C++ Programing Language

About

Year

Fall 2022

BY

A. S. Eldesouky

Table of Content

Diving In

Data Input and Output

Input / Output Manipulators

std::Keyword	Manipulators	Library
boolalpha and noboolalpha	switches between textual and numeric representation of booleans (function)	
showbase and noshowbase	controls whether prefix is used to indicate numeric base (function)	
showpoint and noshowpoint	controls whether decimal point is always included in floating-point representation (function)	
showpos and noshowpos	controls whether the `+` sign used with non-negative numbers (function)	
skipws and noskipws	controls whether leading whitespace is skipped on input (function)	
uppercase and nouppercase	controls whether uppercase characters are used with some output formats (function)	
unitbuf and nounitbuf	controls whether output is flushed after each operation (function)	
internal and left and right	sets the placement of fill characters (function)	
dec and hex and oct	changes the base used for integer I/O (function)	
fixed and scientific and hexfloat (C++11) and defaultfloat (C++11)	changes formatting used for floating-point I/O (function)	
ws	consumes whitespace (function template)	
ends	outputs '**\0**' (function template)	
flush	flushes the output stream (function template)	
endl	outputs '**\n**' and flushes the output stream (function template)	
emit_on_flush and noemit_on_flush (C++20)	controls whether a stream's basic_syncbuf emits on flush (function template)	
flush_emit (C++20)	flushes a stream and emits the content if it is using a basic_syncbuf (function template)	
resetiosflags]	clears the specified ios_base flags (function)	
setiosflags	sets the specified ios_base flags (function)	
setbase	changes the base used for integer I/O (function)	
setfill	changes the fill character (function template)	
setprecision	changes floating-point precision (function)	

std::Keyword	Manipulators	Library
setw	changes the width of the next input/output field (function)	
get_money (C++11)	parses a monetary value (function template)	
put_money (C++11)	formats and outputs a monetary value (function template)	
get_time (C++11)	parses a date/time value of specified format (function template)	
put_time (C++11)	formats and outputs a date/time value according to the specified format (function template)	
quoted (C++14)	inserts and extracts quo	

Variables and Data Type

- **Variables :** A named piece of memory that use to store specific types of data.
- **Data Type :** A particular kind of data item, as defined by the values it can take.

Number Systems

- There is different number systems other than decimal system (Binary, Octal, Hexadecimal, etc.)
- All data is represented by a bench of grouped cells of **0**'s and **1**'s in memory
- Bigger numbers needs bigger memory to represent
- Hexadecimal makes us deals with the streams of data with **0**'s and **1**'s more easier
- Octal like hexadecimal but it's no longer used in modern times

System	Base	Represent	Example	Symbol
Decimal	10	**0** to **9**	5924	non
Binary	2	**0** , **1**	0001011100100100	0b
Octal	8	**0** to **7**	13444	0
Hexadecimal	16	**0** to **9** , **A** to **F**	1724	0x

```
#include <iostream>
int main()
{
   // Representation in C++
   int decimal = 15 ;
   int binary = 0b1111 ;
   int octal = 017 ;
```

```
    int hexa = 0xF ;

    // Print Numbers
    std::cout << "decimal is " << decimal << std::endl ;
    std::cout << "binary is " << binary << std::endl ;
    std::cout << "octal is " << octal << std::endl ;
    std::cout << "hexa is " << hexa << std::endl ;
}

/*  Output
*
*   decimal is 15
*   binary is 15
*   octal is 15
*   hexa is 15
*/
```

Declaration and Initialization

```
// Variable braced initialization
Data_Type Variable_Name {Value} ; //

// Function variable initialization
Data_Type Variable_Name (Value) ;

// Assignment initialization
Data_Type Variable_Name = Value ;
```

> - Size of data type in memory
>
> ```
> std::cout << sizeof(data_type) ;
> ```

> The whole point of initialization with `{}` is to prevent or warn against potential data losses in this case.

Integer

- Stores decimal
- Typically occupies 4 bytes or more in memory (depend on system)

Initialization

1. Variable braced initialization

```
#include <iostream>
int main()
{
    /********************************* Variable braced initialization ***************************/
    // declaration and initialization
    int variable_X {} ;                 // Initializes with zero

    int variable_Y {10} ;               // Initializes with 10
```

```
    int variable_Z {15} ;                   // Initializes with 15
    int variable_F {variable_Y} ;       // Initializes with different exist variable

    // Error and Warning
    //int variable_H {variable_W} ;     // Initializes with different not exist variable -> compiler error
    //int variable_V {2.9} ;              // Initializes with different data type ->  error or warning

    std::cout << "variable_X: " << variable_X << std::endl ;
    std::cout << "variable_Y: " << variable_Y << std::endl ;
    std::cout << "variable_Z: " << variable_Z << std::endl ;
    std::cout << "variable_F: " << variable_F << std::endl ;
}

/*  Output
*
*   variable_X: 0
*   variable_Y: 10
*   variable_Z: 15
*   variable_F: 10
*/
```

2. Function variable initialization

```
#include <iostream>
int main()
{
    /********************************** Function variable initialization ***************************/
    // declaration and initialization
    int variable_Y (10) ;                   // Initializes with 10
    int variable_Z (15) ;                   // Initializes with 15
    int variable_F (variable_Y) ;       // Initializes with different exist variable


    // Error and Warning
    //int variable_H {variable_W} ;     // Initializes with different not exist variable -> compiler error
    int variable_V (2.9) ;              // Initializes with different data type ->  information lost


    std::cout << "variable_Y: " << variable_Y << std::endl ;
    std::cout << "variable_Z: " << variable_Z << std::endl ;
    std::cout << "variable_F: " << variable_F << std::endl ;
    std::cout << "variable_V: " << variable_V << std::endl ;
}

/*  Output
*
*   variable_Y: 10
*   variable_Z: 15
*   variable_F: 10
*   variable_V: 2
*/
```

3. Assignment initialization

```
#include <iostream>
int main()
{
   /********************************** Assignment initialization ***************************/
   // declaration integer
   int variable_name ;
   // initialization integer with 20
   variable_name = 20 ;
   // declaration and initialization
   int variable_X = 0 ;                  // Initializes with zero
   int variable_Y = 10 ;                  // Initializes with 10
   int variable_Z = 15 ;                  // Initializes with 15
   int variable_F = variable_Y ;             // Initializes with different exist variable

   int variable_V = 2.9  ;                  // Initializes with different data type


   std::cout << "variable_name: " << variable_name << std::endl ;
   std::cout << "variable_X: " << variable_X << std::endl ;
   std::cout << "variable_Y: " << variable_Y << std::endl ;
   std::cout << "variable_Z: " << variable_Z << std::endl ;
   std::cout << "variable_F: " << variable_F << std::endl ;
   std::cout << "variable_V: " << variable_V << std::endl ;
}

/*  Output
*
*   variable_name: 20
*   variable_X: 0
*   variable_Y: 10
*   variable_Z: 15
*   variable_F: 10
*   variable_V: 2
*/
```

Modifiers

1. Singed and unsigned

```
signed int variable_X {10} ;       // Initializes with 10
signed int variable_Y {-10} ;      // Initializes with -10

unsigned int variable_Z {10} ;     // Initializes with 10
unsigned int variable_F {-10} ;    // compiler error
```

Type with Modifiers	Bytes in memory	Range
unsigned int	4	**0** to **4,294,967,295**
signed int	4	**-2,147,483,648** to **2,147,483,647**

2. long and short

Type with Modifiers	Bytes in memory	Example
short, short int, signed short, signed short int	2	-32768
unsigned short int	2	455
long, long int, signed long, signed long int	4 or 8	-1588
unsigned long int	4 or 8	33
long long, long long int, signed long long, signed long long int	8	-459874
unsigned long long int	8	44658

```
#include <iostream>
int main()
{
   //short and long
   short short_var {-32768} ; //  2 Bytes
   short int short_int {455} ; //
   signed short signed_short {122}; //
   signed short int signed_short_int {-456}; //
   unsigned short int unsigned_short_int {456};

   int int_var {55} ; // 4 bytes
   signed signed_var {66};//
   signed int signed_int {77};//
   unsigned int unsigned_int{77};

   long long_var {88}; // 4 OR 8 Bytes
   long int long_int {33};
   signed long signed_long {44};
   signed long int signed_long_int {44};
   unsigned long int unsigned_long_int{44};

   long long long_long {888};// 8 Bytes
   long long int long_long_int {999};
   signed long long signed_long_long {444};
   signed long long int signed_long_long_int{1234};
   unsigned long long int unsigned_long_long_int{1234};
```

```
    std::cout << "Short variable, size : " << sizeof (short) << " bytes" << std::endl;
    std::cout << "Short Int, size : " << sizeof (short int) << " bytes" << std::endl;
    std::cout << "Signed short, size : " << sizeof (signed short) << " bytes" << std::endl;
    std::cout << "Signed short int, size : " << sizeof (signed short int) << " bytes" << std::endl;
    std::cout << "unsigned short int, size : " << sizeof (unsigned short int) << " bytes" << std::endl;
    std::cout << "---------------------" << std::endl;

    std::cout << "Int variable, size : " << sizeof (int) << " bytes" << std::endl;
    std::cout << "Signed variable, size : " << sizeof (signed) << " bytes" << std::endl;
    std::cout << "Signed int, size : " << sizeof (signed int) << " bytes" << std::endl;
    std::cout << "unsigned int, size : " << sizeof (unsigned int) << " bytes" << std::endl;
    std::cout << "---------------------" << std::endl;

    std::cout << "Long variable, size : " << sizeof (long) << " bytes" <<std::endl;
    std::cout << "Long int, size : " << sizeof (long int) << " bytes" << std::endl;
    std::cout << "Signed long, size : " << sizeof (signed long) << " bytes" << std::endl;
    std::cout << "Signed long int, size : " << sizeof (signed long int) << " bytes" << std::endl;
    std::cout << "unsigned long int, size : " << sizeof (unsigned long int) << " bytes" << std::endl;
    std::cout << "---------------------" << std::endl;

    std::cout << "Long long, size : " << sizeof (long long) << " bytes" << std::endl;
    std::cout << "Long long int, size : " << sizeof (long long int) << " bytes" << std::endl;
    std::cout << "Signed long long, size : " << sizeof (signed long long) << " bytes" <<std::endl;
    std::cout << "Signed long long int, size : " << sizeof (signed long long int) << " bytes" << std::endl;
    std::cout << "unsigned long long int, size : " << sizeof (unsigned long long int) << " bytes" << std::endl;
    std::cout << "---------------------" << std::endl;

    return 0;
}

/* Output
        Short variable, size : 2 bytes
        Short Int, size : 2 bytes
        Signed short, size : 2 bytes
        Signed short int, size : 2 bytes
        unsigned short int, size : 2 bytes
        ---------------------
        Int variable, size : 4 bytes
        Signed variable, size : 4 bytes
        Signed int, size : 4 bytes
        unsigned int, size : 4 bytes
        ---------------------
        Long variable, size : 4 bytes
        Long int, size : 4 bytes
        Signed long, size : 4 bytes
        Signed long int, size : 4 bytes
        unsigned long int, size : 4 bytes
        ---------------------
        Long long, size : 8 bytes
        Long long int, size : 8 bytes
        Signed long long, size : 8 bytes
        Signed long long int, size : 8 bytes

        unsigned long long int, size : 8 bytes
```

```
    --------------------
*/
```

Fraction

- A numerical quantity that is not a whole number (e.g. 1/2, 0.5).
- Floating point numbers memory representation by **IEEE 754**

Fraction Data Type

Type	Bytes in memory	Precision	Symbol (Suffixes)	Example
float	4	7	f	0.123456f
double	8	15	-	0.12345678912345
long double	12	> double	L	0.123456789123456789L

```
#include <iostream>
#include <iomanip>
int main()
{
   // Declare and initialize floating point
   float X {0.123456789123456789123456789f} ;
   double Y {0.123456789123456789123456789} ;
   long double Z {0.123456789123456789123456789L} ;

   // Size of data types
   std::cout << "Size of float: " << sizeof(float) << std::endl ;
   std::cout << "Size of double: " << sizeof(double) << std::endl ;
   std::cout << "Size of long double: " << sizeof(long double) << std::endl ;

   // Precision of floating point
   std::cout << std::setprecision(20) ;       // control srd::cout precision length
   std::cout << "float: " << X << std::endl ;
   std::cout << "double: " << Y << std::endl ;
   std::cout << "long double: " << Z << std::endl ;
}

/* Output
      Size of float: 4
      Size of double: 8
      Size of long double: 16

      float: 0.12345679104328155518          #valid 7 precision
      double: 0.1234567891234567838          #valid 15 precision
      long double: 0.12345678912345678912     #valid longer than 15 precision
*/
```

Narrowing Conversion

- Precision is a problem for a **float** data type because it's very limited.
- **double & long double** commonly used for floating point representation.

```
#include <iostream>
#include <iomanip>
int main()
{
   float X {123400081945.0f} ;
   double Y {123400081945.0} ;

   std::cout << std::setprecision(20) ;       // control srd::cout precision length
   std::cout << "float: " << X << std::endl ;
   std::cout << "double: " << Y << std::endl ;
}

/* Output
      float: 123400085504
      double: 123400081945
*/
```

Scientific Notation

- Scientific notation is a way of expressing numbers that are too large or too small to be conveniently written in decimal form.


```
#include <iostream>
#include <iomanip>
int main()
{
   double X {192400023} ;
   double Y {1.92400023e8} ;
   double Z {1.924e8} ;

   double A {0.00000000003498} ;
   double B {3.498e-11} ;

   std::cout << std::setprecision(20) ;       // control srd::cout precision length
   std::cout << "X: " << X << std::endl ;
   std::cout << "Y: " << Y << std::endl ;
   std::cout << "Z: " << Z << std::endl ;
   std::cout << "A: " << A << std::endl ;
   std::cout << "B: " << B << std::endl ;
}

/* Output

      X: 192400023
```

```
    Y: 192400023
    Z: 192400000
    A: 3.49799999999999998372e-11
    B: 3.49799999999999998372e-11
*/
```

Infinity and NaN

- Infinity happened when we try to divide **value** over **0**
- NaN happened when we try to divide **0.0** over **0.0**

```
#include <iostream>
int main()
{
   double X {5.6} ;
   double Y {-5.6} ;
   double Z {} ;

   std::cout << "X / Z = " << X / Z << std::endl;
   std::cout << "Y / Z = " << Y / Z << std::endl;
   std::cout << "Z / Z = " << Z / Z << std::endl;
}

/* Output
    X / Z = inf
    Y / Z = -inf
    Z / Z = nan
*/
```

Booleans

- A binary variable that can have one of two possible values, **0** (false) or **1** (true).
- Use in decision statement or function return
- Take one byte in memory

```
#include <iostream>
int main()
{
   bool red {false};
   bool green {true};

   //size of boolean
   std::cout << "Size of bool : " << sizeof(bool) << std::endl;

   if (red == true)
   {
      std::cout << "Stop!" << std::endl;
   }
   else
   {
      std::cout << "Go through!" << std::endl;
```

```
    }

    if (green)
    {
        std::cout << "The light is green!" << std::endl;
    }
    else
    {
        std::cout << "The light is NOT green!" << std::endl;
    }

    //1 -->> true
    //0 -->> false
    std::cout << "red : " << red << std::endl;
    std::cout << "green : " << green << std::endl;

    std::cout << std::boolalpha;            // control std::cout output for booleans data
    std::cout << "red : " << red << std::endl;
    std::cout << "green : " << green << std::endl;
}

/* Output
        Size of bool : 1
        Go through!
        The light is green!
        red : 0
        green : 1
        red : false
        green : true
*/
```

Characters

- A data type that holds one character (letter, number, etc.)
- Take one byte in memory (2^8 = 256 different values **0~255**)
- Use **ASCII** code for representation

ASCII TABLE

Decimal	Hexadecimal	Binary	Octal	Char	Decimal	Hexadecimal	Binary	Octal	Char	Decimal	Hexadecimal	Binary	Octal	Char
0	0	0	0	[NULL]	48	30	110000	60	0	96	60	1100000	140	`
1	1	1	1	[START OF HEADING]	49	31	110001	61	1	97	61	1100001	141	a
2	2	10	2	[START OF TEXT]	50	32	110010	62	2	98	62	1100010	142	b
3	3	11	3	[END OF TEXT]	51	33	110011	63	3	99	63	1100011	143	c
4	4	100	4	[END OF TRANSMISSION]	52	34	110100	64	4	100	64	1100100	144	d
5	5	101	5	[ENQUIRY]	53	35	110101	65	5	101	65	1100101	145	e
6	6	110	6	[ACKNOWLEDGE]	54	36	110110	66	6	102	66	1100110	146	f
7	7	111	7	[BELL]	55	37	110111	67	7	103	67	1100111	147	g
8	8	1000	10	[BACKSPACE]	56	38	111000	70	8	104	68	1101000	150	h
9	9	1001	11	[HORIZONTAL TAB]	57	39	111001	71	9	105	69	1101001	151	i
10	A	1010	12	[LINE FEED]	58	3A	111010	72	:	106	6A	1101010	152	j
11	B	1011	13	[VERTICAL TAB]	59	3B	111011	73	;	107	6B	1101011	153	k
12	C	1100	14	[FORM FEED]	60	3C	111100	74	<	108	6C	1101100	154	l
13	D	1101	15	[CARRIAGE RETURN]	61	3D	111101	75	=	109	6D	1101101	155	m
14	E	1110	16	[SHIFT OUT]	62	3E	111110	76	>	110	6E	1101110	156	n
15	F	1111	17	[SHIFT IN]	63	3F	111111	77	?	111	6F	1101111	157	o
16	10	10000	20	[DATA LINK ESCAPE]	64	40	1000000	100	@	112	70	1110000	160	p
17	11	10001	21	[DEVICE CONTROL 1]	65	41	1000001	101	A	113	71	1110001	161	q
18	12	10010	22	[DEVICE CONTROL 2]	66	42	1000010	102	B	114	72	1110010	162	r
19	13	10011	23	[DEVICE CONTROL 3]	67	43	1000011	103	C	115	73	1110011	163	s
20	14	10100	24	[DEVICE CONTROL 4]	68	44	1000100	104	D	116	74	1110100	164	t
21	15	10101	25	[NEGATIVE ACKNOWLEDGE]	69	45	1000101	105	E	117	75	1110101	165	u
22	16	10110	26	[SYNCHRONOUS IDLE]	70	46	1000110	106	F	118	76	1110110	166	v
23	17	10111	27	[ENG OF TRANS. BLOCK]	71	47	1000111	107	G	119	77	1110111	167	w
24	18	11000	30	[CANCEL]	72	48	1001000	110	H	120	78	1111000	170	x
25	19	11001	31	[END OF MEDIUM]	73	49	1001001	111	I	121	79	1111001	171	y
26	1A	11010	32	[SUBSTITUTE]	74	4A	1001010	112	J	122	7A	1111010	172	z
27	1B	11011	33	[ESCAPE]	75	4B	1001011	113	K	123	7B	1111011	173	{
28	1C	11100	34	[FILE SEPARATOR]	76	4C	1001100	114	L	124	7C	1111100	174	|
29	1D	11101	35	[GROUP SEPARATOR]	77	4D	1001101	115	M	125	7D	1111101	175	}
30	1E	11110	36	[RECORD SEPARATOR]	78	4E	1001110	116	N	126	7E	1111110	176	~
31	1F	11111	37	[UNIT SEPARATOR]	79	4F	1001111	117	O	127	7F	1111111	177	[DEL]
32	20	100000	40	[SPACE]	80	50	1010000	120	P					
33	21	100001	41	!	81	51	1010001	121	Q					
34	22	100010	42	"	82	52	1010010	122	R					
35	23	100011	43	#	83	53	1010011	123	S					
36	24	100100	44	$	84	54	1010100	124	T					
37	25	100101	45	%	85	55	1010101	125	U					
38	26	100110	46	&	86	56	1010110	126	V					
39	27	100111	47	'	87	57	1010111	127	W					
40	28	101000	50	(	88	58	1011000	130	X					
41	29	101001	51	)	89	59	1011001	131	Y					
42	2A	101010	52	*	90	5A	1011010	132	Z					
43	2B	101011	53	+	91	5B	1011011	133	[					
44	2C	101100	54	,	92	5C	1011100	134	\					
45	2D	101101	55	-	93	5D	1011101	135	]					
46	2E	101110	56	.	94	5E	1011110	136	^					
47	2F	101111	57	/	95	5F	1011111	137	_					

> - It's possible to assign a valid ASCII code to a char variable, and the corresponding character will be stored in.
>
>> - ASCII was the first encoding representation text in a computer.
>> - It doesn't represent all characters (e.g. Arabic, east Asian language).
>> - There's better way to represent character in C++ like **Unicode**

```
#include <iostream>
int main()
{
   char character1 {'a'};
   char character2 {'r'};
   char character3 {'r'};
   char character4 {'o'};
   char character5 {'w'};

   std::cout << character1 << std::endl;
   std::cout << character2 << std::endl;
   std::cout << character3 << std::endl;
   std::cout << character4 << std::endl;
   std::cout << character5 << std::endl;

   char value = 65 ;                              // ASCII character code for 'A'

   std::cout << "value : " << value << std::endl;  // A
```

```
    std::cout << "value as intager : " << static_cast<int>(value) << std::endl;
}

/* Output
       a
       r
       r
       o
       w
       value : A
       value as intager : 65
*/
```

Auto

Let the compiler deduce the type

```
#include <iostream>
int main()
{
    auto A {12};       // int
    auto B {13.0};     // double
    auto C {14.0f};    // float
    auto D {15.0l};    // long double
    auto E {'e'};      // char
    auto F { 123u};    // unsigned int
    auto G { 123ul};   //unsigned long int
    auto H { 123ll};   // long long int

    std::cout << "A occupies : " << sizeof(A) << " bytes" << std::endl;
    std::cout << "B occupies : " << sizeof(B) << " bytes" << std::endl;
    std::cout << "C occupies : " << sizeof(C) << " bytes" << std::endl;
    std::cout << "D occupies : " << sizeof(D) << " bytes" << std::endl;
    std::cout << "E occupies : " << sizeof(E) << " bytes" << std::endl;
    std::cout << "F occupies : " << sizeof(F) << " bytes" << std::endl;
    std::cout << "G occupies : " << sizeof(G) << " bytes" << std::endl;
    std::cout << "H occupies : " << sizeof(H) << " bytes" << std::endl;
}

/* Output
       A occupies : 4 bytes
       B occupies : 8 bytes
       C occupies : 4 bytes
       D occupies : 16 bytes
       E occupies : 1 bytes
       F occupies : 4 bytes
       G occupies : 4 bytes
       H occupies : 8 bytes
*/
```

Assignment

- Change the data after declare it with another value
- Don't change the value of **auto** data type with another to avoid garbage value

```
#include <iostream>
int main()
{
   int A{123}; // Declare and initialize
   std::cout << "A : "  << A << std::endl;

   A = 55; // Assign
   std::cout << "A : "  << A << std::endl;

   std::cout << "----------------" << std::endl;

   double B {44.55}; // Declare and initialize
   std::cout << "B : " << B << std::endl;

   B = 99.99; // Assign
   std::cout << "B : " << B << std::endl;

   std::cout << "----------------" << std::endl;

   bool state{false}; // Declare and initialize
   std::cout << std::boolalpha;
   std::cout << "state : " << state << std::endl;

   state = true; // Assign
   std::cout << "state : " << state << std::endl;

   std::cout << "----------------" << std::endl;

   auto C {333u}; // Declare and initialize with type deduction
   std::cout << "C : " << C << std::endl;

   C = -22; // Assign negative number. DANGER!
   std::cout << "C : " << C << std::endl;
}

/* Output
      A : 123
      A : 55
      ----------------
      B : 44.55
      B : 99.99
      ----------------
      state : false
      state : true
      ----------------
      C : 333
      C : 4294967274
*/
```

Enumeration

An enumeration is a user-defined data type that consists of integral constants. To define an enumeration, keyword `enum` is used. Each enumeration is represented by an integral value under the hood.

```
// Declaration
enum Enumeration_Name {variable_0, variable_1, variable_2, ...., variable_n};

// Initilization
Enumeration_Name Variable_Name {variable_x} ;
```

By default, variable_0 is 0, variable_1 is 1 and so on. You can change the default value of an `enum` element during declaration (if necessary).

```
enum Enumeration_Name
{
   variable_0 = 5,
   variable_1 = 3,
   variable_2 = 79
};
```

When you create an enumerated type, only blueprint for the variable is created. Here's how you can create variables of `enum` type.

```
enum boolean { false, true };

// inside function
enum boolean check;

/********************************** using different syntax ***********************************/
enum boolean
{
  false, true
} check;
```

```
enum Colour { RED, GOLD, GREEN, BLUE };  // No typedef required
enum Metal  { GOLD, SILVER, BRONZE };    // Oops re-definition of GOLD!

Colour c = RED;                          // Colour is a new first-
                                         // class type

Metal m = GOLD                           // <= FAIL - this is Colour's
                                         // GOLD, not Metal's!

int a = c;                               // OK
c = 7;                                   // Not allowed
```

enum class

C++11 has introduced `enum` classes (also called scoped enumerations), that makes enumerations both strongly typed and strongly scoped. Class `enum` doesn't allow implicit conversion to int, and also doesn't compare enumerators from different enumerations. We can now specify the underlying type of the enumeration (as long as it's an integer type). The default is integer; as with C++98.

```
enum EE : unsigned long {EE_ONE = 1, EE_TWO= 2, EE_BIG = 0xFFFFFFF0U};
```

```
// Declaration
enum class Enumeration_Name {variable_0, variable_1, variable_2, ...., variable_n};

// Initilization
Enumeration_Name Variable_Name {Enumeration_Name::variable_x} ;
```

```
enum Colour { RED, GOLD, GREEN, BLUE };  // No typedef required
enum Metal  { GOLD, SILVER, BRONZE };    // Oops re-definition of GOLD!

Colour c = RED;                          // Colour is a new first-
                                         // class type

Metal m = GOLD                           // <= FAIL - this is Colour's
                                         // GOLD, not Metal's!

int a = c;                               // OK
c = 7;                                   // Not allowed
```

using enum C++20

A `using-enum-declaration` introduces the enumerator names of the named enumeration as if by a `using-declaration` for each enumerator. When in class scope, a `using-enum-declaration` adds the enumerators of the named enumeration as members to the scope, making them accessible for member lookup.

```
enum class Month
{
   jan, feb, mar, apr,
   may, jun, jul, aug,
   sep, oct, nov, dec
};

// without using enum

std::string_view month_to_string (Month month)
{
```

```
    switch (month)
    {
        case Month::jan : return "january" ;
        case Month::feb : return "february" ;
        .
        .
        .
    }
}

// with using enum

std::string_view month_to_string (Month month)
{
    switch (month)
    {
        using enum Month
        case jan : return "january" ;
        case feb : return "february" ;
        .
        .
        .
    }
}
```

Type Aliases

Type alias is a name that refers to a previously defined type, similar to `typedef`.

```
using Identifier_Name = Data_Type ;
Identifier_Name Variable_Name = Value ;
```

> Recommended in modern C++

Variable Lifetime

The period of time in which a variable is alive in memory. It becomes alive when you declare it and it is killed (wiped out) from memory at some point.

1. Local Duration: dies at the end of the block
2. Static Duration: dies at the end of the program
3. Dynamic Duration: you decide when it dies

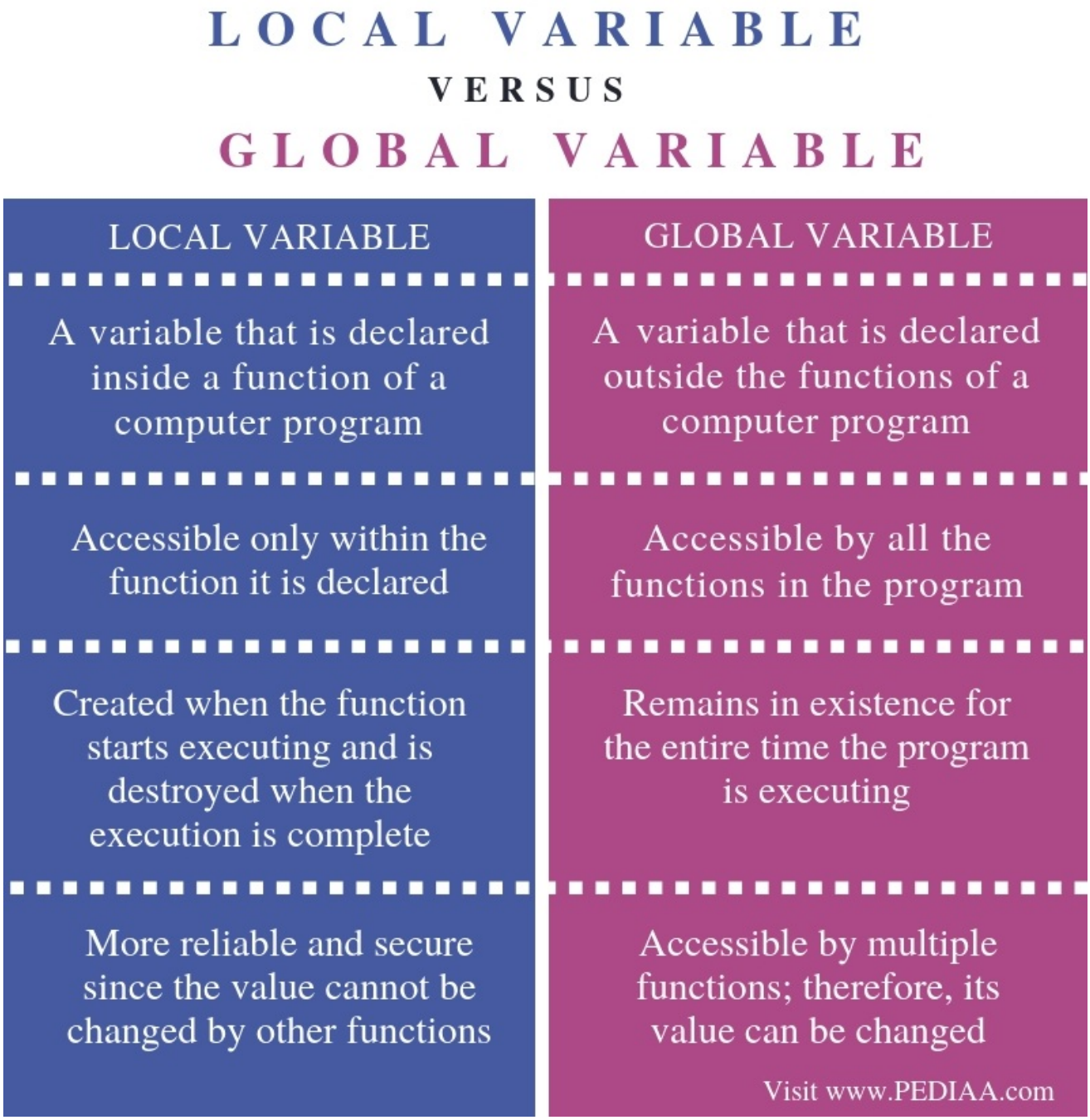
Variable Scope

A region in your code where a variable name can be mentioned. You may be reading from it, writing into it, basically using it in any conceivable way.

> Trying to use a variable out of it's scope will result in a compiler error.

1. Global Variable: it's a variable with global scope, meaning that it is visible throughout the program.

2. Local Variable: it's a variable that is given local scope. Local variable references in the function or block in which it is declared override the same variable name in the larger scope.



Operations on Data

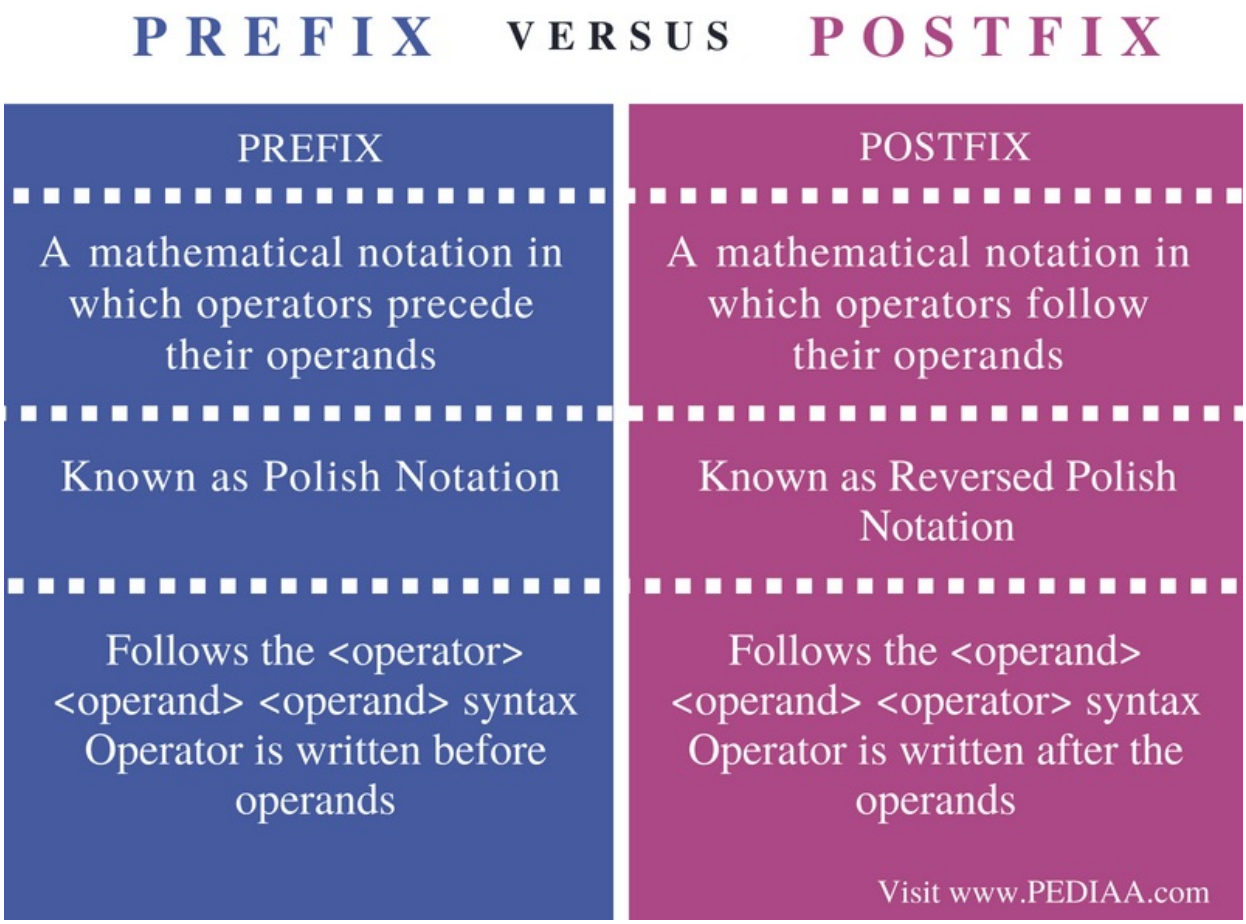
Basic Operations

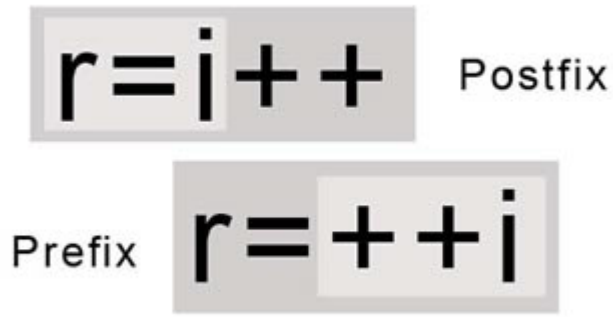
Operation	Symbol	Operands	Example
Addition	+	2	5 + 26 = 31
Subtraction	-	2	36 - 5 = 31
Multiply	*	2	6 * 5 = 30
Division	/	2	31 / 10 = 3
Modulus	%	2	31 % 10 = 1

Increment and Decrement

Operation	Symbol	Operands	Example
Increment	++	1	X++ or ++X
Decrement	--	1	Y-- or --Y

> Prefix vs. Postfix.





Assignment Operators

Operation	Symbol	Operands	Example
sum equal	+=	2	A += 5
subtract equal	-=	2	B -= 5
multiply equal	*=	2	C *= 5
divide equal	/=	2	D /= 5
modulus equal	%=	2	E %= 5

Relational Operators

Operation	Symbol	Operands	Example
less than	<	2	X < Y
less than or equal	<=	2	X <= Y
bigger than	>	2	X > Y
bigger than or equal	>=	2	X >= Y
equal	==	2	X == Y
not equal	!=	2	X != Y

> return Booleans value (**true** or **false**).

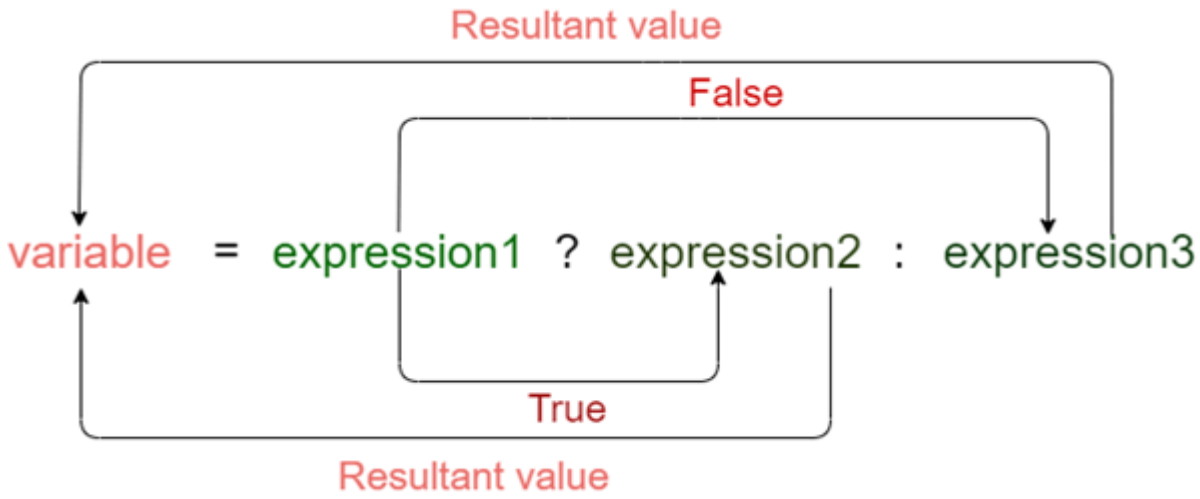
Logical Operator

Operation	Symbol	Operands	Example
AND	&&	2	X && Y
OR	||	2	X || Y
NOT	!	1	!X or !Y

Ternary Operators

The conditional operator is kind of similar to the if-else statement as it does follow the same algorithm as of if-else statement but the conditional operator takes less space and helps to write the if-else statements in the shortest way possible.

```
variable = (Expression1) ? Expression2 : Expression3
```



> all Expressions must be can compared (same data type).

Comma Operator

- The comma operator (represented by the token `,`) is a binary operator that evaluates its first operand and discards the result, it then evaluates the second operand and returns this value (and type). The comma operator has the lowest precedence of any C operator, and acts as a sequence point.
- Comma acts as a separator when used with function calls and definitions, function like macros, variable declarations, enum declarations, and similar constructs.
- We know that in C and C++, every statement is terminated with a semicolon but comma operator also used to terminate the statement after satisfying the following rules.

1. The variable declaration statements must be terminated with semicolon.
2. The statements after declaration statement can be terminated by comma operator.
3. The last statement of the program must be terminated by semicolon.

Precedence and Associativity

Precedence	Operator	Description	Associativity
1	::	Scope resolution	Left-to-right →
2	a++ a-- *type*() *type*{} a() a[] . ->	Suffix/postfix increment and decrement Functional cast Function call Subscript Member access	
3	++a --a +a -a ! ~ (*type*) *a &a sizeof co_await new new[] delete delete[]	Prefix increment and decrement Unary plus and minus Logical NOT and bitwise NOT C-style cast Indirection (dereference) Address-of Size-of[note 1] await-expression (C++20) Dynamic memory allocation Dynamic memory deallocation	Right-to-left ←
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 ≤ and > and ≥ respectively	
10	== !=	For equality operators = and ≠ respectively	
11	a&b	Bitwise AND	
12	^	Bitwise XOR (exclusive or)	
13	|	Bitwise OR (inclusive or)	
14	&&	Logical AND	
15	||	Logical OR	
16	a?b:c throw co_yield = += -= *= /= %= <<= >>= &= ^= |=	Ternary conditional[note 2] throw operator 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	Right-to-left ←
17	,	Comma	Left-to-right →

Control Flow

Control flow is the order in which individual statements, instructions or function calls of an imperative program are executed or evaluated.

if Statement

```
// use for testing one condition
if (condition)
{
   // Code if the condition is true
}
else
{
   // Code if the condition is false
}
```

> - Condition return true or false only
> - True conditions: is any number different than 0, or expression evaluating to something other than 0
> - False conditions: is any number equal to 0, or expression evaluating to 0

else if Statement

```
// use for testing multiple conditions
if (condition 1)
{
   // Code if the condition 1 is true
}
else if (condition 2)
{
   // Code if the condition 2 is true
}
else if (condition 3)
{
   // Code if the condition 3 is true
}
else
{
   // Code if the three conditions is false
}
```

switch Statement

```
// use for testing several different conditions
switch (condition)
{
   case A:
      // code if the variable match with case A
   break ;
   case B:
      // code if the variable match with case B
   break ;
   case C:
      // code if the variable match with case C
   break ;
   default:
```

```
    // code if the variable not matching with any case
  break ;
}
```

> if the **break** isn't existed after the **case** is finished, the program will continue to the next **case** until finding a **break** or end of the **switch** statement

Short Circuit Evaluation

Short-circuiting is a programming concept by which the compiler skips the execution or evaluation of some sub-expressions in a logical expression. The compiler stops evaluating the further sub-expressions as soon as the value of the expression is determined.

```
int main()
{
   int a = 10;
   int b = -1;

   // Here b == -1 is not evaluated as
   // a != 10 is false
   if (a != 10 && b == -1) {
      printf("I won't be printed!\n");
   }
   else {
      printf("Hello, else block is printed");
   }

   a = 0;
   // myfunc(b) will not be called
   if (a != 0 && myfunc(b)) {
      // do_something();
   }

   a = 15;
   // Here since a is 10, calculate_sqrt
   // function will be called
   if (a >= 10 && calculate_sqrt(a)) {
      printf("I will be printed!\n");
   }

   return 0;
}
```

if constexpr

a C++17 feature which allows conditionally compiling code based on template parameters in a clear and minimal fashion.

```
constexpr bool condition {false} ;
if constexpr (condition)
{
   // Code if the condition is true
}
else
{
   // Code if the condition is false
}
```

> condition must be constexpr or exist before compile time

if with Initializer

a C++17 has extended existing if statement's syntax. Now it is possible to provide initial condition within if statement itself. This new syntax is called "if statement with initializer". This enhancement simplifies common code patterns and helps users keep scopes tight. Which in turn avoids variable leaking outside the scope.

```
if (init; condition)
{
  // Code if the condition is true
} else
{
  // Code if the condition is false
}
```

switch with Initializer

```
switch (init; condition)
{
   case A:
      // code if the variable match with case A
   break ;
   case B:
      // code if the variable match with case B
   break ;
   case C:
      // code if the variable match with case C
   break ;
   default:
      // code if the variable not matching with any case
   break ;
}
```

Loops

- A loop is used for executing a block of statements repeatedly until a particular condition is satisfied.
- the main components of a loop are

1. Iterator
2. Starting Point
3. Test(controls when the loop stops)
4. Increment(Decrement)
5. Loop body

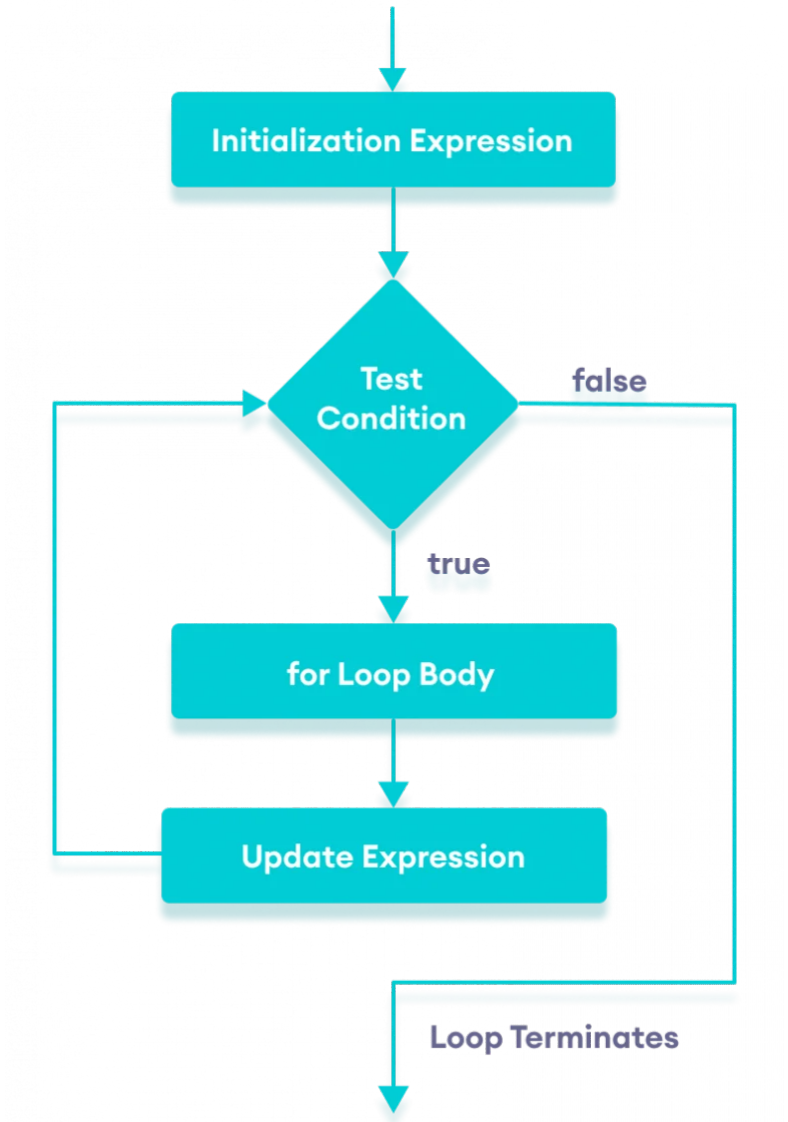
for Loop

```
for (initialization; condition; update)
{
   // body of-loop
}
```

or we can initialize the iterator outside the loop scope

```
Data_Type initialization ;
for ( ; condition; update)
{
   // body of-loop
}
```

- `initialization` - initializes variables and is executed only once
- `condition` - if `true`, the body of `for` loop is executed, if `false`, the for loop is terminated
- `update` - updates the value of initialized variables and again checks the condition
  `Iterator` must be integral data type, we can use `size_t` instead of `unsigned int`



We can use multiple declarations and updates in for loop

```
for (initialization1, initialization2, ...; condition; update1, update2, ...)
{
   // body of-loop
}
```

Rang Based for Loop

In C++11, a new range-based `for` loop was introduced to work with collections such as **arrays** and **vectors**.

```
for (variable : collection)
{
   // body of loop
}
```

Example

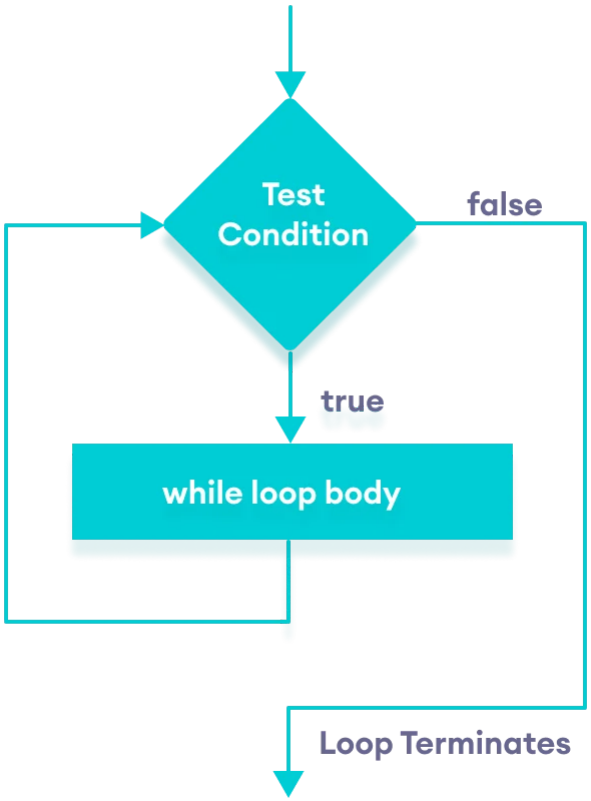
```
#include <iostream>

int main()
{
   int num_array[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

   for (int n : num_array)
   {
      cout << n << " ";
   }
   return 0;
}

/*
* Output:1 2 3 4 5 6 7 8 9 10
*/
```

> Range based for loop with initializer, helper variable in loop scope (C++20)
>
> ```
> for (initialization; variable : collection)
> {
>    // body of loop
> }
> ```

while Loop

```
while (condition)
{
   // body of the loop
}
```

- A `while` loop evaluates the `condition`
- If the `condition` evaluates to `true`, the code inside the `while` loop is executed.
- The `condition` is evaluated again.
- This process continues until the `condition` is `false`.
- When the `condition` evaluates to `false`, the loop terminates.

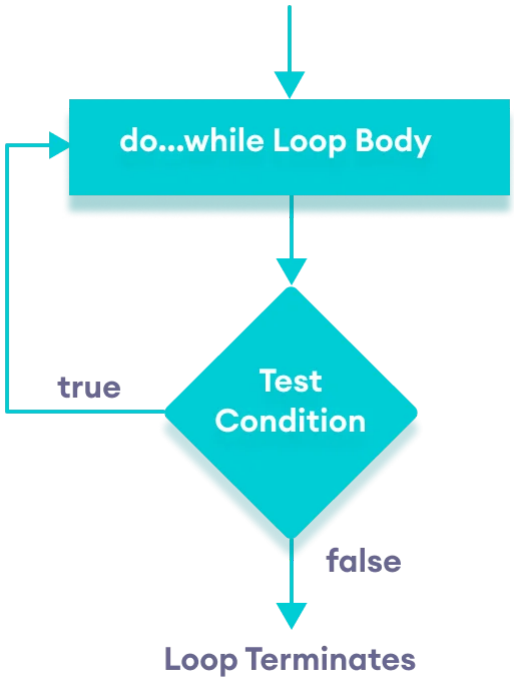


do while Loop

The `do...while` loop is a variant of the `while` loop with one important difference: the body of `do...while` loop is executed once before the `condition` is checked.

```
do
{
  // body of loop;
}
while (condition);
```

- The body of the loop is executed at first. Then the `condition` is evaluated.
- If the `condition` evaluates to `true`, the body of the loop inside the `do` statement is executed again.
- The `condition` is evaluated once again.
- If the `condition` evaluates to `true`, the body of the loop inside the `do` statement is executed again.
- This process continues until the `condition` evaluates to `false`. Then the loop stops.



Infinite Loops

If the `condition` in a loops are always `true`, it runs forever (until memory is full).

```
for( ; ; )
{
    // body of loop;
}
```

```
while ()
{
  // body of the loop
}
/*******************************/
while (true)
{
  // body of the loop
}
```

```
do
{
  // body of loop;
}
while ();
/*******************************/
do
{
  // body of loop;
}
while (true);
```

Nested Loops

A loop within another loop is called a nested loop.

```
for (initialization; condition; update)
{
   // body of-loop
   for (initialization; condition; update)
   {
      // body of-loop
      for (initialization; condition; update)
      {
         // body of-loop
         // we can do this as we need
      }
   }
}
```

```
while (condition)
{
   // body of the loop
   while (condition)
   {
      // body of the loop
      while (condition)
      {
         // body of the loop
         // we can do this as we need
      }
   }
}
```

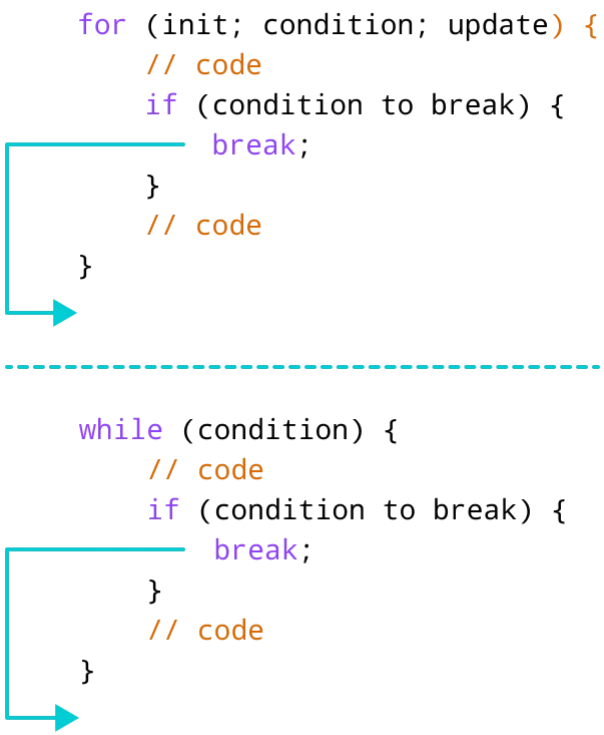
```
do
{
  // body of loop;
   do
   {
     // body of loop;
     // we can do this as we need
   }
   while (condition);
}
while (condition);
```

break

the `break` statement terminates the loop when it is encountered.

```
break ;
```

> `break` statement is usually used with decision-making statements.

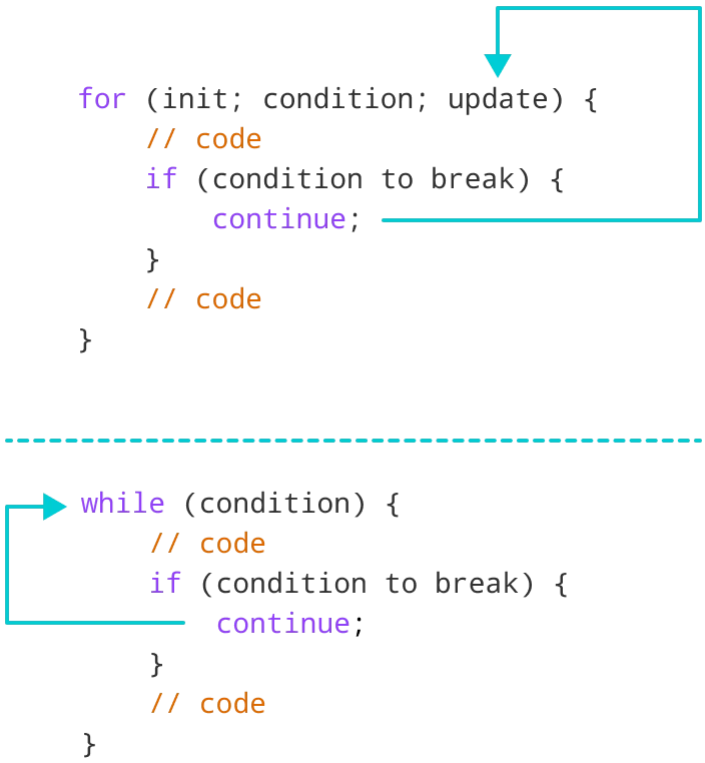


continue

In computer programming, the `continue` statement is used to skip the current iteration of the loop and the control of the program goes to the next iteration.

```
continue ;
```

> `continue` statement is almost always used with decision-making statements.



Arrays

An array is a collection of elements of the same type placed in contiguous memory locations that can be individually referenced by using an index to a unique identifier.

Declaration and Initialization

```
// Declaration
Data_Type arr_name [arr_size] ;

// Declaration and Initialization
Data_Type arr_name [arr_size] {item1, item2, item3, ..., item(arr_size-1)} ;
```

```
// declare and initialize and array
int x[6] {19, 10, 8, 17, 9, 15};

/*************************** or ***************************/

// declare and initialize an array
int x[] {19, 10, 8, 17, 9, 15};
```

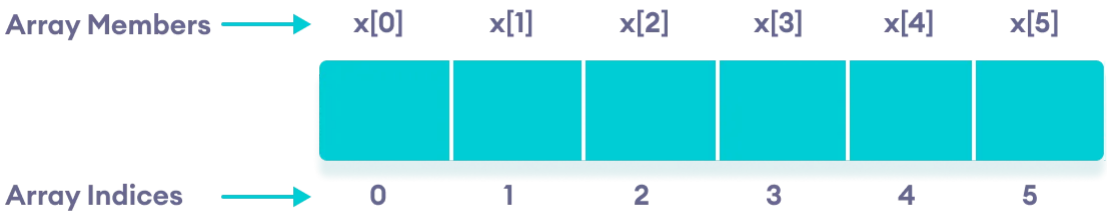
If an array has a size `n`, we can store up to n number of elements in the array. In such cases, the compiler assigns random values to the remaining places. Oftentimes, this random value is simply `0`.

```
// store only 3 elements in the array
int x[6] {19, 10, 8};
```

Array Access

In C++, each element in an array is associated with a number. The number is known as an array index. We can access elements of an array by using those indices.

```
// syntax to access array elements
arr[index];
```



- The array indices start with `0`. Meaning x[0] is the first element stored at index `0`.
- If the size of an array is `n`, the last element is stored at index `(n-1)`.
- Elements of an array have consecutive addresses. For example, suppose the starting address of `x[0]` is 2120d. Then, the address of the next element `x[1]` will be 2124d, the address of `x[2]` will be 2128d and so on.

- If we declare an array of size 10, then the array will contain elements from index 0 to 9, if we try to access the element at index 10 or more than 10, it will result in Undefined Behavior.

Size of Arrays

Before C++ 17 there's only way to find size of an array.

```
size = sizeof(arr_name) / sizeof(arr_name[0]) ;
```

After C++ 17 we got a new way to find the size of an array.

```
size = std::size(arr_name) ;
```

> ### Omit Array Size
>
> You don't have to specify the size of the array. But if you don't, it will only be as big as the elements that are inserted into it, but it's available for only one the first dimension of the array.
>
> ```
> Data_Type arr_name [] ;
> Data_Type arr_name [][2nd_D_arr_size] ;
> Data_Type arr_name [][2nd_D_arr_size][3rd_D_arr_size] ;
> ```

Array of Characters

String is a collection of characters. There are two types of strings commonly used in C++ programming language:

- Strings that are objects of string class (The Standard C++ Library string class)
- C-strings (C-style Strings)

Now we'll just take C-strings. In C programming, the collection of characters is stored in the form of arrays. This is also supported in C++ programming. Hence it's called C-strings.

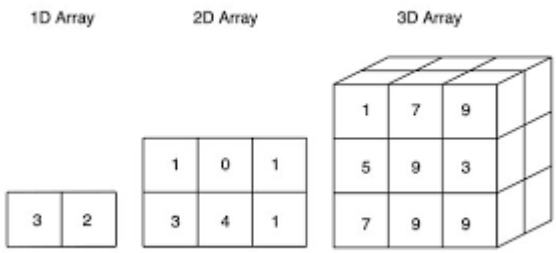
C-strings are arrays of type `char` terminated with null character, that is, `\0` (ASCII value of null character is 0).

```
char string_name[] = "string_data" ;

char string_name[] {"string_data"} ;

char string_name[5] {"Data"} ;

char string_name[] {'D', 'a', 't', 'a', '\0'} ;

char string_name[5] {'D', 'a', 't', 'a', '\0'} ;
```

> `cctype` and `cstring` are a libraries containing different functions to deal with strings.

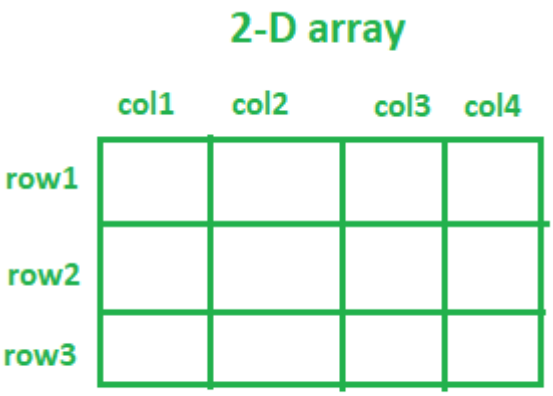
Multi Dimensional Array

In C++, we can create an array of an array, known as a multidimensional array.



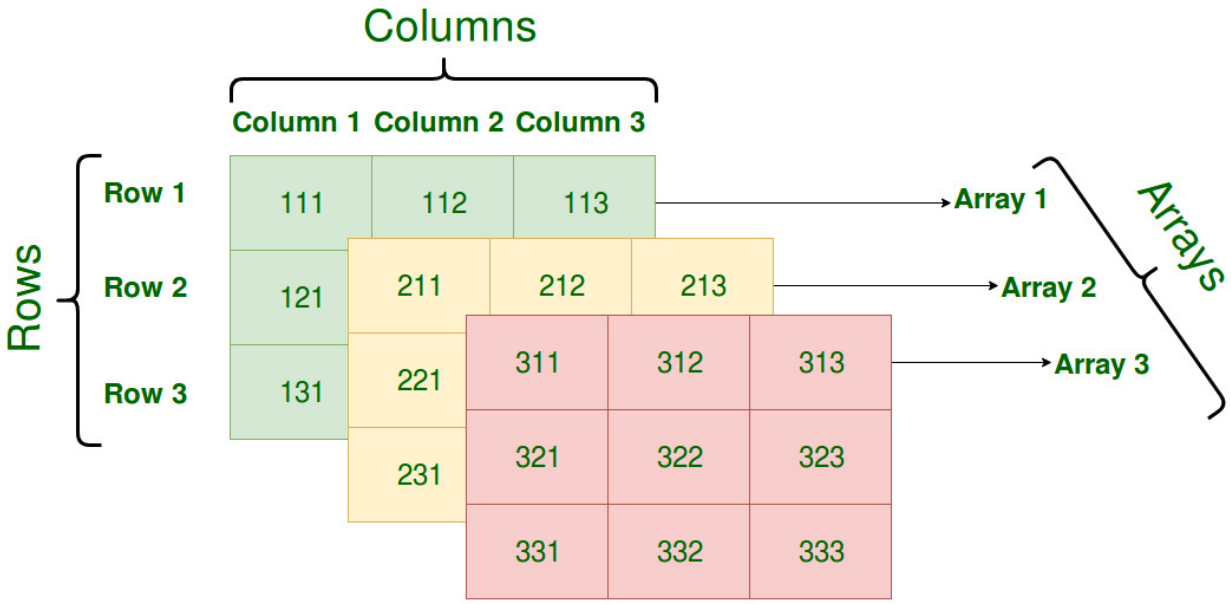
2D Array

```
/***************************** 2D Array ******************************/
// Declaration
Data_Type arr_name [1st_D_arr_size][2nd_D_arr_size] ;

// Declaration and Initialization
Data_Type arr_name [1st_D_arr_size][2nd_D_arr_size] {{1st_D_Data}, {2nd_D_Data}} ;
```



3D Array

```
/***************************** 3D Array ******************************/
// Declaration
Data_Type arr_name [1st_D_arr_size][2nd_D_arr_size][3rd_D_arr_size] ;

// Declaration and Initialization
Data_Type arr_name [1st_D_arr_size][2nd_D_arr_size][3rd_D_arr_size]
{    /*************** 2nd_D_Data ***************/
   { {3rd_D_Data}, {3rd_D_Data},  {3rd_D_Data} }, // 1st_D_Data
   { {3rd_D_Data}, {3rd_D_Data},  {3rd_D_Data}},  // 1st_D_Data
   { {3rd_D_Data}, {3rd_D_Data},  {3rd_D_Data}},  // 1st_D_Data
} ;
```



Pointers

Pointers are variables that store the memory addresses of other variables. If we have a variable `var` in our program, `&var` will give us its address in the memory.

Declaration and Initialization

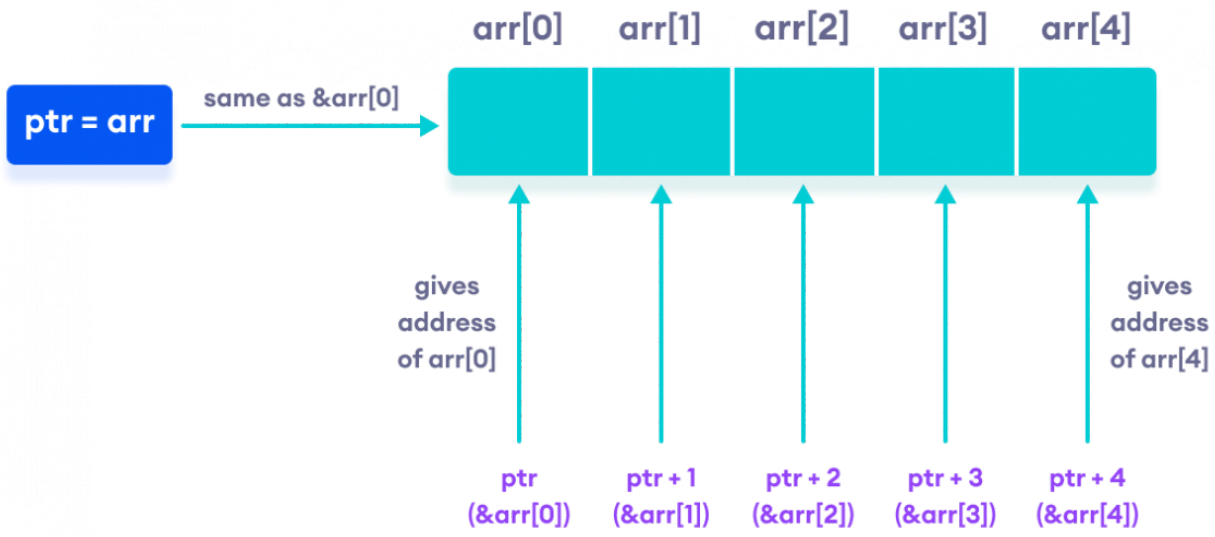
```
// Declaration
Data_Type *ptr ;

// Declaration and Initialization
Data_Type *ptr {Value} ;

// Initialization
int var {} ;
ptr = &var ;
```

- * it's called the **dereference operator**. It operates on a pointer and gives the value pointed by the address stored in the pointer. That is, `*pointVar = var`.
- `ptr` and `*ptr` is completely different. We cannot do something like `*ptr = &var;`
- Pointer to another variable must be same data type or get compiler error.

```
// wirting into uninitialized pointer is very BAD!!!
int * ptr ; // unkown address
* ptr = 5 ; // very BAB, writing in junk address
// must initilize the pointer before use it.
```

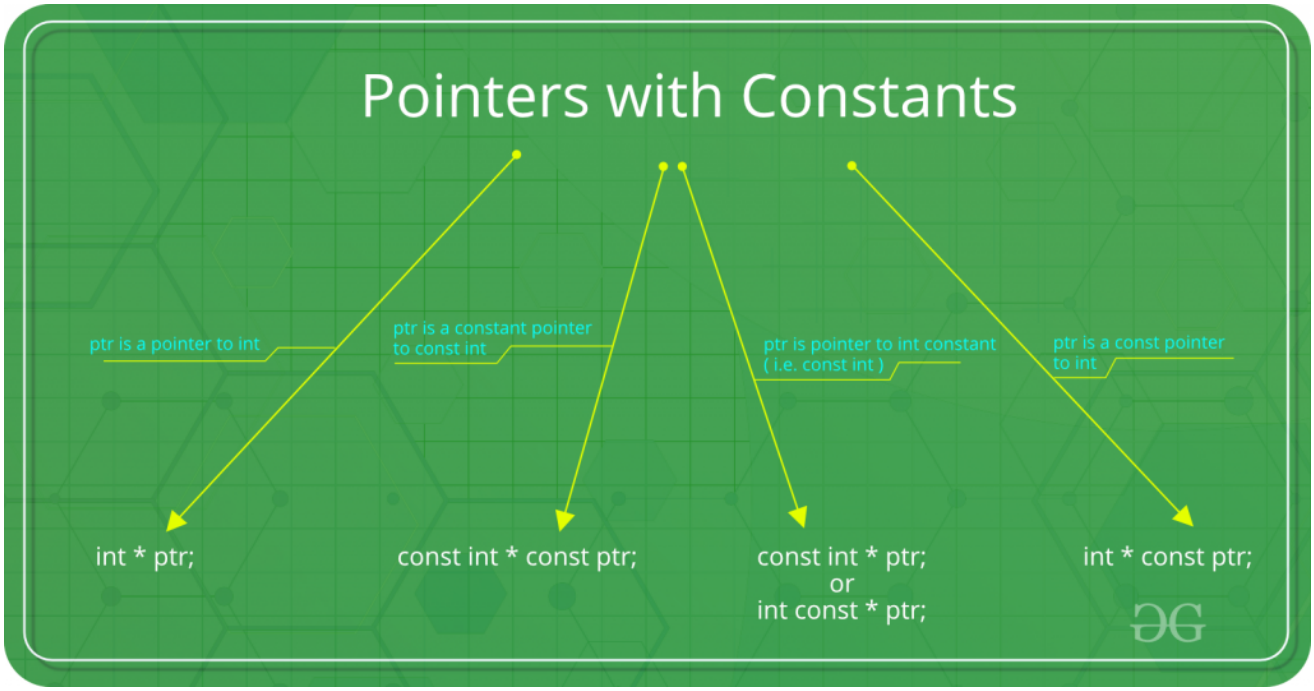
Pointer to Array

```
int *ptr;
int arr[5];
ptr = arr;
```



> Array names decay to pointers. In simple words, array names are converted to pointers. That's the reason why we can use pointers to access elements of arrays.

Constant with Pointers



1. Non const pointer to Non const data

```
/*********** Non const pointer to Non const data ***********/
int * ptr {} ;
int var {10} ;

ptr = &var ;

// can change the pointed value to another
*ptr = 5 ;

// can change the pointer itself
int x {20} ;
ptr = &x ;
```

2. Non const pointer to const data

```
/*********** Non const pointer to const data ***********/
int const * ptr {} ;
const int var {10} ;

ptr = &var ;

// can't change the pointed value to another
*ptr = 5 ; // compiler error

// can change the pointer itself
int x {20} ;
ptr = &x ;
```

3. const pointer to Non const data

```
/*********** const pointer to Non const data ***********/
int * const ptr {} ;
const int var {10} ;

ptr = &var ;

// can change the pointed value to another
*ptr = 5 ;

// can't change the pointer itself
int x {20} ;
ptr = &x ; // compiler error
```

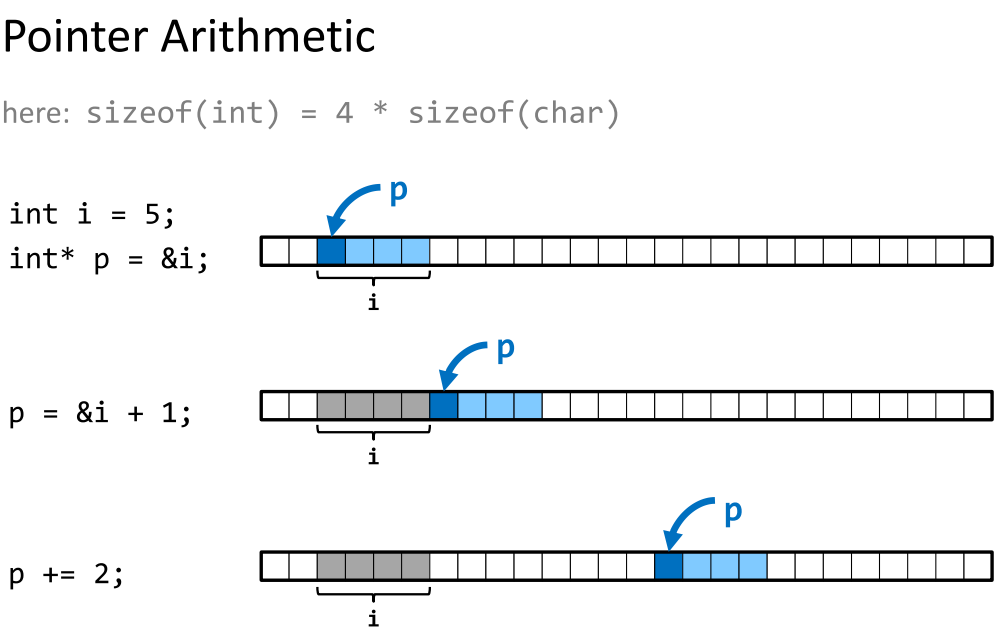
4. const pointer to const data

```
/*********** const pointer to const data ***********/
int const * const ptr {} ;
const int var {10} ;

ptr = &var ;

// can't change the pointed value to another
*ptr = 5 ; // compiler error

// can't change the pointer itself
int x {20} ;
ptr = &x ; // compiler error
```

> One way to remember the syntax (according to Bjarne Stroustrup) is the `spiral rule`.
> The rule says, start from the name of the variable and move clockwise to the next pointer or type.
> Repeat until expression ends.

Pointer Arithmetic

A set of operations we can do on the pointer representing the array to manipulate the array. These operations can include navigating from element to element, computing the distance between elements and comparing addresses of elements.



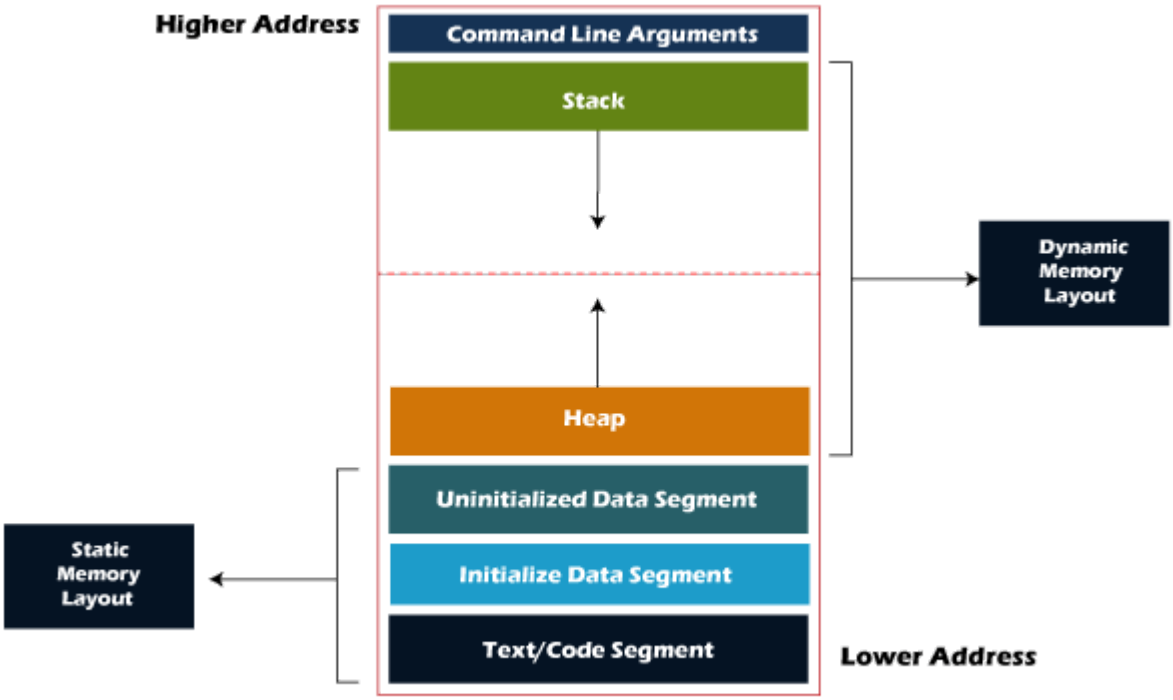
> `std::ptrdiff_t` is the signed integer type of the result of subtracting two pointers.

Dynamic Memory Allocation

Lot's of programs running on our OS. We can quickly run our of memory.

A trick that fools your program into thinking it is the only program running on your OS, and all memory resources belong to it. Each program is abstracted into a process, and each process has access to the memory range 0 ~ 2^N, where N is 32 on 32 bit systems and 64 on 64 bit.

The entire program is not loaded in real memory by the CPU and MMU. Only parts that are about to be executed are loaded. Making effective use of real memory, a valuable and lacking resource. The memory map is a standard format defined by the OS. All programs written for that OS must conform to it. It is usually divided into some sections.



Memory Layout

So far we've only been using memory allocated on the stack. We are going to see how one can allocate memory from the heap, and some of the differences between these mechanisms.

stack vs heap

We can allocate and then deallocate memory dynamically using the `new` and `delete` operators respectively.

1. `new` operator returns the address of the **memory location**.

```
Data_Type *ptr = new Data_Type ;

/************************ Example ************************/
// declare an int pointer
int * ptr;

// dynamically allocate memory
// using the new keyword
ptr = new int;

// assign value to allocated memory
*ptr = 45;
```

2. `delete` operator is used. It returns the memory to the operating system. This is known as **memory deallocation**.

```
delete ptr;

/************************ Example ************************/
// declare an int pointer
int * ptr;

// dynamically allocate memory
// using the new keyword
ptr = new int;

// assign value to allocated memory
*ptr = 45;

// print the value stored in memory
cout << *ptr; // Output: 45

// deallocate the memory
delete ptr;
```

> If the program uses a large amount of unwanted memory using `new`, the system may crash because there will be no memory available for the operating system. In this case, the `delete` operator can help the system from crash.

In some rare cases, the 'new' operator will fail to allocate dynamic memory from the heap. When that happens, and you have no mechanism in place to handle that failure, an exception will be thrown and your program will crash. 'new' fails very rarely in practice and you'll see many programs that assume that it always works and don't check for memory allocation failure in any way. Depending on your application, failed memory allocations can be very bad and you need to check and handle them.

```
int * data = new int[10000000000000000]; //CRASH!
```

There's two ways to check for failed memory allocations:

1. Through the exception mechanism

```
//exception
for(size_t i{0} ; i < 100 ; ++i)
{
   try
   {
      int * data = new int[1000000000];
   }
   catch(std::exception& ex)
   {
      std::cout << "  Something went wrong : " << ex.what() << std::endl;
   }

}
```

2. `std::notthrow` setting

```
//std::nothrow
for(size_t i{0} ; i < 100 ; ++i)
{
   int * data = new(std::nothrow) int[1000000000];
   if(data!=nullptr)
   {
      std::cout << "Data allocated" << std::endl;
   }
   else
   {
      std::cout << "Data allocation failed" << std::endl;
   }
}
```

Dynamic Array

A dynamic array is quite similar to a regular array, but its size is modifiable during program runtime. Dynamic Array elements occupy a contiguous block of memory. Once an array has been created, its size cannot be changed. However, a dynamic array is different. A dynamic array can expand its size even after it has been filled. During the creation of an array, it is allocated a predetermined amount of memory. This is not the case with a dynamic array as it grows its memory size by a certain factor when there is a need.

```
DataType * ptr_array { new DataType[size]{} } ;
```

- Regular arrays have a fixed size. You cannot modify their size once declared.
- With these types of arrays, the memory size is determined during compile time.
- Dynamic arrays are different. Their sizes can be changed during runtime.
- In dynamic arrays, the size is determined during runtime.
- Dynamic arrays in C++ are declared using the `new` keyword.
- We use `square brackets` to specify the number of items to be stored in the dynamic array.
- Once done with the array, we can free up the memory using the delete operator.
- Use the `delete` operator with `[]` to free the memory of all array elements.
- A `delete` without `[]` frees the memory of only a single element.
- There is no built-in mechanism to resize C++ arrays.
- To initialize an array using a list initializer, we don't use the `=` operator.

Dangling Pointer

Dangling pointer is a pointer pointing to a memory location that has been freed (or deleted). There are different ways where Pointer acts as dangling pointer:

1. Uninitialized pointer

```
int * p_number; // Dangling uninitialized pointer

std::cout << std::endl;
std::cout << "Case 1 : Uninitialized pointer : ." << std::endl;
std::cout << "p_number : " << p_number << std::endl;
std::cout << "*p_number : " << *p_number << std::endl; //CRASH!
```

2. Deleted pointer

```
std::cout << std::endl;
std::cout << "Case 2 : Deleted pointer" << std::endl;
int * p_number1 {new int{67}};

std::cout << "*p_number1 (before delete) : " << *p_number1 << std::endl;

delete p_number1;
//undefined behaviour : Crash/ garbage or whatever
std::cout << "*p_number1(after delete) : " << *p_number1 << std::endl;
```

3. Multiple pointers pointing to same memory

```
std::cout << std::endl;
std::cout << "Case 3 : Multiple pointers pointing to same address : " << std::endl;

int *p_number3 {new int{83}};
int *p_number4 {p_number3};

std::cout << "p_number3 - " << p_number3 << " - " << *p_number3 << std::endl;
std::cout << "p_number4 - " << p_number4 << " - " << *p_number4 << std::endl;

//Deleting p_number3
delete p_number3;

//p_number4 points to deleted memory. Dereferncing it will lead to
//undefined behaviour : Crash/ garbage or whatever
std::cout<< "p_number4(after deleting p_number3) - " << p_number4 << " - " << *p_number4 ;
```

Reference

When a variable is declared as a reference, it becomes an alternative name for an existing variable. A variable can be declared as a reference by putting '&' in the declaration.

Declaration and Initialization

```
DataType & ref_name = ref_variable ;
```

Example

```
#include <iostream>
int main()
{
 int x = 10;
 // ref is a reference to x.
 int& ref = x;
 // Value of x is now changed to 20
 ref = 20;
 std::cout << "x = " << x << '\n';
 // Value of x is now changed to 30
 x = 30;
 std::cout << "ref = " << ref << '\n';
 return 0;
}

/* Output
* x = 20
* ref = 30
*/
```

Applications

1. **Modify the passed parameters in a function**: If a function receives a reference to a variable, it can modify the value of the variable. For example, the following program variables are swapped using references.
2. **Avoiding a copy of large structures**: Imagine a function that has to receive a large object. If we pass it without reference, a new copy of it is created which causes wastage of CPU time and memory. We can use references to avoid this.
3. **In For Each Loops to modify all objects** : We can use references in for each loops to modify all elements.
4. **For Each Loop to avoid the copy of objects**: We can use references in each loop to avoid a copy of individual objects when objects are large.

Reference vs. Pointer

ref vs ptr

const Reference

A `const` reference that isn't `const`. We can change the variable only from the original one, not from the reference variable .

```
const DataType & ref_name = ref_variable ;
```

```
#include <iostream>
int main()
{
  int age {27};

  const int& ref_age{age};
```

```
    std::cout << "age : " << age << std::endl;
    std::cout << "ref_age : " << ref_age << std::endl;

    //Mofify through reference -> compiler error
    //ref_age++;

    //Mofify through original variable
    age++;

    std::cout << "age : " << age << std::endl;
    std::cout << "ref_age : " << ref_age << std::endl;
}

/* Output
* age : 27
* ref_age : 27
* age : 28
* ref_age : 28
*/
```

String

Previously we said. String is a collection of characters. There are two types of strings commonly used in C++ programming language:

- Strings that are objects of string class (The Standard C++ Library string class)
- C-strings (C-style Strings)

And we spoke about C-strings in Arrays chapter, now we'll just take the standard C++ library string class. C++ has in its definition a way to represent a sequence of characters as an object of the class. This class is called `std::string`. String class stores the characters as a sequence of bytes with the functionality of allowing access to the single-byte character.

> must include `<string>` for string class.

Declaration and Initialization

```
// Declaration
std::string str ;

// Declaration and Initialization
std::string str {"str Data"} ;

// Initialization
std::string str ;
str = "str Data" ;
```

C-style vs. C++ Class String

1. A character array is simply an array of characters that can be terminated by a null character. A string is a class that defines objects that be represented as a stream of characters.
2. The size of the character array has to be allocated statically, more memory cannot be allocated at run time if required. Unused allocated memory is wasted in the case of the character array. In the case of strings, memory is allocated dynamically. More memory can be allocated at run time on demand. As no memory is preallocated, no memory is wasted.
3. There is a threat of array decay in the case of the character array. As strings are represented as objects, no array decay occurs.
4. Implementation of character array is faster than `std::string`. Strings are slower when compared to implementation than character array.
5. Character arrays do not offer many inbuilt functions to manipulate strings. String class defines a number of functionalities that allow manifold operations on strings.

Operation on string

1. Iterator Functions

Iterators

`begin` `cbegin` (C++11)	returns an iterator to the beginning (public member function)
`end` `cend` (C++11)	returns an iterator to the end (public member function)
`rbegin` `crbegin` (C++11)	returns a reverse iterator to the beginning (public member function)
`rend` `crend` (C++11)	returns a reverse iterator to the end (public member function)

2. Capacity Functions

Capacity

`empty`	checks whether the string is empty (public member function)
`size` `length`	returns the number of characters (public member function)
`max_size`	returns the maximum number of characters (public member function)
`reserve`	reserves storage (public member function)
`capacity`	returns the number of characters that can be held in currently allocated storage (public member function)
`shrink_to_fit` (C++11)	reduces memory usage by freeing unused memory (public member function)

> by default capacity of string is `15`.

3. Access Functions

Element access

`at`	accesses the specified character with bounds checking (public member function)
`operator[]`	accesses the specified character (public member function)
`front` (C++11)	accesses the first character (public member function)
`back` (C++11)	accesses the last character (public member function)
`data`	returns a pointer to the first character of a string (public member function)
`c_str`	returns a non-modifiable standard C character array version of the string (public member function)
`operator basic_string_view` (C++17)	returns a non-modifiable string_view into the entire string (public member function)

4. Manipulating Functions

Operations

`clear`	clears the contents (public member function)
`insert`	inserts characters (public member function)
`erase`	removes characters (public member function)
`push_back`	appends a character to the end (public member function)
`pop_back` (C++11)	removes the last character (public member function)
`append`	appends characters to the end (public member function)
`operator+=`	appends characters to the end (public member function)
`compare`	compares two strings (public member function)
`starts_with` (C++20)	checks if the string starts with the given prefix (public member function)
`ends_with` (C++20)	checks if the string ends with the given suffix (public member function)
`contains` (C++23)	checks if the string contains the given substring or character (public member function)
`replace`	replaces specified portion of a string (public member function)
`substr`	returns a substring (public member function)
`copy`	copies characters (public member function)
`resize`	changes the number of characters stored (public member function)
`resize_and_overwrite` (C++23)	changes the number of characters stored and possibly overwrites indeterminate contents via user-provided operation (public member function)
`swap`	swaps the contents (public member function)

5. Search Functions

Search

`find`	find characters in the string (public member function)
`rfind`	find the last occurrence of a substring (public member function)
`find_first_of`	find first occurrence of characters (public member function)
`find_first_not_of`	find first absence of characters (public member function)
`find_last_of`	find last occurrence of characters (public member function)
`find_last_not_of`	find last absence of characters (public member function)

6. Constant

Constants

`npos` [static]	special value. The exact meaning depends on the context (public static member constant)

7. Numeric Function

Numeric conversions

`stoi` (C++11) `stol` (C++11) `stoll` (C++11)	converts a string to a signed integer (function)
`stoul` (C++11) `stoull` (C++11)	converts a string to an unsigned integer (function)
`stof` (C++11) `stod` (C++11) `stold` (C++11)	converts a string to a floating point value (function)
`to_string` (C++11)	converts an integral or floating point value to `string` (function)
`to_wstring` (C++11)	converts an integral or floating point value to `wstring` (function)

Escape Sequence

Escape sequence	Description	Representation
\'	single quote	byte `0x27` in ASCII encoding
\"	double quote	byte `0x22` in ASCII encoding
\?	question mark	byte `0x3f` in ASCII encoding
\\	backslash	byte `0x5c` in ASCII encoding
\a	audible bell	byte `0x07` in ASCII encoding
\b	backspace	byte `0x08` in ASCII encoding
\f	form feed - new page	byte `0x0c` in ASCII encoding
\n	line feed - new line	byte `0x0a` in ASCII encoding
\r	carriage return	byte `0x0d` in ASCII encoding
\t	horizontal tab	byte `0x09` in ASCII encoding
\v	vertical tab	byte `0x0b` in ASCII encoding

Raw String Literals

A raw string literal is a string in which the escape characters like **\n**, **\t** or **\"** of C++ are not processed. Hence, this was introduced in C++11, a raw string literal which starts with `R(` and ends in `)`. These raw string literals allow a series of characters by writing precisely its contents like raw character sequence.

```
std::string str {R "delimiter( raw_characters )delimiter"} ;

// Ordinary String Literal
std::string str {"\\\\n"} ;

// Raw String Literal
std::string str {R"(\\n)"} ;
```

string_view C++17

The std::string has some demerits, one of the most common situations is constant strings. Below is the program that demonstrates the problem that occurs in dealing with constant strings with std::string:

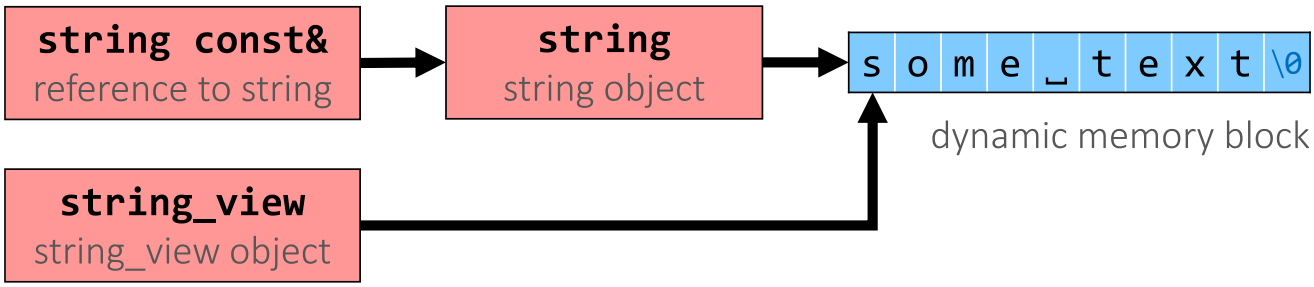
```
// C++ program to demonstrate the
// problem occurred in string
#include <iostream>
#include <string>

// Driver Code
int main()
{
   char str_1[]{ "Hello !!, GeeksforGeeks" };

   string str_2{ str_1 };
   string str_3{ str_2 };

   // Print the string
   std::cout << str_1 << '\n'
      << str_2 << '\n'
      << str_3 << '\n';

   return 0;
}

/* Output
* Hello !!, GeeksforGeeks
* Hello !!, GeeksforGeeks
* Hello !!, GeeksforGeeks
*/

/*
The output is the same as expected. But, in order to view “Hello !!, GeeksforGeeks” twice the std::string performs overheads on the memory twice. But here the task was to read the string (“Hello !!, GeeksforGeeks”), and no write operation is required on it. So just to display a string why assign memory multiple times. In order to deal with the strings more efficiently, C++17 proposed std::string_view() which provides the view of pre-defined char str[] without creating a new object to the memory.
*/
```

Benefits of `std::string_view`:

- **Light and Cheaper:** The `std::string_view` is a very light, cheaper and is mainly used to provide the view of the string. Whenever the `std::string_view` is created there is no need to copy the string in the manner as done in the above example that was inefficient and was causing overhead on the memory. It makes the copying process of the string quite efficient and never creates any copy of the string when the modification is being made in the viewed string the made changes have appeared in the `std::string_view`.
- **Better Performance:** The `std::string_view` is better than the const `std::string&` because it removes the constraint of having a `std::string` object at the very beginning of the string as `std::string_view` is composed of two elements first one is `const char*` that points to the starting position of the array and the second is size.
- **Supports crucial function:** The `std::string_view` supports mostly all crucial function that is applied over the `std::string` like `substr`, `compare`, `find`, overloaded comparison operators `(e.g., ==, <, >, !=)`. So in most of the cases, it removes the constraint of having a `std::string` object declaration when our preference is read-only.

> `std::string_view` C++17 library has proposed a standard type of string (`std::string_view`) which is different from the usual `std::string`.

- The `std::string_view` provides a lightweight object that offers read-only access to a string or a part of a string using an interface similar to the interface of `std::_string` and merely refers to the contiguous char sequence. Unlike `std::string`, which keeps its own copy of the string, It also provides a view of a string that is defined elsewhere in the source code.
- it is composed of two members: a `const char*` that points to the start of the char array, and the _size. It is a non-owning reference to a string in itself.



Function

A function is a block A reusable piece of code that can take a number of optional inputs and produce some desirable output that performs a specific task.

There are two types of function:

1. Standard Library Functions: Predefined in C++
2. User-defined Function: Created by users

Prototype and Definition

C++ allows the programmer to define their own function. A user-defined function groups code to perform a specific task and that group of code is given a name (identifier). When the function is invoked from any part of the program, it all executes the codes defined in the body of the function.

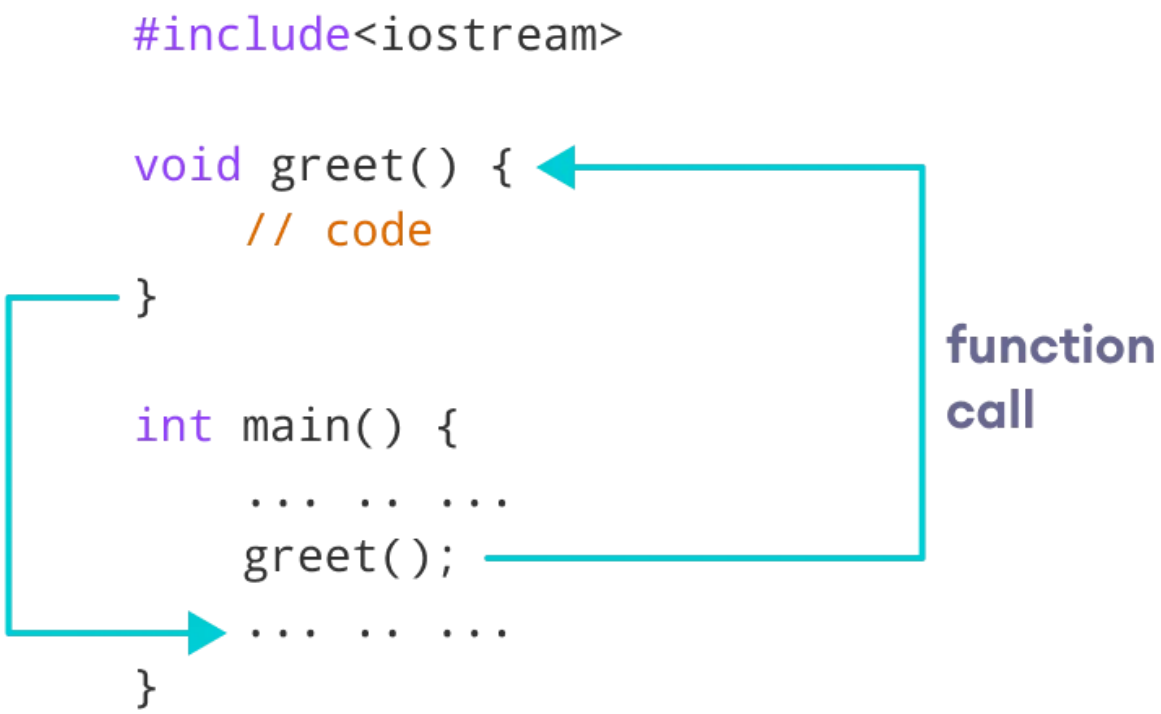
The prototype needs to come BEFORE the function call in your file. Otherwise the compilation will fail. The full function definition coming in front of main() also doubles as a prototype(declaration).

```
// prototype with defination
returnType functionName (parameter1, parameter2,...)
{
   // function body

   return returnData
}

// prototype separate from the defination
returnType functionName (parameter1, parameter2,...); // prototype

int main ()
{
   // main body
}

returnType functionName (parameter1, parameter2,...) // definaion
{
   // function body

   return returnData
}
```

Call Functions

To use the function, we need to call it.

```
returnType functionName (parameter1, parameter2,...)
{
   // function body

   return returnData
}

int main()
{
   // calling the function
   returnVariable = functionName (parameter1, parameter2,...) ;    // if has return value
   functionName (parameter1, parameter2,...) ;                     // if has no return value
}
```

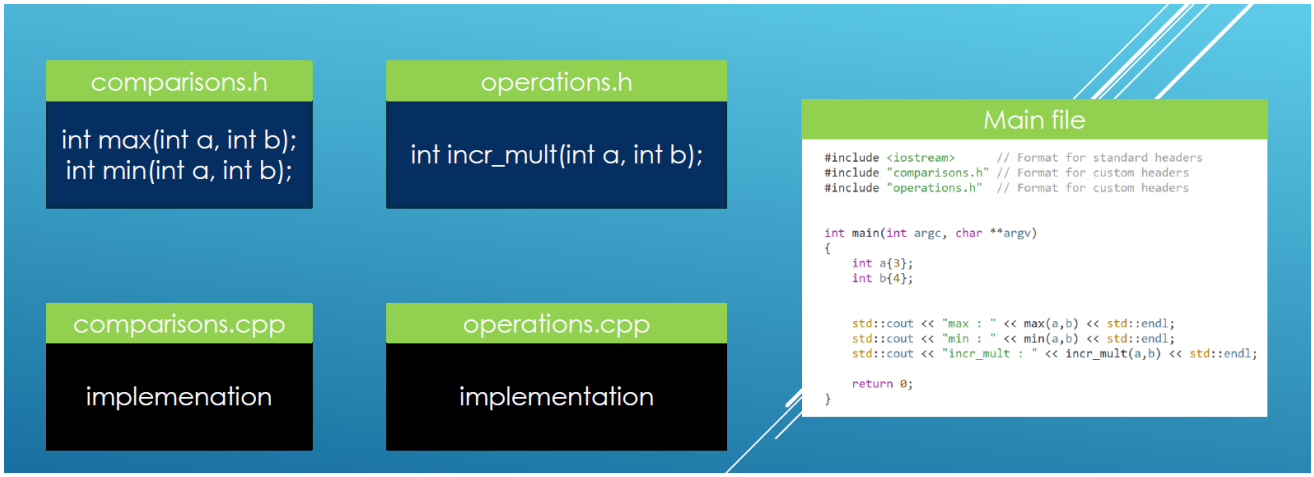


Multiple Source Files

Small programs are typically written in a single `.cpp` file. However, as programs get larger, it's necessary to split the source into several files in order to make the project manageable. Because C++ needs to know the declarations of functions before they are called, function prototypes (declarations) are typically written at the top of a single-file, multiple-function program. This ensures that the call and the definition are both consistent. the compiler compiles each file individually. It does not know about the contents of other code files, or remember anything it has seen from previously compiled code files. So even though the compiler may have seen the definition of function *add* previously (if it compiled *add.cpp* first), it doesn't remember.

> One Definition Rule : The same function implementation can't show up in the global namespace more than once.

The linker searches for definitions in all translation units (.cpp) files in the project. Doesn't have to live in a . cpp file with the same name as the header.



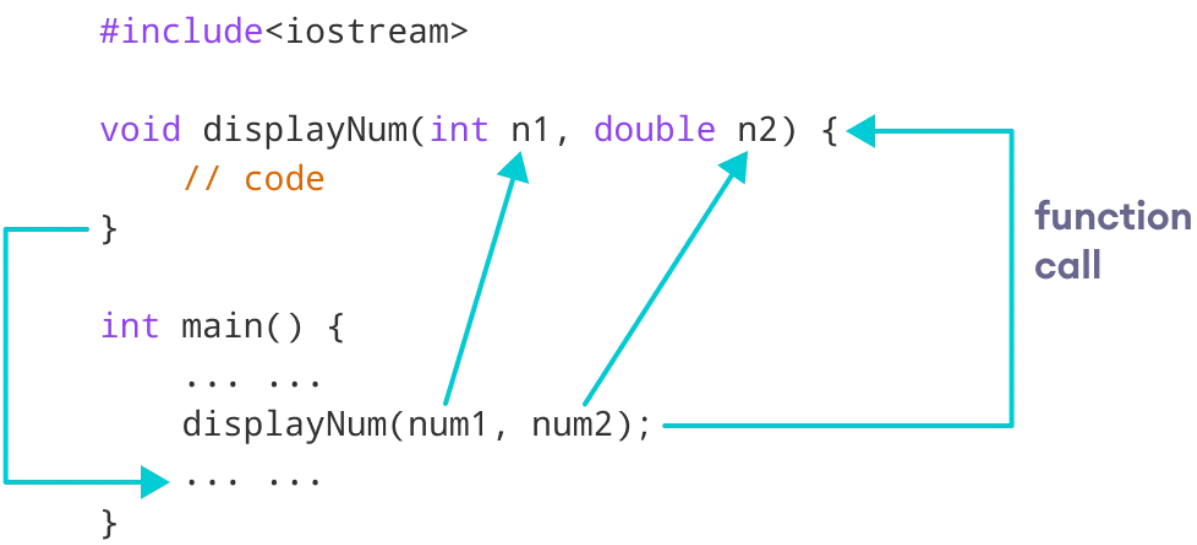
Function Parameters

a function can be declared with parameters (arguments). A parameter is a value that is passed when declaring a function. called parameters. A legal C++ function can have 0 or more parameters.

Output parameters should be passed in such a way that you can modify the arguments from inside the function. Options are passing by reference or by pointer. References are preferred in C++.

Input parameters shouldn't be modifiable from inside a function. The function really need to get input (read) from the arguments. You enforce modification restrictions with the `const`. Options are passing by `const` reference, passing by `pointer` to `const`, or even passing by `const pointer` to `const`

> The `void` keyword, used in the previous examples, indicates that the function should not return a value. If you want the function to return a value, you can use a data type (such as `int`, `string`, etc.) instead of `void`, and use the `return` keyword inside the function.



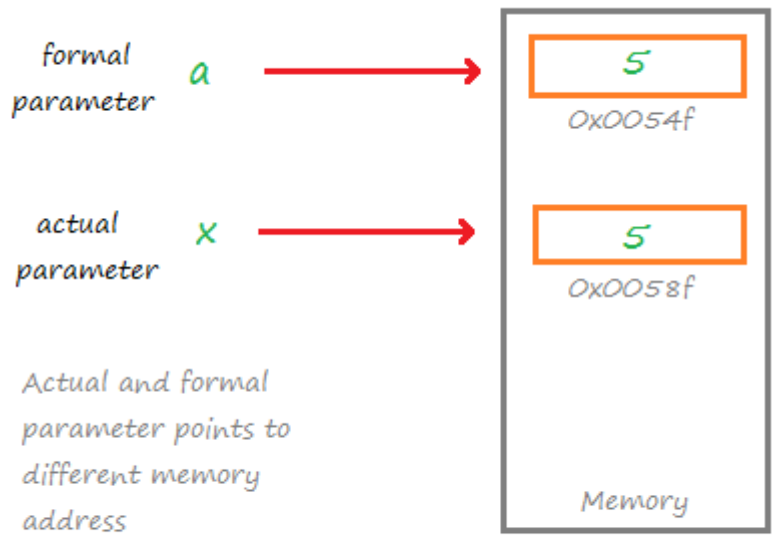
Default Parameter

You can also use a default parameter value, by using the equals sign (`=`). If we call the function without an argument, it uses the default value.

```cpp
#include <iostream>

void myFunction(string country = "Egypt")
{
  std::cout << country << "\n";
}

int main()
{
  myFunction("Sweden");
  myFunction("India");
  myFunction();
  myFunction("USA");
  return 0;
}

/* Output
* Sweden
* India
* Egypt
* USA
*/
```

Call by Value

When a function is called in the call by value, the **value of the actual parameters is copied into formal parameters.** Both the actual and formal parameters have their own copies of values, therefore any change in one of the types of parameters will not be reflected by the other. This is because both actual and formal parameters point to different locations in memory (i.e. they both have different memory addresses).



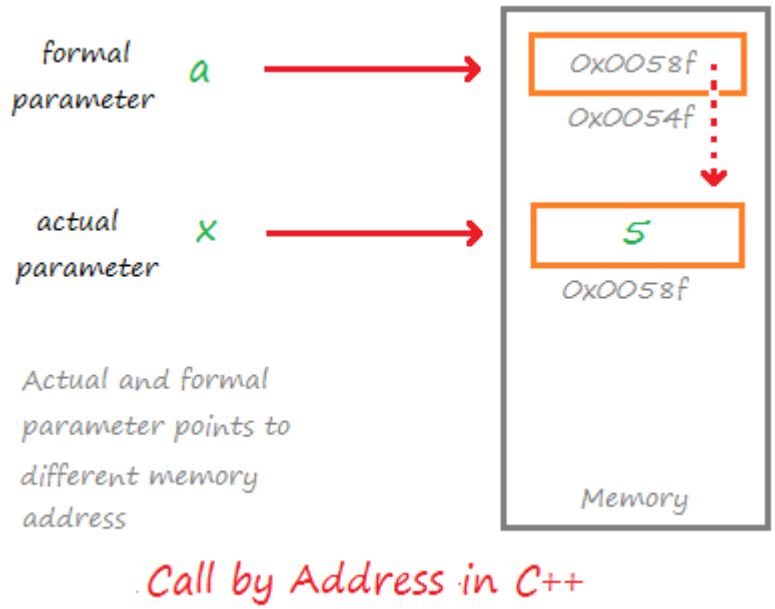
Call by Value in C++

> - If we call by `const value`, we can't change the copy it's cause a compiler error.

> - If the passed parameters doesn't exact data type, the compiler will implement **implicit conversion**.

Call by Address

In the call by address method, **both actual and formal parameters indirectly share the same variable.** In this type of call mechanism, pointer variables are used as formal parameters. The formal pointer variable holds the address of the actual parameter, hence the changes done by the formal parameter is also reflected in the actual parameter, both parameters point to different locations in memory, but since the formal parameter stores the address of the actual parameter, they share the same value.



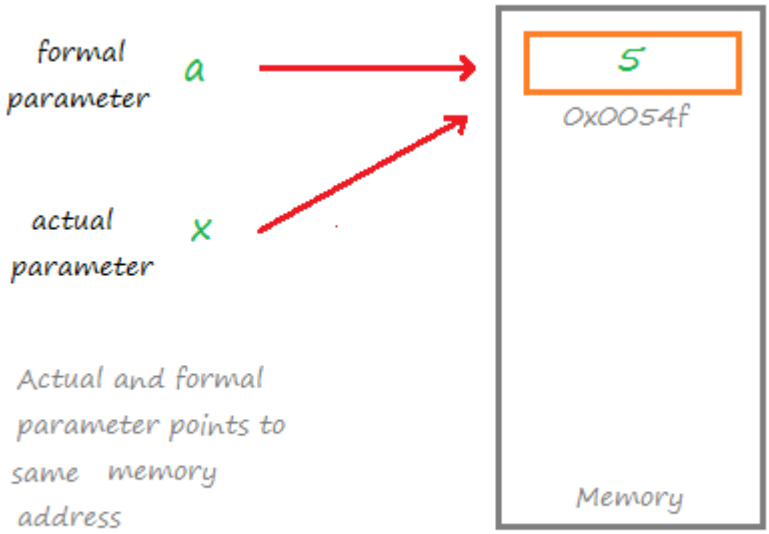
Call by Address in C++

```
// prototype with defination
returnType functionName (*parameter)
{
   // function body

   return returnData
}
```

> - If we call by `Address to const value`, we can't change the value it's cause a compiler error.
> - If we call by `const Address to const value`, we can't change the value and the pointer point it's cause a compiler error.
> - If the passed parameters doesn't exact data type, we'll got a complier error.

Call by Reference

In the call by reference, **both formal and actual parameters share the same value.** Both the actual and formal parameter points to the same address in the memory. That means any change on one type of parameter will also be reflected by other. Calls by reference are preferred in cases where we do not want to make copies of objects or variables, but rather we want all operations to be performed on the same copy.



Call by Reference in C++

```
// prototype with defination
returnType functionName (&parameter)
{
   // function body

   return returnData
}
/
```

- If we call by `Reference to const value`, we can't change the value it's cause a compiler error.
- If the passed parameters doesn't exact data type, we'll got a complier error.

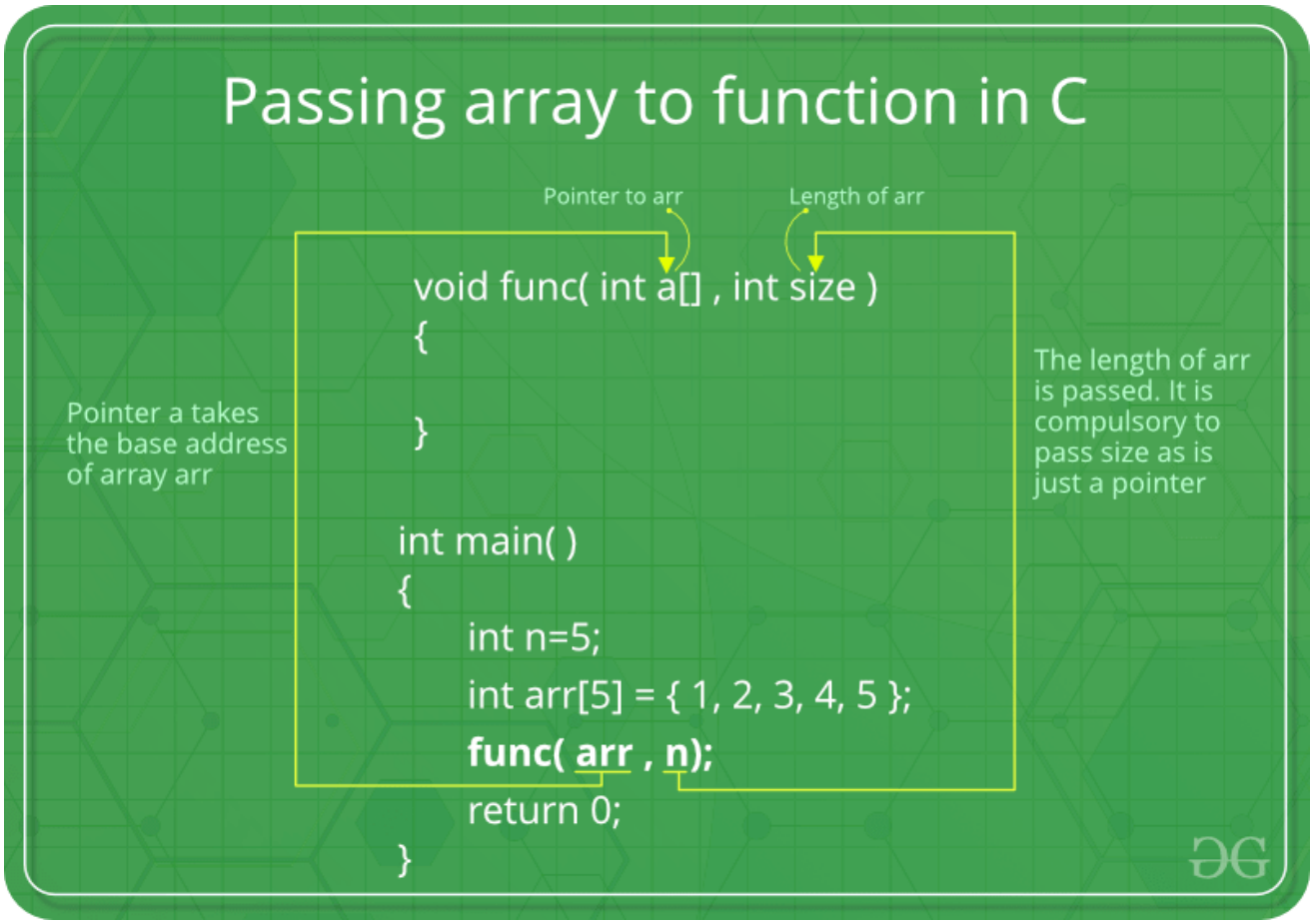
Passing Array

We can pass arrays as an argument to a function. And, also we can return arrays from a function.

```
returnType functionName(dataType arrayName[arraySize, dataType arraySize)
{
   // function body
}
```

> - It is not mandatory to specify the number of rows in the array. However, the number of columns should always be specified. This is why we have used `int n[][2]`, also we can pass arrays with more than 2 dimensions as a function argument.

- We can also return an array from the function. However, the actual array is not returned. Instead the address of the first element of the array is returned with the help of pointers.
- If we pass array by reference we must pass the true array size to avoid the compiler error.
- when an array name is passed as a function parameter, the parameter name inside the body of the function is no longer a "real array" name, and it has lost all size information it had. It becomes just a pointer to the first element of the array. `sizeof(array)` isn't going to give us the size of the entire array, it's just going to be the size of a pointer on your system.
-



Passing String

If You...	Use Parameter Type
always need a copy of the input string inside the function	`std::string` "pass by value"
want **read-only access** ▪ don't (always) need a copy ▪ are using **C++17 / 20**	`#include <string_view>` `std::string_view`
want **read-only access** ▪ don't (always) need a copy ▪ are stuck with **C++98 / 11 / 14**	`std::string const&` "pass by const reference"
want the function to **modify the input string** in-place (you should try to avoid such "output parameters")	`std::string &` "pass by (non-const) reference"

1. Copy Value

Pass By Value!

👍 By value ⇒ One copy per call	👎 By const reference ⇒ Sometimes two copies

`Box by_value (std::string id) {`
`  // sanitize id (already a copy)`
`  replace(begin(id), end(id),`
`  ' ', '-');`
 	
`  return Box{std::move(id)};`
`}`
	`Box by_cref (std::string const& id) {`
	`  // make copy for replacing`
	`  std::string idCopy {id};`
	`  replace(begin(idCopy), end(idCopy),`
	`  ' ', '-');`
	
	`  return Box{std::move(idCopy)};`
	`}`

Call with string literal

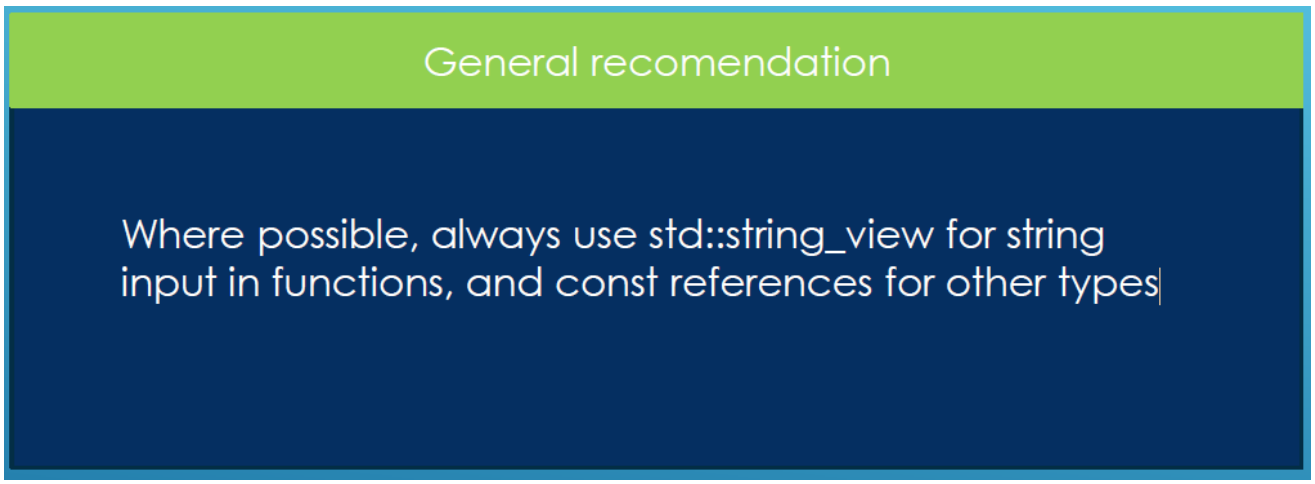
`Box b = by_value("A 2");` 👍 implicitly creates one string object: function parameter `string id{"A 2"}`	`Box b = by_cref("A 2");` 👎 creates **two** string objects: 1. implicit `string{"A 2"}` that is bound to function parameter `string const& id` 2. explicit local copy `idCopy`

2. Read-Only

Use std::string_view C++17

- lightweight (= cheap to copy, can be passed by value)
- non-owning (= not responsible for allocating or deleting memory)
- read-only view
- of a character range or string(-like) object (std::string / "literal" / ...)

```
#include <string>
#include <string_view>

class Pattern { …
public: …
  auto match (std::string_view s) const { … }
};

Pattern p {"ab.*"};

auto r1 = p.match("abx");  // no copy

std::string txt = "abcde";
auto r2 = p.match(txt);    // no copy
```



Function return

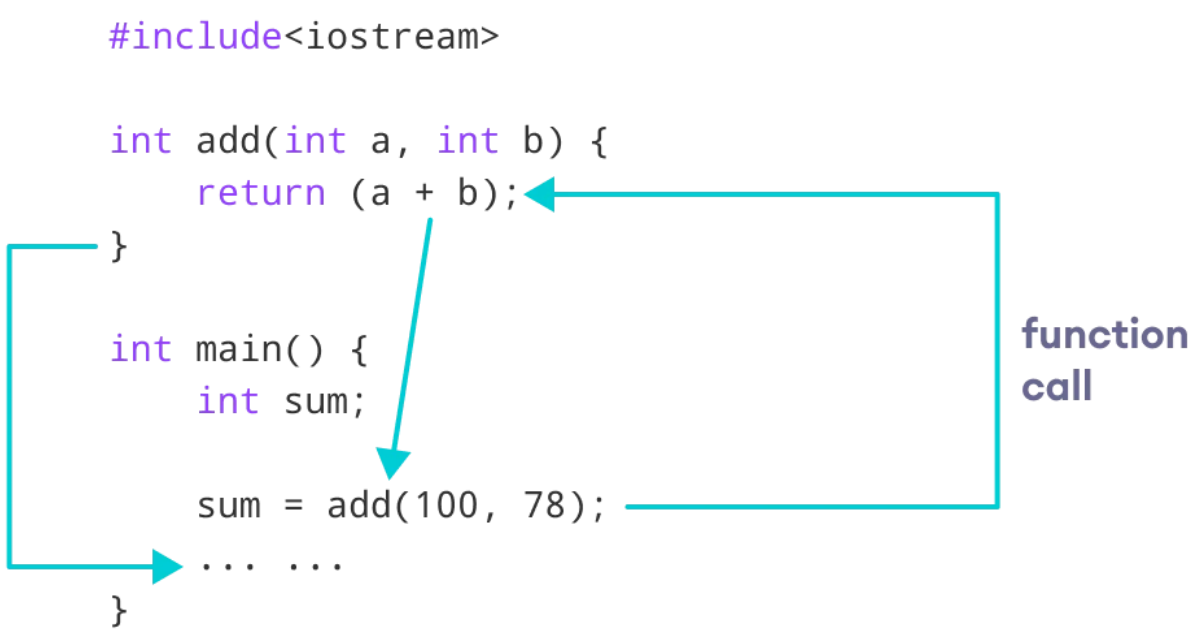
The `void` keyword, used in the previous examples, indicates that the function should not return a value. If you want the function to return a value, you can use a data type (such as `int`, `string`, etc.) instead of `void`, and use the `return` keyword inside the function.

return by value

In modern compilers, return by value is commonly optimized out by the compiler when possible and the function is modified behind your back to return by reference, avoiding unnecessary copies!

```
// program to add two numbers using a function
#include <iostream>

// declaring a function
int add(int a, int b)
```

```
{
   return (a + b);
}

int main()
{
   int sum;
   // calling the function and storing
   // the returned value in sum
   sum = add(100, 78);
   std::cout << "100 + 78 = " << sum << std::endl;
   return 0;
}
```



return by refinance

In C++ Programming, not only can you pass values by reference to a function but you can also return a value by reference.

```
#include <iostream>
// global variable
int num;
// function declaration
int& test();

int main()
{
 // assign 5 to num variable
 // equivalent to num = 5;
 test() = 5;
 cout << num;
```

```
  return 0;
}
// function definition
// returns the address of num variable
int& test()
{
  return num;
}
```

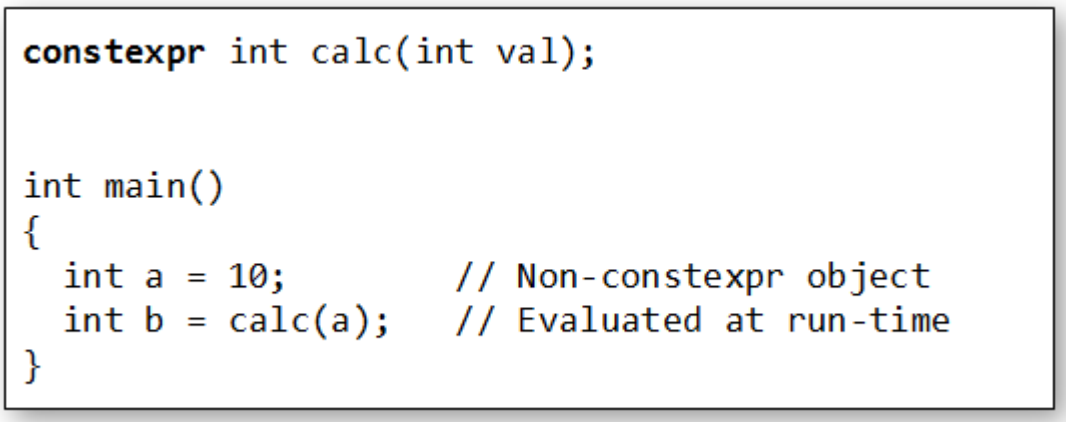
return by pointer

C++ allows you to return a pointer from a function. To do so, you would have to declare a function returning a pointer.

```
int * myFunction()
{
  .
  .
  .
}
```

constexpr Functions

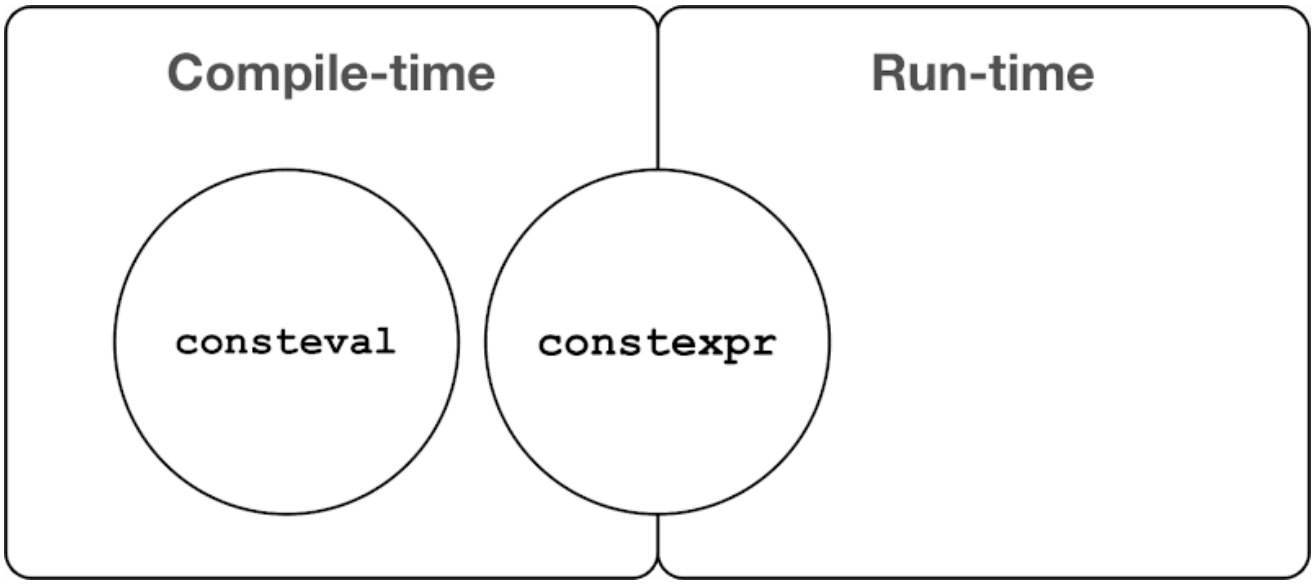
- The `constexpr` specifier declares that it is possible to evaluate the value of a function or variable at compile time.
- Such variables and functions can then be used where only compile-time constant expressions are allowed.
- These functions are used to improve the performance of the program by doing computations at compile time instead of run time.
- These functions can really be helpful, where executing a program multiple times as the constant expressions will only be evaluated once during the compile time.
- The `constexpr` specifies that the value of a variable or function can appear in constant expressions.
- the `constexpr` function it doesn't have to be at compile time.



```
constexpr int calc(int val);


int main()
{
  int a = 10;         // Non-constexpr object
  int b = calc(a);    // Evaluated at run-time
}
```

constvel Functions

- `consteva` function, every call to the function must directly or indirectly produce a compile-time constant expression.
- The `consteval` function is the same as `constexpr` function except that if the call to a `consteval` function doesn't evaluate to a compile-time constant expression, then the program gives an error while it is not so in the case of a `constexpr` function.
- Function can appear in constant expressions, it doesn't say that the function has to be, while a `consteval` specifies that a function is an immediate function, that is, every call to the function must produce a compile-time constant.



Arguments to main Function

We can also give command-line arguments in C and C++. Command-line arguments are given after the name of the program in command-line shell of Operating Systems. To pass command line arguments, we typically define main() with two arguments : first argument is the number of command line arguments and second is list of command-line arguments.

```
int main(int argc, char *argv[])
{
   /* ... */
}

/********* or *********/
int main(int argc, char **argv)
{
   /* ... */
}
```

- `argc` (ARGument Count) is int and stores number of command-line arguments passed by the user including the name of the program. So if we pass a value to a program, value of argc would be 2 (one for argument and one for program name)
- The value of argc should be non negative.
- `argv` (ARGument Vector) is array of character pointers listing all the arguments.
- If argc is greater than zero,the array elements from argv[0] to argv[argc-1] will contain pointers to strings.

- Argv[0] is the name of the program , After that till argv[argc-1] every element is command -line arguments.

Example

```
// Name of program mainreturn.cpp
#include <iostream>
using namespace std;

int main(int argc, char** argv)
{
   cout << "You have entered " << argc
        << " arguments:" << "\n";

   for (int i = 0; i < argc; ++i)
      cout << argv[i] << "\n";

   return 0;
}

/* input
* $ g++ mainreturn.cpp -o main
* $ ./main Abdelrahman Saad Eldesouky
*/

/* Output
* You have entered 4 arguments:
* ./main
* Abdelrahman
* Saad
* Eldesouky
*/
```

Properties of Command Line Arguments:

1. They are passed to main() function.
2. They are parameters/arguments supplied to the program when it is invoked.
3. They are used to control program from outside instead of hard coding those values inside the code.
4. `argv[argc]` is a NULL pointer.
5. `argv[0]` holds the name of the program.
6. `argv[1]` points to the first command line argument and `argv[n]` points last argument.

std::optional

`std:optional` - a new helper type added in C++17. It's a wrapper for your type and a flag that indicates if the value is initialized or not. `std::optional` was added in C++17 and brings a lot of experience from `boost::optional` that was available for many years. Since C++17 you can just `#include <optional>` and use the type.

```
//Declare and Initialization

std::optional<Data_Type> Variable_Name{Value} ;
```

Usually, you can use an optional wrapper in the following scenarios:

- If you want to represent a nullable type nicely.
  - Rather than using unique values (like -1, nullptr, NO_VALUE or something)
  - For example, a user's middle name is optional. You could assume that an empty string would work here, but knowing if a user entered something or not might be important. With std::optionalstd::string you get more information.
- Return a result of some computation (processing) that fails to produce a value and is not an error.

  For example, finding an element in a dictionary: if there's no element under a key it's not an error, but we need to handle the situation.
- To perform lazy-loading of resources.

  For example, a resource type has no default constructor, and the construction is substantial. So you can define it as std::optional (and you can pass it around the system), and then load only if needed later.
- To pass optional parameters into functions.

```
std::optional<std::string> UI::FindUserNick()
{
   if (nick_available)
   {
      return { mStrNickName };
   }
   return std::nullopt; // same as return { };
}

// use:
std::optional<std::string> UserNick = UI->FindUserNick();
if (UserNick)
{
   Show(*UserNick);
}
```

Function Overloading

Function overloading is a feature of object oriented programming where two or more functions can have the same name but different parameters. When a function name is overloaded with different jobs it is called Function Overloading. In Function Overloading "Function" name should be the same and the arguments should be different.

Parameter Differences:

1. Order
2. Number
3. Types

```
#include <iostream>

void print(int i) {
  std::cout << " Here is int " << i << std::endl;
}
void print(double  f) {
  std::cout << " Here is float " << f << std::endl;
}
void print(char const *c) {
  std::cout << " Here is char* " << c << std::endl;
}

int main() {
  print(10);
  print(10.10);
  print("ten");
  return 0;
}

/* Output
* Here is int 10
* Here is float 10.1
* Here is char* ten
*/
```

How Function Overloading works?

- Exact match

  (Function name and Parameter)

- If a not exact match is found

  1. Char, Unsigned char, and short are promoted to an int.
  2. Float is promoted to double

- If no match found:

  C++ tries to find a match through the standard conversion.

- ELSE ERROR

Lambda Function

A mechanism to set up anonymous functions (without names). Once we have them set up, we can either give them names and call them , or we can even get them to do things directly.

```
// Declaring
[ capture clause ] (parameters) -> return-type
{
   definition of method
};

// Assigning
```

```
auto Call_Name [ capture clause ] (parameters) -> return-type
{
  definition of method
};

// Calling
Call_Name(parameters) ;

// Calling Directly After Definition
[ capture clause ] (parameters) -> return-type
{
  definition of method
}();
```

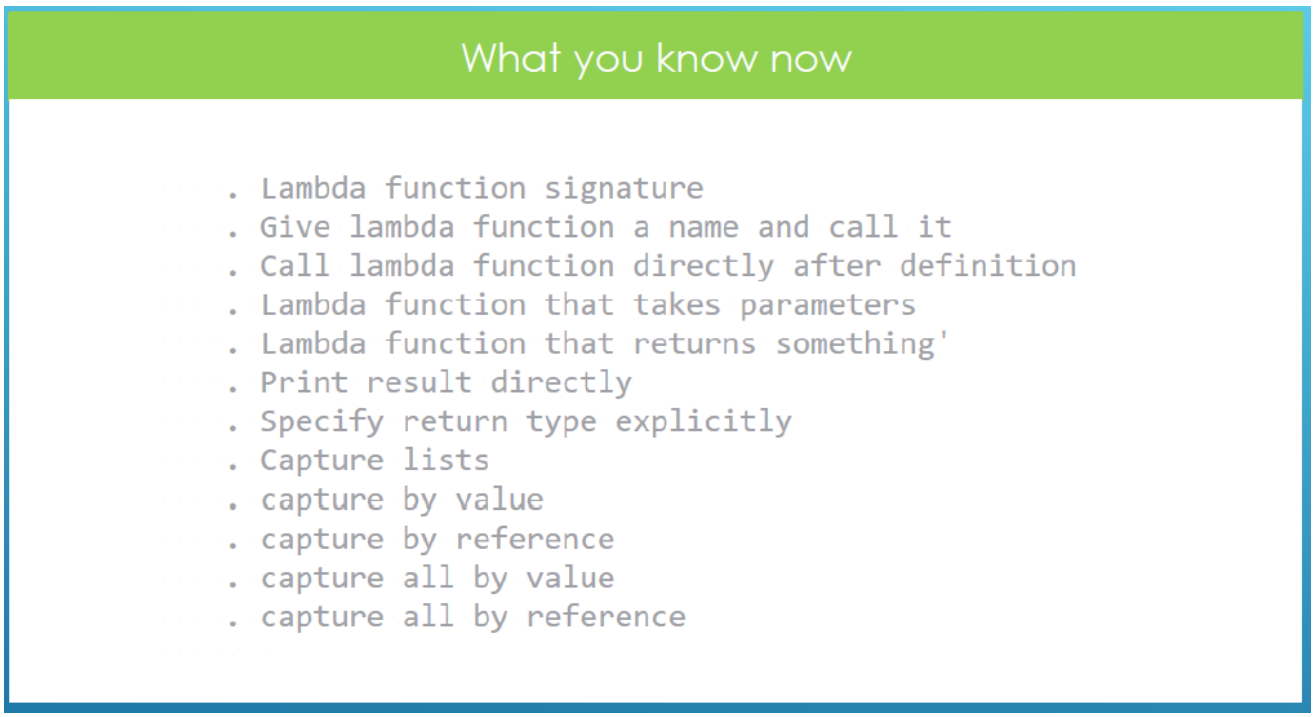
Capture List

A lambda can introduce new variables in its body (in **C++14**), and it can also access, or *capture*, variables from the surrounding scope. A lambda begins with the capture clause. It specifies which variables are captured, and whether the capture is by value or by reference. Variables that have the ampersand (`&`) prefix are accessed by reference and variables that don't have it are accessed by value. An empty capture clause, `[ ]`, indicates that the body of the lambda expression accesses no variables in the enclosing scope. You can use a capture-default mode to indicate how to capture any outside variables referenced in the lambda body: `[&]` means all variables that you refer to are captured by reference, and `[=]` means they're captured by value. You can use a default capture mode, and then specify the opposite mode explicitly for specific variables. For example, if a lambda body accesses the external variable `total` by reference and the external variable `factor` by value, then the following capture clauses are equivalent:

```
[&total, factor]
[factor, &total]
[&, factor]
[=, &total]
```

Only variables that are mentioned in the lambda body are captured when a capture-default is used. If a capture clause includes a capture-default `&`, then no identifier in a capture of that capture clause can have the form `&identifier`. Likewise, if the capture clause includes a capture-default `=`, then no capture of that capture clause can have the form `=identifier`. An identifier or **this** can't appear more than once in a capture clause.

> Values captured by value can't be modified in the body of the lambda function by default. There are ways around this and we'll learn about them later.



Static Variables in Function

When a variable is declared as static, space for **it gets allocated for the lifetime of the program**. Even if the function is called multiple times, space for the static variable is allocated only once and the value of variable in the previous call gets carried through the next function call.

Global variable vs Static variable

- Both global and static variables have static storage duration. They live throughout the entire lifetime of the program.
- Static variables are scoped to the function in which they are declared and used. If you try to access them outside that function, you'll get a compiler error.
- Global variables are scoped to the global scope of the file where they are declared. They are accessible and usable through out the entire file.

Inline Function

When the program executes the function call instruction the CPU stores the memory address of the instruction following the function call, copies the arguments of the function on the stack and finally transfers control to the specified function. The CPU then executes the function code, stores the function return value in a predefined memory location/register and returns control to the calling function. This can become overhead if the execution time of function is less than the switching time from the caller function to called function (callee). For functions that are large and/or perform complex tasks, the overhead of the function call is usually insignificant compared to the amount of time the function takes to run. However, for small, commonly-used functions, the time needed to make the function call is often a lot more than the time needed to actually execute the function's code. This overhead occurs for small functions because execution time of small function is less than the switching time.

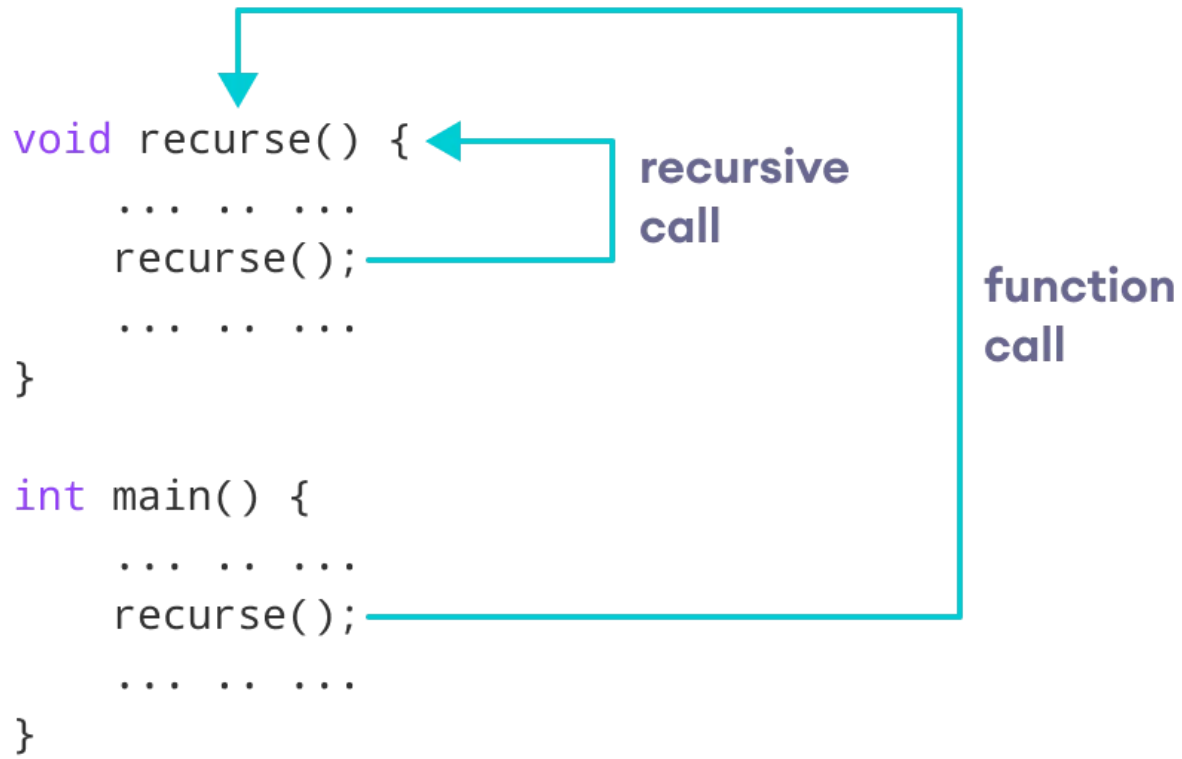
C++ provides an inline functions to reduce the function call overhead. Inline function is a function that is expanded in line when it is called. When the inline function is called whole code of the inline function gets inserted or substituted at the point of inline function call. This substitution is performed by the C++ compiler at compile time. Inline function may increase efficiency if it is small.

```
inline returnType functionName (parameter1, parameter2,...)
{
   // function body

   return returnData
}
```

- Inline functions can increase the size of your application binary.
- It is recommended to use them for short, frequently used functions.
- The programmer (You), should weigh in the benefits against the downsides of inlining your functions.
- Usually only functions of a few lines of code and simple logic, like our max function should be inlined.
- Marking your function as inline is just a suggestion to the compiler. The compiler might agree and inline your function or just ignore you.

Recursion Function

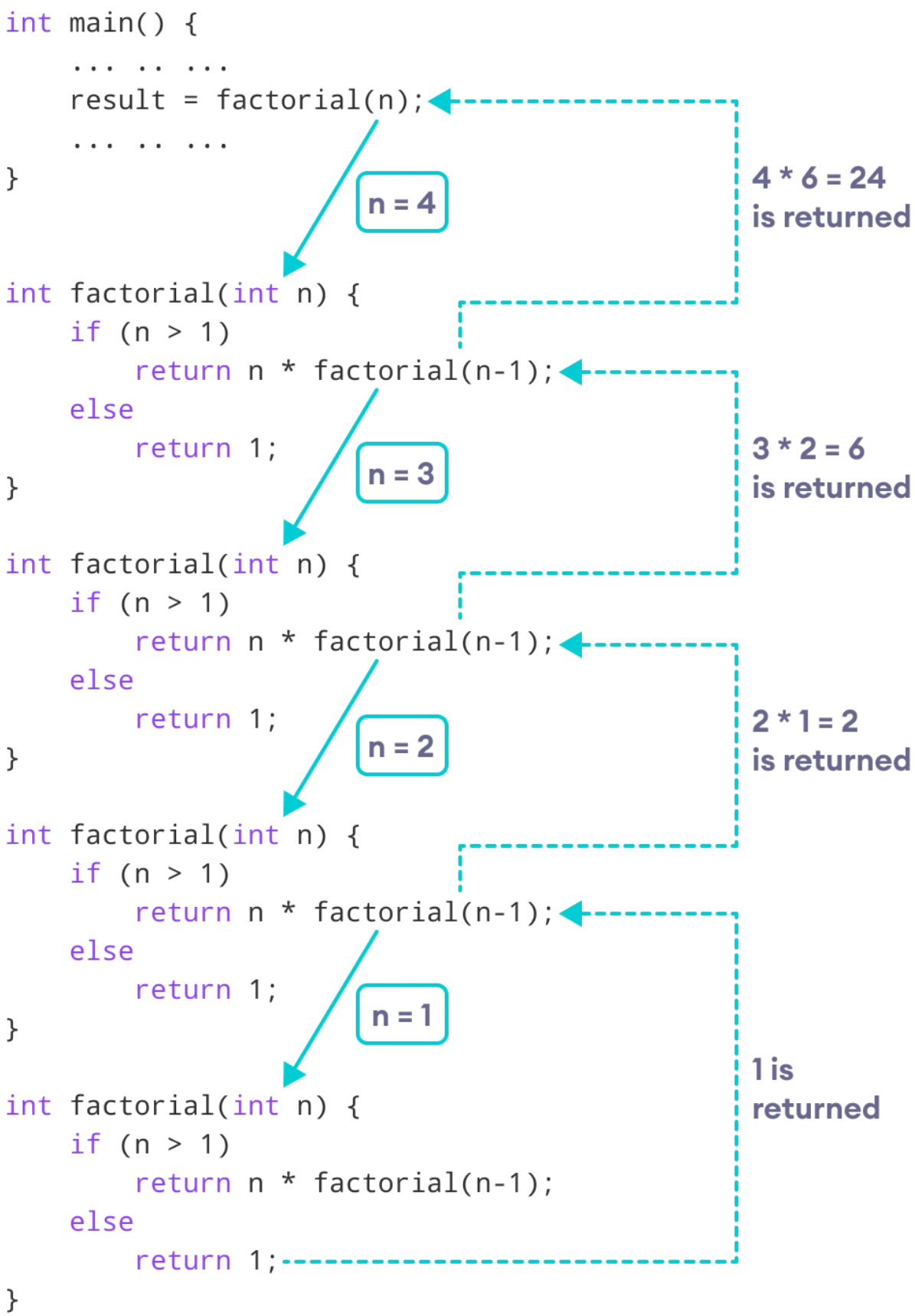
A function that calls itself is known as a recursive function. And, this technique is known as recursion. The recursion continues until some condition is met. To prevent infinite recursion, `if...else statement` (or similar approach) can be used where one branch makes the recursive call and the other doesn't.



Example: Factorial of a Number Using Recursion

```
// Factorial of n = 1*2*3*...*n

#include <iostream>
```

```
int factorial(int);

int main()
{
   int n, result;

   std::cout << "Enter a non-negative number: ";
   std::cin >> n;

   result = factorial(n);
   std::cout << "Factorial of " << n << " = " << result;
   return 0;
}

int factorial(int n)
{
   if (n > 1)
   {
      return n * factorial(n - 1);
   }
   else
   {
      return 1;
   }
}
```



Advantages of Recursion:

1. It makes our code shorter and cleaner.
2. Recursion is required in problems concerning data structures and advanced algorithms, such as Graph and Tree Traversal.

Disadvantages of Recursion:

1. It takes a lot of stack space compared to an iterative program.
2. It uses more processor time.
3. It can be more difficult to debug compared to an equivalent iterative program.

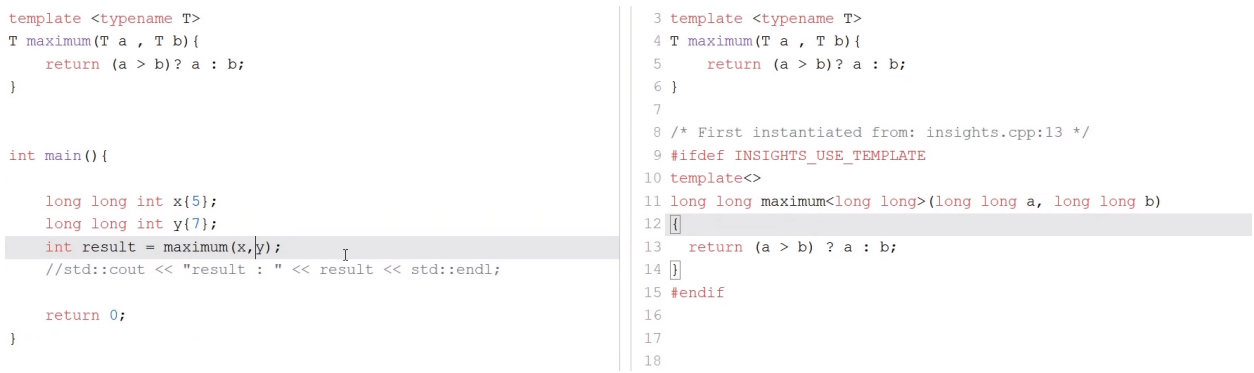
Function Templates

Templates are powerful features of C++ which allows us to write generic programs. We can create a single function to work with different data types by using a template, Instead of function overload.

```
// Definition by value
template <typename T>
T Function_Name(T parameter1, T parameter2, ...)
{
   // code
}

// Definition by reference
template <typename T>
T& Function_Name(T& parameter1, T& parameter2, ...)
{
   // code
}

// Definition by pointer
template <typename T>
T* Function_Name(T* parameter1, T* parameter2, ...)
{
   // code
}

// Calling with deduction
Function_Name(parameter1, parameter2,...);

// Calling with explicitly
Function_Name<dataType>(parameter1, parameter2,...);
```

- Function templates are just blueprints. They're not real C++ code consumed by the compiler. The compiler generates real C++ code by looking at the arguments you call your function template with.
- The real C++ function generated by the compiler is called a template instance.
- A template instance will be reused when a similar function call (argument types) is issued. No duplicates are generated by the compiler.
- Real function declarations and definitions, aka template instances are created when you call the function with arguments.

- If the template parameters are of the same type (T,T), then the arguments you call the function with must also match, otherwise you get a compiler error.
- The arguments passed to a function template must support the operations that are done in the body of the function.
- cppinsights.io that can show you template instantiations. You can even use the debugger to infer that information from the activation record of a template function



```
template <typename T>
T maximum(T a , T b){
    return (a > b)? a : b;
}


int main(){

    long long int x{5};
    long long int y{7};
    int result = maximum(x,y);
    //std::cout << "result : " << result << std::endl;

    return 0;
}
```

```
template <typename T>
T maximum(T a , T b){
    return (a > b)? a : b;
}

/* First instantiated from: insights.cpp:13 */
#ifdef INSIGHTS_USE_TEMPLATE
template<>
long long maximum<long long>(long long a, long long b)
{
  return (a > b) ? a : b;
}
#endif
```

- For the template arguments must be of the same type.

> Template instances won't always do what you want. A good example is when you call our maximum function with pointers. DISASTER!

Template Specialization

Function Templates aren't real C++ code, they're blueprints the compiler uses to generate actual C++ code in template instances. Template specializations however are real C++ code. If multiple function template specialization definitions show up in multiple files, you'll get redefinition errors.

```
// Definition with template specialization
template <>
return_Type Function_Name<Data_Type> (Data_Type parameter1, Data_Type parameter2, ...)
{
   // code
}
```

Template Overloading

Overloaded functions with templates

```
//Function template
template <typename T> T maximum(T a,T b){
    return (a > b) ? a : b ;
}

//A raw overload will take precedence over any template instance
//if const char* is passed to maximum
const char* maximum(const char* a, const char* b){
    //std::cout << "Raw overload called" << std::endl;
    return (std::strcmp(a,b) > 0) ? a : b ;
}

//Overload through templates. Will take precedence over raw T
//if a pointer is passed to maximum
template <typename T> T* maximum(T* a, T* b){
    //std::cout << "Template overload called" << std::endl;
     return (*a > *b)? a : b;
}
```

C++ treats specialization and overloads very differently. This is best explained with an example.

```
template <typename T> void foo(T);
template <typename T> void foo(T*); // overload of foo(T)
template <>           void foo<int>(int*); // specialisation of foo(T*)

foo(new int); // calls foo<int>(int*);
```

The compiler does overload resolution before it even looks at specializations. So, in both cases, overload resolution chooses `foo(T*)`. However, only in the first case does it find `foo<int*>(int*)` because in the second case the `int*` specialization is a specialization of `foo(T)`, not `foo(T*)`.

> overload when you can, specialize when you need to.

Template with Multiple Parameters

```
template <typename return_Type, typename T, typename P>
return_Type Function_Name (T parameter1, P parameter2, ...)
{
  // code
}
```

The order of the arguments matters

```
template <typename ReturnType, typename T, typename P>
//Possible calls
    int max1 = maximum<int,char,long>('c',12L); // OK
    int max2 = maximum<int,char>('d',12L);      // OK
    int max3 = maximum<int>('e',14L);           // OK
    int max4 = maximum('f',14L);                // Error : return type can't be deduced

template <typename T,typename ReturnType,  typename P>
//Possible calls
    int max1 = maximum<char,int,long>('c',12L); // OK
    int max2 = maximum<char,int>('d',12L);      // OK
    int max3 = maximum<char>('e',14L);          // Error : return type can't be deduced
    int max4 = maximum('f',14L);                // Error : return type can't be deduced

template <typename T, typename P, typename ReturnType>
//Possible calls
    int max1 = maximum<char,long,int>('c',12L); // OK
    int max2 = maximum<char,long>('d',12L);     // Error : return type can't be deduced
    int max3 = maximum<char>('e',14L);          // Error : return type can't be deduced
    int max4 = maximum('f',14L);                // Error : return type can't be deduced
```

> We can use `auto` to deduced the `return` type, and the function definition must be before the `main` function.

Decltype with auto

The `decltype` type specifier yields the type of a specified expression. The `decltype` type specifier, together with the `auto` keyword, is useful primarily to developers who write template libraries.
Use `auto` and `decltype` to declare a template function whose return type depends on the types of its template arguments. Or, use `auto` and `decltype` to declare a template function that wraps a call to another function, and then returns the return type of the wrapped function.

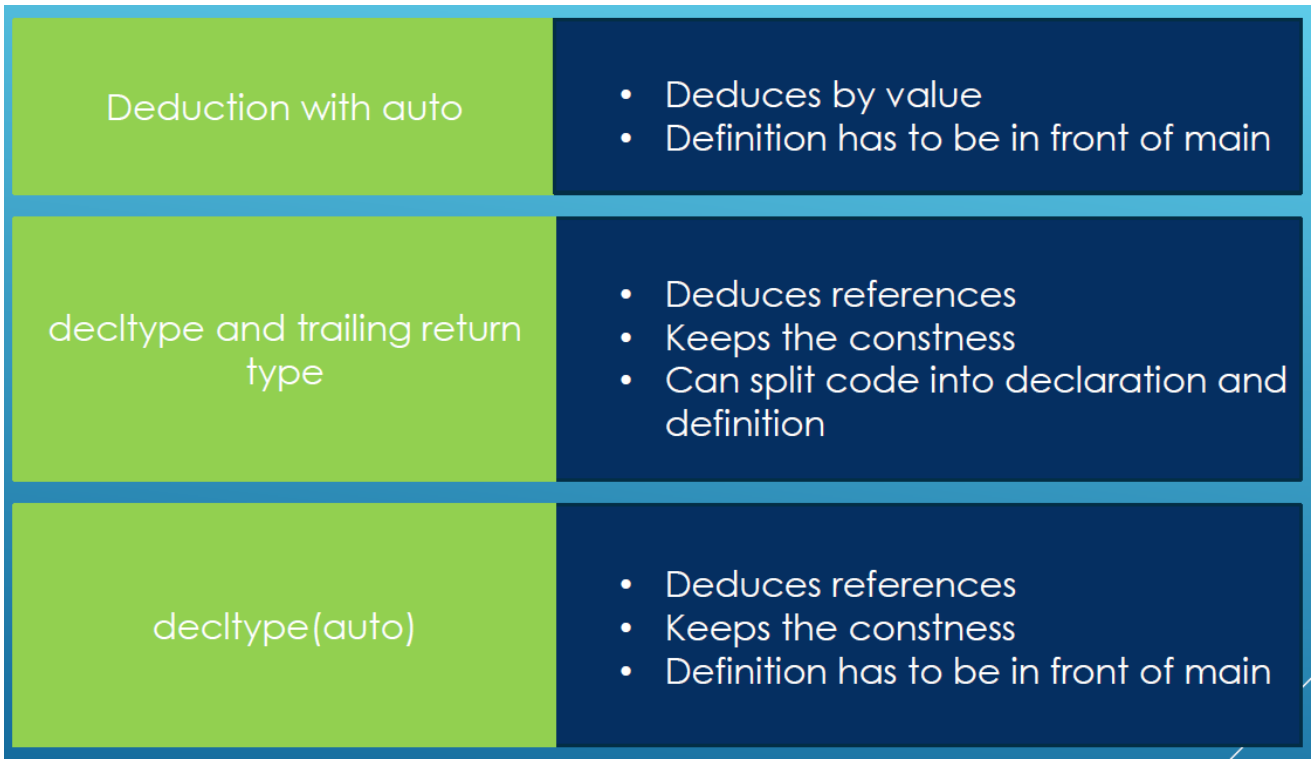
```
decltype( expression ) // its return the type of the expression parameter
```

In C++14, you can use `decltype(auto)` with no trailing return type to declare a template function whose return type depends on the types of its template arguments.

In C++11, you can use the `decltype` type specifier on a trailing return type, together with the `auto` keyword, to declare a template function whose return type depends on the types of its template arguments. For example, consider the following code example in which the return type of the template function depends on the types of the template arguments. In the code example, the *UNKNOWN* placeholder indicates that the return type cannot be specified.

The introduction of the `decltype` type specifier enables a developer to obtain the type of the expression that the template function returns. Use the *alternative function declaration syntax* that is shown later, the `auto` keyword, and the `decltype` type specifier to declare a *late-specified* return type. The late-specified return type is determined when the declaration is compiled, instead of when it is coded.

```
// C++11
template<typename T, typename P>
auto Function_Name (T a, P b) -> decltype(expression)
{
  // code
}

// C++14
template<typename T, typename P>
decltype(auto) Function_Name (T a, P b)
{
  // code
}
```



Non Type Template Parameter

A template non-type parameter is a template parameter where the type of the parameter is predefined and is substituted for a constexpr value passed in as an argument.

```
template<Data_Type T, typename P>
Return_Type Function_Name (T a)
{
  // code
}
```

A non-type parameter can be any of the following types:

- An integral type
- An enumeration type
- A pointer or reference to a class object

- A pointer or reference to a function
- A pointer or reference to a class member function
- std::nullptr_t
- A floating point type (since C++20)

> In c++17 and below, only `int` like types could be used as non type template parameter.

auto Function Template

```
auto Function_Name (auto parameter1, auto parameter2, ..)
{
   // code
   // its return the largest type
}
```

Template Parameter for Lambda

```
auto Function_Name = [] <typename T> (T parameter1, T parameter2, ..)
{
   // code
   // pervent passing different parameter
}
```

C++20 Standard Concepts

The concepts library provides definitions of fundamental library concepts that can be used to perform compile-time validation of template arguments and perform function dispatch based on properties of types. These concepts provide a foundation for equational reasoning in programs.

Most concepts in the standard library impose both syntactic and semantic requirements. It is said that a standard concept is *satisfied* if its syntactic requirements are met, and is *modeled* if it is satisfied and its semantic requirements (if any) are also met.

Core language concepts

Defined in header <concepts>

same_as (C++20)	specifies that a type is the same as another type (concept)
derived_from (C++20)	specifies that a type is derived from another type (concept)
convertible_to (C++20)	specifies that a type is implicitly convertible to another type (concept)
common_reference_with (C++20)	specifies that two types share a common reference type (concept)
common_with (C++20)	specifies that two types share a common type (concept)
integral (C++20)	specifies that a type is an integral type (concept)
signed_integral (C++20)	specifies that a type is an integral type that is signed (concept)
unsigned_integral (C++20)	specifies that a type is an integral type that is unsigned (concept)
floating_point (C++20)	specifies that a type is a floating-point type (concept)
assignable_from (C++20)	specifies that a type is assignable from another type (concept)
swappable **swappable_with** (C++20)	specifies that a type can be swapped or that two types can be swapped with each other (concept)
destructible (C++20)	specifies that an object of the type can be destroyed (concept)
constructible_from (C++20)	specifies that a variable of the type can be constructed from or bound to a set of argument types (concept)
default_initializable (C++20)	specifies that an object of a type can be default constructed (concept)
move_constructible (C++20)	specifies that an object of a type can be move constructed (concept)
copy_constructible (C++20)	specifies that an object of a type can be copy constructed and move constructed (concept)

```
// Syntax 1
template <typename T>
requires std::C++_Concept <T>
T Function_Name (T a, T b)
{
   // code
}

// Syntax 1 by using a type trait
template <typename T>
requires std::Type_Trait <T>
T Function_Name (T a, T b)
{
   // code
}

// Syntax 2
template <std::C++_Concept T>
T Function_Name (T a, T b)
{
   // code
}

// Syntax 3
auto Function_Name (std::C++_Concept auto a, std::C++_Concept auto b)
```

```
{
   // code
}

template <typename T>
T Function_Name (T a, T b) requires std::C++_Concept <T>
{
   // code
}
```

C++20 Custom Concepts

We can build our concepts for custom usage. It can use like the **C++20 Standard Concepts**.

The requires clause can take in four kinds of requirements :

1. Simple Requirements

```
// Simple requirements
template <typename T>
concept Concept_Name = requires (T a, ..)
{
    expression // check for valid condition
 };
```

2. Nested Requirements

```
// Nested Requirements
template <typename T>
concept Concept_Name = requires (T a, ..)
{
   expression // check for valid condition
   requires expression // check if the expression is true
 };
```

3. Compound Requirements

```
// Compound Requirements
template <typename T>
concept Concept_Name = requires (T a, ..)
{
   // check for valid condition and the result is valid for second condition
   expression -> std::C++_Concept
 };
```

We can use `||` and `&&` to compound the concept

```
// Example
#include <iostream>
#include <concepts>
```

```
template <typename T>
concept TinyType = requires ( T t)
{
   sizeof(T) <=4; // Simple requirement
   requires sizeof(T) <= 4; // Nested requirement
};

template <typename T>
//requires std::integral<T> || std::floating_point<T> -> OR operator
//requires std::integral<T> && TinyType<T>          -> AND operator
requires std::integral<T> && requires ( T t)
{
   sizeof(T) <=4; // Simple requirement
   requires sizeof(T) <= 4; // Nested requirement
}
T add(T a, T b)
{
   return a + b;
}

int main()
{
   long long int x{7};
   long long int y{5};

   add(x,y);
   return 0;
}
```

4. Type Requirements

   This's out of scope for now.

> We can use Concepts with `auto` keyword.

```
std::integral auto add (std::integral auto a,std::integral auto b)
{
   return a + b;
}
// or
std::floating_point auto x = add(5,8);
```