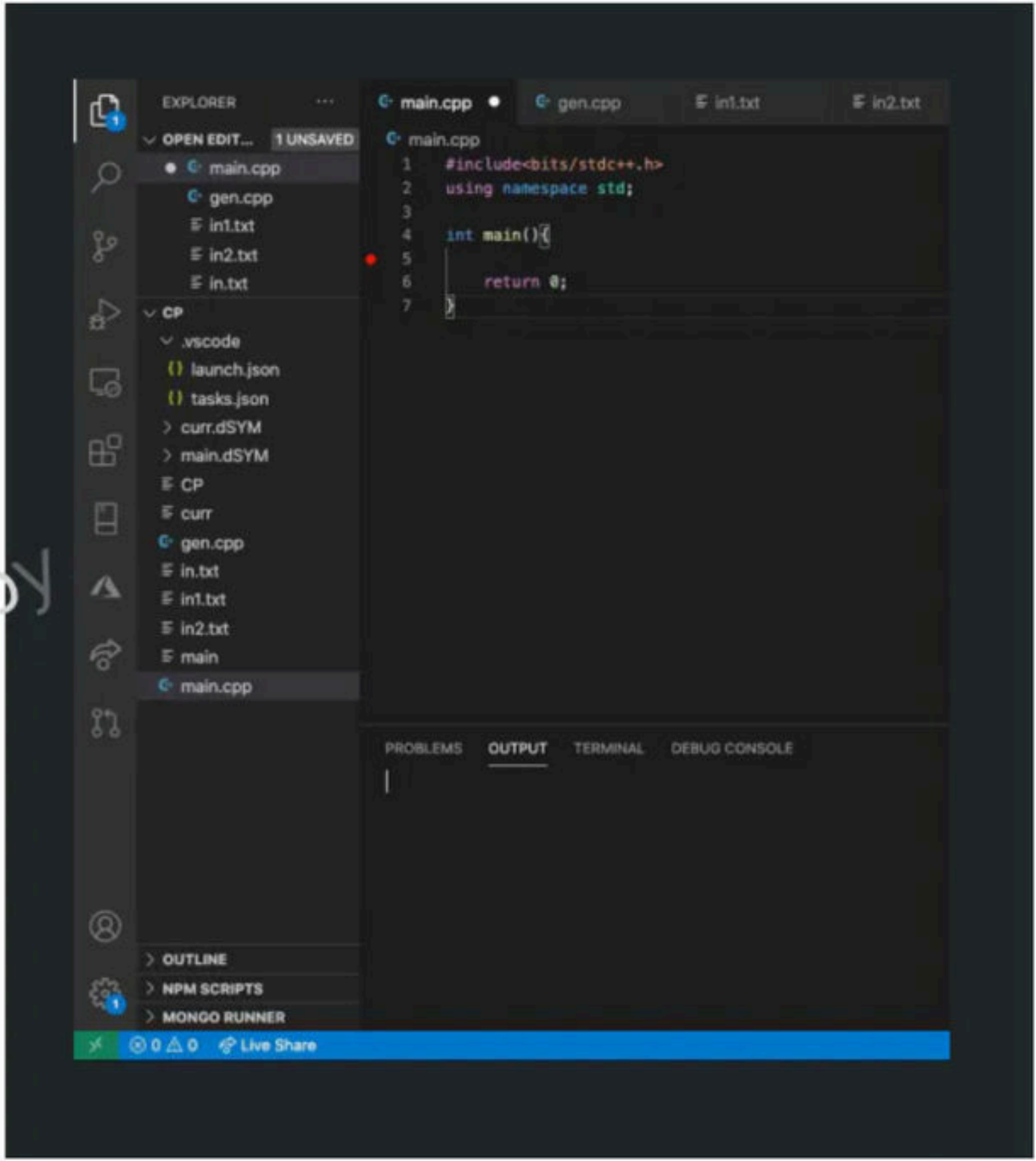
compile and run option, and many other features such as project manager, version control, debugger, etc.



IDE - Installation

CLion:
https://www.jetbrains.com/help/clion/installation-guide.html

VS Code:
https://code.visualstudio.com/download
Go to Extensions and Install C++ extension, you can also download github extensions for version control
https://code.visualstudio.com/docs/editor/extension-gallery



Getting Started with C++

Hello World Program

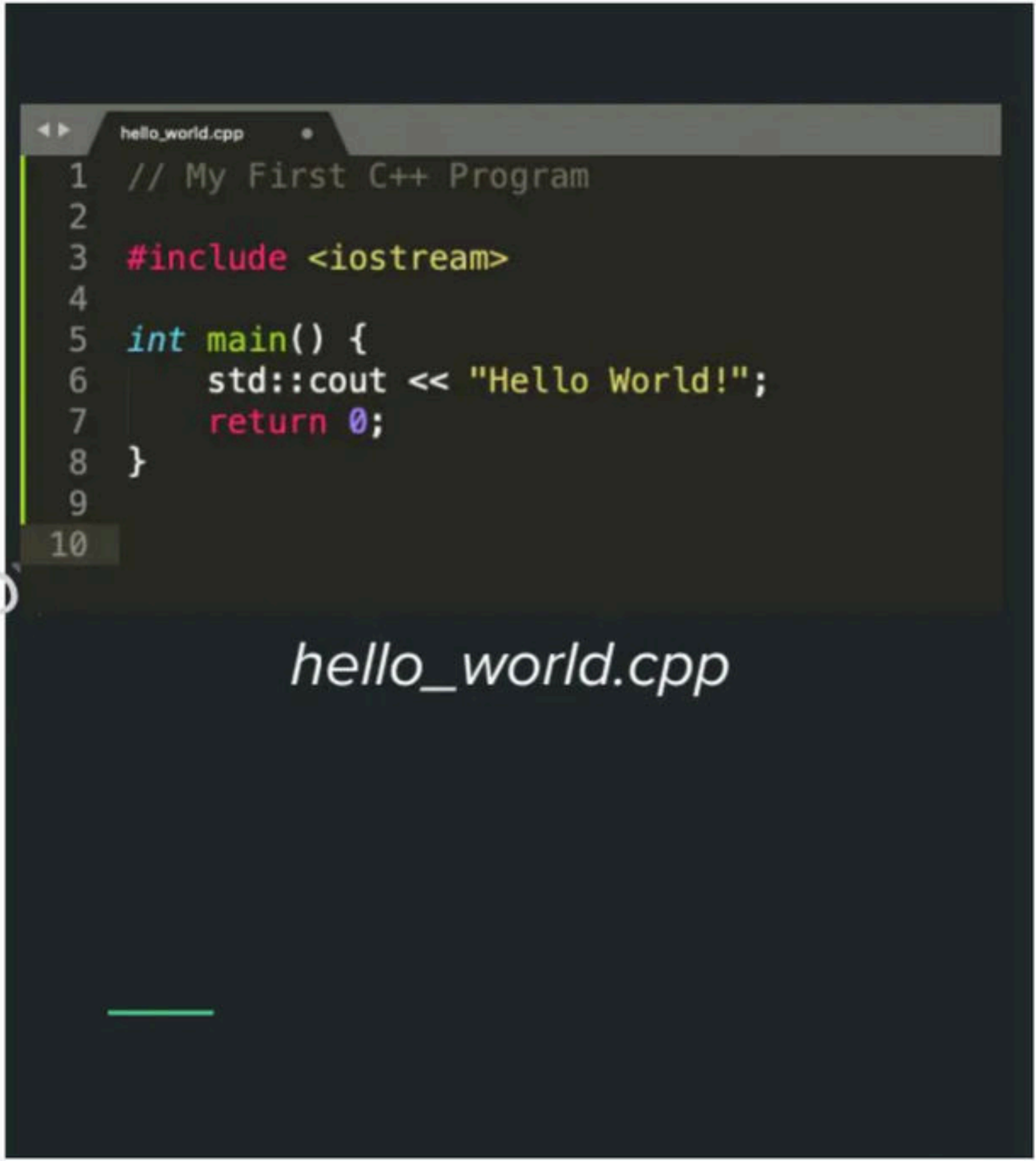
Ravindrababu Ravula
Jay Bansal

unacademy.com/@ravula
unacademy.com/@jay.bansal

"Hello World!" Program

1. In C++, any line starting with // is a comment. Comments are intended for the person reading the code to better understand the functionality of the program. It is completely ignored by the C++ compiler.

2. The **#include** is a preprocessor directive used to include files in our program. The above code is including the contents of the iostream file, it allows us to use **cout** in our program to print output on the screen.

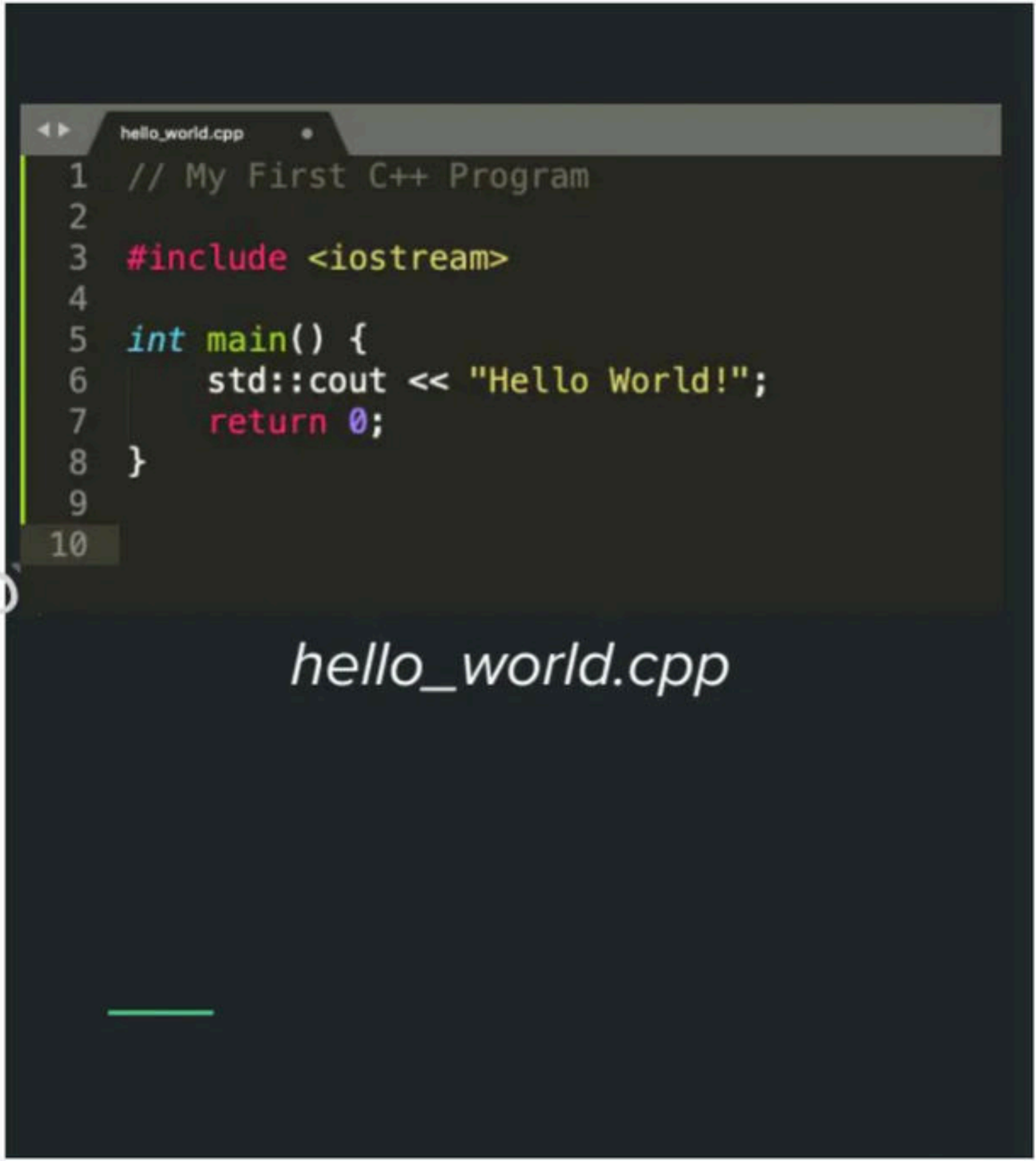


"Hello World!" Program

3. The execution of code beings from **main** function. A valid C++ program must have the main() function. The **curly braces** indicate the start and the end of the function.

4. **std::cout** prints the content inside the quotation marks. It is followed by << followed by the format string.

5. The **return 0;** statement is the "Exit status" of the program. In simple terms, the program ends with this statement.



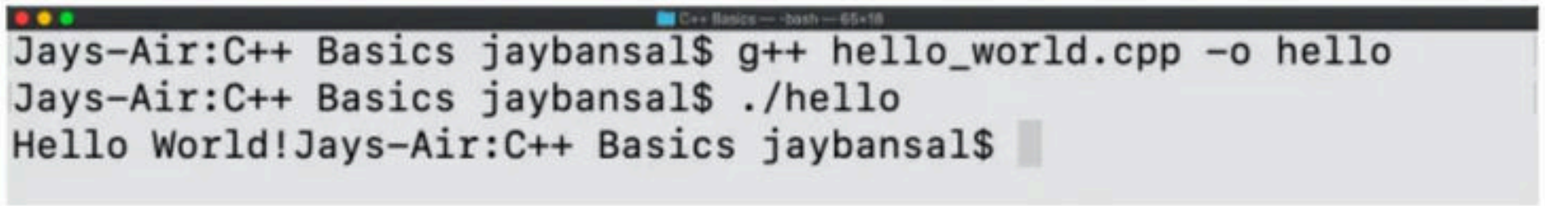
Running Hello World Program

1. We have named the file as `hello_world.cpp`

2. Now you have to open the Linux terminal and move to the directory where you have saved your file. Run the below command to **compile** your file:
   `g++ hello_world.cpp -o hello`

   - hello_world.cpp: name of your source code file
   - hello: can be any name of your choice, will be assigned to the executable file which is created by the compiler after compilation

3. Now to run your program you have to run the below command:
   `./hello`



Getting Started with C++

Basic user input and output

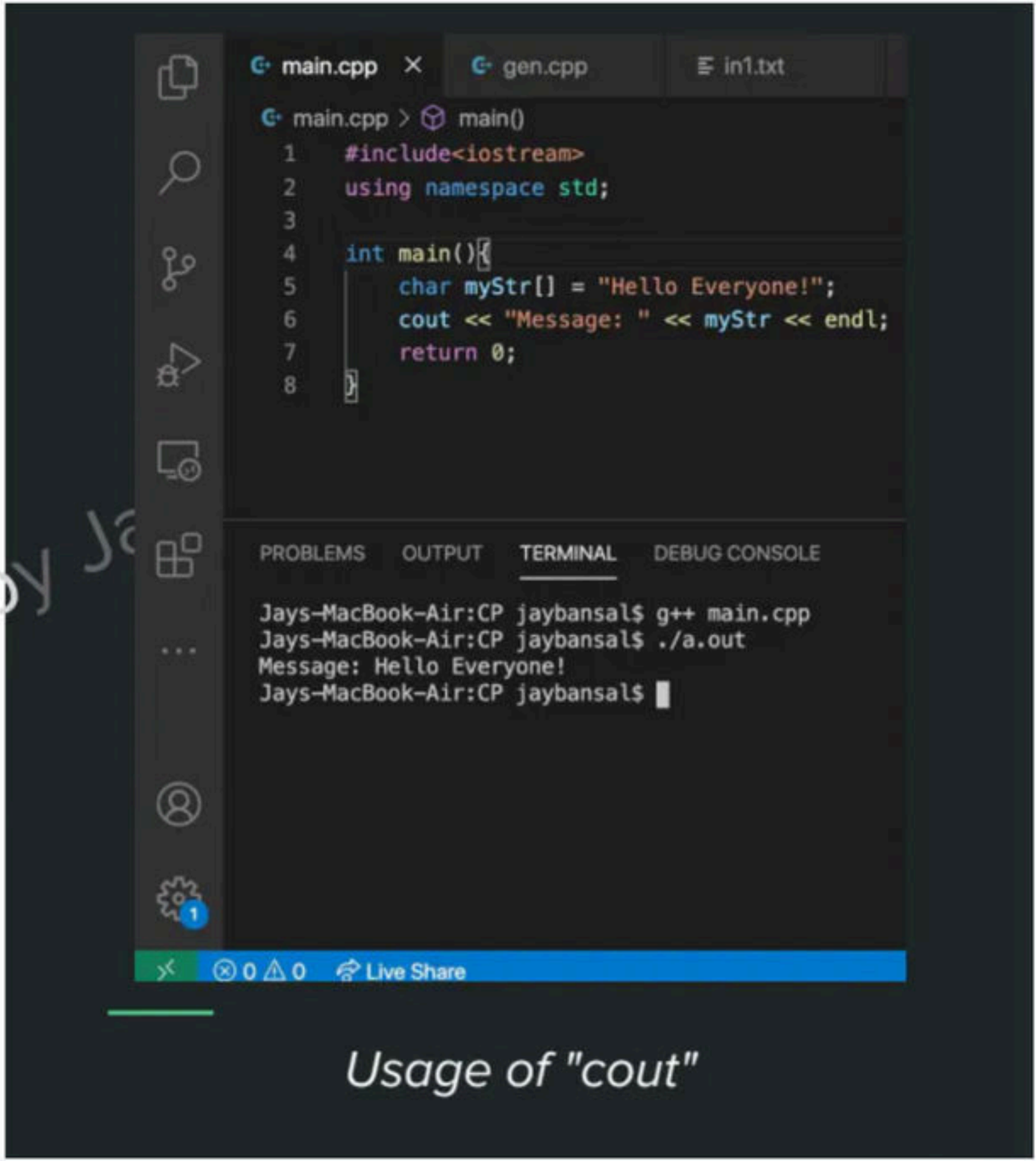
Ravindrababu Ravula
Jay Bansal

unacademy.com/@ravula
unacademy.com/@jay.bansal

Header Files & Functions

- **<iostream> :** C++ input/output streams are primarily defined by iostream, a header file that is part of the C++ standard library
- iostream uses the objects **cin**, **cout**, **cerr**, and **clog** for sending data to and from the standard streams
- As part of the C++ standard library, these objects are a part of the **std** namespace

- **cout** object is of type ostream. The cout object is said to be "connected to" the standard output device, which usually is the display screen. cout is used in conjunction with the stream insertion operator, which is written as << (overloads the left bit-shift operator)

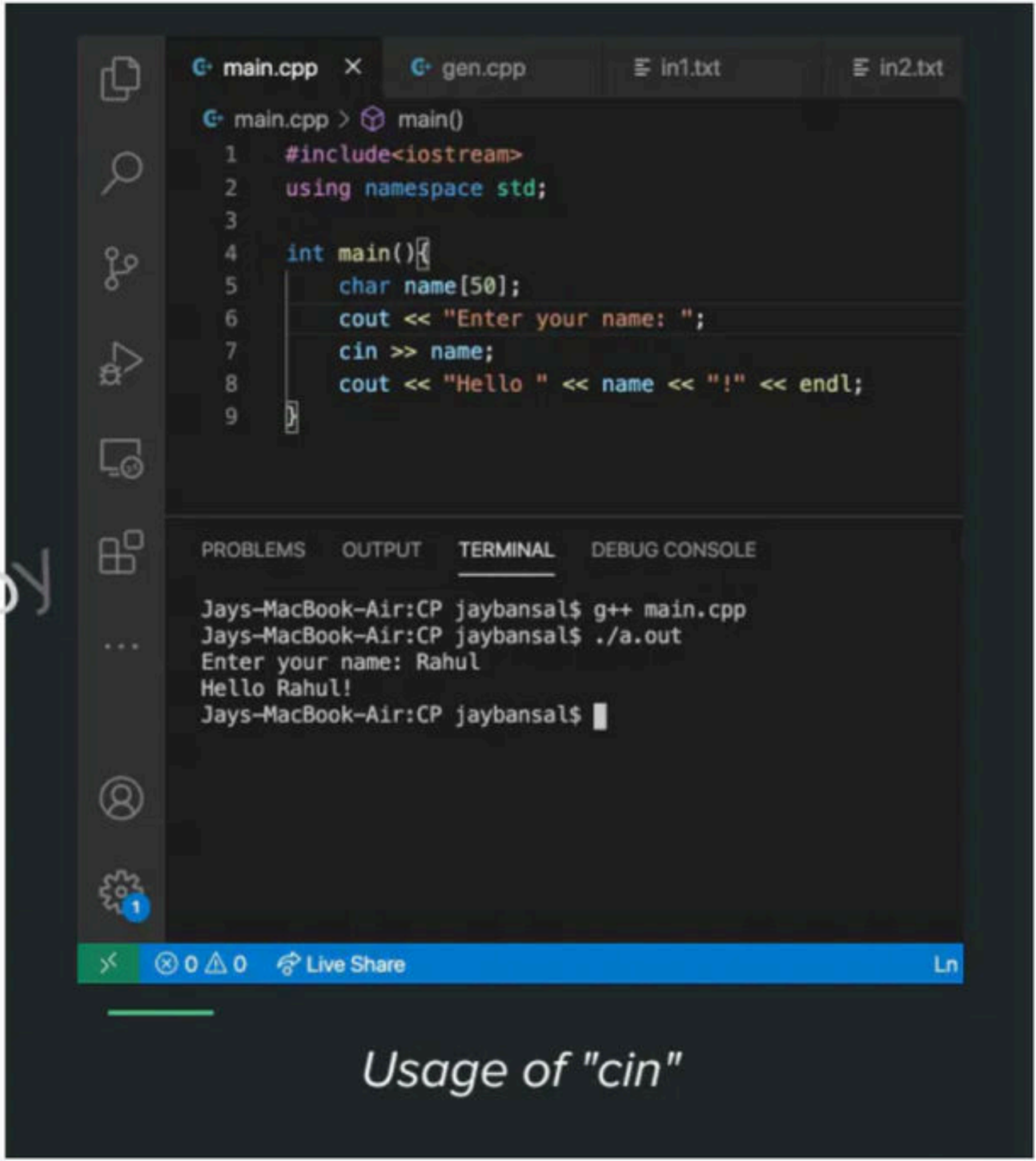


Usage of "cout"

Header Files & Functions

- **cin** object is of type istream, which overloads the right bit-shift operator (called the stream extraction operator), is an instance of istream class

endl ("end line"): inserts a newline into the stream and calls flush.



Usage of "cin"

Namespace in C++

- In computing, a namespace is a set of signs (names) that are used to identify and refer to objects of various kinds. Namespaces are commonly structured as hierarchies to allow reuse of names in different contexts.

- Namespaces are used to organize code into logical groups and to prevent name collisions that can occur especially when your code base includes multiple libraries.

- All identifiers at namespace scope are visible to one another **without qualification**. Identifiers outside the namespace can access the members by using the fully qualified name for identifiers,
  Eg. `std::vector<std::string> vec;`

- You can also use the "using Directive" for all the identifiers in the namespace `using namespace std;`

Getting Started with C++

C++ Data Types

Ravindrababu Ravula
Jay Bansal

unacademy.com/@ravula
unacademy.com/@jay.bansal

C++ variables & Data Types

- Variables are containers for storing data values.
- In C++, there are different **types** of variables (defined with different keywords)
  These are called the "primitive data types"

int - stores integers (whole numbers), without decimals

float, double - stores floating point numbers, with decimals

char - stores single characters, such as 'a' or 'A'.

Char values are surrounded by single quotes

string - stores text, such as "Hello World".

String values are surrounded by double quotes

bool - stores values with two states: true or false

void - void data type represents a valueless entity

Scope of variables

- All variables have two attributes, **scope** and **storage class**

- The scope of a variable is the area of the program where the variable is valid
  - A **global** variable is valid from the point it is declared to the end of the program
  - A **local** variable's scope is limited to the block where it is declared and **cannot** be accessed outside that block

It is possible to declare a local variable with the same name as a global variable
The declaration of the variable in the closest scope will be considered while using it.

By default, Global variables are initialized to 0 (False for bool)
By default, Local variables are initialized to garbage value

int - basic integer type

On 32/64 bit systems it is almost exclusively guaranteed to have width of at least 32 bits.

Limits:

signed: -2,147,483,648 to 2,147,483,647

unsigned: 0 to 4,294,967,295

long or long long: 64 bits (2^63-1 on either side ~ 9x10^18)

Default Value:

Local variable: undefined/garbage

Global variable: 0

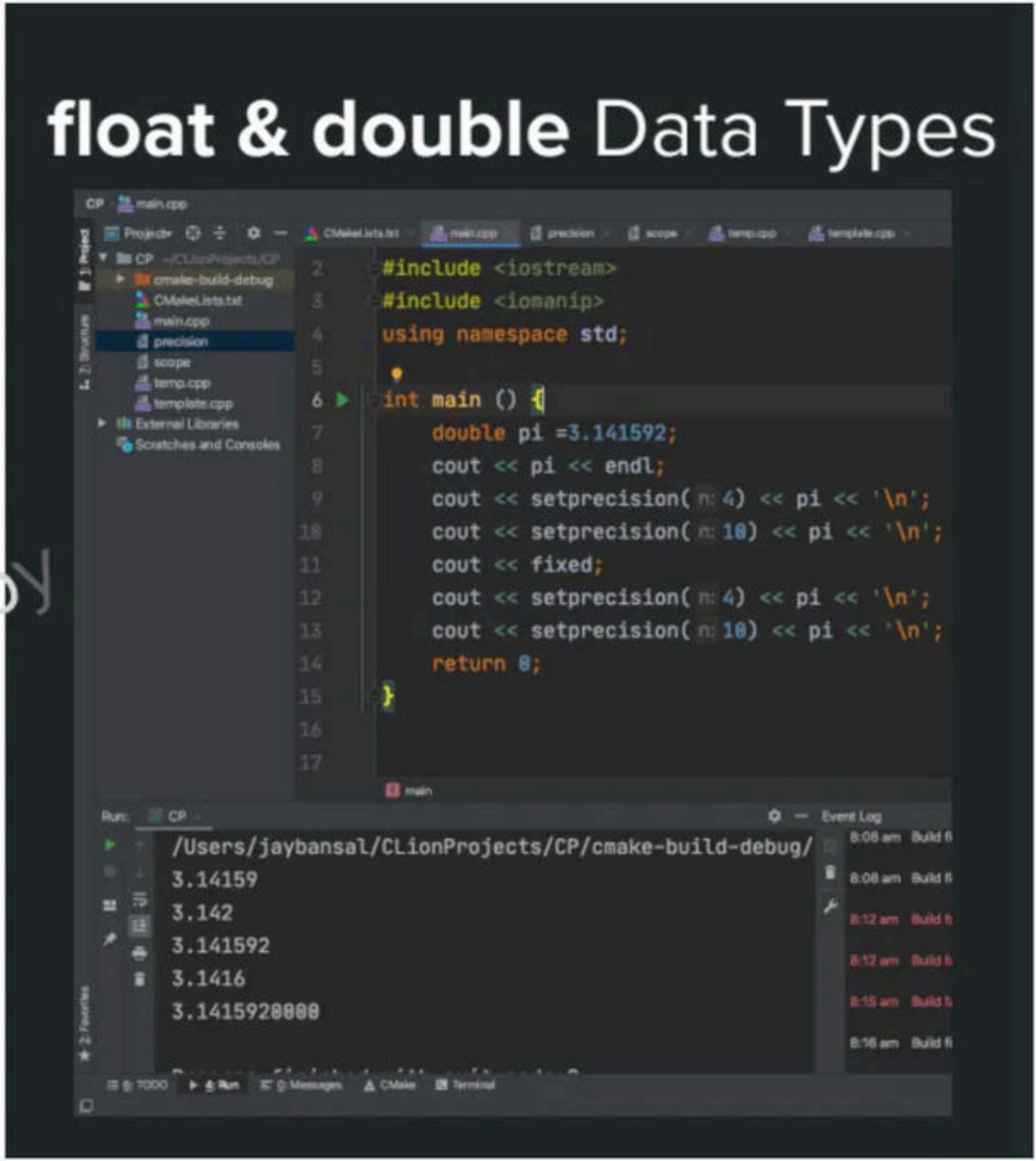
int Data Type (Very useful for Competitive Programming)

double: 64 bit precision(1 bit for the sign, 11 bits for the exponent, and 52 bits for the value), i.e. double has 15 decimal digits of precision.

float: 32 bit precision (8 bits for the exponent, and 23 for the value), i.e. float has 7 decimal digits of precision.

setprecision:
This manipulator is declared in header <iomanip>

Sets the decimal precision to be used to format floating-point values on output operations.



In order to use the string data type, the C++ string header <string> must be included at the top of the program

Basic operations:

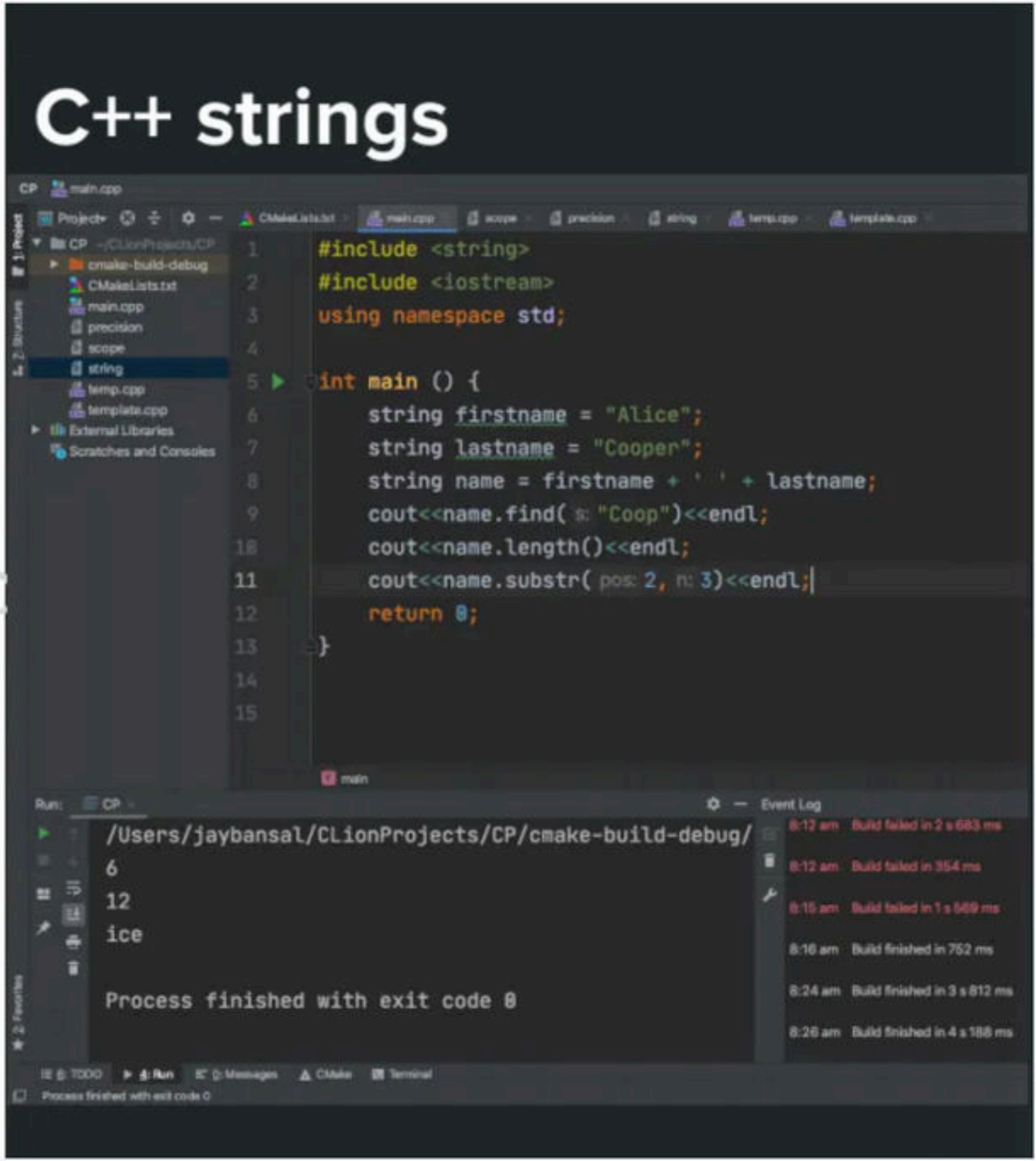
Counting the number of characters in a string: The **length** method returns the number of characters in a string; str.length()

Comparing two strings: You can compare two strings for equality using the == and != operators

Searching within a string: The string member function **find** is used to search within a string for a particular string or character.

str.find(key), str.find(key, n)

Extracting substrings: The **substr** member function creates substrings from pieces of the receiver string. **str.substr(start, length)** returns a new string consisting of the characters from str starting at the position start and continuing for length characters



const

If you make any variable as constant, using const keyword, you cannot change its value. Also, the constant variables must be initialized while they are declared.

```
const int a = 10;

const int b = i + 10;

a = b+1;    // Compile time error
```

Storage Classes of variables in C++

Storage Classes are used to describe the scope, visibility and lifetime which help us to trace the existence of a particular variable during the runtime of a program.

auto: The default storage class for all the variables declared inside a function or a block. Auto variables can be only accessed within the block/function they have been declared and not outside them.

Storage: Stack, Default Initial: Garbage, Scope: Within block, Life: End of the block

Storage Classes of variables in C++

static: used to declare static variables, having a property of preserving their value even after they are out of their scope.
Storage: Data segment, Default Initial: 0, Scope: Within block, Life: End of the program

Storage Classes of variables in C++

register: This storage class declares variables which like auto variables. Difference is that the compiler tries to store these variables in the register of the microprocessor if a free register is available
Storage: Register, Default Initial: Garbage, Scope: Within block, Life: End of the block

Storage Classes of variables in C++

extern: extern keyword extends the visibility of the variables and functions to the whole program, the function can be used (called) anywhere in any of the files of the whole program
Storage: Data segment, Default Initial: 0, Scope: Entire Program, Life: End of the program

Derived & User-Defined Data Types

Derived data types: The data-types that are derived from the primitive or built-in datatypes.
User-Defined data types: Data types that are defined by user itself.

- Arrays
- Pointers

- Structures
- Union
- Enum
- Typedef defined DataType

Getting Started with C++

Control Structures in C++

Ravindrababu Ravula
Jay Bansal

unacademy.com/@ravula
unacademy.com/@jay.bansal

Control Statements

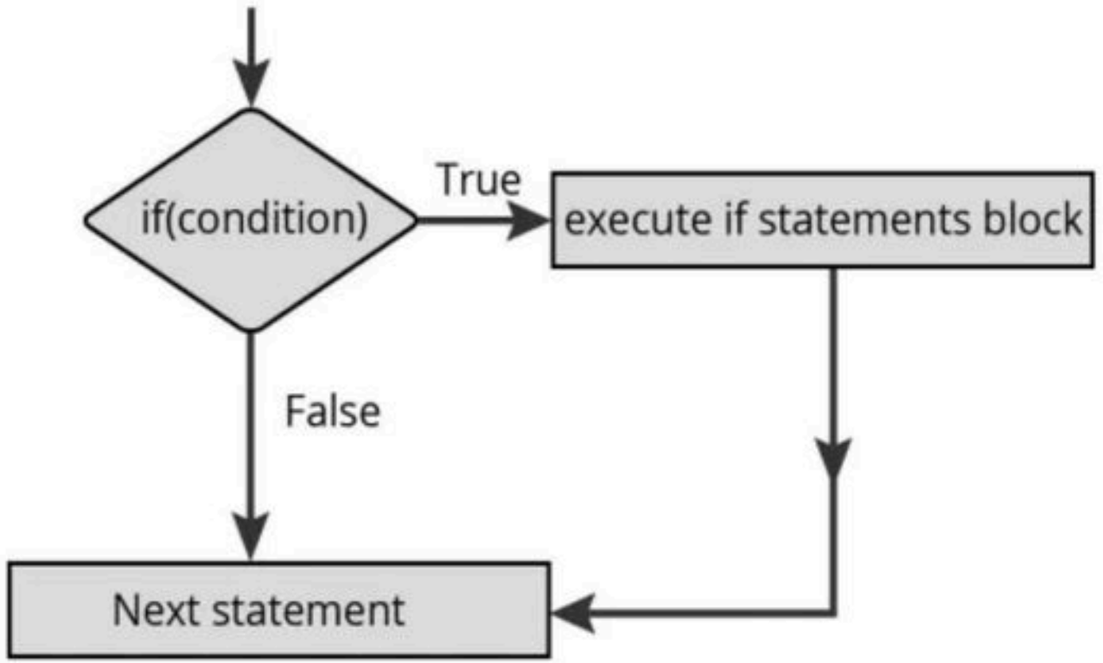
- Control statements are those which are used to control flow of execution of a program
- These don’t affect code at the compilation time

Types of Control Statements

Three Types :

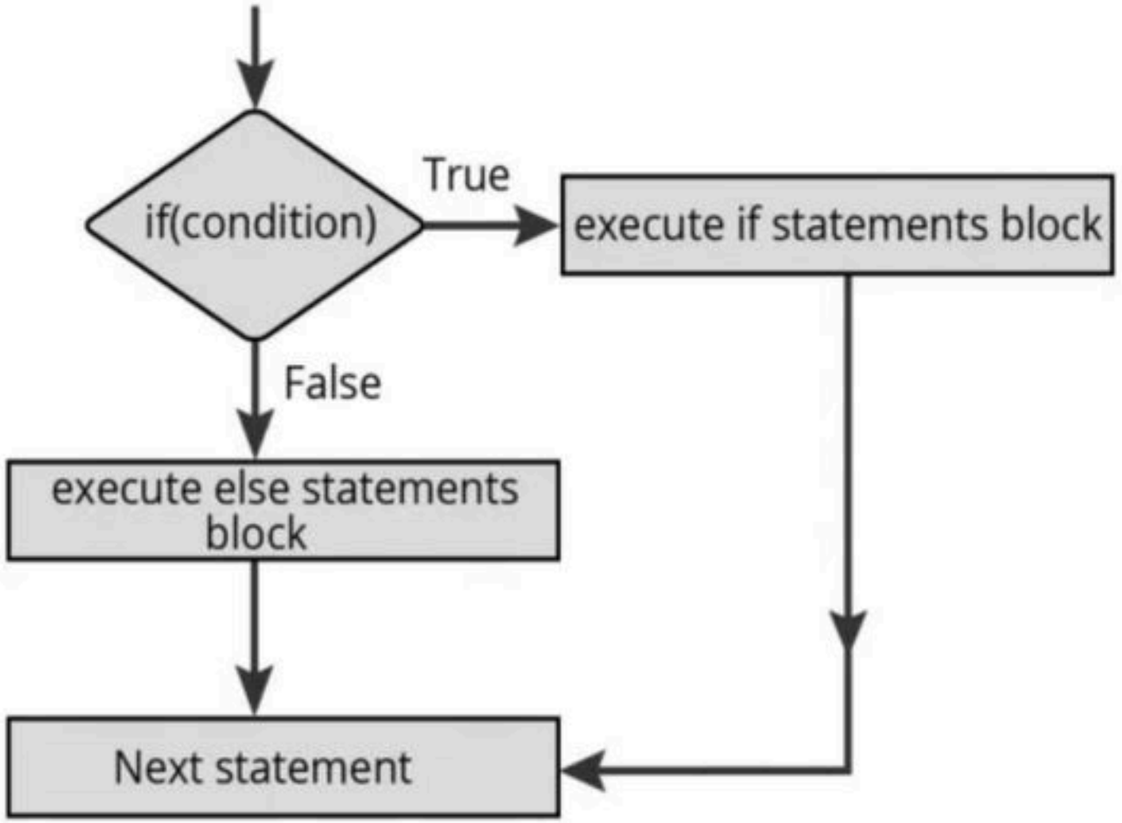
- Selection Statement
  - if
  - if/else
  - switch case
- Repetition Statement
  - for loop
  - while loop
  - do-while loop
- Sequence Statement
  - break
  - continue

- **If Statement**



Selection Statements (If Statement)

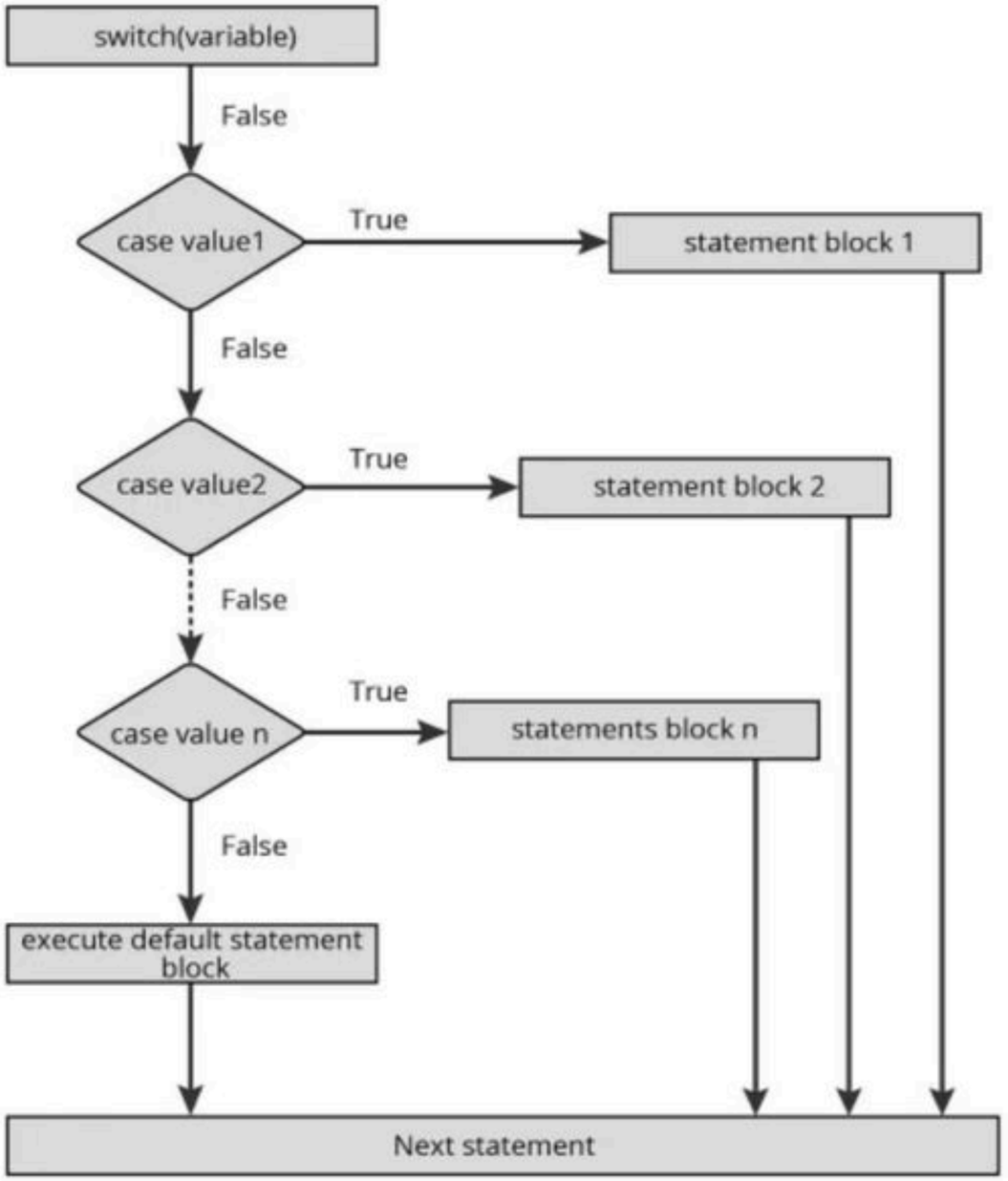
- **If/else Statement**



Selection Statements (If/else Statement)

Selection Statements

- **Switch Statement**
  - Allows control execution of the program based on a value/expression
  - Works as multiple if/else ifs or as nested if/else
  - Any number of cases can be there inside a switch
  - Variable in switch and case should have same data types
  - If no case matches, switch terminates with none case execution
  - For handling cases with no match, we can use default
  - Default is meant to place as last case so as not to miss other case match
  - Default and Break are optional for switch case statements



Sequence Statements

- **Break**
  - It is used to terminate cases' execution in sequence in switch case statements
  - On encounter of break inside any loop statement, loop terminates and the execution of program continues from just next statement after the loop

Selection Statements (break)

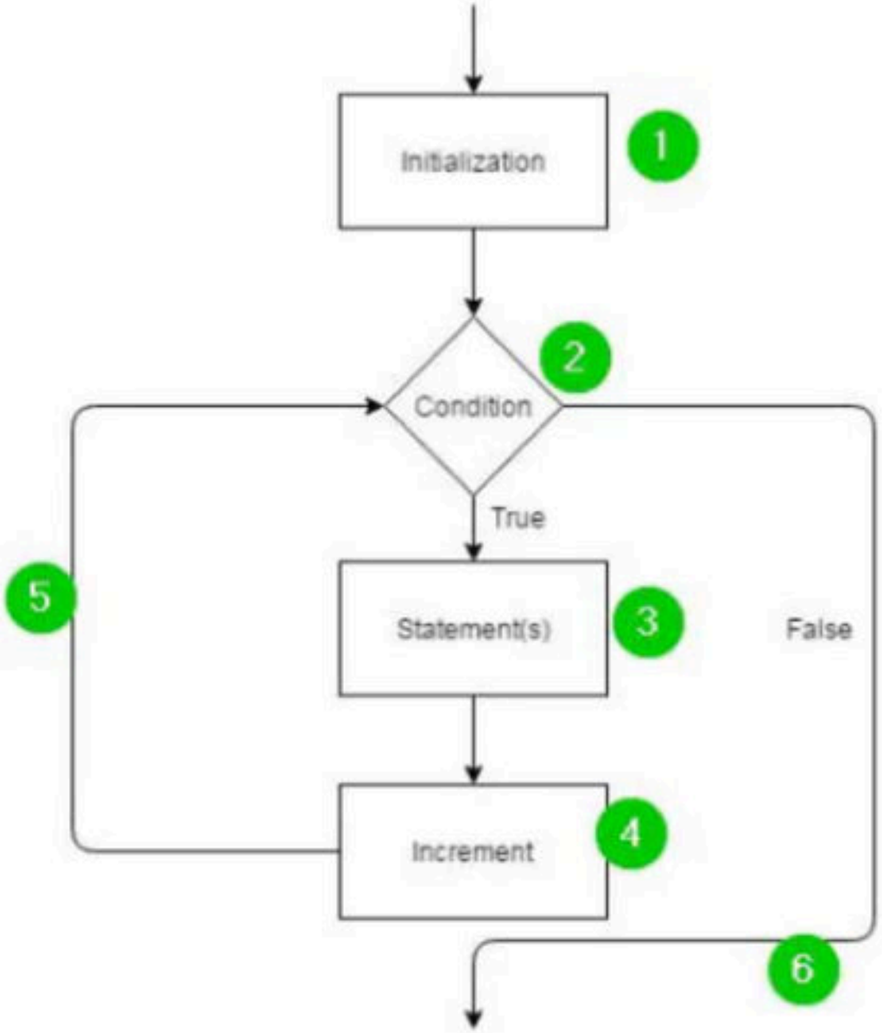
Sequence Statements

- **Continue**
  - used inside loops
  - instead of termination of loop like in break, it forces to skip iteration execution of lines placed just after the encounter of continue inside the loop
  - loop continues execution from next iteration

Selection Statements (continue)

Repetition Statements

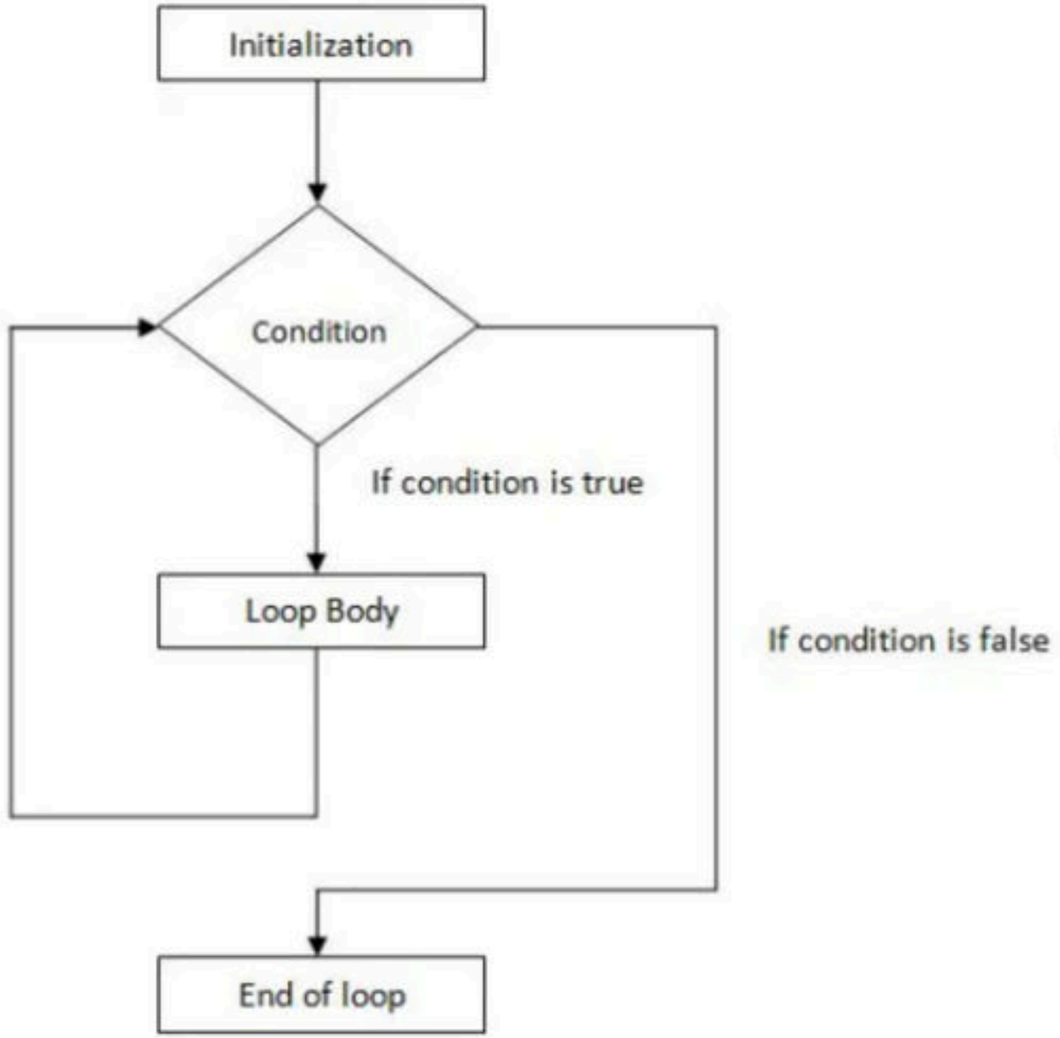
- **For loop**



Repetition Statements (For loop)

Sequence Statements

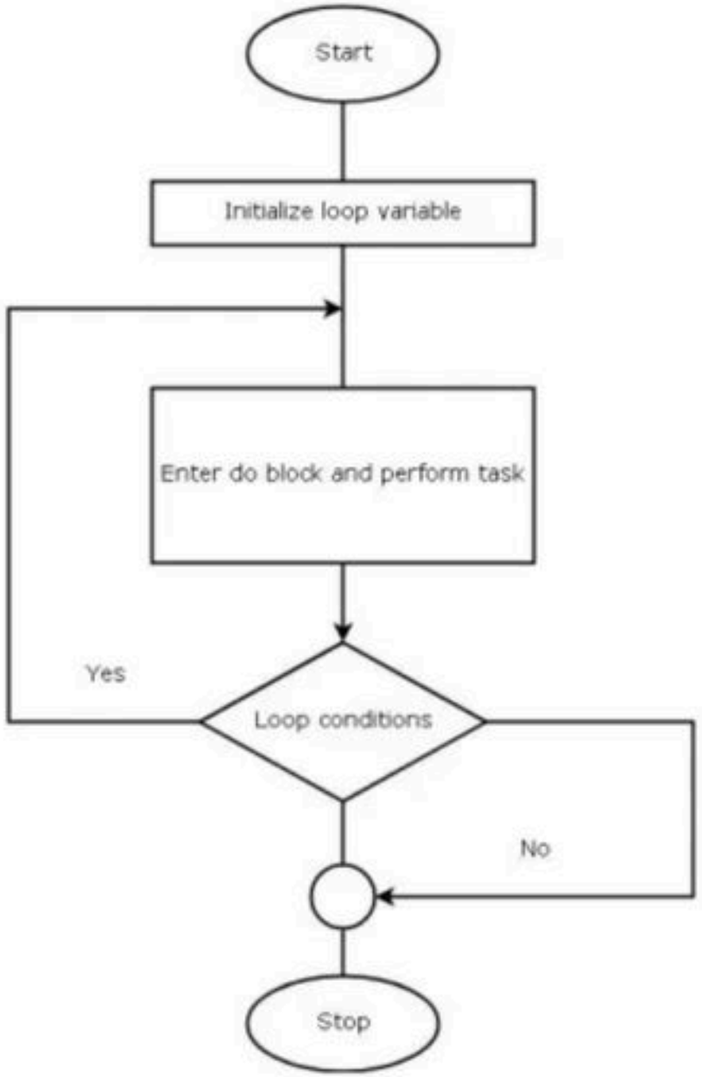
- **While loop**



Repetition Statements (while loop)

Sequence Statements

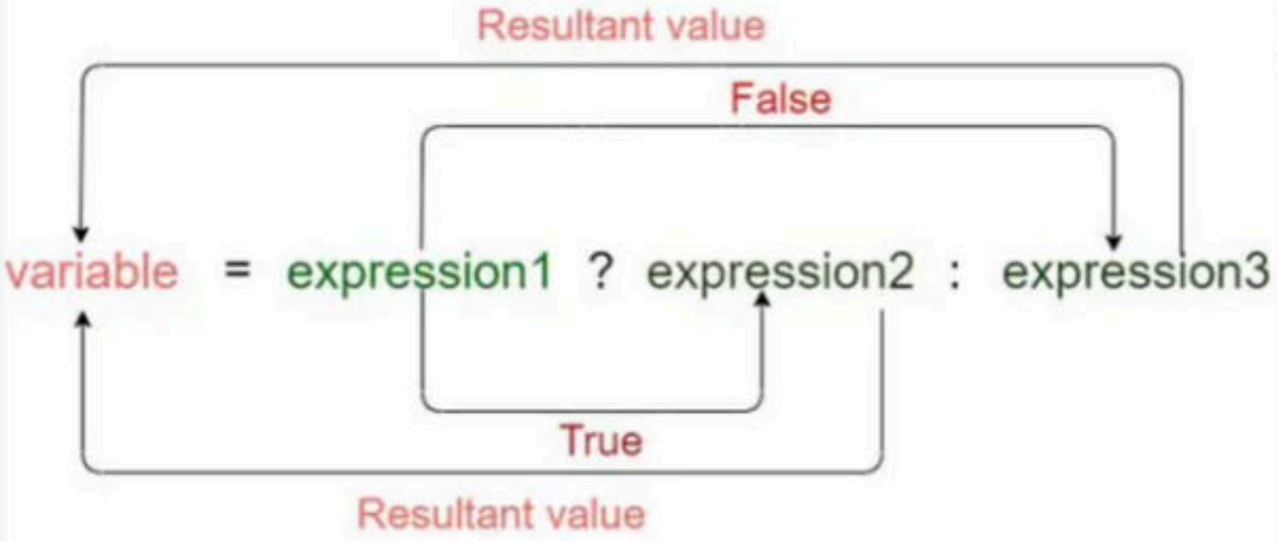
- **Do-while loop**



Repetition Statements (do-while loop)

Conditional Operator

- Also called Ternary Operator, because it has three operands
- It is a decision making statement
- Works like if/else only



Conditional Operator (ternary operator)

Problems

Problem 1 : Take n as input from user and write a program which works as follows :
if n is odd : evaluates and prints factorial value of n
if n is even : evaluates and prints sum of all the whole numbers upto and including n

Problem 2 : Take n as input from user and write a program to evaluate and print sum of all the digits present in n.

Problems

Problem 3 : Take income and number of years with company as input from user and write a program which updates income in the following manner :

- If years of work with the company are more than 5 (or equal)
  - If income is more than 100 and less than 200, 30% increment
  - If income is more than 200 and less than 300, 25% increment
  - If income is more than 300 and less than 500, 20% increment
  - If income is more than 500 15% increment, 15% increment
- If years of work with the company are less than 5 but greater than 2
  - If income is more than 100 and less than 200, 25% increment
  - If income is more than 200 and less than 300, 20% increment
  - If income is more than 300 and less than 500, 15% increment
  - If income is more than 500 15% increment, 10% increment
- If years of work with the company are less than 2 (or equal)
  - If income is more than 100 and less than 200, 20% increment
  - If income is more than 200 and less than 300, 15% increment
  - If income is more than 300 and less than 500, 10% increment
  - If income is more than 500 15% increment, 5% increment

Getting Started with C++

C++ Tokens & Operators

Ravindrababu Ravula
Jay Bansal

unacademy.com/@ravula
unacademy.com/@jay.bansal

Tokens in C++

A token is the smallest element of a program that is meaningful to the compiler.

Tokens classification:

- Keywords
- Identifiers
- Constants
- Strings
- Operators
- Special Symbols

Keywords

Keywords in C++ refer to the pre-existing, reserved words and has a specific function associated with it.

We cannot use C++ keywords for assigning variable names as it would suggest a totally different meaning entirely and would be incorrect

95 reserved words in C++

alignas	alignof	asm	auto	bool	break
case	catch	char	char16_t	char32_t	class
const	constexpr	const_cast	continue	decltype	default
delete	double	do	dynamic_cast	else	enum
explicit	export	extern	FALSE	float	for
friend	goto	if	inline	int	long
mutable	namespace	new	noexcept	nullptr	operator
private	protected	public	register	reinterpret_cast	return
short	signed	sizeof	static	static_assert	static_cast
struct	switch	template	this	thread_local	throw
TRUE	try	typedef	typeid	typename	union
unsigned	using	virtual	void	volatile	wchar_t
while	–	–	–	–	–

C++ Identifiers

C++ allows the programmer to assign names of his own choice to variables, arrays, functions, structures, classes, and various other data structures called identifiers

Rules for C++ Identifiers:

- **First character**: The first character of the identifier in C++ should positively begin with either an alphabet or an underscore. It means that it strictly cannot begin with a number
- **No special characters**: C++ does not encourage the use of special characters while naming an identifier. It is evident that we cannot use special characters like the “!” or the “@” symbol.

C++ Identifiers

Rules for C++ Identifiers:

- **No keywords**: Using keywords as identifiers in C++ is strictly forbidden, as they are reserved words that hold a special meaning to the C++ compiler
- **No white spaces**: Leaving a gap between identifiers is discouraged. White spaces incorporate blank spaces, newline, carriage return, and horizontal tab
- **Word limit**: The use of an arbitrarily long sequence of identifier names is restrained. The name of the identifier **must not exceed 31 characters**, otherwise, it would be insignificant

- Note: In C++, uppercase and lowercase characters connote different meanings (Case sensitive)

C++ Identifiers

Example:

Which of the following is a correct identifier in C++?

a) 7var_name

b) 7VARNAME

c) VAR_1234

d) FALSE

C++ Identifiers

Example:

Which of the following is a correct identifier in C++?

a) 7var_name

b) 7VARNAME

c) VAR_1234

d) FALSE

Special Symbols

Apart from letters and digits, there are some special characters in C++ which help you manipulate or perform data operations:

- [] The opening and closing brackets of an array symbolize single and multidimensional subscripts

- **()** The opening and closing brackets represent function declaration and calls.

- {} The opening and closing curly brackets to denote the start and end of a particular fragment of code which may be functions

- **, (comma)** separates more than one statements, like in the declaration of different variable names

Special Symbols

Apart from letters and digits, there are some special characters in C++ which help you manipulate or perform data operations:

- **# (hash)** The hash symbol represents a preprocessor directive used for denoting the use of a header file
- * **(asterisk)** used in various respects such as to declare pointers, used as an operand for multiplication
- **~ (tilde)** We use the tilde symbol as a destructor to free memory
- **. (period)** use the dot operator to access a member of a structure

Getting Started with C++

C++ Operators

Ravindrababu Ravula
Jay Bansal

unacademy.com/@ravula
unacademy.com/@jay.bansal

Arithmetic Operators

Operator	Name	Description	Example
+	Addition	Adds together two values	a+b
-	Subtraction	Subtracts one value from another	a-b
*	Multiplication	Multiplies two values	a*b
/	Division	Divides one value by another	a/b
%	Reminder	Returns the division remainder	a%b
++	Increment	Increases the value of a variable by 1	a++
--	Decrement	Decreases the value of a variable by 1	a--

Bitwise Operators

Operator	Name	Description	Example
&	Bitwise AND	AND on every bit of two numbers	a&b
|	Bitwise OR	OR on every bit of two numbers	a|b
^	Bitwise XOR	XOR on every bit of two numbers	a^b
<<	Left shift	left shifts the bits of the first operand, the second operand decides the number of places to shift	a<<3
>>	Right shift	right shifts the bits of the first operand, the second operand decides the number of places to shift	a>>3
~	Bitwise NOT	Takes 1's complement of the number	~a

Assignment Operators

Operator	Example	Equivalent to
=	a = 3	a = 3
+=	a += 3	a = a+3
-=	a -= 3	a = a-3
*=	a *= 3	a = a*3
/=	a /= 3	a = a/3
%=	a %= 3	a = a%3
|=	a |= 3	a = a|3
^=	a ^= 3	a = a^3
>>=	a >>= 3	a = a>>3
<<=	a <<= 3	a = a<<3

Comparison Operators

Operator	Name	Example
==	Equal to	a==b
!=	Not equal to	a!=b
>	Greater than	a>b
<	Less than	a<b
>=	Greater than or equal to	a>=b
<=	Less than or equal to	a<=b

Comparison Operators

Three-way comparison operator, <==> (C++ 20)

Also called "**Space Ship operator**"

- (A <=> B) < 0 is true if A < B
- (A <=> B) > 0 is true if A > B
- (A <=> B) == 0 is true if A and B are equal/equivalent.

eg.

```
if (3<=>4 < 0)
        cout<<"Yes"<<endl;
```

Logical Operators

Operator	Name	Description	Example
&&	Logical and	Returns true if both statements are true	a==b && b!=c
||	Logical or	Returns true if any of the statements is true	a!=b || b>c
!	Logical not	Returns true if the result is false	! (a>b)

C++ operator precedence & associativity

Precedence	Operator	Description	Associativity
1	`::`	Scope resolution	Left-to-right
2	`a++` `a--`	Suffix/postfix increment and decrement	
	type`()` *type*`{}`	Functional cast	
	`a()`	Function call	
	`a[]`	Subscript	
	`.` `->`	Member access	
3	`++a` `--a`	Prefix increment and decrement	Right-to-left
	`+a` `-a`	Unary plus and minus	
	`!` `~`	Logical NOT and bitwise NOT	
	(*type*)	C-style cast	
	`*a`	Indirection (dereference)	
	`&a`	Address-of	
	`sizeof`	Size-of[note 1]	
	`co_await`	await-expression (C++20)	
	`new` `new[]`	Dynamic memory allocation	
	`delete` `delete[]`	Dynamic memory deallocation	

C++ operator precedence & associativity

4	`.*` `->*`	Pointer-to-member	Left-to-right
5	`a*b` `a/b` `a%b`	Multiplication, division, and remainder	
6	`a+b` `a-b`	Addition and subtraction	
7	`<<` `>>`	Bitwise left shift and right shift	
8	`<=>`	Three-way comparison operator (since C++20)	
9	`<` `<=` `>` `>=`	For relational operators < and ≤ respectively For relational operators > and ≥ respectively	
10	`==` `!=`	For relational operators = and ≠ respectively	
11	`&`	Bitwise AND	
12	`^`	Bitwise XOR (exclusive or)	
13	`|`	Bitwise OR (inclusive or)	
14	`&&`	Logical AND	
15	`||`	Logical OR	

C++ operator precedence & associativity

16	`a?b:c`	Ternary conditional[note 2]	Right-to-left
	`throw`	throw operator	
	`co_yield`	yield-expression (C++20)	
	`=`	Direct assignment (provided by default for C++ classes)	
	`+=` `-=`	Compound assignment by sum and difference	
	`*=` `/=` `%=`	Compound assignment by product, quotient, and remainder	
	`<<=` `>>=`	Compound assignment by bitwise left shift and right shift	
	`&=` `^=` `|=`	Compound assignment by bitwise AND, XOR, and OR	
17	`,`	Comma	Left-to-right

C++ operator precedence & associativity

Example 1

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

int main() {
  int a = 1;
  int b = 4;

  b += a -= 6;

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

C++ operator precedence & associativity

Example 2

```cpp
#include <iostream>
using namespace std;
int main()
{
    int a = 10, b=22, c=2, x;

    x = a == b != c;
    cout<<"The result of the first expression is = "<< x <<endl;

    x = a == ( b != c );
    cout<<"The result of the second expression is = "<<x <<endl;

    return 0;
}
```

C++ operator precedence & associativity

Example 3

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5      int a = 1, b = 3;
6
7      b += a++ + a++ + --a;
8      cout<< "a = " << a << ", b = " << b << endl;
9
10     return 0;
11 }
12
```

Getting Started with C++

Functions

Ravindrababu Ravula
Jay Bansal

unacademy.com/@ravula
unacademy.com/@jay.bansal

Function

- A block of code with a unique name performing some specific task, which can be used multiple times throughout the execution
- The most important function in c++ is “main” function
- Main function is the entry point of every c++ program execution, because the compiler commands the thread to start execution from main function
- Using functions is a good practice as it provides features of modularity and reusability

Function Structure

```
return_type function_name(parameter1_type p1, parameter2_type p2, ...)
{


    return_type x;
}
```

Function Prototype

- Function prototype only mentions the interface of the function hiding the details of task to be done by the function
- Function prototype is declaration of the function which specifies the following:
  - Function name
  - Data types of the parameters
  - Return type of the function
- Function prototypes are useful for forward declarations (declaring prior to the definition)
  - e.g. circular recursive program
- e.g.

```
int function(int, char, float, int);
```

Function Definition

- Function definition consists of two parts :
  - Function header : Same as function prototype, includes the parameters' variable names as well
  - Function body : Code inside the curly braces written to perform that specific task the function is meant for
  - e.g.

```
int function(int a, char b, float c, int d)   → function header
{
return a+b+c+d;                               } function body
}
```

Calling a Function

- Function call is made when we require to perform the task the function is accomplishing
- While calling the function, we pass real values in accordance with the data type list of parameters of function prototype
- These real values which are passed at function calling are known as **arguments/actual parameters** to the function
- e.g.

```
int out = function(5, 'a', 1.167, 2021)
```

Examples

Function 1 : Write a function to find number of vowels in a string

Function 2 : Write a function to check if a string is a palindrome or not

Calling a Function

There are three ways of calling a function :

1. Call by Value
2. Call by Reference
3. Call by Pointer

Actual and Formal Parameters

Actual parameters : Arguments or values passed to the function while calling

Formal parameters : Variables used by the function to store received values to that function

formal parameters

```
int function(int a, char b, float c, int d)
{
return a+b+c+d;
}

int out = function(5, 1.167, 'a', 2021)
```

actual parameters

Call by Value

- This method of calling **copies the values** from actual parameters to the formal parameters
- Both actual and formal parameters hence take separate memories of their own

```cpp
#include<iostream>
using namespace std;
void swap(int a, int b){
    int temp = a;
    a = b;
    b = temp;
}
int main(){
    int a, b;
    cin>>a>>b;
    cout<<"Values prior to swap (a, b): ("<<a<<", "<<b<<")"<<endl;
    swap(a, b);
    cout<<"Values post to swap (a, b): ("<<a<<", "<<b<<")"<<endl;
    return 0;
}
```

Call by Reference

- In this method of calling actual parameter and corresponding formal parameter, both point to the same memory location
- & is used with the variable name in formal parameter, it creates alias (another name of the same actual parameter)
- Changes made hence effect to the same variable placed at that one location

```cpp
#include<iostream>
using namespace std;
void swap(int &a, int &b){
    int temp = a;
    a = b;
    b = temp;
}
int main(){
    int a, b;
    cin>>a>>b;
    cout<<"Values prior to swap (a, b): ("<<a<<", "<<b<<")"<<endl;
    swap( &a, &b);
    cout<<"Values post to swap (a, b): ("<<a<<", "<<b<<")"<<endl;
    return 0;
}
```

Call by Pointer

- In this method of calling instead of passing variable, address of the memory location that variable is passed
- In formal parameters, pointer is used to store the address
- Hence the changes are all made in the variable placed at that particular memory location only

Examples

Guess the output!

```
#include<iostream>
using namespace std;
int fun(int &a, int b, int c){
    a += 10;
    b = b - ++c;
    c--;
    return a+b+c;

}
int main(){
    int a, b;
    cin>>a>>b;
    cout<<fun(a, b, a)<<" "<<a<<" "<<b<<endl;
    return 0;
}

Output on a = 2, b = 3 ???
```

Examples

Guess the output!

14 12 3

```
#include<iostream>
using namespace std;
int fun(int &a, int b, int c){
    a += 10;
    b = b - ++c;
    c--;
    return a+b+c;

}
int main(){
    int a, b;
    cin>>a>>b;
    cout<<fun(a, b, a)<<" "<<a<<" "<<b<<endl;
    return 0;
}

Output on a = 2, b = 3 ???
```

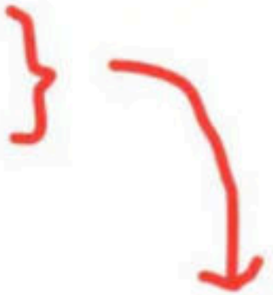
Inline Function

- Inline function are used to save the function switching time at calling
- Keyword “inline” is used just before the function header
- When call is made to an inline function from somewhere, instead of switching function flow, compiler puts the code lines (body) of called function directly in the caller function
- This is used when function definition is considerably small or when function calls are very less in number to that function
- If the code is complex, compiler can also ignore to make function inline by itself, this is compiler implementation dependent

Inline Function

Function definition

```
inline int function(int a, char b, float c, int d)
{
return a+b+c+d;
}
```



Function call

```
int out = function(5, 1.167, 'a', 2021)
```

```
5+1.167+'a'+ 2021
```

Default Arguments

```cpp
#include<iostream>
using namespace std;
int fun(int a, int b){
    return a*b;
}
int main(){
    int a, b;
    cin>>a>>b;
    cout<<fun(a, b)<<endl;
    return 0;
}
```

➔ a = 3, b = 4
OUTPUT ?

Default Arguments

```cpp
#include<iostream>
using namespace std;
int fun(int a, int b){
    return a*b;
}
int main(){
    int a, b;
    cin>>a>>b;
    cout<<fun(a, b)<<endl;
    return 0;
}
```

→ a = 3, b = 4
OUTPUT ?

12

Default Arguments

```cpp
#include<iostream>
using namespace std;
int fun(int a, int b){
    return a*b;
}
int main(){
    int a, b;
    cin>>a>>b;
    cout<<fun(b)<<endl;
    return 0;
}
```

➔ a = 3, b = 4
OUTPUT ?

Default Arguments

```
#include<iostream>
using namespace std;
int fun(int a, int b){
    return a*b;
}
int main(){
    int a, b;
    cin>>a>>b;
    cout<<fun(b)<<endl;
    return 0;
}
```

➔ a = 3, b = 4
OUTPUT ?

error: no matching function for call to 'fun'
(Compile time error)

Default Arguments

```
#include<iostream>
using namespace std;
int fun(int a = 10, int b){
    return a*b;
}
int main(){
    int a, b;
    cin>>a>>b;
    cout<<fun(b)<<endl;
    return 0;
}
```

➔ a = 3, b = 4
OUTPUT ?

Default Arguments

```
#include<iostream>
using namespace std;
int fun(int a = 10, int b){
    return a*b;
}
int main(){
    int a, b;
    cin>>a>>b;
    cout<<fun(b)<<endl;
    return 0;
}
```

➔ a = 3, b = 4
OUTPUT ?

error: missing default argument on parameter 'b'
error: no matching function for call to 'fun'
(2 Compile time errors)

Default Arguments

```cpp
#include<iostream>
using namespace std;
int fun(int a, int b=10){
    return a*b;
}
int main(){
    int a;
    cin>>a;
    cout<<fun(a)<<endl;
    return 0;
}
```

➔ a = 3
OUTPUT ?

Default Arguments

```cpp
#include<iostream>
using namespace std;
int fun(int a, int b=10){
    return a*b;
}
int main(){
    int a;
    cin>>a;
    cout<<fun(a)<<endl;
    return 0;
}
```

➔ a = 3
OUTPUT ?

30

Default Arguments

```cpp
#include<iostream>
using namespace std;
int fun(int a, int b=10){
    return a*b;
}
int main(){
    int a;
    cin>>a;
    cout<<fun(a, a+20)<<endl;
    return 0;
}
```

➔ a = 3
OUTPUT ?

Default Arguments

```cpp
#include<iostream>
using namespace std;
int fun(int a, int b=10){
    return a*b;
}
int main(){
    int a;
    cin>>a;
    cout<<fun(a, a+20)<<endl;
    return 0;
}
```

➔ a = 3
OUTPUT ?

69

Default Arguments

```cpp
#include<iostream>
using namespace std;
int fun(int a, int b=10, int c){
    return a*b+c;
}
int main(){
    int a;
    cin>>a;
    cout<<fun(a, a+20)<<endl;
    return 0;
}
```

→ a = 3
OUTPUT ?

Default Arguments

```
#include<iostream>
using namespace std;
int fun(int a, int b=10, int c){
    return a*b+c;
}
int main(){
    int a;
    cin>>a;
    cout<<fun(a, a+20)<<endl;
    return 0;
}
```

→ a = 3

OUTPUT ?

error: missing default argument on parameter 'c'

error: no matching function for call to 'fun'

(2 compile time errors)

Default Arguments

```cpp
#include<iostream>
using namespace std;
int fun(int a, int c, int b=10){
    return a*b+c;
}
int main(){
    int a;
    cin>>a;
    cout<<fun(a, a+20)<<endl;
    return 0;
}
```

➔ a = 3
OUTPUT ?

Default Arguments

```cpp
#include<iostream>
using namespace std;
int fun(int a, int c, int b=10){
    return a*b+c;
}
int main(){
    int a;
    cin>>a;
    cout<<fun(a, a+20)<<endl;
    return 0;
}
```

→ a = 3
OUTPUT ?

53

Default Arguments

```
#include<iostream>
using namespace std;
int fun(int a=10, int b=20, int c=30){
    cout<<"a: "<<a<<", b: "<<b<<", c: "<<c<<endl;
    return a*b+c;
}
int main(){
    cout<<fun()<<endl;
    cout<<fun(4)<<endl;
    cout<<fun(4, 5)<<endl;
    cout<<fun(4, 5, 6)<<endl;
    return 0;
}
```

GUESS OUTPUT ?

Default Arguments

```
#include<iostream>
using namespace std;
int fun(int a=10, int b=20, int c=30){
    cout<<"a: "<<a<<", b: "<<b<<", c: "<<c<<endl;
    return a*b+c;
}
int main(){
    cout<<fun()<<endl;
    cout<<fun(4)<<endl;
    cout<<fun(4, 5)<<endl;
    cout<<fun(4, 5, 6)<<endl;
    return 0;
}
```

GUESS OUTPUT ?

a: 10, b: 20, c: 30
230
a: 4, b: 20, c: 30
110
a: 4, b: 5, c: 30
50
a: 4, b: 5, c: 6
26

Default Arguments

- A default argument is value provided in the function header that is automatically assigned by the compiler if the caller of the function doesn't provide a value for that argument while calling
- To avoid ambiguity, default arguments are all put at last in the formal parameters' list

Const Arguments

- Function arguments can be declared as const
- A const variable is one whose value cannot be changed once initialised
- To make variable const in the function, put keyword const corresponding the concerned argument's formal parameter
- e.g.

Function definition

```
int function(const int a, char b, float c, int d)
{
        a+=10;
        return a+b+c+d;
}
```

Function call

```
int out = function(5, 1.167, 'a', 2021)
```

error: cannot assign to variable 'a' with const-qualified type

Function Overloading

- We can define multiple functions with same name in c++, the concept is called function overloading
- Function prototype of overloaded functions must vary in one (or more) of the following ways:
  - Number of parameters function take
  - Data type of parameters function take
- Differing in only return type won't be a valid case of function overloading
- On calling overloaded function, compiler matches types of arguments passed and calls the most appropriate matching overloaded function

Function Overloading

Example

Getting Started with C++

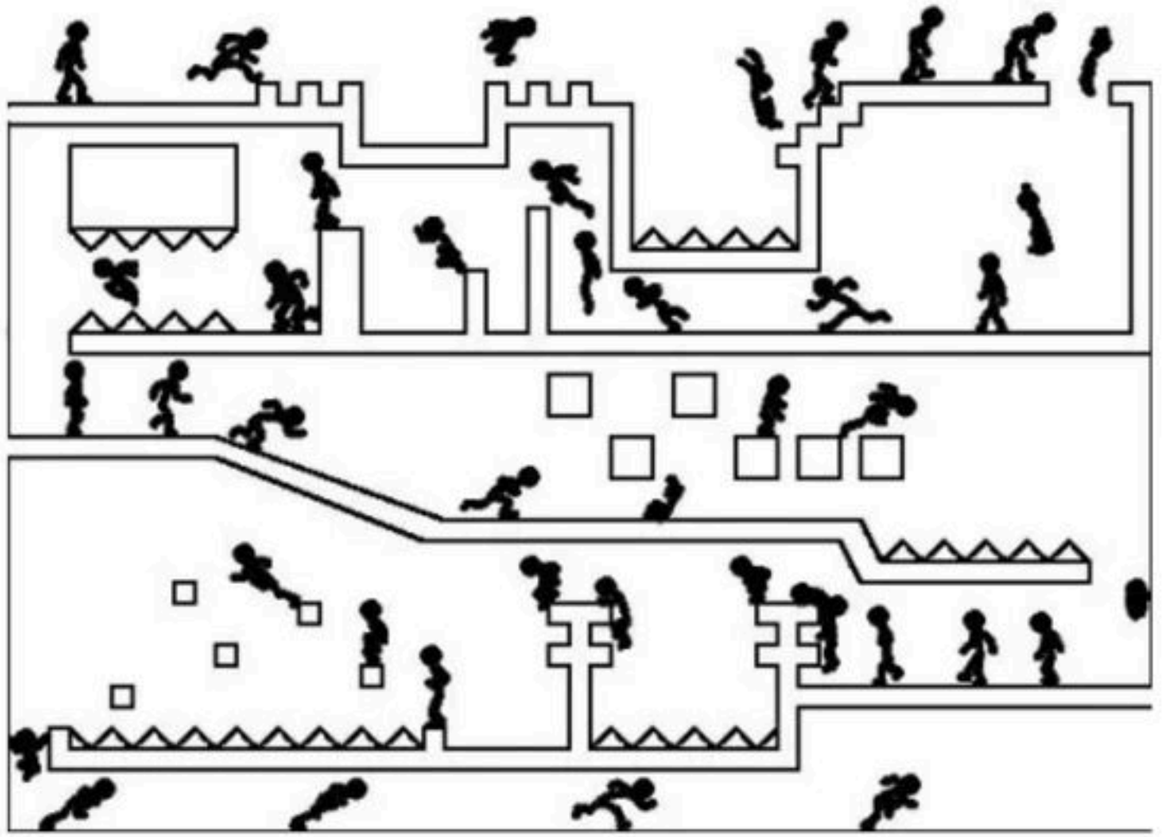
Recursion

Ravindrababu Ravula
Jay Bansal

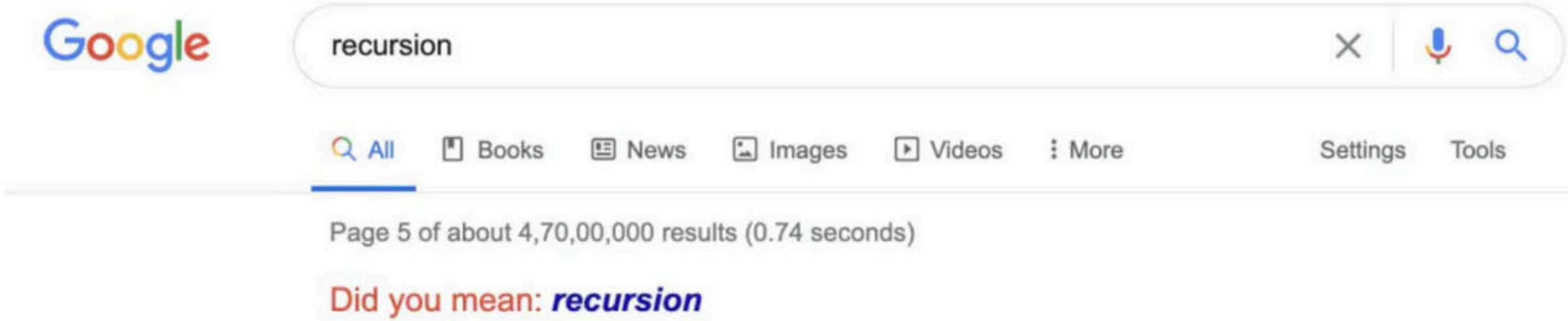
unacademy.com/@ravula
unacademy.com/@jay.bansal

What is Recursion ??

- A problem can be split into several problems of same kind, but simpler ones.
- Allows breaking of complex tasks into simpler problems.
- For eg, suppose you have a box full of 5 rupee coins and you have to count how much money you have. To save some time, you can ask a friend to help and split the task. When you both finish counting, you can aggregate the results.



Have you noticed that



When it comes to programming

- Recursive functions are the functions that call themselves. They have two main parts:
  - General (Recursive) case: Here the problem space is made smaller and smaller
  - Base case: The case for which the solution can be stated directly.

Let's take an example of calculating factorial using recursion,

$n! = (n) * (n-1) * (n-2) * (n-3) \ldots 2 * 1$

We can generalize this expression,

$n! = (n) * (n-1)!$

$$n! = \begin{cases} 1 & \text{if } n = 0 \text{ (base case)} \\ n * (n-1)! & \text{if } n > 0 \text{ (recursive case)} \end{cases}$$

Recursion vs Iteration

- So the question arises here is that how recursion is different from iteration and when should we use which alternative.
- Key differences:
  - **Termination**: In case of recursion, the termination is decided by the base case whereas in iteration, termination depends upon the value of control variable.
  - **Memory**: Recursion takes more memory as for every function call it occupies a memory block in the stack, whereas iteration doesn't requires any extra storage.
  - **Execution Time**: Iteration executes faster than recursion as there is no function calls and return statements.
  - **Code Complexity**: Recursive code is smaller and simples as compared to the iterative counterpart.

Recursion vs Iteration

- Let's see some examples:
  - Factorial using recursion
  - Factorial using iteration

Recursion vs Iteration

- Let's see some examples:
  - Fibonacci using iteration:
  - Problem: Write a cpp code to display fibonacci series till n using recursion.

Practice Problems

- Write the correct option in the chat:

```
int f(int x, int y)
{
    if (x == 0)
        return y;
    else
        return f(x - 1, x + y);
}
What will the above function return for f(10, 20) ?
a) 45
b) 55
c) 65
d) 75
```

Practice Problems

- Write the correct option in the chat:

```
int f(int x)
{
    if (x == 0 || x == 1)
        return x;
    if (x%3 != 0)
        return 0;
    else
        return f(x/3);
}
```

Choose the correct option for the above function?

a) Returns 1 when n is a power of 3, otherwise 0

b) Returns 0 when n is a power of 3, otherwise 1

c) Returns 0 when n is a power of 3, otherwise 0

d) Returns 1 when n is a power of 3, otherwise 1

Practice Problems

- Write the correct option in the chat:

```
void f(int x)
{
    if(x > 0)
    {
        f(--x);
        cout << x <<" ";
        f(--x);
    }
}
```


What will be the output of the above function call for f(5) ?

a) 0 1 2 3 4 3 2 1 0 1 2 3

b) 0 1 2 3 0 1 2 3 0 1 2 3

c) 0 1 2 0 3 0 1 4 0 1 2 0

d) 0 1 2 3 4 3 3 4 3 2 1 0

Practice Problems

- Write the correct option in the chat:

```
int f(int x)
{
    if (x < 0)
        return;
    f(x-2);
    cout << x << " ";
}
```

What will the above function return for f(6) ?

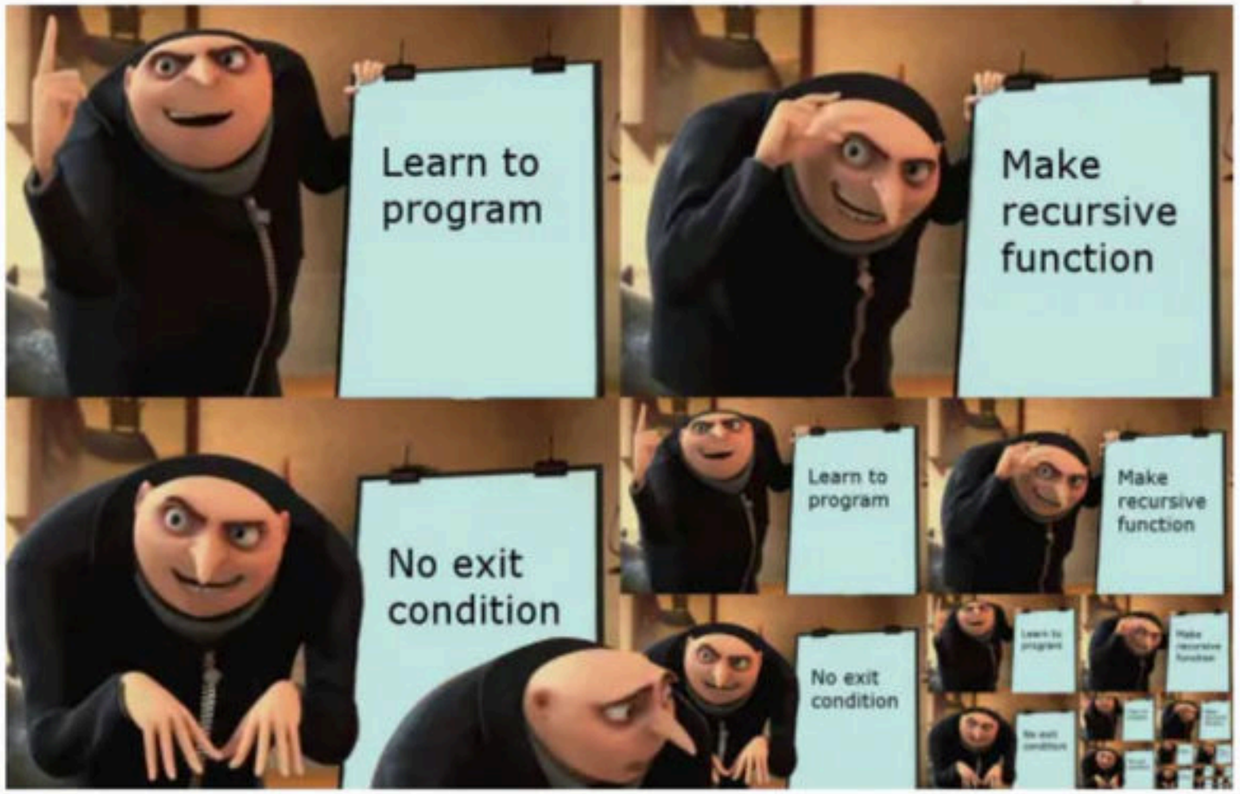
a) 2 4 6

b) 6 4 2

c) 6 6 6

d) None of the above

If not understood, try reading it again, and again... until you reach your base case



Getting Started with C++

Pointers, Arrays and References

Ravindrababu Ravula
Jay Bansal

unacademy.com/@ravula
unacademy.com/@jay.bansal

Derived & User-Defined Data Types

Derived data types: The data-types that are derived from the primitive or built-in datatypes.
User-Defined data types: Data types that are defined by user itself.

- **Arrays**
- **Pointers**

- Structures
- Union
- Enum
- Typedef defined DataType

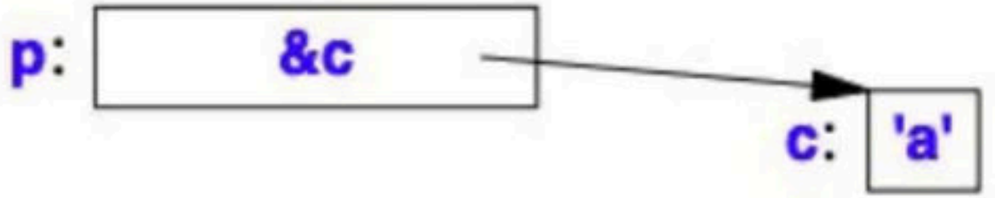
Pointers

For any type **T**, **T*** is the type "**pointer to T**."

That is, a variable of type **T*** can hold the address of an object of type **T**.

For example:

```
char c = 'a';
char* p = &c;  // p holds the address of c; & is the address-of operator
```

Dereferencing or indirection

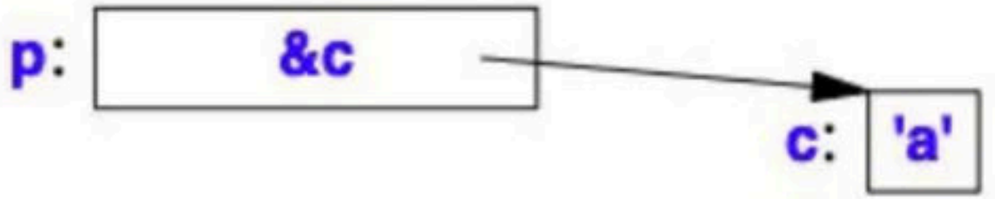
Referring to the object pointed to by the pointer

char c = 'a';

char* p = &c;

char c2 = *p; // c2 == 'a'; * is the dereference operator

The object pointed to by p is c, and the value stored in c is 'a', so the value of *p assigned to c2 is 'a'



void*

- In low-level code, we occasionally need to store or pass along an address of a memory location without actually knowing what type of object is stored there

- A **void*** is used for that, read as 'pointer to an object of unknown type."

- A pointer to any type of object can be assigned to a variable of type void*

- A void* can be assigned to another void*


```
int* pi;
void* pv = pi; // allowed
```

nullptr

- The literal **nullptr** represents the null pointer, that is, a pointer that does not point to an object

- It can be assigned to any pointer type, but **not** to other built-in types

```
int* p = nullptr;
double* p = nullptr;
int p = nullptr; // error: p is not a pointer
```

Using nullptr makes code more readable and avoids potential confusion when a function is overloaded to accept either a pointer or an integer (using nullptr instead of 0)

Arrays

- Sequential collection of elements of same data type.
- As they are stored sequentially in memory, it gives the advantage to access any element in constant time using the index.

For a type **T**, **T[size]** is the type, “array of **size** elements of type **T**”.

The elements are indexed from **0** to **size-1**.

For example:

```
int arr[2];     // an array of two integers: arr[0], arr[1]
char* p[4];  // an array of four pointers to char: p[0], p[1], p[2], p[3]
```

Initialization of Arrays

An array can be initialized by a list of values,

```
int arr[] = {4, 3, 2, 1};
char p[] = {'x', 'y', 'z'};
```

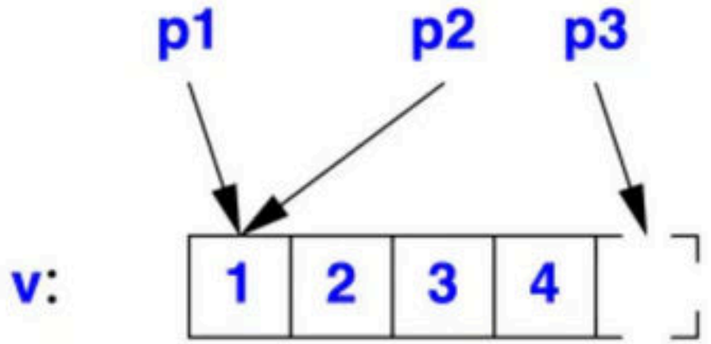
Correct or incorrect: type in chat

1. `int arr1[3] = {1, 2, 3, 4};`
2. `int arr2[6] = {1, 2, 3, 4};`
3. `int arr3[6] = arr2;`

Pointers to Arrays

The name of an array can be used as a pointer to its initial elements,

```
int v[] = {1, 2, 3, 4};
int* p1 = v;            // pointer to initial element
int* p2 = &v[0];        // pointer to initial element
int* p3 = v+4;
```

Multidimensional Arrays

Arrays of arrays are the multidimensional arrays

int marr[3][5]; // 3 arrays with 5 elements each

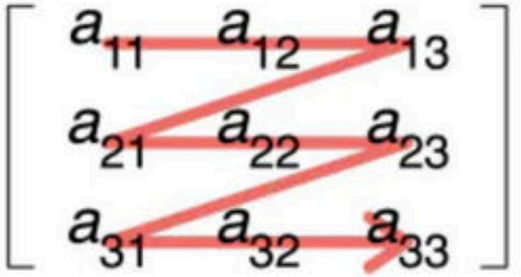
Eg.

```
int arr[3][2] = {{0,1}, {2,3}, {4,5}};
for (int i = 0; i < 3; i++)
{
    for (int j = 0; j < 2; j++)
        cout <<arr[i][j]<<endl;
}
```

Row major vs Column major

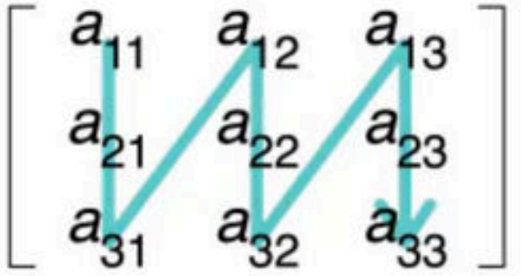
There are two methods of storing multidimensional arrays:

- Row major order: The elements are arranged consecutively along the row.



Address	0	1	2	3	4	5	6	7	8
Element	a_{11}	a_{12}	a_{13}	a_{21}	a_{22}	a_{23}	a_{31}	a_{32}	a_{33}

- Column major order: The elements are arranged consecutively along the column.



Address	0	1	2	3	4	5	6	7	8
Element	a_{11}	a_{21}	a_{31}	a_{12}	a_{22}	a_{32}	a_{13}	a_{23}	a_{33}

Passing Arrays

Arrays cannot be directly passed by value, instead, it is passed as a pointer to its first element.

- **way 1:** Argument as a pointer

  E.g. void func(int* param){ ... }
- **way 2:** Argument as a pointer

  E.g. void func(int param[50]){ ... }
- **way 3:** Argument as a pointer

  E.g. void func(int param[]){ ... }

Pointer to Function

Like a (data) object, the code generated for a function body is placed in memory somewhere, so it has an address

We can have a pointer to a function just as we can have a pointer to an object.

The pointer obtained by taking the address of a function can then be used to call the function

void (*foo)(int);

foo is a pointer to a function taking one argument, an integer, and that returns void.

Pointer to Function

void *(*foo)(int *);

Read inside-out

The innermost element of the expression is *foo, and that otherwise it looks like a normal function declaration.

foo is a pointer to a function that returns a void* and takes an int*.

Initializing Pointer to Function

To initialize a function pointer, you must give it the address of a function in your program:

```
void fun(int x) {
    printf( "%d\n", x );
}
void (*ptr)(int);
ptr = &fun;
[or]
ptr = fun;
```

Using Pointer to Function

To call the function pointed to by a function pointer, you treat the function pointer as though it were the name of the function you wish to call:

```
void fun(int x) {
    printf( "%d\n", x );
}
void (*ptr)(int);
ptr = &fun;
ptr(5);
[or]
(*ptr)(5);
```

Practice Questions

- Displaying the array in reverse direction ?

```
#include <iostream>
using namespace std;
int rev(int* arr, int size)  {
    for (int i = size-1; i >=0; i--)
        cout << arr[i] << " "
}
int main()  {
        int arr1[5] = { 10, 20, 30, 40, 50 };
        rev(arr1, 5);
        return 0;
}
```

Practice Questions

- Write a c++ code for finding the second largest element in the array?

Practice Questions

Write the correct option in the chat.

```
What does the following declaration means ?
int *ptr[10];

a) ptr is an array of pointers to 10 integers
b) ptr is a pointer to an array of 10 integers
c) ptr is an array of 10 integers
d) ptr is a pointer to an array
```

Practice Questions

Write the correct option in the chat.

What does the following declaration means ?

```
int *ptr[10];
```

a) ptr is an array of pointers to 10 integers

b) ptr is a pointer to an array of 10 integers

c) ptr is an array of 10 integers

d) ptr is a pointer to an array

Practice Questions

Write the correct option in the chat.

What does the following declaration means ?

```
int (*ptr)[10];
```

a) ptr is an array of pointers to 10 integers
b) ptr is a pointer to an array of 10 integers
c) ptr is an array of 10 integers
d) ptr is a pointer to an array

Practice Questions

Write the correct option in the chat.

What does the following declaration means ?

```
int (*ptr)[10];
```

a) ptr is an array of pointers to 10 integers

b) ptr is a pointer to an array of 10 integers

c) ptr is an array of 10 integers

d) ptr is a pointer to an array

Practice Questions

Write the correct option in the chat.

What will be the address of the arr[2][3] if arr is a 2-D long array of 4 rows and 5 columns and starting address of the array is 2000 ?
(Size of long = 4 bytes)

a) 2048
b) 2056
c) 2052
d) 2042

Practice Questions

Write the correct option in the chat.

What will be the address of the arr[2][3] if arr is a 2-D long array of 4 rows and 5 columns and starting address of the array is 2000 ?
(Size of long = 4 bytes)

a) 2048
b) 2056
c) 2052
d) 2042

Examples

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

int main(){
    int *p, a;
    cin>>a;
    p = &a;
    //p = a; (error: assigning to 'int *'
    //  from incompatible type 'int')
    cout<<a<<endl;
    *p += 20;
    cout<<a<<endl;
    return 0;
}
```

Output on a = 123 ?

Examples

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

int main(){
    int *p, a;
    cin>>a;
    p = &a;
    //p = a; (error: assigning to 'int *'
    //  from incompatible type 'int')
    cout<<a<<endl;
    *p += 20;
    cout<<a<<endl;
    return 0;
}
```

Output on a = 123 ?

123
143

Examples

```cpp
#include<iostream>
using namespace std;
int main()
{
    int arr[] = {11, 21, 31, 41, 51, 71, 91};
    int *ptr1 = arr;
    int *ptr2 = arr + 6;
    //Number of elements between ptr1 & ptr2: "
    cout<<(ptr2 - ptr1)<<endl;
    //Number of bytes between ptr1 & ptr2
    cout<<(char*)ptr2 - (char*) ptr1<<endl;
    return 0;
}
```

Output ?

Examples

```cpp
#include<iostream>
using namespace std;
int main()
{
    int arr[] = {11, 21, 31, 41, 51, 71, 91};
    int *ptr1 = arr;
    int *ptr2 = arr + 6;
    //Number of elements between ptr1 & ptr2: "
    cout<<(ptr2 - ptr1)<<endl;
    //Number of bytes between ptr1 & ptr2
    cout<<(char*)ptr2 - (char*) ptr1<<endl;
    return 0;
}
```

Output ?

Garbage
6
24

Examples

```
#include<iostream>
using namespace std;
int fun(int p, int *q, int **r){
    ++*q;
    **r*=2;
    p%=11;
    return (p+*q+**r);

}
int main(){
    int *p, a;
    cin>>a;
    p = &a;
    cout<<fun(a, &a, &p)<<endl;
    cout<<a<<endl;
    return 0;
}
```

Output on a = 23 ?

Examples

```
#include<iostream>
using namespace std;
int fun(int p, int *q, int **r){
    ++*q;
    **r*=2;
    p%=11;
    return (p+*q+**r);

}
int main(){
    int *p, a;
    cin>>a;
    p = &a;
    cout<<fun(a, &a, &p)<<endl;
    cout<<a<<endl;
    return 0;
}
```

Output on a = 23 ?

Garbage
97

Examples

Declare the following statement:

"An array of 4 pointers to chars".

A. char *ptr[4]();
B. char *ptr[4];
C. char (*ptr[4])();
D. char **ptr[4];

Examples

Declare the following statement:

"An array of 4 pointers to chars".

A. char *ptr[4]();

B. **char *ptr[4];**

C. char (*ptr[4])();

D. char **ptr[4];

Examples

Declare the following statement?

"A pointer to an array of three chars".

A. char *ptr[3]();
B. char (*ptr)*[3];
C. char (*ptr[3])();
D. char (*ptr)[3];

Examples

Declare the following statement?

"A pointer to an array of three chars".

A. char *ptr[3]();

B. char (*ptr)*[3];

C. char (*ptr[3])();

D. **char (*ptr)[3];**

Examples

Declare the following statement?

"A pointer to a function which receives nothing and returns a pointer to an integer".

A. int **(ptr)*int;

B. int **(*ptr)()

C. int *(*ptr)(*)

D. int *(*ptr)()

Examples

Declare the following statement?

"A pointer to a function which receives nothing and returns a pointer to an integer".

A. int **(ptr)*int;

B. int **(*ptr)()

C. int *(*ptr)(*)

D. int *(*ptr)()

Examples

Is the following declaration correct?

void(*f)(int, void(*)());

A. Yes

B. No

Examples

Is the following declaration correct?

void(*f)(int, void(*)());

A. Yes

B. No

f is a pointer to a function which returns nothing and receives as its parameter an integer and a pointer to a function which receives nothing and returns nothing.

Examples

What do the following declarations mean?

1. void (*x[3]) (int)

2. double (*(*x())[]) ()

3. int *(*(*x[5])()) ()

Getting Started with C++

Dynamic Memory Allocation & Typecasting

Ravindrababu Ravula
Jay Bansal

unacademy.com/@ravula
unacademy.com/@jay.bansal

Dynamic Memory Allocation

DMA in C++ refers to performing memory allocation manually by programmer.

Dynamically allocated memory is allocated on **Heap**

Applications:

- Variable length arrays
- Linked lists, Trees, Graphs, etc.

For normal variables, memory is allocated and deallocated automatically

For dynamically allocated memory, it is our responsibility to deallocate memory when no longer needed

Dynamic Memory Allocation

Dynamic Memory Allocation in C is using malloc(), calloc(), free() and realloc()

C++ supports these functions and has two operators **new** and **delete** that perform the task of allocating and freeing the memory in a easier way!

Operator new acquires its memory by calling an operator **new()**. Similarly, operator delete frees its memory by calling an operator **delete()**

new Operator

new operator requests for memory allocation on the **Free Store**.

If sufficient memory is available, new operator initializes the memory and returns the address of the newly allocated and initialized memory to the pointer.

ptr = new data_type;

eg:

int *ptr = NULL;

ptr = new int;

Free store vs Heap

new Operator

when new needs memory on the free store for an object of type X, it calls operator:

new(sizeof(X))

Similarly, when new needs memory on the free store for an array of N objects of type X, it calls operator:

new[](N*sizeof(X))

int *p = new int[10]

delete Operator

It is our responsibility to deallocate dynamically allocated memory!

If you forget to delete the allocated memory, **memory leak** happens

delete ptr;

delete[] ptr;

Practice Questions

- How to create a dynamic array of integers of size 3 using new in C++?

A int *arr = new int *[3];

B int **arr = new int *[3];

C int arr = new int [3];

D int *arr = new int [3];

Practice Questions

- How to create a dynamic array of integers of size 3 using new in C++?

A int *arr = new int *[3];

B int **arr = new int *[3];

C int arr = new int [3];

D int *arr = new int [3];

Practice Questions

- How to create a dynamic array of **pointers to** integers of size 3 using new in C++?

A int *arr = new int *[3];

B int **arr = new int *[3];

C int arr = new int [3];

D int *arr = new int [3];

Practice Questions

- How to create a dynamic array of **pointers to** integers of size 3 using new in C++?

A int *arr = new int *[3];

B int **arr = new int *[3];

C int arr = new int [3];

D int *arr = new int [3];

Practice Questions

- Correct statement(s)?

A new is an operator

B malloc() is a function

C new allocates memory from free store

D malloc() allocates memory from heap

E None of the above

Practice Questions

- Correct statement(s)?

A new is an operator

B malloc() is a function

C new allocates memory from free store

D malloc() allocates memory from heap

E None of the above

Example

```cpp
#include<iostream>
using namespace std;
int main(){
    int *a, sum = 0;
    a = new int[6];
    for(int i=0; i<6; i++){
        a[i] = i+1;
    }
    for(int i=0; i<6; i++){
        sum+=a[i];
    }
    for(int i=0; i<6; i++){
        sum+=a[i];
    }
    cout<<sum<<endl;
    return 0;
}
```

Output ?

Example

```
#include<iostream>
using namespace std;
int main(){
    int *a, sum = 0;
    a = new int[6];
    for(int i=0; i<6; i++){
        a[i] = i+1;
    }
    for(int i=0; i<6; i++){
        sum+=a[i];
    }
    for(int i=0; i<6; i++){
        sum+=a[i];
    }
    cout<<sum<<endl;
    return 0;
}
```

Output ?

42

Example

```
#include<iostream>
using namespace std;
int main(){
    int *a, sum = 0;
    a = new int[6];
    for(int i=0; i<6; i++){
        a[i] = i+1;
    }
    for(int i=0; i<6; i++){
        sum+=a[i];
    }
    delete[] a;
    for(int i=0; i<6; i++){
        sum+=a[i];
    }
    cout<<sum<<endl;
    return 0;
}
```

Output ?

Example

```
#include<iostream>
using namespace std;
int main(){
    int *a, sum = 0;
    a = new int[6];
    for(int i=0; i<6; i++){
        a[i] = i+1;
    }
    for(int i=0; i<6; i++){
        sum+=a[i];
    }
    delete[] a;
    for(int i=0; i<6; i++){
        sum+=a[i];
    }
    cout<<sum<<endl;
    return 0;
}
```

Output ?

Not known
Dangling Pointer

Example

```
#include<iostream>
using namespace std;
int main(){
    int *a, sum = 0;
    a = new int[6];
    for(int i=0; i<6; i++){
        a[i] = i+1;
    }
    for(int i=0; i<6; i++){
        sum+=a[i];
    }
    delete[] a;
    a = NULL;
    for(int i=0; i<6; i++){
        sum+=a[i];
    }
    cout<<sum<<endl;
    return 0;
}
```

Output ?

Example

```
#include<iostream>
using namespace std;
int main(){
    int *a, sum = 0;
    a = new int[6];
    for(int i=0; i<6; i++){
        a[i] = i+1;
    }
    for(int i=0; i<6; i++){
        sum+=a[i];
    }
    delete[] a;
    a = NULL;
    for(int i=0; i<6; i++){
        sum+=a[i];
    }
    cout<<sum<<endl;
    return 0;
}
```

Output ?

Segmentation fault

Typecasting

Typecasting is making a variable of one type, such as an int, act like another type, a char, for one single operation

To typecast something, **put the type of variable** you want the actual variable to act as **inside parentheses** in front of the actual variable


```
#include <iostream>
using namespace std;
int main() {
        cout<< (char)66 <<endl;
}
```

Implicit type conversion in c++

C++ supports other types to typecast as well.

bool -> char -> short int -> int -> unsigned int -> long -> unsigned -> long long -> float -> double -> long double

Data loss can occur (Signs, overflow)

Practice Questions

Which of these is/are a valid typecast in C++?

A. a(char);

B. char:a;

C. (char)a;

D. to(char, a);

E. char(a);

Practice Questions

Which of these is/are a valid typecast in C++?

A. a(char);

B. char:a;

C. (char)a;

D. to(char, a);

E. char(a);

Demo

```cpp
#include<iostream>
using namespace std;
int main() {
    long a = 1;
    int b = 1e5, c = 1e5;
    a = long(b*c);
    cout<<a<<endl;
    return 0;
}
```

Example

```
#include<iostream>
using namespace std;
int main(){
    int a = 257;
    char ch = (char)a + 'a';
    printf("%c\n", ch);
    return 0;
}
```

Output ?

Example

```
#include<iostream>
using namespace std;
int main(){
    int a = 257;
    char ch = (char)a + 'a';
    printf("%c\n", ch);
    return 0;
}
```

Output ?

b

Example

Output ?

```
#include<iostream>
using namespace std;
int main(){
    int a = 257;
    int *p = &a;
    cout<<(char)(*(char*)p + 'a')<<endl;
    cout<<*p<<endl;
    return 0;
}
```

Example

```
#include<iostream>
using namespace std;
int main(){
    int a = 257;
    int *p = &a;
    cout<<(char)(*(char*)p + 'a')<<endl;
    cout<<*p<<endl;
    return 0;
}
```

Output ?

```
b
257
```

Example

```
#include<iostream>
using namespace std;
int main(){
    int a = 256;
    int *p = &a;
    char *q = (char*)p;
    cout<<char(*q+'a');
    q++;
    cout<<char(*q+'a');
    q++;
    cout<<char(*q+'a')<<endl;
    return 0;
}
```

Output ?

Example

```
#include<iostream>
using namespace std;
int main(){
    int a = 256;
    int *p = &a;
    char *q = (char*)p;
    cout<<char(*q+'a');
    q++;
    cout<<char(*q+'a');
    q++;
    cout<<char(*q+'a')<<endl;
    return 0;
}
```

Output ?

aba

Getting Started with C++

Structures, Unions, and Enumerations

Ravindrababu Ravula
Jay Bansal

unacademy.com/@ravula
unacademy.com/@jay.bansal

Derived & User-Defined Data Types

Derived data types: The data-types that are derived from the primitive or built-in datatypes.

User-Defined data types: Data types that are defined by user itself.

- **Arrays**
- **Pointers**

- **Structures**
- **Union**
- **Enum**

Structures

A structure is a sequence of elements of arbitrary data type.

For example, if we want to store address,

```
struct Address {
        const char* name;           // “Sameer”
        int number;                 // 45
        const char* street;         // “Tilak Nagar”
        const char* town;           // “Chennai”
        char state[2];              // “Tamil Nadu”
        const char* zip;            // “102039”
};
```

Structures

- Let's see how to reference the elements:
  - Using . (dot) operator:

```
Address add;
add.name = "Sameer";
add.number = 45;                    // and so on
```

  - Through pointers using -> (struct pointer dereference) operator:

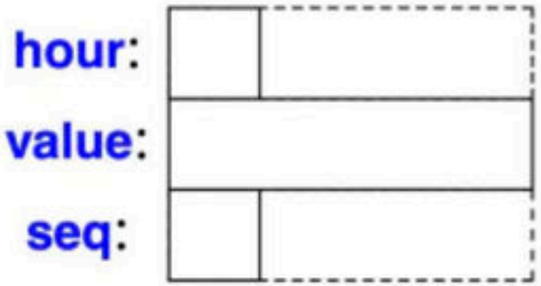
```
void prtAdd(Address* add){
        cout << add->name;
        cout << add->number;        // and so on
}
```

Structures

- Total size of the structure is not always the sum of the sizes of all data members.

  Let's understand this:

```
struct Readout{
        char hour;
        int value;
        char seq;
};
```

So the size of Readout on a 4 byte-int machine would be 12 bytes and not 6 bytes, this is due to the padding added by the compiler to avoid alignment issues.

Unions

- A **union** is a **struct** in which all members are allocated at the same address so that the union occupies only as much space as its largest member.
- It can only hold value for one member at a time.

For example, if we want to store address,

```
union Address {
        const char* name;           // “Sameer”
        int number;                 // 45
        const char* street;         // “Tilak Nagar”
};
```

Unions

- Like structures, their members can also be accessed by:
  - Using . (dot) operator:

```
Address add;
add.name = “Sameer”;
add.number = 45;                    // and so on
```

  - Through pointers using -> (union pointer dereference) operator:

```
void prtAdd(Address* add){
        cout << add->name;
        cout << add->number;   // and so on
}
```

Unions

- The size of a union is taken according the size of largest member in it.

  Let's understand this:

```
union Readout{
        char hour;
        int value;
        char seq;
};
```


So the size of Readout on a 4 byte-int machine would be 4 bytes and not 6 bytes, this is due to using same memory for all members.

Enumerations

- An enum is a type that can hold a set of integer values specified by the user.
- These values are defined at the time of declaring the enumerated type. For example,

```
enum Color {
        red,
        green,
        blue
};
```

Here red, green and blue are called the enumerators with implicit values as 0, 1 and 2 respectively.

Enumerations

- We can also change the default value of an enum element at the time of declaration.

  For example,

```
enum Color {
        red = 10,
        green = 20,
        blue = 30
};
```

  Here red, green and blue have values 10, 20 and 30 respectively.

Enumerations

- Why enums are used ?
  - Because an enum variable takes only one value out of all possible values. This makes enums a good choice to work with flags.

For example,

```
enum light {
        red = 1,
        orange = 2,
        green = 3,
        yellow = 4
  } signal;
```

Here the signal enum variable can take value out of 4 possible values.

Examples

```
int main() {
    struct var {
        int a, b;
    };
    struct var v;
    v.a=10;
    v.b=20;
    printf("%d\n", v.a);
    return 0;
}
```

A. 10
B. 20
C. 30
D. 0

Examples

```
int main() {
    struct var {
        int a, b;
    };
    struct var v;
    v.a=10;
    v.b=20;
    printf("%d\n", v.a);
    return 0;
}
```

A. 10
B. 20
C. 30
D. 0

Examples

```
int main() {
    union var {
        int a, b;
    };
    union var v;
    v.a=10;
    v.b=20;
    printf("%d\n", v.a);
    return 0;
}
```

A. 10
B. 20
C. 30
D. 0

Examples

```
int main() {
    union var {
        int a, b;
    };
    union var v;
    v.a=10;
    v.b=20;
    printf("%d\n", v.a);
    return 0;
}
```

A. 10

B. 20

C. 30

D. 0

Examples

```
int main() {
    enum days {MON=-1, TUE, WED=6, THU, FRI, SAT};
    printf("%d, %d, %d, %d, %d, %d\n", MON, TUE, WED, THU, FRI, SAT);
    return 0;
}
```

A. -1, 0, 1, 2, 3, 4

B. -1, 2, 6, 3, 4, 5

C. -1, 0, 6, 2, 3, 4

D. -1, 0, 6, 7, 8, 9

Examples

```
int main() {
    enum days {MON=-1, TUE, WED=6, THU, FRI, SAT};
    printf("%d, %d, %d, %d, %d, %d\n", MON, TUE, WED, THU, FRI, SAT);
    return 0;
}
```

A. -1, 0, 1, 2, 3, 4

B. -1, 2, 6, 3, 4, 5

C. -1, 0, 6, 2, 3, 4

D. -1, 0, 6, 7, 8, 9

Examples

The following C declarations

```
struct node
{
   int i;
   float j;
};
struct node *s[10] ;
```

define s to be?

A An array, each element of which is a pointer to a structure of type node

B A structure of 2 fields, each field being a pointer to an array of 10 elements

C A structure of 3 fields: an integer, a float, and an array of 10 elements

D An array, each element of which is a structure of type node

[GATE CS 2000]

Examples

The following C declarations

```
struct node
{
   int i;
   float j;
};
struct node *s[10] ;
```

define s to be?

A An array, each element of which is a pointer to a structure of type node

B A structure of 2 fields, each field being a pointer to an array of 10 elements

C A structure of 3 fields: an integer, a float, and an array of 10 elements

D An array, each element of which is a structure of type node

[GATE CS 2000]

Getting Started with C++

Remaining Basic Concepts in C++

Ravindrababu Ravula
Jay Bansal

unacademy.com/@ravula
unacademy.com/@jay.bansal

Structure Member Alignment

```
struct A {
   char         a;
   short int    b;
} oa;
```

```
sizeof(oa)
```

Structure Member Alignment

```
struct A {
   char         a;
   short int    b;
} oa;
```

sizeof(oa) 4

Structure Member Alignment

```
struct B {
   char        a;
   short int   b;
   int         c;
} ob;



sizeof(ob)
```

Structure Member Alignment

```
struct B {
   char        a;
   short int   b;
   int         c;
} ob;
```

sizeof(ob) 8

Structure Member Alignment

```
struct C {
   char         a;
   double    b;
   int          c;
} oc;
```

```
sizeof(oc)
```

Structure Member Alignment

```
struct C {
   char        a;
   double      b;
   int         c;
} oc;
```

sizeof(oc) 24

Structure Member Alignment

```
struct D {
   char        a;
   int         b;
   double      c;
} od;
```

```
sizeof(od)
```

Structure Member Alignment

```
struct D {
   char        a;
   int         b;
   double      c;
} od;
```

sizeof(od) 16

Structure Member Alignment

```
struct A {
        int x;
        double y;
        short int z;
    };
```

sizeof(struct A)?

Structure Member Alignment

```
struct A {
        int x;
        double y;
        short int z;
    };
```

sizeof(struct A)? 24

Structure Member Alignment

```
struct B {
        double z;
        int x;
        short int y;
    };
```

sizeof(struct B)?

Structure Member Alignment

```
struct B {
        double z;
        int x;
        short int y;
    };
```

sizeof(struct B)? 16

typedef

The typedef keyword allows us to create new names for types such as int or, more commonly in C++, templated types

Stands for "type definition"

typedef long long int ll;

typedef struct A sA;

typedef

```
typedef int a;
a b=2, c=8, d;
d=(b*2)/2+8;
printf("%d",d);
```

A) 10
B) 8
C) 16
D) Compiler Error

typedef

```
typedef int a;
a b=2, c=8, d;
d=(b*2)/2+8;
printf("%d",d);
```

A) 10

B) 8

C) 16

D) Compiler Error

Pointers and multidimensional arrays

```
int matrix[2][5] = {{1,2,3,4,5},{6,7,8,9,10}};

int (*pmatrix)[5] = matrix;

void fun(int arr[][5], int rows) {...}

[or]

void fun(int (*arr)[5], int rows) {...}



void fun(int (*arr)[2][4], int rows) {...}
```

Pointers and multidimensional arrays

```
char arr[5][7][6];

char (*p)[5][7][6] = &arr;

cout<< (&arr + 1) - &arr;

cout<< (char *)(&arr + 1) - (char *)&arr;
```

Pointers and multidimensional arrays

```
char arr[5][7][6];

char (*p)[5][7][6] = &arr;

cout<< (&arr + 1) - &arr;

cout<< (char *)(&arr + 1) - (char *)&arr;
```

```
1
42
```

Exception handling in C++

An exception is a problem that arises during the execution of a program.

A C++ exception is a response to an exceptional circumstance that arises while a program is running.

C++ exception handling is built upon three keywords: try, catch, and throw.

throw − A program throws an exception when a problem shows up. This is done using a throw keyword.

catch − A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The catch keyword indicates the catching of an exception.

try − A try block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.

Exception handling in C++

```
try {
   // protected code
} catch( ExceptionName e1 ) {
   // catch block
} catch( ExceptionName e2 ) {
   // catch block
...
} catch( ExceptionName eN ) {
   // catch block
}
```

Exception handling in C++

```
int x = 5;
int y = 0;
double z = 0;
try {
    if( y == 0 ) {
        throw "Division by zero condition!";
    }
    z = x/y;
    cout<<z<<endl;
} catch (const char* msg) {
    cerr << msg << endl;
}
```

Exception handling in C++

```
int x = -1;
try {
        cout << "Inside try \n";
        if (x < 0) {
                throw x;
                cout << "After throw \n";
        }
}
catch (int x ) {
    cout << "Exception Caught \n";
}
cout << "After catch \n";
```

A)
```
Inside try
Exception Caught
After throw
After catch
```

B)
```
Inside try
Exception Caught
After catch
```

C)
```
Inside try
Exception Caught
```

D)
```
Inside try
After throw
After catch
```

Exception handling in C++

```
int x = -1;
try {
        cout << "Inside try \n";
        if (x < 0) {
                throw x;
                cout << "After throw \n";
        }
}
catch (int x ) {
    cout << "Exception Caught \n";
}
cout << "After catch \n";
```

A)
```
Inside try
Exception Caught
After throw
After catch
```

B)
```
Inside try
Exception Caught
After catch
```

C)
```
Inside try
Exception Caught
```

D)
```
Inside try
After throw
After catch
```

Exception handling in C++

```
try {
        try {
            throw 2;
        }
        catch (int n) {
            cout << "Inner Catch \n";
            throw;
        }
}
catch (int x) {
    cout << "Outer Catch \n";
}
```

A) Outer Catch
B) Inner Catch
C) Inner Catch
   Outer Catch
D) Compiler Error

Exception handling in C++

```
try {
        try {
            throw 2;
        }
        catch (int n) {
            cout << "Inner Catch \n";
            throw;
        }
}
catch (int x) {
    cout << "Outer Catch \n";
}
```

A) Outer Catch

B) Inner Catch

C) Inner Catch
Outer Catch

D) Compiler Error

Getting Started with C++

Object Oriented programming in C++

Ravindrababu Ravula
Jay Bansal

unacademy.com/@ravula
unacademy.com/@jay.bansal

What is OOP ?

- Object Oriented Programming is the principle of design & development of programs using modular approach.
- The procedural programming focuses on processing of instructions in order to perform a desired computation, it emphasizes more on doing things like algorithms. Used in programming languages like C & Pascal.
- OOP combines both the data and the functions that operate on that data into a single unit called the object. It follows bottom-up design technique, it is piecing together the smaller systems to give rise to more complex systems.
- Another major component that plays a major role in OOP is the class. It is a template that represents a group of objects which share common properties and relationships.

Why OOP ?

- The main reason is that the traditional procedural programming languages were unable to model real world problems.

C++ as Object Oriented Language ?

- It is an object-oriented, general-purpose programming language, derived from C.
- Existing code on C can be used with C++, hence mode compatible. Even existing pre-compiled libraries can be used with new C++ code.
- Also there is no additional cost for using C++, hence efficient.
- Major improvements over C:
  - Stream I/O
  - Strong typing
  - Inlining
  - Default argument values
  - Parameter passing by reference
  - OOP advantages.

OOP Principles ?

- Object Oriented Programming is a methodology characterized by the following concepts:

- **Encapsulation**: The process of binding data members (variables, properties) and member functions (methods) into a single unit. A class is the best example.
  - Take an example of a pharmacy store.
  - You go to the shop and ask for a medicine.
  - There only the chemist has the access to the medicines of the store and he knows what medicine to give you.
  - This reduces the risk of taking a medicine that is not prescribed to you.
  - Here medicines are the member variables, chemist is the member function and you are the external piece of code.

OOP Principles ?

- **Abstraction**: It refers to represent necessary features without including more details or explaination. Data abstraction is a programming technique that relies on the separation of interface and implementation.
  - Take an example of your laptop.
  - When you press a key on keyboard, the character appears on the screen.
  - You need to know only this.
  - How exactly it works is not required. This is called abstraction.

OOP Principles ?

- **Inheritance**: The mechanism of deriving a new class from an old class is called inheritance. The old class is known as base class while new class is known as derived class or subclass. The inheritance is the most powerful features of OOP.
  - For example, a child inherits the traits of his/her parents.
  - With inheritance, we can reuse the fields and methods of the existing class.
  - Through effective use of inheritance, you can save lot of time in your programming and also reduce errors
  - Which in turn will increase the quality of work and productivity.
  - There are different types of inheritance: single, multiple, multilevel, hybrid.

OOP Principles ?

- **Polymorphism**: It means ability to take more than one form.
  - For example, a + is used to add two numbers, but it can also be used to concatenate two strings.
  - This is known as operator overloading because same operator may behave differently on different instances.
  - Same way functions can be overloaded. For eg, sum() can be used to add two integers as well as two floating point numbers.

Topics Covered

- Classes & Objects
- Member functions & data
- Static data and methods
- Constructors & Destructors
- Data Hiding
- Polymorphism
- Compile-time Polymorphism
- Overloading
- Inheritance
- Abstract Classes
- Run-time Polymorphism

Object Oriented programming in C++

Classes, Objects & Methods

Ravindrababu Ravula
Jay Bansal

unacademy.com/@ravula
unacademy.com/@jay.bansal

C++ Classes and Objects

- C++ is an object-oriented programming language.

- Everything in C++ is associated with classes and objects, along with its attributes and methods.

- Attributes and methods are basically variables and functions that belongs to the class. These are often referred to as "class members".

- A class is a user-defined data type that we can use in our program, and it works as an object constructor, or a "**blueprint**" for creating objects.

Creating a class

To create a class, use the **class** keyword:

For eg:

```
class MyClass {     // Class name
 public:                // Access specifier
   int myNum;       // Attribute
   string myString;  // Attribute
};
```

Creating an Object

In C++, an object is created from a class.

To create an object, specify the class name, followed by the object name.

To access the class attributes, use the dot syntax (.) on the object:

```
MyClass myObj;  // Create an object of MyClass

myObj.myNum = 5;

myObj.myString = "Hello";
```

Class Methods

Methods are functions that belongs to the class. They define the behaviour/ action taken by object.

There are two ways to define functions that belongs to a class:

```
class MyClass {                 // Class
 public:                        // Access specifier
  void myMethod() {             // Method
   cout << "Hello World!";
  }
};
```

Class Methods

Methods are functions that belongs to the class. They define the behaviour/ action taken by object.

There are two ways to define functions that belongs to a class:

```
class MyClass {          // Class
  public:                // Access specifier
    void myMethod();     // Method declaration
};

void MyClass::myMethod() {
  cout << "Hello World!";
}
```

Class Methods

Eg.

```
class Student {
public:
// Declaration of state/Properties.
   string name;       // Instance variable.
   int rollNo;        // Instance variable.
   static int age;    // Static variable.
// Declaration of Actions.
   void display() { // Instance method.
     // method body.
   }
};
```

Real life example of classes & objects

Object: **Person**

State/Properties:

Black Hair Color: hairColor = "Black";

5.5 feet tall: height = 5.5;

Weighs 60 kgs: weight = 60;

Behaviour/Action:

Eat: eat()

Sleep: sleep(5)

Run: run(0.5)

Access specifiers

Access specifiers define how the members (attributes and methods) of a class can be accessed:

In C++, there are three access specifiers:

public - members are accessible from outside the class
private - members cannot be accessed (or viewed) from outside the class
protected - members cannot be accessed from outside the class, however, they can be accessed in **inherited** classes.

By default, all members of a class are private if you don't specify an access specifier, they will be private

Access specifiers

```
class MyClass {
 public:   // Public access specifier
   int x;  // Public attribute
 private:  // Private access specifier
   int y;  // Private attribute
};

int main() {
 MyClass myObj;
 myObj.x = 5;  // Allowed (public)
 myObj.y = 5;  // Not allowed (private)
 return 0;
}
```

error: y is private

Default Copying

By default, objects can be copied. In particular, a class object can be initialized with a copy of an object of its class. For example:

```
Date d1 = my_birthday;   // initialization by copy

Date d2 {my_birthday};   // initialization by copy
```

By default, the copy of a class object is a copy of each member.

If that default is not the behavior wanted for a class X, a more appropriate behavior can be provided using overloading.

Static Members of a C++ Class

We can define class members static using static keyword.

When we declare a member of a class as static it means no matter how many objects of the class are created, **there is only one copy of the static member.**

A static member is shared by all objects of the class.

All static data is **initialized to zero** when the first object is created, if no other initialization is present.

Static Members of a C++ Class

We can't put it in the class definition but it can be initialized outside the class as done in the following example by redeclaring the static variable, using the scope resolution operator :: to identify which class it belongs to.

Eg:

```
class Box {
  public:
    static int objectCount;
};
// Initialize static member of class Box
int Box::objectCount = 0;
```

Static Function Members

By declaring a function member as static, you make it independent of any particular object of the class.

A static member function can be called even if no objects of the class exist and the static functions are accessed using only the class name and the scope resolution operator **::**

A static member function can only access static data member, other static member functions and any other functions from outside the class.

Static member functions have a class scope and they do not have access to the **this** pointer of the class.

Static Function Members

```
class Box {
  public:
       static int objectCount;
       static int getCount() {
        return length;
    }
  private:
    double length;    // Length of a box
};


Box::getCount();
```

Static Function Members

```
class Box {
  public:
       static int objectCount;
       static int getCount() {
        return objectCount;
    }
  private:
    double length;    // Length of a box
};


Box::getCount();
```

Passing and Returning Objects in C++

In C++ we can pass class objects as arguments and also return them from a function the same way we pass and return other variables.

Passing an Object as argument

To pass an object as an argument we write the object name as the argument while calling the function the same way we do it for other variables.

Syntax:

```
fun(object_name);
```

Passing and Returning Objects in C++

Passing an Object as argument

```
class Example {
public:
   int a;
   void add(Example E)
   {
      a = a + E.a;
   }
};
```

```
Example E1, E2;
E1.a = 5;
E2.a = 10;
E2.add(E1);
cout << E1.a << E2.a << "\n";
```

Passing and Returning Objects in C++

Returning Object as argument

```
class Example {
public:
   int a;
Example add(Example Ea, Example Eb) {
      Example Ec;
      Ec.a = Ec.a + Ea.a + Eb.a;
      return Ec;
   }
};
```

```
Example E1, E2,E3;
E1.a = 5;
E2.a = 10;
E3 = E3.add(E1, E2);
cout << E3.a << "\n";
```

JAY BANSAL SIR

Expert in Competitive Programming

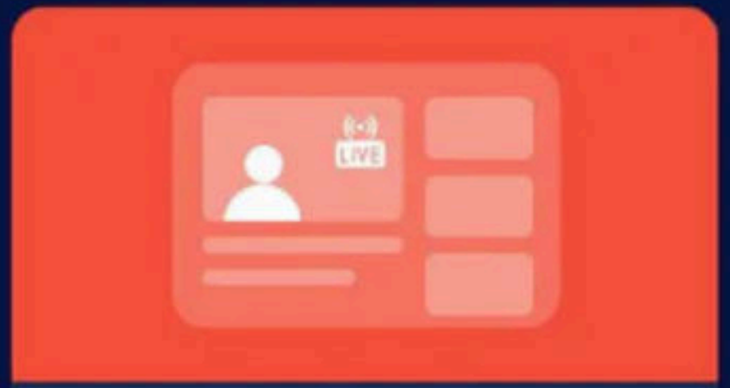
GATE CSE 2019 - AIR 2

Incoming SDE @ Google
ACM ICPC 2019 - AIR 39
B.Tech Gold Medalist

Subjects Taken:

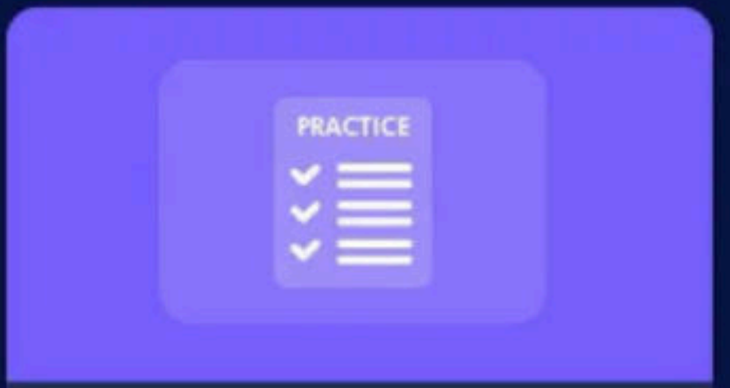
Competitive Programming
C++
Data Structures
Algorithms
Interview Preparation

What You Will Get?



Live Interactive Classes

Attend live interactive classes with our top educators. Interact during class with educators to get all your doubts resolved.

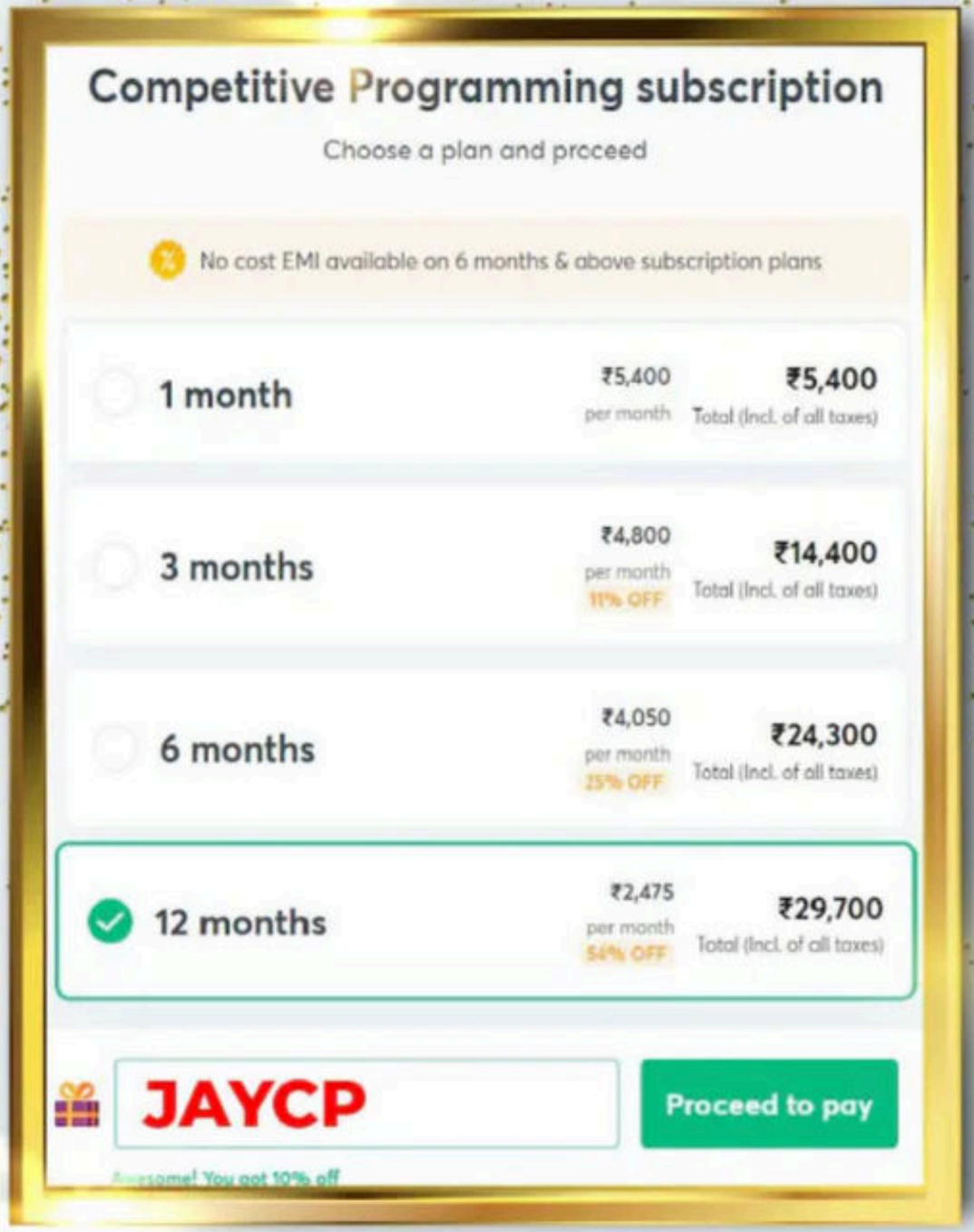


Practice Relevant Problems @ Codechef

Each class comes with a set of curated practice problems to help you apply the concepts in real time.

Doubt Support

If you get stuck in any problem post class- Get your doubts resolved by our expert panel of teaching assistants and community members instantly.


Competitive Programming subscription
Choose a plan and proceed
No cost EMI available on 6 months & above subscription plans
1 month
₹5,400
per month
₹5,400
Total (Incl. of all taxes)
3 months
₹4,800
per month
11% OFF
₹14,400
Total (Incl. of all taxes)
6 months
₹4,050
per month
25% OFF
₹24,300
Total (Incl. of all taxes)
12 months
₹2,475
per month
54% OFF
₹29,700
Total (Incl. of all taxes)
JAYCP
Proceed to pay
Awesome! You got 10% off

Follow Me on

unacademy

https://unacademy.com/@jay.bansal

Thank You